Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät
Department Informatik
Lehrstuhl 11 für Software Engineering

Zusammenfassung der Veranstaltung

Softwareentwicklung in Großprojekten (SoSy3)

gehalten im Wintersemester 2017/18
von Prof. Dr. Francesca Saglietti

Vorbemerkung

Diese(s) Zusammenfassung enthält den Inhalt der Vorlesung „Softwareentwicklung in Großprojekten“ des Wintersemester 2017/18 bei Prof. Dr. Francesca Saglietti. Es wurde anhand von Mitschriften in LaTeX gesetzt und erhebt daher weder Anspruch auf Korrektheit noch Vollständigkeit und ist ***offensichtlich inoffiziell***. Bei Unstimmigkeiten und evtl. vorhandenen Fehlern bitte ich um eine Email an untenstehende Adresse. Dieses Skript stellt damit insbesondere **keine** offizielle Veröffentlichung des Lehrstuhl 11 für Software Engineering am Department Informatik der Friedrich-Alexander-Universität Erlangen-Nürnberg dar.

Florian Frank — florian.ff.frank@fau.de
Version vom 14. Oktober 2018

Literaturvorschläge

[Bal09] Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3. Aufl. 2009, XVIII, 624 S. ISBN: 978-3-8274-1705-3 3-8274-1705-8. URL: http://www.springer.com/spektrum+akademischer+verlag/informatik/informatik+und+it+%C3%BCbergreifend/book/978-3-8274-1705-3.

[Bal11] Helmut Balzert. *Lehrbuch der Softwaretechnik Entwurf, Implementierung, Installation und betrieb*. 3. Aufl. Stuttgart: Spektrum Akad. verlag, 2011, XVIII, 596 S. ISBN: 978-3-8274-1706-0.

[Bal96] Helmut Balzert. *Lehrbuch der Software-Technik: Software-Entwicklung*. Lehrbücher der Informatik. 1996, XVI, 1009 S. Ill., graph. Darst. ISBN: 3-8274-0042-2.

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Lehrbücher der Informatik. 1998, XX, 769 S. graph. Darst. ISBN: 3-8274-0065-1.

[GHJ15] Erich Gamma, Richard Helm und Ralph Johnson. *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. 1st ed. mitp Professional. 2015, 1 online resource (482 pages). ISBN: 978-3-8266-9903-0 978-3-8266-9700-5.

[Hit05] Martin Hitz. *UML@work objektorientierte Modellierung mit UML 2*. 3., aktualisierte und überarb. Aufl. dpunkt.lehrbuch. 2005, XIII, 422 S. ISBN: 3-89864-261-5. URL: http://www.gbv.de/du/services/agi/7C9CA380D348685FC12571070048258F/0124284.

[Kah01] Bernd Kahlbrandt. *Software-Engineering mit der Unified Modeling Language*. 2. Auflage. 2001, 1 Online-Ressource (XV, 519S. 215 Abb). ISBN: 978-3-642-56661-5 978-3-540-41600-5. URL: https://doi.org/10.1007/978-3-642-56661-5.

[Lar08] Craig Larman. *Applying UML and patterns an introduction to object-oriented analysis and design and iterative development*. 3. Always learning. 2008, XXV, 703 S. ISBN: 81-775-8979-2 978-81-775-8979-5.

[RI14] Chris Rupp und SOPHIST-Gesellschaft für Innovatives Software-Engineering (Nürnberg). *Requirements-Engineering und -Management aus der Praxis von klassisch bis agil*. 6., aktualisierte und erw. Aufl. 2014, XIV, 556 S. ISBN: 978-3-446-43893-4 978-3-446-44313-6 3-446-43893-9. URL: http://deposit.d-nb.de/cgi-bin/dokserv?id=4762630&prov=M&dok_var=1&dok_ext=htm%20http://www.hanser-fachbuch.de/buch/Requirements+Engineering+und+Management/9783446438934%20http://files.hanser.de/hanser/docs/20140926_2149269455-117_978-3-446-43893-4_Leseprobe.pdf.

[Sch02] Stephen R. Schach. *Object oriented and classical software engineering*. 6th. 2002, XIX, 628 S. graph. Darst. ISBN: 0-07-239559-1.

[SW02] Harry M. Sneed und Mario Winter. *Testen objektorientierter Software das Praxishandbuch für den Test objektorientierter Client/Server-Systeme*. 2002, XIV, 418 S. ISBN: 3-446-21820-3.

[Som16] Ian Sommerville. *Software engineering*. Tenth , global. Always learning. 2016, 1 Online-Ressource. URL: http://lib.myilibrary.com?id=910957.

Inhaltsverzeichnis

KAPITEL 1

EINFÜHRUNG

Ziel dieser Veranstaltung ist es **hochwertige** und **komplexe** Software unter Berücksichtigung von **Zeit** und **Budget**, die den wirklichen **Bedürfnissen** des Kunden entspricht, zu entwickeln. Wir halten uns dazu an die 4 P's:

P wie **Produkt** – *Was ist das Ergebnis?*

P wie **Prozess** – *Wie wird entwickelt?*

P wie **Personal** – *Wer entwickelt?*

P wie **Projektmanagement** – *Wie lange dauert es und wie teuer wird es?*

Der Einsatz von komplexer und vor allem sicherheitskritischer Software hat im letzten Jahrzehnt zugenommen, ein chaotisches Verhalten[1] führt hier zwangsläufig zum Scheitern, was wirtschaftliche und Imageverluste mit sich führt. Wir lernen deshalb bei der Planung von Projekten strukturiert vorzugehen.

1.1 Beherschung von Softwarefehlern

Wir unterscheiden generell zwischen vier Fehlerarten:

Irrtum – *mistake*

Wir bezeichnen den mentalen Vorgang, der zur Entstehung von Programmfehlern fuhrt, als Irrtum.
Ein Irrtum führt stets zu einem oder mehreren Fehlern.

(Produkt)fehler – *fault*

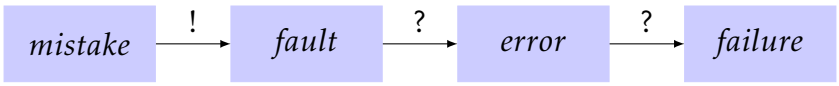
Eine Abweichung zwischen beabsichtigtem und realisiertem Produkt wird als Produktfehler (*fault*) bezeichnet.
Ein Fehler kann zu einem fehlerhaften Zustand und/oder Versagen führen.

Fehlerhafter Zustand – *error*

Eine Abweichung zwischen beabsichtigtem und realisiertem internen Zustand wird als fehlerhafter Zustand (*error*) bezeichnet.

Versagen – *failure*

Eine Abweichung zwischen beabsichtigtem und tatsächlichem Verhalten mit der Offenbarung eines Fehlers wird als Versagen (*failure*) bezeichnet.



[1] unstrukturiert

Wie kann man Fehler „beherrschen“?

- ...auf **konstruktiven** Weg – zur **Irrtumsvermeidung** – durch ein *kontrolliertes*, *durchgängiges*, *rückverfolgbares*, *transparentes Vorgehen*
- ...auf **analytischem** Weg – zur **Fehlererkennung** – durch *rigorose*, *dokumentierte und reproduzierbare Qualitätssicherung vor Verlassen jeder Prozessphase*
- ...mittels **redundanten Maßnahmen zur Fehlertolerierung**

1.2 Vorgehensmodelle

Definition 1 (*Softwareprozess*)

Ein **Softwareprozess** ist eine Abfolge von Aktivitäten und daraus resultierenden Ergebnissen, die zur Herstellung eines Softwareprodukts führen.

Definition 2 (*Prozessmodell*)

Ein **Prozessmodell** ist eine vereinfachte und abstrahierte Beschreibung eines Softwareprozesses und soll insbesondere folgendes festlegen:

- Reihenfolge des Arbeitsablaufs
- jeweils durchzuführende Aktivitäten
- Definition der Teilprodukte einschließlich Layout und Inhalt
- Fertigstellungskriterien
- Notwendige Qualifikationen der Mitarbeiter
- Verantwortlichkeiten und Kompetenzen
- Anzuwendende Standards, Richtlinien, Methoden und Werkzeuge

Ein **Prozessmodell** legt systematische Vorgehensweise für die *Softwareerstellung und Qualitätssicherung*, das *Konfigurations-* sowie *Projektmanagement* fest.

Der **Software-Lebenszyklus** kennzeichnet alle Phasen und Stadien von Softwareprodukten, wie die ...

... **Anforderungsphase** — *Untersuchung des Anwendungszwecks*

... **Spezifikationsphase** — *Beschreibung der zu erbringenden Dienste*

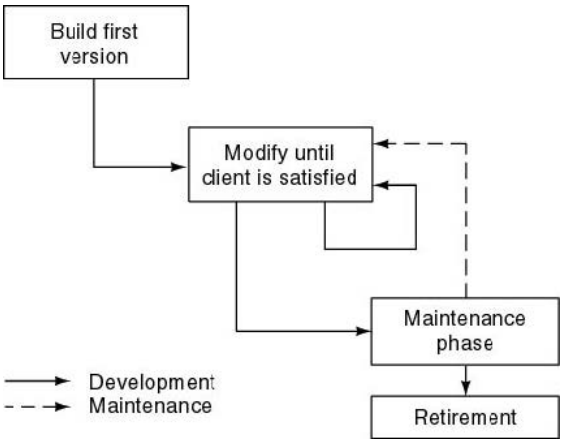
... **Entwurfsphase** — *Umsetzung in Logik*

... **Implementationsphase** — *Kodierung der Komponenten*

... **Integrationsphase** — *Zusammensetzung der Komponenten*

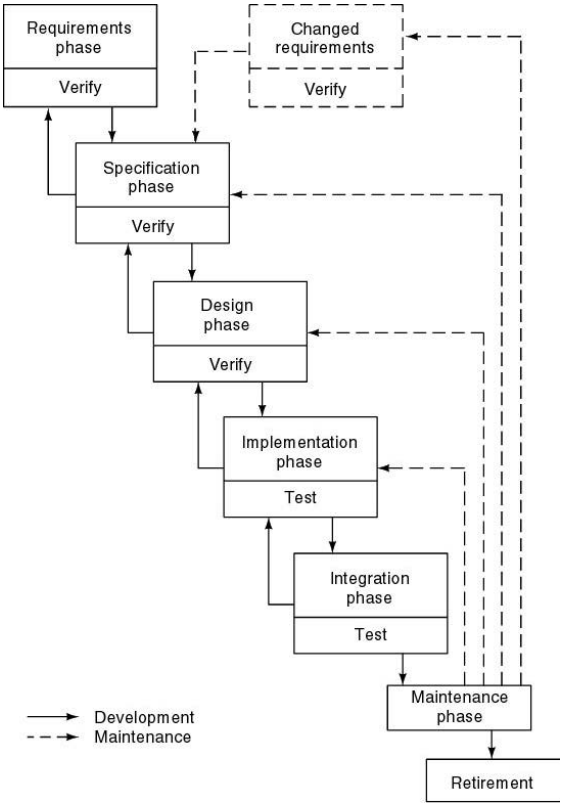
... **Installations-, Nutzungs-, Wartungs- und Ablösungsphasen**

1.2.1 Build-And-Fix Modell



Im Allgemeinen beschreibt das Build-And-Fix Modell die „chaotische Vorgehensweise“, als dass weder Analyse, Design, Test noch das Lebenszyklusmodell hier abgebildet werden. Damit ist es für **größere Projekte absolut ungeeignet!**

1.2.2 Wasserfallmodell



Beim Wasserfallmodell haben wir ein **dokumentationsgetriebenes**, über **strikte Phasen** verfügendes Modell mit **Feedback Loops**. Massive Vorteile sind die **Meilensteine**, die **Dokumentation**, **saubere Änderungen** sowie das **integrierte Testen**. Ein Nachteil des Modells ist das Fortpflanzen sich einschleichender Fehler in der „Kette“.

1.2.3 V-Modell

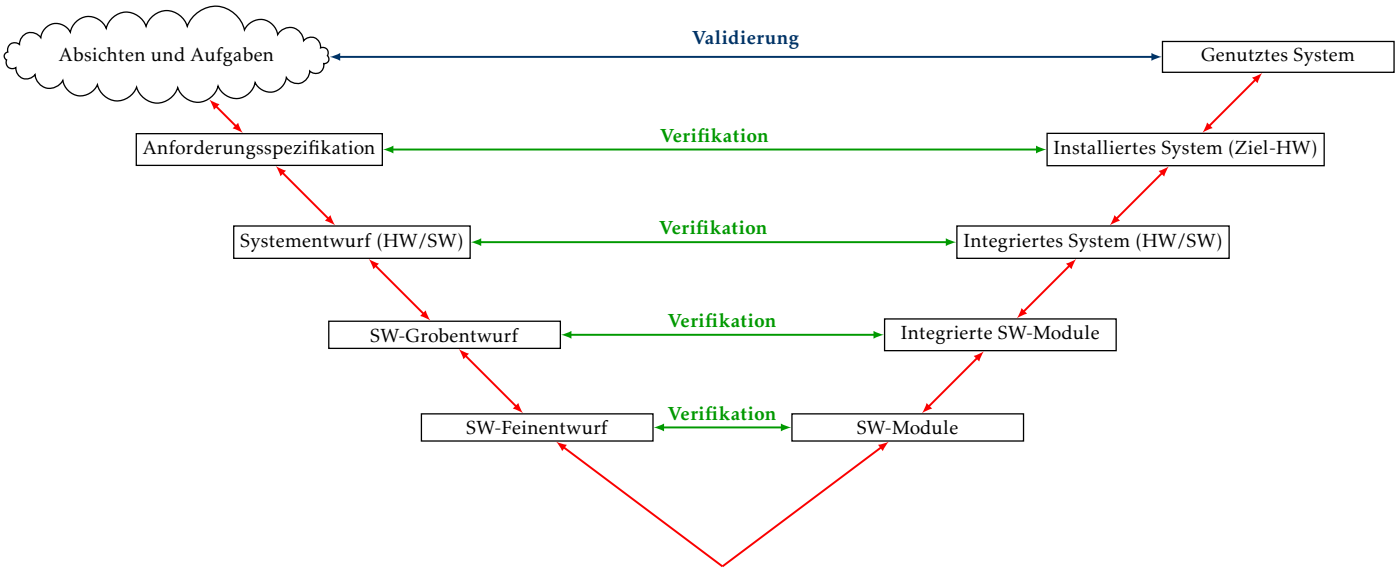
Das V-Modell stellt eine Erweiterung des Wasserfallmodells im Hinblick auf die Qualitätssicherungsaspekte der **Verifikation** und **Validierung** dar.

Definition 3 (*Verifikation*)

Die Überprüfung der Übereinstimmung zwischen Softwareprodukt und dessen Spezifikation wird als **Verifikation** bezeichnet.
*Are wie **doing** things **right**?*

Definition 4 (*Validierung*)

Die Überprüfung der Eignung eines Softwareprodukts bezogen auf seinen Einsatzzweck wird als **Validierung** bezeichnet.
*Are wie doing **right things**?*



Generell gilt der Nutzen des getriebenen Aufwands stellt sich erst in der **Wartungsphase** ein.

Bei dem Verifikationsschritt von „**SW-Module**“ zu „**SW-Feinentwurf**“ sprechen wir im Allgemeinen von sogenannten „*Modultests*“, zwischen „**SW-Grobentwurf**“ und „**Integrierte SW-Module**“ von „*Integrationstests*“, zwischen „**Systementwurf**“ und „**Integriertes System**“ von einem „*Systemtest*“ und schlussendlich von der „**Anforderungsspezifikation**“ zu dem „**installiertem System**“ von einem „*Abnahmetest*“.

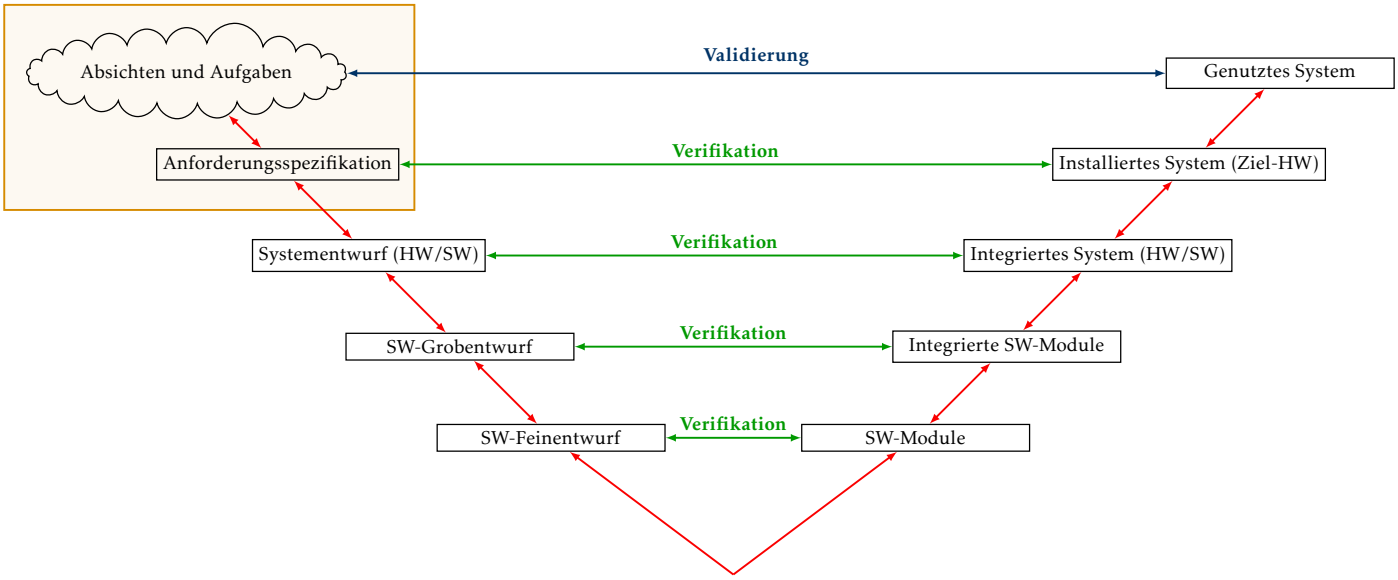
1.2.4 Agile Vorgehensmodelle

Eine agile Vorgehensweise sei als Alternative zum strukturierten Vorgehen genannt, sie ist charakterisiert durch informelle Angabe der Anforderungen, etwa nur durch eine Testfallmenge, sowie eine in kurz aufeinander folgenden Iterationen durch jeweilige Inkrementierung des bereits erzielten Zwischenproduktes erfolgende Entwicklung.
Einsatzzwecke sehen wir vor allem bei kleinen Teams und Projekten, bei einem Drang zu schnellen, eventuell aber noch unvollständigen, Zwischenprodukten sowie bei einer gewissen Flexibilität auf Anforderungsänderungen.
Die Einsatzzwecke spiegeln sich auch in den Zielen wieder, zu welchen die Verringerung des „*time-to-market*“, die engere Einbeziehung des Anwenders sowie die Einleitung von Gegenmaßnahmen bei frühzeitiger Risikoerkennung zählen. Als Beispiel für eine agile Vorgehensweise sei das **extreme programming** oder das „*test-driven development*“ genannt.
Hieraus lassen sich Vorteile und Nachteile erkennen. Zu den Vorteilen agiler Softwaremodelle zählen die Flexibilität der ständigen Anpassung an sich ändernde Voraussetzungen und Anforderungen, die enge Einbeziehung des Auftraggebers, wodurch es zu einer Akzeptanzsteigerung führen kann, das Einfließen der Erfahrung des Echteinsatzes durch Vorabveröffentlichungen sowie die Risikoreduktion vor Produktion durch eine frühzeitige Überprüfung von Teilveröffentlichungen in einem Pilotbetrieb.
Zu den Nachteilen agiler Softwaremodelle zählen die Nichteignung für komplexe Systeme mit hohen Zuverlässigkeitsanforderungen aufgrund fehlender vollständiger Spezifikation, die Gefahr eines unzureichenden Designs, ein relativ hoher Kommunikationsaufwand aus dem eine Nichteignung für große Teams folgt, die höhere Anforderung an Kreativität statt Disziplin, ein hoher Aufwand für das Änderungsmanagement, sowie die ständige Gefahr des „*feature creep*“. Es eignet sich zudem nur bei vorhandener Nähe zum Kunden.

KAPITEL 2

ANFORDERUNGSSPEZIFIKATION

Wo befinden wir uns gerade?



