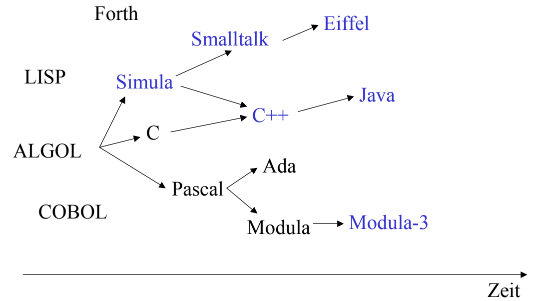
Objektorientierung

Beobachtungen bei der Softwareentwicklung:

- ähnliche Prozesse werden ständig wiederholt, das Rad wird immer wieder neu erfunden
- was der Kunde sich vorgestellt hat und was am Ende herauskommt unterscheidet sich deutlich
- Es ist schwierig abzuschätzen, wieviel Aufwand die Realisierung eines Projektes machen wird
- Für größere Aufgaben werden gerne Bibliotheken, Schablonen, vorgefertigte Teillösungen verwendet

Einige Programmiersprachen



Objektorientierung

Bemerkung:

- Auch mit „nicht objektorientierten Sprachen“ (sogar in Maschinencode) ist es möglich, objektorientiert zu programmieren
- OO-Sprachen machen OO-Programmierung leichter, und besser lesbar
- Aufgaben und Probleme sind weder OO noch nicht-OO, aber deren Analyse, Design und Implementierung
- Manche Aufgaben eignen sich mehr oder weniger als andere um mit den OO-Paradigmen behandelt zu werden

Motivation für Objektorientierung

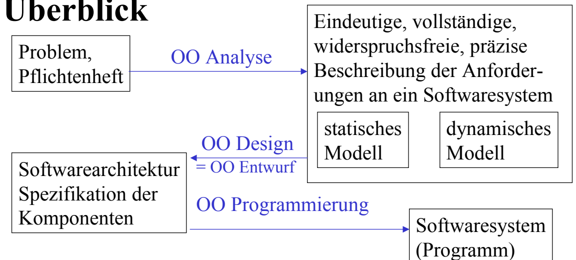
In der Regel schlechte Lösungen bei nicht OO-Sprachen für

- **Nebenläufigkeit** von Prozessen
- Konzept des **Ereignisses** (fehlt oder ist kompliziert)
- Einsatz von **Schablonen** (Templates)
- **Vererbung / Verfeinerung** von Eigenschaften
- **APIs** (application programming interface) und **Interfaces**
- **Modularisierung** und **Kapselung**
- Zusammenfassen mehrere Teile zu einem Ganzen (**Aggregation**)
- **Lesbarkeit**, **Portierbarkeit** und Fehlersuche (**Debugging**)

Ziele der Objektorientierung

- Systemanalytiker ist Schnittstelle zwischen Auftraggeber und Entwickler
- Systemanalytiker muß Auftraggeber verstehen (nicht umgekehrt)
- Beschreibung der Aufftraggeberwünsche sollte unabhängig von technischen Möglichkeiten sein (perfekter Rechner)
- Auftraggeber sollte seinen Auftrag im OO-Modell erkennen
- OO-Design (= OO-Entwurf) und OO-Programmierung werden meist beide von Entwickler(n) nicht Analytiker(n) gemacht

Objektorientierte Softwareentwicklung Überblick



- Analyse, Design und Programmierung gibt es auch ohne OO
- OO: gleiche Denkweisen in allen Entwicklungsphasen

Objektorientierte Softwareentwicklung

Programmieren ≙ Sprechen in einer „Fremdsprache“ (die der Computer versteht)

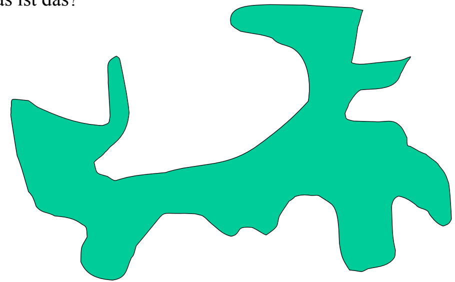
Softwareentwicklung = Produzieren eines Dokuments, mit dessen Hilfe Computer ein bestimmtes Problem lösen können

