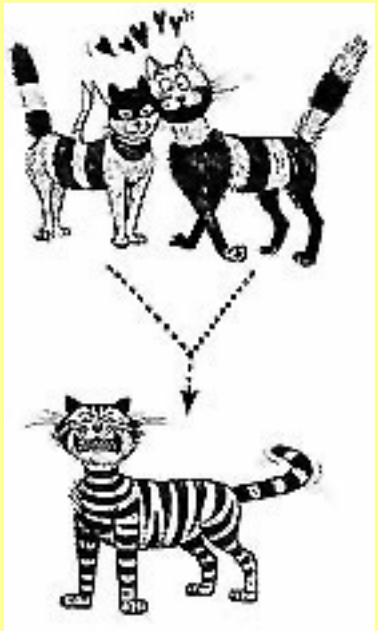
Software Engineering 2
Einleitung



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Warum Objektorientierung?

- Software wird immer komplexer, Benutzer werden anspruchsvoller
  - z.B. Umstieg von der einfachen grafischen Benutzerschnittstelle zur mobilen Multitouch-Anwendung
  - Entwicklung komplexer Technologien ermöglicht Softwareentwicklung mit Bibliotheken und Rahmenwerken
  - Programme müssen verständlich sein und über Technologiegrenzen hinweg modernisiert werden
    - Produktivzeiten von 15-30 Jahren sind keine Seltenheit mehr!
- Aber:
  - Komplexität in der Softwareentwicklung steigt!
  - einfaches Programmieren führt nicht mehr zum Ziel

Idee der Objektorientierung

- Verringerung der Komplexität großer Systeme durch Abstraktion
  - reale Gegenstände werden als Objekte betrachtet
- Prinzip der Abstraktion
  - Betrachtung von tatsächlich relevanten Aspekten eines Gegenstands (eines Objekts)
  - Objekte können bei unterschiedlicher Betrachtungsweise verschiedene Eigenschaften haben.
- Architektur: Abstraktion eines Softwaresystems
  - (mehrere) System-Modelle (Sichten), die konkrete Aspekte eines Gesamtsystems formulieren
  - beschreibt das Wissen bzw. die Prozesse
    - UML ist derzeitiger Stand der Technik, aber nicht Lösung aller Probleme!

Inhalt der Vorlesung

- Wiederholung
  - Vorgehensmodelle
- Grundkonzepte der Objektorientierung
  - Taxonomie
  - Aggregation vs. Assoziation
  - Sammlungen (engl. collections)
  - Polymorphie
  - Persistenz
  - Klassifizierung von Klassen
- Entwurfsmuster (Auswahl)

Inhalt der Vorlesung

- OO Methodik
  - Analyse
  - Design
- Technische Projektkonzeption
  - systematische Technologieentscheidungen
- Wiederholung UML
  - Anforderungen und Fälle
  - Beziehungen
  - Strukturdiagramme

Voraussetzung

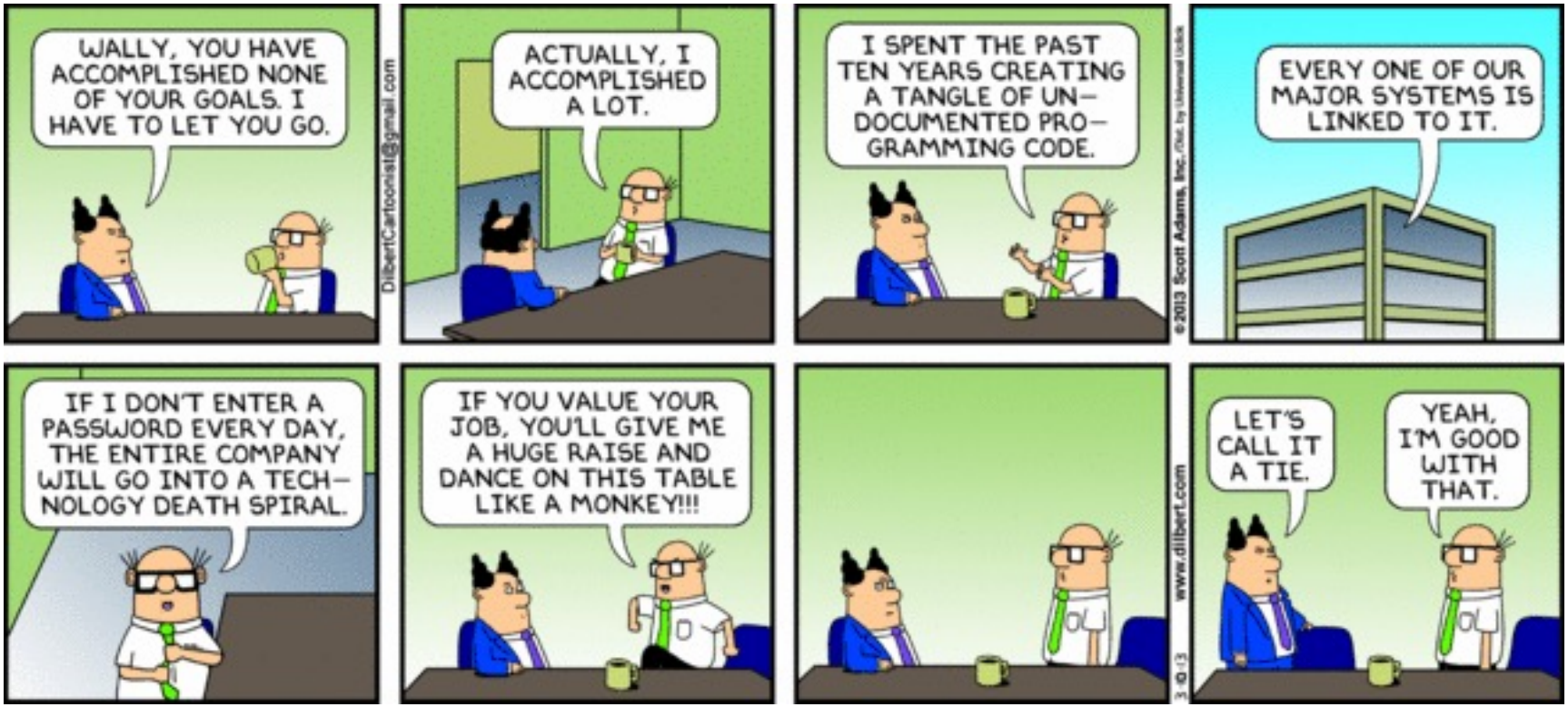
- Grundkonzepte der Objektorientierung
  - Klassen, Schnittstellen
  - Vererbung
- Wiederholung: UML Basiswissen
  - Klassendiagramme
  - Aktivitätsdiagramm
  - Zustandsdiagramm

Falsche Annahmen bei objektorientierter Softwareentwicklung

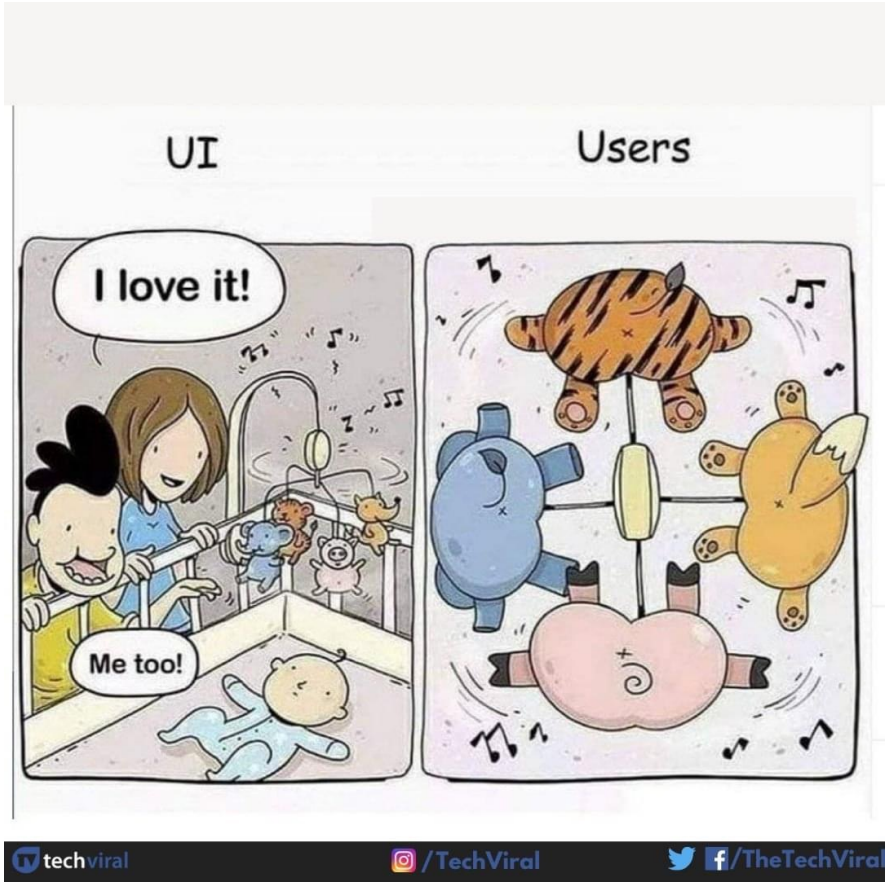
- Die Tiefe der Vererbung spielt keine Rolle bei der Wartung von Software
  - vgl: A Controlled Experiment on Inheritance Depth as a Cost Factor for Code Maintenance; Prechelt, Unger, Philippsen und Tichy; 2003
- Das Bilden einer Unterklasse aus einer API ist der richtige Ansatz zur Programmierung.
  - Objektorientierung wird oft missbraucht.
  - Aggregation vs. Vererbung wird falsch angewendet.

Management-Fehler

Abhängigkeit von Personen im Projekt…



Entwickler unter sich…



Selbstkontrolle

1. Überlegen Sie sich Gründe, warum reine Programmierleistung große Projekte nicht zum Erfolg führen kann.

2. Überlegen Sie, warum Objektorientierung zur Entwicklung von Software mit hoher Qualität geeignet ist.

3. Objektorientierung wird in Projekten oft als Garantie für hohe Softwarequalität verkauft. Begründen Sie, warum diese Aussage falsch ist.

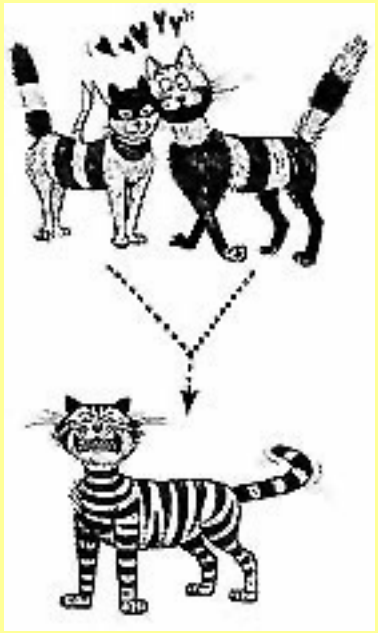
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de

Software Engineering 2
Wiederholung: Vorgehensmodelle



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Vorgehensweise / Prozess-Modell

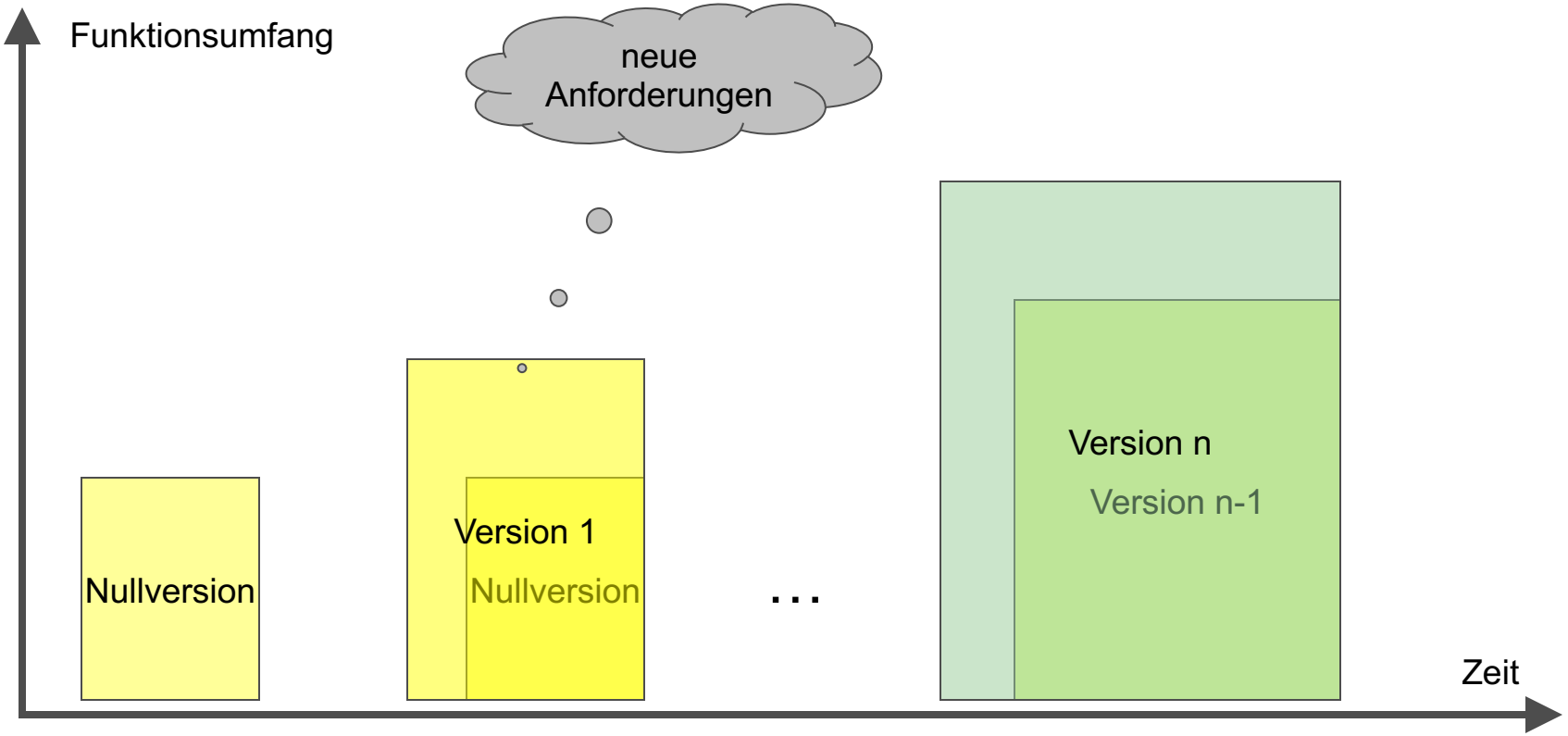
- Legt den organisatorischen Rahmen fest.
  - Welche Aktivitäten sollen in welcher Reihenfolge von welchen Mitarbeitern bearbeitet werden?
- Definiert die Ergebnisse aller Teilschritte.
  - auch Artefakte (engl. artifacts)
- Aktivitäten werden von Mitarbeitern ausgeführt, die jeweils eine oder mehrere Rollen einnehmen.
  - Beachtung von Methoden, Richtlinien, Konventionen, Checklisten und Mustern

- Typische Prozess-Modelle:
  - Wasserfall-Modell, V-Modell (XT), Spiral-Modell, Synchronisieren und Stabilisieren
  - vgl. Software Engineering 1

Iterative Vorgehensweise: evolutionär

- Ausgangspunkt: Kern- und Muss-Anforderungen des Auftraggebers
  - wird als Nullversion des Produkts entwickelt
- Auftraggeber verfeinert seine Anforderungen aus den Erfahrungen mit der Nullversion.
  - Entwicklung einer Produktversion
- Guter Ansatz, wenn Auftraggeber noch nicht alle Anforderungen kennt.
  - Eventuell ist die Nullversion zu schwach: Nicht alle Kern- und Muss-Anforderungen wurde umgesetzt!
  - Zwischen den Versionen kann eine vollständige Überarbeitung möglich sein!

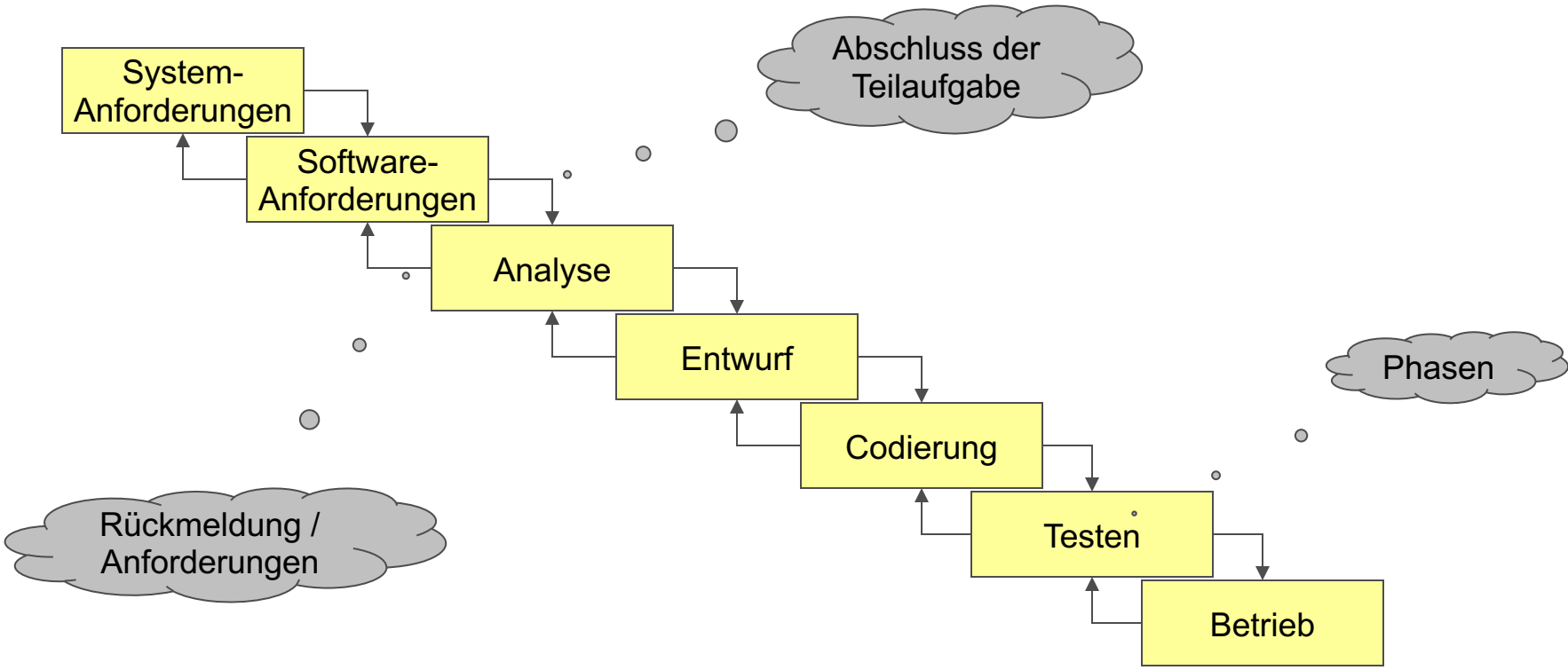
Iterative Vorgehensweise: evolutionär



Iterative Vorgehensweise: inkrementell

- Für die Nullversion sind die vollständigen Anforderungen bekannt.
  - Nur ein Teil davon wird für die Version umgesetzt.
- Erfahrungen des Auftraggebers fließen in die nächste Version ein.
  - Kenntnis aller Anforderungen verhindert hohe Überarbeitungsaufwände.
  - Inkrementelle Erweiterungen lassen sich leicht einbauen.

Vorgehensmodell: Wasserfall



Vorgehensmodell: Wasserfall

- Jede Phase muss vollständig abgeschlossen sein, bevor die nächste Phase beginnt.
- Ergebnisse einer Phase fallen immer nur in die nächste Phase.
  - Rückkopplung und Anforderungen nur in die angrenzende, übergeordnete Phase.
- Wasserfall ist dokumentgetrieben
  - Am Ende jeder Phase steht ein Ergebnisdokument.
- Bewertung: Weit verbreitete Vorgehensweise mit klaren Nachteilen.
  - Trennung in Phasen nicht immer sinnvoll
  - Sequenzielle Ausführung von Phase nicht immer sinnvoll
  - Das eigentlich zu entwickelnde Softwaresystem könnte vernachlässigt werden.
  - Risikofaktoren werden evtl. zur wenig berücksichtigt.

Vorgehensmodell: Rational Unified Process (RUP)

- etabliert von Rational, seit 2003 zu IBM gehörig

- objektorientiertes Vorgehensmodell
  - anwendungsfall-basiert (use case)
    - Menge der Anforderungen beschreibt das zu entwickelnde Softwaresystem
  - architektur-zentriert
    - Modellierung erfolgt mit UML
  - iterativ und inkrementell
    - Entwicklungsprozess wird in Zyklen iteriert.
    - Jeder Zyklus inkrementiert den Funktionsumfang.
      - Zyklen werden so lange ausgeführt, bis alle Anwendungsfälle entwickelt wurden.

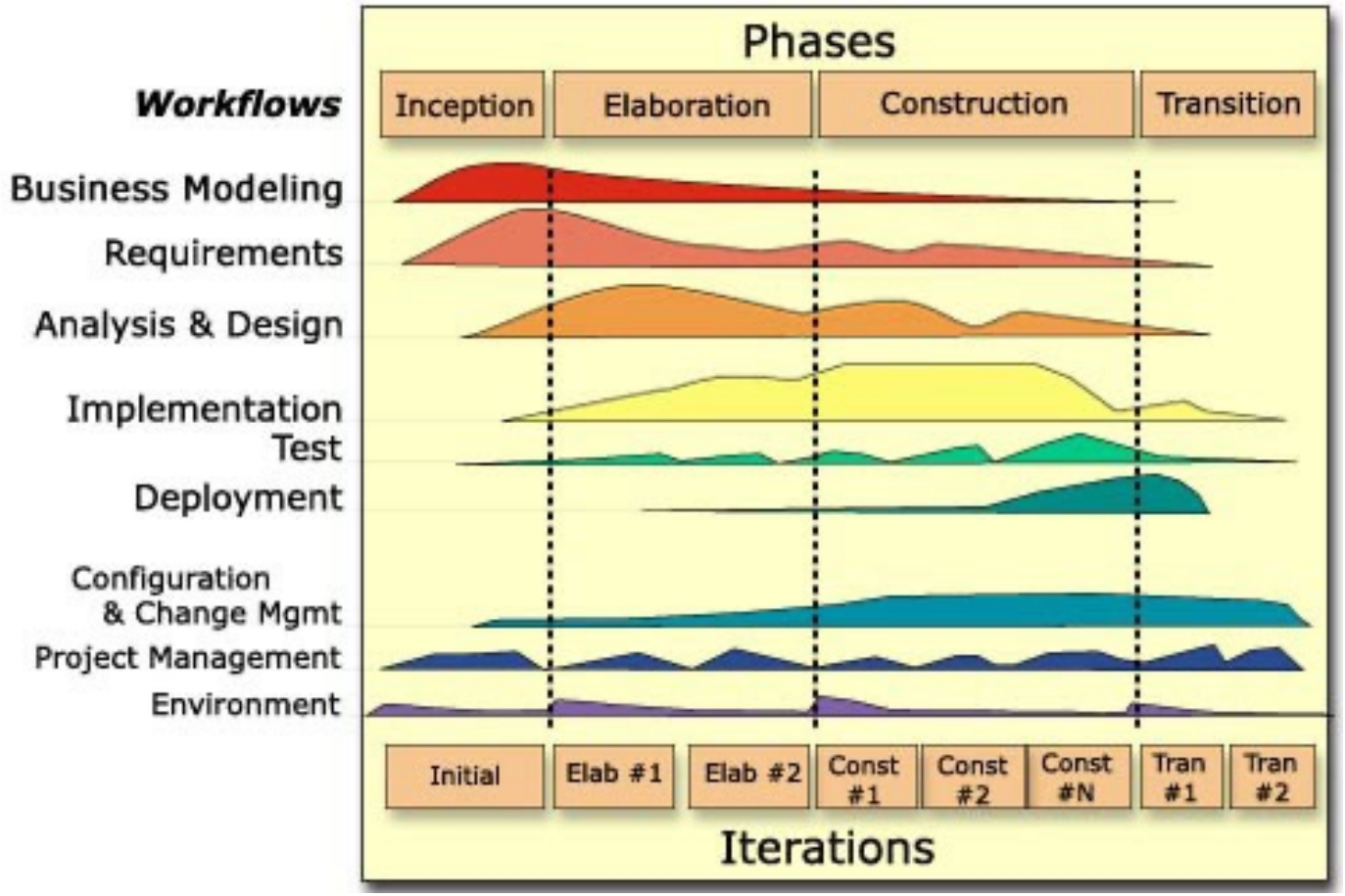
Vorgehensmodell: Rational Unified Process (RUP)

- Phasen eines Zyklus:
  1. Einführung (engl. inception)
     - Anwendungsfälle auswählen, Systemarchitektur entwerfen, Risikoanalyse, Präsentation beim Kunden
     - Ergebnis: Zieldefinition
  2. Entwurf (engl. elaboration)
     - Werkzeuge und Ressourcen festlegen, Hauptrisiken identifizieren, Prototyp der Architektur entwickeln
     - Ergebnis: Architekturbeschreibung
  3. Konstruktion bzw. Entwicklung (engl. construction)
     - Implementierung und Test, Behandlung der Hauptrisiken
     - Ergebnis: lauffähige Software
  4. Übergang (engl. transition)
     - Übergabe der Software an den Kunden, Dokumentation, Schulung, Datenübernahme aus Altsystemen
     - Ergebnis: Projektabschluss

Vorgehensmodell: Rational Unified Process (RUP)

- Umsetzung von Projekterfahrungen (engl. best practices) in folgende Schritte:
  - Geschäftsprozessmodellierung (engl. business modelling)
  - Anforderungsdefinition (engl. requirements)
  - Analyse und Design (engl. analysis and design)
  - Implementierung (engl. implementation)
  - Test (engl. test)
  - Betrieb (engl. deployment)
- begleitende Schritte:
  - Konfigurations- und Änderungs-Management (engl. configuration and change management)
  - Projektmanagement (engl. project management)
  - Umwelt (engl. environment)

Vorgehensmodell: Rational Unified Process (RUP)



Vorgehensmodell: Rational Unified Process (RUP)

- Bewertung: Vorteile:
  - Risiken werden bereits früh erkannt und behandelt.
  - Projekterfahrungen sichern die Praxistauglichkeit.
    - Weiterentwicklungen durch neue Erfahrungen.
  - UML ist heute weit verbreitet und kann als Standard für die Modellbildung angesehen werden.
- Bewertung: Nachteile:
  - Phasen und Zwischenziele (Meilensteine) sind schwer zu definieren.

Selbstkontrolle

1. Warum gibt es unterschiedliche Vorgehensmodelle?
2. Das Wasserfallmodell ist sehr intuitiv. Warum ist es aber nicht für große Projekte geeignet?
3. Viele öffentliche Auftraggeber verlangen Projekte ausschließlich nach V-Modell zu bearbeiten. Begründen Sie, ob diese Forderung sinnvoll ist.
4. Vergleichen Sie das V-Modell mit dem RUP. Wo sehen Sie einen elementaren Unterschied?

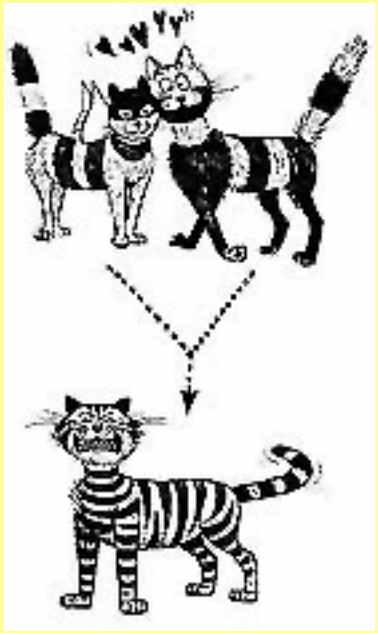
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de

Software Engineering 2
Grundkonzepte der Objektorientierung



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Was zeichnet ein Objekt aus?

- Warum wird eine Lok einer Modelleisenbahn als solche erkannt?

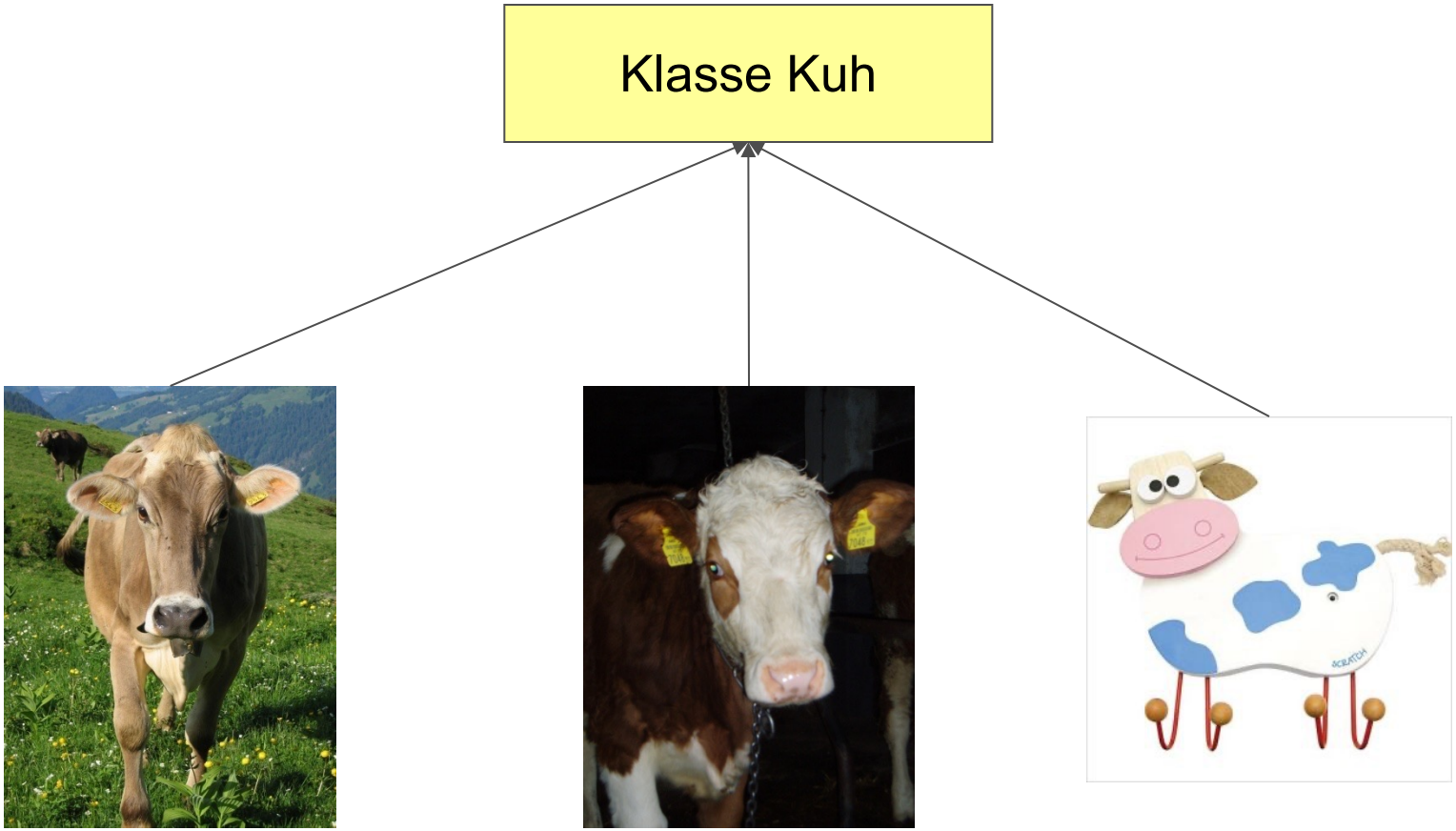
- Objekte sind Abstraktionen der realen Welt. Sie sind Instanzen eines konkreten Bauplanes.

Objekt-Klasse Prinzip

- Eine Klasse beschreibt die Struktur und das Verhalten einer Menge gleichartiger Objekte.

- Ein Objekt ist eine zur Ausführungszeit vorhandene und für ihre Instanzvariablen Speicher allokierende Instanz, die sich entsprechend des Bauplans ihrer Klasse verhält.

Beispiel: Welche wesentlichen Eigenschaften hat eine Kuh?



Kapselung

- Objekte fassen Attribute und Operationen zu einer Einheit zusammen. Attribute sind nur indirekt über die Operationen einer Klasse zugänglich.
  - Wird in der Praxis oft umgangen.
    Beispiel: Deklaration public in C++, Java, C#.

- Attribute
  - enthalten Daten bzw. Informationen
- Operationen (nicht Funktionen!)
  - beschreiben das Verhalten eines Objekts
- Zusicherungen
  - Bedingungen, Voraussetzungen und Regeln, die ein Objekt erfüllen muss
- Beziehungen
  - Verhältnis, das eine Klasse zu anderen Klassen hat, z.B. Vererbung, Assoziation)

Identität: Objekte eindeutig unterscheiden

- Die Identität ist eine Eigenschaft, mit der ein Objekt von allen anderen Objekten unterschieden werden kann.
- Jedes Objekt ist per Definition unabhängig von seinen konkreten Attributwerten von allen anderen Objekten eindeutig zu unterscheiden.

- Beispiele:
  - Ist eine Fahrzeugkennzeichen ein Attribut oder die Identität?
  - Ist die Nummer des Personalausweises die Identität einer Person?

Verantwortlichkeiten: Kohärenzprinzip

- Jede Klasse soll für genau einen Aspekt des Systems verantwortlich sein.
  - Verantwortlichkeit möglichst prägnant formulieren!
- Alle Eigenschaften einer Verantwortlichkeit sollen in einer Klasse zusammengefasst sein.
- Eine Klasse darf keine Eigenschaften besitzen, die nicht zu ihrer Verantwortlichkeit gehört.
- Prinzip: Teile ein Problem und Du hast es zur Hälfte gelöst.