2.1 Einführung

Eine **Anforderung** ist eine *Bedingung oder Fähigkeit*, die von *einer Person* zur *Lösung eines Problems* oder zur *Erreichung eines Ziels* benötigt wird.
Eine **Software-Anforderung** ist eine *Bedingung oder Fähigkeit*, die *eine Software erfüllen oder besitzen* muss, um *einen Vertrag*, *eine Norm* oder ein *weiteres formelles Dokument* zu erfüllen.
Man kann **Anforderungen** unterschiedlich **priorisieren**:

MUSS **unverzichtbare** Anforderungen

SOLL **wichtige** Anforderungen, auf die aber aufgrund von **zu hohen Kosten** verzichtet werden kann

WUNSCH **nice-to-have**, nicht essentiell

Definition 5 (*Softwarespezifikation*)

> Eine **Softwarespezifikation** ist eine Zusammenstellung **aller** Anforderungen an eine Software **und** der **Randbedingungen** für ihren Einsatz, deren Aufgabe es ist, die **angestrebte Dienstleistung** zu beschreiben, aber dabei **nicht** mögliche Lösungsansätze zur Realisierung der Diensterbringung zu beschreiben.

Wir unterteilen die Anforderungen an Softwaresysteme in **funktionale** und **nichtfunktionale Anforderungen**.

funktionale Anforderungen beschreiben die Dienste, welche die Software erbringen soll, sind also **Anforderungen** zum erwünschten oder auch unzulässigem Verhalten der Software in bestimmten Situationen

nichtfunktionale Anforderungen beschreiben weitere Einschränkungen der durch die Software zu erbringenden Dienste, also beispielsweise **einzuhaltende Normen** oder bestimmte Umgebungen

Im **Pflichtenheft** werden diese Vereinbarungen zwischen dem Auftraggeber und Entwickler festgehalten. Es dient zum Vermeiden von Missverständnissen, zur detaillierten Beschreibung des Funktionsumfangs sowie zum eindeutigen Formulieren der Softwareeigenschaften und besitzt dabei Vertragscharakter.
Der finale **Abnahmetest** sollte im Pflichtenheft ebenfalls klar definiert werden.

Eigenschaften von Anforderungen Wir unterscheiden auch hier in:

Volständigkeit: vollständige Funktionalitätsbeschreibung mit detaillierter Systemreaktionsbeschreibung für **alle möglichen Eingaben(-klassen)** und **potentiell eintreffenden Ereignissen** inklusive deren **möglichen Kombinationen**

Konsistenz: Widerspruchsfreiheit von Anforderung in sich als auch untereinander

Korrektheit: Jede Anforderung muss die **Absicht des Auftraggebers** vollständig und konsistent wiedergeben. Überprüfung durch Validierung.

Eindeutigkeit: Eindeutige **Interpretation** einer jeden Anforderung

Realisierbarkeit: Umsetzbarkeit einer jeden Anforderung unter **Randbedingungen** mit Rücksicht auf **Systemgrenzen**, **Umsetzungskosten**, in ablauffähiges System.

Verfolgbarkeit: Eindeutige Identifizierbarkeit einer jeden Anforderung sowie die **durchgängige Auffindbarkeit** deren schrittweisen Umsetzung durch den SW-Lebenszyklus

Nachweisbarkeit: Existenz von eindeutigen Kriterien zur Überprüfung der Anforderungserfüllung einer jeden Anforderung.

2.2 Allgemeine Anforderungsermittlung

Die Aufgabe der Anforderungsermittlung ist es die Wünsche und Bedürfnisse der Beteiligten zu erkennen, zu analysisen und darzustellen, sowie den Beteiligten eventuell noch nicht erkannte Möglichkeiten aufzuzeigen, bei Bedarf einen IST-Zustand zu erheben, sowie das Marktpotential zu klären und Randbedingungen zu erkennen, zu analysieren und zu dokumentieren. Die allgemeine Vorgehensweise lässt zuerst **das Problem definieren** mit einer *einheitlichen groben Beschreibung des zu behandelnden Problems*, um anschließend die **Interessensgruppen** zu **identifizieren** und dabei *verschiedene Perspektiven mit einzubeziehen* und zum Schluss dann die Anforderungen und Systemgrenzen zu ermitteln und festzulegen.

2.2.1 Problemdefinition

Das Ziel ist es **eine Einigung** über das zu lösende Problem zu erzielen, dazu soll die Problemdefinition aus der Sicht des Anwenders **beschrieben** werden. Dabei ergeben sich zwei hauptsächliche Probleme mit der zu verwendenden Sprache/Terminologie und des *moving targets* (*what you wish for is not necessarily what you really need*).

2.2.2 Identifizieren der Interessensgruppen (en. *stakeholder*)

Definition 6 (*Interessensgruppen*)

Interessensgruppen (en. *stakeholder*) sind alle Personen, die von der Systementwicklung sowie vom Einsatz und Betrieb des Systems betroffen sind. Damit insbesondere Softwarenutzer, Wartungspersonal, sowie Ausbilder für diese Software.
Interessensgruppen sind damit **Informationslieferanten** für eventuelle Ziele, Anforderungen sowie Randbedingungen.

Beispiele für typische Interessensgruppen:

- Endbenutzer
- Auftraggeber
- Betreiber
- Entwickler
- Projektleitung
- Wartungspersonal

2.2.3 Anforderungsermittlung (*requirements elicitation*)

Hierbei ist das Verstehen der Probleme sowie das Festlegen der Systemgrenzen wichtig. Man definiert dazu die Schnittstellen zwischen dem Softwaresystem und deren Umgebung. Wir unterscheiden folgende Techniken:

Brainstorming Das Ziel hierbei ist es möglichst viele Ideen und Beiträge in einer möglichst kurzen Zeit zu erhalten. Dabei nimmt man meist eine relativ kleine Teilnehmergruppe mit einem Moderator, welcher möglichst alle Beiträge zulassen solle und bespricht und wertet anschließend Einfälle aus. Die Effektivität lässt sich durch das Teilnehmen unterschiedlicher Interessensgruppengruppenrepräsentanten steigern. Ein Vorteil dieser Methodik ist die Ideenrate pro Zeit sowie das gegenseitige Anregen zu immer neuen Ideen. Nachteile stellen das starke Hervorkommen von sprach- und redegewandte Teilnehmer sowie das Risiko der Nichtpraktikabilität vieler Ideen und das meist unstrukturierte Ergebnis mit dem ein beträchtlicher Nachbearbeitungsaufwand einhergeht.

Fragebogen Hier wird eine Liste von Fragen zum Themengebiet im Antwort-Wahl-Verfahren oder als offene Fragen an verschiedene Personen – nach vorherigem Test der Verständlichkeit in kleinem Kreis – ausgegeben. Die Vorteile sind die einfache und effiziente Auswertungsmöglichkeit sowie das Einbeziehen einer sehr großen Anzahl von Interessensgruppen mit sehr geringem Zeit- und Kostenaufwand. Nachteile ergeben sich in der Schwierigkeit einer nicht suggestiven, aber immer noch eindeutigen und aussagekräftigen Fragestellung, dem notwendigen Hintergrundwissen als Fragesteller sowie -beantworter und der Nichteignung für implizite Anforderungen, komplexe Systemabläufe sowie Anforderungen an Dienstqualitäten.

Interview Hierbei stellt ein Analytiker den Beteiligten vorgegebene Fragen, wobei im Gespräch auftretende Probleme gleich geklärt werden können. Der Zweck ist meist die Erfassung einer groben Systembeschreibung, wobei aber auch neue Anforderungen während des Interviews entdeckt werden können. Der Vorteil liegt in einer starken Individualität des Gesprächsablaufes, der Nachteil hingegen im enormen Zeitaufwand, der mit einem solchen Interview einhergeht.

Simulationsmodelle Mittels Prototypen sollen Benutzern Anforderungen klar werden, die sie nur bei der ersten Bedienung erkennen. Ein Vorteil dieses Systems sind motiviertere Interessensgruppen, gleichzeitig zeichnet sich der Nachteil bei nicht inkrementell durchgeführen Systemerstellungen mit einem hohen Kosten- und Zeitaufwand aus.

Anforderungsreview Hierbei werden die aus bereits durchgeführten Befragungstechniken gewonnenen Anforderungen Interessensgruppen vorgeschlagen und zur Diskussion und zum verpflichtenden Votum gestellt. Dies soll vor allem die Qualität der Anforderungen sicherstellen. Vorteile zeichnen sich mit einer Verbesserung der Korrektheit und

Verständlichkeit der Anforderungen sowie der Identifizierung von Lücken in oder noch fehlenden Anforderungen durch Interessensgruppen ab, der Nachteil liegt auch hier in massiven Zeitkosten.

Workshop Hierbei soll durch eine gemeinschaftliche Erarbeitung der Anforderungen mit relevanten Interessensgruppen ein von allen Teilnehmern abgezeichnetes Anforderungsdokument entstehen. Der Vorteil liegt hierbei in der Förderung des gegenseitigen Verständnisses sowie der Kompromissbereitschaft durch direkte Kommunikation, sowie in der Möglichkeit des Erhaltens von genaueren und hinterfragbaren Informationen. Der Nachteil ist eine starke Gruppendynamik mit eventuell auftretendem Gruppenzwang und sich herauskristalisierenden Meinungsbildern und Ja-Sagern, sowie einer sinkenden Realisierbarkeitswahrscheinlichkeit mit steigender Anzahl an Interessensgruppen.

2.3 Konkrete Vorgehensweise

Unsere Vorgehensweise soll hierbei am Beispiel des Voleremodells dargestellt werden. Zuerst werden wir dabei die Anwendungsfälle modellieren, um anschließend die Anforderungen festzuhalten und zu Schluss die Spezifikation zusammenzustellen.

2.3.1 Anwendungsfallsmodellierung

Das **System** charakterisiert hierbei die Abgrenzung der zu erstellenden Anwendung gegenüber der Außenwelt.

Definition 7 (*Akteur*)

Wir bezeichnen hierbei eine Person oder ein System, das mit dem System interagiert, als **Akteur**[1].

Definition 8 (*Anwendungsfall*)

Ein **Anwendungsfall** beschreibt eine anwendersichtbare Funktionalität des Systems und stellt eine Abstaktion einer typischen Interaktion zwischen Anwender und System dar. Er dient zur Darstellung des externen Systemverhaltens in einer begrenzten Arbeitssituation aus der Sicht des Anwenders.

Setzen wir Anwendungsfälle und Akteure zueinander in Beziehung, so erhalten wir ein Anwendungsfalldiagramm, welches genauere Informationen über **mögliche Anwendungsfälle**[2], **relevante externe Aktionen**[3] sowie die **Charakterisierung der Systemgrenzen**[4]. Sie sind gemäß UML-Standard darzustellen, was in der Veranstaltung „Konzeptionelle Modellierung“ gelehrt wurde. Der Vollständigkeit sei hierzu auf einen Auszug aus dem Merkblatt zur Verhaltensmodellierung mit der UML im Anhang verwiesen.

2.3.2 Festhalten der Anforderungen

Für jeden Anwendungsfall werden nun an dieser Stelle die mit ihm in Zusammenhang stehenden Anforderungen ermittelt. Diese werden dann auf Karten, zumindest nach dem Volere-Modell, festgehalten. Die Karten enthalten dabei folgende Informationen:

Anforderungsnummer Jede Anforderung wird **eindeutig** durchnummeriert

Anwendungsfallnummer Implizite Festlegung der betroffenen Anwender durch Verweis auf bezogene Anwendungsfälle.

[2] also wie das System auf bestimmte Benutzerinteraktionen reagieren soll
[3] also mit wem das System unter welchen Umständen kommunizieren kann
[4] also wie das System von seiner Systemumgebung abzugrenzen ist

Beschreibung der Anforderung Formulierung der Anforderung **in der Sprache des Anwenders**

Anforderungsart Charakterisierung der Anforderungsklasse

Motivation Begründung für Anforderung, setzt gleichzeitig Wichtigkeit und Verständnis der Anforderung fest

Urheber Name der Person, die diese Anforderung erstellt hat, kann für weitere Rückfragen bezüglich der Anforderung kontaktiert werden

Abnahmekriterien stellen die Akzeptanzkriterien für den Abnahmetest dar und sollten möglichst präzise formuliert werden

Kunden(un)zufriedenheit soll das Gefühl von Freude oder respektive Enttäuschung ausdrücken, dass der Kunde bei Erfüllung oder Nichterfüllung spürt. Zufriedenheit ist dabei nicht mit Unzufriedenheit in Relation zu setzen!

Abhängigkeiten stellen Anforderungen dar, die von dieser Anforderung abhängen, sprich von Änderung betroffene Anforderungen oder Anforderungen, deren Existenzberechtigung alleinig von dieser Anforderung abhängt.

Konflikte stellen Anforderungen dar, die potentiell im Konflikt mit dieser Anforderung stehen.

Unterlagen auf die Bezug genommen wird

Historie Erstellungsdatum und Änderungshistorie

2.3.3 Zusammenstellen der Spezifikation

Aus den Anforderungen und den Anwendungsfällen sowie einem *„volere specification template“* wird die Spezifikation zusammengestellt.

2.4 Anforderungsverwaltung

Anforderungen obliegen einem gewissen Evolutionsprozess, als dass es immer wieder zu Anforderungsänderungen kommen kann. Generell gilt: Anforderungen müssen im Allgemeinen **stabil** gehalten werden, gleichzeitig aber **notwendige Änderungen** *kontrolliert* zugelassen werden. Dazu sollten Anforderungen **einzeln identifizierbar sein**, es einen **geordneten Änderungsprozess** sowie einen **geregelten Dokumentenfluss** geben, die Zuständigkeiten sowie Verantwortlichkeiten **klar geregelt** sein und **alle Entscheidungen und Änderungen** rÜckverfolgbar sein.

2.5 Spezifikationssprachen

Wir unterteilen Spezifikationssprachen gemäß ihrer Formalität, so gibt es **informale** – typischerweise natürliche Sprachen – **semiformale** – welche Elemente mit eindeutiger, wohldefinierter Semantik enthalten, beispielsweise ER-Diagramme, funktionale Tabellen oder Ablaufdiagramme – und **formale Spezifikationssprachen**, welche eine durchgehend eindeutige und wohldefinierte Semantik haben, als Beispiele seien endliche Zustandsautomaten, Petrinetze und die OCL genannt. Generell gibt es bei der Auffassung von Spezifikationen zwei Fehlerquellen, einerseits die **Wahrnehmung** (von der Realität auf persönliches Wissen), andererseits die **Wissensdarstellung** (von dem persönlichen Wissen zum *sprachlichen Ausdruck des Wissens*). Bei der **Wahrnehmungsdarstellung** kommt es des öfteren zu folgenden Problemem:

Tilgung: Ein Prozess, durch den wir unsere Aufmerksamkeit selektiv bestimmten Dimensionen unserer Erfahrung zuwenden und andere ausschließen. **Gefahr** der unzulässigen Informationsreduktion.

Generalisierung: Verallgemeinerung individueller Realitätswahrnehmung. **Gefahr** der unzulässigen Verallgemeinerung.

Verzerrung: Eine unzulässige Umgestaltung der Darstellung erfahrener Realität, was gleichzeitig auch die **Gefahr** ist.

2.5.1 Semiformale Spezifikationssprachen

Diese sollen einen Kompromiss zwischen informalen und formalen Spezifikationssprachen darstellen, sie enthalten dementsprechend sowohl Teile in natürlicher Sprache als auch Elemente mit eindeutige, wohldefinierter Semantik.

2.5.1.1 Entitäts-Beziehungs-Diagramme (*entity-relationship-diagrams*)

Genauer behandelt in der Veranstaltung „Konzeptionelle Modellierung“. Dieses Modell findet weite Verbreitung bei der Spezifikation von **Datenbanken**, es kann aber auch für die **objektorientierte Analyse** verwendet werden. Es wird an dieser Stelle explizit nicht näher darauf eingegangen, das Merkblatt aus der oben genannten Veranstaltung sei im Anhang zu finden.

2.5.1.2 Tabellen

Zur Strukturierung von Spezifikationen verwendet man häufig Tabellen, sie sind genauso aufgebaut wie man sich sie vorstellt.

2.5.1.3 Ablaufdiagramm

Ein **Ablaufdiagramm** ist eine graphische Darstellung des zeitlichen Ablaufs eines Prozesses in **abstrahierter** Form. Je nach Prozessart und Abstraktionsebene gibt es verschiedene Arten von Ablaufdiagrammem. Es eignet sich besonders zur Veranschaulichung der **Mensch-Maschinen-Interaktion**. Ein Beispiel sind Prozessablaufpläne, wie sie bereits in anderen Veranstaltungen wie „Algorithmen und Datenstrukturen“ verwendet wurden.

2.5.2 Formale Spezifikationssprachen

Formale Spezifikationssprachen bauen auf der Grundlage mathematisch-logischer Formalismen mittels formal definierter Synatx **und Semantik** auf. Zu ihren **Vorteilen** zählt die ständige **Eindeutigkeit**, die **formale Prüfbarkeit** auf **Widerspruchsfreiheit**, die **Beweisbarkeit** der **Erfüllung wichtiger Eigenschaften** sowie die **Möglichkeit der formalen Verifikation** und der teilweisen Automatisierbarkeit der Analysen.
Zu ihren **Nachteilen** zählen wohl oder übel die hohe Lernkurve, ein aufwändiger Erstellprozess sowie die Schwierigkeit der Verifikation.

2.5.2.1 Endliche Zustandsautomaten

Endliche Zustandsautomaten oder auch *finite state machines* oder auch *endliche deterministische Automaten* sollten aus verschiedenen Veranstaltungen wie die „Grundlagen der technischen Informatik“ oder auch „Berechenbarkeit und Formale Sprachen“ durchaus bekannt sein. Wir verzichten an dieser Stelle deshalb **komplett** auf eine nähere Beschreibung, sie sei in den Folien zu dieser Veranstaltung oder in den Skripten zu einer der eben genannten Veranstaltungen zu finden.

Vorteile Zu den Vorteilen zählen die eindeutige Notation, die intuitiv verständliche graphische Darstellung, die vollständige Analysierbarkeit sowie die existierende *tool*-basierte Unterstützung.

Nachteile Zu den Nachteilen zählen die Möglichkeit der „Explosion der Zustandsmenge“, wogegen sogenannte „Hyperknoten“ helfen könnten, sowie die eingeschränkte Ausdruckskraft bei zeitlichen sowie nebenläufigen Anforderungen.

2.5.2.2 Petrinetze

Auch Petrinetze wurden bereits in der Veranstaltung „Parallele und Funktionale Programmierung“ eingeführt, dies soll an dieser Stelle also auch nicht noch einmal passieren. Es sei aber auf ein entsprechendes Merkblatt im Anhang verwiesen.