Analyse ≙ selber Verstehen, was das Problem ist

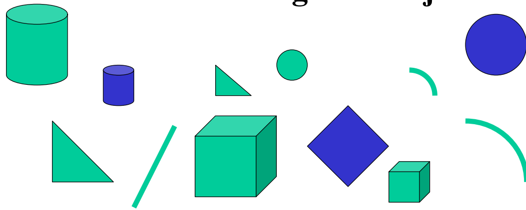
Design/Entwurf ≙ Überlegen, was man in der Fremdsprache überhaupt sagen möchte

Objekte

Was ist das?



Klassifizierung von Objekten



Zu welchen Klassen gehören diese Objekte?

- Form: rund, eckig
- Farbe: blau, grün
- Größe: groß, klein
- Dimension: Strich, Fläche, Körper
- Erscheinung: schön, häßlich

Gefahren bei OO-Programmierung

Beliebter Fehler: übertriebene Nutzung der OO-Fähigkeiten

Standard Programmierung:

```
Uhrzeit
    int stunde;
    int minute;
```

OO-Programmierung:

```
Uhrzeit:
    Stunde:
        System24:
            int stunde;
        System12:
            int stunde;
            boolean nachmittag;
    Minute:
        int zehner;
        int einer;
```

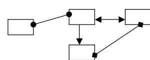

Anwendbarkeit der OO-Paradigmen

Beliebtes Gebiet: Datenbanken

- Datensätze sind Objekte einer Klasse
- großer Bedarf in der Wirtschaft

Extreme Sichtweisen für Programme:

Programm als Funktion: $\varphi_i(j)=k$ ←→ Programm als OO-System:



geeignet z.B. für Mathematik — geeignet z.B. für Büropakete

Beobachtung: großer Teil der OO-Modellierung für Datenbankprojekte

Anwendbarkeit der OO-Paradigmen

Aufgabe: Berechne GGT(x,y)

OO-Model:

:GGT
x:int
y:int
ggt:int

Restriktion: {ggt = *ggt*(x,y)}

Aufgabe: Verwaltung riesiger Kundenkartei

Funktionales Modell:

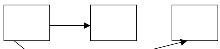
```
Karte := {Name, Adresse, ...}
Add(Kartei, Karte)
Delete(Kartei, Karte)
Search(Kartei, Karte)
```

Ergebnis der Analyse: Ein Modell

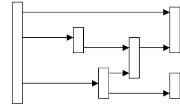
- sollte vom Systemanalysten (aber auch vom Kunden) verstanden werden
- sollte unabhängig von zu verwendender Programmiersprache und Entwicklungsumgebung sein

Also: Darstellung als Grafik

Statisches Modell



Dynamisches Modell



Standards der OO-Analyse

- Ende 80er, Anfang 90er: erste Bücher über OO-Analysis
- mehrere verschiedene graphische Notationen, zunächst nur wenige Anwender
- Bestrebungen, Standards einzuführen; Booch (91, 94), Rumbaugh (91) und andere; die Analyse unterstützende Werkzeuge werden entwickelt
- 1994: Booch und Rumbaugh einigen sich auf „**Unified Method**“
- 1996: Booch, Rumbaugh, Jacobson veröffentlichen „**Unified Modeling Language**“ (UML)

OO-Modellierung für Datenbanken

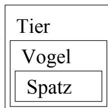
- großer Teil der OO-Arbeit für Datenbankanwendungen
- besonderes Interesse: verteilte Datenbanken (v.a. im Internet-Zeitalter)
- Internet-Quellen: URL = Uniform Resource Locator
Protokoll dazu: HTTP (hypertext transfer protocol)
- Datenbankobjekte: ORB = Object Request Broker
Protokoll dazu: CORBA (common ORB architecture)

