



Hochschule Karlsruhe  
Technik und Wirtschaft  
**UNIVERSITY OF APPLIED SCIENCES**

# Software-Engineering

Prof. Dr. Thomas Fuchß  
Hochschule Karlsruhe – Technik und Wirtschaft  
Fakultät für Informatik und Wirtschaftsinformatik





# Inhaltsverzeichnis

- **Grundlagen**
- **Objektorientierung**
- **Entwicklungsprozesse**
- **Objektorientierte Analyse**
- **Objektorientiertes Design**

**Labor**

**Die Anmeldung zum Labor erfolgt später.**



# Literatur (Grundlagen)

Die Vorlesung basiert im Wesentlichen auf Literatur und Materialien aus folgenden Quellen.

- **Larman, Craig**  
*Applying UML and patterns : an introduction to object-oriented analysis and design and the Unified Process*, 1. ed. – Upper Saddle River, NJ : Prentice Hall, 1998.
- **Larman, Craig**  
*Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 2. ed. – Upper Saddle River, NJ : Prentice Hall, 2002.
- **Larman, Craig**  
*Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3. ed. – Upper Saddle River, NJ : Prentice Hall, 2004.
- **Jacobson, I., Booch, G. and Rumbaugh, J.**  
*The unified software development process* – Reading, Mass.: Addison-Wesley, 1999.
- **OMG Object Management Group**  
*OMG Unified Modeling Language (OMG UML) Version 2.5* – Needham Ma: OMG, 2013.  
(<http://www.omg.org>)
- **Rupp,C., Queins, S. und Zengler, B.**  
*UML 2 glasklar: Praxiswissen für die UML-Modellierung*, 3. Auflage – Hanser, 2007.



# Literatur

- Jim Arlow, Ila Neustadt  
UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design, 2. ed. – Addison-Wesley Professional, 2005.
- Schwaber, K. and Sutherland, J.  
The Scrum Guide: The Definitive Guide to Scrum – Scrum.org, 2011.
- Doug Shimp D. and Rawsthorne D.  
Exploring Scrum: The Fundamentals – CreateSpace, 2011.
- Booch, G., Rumbaugh J. and Jacobson I.  
Das UML-Benutzerhandbuch – Bonn : Addison-Wesley, 1999. Original: The unified modeling language user guide
- Fowler, Martin and Kendall, Scott  
UML distilled : a brief guide to the standard object modeling language – Reading, Mass. : Addison Wesley, 2000.
- Martin, James and Odell, James  
Object oriented methods : a foundation – UML ed., 2. ed. – Englewood Cliffs, N.J. : Prentice Hall, 1998.



# Literatur

- Bernd Oestereich  
Developing Software with UML: Object-Oriented Analysis and Design in Practice 2. ed.: Addison-Wesley Professional, 2003.
- Bernd Oestereich  
Analyse und Design mit UML 2.1: Objektorientierte Softwareentwicklung, 8. ed. – München; Wien: Oldenbourg, 2006
- Die UML-Kurzreferenz für die Praxis : kurz, bündig, ballastfrei – München; Wien: Oldenbourg, 2001.
- Oestereich, Bernd  
Erfolgreich mit Objektorientierung : Vorgehensmodelle und Managementpraktiken für die objektorientierte Softwareentwicklung (Hrsg.). Peter Hruschka – München; Wien: Oldenbourg, 2001.
- Rumbaugh, J. et. al.  
Objektorientiertes Modellieren und Entwerfen – München: Hanser; London: Prentice-Hall Internat., 1994. Original: Object-oriented modeling and design
- **Sommerville, Ian**  
**Software Engineering (9. Ausgabe): Pearson Studium, 2012**



# Literatur

- Gamma, Erich et. al.  
Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software – München: Addison-Wesley, 2001.
- **Gamma, Erich et.al.**  
**Design patterns : elements of reusable object-oriented software – Reading, Mass.: Addison-Wesley, 1995.**
- Fowler, Martin  
Analysemuster: wiederverwendbare Objektmodelle ; [ein Pattern-Katalog für Business-Anwendungen]. Dt. Übers. von Andrea Dauer – Bonn : Addison-Wesley-Longman, 1999.
- Fowler, Martin  
Refactoring : Improving the Design of Existing Code – Reading, Mass.: Addison Wesley, 2000.
- (Pattern-oriented software architecture) Band: 1  
Buschmann, Frank. A system of patterns: John Wiley & Sons. 1996
- (Pattern-oriented software architecture) Band: 2  
Schmidt, Douglas C. Patterns for concurrent and networked objects: John Wiley & Sons.



**Hochschule Karlsruhe  
Technik und Wirtschaft**  
**UNIVERSITY OF APPLIED SCIENCES**

# **Software-Engineering**

## **Grundlagen**

Prof. Dr. Thomas Fuchß  
Hochschule Karlsruhe – Technik und Wirtschaft  
Fakultät für Informatik und Wirtschaftsinformatik





# Wieso, weshalb, warum ...?

Kommentare zu **iX-Workshops mit UML-Erfinder Ivar Jacobson**, heise News 10.3.2008  
(<http://www.heise.de/newsticker/iX-Workshops-mit-UML-Erfinder-Ivar-Jacobson--/meldung/104782>)

## 10. März 2008 12:06 Wer verwendet UML – und zwar richtig, ausgefeilt und detailliert?

**Ulriko** (mehr als 1000 Beiträge seit 26.06.01)

UML ist seit Jahren bekannt und ein „Standard“ in der EDV nur habe ich es nahezu nie in extenso im Realeinsatz gesehen, eher als „Anhängsel“ und Beigabe. Wenn doch, dann hat man es dennoch meist mehr schlecht als recht, sozusagen in einer Näherungsform, verwendet. Irgendwie schien es nie recht sinnvoll, die Zeit in die Erlernung der Notation zu stecken und dann die UML zum Zentrum der Modellierungsprozesse zu machen, man verwendet halt gängige Symbole daraus je nach Laune und Gefühl und mischt es mit anderen gängigen Notationen (z.B. der „MS Powerpoint-Autoformen-Flussdiagramm-Notation“ ;-). Klar, viele Tools verwenden UML oder UML-artige Symbole, so dass man z.B. Datenmodelle oder Geschäftsprozesse darin oft ausgedruckt bekommt. Aber ich kenne wenige Leute, die die Notation wirklich lesen können, die also wissen, was all die vielen kleinen Striche und Kreuzchen usw. an den Kästen bedeuten (so grob weiß es fast jeder, klar). Wenn ich durch UML-Lehrbücher kucke, habe ich den Eindruck, dass die 20/80-Regel auch hier gilt: 80% aller Verwendungen der UML nutzen nur 20% der dort vorgestellten Symbole. Vielleicht sind es aber sogar 95% aller Verwendungen und 10% der Symbole... Kann da jemand über andere Erfahrungen berichten?

**Ulriko** 14:15 14.03.2008

## Re: Wer verwendet UML – und zwar richtig, ausgefeilt und detailliert?

**Nasan** (mehr als 1000 Beiträge seit 04.12.00)

Na ist doch klar warum – UML ist etwa so tauglich für SW-Entwicklung wie linux für Bildbearbeitung...



# Wieso, weshalb, warum ...?

10. März 2008 14:12 Re: Wer verwendet UML – und zwar richtig, ausgefeilt und detailliert?  
*redrocker* (mehr als 1000 Beiträge seit 17.02.05)

Oliver\_\_ schrieb am 10. März 2008 14:00

> In der Praxis dann erst mal nichts von UML. Bis, ja bis die Firma  
> sich irgendwann entschloss, „wir machen jetzt auf RUP!“ (Rational  
> Unified Process, ein weiteres buzzword). Also alle Manager und  
> Entwickler mal kurz in einen 1-wöchigen RUP Kurs geschickt, ein  
> schönes tool gekauft und los ging's! Alle 6 Wochen die gleiche  
> PowerPoint Folter: dutzende von Kästchen, Pfeilen, Parallelogrammen,  
> zu kleinen Schriften... „Ist das dort eine 1, ein \*... Oder bloss ein  
> Fliegenschiss?“ – „Oh, eigentlich eine 1, aber das ist hier nicht so  
> wichtig, es geht bloss um die Illustration.“ – „Achso, ja  
> kaaaaaar....“

Da kann ich Dir nur zustimmen. UML ist schön für Powerpoint-Fetischisten und sonstigen Wichtigschwätzern. Nur um irgendwo mal bisschen was dazulabern zu können. Solche Präsentationen sind reine Zeitverschwendungen, aber bringen einigen Beteiligten ein gewisses Gefühl der bedeutsamen Einflussnahme und Unentbehrlichkeit.

> Ja, um die Illustration. Meiner Erfahrung bin ich mit Bleistift und  
> Papier immer noch 1947284 mal schneller, um mit ein paar Kreisen und  
> Pfeilen meine Idee rüberzubringen.

ACK!



# Was ist Software-Engineering ?

**Die Technologie- und Management-Disziplin  
für die systematische Erstellung und  
Wartung von Software.**

**Vorlesungsziel:**

**Die Vermittlung von Techniken zur strukturierten und  
objektorientierten Entwicklung von Software im Großen.**



# Was ist Software-Engineering?

Software-Engineering zielt auf die ingenieurmäßige Entwicklung, Wartung, Anpassung und Weiterentwicklung **großer Software-Systeme** unter Verwendung bewährter, systematischer Vorgehensweisen, Prinzipien, Methoden und Werkzeuge hin.

M. Broy und D. Rombach.

Software Engineering: Wurzeln, Stand und Perspektiven.

Informatik Spektrum 25(6), Springer-Verlag, Heidelberg, 2002.



# Eigenschaften großer SW-Systeme

- **große Anzahl von Benutzern und viele Entwickler**
- **Entwickler und Benutzer sind disjunkt**
- **sind in der Regel so groß, dass sie von einer einzelnen Person nicht mehr in allen Details verstanden werden**
- **die Entwicklungsressourcen sind beträchtlich**  
(Zeit, Personal und damit auch die Kosten)
- **haben in der Regel eine hohe Lebenserwartung**  
(Jahr 2000 Problem)
  - viele Änderungen (Change-Requests) sind erforderlich
  - Systemfamilien entstehen mit vielen Versionen



# Ziele der systematischen SW- Entwicklung

- Erhöhung der Produktivität
- Verbesserung der Qualität
- Reduktion der Kosten
- Automatisierung der Entwicklung

**Zur Beherrschung dieser Aufgaben ist ein  
systematisches Vorgehen unabdingbar.**

durch:

- einen systematischen Entwurf
- erhöhte Wiederverwendung (Reuse)
- bessere Wartbarkeit (etwa 60% des Aufwands)



# Risiken bei der SW-Entwicklung

- **Fähigkeiten der Entwickler** Kompetenz, Systemkenntnis

**Probleme:** Personalmangel

**Lösung:** Schulung, Verbesserung der Gruppendynamik, Wahl des Besten

- **Problemverständnis**

**Probleme:** Erstellung falscher Funktionalität, Benutzerschnittstellen; Überausstattung

**Lösung:** Absprachen mit dem Auftraggeber, Prototypen, genaue Analyse

- **Komplexität**

Anwenderprogramme < Frameworks und Dienstprogramme < Systemprogramme

**Probleme:** Unrealistischer Zeitplan und Budget

**Lösung:** Detaillierte Kostenschätzung unter Berücksichtigung der Komplexität und Produktivität, inkrementeller Entwurf

- **Qualität und Zuverlässigkeit** unterschiedliche Ansprüche und Anwendungsgebiete erfordern unterschiedliche Anforderungen an Prüfung und Tests

**Probleme:** Mangelnde Qualität

**Lösung:** Detaillierteres Design, verbesserte Testpläne, Einsatz formaler Methoden, inkrementeller Entwurf, **Reduktion auf das Wesentliche**



# Risiken bei der SW-Entwicklung

- **Werkzeuge und Technologie**

**Problem:** Neue Technologien und Werkzeuge werden nicht verstanden oder falsch eingesetzt  
**Lösung:** Schulungen

- **Änderungsverhalten**

**Problem:** Eine Lawine von Änderungswünschen gefährdet Zeit, Budget und Qualität  
**Lösung:** formelle Änderungsprozeduren, Prototypen, inkrementelle Entwicklung

- **Kommunikation und Teamgröße**

**Problem:**

- Kommunikation ist notwendig wächst aber quadratisch mit der Anzahl der Teammitglieder
- Die Projektdauer, kann durch Hinzufügen von Personen nicht beliebig reduziert werden.



# Wie können Risiken reduziert werden?

**Sorgfältiges Planen vermeidet hohe Kosten,  
schlechte Qualität, Verzögerungen usw.**

- **Festgelegt werden**

Ziele, Anforderungen, Beschränkungen, Lösungsstrategien, Budget, Organisation usw.

- **typische Vorgehensweise**

- Problemdefinition und Analyse
- Design
- Implementierung und Tests
- Betrieb



# Fortschritte müssen erkennbar sein

- Wie können Fortschritte gemessen werden?
- Wie können Verzögerungen erkannt werden?
- Die Entwicklung eines Softwaresystems muss in klar abgegrenzte Abschnitte unterteilt werden.
- Die Übergänge zwischen Abschnitten müssen leicht erkennbar sein.
- Die Anzahl der Abschnitte ist variabel aber nicht beliebig (eher fest).



Dies widerspricht nicht der Forderung nach Agilität!



# Aufgaben in der SW-Entwicklung

## Aufgaben des Managements:

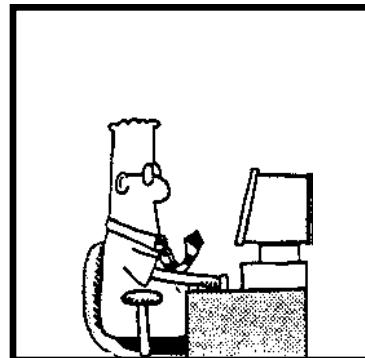
- Kostenüberwachung
- Ressourcenverteilung
- Planung
- Terminüberwachung
- Risikoabschätzung



Copyright © 2002 United Feature Syndicate, Inc.

## Aufgaben der Entwickler:

- Analyse
- Design
- Implementierung
- Test
- Wartung



rein technische  
Aufgaben



# Aufgaben in der SW-Entwicklung

**Diese Aufgaben müssen sich in den Übergängen  
widerspiegeln, um Entscheidungsmöglichkeiten  
und Steuermechanismen zu bieten.**



# Übergänge planen

Diese Übergänge zwischen Entwicklungsstufen werden meist als Meilensteine bezeichnet.

**Planen der Übergänge und Denken in Meilensteinen führt zur Definition von Entwicklungsprozessen. Diese ermöglichen es, ein Projekt zu steuern.**

- Probleme frühzeitig erkennen
- Gegenmaßnahmen planen und einleiten
- Ressourcen anfordern (Geld, Personal, Zeit,...)
- Kürzungsmöglichkeiten wahrnehmen
- Projekt einstellen



# Prozessmodelle

Anforderungen  
(Requirements)



Produkt  
**(hoffentlich)**

Anforderungen  
(Requirements)



Produkt



# Klassische Entwicklungsmodelle

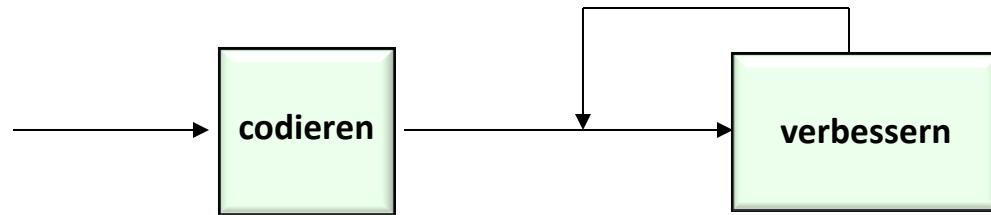
- Codieren und Verbessern
- Wasserfallmodell
- Prototypmodell
- Spiralmodell
- Versionsmodell



# Codieren und Verbessern

Die naive Methode:

**Man codiert erst, und macht sich dann Gedanken über Anforderungen, Design, Testmöglichkeiten und Wartung.**

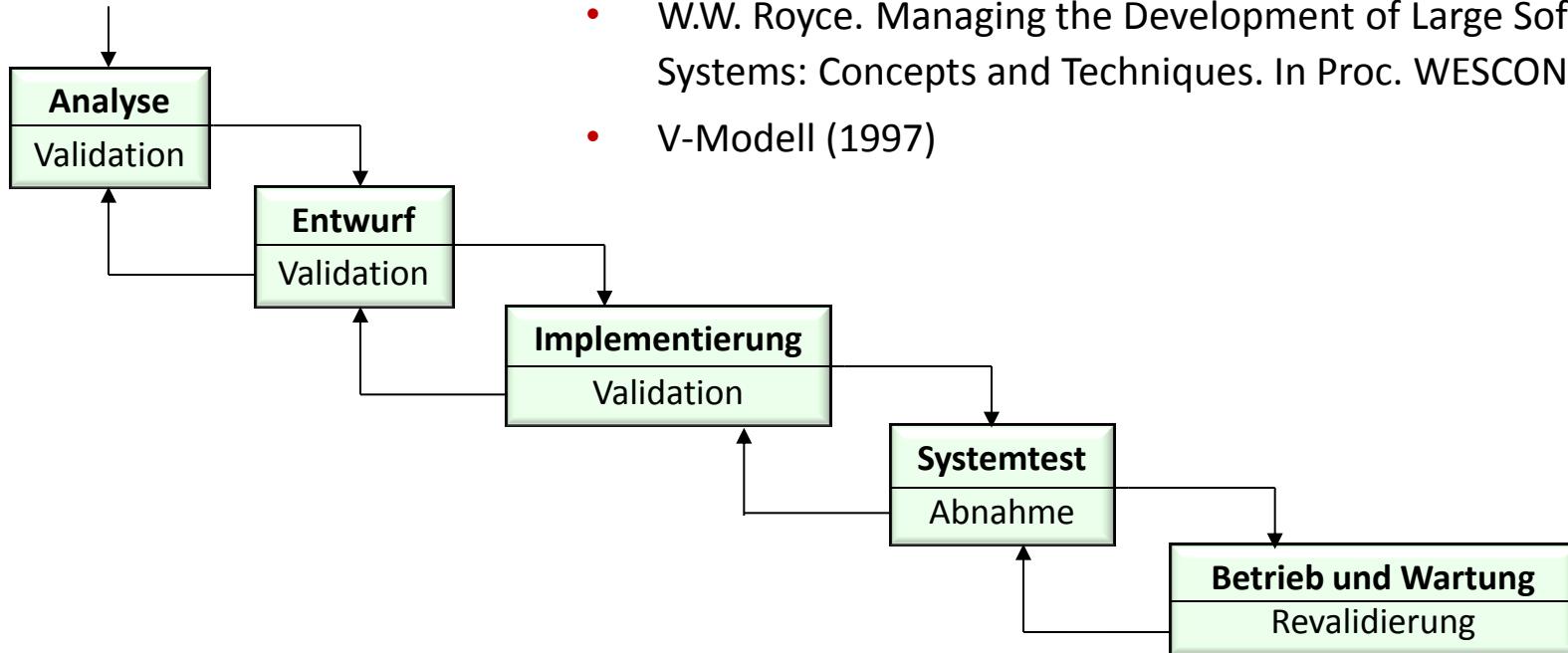


- Nach vielen Verbesserungen wird der Code unbrauchbar (Spaghetti-Code).  
    → **Entwurfsphase ist notwendig**
- Selbst gut entworfene SW kann zu weit von den Bedürfnissen entfernt sein.  
    → **Analysephase ist notwendig**
- Korrekturen und Änderungen sind kostspielig, Programme sind nicht auf Tests und Änderungen vorbereitet.  
    → **Einplanung von Testphasen und Testschnittstellen**



# Wasserfallmodell

auch bekannt als: **Software life cycle**



- **Feste Phasen mit genau definierten Übergängen**
- **Rückkopplung ist nur zwischen benachbarten Phasen möglich**



# Probleme des Wasserfallmodells

- Phasen sind starr
- Pflichtenhefte und Dokumente müssen vollständig ausgearbeitet werden
- genaue Spezifikationen für schlecht verstandene Anforderungen gefolgt von der Implementierung großer Mengen nutzlosen Codes
- hohe Kosten für iterative Systeme
- Zielsystem wird als ganzes betrachtet

## Vorteile:

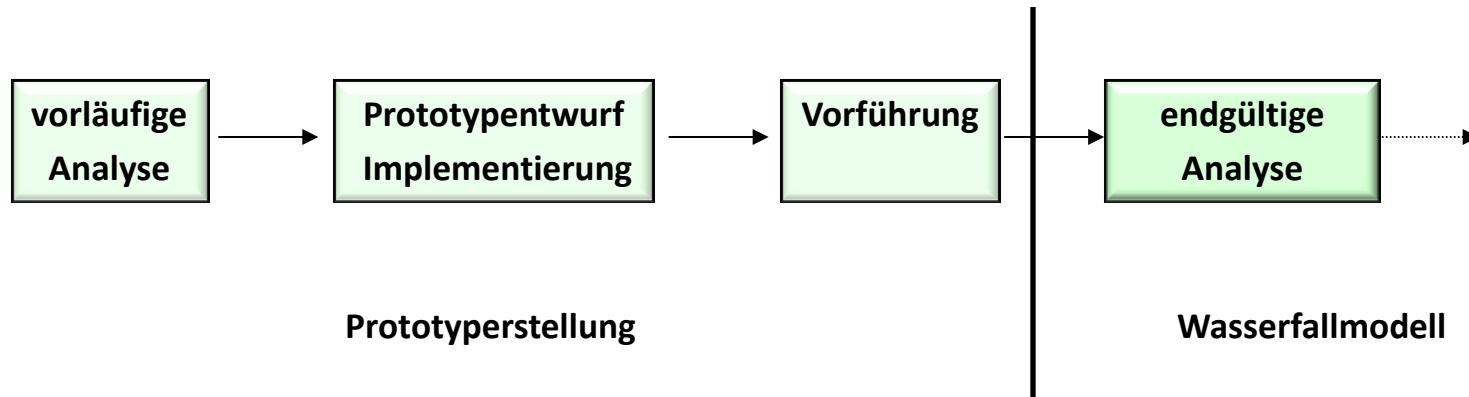
- Projektverlauf ist deutlich ersichtlich
- Verzug ist früh erkennbar
- Meilensteine für Management und Entwicklung liegen jeweils am Ende einer Phase



# Das Prototypmodell

## Idee:

Entwicklung eines Prototyps klärt die offenen Fragen bzgl. der Kundenwünsche und Anforderungen, schafft Planungssicherheit und reduziert das Risiko einer Fehlentwicklung.



schnell entsteht ein eingeschränkt funktionsfähiges Modell



- Arbeitsmoral
- Vertrauen



# Spiralmodell

Boehm (1988) ein neuer Ansatz:

Risikomanagement bestimmt den Entwicklungsverlauf

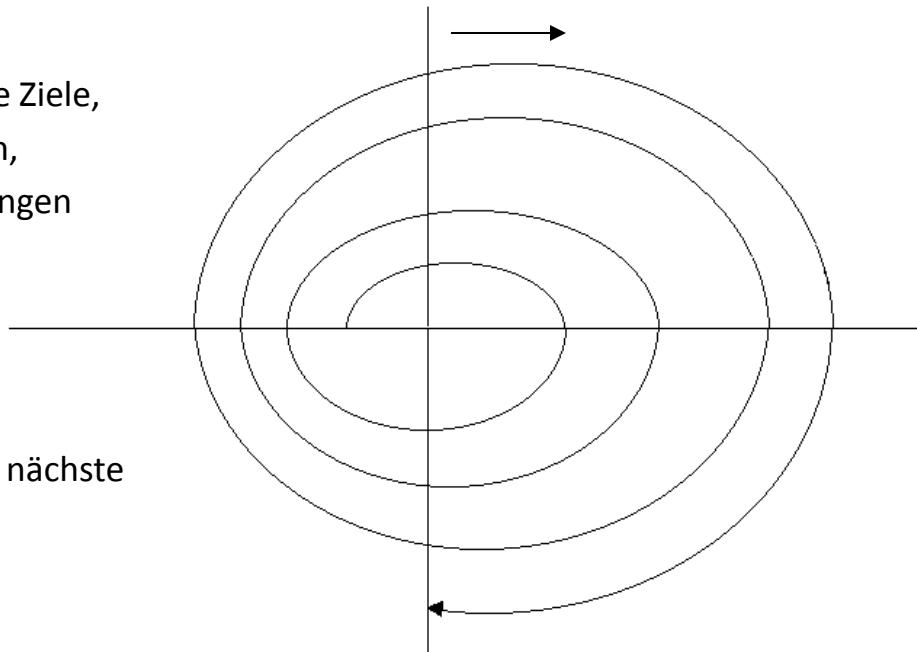
Barry Boehm. A Spiral Model of Software Development and Enhancement, IEEE Computer, May 1988

1. Bestimme Ziele,  
Alternativen,  
Beschränkungen

2. Evaluiere Alternativen,  
identifiziere und beseitige Risiken

3. Entwickle und verifiziere  
das Produkt

4. Plane die nächste  
Phase



- **jede Schleife bestimmt eine Phase**
- **die Phasen sind nicht fix, das Management entscheidet über den Ablauf, aufgrund der bisherigen Resultate**



# Was sind Risiken?

Der größte Unterschied zu anderen Modellen liegt in der expliziten Betrachtung des Risikos als entscheidende und treibende Kraft bei der Entwicklung der Software.

## Was sind Risiken?

Entscheidungen, die auf unvollständigen und falschen Annahmen und Informationen beruhen.

→ **Informationen sammeln und erarbeiten, die Unsicherheiten beseitigen**

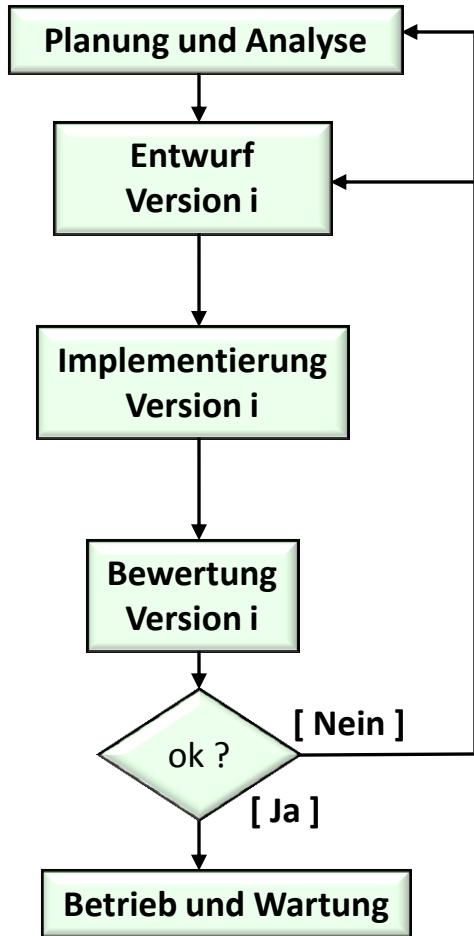
**Leider werden falsche Annahmen und Informationen nicht immer als solche erkannt.**



# Versionsmodell

## Idee: Wasserfallmodell iterativ

D.D. McCracken and M.A. Jackson. Life-Cycle Concepts Considered Harmful. ACM Software Engineering Notes, April 1982



**Jede Version ist ein Teilsystem, die Teilsysteme bauen aufeinander auf und bilden zusammen das Gesamtsystem.**

- Wer bestimmt den Umfang von Version i ?
- Wie erfolgt die Aufteilung ?
- Wie viele Versionen gibt es ?