2.5.2.3 OCL

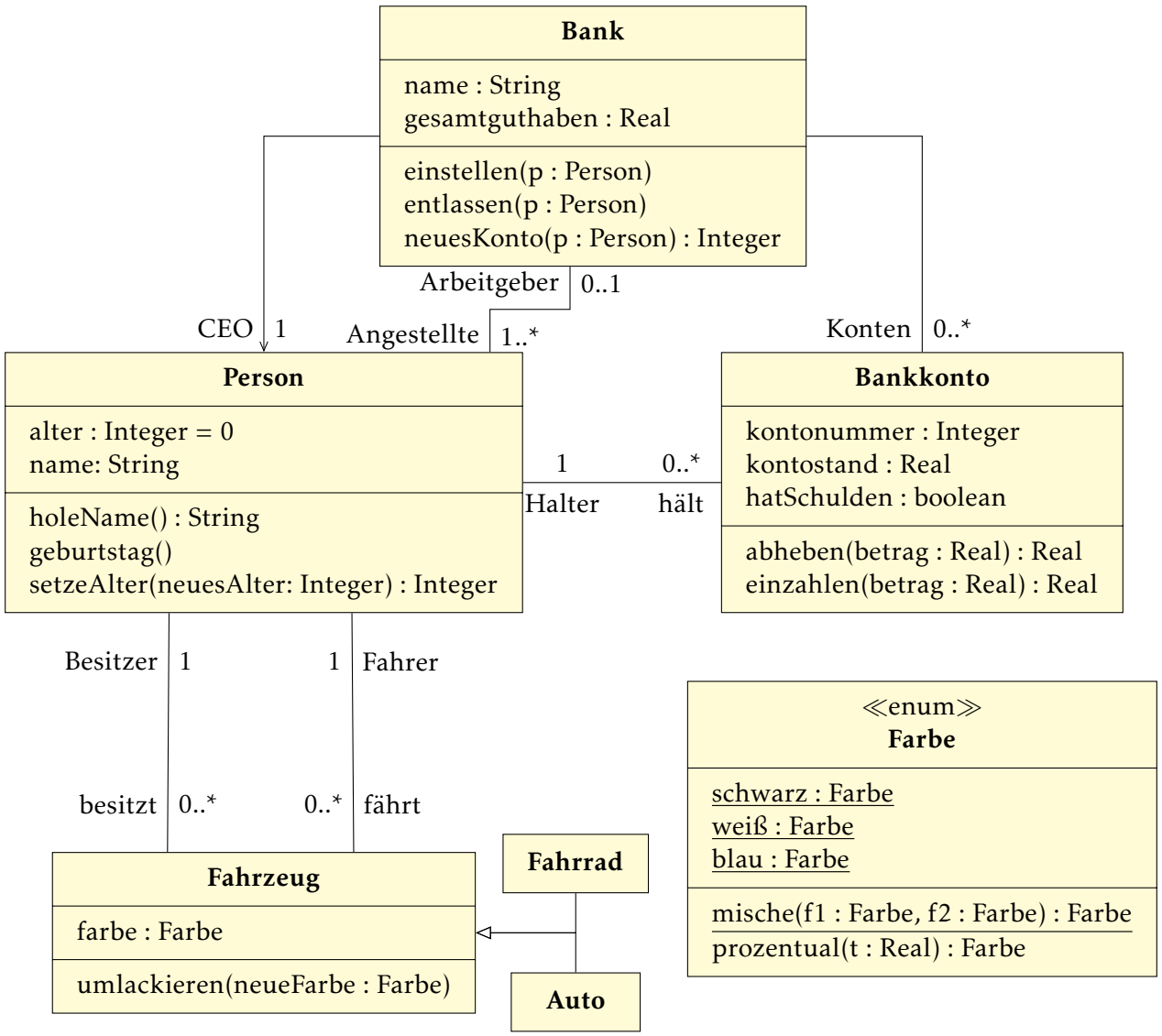
UML wurde um die **Object Constraint Language** (OCL) ergänzt, um zulässiges Verhalten durch formale Einschränkungen zu beschreiben.

Datentypen und Operationen Generell gilt: **Alle** im zugrundeliegenden UML-Modell verwendeten Datentypen und deren Operationen sind auch in den zugehörigen OCL-Annotationen verwendbar, es gibt allerdings ein paar *vordefinierte Datentypen mit Operationen*.

Typ	**Beispielwerte**
`Boolean`	`true`, `false`
`Integer`	`1`, `-5`, `4711`, `...`
`Real`	`2.49`, `49.2`, `23`, `...`
`String`	`'Sein oder nicht sein ...'`, `...`

Typ	**Operationen**
`Boolean`	`and`, `or`, `not`, `xor`, `implies`, `if-then-else-endif`, `...`
`Integer`	`+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=`, `<>`, `abs`, `div`, `mod`, `max`, `min`, `...`
`Real`	`+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=`, `<>`, `abs`, `div`, `mod`, `max`, `min`, `...`
`String`	`size`, `concat`, `toUpper`, `substring`, `...`

Die Zusammenfassung von **mehreren Objekten** nennt man dann **Collection**. Eine **Collection** ist dabei ein **abstrakter Datentyp**, wobei konkrete, von ihr abgeleitete, Datentypen mit **Set** – einer Menge im mathematischen Sinn – **Bag** – ein Set, in welchem Duplikate vorkommen können – und **Sequence** – einem geordneten Bag – gegeben. Operationen auf Collections werden stets durch einen **Pfeil** `->` eingeleitet.
Die folgenden Konstrukte sollen nun am folgenden Beispiel näher erläutert werden:



Kontext Die Auswertung eines OCL-Ausdrucks erfolgt stets in einem **Kontext**. Dies kann sowohl eine Klasse, ein Interface oder aber auch eine Methode sein. Die Syntax sieht wie folgt aus:

```
context <<class-/interface-name>> | typename::operationName(p1 : t1, ..., pn : tn) :
    returntype
```

In unserem Beispiel:

```
context Person
```

Die Auswertung findet somit in der Klasse `Person` statt, soll die Auswertung aber in der Methode `abheben(betrag : Real)` der Klasse `Bankkonto` stattfinden, so schreiben wir:

```
context Bankkonto::abheben(betrag : Real) : Real
```

Navigation Wir müssen hier in `Collection`-Typen und andere Typen unterscheiden. Für andere Typen erfolgt der Zugriff auf Attribute oder Methode durch die – zum Beispiel aus Java bekannte – Punktnotation. Für `Collection`-Typen erfolgt sie über die Pfeilschreibweise. An unserem Beispiel: Es soll auf das Attribut `alter` der Klasse `Person` zugegriffen werden und es soll die Operation `forAll` auf das Attribut `hält` der Klasse `Person` aufgerufen werden.

```
Person.name
Person.haelt->forAll( ... )
```

Invarianten Eine Invariante eines Objektes ist eine Aussage, die wahr sein muss, damit der Zustand des Objektes wohldefiniert ist. Sie wird stets mit `inv` eingeleitet und ist ein Wahrheitsausdruck. Es kann dabei pro Kontext mehr als eine Invariante definiert werden. Sie müssen immer vor und nach der Ausführung von Operationen gelten. An unserem Beispiel soll sichergestellt werden, dass der Besizter eines Autos mindestens 18 Jahre alt ist.

```
1 context Auto
2 inv: Besitzer.alter >= 18
```

Instanzen Das Schlüsselwort `self` wird verwendet, um explizit auf den momentanen Kontext zu verweisen (es verhält sich ähnlich dem Schlüsselwort `this` aus Java), es ist redundant und kann damit weggelassen werden, wenn klar ist, dass das folgende Attribut sich auf den Kontext bezieht. Die Operation `allInstances` liefert eine Collection zurück, die alle Instanzen einer Klasse enthält. In unserem Beispiel soll sichergestellt werden, dass es mindestens eine Bank gibt, sowie der Name der Bank mehr als zwei Zeichen enthält.

```
1 context Bank
2 inv: self.name.size() > 2 and Bank.allInstances()->size() >= 1
```

Initialisierungen Mittels Initialisierungen, welche mit `init` eingeleitet werden, können Anfangswerte für Attribute definiert werden. Der Kontext hierfür ist das zugehörige Attribut, nicht die Klasse. In unserem Beispiel soll der Anfangswert 0 des Attributes `alter` der Klasse `Person` definiert werden.

```
1 context Person::alter : Integer
2 init: 0
```

Vor- und Nachbedingungen Mit OCL-Ausdrücken können Vor- und Nachbedingungen für Operationen formuliert werden. Vorbedingungen werden durch `pre` eingeleitet, sie müssen zum Zeitpunkt des Operationsaufrufs wahr sein, damit die Nachbedingungen, eingeleitet durch `post`, unmittelbar nach der Ausführung der Operation gelten müssen. Das Schlüsselwort `result` bezeichnet dabei – bei Existenz – den Rückgabewert der Operation. Werden in den Nachbedingung Werte benötigt, die zum Zeitpunkt der Vorbedingung gelten, so werden sie mit dem Suffix `@pre` versehen.
In unserem Beispiel soll für einen positiven Wert von `neuesAlter` die Methode `setzeAlter` das Alter `alter` der Person auf das übergebene Alter setzen, und das alte Alter ausgeben.

```
1 context Person::setzeAlter(neuesAlter: Integer) : Integer
2 pre: neuesAlter > 0
3 post: result = self.alter@pre and alter = neuesAlter
```

Hilfsvariablen Mit `let ... in ..` kann eine Hilfsvariable für einen Ausdruck (nach `let`) eingeführt werden, die im Folgenden (nach `in`) verwendet werden kann. In unserem Beispiel soll beim `umlackieren` eines Fahrzeugs die Farbe danach eine Mischung aus der neuen Farbe sowie noch fünf Prozent der alten Farbe (`Farbe::prozentual`) sein.

```
1 context Fahrzeug::umlackieren(neueFarbe : Farbe) : void
2 post: let oldncol = farbe@pre::prozentual(0.05) =
3       in farbe = Farbe::mische(oldncol, neueFarbe)
```

Selektionen Mit der `select`-Operation werden die Objekte einer **Collection** gewählt, die einen bestimmten boolschen Ausdruck erfüllen. Das Ergebnis des Ausdrucks ist wiederum eine **Collection**. Die allgemeine Syntax eines solchen Ausdrucks ist:

```
1 collection->select(v : Type | booleanexpression) : collection
```

Allquantor Die `forAll`-Operation dient zur Einschränkung über alle Objekte (er stellt den Allquantor dar), ihr Rückgabewert ist ein Wahrheitswert. Die allgemeine Syntax eines solchen Ausdrucks ist:

```
collection->forAll(v : Type | booleanexpression) : boolean
```

Existenzquantor Der Existenzquantor wird mittels der `exists`-Operation ausgedrückt, deren Rückgabewert ein Wahrheitswert ist. Die allgemeine Syntax eines solchen Ausdrucks ist:

```
collection->exists(v : Type | booleanexpression) : boolean
```

Größen Die `size`-Operation liefert die Größe einer Collection, ihr Rückgabewert ist ein `Integer`. Die allgemeine Syntax eines solchen Ausdrucks ist:

```
collection->size() : integer
```

Die `count`-Operation liefert die Anzahl eines Elements in einer Collection, ihr Rückgabewert ist ein `Integer`. Die allgemeine Syntax eines solchen Ausdrucks ist:

```
collection->count(v : Type) : integer
```

Element einer Collection Mit den Operationen `include` und `exclude` kann man überprüfen, ob ein Element in oder nicht in einer Collection ist. Die allgemeine Syntax eines solchen Ausdrucks ist:

```
collection->include(v : Type) : boolean
collection->exclude(v : Type) : boolean
```

In unserem Beispiel soll eine Person bei der Einstellung zu den Angestellten hinzugefügt werden, wenn sie davor noch nicht Teil der Angestellten waren.

```
context Bank::einstellen(p : Person) : void
pre: not self.Angestellte->includes(p)
post self.Angestellte->includes(p)
```

Teilmengen und Disjunktheit Mit den Operationen `includesAll` und `excludesAll` kann man überprüfen, ob eine andere Collection Teilmenge einer oder disjunkt zu einer Collection ist. Die allgemeine Syntax eines solchen Ausdrucks ist:

```
collection->includesAll(c : Collection) : boolean -- Teilmengen
collection->excludesAll(c : Collection) : boolean -- Disjunktheit
```

Leerheit Mit den Operationen `isEmpty()` und `notEmpty()` kann überprüft werden, ob eine Collection leer oder nicht leer ist.

```
collection->isEmpty() : boolean
collection->notEmpty() : boolean
```

Vor- und Nachteile der OCL Die OCL ist als Schnittstellenbeschreibungssprache gut geeignet, sowie lassen sich Konsistenzeigenschaften statisch überprüfen. Dadurch erlaubt sie eine dynamische Überprüfung der Einhaltung der Constrains sowie die teilweise automatisierte Generierung von Testtreiben sowie die Implementierung von Fehlertoleranzmechanismen.
Als **Nachteile** sind die ausdruckshafte Beschränkung auf Beschreibungen von stets oder punktuell einzuhaltenden Einschränkungen, die fehlende Kombination von Operationen oder eines Zeitbezugs, die reine Datenbasiertheit sowie die fehlende Berücksichtigung der zur Ausführung erforderlichen Dienste zu nennen.

2.6 Graphische Benutzeroberflächen

Definition 9 (*Bedienoberflächen*)

Eine **Bedienoberfläche** ist eine Sammlung von Techniken und Mechanismen zur **Interaktion** mit einem System. Eine **Aktion** ist eine Operation, die ein Benutzer dabei auf Oberflächenobjekte ausführen kann.

Trivialerweise ist eine gut gestaltete Bedienoberfläche entscheidend für den Erfolg eines Systems.

Softwareergonomie Ziel der **Softwareergonomie** ist die Anpassung der Eigenschaften von Software an die psychischen Eigenschaften der menschlichen Nutzer. Für die Dialoggestaltung gibt es mit der **EN ISO 9241-10** Richtlinien für die Dialoggestaltung.

Aufgabenangemessenheit Unterstützen des Nutzers bei der effektiven und effizienten Arbeitsaufgabenerledigung

Selbstbeschreibungsfähigkeit Jeder einzelne Dialogschritt ist durch Rückmeldung des Dialogsystems **unmittelbar verständlich** oder wird auf Nachfrage erklärt.

Steuerbarkeit Der Benutzer ist in der Lage den Dialogablauf solange zu steuern sowie die Richtung und Geschwindigkeit zu beeinflussen bis das Ziel erreicht ist.

Erwartungskonformität Der Dialog ist konsistent und entspricht den Merkmalen des Benutzers.

Fehlertoleranz Erreichen des beabsichtigten Arbeitsergebnisses trotz fehlerhafter Eingabe nur mit minimalen oder keinem Korrekturaufwand.

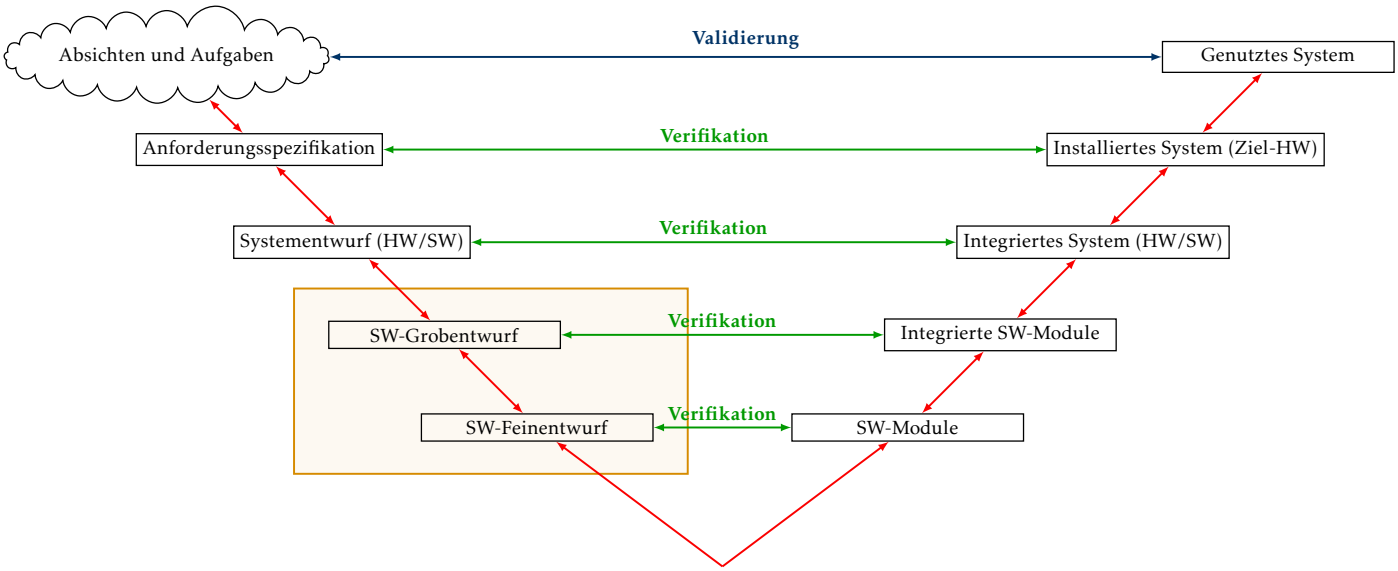
Individualisierbarkeit Zulassen des Dialogsystems von Anpassungen an die Erfordernisse der Arbeitsaufgabe sowie an individuelle Fähigkeiten und Vorlieben des Benutzers.

Lernförderlichkeit Der Dialog unterstützt und leitet den Benutzer zum Erlernen des Dialogsystems an.

KAPITEL 3

SOFTWAREENTWURF

Wo befinden wir uns gerade?



3.1 Einführung

Aus dem allgemeinen Ziel des Softwareentwurfs ergeben sich die tatsächlichen Ziele und Aufgaben der Entwurfsphase. Sie seien definiert als das „Entwerfen einer Software**architektur**“, wozu das definierte System in Systemkomponenten zerlegt, durch geeignete Anordnung jener strukturiert, sowie die Beziehungen sowie die kommunizierenden Schnittstellen jeder Komponenten beschrieben und festgelegt werden, die Spezifikation des Funktions- resp. Leistungsumfangs jener **Komponenten** sowie die Detaillierung der Systemkomponenten durch Beschreibung ihrer Operationen.

3.2 Softwaregrobentwurf

Ziel ist es eine Softwarearchitektur zu erstellen, welche sowohl die **Anforderungen der Spezifikation** – sowohl Produkt- als auch Qualtitätsanforderungen – erfüllt als auch die **Schnittstellen zur Umgebung** bereitstellt.

anzuwendende Prinzipien sind sowohl die **Zerlegung**, welche das System in miteinander interagierende Bausteine **zerlegt**/aufteilt, als auch die **Abstraktion**, welche ein Teilsystem mit seiner im Gesamtsystem zu übernehmenden funktionalen Rolle – ohne Berücksichtigung jeglicher Implementierungsdetails – indentifiziert.

Vorgehensweisen sind mit **Top-Down** und **Bottom-Up** gegeben, welche nicht exklusiv aufeinander wirken. Mit **Top-Down** spezialisiert man immer weiter, so dass das Gesamtsystem schrittweise

in Teilsysteme zerlegt wird, mit **Bottom-Up** generalisiert man immer weiter, so dass das Gesamtsystem schrittweise aus Teilsystemen zusammengesetzt wird.

Ergebnis ist eine **Architektur**, also eine strukturierte/hierarchische interrelationsinklusive Anordnung der Systemkomponenten, sowie eine **Spezifikation** einer jeden **Komponente**, also eine Festlegung von Schnittstelle sowie Funktions- und Leistungsumfang.

3.3 Klassische Architekturmodelle

Blockdiagramm als stärkste Vereinfachung des Architekturentwurfs. Jeder Block stelle dabei ein eigenständiges Subsystem dar, welches wiederum aus mehreren Subsystemen bestehen kann (sogenannte „Blöcke in Blöcken“). Kommunikation für Daten oder Steuerungsbefehle werden über *gerichtete Kanten* zwischen den kommunizierenden Blöcken dargestellt. Die Kommunikation verläuft in Kantenrichtung.
Wir verwenden das Blockdiagramm in der ersten Phase des Architekturentwurfs um einen **Überblick über die Systemstruktur** zu erhalten und das System in mehrere miteinander zusammenarbeitende Subsysteme aufzuteilen.

Datenmodelle — Ein Gesamtsystem aufspannende Subsysteme **müssen** einen Informationsaustausch für eine effektive Arbeitsweise vornehmen. Wir wollen zwei grundlegende Methoden unterscheiden:

Datenspeichermodell — Hierbei werden alle gemeinsam benutzten Daten in **einer zentralen Datenbank** gespeichert, welche wiederum für alle Subsysteme zugänglich ist. Damit eignet sich dieses Modell besonders für auf globale Datenbanken oder Quelllager basierende Systeme.

Dezentrales Datenmodell — Hierbei erhält **jedes** Subsystem seine **eigene Datenbank**. Die Intersubsystemdatenkommunikation findet dann über das Versenden von Nachrichten statt.

Schichtenmodell als ein auf einer Hierarchie aufsteigender Abstraktionsebenen **schichtweise** aufgebautes Architekturmodell. Dabei soll eine **jede Schicht des Modells** ein Paket von Diensten, zu deren Implementierung nur Dienste der **darunterliegenden** Schichten verwendet werden sollen, bereitstellen. Als Beispiel sei das OSI-Referenzmodell genannt.
Ein solcher Ansatz begünstigt selbstverständlich die **schrittweise Entwicklung** von Systemen, als dass nach Entwicklung *einer Schicht*, dessen Dienste Nutzern verfügbar gemacht werden können. Eine solche Architektur ist dementsprechend auch **veränderbar** und **portierbar**. Wichtig hierfür ist eine Nichtveränderung der Schnittstelle, da dann die komplette Schicht einfach ausgetauscht werden kann. Bei Änderungen betrifft dies aber **nur** die **angrenzende** Schicht.