Was sind Objekte

- entsprechen in der Regel irgendetwas aus der realen Welt
- gehören genau einer **Klasse** an
- können verschiedene **Zustände** (states) haben
- mehrere Objekte können können **verschieden**, **gleich** oder **identisch** sein
- haben ein **Verhalten** (behavior) = Menge von Operationen
- können einen Namen haben
- enthalten in der Regel Daten (**Attribute**)
- können **erzeugt** und zerstört **werden**
- können **referenziert** werden
- ermöglichen in der Regel Änderung/Auslesen ihrer Attribute
- können Attribute haben, die selbst Objekte (bzw. Referenzen auf Objekte) sind

Was sind Klassen

- beschreiben was Objekte gemeinsam haben
- definieren Objektstruktur (Attribute)
- definieren Verhalten der Objekte (Menge der Operationen)
- definieren Botschaften, auf die reagiert werden kann
- können Verallgemeinerungen / Spezialisierungen anderer Klassen sein



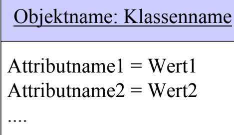
- haben in der Regel einen Namen (Substantiv)
- verwalten (erzeugen, vernichten, finden) *ihre eigenen* Objekte
- kapseln klassenspezifische Interna von der Außenwelt ab

Gründe für Kapselung

- Geheimnisprinzip
  - Unmöglichkeit der Manipulation von „außen“
  - Kenntnisse über Modul-/Klasseninterna nicht nötig
- Unabhängigkeit
  - Programmierung ohne Rücksicht auf andere Module / Klassen
  - Änderungen möglich ohne Effekt auf andere Module / Klassen
- Disziplinierung
  - weniger Spaghetticode
  - durchdachte Programme, leichtere Lesbarkeit

Darstellung von Objekten in UML

graphische Darstellung als Rechteck, geteilt in zwei Felder:



Oberes Feld:

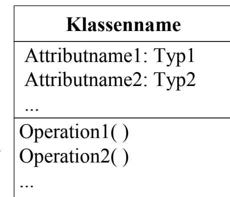
enthält auf jeden Fall den Klassennamen (anonymes Objekt), ggf. auch den Objektnamen, z.B. :Klassenname

Unteres Feld:

enthält eine Liste der (interessanten) Objektattribute mit ihren Werten

Darstellung von Klassen in UML

graphische Darstellung als Rechteck mit drei Teilen:



Oberes Feld:

Name der Klasse (nicht unterstrichen), sollte i.d.R. ein Substantiv o.ä. sein

Mittleres Feld:

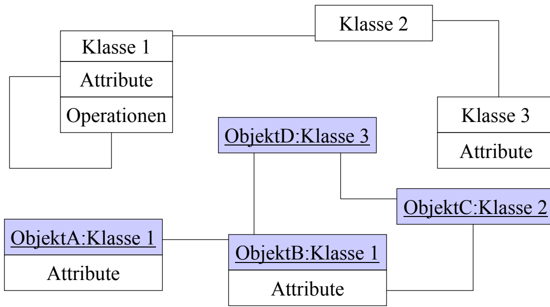
Liste der Attribute (ggf. gefolgt von „:“ und Typangabe)

Unteres Feld:

Liste der Operationen (ohne genauere Details)

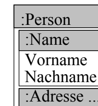
Geflechte von Objekten und Klassen

veranschaulichen Zusammenhänge der Bestandteile eines Systems

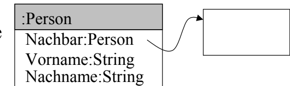


Verschiedene Arten von Attributen

- elementare Typen (auch Listen von elementaren Typen)
  z.B. int, float, boolean, ...
- Objekte
  (=> komplexe Objekte)



- Anfangswerte, Zusicherungen, Restriktionen, Invarianten
- Referenzen auf Objekte



