

SWT Skript SoSe 2018


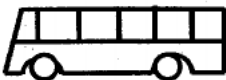




Riebisch

1. Anforderungsbeschreibung mit UML

1.1. Die „frühen Phasen“ im Vorgehensmodell der Softwareentwicklung

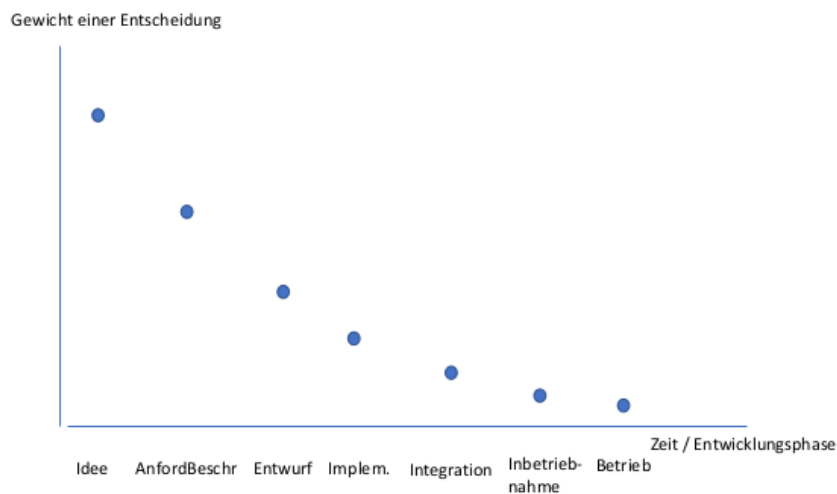
Größere industrielle Projekte erfordern mehr als nur das Beherrschen einer Programmiersprache:

- **Arbeitsteilung** (mehrere Personen/Spezialisten)
- unterschiedliche teilweise unvollständige Wahrnehmung: **gemeinsame Konzepte entwickeln**
- **Kommunikation**: Konzepte gleichartig verstehen
 - verstehen komplexer Zusammenhänge trotz begrenztem Aufnahmevermögen
 - Arbeitsteilung zwischen Personen mit teilweisem Wissen oder unterschiedlichen Sichtweisen
- **Wissen über lange Zeit aufbewahren**
- Komplexität: Anforderungen wie **Verfügbarkeit, Sicherheit, Korrektheit, heterogene Technologien** bergen wichtigen Entscheidungen und große Risiken
 - **Risikominimierung**: durch Modelle als frühzeitige Gegenstände der Prüfung als First Class artefacts
 - **Methodischen Vorgehen**: frühere Erfahrungen nutzen, Lösungen wiederverwenden, wichtige Aspekte nicht vergessen

Was der Anwender wollte 	Wie es der Anwender dem Programmierer sagte 	Wie es der Programmierer verstanden hat 
Was der Programmierer bauen wollte 	Was der Programmierer tatsächlich gebaut hat 	Was der Anwender tatsächlich gebraucht hätte 

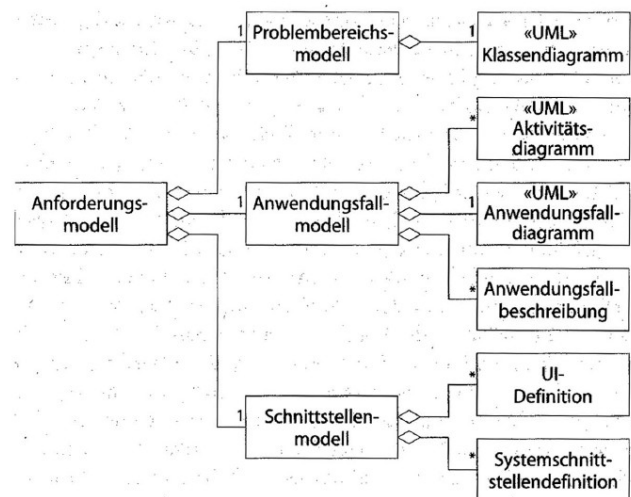
Gewicht der frühen Phasen

- Grundlegende Entscheidungen haben Auswirkungen auf nachfolgende
- Änderungen dieser Entscheidungen haben Veränderungen der darauf aufbauenden Arbeiten zur Folge → Rework
- Falsch verstandene Anforderungen führen zum Bau eines falschen Systems



Ziele der Phase Anforderungsbeschreibung / -erhebung:

- Erreichen von gleichartigem Verständnis der Aufgabenstellung bei Auftraggeber und Entwickler
- Entwickler lernen Anwendungsfachgebiet kennen (sind keine Experten!)
- Sammeln von Informationen über Aufgabenstellung (Widersprüche ausräumen, Aufgabe beschreiben, Verständnis zwischen AG und AN angleichen) → Prototyp
- Entwickler lernen gegenwärtigen Zustand kennen
- Modellieren der Anforderungen mit Anforderungsmodell (Requirements specification)
- Beschreiben der Aufgabenstellung in Pflichtenheft, + Übereinkommen mit Auftraggeber / Vertrag
- Anwender mit Möglichkeiten und Komplexität der Lösung vertraut machen ® Beraten über optimale Aufgabenstellung
- Festlegen der Nutzerschnittstelle



- Bereitstellen der Basis für Planung

Techniken der Anforderungsbeschreibung/ -erhebung

- Fragen Stellen
- Zuhören
- Infos(Anforderungsmodell) festhalten und strukturieren

Ergebnisse dieser Phase

- Anforderungsmodell
- Schnittstellen modell (Wir haben immer schnittstellen nach außen die wir bedienen müssen)
- Lastenheft/Pflichtenheft (Musterspezifikation, Mustergliederung)

Spezifikation am Beispiel Volere

PROJECT DRIVERS

1. The Purpose of the Product
2. Client, Customer and other

Stakeholders

3. Users of the Product

PROJECT CONSTRAINTS

4. Mandated Constraints
5. Naming Conventions and Definitions
6. Relevant Facts and Assumptions

FUNCTIONAL REQUIREMENTS

7. The Scope of the Work
8. The Scope of the Product
9. Functional and Data Requirements

NON-FUNCTIONAL REQUIREMENTS

10. Look and Feel Requirements
11. Usability Requirements
12. Performance Requirements
13. Operational Requirements
14. Maintainability and Portability Requirements
15. Security Requirements
16. Cultural and Political Requirements

17. Legal Requirements

PROJECT ISSUES

- 18. Open Issues
- 19. Off-the-Shelf Solutions
- 20. New Problems
- 21. Tasks
- 22. Cutover
- 23. Risks
- 24. Costs
- 25. User Documentation and Training

1.2. Use-Case-Modellierung mit UML

Definition Use Case (Struktur, Tatbestand, Anwendungsfall):

Wir wollen erreichen dass Softwaresysteme ein bestimmtes Verhalten haben. Dieses Verhalten müssen wir uns beschreiben lassen.

Ein Use Case beschreibt eine Menge von Aktivitäten (etwas was getan wird) eines Systems aus der Sicht seiner Akteure (geht nicht um die technische Seite sondern um das was der Benutzer wahrnimmt), die für die Akteure zu einem wahrnehmbaren Ergebnis führen. Ein Use Case ist ansonsten eine komplette, unteilbare Beschreibung. (Das was nur innerhalb der Software passiert ist irrelevant)

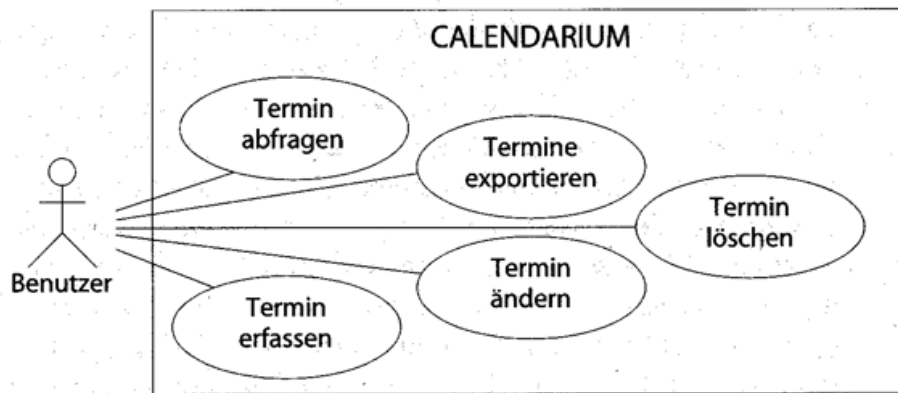
Definition Use-Case-Diagramm (Anwendungsfalldiagramm):

Ein Use-Case-Diagramm zeigt die Beziehungen zwischen Akteuren und Use Cases.

Das Anwendungsfalldiagramm beschreibt die Abläufe der Software.

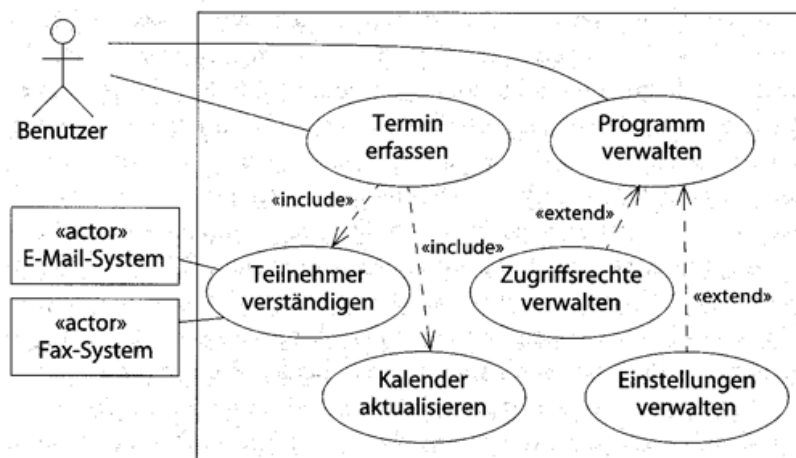
Identifikation von Use Cases

- Sammeln der Kundenwünsche (mit Auftraggeber Sprechen, Endbenutzer etc, gab es schonmal ein System? Tätigkeite zu finden die die Software unterstützen)
- Analysieren von Textdokumenten: Verben mit Tätigkeitsbeschreibung
- "echte" Use Cases finden, verallgemeinern und zusammenfassen
- zunächst recht vollständig, weitere Verfeinerung (erstmal so volsätngi machen, wie es mit den vorhandenen Infos möglich ist)
- für jeden Use Cases Textbeschreibung angeben → Vervollständigung (welche Strukturen gibt es,
- Aufnahme in Use-Case-Diagramm

**Abb. 2-52**

Anwendungsfalldiagramm
für CALENDARIUM

Abb. 2-53
include-Beziehung und
extend-Beziehung zwischen
Anwendungsfällen



11

Use-Case-Diagramm: Elemente

System: Rechteck mit Namen

Aktor: Strichmännchen oder Rechteck mit <<aktor>>, und Name, ist ein Stereotyp

Use Case: Ellipse mit Name

Assoziation: $A \leftarrow B$, B hat zu tun mit A, Aussagekräftiger mit Namen und Richtung, (wenn man da den Namen „ist Teil von“ ranschreibt, ist es auch eine Aggregation)

Aggregation: $A \diamond \leftarrow B$ B ist Teil von A, A: ganzes, B: Teil,

Komposition: $A \blacklozenge \leftarrow B$ Teil-von, Existenzabhängigkeit, B ist Teil von A, Existenz hängt von A ab, wenn ich A lösche ist B auch weg

Kommunikationsbeziehung: einfache Linie

include-Beziehungen: $A - - - > B$ gestrichelte Linie mit Pfeil; zwischen Use Cases, Teil-von-Beziehung; bedeutet Abhängigkeit, A hängt ab von B, wenn b geändert wird muss auch A geändert werden

extend-Beziehung: kennzeichnet Varianten; damit ist gemeint dass man eine Small, medium und enterprise variante der software haben kann. Dann unterscheiden sich die Use Cases die dann mit der extend beziehung hinzugefügt werden können

Generalisierungsbeziehung: Verallgemeinerung/Generalisierung, ist-ein, **A** ←^{is ein} **B**

A ist das allgemiene elemten, B das speziellere, B ist eine Spezialisierung von A und A eine Verallgemeinerung von B

<<name>> Stereotyp: UML-Elemente genauer beschreiben

- - - ><< call >> Aufrufbeschreibung (genaue Beschreibung einer Abhängigkeit) als Beispiel für Stereotyp

Paket: „Ordner: mit name und <<package>

FRAGE: Welche Erweiterungsmöglichkeiten für UML gibt es? z.b Stereotyp

Use-Case-Beschreibung

→ als Teil der Use Case Modellierung (UML ist ein standard mit verschiedenen Versionen)

→ Use Case bscheibung ist in Textform und Teil des Standards, kann sich also immer unterscheiden

- Anwendungsfall-Nº Name des Anwendungsfalls
- Akteure
- Vorbedingungen
- Nachbedingungen
- Invarianten (Dateien sollen am ende geschlossen sein, ...)
- Qualitätsmerkmale (Anforderungen wie Performance, Sicherheit, ..)
- Ablaufbeschreibung / Durchführung „Sunny Day“
- Ausnahmen, Fehlersituation
- Alternativabläufe (Alternativsituation: will etwas bestellen, habe aber keinen Account und werde dann weitergeleitet auf Account erstellen)

Paket in der UML

Ziel von Pakete: Gliederung, wenn wir viele Use Cases haben wird es leicht unübersichtlich.

→ Ein Paket ist ein Stereotyp

Definition: Pakete sind **Ansammlungen von Modellelementen beliebigen Typs**, mit denen das Gesamtmodell in kleinere, überschaubare Einheiten gegliedert wird. Ein Paket definiert einen

Namensraum (damit ist Eindeutigkeit gegeben) (hierarchisch) wiederum Pakete enthalten. Das oberste Paket beinhaltet das ganze System. Jedes Modellelement kann in anderen Paketen referenziert werden, gehört aber zu genau einem (Heimat-)Paket.

1.3. Aktivitätsdiagramm

Definition: Eine Aktivität beschreibt einen Ablauf und wird definiert durch verschiedene Arten von Knoten, die durch Objekt- und Kontrollflüsse miteinander verbunden sind. Es werden Aktions-, Objekt- und Kontrollknoten unterschieden.

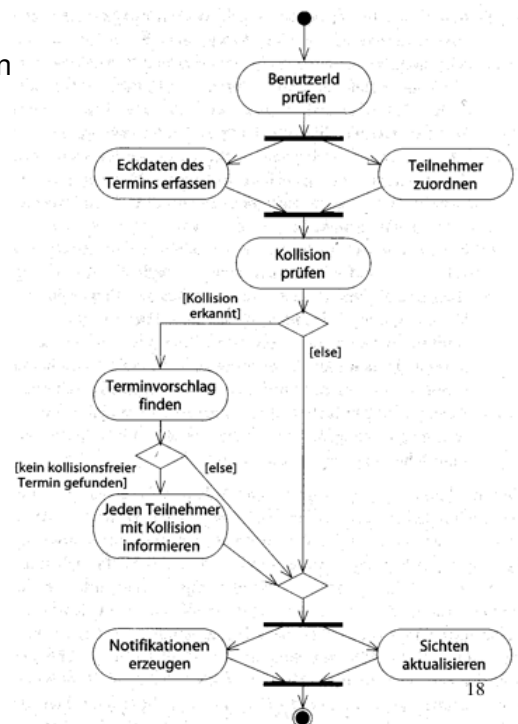
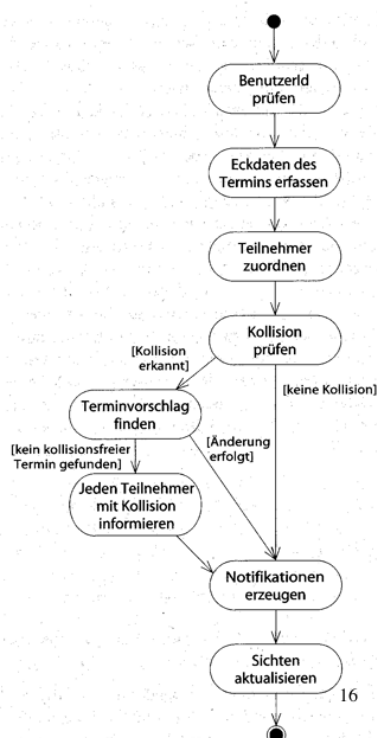
→ sind alles Tätigkeiten die da stehen, wir wollen Tätigkeiten erfassen

Aktivitätsdiagramm Beispiel (links)

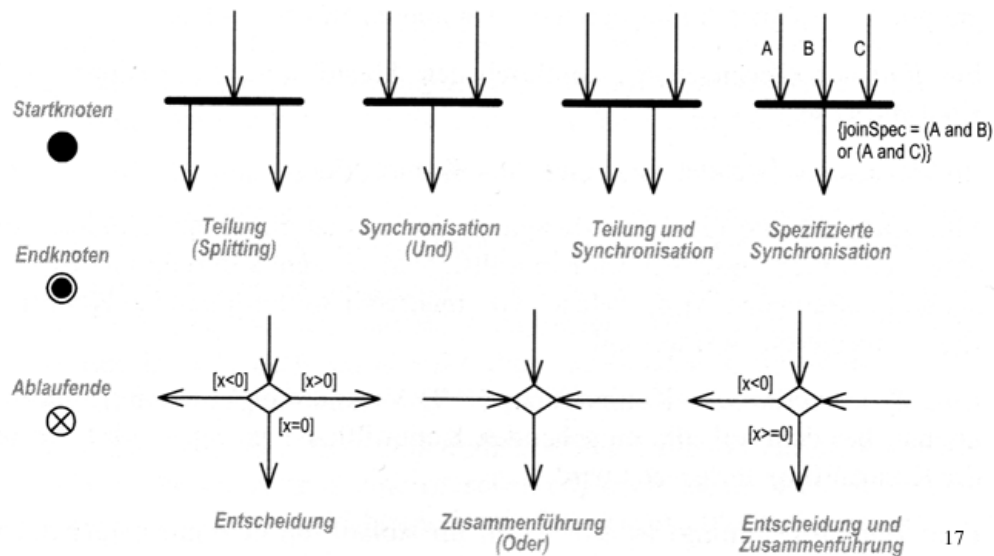
- Ablauf des Use Case „Termin erfassen“ als Aktivitätsdiagramm Version 1
- hier teilt sich der Fluss auf unter Bedingungen,
- ein Aktivitätsknoten bezeichnet Aktion/Handlung

Steuerfluss Beispiel (rechts)

- Ablauf des Use Case „Termin erfassen“ als Aktivitätsdiagramm Version 2
- Steuerfluss mit Entscheidungen und Zusammenführungen/Synchronisation zur Darstellung von Nebenläufigkeiten
- **Synchro:** Fortsetzung erst wenn beide Pfade angekommen sind, nicht unbedingt Parallelisierung, Reihenfolge ist egal
- **Partition:** Verantwortungsbereich mit zwei Akteuren



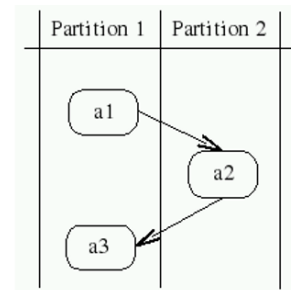
Kontrollknoten: Steuerfluss



17

Verantwortlichkeitsbereiche: Partitionen

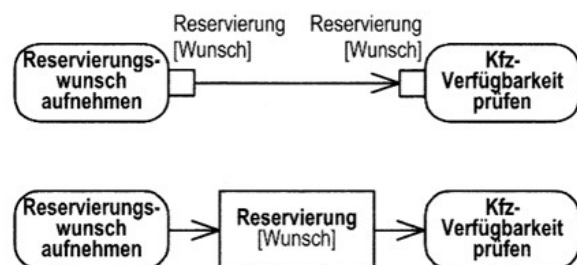
Definition: Eine Partition (Verantwortlichkeits- oder Funktionsbereich) beschreibt innerhalb des Aktivitätsmodells, wer oder was für einen Knoten verantwortlich ist oder welche gemeinsame Eigenschaft Knoten kennzeichnet.