Taxonomie und Vererbung

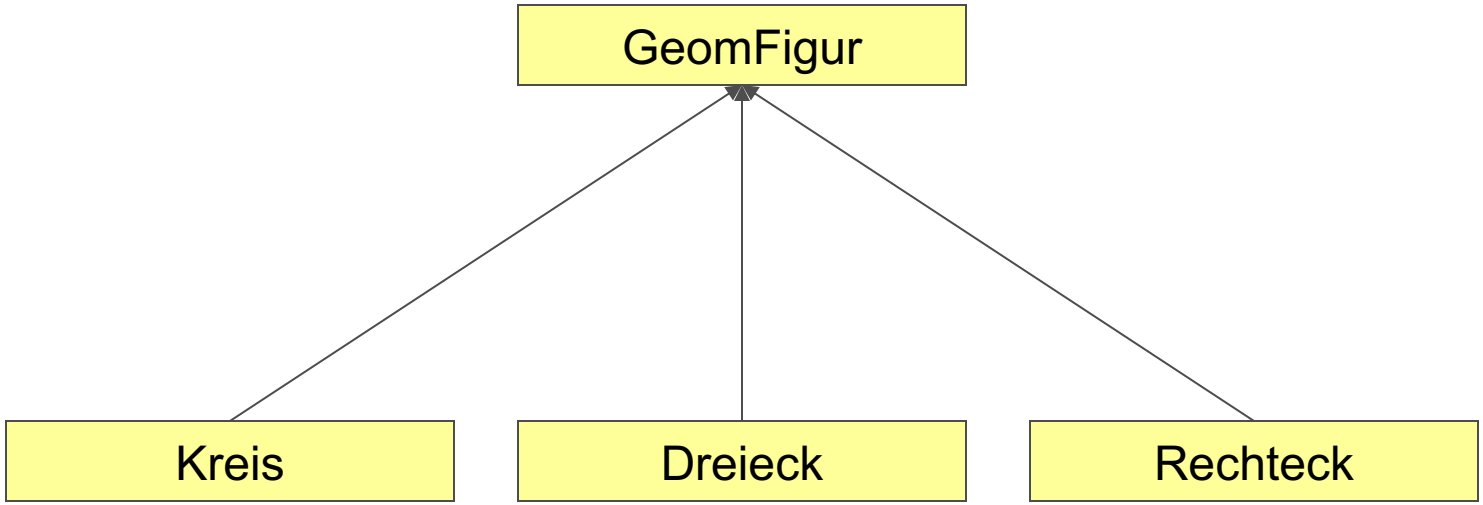
- Vererbung
  - Klassen können Spezialisierungen anderer Klassen darstellen
    - hierarchische Anordnung: Klassen „erben“
  - Klassen erben alle Eigenschaften der übergeordneten Klasse
    - Eigenschaften können spezialisiert werden
    - Eigenschaften können nicht eliminiert werden
- Substitution
  - Objekte von Unterklassen müssen jederzeit anstelle von Objekten ihrer Oberklasse(n) eingesetzt werden können
- Vererbung richtig anwenden!
  - Vererbung ist in den meisten Fällen der falsche Weg!
- Diskriminator
  - Eigenschaft, nach der Unterklassen gebildet werden

Beispiel: ein Kreis und ein Dreieck

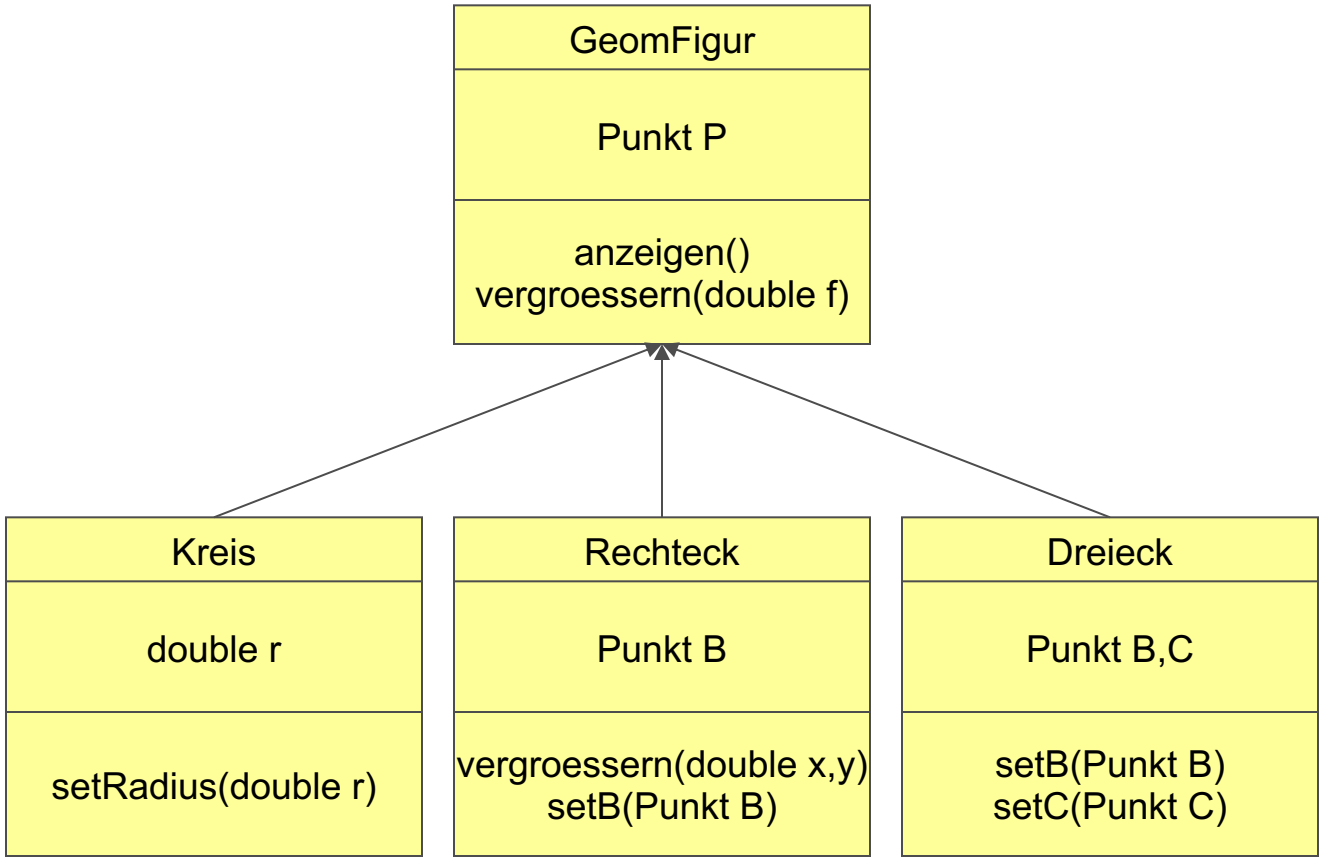
- Kreis
  - Attribute: Mittelpunkt und Radius
  - Operationen: z.B. anzeigen, verschieben, vergrößern
  - Zusicherungen: der Radius kann nicht negativ sein
  - Beziehungen: keine

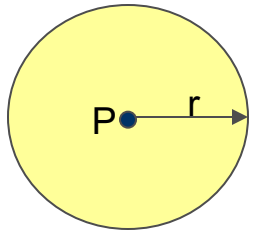
- Dreieck
  - Attribute: Ecken A,B,C (oder Ecke A und 2 Vektoren b,c)
  - Operationen: anzeigen, verschieben, vergrößern
  - Zusicherung: Die Fläche muss größer als 0,001 LE^2 sein
  - Beziehungen: keine

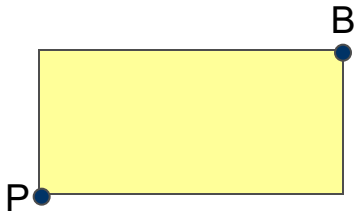
Klassisches Beispiel: geometrische Figuren

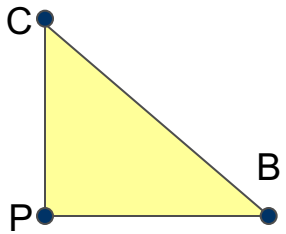


Klassisches Beispiel: geometrische Figuren

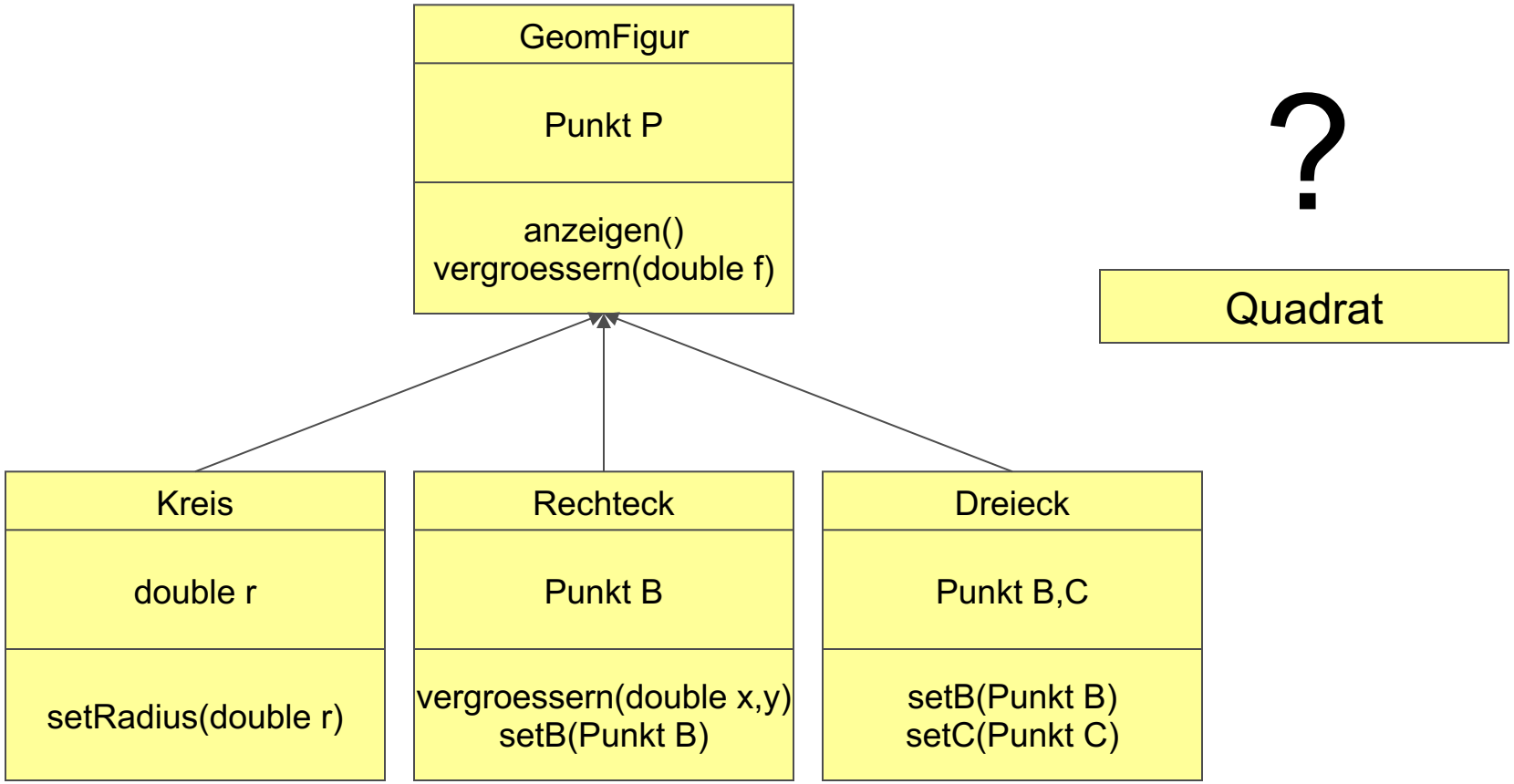








Klassisches Beispiel: geometrische Figuren



Übung: ein Quadrat modellieren

- Modellieren Sie ein Quadrat und betrachten Sie alle Möglichkeiten für eine Einbettung in die Klassenhierarchie der geometrischen Figuren.
  - Welche Variante wählen Sie als Lösung aus?
- Statten Sie das Quadrat mit der passenden Zusicherung aus.
- Welche Alternative sehen Sie zur Modellierung einer eigenen Klasse?
- Fügen Sie zusätzlich einen Diskriminator ein: Anzahl der Ecken.
- Wählen Sie einen alternativen Diskriminator und passen Sie Ihr Klassenmodell an.
- Welche Entwurfsentscheidung müssen Sie ändern, wenn Sie statt eines Quadrates ein Rundeck modellieren?

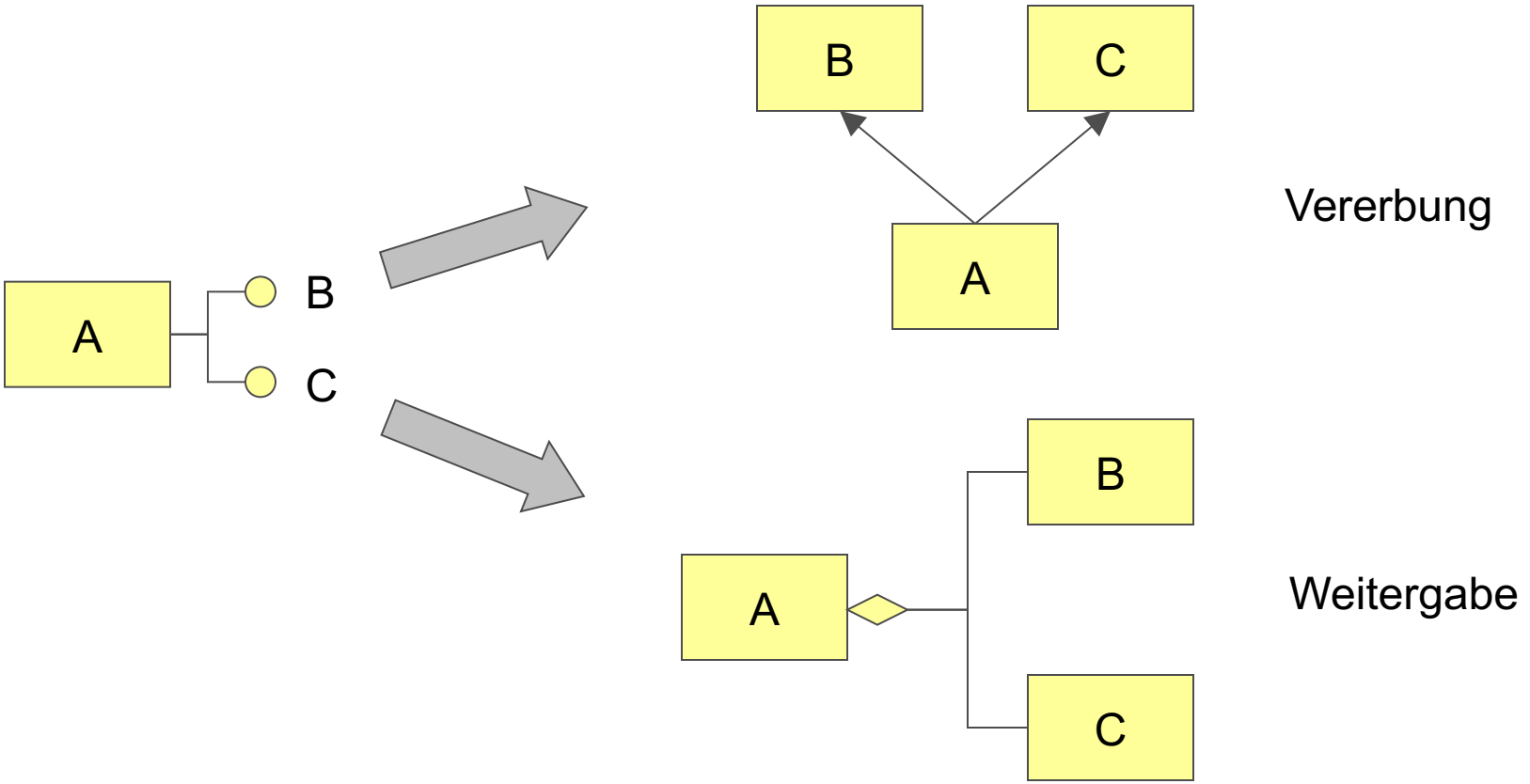
Vererbung: Restriktionen und Probleme

- Vererbung ist einfach, aber nicht ohne Tücken!
- Unterklassen müssen alle Zusicherungen ihrer übergeordneten Klassen erfüllen.
  - Zusicherungs-Verantwortlichkeits-Prinzip
- Eine Unterklasse darf nicht nur um Zusicherungen spezialisieren!

Delegation

- Delegation erlaubt die Mitbenutzung von Eigenschaften anderer Klassen
  - durch Bildung einer Unterklasse (ggfs. mit Mehrfachvererbung)
  - durch die Implementierung einer Delegat-Schnittstelle
- Bildung von Unterklassen
  - Substitutionsprinzip beachten!

Delegation



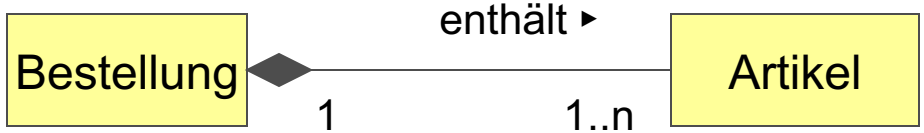
Assoziation, Aggregation und Komposition

- Eine Assoziation repräsentiert eine Beziehung zwischen Objekten einer oder mehrerer Klassen.
  - Kardinalität: eine Mengenangabe für Objekte: z.B. 0, 1, 0..1, 0..n, 1..n
- Eine Aggregation ist ein Spezialfall der Assoziation.
  - definiert eine Beziehung in Form einer Teil/Ganzes Beziehung
- Eine Komposition ist ein Spezialfall der Aggregation
  - definiert eine existenzabhängige Teil/Ganzes Beziehung

Assoziation, Aggregation und Komposition

Assoziation

Aggregation



Komposition

Moderne Sprachkonzepte (1/2)

- Moderne Programmiersprachen mischen Objektorientierung mit funktionaler Programmierung und weiteren Programmierparadigmen.

- Moderne Programmiersprachen besitzen typischerweise riesige Funktionsbibliotheken und sind aufgrund der Mischung recht einfach programmierbar.
  - z.B. Steuerungsklassen funktional
  - z.B. Datenstrukturen objektorientiert

Moderne Sprachkonzepte (2/2)

- Moderne Programmiersprachen nutzen Vorteile aus verschiedenen Welten zur Vereinfachung des Programmcodes und ihrer Programmierung.
- Die Verwendung von Interpretern reduziert die Entwicklungszeit von Software erheblich.
- Prominente Vertreter:
  - EcmaScript, Python

Sammlungen (engl. collections)

- auch Behälter oder Container (engl.)
- Standard in der Objektorientierung
  - z.B. bei grafischen Benutzerschnittstellen
  - z.B. bei Unternehmensanwendungen
- Behälter speichern Objekte eines konkreten Typs
  - passende Operationen werden bereitgestellt
- Beispiele
  - Array, Hashtable, Map

Persistenz

- dauerhafte Speicherung von Objekten auf einem Sekundärspeicher, z.B. Festplatte
  - Objekte müssen das Kriterium „serialisierbar“ erfüllen
    - in Java: java.io.Serializable (hat sich über die Versionen geändert!) implementieren
  - Objekte können zu einem beliebigen Zeitpunkt wiederhergestellt werden.
- Kriterium „serialisierbar“:
  - Ein Objekt ist serialisierbar, wenn
    1. es nur Attribute primitiver Datentypen besitzt und
    2. alle objektwertigen Attribute serialisierbar sind.
  - Achtung: Definition ist rekursiv! Wichtig bei Echtzeitsystemen!

Übung: Persistenz

- In Java ist die Identität eines Objekts über die Methode hashCode() der Klasse java.lang.Object erhältlich. Was geschieht mit der Identität bei der Speicherung eines Objekts auf der Festplatte?
- Entwerfen Sie ein Java-Programm, dass ein Objekt in eine Datei serialisiert und anschließend aus der Serialisierung wieder ein Objekt erzeugt. Überprüfen Sie, ob sich beim Wiederherstellen die Identität ändert:
  - im gleichen Programm in der gleichen Java Virtual Machine (JVM) und
  - in 2 Programmen, die separat gestartet werden.
- Erklären Sie diesen Effekt.
- Die Methode hashCode liefert die Identität im Datentyp int. Überprüfen Sie, ob die damit verbundene Beschränkung von Objekten gleichen Datentyps in der Praxis Auswirkungen besitzt.

Polymorphie

- Polymorhie-Prinzip
  - Eine Operation kann sich in unterschiedlichen Klassen unterschiedlich verhalten.
- statische Polymorhie (Überladung)
  - gleiche Operationen können mit unterschiedlichen Daten ausgeführt werden
    - Operationen mit unterschiedlicher Signatur
  - z.B. + in Java: für die Typen int, float, double, String
- dynamische Polymorphie (Überlagerung)
  - auch als späte Bindung bezeichnet
  - Speicherort einer Operation wird nicht zur Übersetzung sondern zur Laufzeit bestimmt.
  - z.B. anzeigen() bei GeomFigur

Klassifizierung von Klassen und Objekten

- Klassen und Objekte besitzen unterschiedliche Verwendungszwecke.
- Klassifizierung ist der Weg zu strukturierten Modellen und standardisierten Lösungen.
  - z.B. Entwurfsmuster
- Die Trennung der nachfolgenden Klassifizierung ist nicht immer scharf!

Klassifizierung: Entitätsklasse (engl. entity) <<entity>>

- stellt einen Sachverhalt oder realen Gegenstand dar
  - besitzt viele Attribute
  - besitzt viele primitive Operationen, z.B. set/get
  - wird in Analyse- und Strukturmodellen verwendet
  - besitzt ein einfaches Zustandsmodell
- z.B. Auto, Benutzer, Bestellung

Klassifizierung: Steuerungsklasse (engl. control) <<control>>

- definiert eine Ablauf, eine Steuerung oder eine Berechnung
  - besitzt wenige oder keine Attribute
  - existiert oft nur für eine Berechnung
    - wird ggfs. in einem Pool bereitgestellt
  - greift auf Entitätsklassen zur Anforderung von Daten zu und speichert Ergebnisse darin zurück
  - besitzt keine Zustände
- z.B. Klassen in mathematischen Bibliotheken

Klassifizierung: Schnittstellenklasse (engl. interface) <<interface>>

- abstrakte Definition von Schnittstellen
  - keine Attribute
  - keine Assoziationen
  - definiert eine Menge von Operationen
  - werden von Entitäts- und Steuerklassen implementiert
- z.B. java.rmi.Remote

Klassifizierung: Schnittstellenobjekt (engl. boundary) <<boundary>>

- liefert eine Sicht auf eine Menge anderer Objekte
  - bildet eine Zusammenstellung von Eigenschaften
  - delegiert Operationen weiter
  - haben keine Zustande
  - sind gewöhnlich als Singleton implementiert
    - siehe Entwurfsmuster
- z.B. Schnittstelle zu Hardware

Klassifizierung: Typ (engl. type) <<type>>

- definiert eine Menge von Attributen und Operationen.
  - kann (anders als Schnittstellenklassen) Attribute und Assoziationen besitzen
  - besitzt oft abstrakte Operationen
  - wird üblicherweise von Entitätsklassen implementiert und besitzt einen ähnlichen Namen
  - Analysemodelle bestehen hauptsächlich aus Typen

- z.B. Person, Beitragszahler

Klassifizierung: primitive Klasse (engl. primitive) <<primitive>>

- ist eine elementare Klasse in einer Programmiersprache
  - besitzt keine eigenen Persistenzmechanismen
- z.B. java.lang.String

Klassifizierung: Datentyp oder Datenstruktur (engl. data type) <<dataType>>

- Klasse, die einfache Attribute oder primitive Klassen enthält
  - besitzt wenige Attribute und einfache Operationen
  - besitzt oft Methoden zur Umwandlung in andere primitive Klassen
    - z.B. getIntValue, asDate
- z.B. Geld, KundenNr, Dauer

Klassifizierung: Aufzählung (engl. enumeration) <<enumeration>>

- ist eine aufzählbare Wertmenge
  - ist oft dynamisch änderbar
  - wird für die Deklaration von Attributen benötigt
  - besitzt eine frei definierbare Reihenfolge
  - in der Praxis oft als Folge von definierbaren int-Konstanten realisiert
    - z.B. true=42, false=0
      (vgl. Douglas Adams: Per Anhalter durch die Galaxis)

- z.B. Entscheidung = {ja, nein, vielleicht}

Selbstkontrolle

1. Warum sind im Beispiel zu Anfang des Kapitels alle drei Kühe als solche erkennbar?
2. Entwickeln Sie in der Klassenhierarchie GeomFigur eine Klasse Polygon. Welche Varianten gibt es und welche davon sind sinnvoll?
3. Erläutern Sie den Unterschied zwischen Assoziation, Aggregation und Komposition.
4. Welchen Sinn macht der Einsatz von Kontrollklassen?
5. Sind Java-Objekte vom Typ java.lang.Object serialisierbar?

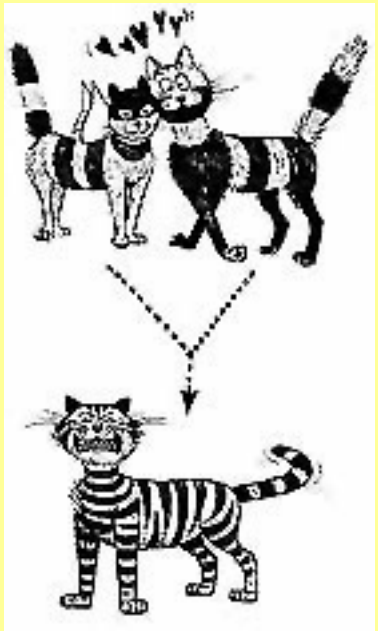
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de

Software Engineering 2
Entwurfsmuster



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

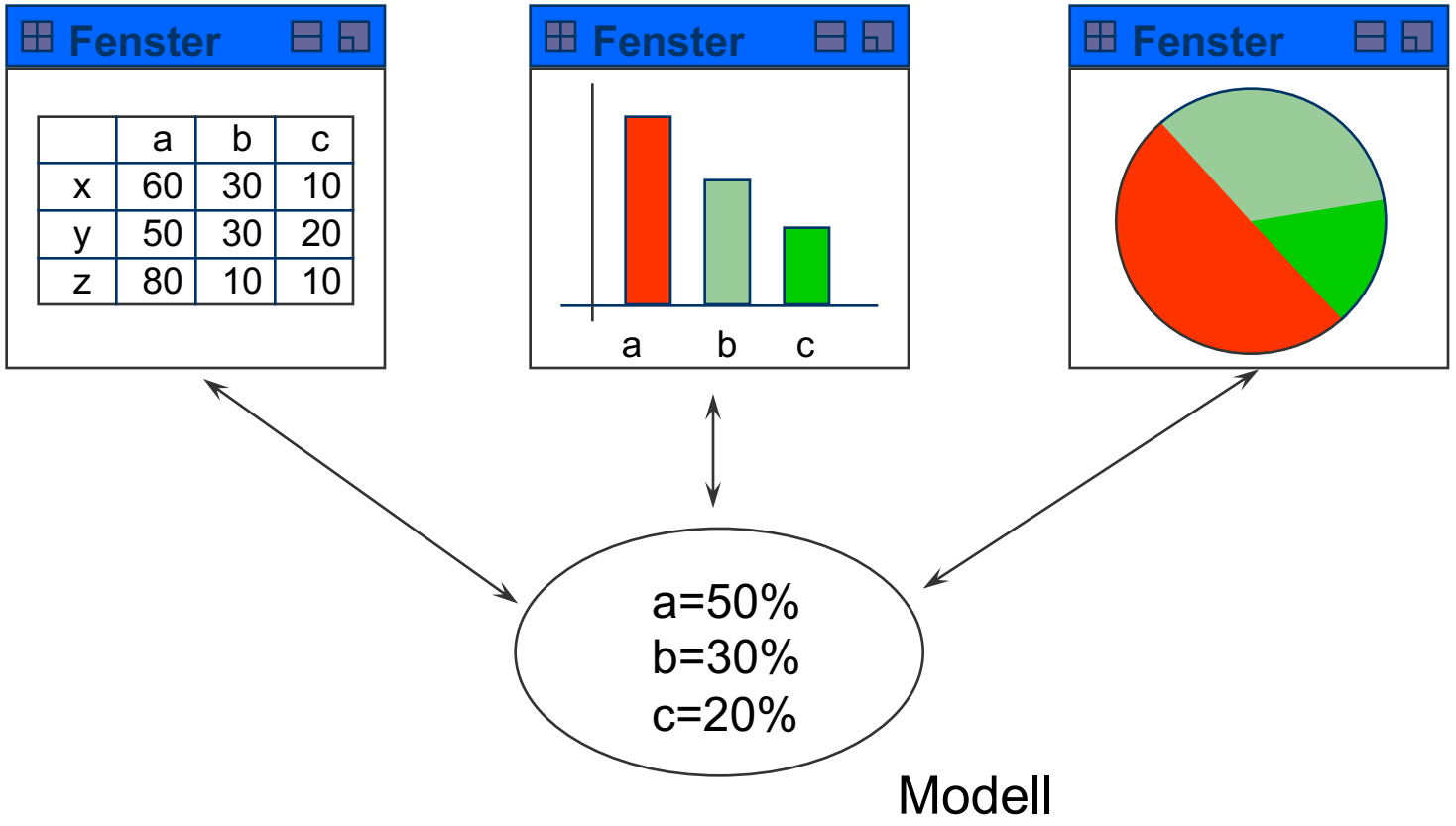
Was ist ein Entwurfsmuster?

- Ein Software-Entwurfsmuster beschreibt eine Familie von Lösungen für ein Software-Entwurfsproblem.
- Das Ziel eines Entwurfsmusters ist die Wiederverwendbarkeit von Entwurfswissen.
- Entwurfsmuster sind für den Entwurf (oder das Programmieren im Großen) was Algorithmen für das Programmieren im Kleinen sind.

Beispiel: Model/View/Controller (MVC)

- MVC ist eine Klassenkombination zur Konstruktion von Benutzerschnittstellen (entstanden zuerst in Smalltalk).
  - Modell (Model):
    - Anwendungsobjekt
  - Sicht (View):
    - Darstellung des Modells auf dem Bildschirm (evtl. mehrfach)
  - Steuerung (Controller):
    - Definiert Reaktion der Benutzerschnittstelle auf Eingaben
- MVC ist ein Entwurfsmuster, das die drei Entwurfsmuster Beobachter, Kompositum und Strategie kombiniert.

Muster 1 im MVC: Beobachter (*Observer*)



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Muster 1 im MVC: Beobachter

- Muster 1: Die Beziehung zwischen Modell und Sichten ist ein Beispiel für das Beobachter-Muster.
  - Es gibt eine Registrierungs- und Benachrichtigungs-Interaktion zwischen dem Modell (dem „Subjekt”) und den Sichten (den „Beobachtern”).
  - Wenn Daten im Modell sich ändern, werden die Sichten benachrichtigt. Daraufhin aktualisiert jede Sicht sich selbst durch Zugriff auf das Modell.
  - Die Sichten wissen nichts voneinander. Das Modell weiß nicht, in welcher Weise die Sichten die Daten des Modells verwenden (Entkopplung).

Muster 2 im MVC: Kompositum

- Muster 2: Sichten können geschachtelt sein. Eine zusammengesetzte Sicht ist ein Beispiel für das Muster Kompositum.
  - Zum Beispiel kann ein Objektinspektor aus geschachtelten Sichten bestehen und kann selbst in der Benutzerschnittstelle eines Debuggers enthalten sein.
  - Die Klasse ZusammengesetzteSicht ist eine Unterklasse von Sicht. Ein Exemplar von ZusammengesetzteSicht kann genauso wie ein Exemplar von Sicht benutzt werden. Es enthält und verwaltet geschachtelte Sichten.

Muster 3 im MVC: Strategie

- Muster 3: Die Beziehung zwischen Sicht und Steuerung ist ein Beispiel für ein Strategie-Muster.
  - Es gibt mehrere Unterklassen von Steuerung, die verschiedene Antwortstrategien implementieren. Zum Beispiel werden Tastatureingaben unterschiedlich behandelt, oder ein Menü wird anstelle von Tastaturkommandos benutzt.
  - Eine Sicht wählt eine gewisse Unterklasse aus, die die entsprechende Antwortstrategie implementiert. Diese kann auch dynamisch ausgewählt werden (zum Beispiel um ausgeschaltete Zustände zu ignorieren).

Wozu überhaupt Entwurfsmuster?

- Muster verbessern Kommunikation im Team.
  - Entwurfsmuster bilden eine nützliche Terminologie, d.h. bieten Begriffe und Kurzformeln für die Diskussion zwischen Entwicklern über komplexe Konzepte.

- Muster erfassen wesentliche Konzepte und bringen sie in eine verständliche Form
  - Muster helfen Entwürfe zu verstehen,
  - dokumentieren Entwürfe kurz und knapp,
  - verhindern unerwünschte Architektur-Drift,
  - verdeutlichen Entwurfswissen.

Wozu überhaupt Entwurfsmuster?

- Muster dokumentieren und fördern den Stand der Kunst
  - Muster helfen weniger erfahrenen Entwerfern.
  - Muster vermeiden die Neuerfindung des Rades.
- Ein Muster ist keine feste Regel, der man blind folgt, sondern ein Vorschlag und ein Satz von Alternativen zur Lösung eines Problems. Anpassung erforderlich.
  - Muster können Code-Qualität, und -Struktur verbessern.
- Muster fördern gute Entwürfe und guten Code durch Angabe konstruktiver Beispiele.

Die Entwicklung guter Muster ist schwierig und zeitaufwendig

- Herausstellen der wesentlichen Merkmale ist schwierig.
- Die Zusammenstellung ist ein iterativer Prozess und erfordert eine Anwendung und Verfeinerung.
- Erfahrung im Entwurf und Dokumentieren von Mustern ist hilfreich.
- Entwurfsmuster sind nicht notwendigerweise objektorientiert.

Beschreibungsstruktur für Entwurfsmuster

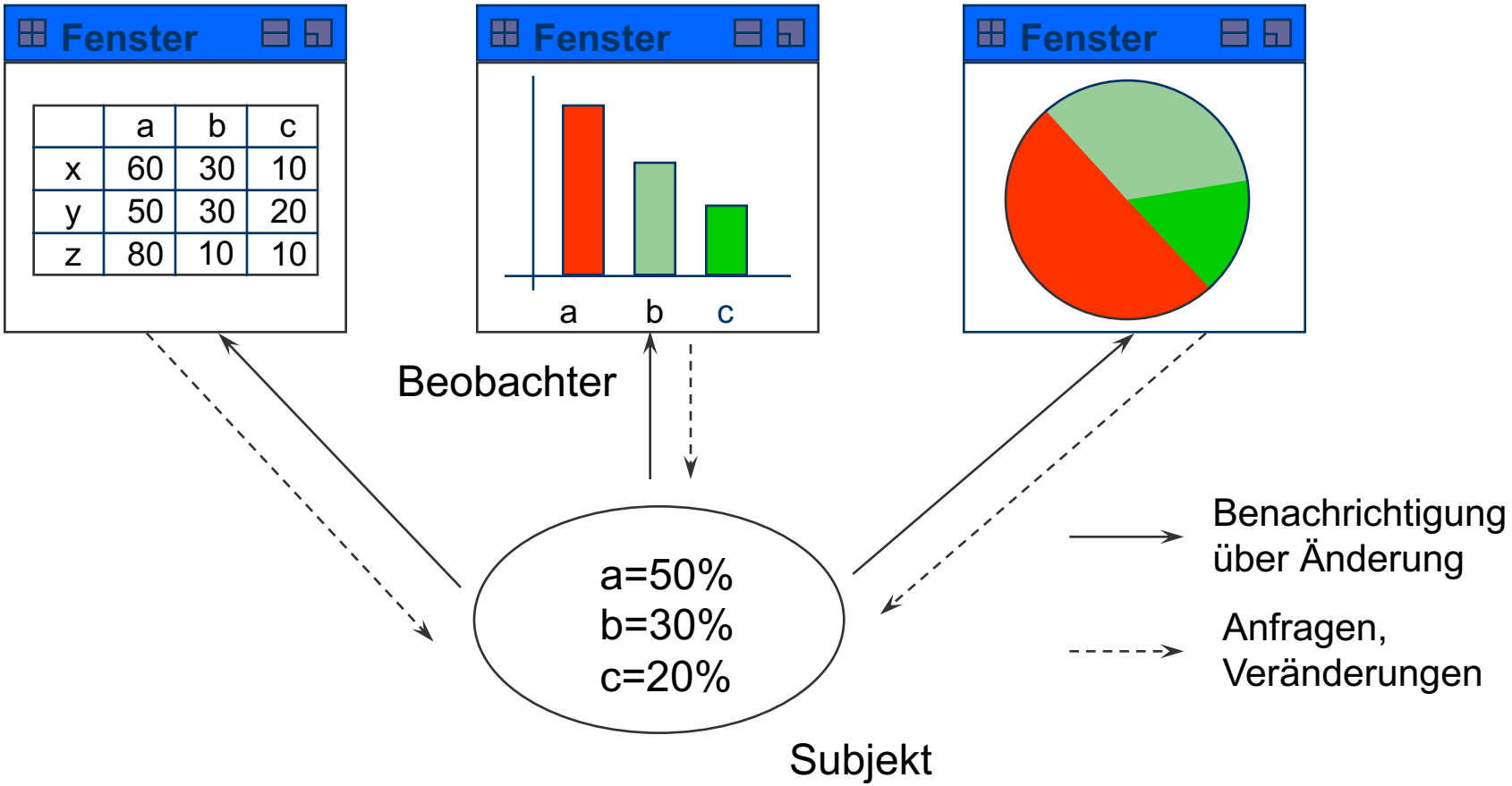
- Name (einschließlich Synonyme)
- Aufgabe und Kontext
- Beschreibung der Lösung
  - Struktur (Komponenten, Beziehungen)
  - Interaktionen und Konsequenzen
  - Implementierung
  - Beispielcode

Beobachter (Observer)

- Zweck
  - Definiert eine 1-zu-n Abhängigkeit zwischen Objekten, so dass die Änderung eines Zustandes eines Objektes dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

- Auch bekannt als Abhängigkeit, Publizieren-Abonnieren, Subjekt-Beobachter

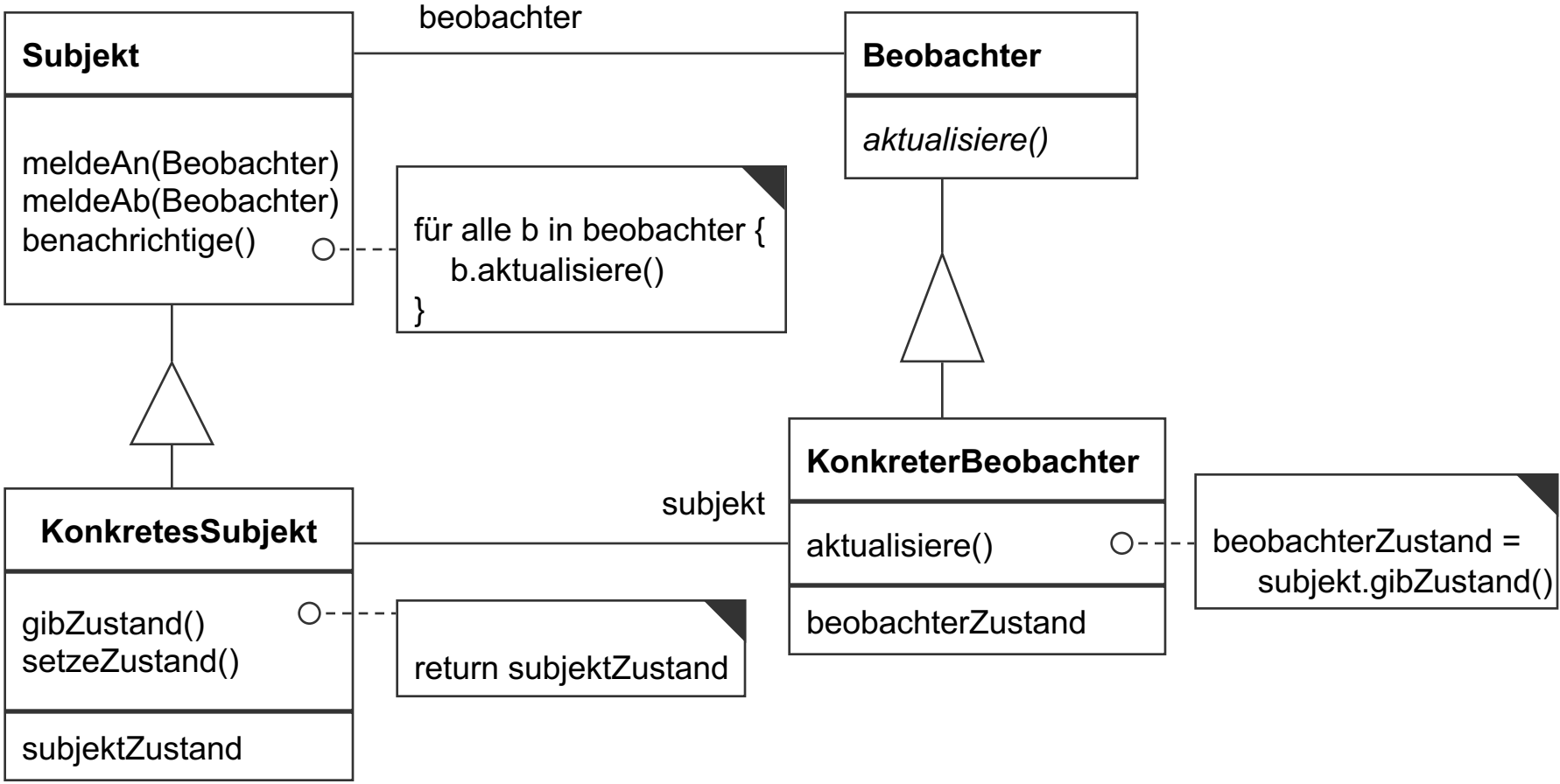
Beobachter


Fenster
Fenster
Fenster
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a b c
Beobachter
a=50%
b=30%
c=20%
Subjekt
Benachrichtigung über Änderung
Anfragen, Veränderungen

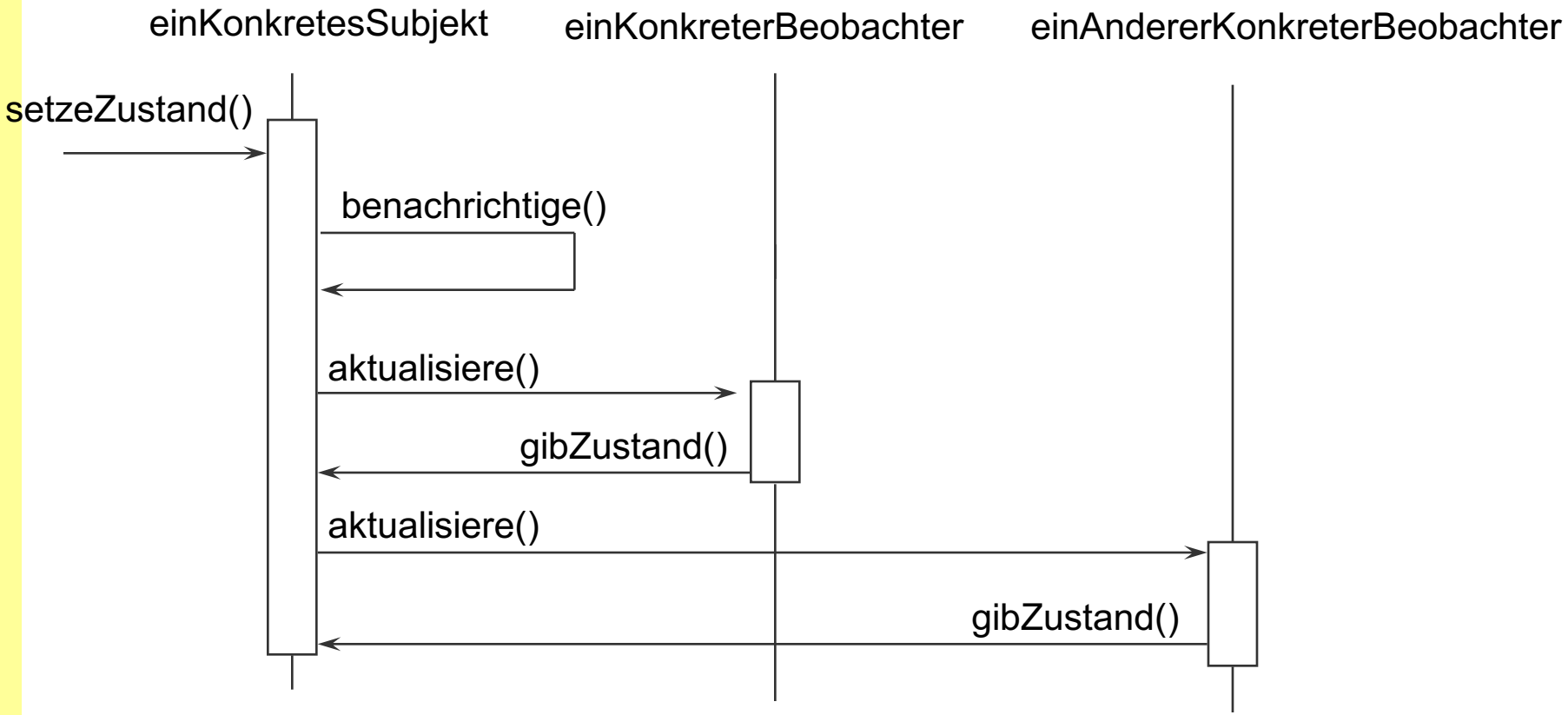
Beobachter: Motivation

- Teilt man ein System in eine Menge von interagierenden Klassen auf, so muss die Konsistenz zwischen den miteinander in Beziehung stehenden Objekten aufrechterhalten werden.
- Eine enge Kopplung dieser Klassen ist nicht empfehlenswert, weil dies die individuelle Wiederverwendbarkeit einschränkt.
- Im MVC-Beispiel wissen die Tabellendarstellung und die Säulendarstellung nichts voneinander. Damit können sie unabhängig voneinander wiederverwendet werden. Trotzdem verhalten sich beide Objekte so, als ob sie einander kennen würden.