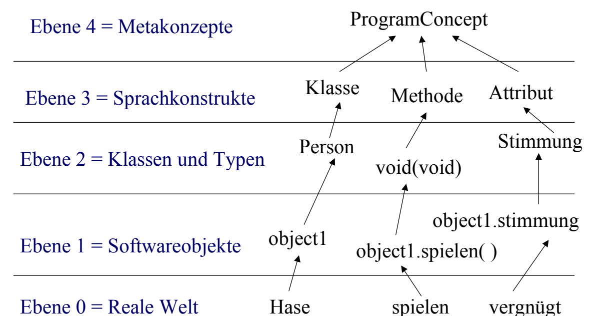
- Aufzählungstypen
  (z.B. Geschlecht: { values: M, F, other, select: 1..1, noAdd })
- Sichtbarkeit, Änderbarkeit, Änderungsrechte

Verschiedene Arten von Operationen

Es gibt:

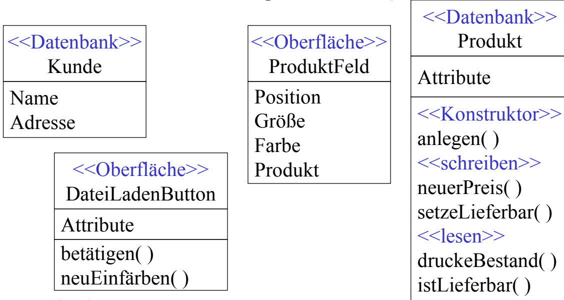
- **Konstruktoroperationen**
  erzeugen neue Objekte und initialisieren sie
  z.B. eröffne Konto ( )
- **Klassenoperationen**
  sind der Klasse zugeordnet und nicht einem bestimmten Objekt
  sie werden in UML unterstrichen dargestellt
  z.B. erhöhe Überziehungszins ( )
- **Objektoperationen**
  (auch kurz „Operationen“), beziehen sich immer auf genau ein Objekt - dazu gehören auch Operationen, die Objekte löschen
  z.B. löse Konto auf ( ) oder überweise Geld ( )

Verschiedene Modellierungsebenen



Stereotypen

dienen der Zusammenfassung irgendwie zusammengehörender Elemente (z.B. ähnliche Klassen, ähnliche Operation, u.s.w.)



Ereignisse

Standadrdbegriff des Algorithmus kommt ohne Ereignisse aus:
in der Regel genügen:

- Prozedur-/Funktionsdefinition
- Prozedur-/Funktionsaufruf
- Zuweisung von Ausdrücken zu einer Variablen
- Bedingte Verzweigung des Programmablaufs
- keine nebenläufigen Aktivitäten

Objektorientiertes Modell (meist):

- kein fest definierter Programmablauf
- Aktionen finden nur aufgrund von Ereignissen statt
- Nachrichten verschicken entspricht Methodenaufruf
- beliebig viele (teils implizite) nebenläufige Aktivitäten

Ereignisse

Standardalgorithmus:

```
START:
    tue dies
    tue das
    wenn x=y
        tue jenes
    sonst
        tue dieses
        springe zu ENDE
    springe zu START
ENDE:
    drucke ergebnis
```

Objektorientiert:

```
START:
      erzeuge diese Objekt
      erzeuge jenes Objekt
-----------------------------
dieses Objekt:
      falls Mausklick hier
            tue das
      falls diese Nachricht
            tue jenes
-----------------------------
jenes Objekt:
      ...
```


Kommunikation in OO-Systemen

Nachrichten (Methodenaufruf) u. Zugriff auf Werte (Attribute) verlaufen **synchron**:

Methodenrumpf wird sofort nach den Aufruf ausgeführt

Behandlung von Ereignissen verläuft **asynchron**:

Ereigniserzeuger übergibt dieses an das Laufzeitsystem, diese puffert es bis der Verbraucher / Abhörer bereit ist.

Typische Anwendung: Benutzeroberfläche, bei der der Benutzer Ereignisse erzeugt.

Objektorientierte Analyse

Typische Szene:
Kunde braucht Softwaresystem zur Lösung bestimmter Probleme.
Softwareentwickler hört zu und versucht Modell zu entwickeln.