Datenfluss: Objektknoten und Objektfluss

→ UML hat keine Möglichkeit Datenfluss darzustellen

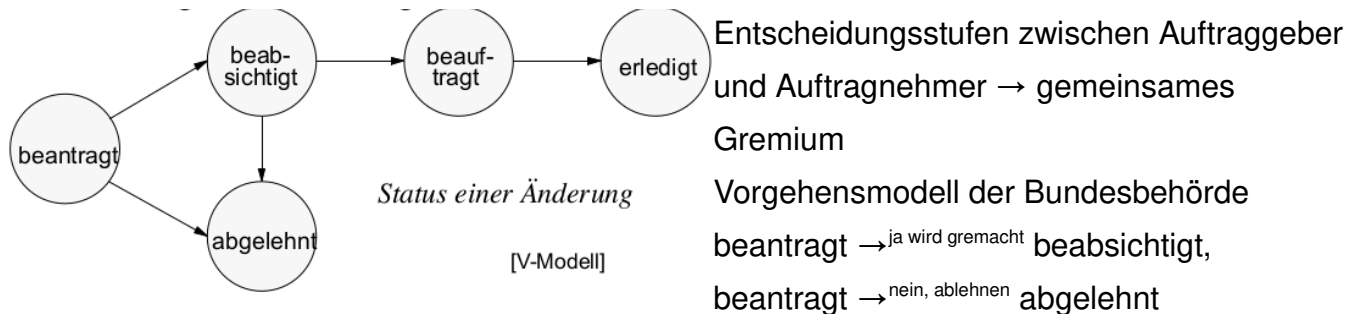
Ein Objektknoten gibt an, dass ein Objekt oder eine Menge davon existiert. Objektknoten können als ein- oder ausgehende Parameter in Aktivitäten verwendet werden. Ein Objektfluss ist wie ein Kontrollfluss, bei dem jedoch Objekte transportiert werden.



Änderungsmanagement Ziele

- Änderungen (von Anforderungen) koordinieren
- vor ungeplanten Änderungen schützen
- und am Ende die verschiedenen Versionen der Änderungen auch zu behalten (ins Konfigurationsmanagement übernehmen)

Vorgehensweise:



Prüfkriterien Anforderungsbeschreibung

→ geben mir Überblick darüber was Anforderungsbeschreibung ausmacht

Eindeutigkeit - nur eine einzige Interpretation möglich? Aus V-Modell

Vollständigkeit - alle notwendigen Funktionen berücksichtigt? Wie findet man raus was fehlt?

Überprüfbarkeit - Erfüllung der Anforderungen überprüfbar?

Widerspruchsfreiheit - Konflikte zwischen den Anforderungen?

Verständlichkeit - für alle Beteiligten?

Herkunft - Herkunft/Begründung der Anforderung klar beschrieben? Der Urheber muss klar sein, mit den Personen die man reden muss haben verschiedene Interessen, muss man bei jeder Anforderung wissen wer das entschieden hat

Flexibilität und Abhängigkeiten - ohne Auswirkung auf andere Anforderungen änderbar?

Rückverfolgbarkeit - eindeutig zu identifizieren? Damit man Anforderungen abhaken kann

Abstrahierbarkeit - Ist die Anforderung implementierungsunabhängig?

Klassifizierbarkeit bezüglich Bedeutung - Risiken bezüglich

Realisierbarkeit; Stabilität über gesamten Lebenszyklus?

Angemessenheit - Wünsche und Bedürfnisse des Benutzers abgedeckt?

1.4. Problembereichsmodellierung (siehe. Teile des Anforderungsmodell)

Konzeptionellen Beschreibung („was“) statt Darstellung der

Realisierung („wie“): Prüfkriterien sagen nichts zur technischen Implementierung

- **Struktur:** Klassendiagramm (Komponentendiagramm)
 - struktur heißt welche Teile gibt es und wie hängen sie miteinander zusammen, nicht welche Abläufe gibt es
- **Verhalten:** Zustands-, Kommunikations-, Aktivitätsdiagramm
- **Elemente der Strukturmodelle:** Klasse, Objekt, Attribut, Methode, Paket, (Komponente)
- **Beziehungen:** Generalisierung (Vererbung), Assoziation, Rolle, Aggregation, Komposition, Abhängigkeit

Unterschied Problembereich und Lösungsbereich:

- mit dem **Problembereich/ Problem domain** wollen wir darstellen
 - Zielstellung
 - Spezifikation/Anforderungsbeschreibung
 - fachliche Zusammenhänge
 - Was soll gebaut werden?
 - Beispiele:
 - Organisatorisches Umfeld/Struktur, (Wer macht was in der Uni, Wie hängen Studiengänge zusammen)
 - Abläufe wie Use Cases
 - Reaktionen auf Fehlerfälle und Änderungen
 - Daten mit Beziehungen zwischen diesen Daten und Wertebereich
 - Geschäftsregeln mit Kreditvergabe
- mit dem **Lösungsbereich/ solution domain** wollen darstellen wie technische Seite aussieht, alles was man von außen nicht sieht
 - Entscheidungen über technische Lösung beschreiben (Warum bauen wir es so)
 - interna der Lösung / wie es intern funktioniert
 - Wie soll es gebaut werden? Wie ist es gebaut?
 - Beispiele:

- technischer Aufbau eines Systems
- Datenspeicherung und Synchro
- Schnittstellen zwischen Komponenten
- Mechanismen der Fehlerbehandlung
- Datenmodell des DBMS

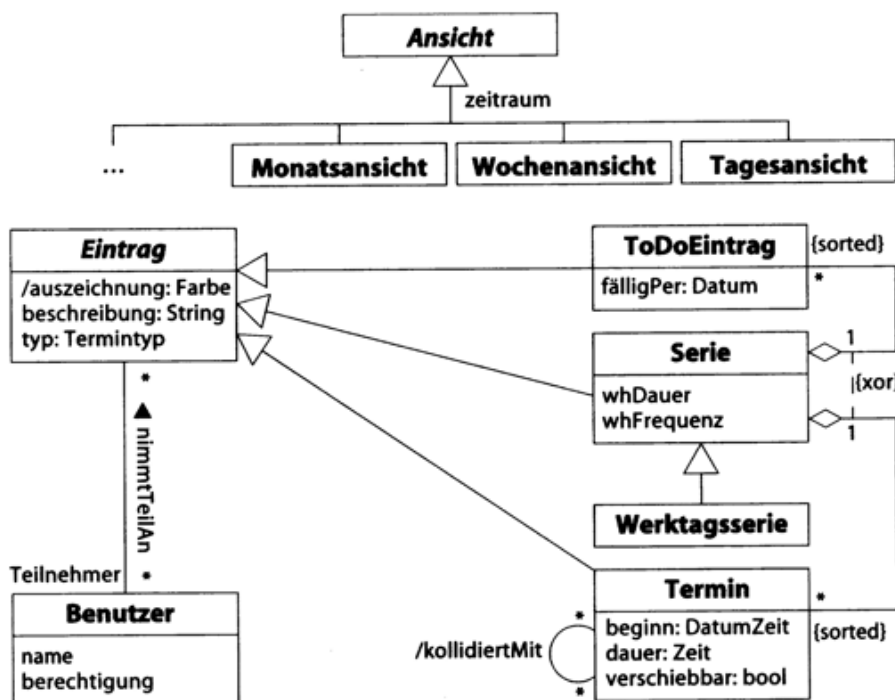
1.5. Klassendiagramm

Generalisierung

Generalisierung als Beziehung zwischen Klassen wird wie Taxonomie und Vererbung genutzt, um **Eigenschaften zusammenzufassen** und zu **strukturieren**. Dadurch wird erreicht, dass das Beschreiben bzw. Ändern einer Beschreibung der Eigenschaften weniger Aufwand erfordert, weil **Redundanzen verringert** werden. Durch Vererbung werden Eigenschaften auf untergeordnete Klassen übertragen, dabei können sie von diesen ergänzt und verändert werden (Spezialisierung).

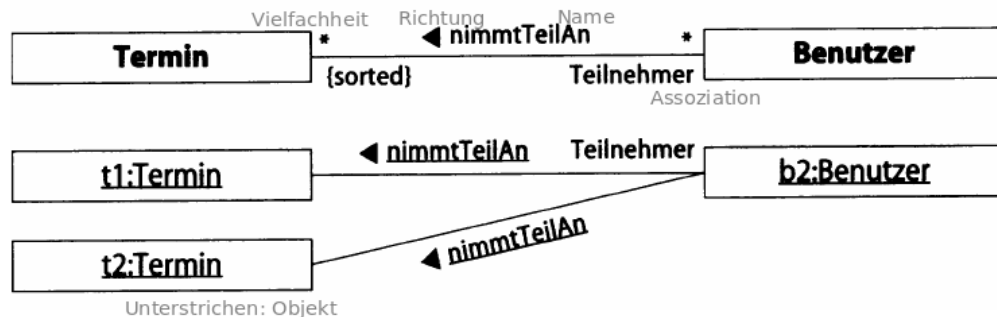
→ Hierarchie als Mittel zur Behandlung von Komplexität

Beispiel:



Abstraktion: Beziehungen zwischen Klassen

Verallgemeinert: Klassenbeziehung



Beziehungen zwischen Objekten

Assoziation

Die Assoziation stellt eine statische Beziehung zwischen Objekten dar.

Sie wird beschrieben durch:

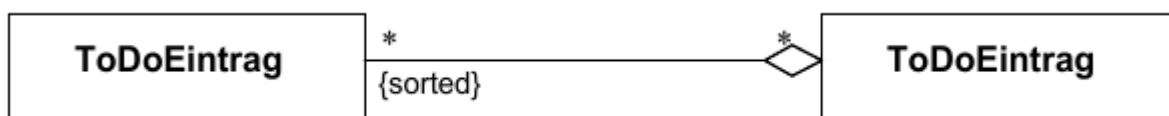
- Name (Semantik) und Richtung
- Vielfachheit: Anzahl der Partnerobjekte
- Evtl. Rolle beteiligter Objekte



→ Zwischen zwei Klassen kann es mehrere Beziehungen geben

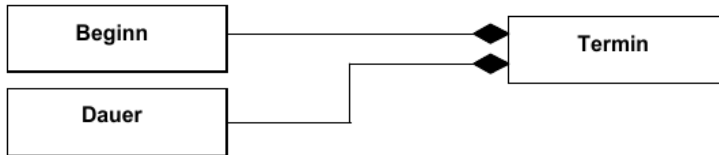
Aggregation und Komposition

Die Aggregation (part-of, Teil-ganzes-Beziehung) stellt eine besondere Assoziation dar, bei der ein zsmgesetztes Objekt in Beziehung zu seinen Teilen dargestellt wird. Es handelt sich um eine asymmetrische Beziehung zwischen nicht gleichwertigen Partnern.



Die Komposition stellt eine spezielle Form der Aggregation dar.

- Ein Teil darf nur zu einem Ganzen gehören
- Ein Teil existiert nur, solange sein Ganzes existiert



1.6. Zustandsdiagramm

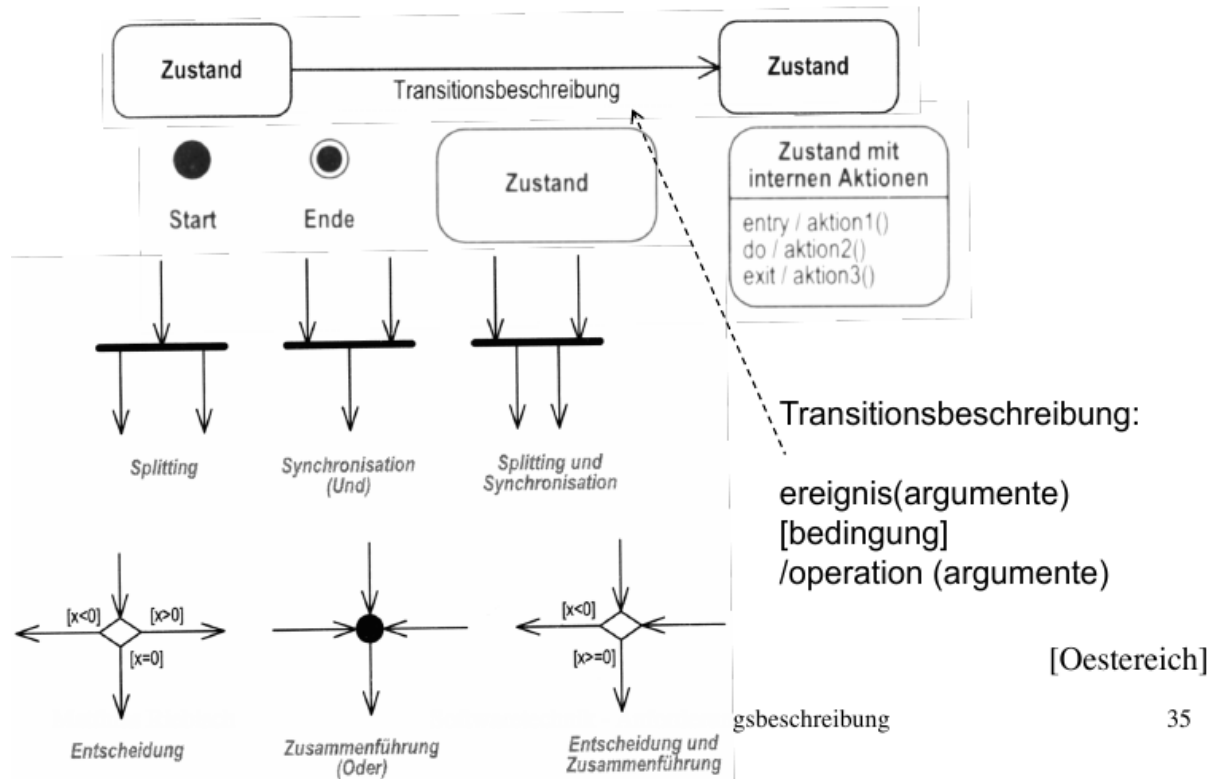
Was macht das Zustandsdiagramm aus?

Beschreibt das Gesamtverhalten → menge aller Zustände und Zustandsübergänge, die ein Objekt im Laufe seines Lebenszyklus einnehmen kann, sowie Stimuli und davon bewirkte Zustandsänderungen. Es ist immer nur auf einen Bereich/Element/System/Komponente/Klasse bezogen. Es ist ein weiteres Diagramm der UML. Es hat den endlichen Automaten als Vorbild.

Hypothetische Maschine – endlicher Automat

- Endliche Menge von Zuständen
- Endliche Menge von Ereignissen
- Zustandsübergänge
- Anfangszustand
- Menge von Endzuständen

Zustandsdiagramm Elemente



35

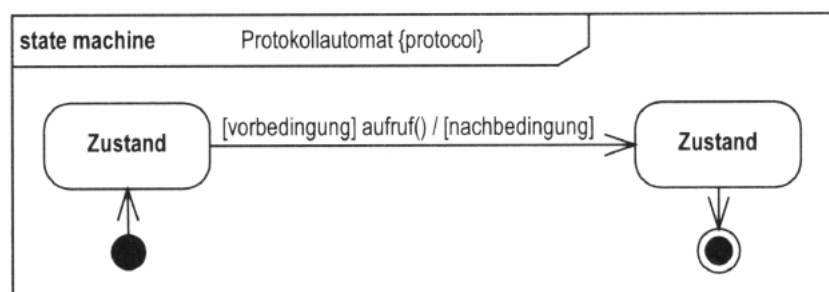
→ beim Zustand mit internen Aktionen können wir Kästen teilen und Aktionen reinschreiben, was beim Aktivitätsdiagramm nicht möglich war

→ ein Pfeil zeigt eine Aktion

→ bei komplexen Systemen kann man Sachen zu Blöcken zsmfassen und sie als Unterzustand kennzeichnen der eine zsmgefasste Aktion darstellt

Protokoll-Zustandsautomat (Protocol State Machine)

Spezielle Form des Zustandsdiagramms, beschreibt lediglich die möglichen und verarbeitbaren Ereignisse ohne weitergehendes Verhalten



2. Entwurf

Wir wollen vom Entwurf in die Implementierung gehen. Fachliche Konzepte können meist nicht eins zu eins in die technische übernommen werden.

Aufgaben Entwurf

- (beschreiben der Probleme aus dem Problembereich und) Überführen der Anforderungen in technische Lösung
- Erfüllung funktionaler Anforderungen
 - Workflows und Aktionen, Datenstrukturen, GUI
 - Abläufe, Strukturelle Sachen
- Erfüllung von Randbedingungen
 - Plattform, Schnittstellen zu anderen System
 - Datenbanken, Umgebungssysteme
 - Welche Zahlungsdienstleister arbeiten mit uns
 - OwnCloud → Aus Sicherheitsgründen an Spezialisten übergeben
- Erfüllung nicht-funktionaler Anforderungen
 - Zeitverhalten
 - Skalierbarkeit
 - Robustheit
 - Änderbarkeit
 - Sicherheit
- Risiken
 - Verständnis schaffen
 - Kernwissen

Ziele:

- Erfüllung der Anforderungen
 - funktionale
 - Randbedingungen
- nicht funktionale
 - Risiken
 - Verständnis schaffen, Kernwissen

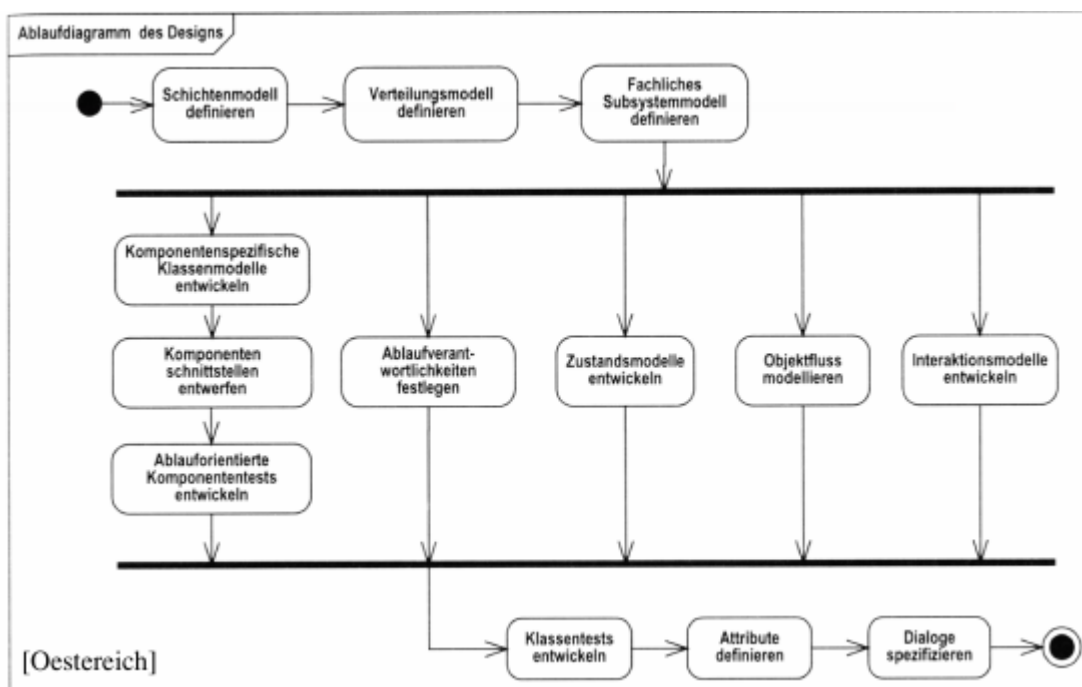
Quasar: Qualitätsorientierte SW Architektur

Empfehlung zur Aufteilung in Komponenten

- A-fachliche, Anwendungsbezogene
- T-technische
- O-weder fachlich noch technisch
- R-Transformatoren/Generatoren (Templates)

→ A und T werden stark von einander getrennt, auch keine Referenzen, kein Bezug aufeinander, auf die beiden Komponenten konzentrieren wir uns

Ablauf Entwurf



Techniken

Techniken kommen daher, dass wir methodisch/systematisch vorgehen und bestimmte Schritte einhalten um nichts zu vergessen

- das sind die Entwurfsmetapher die wir anwenden.
 - → WAM, DDD, Jacobsen, Service orientierter Entwurf, Quasar
- Etablierte Bausteine einsetzen
 - abstrakt: Design pattern, Architekturstile, Taktiken
 - konkrete: Klassenbibliothek, Framework, Tools,

- Beschreibung erstellen
 - Entwurfsmodell: Sichten für die Struktur (also die statischen Sachen), Verhalten, Kontext, Deployment etc
 - Entscheidungen beschreiben: Ziele, Alternativen, Bewertung der Alternativen, Beziehungen zu anderen Entscheidungen

Ziele Entwurf und Architekturentwicklung

Effiziente Entwicklung

- Iterativ, inkrementelle Entwicklung zulassen
- Grundlage der Projektplanung und Management:
 - Organisation, aktive Führung, Einblick, Verhandlungsbasis
- unabhängige, verteilte Implementierung

Risiken minimieren

- Reihenfolge nach Risiken
- Randbedingungen früh berücksichtigen
- Anforderungserfüllung früh prüfen

Verständnis schaffen

- Kommunikation zwischen Stakeholdern: Forderungen, Sichten
- Basis für Schulungen
- Leitbild und Referenz: Dokumentation aus verschiedenen Perspektiven

Kernwissen des Systems konservieren

- Übertragbares Modell: für ähnliche Systeme, Wiederverwendung
- KnowHow, geistiges Eigentum
- Verbesserung ermöglichen

Anmerkungen zu den Zielen:

- Wir wollen effizient arbeiten, wenig Aufwand viel Ergebnis
- Projektplanung: Kunde setzt Deadline, aber SWT schafft es nicht, dann machen wir Dinge provisorisch. Projektleiter setzt Termine und SWT Entwickler versteht diese Termine gar nicht. Projektleiter kann nicht sagen, dass wir jetzt schneller arbeiten oder dass Überstunden gemacht werden müssen. Was er machen kann aus der aktuellen

Version mit Auftraggeber zu besprechen und die unwichtigeren Sachen in die nächste Version zu verschieben. Spätere Features sind vllt möglich.