→ Zeitplan vage

Frühe Entwicklung einer SW-Architektur ist notwendig



# Vorteile des Versionsmodells

- **Neu auftretende Nutzeranforderungen können integriert werden.**  
(erste Erfahrungen mit lauffähigen Zwischenprodukten)
- **Anforderungen können im Laufe der Versionen verfeinert werden.**
- **Der frühe Einsatz eines Kernsystems beim Auftraggeber ist möglich.**
- **Der Auftraggeber kann in die Entscheidung über die Versionsreihenfolge einbezogen werden.**  
(wichtige und kritische Aspekte zuerst)



# Neue Phasen

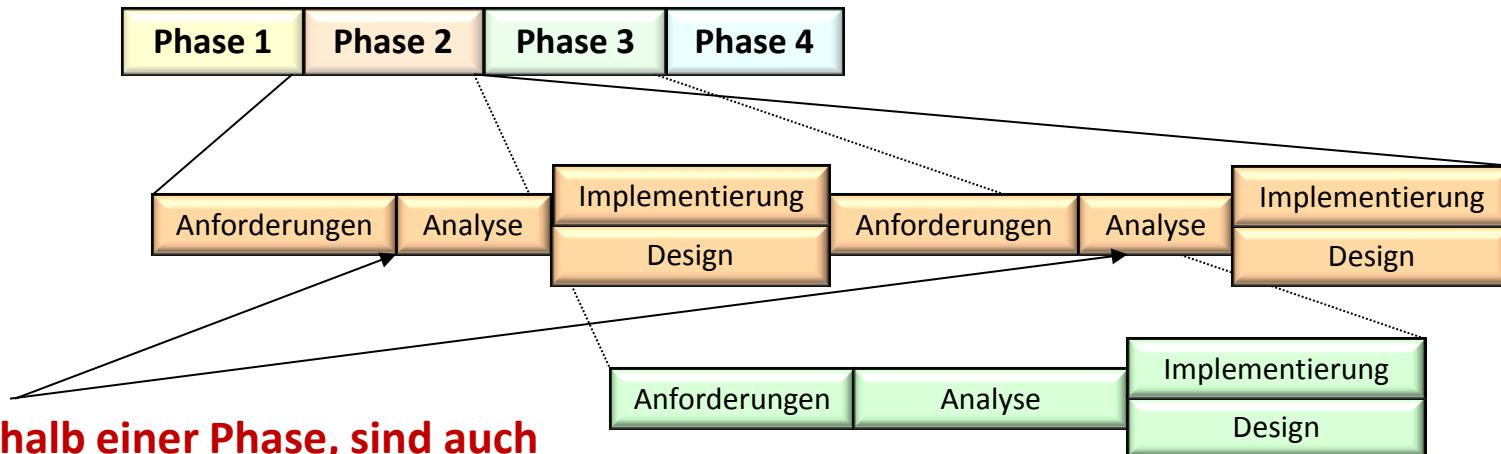
Offensichtlich entfällt der Begriff der Phase als zusammenhängender zeitlicher Bereich für Analyse, Design, Implementierung, usw. Da aus technischer Sicht Analyse, Design, Implementierung weiterhin Bestand hat ist eine neue Phaseneinteilung notwendig. Statt Phasen für Analyse, Design und Implementierung können neue Phasen konzipiert werden wie:

- Projektinitiierung,
- Projektvorbereitung,
- Projektdetaillierung I,
- Projektdetaillierung II, ...

**Innerhalb dieser Phasen werden wiederum die Aufgaben Analyse, Design, Implementierung, usw. durchgeführt.**



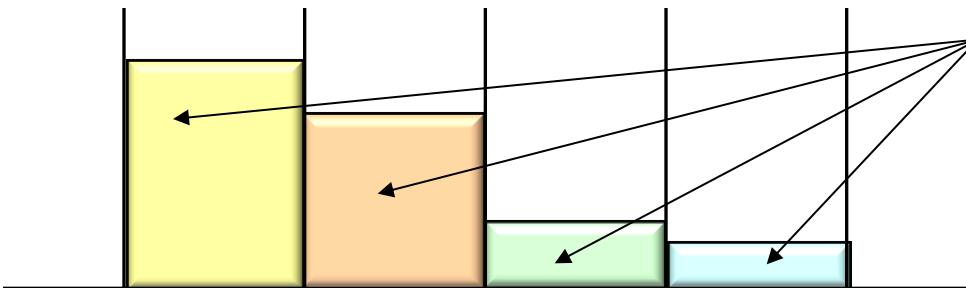
# Verteilung der Aktivitäten



innerhalb einer Phase, sind auch mehrere Iterationen möglich.

Phase 1    Phase 2    Phase 3    Phase 4

Bsp.: Aufwand zur Ermittlung von Nutzeranforderungen innerhalb der verschiedenen Phasen.





# Prozesse sind Software

Leon Osterweil. Software Processes are Software too. In Proc. Ninth International Conference on Software Engineering, 1987

Unterschiedliche Projekte und unterschiedliche Firmen stellen unterschiedliche Anforderungen und haben unterschiedliche Voraussetzung.



## Den universellen Softwareprozess kann es nicht geben.

- Der Softwareprozess muss sich am Problem orientieren.
- Er muss zum Projekt und der aktuellen Situation passen.



## Entwicklung von:

- Process-centered Software Engineering Environments (PSEE) und
- Process Modelling Languages (PML)



# Objektorientierte Softwareentwicklung

**Komplexe objektorientierte Systeme werden**

- **iterativ entwickelt**
- **architekturorientiert entwickelt**

**Die Architektur eines Softwaresystems ist ein Bauplan für die Organisation des Gesamtsystems (Beschreibung der Komponenten und Subsysteme und ihrer Beziehungen)**

**Auch die Erstellung der Architektur erfolgt inkrementell !**

- Zu Beginn hat man eine erste grobe Architektur, die aber die angestrebte Gesamtfunktionalität berücksichtigen muss.
- Im Laufe des Projekts wird sie so ausgearbeitet, dass alle Teilprodukte zu einem einheitlichen System integriert werden können.



# Die Säulen der Objektorientierung

- **Natürliche Modellierung (Simula)** Ole-Johan Dahl and Kristen Nygaard 1962 - 67 )

Objekte dienen der Modellierung der Realität

```
Class Rectangle (Width, Height); Real Width, Height; ! Parameters;
Begin      Real Area;           ! Attributes;
            Procedure Update;   ! Methods;
                Begin Area := Width * Height; End of Update;
            Update; ! Body;
            OutText("Rectangle created: ");
        End of Rectangle;
```

- **Sharing (Smalltalk-80)** Xerox Palo Alto Research Center (Parc) 1972 - 80 )

Objekte teilen sich „gemeinsame“ Eigenschaften

super und self steuern den Zugriff

- **Abstrakte Daten Typen (ADTs) (Eifel)** Bertrand Meyer 1985 )

Objekte kapseln ihren internen Zustand. Der Zugriff erfolgt über eine definierte Schnittstelle

```
feature { NONE } privateCounter: INTEGER
```



# The Treaty of Orlando (1989)

L. Stein, H. Lieberman and D.Ungar. A Shared view of Sharing: The Treaty of Orlando. In Object-Oriented Concepts, Databases and Applications, ACM Press/Addison-Wesley,1989.

Sharing basiert auf zwei fundamentalen Mechanismen:

- **empathy (Nachempfinden)**

*The ability of one object to share the behavior of another object without explicit redefinition.*

A ahmt B im Rahmen der Nachricht M nach, falls A kein eigenes Protokoll für M hat, sondern als Antwort das von B benutzt.

**Wenn im Antwortprotokoll von B eine Nachricht an sich selbst geschickt wird, so wird diese bei der Antwort von A an A geschickt.**

- **templates**

*The ability to create a new object based on a template, which guarantees characteristics of the newly created object.*



# Grundprinzipien

- **ganzheitliche Vorgehensweise** – beschrieben werden:
  - Daten,
  - Funktion und
  - deren Zusammenhänge
- **intuitiv und kommunikativ** – das Problem wird gelöst durch:
  - Modellieren und
  - Simulieren
- **methodische Durchgängigkeit**
  - Ergebnisse der Aktivität  $i$  lassen sich in  $i+1$  übernehmen

das Programm als Abbild der realen Welt



# Beispiel für methodische Durchgängigkeit

**Am Anfang steht die Erfassung der Anforderungen**

## Ein Dialog:

**Entwickler:** Was ist Ihnen wichtig?

**Anwender:** Unser Kunde?

**Entwickler:** Was ist denn ein Kunde, welche Merkmale zeichnen ihn aus?

**Anwender:** Ein Kunde hat einen Namen, eine Anschrift und eine Bonität, die muss stimmen?



# Beispiel für methodische Durchgängigkeit

Im Analyse- und Designmodell erscheint der Kunde als Klasse

Klasse in UML:





# Beispiel für methodische Durchgängigkeit

In der Implementierung gibt es immer noch die Klasse Kunde

etwa der Art:

```
class Kunde
{
    String name;
    Anschrift anschrift;
    int bonitaet;
    ...
    public boolean bonitaetPruefen ()
    {
        ...
    }
    ...
}
```



- **der iterative Prozess ermöglicht es auf Änderungen adäquat zu reagieren**
- **die Kommunikation zwischen Anwendern, Experten und Entwicklern wird verbessert**
- **die durchgängige Modellierung verbessert die Qualität und Konsistenz**
- **dank der ganzheitlichen Sichtweise, werden die Strukturen der realen Welt (Problemraum) besser erfasst**



Hochschule Karlsruhe  
Technik und Wirtschaft  
**UNIVERSITY OF APPLIED SCIENCES**

# **Software-Engineering**

## **Objektorientierung (Wiederholung)**

Prof. Dr. Thomas Fuchß  
Hochschule Karlsruhe – Technik und Wirtschaft  
Fakultät für Informatik und Wirtschaftsinformatik





# Grundbegriffe der Objektorientierung

- Klassen, Objekte, Instanzen
- Attribute, Operationen
- Vererbung, Polymorphie, abstrakte Klassen
- Assoziationen, Aggregationen, Kompositionen



# Was sind Objekte ?

Objekte stehen für Dinge, Personen, abstrakte Begriffe der realen Welt.

## Bsp.:

- ein Fahrrad, ein Reifen, ein Auto, eine Versicherungspolice,
- ein Fisch, ein Mensch, Peter,
- ein Vektor, eine Matrix, ein Polynom

Objekte haben Eigenschaften und setzen sich wiederum aus Objekten zusammen. Ein Fahrrad hat eine Größe und eine Farbe, es besteht aus einem Rahmen, aus Rädern. Räder wiederum aus Speichen, usw.



# Realität und Modell

Realität:



Auto

Fußballer

Ball

Modell:

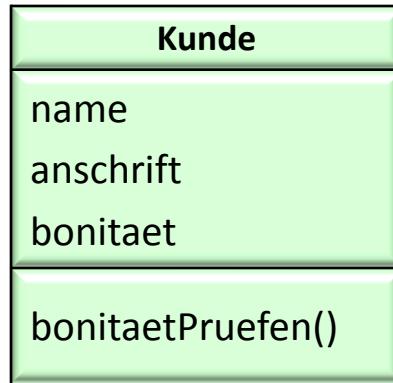




# Abstraktion

Dinge der realen Welt sind komplex. So besteht ein Mensch nicht nur aus einem Rumpf, Armen, Beinen und einem Kopf. Er hat Organe, Leber, Nieren, Blut, Nervenzellen, Ganglien, T-Helferzellen, ...

**Manchmal ist es aber nur wichtig, dass es ein Kunde ist, der über einen Namen, eine Adresse und eine gewisse Bonität verfügt.**



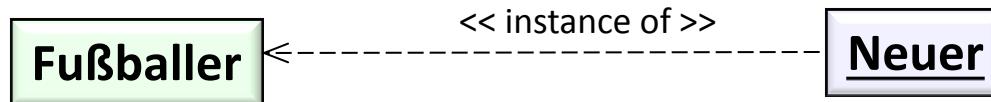
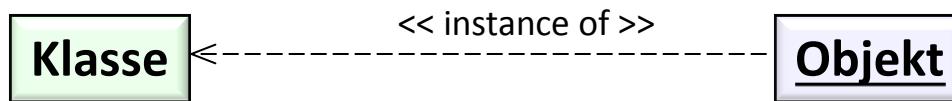
**Ein Modell ist ein Abbild der Realität, das von irrelevanten Dingen abstrahiert. In einem Modell werden nur die für das Problem relevanten Eigenschaften betrachtet.**



# Klassen und Klassifikation

**Klassen beschreiben die Struktur und das Verhalten vieler gleichartiger Objekte. Objekte bezeichnet man auch als Instanzen einer Klasse.**

**Die Zusammenfassung vieler gleichartiger Objekte zu einer Klasse bezeichnet man als Klassifikation.**





# Attribute, Operationen, ...

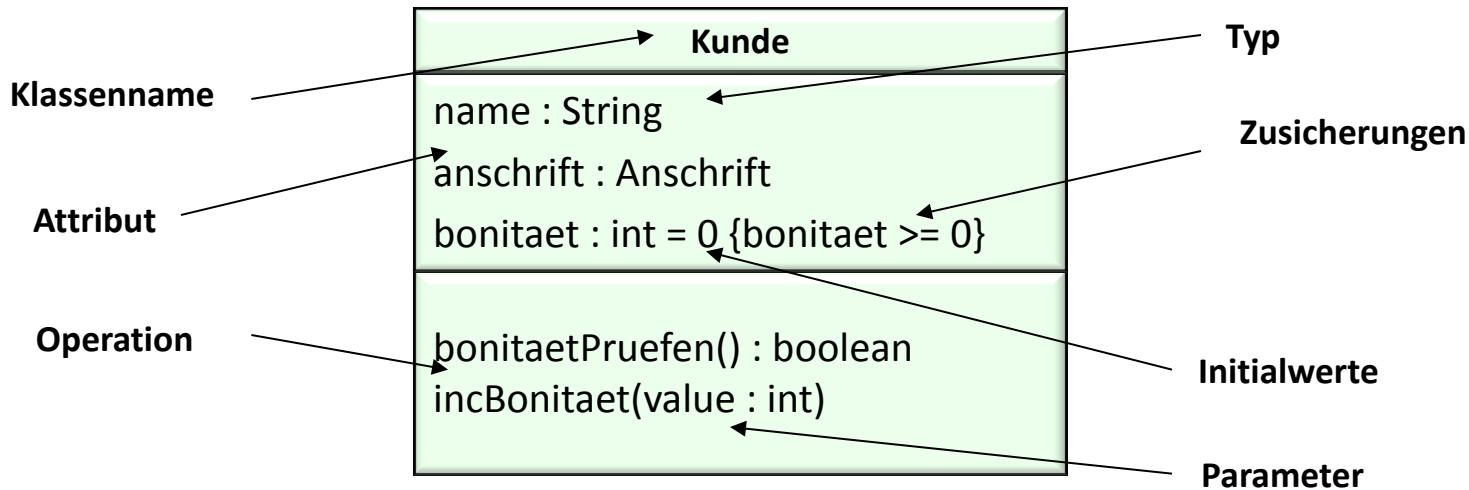
**Dinge der realen Welt haben Eigenschaften.  
Auf gleichen Eigenschaften beruht die Klassifikation.**

## **Doch was sind Eigenschaften?**

- **Attribute:** Beschreiben die Struktur von Objekten:  
Bestandteile, Information, Daten
- **Operationen:** Beschreiben das Verhalten von Objekten
- **Zusicherungen:** Regeln und Zwänge (Constraints), die die Objekte erfüllen müssen.
- **Beziehungen:** Beziehungen zwischen Klassen, **hat-ein**, **ist-ein**, **ist-Teil-von** ...



# Beispiel



**Klassen sind Einheiten aus Attributen, Operationen und Zusicherungen. Attribute sind nur über die Operationen einer Klasse zugänglich. Im Vergleich zu imperativen oder funktionalen Programmen, hängen also Daten und Funktionen in einer Einheit zusammen.**



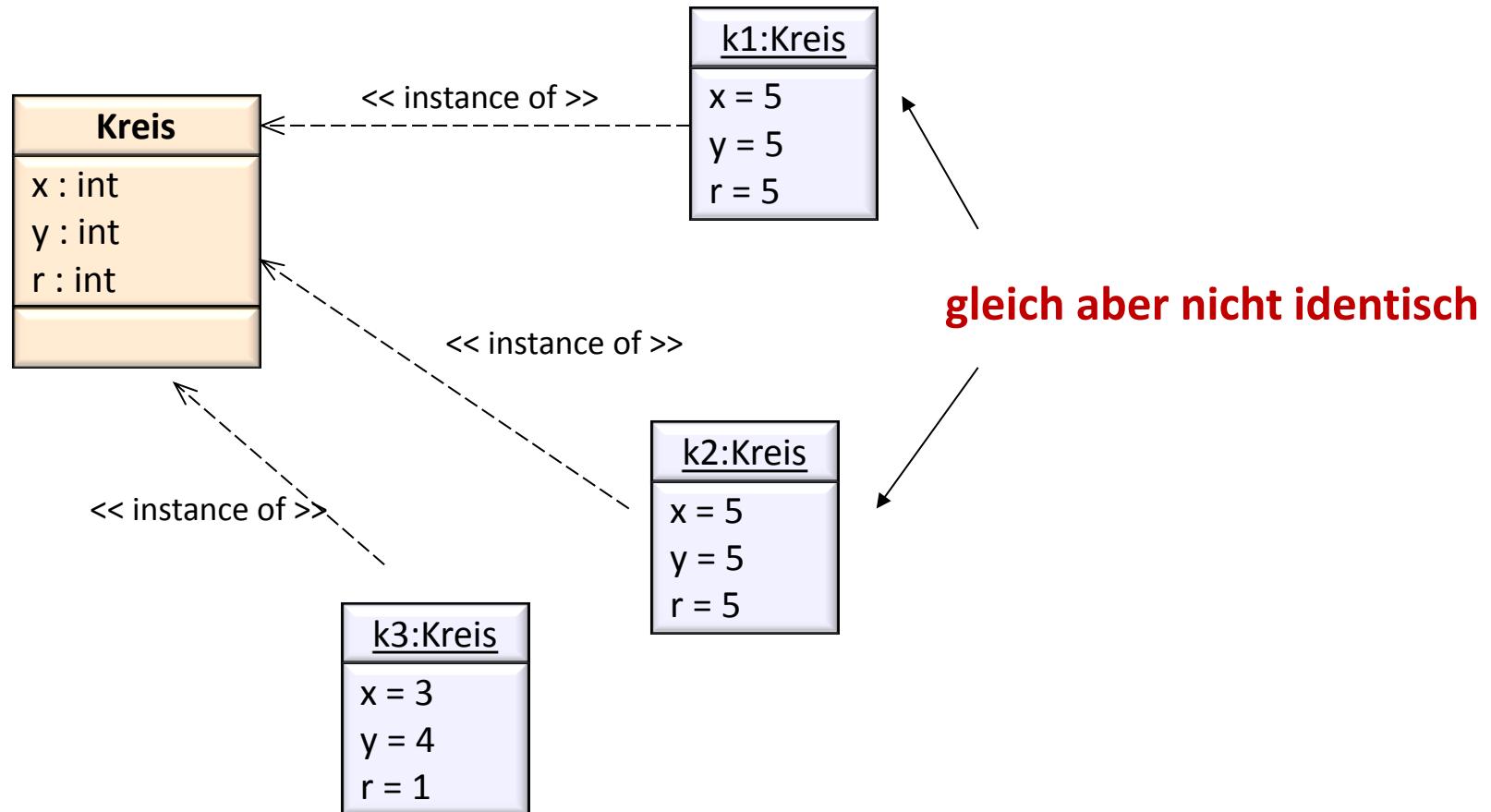
**Prinzip: Die Existenz eines Objektes ist unabhängig von den konkreten Attributwerten**



- zwei Objekte mit gleichen Attributwerten sind zwei Objekte (das gleiche ist nicht dasselbe)
- Objekte können nicht über die Werte ihre Attribute identifiziert werden.



# Beispiel





# Vererbung

## **Das Vererbungsprinzip:**

Klassen können Spezialisierungen anderer Klassen sein, d.h. Klassen können hierarchisch angeordnet werden. Sie übernehmen (erben) dabei die Eigenschaften (Attribute, Operationen, ...) ihrer übergeordneten Klassen. Sie können diese Eigenschaften überschreiben und erweitern, jedoch nicht eliminieren.

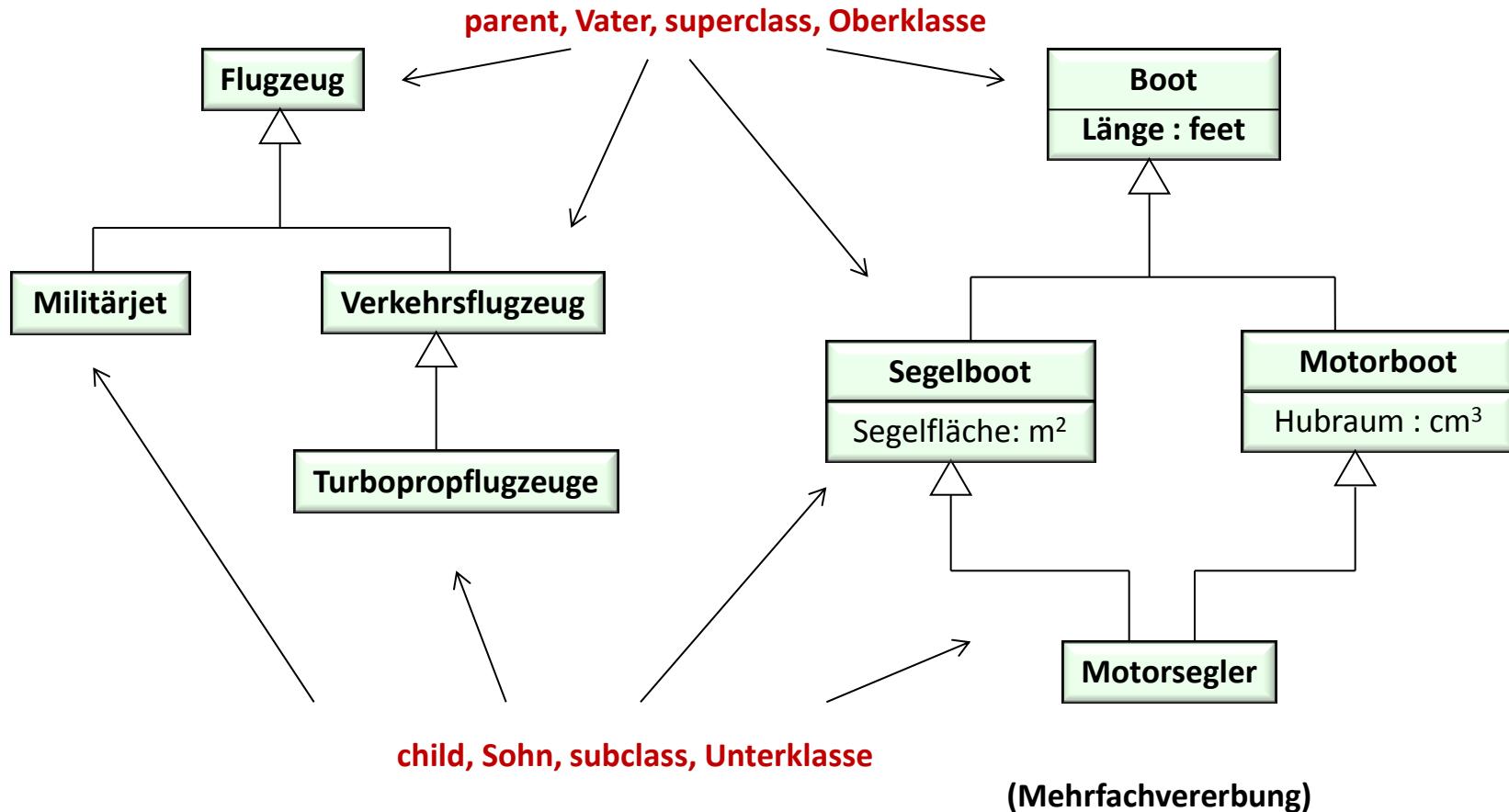
## **Man erbt auch Beziehungen:**

## **Das Substitutionsprinzip in Programmiersprachen:**

Objekte von Unterklassen können immer für Objekte ihrer Oberklasse eingesetzt werden.



# Beispiel





# Polymorphie

**Objekte (verschiedener Klassen) können auf gleiche Nachrichten unterschiedlich reagieren.**

- **statische Polymorphie (Overloading):**
  - Gleiche Namen für Operationen in verschiedenen Klassen.
  - Gleiche Namen für Operationen unterschiedlicher Signatur (in derselben Klasse).
- **dynamische Polymorphie:**
  - Zuordnung der Nachricht zur empfangenden Operation zur Laufzeit.
  - Die spezifischste Operation kommt zur Anwendung.



# Polymorphie

## Beispiel:

Die Nachricht „zeichne dich“ kann an jedes Objekt der Klasse Körper gesendet werden. Während Objekten der Klasse Kegel einen Kegel zeichnen, zeichnen Objekte der Klasse Würfel offensichtlich einen Würfel.