Klient/Server-Modell als ein *verteiltes Systemmodell* zur Darstellung der Verteilungsmöglichkeiten von Daten und Prozessen auf eine Menge von Prozessoren, dessen Hauptbestandteile die Folgenden sind:

- Eine Menge unabhängiger **Auftragnehmer** (*server*), die Dienste für andere Subsysteme anbieten
- Eine Menge *im Allgemeinen* unabhängiger **Klienten** (*clients*), welche die von den Auftragnehmern angebotenen Dienste abrufen.
- Ein **Netzwerk** zur Kommunikation zwischen Auftragsnehmer und -geber, also ein Netzwerk über welches die Klienten auf die Dienste zugreifen können.

Letzterer Punkt ist redundant bei einer Ausführung von Klienten und Auftragsnehmern auf *demselben* Computer.

Verteiltes System — Bei einem **verteilten System** läuft die Software auf einer durch das Netzwerk nur lose miteinander verbundenen Gruppe von Rechnern, sie bestehen damit aus **lose integrierten, unabhängigen Teilen**. Eine plattformübergreifende Kommunikationsermöglichung sowie eine teilweise Systemverwaltung bietet die **Middleware**. Vorteile bilden sich mit der …

… Ressourcenteilung	… Nebenläufigkeit	… Fehlertoleranz
… Offenheit	… Skalierbarkeit	… Transparenz

Nachteile ergeben sich mit einer **erhöhten Komplexität**, einer gewissen **Unvorhersagbarkeit**, einer **erschwerten Verwaltbarkeit** sowie einem **erschwerten Sicherstellen der Informationssicherheit**.

3.4 Kohäsion und Kopplung

3.4.1 Kopplung

Definition 10 (*Kopplung*)

Der Grad der Interaktion zweier Komponenten heiße **Kopplung**. Je geringer die Kopplung, desto besser ist die **Transparenz** und **Wartbarkeit**. Deshalb sei es das Ziel des Grobentwurfs eine möglichst **geringe** Kopplung zu erzielen.

Wir unterteilen die Kopplung in fünf Kategorien:

Inhaltskopplung (sehr schlecht) Zwei Module heißen **inhaltsgekoppelt**, wenn *eines der Module unmittelbare Auswirkungen auf den Inhalt oder auf den auszuführenden Kodeteil eines anderen Moduls* haben kann. Die Inhaltskopplung beruht dabei **nicht auf einem eigentlichen Aufrufs des anderen Moduls**, die Änderungen werden vielmehr **direkt initiiert**.

Globalkopplung (schlecht) Zwei Module heißen **datenextern** oder **global** gekoppelt, wenn **beide** auf **gemeinsame globale Daten lesend und schreibend zugreifen** können.

Kontrollkopplung Zwei Module heißen **kontrollgekoppelt**, wenn *ein Modul ein **kontrollflussbestimmendes Element** an ein anderes Modul sendet*. Das Datum wird damit nur gesendet um die interne Logik des anderen Moduls zu steuern.
Dies führt zur Abhängigkeit der beiden Module und widerspricht dem *black-box*-Prinzip, wodurch die Wiederverwendbarkeit einschränkt ist. (tritt vor allem bei **logischer Kohäsion** auf)

Datenstrukturkopplung (gut) Zwei Module heißen **datenstrukturgekoppelt**, wenn *ein Modul eine Datenstruktur als Parameter an ein weiteres Modul übergibt, das aufgerufene Modul aber **nur Teile der Datenstruktur verwendet***.
Dies erzeugt wieder eine Abhängigkeit zwischen ansonst unabhängigen Modulen und macht sich insbesondere bei Änderungsvorgängen unangenehm bemerkbar.

Datenkopplung (sehr gut) Zwei Module heißen **datengekoppelt**, wenn *sie Informationen über Parameter austauschen, welche homogene Daten* sind. Von einem *homogenen Datum* spricht man bei *einfachen Daten* oder *Datenstrukturen, deren Elemente **alle** vom aufgerufenen Modul verwendet werden*.

3.4.2 Kohäsion

Definition 11 (*Kohäsion*)

> Der Grad der funktionalen Bindung **innerhalb** einer Komponente heiße **Kohäsion**. Je höher die Kohäsion, desto besser ist die **Transparenz** und **Wartbarkeit**. Deshalb sei es das Ziel des Grobentwurfs eine möglichst **hohe** Kohäsion zu erzielen.

Wir unterteilen die Kohäsion in sieben Kategorien:

Zufällige Kohäsion $(---)$ Ein Modul hat **zufällige Kohäsion**, wenn es *mehrere, vollkommen unzusammenhängende, Aktionen durchführt*. Dadurch ergibt sich eine schlechte Änderbarkeit und Wiederverwendbarkeit.

Logische Kohäsion $(--)$ Ein Modul hat **logische Kohäsion**, wenn es *eine Menge verwandter und zum Teil alternativer Funktionalitäten anbietet, wobei einzelne Kodeteile im Allgemeinen zur Realisierung verschiedener Funktionalitäten beitragen*. Dadurch ergibt sich ein schweres Verständnis von Schnittstellen und eine schlechte Wiederverwendbarkeit.

Zeitliche Kohäsion $(-)$ Ein Modul hat **zeitliche Kohäsion**, wenn es *eine Reihe von Aktionen in zeitlichem Zusammenhang* ausführt. Die Reihenfolge ist dabei irrelevant. Die Aktionen haben dabei nur schwachen Bezug aufeinander, aber einen starken Bezug zu Aktionen in anderen Modulen, sind deswegen auch nur schlecht wiederverwendbar.

Prozedurale Kohäsion Ein Modul hat **prozedurale Kohäsion**, wenn es *eine Reihe von Aktionen auf **unterschiedlichen Daten** in zeitlicher Abfolge durchführt*. Dabei ist die Reihenfolge **relevant**. Die Aktionen haben dabei nur schwachen Bezug aufeinander, aber einen starken Bezug zu Aktionen in anderen Modulen, sind deswegen auch nur schlecht wiederverwendbar.

Kommunikative Kohäsion $(+)$ Ein Modul hat **kommunikative Kohäsion**, wenn es *eine Reihe von Aktionen auf **gemeinsame** Daten ausführt, wobei die Reihenfolge **irrelevant** ist*. Es ergibt sich eine schlechte Wiederverwendbarkeit.

Sequentielle Kohäsion $(++)$ Ein Modul hat **sequentielle Kohäsion**, wenn es *eine Reihe von Aktionen in **sequentieller** Abfolge ausführt*. Die Ausgabedaten werden als Eingabedaten weiterverarbeitet.

Funktionale Kohäsion $(+++)$ Ein Modul hat **funktionale Kohäsion**, wenn *alle seine Elemente zur Ausführung einer **und nur einer** Aufgabe beitragen*.

Zusammenfassend ergibt sich folgendes Entscheidungsmuster:
Man stellt sich zuerst die Frage: *„Gibt es mehrere Funktionen?“*

- ↪ **JA**, so stellt man sich die Frage: *„Arbeiten diese auf gemeinsamen Daten?“*
  - ↪ **JA**, so stellt man sich die Frage: *„Ist die Reihenfolge dieser relevant?“*
    - ↪ **JA**, so ist es **sequentielle Kohäsion**
    - ↪ **NEIN**, so ist es **kommunikative Kohäsion**
  - ↪ **NEIN**, so stellt man sich die Frage: *„Werden sie in einem zeitlichen Zusammenhang aktiviert?“*
    - ↪ **JA**, so stellt man sich die Frage: *„Ist die Reihenfolge dieser relevant?“*
      - ↪ **JA**, so ist es **prozedurale Kohäsion**
      - ↪ **NEIN**, so ist es **temporale Kohäsion**
    - ↪ **NEIN**, so stellt man sich die Frage: *„Bilden sie verwandte, alternative Funktionalitäten?“*
      - ↪ **JA**, so ist es **logische Kohäsion**
      - ↪ **NEIN**, so ist es **zufällige Kohäsion**

↪ **NEIN**, so ist es **funktionale Kohäsion**

Man kann zudem noch von **Informationskohäsion** sprechen. Ein Modul hat **Informationskohäsion**, wenn es *eine Reihe von Operationen – jede davon mit eigenem Ein- und Ausgang sowie **unabhängigem** Kode – auf **derselben** Datenstruktur durchführen kann*. Im Wesentlichen handelt es sich hierbei um abstrakte Datentypen.

3.5 Softwarefeinentwurf

Ziel ist es die im Softwaregrobentwurf erstellte Softwarearchitektur zu verfeinern. Dabei sollte der Feinentwurf dann so detailliert sein, dass ein Programmierer den Programmkode **ohne weitere Interpretation der Aufgabenstellung** schreiben kann.

3.6 Programmiersprachenneutrale Notationen

Ziel ist eine Beschreibung von Operationen durch das Abstrahieren syntaktischer Programmiersprachendetails.

Dadurch ist eine Wahl der Programmiersprache noch nicht zwingend erforderlich und man kann die Aufmerksamkeit auf die Korrektheit des Algorithmus führen. Wir beleuchten zwei Beispiele näher, Nassi-Schneidermann-Diagramme (Struktogramme) und Pseudocode.

3.6.1 Struktogramm

Ein Struktogramm ist eine graphische Darstellung des Kontroll- und Datenflusses eines in **imperativer Sprache** darzustellenden Algorithmus. Dabei werden die vier Elemente eines wohlstrukturierten Programms, der **Befehl**, die **Sequenz**, die **Verzweigung** und die **Schleife** durch die im Folgenden angegebenen Diagrammarten versinnbildlicht.

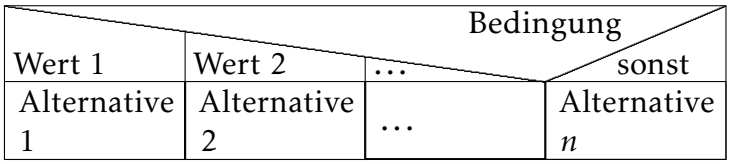
Sequenz von Anweisungen

Anweisung 1
Anweisung 2
Anweisung 3
…

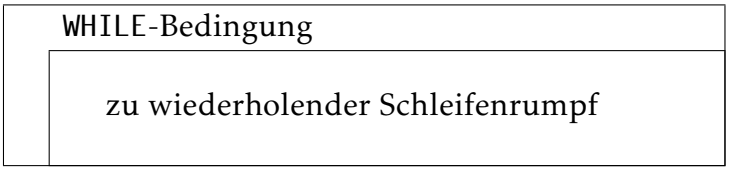
Verzweigungen mit `if`

Bedingung: Ja	Bedingung: Nein
`THEN`-Teil	`ELSE`-Teil, kann auch leer sein.
∅	

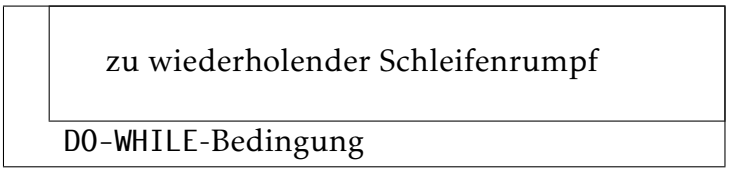
Verzweigungen mit `case`



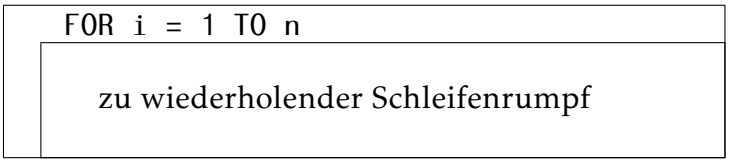
Schleifen mit `while`



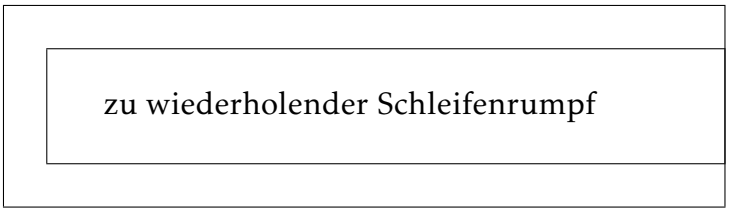
Schleifen mit `do-while`



Schleifen mit `for`



Endlosschleifen



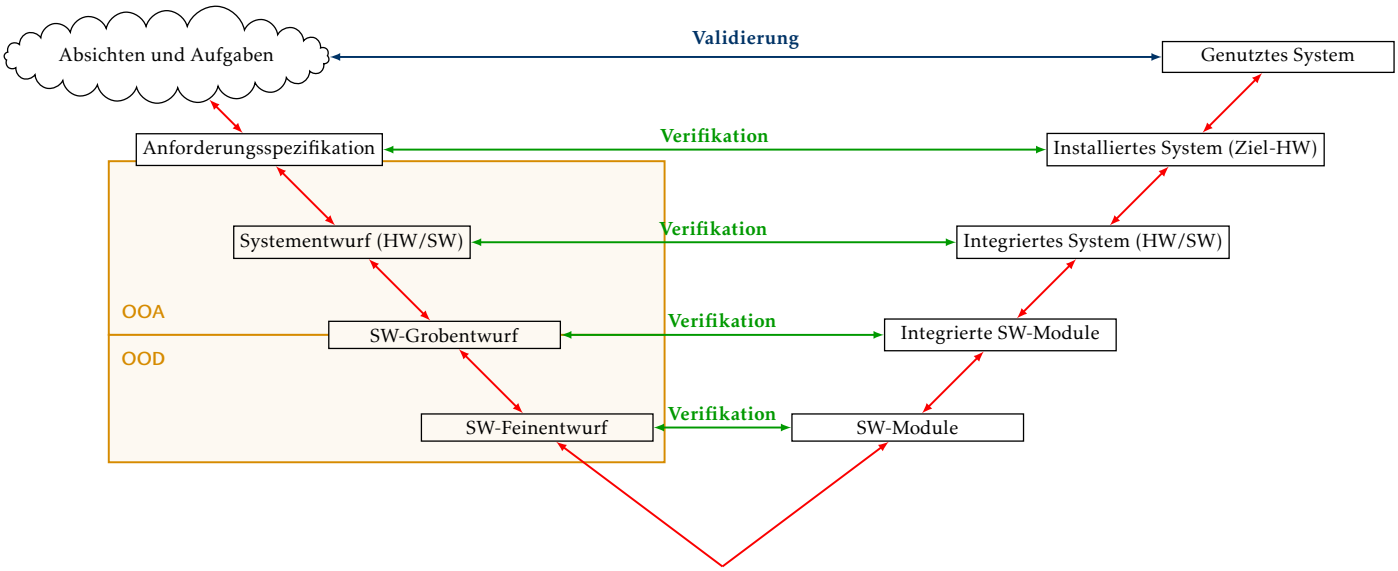
3.6.2 Pseudocode

Pseudocode ist eine Darstellung eines Algorithmus in einer intuitiv verständlichen Sprache, die an eine Programmiersprache angelehnt ist, aber **noch leichter lesbar ist als ausformulierter Programmcode**.

KAPITEL 4

Objektorientierte Analyse und Design

Wo befinden wir uns gerade?



4.1 Einführung

Definition 12 (*Klasse*)

Eine **Klasse** ist in der objektorientierten Programmierung eine Zusammenfassung einer Datenstruktur mit den auf ihr anwendbaren Operationen zu einer Einheit.

Definition 13 (*Objekt*)

Ein **Objekt** ist in der objektorientierten Programmierung ein Softwaregebilde mit individuellen Merkmalen. Es ist definiert durch seine*Identität*, seinen *Zustand* und sein *Verhalten*.
Der Zustand ist dabei durch die aktuellen Werte der Instanzvariablen festgelegt, das Verhalten eines Objektes wird durch Methoden implementiert.

4.2 Objektorientierte Analyse — OOA

Ziel ist es, *geeignete Klassen zur Modellierung des Problems* und *ihre Beziehungen untereinander* zu identifizieren, was durch **statische** und *dynamische* Modellierung erfolgt. Im Wesentlichen entspricht die objektorientierte Analyse damit der Zielsetzung der Spezifikation und zum Teil des Grobentwurfs, denn sie liefert eine Antwort auf die Frage, was das System macht, sowie eine Definition des Grundgerüsts des Systems **ohne** auf die Verhaltensrealisierung im Einzelfall einzugehen.

4.2.1 OOA: Statische Modellierung

Gegenstand eines statischen Systemmodells ist die Modellierung der **statischen Systemstruktur**:

Ziel ist die Modularisierung des Systems durch die *Identifikation der relevanten Klassen*, der *Beschreibung ihrer **statischen** Eigenschaften*, die *Analyse der Verantwortlichkeiten der Klassen*, die *Beschreibung der Beziehungen zwischen Klassen* sowie die anschließende *Definition der Eigenschaften der Datenspeicherung und -persistenz*.

Wir schaffen dies graphisch durch die UML-*Klassendiagramme*, auch hierzu sei auf ein Merkblatt aus KonzMod verwiesen.
Im Allgemeinen kommt man durch drei Schritte zum statischen Modell:

1. Identifikation der Klassenkandidaten
2. Identifikation der Assoziationen
3. Spezifikation der Attribute

4.2.2 OOA: Dynamische Modellierung

Bei der **dynamischen Modellierung** soll das Verhalten des Systems modelliert werden. Dazu werden Anwendungsfälle mit Hilfe von Sequenz- und Kommunikationsdiagrammen verfeinert. Diese sogenannten **Szenarien** tragen im Wesentlichen zur Definition des Flusses der **Botschaften** durch das System bei.

4.2.2.1 Sequenzdiagramme

Für genauere Informationen siehe das Merkblatt am Ende. Hier werden Nachrichten zwischen Objekten **im zeitlichen Verlauf** dargestellt. Es **betont** damit den **dynamischen Ablauf** und die **zeitliche Abfolge des Nachrichtenaustauschs**.

4.2.2.2 Kommunikationsdiagramm

In Kommunikationsdiagrammen werden beteiligte **Objekte** und ihre Beziehungen **untereinander** dargestellt. Der Nachrichtenaustausch erfolgt **entlang den Verknüpfungskanten**, welche *zeitlich* durch ein Nummerierungsschema geordnet sind. Es **betont** damit die **statischen Verknüpfungen** zwischen den **interagierenden Objekten**.

4.3 Objektorientiertes Design — OOD

Ziel ist es einen bereit bestehenden Grobentwurf mit einer Architektur gemäß dem OOA-Modell durch *Vervollständigung der Klassendiagramme*, sowie der *Interaktionsdiagramme* und bei Bedarf *Modellierung des internen Verhaltens einer Klasse* zum Feinentwurf zu detaillieren. Dies erfolgt ebenfalls in iterativen, sich gegenseitig ergänzenden **statischen** und **dynamischen** Modellierungsphasen. Die logische und physikalische Struktur des Systems wird dabei in der **Architekturmodellierungsphase** festgelegt.

4.3.1 OOD: Architekturmodellierung

Bislang war es uns noch nicht möglich, ein System zu modularisieren. Wir wollen das, da dies die Skalierbarkeit des Systems unterstützt, sowie die Wartbarkeit deutlich verbessert. Wir unterscheiden zwischen den folgenden Teilaspekten:

Logische Systemarchitektur beschreibt die Zusammenfassung der Klassen zu Paketen und Komponenten

Physikalische Systemarchitektur beschreibt die Verteilung und Kommunikation mehrerer Kompoenten auf mehrere Rechner

Für die logische Sicht gibt es **Paket**- und **Komponentendiagramme**, für die physikalische Sicht **Einsatzdiagramme**.

4.3.2 OOD: Statische Modellierung

Ziel ist es, die in der OOA entwickelten UML-Klassendiagramme zu **präzisieren**. Dabei werden eventuell *Klassen ergänzt*, Klassen um *Operationen und Attribute ergänzt*, sowie die zum Teil bereits ermittelten *Assoziationen präzisiert*.

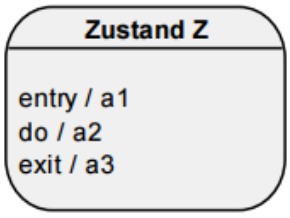
Definition 14 (*Interface*)

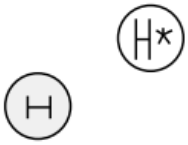
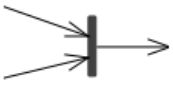
Eine abstrakte Klasse mit *keiner implementierten Methode*, sowie *keinen Attributen* heiße **Interface**.

4.3.3 OOD: Dynamische Modellierung

Ziel ist es, die in der OOA entstandenen, zum Teil noch unvollständigen Interaktionsdiagramme noch weiter zu verfeinern.

Zustandsdiagramme Ein **Zustandsdiagramm** ist die graphische Repräsentation eines Zustandsautomaten. Er beschreibt das *dynamische Verhalten von Objekten anhand von ereignisgesteuerten Übergängen zwischen diskreten Zuständen*.