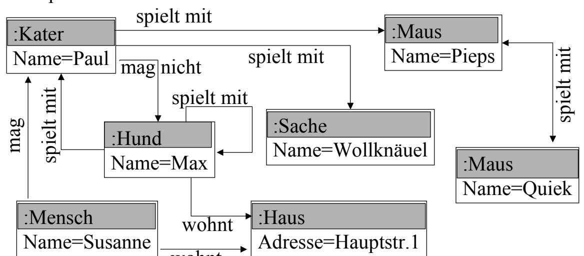
Situation birgt die Gefahr, daß

- Sachverhalte (Klassen, Objekte, Methoden, Attribute) übersehen und falsch eingeschätzt werden
- Zusammenhänge (Relationen zwischen Klassen / Objekten) nicht erkannt werden
- unnötige Zusammenhänge gesehen werden, unnütze Klassen-definitionen u.ä. gemacht werden

Deshalb wünschenswert: **Methodisches Vorgehen**, **Checklisten**

Assoziationen

Sind Verbindungen verschiedenster Art zwischen Objekten.
Beispiele:

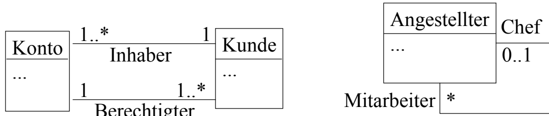


Assoziationen zwischen Klassen

Typischerweise: Assoziation zwischen Objekten (beliebiger Klassen)

Assoziationen zwischen Klassen beschreiben, wie Objekte dieser Klassen assoziiert sind:

Beispiel:

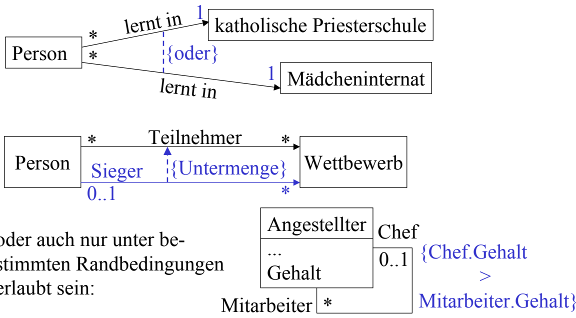


Eigenschaften von Assoziationen

- Wertigkeit (wie viele Objekte sind assoziiert, normal: zwei)
- Richtung:
  - reflexiv (selbes Objekt/Klasse)
  - bidirektional (entspricht „kommutativ“)
  - unidirektional (nicht umkehrbar)
- Kardinalität (wie viele „Ziel-/Quellobjekte mindestens/höchstens)
- Rolle (was bedeutet die Assoziation, Beschriftung der Verbindung)
- Restriktionen (einzuhaltende Bedingungen)

Eigenschaften von Assoziationen

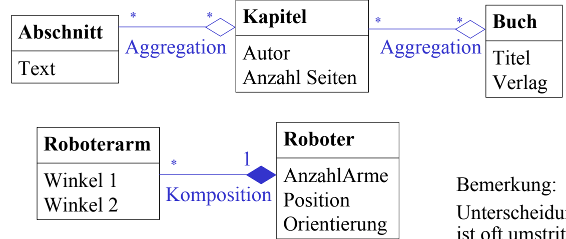
Assoziationen können auch untereinander in Beziehung stehen:



oder auch nur unter bestimmten Randbedingungen erlaubt sein:

Aggregationen und Kompositionen

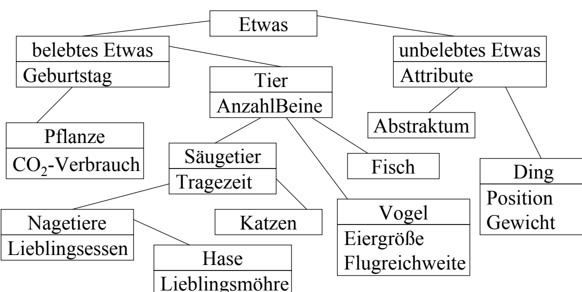
sind spezielle Formen von Assoziationen, drücken immer eine „ist-Teil-von-Beziehung“ aus (gerichteter Graph ohne Zyklen):