Struktur des Beobachters



Interaktionsdiagramm Beobachter



Beobachter: Anwendbarkeit

- Wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt und man nicht weiß, wie viele und welche Objekte geändert werden müssen.
- Wenn ein Objekt andere Objekte benachrichtigen muss, ohne Annahmen über diese Objekte zu treffen.
- Wenn eine Abstraktion zwei Aspekte besitzt, wobei einer von dem anderen abhängt. Die Kapselung dieser Aspekte in separaten Objekten ermöglicht unabhängige Wiederverwendbarkeit.

Beobachter: Konsequenzen

- Subjekte und Beobachter können unabhängig voneinander wiederverwendet werden.
- Beobachter können neu hinzugefügt oder entfernt werden, ohne das Subjekt oder andere Beobachter zu ändern.
- Die abstrakte Kopplung zwischen Subjekt und Beobachter wird durch die Benachrichtigung erreicht. Subjekt und Beobachter gehören verschiedenen Schichten der Benutzt-Hierarchie an, ohne dabei Zyklen zu erzeugen. (Subjekt benutzt nicht die Beobachter, aber umgekehrt.)

Beobachter: Konsequenzen

- Automatischer Rundruf von Änderungen.
- Beobachter entscheiden selbst, ob sie die Benachrichtigung ignorieren oder nicht.
- Der Aufwand der Aktualisierung kann versteckt sein.
  - Eine einfache Benachrichtigung kann zu einer Kaskade von Aktualisierungen bei den Beobachtern führen.
  - Die Botschaft enthält keinen Hinweis was geändert wurde. Ein erweitertes Protokoll kann verwendet werden, um den Beobachtern die konkrete Änderung mitzuteilen.

Beobachter: Implementierung

- Wenn mehr als ein Subjekt beobachtet wird:
  - Verwende Subjekt als Parameter von aktualisiere(Subjekt s).
- Auslösen der Aktualisierung:
  - setzeZustand() ruft benachrichtige(),
  - oder Klienten rufen benachrichtige() explizit.
- Löschen eines Beobachters:
  - Als erstes beim entsprechenden Subjekt abmelden.

Beobachter: Implementierung

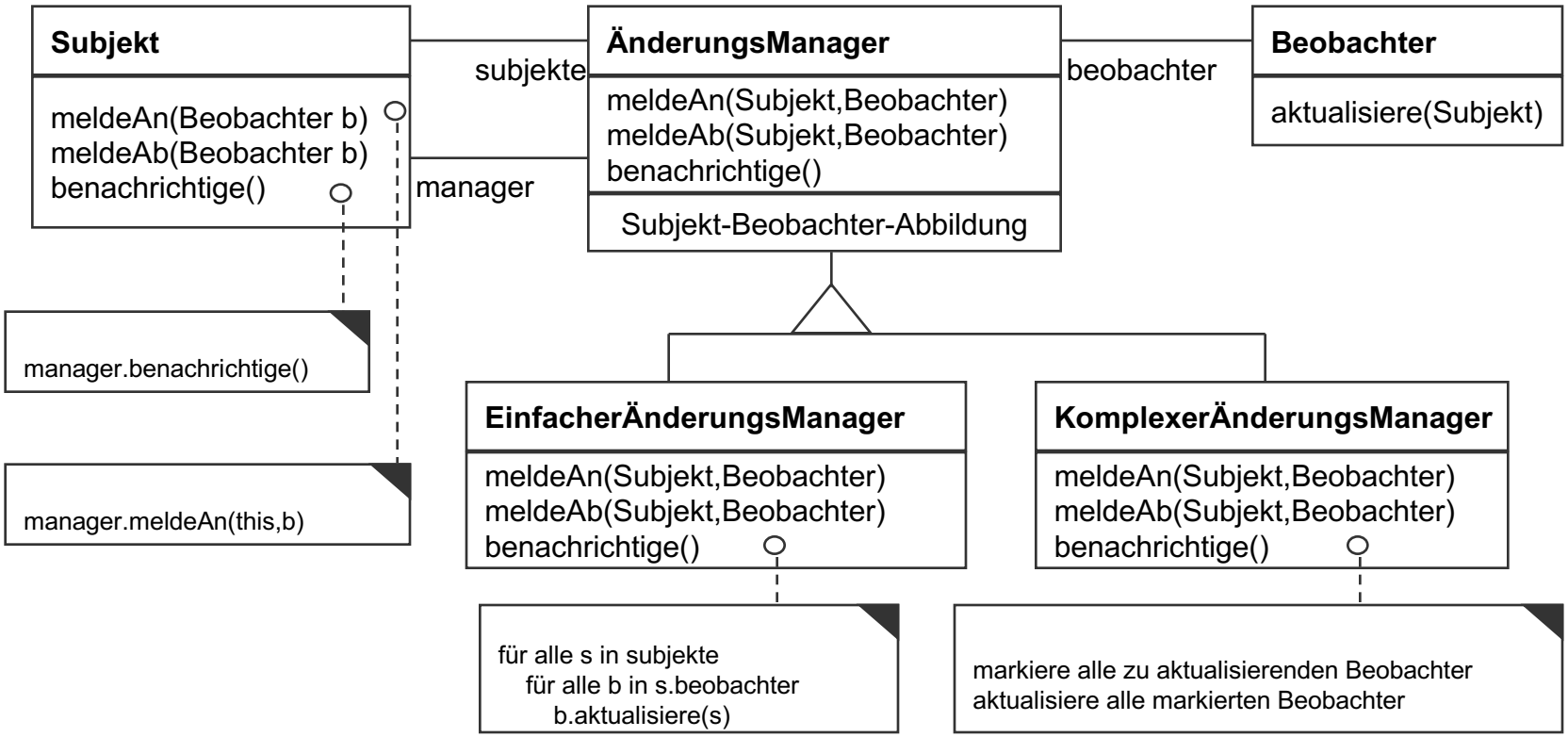
- Subjekte müssen vor der Benachrichtigung konsistent sein. Wenn eine Subjekt-Unterklasse geerbte Operationen aufruft, kann versehentlich die Oberklasse eine Benachrichtigung auslösen bevor das Unterklassenobjekt komplett konsistent ist.

- Aktualisierung: Pull- und Push-Modell
  - Pull-Modell: Beobachter holt alle Daten direkt vom Subjekt (kann ineffizient sein)
  - Push-Modell: Subjekt schickt Änderungsdaten an die Beobachter in aktualisiere() (kann Wiederverwendbarkeit reduzieren)

Beobachter: Implementierung

- ÄnderungsManager zwischen Subjekt und Beobachtern:
  - bildet Subjekt auf seine Beobachter ab und bietet eine Schnittstelle zur Verwaltung dieser Abbildung,
  - aktualisiert bei Anforderung durch das Subjekt alle abhängigen Beobachter,
  - vermeidet mehrfache Aktualisierungen,
  - kapselt komplexe Aktualisierungssemantik.

Beobachter mit ÄnderungsManager



Kompositum (Composite)

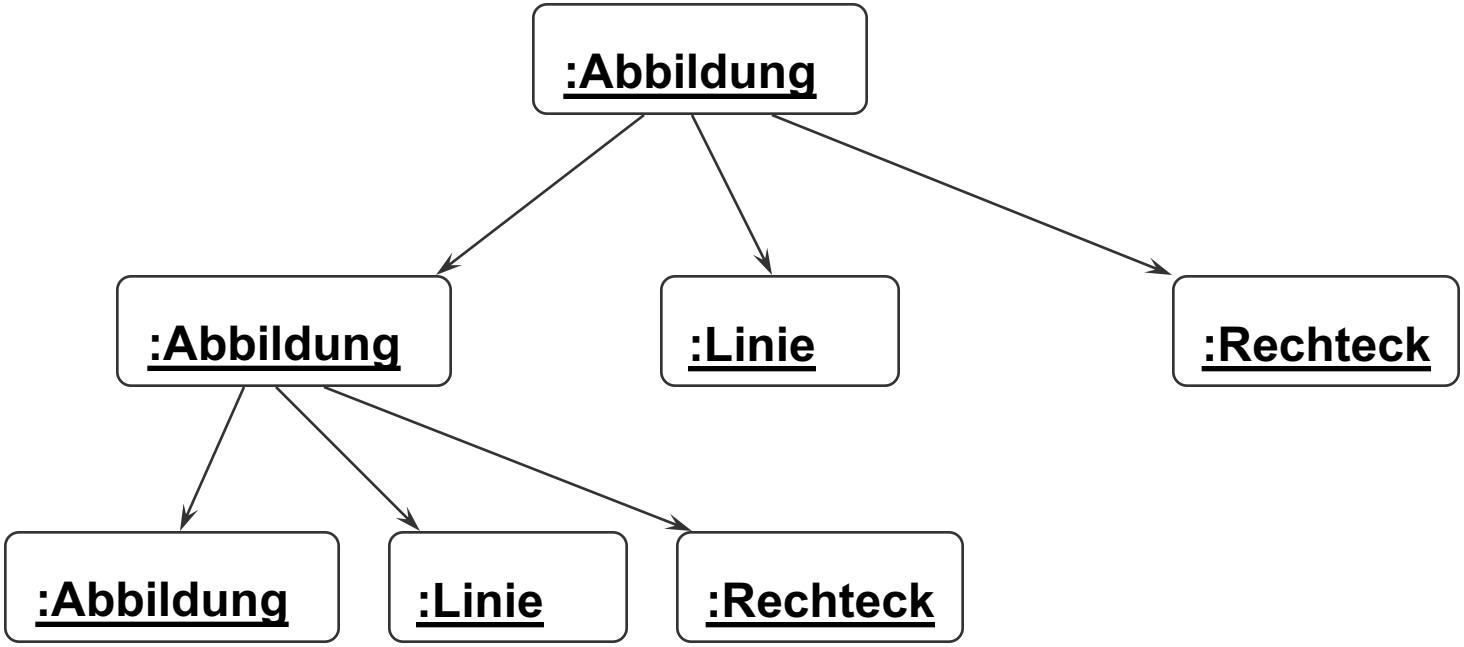
- Zweck
  - Füge Objekte zu Baumstrukturen zusammen, um Bestands-Hierarchien zu repräsentieren. Das Muster ermöglicht es Klienten, sowohl einzelne Objekte als auch Aggregate einheitlich zu behandeln.

Kompositum: Motivation

- Bestands-Hierarchien treten überall dort auf, wo komplexe Objekte modelliert werden, wie beispielsweise Datei-Systeme, graphische Anwendungen, Textverarbeitung, CAD, CIM, ...
  Bei diesen Anwendungen werden einfache Objekte zu Gruppen zusammengefasst, welche wiederum zu größeren Gruppen zusammengefügt werden können.

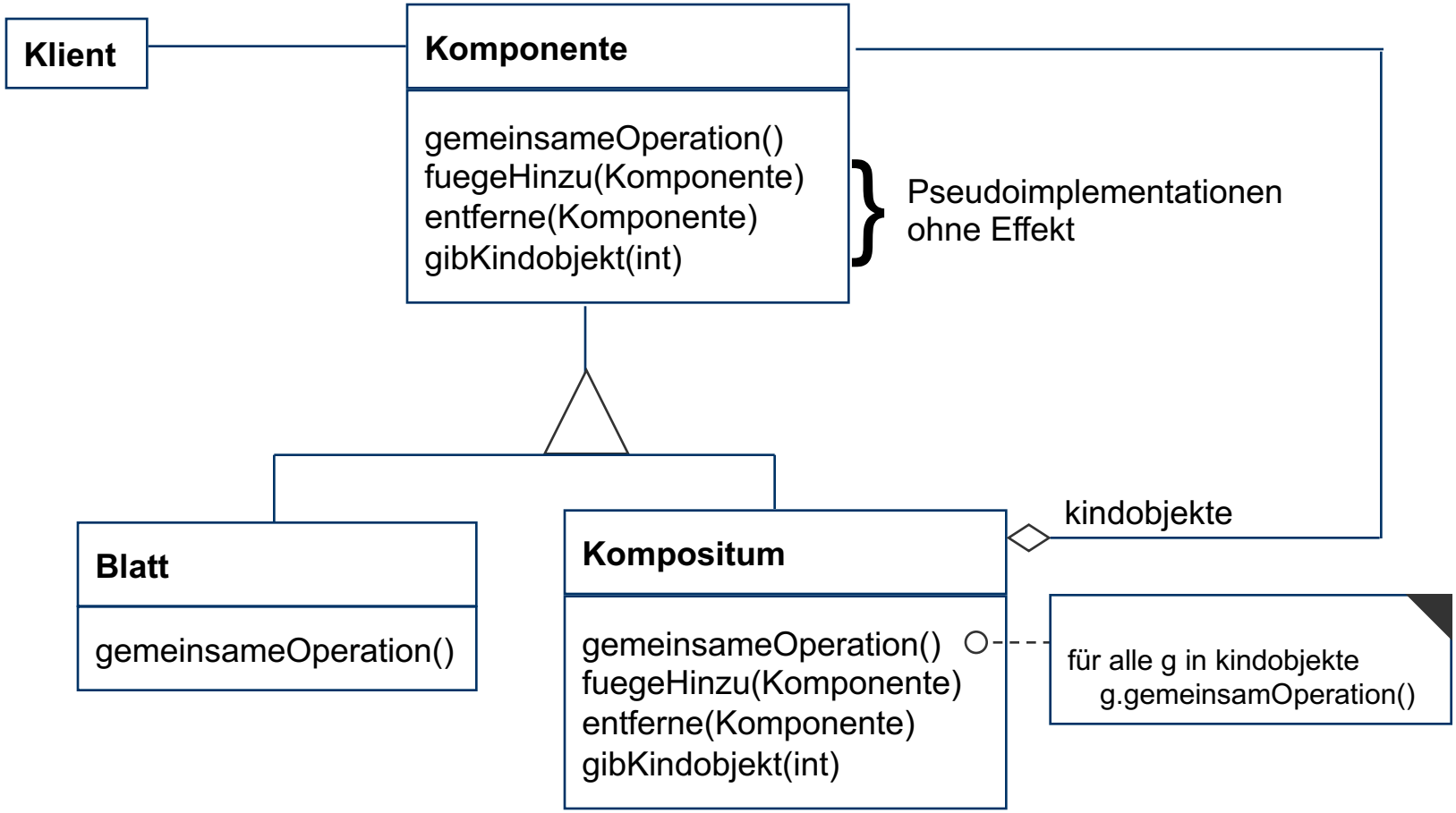
- Häufig soll dabei die Behandlung von Objekten und Aggregaten durch das Programm einheitlich sein. Das Kompositum isoliert die gemeinsamen Eigenschaften von Objekt und Aggregat und bildet daraus eine Oberklasse.

Beispiel für Kompositum: Zusammengefügte Graphik-Objekte

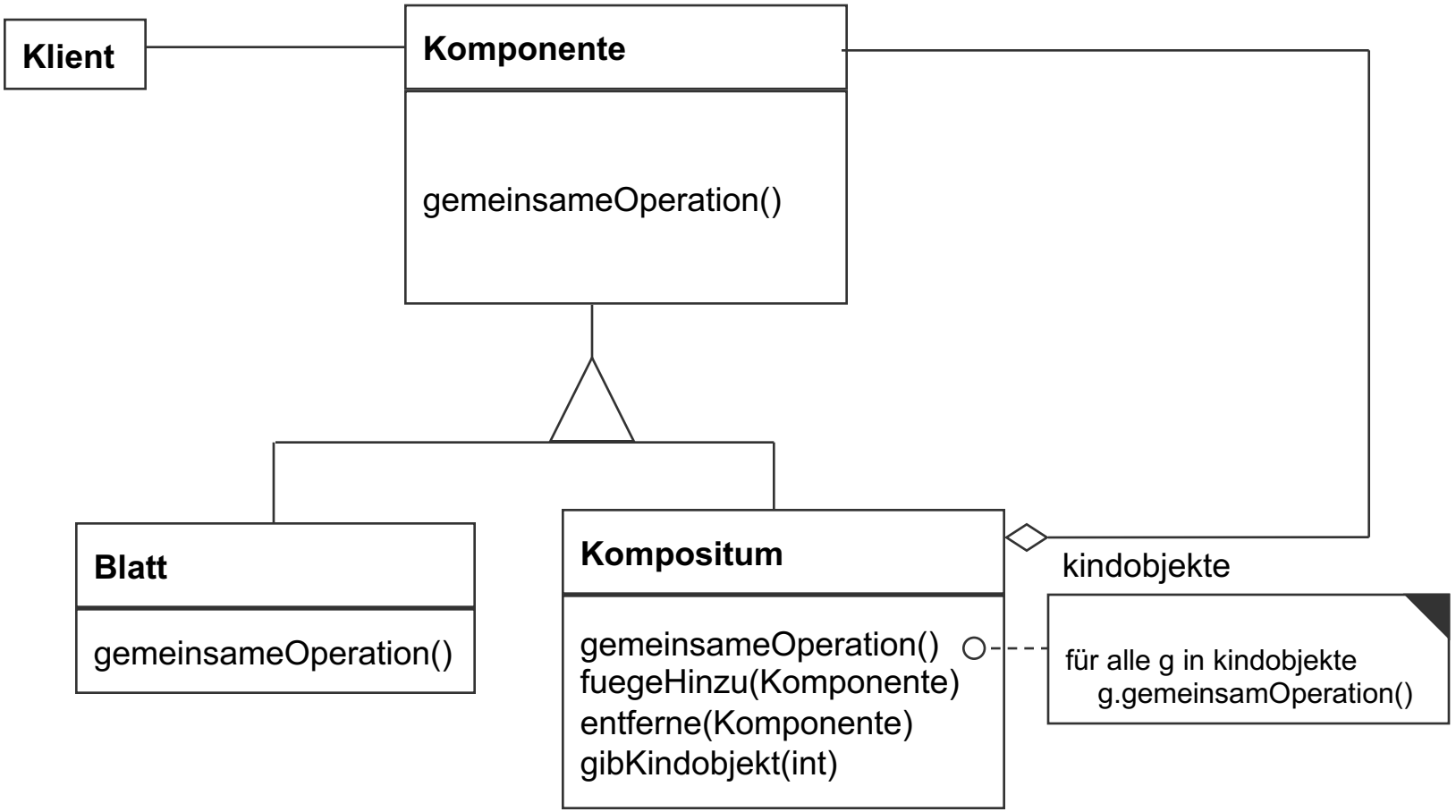


gemeinsame Operationen: zeichne(), verschiebe(), lösche(), skaliere()

Struktur für Kompositum: Kompositum-Operationen in der Komponente



Struktur für Kompositum 2
Kompositum-Operationen im Kompositum

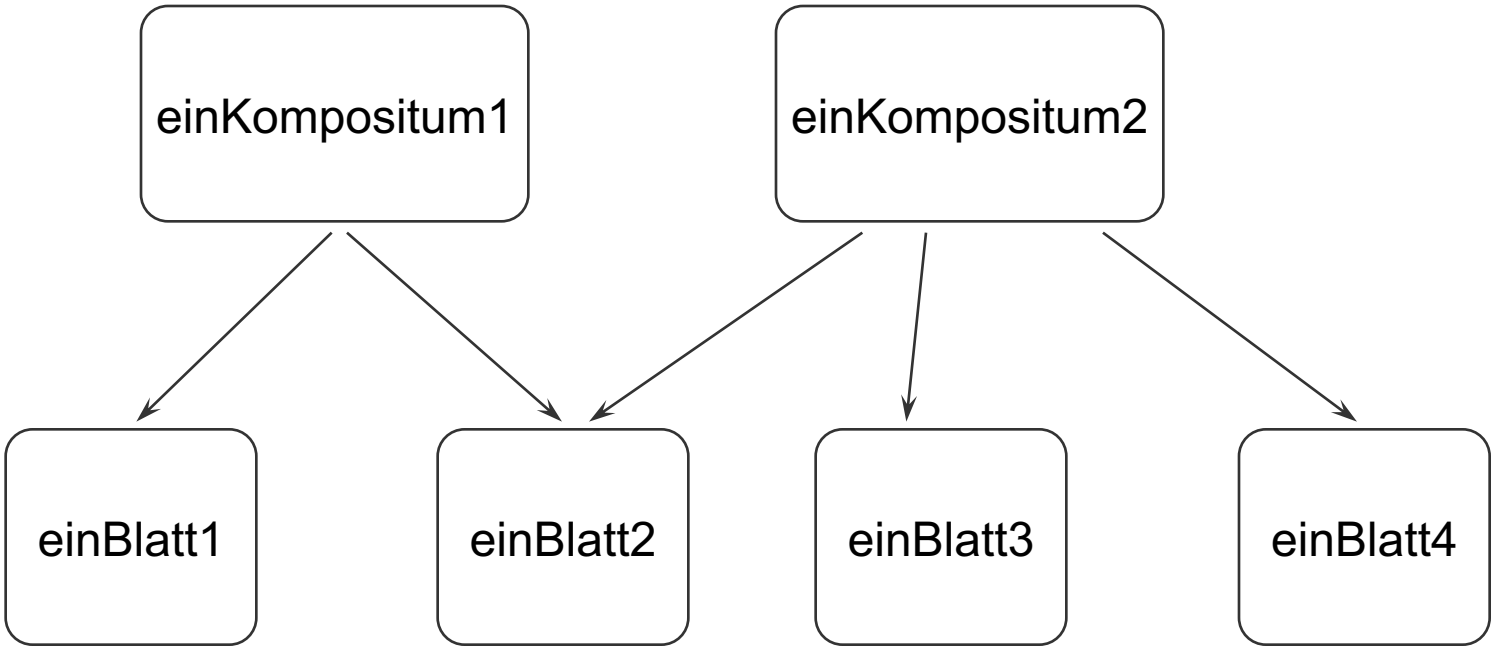


Kompositum: Anwendbarkeit

- Die Klasse Kompositum enthält und manipuliert die Behälter-Datenstruktur, die die Komponenten speichert.

- Anwendbarkeit
  - Wenn Bestands-Hierarchien von Objekten repräsentiert werden sollen.
  - Wenn die Klienten in der Lage sein sollen, die Unterschiede zwischen zusammengesetzten und einzelnen Objekten zu ignorieren.

Kompositum: Implementierung



Kompositum: Implementierung

- Es ist häufig nützlich eine Eltern-Referenz in jeder Komponente zu führen.
  Diese Referenz kann von den fügeHinzu und entferne Methoden des Kompositums gepflegt werden.

- Das Teilen von Komponenten kann zu mehreren Eltern und damit zu Zweideutigkeiten führen.

Kompositum: Implementierung

- Maximieren der Komponenten-Schnittstelle
  - Die Komponenten-Schnittstelle sollte so viele gemeinsame Methoden des Kompositums und der Blätter wie möglich definieren um Transparenz zu garantieren.
  - Wenn Methoden des Kompositums in der Komponente definiert werden, sollte gibKindobjekt bei Blättern nichts zurückgeben – das kann auch durch eine entsprechende Implementierung in der Komponente erreicht werden.
  - fuegeHinzu und entferne sollten bei Blättern fehlschlagen (und einen Fehler zurückgeben oder eine Ausnahme generieren).

Kompositum: Implementierung

- Speichern der Kinder:
  - Felder, Listen oder Hash-Tabellen – abhängig von der Anwendung und benötigten Effizienz.
  - Bei einer festen Anzahl Kinder verwende explizite Variablen (und spezialisierte fuegeHinzu / entferne / gibKindobjekt Operationen).
  - Die Kinder müssen unter Umständen in einer bestimmten Reihenfolge gelassen werden (beispielsweise bei Anordnern (Layout Manager)).

Übung: Kompositum bei GUI

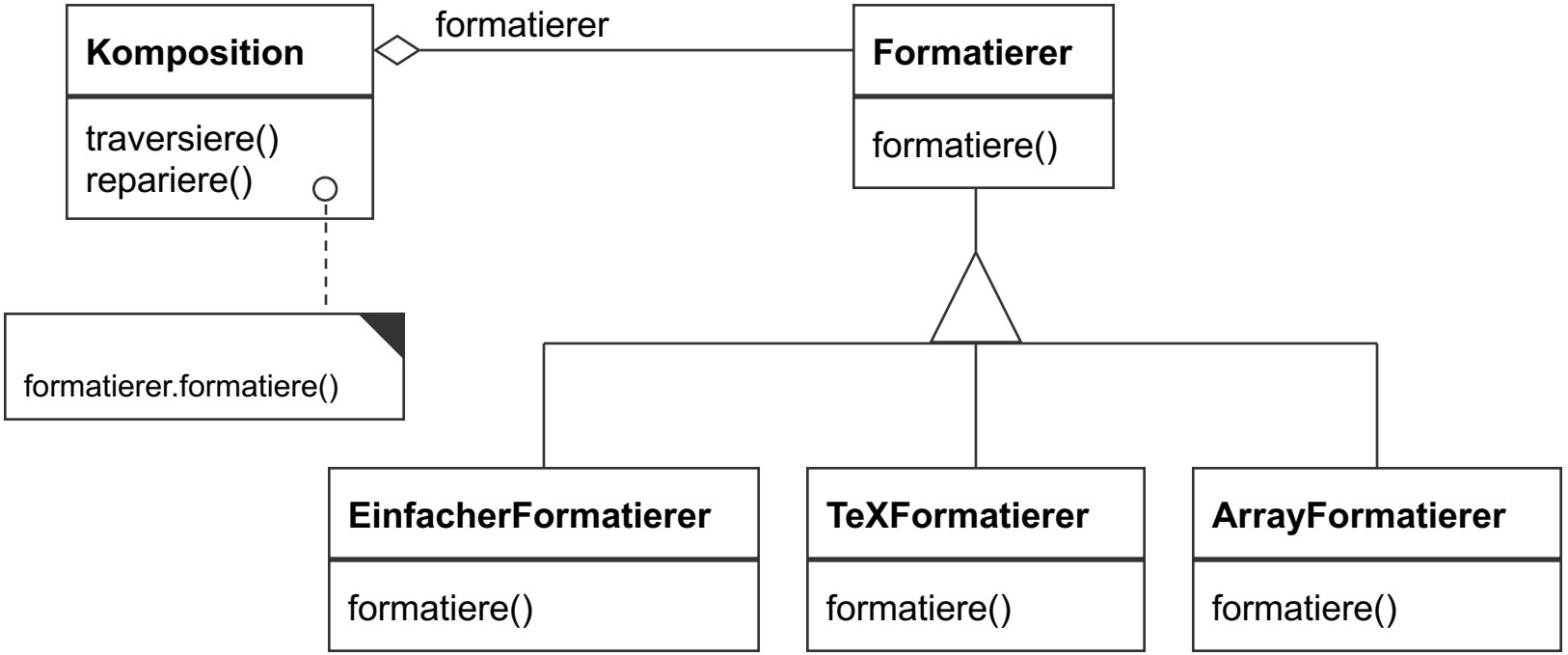
- Java Swing ist eine klassische Kompositium-Implementierung.

- Erläutern Sie, welche Klassen in Swing welche Aufgabe im Kompositum erfüllen. Implementieren Sie dazu ein Beispiel.

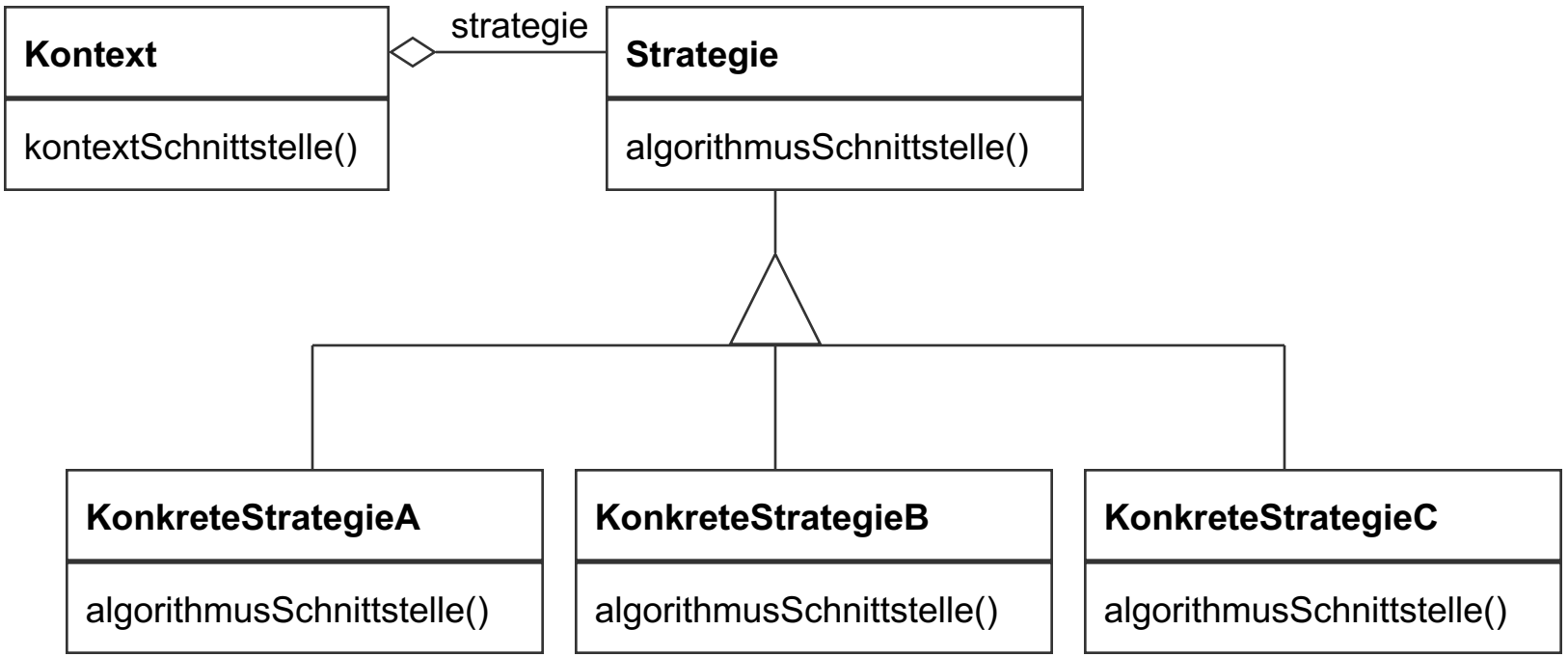
Strategie (Strategy)

- Zweck
  - Definiere eine Familie von Algorithmen, kapsele sie und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von nutzenden Klienten zu variieren.
- Auch bekannt als
  - Policy
- Motivation
  - Manchmal müssen Algorithmen, abhängig von der notwendigen Performanz, der Anzahl oder des Typs der Daten, variiert werden.

Beispiel für Strategie: Kapselung von Zeilenumbrechalgorithmen in Klassen



Struktur für Strategie



Strategie: Anwendbarkeit

- Wenn sich viele verwandte Klassen nur in ihrem Verhalten unterscheiden. Strategieobjekte bieten die Möglichkeit, eine Klasse mit einer von mehreren möglichen Verhaltensweisen zu konfigurieren.
- Wenn unterschiedliche Varianten eines Algorithmus benötigt werden.
- Wenn ein Algorithmus Datenstrukturen verwendet, die Klienten nicht bekannt sein sollen.
- Wenn eine Klasse unterschiedliche Verhaltensweisen definiert und diese als mehrfache Bedingungsanweisungen in ihren Operationen erscheinen.

Strategie: Anwendbarkeit

- Alternativ zur Ableitung der Klasse Strategie kann man auch die Klasse Kontext ableiten, um verschiedene Verhaltensmuster zu implementieren.
  - Das Ergebnis sind viele Klassen, die sich nur im Verhalten unterscheiden, welches für jede Klasse fest ist. Das Strategie-Muster erlaubt demgegenüber auch eine dynamische Veränderung des Verhaltens.

Entwurfsmuster - Kategorien

- Entkoppelungs-Muster
- Varianten-Muster
- Zustandshandhabungs-Muster
- Steuerungs-Muster
- Virtuelle Maschinen
- Bequemlichkeits-Muster

Entwurfsmuster-Kategorien nach Verwendungszweck

- Entkopplungs-Muster
  - Entkopplungs-Muster teilen ein System in mehrere Einheiten, so dass einzelne Einheiten unabhängig voneinander erstellt, verändert, ausgetauscht und wiederverwendet werden können. Der Vorteil von Entkopplung ist, dass ein System durch lokale Änderungen verbessert, angepasst und erweitert werden kann, ohne das ganze System zu modifizieren.
  - Mehrere der Entkopplungsmuster enthalten ein Kopplungsglied, das entkoppelte Einheiten über eine Schnittstelle kommunizieren lässt. Diese Kopplungsglieder sind auch für das Koppeln unabhängig erstellter Einheiten brauchbar.

Entwurfsmuster-Kategorien nach Verwendungszweck

- Varianten-Muster
  - In Mustern dieser Gruppe werden Gemeinsamkeiten von verwandten Einheiten aus ihnen herausgezogen und an einer einzigen Stelle beschrieben. Aufgrund ihrer Gemeinsamkeiten können unterschiedliche Komponenten im gleichen Programm danach einheitlich verwendet werden, und Wiederholungen desselben Codes werden vermieden.

- Zustandshandhabungs-Muster
  - Die Muster dieser Kategorie bearbeiten den Zustand von Objekten, unabhängig von deren Zweck.

Entwurfsmuster-Kategorien nach Verwendungszweck

- Steuerungs-Muster
  - Muster dieser Gruppe steuern den Kontrollfluss. Sie bewirken, dass zur richtigen Zeit die richtigen Methoden aufgerufen werden.

- Virtuelle Maschinen
  - Virtuelle Maschinen erhalten Daten und ein Programm als Eingabe und führen das Programm selbständig an den Daten aus. Virtuelle Maschinen sind in Software, nicht in Hardware implementiert.

Entwurfsmuster-Kategorien nach Verwendungszweck

- Bequemlichkeits-Muster
  - Muster dieser Gruppe vereinfachen die Verwendung von Klassen, Methoden oder Subsystemen durch Bereitstellung geeigneter Schnittstellen

Entkoppelungs-Muster

- Abstrakter Datentyp
- Modul
- Datenablage (Repository)
- Iterator
- Schichtenarchitektur
- Vermittler (Mediator)
- Brücke
- Adapter
- Stellvertreter (Proxy)
- Fließband
- Ereigniskanal
- Rahmenprogramm (Framework)

Entkopplungs-Muster: Abstrakter Datentyp

- Ein abstrakter Datentyp (ADT) definiert einen neuen Datentyp zusammen mit geeigneten Operationen. Die Implementierung dieses Datentyps ist wie beim Modul hinter einer abstrakten (änderungs-unempfindlichen) Schnittstelle verborgen.

- Unterschiede zum Modul:
  - Modul ist i.d.R. eine größere Einheit. Es kann z.B. mehrere voneinander abhängige ADTs zusammenfassen.
  - Von einem ADT kann man beliebig viele Exemplare anlegen; von einem Modul gibt es in jedem Programm nur ein Exemplar.

Entkopplungs-Muster: Modul

- Ein Modul ist eine Menge von Programmkomponenten, die gemeinsam entworfen und verändert werden; diese Komponenten werden hinter einer änderungs-unempfindlichen Schnittstelle verborgen (“Geheimnisprinzip”).
- Ziel der Modularisierung ist eine Entkopplung: modul-interne Komponenten können verändert oder ersetzt werden, ohne die Benutzer des Moduls anpassen zu müssen.
- Um effektiv zu sein, müssen die möglichen Änderungen vorausgesehen und in die Modulstruktur und Schnittstellen hineingeplant werden.