Modellierungselemente	Bedeutung
	Zustand: • Beim Betreten wird Aktion a1, während des Zustandes Aktion a2 und beim Verlassen Aktion a3 ausgeführt. • Zustände können selbst wieder durch Zustandsmaschinen näher beschrieben werden. Der Zustand kann dann erst bei Erreichen des Endzustandes vom inneren Automat verlassen werden. • Zustände können durch mehrere parallele Automaten verfeinert werden. Diese werden durch horizontale Linien getrennt
	Zustandsübergänge können mit *dem auslösenden Ereignis*, *dem ausgelösten Ereignis* sowie einer *Wächterbedingung* versehen werden.

Modellierungselemente	Bedeutung
	Startzustand — Hier beginnt das Zustandsdiagramm
	Endzustand — Hier endet das Zustandsdiagramm
	Terminationsknoten — Das modellierte Objekt höhrt auf zu existieren.
	„Rücksprungadressen“
	Knoten von dem aus mehrere alternative Transitionen ausgehen können.
	Aufspaltung des Kontrollflusses in mehrere parallele Zustände
	Zusammenführung des Kontrollflusses von mehreren parallelen Zuständen

KAPITEL 5

Entwurfsmuster

Entwurfsmuster lassen sich nach Aufgabe und Gültigkeitsbereich unterteilen. Wir unterscheiden als Aufgabe:

Erzeugungsmuster mit *erzeugenden Aufgaben* betreffen den **Prozess der Objekterzeugung**

Strukturmuster mit *strukturorientierten Aufgaben* befassen sich mit der **Zusammensetzung von Klassen und Objekten**

Verhaltensmuster mit *verhaltensorientierten Aufgaben* charakterisieren die **Art und Weise der Zusammenarbeit und Verantwortungsaufteilung zwischen Klassen und Objekten**

Wir unterscheiden folgende Gültigkeitsbereiche:

Klassenbasierte Entwurfsmuster befassen sich im Wesentlichen mit *statischen auf **Vererbung** basierenden **Klassen**beziehungen*

Objektbasierte Entwurfsmuster befassen sich im Wesentlichen mit *dynamischen **Objekt**beziehungen*

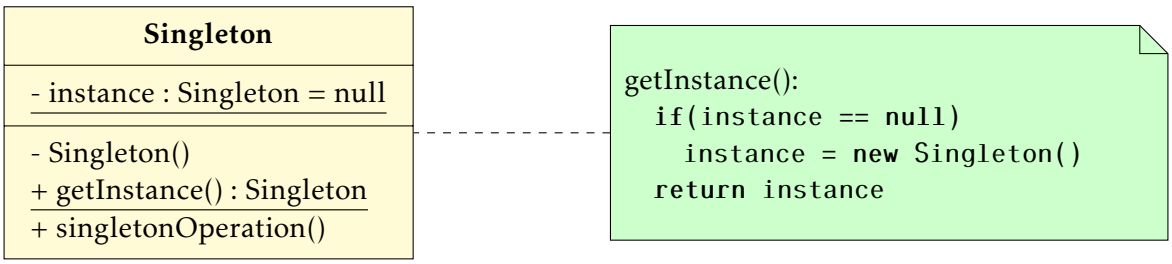
Die im Folgenden besprochenen Entwurfsmuster lassen sich so untergliedern. Wir zeigen dies mit folgender Matrix:

	Erzeugungsmuster	**Strukturmuster**	**Verhaltensmuster**
klassenbasiert		Adapter	Schablonenmethode
objektbasiert	Singleton Abstrakte Fabrik	Adapter Brücke Dekorierer Kompositum Proxy	Befehl Beobachter Strategie Zuständigkeitskette

5.1 Erzeugungsmuster

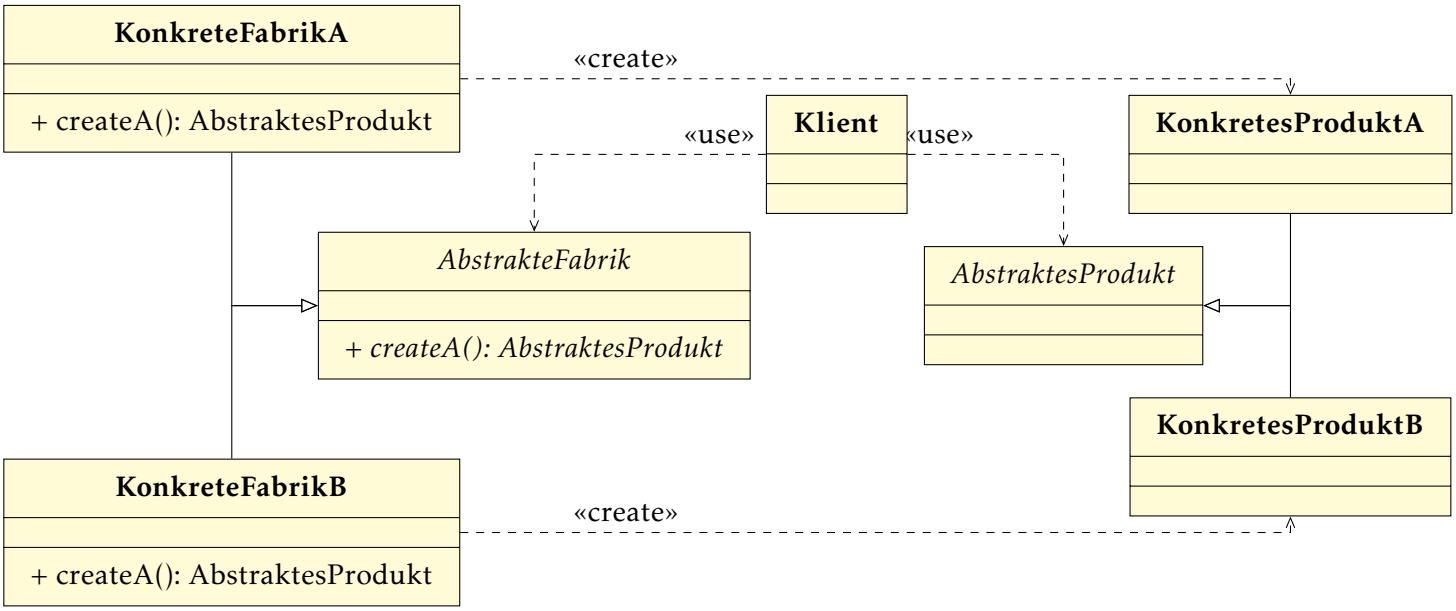
5.1.1 Singleton

Von einer Klasse soll **immer nur eine Instanz** existieren. Zugriff auf diese Instanz soll **global** erfolgen.



5.1.2 Abstrakte Fabrik

Es sollen **verwandte** oder auch **voneinander abhängige Objekte** durch einen **einheitlichen Erzeugungsprozess**, welcher *zur Laufzeit* aus einer vorgegebenen Menge ausgewählt wird, **erstellt werden**.

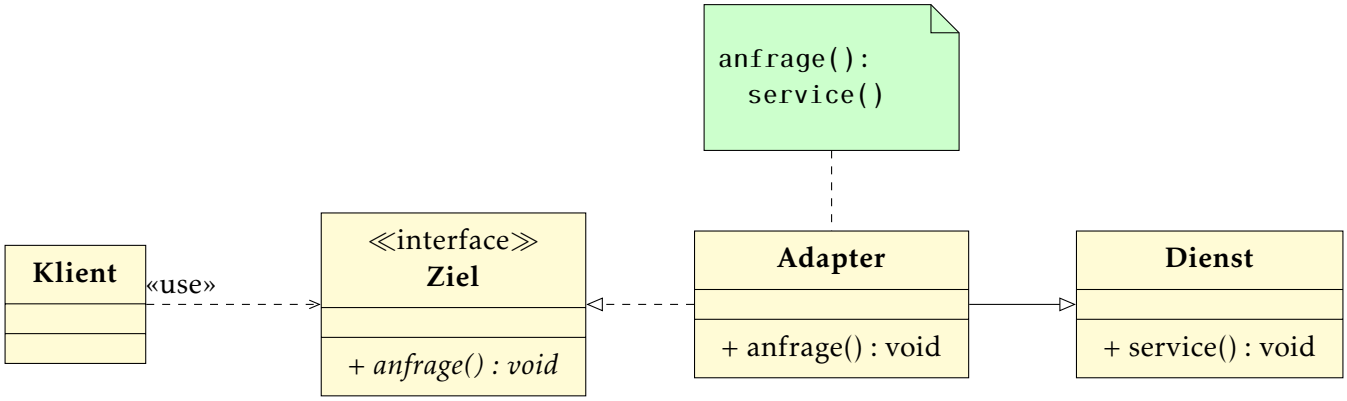


5.2 Strukturmuster

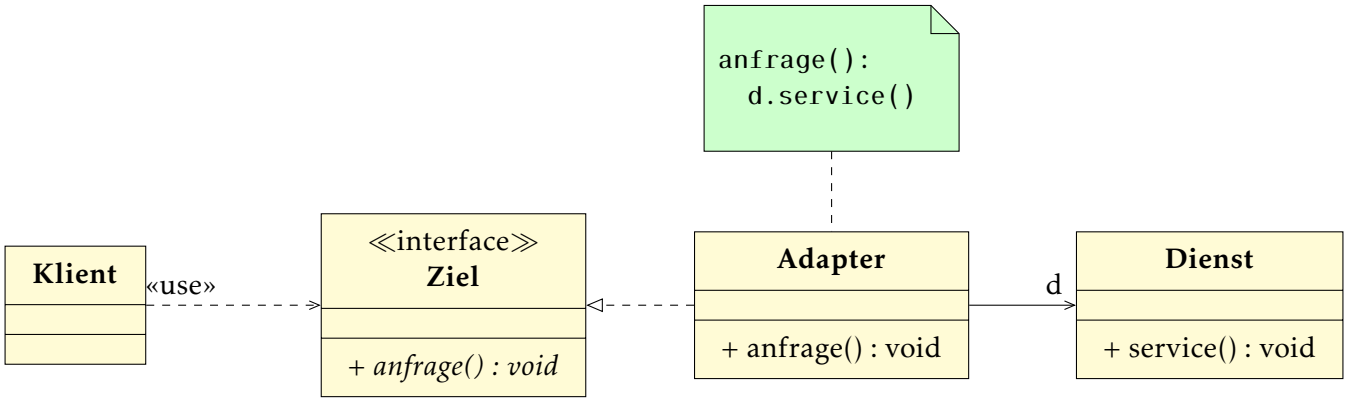
5.2.1 Adapter

Es soll eine **vorgegebene Schnittstelle** einer Klasse in eine *andere Schnittstelle*, wie sie vom Dienstnutzer erwartet wird, **übersetzt werden**. Dies ist einerseits über *Vererbung* – man spreche dann von einem *klassenbasierten Adapter* – andererseits über *Delegation* – hier von einem *objektbasiertem Adapter* – möglich. Bei ersterem kommt es zur **Selbstdelegation** bei letzterem **nicht**.

5.2.1.1 Klassenbasierter Adapter

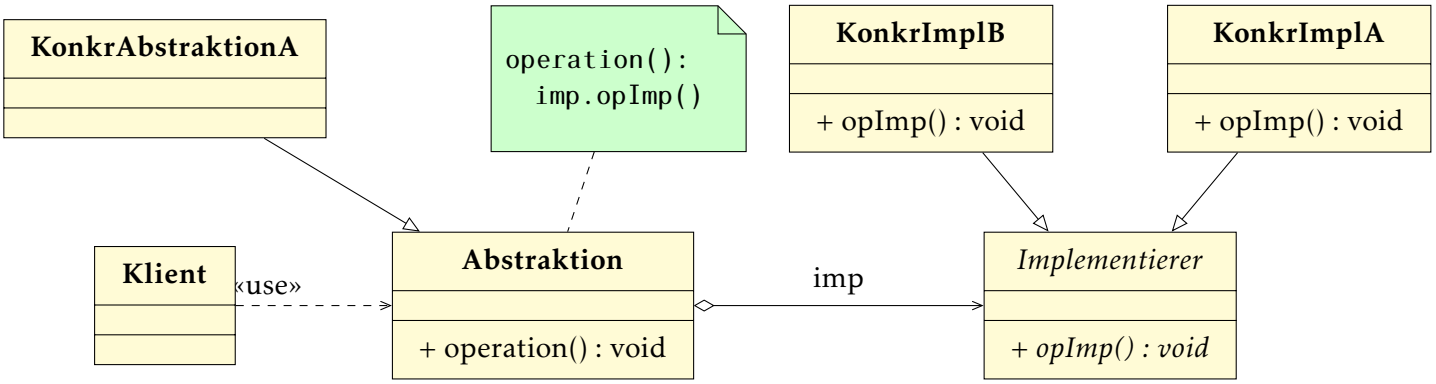


5.2.1.2 Objektbasierter Adapter



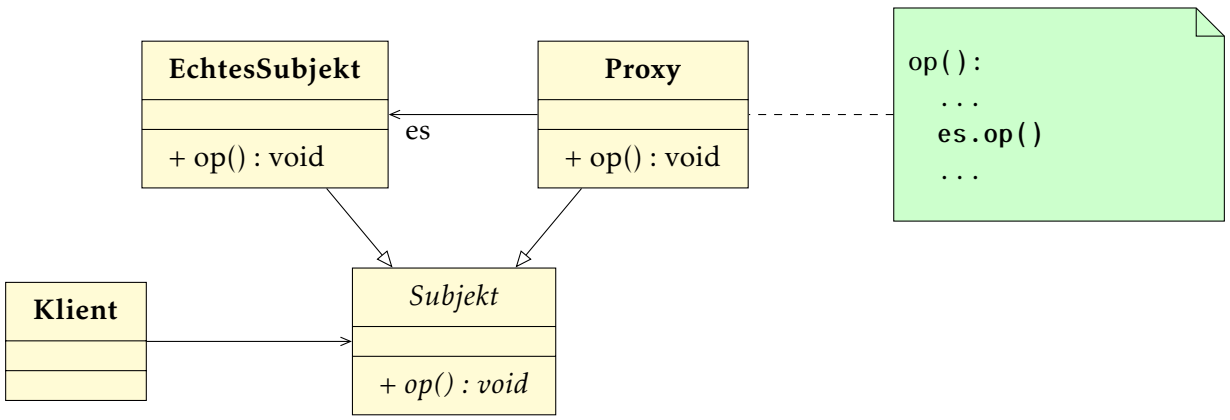
5.2.2 Brücke

Es soll die „Polymorphie aufgetrennt“ werden. Hierzu möchte man hinter einer – eine Schnittstelle anbietenden – **Abstraktion** unterschiedliche, nicht per se auf *eine Abstraktion festgelegte*, **Implementierer** verbergen, wodurch Schnittstellen und Implementierungen in **getrennten** Vererbungshierarchien organisiert.



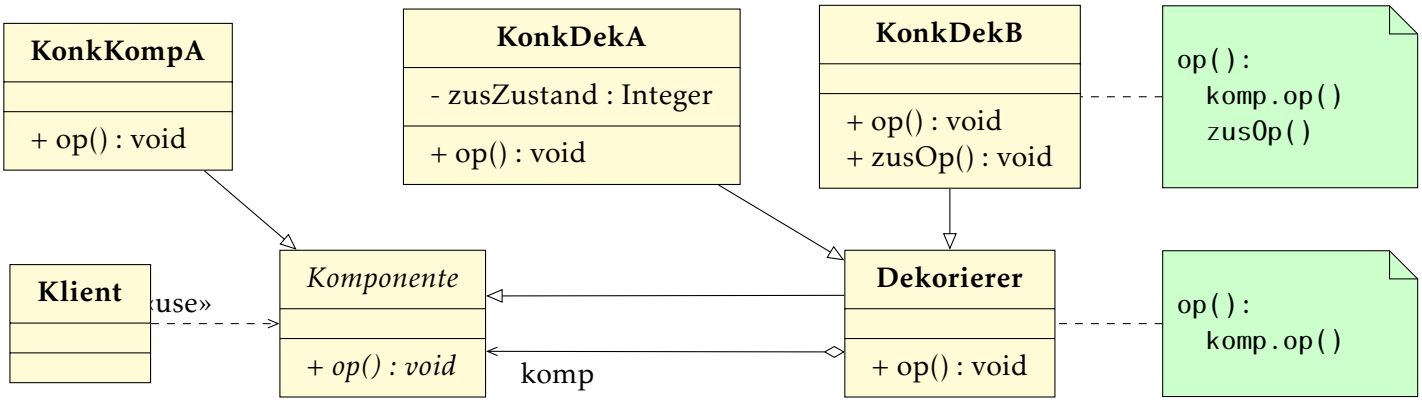
5.2.3 Proxy

Es soll der Zugriff auf ein Objekt mittels eines **vorgelagerten Stellvertreterobjekts** kontrolliert werden und dabei in Abhängigkeit vom Kontrollergebnis eventuell zusätzliche Dienste erbracht werden.



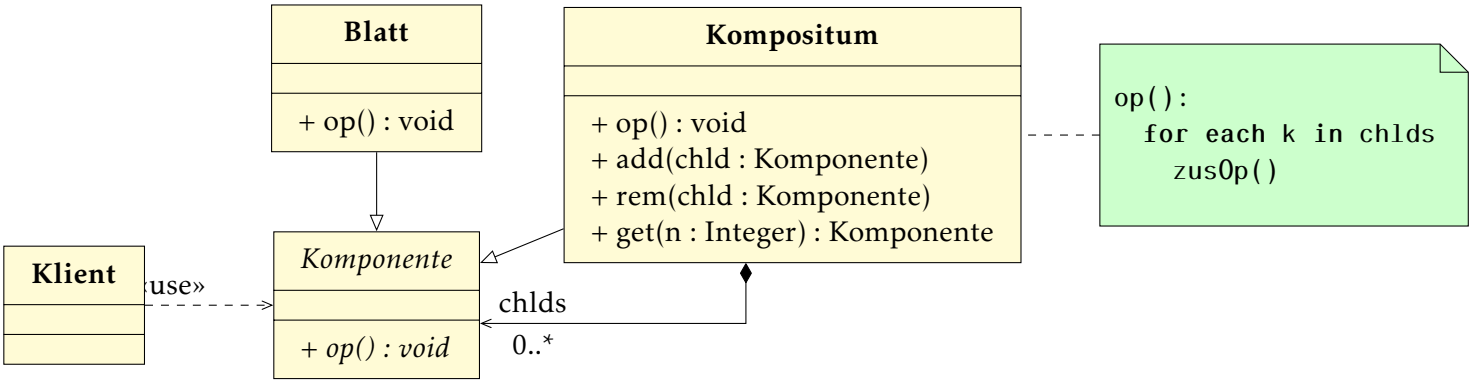
5.2.4 Dekorierer

Die Zuständigkeit eines Dienstes eines Objekts soll **zur Laufzeit *dynamisch*** erweitert werden.



5.2.5 Kompositum

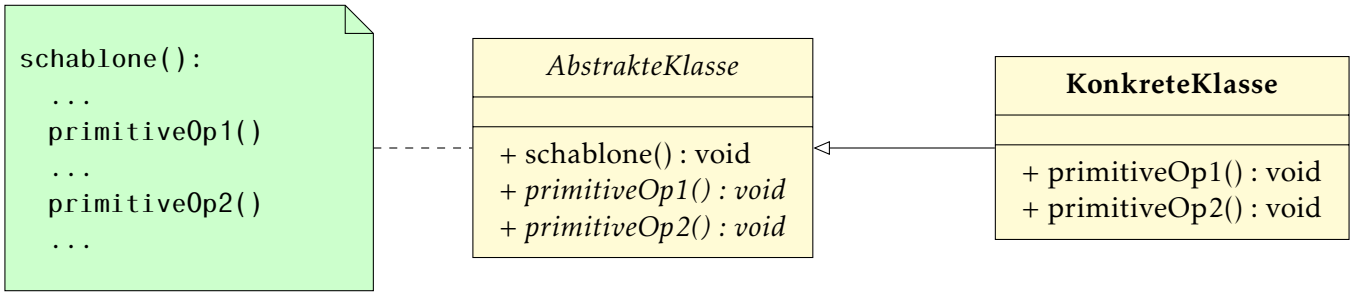
Objekte sollen zu einer **Baumstruktur** zusammengefügt werden, um eine **Teil-Ganzes-Hierarchie** zu repräsentieren. Damit sollen *sowohl einzelne Objekte* als auch *Kompositionen von Objekten* eine **einheitliche Behandlungsweise** zugestanden werden.