- Voraussetzungen/Abhängigkeiten zwischen Aufgaben klären, sonst scheitert die Planung
- Implementierung: paar Sachen Outsourcen, dann Schnittstellen bauen, Schnittstellen kommunizieren
- Risiken minimieren: Projekt könnte Scheitern, Termin wird nicht eingehalten
→ Vertragsstrafe
- Bausteine funktionieren vllt nicht zusammen, Qualitätseigenschaften werden nicht eingehalten
- Sicherheitsfragen werden nicht so beantwortet wie es richtig wäre
- Riskante Aufgaben sollten zuerst bearbeitet werden
- Entwurf beschreibt warum gewisse Entscheidungen getroffen wurden
- Leitbild wird gebraucht für Orientierung, weil man immer abweichen muss und dann hat man eine Referenz für das was wichtig ist
- Know how: was machen wir besser als andere Firmen, Erfahrungen sammeln für spätere Projekte--> Kernwissen

Ergebnisse

- Modell der Lösung: z.b Komponenten und Schnittstellen → Architektur
- Entscheidungen der Realisierung
- Technologie-Entscheidungen

Übergang Problembereichsmodell → Entwurf

Jacobsen (UML) ist ein Entwurf um von dem Problembereich in den Lösungsbereich zu kommen. Eine fachliche Klasse hat drei technische Klassen nach Jacobsen.

Die drei Dimensionen:

- Information: Informationsobjekte (Entity relationship model, Entity Object)
- Verhalten: Steuerungsobjekte (control object)
- Darstellung: Schnittstellenobjekte (boundary Object, GUI, Nutzerschnittstelle)

Für jedes Objekt (fachliche Klasse im Problembereichsmodell) machen wir drei technische Objekte (diese drei Dimensionen) die wir dann im Lösungsbereich darstellen.

→ diese Aufteilung ist ähnlich zu MVC

Zerlegungs- und Modularisierungskriterien SOA

	Cohesiveness	Compatibility	Constraints	Communication
Domain	<div>Identity & Lifecycle Commonality</div> <div>Semantic Proximity</div> <div>Shared Owner</div>	<div>Structural Volatility</div>		
Quality	<div>Latency</div>	<div>Consistency Criticality</div> <div>Availability Criticality</div> <div>Content Volatility</div>	<div>Consistency Constraint</div>	<div>Mutability</div>
Physical		<div>Storage Similarity</div>	<div>Predefined Service Constraint</div>	<div>Network Traffic Suitability</div>
Security	<div>Security Contextuality</div>	<div>Security Criticality</div>	<div>Security Constraint</div>	

CC-8 Content Volatility

Description

A nanoentity can be classified by its volatility which defines how frequent it is updated. Highly volatile and more stable nanoentities should be composed in different services.

User Representation

- Volatility can be calculated from use case definitions if they are equipped with a frequency information.
 - Nanoentities can be classified by data types to determine the volatility: Master Data (regularly), Reference Data (rarely), Transaction Data (often) and Inventory Data (often) to determine the volatility.

Type

Compatibility

Perspective

Quality

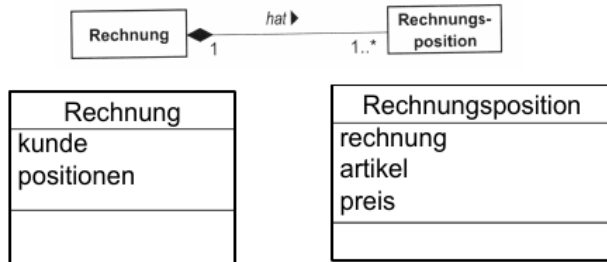
Characteristics

Often, Regularly (*default*), Rarely

Realisierung von Vielfachheiten

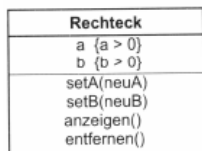
1 – Attribut mit Referenz (Typ des verbundenen Objekts)

* – Attribut mit Liste von Referenzen (Collection-Typ)



Spezialisierung Versuch 1

Quadrat als
spezielles Rechteck
durch Vererbung abgeleitet:



**Zusicherungs-
Verantwortlichkeitsprinzip:**
Eine Unterklasse sollte keine
Zusicherungen auf
Eigenschaften einer Oberklasse
machen

?

[Oestereich]

Spezialisierung Versuch 2

Umgekehrte Vererbungsbeziehung:

Rechteck ist Quadrat, jedoch mit einer
zweiten Kante b

→ möglich aber unverständliche Zuweisungen erlaubt:

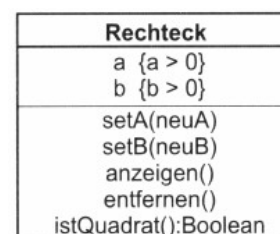
```

class Rechteck extends Quadrat { ... }
Rechteck r;
Quadrat q;

...
q = r; // Zuweisung zulässig, da typkompatibel,
       // jedoch nicht sinnvoll
  
```

Spezialisierung Versuch 3

Vererbung nicht verwendet, Quadrat als Spezialfall in Attribut
modelliert

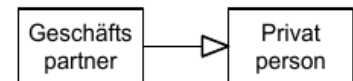
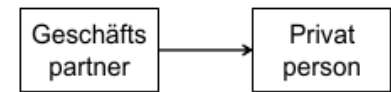


Delegation statt Vererbung

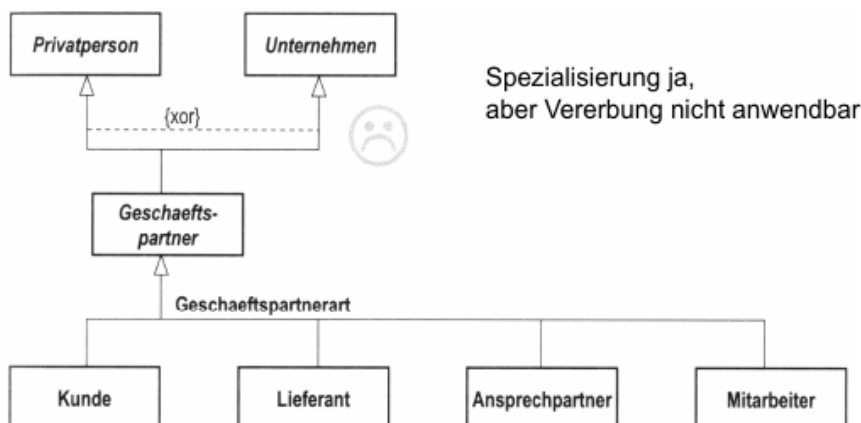
Entwurf: Generalisierung fasst Gemeinsames zusammen

Implementierung:

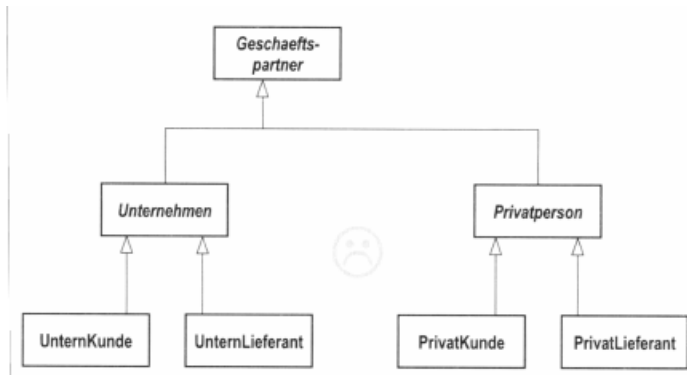
- Delegation – Aufruf einer Methode (bedeutet: ich gebe etwas weiter)
 - In Aufgerufenem Exemplar zsmgefasstes Verhalten und Eigenschaften nutzen
 - Details ausgelagert – Info Hiding als Vorteil
 - Höhere Flexibilität – Austausch zur Laufzeit möglich
 - Generell bevorzugt, es ist die bessere Lösung bei der Spezialisierung im Falle von eben, generell gilt wenn man eine Zsmfassung von Eigenschaften hat, sollte man zuerst über Delegation nachdenken
 - der Geschäftspartner hat eine Referenz auf Privatperson
- Vererbung – Code übernehmen
 - Java-Typsistem nutzen
 - durch Erben in Superklasse zsmgefassten Code übernehmen
 - Verhalten eingebettet in einer Klasse
 - vor allem wenn Verhalten fast unverändert, ist die einfachere Lösung indem die Privatperson ein paar Eigenschaften vom Geschäftspartner erbt



Spezialisierung wechseln: Rollen

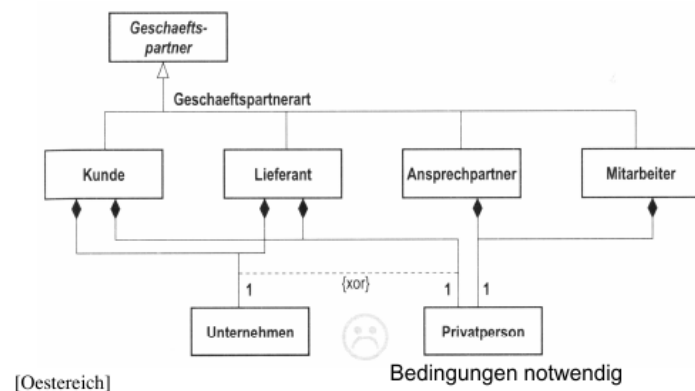


Vererbung Versuch 2

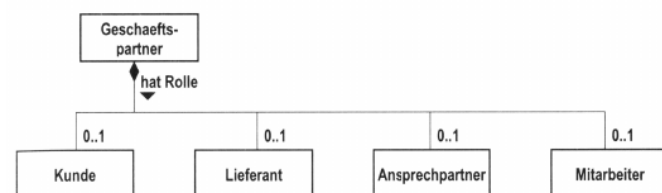


Kombinatorische Explosion

Komposition Versuch 3



Rollen Versuch 4



Rolle = Sichtweise

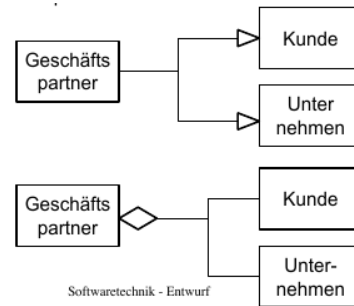
Aggregation statt Vererbung

Entwurf: Aggregation führt Teil-Eigenschaften zusammen

Implementierung: durch Mehrfach-Erben Code übernehmen

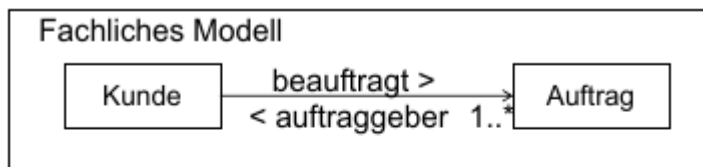
Aggregation: durch Aufruf separat realisiertes Verhalten nutzen

Mehrfachvererbung: in C++ riskant, in Java möglich

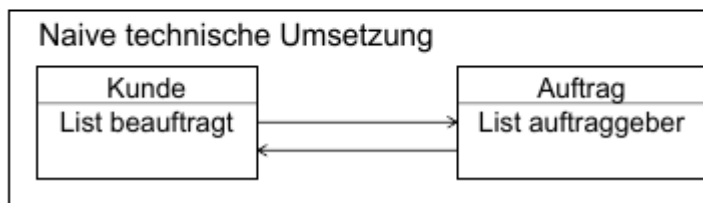


Mehrfach-Aufruf: Information Hiding als Vorteil, Austausch zur Laufzeit möglich

Enge Kopplung durch Objekt-Referenzen



Wechselseitige Beziehungen



Wechselseitige Objektreferenzen

Abhängigkeiten!!

Problemlösung: gemeinsame Komponente (kleine Systeme), lose Kopplung: Übergabe von Werten statt Referenzen

2.4 Kommunikationsdiagramm

- Reihenfolge durch Zahlen
- Fokus auf Objektbeziehungen
- Kästen stehen hier nicht für Klassen sondern für Objekte
- es ist ein Verhaltensdiagramm, es ist also dynamisch (nicht statisch), deshalb muss man hier konkretisieren und braucht Instanzen
- unterschiedliche Pfeilarten zeigen synchron und asynchron

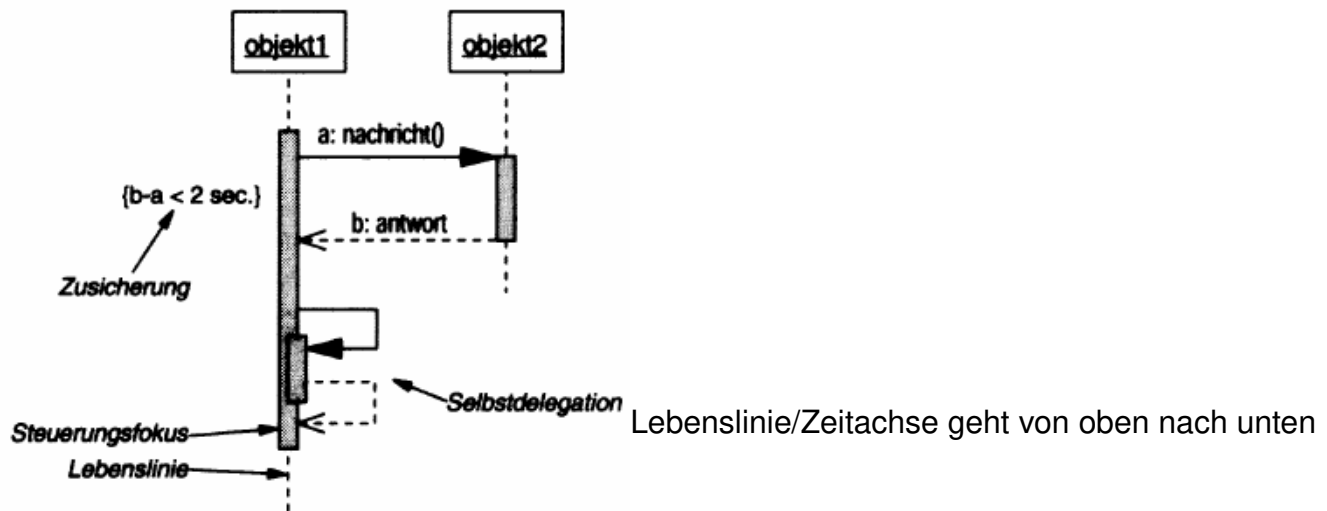
Dieses Diagramm als Klassendiagramm

- Die Objekte würde man so als Klassen übernehmen
- nachricht1() steht in Klasse1 drin

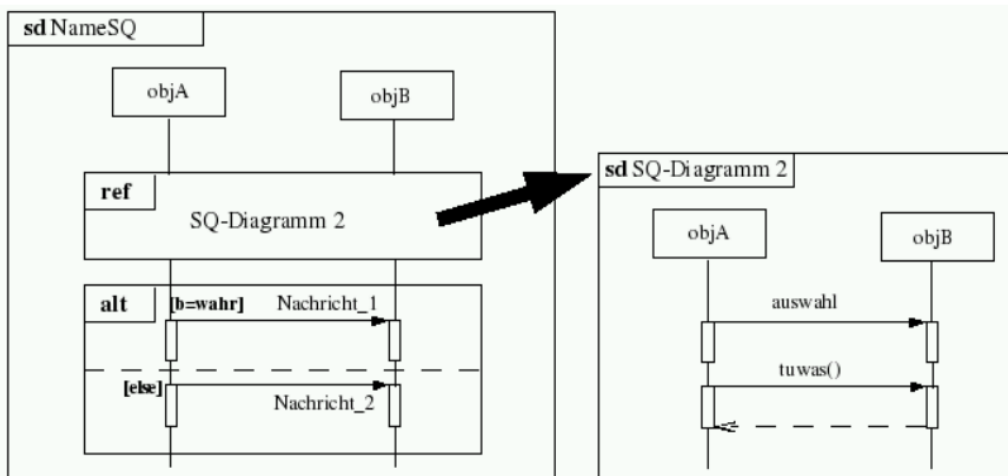
- `nachricht2()`, `nachricht3()` stehen in Klasse2 drin
- Lebenslinie wird zuerst `nachricht1()` nach Klasse1 aufrufen

Dieses Diagramm als Sequenzdiagramm: siehe Foto

2.5 Sequenzdiagramm



Mit Verfeinerung



3. SW-Qualitätsmanagement

Qualitätsmanagement soll ein Projekt mit Qualität zum Erfolg führen.