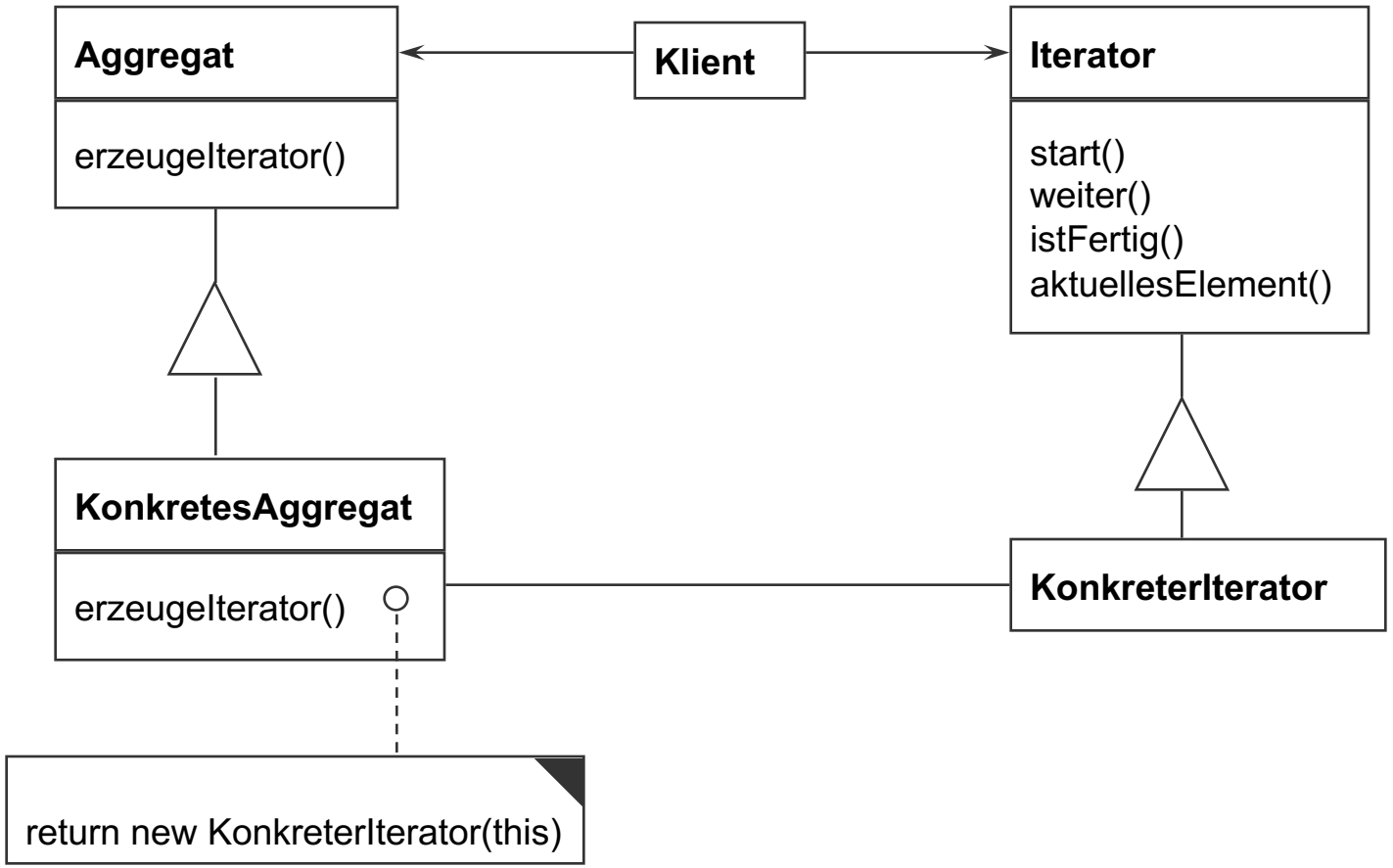
Entkopplungs-Muster: Modul

- Kandidaten für Veränderung/Verbergung:
  - Datenstrukturen und Operationen, Größe der Datenstrukturen, Optimierungen
  - maschinennahe Details
  - betriebsystemnahe Details
  - Ein- und Ausgabeformate
  - Benutzerschnittstellen
  - Dialog- und Fehlermeldungstexte, Maßeinheiten (Internationalisierung)
  - Reihenfolge der Verarbeitung, Vorverarbeitung, inkrementelle Verarbeitung
  - Zwischenpufferung

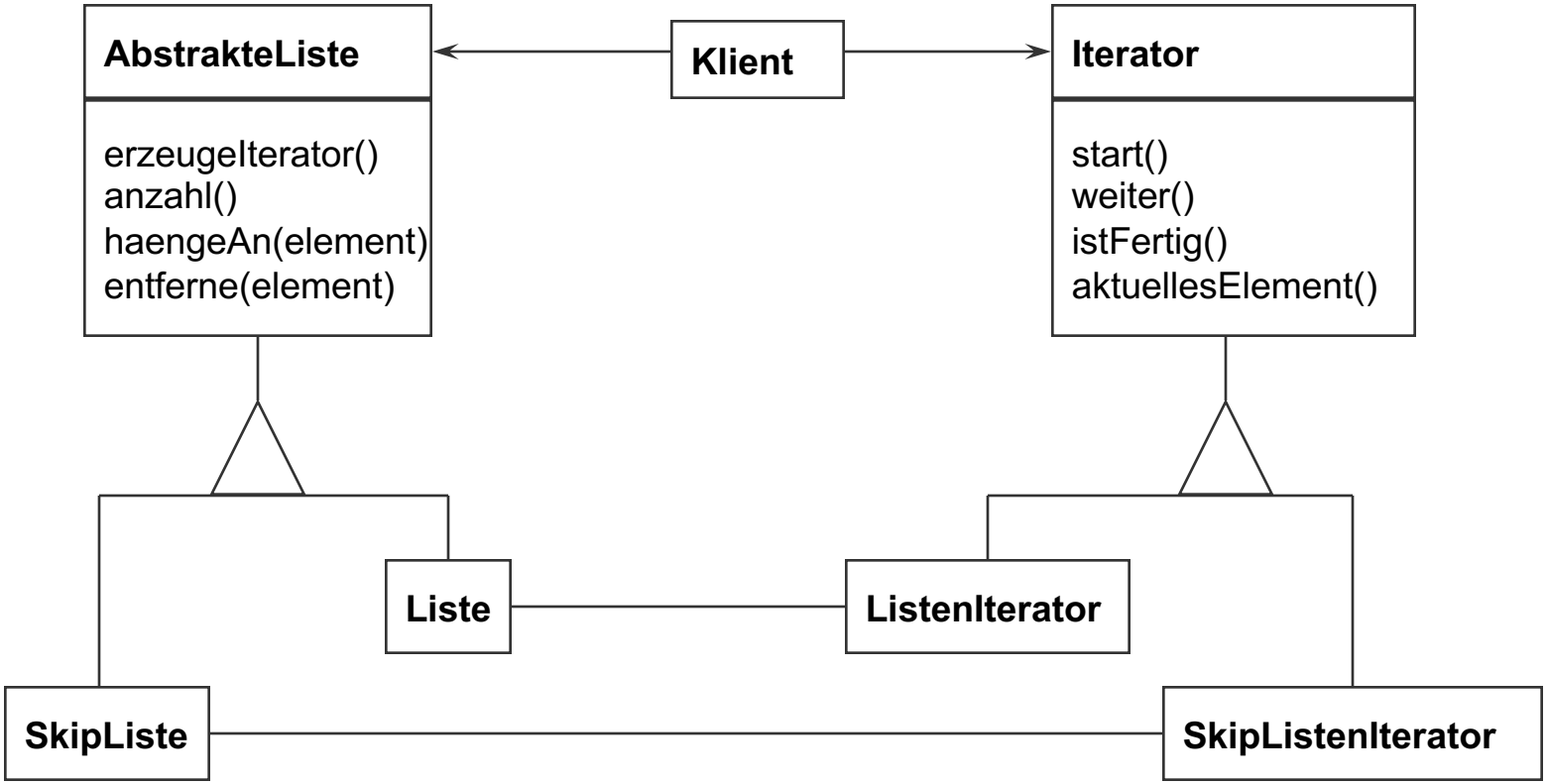
Entkopplungsmuster: Iterator (Iterator)

- Zweck
  - Ermögliche den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen.

Entkopplungs-Muster: Struktur des Iterators



Entkopplungs-Muster: Beispiel Iterator für eine Liste



Entkopplungs-Muster: Iterator: Anwendbarkeit

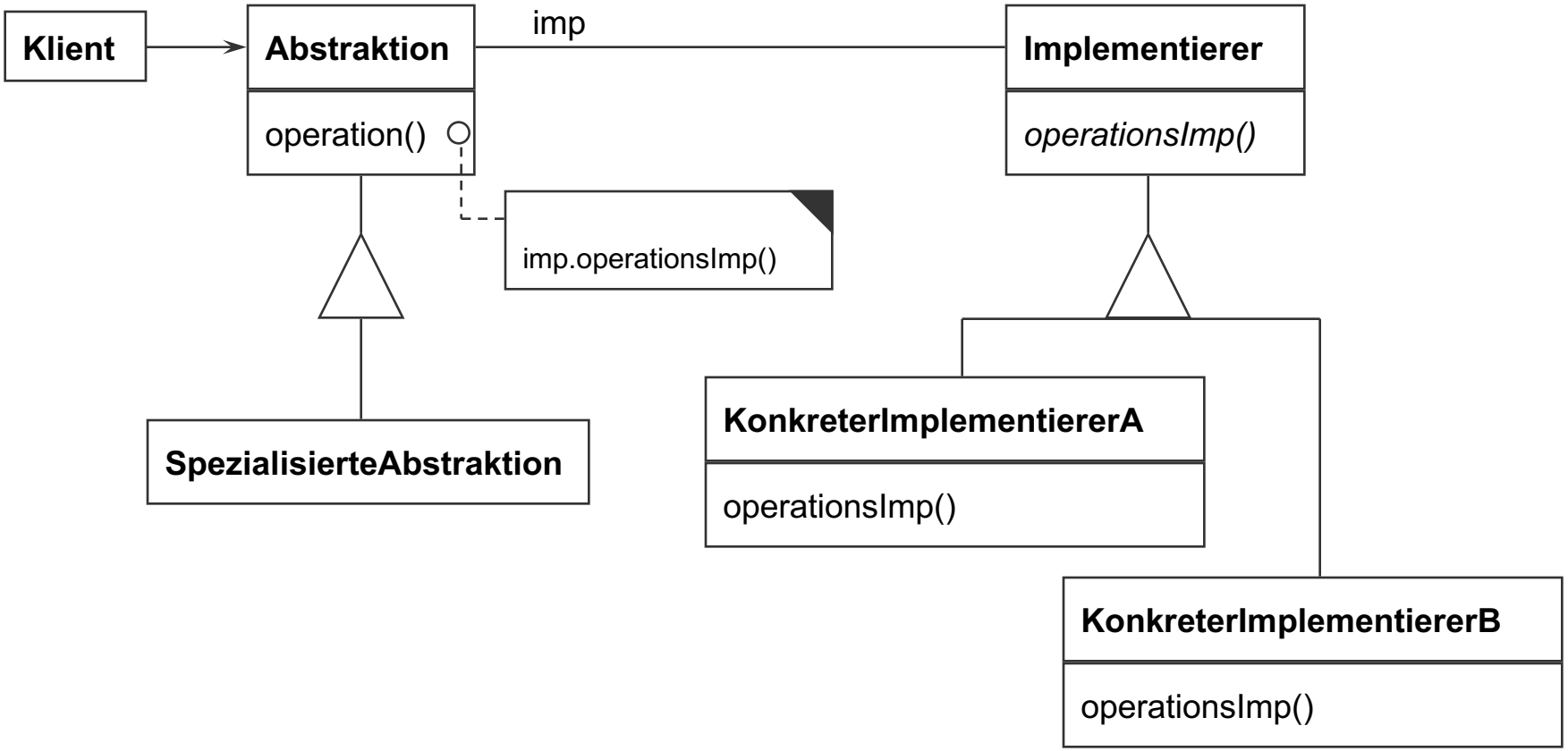
- Um den Zugriff auf den Inhalt eines zusammengesetzten Objekts zu ermöglichen, ohne dabei seine interne Struktur offenzulegen.
- Um mehrfaches gleichzeitiges Durchlaufen von zusammengesetzten Objekten zu ermöglichen.
- Um eine einheitliche Schnittstelle zum Durchlaufen unterschiedlicher zusammengesetzter Strukturen anzubieten (das heißt, um polymorphe Iteration zu ermöglichen).

Entkopplungs-Muster: Brücke (Bridge)

- Zweck
  - Entkopple eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.

- Auch bekannt als
  - Handle/Body

Entkopplungs-Muster: Struktur der Brücke



Entkopplungs-Muster: Brücke: Anwendbarkeit

- Wenn eine dauerhafte Verbindung zwischen Abstraktion und Implementierung vermieden werden soll.
- Wenn sowohl Abstraktion als auch Implementierungen durch Unterklassenbildung erweiterbar sein soll.
- Wenn Änderungen in der Implementierung einer Abstraktion keine Auswirkung auf Klienten haben sollen.
- Wenn die Implementierung einer Abstraktion vollständig vom Klienten versteckt werden soll.
- Wenn eine starke Vergrößerung der Anzahl der Klassen vermieden werden soll.
- Wenn eine Implementierung von mehreren Objekten aus gemeinsam benutzt werden soll.

Varianten-Muster

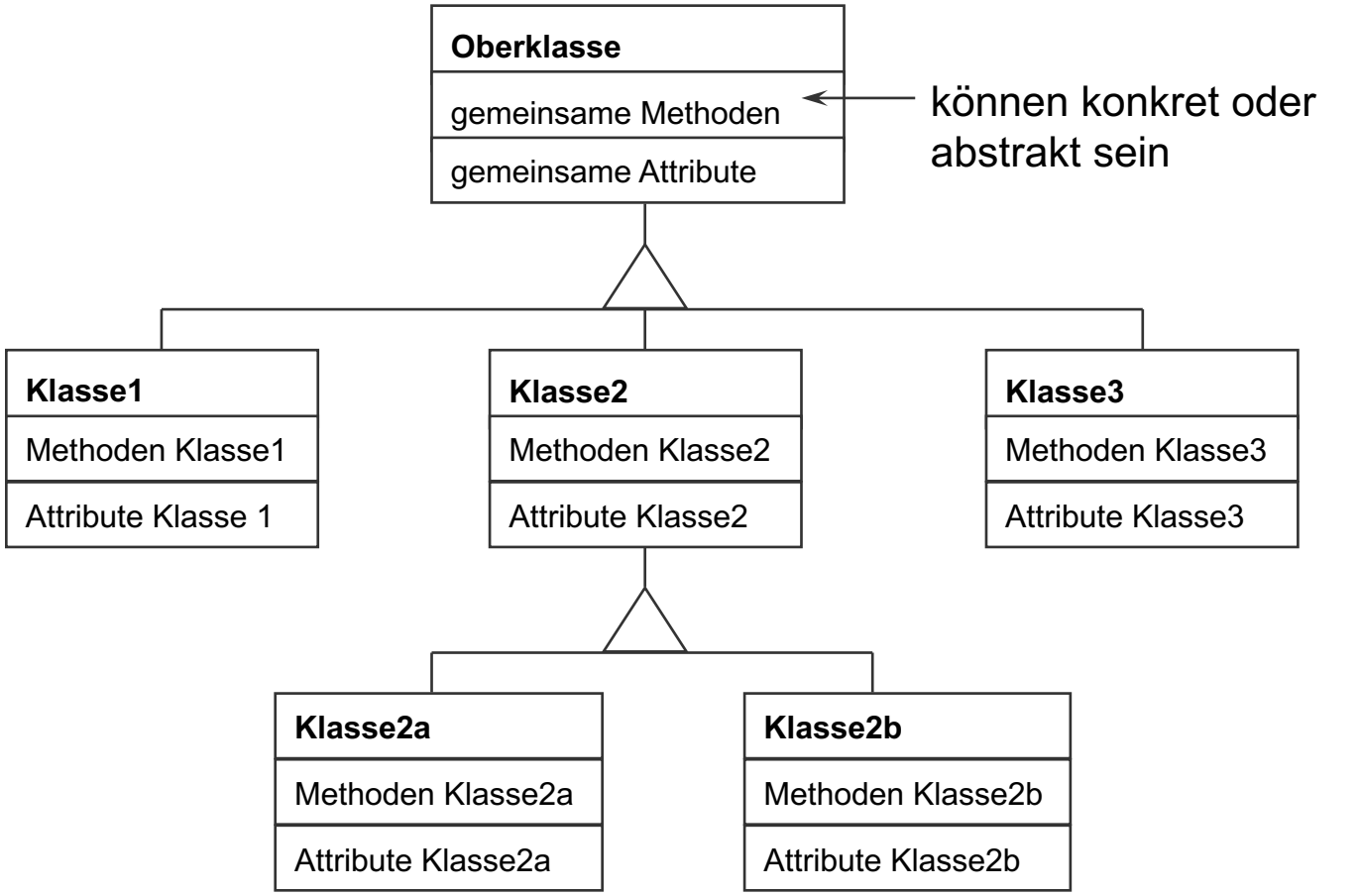
- Gemeinsamkeiten von verwandten Einheiten werden aus ihnen herausgezogen und an einer einzigen Stelle beschrieben. Aufgrund ihrer Gemeinsamkeiten können unterschiedliche Komponenten im gleichen Programm einheitlich verwendet werden, und Code-Wiederholungen werden vermieden.

- Oberklasse
- Besucher
- Schablonenmethode
- Fabrikmethode
- Erbauer
- Abstrakte Fabrik

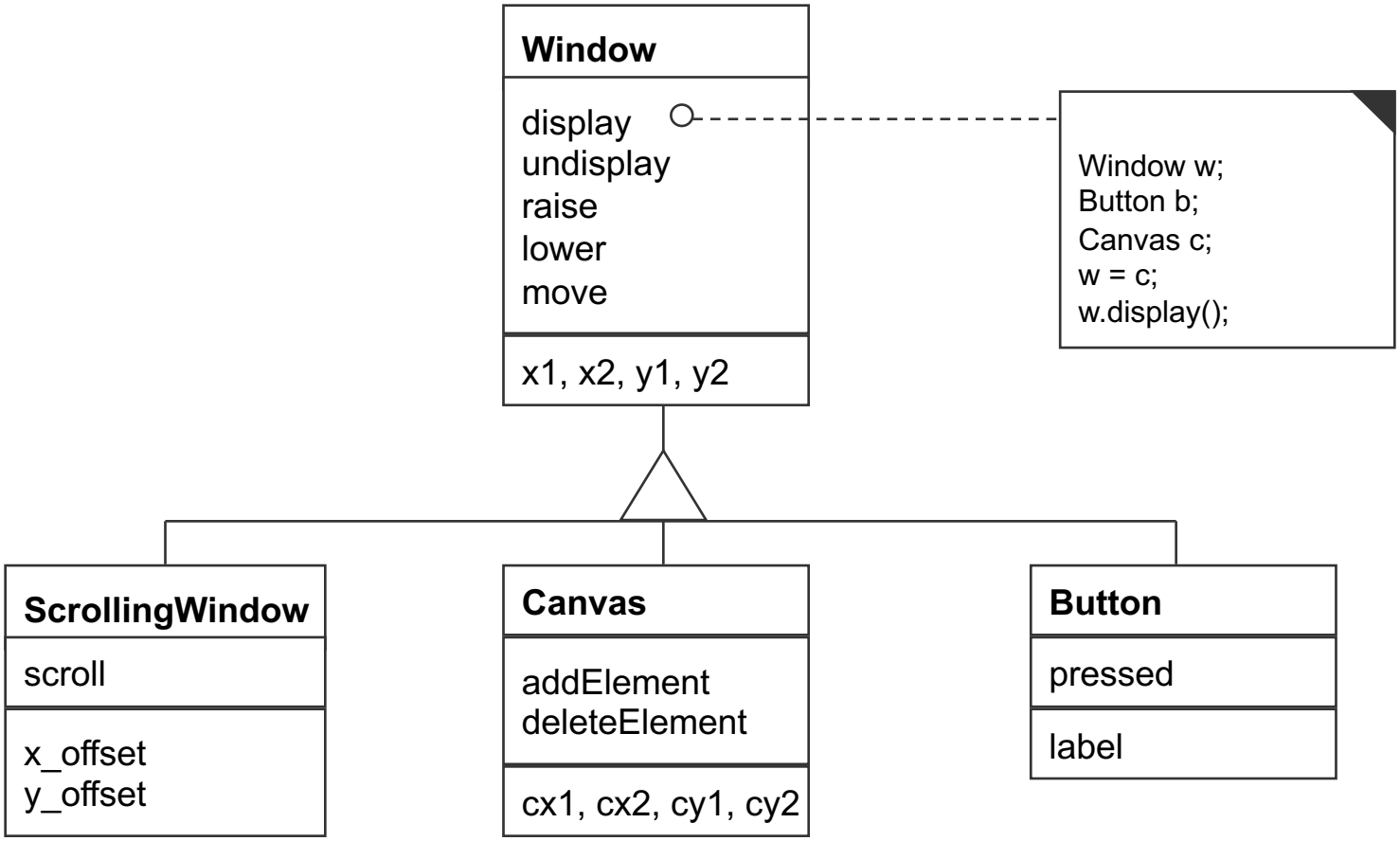
Varianten-Muster: Oberklasse

- Zweck
  - Einheitliche Behandlung von Objekten, die unterschiedlichen Klassen angehören, aber gemeinsame Attribute oder Methoden besitzen.

Varianten-Muster: Struktur der Oberklasse



Varianten-Muster: Beispiel einer Oberklasse



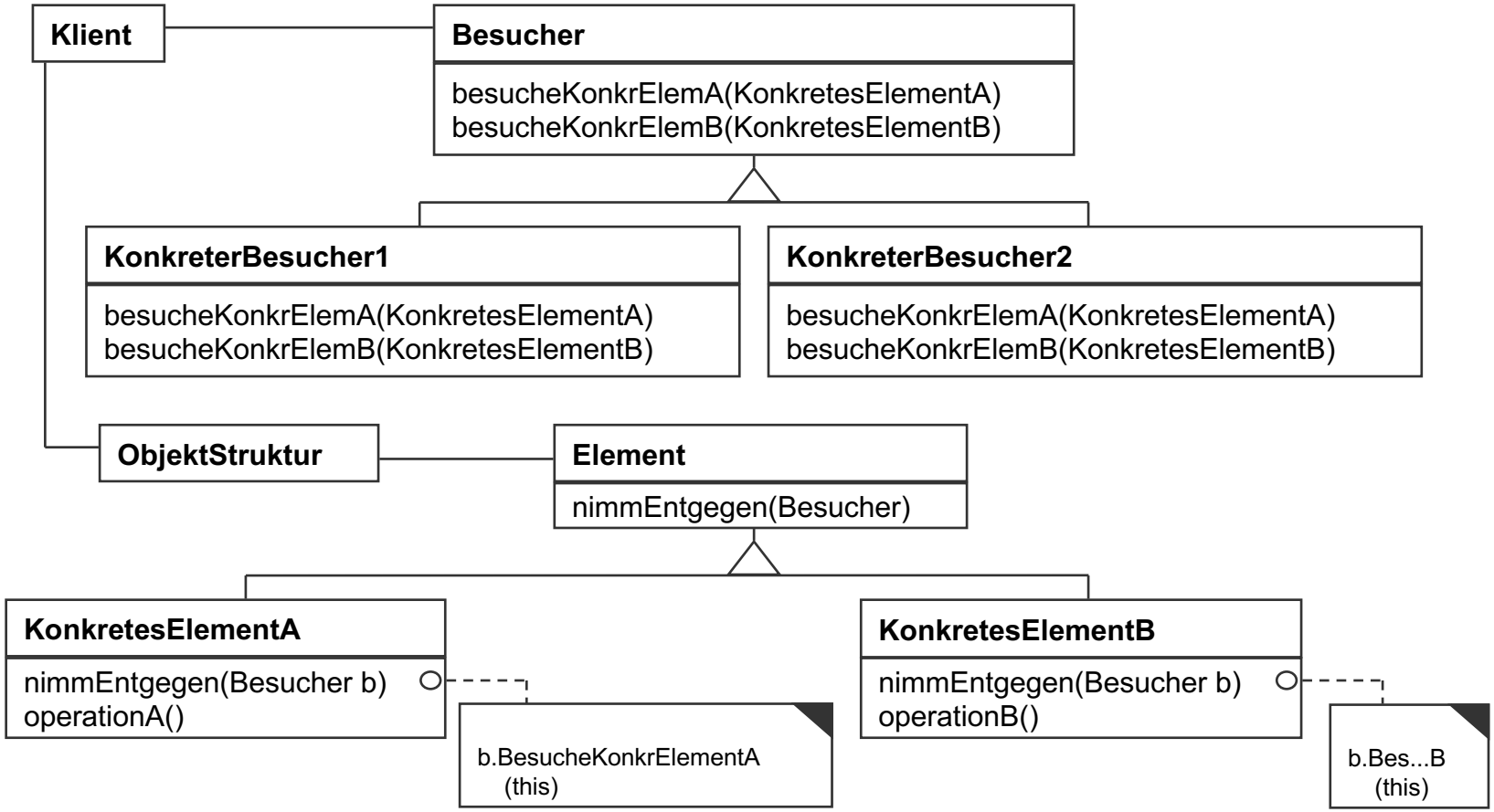
Varianten-Muster: Oberklasse: Anwendbarkeit

- Wenn Objekte verschiedener Klassen gemeinsame Attribute, Methoden oder Schnittstellen haben.
- Wenn Objekte verschiedener Klassen einheitlich in einem Programm behandelt werden sollen.
- Wenn es möglich sein soll, weitere Klassen hinzuzufügen, ohne den bestehenden Quelltext zu verändern.

Varianten-Muster: Besucher (Visitor)

- Zweck
  - Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

Varianten-Muster: Struktur des Besucher



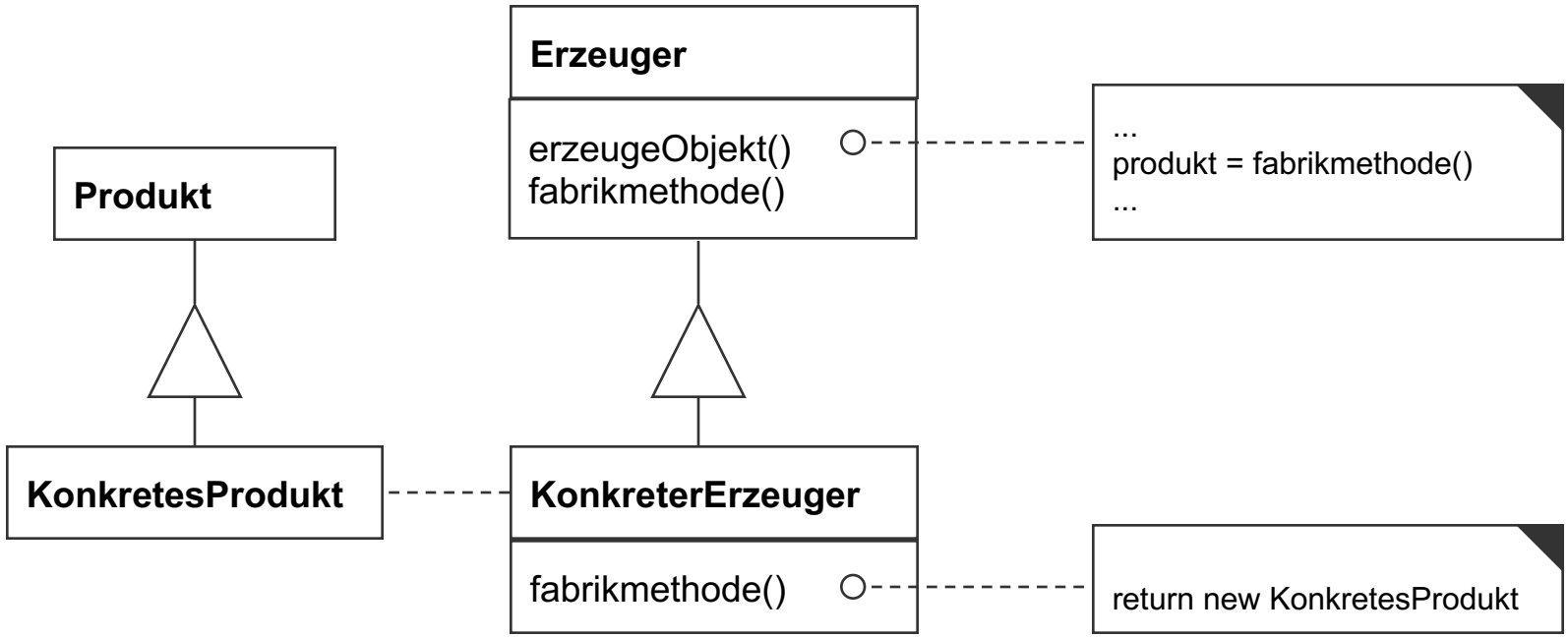
Varianten-Muster: Besucher: Anwendbarkeit

- Wenn eine Objektstruktur viele Klassen von Objekten mit unterschiedlichen Schnittstellen enthält und Operationen auf diesen Objekten ausgeführt werden sollen, die von ihren konkreten Klassen abhängen.
- Wenn viele unterschiedliche und nicht miteinander verwandte Operationen auf den Objekten einer Objektstruktur ausgeführt werden müssen und diese Klassen nicht mit diesen Operation „verschmutzt" werden sollen.
- Wenn sich die Klassen, die eine Objektstruktur definieren, praktisch nie ändern, aber häufig neue Operationen für die Struktur definiert werden.

Varianten-Muster: Fabrikmethode (Factory Method)

- Zweck
  - Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.

Varianten-Muster: Struktur der Fabrikmethode



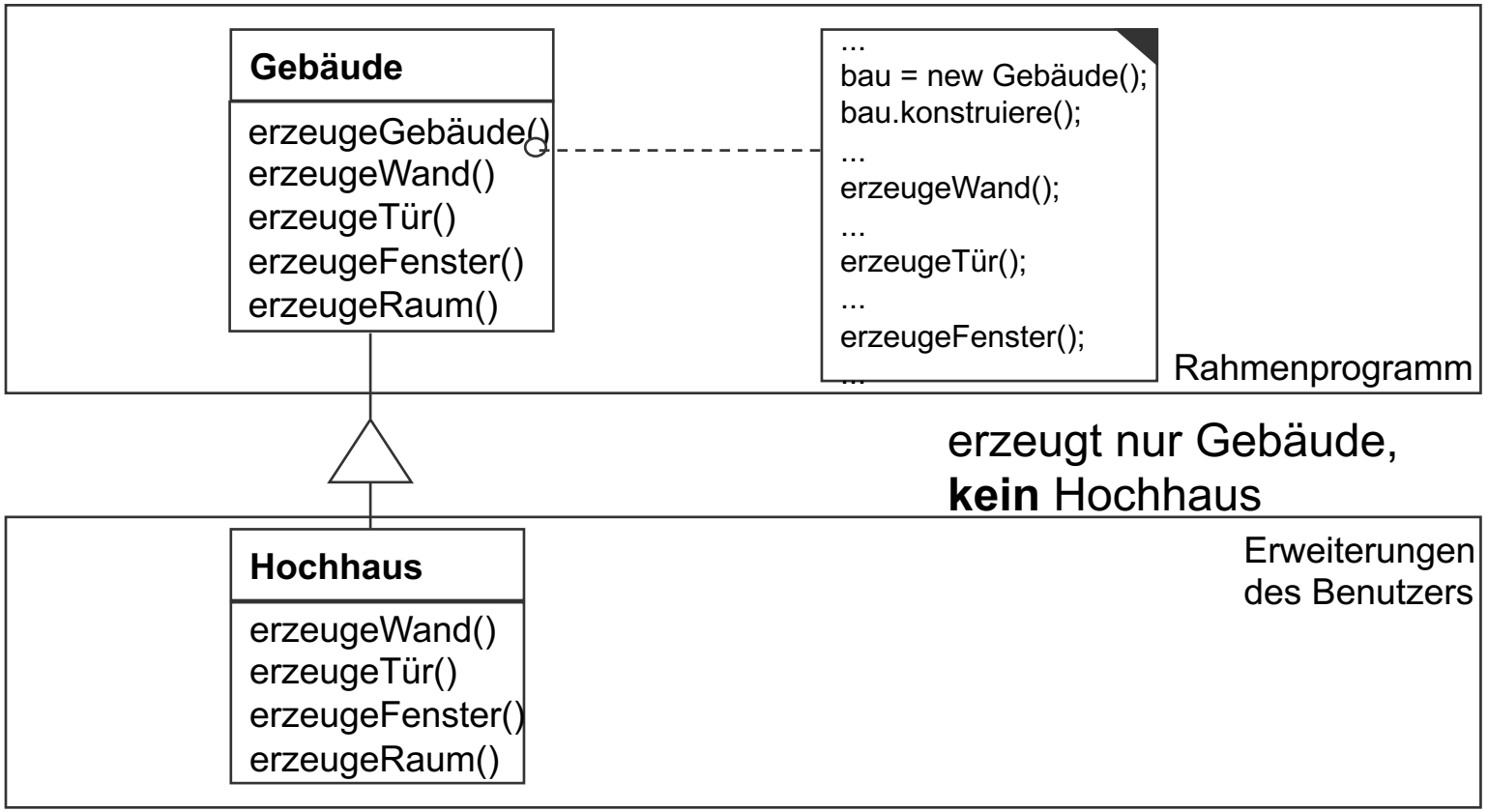
Varianten-Muster: Beispiel einer Fabrikmethode

- Angenommen, „Architekt“ sei ein Rahmenprogramm zum Planen von Gebäuden. Es enthält die Klasse Gebäude, die eine Methode konstruiere enthält. Diese Methode ist ein interaktives Programm, um einen Plan für ein Gebäude zu erstellen.

- Dem Benutzer des Gebäudes ist es gestattet, Unterklassen von Gebäude anzulegen. Diese Unterklassen müssen Implementierungen für die abstrakten Methoden erzeugeWand, erzeugeRaum, erzeugeTür, erzeugeFenster enthalten.

- Problem: Wie kann der Benutzer dem Rahmenprogramm mitteilen, wie seine Unterklasse heißt?

Varianten-Muster: Beispiel einer Fabrikmethode



Varianten-Muster: Beispiel einer Fabrikmethode Variante 1

- Übergib den Klassennamen Hochhaus als Zeichenkette und wandle ihn in die entsprechende Klasse um.
  - in Java: Class.forName(String name)

```
konstruiere() {
    String KlassenName = holeKlassenname()
    // holeKlassenName liest z.B. eine Konfigurationsdatei
    // oder nutzt einen Aufrufparameter
    bau = (Gebäude) Class.ForName(
      KlassenName).newInstance();
    ...
}
```

Varianten-Muster: Beispiel einer Fabrikmethode Variante 2

- Gib Unterklasse von Gebäude im Rahmenprogramm vor, und parametrisiere konstruiere() entsprechend.

```
konstruiere (Gebäudetyp t) {
    switch (t) {
      case HOCHHAUS:  bau = (Gebäude) new Hochhaus(); break;
      case BANK:          bau = (Gebäude) new Bank(); break;
      case WOHNHAUS: bau = (Gebäude) new Wohnhaus(); break;
    }
    ...
}
```

- Nachteil:
  - Existenz und Anzahl der Unterklassen muss bekannt sein

Varianten-Muster: Fabrikmethode: Anwendbarkeit

- Wenn eine Klasse die Klasse von Objekten, die sie erzeugen muss, nicht im voraus kennen kann.
- Wenn eine Klasse möchte, dass ihre Unterklasse die von ihr zu erzeugenden Objekte festlegen.
- Wenn Klassen Zuständigkeiten an eine von mehreren Hilfsunterklassen delegieren sollen und das Wissen, an welche Hilfsunterklasse die Zuständigkeit delegiert wird, lokalisiert werden soll.

Zustandshandhabungs-Muster

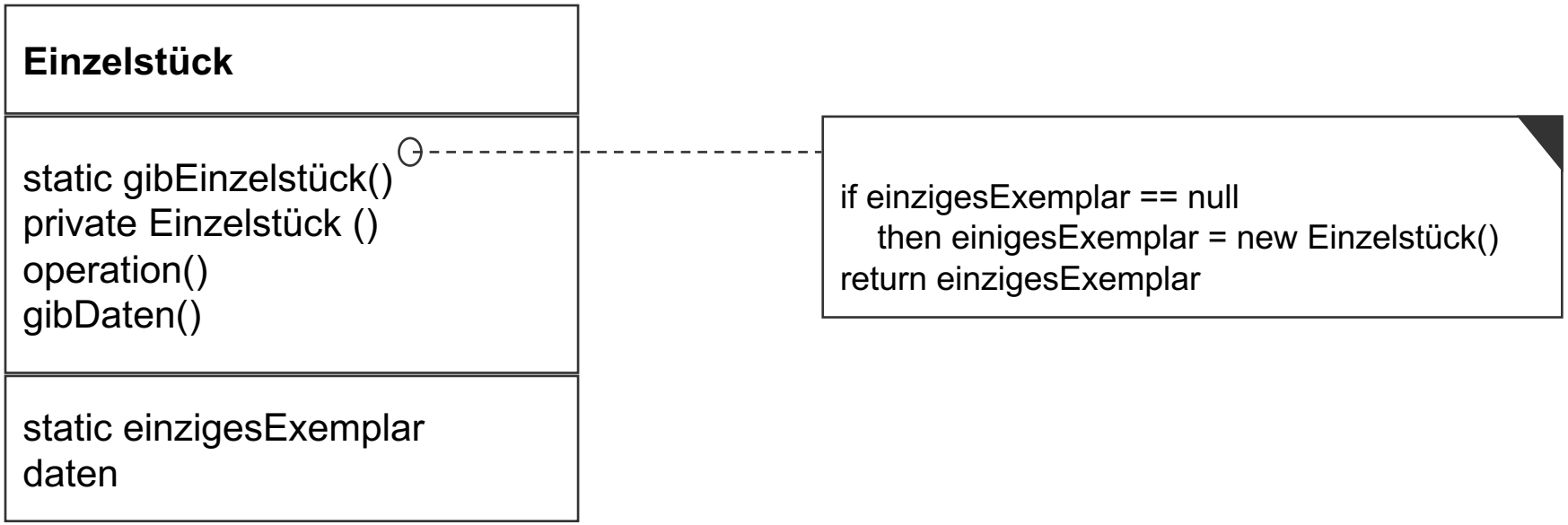
- Memento
- Prototyp
- Fliegengewicht
- Einzelstück

Zustandshandhabungs-Muster: Einzelstück (Singleton)

- Zweck
  - Sichere zu, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.

- Motivation
  - Die Klasse ist selbst für die Verwaltung ihres einzigen Exemplars zuständig. Die Klasse kann durch Abfangen von Befehlen zur Erzeugung neuer Objekte sicherstellen, dass kein weiteres Exemplar erzeugt wird.

Zustandshandhabungs-Muster: Struktur des Einzelstücks



Zustandshandhabungs-Muster: Einzelstück: Anwendbarkeit

- Wenn es von einer Klasse nur eine einzige Instanz geben darf und diese Instanz den Klienten an einer bekannten Stelle zugänglich gemacht werden soll.
- Wenn die einzige Instanz durch Unterklassenbildung erweiterbar sein soll und die Klienten ohne Veränderung ihres Quelltextes diese nutzen sollen.
- Wenn es schwierig oder unmöglich ist, festzustellen, welcher Teil der Anwendung die erste Instanz erzeugt.

Steuerungs-Muster

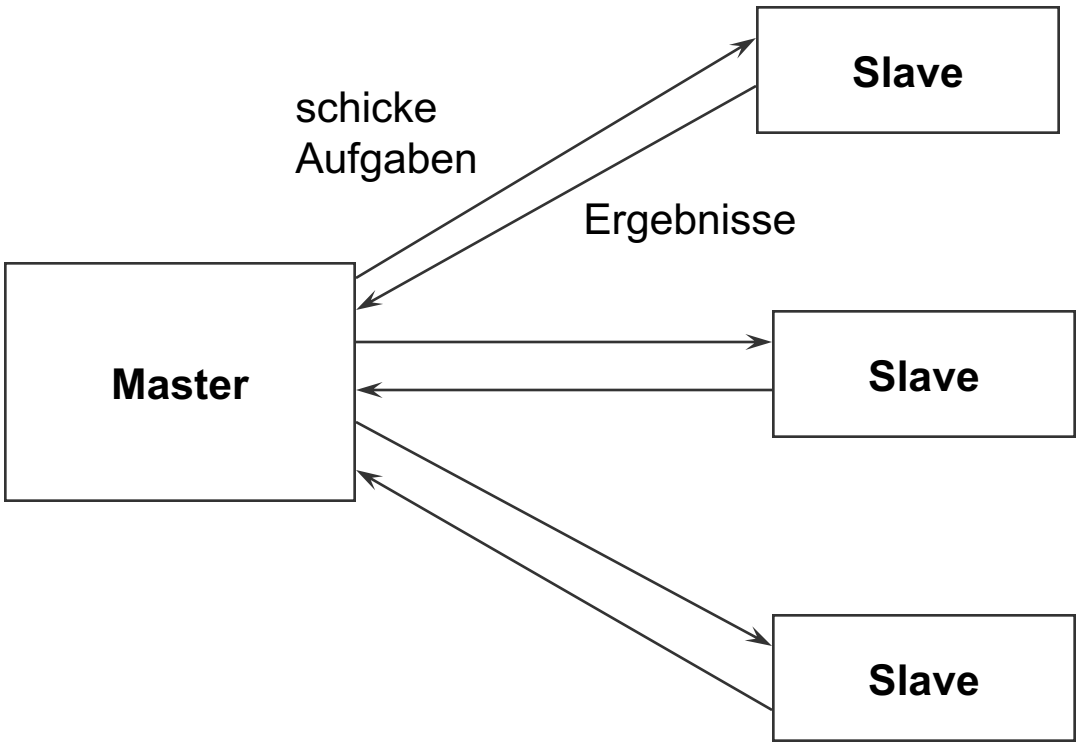
- Tafel
- Befehl
- Zuständigkeitskette
- Master/Slave
- Prozesssteuerung
- System ohne Rückkoppelung
- System mit Rückkoppelung
- Regelung mit Rückführung
- Regelung mit Störgrößenaufschaltung

Steuerungs-Muster: Master/Slave

- Zweck
  - Das Master/Slave-Muster unterstützt Fehlertoleranz und parallele Berechnung. Eine Master-Komponente (Auftraggeber) verteilt die Arbeit an identische Slave-Komponenten (Arbeiter, Auftragnehmer) und berechnet das Endergebnis aus den Teilergebnissen, welche die Arbeiter zurückliefern.