5.3 Verhaltensmuster

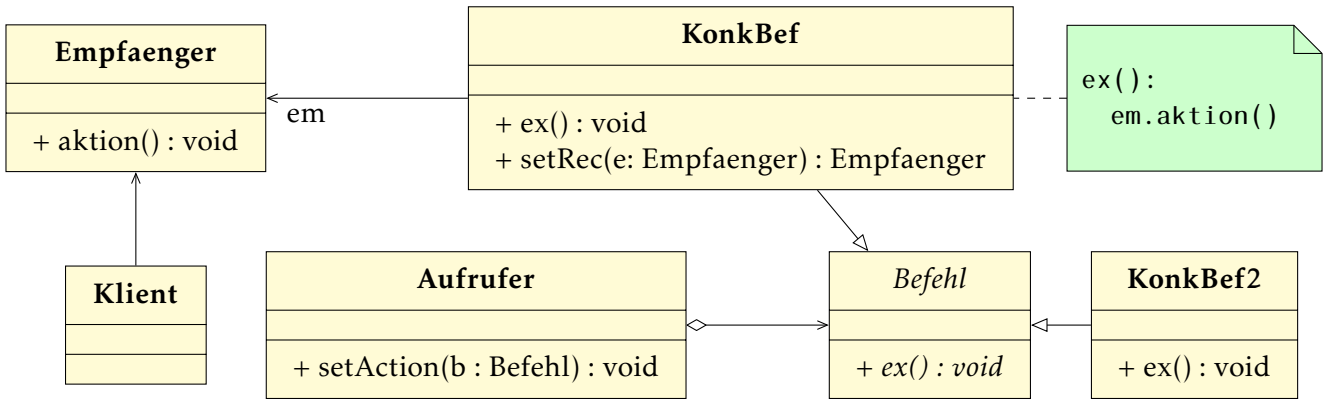
5.3.1 Schablonenmethode

Hier wird ein Algorithmus **schablonenhaft** beschrieben, so dass bezüglich einigen Implementierungsdetails **Freiheitsgrade** bestehen.



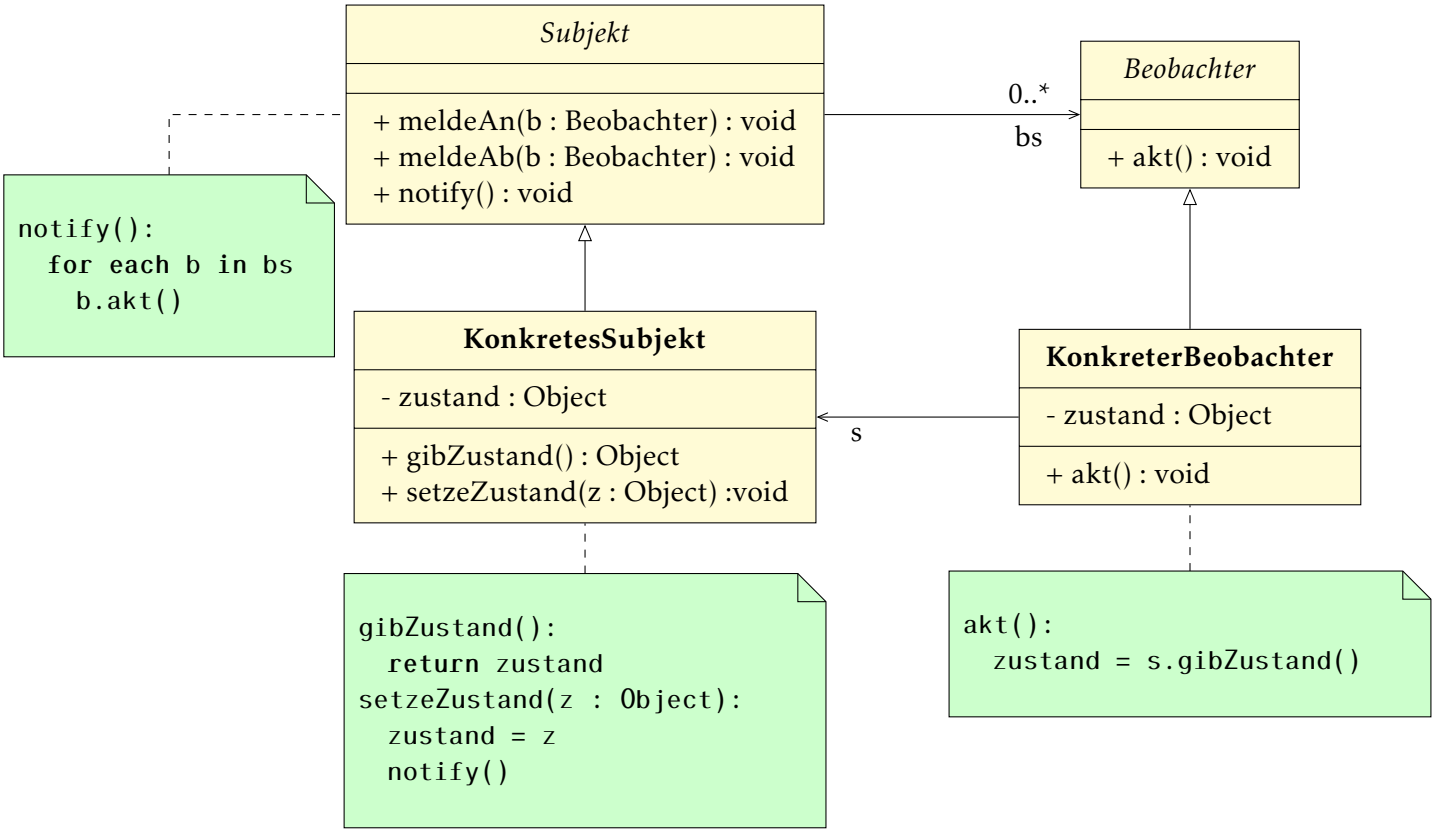
5.3.2 Befehl

Die Information, welche Operation auszuführen ist, soll als Parameter übergeben werden können. **Jeder** Befehl soll vor dessen Ausführung erfasst und verwaltet werden können.



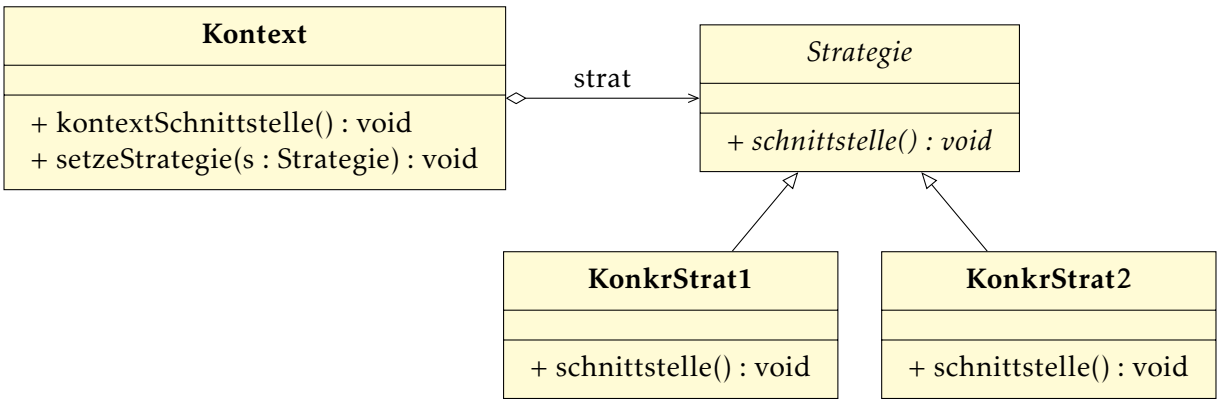
5.3.3 Beobachter

Konsistente Änderungsübernahme – **ohne Vereinheitlichung** – von einem Objekt bei mehreren *„Beobachtern"*. Dazu gibt es einen **Benachrichtiger**, welcher die vorher angemeldeten *Beobachterobjekte* bei **Änderungen benachrichtigt**.



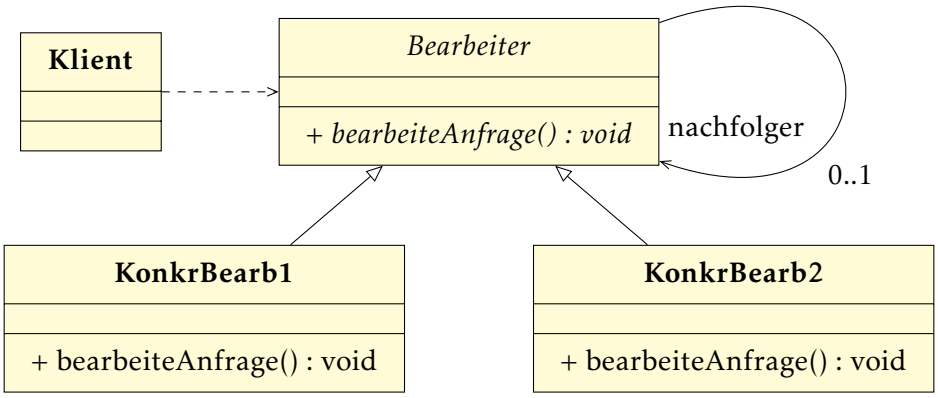
5.3.4 Strategie

Das grundsätzliche Problem ist auf verschiedene Weisen lösbar, wobei die tatsächlich verwendete Variante **zur Laufzeit** ausgewählt werden soll. Der **nichtvariable Teil** soll allerdings nur einmal *zur Redundanzvermeidung* gespeichert werden.



5.3.5 Zuständigkeitskette

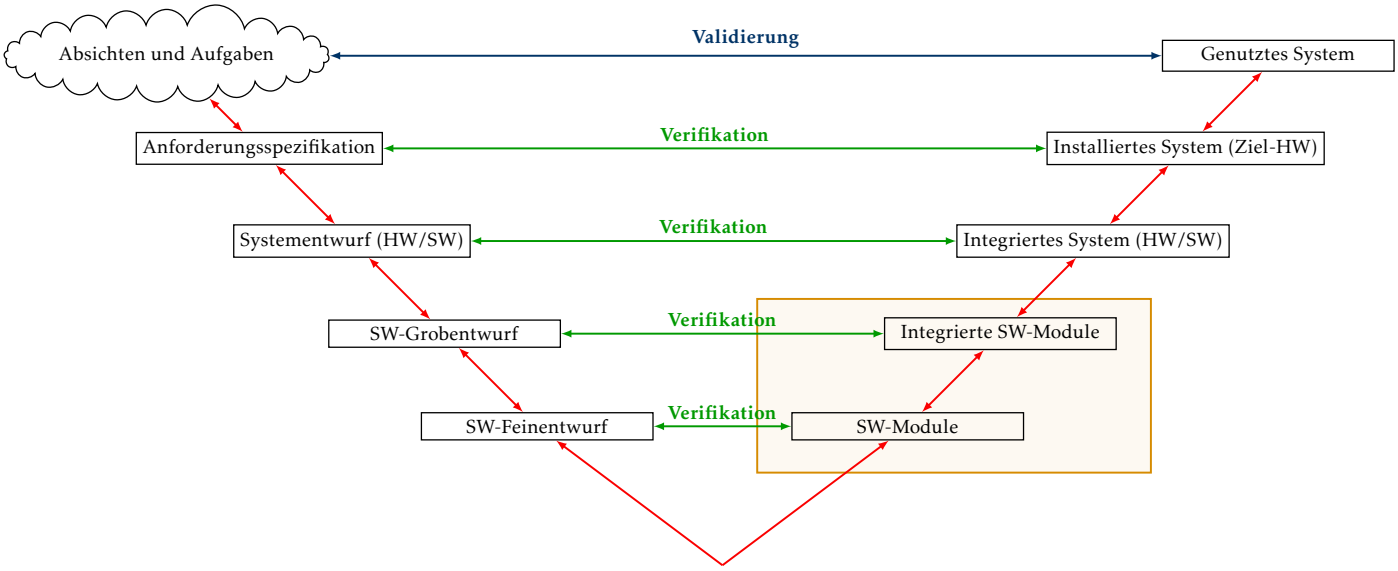
Zur Laufzeit ist dem Klienten die Bearbeiter der von ihm gestellten Aufgabe bekannt, wobei zur Bearbeitung spezieller Anfragen unterschiedliche Dienstgeber in Frage kommen. Die Menge der für einen **speziellen** Dienst möglichen Bearbeiter sei zudem **dynamisch festlegbar**.



KAPITEL 6

IMPLEMENTIERUNG

Wo befinden wir uns gerade?



6.1 Überblick über Programmiersprachen

In Programmiersprachen können besonders *fehleranfällige Konstrukte* enthalten sein, um dem entgegenzuwirken gibt es „sichere Teilmengen“ (en. *safe subsets*) der Sprache. Mit ihnen werden Regeln definiert, die den ursprünglichen Wortschatz zwar einschränken, aber dafür den fehleranfälligen Konstrukten entgegenwirkt.

6.2 Kodierungsregeln

In der Mitte stehen die zentralen Thesen der **Verständlichkeit** und **Lesbarkeit**, die allesamt für **jeden Beteiligten wichtiger** sind als ein geringer Schreibaufwand oder Betriebsmittelbedarf.
Ebenso wichtig ist ein **problemorientierten Aufbau**, sowie eine **Stileinheit der Programmierung, Dokumentation und des Layouts in jedem Großprojekt**.
Man **darf keine Tricks verwenden**! **Bezeichner** sollen Objekte **eindeutig** identifizieren, die Bedeutung *leicht* erkennen lassen und **konsistent** geführt werden.
Getrennte **Deklaration und Initialisierung** von Variablen sollte nach Möglichkeit **vermieden werden**. Ebenso sollten **Konstanten** statt *„hard-coded-values“* verwendet werden.

6.3 Kodegenerierung aus UML-Konstrukten

Man setzt es sich zum Ziel aus den UML-Diagrammen, die zu Ende der Objektorientierten Analyse und dem Objektorientiertem Design vorliegen, **automatisch** Quellkode zu generieren.

Wir unterscheiden drei Arten des *„engineering“*:

Forward Engineering Fertiges Softwaresystem ist **Ergebnis** des Entwicklungsprozesses, dazu wird der Programmkode aus den Entwurfsmodellen erzeugt.

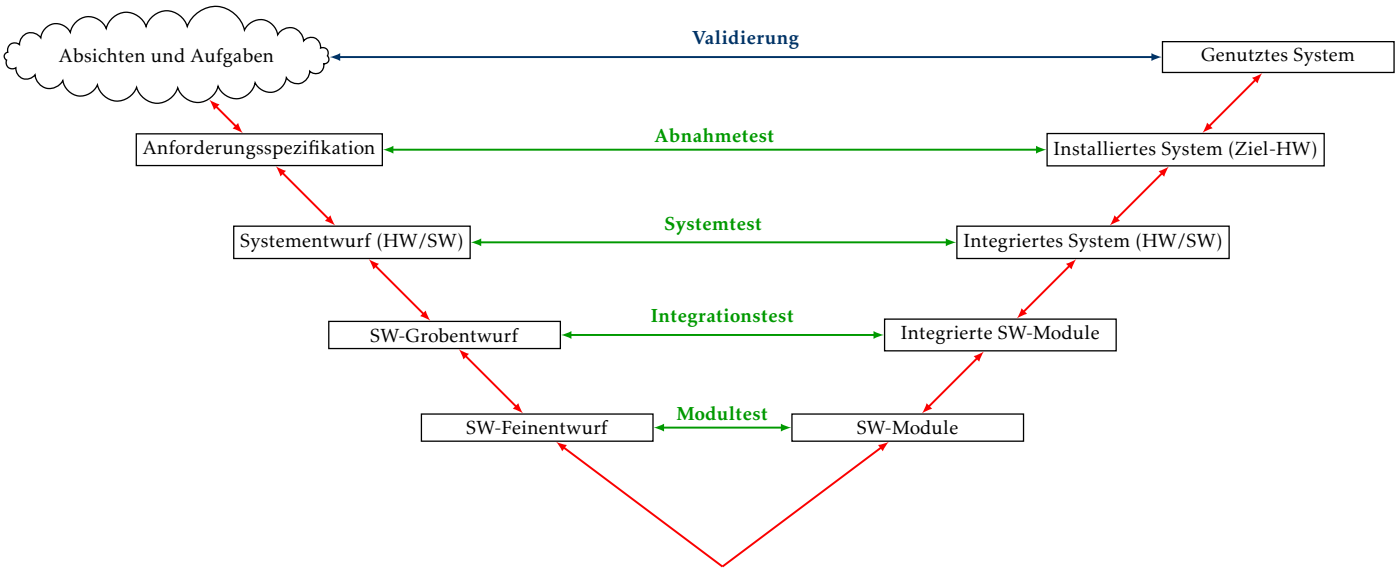
Reverse Engineering Das **vorhandene** Softwaresystem ist der Ausgangspunkt der Analyse, dazu wird das Entwurfsmodell aus dem Programmkode generiert.

Round-Trip Engineering bezeichnet eine Kombination aus **Forward** und **Reverse Engineering**.

KAPITEL 7

TESTEN

Wo befinden wir uns gerade? Wir befinden uns jetzt in der Verifikationphase:



7.1 Einführung

Wir unterscheiden zwischen folgenden Testarten in unserem V-Modell:

Modultest zwischen den Phasen „SW-Feinentwurf“ und „SW-Module“. Hierbei findet eine Prüfung vom Modulen gegen die im Feinentwurf festgelegten Funktionen und Schnittstellen. Diese kann dabei aufgrund der überschaubaren Größe der Module **gründlicher, feiner und intensiver** gestaltet werden als der **Integrationstest**. Im Allgemeinen erfordert dies dann aber auch *Zusatzelemente*, die die *Interaktion* des zu testenden Moduls *simulieren*, was mit Stümpfen (en. *stubs*) von aufgerufenen Modulen und Treiber (en. *drivers*) der aufrufenden Module geschieht.

Integrationstest zwischen den Phasen „SW-Grobentwurf“ und „Integrierte SW-Module“. Hierbei findet die Prüfung des Modulzusammenspiels statt. Die Integration der zu testenden Module findet dabei meist **inkrementell statt**, um eine frühzeitige Erkennung von Fehlern an Schnittstellen zu erleichtern.

Systemtest zwischen den Phasen „Systementwurf“ und „Integriertes System“. Hierbei findet eine Prüfung des auf Zielhardware laufenden Softwaresystems auf Konsistenz zwischen **tatsächlicher** und im Systementwurf *verlangter* Leistung. Dies umfasst einen **Vollständigkeits-**, **Leistungs-**, **Volumen-** sowie **Stresstest**.

Abnahmetest zwischen den Phasen „Anforderungsspezifikation“ und „Installiertes System“. Hierbei findet eine Prüfung des beim Kunden **installierten Systems** auf die **Erfüllung der spzifizierten Anforderungen** statt.

7.2 Systematische Vorgehensweise

Beim Testen wollen wir wie folgt vorgehen:

Schritt 1: Testplanung — Hierbei wird das **Testziel** sowie das **Testendekriterium** definiert.

Schritt 2: Testerstellung — Hierbei werden die **Eingabedaten** für das zu testende Programm **ausgewählt**.

Schritt 3: Testdurchführung — Hierbei wird das Programm mit den eben ausgewählten Eingabedaten **ausgeführt**.

Schritt 4: Testauswertung — Die Ausgabedaten werden mit den aufgrund der Spezifikation zu **erwartenden** Ergebnissen **verglichen**.

Wir können dazu unterschiedlich vorgehen, so unterscheiden wir auch zwischen drei Teststrategien:

Funktionales Testen (en. *black-box-testing*) beschreibt eine Prüfung auf die Realisierung **aller** spezifizierten Funktionalitäten, die Testauswahl findet nur aufgrund der Spezifikation, **nicht jedoch aufgrund der Programmstruktur**, statt. Die Testendekriterien werden anhand des Anteils der Anforderungen, deren korrekte Realisierung durch Test überprüft wurden, erstellt.

Modellbasiertes Testen (en. *grey-box-testing*) beschreibt eine Prüfung der Umsetzung des im Entwurf festgehaltenen Verhaltens, die Testauswahl findet hierbei nach dem Entwurf statt, beispielsweise anhand von UML-Diagrammen aus der Phase der objektorientierten Analyse. Testendekriterien erstellt man hierbei anhand der erzielten Überdeckung der Diagramme.