3.1 Software-Qualitätsmerkmale

ISO/IEC 25000: Produktmerkmale

Äußere Qualitätsmerkmale

- Funktionalität: inwieweit besitzt die Software die geforderten Funktionen
 - Angemessenheit: Eignung von Funktionen für spezifizierte Aufgaben
 - Richtigkeit: Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen
 - Interoperabilität: Fähigkeit, mit vorgegebenen Systemen zsmzuwirken
 - Sicherheit: Fähigkeit, unberechtigten Zugriff, sowohl versehentlich als auch vorsätzlich, auf Programme und Daten zu verhindern
 - Ordnungsmäßigkeit: anwendungsspezifische Normen oder Vereinbarungen oder gesetzliche Bestimmungen erfüllen
- Zuverlässigkeit: ein bestimmtes Leistungsniveau unter bestimmten/erschweren Bedingungen über einen bestimmten Zeitraum aufrechterhalten
 - Reife: Geringe Versagenshäufigkeit durch Fehlerzustände
 - Fehlertoleranz: spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle bewahren
 - Wiederherstellbarkeit: bei Versagen Leistungsniveau wiederherzustellen
 - Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Zuverlässigkeit erfüllt
- Effizienz: Verhältnis zwischen Leistungsniveau der Software und eingesetzten Betriebsmitteln
 - Zeitverhalten: Antwort- und Verarbeitungszeiten sowie Durchsatz bei der Funktionsausführung
 - Verbrauchsverhalten: Anzahl und Dauer der benötigten Betriebsmittel bei der Erfüllung der Funktionen
 - Ressourcenverbrauch: CPU-Zeit, Festplattenzugriffe, Netzwerkübertragungsmenge
 - mit unterschiedlichen Ressourcen wird unterschiedlich gut umgegangen

- Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Effizienz erfüllt
- Benutzbarkeit: Aufwand zur Benutzung, bekommt der Benutzer mit
 - Wie ist es an den Menschen angepasst? (Usability, Ergonomie)
 - Verständlichkeit: Aufwand für Verständnis von Konzept und Anwendung
 - Erlernbarkeit: Aufwand für Erlernen der Anwendung für erstmalige Benutzung
 - Bedienbarkeit: Aufwand für Bedienen der Anwendung (Hotkeys)
 - Attraktivität: Anziehungskraft gegenüber dem Benutzer
 - Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Benutzbarkeit erfüllt

Innere Qualitätsmerkmale

→ wenn Änderungen schwer sind oder Importierungen, sind es innere Merkmale da es nur der Entwickler bemerkt

- Wartbarkeit: Aufwand für Durchführung vorgegebener Änderungen an der Software
 - Bsp: Windows 7 wird nicht mehr unterstützt, Datenbank muss verändert werden
 - Separation of Concerns zielt darauf ab, darzustellen welche fachliche Anforderung zu welcher technischen gehört und wählt dafür gute Bezeichner
 - Analysierbarkeit: Aufwand für Diagnose von Mängeln oder Ursachen von Versage
 - Patterns als Entwurfsmuster können beim analysieren helfen da Elemente dann wiedererkannt werden können und besser gearbeitet werden kann
 - Änderbarkeit: Aufwand zur Ausführung von Veränderungen
 - Stabilität: Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen
 - Testbarkeit: Aufwand zur Prüfung der geänderten Software
 - Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Änderbarkeit erfüllt
- Übertragbarkeit: Aufwand für Übertragung der Software in eine andere Umgebung
 - Anpassbarkeit: Aufwand zur Anpassung an verschiedene Umgebungen anzupassen
 - Installierbarkeit: Aufwand zum Installieren in festgelegter Umgebung

- Koexistenz: Fähigkeit der Arbeit neben anderen Systemen mit ähnlichen oder gleichen Funktionen
- Austauschbarkeit: Möglichkeit der Verwendung der Software in anderer Umgebung jeder Software zu verwenden, sowie der dafür notwendige Aufwand
- Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Übertragbarkeit erfüllt

Merkmale der Prozessqualität

Prozess-Reifegrad

- Reifegradmodell

Terminplanung

- Termineinhaltung
- Güte der Abschätzung

Budgetplanung

- Budgeteinhaltung
- Güte der Abschätzung

Produktivität

- Ergebnis - Aufwand

Aufgaben-Koordination

- Anteil Leerlaufzeiten
- Grad Paralleltätigkeiten

Organisation und Kommunikation

- Regelungsgrad
 - die Effizienz ist schwach bei zu wenigen Regelungen und bei zu vielen. Man muss ein gesundes Maß finden
- Informationsweg-Länge
- Spezialisierungsgrad
- Kommunikationsaufwand
 - je mehr Personen beteiligt sind, desto mehr Kommunikationswege gibt es

Erfahrungsmanagement

- Einarbeitungsaufwand

- Fehlervermeidung/Anteil
- Rework: etwas nochmal machen
- Optimierungsgeschwindigkeit

Kundenzufriedenheit

Mitarbeiterzufriedenheit

Fehlerfortpflanzung

- Fehlererkennung häufig erst an Folgewirkung
- Rückverfolgung und Ursachenerkennung bei komplexen Systemen schwierig
 - deutlich aufwendiger als Behebung
- Fehler pflanzen sich in Folgeschritten fort:
 - Entwicklungsschritte Vorläuferprodukte zu wiederholen „Rework“, die Arbeitsschritte bauen auf einander auf und müssen deshalb wiederholt werden
 - Je später erkannt, umso umfangreicher Rework, also möglichst früh erkennen, dann baut sich nicht so viel Arbeit auf
- Fehlerverweilzeit senken = Produktivität erhöhen
- Aufwand für Fehlerbehebung wächst bei späterer Erkennung im Entwicklungsprozess
 - Anforderungsanalyse
 - Entwurf
 - Implementierung
 - Integration
 - Inbetriebnahme
 - Betrieb

Qualitätsmanagementmaßnahmen

Konstruktive Maßnahmen (Beim erstellen)

Kommen meist zu kurz, sind aber viel wichtiger, weil man die Software dann gleich richtig entwickelt

- Technisch-konstruktive Maßnahmen
 - Methoden- und Werkzeuganwendung (Softwareengineering)

- Programmiersprachen
- Organisatorische Maßnahmen
 - Projektmanagement (z.B. Pläne und Koordinierung)
 - Konfigurationsmanagement (z.B. Schutz vor Veränderungen und Inkonsistenzen)
 - ist wichtig um Fehler zu vermeiden
 - Vorgehensmodell
 - Welche Schritte/Dokumente/Beteiligte/Teams gibt es
- Psychologisch orientierte Maßnahmen sind sehr wichtig denn wenn Motivation erstmal weg ist, kommt sie nicht wieder
 - Schulungen (z.B. zu Qualitätsmaßnahmen und Zielen)
 - Motivationsfördernde Maßnahmen (z.B. Qualitätszirkel)
 - Kommunikationsverbessernde Maßnahmen

Analytische Maßnahmen (Nach dem Erstellen)

Qualität prüfen und bewerten

- Ziele: alle Beteiligten haben unterschiedliche Wünsche/Ziele
 - Kunde: erfüllt Produkt meine Anforderungen?
 - Projektleiter: Wie ist Qualität der Zwischen- und Endprodukte?
 - Entwickler: Habe ich gute Arbeit geleistet?
- Verifikation: Übereinstimmung der Ergebnisse einer Phase mit der vorigen
- Validation: Übereinstimmung der Ergebnisse mit Anforderungen
 - Frühzeitig analysieren: Risiken und Rework-Aufwand reduzieren
- statisch z.B. Review oder Inspektion
- dynamisch z.B. Tests

Statische und Dynamische Maßnahmen – Gegenüberstellung

- Statische: Review
 - Ablauf: Prüfobjekt liegt vor, wird begutachtet

- für Produkte: Alle Arten (Dokus, Architekturbeschreibung)
- In Phasen: Frühzeitig möglich
- Fehler, Mängel: auch innere Qualitätsmerkmale: Wartbarkeit etc.
- Effizienz: hoch
- Dynamische: Tests
 - Ablauf: Prüfobjekt wird ausgeführt, Verhalten beobachtet
 - für Produkte: nur ausführbare Produkte (Code)
 - In Phasen: erst nach teilweiser Implementierung möglich
 - Fehler, Mängel: nur Fehlverhalten (äußere Qualitätsmerkmale)
 - Effizienz: gering

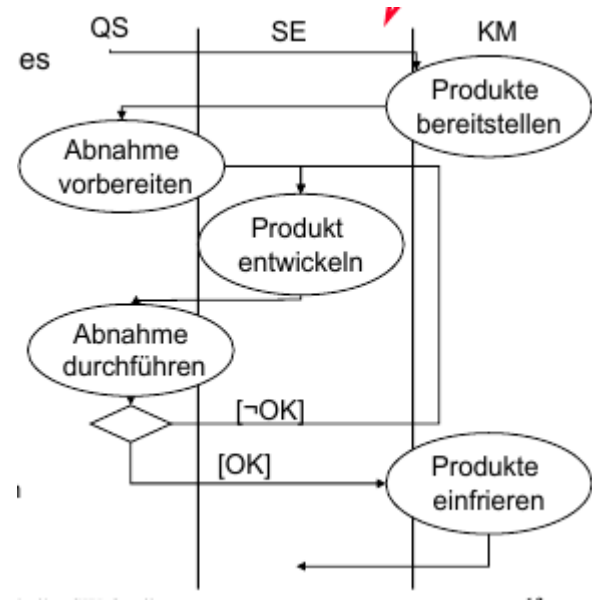
→ Tests sind also nicht genug, da sie nicht alle Produkte abdecken und erst spät durchgeführt werden können. Ist für Fehlerfortpflanzung eher schlecht. Reviews können wir direkt nach jeder Entwicklungsphase durchführen

Einbindung in Entwicklungsprozess

- Möglichkeiten für inkrementelles Vorgehen nutzen
- Abnahmekriterien, Testszenarien, Testfälle vor Entwicklung bereitstellen
- Wartbarkeit, Erweiterbarkeit, Flexibilität usw. als explizite Ziele aufnehmen
- Zugehörige Dokumente einbeziehen
- Schrittweise „Reifegrad“ erhöhen
 - Dokumentationsbasis verbessern
 - Architekturqualität verbessern
 - Modell-Qualität verbessern

V-Modell als Aktivitätsdiagramm mit zwei Swimlanes und drei Arbeitsbereichen:

- Qualitätssicherung
- Softwareentwicklung
- Konfigurationsmanagement



Ablauf eines Reviews

1. Planung: Teilnehmer Termin, Kriterien, Gegenstand
2. Individuelle Vorbereitung: Jeder teilnehmer prüft anhand der kriterien
3. Review Sitzung: gemeinsame Prüfung, Autor stellt vor, Experten fragen, Ergebnis: Mängelliste, es werden keine lösungen diskutiert
4. Nachbereitung: Autor behebt die Mängel
→ Dauer insgesamt: max 2Std

Peer Review: unter gleichen kollegen ohne projektleiter

→ Wirkprinzip: Vier augen sehen mehr als zwei, Kognitive Fähigkeiten der Beteiligten sind wichtig

Nebeneffekt: Einarbeitung, Know-How Transfer

Statische und Dyamische Maßnahmen – Gefundene Fehler

Review in Entwicklungsphase	gefundene Fehler
Review der funktionalen Spezifikation	2 ... 5 %
Review des Grobentwurfes Architektur	10 ... 16 %
Review des Feinentwurfs	17 ... 22 %
Review des Codes	20 ... 32 %
Modultest	12 ... 17 %
Funktionstest	10 ... 14 %
Komponententest	8 ... 14 %
Systemtest	0,5 ... 7 %

Testumgebung und Testdurchführung

Test im eigentlichen Sinne: Ausführung mit Testdaten

- Testrahmen erforderlich
 - Treiber (driver) zum Aufrufen des Testobjektes und zur Übergabe von Eingabedaten und Resultaten
 - Stellvertreter (stubs) zur Simulation der externen Funktionen

- Tests müssen wiederholbar sein (z.B. für Regressionstests) und dafür ist das Aufheben von Testdaten wichtig
- nachvollziehbar sein
- dokumentiert werden
- d.h. geplant und kontrolliert werden
- Aufwand von Tests in Relation zum Codierungsaufwand: 3:1

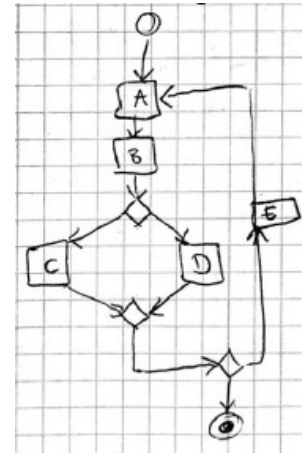
Testmethoden

- Beobachten des Verhaltens bei Ausführung
- Stichprobenverfahren, kein Nachweis der Korrektheit
- viele mögliche Kombinationen → viele Testfälle → Extrem hoher Aufwand
- Verringerung der Anzahl der Testfälle
- Durchführung von
 - Positivtests → erwartetes Verhalten
 - Negativtests → Wird unerlaubtes Verhalten verhindert?
 - Lasttests → Lasts steigen bis „Ende“, also bis es nicht mehr performant läuft
- Abdeckungsmaße zur Bewertung der Tests

Testabdeckungsmaße

- Anweisungsabdeckung
 - Jede Anweisung im Testobjekt mind. Einmal ausgeführt Maß (C0) = Anzahl der ausgeführten Anweisungen/Anzahl aller Anweisungen → **nicht ausreichend**
 - sehr aufwändig, müssen bspw. dafür Schelifen nur einmal durchlaufen werden → Grenzfälle werden nicht aufgezeigt
- Zweigabdeckung
 - Jede Bedingung, die zur Änderung des Ablaufs führt, soll mind. Einmal den Wert TRUE und einmal den Wert FALSE liefern. → **Mindesanforderung**
- Pfadabdeckung
 - Alle möglichen unterschiedlichen Abläufe des Testobjektes sollen in der Testphase einmal ausgeführt werden → **nicht realistisch**

- Zweigabdeckung/Pfadabdeckung
 - Zweig 1: ABC Zweig 2: ABD Pfad 1: ABCEABC
 - Pfadabdeckung nicht realistisch, da unendliche Pfade geprüft werden müssen



Gegenüberstellung Testtechniken

	Black Box „Funktionaler Test“	White Box „Strukturtest“
Ablauf	Testgegenstand mit Testdaten beschicken, Verhalten beobachten, Resultate mit Erwartungen vergleichen	
Quelle der Testdaten	Spezifikation, Schnittstellenbeschreibung	Struktur, Steuerflußanalyse
Art gefundener Fehler	Falsche Funktionalität, auch Entwurfsfehler, datenabhängige Fehler	Falsche Funktionalität, auch verborgene Funktionalität (Viren, Trojaner)
Anwendung in Phasen	Subsystem-, Integrations-, Abnahmetest	„Entwicklertest“ Klassen-, Modultest

Unterschied zu Black-box-Tests liegt vor allem in der Wahl der Testdaten

- Black-Box: typische Use-Case Daten
- White-Box: Daten, um bestimmte Abläufe zu überprüfen (Test der Struktur)

→ White-Box Tests ergänzen die Black-Box Tests

Testebenen

- Modultest/Klassentest – Testen im Kleinen
 - Verhalten von Funktionen, Programmteilen, Modulen
- Integrationstest – Testen im Großen
 - Montage der fertig getesteten Module
 - Test des Zsmspiels der Module (Schnittstellentest)

- Systemtest – Test des kompletten Systems
 - Außenverhalten des Gesamtsystems
 - Laufzeit- und Stresstests
 - Bewertung der Benutzungsfreundlichkeit
 - ähnlich zu Abnahme- und Akzeptanztests

Integration der Testtechniken

1. funktionsorientierter Klassentest
 - Instrumentierung für Zweigüberdeckung (Anweisungen zur Messung → z.B. Konsolenausgabe)
 - Äquivalenzklassen-Tests nach Spezifikation
 - Kontrolle der erreichten Zweigüberdeckung (typisch: 70-80%)
2. strukturorientierter Klassentest
 - Testfälle für noch nicht durchlaufende Zweige
 - garantierte Erfüllung der Minimalbedingungen

Äquivalenzklassen

- Partitionierung des Eingangsraumes einer Methode durch Bildung von Äquivalenzklassen für die Eingangswerte
- Werte aus einer Äquivalenzklasse
 - verursachen identisches funktionales Verhalten
 - testen identische spezifizierte Funktionen

→ Verringerte Anzahl der Testfälle, gleiche Testabdeckung

Regeln zur Bildung der Äquivalenzklassen:

- Positiv-Klassen: Testfälle aus Kombination möglichst vieler Testdaten aus gültigen Äquivalenzklassen
- Negativ-Klassen: Testfälle aus
 - einem Testdatum einer ungültigen Äquivalenzklasse
 - weiteren Testdaten ausschließlich aus gültigen Äquivalenzklassen

Beispiel für abgeleitete Testfälle

Portorechner eines Paketdienstes

Päckchen:

Gewicht > 0,1 .. 2 kg;

Preis 4,99 EUR

Paket:

Gewicht > 2 .. 15 kg;

Preis 6,99 EUR



4 Äquivalenzklassen

zuLeicht: Gewicht $\leq 0,1$ kg

Päckchen: Gewicht > 0,1 .. ≤ 2 kg;

Paket: Gewicht > 2 .. ≤ 15 kg;

zuSchwer: Gewicht > 15 kg

4 Testfälle:

zuLeicht: Gewicht = 0,1 kg: ungültig

Päckchen: Gewicht = 1,4 kg: gültig

Paket: Gewicht = 11 kg: gültig

zuSchwer: Gewicht = 15,01 kg: ungültig

- Nur vier statt unendlich viele Testfälle nötig durch Nutzung von Äquivalenzklassen
→ zumindest wenn das Gewicht der einzige Testparameter ist
- Wenn in der Klausur nach Testfällen für Äquivalenzklassen gefragt wird, müssen konkrete Testwerte angegeben werden
- Möglich Test von Grenzwerten

Grenzwertanalyse

- Fehlerwahrscheinlichkeit höher an den Grenzen einer Äquivalenzklasse als innen
- Testen an Grenzen erfolgreicher
- Testfälle für Grenzen gültiger Äquivalenzklasse, ungültiger Äquivalenzklassen

Profiler

- Programmierwerkzeug um Laufzeitverhalten zu analysieren
- Messung und Ausgabe von Verweilzeit in einzelnen Methoden
- Angabe, welche Zweige durchlaufen worden sind

Beispiel für abgeleitete Testfälle

→ Hier Black-Box Tests

Bußgeld Geschwindigkeitsüberschreitung

Strafen bei
Geschwindigkeitsüberschreitung Außerorts:

bis 10 km/h: 10 €, 0 Monate, 0 Punkte
 11 bis 15 km/h: 20 €, 0 Monate, 0 Punkte
 16 bis 20 km/h: 30 €, 0 Monate, 0 Punkte
 21 bis 25 km/h: 70 €, 0 Monate, 1 Punkt
 26 bis 30 km/h: 80 €, 0 Monate, 1 Punkt
 31 bis 40 km/h: 120 €, 0 Monate, 1 Punkte
 41 bis 50 km/h: 160 €, 1 Monat, 2 Punkte
 51 bis 60 km/h: 240 €, 1 Monat, 2 Punkte
 61 bis 70 km/h: 440 €, 2 Monate, 2 Punkte
 über 70 km/h: 600 €, 3 Monate, 2 Punkte

Erstellte Testfälle

v = -1 (Negativ-Test)
 v = 1, v= 9 (nahe Grenze)
 v = 11, v= 15 (nahe Grenze)
 v = 16, v= 19 (nahe Grenze)
 v = 21, v= 25 (nahe Grenze)
 v = 26, v= 30 (nahe Grenze)
 v = 31, v= 39 (nahe Grenze)
 v = 41, v= 49 (nahe Grenze)
 v = 51, v= 59 (nahe Grenze)
 v = 61, v= 69 (nahe Grenze)
 v = 70, v= 110 (nahe Grenze)

Entscheidungstabellen – basierter Test

Entscheidungstabelle für:

- Aufstellung der möglichen Kombinationen
- Prüfung der Vollständigkeit
- Abgrenzung von Äquivalenzklassen

vier Teilbereiche:

- Bedingungen: mögliche Zustände
- n mögliche Bedingungen
- 2ⁿ mögliche Bedingungskombinationen, Regeln
- Aktionsanzeiger: Zuordnung der Aktivitäten zu Bedingungskombinationen

Entscheidungstabellen-basierter Test – Vorgehen

→ das Aufstellen, Vereinfachen und Prüfen schon während der Anforderungsbeschreibung anlegen

Aufstellung Entscheidungstabelle:

1. Bedingungen festlegen

2. Aktionen angeben
3. Regeln und Aktionsanzeiger setzen

Vereinfachen und prüfen:

4. Konsolidierung der Entscheidungstabelle à je Gruppe zusammengefasster Regeln eine Äquivalenzklasse
5. Prüfung auf Widerspruchsfreiheit und Vollständigkeit

Testfälle bilden:

6. je Äquivalenzklasse einen Testfall bilden

Entscheidungstabelle Beispiel:

Bereichsleiter Schmid möchte eine Mitarbeiterin im Krankenhaus besuchen. Er informiert sich telefonisch an der Information über die Besuchsmöglichkeiten und erhält folgende Antwort: Die Patientin kann ohne Einschränkungen innerhalb der Besuchszeit besucht werden, sofern keine ansteckende Krankheit vorliegt und sie kein Fieber hat. Außerhalb der Besuchszeit ist in diesem Fall eine Schwester als Begleitung erforderlich. Falls die Patientin eine ansteckende Krankheit hat, werden Besuche ganz abgelehnt. Wenn die Krankheit nicht ansteckend ist, die Patientin aber Fieber hat, darf der Besuch innerhalb der Besuchszeit maximal 30 Minuten betragen, außerhalb der Besuchszeit dürfen Patienten mit Fieber nicht besucht werden.