Bemerkung: Unterscheidung ist oft umstritten

Vererbung

- handelt von **allgemein**e(re)n und **spezialisiert**e(re)n Klassen
- definiert eine **Klassenhierarchie**

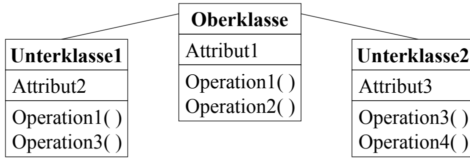


Vererbung

- Eine spezialisierte Klasse „erbt“ alle Eigenschaften aller allgemeineren Klassen
- Die spezielle Klasse ist **Unterklasse** der allgemeinen
  Die allgemeine Klasse ist **Oberklasse** der speziellen
- Unterklassen haben außer den Eigenschaften der Oberklasse noch weitere Eigenschaften (**Spezialisierung**)
- **Multiple Vererbung** ist möglich:
  Eine Klasse kann mehrere direkte Oberklassen haben

Vererbung und Überschreiben

Klassen können das Verhalten ihrer Oberklassen überschreiben

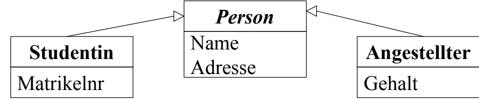


Operation1( ) ist für Unterklasse1 eine andere als für Unterklasse2, Unterklasse2 und Oberklasse haben dieselbe Operation1( ).

Bemerkung: Überschreiben ≠ Überladen = Methoden mit gleichem Namen aber unterschiedlicher Parameterliste

Abstrakte Klassen

Abstrakte Klassen sind Klassen von denen es keine Objekte gibt.



Es ist möglich, Objekte der Klasse **Studentin** oder der Klasse **Angestellter** anzulegen, aber nicht der Klasse ***Person***.

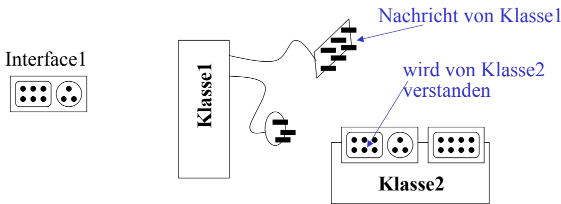
In der UML-Notation:

werden abstrakte Klassen ***kursiv*** geschrieben,
Verbindung von Unterklasse zu Oberklasse mit
——▷ gekennzeichnet

Interfaces (Schnittstellen)

Interfaces definieren einen Teil der Signaturen einer Klasse
(d.h. einen Teil ihres Erscheinens / ihrer Schnittstelle „nach außen“)

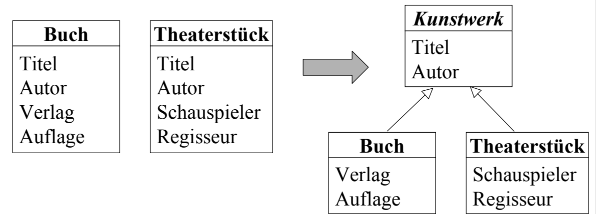
ist wie eine abstrakte Klasse ohne Assoziationen und ohne Attribute
(nur abstrakte Operationen)



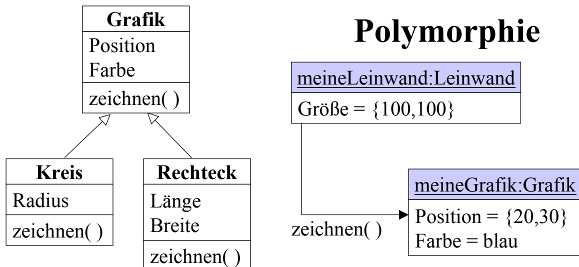
Konkretisierung und Faktorisierung

Unterklassen entstehen v.a. durch Konkretisierung (Spezialisierung) der Oberklasse, d.h. hinzufügen von Attributen und Operationen

oder durch Faktorisierung:



Polymorphie



Dynamisches Binden ist nötig, wenn erst zur Laufzeit feststeht, welche Operation tatsächlich aufgerufen werden soll.

meineGrafik kann von der Klasse **Kreis** oder **Rechteck** sein.