Strukturelles Testen (en. *white-box-testing*) beschreibt eine Prüfung auf die ausschließliche Realisierung **spezifizierter Funktionalität**, wobei die Testauswahl aufgrund der **Kontroll- und Datenflüsse des Kodes** stattfindet und damit die **Programmstruktur besonders relevant** ist. Testendekriterien werden hierbei anhand der erzielten Kodeüberdeckung ausgewählt.

7.2.1 Funktionale Testarten

Äquivalenzklassentest Man testet hierbei Eingabebereiche, welche auf **äquivalente** Ergebnisse führen sollen.

Grenzwerttest Man testet hierbei Eingaben an den Grenzen der *Äquivalenzklassen*.

Ursachewirkungsüberlegungen (en. *cause-effect-graphing*) Man erstellt einen Ursachewirkungsgraphen (en. *cause-effect-graph*) aufgrund der **Ursachewirkungsüberlegungen**.

Fehlerraten (en. *error guessing*) Man führt einen spezifikationsbezogenen **Fehlererwartungstest** aus Erfahrung durch.

Test mit Zufallswerten Die Testdaten werden hierbei nach statistischen Verteilungen gewählt.

7.2.2 Strukturelle Testarten

7.2.2.1 Überdeckungstesten

Anweisungsüberdeckung — Hierbei muss **jede Anweisung** einmal überdeckt werden.

Verzweigungsüberdeckung — Hierbei müssen **alle Verzweigungen** mindestens einmal überdeckt werden.

Pfadüberdeckung — Hierbei müssen **alle Pfade** mindestens einmal überdeckt werden.

Mehrfachbedingungstest — Bei zusammengesetzten Verzweigungsbedingungen werden die Kombinationen ihrer atomaren Bestandteile mindestens einmal getestet.

Grenzwerttest — Die Grenzen der durch die Verzweigungsbedingungen definierten Domänen werden hierbei mindestens einmal getestet.

7.2.2.2 Datenflussbasiertes Testen

Zusätzlich zu Kontrollflusseigenschaften werden die Positionen und Zeiten einer neuen Variablenbelegung sowie die Positionen einer Verwendung der neuen Werte berücksichtigt.

KAPITEL 8

Integrationstest

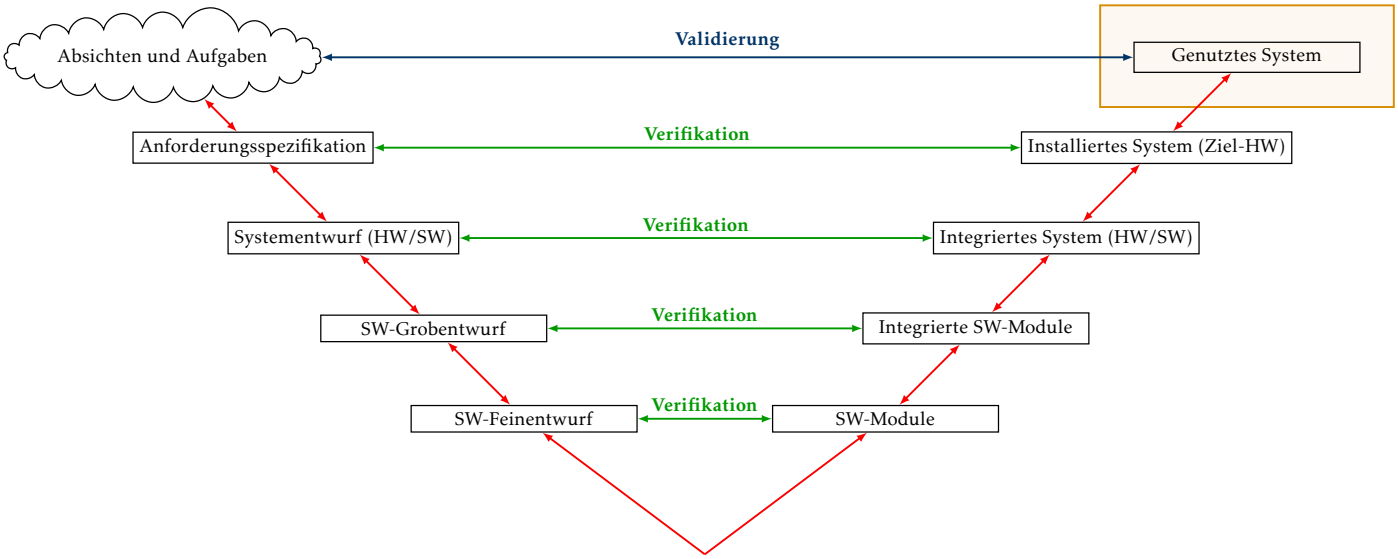
Wir unterscheiden Implementierung und Integration:

Herangehensweise	Stärken	Schwächen
Implementierung mit anschließender Integration	—	− Keine Fehlerisolation − Große Fehler im Design treten erst relativ spät auf
Top-Down Implementierung und Integration	+ Fehlerisolation + Große Fehler im Design treten bereits früh auf	− Wiederverwendbare Module werden adäquat getestet
Bottom-Up Implementierung und Integration	+ Fehlerisolation + Wiederverwendbare Module werden adäquat getestet	− Große Fehler im Design treten erst relativ spät auf
Sandwich Implementierung und Integration	+ Fehlerisolation + Große Fehler im Design treten bereits früh auf + Wiederverwendbare Module werden adäquat getestet	—

KAPITEL 9

WARTUNG

Wo befinden wir uns gerade?



9.1 Einführung

Definition 15 (*Wartung*)

In der **Wartung** beschätigen wir uns mit der *Lokalisierung und Behebung von Fehlerursachen* bei *in Betrieb befindlichen Softwareprodukten*, wenn die **Fehlerwirkung** bekannt ist sowie mit der *Lokalisierung und Durchführung von Änder- und Erweiterungen* von *in Betrieb befindlichen Softwareprodukten*, wenn die **Art der gewünschten Änderungen/Erweiterungen** festliegt.

9.2 Wartungsaufgaben

Korrektur (en. *corrective maintenance*) mit der Entfernung aktueller Fehler

Anpassung an neue Benutzeranforderungen einschließlich vorbeugenden Maßnahmen (en. *perfective and preventive maintenance*)

Anpassung an neue Umgebungen (en. *adaptive maintenance*)

Dabei gilt selbstverständlich, dass die Wartung keine **einmalige Tätigkeit** ist, sonder über den **gesamten Lebenszyklus eingeplant** werden muss. Mit der möglichst früh stattfindenden Änderungsfesthaltung, einer **vollständigen Dokumentation**, einem Softwaresystem mit *hoher Kohäsion* und *loser Kopplung*, mit **Kapselung** (en. *information hiding*), der **Entkopplung von Grob- und Feinentwurf** und **Schichtenarchitekturen**, sowie der Verwendung von **Entwurfsmustern**

und die Wiederverwendung generierter Testfälle durch **Regressionstest**strategien, lässt sich der Wartungs*aufwand* **reduzieren**.

9.3 Refactoring

Definition 16 (*Refaktorisierung*)

> Eine Änderung an der **inneren Struktur** einer Software, um sie **leichter verständlich** zu machen unnd **einfacher zu verändern**, **ohne ihr beobachtbares Verhalten zu ändern** heiße **Refaktorisierung**.

Methode extrahieren — Kode der öfters vorkommt wird in eine eigene Methode verschoben

Methode integrieren — „Einfache“ Methoden ohne wirklichen Verständnisgewinn werden zum Aufrufer hin aufgelöst.

Methode in ein Objekt umwandeln — **Methode extrahieren** lohnt sich nicht aufgrund vieler lokaler Variablen, dann löst man eine Methode auf, indem man sie durch ein Objekt ersetzt. Lokale Variablen werden dabei zu Feldern.

Klasse extrahieren — Eine Klasse wird aufgeteilt, weil sie mehrere Zuständigkeiten besitzt. Dazu wird eine neue Klasse erstellt.

Klasse integrieren — Gegenstück der „Denormalisierung“: Eine Klasse wird, weil sie keine eigenständigen Verantwortlichkeiten mehr besitzt, in eine andere Klasse aufgelöst.

Delegation verbergen — Dem Dienstleisterobjekt sei eine Methode hinzuzufügen, mit dem auf Objekte, die vorher extra geholt wurden um Methoden auf ihnen aufzurufen, zugegriffen wird.

Collection kapseln — Kapselung einer Collection in einer Klasse: Dazu wird eine Klasse so abgeändert, dass sie nur eine lesende Sicht der Collection übergibt, alle Änderungen muss über Methoden der Klasse geschehen.

Bedingten Ausdruck in Polymorphie überführen — Eine Methode definiert in Abhändigkeit vom Typ eines Objekts unterschiedliches Verhalten, so verschiebt man jeden Zweig der Entscheidung in eine überschreibende Methode einer Unterklasse, die Orginalmethode wird dann abstrakt.

Feld „nach oben“ verschieben — Mehrere Unterklassen besitzen dasselbe Feld, so wird dies in die Oberklasse verschoben.

Methdoe „nach oben“ verschieben — Mehrere Unterklassen besitzen dieselbe Methode mit **identischem Verhalten**, so wird diese in die Oberklasse verschoben.

Unterklasse extrahieren — Eine Klasse weist Attribute oder Methoden auf, die nicht **für alle Instanzen der Klasse** von Bedeutung sind, so wird eine neue Unterklasse für die Instanzen, die diese Attribute oder Methoden benötigen, erstellt und die betroffenen Attribute oder Methoden in die Unterklasse verschoben.

Oberklasse extrahieren — Mehrere Klassen haben gemeinsame Attribute oder Methoden, so werden diese in eine **gemeinsame zu erstellende Oberklasse** verschoben.

Vererbungsstruktur entzerren — Eine mehrstufige Vererbungshierarchie, mit redundantem Kode durch Kombination mehrerer Merkmale, wird in mehrere **getrennte Vererbungshierarchien** mit Assoziationen aufgeteilt.

Indirektionen Wir definieren:

Definition 17 (*Indirektion*)

> Eine **Indirektion** liegt vor, wenn eine Funktionalität eines Dienstleisters nicht direkt von einem Dienstnutzer aufgerufen wird, sondern wenn ein **Vermittler** den Aufruf weiterleitet. Insbesondere liegt damit eine **Indirektion** auch bei Vererbung vor

Indirektionen können durch Refaktorisierung entstehen, sind auch in gewissen Maßen gut, als dass …

… redundanter Kode im noch unzerlegten Objekt durch die Zerlegung eliminiert wird.

… Redundanzen beseitigt und Übersichtlichkeit und Flexibilität erhöht wird.

Aber durch Indirektionen wird die Software auch unübersichtlicher. Deshalb ist es in Maßen einzusetzen.

Anhang A

Merkblätter

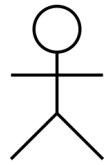
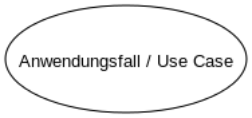
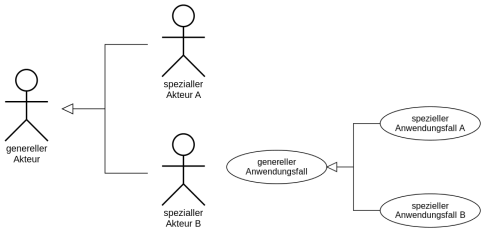
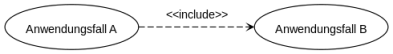
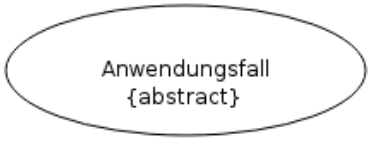
Hier sollen verschiedene Merkblätter von anderen Veranstaltungen folgen, dessen Stoff in dieser Zusammenfassung nicht näher behandelt wurde. Es finden sich folgende Merkblätter:

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Anwendungsfalldiagramm (Use-Case-Diagramm)

Das Ziel eines jeden Anwendungsfalldiagramms ist die funktionale Beschreibung eines Systems. Dabei beschreiben die Use Cases die **Benutzeranforderungen**.

Modellierungselemente	Bedeutung
	Akteure: • Akteure benutzen Funktionen • Akteure sind immer außerhalb jeglicher Systeme • Akteure werden **nicht** implementiert (sie stellen den Benutzer dar)
	Konkreter Anwendungsfall (Funktionen), stellt eine Aktivität oder Aktion dar.
	Systemkontext: Alles innerhalb dieses Rechtecks gehört zu einem System.
	Generalisierung: Im Prinzip ähnlich der Vererbung bei Klassen, so werden bei Akteuren alle Eigenschaften weiter vererbt und alle möglichen Anwendungsfälle. Bei Anwendungsfällen ist dies eine IS_A-Beziehung.
	Beziehungen zwischen Anwendungsfällen: «include» (auch «benutzt»): Ein Anwendungsfall wird in mehreren anderen Anwendungsfällen wiederverwendet. Wichtig: Diese Beziehung ist ein **MUSS**! (**Funktionale Dekomposition**)
	Beziehungen zwischen Anwendungsfällen: «extends» (auch «erweitert»): Beschreibt eine Variation des normalen Verhaltens. Nutzen: Beschreiben von Ausnahmefällen. Bei notwendigen Bedingungen für Ausnahmen steht dabei ein Extensionpoint zur Verfügung.
	A**bstrakter Anwendungsfall**. Ähnlich der Klassendefinition kann man diesen Anwendungsfall nicht ausführen. Er dient der Definition der Wiederverwendbarkeit eines gemeinsamen Verhaltens.

Merkblatt zur Veranstaltung

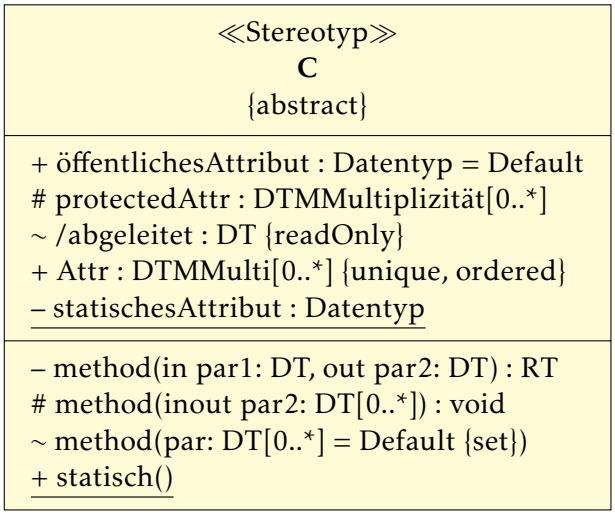
Konzeptionelle Modellierung

Merkblatt: „Modellierung mit UML – Augenmerk: Strukturdiagramme“

Um die Struktur eines Systems in ihren verschiedenen Aspekten zu beschreiben, bietet UML eine Reihe von Diagrammarten an, welche in den folgenden Abschnitten definiert werden.

Definition (Klassendiagramm)
Klassendiagramme entstammen der konzeptionellen Datenmodellierung und entsprechen somit den Entitätstypen im E/R-Modell. Der einzige **Unterschied**: „Objekte“ stellen im Gegensatz zu den Entitäten nun **Software-Artefakte** dar. Wir verwenden es deswegen für die Modellierung statischer Strukturen eines Systems.

Basisnotation für Klassen



Namensfelder
Innerhalb des Namensfelds findet sich der Klassenname. Zudem können Stereotype (Schlüsselwörter) angegeben werden, welche die Klasse in eine Kategorie einteilen. Eigenschaftsangaben sind ebenfalls erlaubt. Hier wird gerade eine abstrakte Klasse definiert ({abstract}).

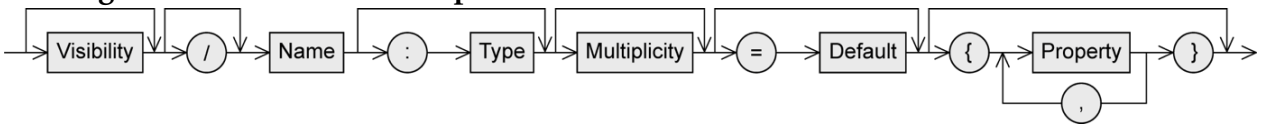
Attribute
Innerhalb der zweiten Kategorie finden sich die Attribute wieder. Allgemein gilt auch hier: Attribute beschreiben Eigenschaften. Während Instanzattribute nur für ein Objekt spezifisch sind, sind Klassenattribute (unterstrichen darzustellen) für eine ganze Klasse spezifisch.

Abgeleitete Attribute
Abgeleitete Attribute sind Attribute, deren Wert sich durch andere Attribute bestimmen lässt. Man kennzeichnet sie durch einen vorangestellten Schrägstrich **/**.

Mehrwertige Attribute
Mittels der Multiplizitätsangabe lassen sich auch mehrwertige Attribute modellieren. So beschreibt jene die Anzahl der Werte, die ein Attribut annehmen kann.

Eigenschaften bei Attributen
Bei Attributen kann ein Datentyp sowie ein Initialwert angegeben werden. Zusätzliche Eigenschaften ergeben sich über die Eigenschaftsliste zu notieren in geschweiften Klammern. Achtung: Die Eigenschaft **unique** unterscheidet sich von der Eigenschaft `UNIQUE` im Relationenmodell! In UML heißt unique, dass ein Attributwert in **einer** Instanz in **einem mehrwertigen** Attribut nur einmal vorkommen darf!

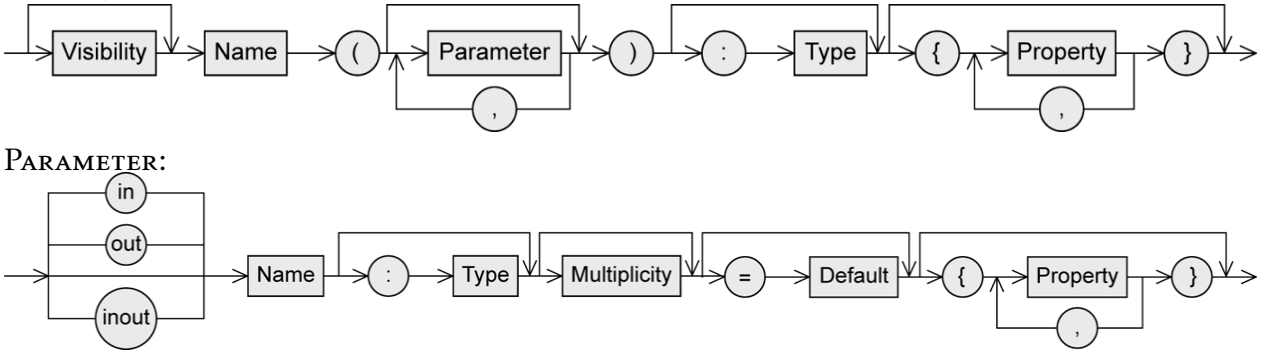
Syntaxdiagramm für die Attributspezifikation



Operationen

Neben Attributen haben die meisten Klassen noch Operationen (↯ Unterschied zum EE/R-Modell, denn dort gab es **keine** Operationen). Neben dem Namen und der Sichtbarkeit gibt es hier im Gegensatz zu der Attributspzifikation noch die Möglichkeit eine Parameterliste anzugeben. Jeder Parameter ist ähnlich wie ein Attribut aufgebaut (von der Spezifikation her zumindest), statt der Sichtbarkeit lässt sich hier aber eine Richtung angeben (`in`, `out`, `inout`).

Syntaxdiagramm für die Operationenspezifikation



Sichtbarkeiten

Notation	Name	Bedeutung
+	`public`	Der Zugriff ist durch Objekte beliebiger Klassen erlaubt.
–	`private`	Der Zugriff ist nur innerhalb des Objekts selbst erlaubt.
#	`protected`	Der Zugriff ist nur durch Objekte derselben Klasse und deren Subklassen erlaubt.
~	`package`	Der Zugriff ist nur durch Objekte, deren Klassen sich im selben Paket befinden, erlaubt.

Aktive Klassen

Wir wollen grundsätzlich zwischen passiven und aktiven Klassen unterscheiden. Als aktiv bezeichnen wir eine Klasse, für welche ein eigenes Verhalten (bspw. mittels Zustands- oder Aktivitätsdiagrammen) definiert ist. Deren aktive Instanzen sind in ihrem eigenen Operationsfaden resp. können den Kontrollfluss starten oder modifizieren. Aktive Objekte sind somit sequenziell und machen etwas (bspw.: Variablenmodifizieren, Verhaltensänderung etc.). Sie sind zu kennzeichnen mit der Eigenschaft {active} oder einem vertikal doppelten Rahmen.