Bedingungen:

- | Patientin hat ansteckende Krankheit (j/n)
- | Besuch innerhalb Besuchszeit (j/n)
- | Patientin hat Fieber (j/n)

Aktionen:

- | Besuchszeit maximal 30 Minuten
- | Besuch ablehnen
- | Besuch mit Begleitung einer Schwester
- | Normalbesuch in Besuchszeit

3 Bedingungen → $2^3 = 8$
Bedingungskombinationen

Bedingungen	1	2	3	4	5	6	7	8
Ansteckende Krankheit (j/n)								
innerhalb Besuchszeit (j/n)								
Fieber (j/n)								
Aktionen								
Besuchszeit maximal 30 Minuten								
Besuch ablehnen								
Mit Begleitung einer Schwester								
Normalbesuch in Besuchszeit								

Bedingungen:

- | Patientin hat ansteckende Krankheit (j/n)
- | Besuch innerhalb Besuchszeit (j/n)
- | Patientin hat Fieber (j/n)

Aktionen:

- | Besuchszeit maximal 30 Minuten
- | Besuch ablehnen
- | Besuch mit Begleitung einer Schwester
- | Normalbesuch in Besuchszeit

Bedingungen	1	2	3	4	5	6	7	8
Ansteckende Krankheit (j/n)	j	j	j	j	n	n	n	n
innerhalb Besuchszeit (j/n)	j	j	n	n	j	j	n	n
Fieber (j/n)	j	n	j	n	j	n	j	n
Aktionen Besuchszeit > 30 Minuten					x			
Besuch ablehnen	x	x	x	x			x	
Mit Begleitung einer Schwester								x
Normalbesuch in Besuchszeit						x		

Bedingungen:

- | Patientin hat ansteckende Krankheit (j/n)
- | Besuch innerhalb Besuchszeit (j/n)
- | Patientin hat Fieber (j/n)

Aktionen:

- | Besuchszeit maximal 30 Minuten
- | Besuch ablehnen
- | Besuch mit Begleitung einer Schwester
- | Normalbesuch in Besuchszeit

Bedingungen	1	2	3	4	5	6	7	8
Ansteckende Krankheit (j/n)	j	j	j	j	n	n	n	n
innerhalb Besuchszeit (j/n)	j	j	n	n	j	j	n	n
Fieber (j/n)	j	n	j	n	j	n	j	n
Aktionen Besuchszeit > 30 Minuten					x			
Besuch ablehnen	x	x	x	x			x	
Mit Begleitung einer Schwester								x
Normalbesuch in Besuch								

Entscheidungstabellen: informat.de]

konsolidieren:
gleiche Aktion, „don't
care“ Bedingungen

nicht konsolidierbar:
keine „don't care“
Bedingungen

Bedingungen:

- Patientin hat ansteckende Krankheit (j/n)
- Besuch innerhalb Besuchszeit (i/n)

Mind. 1 Testfall je Äquivalenzklasse;

Aktionen:

- Besuchszeit maximal 30 Minuten
- Besuch ablehnen
- Besuch mit Begleitung einer Schwester
- Normalbesuch in Besuchszeit

Bedingungen	1 bis 4 konsolidiert	5	6	7	8
Ansteckende Krankheit (j/n)	j	n	n	n	n
innerhalb Besuchszeit (j/n)	„don't care“	j	j	n	n
Fieber (j/n)	„don't care“	j	n	j	n
Aktionen					
Besuchszeit > 30 Minuten		x			
Besuch ablehnen	x			x	
Mit Begleitung einer Schwester					x
Normalbesuch in Besuchszeit			x		

Erwartetes Ergebnis

Softwaretechnik - SW-Qualitätsmanagement

33

Bedingungen:

- Patientin hat ansteckende Krankheit (j/n)
- Besuch innerhalb Besuchszeit (i/n)

Mind. 1 Testfall je Äquivalenzklasse;

bei „don't care“ Werte nahe Grenze wählen

Aktionen:

- Besuchszeit maximal 30 Minuten
- Besuch ablehnen
- Besuch mit Begleitung einer Schwester
- Normalbesuch in Besuchszeit

Bedingungen	4 konsolidiert	5	6	7	8
Ansteckende Krankheit (j/n)	j	n	n	n	n
innerhalb Besuchszeit (j/n)	j	j	j	n	n
Fieber (j/n)	j	j	n	j	n
Aktionen					
Besuchszeit > 30 Minuten		x			
Besuch ablehnen	x			x	
Mit Begleitung einer Schwester					x
Normalbesuch in Besuchszeit			x		

Erwartetes Ergebnis

Softwaretechnik - SW-Qualitätsmanagement

34

→ Nur noch fünf Testfälle benötigt, da die Fälle 1 bis 4 zu einem Fall konsolidiert worden sind

Zustandsbasierter Test

Anwendungsgebiet: Test von Methodensequenzen, die als Zustandsautomat spezifiziert werden.

→ ist Black-Box-test, da Spezifikation betrachtet wird

Abdeckungsmaße:

- typische Folgen von Zuständen
- alle Zustände (Minimum)
- alle Zustandsübergänge einmal durchlaufen (besser)
 - Testfälle als Folge von Ereignissen → decken Zustände / Zustandsübergänge ab
- typische Folgen von Zustandsübergängen
- alle Ereignisse (noch besser)
- ungültige Zustandsübergänge

Use-Case-basierter Test

- Interaktionen zwischen Benutzer und System führt zu Reaktionen
- Vorbedingungen für Eintreten des Use Case
- Nachbedingungen nach Ende des Use Case (soweit sichtbar)
- Mehrere Szenarien: Normalfall, Ausnahmefall, Fehlerfall

→ Für Prüfung von Abläufen (z.B. Workflows)

→ Für Prüfung des Zusammenspiels von Komponenten

→ Meist in Kombination mit anderen Methoden

Testen objektorientierter Software

- Kapselung von Objekten behindert das Testen
- Fehlerbehandlung
 - gegenüber Benutzer sinnvoll
 - gegenüber Test-Software nicht sinnvoll, da
 - Fehler maskiert werden
 - Testbarkeit reduziert wird (Testbarkeit beschreibt Aufwand um Tests durchzuführen und nicht direkt, ob Tests durchgeführt werden können)

→ Fehlerwirkung forcieren

→ Zusicherungen erhöhen Test- und Beobachtbarkeit

Ziel: Kein zusätzlicher Code für Fehlerbehandlung

Klassentest

- Testen der Methoden eines Objekts, welches Instanz der zu testenden Klassen ist.
- Im Vergleich zu Modulen aus imperativen Programmierparadigmen zeigen Methoden
 - eine einfache Kontrollstruktur.
 - eine starke Verflechtung über die Attribute der Klasse.
- Eine Methode sollte
 - eine gewisse Mindestkomplexität aufweisen.
 - keine zu große Abhängigkeit von anderen Teilen des Objektes oder zu anderen Objekten besitzen.

Abdeckungsmaße	Whitebox	Blackbox	Umfang(bzw. Anzahl der Tests)
Zweigabdeckung	x		mittel
Anweisungsabdeckung	x		mittel
Äquivalenzklassen-Abdeckung		x	gering
Use-Case-Abdeckung		x	sehr gering
Zustandsübergangs-Abdeckung	(x)	x	gering
Bedingungskombinations-Abdeckung	(x)	x	extrem hoch
Zustands-Abdeckung	(x)	x	sehr gering

→ (x) bedeutet „falls interne Zustände betrachtet werden

→ Pfadabdeckung gibt es eigentlich nicht

Voraussetzung für den Test einer einzelnen Methode

Die Methode sollte eine gewisse Mindestkomplexität aufweisen und keine zu große Abhängigkeit von anderen Teilen des Objektes oder zu anderen Objekten besitzen.

Ziel des OO-Testens ist nicht das Testen der Methoden an sich, sondern das Testen von deren Zusammenspiel über die Attribute einer Klasse.

Zusammenfassung Testen objektorientierter Software

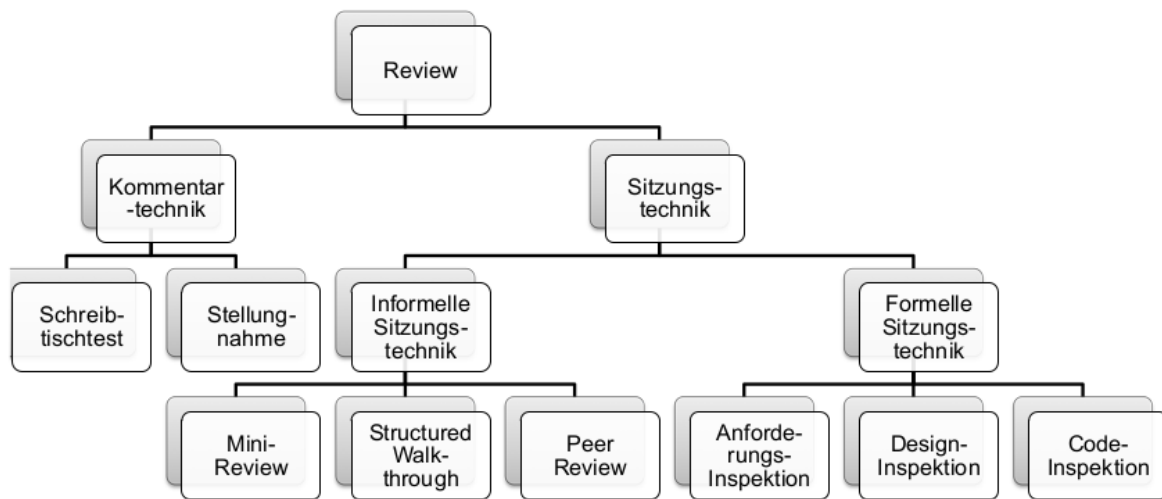
- Gleiche Testtechniken wie bei imperativen Programmiersprachen

- funktionaler Test
- kontrollflußorientierter Test
- Unterschiedliche Vorgehensweise beim OO-Testen
- Integration der Testtechniken zur simultanen Testdurchführung

3.3 Analytische statische Maßnahmen

- Audit
 - Prüfung der Einhaltung von Vorgaben
 - Prüfung der Sinnhaftigkeit, Wirksamkeit
- Review, Inspektion
 - Formaler Analyseprozeß vor Gutachtern
- Walkthrough
 - Durchspielen von Abläufen
 - ähnlich wie Review, aber es werden einzelne Abläufe nach und nach geprüft; bei Review wird eher das Gesamtsystem betrachtet
- Korrektheitsbeweiser
 - Gegenüber formaler Spezifikation
- Symbolische Programmausführung

Statische Maßnahmen: Review-Techniken



Statische Maßnahme: Review, Inspektion

Insbesondere Peer Review:

- Gemeinsame Analyse mit Fragen und Erläuterung durch Autor
- Fremde Fragestellungen: neue Erkenntnisse
- Kognitive Fähigkeiten vereinigt - 4 Augen
- Anerkennung der Ergebnisse und Verbesserungshinweise
- Effektiver als Test: mehr Fehler pro Zeiteinheit
- Motivation für Dokumentation und Programmierstil
- häufig erkennt Programmautor beim Erklären seines Codes selbst Verbesserungsvorschläge

→ Jedoch: Erfolg personen- und klimaabhängig

Vorteile

- ca. 80% der Softwarefehler entdeckt
- Mängel entdeckt: Wartbarkeit, Tricks
- einfach durchzuführen, keine spezielle Software benötigt
- Collective Code Ownership statt Einzelkämpfertum
- Unterstützt praktischen Wissenstransfer zwischen erfahrenen und unerfahrenen Mitarbeitern
 - Entwurfsmethoden
 - Defensiver Programmierstil
- Qualitätskriterien schon während des Implementierungsprozesses überprüft

Zeitpunkte Inspektion

- vor Freigabe
- sobald eine gewisse Qualität garantiert werden muss
 - vor dem Besuch des Kunden
 - bevor der Autor in Urlaub geht
- auf Wunsch eines Mitarbeiters, falls selbst unsicher über Qualität seines Codes

- nach ca. 2, höchstens 4 Implementierungswochen (bei Anfänger öfters, bei Profis seltener)

→ Zeitpunkt der Inspektion frühzeitig bekannt geben, um Vorbereitung zu ermöglichen

Inspektion – Ablauf

Vorbereitung:

- Verstehen, Formale Fehler anhand Checkliste sammeln

Sitzung:

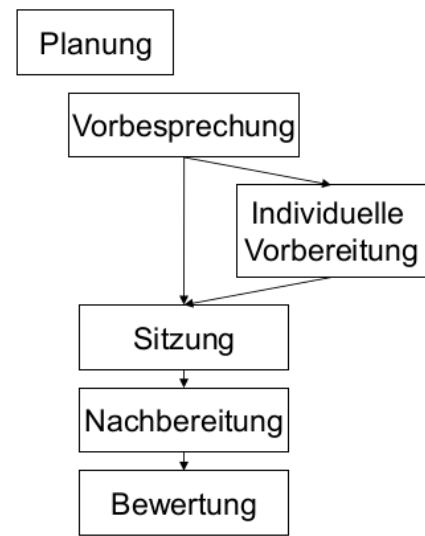
- Mängel nur entdecken und sammeln, nicht diskutieren
- Dauer max. 2 h
- Konstruktives Klima

Nachbereitung:

- Autor behebt Mängel

Bewertung:

- Prüfung der Behebung



Review-Kriterien sind zur Anfertigung einer Check-Liste nötig

- Zielorientiertes Qualitätsmanagement: Anforderungen der Stakeholder als Quelle
- Auftraggeber, Kunden, Benutzer:
 - äußere Q-Merkmale, wie Funktionalität, Benutzbarkeit, Effizienz
- Entwickler, Management des Auftragnehmers:
 - Prozessmerkmale wie Effizienz, Fehlervermeidung
 - innere Q-Merkmale wie Wartbarkeit, Verständlichkeit
- Konzentration auf wenige Kriterien für effiziente Reviews
 - Priorisierung der Kriterien wichtig
 - Erweiterung nach Auftreten von Problemen
- Ziele von Reviews
 - Projektziele erfüllen, Collective Code Ownership verbessern
 - Know-How-Transfer, Projekteffizienz erhöhen
 - Produktivität erhöhen
 - Quality Awareness steigern

Checkliste und Protokoll

Checkliste

- Guter Codierstil mit Facetten:
 - Do's und Don'ts der Sprache: Fehler und Mängel
 - Lesbarkeit
 - Namenskonventionen (allgemein, sprach-, projektspezifisch)
 - Defensiver Programmierstil → mögliche Fehlerfälle abdecken, ggf. Aktualparameter auf einen Wertebereich prüfen
 - Verständlichkeit: Kompliziertes vermeiden
- Portabilität, Nebenläufigkeit, Fehlerbehandlung

→ Protokoll als Liste gefundener Fehler und Mängel

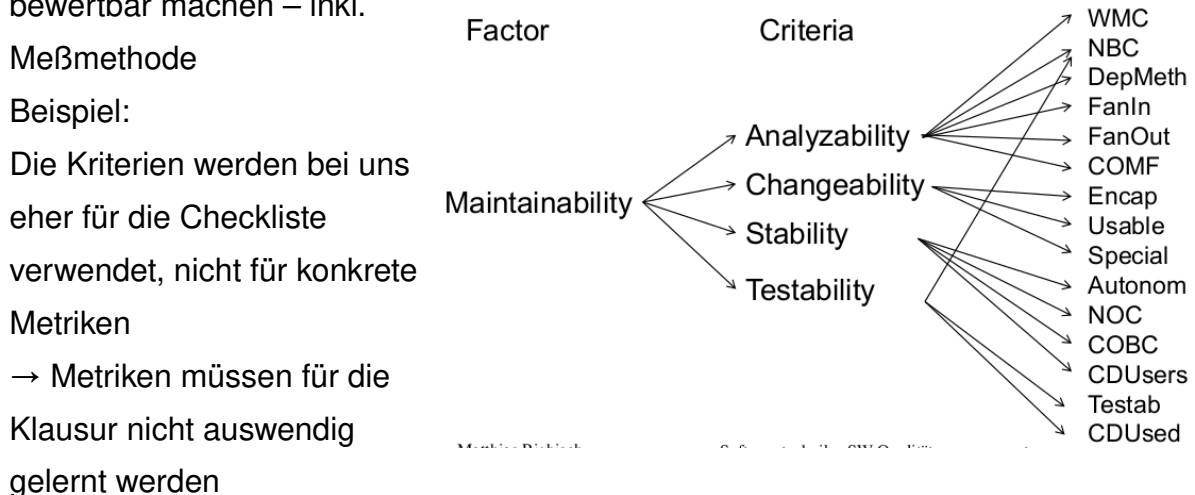
Werkzeugunterstützung: Findbugs, Checkstyle, PMD

- Werkzeuge zur maschinelle Prüfung von Stil-Richtlinien
 - Sun-Richtlinien für Java

- Possible bugs - empty try/catch/finally/switch statements
- Dead code - unused local variables, parameters and private methods
- Suboptimal code - wasteful String/StringBuffer usage
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- Duplicate code - copied/pasted code means copied/pasted bugs

Factor-Criteria-Metrics FCM

- Quality Factors: benutzerorientierte Qualitätsmerkmale der Software.
- Quality Criteria:
 - Teilmerkmale, die Qualitätsmerkmale verfeinern oft Einfluss auf mehrere Qualitätsmerkmale
 - hilfreich bei Priorisierung
- Quality Metrics: Qualitätsindikatoren / Qualitätsmetriken, die Teilmerkmale meß- und bewertbar machen – inkl.