- **Aktionen stehen im Vordergrund**
- **Wartbarkeit und Erweiterbarkeit werden verbessert**
- **Aufrufstellen müssen nicht verändert werden**



# Beispiel

statt

```
ptr := list;  
WHILE (ptr <> nil) DO BEGIN  
  CASE ptr^.type OF  
    CONE : drawCone(ptr^.obj);  
    CUBE : drawCube(ptr^.obj);  
    ...  
  END;  
  ptr := ptr^.next;  
END;
```

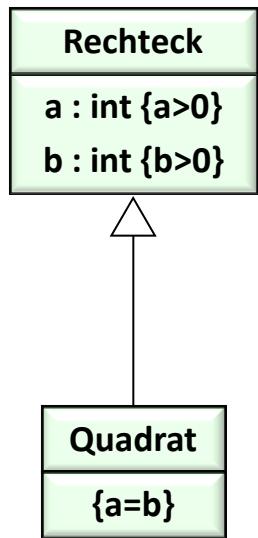
jetzt

```
obj = list.first();  
while (obj != NULL) {  
  obj.draw();  
  obj = list.next();  
}
```

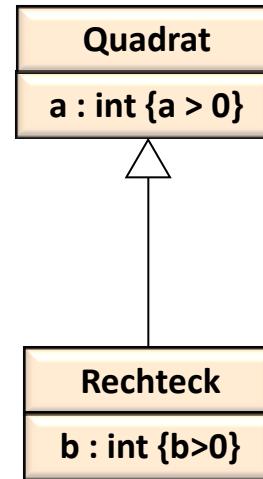


# Beispiel

## Was ist besser?



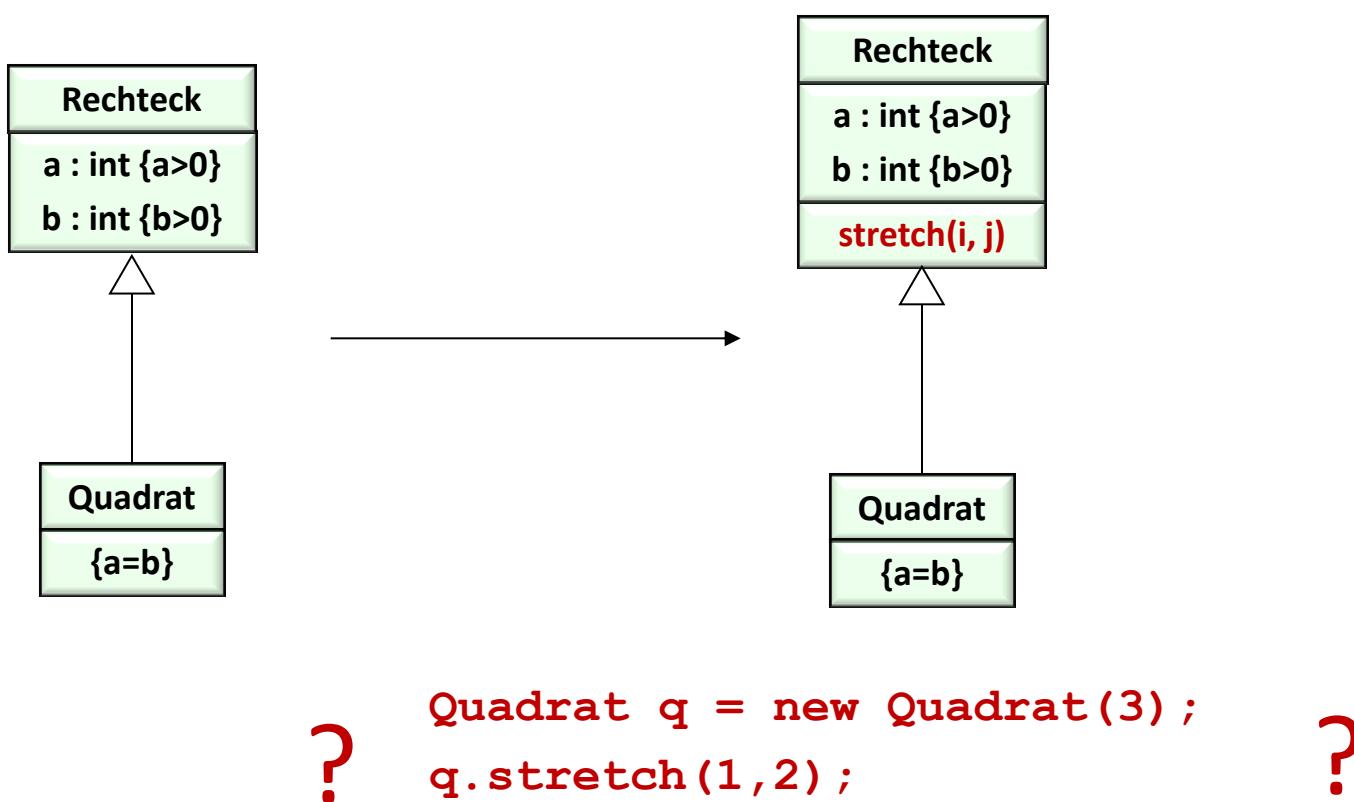
?





# Beispiel

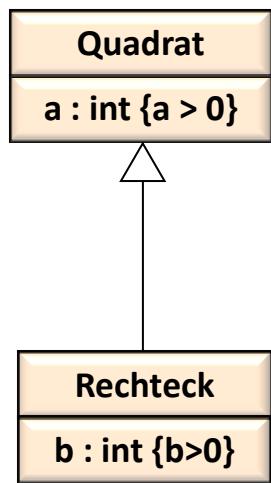
sicher ?





# Beispiel

sicher ?



?

Quadrat q;

Rechteck r = new Rechteck(3,4);

q = r;

?



# Beispiel

oder doch ohne Vererbung

Rechteck
a : int {a>0}
b : int {b>0}
isQuadrat() : boolean

```
public boolean isQuadrat()
{
    return (a==b)
}
```



# Abstrakte Klassen

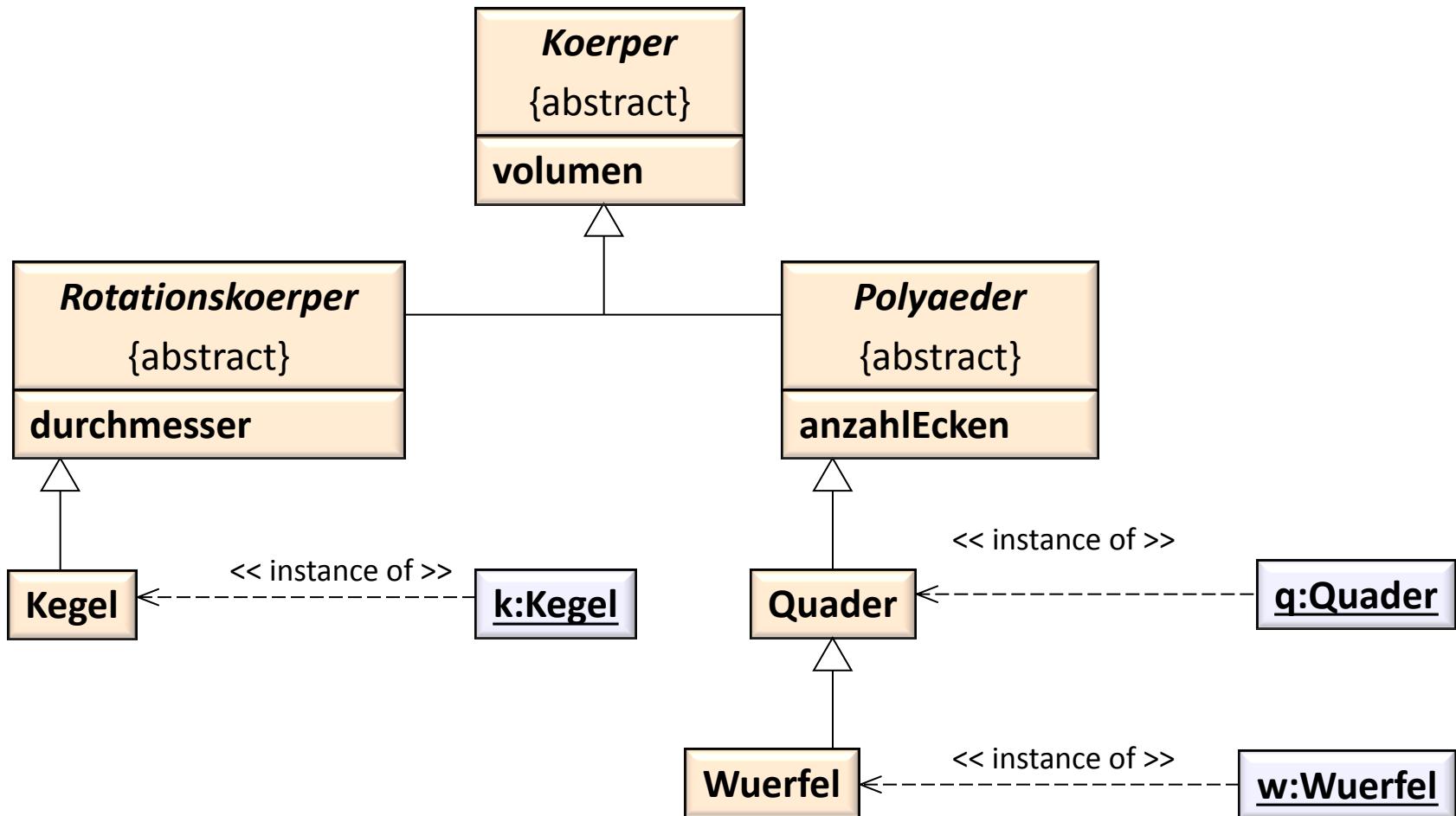
**Eine besondere Form von Klassen stellen abstrakte Klassen dar,  
von Ihnen existieren niemals Objekte.**

**Abstrakte Klassen werden verwendet, um gemeinsame  
Eigenschaften bestehender Klassen zu abstrahieren. Sie stellen  
damit eine effiziente Möglichkeit dar, Eigenschaften nur einmal  
zu definieren.**

**Abstrakte Klassen sind Schablonen für konkrete Klassen.**



# Beispiel





# Assoziationen

**Assoziationen modellieren Beziehungen zwischen verschiedenen Objekten einer oder mehrerer Klassen, die nicht auf Vererbung beruhen**



**Eine Assoziation enthält üblicherweise:**

- einen Namen
- Stelligkeit oder Multiplizität, die angibt, wie viele Objekte der gegenüberliegenden Seite mit einem Objekt der Ausgangsseite verbunden sind
- Rollenbezeichner, die die Bedeutung der beteiligten Klassen näher beschreiben



# Aggregation

Eine Aggregation ist eine spezielle Assoziation. Sie modelliert eine Teile-Ganzes-Beziehung zwischen verschiedenen Objekten einer oder mehrerer Klassen.

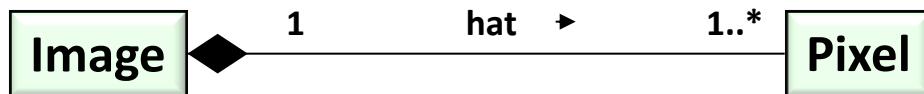


Eine Aggregation dient der Zusammensetzung eines Objekts aus einer Menge von Einzelteilen. Ein Aggregat handelt stellvertretend für seine Teile



# Komposition

Eine Komposition ist eine spezielle Aggregation. Die Einzelteile sind vom Aggregat existenzabhängig.

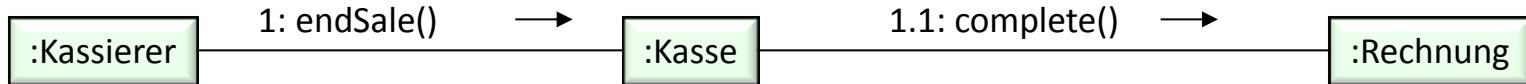


**keine Teile ohne das Ganze**



# Nachrichtenaustausch

Objekte sind eigenständige Einheiten, deren Zusammenarbeit über den Austausch von Nachrichten organisiert wird, die sich die Objekte untereinander zuschicken. Damit ein Objekt auf eine Nachricht reagieren kann, ist es notwendig, dass es über eine entsprechende **Operation** verfügt.





Hochschule Karlsruhe  
Technik und Wirtschaft  
**UNIVERSITY OF APPLIED SCIENCES**

## Software-Engineering

# Entwicklungsprozesse und die UML

Prof. Dr. Thomas Fuchß  
Hochschule Karlsruhe – Technik und Wirtschaft  
Fakultät für Informatik und Wirtschaftsinformatik





# Prozessmodelle

- **Klassische Prozessmodelle**
  - Codieren und Verbessern
  - Wasserfallmodell
  - Prototypmodell
  - Spiralmodell
  - Versionsmodell
- **Objektorientierte Prozessmodelle**
  - Rumbaugh et. al. (OMT)
  - **The Unified Software Development Process (RUP)**  
(Jacobson, Booch, Rumbaugh)
  - SCRUM (Schwaber, Sutherland)
  - FDD, TDD, ...



# Die Aufgabe eines Entwicklungsprozesses

Ein Entwicklungsprozess definiert „wer tut was, wann und wie“, um das vorgegebene Ziel zu erreichen. Ein effektiver Prozess stellt Richtlinien bereit, die eine effiziente Entwicklung ermöglichen. Er reduziert die Risiken und erhöht die Vorhersehbarkeit.

- **Technologie**

Der Prozess muss auf der Technologie beruhen, die heute verfügbar ist. (Sprachen, Betriebssysteme, Entwicklungsumgebungen, Netzwerke, ...)

- **Werkzeuge**

Der Prozess ist Werkzeug unterstützt. Werkzeug und Prozess entstehen parallel. Der Prozess unterstützt die Entwicklung des Werkzeugs und umgekehrt.



# Die Aufgabe eines Entwicklungsprozesses II

- **Personen**

Der Prozess muss auf die Fähigkeiten der Personen ausgelegt sein, die ihn anwenden sollen.

- **Organisationsstrukturen**

Der Prozess muss die heutigen Unternehmensstrukturen berücksichtigen

- virtuelle Organisationen verteilt Entwicklungsstandorte,
- Outsourcing und Unterauftragnehmer
- kleine Startups, mittlere Unternehmen, große Konzerne
- festangestellte und freie Mitarbeiter



# Die Spezifikationssprache UML

**Die UML ist eine graphische Spezifikationssprache. Sie ist keine Entwicklungsmethode und auch kein Softwareprozessmodell**

OMG Object Management Group

UML 2.4.1 Superstructure Specification – Needham Ma: OMG, 2011 ([www.omg.org](http://www.omg.org))

**Im Sinne des objektorientierten Paradigmas ist sie:**

- **ganzheitlich**  
beschrieben werden: Daten, Funktion und deren Zusammenhänge
- **intuitiv und kommunaktiv**  
da graphisch
- **methodische Durchgängigkeit**  
Es gibt Stilmittel für so gut wie alle Bereiche und Aufgaben im Rahmen von Analyse, Entwurf und Design. Diese Beschreibungselemente können über die Grenzen der Diagramme hinweg miteinander in Beziehung gesetzt werden.



# Diagramme

- **Strukturdiagramme**
  - Klassendiagramme (Class Diagrams)
  - Objektdiagramme (Object Diagrams) ergänzend
  - Kompositionsstrukturdiagramme (Composite Structure Diagrams)
  - Kollaborationsdiagramm
- **Verhaltensdiagramme**
  - Interaktionsdiagramme
    - Sequenzdiagramme (Sequence Diagrams)
    - Kommunikationsdiagramme (Communication Diagrams)
  - Zustandsdiagramme (Statechart Diagrams)
  - Aktivitätsdiagramme (Activity Diagrams)
- **Anwendungsfalldiagramme (Use Case Diagrams)**
- **Implementierungsdiagramme**
  - Komponentendiagramme (Component Diagrams)
  - Verteilungsdiagramme (Deployment Diagrams)



# Modellmanagement

Zur Strukturierung der Klassen, Objekte, und Komponenten gibt es

- Packages,
- Subsysteme und
- Modelle

Ein integrierter Erweiterungsmechanismus ermöglicht ein Customizing. Es können neue und eigene Anforderungen berücksichtigt werden.



# Die Semantik der UML

Die UML hat eine (wohldefinierte) Semantik:

In einem Meta-Modell sind alle Beschreibungselemente verankert.

Im Sinne einer methodische Durchgängigkeit ist dieses Meta-Modell in UML Notation beschrieben.



Erweiterung der UML  
durch  
Erweiterung des Meta-Modells

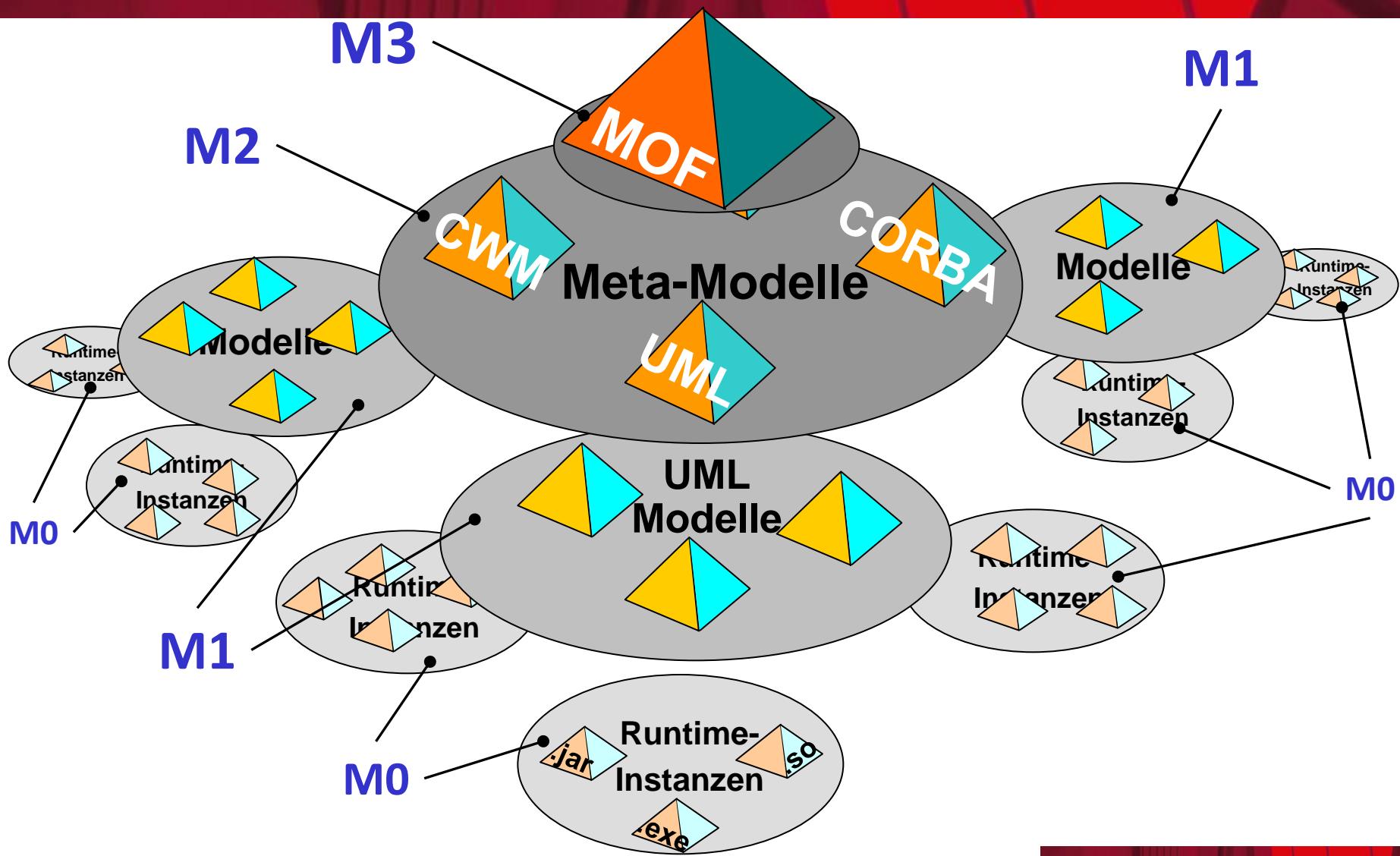


# Das Meta-Modell

Schicht	Beschreibung	Beispiel
M3 Meta-Meta-Modell	die Infrastruktur – beschreibt die Sprache, um Meta-Modelle zu spezifizieren	MetaKlasse, MetaAttribut, MetaOperation,
M2 Meta-Modell (abstrakte Syntax)	eine Instanz des Meta-Meta-Modells, Das Modell einer Spezifikationssprach wie etwa UML	Klasse, Attribut, Operation, Assoziation ...
M1 Modell	eine Instanz des Meta-Modells ein Diagramm mit Klassen, Attributen usw.	Auto, Kunde, Name, Anschrift, bonitaetPruefen
M0 reale Objekte (Ziel)	ein ausführbares Programm (Quellcode) Daten in Datenbanken.	„Mustermann“, 1234, ...

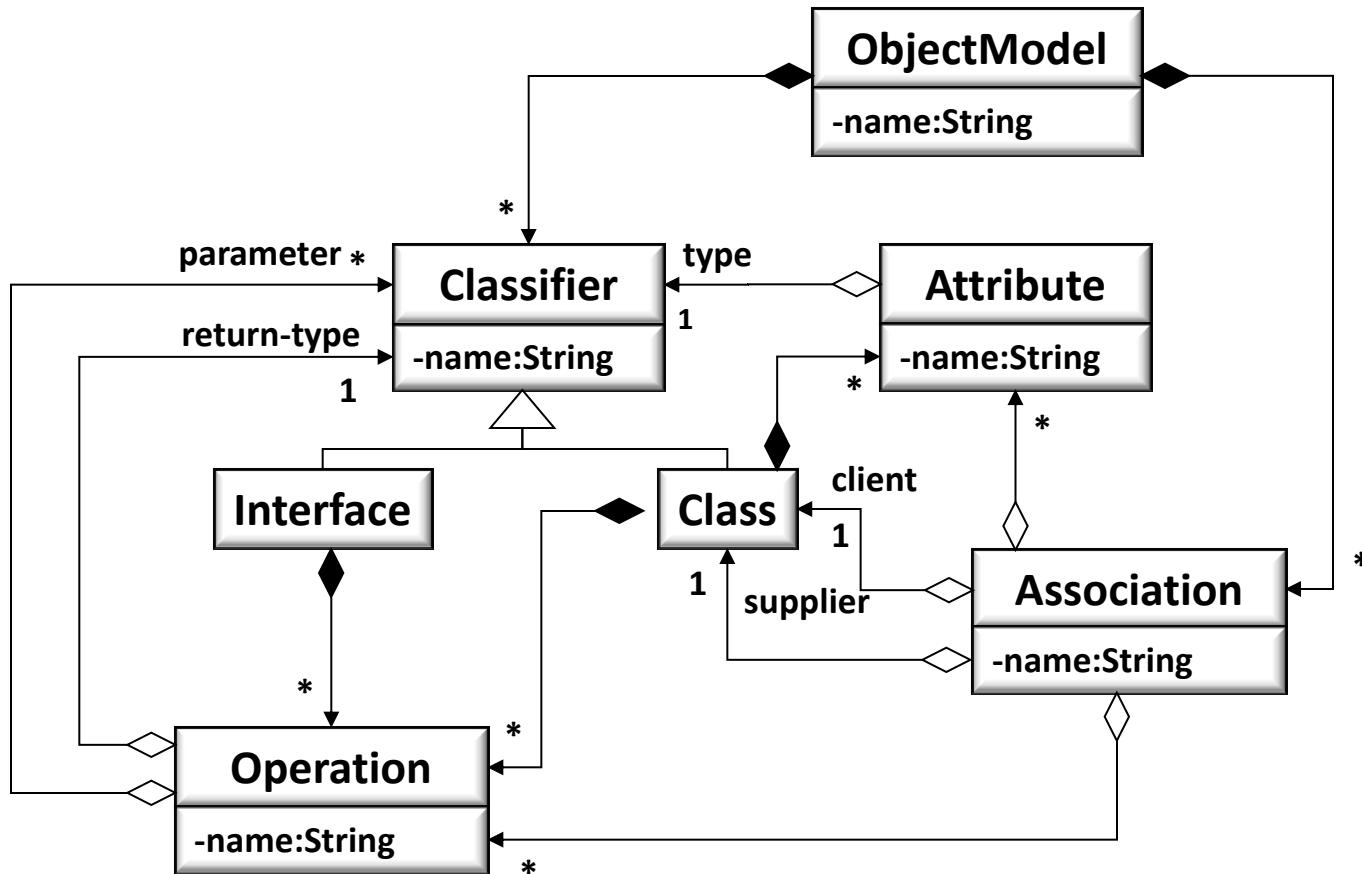


# Die (semantischen) Hierarchien der UML





# Beispiel: Ein einfaches Meta-Modell





Hochschule Karlsruhe  
Technik und Wirtschaft  
**UNIVERSITY OF APPLIED SCIENCES**

# Software-Engineering

(Rational) Unified Process

Prof. Dr. Thomas Fuchß  
Hochschule Karlsruhe – Technik und Wirtschaft  
Fakultät für Informatik und Wirtschaftsinformatik





# Der Unified Process

Jacobson I., Booch, G. and Rumbaugh

The unified software development process – Reading, Mass.: Addison-Wesley, 1999.

## Eigenschaften:

- use-case-driven,
- architekturorientiert,
- iterativ,
- inkrementell und
- komponentenbasiert



# Die Rolle der Use-Cases

Ein Softwaresystem wird erstellt, um Bedürfnisse seiner Anwender (Menschen, Systeme, ...) zu befriedigen. D.h., um ein erfolgreiches System zu entwickeln, ist es notwendig, dass diese Bedürfnisse (Requirements) erfasst und verstanden werden.

Zum Erfassen der funktionalen Anforderung an ein System verwendet man Use-Cases.

„What is the system supposed to do for each user?“

Alle Use-Cases zusammen bilden das Use-Case-Modell. Dies ersetzt das klassische funktionale Modell und dient als Basis für:

- die Analyse
- das Design
- die Implementierung
- die Tests



# Use-Cases

**Ein Use-Case ist im Allgemeinen eine sehr grobe Beschreibung eines Ablaufs aus Benutzersicht. Diese Beschreibung enthält viele Schritte und Aktionen.**

## Aufbau eines High-Level-Use-Case:

- Name: „Verb ...“
- Akteure:
- Typ: Prio 1, 2 oder 3
- Beschreibung: 3-4 Sätze
- Referenz: F1.1, ...

**Oft ist es notwendig, einen höheren Detaillierungsgrad zu verwenden, um ein tieferes Problemverständnis zu erreichen. Dies geschieht über „erweiterte Use-Case“; diese enthalten zusätzlich eine detaillierte Ablaufbeschreibung.**



# Die Rolle der Architektur

**Use-Cases werden nicht isoliert betrachtet und entwickelt. Sie stehen in engem Zusammenhang zur Architektur.**

Architektur und Use-Cases beeinflussen sich gegenseitig und reifen während des Entwicklungsprozesses.