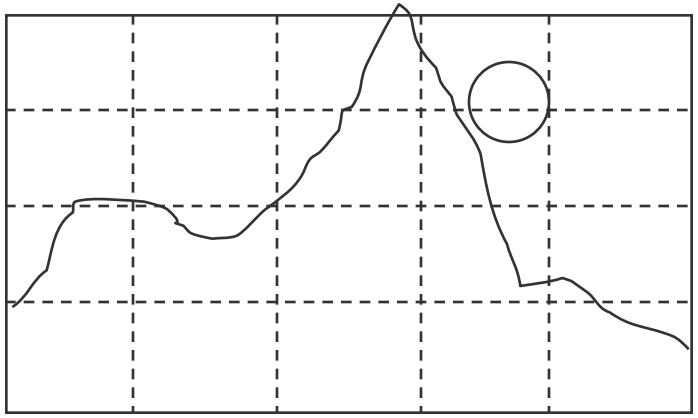
- Auch bekannt als
  - Master/Workers, Auftraggeber/Auftragnehmer

Steuerungs-Muster: Struktur von Master/Slave



Steuerungs-Muster: Beispiel von Master/Slave

Parallele Berechnung eines 3D Bildes



Der Master gibt rechteckige Teile des Bildes zur Berechnung an seine Arbeiter weiter und setzt das gesamte Bild aus den einzelnen Teilen zusammen.

Steuerungs-Muster: Master/Slave: Anwendbarkeit

- Wenn es mehrere Aufgaben gibt, die unabhängig voneinander bearbeitet werden können.
- Wenn mehrere Prozessoren zur parallelen Verarbeitung zur Verfügung stehen.
- Wenn die Belastung der Arbeiter ausgeglichen werden soll.

Virtuelle Maschinen

- Interpretierer
- Regelbasierter Interpretierer

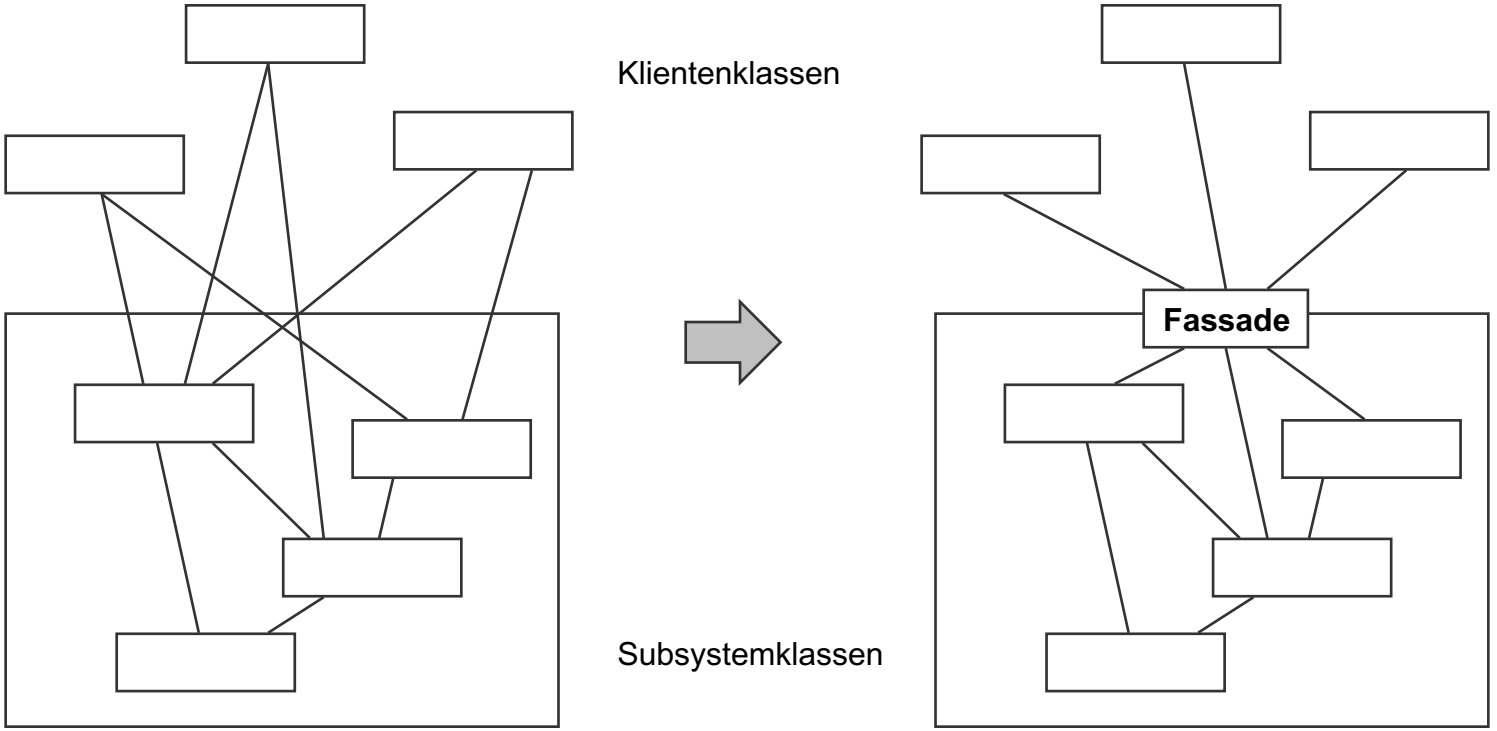
Bequemlichkeits-Muster

- Bequemlichkeits-Methode
- Bequemlichkeits-Klasse
- Fassade
- Null-Objekt

Bequemlichkeits-Muster: Fassade (Facade)

- Zweck
  - Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.

Bequemlichkeits-Muster: Fassade



Bequemlichkeits-Muster: Fassade: Anwendbarkeit

- Wenn eine einfache Schnittstelle zu einem komplexen Subsystem angeboten werden soll. Eine Fassade kann eine einfache voreingestellte Sicht auf das Subsystem bieten, die den meisten Klienten genügt.
- Wenn es viele Abhängigkeiten zwischen den Klienten und den Implementierungsklassen einer Abstraktion gibt. Die Einführung einer Fassade entkoppelt die Subsysteme von Klienten und anderen Subsystemen.
- Wenn Subsysteme in Schichten aufgeteilt werden sollen. Man verwendet eine Fassade, um einen Eintrittspunkt zu jeder Subsystemschicht zu definieren.

Bequemlichkeits-Muster: Null-Objekt

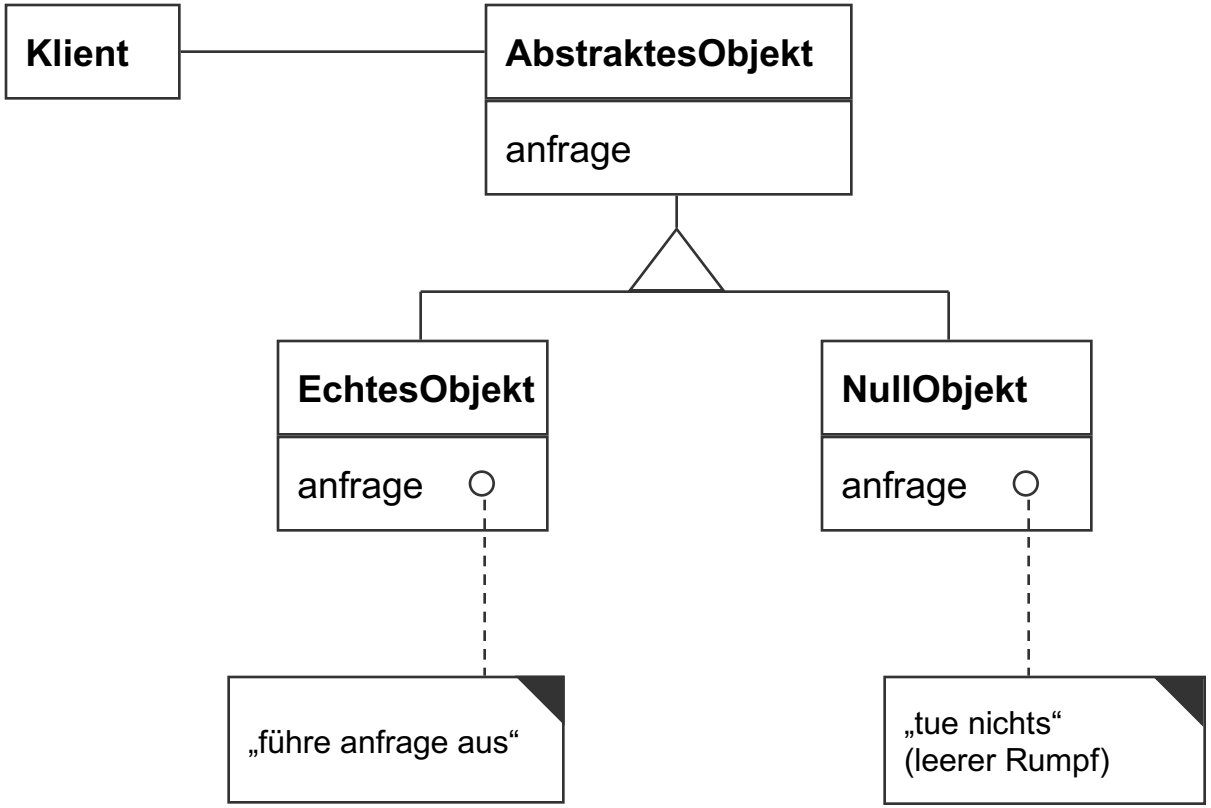
- Zweck
  - Stelle einen Stellvertreter zur Verfügung, der die gleiche Schnittstelle bietet, aber nichts tut. Das Null-Objekt kapselt die Implementierungs-Entscheidungen (wie genau es „nichts tut“) und versteckt diese Details vor seinen Mitarbeitern.

- Motivation
  - Es wird verhindert, dass der Code mit Tests gegen Null-Werte verschmutzt wird, wie:

```
if (thisCall.callingParty != NULL)
  thisCall.callingParty.action()
```

- Bemerkung
  - Zu Beginn einer Implementierung kann ein System vollständig aus Null-Objekten zusammengesetzt werden.

Bequemlichkeits-Muster: Struktur des Null-Objektes



Bequemlichkeits-Muster: Null-Objekt: Anwendbarkeit

- Wenn ein Objekt Mitarbeiter benötigt und einer oder mehrere von ihnen nichts tun sollen.
- Wenn Klienten sich nicht um den Unterschied zwischen einem echten Mitarbeiter und einem der nichts tut kümmern sollen.
- Wenn das „tue nichts“-Verhalten von verschiedenen Klienten wiederverwendet werden soll.

Zusammenfassung

- Entwurfsmuster
  - bieten ein Vokabular zur effizienten Kommunikation zwischen Teammitgliedern,
  - erfassen den „Stand der Kunst“,
  - dokumentieren Entwürfe,
  - machen aus Anfängern gute Entwerfer,
  - machen gute Entwerfer noch besser,
  - verbessern Qualität und Produktivität,
  - sind schwierig zu finden und zu beschreiben.

Selbstkontrolle

1. Erläutern Sie den Unterschied zwischen den beiden Varianten eines Kompositums.
2. Erläutern Sie, wie ein Änderungsmanager in Beobachter funktioniert?
3. Was sind Bequemlichkeitsmuster?
4. Erläutern Sie den Sinn eines Nullobjekts in der objektorientierten Softwareentwicklung.
5. Skizzieren Sie das Klassenmodell eines Singletons.

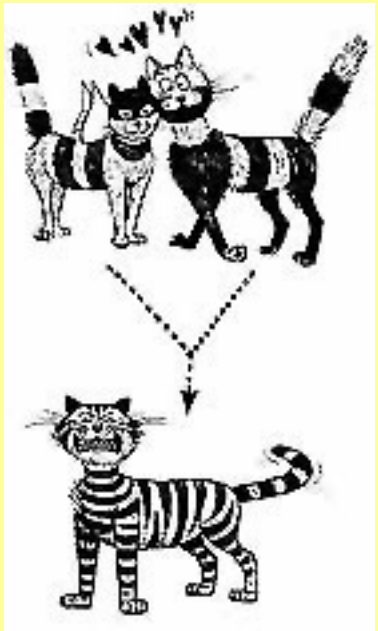
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de

Software Engineering 2
Softwaremetriken



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Warum Software messen?

- Kontrolle der Softwareentwicklung
  - während der Entwicklung
  - nach Projektabschluss

- typische Fragestellungen
  - wie weit ist die Entwicklung verglichen mit dem Zeitplan?
  - wie komplex ist die Software?
  - rechtfertigt die Komplexität den Zeitaufwand?
  - wurde die effizienteste Lösung implementiert?

- Was kann überhaupt gemessen werden?

Was ist eine Metrik?

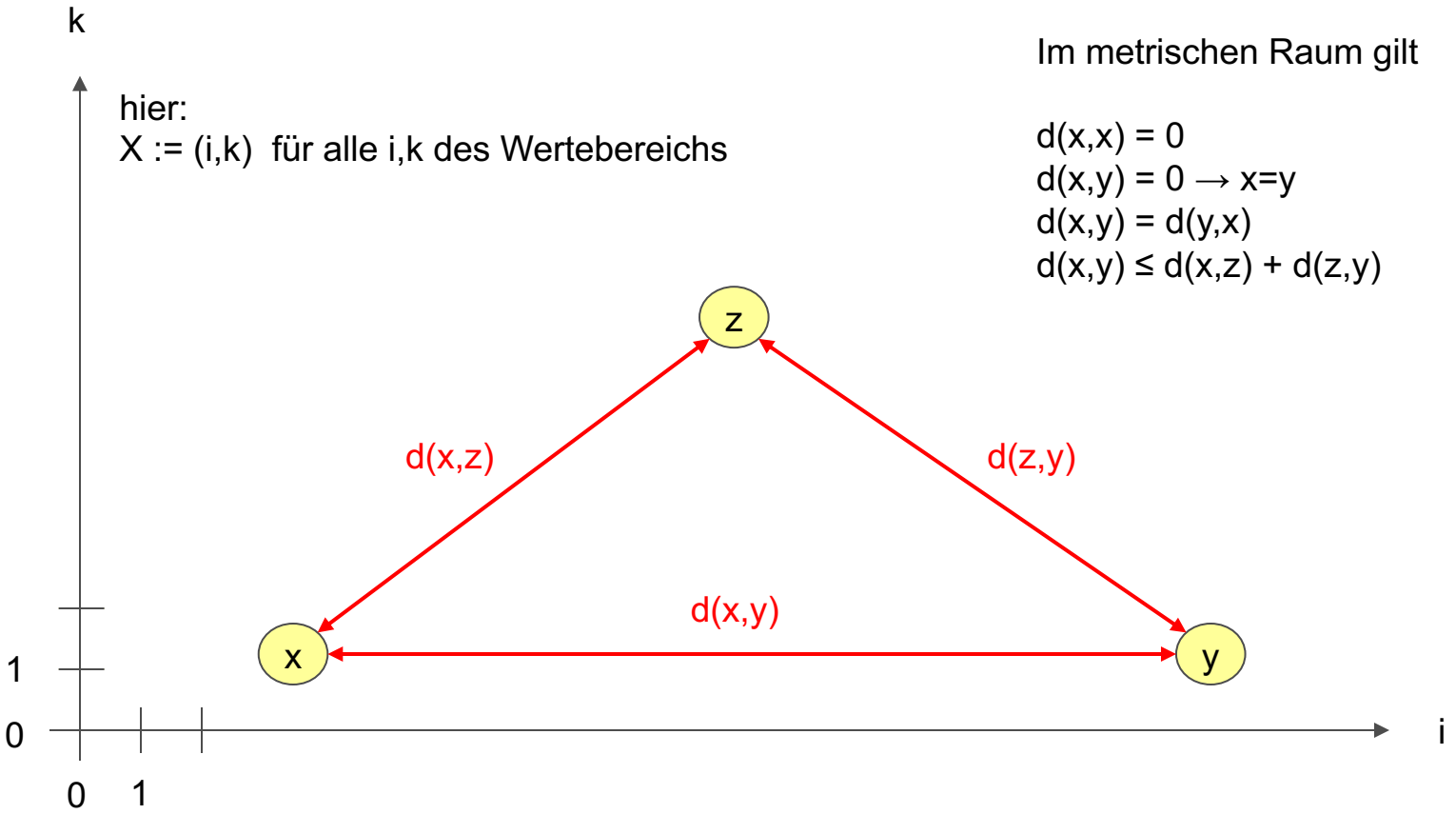
- Metrik
  - Eine Metrik ist eine mathematische Funktion, die je zwei Elementen eines Raums einen nicht negativen skalaren Wert zuordnet, der als Abstand der beiden Elemente von einander aufgefasst werden kann.
  - in der Mathematik auch als Abstandsfunktion bezeichnet.

- Metrischer Raum
  - Ein Raum ist eine Menge, deren Elemente in geometrischer Interpretation als Punkte aufgefasst werden. Ein metrischer Raum ist ein Raum, auf dem eine Metrik definiert ist.

Mathematische Definition

- Sei X eine beliebige Menge. Eine Abbildung d: $X \times X \rightarrow R$ heißt Metrik, wenn gilt
  - $d(x,x) = 0$ (identische Punkte haben den Abstand 0)
  - $d(x,y) = 0 \rightarrow x=y$ (nicht-identische Punkte haben nicht den Abstand 0)
  - $d(x,y) = d(y,x)$ (Symmetrie)
  - $d(x,y) \leq d(x,z) + d(y,z)$ (Dreiecksungleichung)

Abstandsfunktion anschaulich



Was kann man messen?

- im Software-Prozess
  - Ressourcenaufwand (Mitarbeiter, Zeit, Kosten, …)
  - Fehler
  - Kommunikationsaufwand

- im Produkt
  - Umfang (LOC, Wiederverwendung, Anzahl von Klassen und Methoden, …)
  - Lesbarkeit
  - Entwurfsqualität (Modularität, Abhängigkeiten, …)
  - Produktqualität (Testabdeckung, Testergebnisse, …)

Gütekriterien für Software-Metriken

- Objektivität
  - keine subjektiven Einflüsse des Messenden
- Zuverlässigkeit
  - Wiederholung liefert die gleiche Ergebnisse
- Normierung
  - es gibt eine Skala für Messergebnisse und eine Vergleichbarkeitsskala (≤ - Beziehung)
- Vergleichbarkeit
  - man muss das Maß mit anderen Maßen in Relation setzen können

Gütekriterien für Software-Metriken

- Ökonomie
  - Messung hat geringe Kosten
- Nützlichkeit
  - Messung erfüllt praktische Bedürfnisse
- Validiät
  - Messergebnisse ermöglichen Rückschluss auf eine Kenngröße

Umfangs-Metriken

- LOC = lines of code
  - zählt die Zeilen der Quelldateien
- NCSS = non commented source statements
  - zählt die Quellzeilen ohne Kommentare und Leerzeilen
  - SLOC ≤ NCSS

- Umfangsmetriken eignen sich zur Messung des geistigen Eigentums, abzüglich
  - generierter Code
  - kopierter Code
- Qualitätsaussagen auf Basis von Umfangsmetriken sind unrealistisch.

Halstead-Metriken, 1972

- messen die textuelle Komplexität

- Einteilung des Programms in Operanden und Operatoren
  - Operatoren kennzeichnen Aktionen und sind typischerweise Sprachelemente des Programms, z.B. int, (,), return, +;
  - Operanden kennzeichnen Daten und sind Bezeichner oder Literale des Programms, z.B. x, y, z, 0, 1

Halstead-Metriken

- Bestimmung von Parametern
  - N1 = Anzahl der Operatoren
  - n1 = Anzahl der verschiedenen Operatoren
  - N2 = Anzahl der Operanden
  - n2 = Anzahl der verschiedenen Operanden
  - n = n1+n2 = Größe des Vokabulars
  - N = N1+N2 = Länge der Implementierung

Halstead-Metriken

- Schwierigkeit, ein Programm zu verstehen (engl. difficulty)
  - D = n1/2 * N2/n2
  - wobei N2/n2 das durchschnittliche Auftreten von Operanden ist
- Umfang des Programms (engl. volume)
  - V = N * log2 (n)
  - auch als „Größe des Algorithmus in Bits“ oder Halstead-Volumen bezeichnet
- Aufwand, das gesamte Programm zu verstehen (engl. effort)
  - E = D * V

Halstead-Metriken: Bewertung

- Vorteile
  - einfach zu ermitteln und zu berechnen
  - universell einsetzbar
  - Halstead-Metriken wurden experimentell als gutes Maß für die Komplexität bestätigt

- Nachteile
  - es wird nur die textuelle Komplexität betrachtet.
  - moderne Sprachkonzepte bleiben unberücksichtigt
  - Anzahl und Aufteilung von Operatoren und Operanden ist sprachabhängig

McCabe Metriken, 1976

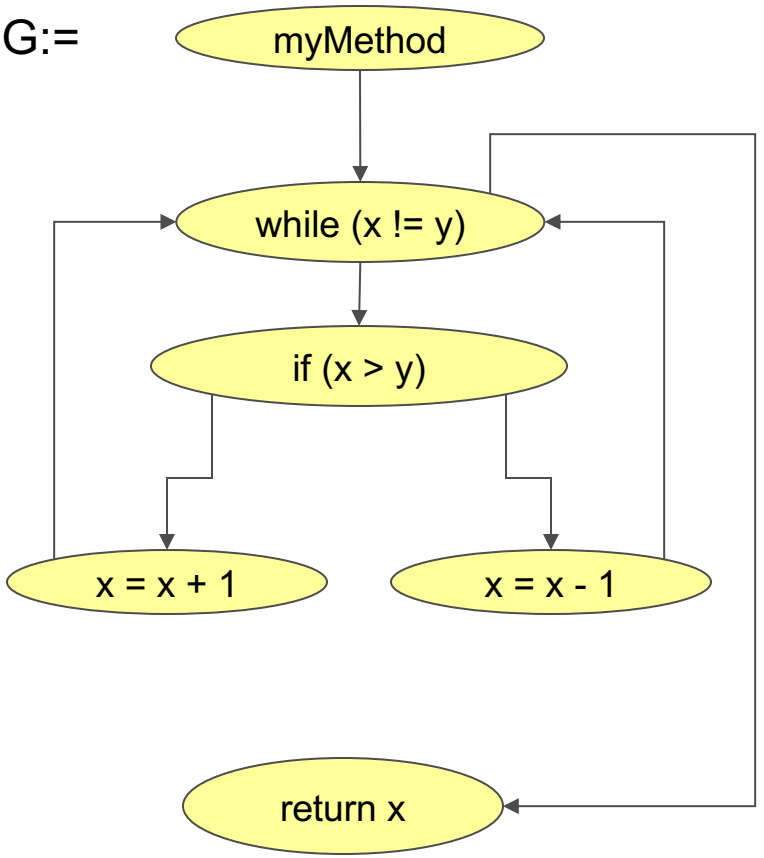
- Idee: strukturelle Komplexität bewerten
- Eingabe: Kontrollflussgraph G (engl. control flow graph, CFG)
- Ergebnis: zyklomatische Komplexität V (G)
- Bestimmung von Parametern
  - e = Zahl der Kanten (engl. edges)
  - n = Zahl der Knoten (engl. nodes)
  - p = Zahl der Zusammenhangskomponenten (engl. partial graph)
- Zyklomatische Komplexität:
  - V (G) = e – n + 2p

Kontrollflussgraph (CFG)

- Kanten
  e = 7
- Knoten
  n = 6
- Zusammenhangs-
  komponenten
  p = 1
- Bedingungen
  c = 2

- V(G) = e-n + 2p = 7-6+2 = 3

G:=



McCabe Metriken: Bewertung

- Klassifizierung:
  - V(G) in [1..10]:
    einfaches Programm, geringes Risiko
  - V(G) in [11..20]:
    komplexeres Programm, moderates Risiko
  - V(G) in [21..50]:
    komplexes Programm, hohes Risiko
  - V(G) in [51..]:
    untestbares Programm, extrem hohes Risiko

McCabe Metriken: Bewertung

- Vorteile
  - einfach zu berechnen
  - zyklomatische Komplexität korreliert gut mit der Verständlichkeit einer Komponente
- Nachteil
  - nur Berücksichtigung des Kontrollflusses, kein Datenfluss
  - ungeeignet für objektorientierte Programme (zahlreiche triviale Aufrufe)

Hybride Metriken: Wartbarkeitsindex von Hewlett-Packard

- Idee: mehrere Metriken kombinieren
- Bestimmung von Parametern
  - V‘ = durchschnittliches Halstead-Volumen pro Modul
  - V(G)‘ = durchschnittliche zyklomatische Komplexität pro Modul
  - L‘ = durchschnittliche LOC pro Modul
  - C‘ = durchschnittlicher Prozentsatz an Kommentarzeilen

- Wartbarkeitsindex MI:
  - MI = 171 – 5,2*ln(V‘) – 0,23*V(G)‘ – 16,2*ln(L‘) + 50 sin(sqrt(2,4*C‘))

  - je größer MI, desto besser die Wartbarkeit
  - MI < 30: Restrukturierung

Metriken für objektorientierte Komponenten

- McCabe Metriken versagen bei objektorientierter Programmierung: Kontrollflusskomplexität oft gering (V(G)=1)
- Idee: Metriken müssen das Zusammenspiel der Klassen betrachten.
  - typisch: anhand des statischen Objektmodells
- Für dynamische Aspekte gibt es noch keine Metriken.
  - z.B. Zustandsautomaten, Sequenzdiagramme

objektorientierte Metriken: Auswahl

- DIT = depth of inheritance tree
  - Anzahl Oberklassen einer Klasse
  - DIT größer = Fehlerwahrscheinlichkeit größer
- NOC = number of children of a class
  - Anzahl direkter Unterklassen
  - NOC größer = Fehlerwahrscheinlichkeit geringer
- RFC = response of a class
  - Anzahl Funktionen, die nur direkt über Klassenmethoden aufgerufen werden
  - RFC größer = Fehlerwahrscheinlichkeit größer

objektorientierte Metriken: Auswahl

- WMC = weighted methods per class
  - Anzahl definierter Methoden
  - WMC größer = Fehlerwahrscheinlichkeit größer
- CRO = coupling between object classes
  - Anzahl Klassen auf deren Dienste die Klasse zugreift
  - CRO größer = Fehlerwahrscheinlichkeit größer
- Offene Frage Validität:

  misst die Metrik etwas Sinnvolles?

Bewertung

- Das was wirklich interessant ist, kann nicht direkt gemessen werden.
- Metriken basieren oft nur auf Hypothesen.
- Metriken sind oft zu einfach.
  - management-tauglich?
- Metriken müssen in der Praxis validiert werden!

Standards für Software Metriken

- IEEE: Standard Dictionary of Measures to Produce Reliable Software, 1989
- IEEE: The Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software
- IEEE Standard 1061: Software Quality Metrics
  - Funktion, die Software auf einen Zahlenwert abbildet
- US Air Force policy on software metrics

Übung: LOC Metrik

- Stellen Sie die Metrik für LOC (lines of code) auf.
- Implementieren Sie die Metrik in Java.
  - Eingabe 1: Name einer Programmdatei im Betriebssystem
  - Eingabe 2: Name einer Programmdatei im Betriebssystem
  - Ausgabe: Messwert (Wert d von LOC)
- Beweisen Sie, dass LOC eine Metrik ist.
- Bewerten Sie die LOC Metrik.

Übung: NCSS Metrik

- Stellen Sie die Metrik für NCSS (non commented source statements) auf.
- Implementieren Sie die Metrik in Java.
  - Eingabe 1: Name einer Programmdatei im Betriebssystem
  - Eingabe 2: Name einer Programmdatei im Betriebssystem
  - Ausgabe: Messwert (Wert d von NCSS)
- Beweisen Sie, dass NCSS eine Metrik ist.
- Bewerten Sie die NCSS Metrik.
- Vergleichen Sie die Metriken LOC und NCSS.

Zusammenfassung

- Software-Metriken definieren, wie eine Kenngröße in einem Software-Produkt oder einem Software-Entwicklungsprozess gemessen wird.
- Validität ist wesentlich:
  - misst die Metrik wirklich, was gemessen werden soll?
- Bei Metriken sind heute noch keine Standards etabliert.
- Messungen ersetzen nicht Test oder Verifikation.

Eine Definition…

- „Handbuch der Raumfahrttechnik“, Hanser Verlag, 3. Auflage 2008, S.364 (ausleihbar in der DHBW Bibliothek)

- „Zur Zeit gibt es ca. 400 Metriken, um die Eigenschaften von Software zu messen, aber sie sind alle sehr umstritten und kaum akzeptiert. Alles dies macht die Beurteilung von Software recht schwierig.“

  =>>[im Original fortlaufender Text]

  „Man weiß nur: Sie [Software] ist immateriell, abstrakt, komplex, überall, groß, teuer und dazu immer mit Fehlern behaftet.“

Selbstkontrolle

- Wie ist eine Metrik definiert?
- Warum eigenen sich Umfangsmetriken kaum für die Bewertung der Softwarequalität?
- Erläutern Sie, wie Halstead die Schwierigkeit eines Programms definiert.
- Erläutern Sie, warum die Messung der zyklomatischen Komplexität eine gute Vergleichsmethode für ähnliche Programme ist.
- Begründen Sie, die der Wartbarkeitsindex von HP entstanden ist.

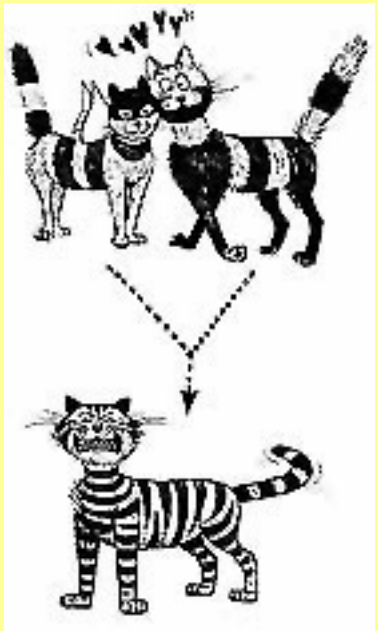
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de

Software Engineering 2
Objektorientierte Analyse



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Was ist Objektorientierte Analyse (OOA)?

- Vorgehensmodell im Softwareentwicklungsprozess
  - Zielsystem soll möglichst präzise in seinen Aspekten beschrieben werden
- Bestandteile der OOA
  - Idee und Ziel formulieren
  - Anwendungsfälle identifizieren und formulieren
  - Fachklassen modellieren
  - Glossar erstellen
  - Ablaufmodell entwickeln
  - Schnittstellen beschreiben

Systemidee und Zielsetzung

- Systemidee entwickeln
  - Auftraggeber, Entwickler und Anwender einbeziehen
  - Widersprüche beseitigen
  - Interessenskonflikte klären

- Systemidee
  - Was soll mit dem zu entwickelnden System erreicht werden?
  - vollständige Beschreibung zu diesem Zeitpunkt oft noch nicht möglich

Systemidee schriftlich fixieren

- Systemvoraussetzungen
- Leistung und Eigenschaften
  - 5-15 wichtigste Eigenschaften und Merkmale formulieren
- Name des Produkts
- Voraussetzungen
  - z.B. Hardware, Betriebssystem, Werkzeuge, Fähigkeiten
- Preis des fertigen Systems bzw. Produkts
- Entwicklungsstufen definieren
  - welche Eigenschaften sind in der ersten Version verfügbar, welche erst in der Weiterentwicklung?

Interessenhalter (Stakeholder)

- Interessenhalter finden und bewerten
  - Relevanz
  - Risiko
- Projekt-Ansprechpartner finden und kategorisieren
  - Fachexperten
    - Experten des Anwendungsgebiets, aber nicht unbedingt Anwender
  - Anforderungsverantwortliche
    - Personen, die befugt sind, Anforderungen an das System festzulegen
  - Systembetroffene
    - z.B. Anwender

Interessenhalter

- Interessenhalter klassifiziert
  - Anwender
  - Fachabteilung
  - Auftraggeber, Management, Geldgeber
  - Gesetzgeber, Standards
  - Administration, Service, Schulung, Hotline
  - Entwickler
  - Käufer
  - Marketing, Vertrieb
  - Projektgegner, Projektbefürworter

Bewertung der Interessenhalter

- Idee: behandle die wichtigen Interessenhalter bevorzugt.

- Bewertung:
  1. Wie groß ist das Risiko, wenn dieser Interessenhalter nicht berücksichtigt wird?
     - Risiko: 0=kein, 1=gering, 2=mittel, 3=hoch, 4=sehr hoch, 5=fatal
  2. Wie groß ist der Aufwand, diesen Interessenhalter zu berücksichtigen?
     - Aufwand: 0=vernachlässigbar, 1=gering, 2=mittel, 3=hoch, 4=sehr hoch, 5=extrem
       - alternativ: absteigend priorisieren
- Berechnung der Priorität: Abstand zum Nullpunkt, Randbedingungen beachten
  Priorität = sqrt(Risiko^2 + Aufwand^2) $\rightarrow$ [0..7]

Vereinfachung der Priorität

- muss
  - Priorität > 5
  - Wenn Interessenhalter nicht angemessen berücksichtigt werden können, muss das Projekt hinterfragt werden.
- sollte
  - 2 < Priorität ≤ 5
  - Wenn Interessenhalter nicht angemessen berücksichtigt werden können, so ist das noch akzeptabel.
- könnte
  - Priorität ≤ 2
  - Interessenhalter können weitgehend vernachlässigt werden.

Interessen der Interessenhalter

- Ziele und Interessen der einzelnen Interessenhalter beschreiben
  - Interessen können sich ergänzen, widersprechen oder überraschen
    - z.B. Gefährdung des eigenen Arbeitsplatzes
  - Befugnisse und Verantwortungen im Projekt betrachten
  - Was darf auf keinen Fall passieren?
- Probleme und Schwachstellen aus Sicht der Interessenhalter identifizieren
- die wichtigsten Eigenschaften aus Sicht der Interessenhalter beschreiben

Geschäftsanwendungsfälle

- Geschäftsanwendungsfall (kurz Geschäftsfall, engl. business use case)
  - Ein Geschäftsfall beschreibt einen geschäftlichen Ablauf, der von einem geschäftlichen Ereignis ausgelöst wird. Ergebnis ist die Erreichung eines Teils der Systemziele.
  - Ein Geschäftsfall ist ein abstrakter Anwendungsfall ohne Bezug zu einer systemtechnischen Umsetzung.
- Beispiel: einen Artikel in den Warenkorb aufnehmen.
- Anwendungsfälle identifizieren
  - Anfang, Ende, Auslöser
  - möglichst abstrakt und allgemein formulieren

Geschäftsfälle notieren

- für jeden Geschäftsfall notieren
  - Name
  - Anwendungsfall-Typ (fest: Geschäftsfall)
  - Kurzbeschreibung
  - Auslöser, evtl. Motivation
  - Ergebnis
  - Akteure

- Regeln
  - Keine Lösungsansätze in Anwendungsfällen formulieren.
  - Auch auszuschließende Geschäftsfälle formulieren.
  - Keine Reihenfolge der Anwendungsfälle festlegen.

Geschäftsfälle notieren

- Abgrenzung: wo beginnt/endet ein Anwendungsfall?
  - kein einzelner Schritt
  - keine einzelne Operation
  - möglichst großen und zusammenhängenden Ablauf darstellen
  - z.B. neuen Kunden erfassen

Anwendungsfälle verfeinern

- für jeden Anwendungsfall
  - eigentliche geschäftliche Intention formulieren
    - möglichst abstrakt
    - aus Sicht des Geschäftstreibenden beschreiben
  - stabile von dynamischen Anforderungen unterscheiden
    - dynamische Anforderungen werden später durch Wartung bzw. Erweiterung modifiziert
    - kurz- und mittelfristig veraltete Sachverhalte kennzeichnen
  - Auslöser, Vorbedingung und eingehende Informationen definieren
  - Ergebnis, Nachbedingung und ausgehende Informationen definieren
  - Anwendungsfälle klar abgrenzen (Kohärenz)
    - evtl. Enthält-Beziehung (<<include>>) verwenden

Materialien und Gegenstände erfassen

- Materialien und Gegenstände im Anwendungsbereich erfassen
  - Beispiele und Muster notieren
  - bei der Erfassung eindeutig nummerieren
- Bewertung: brauchbar und relevant?
  - aktuell
  - verbindlich
  - korrekt
  - wichtig

- Beispiele
  - Kundenakten
  - Verträge
  - Rechnungen
  - Allgemeine Geschäftsbedingungen

Systemanwendungsfälle identifizieren

- kurz: Systemfälle
  - Systemfälle sind diejenigen Geschäftsfälle, die ganz oder teilweise systemtechnisch umgesetzt werden sollen.
  - Systemfälle erweitern Geschäftsfälle um die technische Umsetzung

- Systemfälle sind zeitlich kohärent aus den Geschäftsfällen erzeugt.

Zeitlich kohärente Systemfälle

- zusätzlich zu Anfang, Ende, Auslöser aus dem Geschäftsfall:
  - fachliche Transaktionseinheiten definieren, die entweder ganz oder gar nicht ausgeführt werden
  - vor, nach und nach Abbruch der Transaktionseinheit muss ein konsistenter Systemzustand vorliegen

Systemfälle ergänzen

- weitere Informationen ergänzen:
  - Ansprechpartner
  - Risiko
    - z.B. Kostenüberschreitung, Terminverzug
  - Wichtigkeit
    - unverzichtbar, wichtig, nützlich
  - geschätzter Aufwand
    - in Personentagen
  - Stabilität
    - Wahrscheinlichkeit, dass sich der Anwendungsfall ändert
  - Zeitpunkt, Dringlichkeit
    - wann muss der Systemfall spätestens umgesetzt sein?

Fachklassen identifizieren

- Eine Fachklasse beschreibt einen Gegenstand, ein Konzept, einen Ort oder eine Person aus dem Anwendungsbereich so detailliert, dass er von Fachabteilungen und Entscheidungsträgern verstanden werden kann.

- die wichtigsten fachlichen Gegenstände identifizieren, die vom System repräsentiert werden sollen
  - als Klassen betrachten
  - Zusammenhänge in einem Klassendiagramm modellieren
    - wesentliche Attribute und Operationen
  - Klassennamen vergeben
  - Assoziationen und Kardinalitäten beschreiben
    - z.B.: jeder Prozessor hat höchstens eine Seriennummer = 1:0..1

Fachklassen verfeinern

- Sachverhalte untersuchen bzw. dokumentieren
  - Bestandteile, Untergliederung
  - Bedingungen, Ereignisse, unerwünschte Ereignisse
  - Lebenszyklus: Konstruktions- und Destruktionszeitpunkte
  - Zusicherungen, Rollen, Operationen, Datentypen
  - Beziehungen zu anderen Objekten
  - Mengenangaben bzw. Mengengerüst
  - Berechtigungen
  - Übernahme der Daten aus bestehenden Systemen

Fachklassen verfeinern

- Verantwortlichkeiten
- Überschneidungen mit anderen Systemen
- Wichtigkeit

- Standardwerte für Attribute definieren
- Namen aller Fachklassen und damit verbundenen Rollen im Glossar eintragen
- Abbildungsverzeichnis einfügen

Fachliches Glossar

- Problem
  - Verständnis für Vorgänge, Begriffe und Gegenstände für alle Beteiligten verbindlich formulieren.
    - Vieles wird als intuitiv klar angenommen und schafft im Projektverlauf große Schwierigkeiten
- alle wichtigen fachlichen Begriffe definieren
  - Klassen und Rollen des Klassenmodells
  - wichtige allgemeine und fachliche Prozessworte

Glossar überprüfen

- Glossar prüfen: sind die Formulierungen vollständig und verständlich?
  - Prozessworte sind Verben und substantivierte Verben
    - W-Fragen zur Verfeinerung stellen
  - Vergleiche und Steigerungen vervollständigen
    - besser als was?
  - Allquantoren (formulieren Extremwerte) prüfen
    - Wirklich jede Woche Auto waschen?
  - Bedingungen Prüfen: alle Varianten formuliert?
    - nur sonntags ausschlafen?