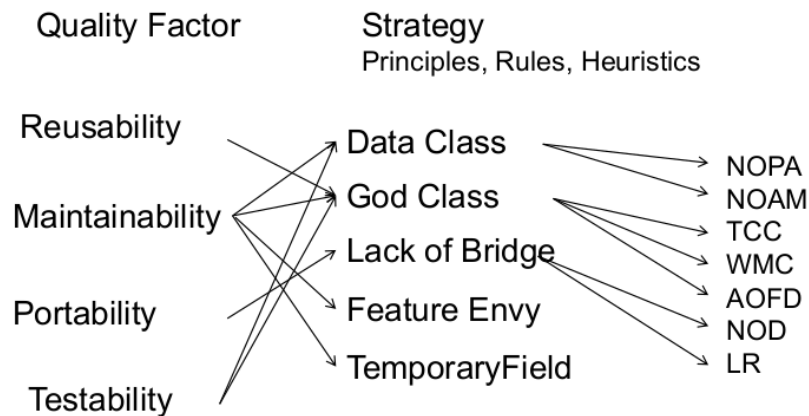
- Beispiel:
- Die Kriterien werden bei uns eher für die Checkliste verwendet, nicht für konkrete Metriken
- Metriken müssen für die Klausur nicht auswendig gelernt werden
- WMC = wie groß sind die Methoden
- Encap = Grad der Kopplung

Factor-Strategy-Modell

- Quality Factors: Qualitätsmerkmale wie bereits behandelt
- Strategy:

- Erkennungsstrategie, zugeordnet zu Factor
- Aufdeckung eines Entwurfsproblems
- Ansätze für Verbesserung
- Kriterien für Inspektion
- Kriterien für Refactoring: Bad Smells

- Beispiel:



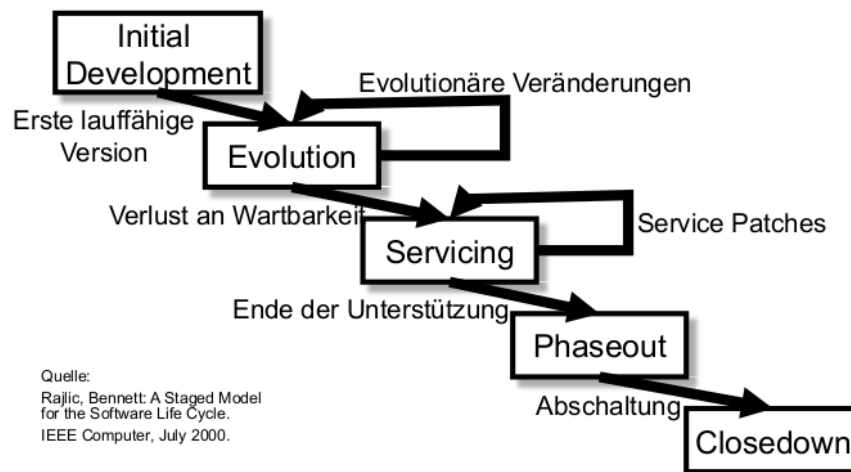
Beispiele für Styleguides und Checklisten

- Sprachspezifisch:
 - Java Styleguide von Sun, bei Oracle:

<http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>
- Vorgaben für Namen
 - Styleguide der Fa. Geosoft <http://geosoft.no/development/javastyle.html>
 - bei starkem technischem Bezug: Namen auf jeden Fall Englisch
 - bei starkem fachlichen Begriff: ggf. Deutsch falls durch Verwirrung entstehen könnte
- Styleguide für die Verwendung von Bibliotheken, APIs:
 - Android Styleguide <https://sites.google.com/a/android.com/opensource/submit-patches/code-style-guide>
 - Verwendung von GIT <http://www.jnode.org/book/export/html/3> (ab Git Etiquette bis Configuration Files)

4.1 Wartung und Reengineering

Staged Model



Tst ein Zustandsdiagramm.
Alles an Entwicklung ist im
ersten Kasten. Alles nach dem
ersten Zustandsübergang ist
nur noch Wartung etc.
Evo Änderungen: weitere
Änderungen sind möglich
(also mit Struktur-
anpassungen) und

Änderungen haben keine negativen Auswirkungen

Service Patches: Funkionalität erreichen auch ohne Strukturanpassungen, sind
notgedrungen, damit es gerade noch so funktioniert.

Man will immer im Zustand der Evolution zu bleiben. Mit Reengineering kann man die Leiter
wieder hochklettern.

Ablösung eines Systems durch Neuentwicklung

- Hohe Kosten, ohne dass zusätzliche Funktionalität entsteht
- Risiko der Unreife des neuen Systems
- Risiko der Kosten- und Terminüberschreitung
- Anforderungen (teilweise) unbekannt → Auftragsvergabe nur schwierig möglich
- tatsächliche Verwendung des (Alt-)Systems ist nicht vollständig bekannt
 - manchmal wird Software für andere Dinge verwendet, als sie eigentlich entwickelt wurde

Begriff: Software-Wartung (Maintenance)

- ANSI/IEEE Standard 729-1983:
 - "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment."

- Üblicher Sprachgebrauch:
 - Änderungen am System nach dessen Auslieferung. Schließt Anpassungen an neue Anforderungen ein
- um mit existierender Software umzugehen

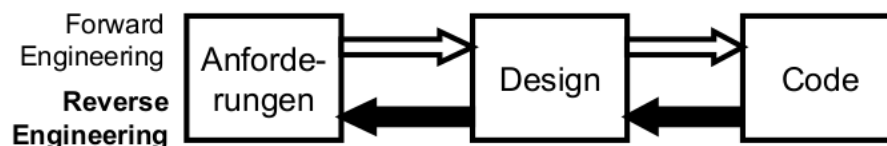
Strukturverlust bei Wartungs-/ Änderungsaktivitäten

- hohe Komplexität
- Verständlichkeit schwierig/aufwendig
- hoher Zeitdruck
- hohes Risiko führt zum Unterbleiben von Strukturanpassungen
 - Vermischung von Zuständigkeiten
 - große Klassen, Methoden

→ Schlechte Verständlichkeit und Verlust der Architekturqualität z.B. Entwurfsprinzipien verletzt

Begriff: Reverse Engineering

- Identifikation der Systemkomponenten und deren Beziehungen. Ziel ist die Beschreibungen des Systems in einer anderen Form oder auf höherem Abstraktionsniveau.
- Aktivitäten zum Erhöhen des Verständnisses und Verbesserung von Wartbarkeit, Wiederverwendbarkeit, nach Inbetriebnahme eines Programmes
- ist der Rückweg



Begriff Geschäftskritische Softwaresysteme

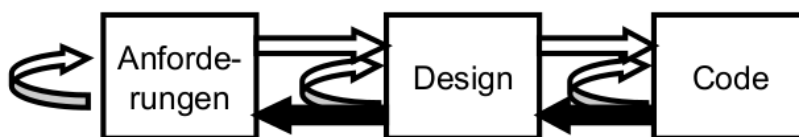
- Nichts privates sondern entscheidend für die wirtschaftliche Tätigkeit eines Unternehmens
 - wenn Firma Dienstleistungen oder Produkte anbietet wo die Software drin enthalten ist

- Beispiel: Paketdienst mit Ein- und Ausgabestellen, Sortierung, Planung etc würde ohne Software nicht funktionieren
- Geschäftsprozesse sind von Software abhängig
- hohe Anforderungen an Zuverlässigkeit und Korrektheit
- lange „Lebensdauer“ (Nutzungsdauer)
- meist hohe Komplexität
- Änderungen mit großer Bedeutung – sonst keine Nutzbarkeit
 - Optimierung von Geschäftsprozessen
 - Öffnung von Systemen: Mitwirkung von Kunden, Unterauftragnehmer
 - Kopplung mit externen Systemen (Datenaustausch)
 - Organisatorische Änderungen: Split oder Zusammenlegung von Unternehmen (aus Steuerrechtlichen Gründen oder zu großer Verantwortung)
 - Änderungen gesetzlicher Rahmenbedingungen
 - Änderungen der technischen Plattform

Begriff: Reengineering (Renovation, Reclamation)

- Untersuchung (Reverse Engineering) und Änderung des Systems, um es in neuer Form zu implementieren.
- Keine Änderung der Funktionalität
- Erweitertes Reengineering: analysieren / restrukturieren, um dann Funktionalität zu ändern
- Wenn mans nicht macht laufen Geschäftsprozesse nicht mehr und die Software ist nicht mehr anwendbar
-

Forward & Reverse Engineering & Restrukturierung



Begriff: Technical Debt

Kurzfristige Lösungen oft mit langfristigen Nachteilen

- Feature durch Lösung mit geringstem Aufwand realisiert
- Z.B. enge Kopplung oder ungeeignetes Framework in Kauf genommen
- Nachteile langfristig: Kosten für Änderungen hoch → Wartbarkeit verringert
- Technical Debts sind die unterbliebenen Strukturanpassungen und befinden sich im Servicing im Staged Model.

Kein Technical Debt:

- (Programmier-)Fehler
- noch nicht implementierte Features
- schlechter Entwicklungsprozess

Technical Debt als Unsichtbares

	sichtbar	nicht sichtbar
positiver Wert	Neue Features, neue Funtionalität	Architektur- und Struktur-Features
negativer Wert	Fehler und Defekte	Technical Debt

4.2 Refactoring

Definition

A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour

- Verbessern der Struktur
- Verringern des Wartungsaufwands
- Verbessern der Verständlichkeit
- Risiko ist hoch. Verringern durch:
 - Folge von kleinen Veränderungen und jede ist klar verständlich und kann rückgängig gemacht werden → in der Summe ist es dann eine große Veränderung
 - durch Tests absichern
 - klare Trennung von funktionalen Änderungen
 - Refactoring zuerst, existierende Tests können genutzt werden
 - funktionale Änderung anschließend, auf Basis verbesserter Struktur bzw. Verständlichkeit
 - Refactorings aus Katalog – erprobte, erfolgreiche Vorgehensweisen
 - Bad Smells als Indikator

„A poorly designed system is hard to change“

- Wartung des Systems
 - Anpassen an neue technische und fachliche Anforderungen
- Änderungen mit hohem Risiko verbunden
 - Fehler werden eingebaut, Änderungen unvollständig, Struktur zerfällt
- Software ist nicht verständlich
 - Schlechte Codestruktur
 - Hohe Komplexität
- Ziel: Schnelle und einfache Änderungen an der Software
- Deshalb: Umstrukturierung des Code

Bad Smells

- Mängel bezüglich Wartbarkeit, Verständlichkeit, Verletzung von Entwurfsprinzipien

- keine Funktionalen Fehler
- Wir verwenden sie als Entscheidungskriterien für Refactoring/Reengineering
 - Experten wären gut, sind aber selten verfügbar
 - Ästhetik als Entscheidungskriterium ungeeignet
 - Intuition als Entscheidungskriterium schlecht geeignet
 - Bad Smells aus Katalog von gesammeltem Wissen

Zuordnung von Qualitätsmerkmalen zu Bewertungskriterien und Bad Smells

Qualitätsmerkmal (und Verfeinerung)

- Reusability
 - Entwurfsziele: Einfachheit
 - Bewertungskriterien: Smell3, Guideline1, Guideline2
- Maintainability
 - Entwurfsziele: Lesbarkeit, Einfachheit, Modularisierung, Sprachstandards, Konformität zu Projekt-Regeln
 - Bewertungskriterien: Smell3, Guideline1, Guideline2
- Efficiency
 - Entwurfsziele: Einfachheit, Modularisierung
- Testability
 - Entwurfsziele: Einfachheit, Modularisierung

Bad Smells: The Bloaters

Something that has grown so large that it cannot be effectively handled

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

Qualitätsmerkmale:

Wartbarkeit

Lesbarkeit

Einfachheit

Bad Smells: The Object-Orientation Abusers

Solution does not fully exploit the possibilities of object-oriented design

- Switch Statements
- Temporary Field
 - Member variables used only occasionally
- Refused Bequest
 - Subclass does not use much behaviour of super class
- Alternative Classes with Different Interfaces

Qualitätsmerkmale:

- Wartbarkeit
- Verständlichkeit
- Einfachheit
- Erweiterbarkeit
- Konformität

Bad Smells: The Change Preventers

Solution hinders changing or further developing the software by tangling and scattering

- Parallel Inheritance Hierarchies
- Divergent Change
 - a single class to be modified by many different changes
- Shotgun Surgery
 - need to modify many classes when making a single change

Qualitätsmerkmale:

- Wartbarkeit
- Erweiterbarkeit
- Einfachheit
- Verständlichkeit

Bad Smells: The Dispensables

Something unnecessary that should be removed from the source code

- Lazy Class
- Data Class (haben nur getter und setter)
- Duplicated Code (Änderungen müssen in allen Klassen gemacht werden)
- Dead Code
- Speculative Generality

Qualitätsmerkmale:

- Wartbarkeit
- Einfachheit
- Verständlichkeit
- Erweiterbarkeit
- Testbarkeit

Bad Smells: The Couplers

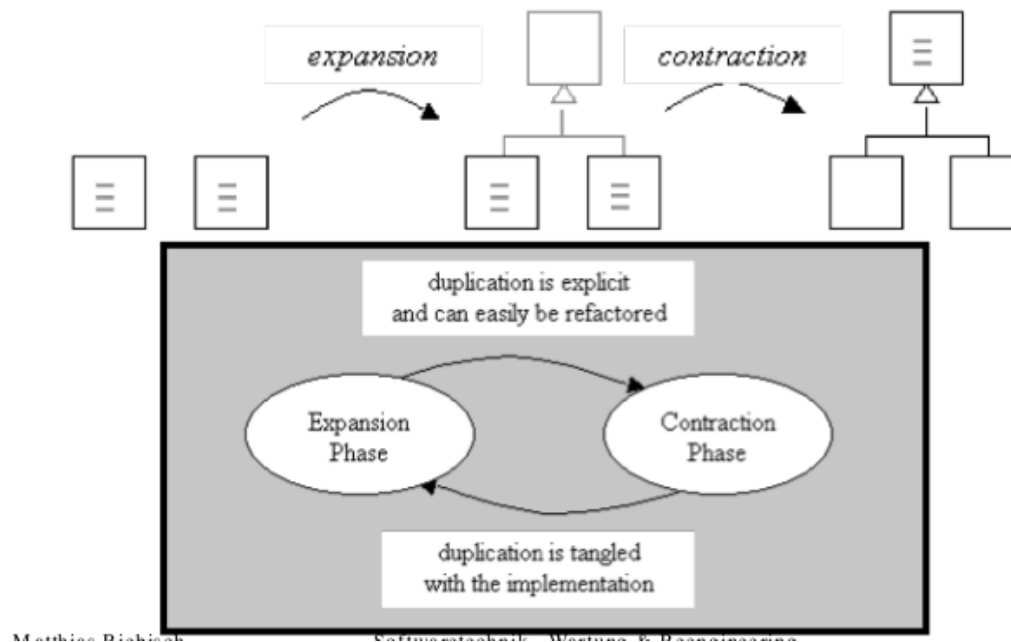
Smells representing inappropriate coupling

- Feature Envy
 - a method is more heavily coupled to other classes than the one that it is
- Inappropriate Intimacy
- Message Chains: Immer über Referenzen weitergeben
 - `A.getB().getC().getD().getTheNeededData()`
- Middle Man
 - trying to avoid high coupling with constant delegation

Qualitätsmerkmale:

- Wartbarkeit
- Verständlichkeit
- Einfachheit
- Erweiterbarkeit
- Testbarkeit

Ablauf Refactoring



Expansion: Oberklasse erstellen

Contraction: Methoden in Oberklasse kopieren

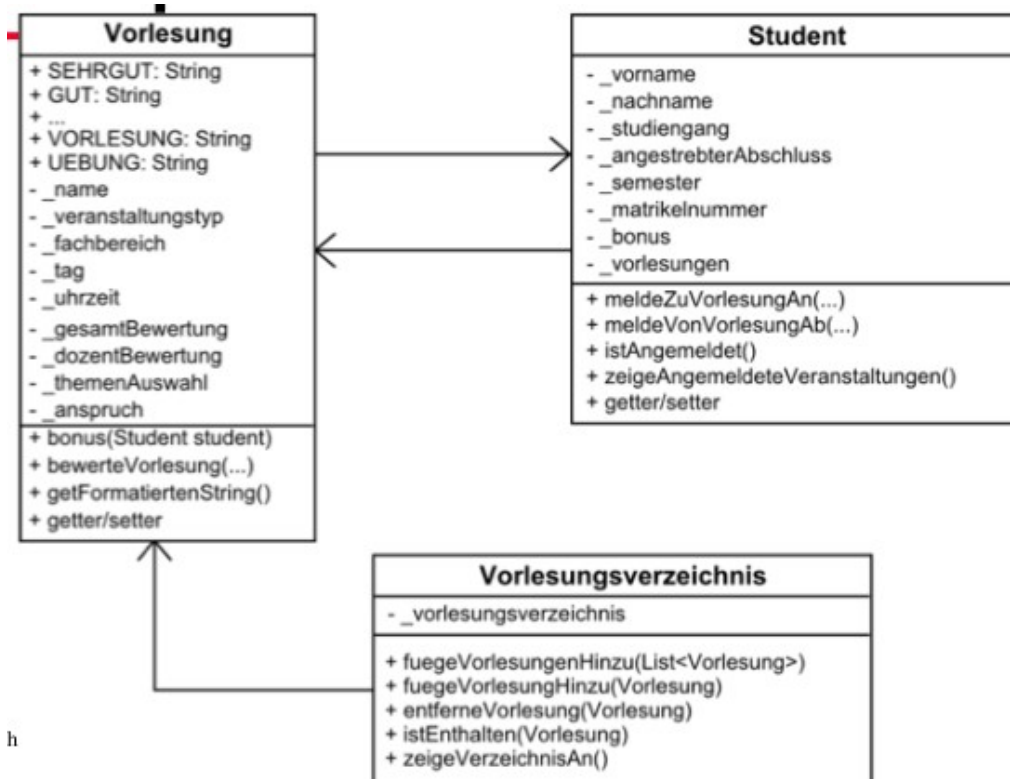
Refactoring-Katalog : <http://www.refactoring.be/thumbnails.html>

- Pull Up Feature
- Push Down Feature
- Encapsulate Feature
- Inline Feature
- Collapse Hierarchy
- Extract Hierarchy
- Introduce Indirection
- Inline Indirection
- Introduce Duplication
- Extract Variation
- Duplicate Feature
- Consolidate Abstraction
- Consolidate Interface
- Eliminate Duplication By Inheritance

- Eliminate Duplication By Composition
- Replace Singletons With Singleton
- Enable Substitution With Interfaces
- Generalize Behavior With Inheritance
- Specialize Behavior With Inheritance
- Split Implicit Layer
- Encapsulate Multiplicity
- Hide Implementation With Interface
- Trade Variation For Duplication
- Trade Duplication For Variation
- Replace Implementation Inheritance With Composition
- Replace Specializing Composition With Inheritance
- Separate Interface From Implementation
- Separate Module Dependencies With Adapter
- Hide Subsystem Complexity With Facade
- Enable Configurable Behavior With Plugin
- Enable Component Subcomponent Substitution
- Replace Template With Strategy
- Replace Concrete Interfacing Class With Explicit Interface
- Replace Concrete Template Class With Abstract Template

Bewertung von Vorlesungen: Code-Beispiel

Code Beispiel: Ist-Zustand



3 fachliche Klassen

Unsere Änderungswünsche

1. Mehr als eine Bewertung für eine Vorlesung speichern → bisher gab es keine Liste von Bewertungen
 - Umstrukturierung der Klasse Vorlesung
 - Bis jetzt ist diese Klasse für alles zuständig
2. Berechnung von Bonuspunkten für Studenten
 - Abhängig vom Veranstaltungstyp (Vorlesung, Übung, Seminar etc.) Bonuspunkte berechnen
 - Bis jetzt: Vorlesung und Übung; es gibt noch Seminar, Praktikum etc.
 - Vorher die Methode `bonus(Student student)` umstrukturieren