Die Aufgabe der Architektur ist es, die signifikanten (hervorstechenden) statischen und dynamischen Aspekte des Gesamtsystems zu beschreiben, d.h., die Struktur der Subsysteme und Komponenten, deren Interfaces, Verbindungen, Interaktion, Verantwortlichkeiten und wie sie zusammenarbeiten. Hinzu kommen Aspekte, die die

- **Verwendung,**
- **Leistung,**
- **technologische Randbedingungen**
- **Kompromisse,**
- **Verständlichkeit**

**betreffen**



# Formen der Architektur

- **funktionale Architektur**

(Gegenstand der Analyse)

- funktionale Bereiche
- Datenfluss

- **logische Architektur**

(Übergang zwischen Analyse und Design)

- Komponenten
- Schichten

- **technische Architektur**

(Übergang zwischen Design und Implementierung)

- Komponenten
- Schichten



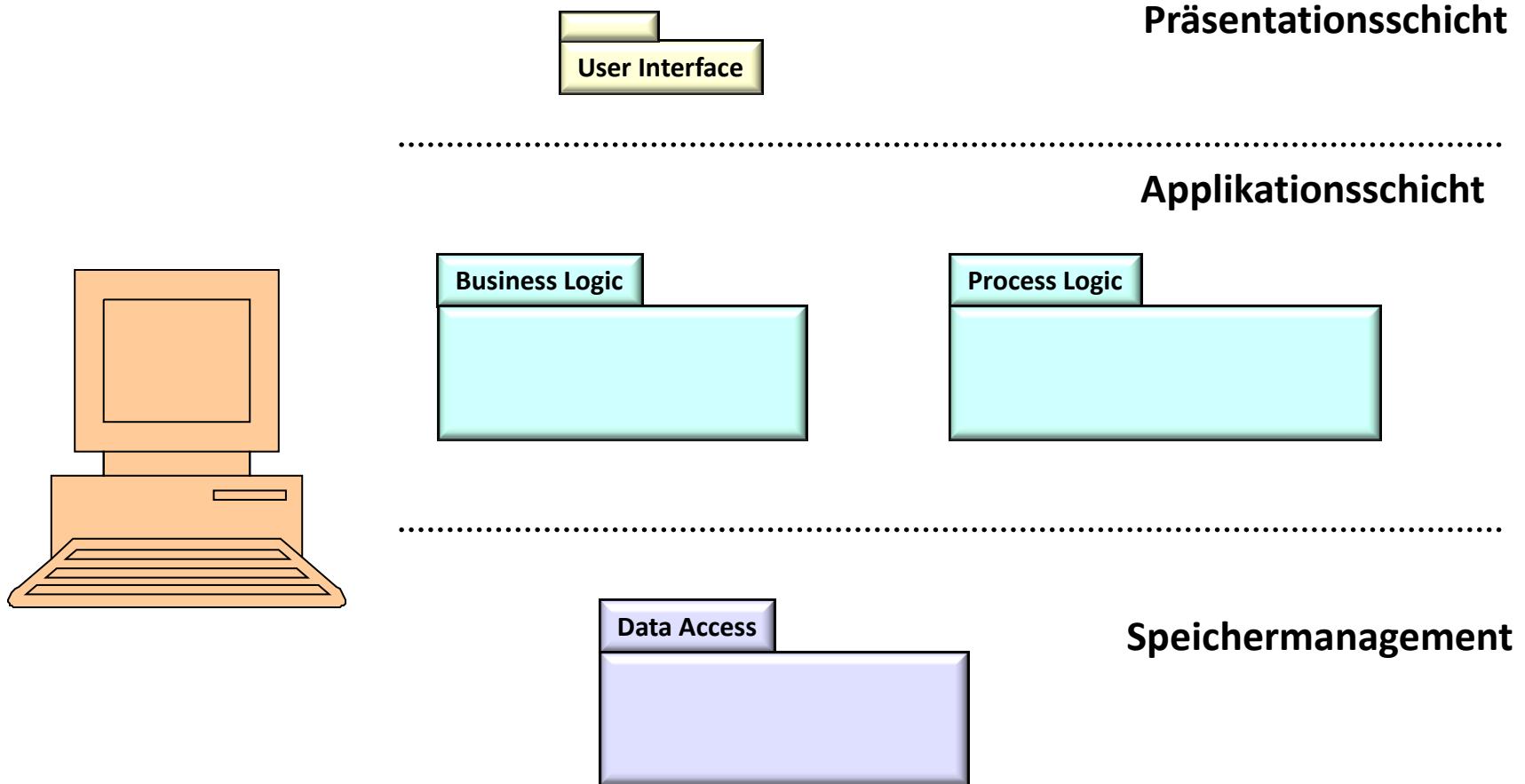
# Eigenschaften einer Architektur

**Eine gut entworfene Architektur hat folgende Eigenschaften:**

- sie besitzt mehrere Schichten
- sie reduziert die Kopplung zwischen Subsystemen
- sie hat wohldefinierte Schnittstellen zwischen Systemen und Subsystemen
- sie ist leicht verständlich
- sie ist robust und skalierbar
- sie hat eine Vielzahl wiederverwendbarer Komponenten
- sie basiert auf den wichtigsten Use-Cases

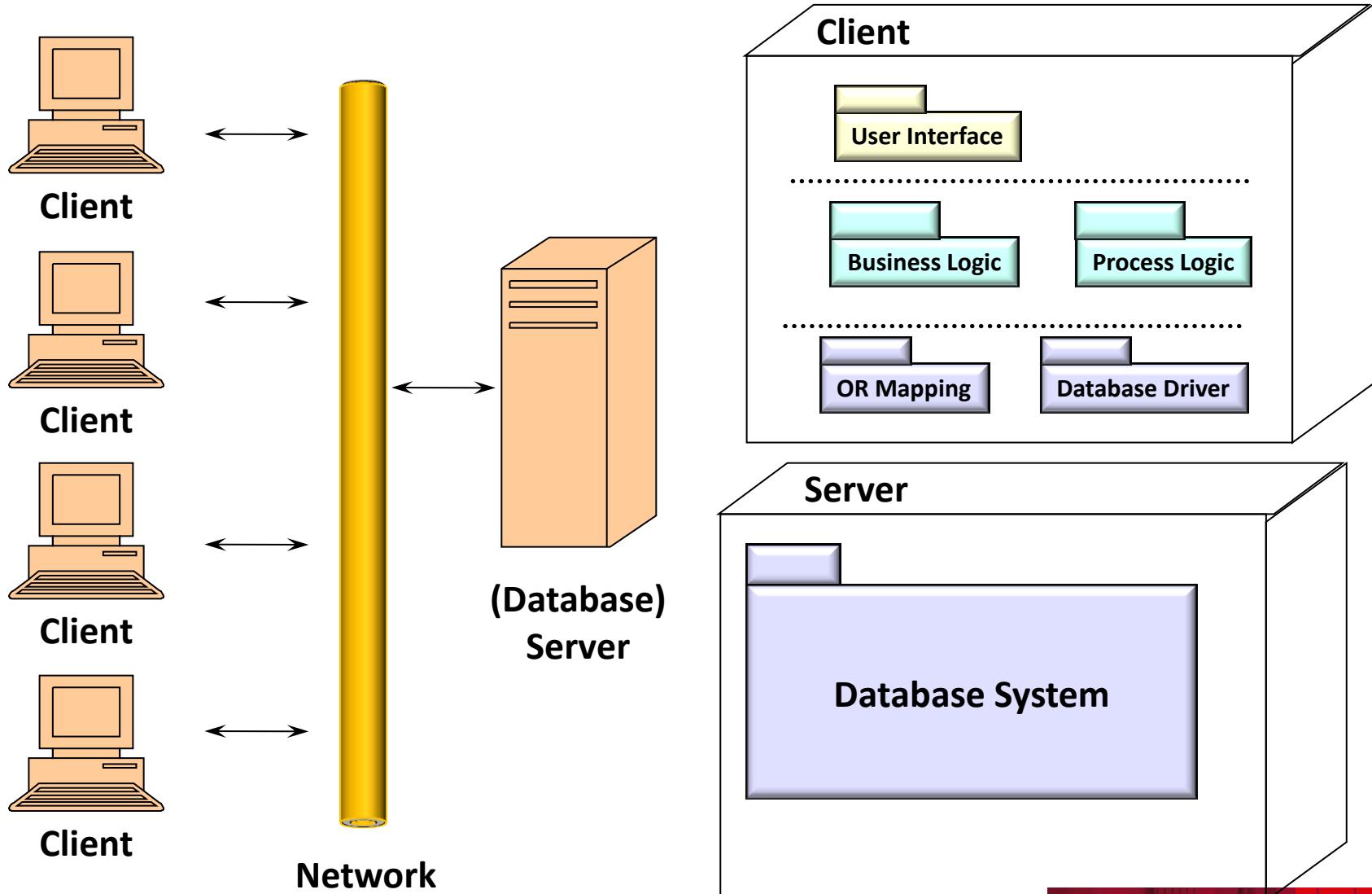


# Beispiel: Eine 3-Schichten-Architektur



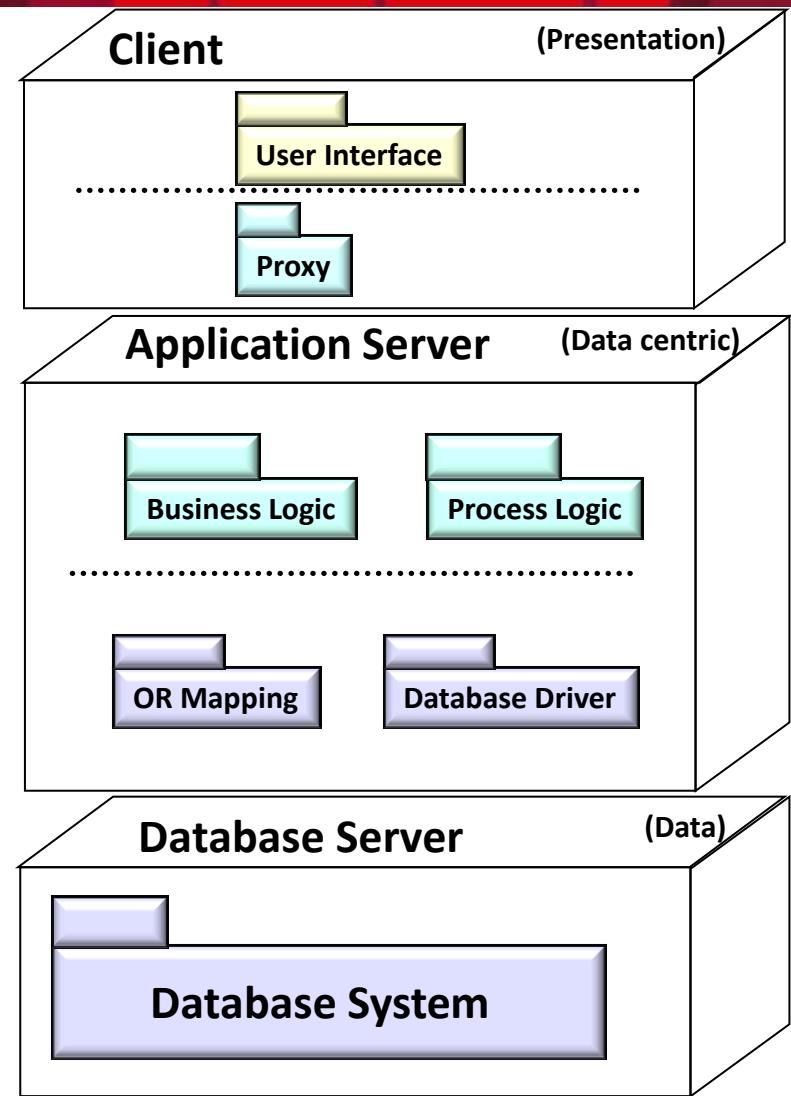
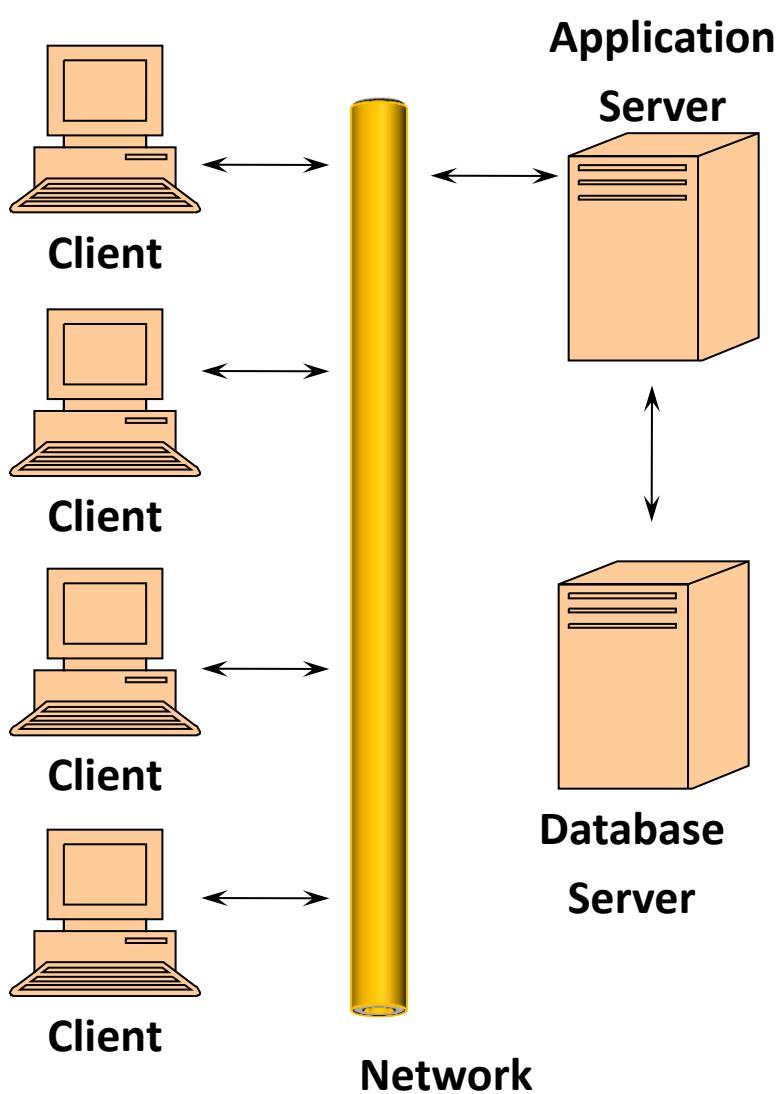


# Beispiel: Eine 2-Tier-Architektur



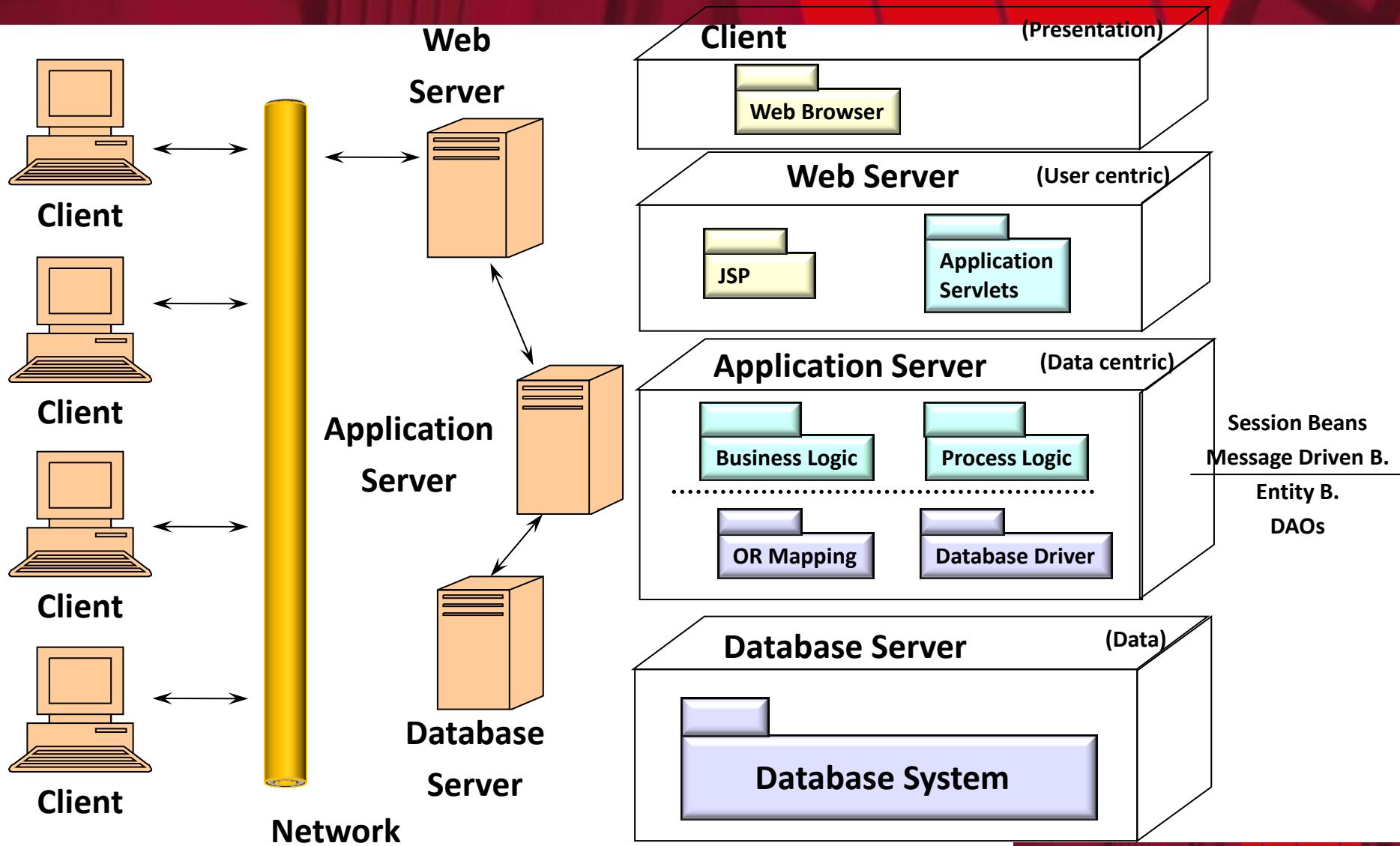


# Beispiel: Eine 3-Tier-Architektur





# Beispiel: Eine 4-Tier-Architektur





# Die Beziehung zwischen Use-Cases und Architektur

**Jedes Ding besteht aus Funktion und Form  
– nur eines davon ist nicht genug.**

## Ablauf:

**Funktion ≡ Use-Cases**

**Form ≡ Architektur**

- Man erstellt einen ersten Ansatz für die Architektur, dieser ist noch unabhängig von Use-Cases. Er beruht auf der Art des Systems, den Plattformen, dem Einsatzbereich usw. Ein Grundverständnis für die Anforderungen des Systems ist erforderlich.
- Use-Cases für die Kernfunktionalität (5%-10% aller Use-Cases) werden erstellt unter Berücksichtigung der (noch sehr) groben Architektur.  
(Was ist System, was ist Umgebung?)
- Jeder Use-Case wird (detailliert) spezifiziert und designed. Dies führt zu Subsystemen, Partitionen und Schichten.
- Aus dieser Architektur entstehen neuen Anforderungen und daraus neue Use-Cases.
- ...

**Architektur und Use-Cases reifen.**



# Die Notwendigkeit der Iteration

Die Entwicklung eines großen Systems ist meist langwierig  
(mehrere Personenjahre)



man zerlegt die Aufgabe in Mini-Projekte

- Jedes Mini-Projekt ist eine Iteration im Workflow und ein Inkrement des Gesamtsystems.
- Jedes Mini-Projekt bezieht sich auf Aspekte mit großen Risiken
- Jedes Mini-Projekt realisiert einen zusammenhängenden Satz von Use-Cases
- Jedes Mini-Projekt schreibt die **Artefakte (Modellelemente)** des vorherigen weiter.

Mini-Projekte sind nicht immer additiv, Designs können geändert werden.

Die Iterationen sollten geplant sein und einer durchgängigen Linie folgen. Schritte, die das Produkt dem Ziel nicht näher bringen, müssen unterbleiben.

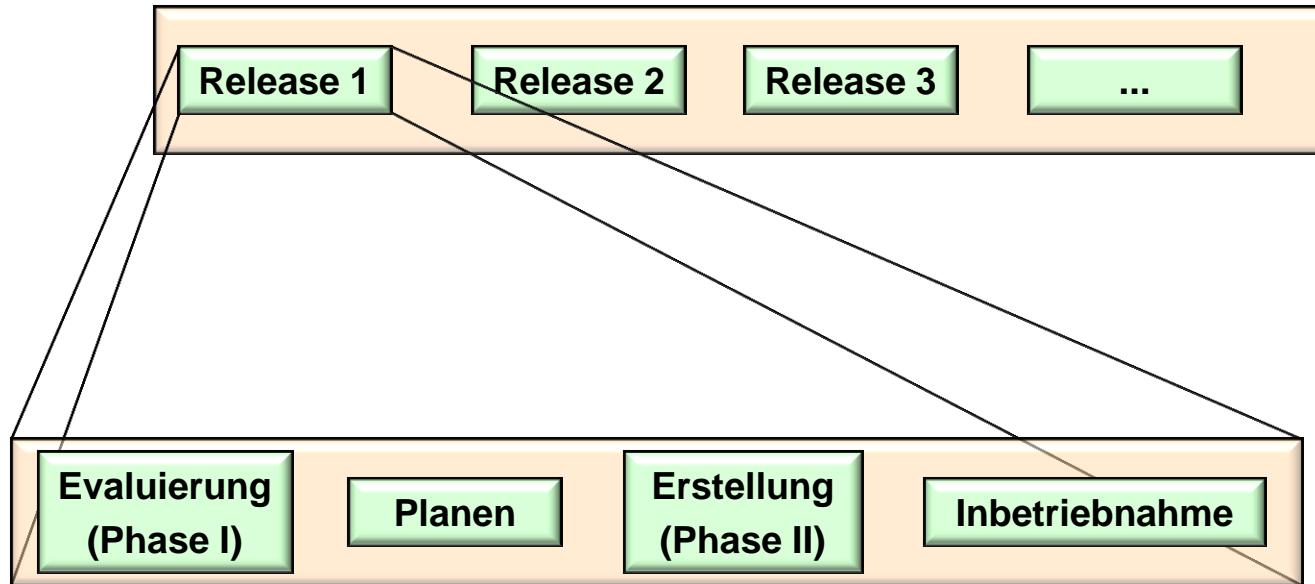


# Ein Iterativer Entwicklungsprozess

Larman, Craig.

Applying UML and patterns : an introduction to object-oriented analysis and design and the Unified Process, 2. ed. – Upper Saddle River, NJ : Prentice Hall, 2002.

## In vier Schritten zum Release





# Der grobe Ablauf

- **Evaluierung:**

Anforderungen (Requirements) definieren und mehrere Entwicklungszyklen durchführen, bis ein realistischer Plan erstellt werden kann. Ziel ist es, das Problem zu verstehen und die Komplexität der Aufgabe einzuschätzen. Dies schafft Planungssicherheit.

- **Planen:**

Zeitplan, Budget, Personalplan usw. anpassen, basierend auf den Ergebnissen der Evaluierung.

- **Erstellung:**

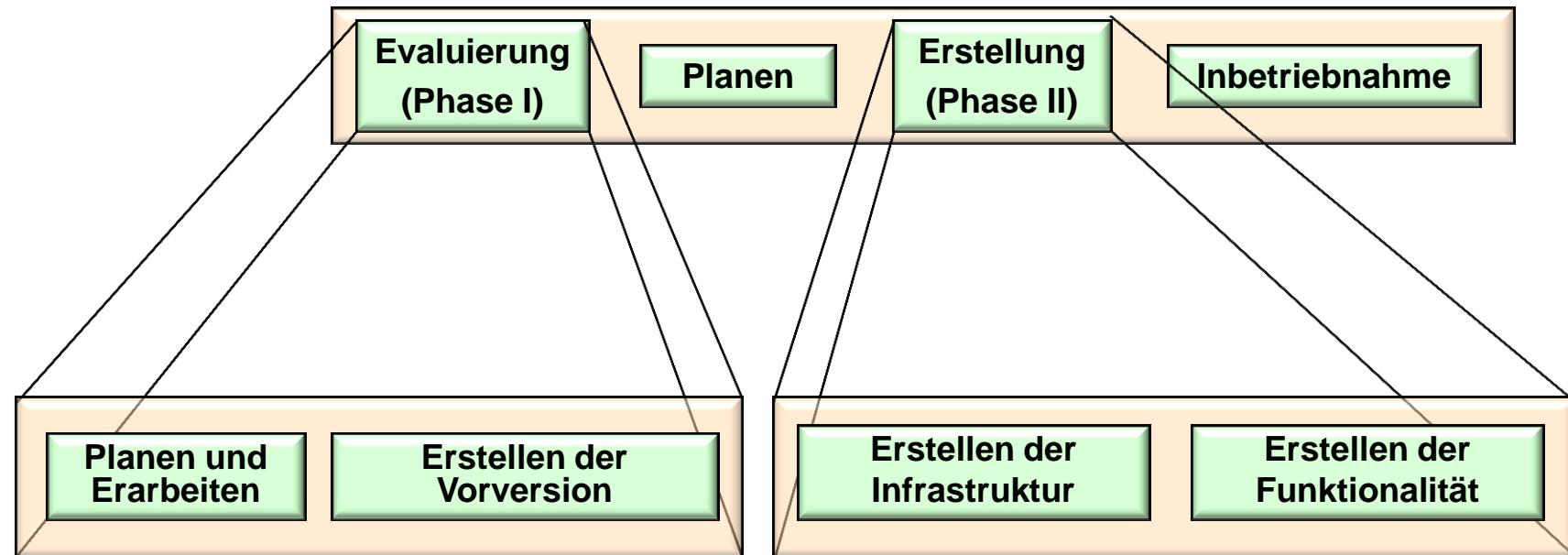
Die Hauptphase, hier wird das eigentliche Produkt realisiert.

- **Inbetriebnahme (Auslieferung):**

Das System geht in Betrieb, Übergang zur Gewährleistung und Wartung



# Die Hauptaufgaben





# Die Hauptaufgaben Phase I

- **Planen und Erarbeiten**

Die Anforderungen erfassen, sammeln, zusammenstellen, gruppieren, gewichten. Die zentralen Requirements herausarbeiten, das Problem erarbeiten und ein Problemverständnis gewinnen.

Erste Use-Cases und die prinzipielle Architektur festlegen.

- **Erstellen der Vorversion**

Die ersten Entwicklungszyklen durchführen (etwa 1-3 Monate). Ziel ist es, primäre Use-Cases umzusetzen und Architekturgesichtspunkte zu fixieren.

**Das Problemverständnis wächst,  
die Komplexität der Aufgabe wird ersichtlich,  
Planungssicherheit entsteht.**



# Die Hauptaufgaben Phase II

- **Erstellen der Infrastruktur**

In den ersten Zyklen der eigentlichen Entwicklungsphase werden die Domain Use-Cases umgesetzt. Dies ist verbunden mit der Konsolidierung der Systemarchitektur. Die Subsysteme und Schichten werden definiert und festgelegt.