Struktur eines Glossareintrags

1. Begriff
2. mögliche Synonyme
3. Definition (max. 5 Sätze)
4. ähnliche oder gegensätzliche Begriffe mit Kontext und Abgrenzung
5. Einschränkung auf Anwendungsbereiche
6. Ansprechpartner, Historie, Herkunft
7. Status (Entwurf, final)
8. Änderungshistorie

Beispiel für einen Glossareintrag

Begriff	Motorrad
Synonyme	Kraftrad, Töff, Bike, Moto
Definition	Ein Motorrad ist ein motorisiertes, nicht an Schienen gebundenes einspuriges Landfahrzeug mit einer Höchstgeschwindigkeit von mindestens 45 km/h oder einem Hubraum größer als 50 cm^3.
Abgrenzung	Mofa, Moped: Begrenzung von Hubraum, Leistung und Höchstgeschwindigkeit
Einschränkungen	keine
Ansprechpartner	Andreas Judt
Status	Entwurf
Änderungen	15.08.2007: erstellt

Systemablaufmodell

- Systemablauf modellieren
  - jeden Anwendungsfall als Aktivitätsdiagramm modellieren

- Schritte im Aktivitätsdiagramm ermitteln
  1. Standardablauf beschreiben
     - keine Ausnahmen oder Varianten
  2. vollständigen Ablauf beschreiben
     - alle Ausnahmen und Varianten aufnehmen
  3. Testfälle erzeugen
  4. Objektfluss beschreiben
     - zu allen Aktivitäten ein- und ausgehende Objekte bzw. Objektzustände notieren
     - gehört eigentlich zu Design

Vollständiges Systemablaufmodell

- jeder Anwendungsfall besitzt ein Aktivitätsdiagramm
- jedes Aktivitätsdiagramm besitzt einen Anfangs- und einen Endknoten
- der Endknoten besitzt einen aussagekräftigen Namen
- jeder Schritt eines Aktivitätsdiagramms wurde eindeutig nummeriert
- jeder ausgehende Kontrollfluss wurde eindeutig nummeriert
- alle fachlichen Ausnahmen wurden berücksichtigt
  - keine technischen Ausnahmen!
- alle Abbruchkriterien wurden berücksichtigt

Systemanwendungsfallmodell

- Systemanwendungsfallmodell erzeugen
  - alle Systemfälle in einem Anwendungsfallmodell darstellen
  - gleiche Schritte in Systemfällen finden und als sekundäre Systemfälle zentral modellieren
    - vermeidet Redundanzen
  - Abhängigkeiten zwischen Anwendungsfallpaketen minimieren.
    - zyklische Abhängigkeiten sind problematisch

Regeln für Paketzuordnung bei sekundären Systemfällen

- Ein sekundärer Systemfall, der nur von Systemfällen eines Pakets verwendet wird, gehört in dieses Paket.
- Liegt eine zyklische Abhängigkeit vor, ist zu prüfen, ob der sekundäre Systemfall in ein anderes Paket verschoben werden kann, so dass nur eine einfache Abhängigkeit besteht.
- Kann eine zyklische Abhängigkeit nicht durch obigen Schritt beseitigt werden, sollte ein neues Paket erzeugt werden, das Zyklen auslösende Systemfälle sammelt.

Nicht modellierte funktionale Anforderungen

- alle verbleibenden Anforderungen an das System detailliert formulieren

1. Nummer der Anforderung
2. Art (fest: funktionale Anforderung)
3. Beschreibung
4. Stabilität (stabil, instabil)
5. Verbindlichkeit (Pflicht, Option)
6. Priorität (sehr hoch, hoch, mittel, niedrig)
7. Verweise auf Regeln, Anforderungen, Modellelemente

Schnittstellen beschreiben

- Basis: Beschreibung in den Anwendungsfällen
- Schnittstelle beschreiben
  1. Name der Schnittstelle
  2. Art
  3. Beschreibung
  4. Komplexität (komplex, normal, einfach)
  5. Hinweise zur Verwendung
- Schnittstellenarten
  - Dialogschnittstelle
  - Ausgabe (z.B. Ausdruck)
  - Datenschnittstelle von/zu externen Systemen
  - funktionale Schnittstellen zu externen Systemen

Schnittstellen-Prototypen

- üblicherweise eine Sequenz von Benutzerschnittstellen
- illustrieren Anwendungsfälle
- eignen sich für die Diskussion mit den Anwendern
  - Anwendungsfälle verstanden?
  - Abläufe verstanden?
  - gibt es aus Sicht der Anwender bessere Lösungen?

Selbstkontrolle

1. Welchen Zweck erfüllt die OOA?
2. Warum ist es dringend erforderlich, die Interessenshalter genau zu analysieren?
3. Was unterscheidet Systemfälle von Geschäftsfällen?
4. Warum ist ein Glossar wichtig für den Projekterfolg?
5. Wie hängen OOA und UML zusammen?

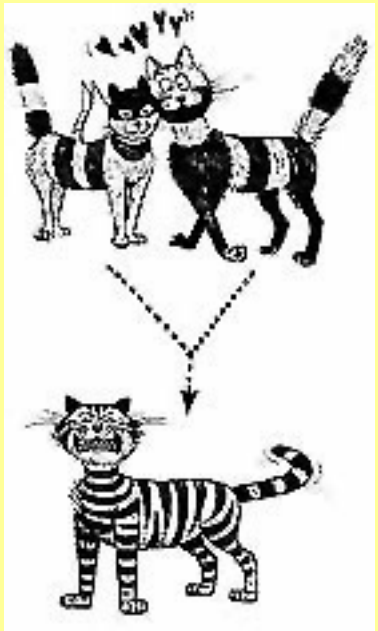
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de

Software Engineering 2
Design



Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

Design und Architektur

- Design
  - Design ist die kreative Erarbeitung eines Lösungskonzepts für ein gegebenes Problem innerhalb eines gegebenen Lösungsraums.
- Architektur
  - Architektur ist die Festlegung der grundlegenden übergreifenden Eigenschaften und Möglichkeiten einer Lösung.
- Bestandteile einer Softwarearchitektur
  - Eine Softwarearchitektur sollte durch folgende Sichten beschrieben werden:
    - Schichtenmodell
    - Verteilungsmodell
    - fachliches Subsystemmodell

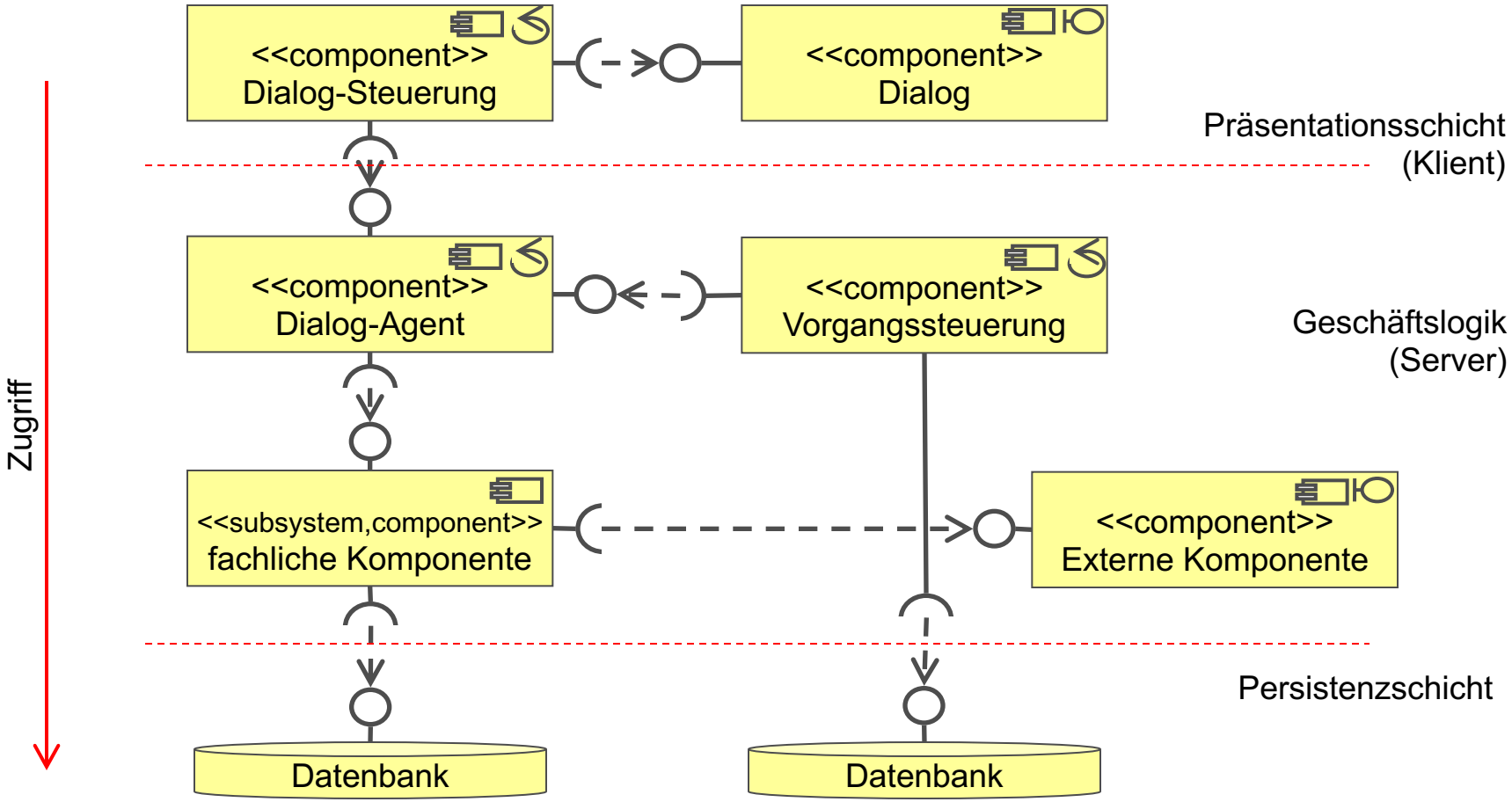
Sichten einer Softwarearchitektur

- Schichtenmodell
  - Ein Schichtenmodell beschreibt, aus welchen grundsätzlichen Hard- und Softwareschichten das Zielsystem besteht
- Verteilungsmodell
  - Ein Verteilungsmodell gibt an, welche Hard- und Software vorhanden ist, auf welcher Hardware welche Software läuft und welche Verbindungen zwischen diesen Teilen bestehen.
- fachliches Subsystemmodell
  - ein fachliches Subsystemmodell partitioniert das Zielsystem in fachliche Untereinheiten und beschreibt ihre Zusammensetzung.

Schichtenmodell definieren

- Vorgehensweise
  - einzelne Elemente und Beziehungen der Architektur identifizieren und in einem Modell darstellen
    - Schichten,
    - Komponenten
    - Klassenarten
  - für jedes Element:
    - Verantwortlichkeiten, Ausgaben und Besonderheiten beschreiben
  - Kommunikationsmechanismen mit Interaktionsdiagrammen beschreiben

Verwendetes Schichtenmodell



Schichtenmodell: Dialog, Dialogsteuerung

- Dialoge bilden die Schnittstelle zum Anwender.
  - Ausgabe von Informationen
  - Eingabe neuer Daten
- Grafische Dialoge sind typischerweise komponentenbasierte Systeme mit verschiedenartigen Interaktionskomponenten
  - z.B. Java Swing, AWT, SWT, C++ Motif, Qt
- Viele Dialoge sind heute nicht mehr grafisch.

Schichtenmodell: Dialog-Agent

- Stellvertreter-Objekt, dass einen Klienten auf dem Server vertritt.
- Der Dialog-Agent steuert die Kommunikation zwischen dem Dialog und serverseitigen Komponenten.
- Zweck
  - Kommunikation zwischen Klient und Server optimieren.

Schichtenmodell: Fachliche Komponenten

- repräsentieren den eigentlichen Anwendungsbereich
- beinhalten und kapseln fachliche Klassen
- Teil der fachlichen Komponenten:

  Anwendungsfallsteuerung

Schichtenmodell: Anwendungsfallsteuerung

- Teil einer Fachkomponente

- repräsentiert einen Systemfall und die im Aktivitätsmodell festgelegten Abläufe
  - steuert die Interaktion der fachlichen Klassen
  - erhält Interaktionen vom Dialog in Form von Ereignissen
  - steuert die Darstellung neuer Dialoge
  - übergibt Daten an den Dialog in Datentransferobjekten

Schichtenmodell: Vorgangssteuerung (engl. workflow)

- initiiert, überwacht und steuert die Abarbeitung eines Geschäftsfalls aus fachlicher Sicht
  - Anwendungsfälle: zeitlich zusammenhängende Teile eines Geschäftsprozesses
- persistiert Zustände der Komponenten bei zeitlicher Unterbrechung
  - z.B. Scheduling von Komponenten
  - z.B. Herunterfahren der Anwendung

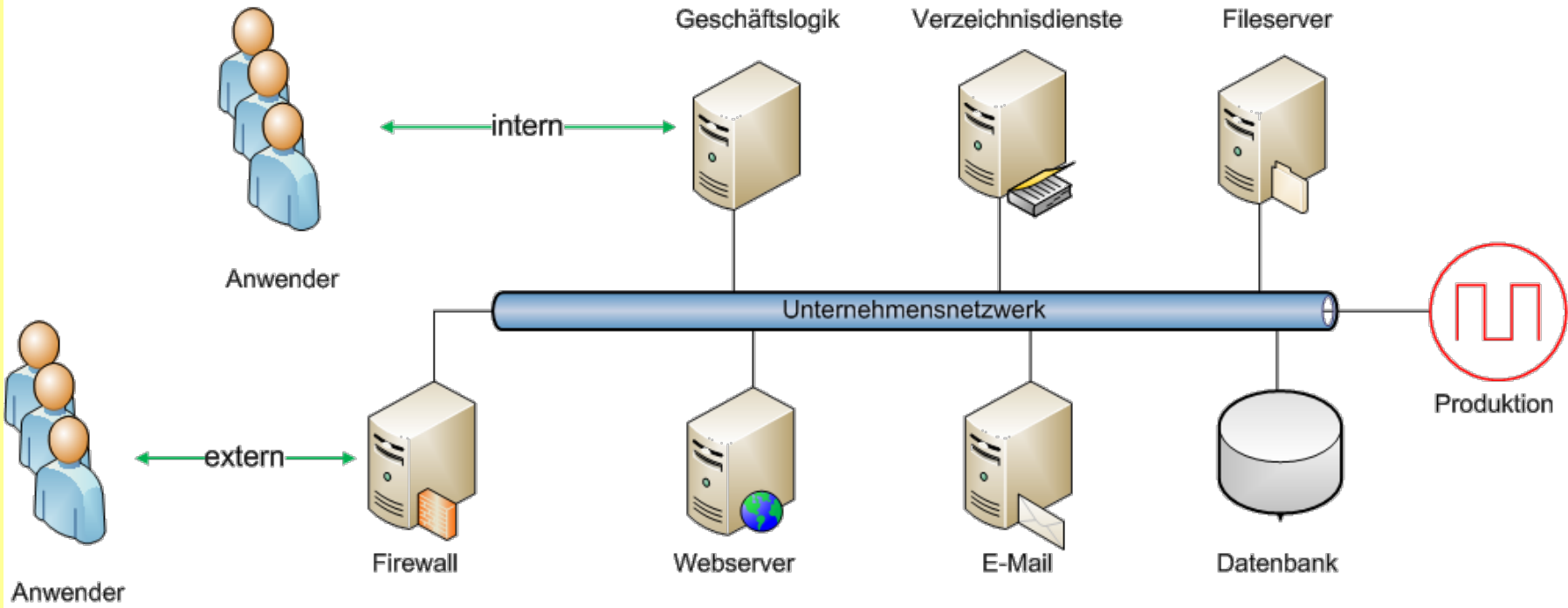
Schichtenmodell: Anforderungen an eine Anwendungsarchitektur

- Schichtenmodell erstellen
- Verantwortlichkeiten (Aufgaben) aller Architektur-Bestandteile dokumentieren
- Interaktionen und Schnittstellen der Architektur-Bestandteile beschreiben
- Verwendung von Entwurfs- und Architekturmustern dokumentieren
- Löst die Architektur das gestellte Problem überhaupt?
- Wurde die erlaubte Zugriffsrichtung eingehalten?

Verteilungsmodell definieren

- Zielhardware definieren (sog. Knoten eines verteilten Systems)
- festlegen, welche Bestandteile des Zielsystems auf welchem Knoten ablaufen werden
- technische Kommunikationswege festlegen
- Beschreibung: Verteilungsdiagramm
  - auch Einsatzdiagramm

Verteilungsmodell: Beispiel



Fachliches Subsystemmodell definieren

- Basis: Klassenmodell aus der Analyse
- zu jedem Geschäftsprozess eine Vorgangssteuerung definieren
- zu jedem Anwendungsfall eine Anwendungsfallsteuerung definieren
- jedes externe System durch einen Adapter repräsentieren
- fachlich zusammenhängende Klassen des Analysemodells zu fachlichen Komponenten oder Subsystemen zusammenfassen
  - Komponente: zusammenhängender, austauschbarer Teil des Systems mit wohldefinierten Schnittstellen
  - Subsystem: grobgranularer Teil des System, der aus Komponenten besteht und eine Menge von Schnittstellen bereitstellt

Fachliches Systemmodell: externe Systeme repräsentieren

- Interaktionen können mit externen Systemen ausgeführt werden.
  - sog. Legacy-Systeme
- Externe Systeme werden als Komponente gekapselt.
  - ermöglicht die Beschreibung einer Schnittstelle
  - externe Systeme müssen nicht objektorientiert sein!

Verantwortlichkeiten festlegen

- Aus den Abhängigkeiten zwischen den Klassen des Analysemodells werden strukturelle Abhängigkeiten zwischen den Komponenten abgeleitet.
  - Kriterium für Komponentenbildung
    - Minimierung der Kopplung und Abhängigkeiten (Schnittstellen) zwischen den Komponenten.
  - Ablaufdiagramm liefert weitere Aussagen über Zusammenhänge zwischen den Komponenten.
    - jeden Schritt eindeutig einer Komponente zuordnen
    - bei Unklarheiten hat die Minimierung der Schnittstellen Vorrang.

Komponentenspezifische Klassenmodelle entwickeln

- Basis: Klassenmodell der Analyse
  - auch Problembeschreibung genannt
- für jede Komponente:
  - Design-Klassenmodell entwickeln
    - auch Lösungskonzept genannt
  - Verantwortlichkeiten (=Aufgaben) aller komponentenspezifischen Klassen definieren
  - Assoziationen mit Kardinalitäten zwischen den Klassen formulieren
  - ggf. komponentenspezifisches Klassenmodell restrukturieren
- Ergebnis:

  Klassenmodell des Designs (Lösungskonzept)

Komponentenschnittstellen entwickeln

- Basis: Aktivitätsdiagramm
- für jede Komponente die erforderlichen Schnittstellen modellieren
  - Schnittstelle der Anwendungsfallsteuerung
    - wird vom Dialog-Agenten benutzt
  - Schnittstellen zu anderen fachlichen Komponenten
    - wird von der Anwendungsfallsteuerung benötigt
- Anforderungen an die Komponentenschnittstellen
  1. Alle im Aktivitätsdiagramm enthaltenen Objekte sind in den Schnittstellen wiederzufinden.
  2. Die Schnittstellen enthalten alle für die Ablaufsteuerung notwendigen Operationen.

Zustandsmodelle verfeinern

- Mögliche Zustände eines Objekts identifizieren.
  - welche Zustände sollen modelliert werden?
- Modellierung ist nur sinnvoll, wenn
  - ein Objekt viele Zustände besitzt
  - wenn ein Objekt im System wichtig ist
- Modellierung von Zuständen
  - Operationen als zustandsabhängig oder zustandsunabhängig kennzeichnen
  - z.B. mit Zustandsautomat, Zustandstabelle
- Überflüssige Modellierungen steigern die Entwicklungskosten unnötig.
  - Einfache Zustandsmodelle können ad hoc umgesetzt werden.

Objektfluss modellieren

- Idee: Aktivitätsdiagramme durch Modellierung der beteiligten Objekte verfeinern
  - Aus Aktivitätsdiagrammen werden Objektflussdiagramme.
  - für jeden Schritt:
    - eingehende Objekte erfassen
    - ausgehende Objekte erfassen
  - Jeder Bearbeitung eines Objekts kann genau ein Schritt und damit genau eine Operation zugeordnet werden.

- Objekte erfassen
  - Kontrollflüsse durchgehen
  - erzeugte und veränderte Objekte betrachten

Interaktionsmodelle entwickeln

- für jeden Anwendungsfall ein Sequenz- oder Kommunikationsdiagramm entwickeln, das den Standardfall darstellt.
  - zusätzlich für die wichtigsten Ausnahmefälle jeweils ein Kommunikationsdiagramm entwickeln
  - alle übrigen Ausnahmefälle werden ad hoc entwickelt
- noch fehlende Eigenschaften identifizieren:
  - Attribute
  - Assoziationen
  - Operationen
  - Klassen

Ablauforientierte Komponententests

- Anforderungen
  - zu jedem Anwendungsfall notwendige Tests für alle Abläufe definieren
  - Design muss automatisierbare Tests erlauben
    - z.B. mit JUnit für Java-Programme

- Warum hier testen?
  - fehlende Operationen werden leicht erkannt
  - Entwicklung von Testfällen ergibt eine vollständige Menge von Tests
  - Später entwickelte Testszenarien liefern typischerweise keine vollständige Abdeckung der zu testenden Funktionalität.

Klassentests

- Operationen der Komponentenschnittstelle jeweils einer Klasse zuordnen
- für jede Operation definieren:
  - Vorbedingung
  - Nachbedingung
- für die zugehörige Klasse definieren:
  - Invariante
- Vor-, Nachbedingung und Invariante werden als Objektzustände definiert.
- für jede Operation automatisierbare Tests entwickeln

Anforderungen an die Vollständigkeit von Design-Klassen

1. Zu jeder Operation wurde eine Testmethode entwickelt.
2. Die Semantik jeder Operation wurde beschrieben.
3. Vor-, Nachbedingung oder Operation und die Invariante der Klasse wurde beschrieben.
4. Es werden keine Informationen über globale Variable erfragt, sondern nur in die Operationen übergeben.
5. Jede Operation erfüllt nur eine einzige Aufgabe.
6. Erfüllen überschriebene Operationen das Vererbungsprinzip?
7. Die Extremwerte aller Parameter wurden berücksichtigt.

Attribute definieren

- benötigte Attribute ermitteln und jeweils einer Klasse eindeutig zuordnen
  - überprüfen, ob ein Attribute besser eine eigene Klasse sein sollte
- Attribute unterscheiden in:
  - gewöhnliche Attribute
  - fachliche Schlüssel (Attribute vom Type einer Fachklasse)
  - Aufzählung
  - primitiver Datentyp
- für jedes Attribut definieren:
  - Name
  - Typ
  - Initialwert
  - Grenzwerte
  - Zusicherung
  - Zuordnung zu einer Verantwortlichkeit (Aufgabe) der Klasse

Dialoge spezifizieren

- Anforderungen an Dialoge ermitteln
  - bei welchen Anwendungsfällen ist ein Dialog erforderlich?
    - wird der Dialog in verschiedenen Kontexten verwendet?
  - welche Informationen müssen präsentiert werden?
  - welche Informationen müssen erfragt werden?

- für jede Information definieren:
  - Name
  - Datentyp
  - Initialwert
  - Grenzwerte

Was bringt OO Analyse und Design?

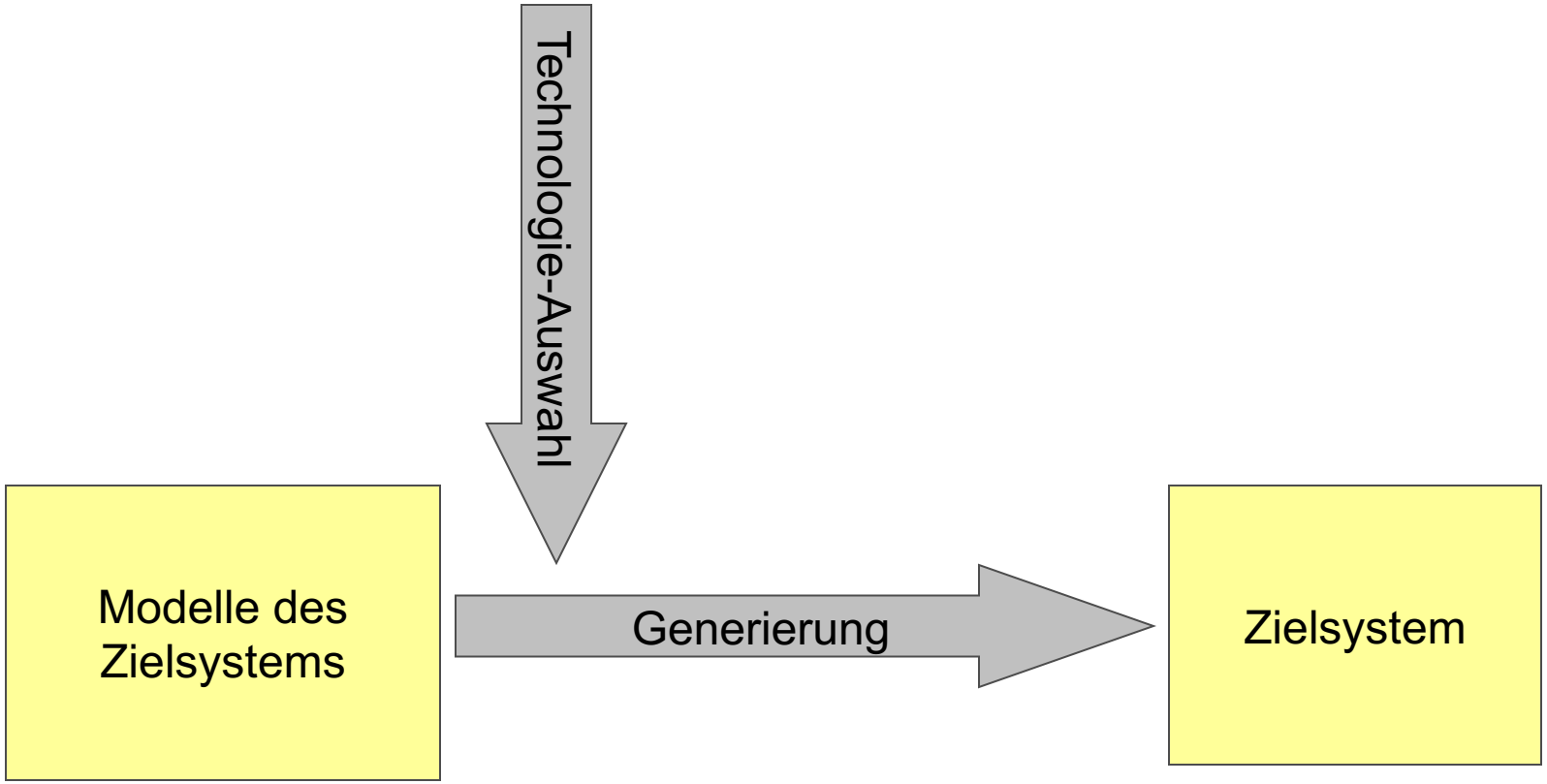
- Zerlegung von Wissen und Anforderungen über das Zielsystem in viele kleine Teilinformationen.
- Ein Weg, gewonnene Informationen strukturiert zusammenzufügen.
- Kontrollmechanismen
  - Vollständigkeit
  - Tests

Was erreicht wurde

Was bringt OO Analyse und Design nicht?

- den Weg in eine vollständige Implementierung
  - Viele unwichtige Informationen sind in den Modellen nicht enthalten.
  - Das Design-Modell liefert keine enge Vorschrift für die Implementierung.
    - Die Modelle können auf vielfältige Weise in Implementierungen umgesetzt werden.
  - Eine Implementierung kann nicht automatisch auf Vollständigkeit bezüglich eines Modells geprüft werden.
    - Stichworte: Architekturvergleich und -bewertung

Was nicht erreicht wurde



Selbstkontrolle

1. Warum werden verschiedene Sichten auf das Zielsystem benötigt?
2. Was ist ein Verteilungsmodell?
3. Skizzieren Sie das vorgeschlagene Schichtenmodell.
4. Warum ist die Zugriffsrichtung in der Softwarearchitektur wichtig?
5. Wie funktioniert ein Klassentest?
6. Wann ist eine Design-Klasse vollständig?
7. Warum kann Software aus einer OOA/D nicht automatisch generiert werden?

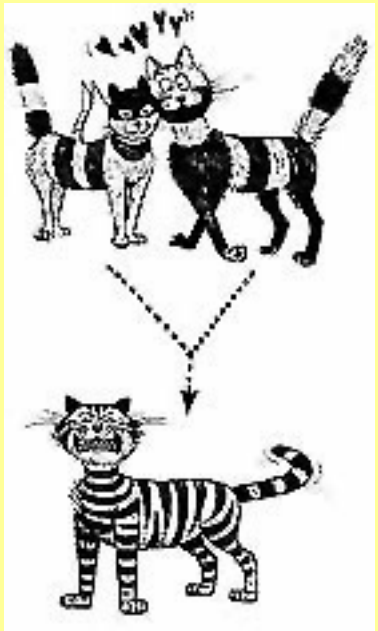
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de

Software Engineering 2
Technische Projektkonzeption



Prof. Dr. Andreas Judt

Inhalt

- Technologieentscheidungen: Einflussfaktoren
- Wissen über das Zielsystem erfassen
- Technische Machbarkeit untersuchen

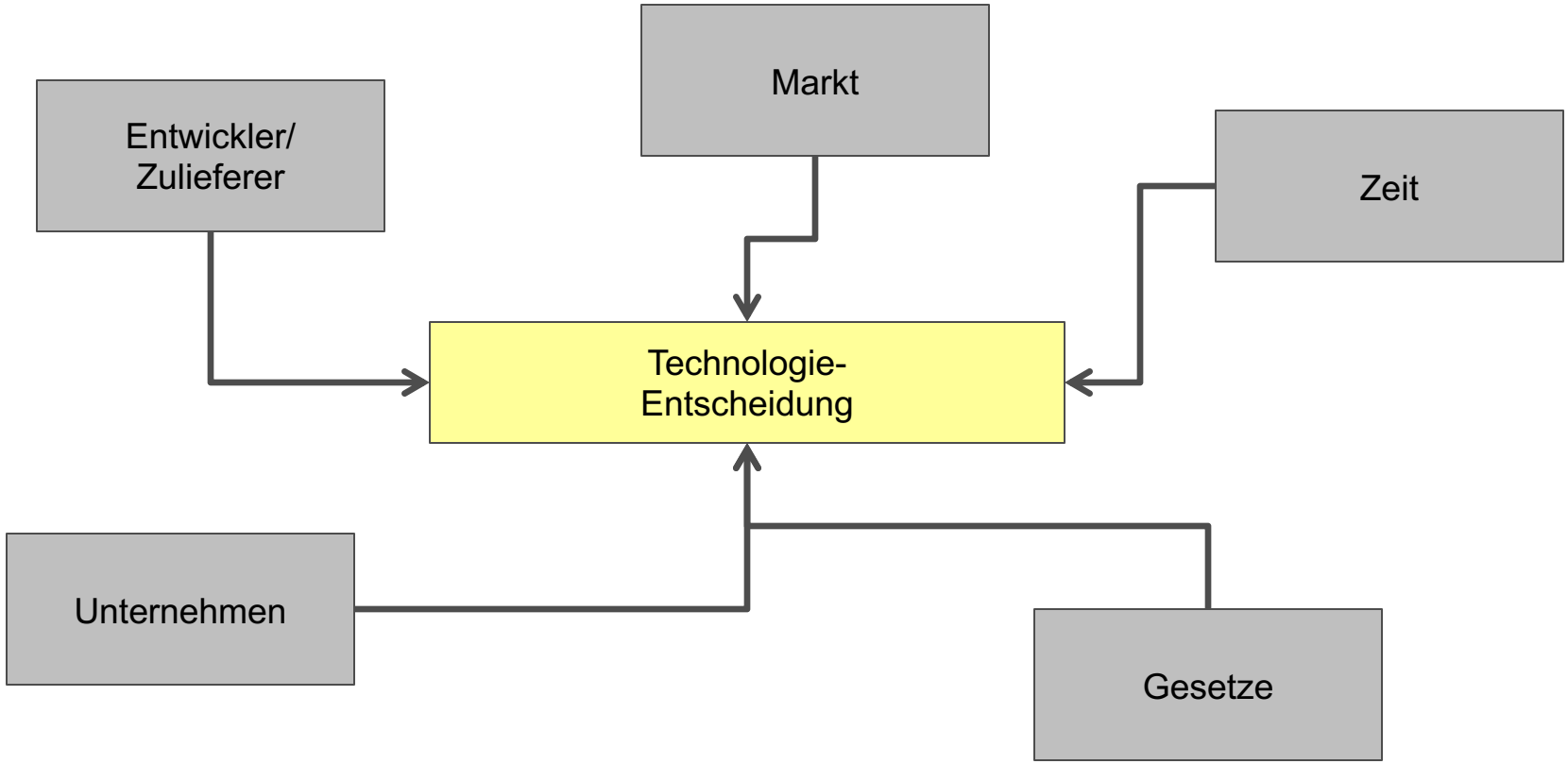
Technologie-Entscheidungen:

Einflussfaktoren

Wie wählt man eine geeignete Technologie aus?

- Es gibt viele Ansätze, welcher ist geeignet?
- Mehrere Technologien erscheinen vielversprechend.
- Tutorials und Foren im Internet scheinen die Funktionsfähigkeit zu belegen.
  - z.B. stackoverflow.com
- Oft stellt man erst im Projektverlauf fest, dass wichtige Produktfunktionen nicht implementierbar sind.

Was beeinflusst Technologie-Entscheidungen?



Einflussfaktor Entwickler / Zulieferer

- Entwickler / Zulieferer bringt Expertise für ein konkretes Produktumfeld mit

- Wechsel der Technologie unwahrscheinlich
  - Erarbeitung von neuem Wissen ist teuer
  - vom Zulieferer / Entwickler selbst unerwünscht

Einflussfaktor Unternehmen

- Unternehmen haben bereits konkrete Technologien im Einsatz.

- Die Einführung einer neuen Technologie ist unwahrscheinlich.
  - Integrationsprobleme durch Technologiesprünge sind zu erwarten.
  - Die Weiterverwendung von bestehendem Technologiewissen ist für Unternehmen kostengünstiger (vgl. Einsatz von PHP)

Einflussfaktor Markt

- Unattraktive Technologien werden am Markt nicht akzeptiert.
- Die Coolness von Produkten und Anwendungen kann sich durch äußere Ereignisse extrem schnell umkehren.
  - Bestehende Entwicklungen müssen im schlechtesten Fall verworfen werden.
- Technologien können aus produktpolitischen Gründen abgekündigt werden.
  - Die Zeitpunkte dafür sind fast immer unkalkulierbar.

Einflussfaktor Zeit

- Wie beständig sind die aktuell eingesetzten Technologien?
  - Prozessoren
  - Betriebssysteme (z.B. Windows Phone)
  - Programmiersprachen

- Welche Innovationen kommen?
  - neue Hardware, z.B.
    - Druckempfindlichkeit von Displays
    - Datenbrillen
    - Smart Watches, bald ohne verknüpftes Smartphone
  - Neue Programmiersprachen, z.B. EcmaScript vs. JavaScript

Einflussfaktor Gesetze

- Änderungen von Gesetzen können mediale Angebote wesentlich beeinflussen, z.B.
  - verändertes Urheberrecht, vgl. Einigung YouTube mit Gema in Deutschland
  - Änderung in der Mitstörerhaftung, vgl. urbane offene WLANs

Vorgehensweise zur technischen Projektkonzeption

1. Technologie beeinflussende Aspekte dokumentieren und bewerten
   - Entwickler / Zulieferer
   - Unternehmen
   - Markt
   - Zeit
   - Gesetze
2. Ausschlusskriterien für Technologien dokumentieren
3. Technologien ausschließen

Wissen über das Zielsystem erfassen

Warum Wissen formulieren?

- Software soll nach präzisen Vorgaben mit einem spezifizierten Funktionsumfang entwickelt werden.
  - überprüfbares / testbares Verhalten
  - verschiedene Implementierungen (z.B. iOS / Android) verhalten sich identisch

- Software soll auf zukünftige, beliebige Plattformen migriert werden
  - zu erwarten: Wechsel des Programmierparadigmas
  - alter Quellcode ist als Vorlage untauglich

Wie kann Wissen formuliert werden?

- Klassenmodelle der Unified Modeling Language (UML) sind untauglich
  - Selbst objektorientierte Sprachen lassen sich nicht über Klassenmodelle migrieren.
  - Bei neuem Programmierparadigma (z.B. objektorientiert – hybrid) sind Klassenmodell und alter Quellcode wertlos.

- Wissen kann über die Bedienung einer Software beschrieben werden.
  - Es gibt viele Ansätze, Geschäftsprozesse zu formulieren.

Kern des Wissens: Abläufe

- Aus Geschäftsprozessen werden sukzessiv Abläufe generiert.
  - z.B. Aktions-/Aktivitätsdiagramme mit der UML

- Aus Abläufen werden Bestandteile des Zielsystems entwickelt, u.a.:
  - Klassen und Methoden
  - Schnittstellen
  - Verteilungs- und Kommunikationsmodelle
  - Testspezifikationen

Vorgehensweise zur technischen Projektkonzeption

1. Nutzung eines geeigneten Verfahrens zur Ablaufbeschreibung

1. Erfassung von Schnittstellen zu Kommunikation
   a) über Netzwerke
   b) zu Anwendern
   c) mit Fremdsystemen

2. Erstellen von Testszenarien
   – Tests sind später Basis zur Erstellung von Prototypen der Machbarkeitsanalyse / -studie

Technische Machbarkeit untersuchen

Machbarkeitsstudien erstellen

- Zur Erstellung von Machbarkeitsstudien bei Technologieentscheidungen gibt es keine Vorgehensweise nach Lehrbuch.

- Die Fragestellung der Machbarkeitsstudie:
  - Sind alle Anforderungen mit der gewählten Technologie tatsächlich umsetzbar?

- Zu lösende Aufgabe:
  - Alle Anforderungen, von denen fraglich ist, ob sie in der Technologie machbar sind, unbedingt verifizieren!

Der Weg zur Machbarkeitsstudie

- Ausgangspunkt: Ablaufbeschreibung
- Für jeden beschriebenen Ablauf muss untersucht werden, ob er sich mit der Technologie umsetzen lässt.
- Erst wenn alle Abläufe sicher implementierbar sind, kann die Entscheidung für die Technologie erfolgen.

Reduktion des Aufwandes

- Bei Abläufen mit völlig gleichen Kommunikationsschnittstellen genügt ein einziger Repräsentant.
  - Die Menge aller zu untersuchenden Abläufe reduzieren.
  - Teilweise gleiche Kommunikationsschnittstellen genügen dabei nicht, es müssen in diesem Fall beide Abläufe untersucht werden.