Definition (Objekt)

object:Type

Wir wollen die Instanzen einer Klasse als **Objekt** bezeichnen. Die Ausprägungen der von der Klasse definierten Struktur (Attribute) weisen ein definiertes Verhalten auf (Operationen). Attributwerte können unter dem Namen in einem Kasten vermerkt werden. Wir notieren ein Objekt als dass der Objektname dem Klassennamen vorangestellt wird. Entweder der Objektname oder der Klassenname kann aber entfallen. Bei letzterem lässt man auch den Doppelpunkt weg. Fällt der Objektname weg, so spricht man auch von einem **anonymen Objekt**. Beides wird aber unterstrichen.

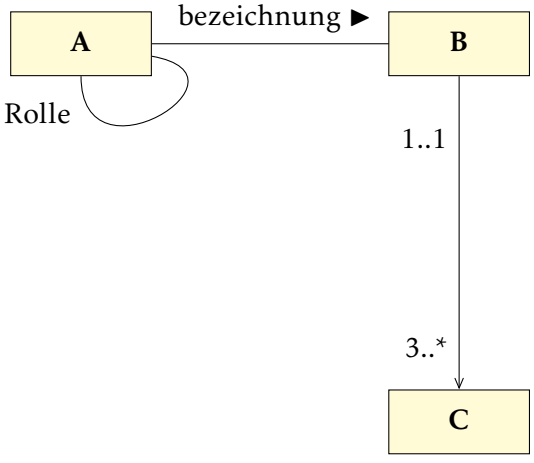
Sollten zwei **Objekte** eine Beziehung eingehen, so spricht man von einem Link. Dieser ist durch eine einfache Kante zwischen zwei Objekten darzustellen.

Definition (Assoziationen)

Als Assoziation bezeichnen wir eine Beziehung auf Klassenebene. Zu vergleichen ist dies mit der eigentlichen Definition des Beziehungstyps im E/R-Modell. Wir stellen eine solche dar als einfache Kante zwischen zweier Klassen und geben eine Leserichtung mit einer Bezeichnung an.

Definition (Navigierbare und Rekursive Assoziationen)

Für rekursive Assoziationen benötigt es Rollen, damit die einzelnen Beziehungspartner voneinander getrennt werden können. Eine Pfeilspitze zeigt grundsätzlich die Navigationsrichtung an. Salopp gesagt: Ein Objekt kennt seine Partnerobjekte und kann dadurch auf die sichtbaren Merkmale zugreifen. Ein nicht navigierbares Assotiationsende kann durch die explizite Angabe eines X spezifiziert werden. Dies bedeutet, dass dann das Partnerobjekt nicht auf die Attribute und Operationen des Objektes – inklusive der öffentlichen – zugreifen kann.



Multiplzitäten

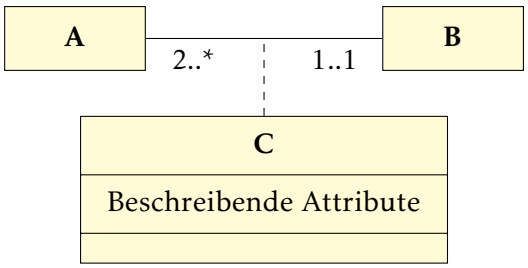
Die Multiplizitäten sind bei UML-Klassendiagrammen im Prinzip genauso zu lesen wie deren Chen-Notationsäquivalente (heißt: "gegenüberliegende"Klasse!). An sich sind sie aber der (min,max)-Notation von der Schreibweise ähnlicher. Zu notieren ist eine Multiplizität wie folgt:

```
zahl [.. max] {, zahl [.. max]}
```

Wichtig: Bei mehrwertigen Assoziationsenden kann man Eigenschaften angeben (Ordnung, Eindeutigkeit). Anders als im E/R-Diagramm gibt es die Möglichkeit, dass ein Objekt mehrere Links zum selben Partnerobjekt hat (Expizite Angabe von {nonunique} erforderlich).

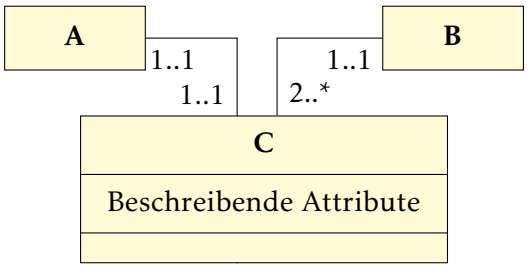
Definition (Assoziationsklasse)

Sollte eine Assoziation durch Attribute näher beschrieben werden, so erstellen wir eine Assoziationsklasse (zu sehen rechts oben). Diese kann dann, wie rechts mitte dargestellt, auch in eine normale Klasse umgewandelt werden.



Definition (N-äre Assoziationen)

Sind mehr als 2 Partnerobjekte an einer Beziehung beteiligt, so kann dies durch eine n-äre Assooziation modelliert werden. Notiert wird eine solche Assoziation durch eine ungefüllte Raute im Zentrum, welche mit allen Partnerobjekten verbunden ist. Während es hier keine Navigationsrichtungen gibt, kann es durchaus Multiplizitäten und Assoziationsklassen geben (siehe Beispiel rechts unten).



Definition (Aggregation)

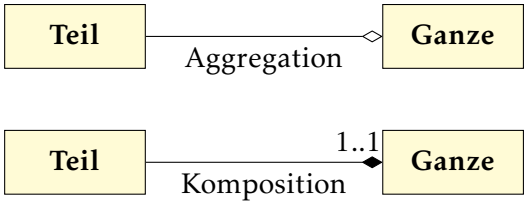
Als eine Aggregation bezeichnen wir eine spezielle Assoziation, mit der eine Teil-Ganze-Beziehung ausgedrückt werden kann. Wir unterscheiden zwischen der schwachen und starken Aggregation. Beide werden durch eine Raute am Assoziationsende des Ganzen ausgedrückt. Beide Aggregationstypen sind reflexive und a(nti)symmetrische Assoziationen.

Definition (Schwache Aggregation)

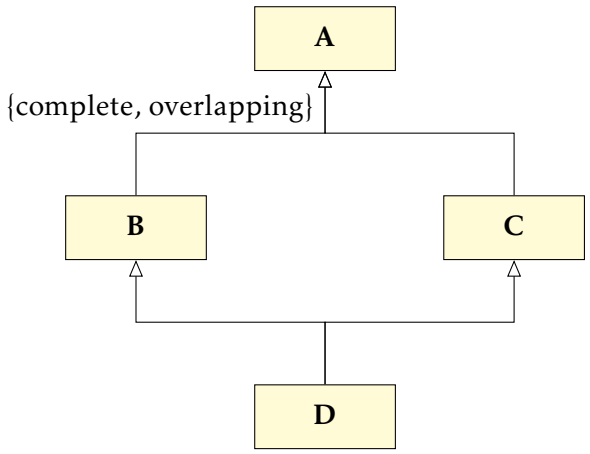
Die schwache Aggregation drückt eine schwache Zugehörigkeit der Teile zum Ganzen aus, heißt ein Teil existiert auch unabhängig vom Ganzen und kann Teil mehrerer Ganzer sein. Die Raute ist **nicht** ausgefüllt!

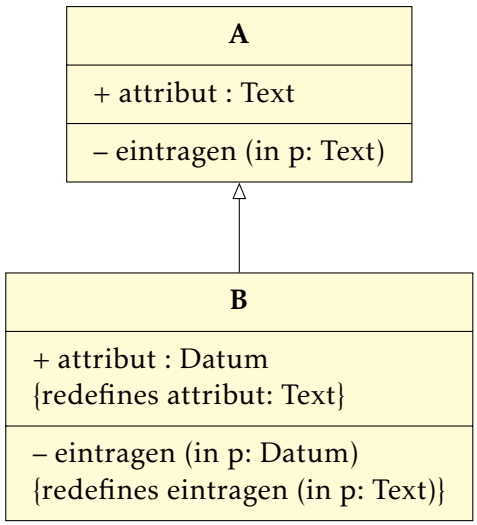
Definition (Komposition)

Im Gegensatz zur schwachen Aggregation darf bei der starken Aggregation (auch: Komposition) ein Teil in maximal einem Kompositum enthalten sein. Mögliche Multiplizitäten sind somit: 0..1 und 1..1. Bei letzterem ist das Teil vom Ganzen existenzabhängig. Bei ersterem kann das Teil zwar ohne Ganzes existieren, aber sobald es zu einem Ganzen zugeordnet ist ist es auch existenzabhängig.



Definition (Generalisierung)





Die IS_A-Beziehung kann ebenfalls in UML modelliert werden. Wir unterscheiden generell zwischen direkten und indirekten Instanzen. Als **direkte Instanz** bezeichnen wir das Objekt in Bezugnahme auf den direkten Typen, also des „dynamischen Typen“. Eine Instanz einer Klasse ist aber **indirekte Instanz** ihrer Oberklassen.
Totale/Disjukte/Überlappende Vererbung kann über Eigenschaften modelliert werden. Zur Auswahl stehen hier: `complete` und `incomplete` sowie `disjoint` und `overlapping`.
Von **Mehrfachklassifikation** sprechen wir, wenn ein Objekt direkte Instanz mehrerer Klassen ist (bspw. bei „overlapping“).
Von **Mehrfachvererbung** sprechen wir, wenn eine Klasse Unterklasse mehrerer Oberklassen ist.
Die Klassifikation kann durch einen **Klassifikationstypen** näher beschrieben und modelliert werden. Dazu schreibt man einen Diskriminator-Typ an die Generalisierungsbeziehung (Wichtig: Notation ähnlich dem anonymen Objekt).

Redefinition

Geerbte Merkmale können per se in Unterklassen redefiniert werden. Dies ist entsprechend der Abbildung links zu notieren. Geerbte Merkmale umfassen, beschränken sich aber nicht auf: Navigierbare Enden von Assoziationen, Attribute oder Operationen.

Definition (Paket-Diagramm)

Für nichttriviale Problemstellungen können die einzelnen Diagramme, wie z.B. das Klassendiagramm, schnell sehr unübersichtlich werden. Für solche Fälle bietet UML einen Strukturierungsmechanismus in Form von Paketen an.
Ein solches Paket erlaubt es, eine **beliebige Anzahl** von paketierbaren **(semantisch zusammengehörigen) UML-Modellelementen zu gruppieren**, dazu zählen u.a. *Klassen*, *Datentypen*, *Aufzähltypen und Komponenten*. **Beachte**: Pakete können selbst wieder Pakete enthalten, da Pakete paketierbare Elemente sind.
Das Paket-Diagramm dient nicht zur Spezifikation der eigentlichen Systemstruktur, sondern der Unterstützung der Modellstrukturierung. Hierbei wird v.a. auf die Gruppierung und die hierarchische Anordnung von Modellelementen Wert gelegt.

Definition (Komponenten-Diagramm)

Das Komponentendiagramm unterstützt das Paradigma der komponentenorientierten Softwareentwicklung und eignet sich somit bestens zur Beschreibung von Architekturen von Softwaresystemen.
Eine **Komponente** ist ein modularer Teil **(eigenständig ausführbare Einheit)** eines Systems, der zur Abstraktion und Kapselung einer beliebig komplexen Struktur dient, die nach außen wohldefinierte Schnittstellen zur Verfügung stellt. Eine Komponente kann hierin im gesamten

Softwareentwicklungszyklus modelliert und zunehmend verfeinert werden, bis eine **Software-Installation** vollständig abgeschlossen ist. Somit kann eine Komponente z.B. die Modellierung der Funktionalität eines Systems mittels Anwendungsfällen (logische Modellierungsaspekte) bis hin zu derer Realisierung durch physische Artefakte umfassen.
Das Komponentendiagramm zeigt also die Definition von Komponenten und Abhängigkeiten zwischen diesen.

Merkblatt zur Veranstaltung

Konzeptionelle Modellierung

Merkblatt: „Modellierung mit UML – Augenmerk: Verhaltensdiagramme“

Interaktionsdiagramme – Sequenzdiagramme

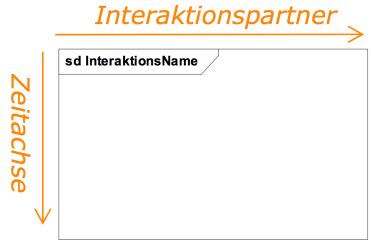
Definition (Interaktionsdiagramm)

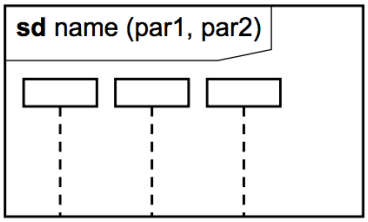
Wir definieren ein Interaktionsdiagramm als das Diagramm, welches das **Interobjektverhalten** in Form von Nachrichten zwischen Objekten in bestimmten Rollen spezifizieren.

Definition (Interaktion)

Wir bezeichnen die Art und Weise, wie **Nachrichten** und Daten zwischen verschiedenen Interaktionspartnern in einem bestimmten Kontext ausgetauscht werden, um eine bestimmte Aufgabe zu erfüllen als **Interaktion**. Der Nachrichtenaustausch wird im Allgemeinen auf Typebene modelliert. Interaktionen erfolgen durch Nachrichten in Form von Signalen und Operationsaufrufen oder wird durch Bedingungen und Zeitereignisse gesteuert.
In anderen Worten: Interaktion ≡ Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?

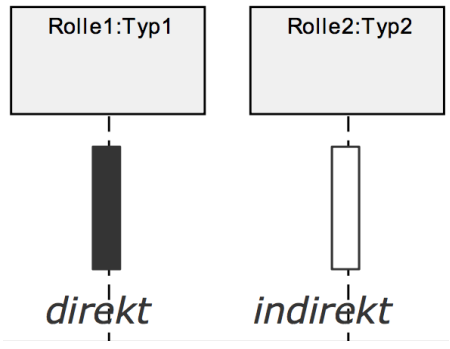
Das Sequenzdiagramm als solches hat viele verschiedene Einsatzmöglichkeiten, so soll es die Interaktionen eines Systems mit seiner Umwelt, die Realisierung eines Anwendungsfalls, das Zusammenspiel der inneren Struktur einer Klasse, Komponente oder Kollaboration, die Spezifikation von Schnittstellen zwischen Systemteilen oder die Operationen einer Klasse modellieren.
Wir unterscheiden im Sequenzdiagramm zwischen der horizontalen Interaktionspartnerachse und der vertikalen Zeitachse. Die Notation sieht wie folgt aus:





Definition (Interaktion, Trace und Ausführungsspezifikation im Sequenzdiagramm)

Als **Interaktion** bezeichnen wir eine Abfolge so genannter Ereignisspezifikationen. Diese partitionieren eine Lebenslinie in Zeitsegmente und deren Reihenfolge ist durch die Lebenslinie festgelegt. Sie definieren einen potenziellen Ereigniseintritt zur Laufzeit.
Als Folge von konkreten Ereigniseintritten bezeichen wir einen **Trace**. Er gibt den Ablauf einer Interaktion zur Laufzeit wieder
Die Periode, in der ein Interaktionspartner direkt oder indirekt ein bestimmtes Verhalten ausführt, nennt man Ausführungsspezifikation. Notation:



Definition (Nachrichtentypen)

Wir wollen zwischen synchronen, asynchronen, verlorenen sowie gefundenen Nachrichten unterscheiden.

Definition (Synchrone Nachricht)

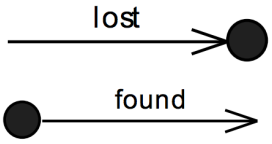
Bei einer **synchronen Nachricht** (ausgefüllte Pfeilspitze) wartet der Sender, bis die durch die Nachricht ausgelöste Interaktion beendet ist. Meist geschieht dies über eine **Antwortnachricht** (gestrichelter Pfeil).

Definition (Asynchrone Nachricht)

Bei einer **asynchronen Nachricht** (leere Pfeilspitze) wartet der Sender eben nicht auf die Antwort des Empfängers.

Definition (Verlorene und Gefundene Nachricht)

Von verlorenen Nachrichten spricht man dann, wenn man die Nachricht an einen unbekannten, außenstehenden oder nicht relevanten Interaktionspartner verschicken möchte (siehe Pfeil «lost»). Außenstehend meint hier außerhalb des zu modellierenden Systems. Von einer gefundenen Nachricht spricht man, wenn man eine Nachricht von einem solchen Interaktionspartner empfängt.

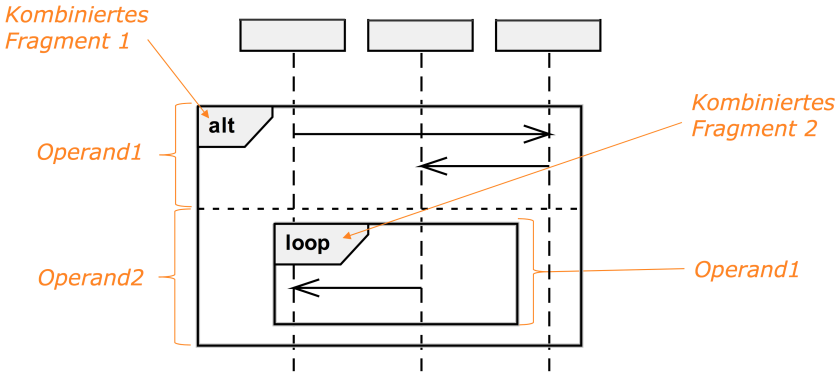


Definition (Zustandsinvariante)

Als **Zustandsinvariante** bezeichnen wir eine Zusicherung, dass eine bestimmte Bedingung zu einem bestimmten Zeitpunkt erfüllt ist. Dabei bezieht sich jene immer auf eine ganz bestimmte Lebenslinie und wird vor Eintritt des darauffolgenden Ereignisses ausgewertet.
Wichtig: Falls die Invariante nicht erfüllt ist kommt es zu einem Fehler!
Notation:

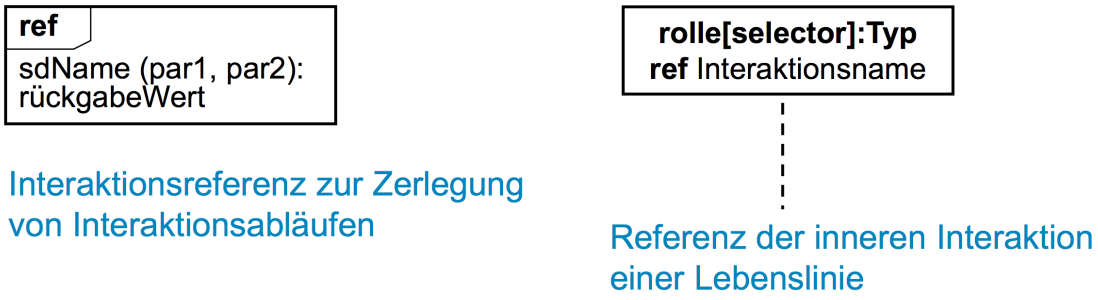
Definition (Kombinierte Fragmente)

Ein kombiniertes Fragment wird wie ein Sequenzdiagramm als Rechteck mit einem Pentagon oben links dargestellt. Es dient der Modellierung komplexer Kontrollstrukturen und beinhaltet einen Interaktionsoperator und einen oder mehrere Interaktionsoperanden. Die Operanden werden durch gestrichelte Linien voneinander getrennt.



Operator	Zweck
`alt`	Alternative Interaktionen
`opt`	Optionale Interaktionen
`break`	Ausnahme Interaktionen
`loop`	Iterative Interaktionen
`seq`	Sequentielle Interaktionen mit schwacher Ordnung
`strict`	Sequentielle Interaktionen mit strenger Ordnung
`par`	Nebenläufige Interaktionen
`critical`	Atomare Interaktionen
`ignore`	Irrelevante Interaktionen
`consider`	Relevante Interaktionen
`assert`	Zugesicherte Interaktionen
`neg`	Ungültige Interaktionen

Definition (Interaktionsreferenzen)
Als Interaktionsreferenz wird die Referenzierung anderen Sequenzdiagramme bezeichnet, wodurch ganze Interaktionsabläufe als auch einzelne Lebenslinien zerlegt werden. Notation:



Auch möglich sind Start- und Zielmarken (ähnlich der Zustandsinvarianten bloß runder), sie sind das Äquivalent zu der goto-Anweisung.

Definition (Verknüpfungspunkte)
Als Verknüpfungspunkt bezeichnen wir die Verbindung von Nachrichten zwischen Sequenzdiagrammen, Interaktionsreferenzen oder kombinierten Fragmenten. Dies können durchaus mal verlorene und gefundene Nachrichten gewesen sein, sie sind nun aber benannt und verknüpft!