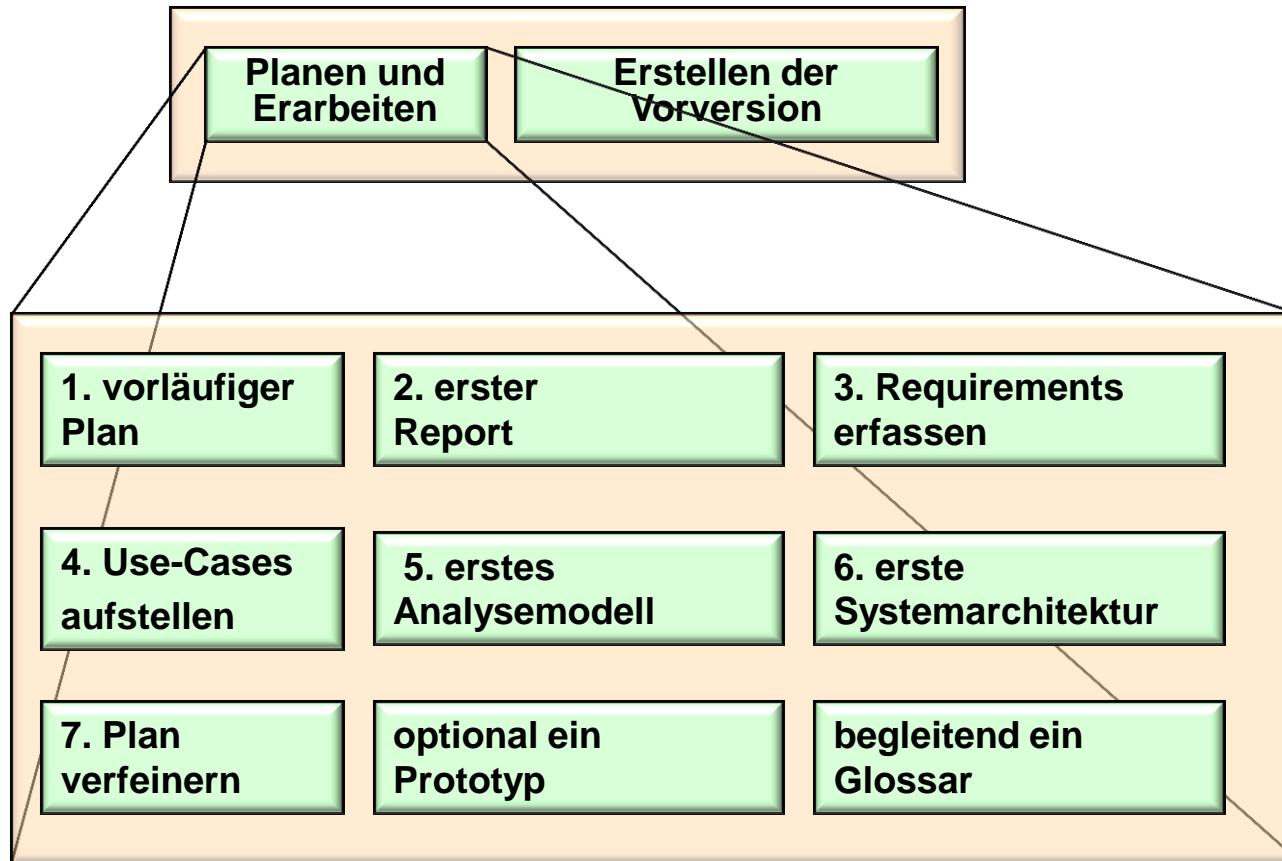
## Architektur und Use-Cases reifen

- **Erstellen der Funktionalität**

Die späteren Zyklen konzentrieren sich dann auf die Vervollständigung der Funktionalität. Dies beeinflusst die Systemarchitektur nur unwesentlich. (Schnittstellen werden gegebenenfalls erweitert)



# Planen und Erarbeiten im Detail





# Planen und Erarbeiten im Detail II

- **vorläufiger Plan:**  
geschätzte Zeit, geschätztes internes Budget, geschätzter Personalbedarf usw. (Aufgabe realistisch?)
- **erster Report:**  
Motivation, Alternativen, Marktpotential, Zielsetzung, ...  
(Aufgabe erfolgversprechend)
- **Zentrale Requirements erfassen:**  
Meist eine deklarative Auflistung der Anforderungen aus Kundensicht (Lastenheft). Diese können in Form eines Workshops zusammen mit dem Auftraggeber erarbeitet werden.



# Planen und Erarbeiten im Detail III

- **Use-Cases aufstellen:**

Beschreibung der (funktionalen) Anforderung an das System und deren Beziehungen (aus Entwicklersicht).

- **erstes Analysemodell:**

Sehr grob und vorläufig. Dient als Hilfestellung, um das Problem, seine Struktur, die Bedürfnisse des Kunden besser zu verstehen. Es ist eng korreliert zu den Use-Cases und Requirements.

- **erste Systemarchitektur:**

Ergibt sich aus dem Analysemodell und den Requirements.

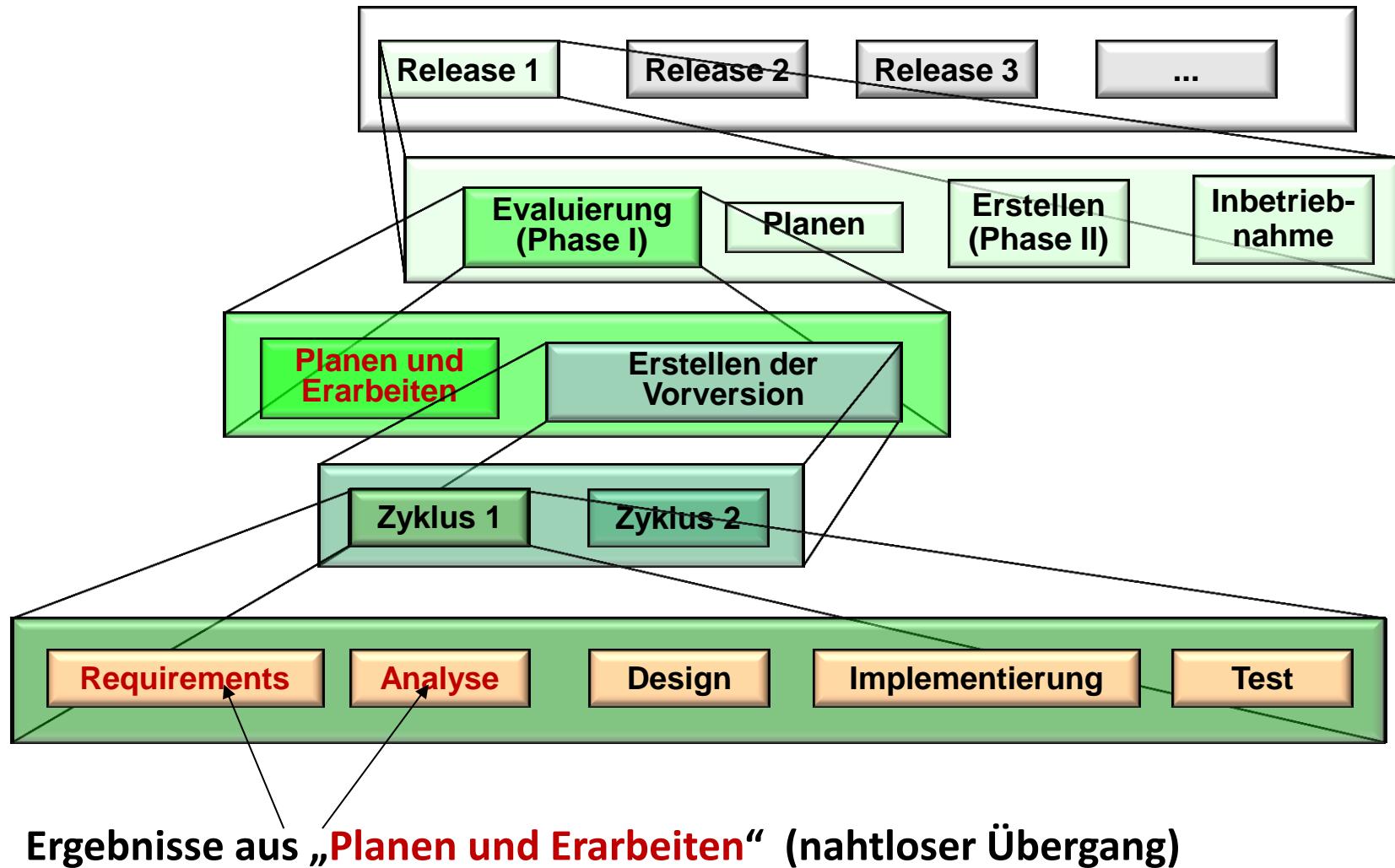


# Planen und Erarbeiten im Detail IV

- **Plan verfeinern:**  
Festlegen, welches die zentralen Aspekte sind, die zuerst realisiert werden sollen. Die Zyklen, Zeitdauer und Umfang der Vorversion festlegen.
- **optional ein Prototyp:**  
Gegebenenfalls kann ein Prototyp erstellt werden. Ein Prototyp klärt offene Fragen bzgl. der Kundenwünsche und Anforderungen, schafft Planungssicherheit und reduziert das Risiko einer späteren Fehlentwicklung.
- **begleitend ein Glossar:**  
Alle Bezeichnungen, Klassen, usw. werden in einem Glossar gelistet und definiert (Data-Dictionary). Dies schafft Klarheit und reduziert das Risiko von Missverständnissen.



# Erstellen der Vorversion





# Der Unified Process ist ein Framework

**Er muss angepasst werden an:**

- die Größe des zu erstellenden Systems
  - das Einsatzgebiet des zu erstellenden Systems
  - die Fähigkeiten und Erfahrungen der Entwickler
  - die Unternehmensstruktur
- • Wie lang sind die einzelnen Phasen?  
• Wie viele Iterationen sind notwendig?  
• Wann ist die Architektur abgeschlossen?  
• ...

**Im Unified Process verbinden sich Objektorientierung und die Erkenntnisse über Prozessmodelle.**



# Der endgültige Projektplan

**Nach der Evaluierungsphase sollte Planungssicherheit bestehen.  
Die Ergebnisse der Evaluierungsphase müssen mit dem Auftraggeber  
abgestimmt werden.**

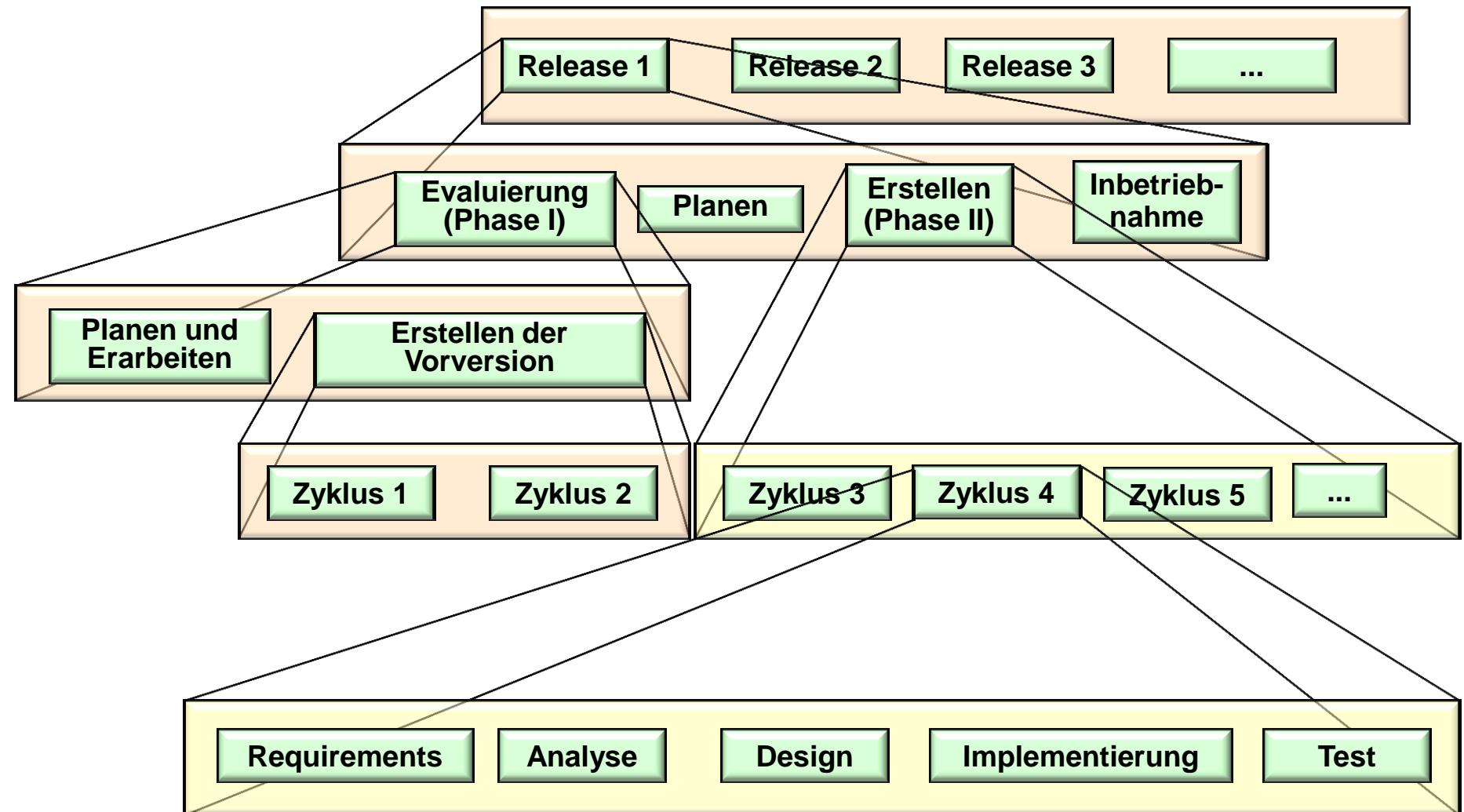
**Jetzt fällt die Entscheidung über Realisierung oder nicht  
(Kosten und Zeitverlauf lassen sich bestimmen)**

**Wenn das Projekt realisiert wird, dann erfolgt jetzt die Erstellung des  
endgültigen Projektplans.**

- Die Entwicklungszyklen mit den Meilensteine werden definiert (Netzplan),
- Die Zahlungspläne werden ausgehandelt,
- Personalverteilung und die Teams werden aufgestellt (Organisationsstruktur),
- Externe Kräfte, Zulieferer und Qualitätssicherung werden gebucht,
- Entscheidungen über Programmiersprachen und Werkzeuge werden getroffen,
- Schulungen werden geplant,
- Hardwarebeschaffung wird geplant ...



# Die iterative Entwicklung (Zyklen)





# Die Phasen eines Zyklus

- **Requirements:**  
Die relevanten Use-Cases bestimmen und das Use-Case-Modell erstellen.
- **Analyse:**  
Die Use-Cases analysieren, das Analysemodell erstellen, die Architektur anpassen.
- **Design:**  
Ein Design auf der Basis der gewählten Architektur erstellen.
- **Implementierung:**  
Das Design komponentenorientiert implementieren.
- **Test:**  
Verifizieren, dass die Komponenten die Use-Cases erfüllen.

**Modelle und Dokumente (Artefakte) synchronisieren und konsistent halten (zyklenübergreifend)**



# Die Länge eines Entwicklungszyklus

Sind sie zu kurz, wird es schwierig eine sinnvolle Menge von Use-Cases umzusetzen. Sind sie zu lang, läuft man Gefahr die Komplexität falsch einzuschätzen und das Feedback auf die ersten Ergebnisse entfällt.

**Faustregel:**

**Ein Zyklus sollte nicht kürzer als 2 Wochen und nicht länger als 2 Monate sein.**

## Faktoren, die Zyklen gefährden:

- Unterschätzen der Komplexität – insbesondere in frühen Phasen. Die Erarbeitung der Architektur ist meist aufwendiger als gedacht.
- Einführung neuer Technologien
- Personalprobleme (Mangel an Experten)
- Parallele Entwicklungsteams erfordern eine erhöhte Koordination und Kommunikation (dies muss beachtet werden)
- Zu große Teams



# Feste Zeitschränken

Eine sinnvolle Strategie ist es, einen Entwicklungszyklus mit festen Zeitschränken zu versehen. Innerhalb dieser Schranken, muss der komplette Zyklus erarbeitet werden.

Vier Parameter beschreiben die Entwicklung:

- Kosten
- Zeit
- Umfang
- Qualität

Funktionalität

Alle vier können nicht gleichzeitig fixiert werden.  
Meistens wird versucht Kosten, Zeit und Umfang  
festzulegen.

verschätzt → schlechte Qualität



# Feste Zeitschränken

Wird die Zeit fixiert und gehen wir davon aus, dass die Qualität nicht wirklich variabel ist, dann bleiben zwei Steuergrößen:

- Kosten (Personal)
- Umfang

Da „*auch neun Frauen kein Kind in einem Monat gebären*“ und die Hinzunahme weiterer Personen zu einem verspäteten Projekt dies noch mehr verspätet, bleibt als einzige realistische Steuergröße, der **Umfang**.

verspätet →

**Umfang reduzieren und auf spätere Entwicklungsschritte verschieben**

[Frederick P. Brooks]



# Der Inhalt eines Zyklus

**Die Entscheidung darüber, welche Anforderungen innerhalb eines Entwicklungszyklus realisiert werden, sollte immer in Zusammenarbeit mit dem Entwicklungsteam erfolgen – sie sind diejenigen, die die Ergebnisse liefern müssen.**



# Parallele Entwicklungsschritte

**Ein Großprojekt wird selten linear entwickelt, meist arbeiten mehrere Teams parallel an verschiedenen Aufgaben.**

**Idealerweise werden die einzelnen Teams über die Architektur organisiert und koordiniert.**

**z.B.: anhand der Schichten**

- Domain-Entwicklungsteam (entwickeln die eigentliche Anwendungslogik)
- User-Interface-Entwicklungsteam (entwickelt die Schnittstelle zum Anwender)
- Service-Team (entwickelt die unteren Schichten)

**Der gesamte Verlauf der Aktivitäten der einzelnen Entwicklungsteams kann in einem Activity-Diagramm modelliert werden.**

**Jedes Team bekommt seine eigene Swimlane.**



# Parallele Entwicklungsschritte II

**Sobald mehr als ein Team an einem Projekt arbeitet, gilt es Abhängigkeiten zu beachten, zu analysieren und die Zusammenarbeit zu koordinieren.**

- Benutzerschnittstelle und Anwendungslogik muss aufeinander abgestimmt sein.
- Anwendungslogik verwendet die Service-Schichten (Speichermanagement, Kommunikationsprotokolle usw.)
- Das Testteam kann nur testen, wenn entsprechende Funktionalität fertiggestellt ist.
- usw.



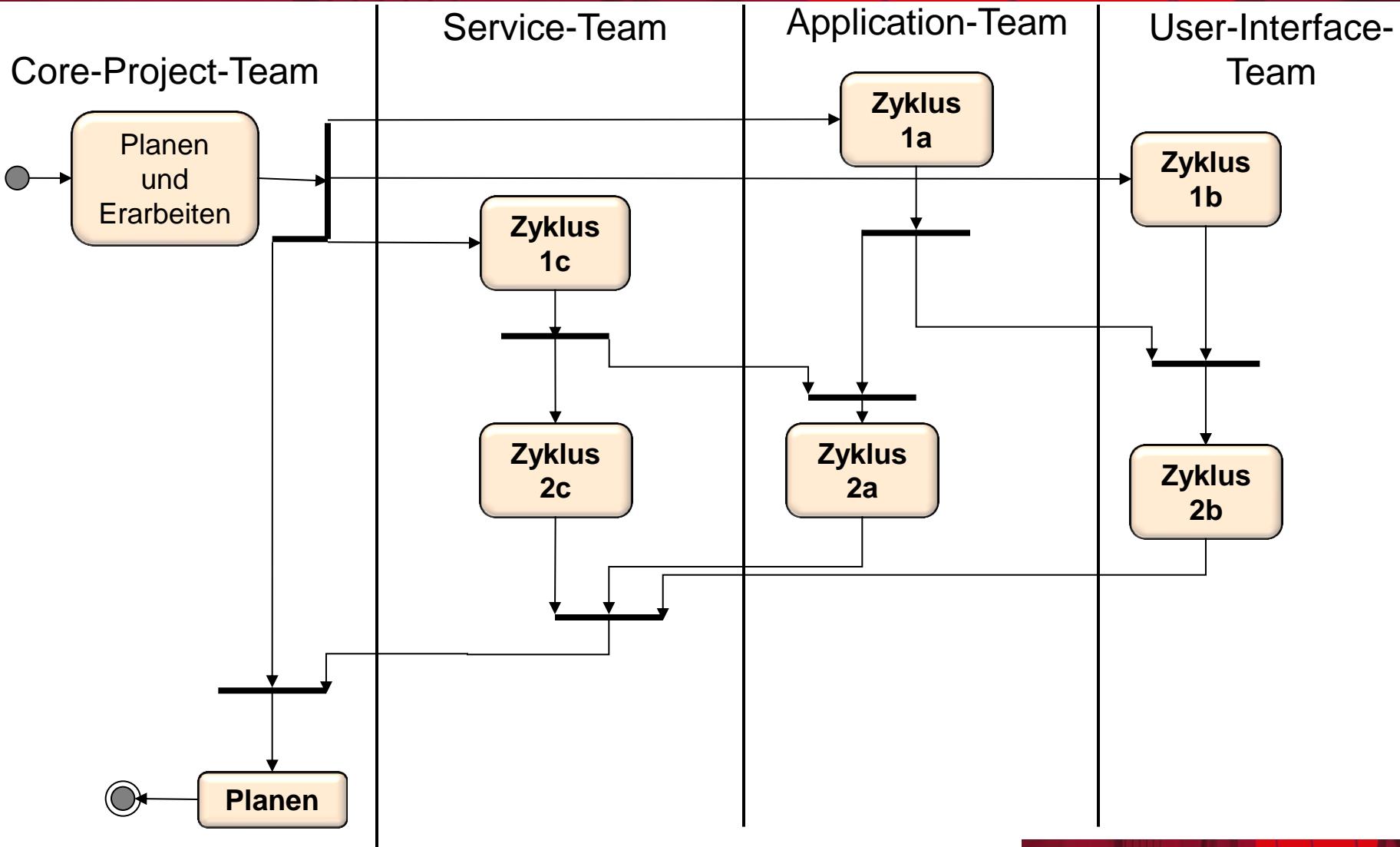
# Parallele Entwicklungsschritte III

Um zuviel Linearisierung zu vermeiden und den Vorteil der parallelen Teams zu erhalten, gibt es mehrere Möglichkeiten:

- **Stubs (Stummel):**  
Notwendige aber noch nicht vorhandene Funktionalität wird mittels Dummys überbrückt (liefern immer konstante Werte o.ä.)
- **Gestaffelte Entwicklung:**  
Entwicklungsteams verwenden Resultate früherer Zyklen anderer Teams.



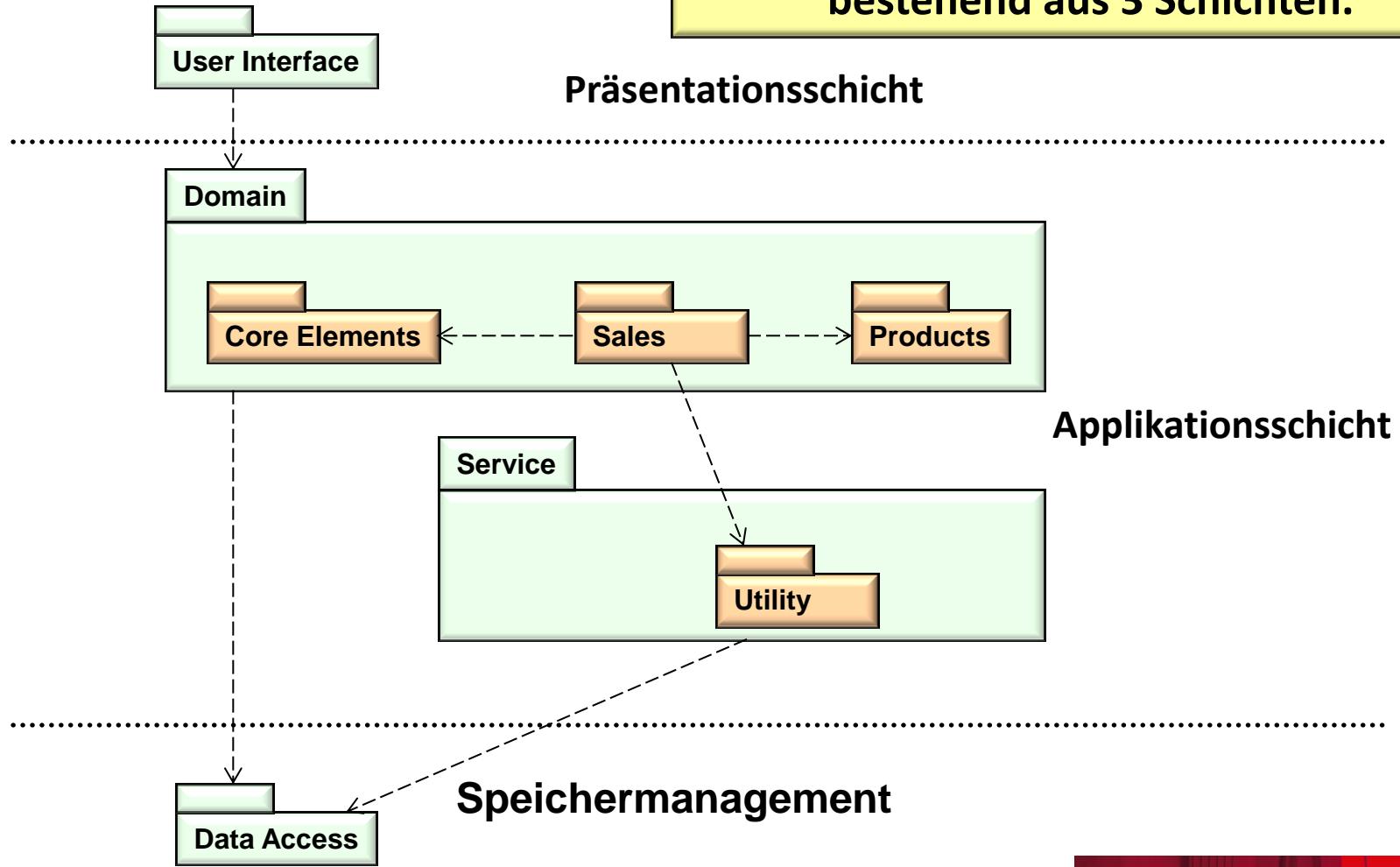
# Beispiel:





# Aufteilung der Zyklen

Ein typisches Anwendungssystem  
bestehend aus 3 Schichten.





# Aufteilung der Zyklen II

- Das User-Interface ist die sichtbare Realisierung des Systems. Es ist das was Nicht-Techniker vom System sehen und für das System halten. Ein ansprechendes User-Interface (bereits in frühen Zyklen) erzeugt zufriedene Kunden.  
**(Nachteil: Der Kunde glaubt das System sei fertig.)**
- Falls Domain- und Service-Layer (innerhalb eines Zyklus) vom selben Team realisiert werden, beginnt man mit dem Domain-Layer. Bis die Funktionalität der zu erfüllenden Use-Cases realisiert ist. (Eventuell benötigte Service-Funktionen werden als Dummys bereitgestellt). Erst nach der Realisierung der Domain-Funktionalität werden die benötigten Service-Funktionen realisiert.