Typische Fehler bei der Machbarkeitsstudie

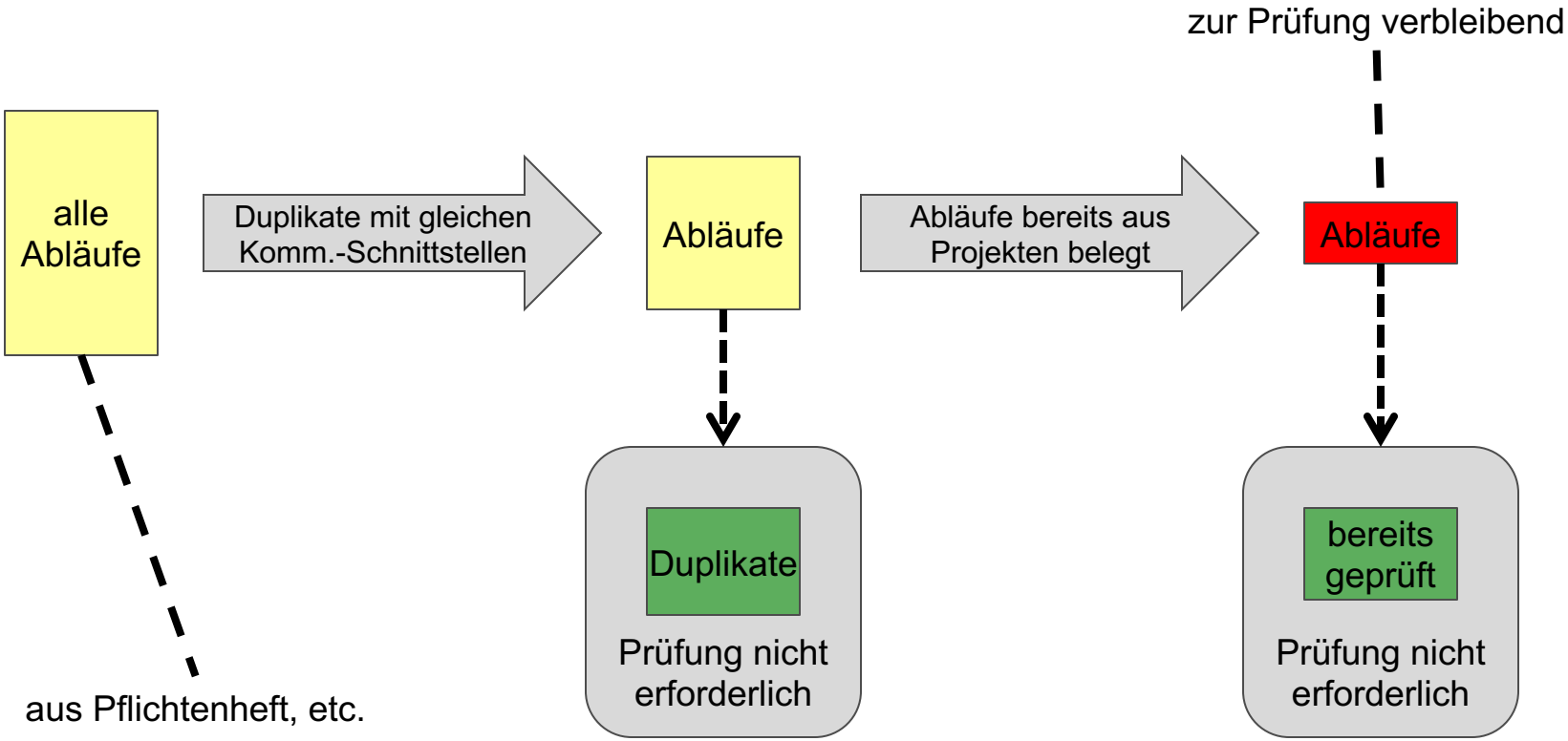
- Machbarkeit wird oft zu früh positiv bewertet!
  - durch Beispielcode in Foren
  - durch Tutorials vom Technologienanbietern oder aus anderen Informationsquellen

- Auch wenn Beispielprogramme vielversprechend aussehen, muss die Machbarkeit selbst untersucht werden!
  - Änderungen durch neuere Versionen können auch negative Folgen haben.
  - Programme funktionieren nicht mit dem ausgewählten Betriebssystem.

Vorgehensweise zur technischen Projektkonzeption

- Schritte der Machbarkeitsstudie
  1. Alle Abläufe in einer mit einem geeigneten Beschreibungsverfahren erfassen
  2. Aus Abläufen mit identischen Kommunikationsschnittstellen einen einzigen Repräsentanten in der Machbarkeitsstudie belassen.
  3. Für jeden verbleibenden Ablauf einen Prototypen implementieren bzw. mit bereits bestehender Implementierung aus anderen Projekten bei gleichen Kommunikationsschnittstellen belegen.

Vorgehensweise zur technischen Projektkonzeption



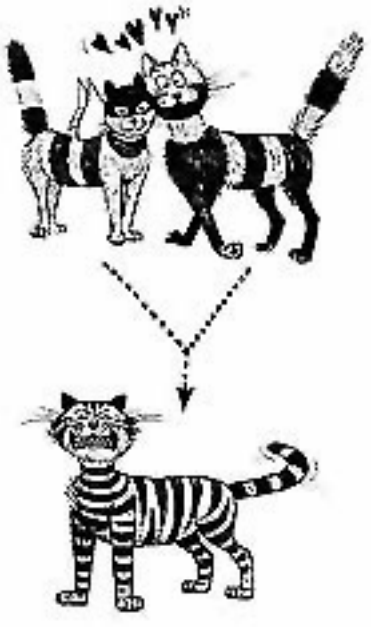
Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de

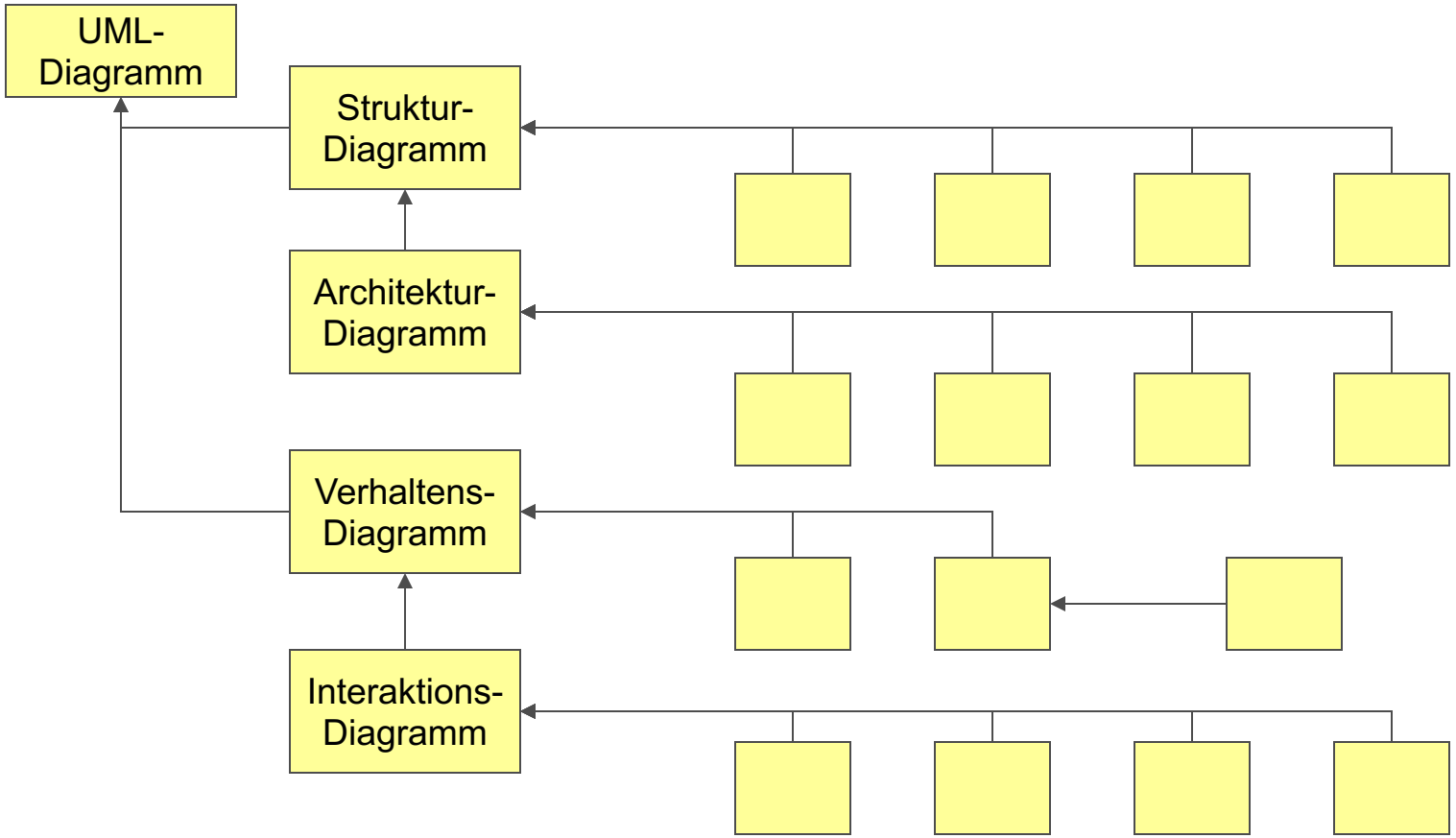
Software Engineering 2
Unified Modeling Language (UML)



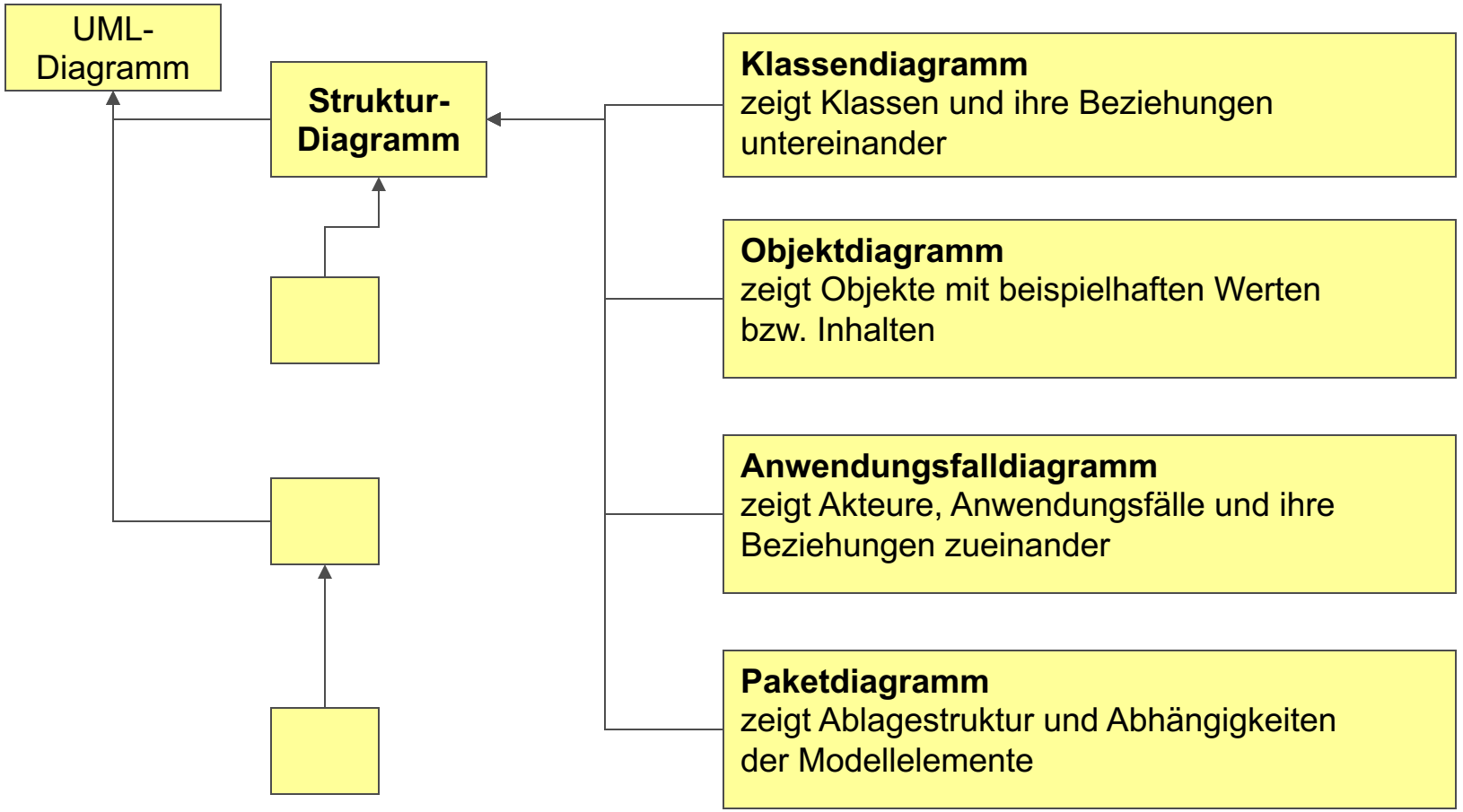
Prof. Dr. Andreas Judt

DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg

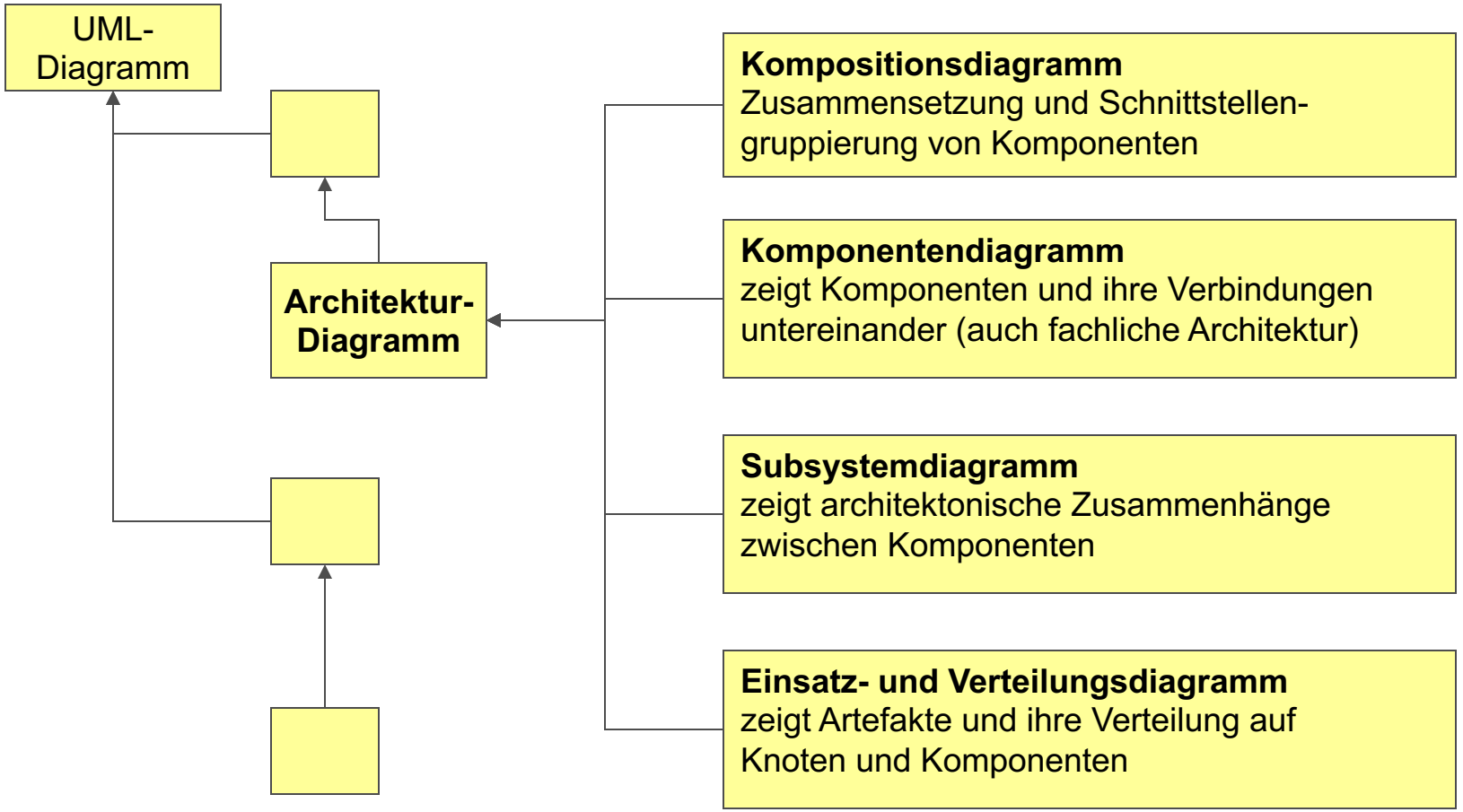
UML Diagramme (Auswahl)



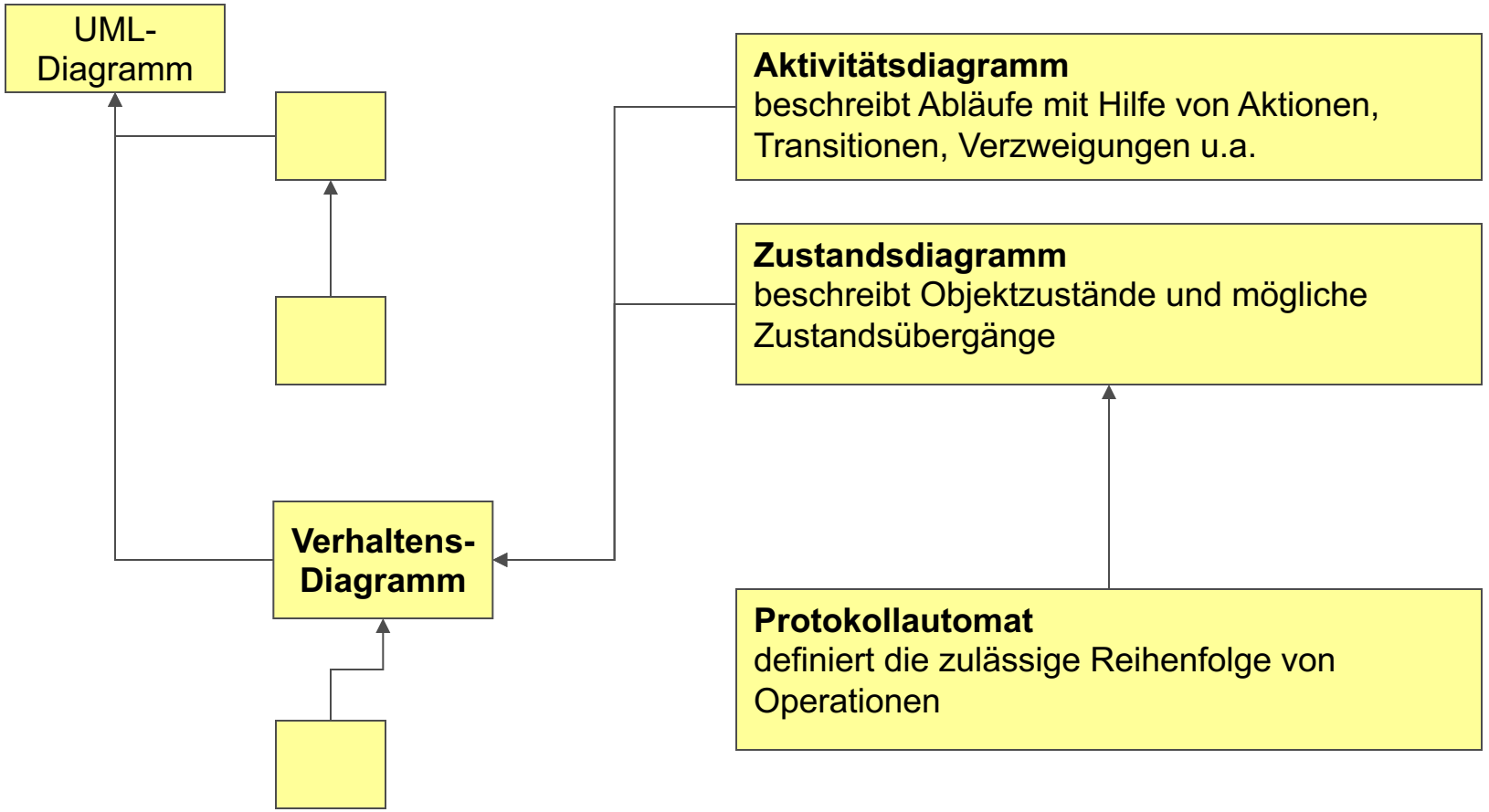
Strukturdiagramme



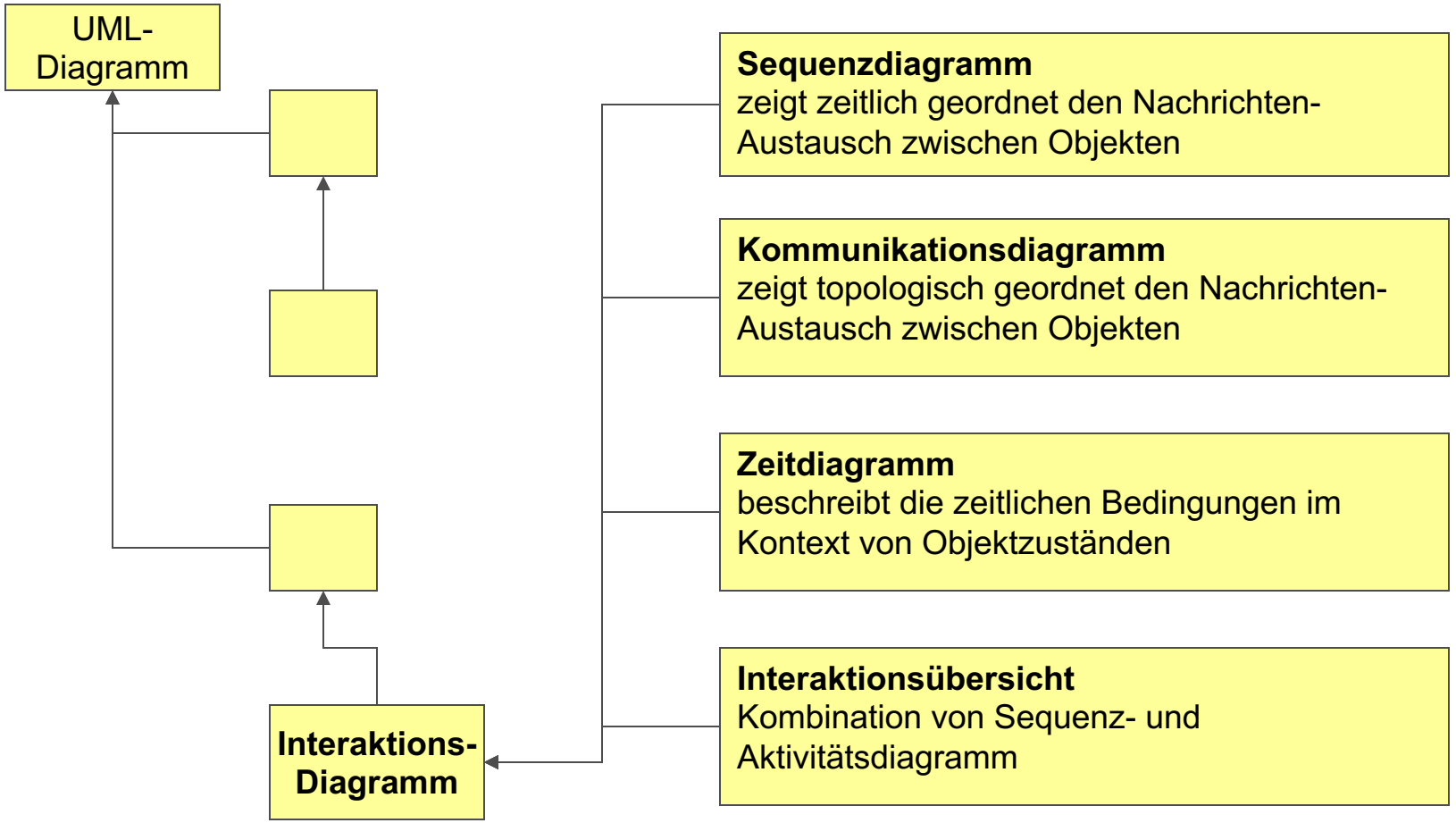
Architekturdiagramme



Verhaltensdiagramme



Interaktionsdiagramme



Objektdiagramm (engl. object diagram)

- besitzt eine ähnliche Struktur wie das Klassendiagramm
- zeigt für einen bestimmten Zeitpunkt existierende Objekte (nicht Klassen)
  - Objektdiagramm ist ein Schnappschuss des Systems zu einem bestimmten Zeitpunkt
- beschreibt insbesondere die Objektzustände mit ihren Attributwerten

Anwendungsfalldiagramm (engl. use case diagram)

- verwandte Begriffe
  - engl. use case model
  - Nutzungsfalldiagramm

- Ein Anwendungsfalldiagramm zeigt Akteure, Anwendungsfälle und ihre Beziehung zueinander.
  - zeigt, wie mit einem Anwendungsfall umgegangen werden soll
  - beschreibt keine Abläufe, sondern nur Zusammenhänge!

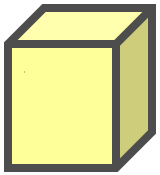
- spezielle Anwendungsfalldiagramme
  - Systemkontext-Diagramm

Akteur (engl. actor)

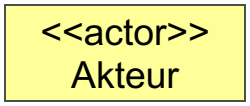
- verwandte Begriffe
  - engl. stakeholder
  - Beteiligter, Systemakteur, Ereignis, externes System, Dialog
- Akteure besitzen eine Multiplizität
  - Standardwert: 0..1 (wenn nicht explizit angegeben)
- Akteure sind in Anwendungsfalldiagrammen zu finden.

Akteure

menschlicher Akteur



Fremdsystem als Akteur



Akteur allgemein

Zeit-Ereignis

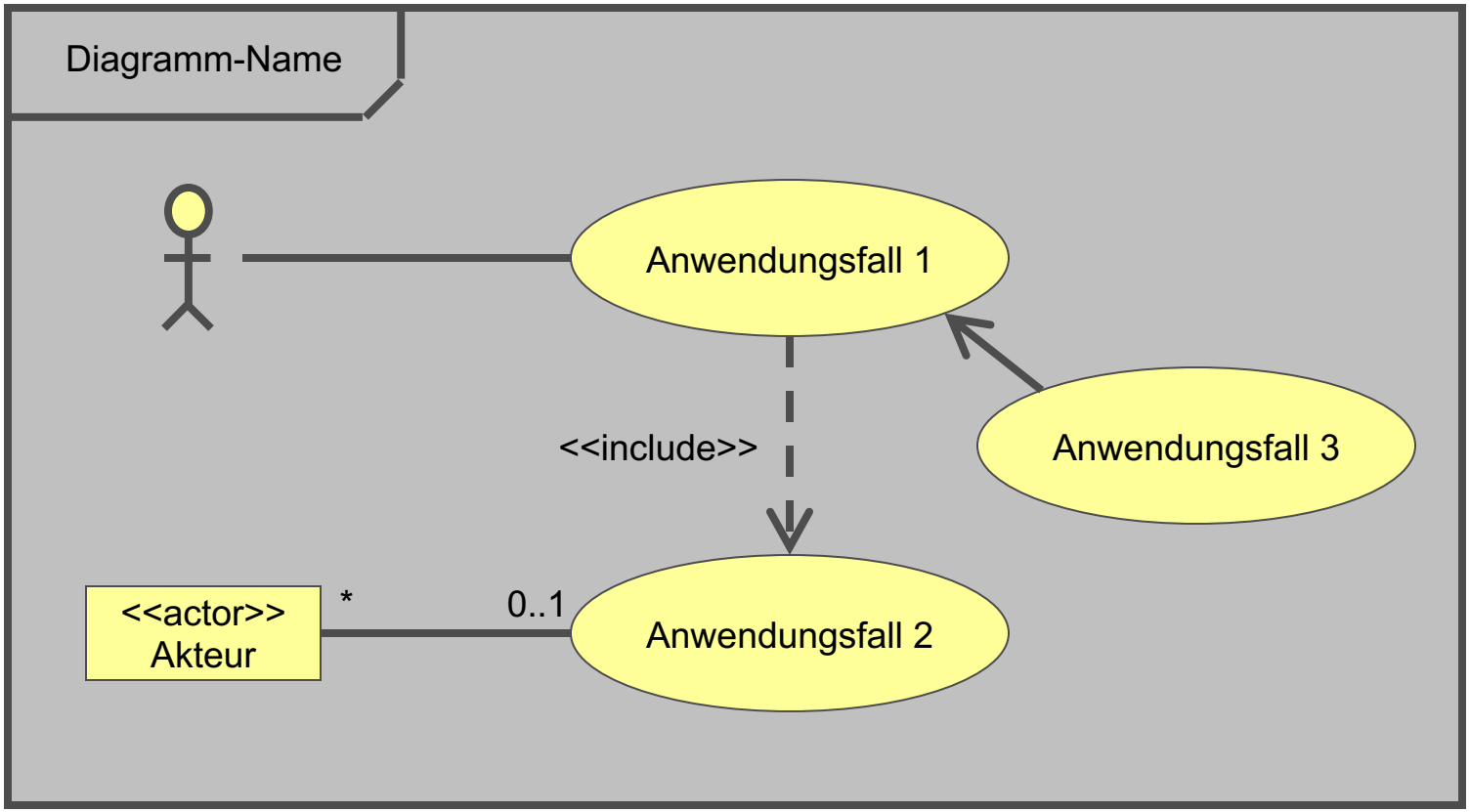
Klassendiagramm

- wurden bereits im Studium behandelt

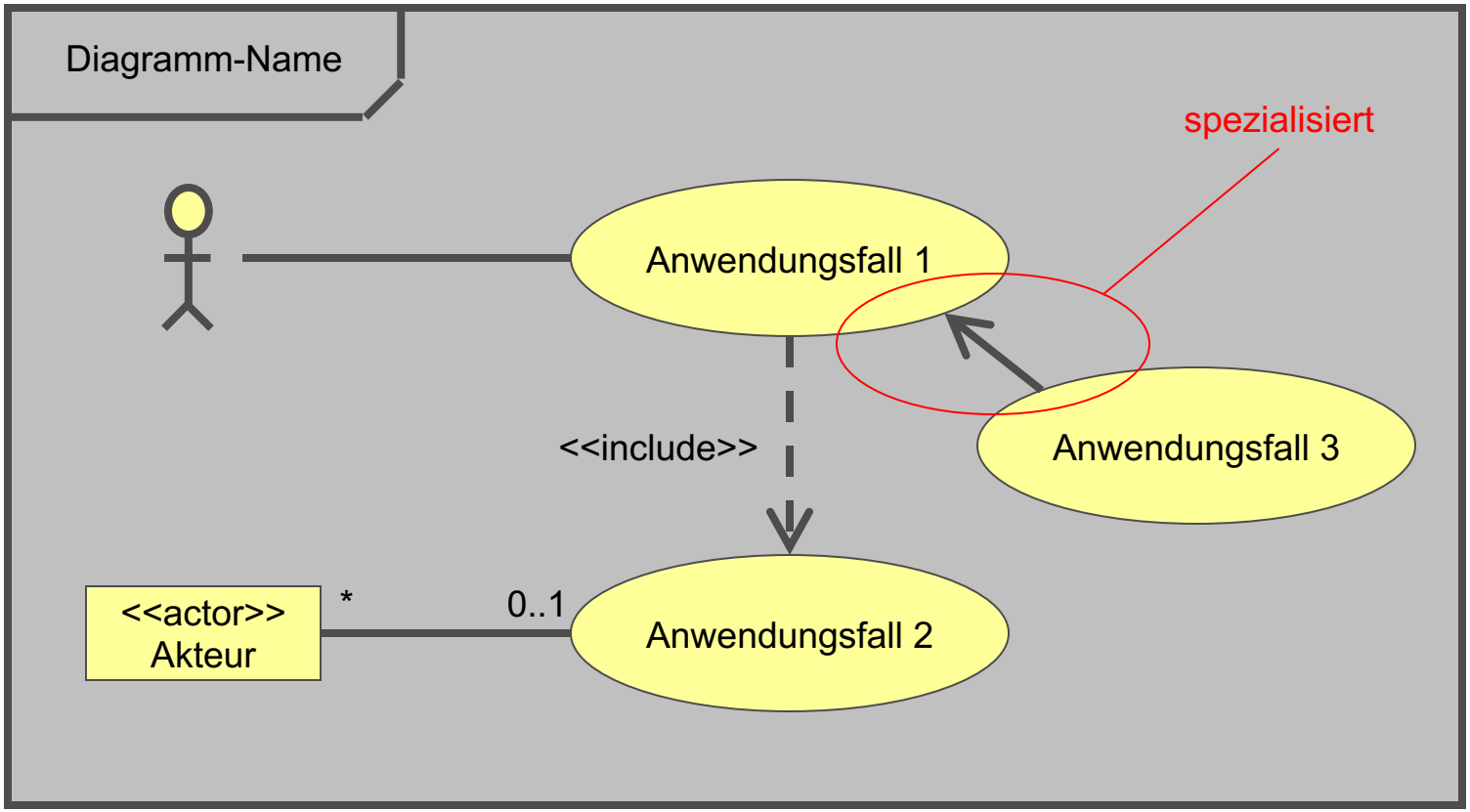
Anwendungsfalldiagramm

- spezielle Anwendungsfälle
  - Geschäftsfall
    - geschäftlicher Ablauf, der von einem geschäftlichen Ereignis ausgelöst wird und ein geschäftliches Ergebnis besitzt
  - Systemfall
    - Anwendungsfall, der speziell für außen stehende Akteure ein wahrnehmbares Verhalten beschreibt
  - abstrakter Anwendungsfall
    - Verallgemeinerung ähnlicher Anwendungsfälle
  - sekundärer Anwendungsfall
    - unvollständiger Teilablauf, der Bestandteil anderer Anwendungsfälle ist

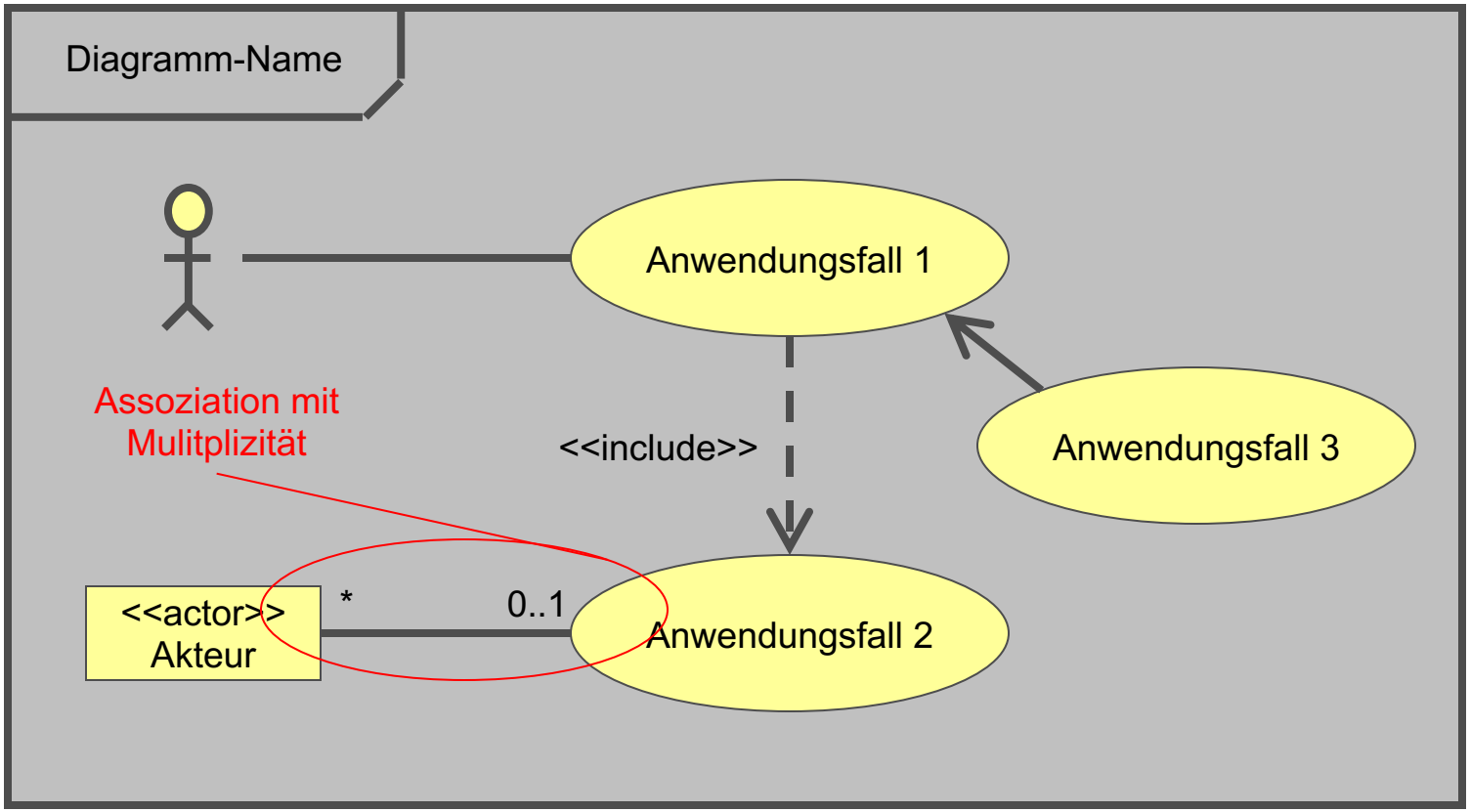
Anwendungsfalldiagramm



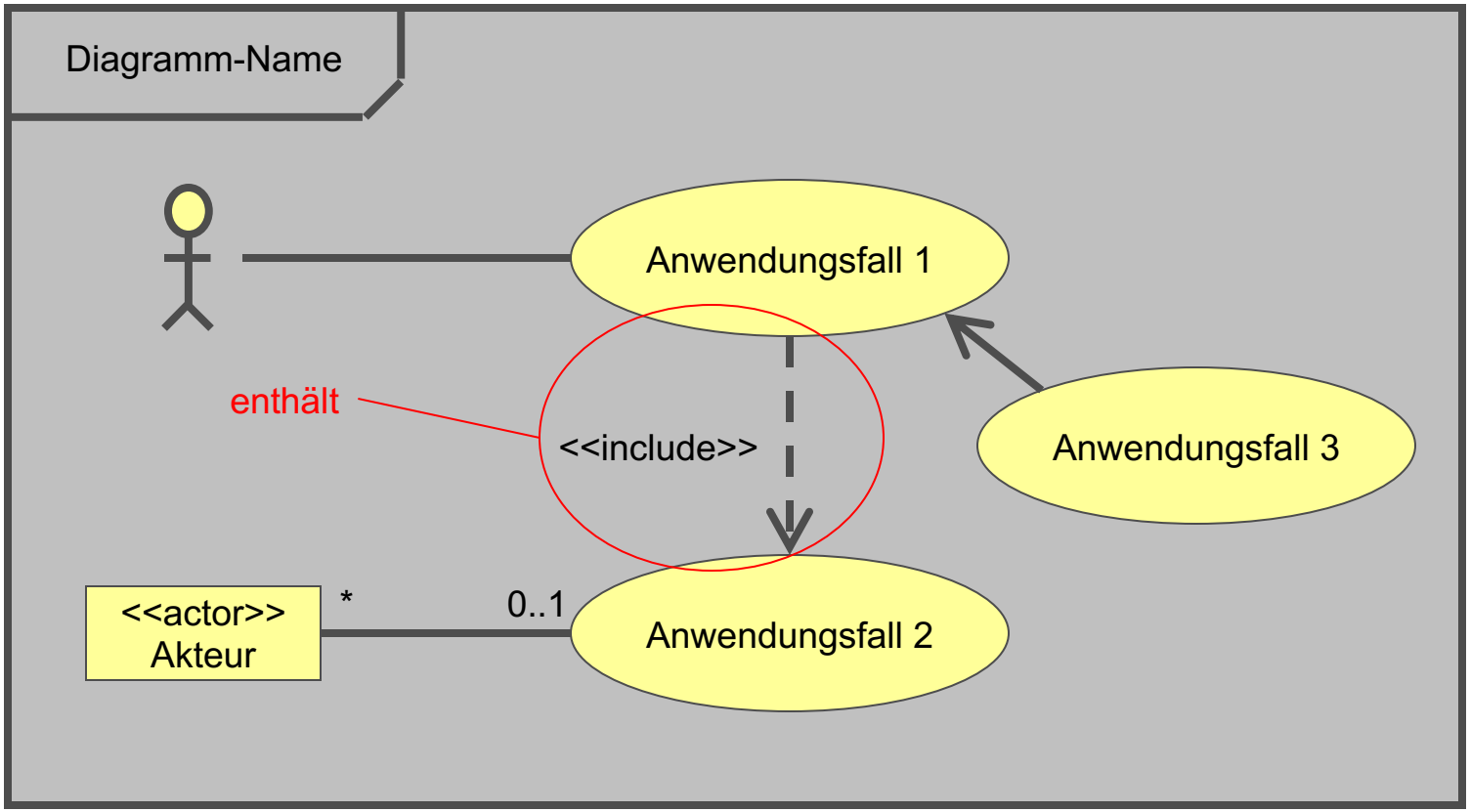
Anwendungsfalldiagramm



Anwendungsfalldiagramm



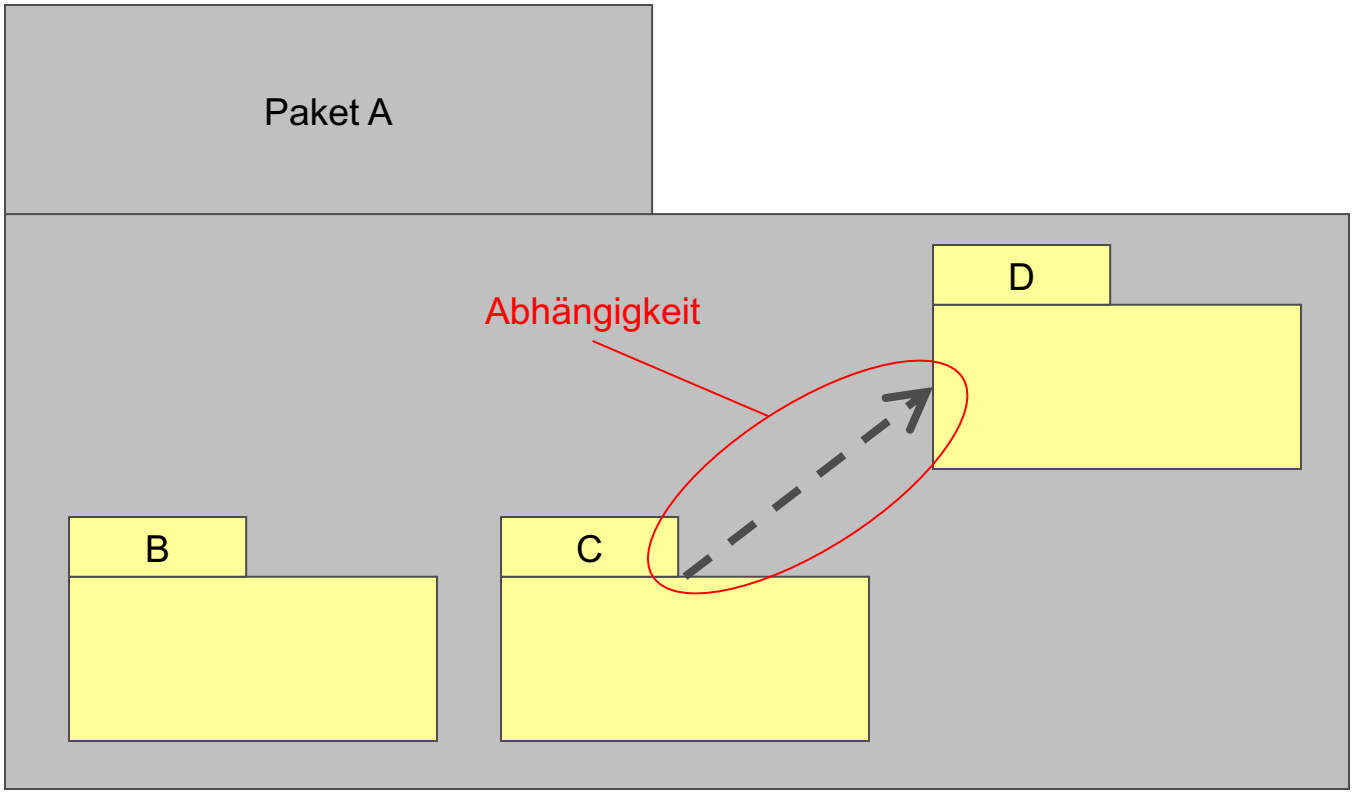
Anwendungsfalldiagramm



Paketdiagramm

- Paketdiagramme gliedern das System in überschaubare Einheiten
- Pakete können beliebige Modellteile beliebigen Typs enthalten
  - z.B. Klassen und Anwendungsfälle
- Pakete können Pakete enthalten und Abhängigkeiten besitzen
  - Abhängigkeit: ein Paket nutzt Klassen eines anderen Pakets