Pakete

dienen verschiedenen Zwecken:

- Schaffung eines neuen Freiraums für Namensgebung
- leichtere Identifizierung von Softwarekomponenten
- zusätzliche Kapselungsebene

Häufiges Vorgehen beim Java-Paketen:

1. Definieren der Signatur der Klassen
   (wie sehen sie „von außen aus“, wie verhalten sie sich)
2. Implementieren der Klassen
   („Ausfüllen“ der Klassenskelette)

Das Szenario

Schaubild von Objekten und Klassen ist nur eine statische Beschreibung
=> **statisches Modell**

Geschäftsprozesse und Abläufe können so nicht dargestellt werden.

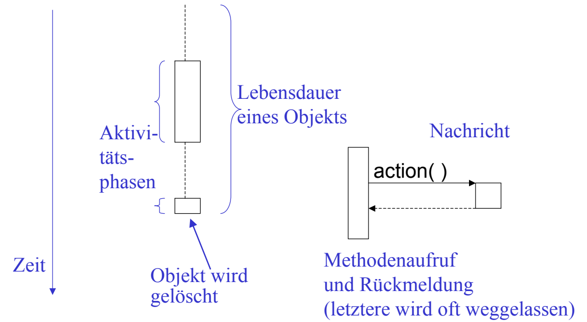
Zur vollständigen Systembeschreibung gehören noch **Szenarien**:

Ein System kann durch mehrere Szenarien beschrieben werden

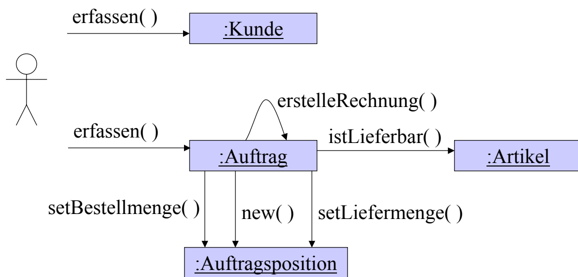
als **Kollaborationsiagramm**
ohne Berücksichtigung der zeitlichen Abhängigkeiten