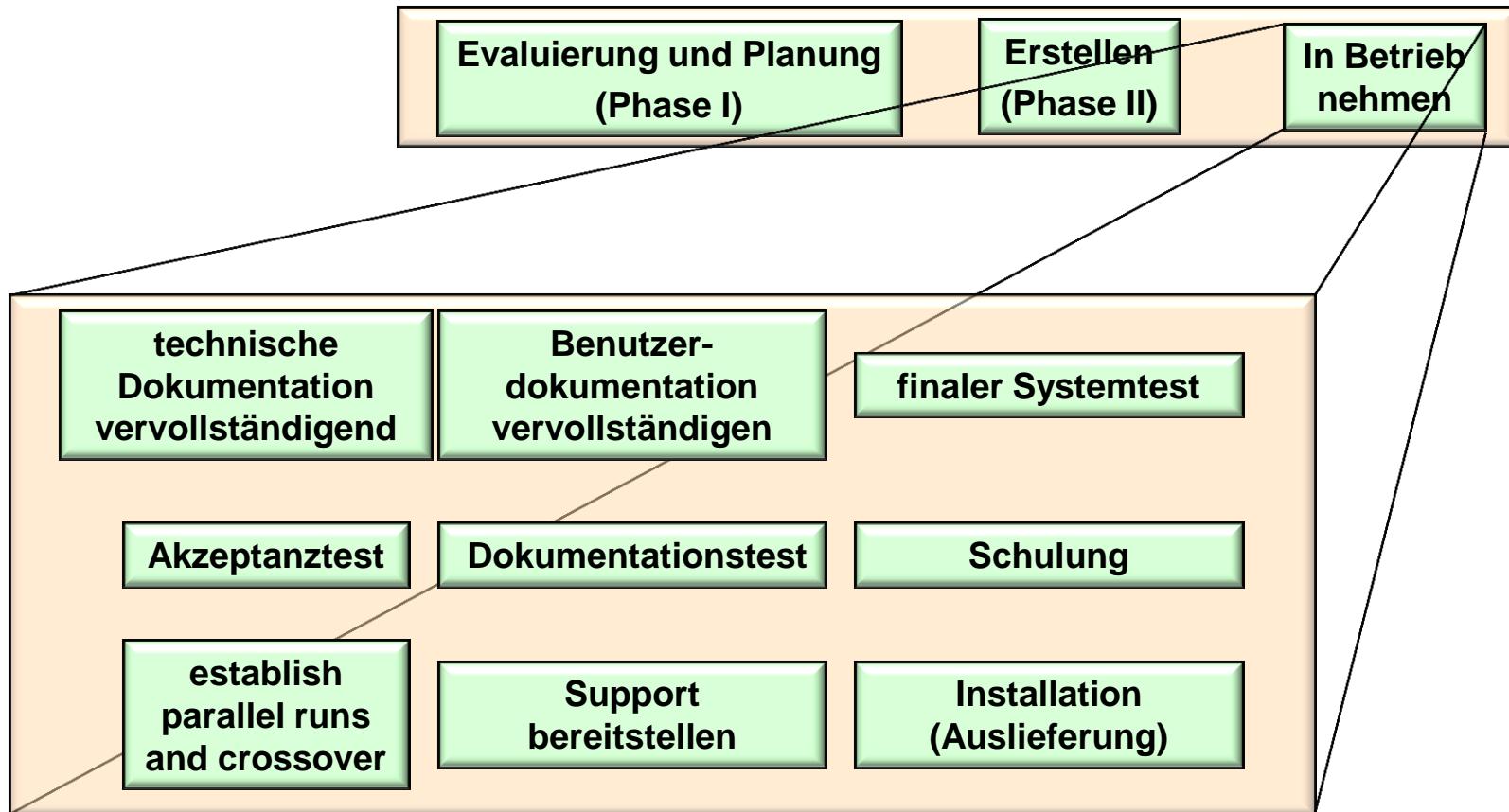


# Aufteilung der Zyklen III

- Die Realisierung des Speichermanagements sollte zeitig aber nicht sofort beginnen.
  - Datenbankentwicklungen sind zeitraubend.  
Dies kann zur Verlagerung von Prioritäten führen (weg vom Design).
- Die Realisierung des Speichermanagements sollte nicht zu spät beginnen.
  - Dies kann aufgrund der Verzahnung zwischen Speichermanagement und Applikationsschicht zu größeren Designänderungen führen.
- Innerhalb eines Zyklus realisiert man nur das, was gebraucht wird.
  - Das Speichermanagement orientiert sich stets an den Bedürfnissen des aktuellen Zyklus.

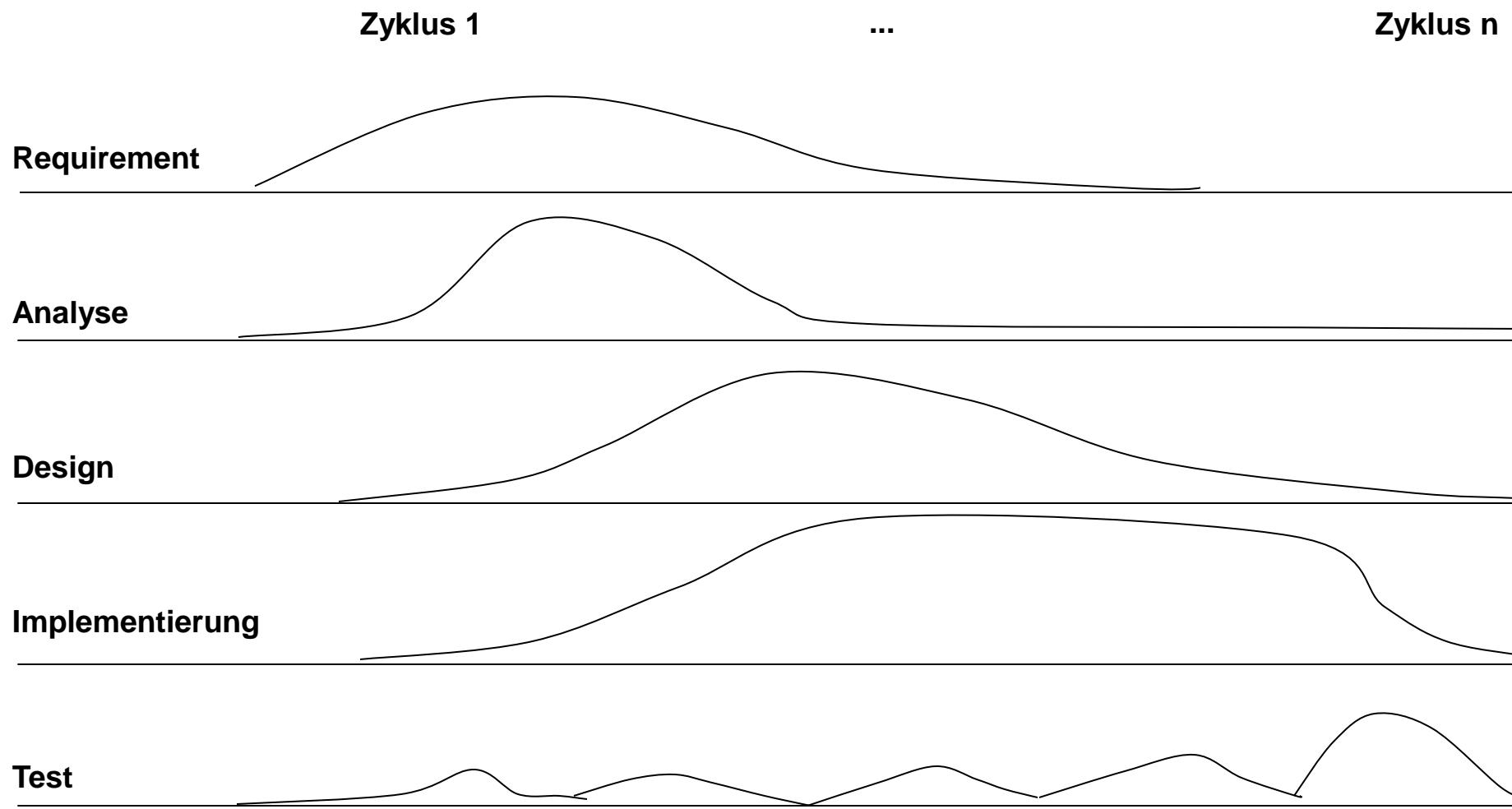


# Die Inbetriebnahme





# Verteilung der Aktivitäten





Hochschule Karlsruhe  
Technik und Wirtschaft  
**UNIVERSITY OF APPLIED SCIENCES**

## **Software-Engineering**

# **Objektorientierte Analyse (OOA nach Craig Larman)**

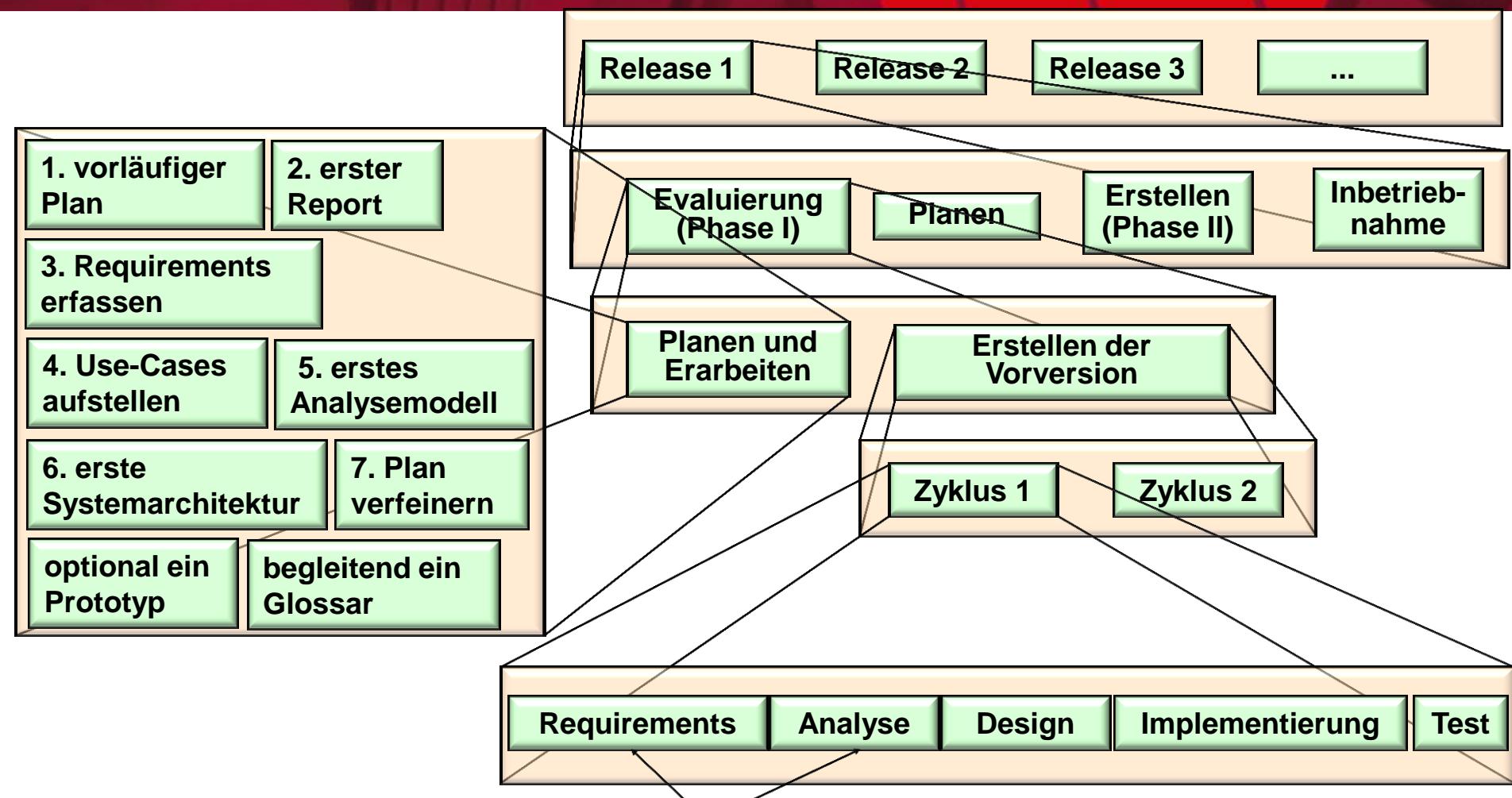
Prof. Dr. Thomas Fuchß  
Hochschule Karlsruhe – Technik und Wirtschaft  
Fakultät für Informatik und Wirtschaftsinformatik





# Analyse Zyklus 1

(nach Craig Larman: Applying UML and patterns)



Ergebnisse aus „Planen und Erarbeiten“ (nahtloser Übergang)



# Analyse

## Ziel:

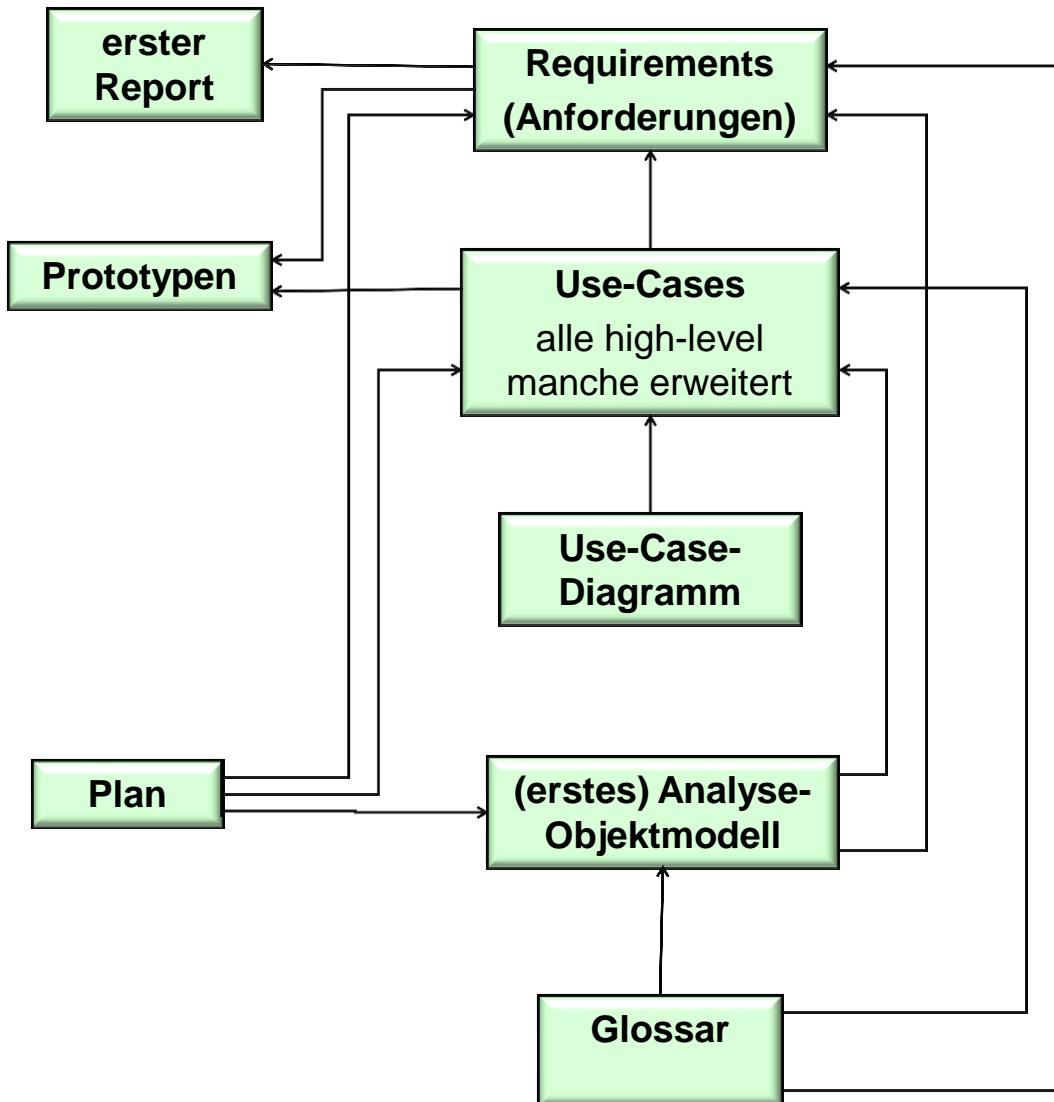
**Das Problem verstehen und beschreiben, so dass ein korrekter Entwurf möglich wird. Hierzu wird ein Analysemodell erstellt.**

## Dies umfasst:

- **Das Analyse-Use-Case-Modell**, die Beschreibung der Anforderungen
- **Das Analyse-Objektmodell**, die statische Struktur der realen Welt
  - Klassen finden und organisieren
  - Verantwortlichkeiten und Beziehungen aufzeigen
- **Das funktionale Modell**,
  - das Systemverhalten beschreiben
  - die Interaktionen bestimmen, Reihenfolgen festlegen, wichtige Parameter berücksichtigen
- **das Analyse-State-Modell**, Zustandsübergangs- und Activity-Diagramme zentraler Objekte und Use-Cases.



# Abhängigkeiten der Artefakte





# Requirements

**Requirements** sind Beschreibungen der Anforderungen und Wünsche für und an das **Produkt; was soll es tun, wie soll es sein?**

**Problem verstehen, bedeutet Anforderungen erkennen, analysieren und bewerten.**

- Aufgabenbeschreibung (um was geht es)
- Anwender bestimmen
- Ziele bestimmen
- Systemfunktionalität erfassen (das System macht „X“)
- Systemeigenschaften erfassen (das System soll „y“ sein)
- Systemgrenzen definieren

**Anforderungen sind so zu formalisieren, dass sie unzweideutig sind.**



# Beispiel: Eine Kassenanlage

- **Aufgabenbeschreibung:** Entwicklung eines Kassensystems mit Lagerverwaltung für einen Supermarkt.
  - **Anwender:** Gut-und-Billig, ... als Ersatz für das bisherige System ...
  - **Ziele:** Verbesserung des Erfassungs- und Abrechnungsvorgangs an den einzelnen Kassen. Durchsatz soll erhöht werden, ...  
(schöner, schneller, weiter)
    - Schnelle Abfertigung für die Kunden
    - Leichte Bedienbarkeit durch das Personal
    - Direkte Ankopplung an die bestehende Lagerverwaltung
- ...



# Systemfunktionalität erfassen

(das System macht „X“)

Funktionen müssen kategorisiert und gruppiert werden.  
Nur dann ist eine Priorisierung möglich.

- **Kategorien** könnten sein:
  - **sichtbar**: Funktionalität ist für den Anwender ersichtlich
  - **versteckt**: Wird von Anwender nicht wahrgenommen (Service-Funktionen)
  - **optional**: nice-to-have (beeinflusst nicht andere Funktionen)
- **Gruppierungen** entstehen durch Zuordnungen im Anwendungsgebiet
  - **Basisfunktionalität**:
    - Rechnungssumme bestimmen
    - Transaktion speichern
    - ...
  - **Bezahlfunktionalität**
    - Barzahlung abwickeln
    - Kreditkartenzahlung loggen
    - ...



# Die ersten Einträge ins Glossar

## Basisfunktionen

<u>Ref.#</u>	<u>Funktion</u>	<u>Kategorie</u>
F1.1	Rechnungssumme bestimmen: Die ...	sichtbar
F1.2	Transaktion speichern: Die ...	versteckt

## Bezahlfunktionen

<u>Ref.#</u>	<u>Funktion</u>	<u>Kategorie</u>
F2.1	Barzahlung abwickeln: Die ...	sichtbar
F2.2	Kreditkartenzahlung loggen : Die ...	versteckt



# Requirements

**Problem verstehen, bedeutet Anforderungen erkennen, analysieren und bewerten**

- Systemfunktionalität erfassen (das System macht „X“)
  - Systemeigenschaften erfassen (das System soll „y“ sein)
  - Systemgrenzen definieren
- 
- High-Level-Use-Cases aufstellen,
  - Use-Cases bewerten sehr wichtig, wichtig, weniger wichtig
  - Use-Case-Diagramm aufstellen und Beziehungen beachten
  - Zentrale Use-Cases ausarbeiten

**Jede Systemfunktion sollte einem Use-Case zugeordnet werden können**



# Systemeigenschaften erfassen

(das System soll „y“ sein)

Systemeigenschaften sind Randbedingungen und Charakteristika, die das System erfüllen muss. Oft in Form von quantitativen Einschränkungen, Echtzeitbedingungen, Verfügbarkeitsanforderungen usw.

**Typische Anforderungen sind:**

„easy of use“, Fehlertoleranz, Antwortzeit, Plattformen, ...

**Diese Anforderungen stehen meist direkt in Bezug zu den Systemfunktionen.**



# Das Glossar wächst

## Basisanforderungen

<u>Ref.#</u>	<u>Eigenschaft</u>		<u>Kategorie</u>
A1.1	Antwortzeit: Die ...	< 5 Sek. ...	want
A1.2	Fehlertoleranz: Die ...	98% bei 7/24 ...	must

## Anforderungen an Interfaces

<u>Ref.#</u>	<u>Funktion</u>		<u>Kategorie</u>
A2.1	GUI ... Dialogboxen ...		want
A2.2	HGCI-Schnittstelle zum Hyper-Gateway-Customer-Service ...		must



# Beziehungen werden festgehalten

## Basisanforderungen

<u>Ref.#</u>	<u>Eigenschaft</u>	<u>Kategorie</u>	<u>Rel.</u>
A1.1	Antwortzeit: Die ... < 5 Sek. ...	want	F1.1, F2.1, ...
A1.2	Fehlertoleranz: Die ... 98% bei 7/24 ...	must	F6.4, ...

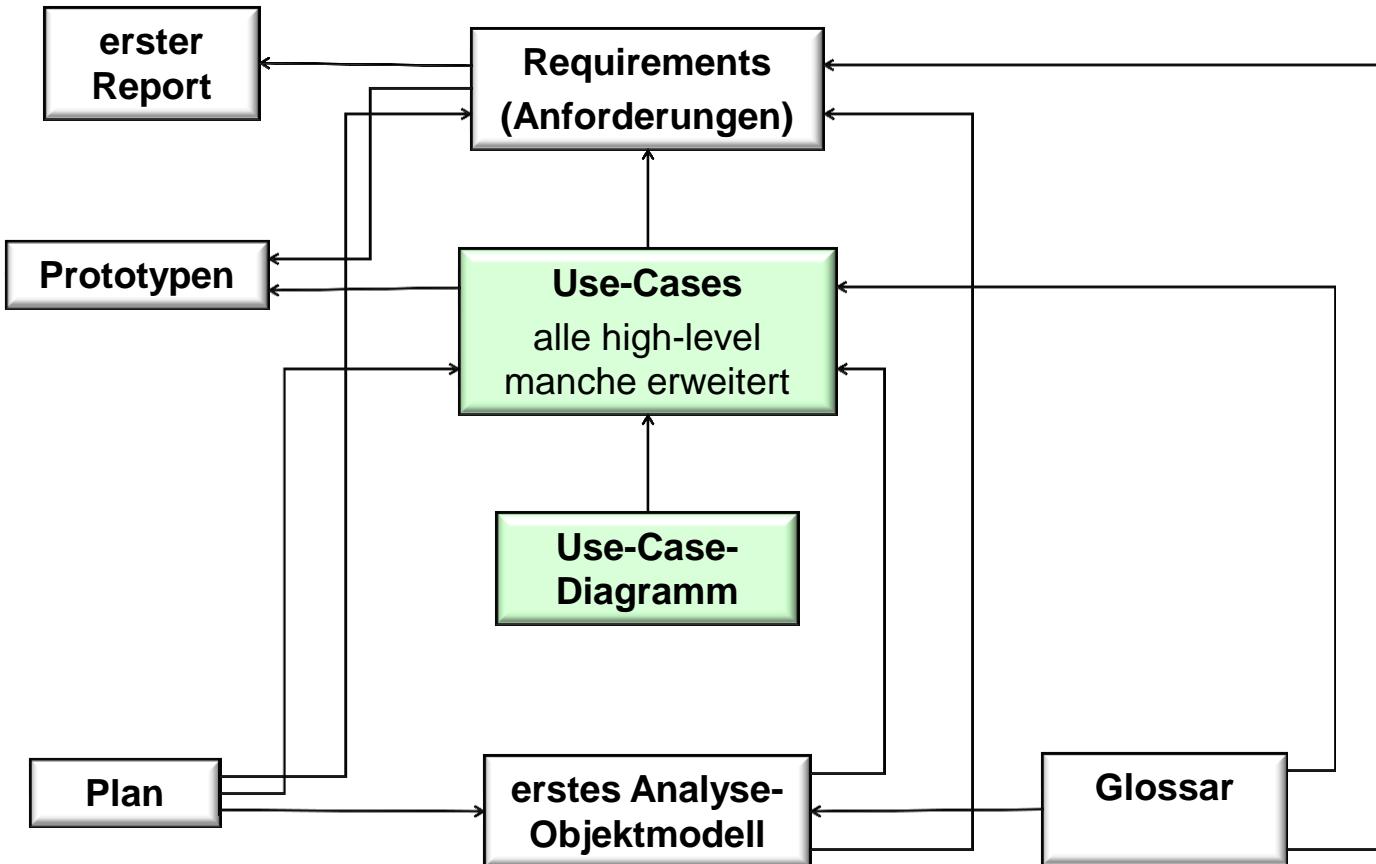
## Bezahlfunktionen

<u>Ref.#</u>	<u>Funktion</u>	<u>Kategorie</u>	<u>Rel.</u>
F2.1	Barzahlung abwickeln: Die ...	sichtbar	A1.1,A1.2
F2.2	Kreditkartenzahlung loggen : Die ...	versteckt	A2.2,...



# Requirements analysieren

Die Analyse der Anforderungen erfolgt über das Aufstellen von Use-Cases.





# Use-Cases

**Ein Use-Case ist im Allgemeinen eine sehr grobe Beschreibung eines Ablaufs aus Benutzersicht. Diese Beschreibung enthält viele Schritte und Aktionen.**

**Use-Cases müssen priorisiert werden. Dies erlaubt es, sie im Rahmen des Entwicklungsprozesses, geeignet einzuplanen.**

**Es gilt folgendes Prinzip:**

**Zuerst werden Use-Cases realisiert, die die zentrale Architektur beeinflussen, (Applikationsschicht und hohe Service-Schichten) oder als besonders kritisch eingestuft werden müssen.**



# Use-Cases

## Aufbau eines High-Level-Use-Case:

- Name: „Verb ...“
- Akteure:
- Typ: Prio 1, 2 oder 3
- Beschreibung: 3-4 Sätze
- Referenz: F1.1, ...
- Vorbedingung

Oft ist es notwendig, einen höheren Detaillierungsgrad zu verwenden, um ein tieferes Problemverständnis zu erreichen. Dies geschieht über „erweiterte Use-Cases“; diese enthalten zusätzlich eine detaillierte Ablaufbeschreibung z. B.:

- **textuell**
- **in Form eines Activity-Diagramms**
- **in Form eines Zustandsdiagramms**



# Beispiel Registrierkasse

**Ein High-Level-Use-Case, der den Vorgang des Kaufens und Bezahlens von Waren an einem Kassenterminal beschreibt.**

- **Name:** Waren kaufen
- **Boundary:** Kassensystem
- **Akteure:** Kunde, Kassierer
- **Prio:** 1 (hoch)
- **Beschreibung:** Der Kunde bringt die Waren zur Kasse, dort werden sie vom Kassierer aufgenommen. Der Kassierer wickelt den Bezahlvorgang ab und übergibt dem Kunden die Waren nachdem dieser bezahlt hat. Der Kunde verlässt das Geschäft mit der bezahlten Ware.
- **Referenz:** F1.1, F 1.2, F 2.1, ...
- **Vorbedingung:** Kassierer ist angemeldet.