Paketdiagramm

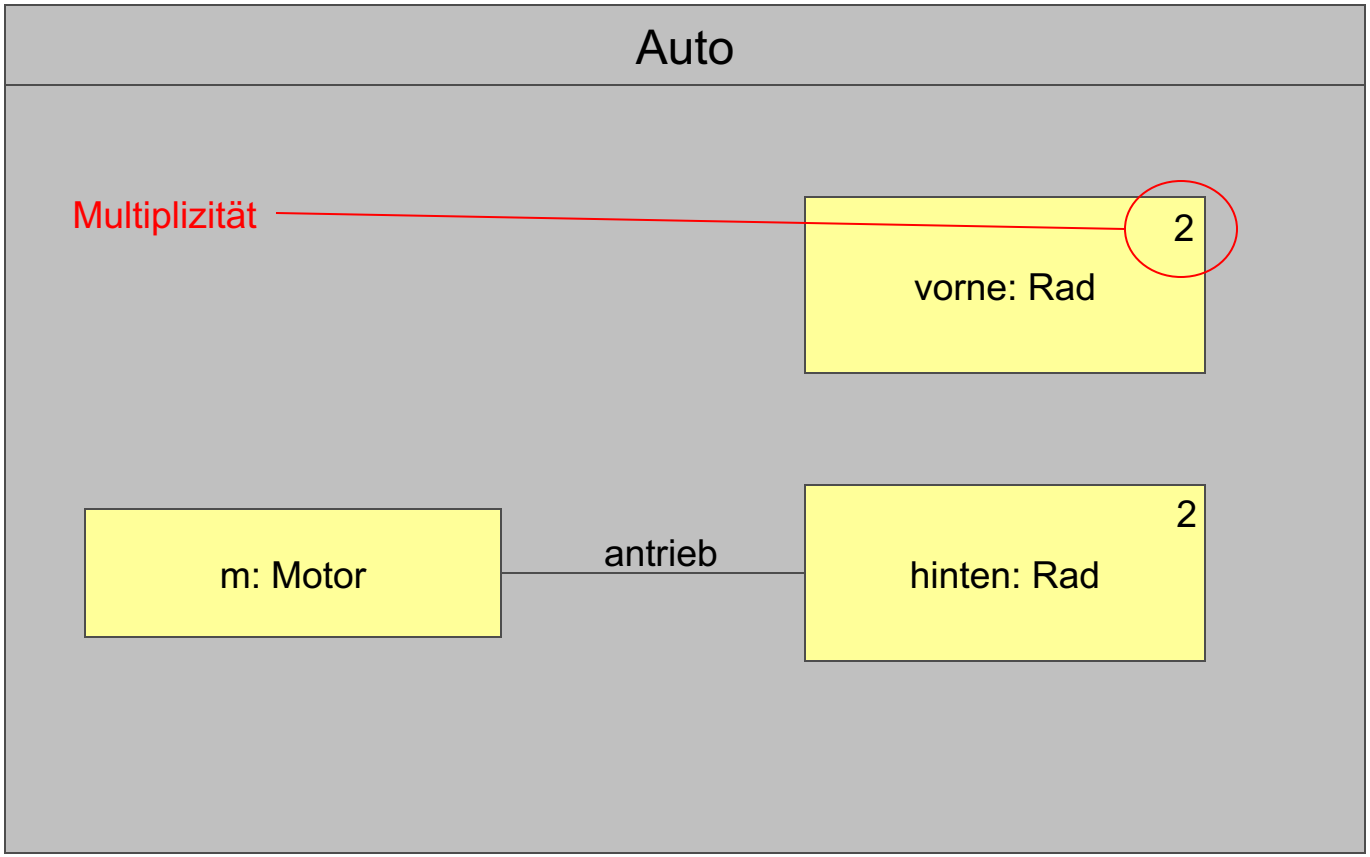


Kompositionsdiagramm

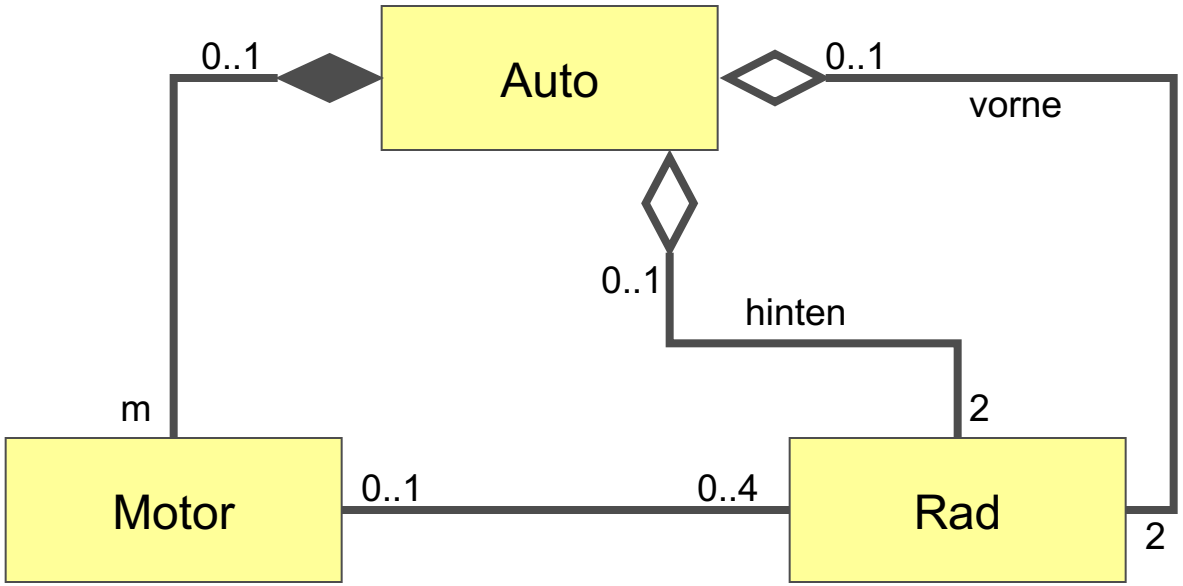
- zeigt die interne Zusammensetzung einer Komponente oder Klasse
  - Teile (engl. parts)
  - detaillierte Informationen über die Verbindung der Teile

- Warum reicht ein Klassendiagramm nicht aus?
  - Kardinalitäten von Kompositionen und Aggregationen sind zu wenig präzise.
  - Bei komplexen Klassendiagrammen können fachlich falsche Kompositionen entstehen.

Kompositionsdiagramm



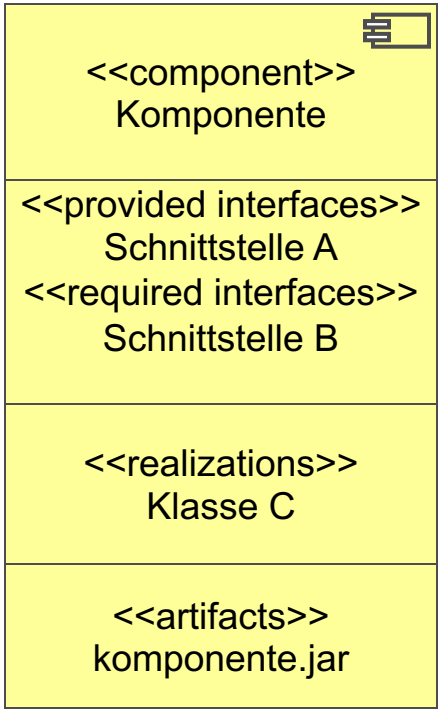
zum Vergleich: das Klassendiagramm

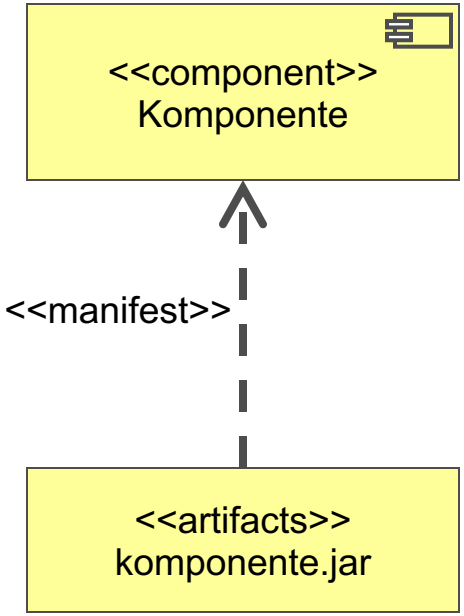


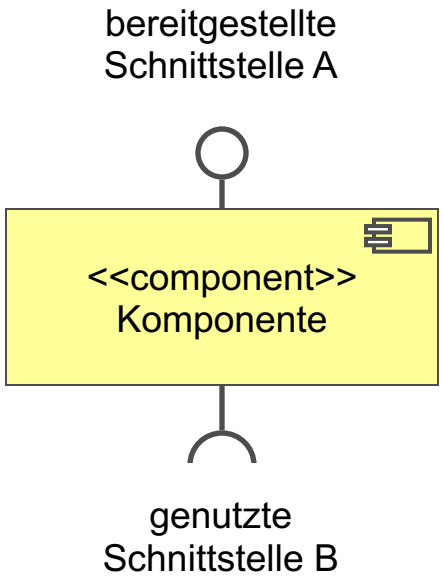
Komponentendiagramm

- Eine Komponente (engl. component) ist eine spezielle Klasse, die eine austauschbare Einheit in einem System repräsentiert, deren Bestandteile gekapselt sind.
  - Funktionalität wird über eine Schnittstelle (engl. interface) bereitgestellt
- Funktionalität, die von außen benötigt wird, wird ebenfalls über Schnittstellen definiert.

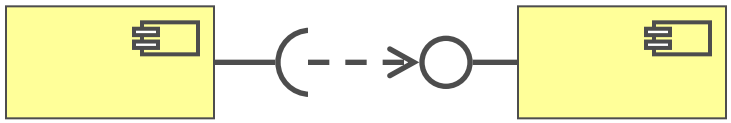
Komponentendiagramm: Notationen



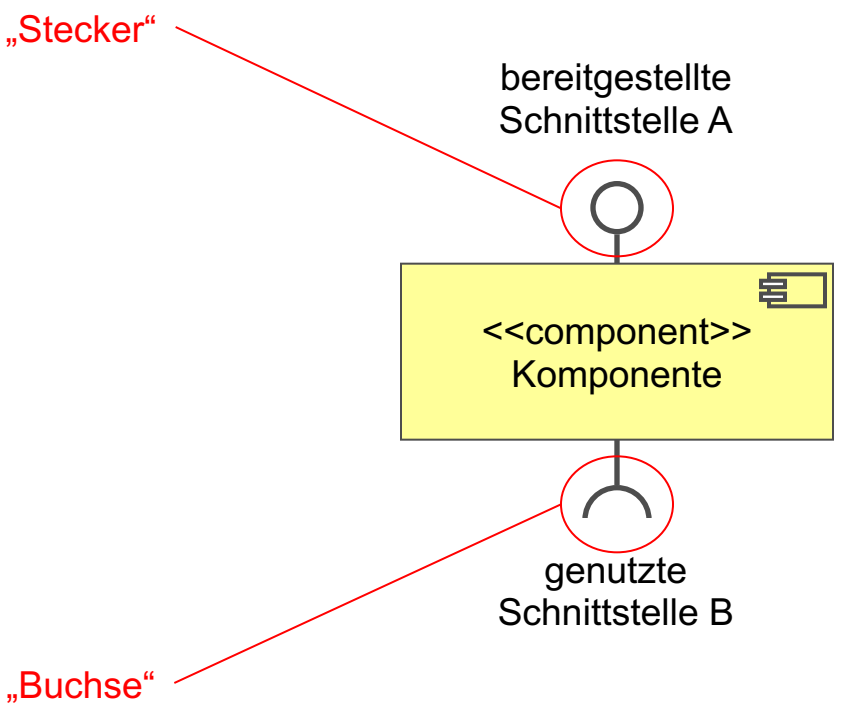




Komponentendiagramm: Notationen



Kopplung von
Komponenten



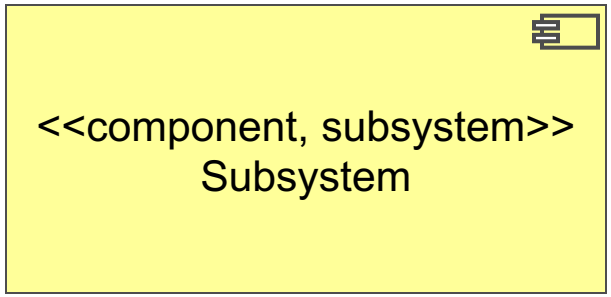
Subsystem-Diagramm

- Ein Subsystem ist eine Komponente, die eine architektonische Einheit darstellt.
  - besitzt die Merkmale einer Komponente

- Hierarchie in der Architektur

  System → Subsystem → Komponente

Subsystem-Diagramm



Aktivitätsdiagramm (engl. activity diagram)

- verwandte Begriffe
  - engl. flow chart
  - Ablaufdiagramm, Programmablaufplan, Objektflussdiagramm
- besteht aus
  - Start- und Endknoten
  - Aktions-, Kontroll- und Objektknoten
  - Kontrollflüsse
- seit UML 2.0: Aktionen, Aktionsdiagramm

Aktivitätsdiagramm: Notationen

Startknoten

Kontrollfluss

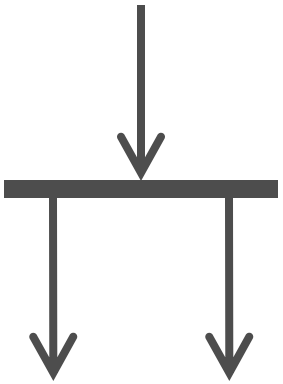
Endknoten, normaler Ablauf

Aktivität
(=Aktion)

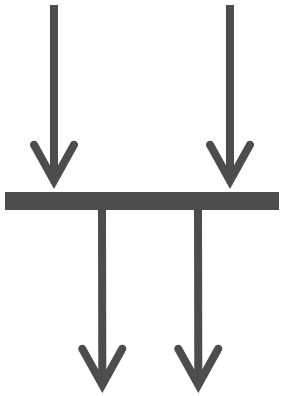
Ablaufende, z.B. Abbruch

Objekt

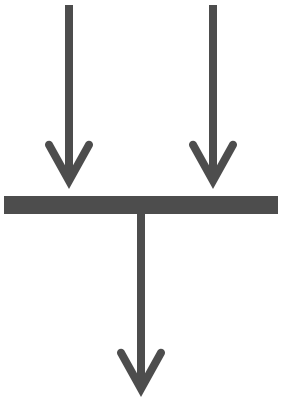
Aktivitätsdiagramm: Notationen



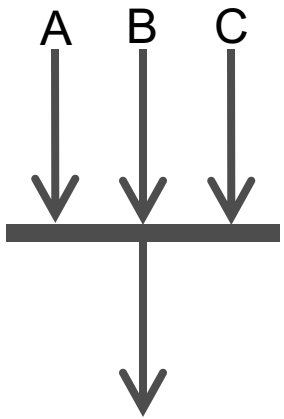
Teilung



Teilung und
Synchronisation



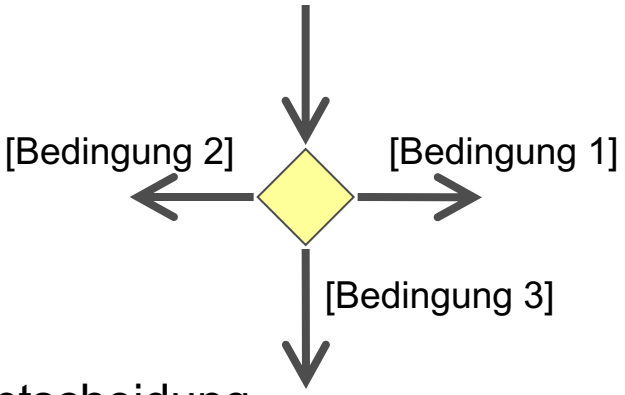
Synchronisation
(logisches UND)



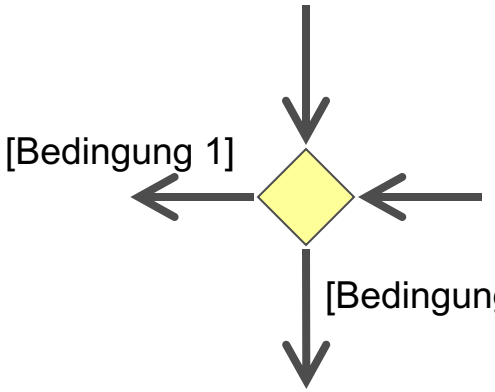
{joinSpec=(A and B) or (B and C)}

spezifizierte
Synchronisation

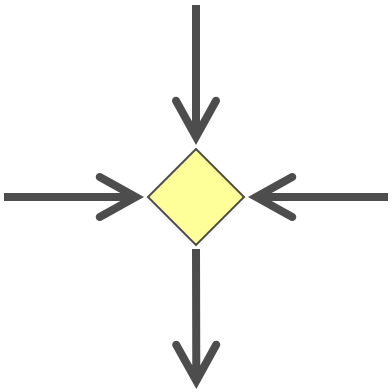
Aktivitätsdiagramm: Notationen



Entscheidung

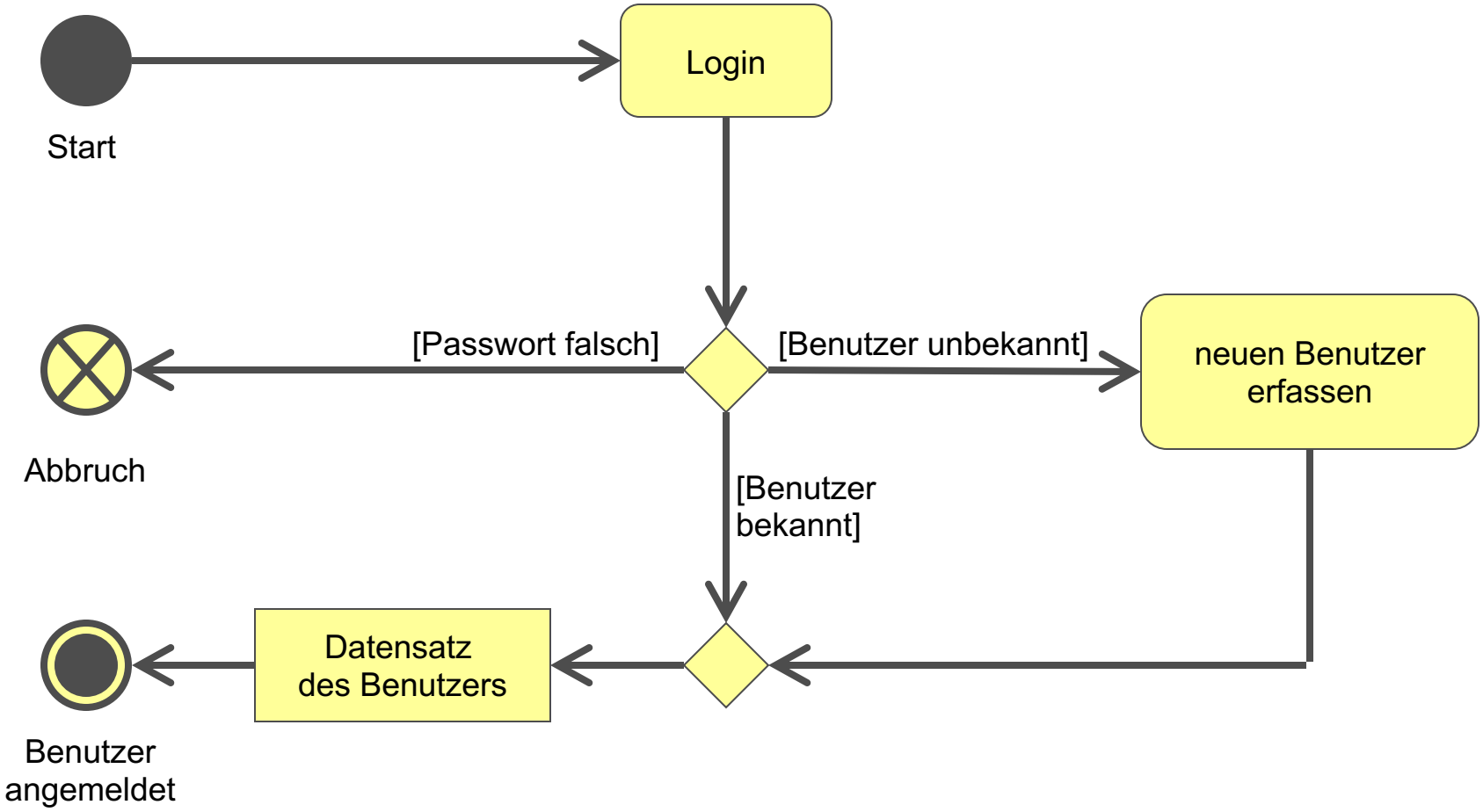


Entscheidung und Zusammenführung



Zusammenführung (logisches ODER)

Beispiel: Benutzeranmeldung



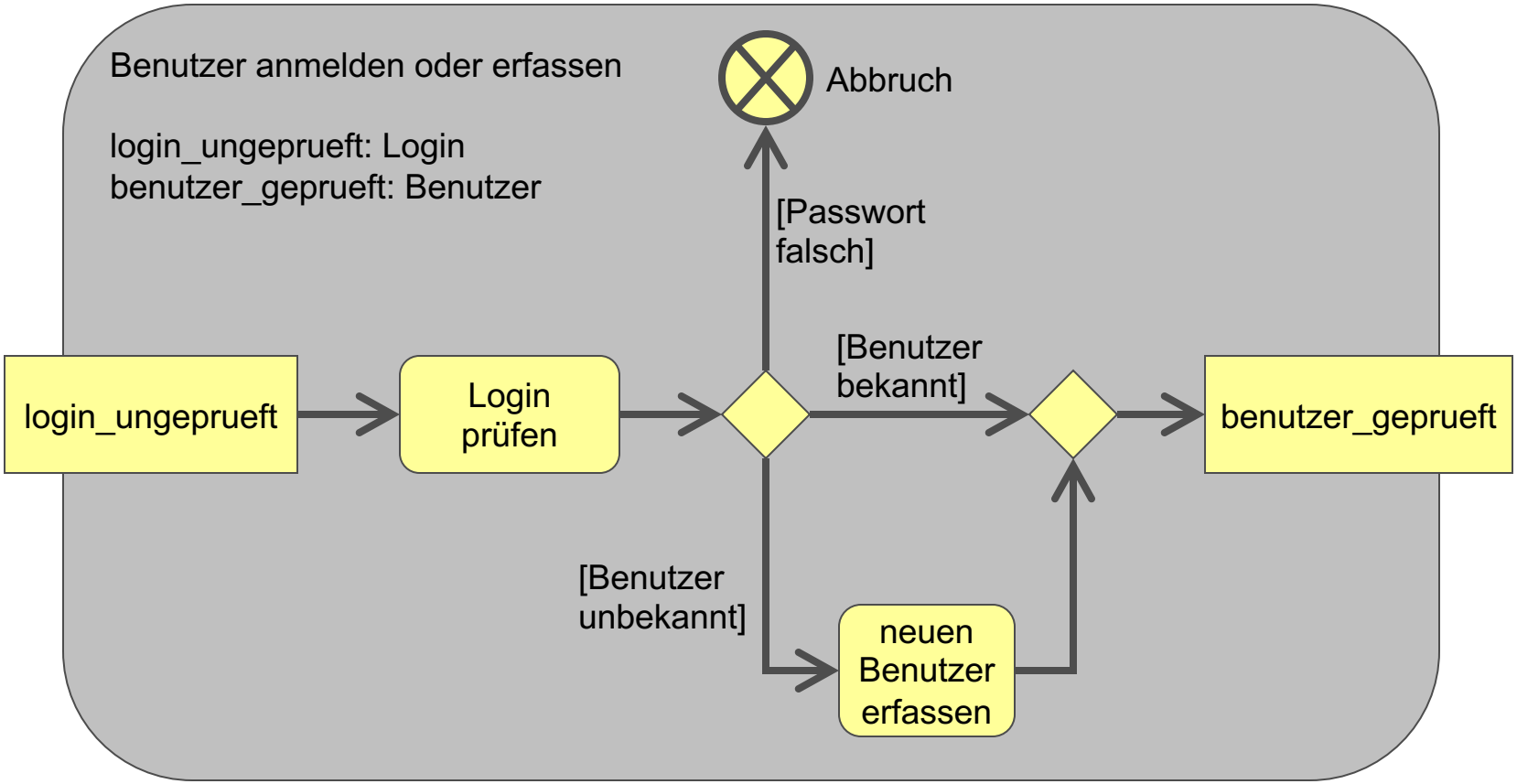
Übung

- Verfeinern Sie das Aktivitätsdiagramm der Benutzeranmeldung um folgende Eigenschaft:

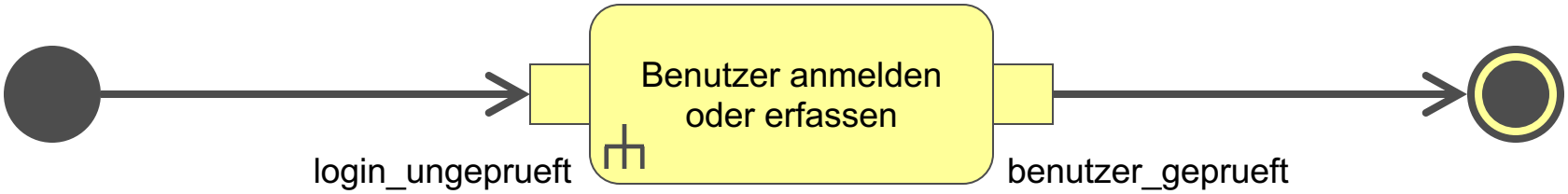
  1. Ein Benutzer hat maximal 3 Anmeldeversuche. Nach dem dritten Fehlschlag wird das Konto gesperrt.

  2. Ein Administrator kann das Konto wieder entsperren.

Aktivitäten zusammenfassen



Aktivitäten zusammenfassen



Zustandsdiagramm (engl. state diagram)

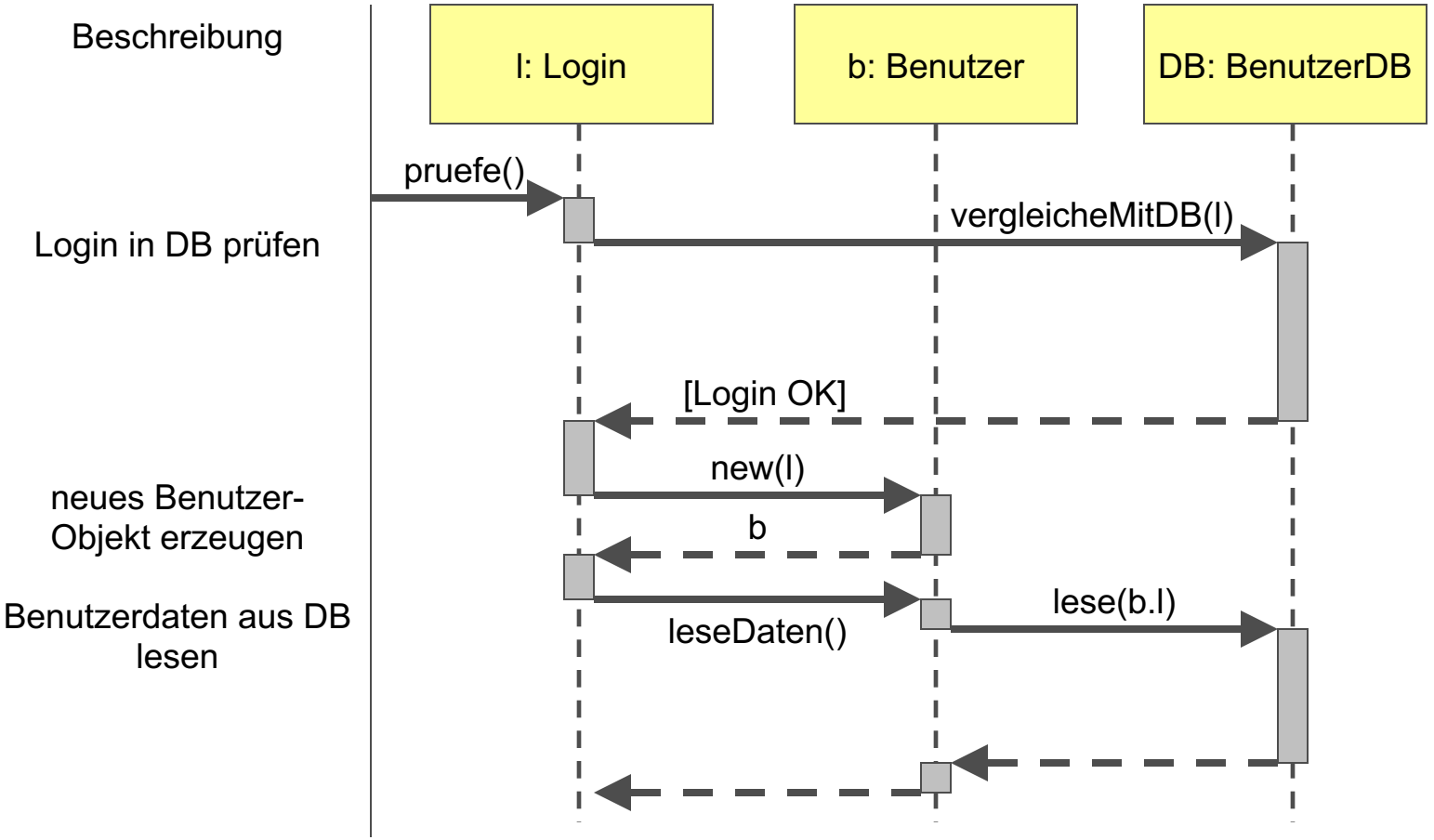
- Ein Zustandsdiagramm zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann und aufgrund welcher Stimuli Zustandsänderungen stattfinden.
- Ein Zustandsdiagramm ist äquivalent zu einem Zustandsautomaten (endlicher Automat)
- Elemente eines Zustandsdiagramms
  - endliche, nicht-leere Menge von Zuständen
  - endliche, nicht-leere Menge von Ereignissen
  - Zustandsübergänge
  - Anfangszustand
  - Menge von Endzuständen
- Notation
  - analog zu einem Aktivitätsdiagramm

Sequenzdiagramm (engl. sequence diagram)

- Eine Sequenz zeigt eine Reihe von Nachrichten, die eine ausgewählte Menge von Beteiligten (Rollen und Akteuren) in einer zeitlich begrenzten Situation austauscht, wobei der zeitliche Ablauf betont wird.
  - vergleichbar mit Kommunikationsdiagramm
    - Kommunikationsdiagramm zeigt die Zusammenarbeit der Rollen.

- Notation
  - Rollen werden durch gestrichelte senkrechte Linien dargestellt.
  - Oberhalb der Linie steht der Name bzw. das Rollensymbol.
  - Nachrichten werden als waagerechte Pfeile zwischen den Rollenlinien gezeichnet.
  - Nachrichten: nachricht(argumente)

Sequenzdiagramm



Kommunikationsdiagramm (engl. collaboration diagram)

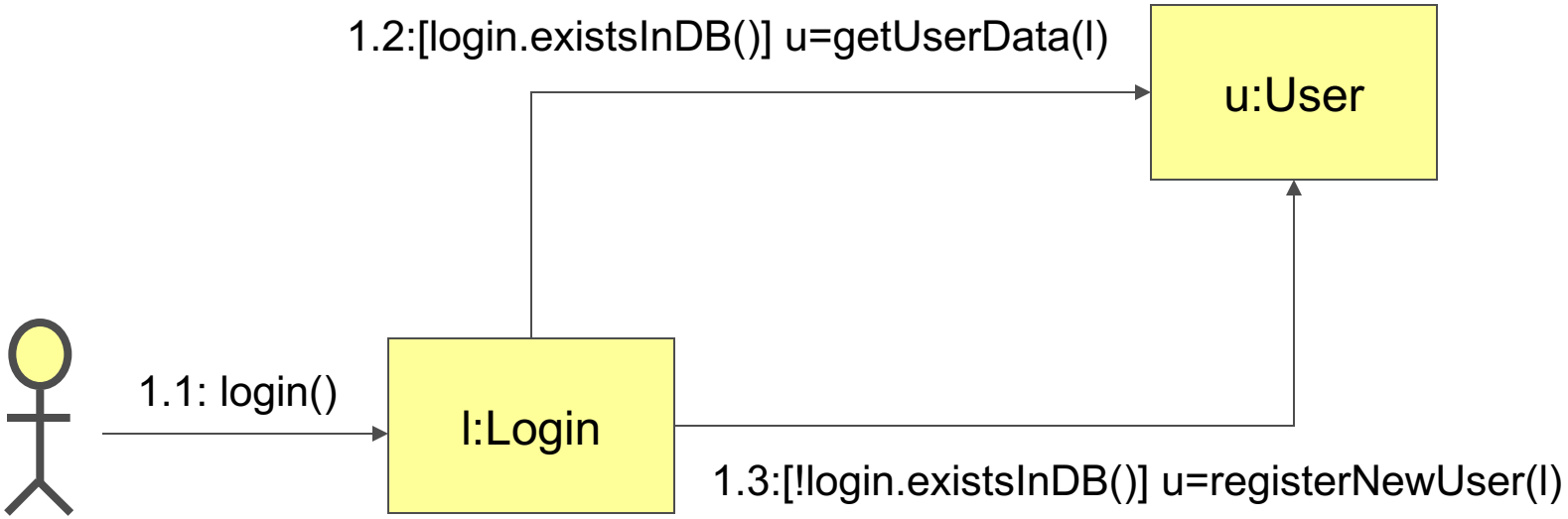
- Ein Kommunikationsdiagramm zeigt eine Menge von Interaktionen zwischen ausgewählten Rollen in einer bestimmten begrenzten Situation (=Kontext) unter Betonung der Beziehungen zwischen den Rollen und ihrer Topographie.
  - zeitliche Abfolge der Nachrichten und Antworten werden dargestellt
  - auch in Form von Iterationen und Schleifen
- Ein Kommunikationsdiagramm ist eine Projektion des dahinter stehenden Gesamtmodells und ist zu diesem konsistent.

Kommunikationsdiagramm (engl. collaboration diagram)

- Unterschied zum Sequenzdiagramm
  - Iterationen und Schleifen sind leichter darstellbar
  - Mehrere Kommunikationspfade (= mehrere Sequenzen) können durch geeignete Nummerierung dargestellt werden.

- Gemeinsamkeit mit Sequenzdiagramm
  - beide Diagramm können Ablaufvarianten beschreiben
  - beide Diagramm eignen sich in der Praxis nicht zur vollständigen Beschreibung des Systemverhaltens
  - zeitlicher Verlauf der Kommunikation zwischen den Rollen steht im Vordergrund
    - Nachrichten werden im Diagramm nummeriert

Beispiel für ein Kommunikationsdiagramm



Interaktionsübersicht

- Eine Interaktionsübersicht ist eine Aktivitätsdiagramm, in dem Teilabläufe durch referenzierte oder eingebettete Sequenzdiagramme repräsentiert sind.

- Die Interaktionsübersicht hilft, eine Menge von Sequenzdiagrammen mit Hilfe des umgebenden Aktivitätsdiagramms in einen zeitlogischen Kontext zu setzen.

Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Studiengangsleiter Informatik

judt@dhbw-ravensburg.de