oder als **Sequenzdiagramm**
zum Hervorheben des dynamischen Verhaltens

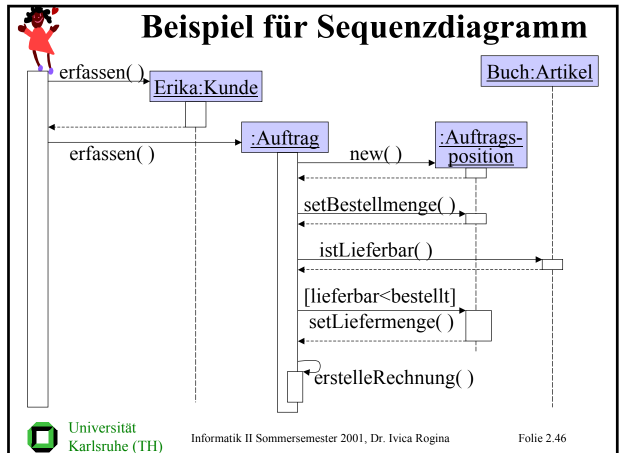
Darstellung dynamischer Modelle



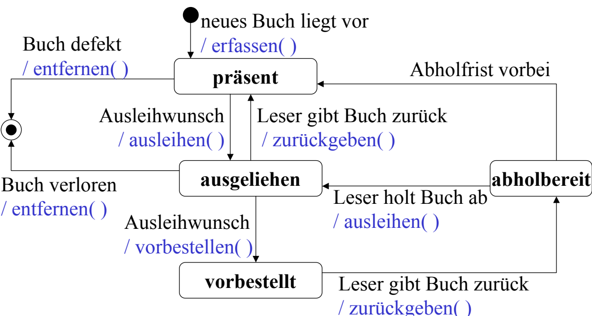
Beispiel für Kollaborationsdiagramm



Beispiel für Sequenzdiagramm



Darstellung mittels Zustandsautomaten



Methodische Vorgehensweise

Nicht nur Darstellung von OO-Modellen (UML) sollte standardisiert sein, sondern auch der Analyseprozeß selbst (nicht in UML):

verschiedene „Schulen“:

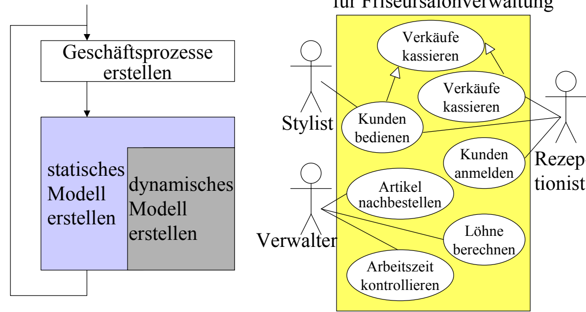
- **anarchists** keine Regeln (nur eigene Kreativität)
- **behaviorists** Konzentration auf Rollen und Verantwortlichkeiten
- **storyboarder** alles ist ein Geschäftsprozeß
- **information modeller** Konzentration auf Daten statt Prozesse
- **architects** Vorliebe für „Bausteine“ (patterns, frameworks)

Abgesehen von der anarchistischen Schule profitieren alle Vorgehensweisen von einer Methodik. Typisches Vorgehen besteht aus

Makroprozeß und **Checklisten**

Der Makroprozeß

Beispiel Geschäftsprozeßdiagramm für Friseursalonverwaltung



Sechs Schritte zum statischen Modell

1. **Klassen identifizieren**
   → Klassendiagramme, Kurzbeschreibungen
2. **Assoziationen identifizieren**
   → Klassendiagramme vervollständigen
3. **Attribute identifizieren**
   → Klassendiagramme vervollständigen
4. **Vererbungsstrukturen identifizieren**
   → Klassendiagramme vervollständigen
5. **Assoziationen vervollständigen**
   → Klassendiagramme, Objektdiagramme
6. **Attribute spezifizieren**
   → Attributspezifikationen und Beschreibungen

Drei Schritte zum dynamischen Modell

1. **Szenarios erstellen**
   → Sequenzdiagramme, Kollaborationsdiagramme
2. **Zustandsautomat erstellen**
   → Zustandsdiagramm
3. **Operationen beschreiben**
   → Klassendiagramme vervollständigen, fachliche Beschreibung der Operationen, Aktivitätsdiagramme

Checklisten

- helfen beim Erstellen der Diagramme und Spezifikationen
- sind nicht immer ohne Änderungen für alle Probleme anwendbar

1	**Akteure** ermitteln (wer tut was über welche Schnittstellen)
2	primäre und sekundäre **Geschäftsprozesse** ermitteln
2a	mittels Akteure (Akteure = Personen?, welche Arbeitsabläufe?)
2b	mittels Ereignissen (Ereignisliste erstellen, welche extern/zeitlich?)
2c	mittels Aufgabenbeschreibungen (Gesamtziele, Ziel jeder Aufgabe)
3	Geschäftsprozesse für **Sonderfälle** formulieren
4	komplexe Geschäftsfälle **aufsplitten**
5	**Gemeinsamkeiten** von Geschäftsprozessen ermitteln
6	**gute Beschreibung** (verständlich für Auftraggeber)
7	**Konsistenz** mit Klassendiagramm prüfen
8	**Fehlerquellen** (Zu viele Prozesse? Zu detailliert?)