**High-Level-Use-Cases erzeugen schnell ein Problemverständnis.**



# Oft gemachte Fehler

**Ein Use-Case ist im Allgemeinen keine einzelne Aktion oder ein einzelner Schritt.**

**Keine Use-Cases sind:**

***Waren aufnehmen und deren Quantität bestimmen und eingeben.***

***Den Gesamtpreis ermitteln und auf dem Display anzeigen.***

**Use-Cases sind sehr grobe Beschreibungen**



# Wie findet man Use-Cases

- **Über die Akteure**
  - Man identifiziert die Akteure, die zu einem System oder zu einer Organisation in Verbindung stehen.
  - Für jeden Akteur bestimmt man die Prozesse, an denen er beteiligt ist oder die er initiiert.
- **Über die Ereignisse**
  - Man bestimmt die externen Ereignisse auf die ein System reagieren muss.
  - Die Ereignisse werden dann zu Akteuren und Use-Cases in Beziehung gesetzt.

**(Wie reagieren System und Akteure auf die Ereignisse)**



# Anwendungsfalldiagramme der UML

## (Use Case Diagram)

Anwendungsfälle sind Interaktionen zwischen Benutzern und Systemen. Ein Anwendungsfall beschreibt die Funktionalität eines Systems aus Benzersicht. Er beschreibt damit die Systemanforderungen, wie sie sich aus der Problembeschreibung ergeben.

### Ein Anwendungsfall beschreibt:

- die Funktionalität, die vom System erbracht werden soll
- die Abgrenzung des System von der Umgebung
- die Entwicklersicht auf die Anforderungen des Anwenders



# Anwendungsfalldiagramme

## Elemente

- **das System**

begrenzt das zu erstellende System von der Umgebung

- **die Akteure**

Personen (Rollen) oder Systeme, die mit dem System interagieren

- Akteur (Kunden, Mechaniker)
- Dialoge
- Systeme (Interfaces zu Systemen)

- **die Use-Cases**

einzelne Anwendungsfälle, die „sichtbare“ Funktionalität

- **die Abhängigkeiten**

Abhängigkeiten zwischen Anwendungsfällen, zwischen Akteuren und  
Abhängigkeiten zwischen Akteuren und Anwendungsfällen



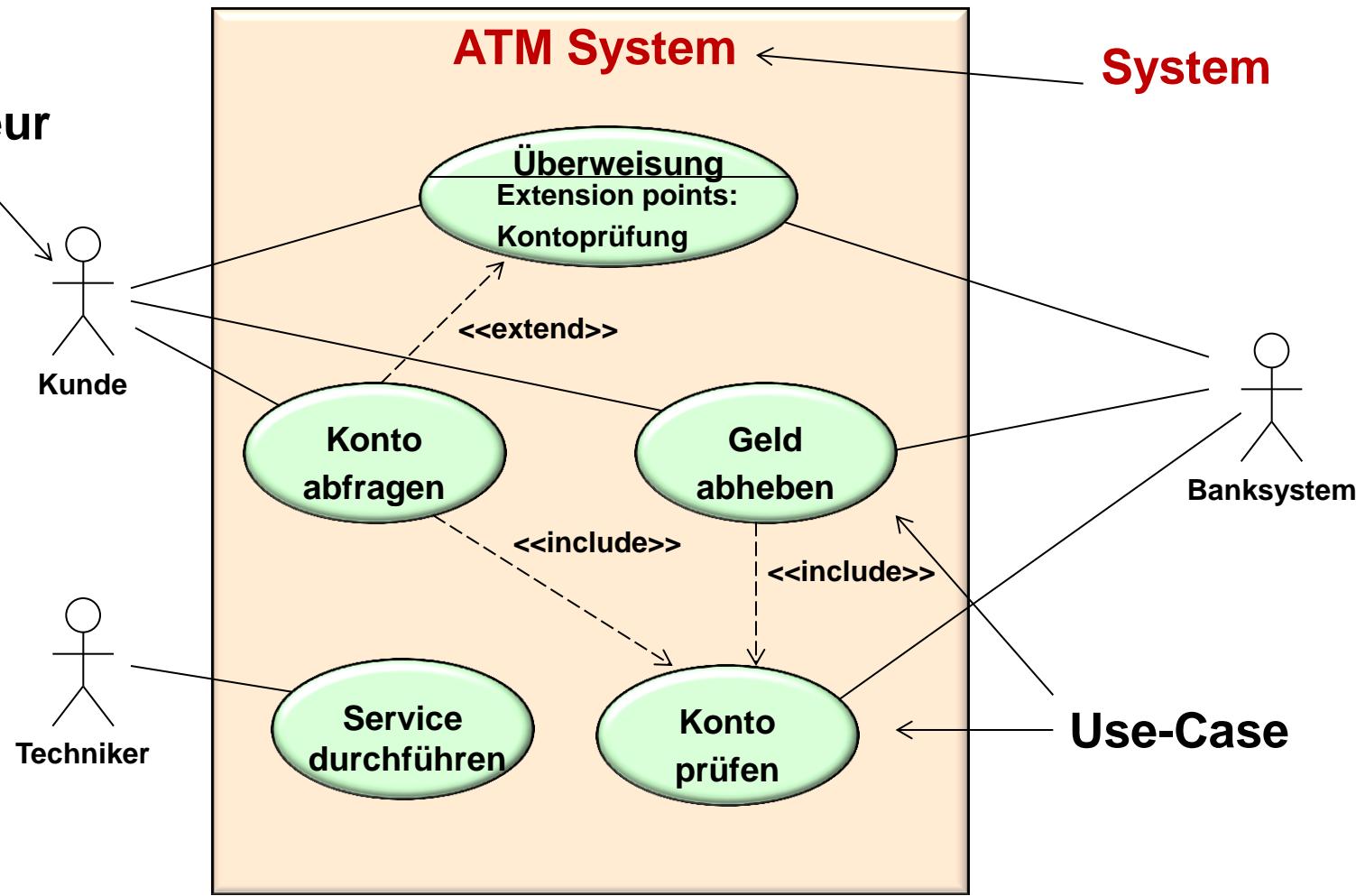
# Anwendungsfalldiagramme

Akteur

System

Banksystem

Use-Case





# Akteure

Akteure führen Anwendungsfälle durch. Ein Akteur kann an vielen Anwendungsfällen beteiligt sein. An einem Anwendungsfall können viele Akteure beteiligt sein; im Allgemeinen jedoch mindestens einer.

**Ein Anwendungsfall ohne Akteure ist meist kein Anwendungsfall.**



# Beziehungen bei Akteuren

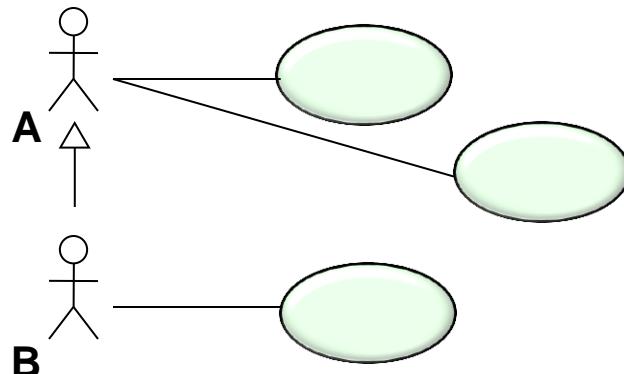
- **Assoziation (Akteur <-> Use-Case)**

Teilnahme eines Akteurs an einem Use-Case. Eine Instanz eines Akteurs und eines Use-Case kommunizieren miteinander.



- **Generalisierung**

Die eine Instanz des Akteurs „B“ kann mit allen Use-Cases kommunizieren mit denen auch „A“ kommunizieren kann (plus die, an denen „B“ beteiligt ist)

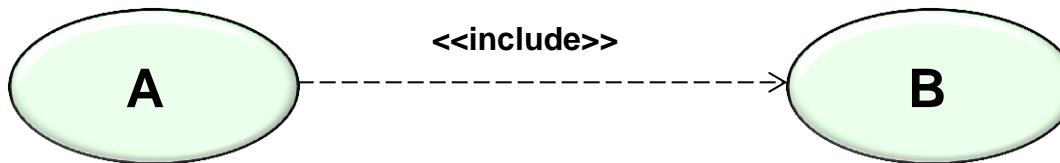




# Beziehungen zwischen Use-Cases

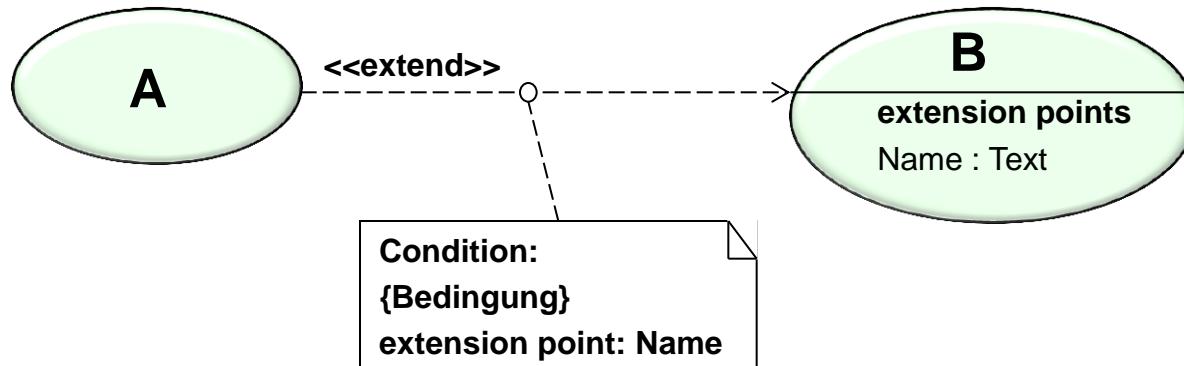
- **Beinhalten (<<include>>)**

Die Aktionen des Use-Case „B“ werden im Use-Case „A“ eingebunden (als Ganzes)



- **Erweitern (<<extend>>)**

Die Aktionen des Use-Case „A“ kommen im Use-Case „B“ hinzu, falls die entsprechende Bedingung (des Extension Points) erfüllt ist

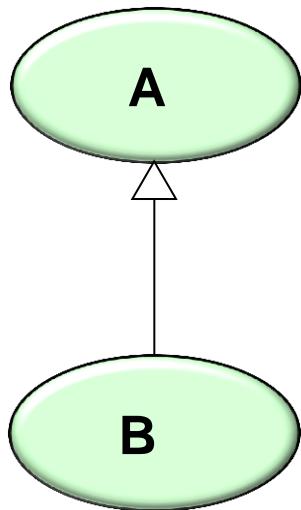




# Beziehungen zwischen Use-Cases

- **Generalisierung**

Die Aktionen des Use-Case „B“ stellen einen Spezialfall des Use-Case „A“ dar.



Wird benutzt, wenn der Spezialfall „B“ so eigenständig ist, dass er durch einen eigenen Use-Case ausgedrückt werden soll und nicht adäquat durch eine *extend* oder *include* Beziehung modelliert werden kann.

**B erbt die Beziehungen von A**



# Faustregel:

- **include**

Man verwendet **include**, wenn man sich in mehreren Anwendungsfällen wiederholt und dies vermeiden möchte.

- **extend**

Man verwendet **extend**, wenn man eine Variation eines normalen Verhaltens in einer kontrollierten Form beschreiben möchte, wobei die Erweiterungsstellen im Basis-Use-Case gekennzeichnet werden können.

- **Generalisierung / Spezialisierung**

Man verwendet **Verallgemeinerung / Spezialisierung**, wenn man eine Variante eines normalen Verhaltens beiläufig beschreiben möchte.



# Szenarien beschreiben Use-Cases

**Ein Szenario ist eine Folge von Schritten, die die Interaktion zwischen einem Benutzer und einem System in einer besonderen Situation beschreibt.**

**Bsp.:** der erfolgreiche Kauf eines Produkts  
der Kauf eines Produkts mit Kreditkarte  
das Scheitern aufgrund falscher PIN-Eingabe

**Ein Anwendungsfall ist eine Menge von Szenarien, die durch ein gemeinsames Ziel verbunden sind.**

**Man analysiert Anwendungsfälle durch zentrale Szenarien mittels Interaktionsdiagrammen, Zustandsdiagrammen und Aktivitätsdiagrammen (auch textuell).**



# Use-Cases während der Analyse

- **Die frühen Phasen**

- Nachdem die Systemfunktionen erfasst wurden, werden die Systemgrenzen und die Akteure identifiziert.
- Alle zentralen Use-Cases werden erfasst und priorisiert.
- Use-Case-Diagramme aufstellen und Beziehungen erfassen.
- **Die kritischsten und essentiellsten Use-Cases werden ausgearbeitet.  
(Sie definieren das Problem)**

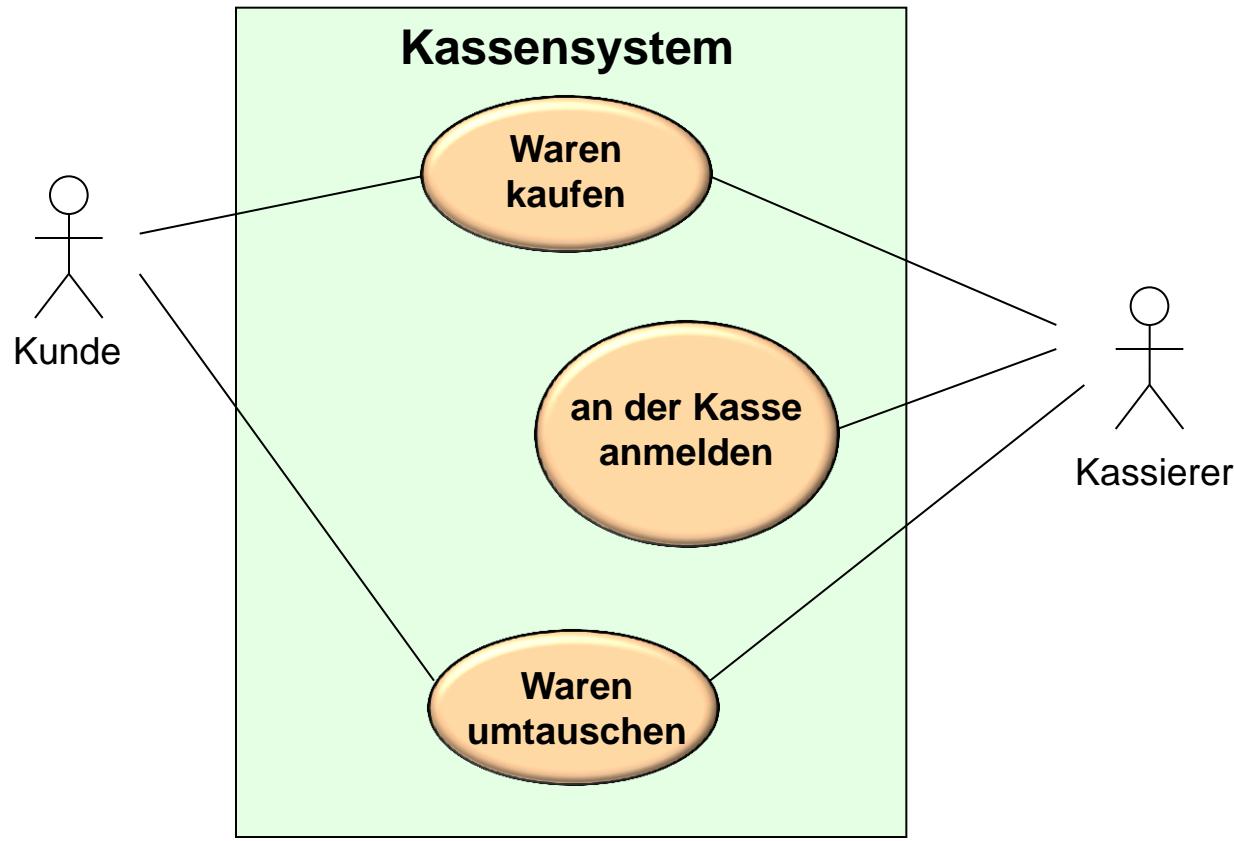
- **In späteren Phasen**

- **Die Use-Cases ausarbeiten, die in dieser Entwicklungsphase realisiert werden sollen.**