Large Class (Bad Smell): für Punkt 1.

```
public class Vorlesung
{
    public final static String SEHRGUT = „sehr gut“;
    public final static String GUT = „gut“;
    public final static String MITTEL = „mittel“;
    public final static String SCHLECHT = „schlecht“;
    public final static String WEISS_NICHT = „weiß nicht“;

    //...

    private String _gesamtBewertung;
    private String _dozentBewertung;
    private String _themenAuswahl;
    private String _anspruch;

    //...
}
```

Extract Class

```
public class Bewertung
{
    public Bewertung()
    {
    }
}
```

Extract Class ist der Refactoring Schritt. Klasse aber erstmal leer. Test durchführen und wird erfolgreich sein.

```
public class Vorlesung
{
    public final static String SEHRGUT = „sehr gut“;
    public final static String GUT = „gut“;
    public final static String MITTEL = „mittel“;
    public final static String SCHLECHT = „schlecht“;
    public final static String WEISS_NICHT = „weiß nicht“;

    //...

    private String _bewertung = new Bewertung();
    private String _gesamtBewertung;
    private String _dozentBewertung;
    private String _themenAuswahl;
    private String _anspruch;

    //...
}
```

Extract Class

```
public class Bewertung
{
    public Bewertung()
    {
    }
}
```

```
public class Vorlesung
{
    public final static String SEHRGUT = „sehr gut“;
    public final static String GUT = „gut“;
    public final static String MITTEL = „mittel“;
    public final static String SCHLECHT = „schlecht“;
    public final static String WEISS_NICHT = „weiß nicht“;

    //...

    private String _bewertung = new Bewertung();
    private String _dozentBewertung;
    private String _themenAuswahl;
    private String _anspruch;

    //...

    public String getGesamtBewertung()
    { return _bewertung.getGesamtBewertung(); }
}
```

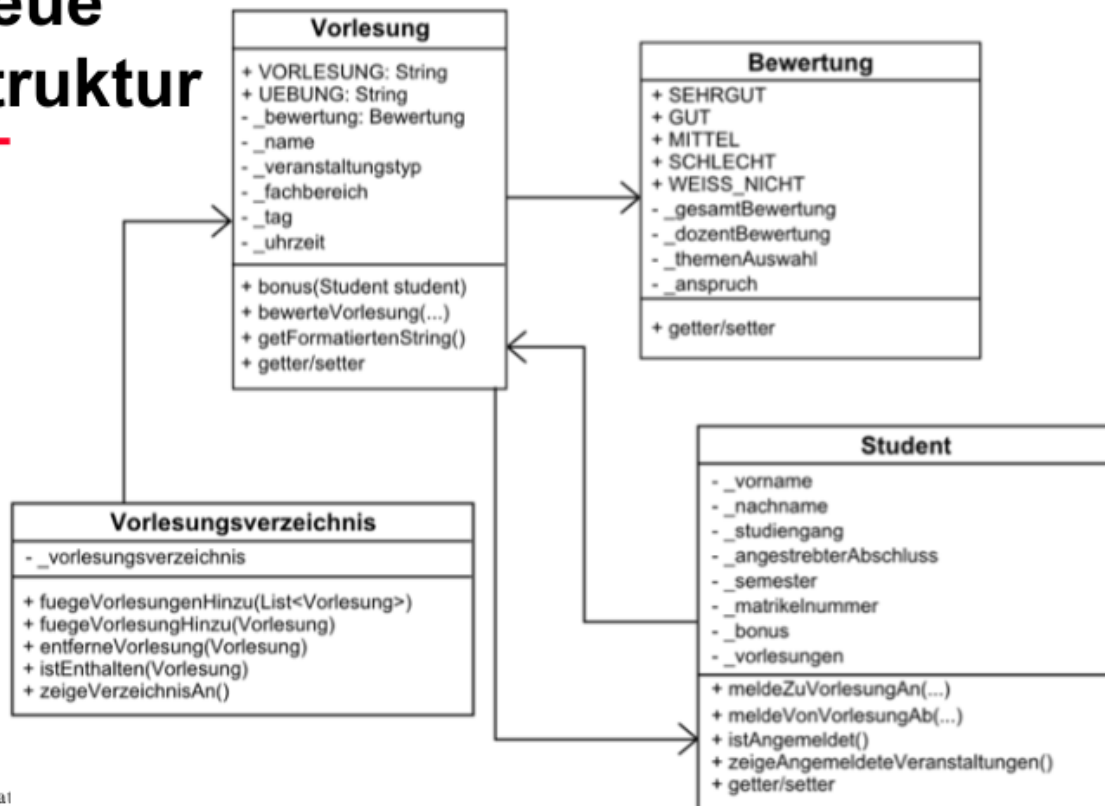
Move Field

```
public class Bewertung
{
    private String _gesamtBewertung;
    public Bewertung()
    {
    }

    public String getGesamtBewertung()
    {
        return _gesamtBewertung;
    }
}
```

Erstmal getter kopieren.

Neue Struktur



Ma1

→ Bewertungsteil ist ausgelagert und dadurch ist es möglich mehrere Bewertungen anzulegen

Änderungen einfügen: Mehrere Bewertungen

```

public class Vorlesung
{
    private List<Bewertung> _bewertungen;

    //...

    public Vorlesung(String name, String typ, String fachbereich, String tag, String uhrzeit)
    {
        //...
        _bewertungen = new ArrayList<Bewertung>();
    }

    //...

    public void bewerteVorlesung(String gesamt, String dozent, String themen, String anspruch)
    {
        Bewertung bewertung = new Bewertung(gesamt, dozent, themen, anspruch);
        _bewertungen.add(bewertung);
    }
}
  
```

Konstruktor erweitern (funktionale Erweiterung) kann nun nach Refactoring erfolgen

Switch Statement (Bad smell): für Punkt 2.

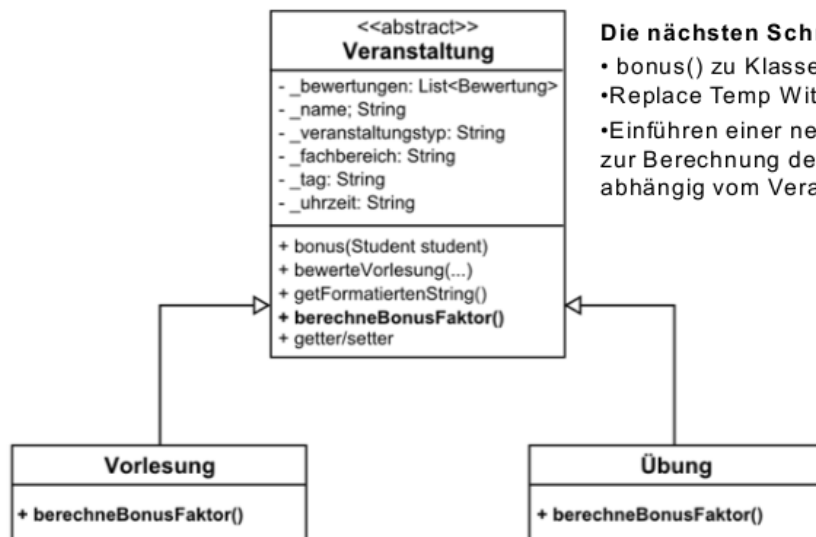
```
public class Vorlesung
{
    public final static String VORLESUNG = "Vorlesung";
    public final static String UEBUNG = "Uebung";
    //...
}
```



- Verhalten abhängig von Type Code
- Vererbungshierarchie einführen
- Switch-Anweisung eliminieren
- *Replace Type Code With Subclasses*
- *Replace Conditional With Polymorphism*

```
switch (veranstaltungstyp)
{
    case Vorlesung.VORLESUNG:
        bonusTotal += 2 * bonusFactor;
        break;
    case Vorlesung.UEBUNG:
        bonusTotal += 2*bonusFactor + 3;
        break;
    default:
        break;
}
```

Replace Type Code With Subclasses



Die nächsten Schritte:

- bonus() zu Klasse Student
- Replace Temp With Query
- Einführen einer neuen Methode zur Berechnung des Bonusfaktors abhängig vom Veranstaltungstyp

Klasse Vorlesung und Übung haben die Funktion drin und es wird kein Switch Statement gebraucht.

Vorlesung wurde zu Veranstaltung umbenannt und ist eine abstract Class.

Methode Bonus (Student student)

```
public int bonus(Student student)
{
    int bonusTotal = 0;
    int bonusFactor = 2;
    List<Vorlesung> vorlesungen = student.getVorlesungen();
    int semester = student.getSemester();
    String matrikelnummer = student.getMatrikelnummer();
    String vorname = student.getVorname();
    String nachname = student.getNachname();
    String studiengang = student.getStudiengang();

    //...
}
```

Feature Envy (Bad Smell): Methode ist eher an Werte der Exemplarvariablen von Student interessiert

Move Method zu Klasse Student (Refactoring)

Zwischenstand

```
public int bonus() //Removed Parameter
{
    int bonusTotal = 0;
    int bonusFactor = 2;

    //Lokale Variablen entfernt
    //...

    int basisBonus = _semester * 2 + 1;
    //...

    if(_studiengang.equals("Informatik"))
    {
        bonusTotal += basisBonus;
    }
    else
    {
        bonusTotal += basisBonus - 2;
    }
}
```

- Removed Parameter student
- Lokale Variablen bzgl. Student nicht mehr benötigt
- Exemplarvariablen können jetzt genutzt werden
- Nächstes Ziel: Weitere lokale Variable basisBonus eliminieren

Replace Temp With Query

```
public int bonus() //Removed Parameter
{
    int bonusTotal = 0;
    int bonusFactor = 2;

    //Lokale Variablen entfernt
    //...

    int basisBonus = _semester * 2 + 1;
    //...

    if(_studiengang.equals("Informatik"))
    {
        bonusTotal += basisBonus;
    }
    else
    {
        bonusTotal += basisBonus - 2;
    }
}
```

Extract Method

```
private berechneBasisBonus()
{
    return _semester * 2 + 1;
}
```

Replace Temp With Query

```
if(_studiengang.equals("Informatik"))
{
    bonusTotal += berechneBasisBonus();
}
else
{
    bonusTotal += berechneBasisBonus() - 2;
}
```

Replace Conditional With Polymorphism

Ausschnitt aus bonus():

```
switch (veranstaltungstyp)
{
    case Veranstaltung.VORLESUNG:
        bonusTotal += 2 * bonusFactor;
        break;
    case Veranstaltung.UEBUNG:
        bonusTotal += 2*bonusFactor + 3;
        break;
    default:
        break;
}
```

```
public abstract class Veranstaltung
{
    public abstract berechneBonusFaktor
        (int factor);
}
```

extends

```
public class Vorlesung
{
    @Override
    public berechneBonusFaktor(int factor)
    {
        2 * factor;
    }
}
```

Eine logische Bedingung durch Polymorphismus ersetzen.

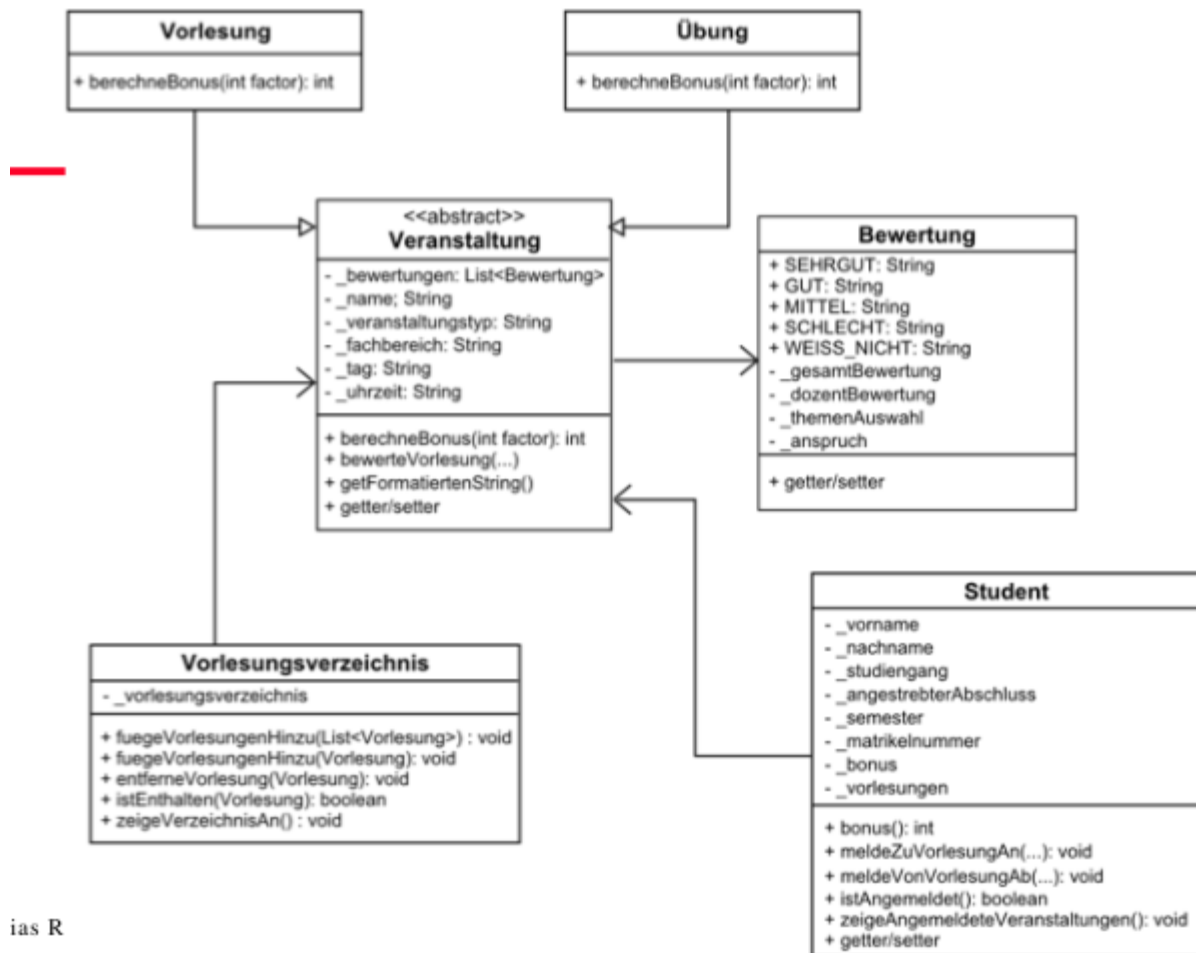
Ausschnitt aus bonus():

```
switch (veranstaltungstyp)
{
    case Veranstaltung.VORLESUNG:
        bonusTotal += 2 * bonusFactor;
        break;
    case Veranstaltung.UEBUNG:
        bonusTotal += 2*bonusFactor + 3;
        break;
    default:
        break;
}
```

```
public abstract class Veranstaltung
{
    public abstract berechneBonusFaktor
        (int factor);
}
```

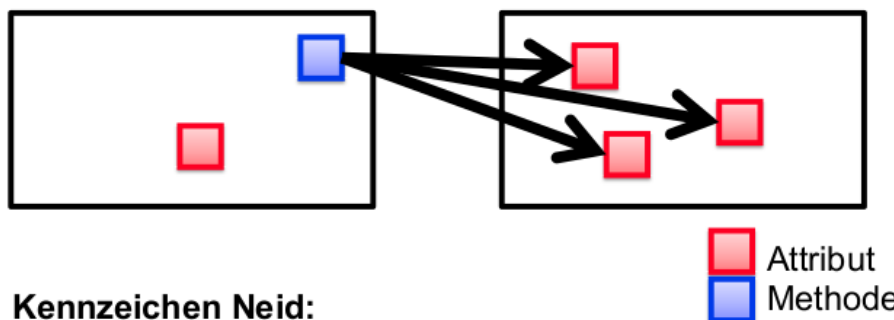
```
public class Vorlesung
{
    @Override
    public berechneBonusFaktor(int factor)
    {
        2 * factor;
    }
}
```

```
veranstaltung.berechneBonusFaktor(factor)
```



ias R

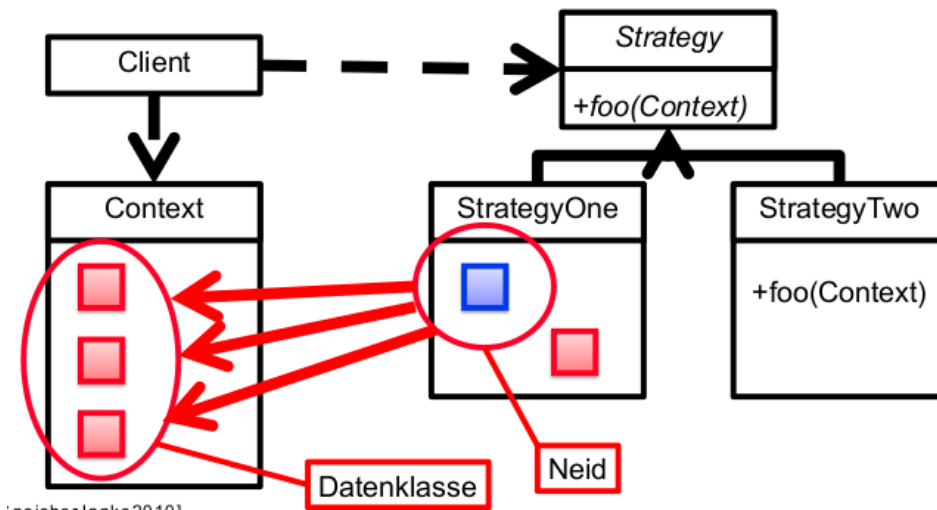
Smells im Kontext: Beispiel Neid (Feature Envy)



Kennzeichen Neid:

- | **Einige** (3) fremde Attribute
- | Verhältnis 0,25 → **unterhalb einem Drittel**
- | Daten gehören zu **Einigen** (1) Klassen

Strategy Pattern impliziert Neid



Embedded DSLs (Domain Specific Language)

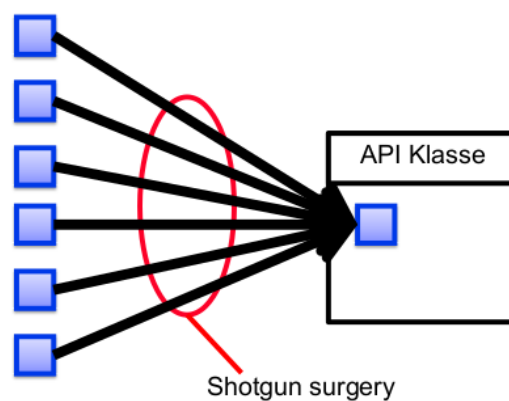
Architekturstile:

- Methodenketten
- Geschachtelte Funktionen
→ Ziel: Lesbarkeit

Aber: charakteristische Eigenschaften von Smells:

- Methoden-Ketten
- Verletzung Law of Demeter
- Evtl. Gestreute Kopplung
- Evtl. Enge Kopplung

APIs weisen Smells auf:



5. Vorgehensmodelle der Softwareentwicklung

5.1 Vorgehensmodelle (= Prozessmodelle)

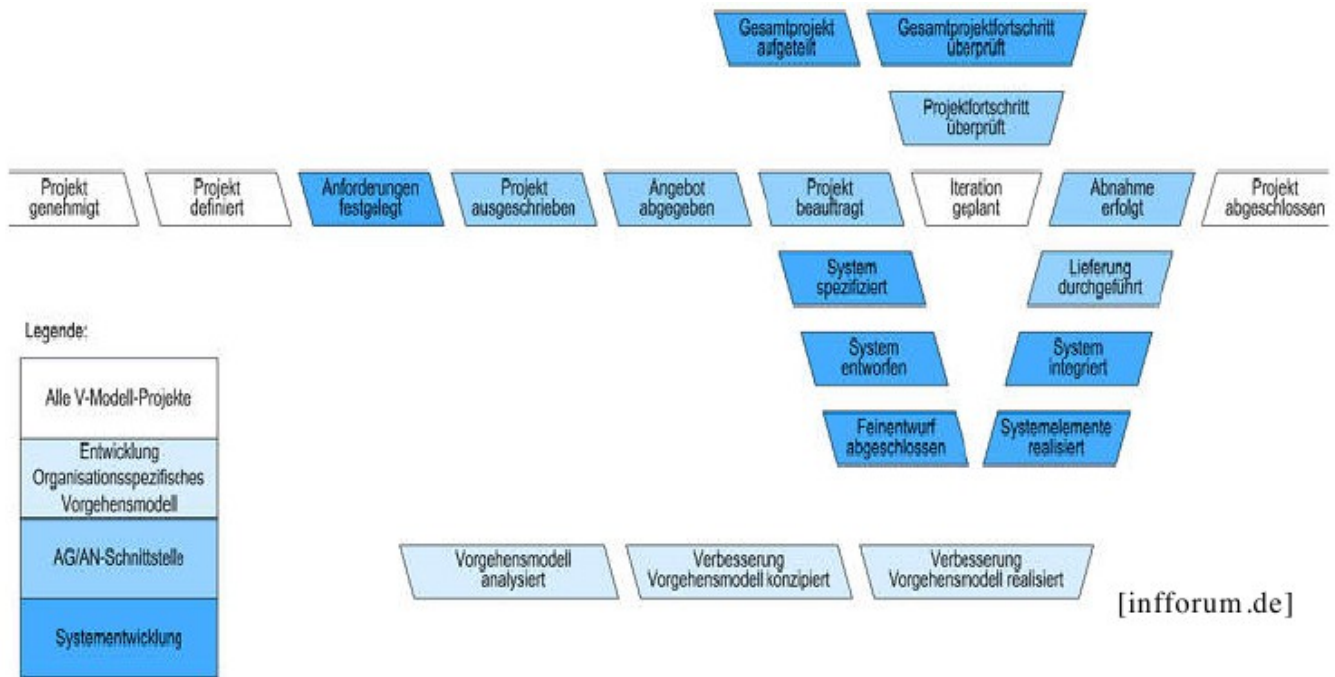
- Ziele: Projekte steuerbar machen → Projektmanagement
 - Softwareentwicklung zeitlich gliedern
 - Teilaufgaben abgrenzen
 - Projektfortschritt messen
 - Arbeitsteilung steuern
 - frühzeitig Qualität erreichen
- Bestandteile:
 - Rollen: Aufgaben, Kompetenzen, Kommunikation
 - z.B. Architekt: was tut er mit wem und wie
 - Aktivitäten: Reihenfolge, Beschreibung, Anleitung
 - Softwaretest: wann wie womit
 - Produkte: Vorlagen, Zusammenhänge

5.2 Schwergewichtige Vorgehensmodelle am Beispiel V-Modell

Vorgehensmodell der Bundesbehörden

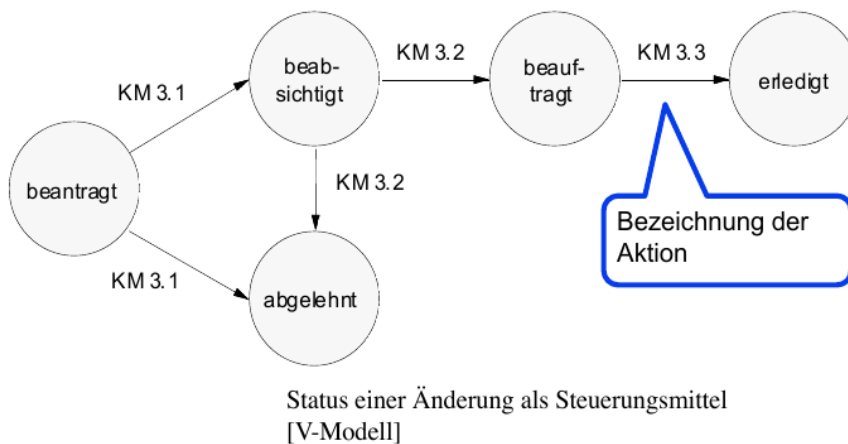
- Allgemeingültig für alle Arten von Entwicklung
- Anpassung an Eigenschaften von Projekten als Teil des Modells: Tailoring
 - Streichbedingungen für kleinere Projekte
- Geeignet für große, langdauernde, verteilte Projekte
- Geeignet wenn Anforderungen und Risiken klar 2005 V-Modell XT – eXtreme Tailoring
- Unterstützung für iteratives Vorgehen: Reihenfolge fehlt

Entwicklungsprozess V-Modell XT



→ Produkt in zentraler Position

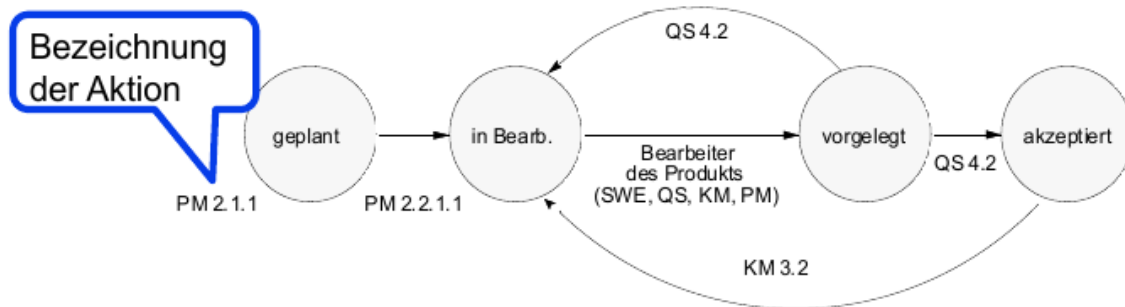
Änderungs(Auftrags-)zustände



Produktzustände im V-Modell

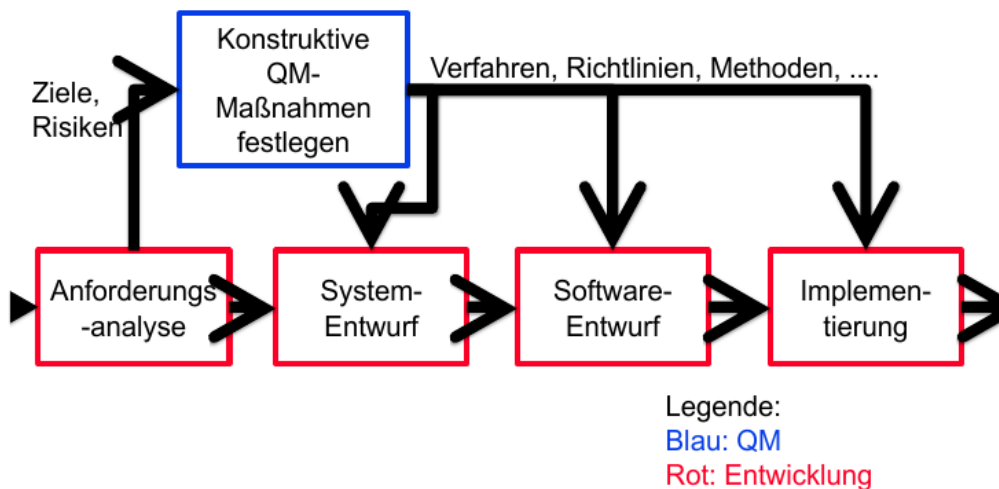
Freigabestatus von Produkten für Planverfolgung, vollständige Prüfungen und konfliktfreie Bearbeitung

Produkt: Begriff des V-Modells, besser: Artefakt



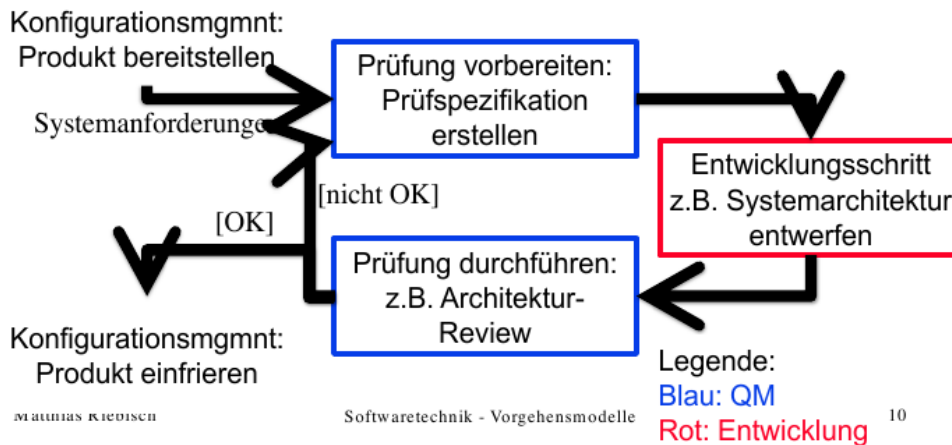
Unterstützung PM: z.B. Zeitpunkt der 100% Fertigstellung eines Moduls anhand erfolgreicher Prüfung erfassbar

Qualitätsmanagement im Entwicklungsprozess: Konstruktive Maßnahmen



Qualitätsmanagement im Entwicklungsprozess: Analytische Maßnahmen

- Prüfung vorbereiten vor jedem Entwicklungsschritt
- Prüfung durchführen während/nach jedem Entwicklungsschritt
- Rückmeldung an Projektmanagement



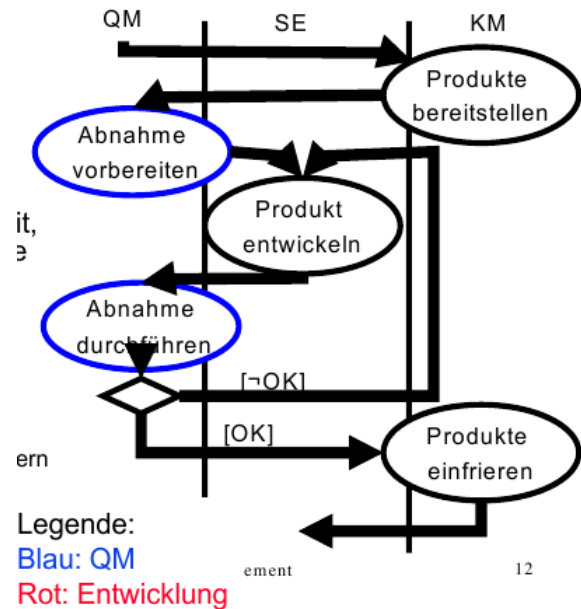
Konstruktive Maßnahmen: Abhängig von Risiken

	Art der Vorgaben für den Entwickler	Verpflichtungen seitens des Entwicklers
Stufe "niedrig"	keine Vorgaben	keine Verpflichtungen
Stufe "mittel"	statistische Vorgaben (z.B. Mindestabdeckung) zur Durchführung der Prüfung	Dokumentation muß den Vorgaben genügen
Stufe "hoch"	genaue Spezifikation der Vorbereitung, Durchführung und Auswertung der Prüfung	Prüfprotokoll gemäß Vorgaben QM

Matthias Riebisch Softwaretechnik - Vorgehensmodelle 11

Analytische Maßnahmen: Verzahnen mit Entwicklung SE und KM

- Möglichkeiten für inkrementelles Vorgehen nutzen
- Abnahmekriterien, Testszenarien / -fälle vor Entwicklung bereitstellen
- Wartbarkeit, Erweiterbarkeit, Flexibilität usw. als explizite Ziele aufnehmen
- Zugehörige Dokumente einbeziehen
- Schrittweise „Reifegrad“ erhöhen
 - Dokumentationsbasis verbessern
 - Architekturqualität verbessern
 - Modell-Qualität verbessern



5.3 Agile Vorgehensmodelle am Beispiel Scrum

Vorteile

- Beherrschung sich entwickelnder, veränderlicher, schwer definierbarer Anforderungen (wenn der Auftraggeber nicht weiß was er will)

Voraussetzungen: Team

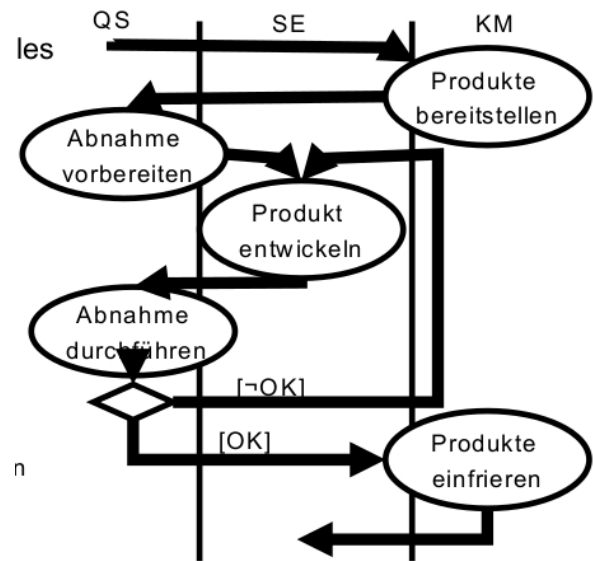
- Größe: kleine Teams: 7 +/- 2 Mitglieder
- Verantwortungskultur: gemeinsame Verantwortung
- Selbstorganisation: Team entscheidet
- Vollständigkeit: Team verfügt über alle Kenntnisse
- Kunden-Feedback ins Team

Herausforderungen:

- Nachhaltige Ergebnisse (Architektur, Doku u.ä.), insbesondere bei langer Dauer und Fluktuation
- Planung und Vorhersage

Einbindung in Entwicklungsprozess

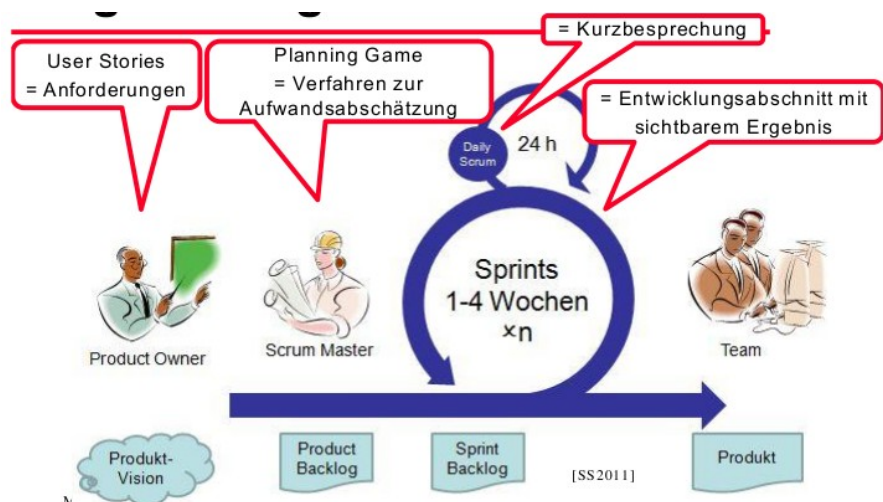
- Möglichkeiten für inkrementelles Vorgehen nutzen
- Abnahmekriterien, Testszenarien / -fälle vor Entwicklung bereitstellen
- Wartbarkeit, Erweiterbarkeit, Flexibilität usw. als explizite Ziele aufnehmen
- Zugehörige Dokumente einbeziehen
- Schrittweise „Reifegrad“ erhöhen
 - Dokumentationsbasis verbessern
 - Architekturqualität verbessern
 - Modell-Qualität verbessern



Agile Prinzipien

- Kleine Schritte statt Big Bang
- Alles verifizieren
- Eigenem Können immer misstrauen
- Nächste Schritte ehrlich besprechen
- Konsequenz vereinfachen
- Überarbeiten und Refactoring
- Pareto: das Wichtigste (Ergebniswirksamste, Riskanteste, Schwierigste) zuerst
- Erfahrungen gemeinsam machen und teilen

Agiles Vorgehen: Scrum



Grundlegende Rollen in Scrum:

Product Owner: stellt fachliche Anforderungen und priorisiert sie + Abnahme und Feedback

Stakeholder: Beobachter und Ratgeber

Team: entwickelt das Produkt

Scrum Master: managed den Prozess und beseitigt Hindernisse

Scrum und Testen?

- Pro:
 - Nutzerzufriedenheit mit großer Bedeutung
 - Unit Tests, frühzeitiges Testen und testgetriebene Entwicklung vorgesehen
- Kontra:
 - Keine festen Phasen, sondern fast kontinuierliche Freigaben
 - keine formalen Prozesse (außer Sprint)
 - Wenige formal definierte Artefakte
 - Langfristige Ziele im Hintergrund
- Maßnahmen für Steigerung des Pro:
 - Akzeptanztests mit Mehr-Augen-Prinzip
 - Externes QM-Team für Systemtests, Regressionstests, Testautomation sowie Defect Tracking
 - Testen innerhalb des gleichen Sprint
 - Gemeinsame „Definition of Done“: Aktivitäten, Ergebnisse und Qualitäten
 - Pair Testing analog zum Pair Programming
 - User Stories für Qualitätsziele

Scrum und konstruktives Qualitätsmanagement?

- Pro:
 - Nutzerzufriedenheit mit großer Bedeutung
 - Hohe Motivation des Teams
 - Gute Qualifikation des Teams
 - Lernen, Fehlervermeidung, Effizienz des Entwicklungsprozesses im Fokus
 - Einfachheit, Wirksamkeit als Ziele

- Kontra:
 - Methoden nicht formal festgehalten, keine formal definierte Artefakte
 - Langfristige und Qualitätsziele eher im Hintergrund
- Maßnahmen für Steigerung des Pro:
 - Bewusste Reflektion von Best Practices
 - Bewusste Auswertung von Fehlern und Mängeln
 - User Stories für Qualitätsziele
 - Checklisten als vorgefertigte Arbeitsmittel als Speicher für Best Practice einsetzen

5.4 Objektorientierte Vorgehensmodelle am Beispiel Unified Process

Rational Unified Process: kommerzielles Produkt von IBM

- Beschreibt Rollen, Aktivitäten, Vorgehensweisen
- Spezifisch für Entwicklung objektorientierter Software
- Architekt: Aufgaben, Aktivitäten, Artefakte

5.5 Reifegrad-Modelle am Beispiel CMMI

- CMMI: Capability Maturity Model Integration
- CMMI-DEV: CMMI for Development
 - Referenzmodell für Verbesserung von Organisationen, die Software, Systeme oder Hardware entwickeln
 - Herausgegeben vom SEI Software Engineering Institute
- Bewertung anhand Prozessgebiete – Process Areas
 - Erlaubt Bewertung des Reifegrads: Sind Elemente der Prozesse vorhanden?
 - Liefert Empfehlungen für Weiterentwicklung: Fehlende Elemente je Prozessgebiet einführen

CMMI Reifegrade – Maturity Levels

5 – Optimizing

Die Arbeit und Arbeitsweise werden mit Hilfe einer statistischen Prozesskontrolle verbessert.

4 – Quantitatively Managed

Es wird eine statistische Prozesskontrolle durchgeführt.

3 – Defined

Die Projekte werden nach einem angepassten Standardprozess durchgeführt und es gibt eine organisationsweite kontinuierliche Prozessverbesserung.

2 – Managed

Die Projekte werden geführt. Ein ähnliches Projekt kann erfolgreich wiederholt werden.

1 – Initial

Keine Anforderungen. Diesen Reifegrad hat jede Organisation automatisch.

Maturity Level Process Areas