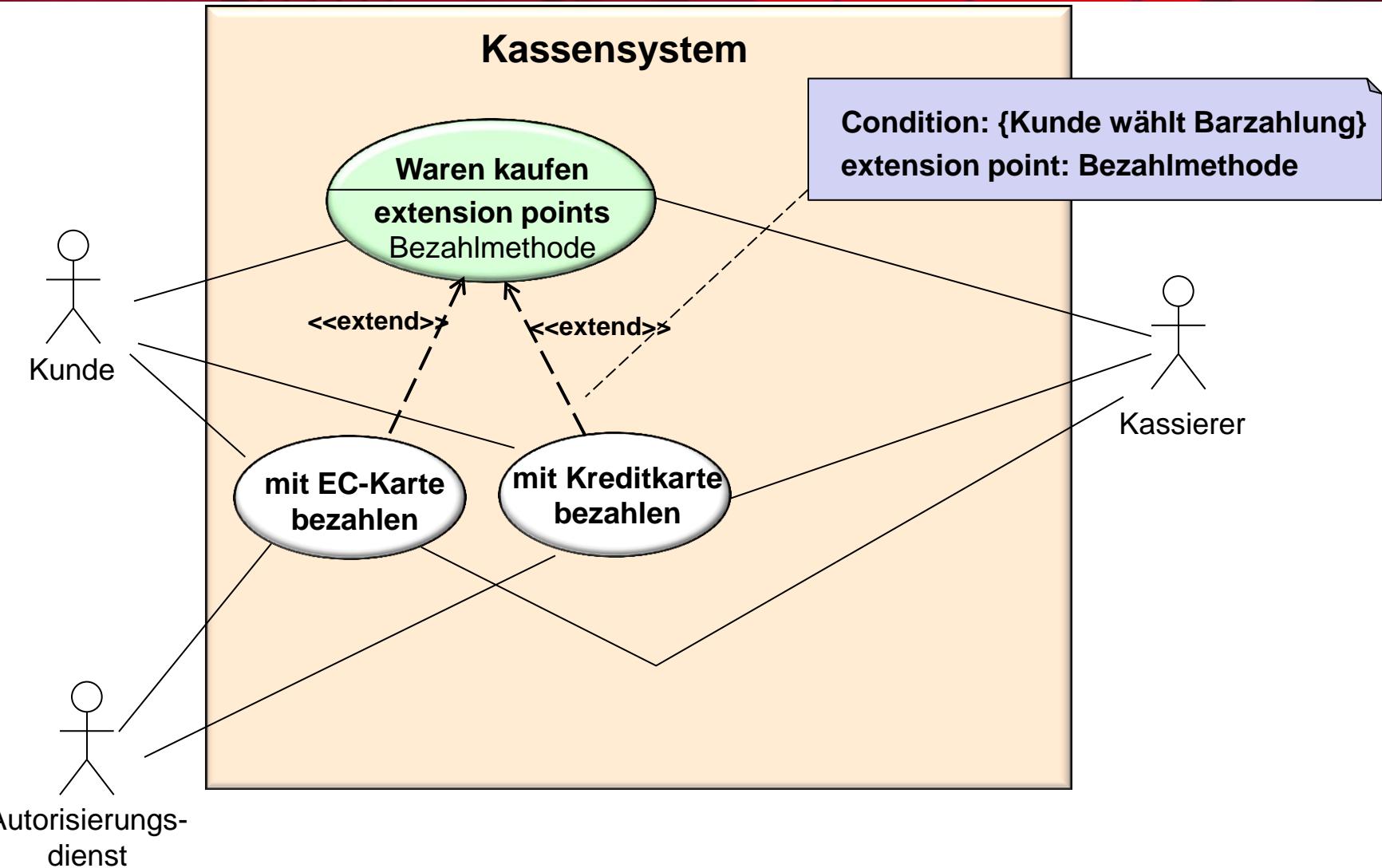
# Beispiel      Registrierkasse

## einige typische Use-Cases



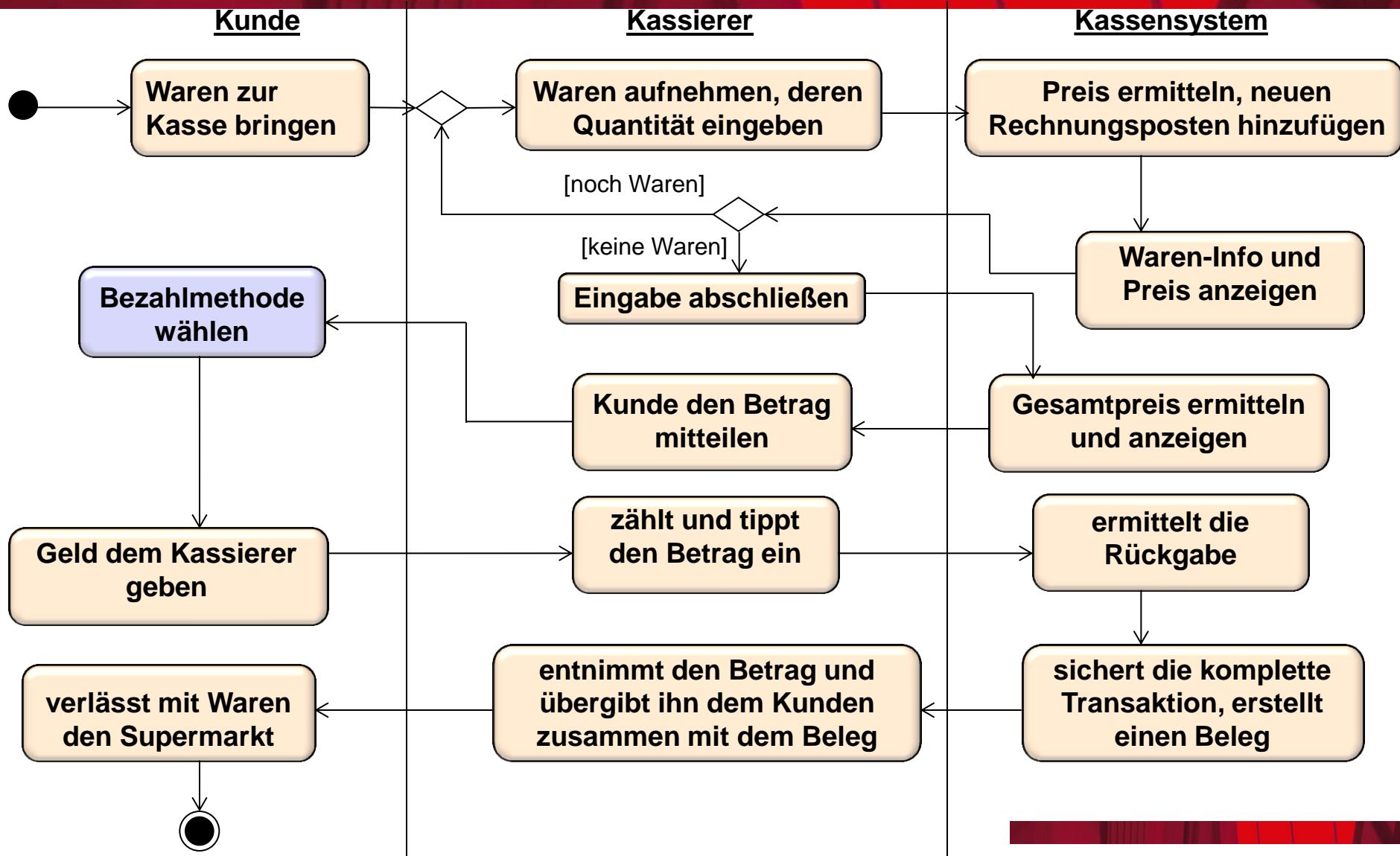


# Registrierkasse (spätere Phase)



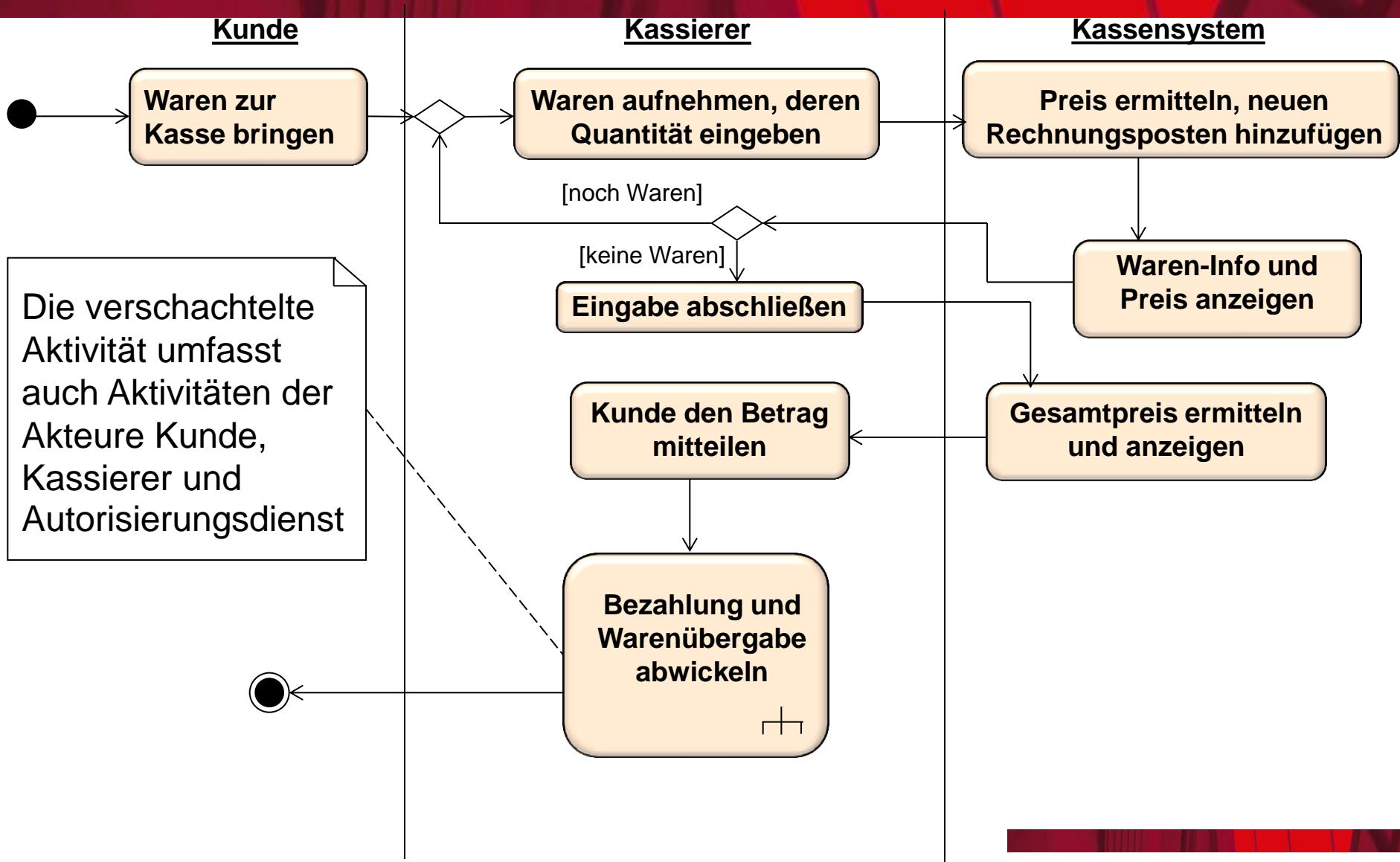


# Beispiel Registrierkasse II



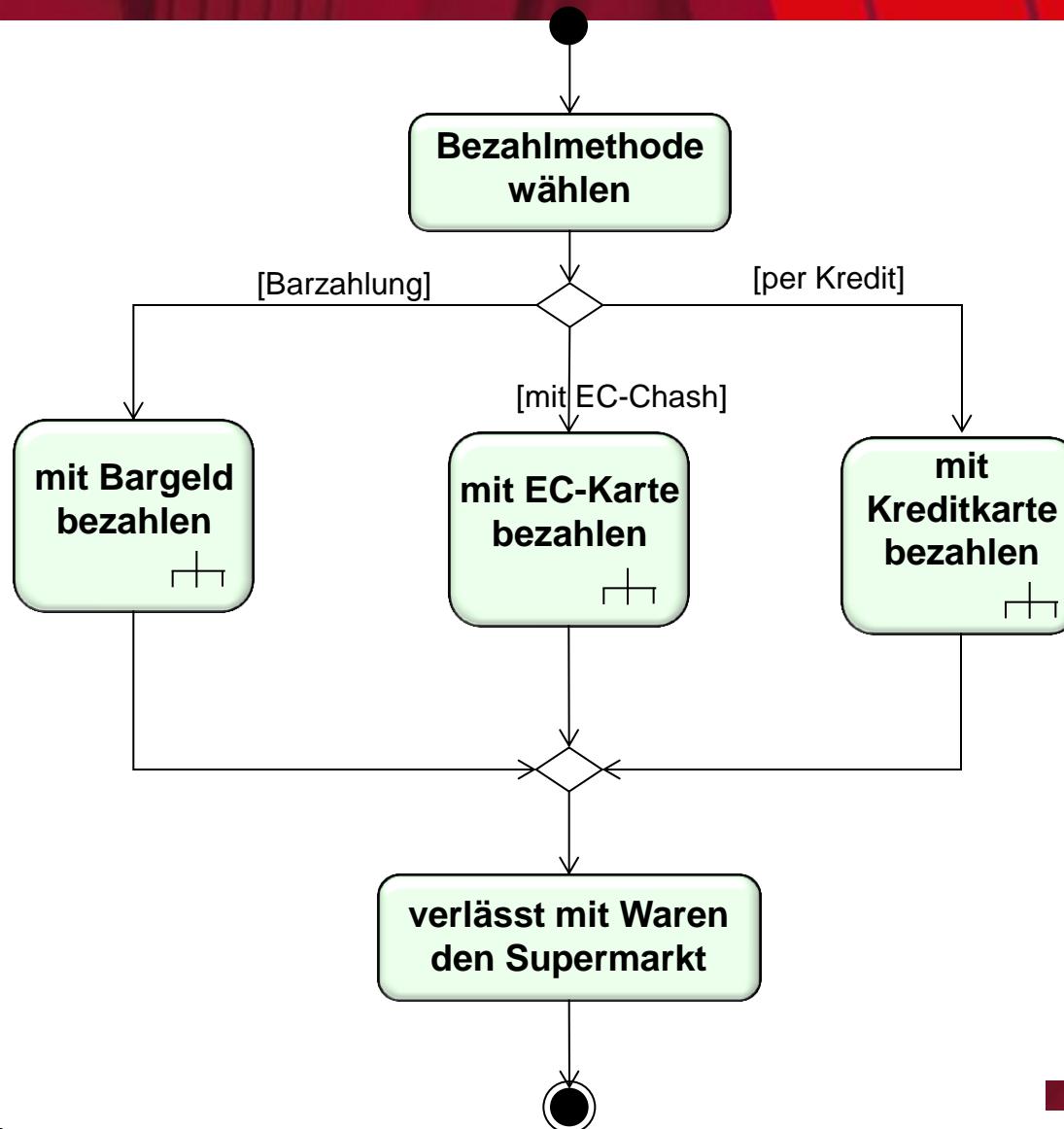


# Beispiel Registrierkasse III





# Beispiel Registrierkasse IV





# Aktivitätsdiagramme (UML)

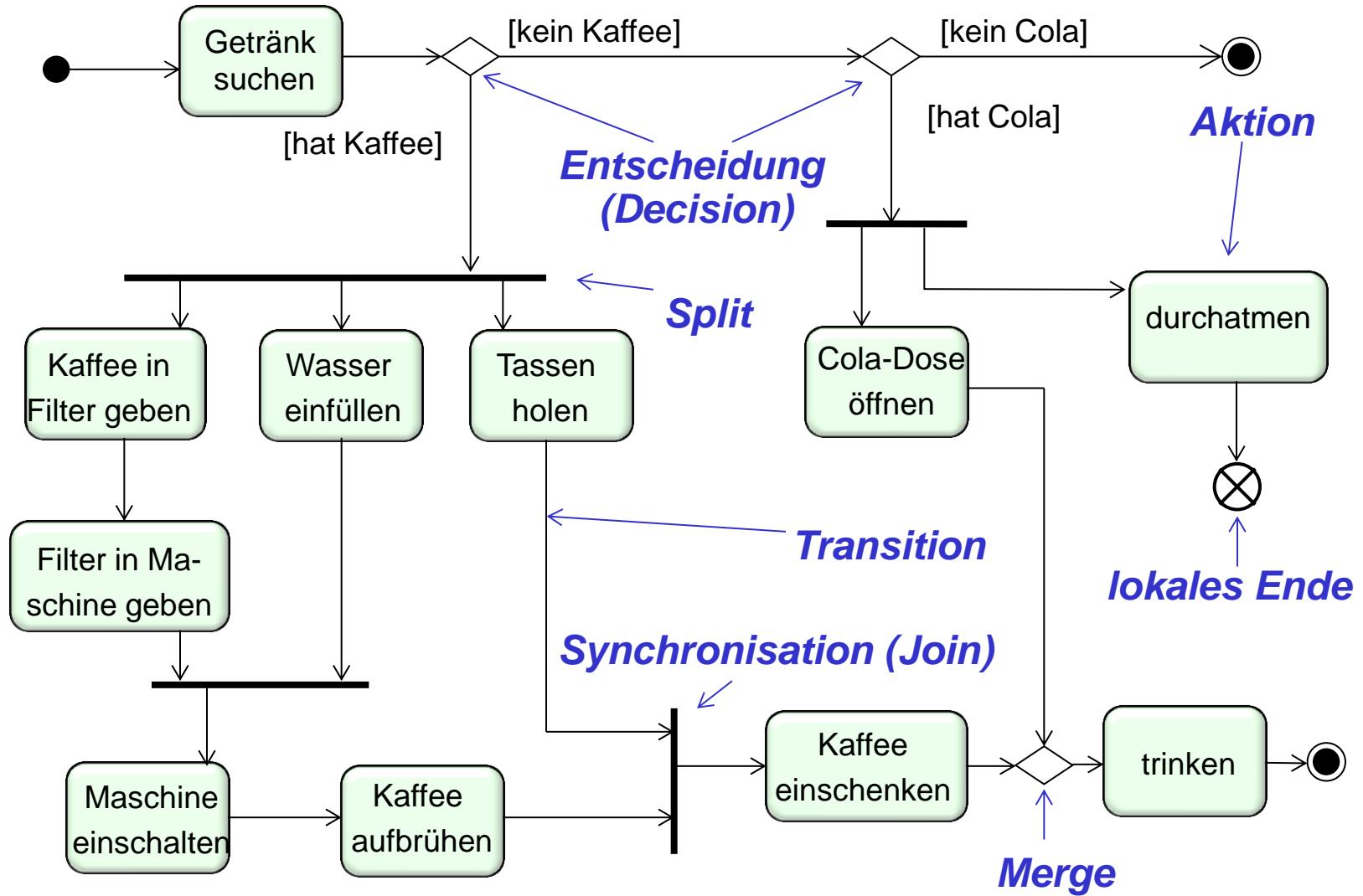
Ein Aktivitätsdiagramm ist ein Diagramm von zusammenhängenden Aktionen beliebiger Granularität. Ein Übergang (Transition) ist nicht verbunden mit einem Ereignis, sondern mit dem Abschluss der entsprechenden Aktion.

Aktivitätsdiagramme sind geeignet unterschiedlichste Abläufe darzustellen. Sie können etwa fachliche Zusammenhänge und Abläufe eines Anwendungsfalls vollständig beschreiben. Sie können Sachverhalte aus mehreren Anwendungsfällen übergreifend beschreiben. Sie können Geschäftsregeln und Entscheidungslogiken definieren, komplexe Algorithmen, ...

Sie können Klassen, Operationen, Anwendungsfälle, ... beschreiben



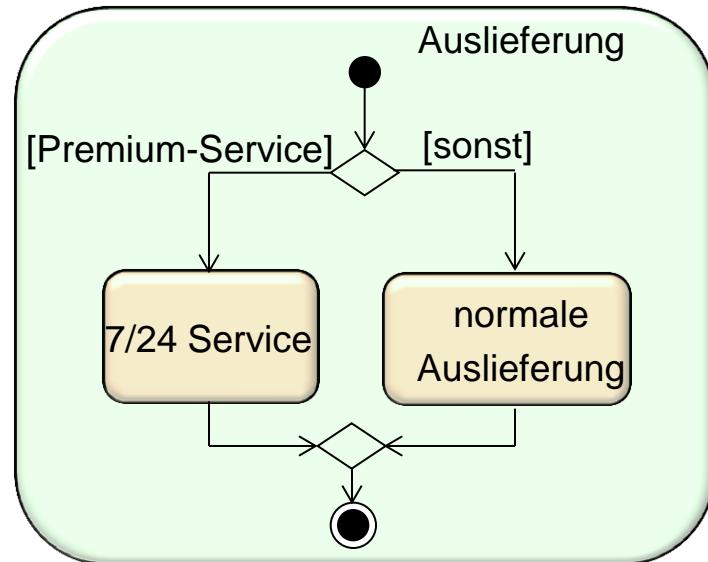
# Aktivitätsdiagramme



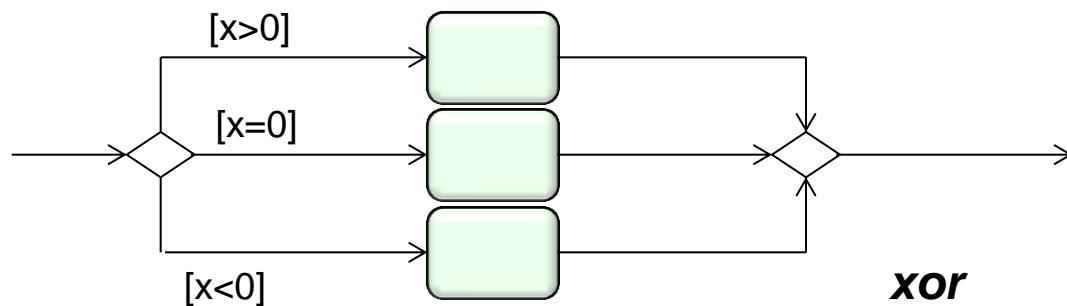


# Besondere Aktionen

- **Geschachtelte Aktionen**



- **Entscheidung und Merge**

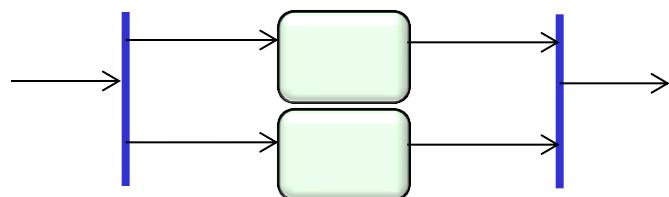


*xor*



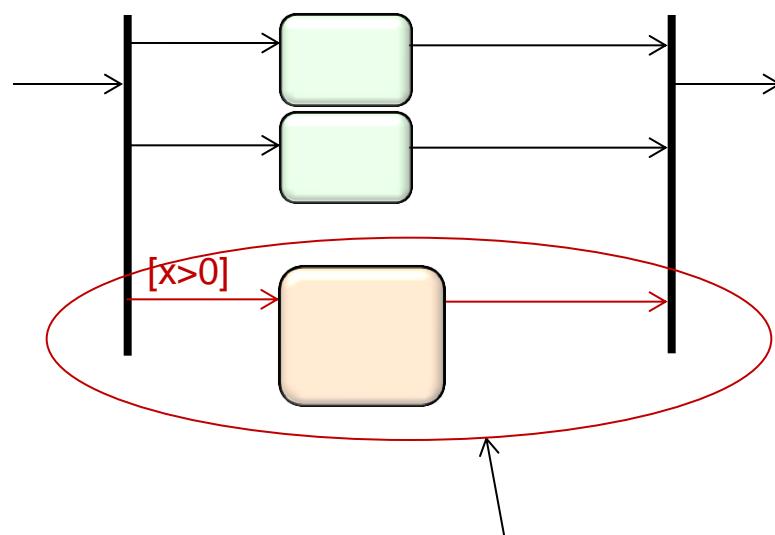
# Besondere Aktionen

- **Split und Join**



*split*

*and*

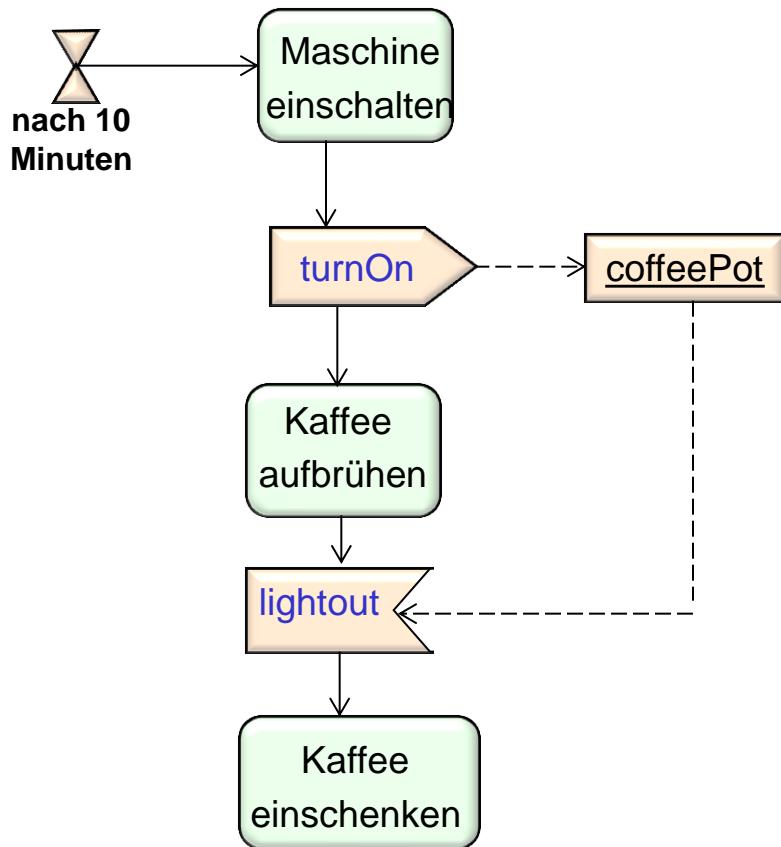


**bedingter Thread**



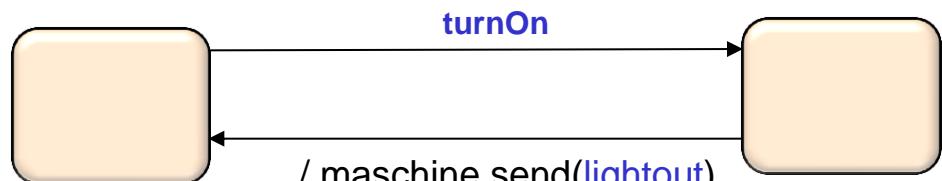
# Besondere Aktionen

- Signale senden und empfangen



entspricht einer markierten  
Transition im  
Zustandsübergangsdiagramm

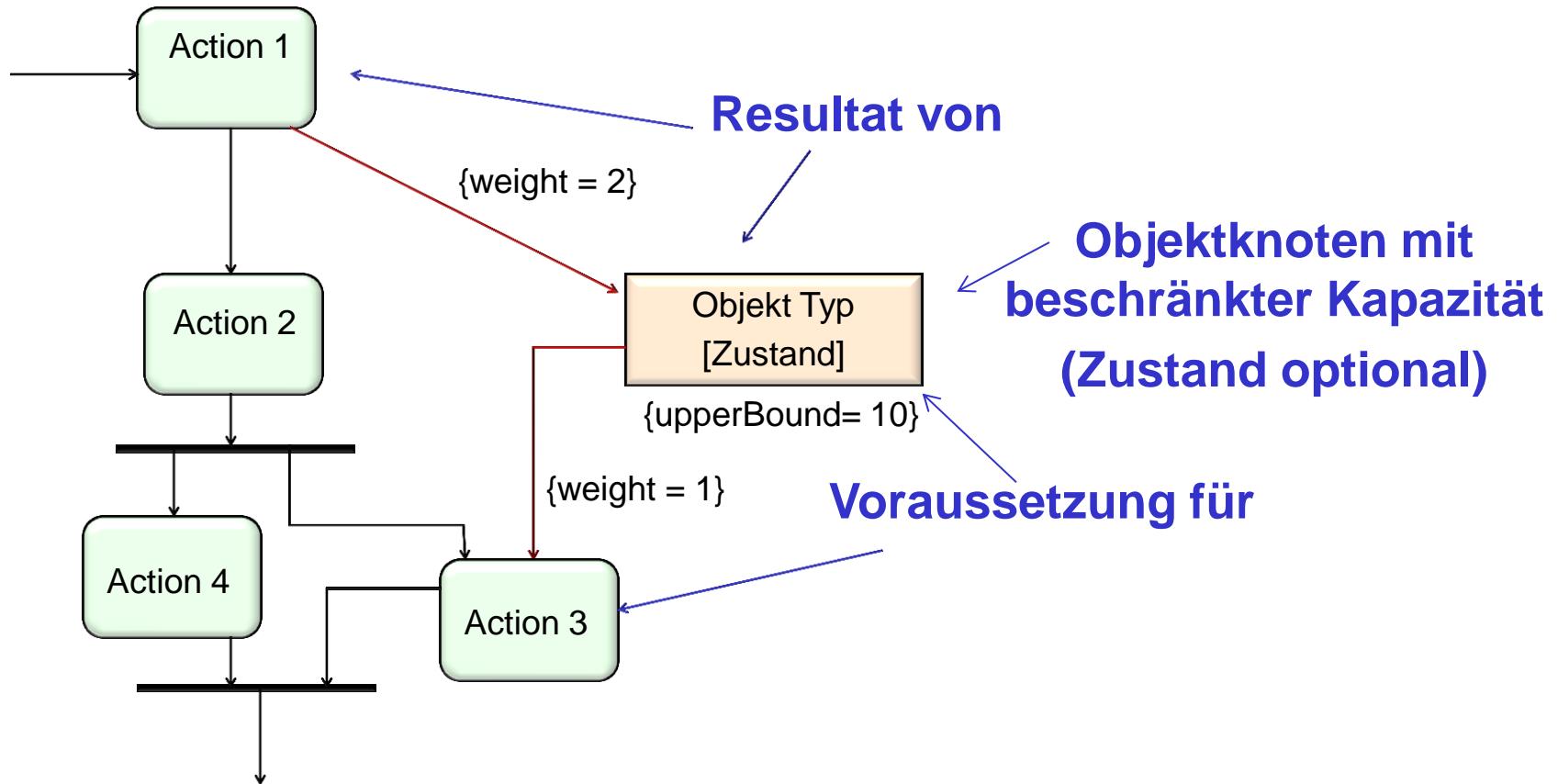
*event [guard-condition] / action-expression*





# Besondere Aktionen

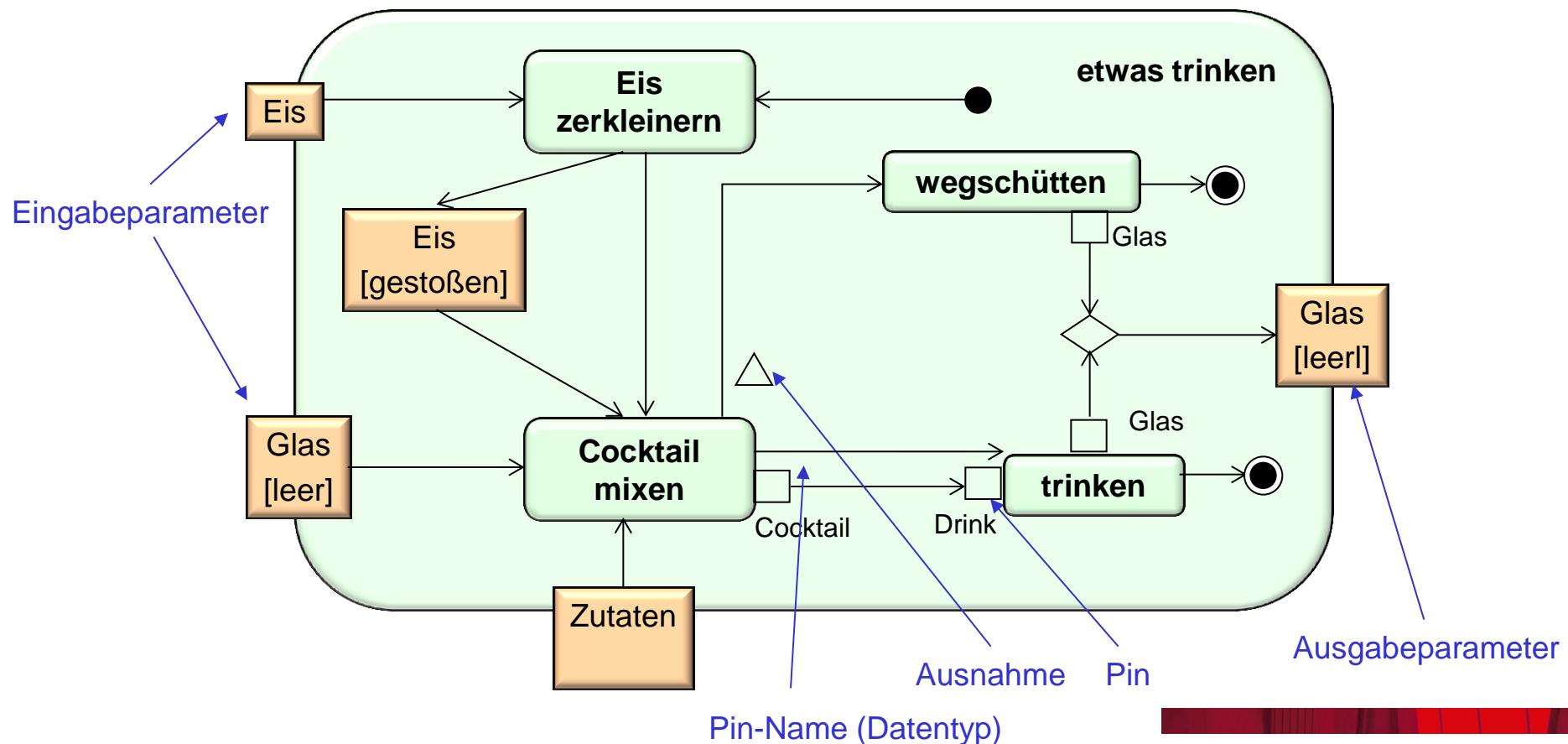
- Objektknoten dienen der Modellierung des Datenflusses





# Besondere Aktionen

- Pins verdeutlichen den Objektfluss und passen (verträgliche) Parameter an (ausgefüllte Pins beschreiben kontinuierliche Datenströme)



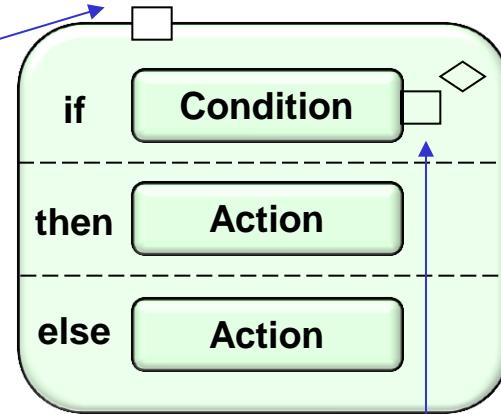


# Besondere Aktionen

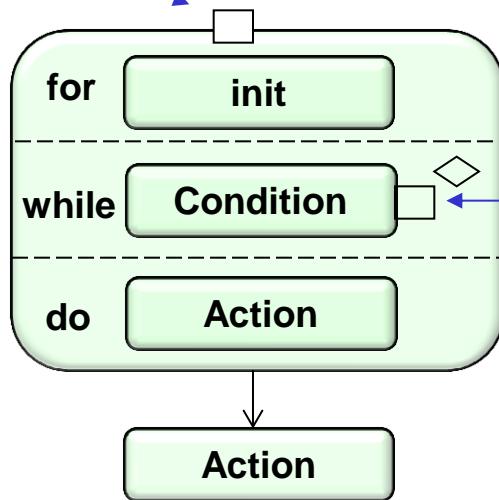
- **Programmelemente**

Parameter nach Bedarf

- **If-Then-Else**

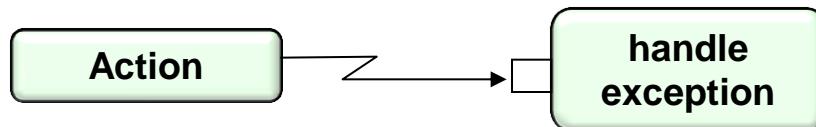


- **Schleife**



Eventuell ein bool'scher Ausgabeparameter

- **Exception**





# Swimlanes

Aktivitäten können in **Swimlanes** (Verantwortlichkeitsbereiche) aufgeteilt werden. Verantwortlichkeitsbereiche korrespondieren oft mit organisatorischen Einheiten in Businessmodellen oder Akteuren und Systemen in Use-Case-Modellen, einzelnen Objekten o.ä.

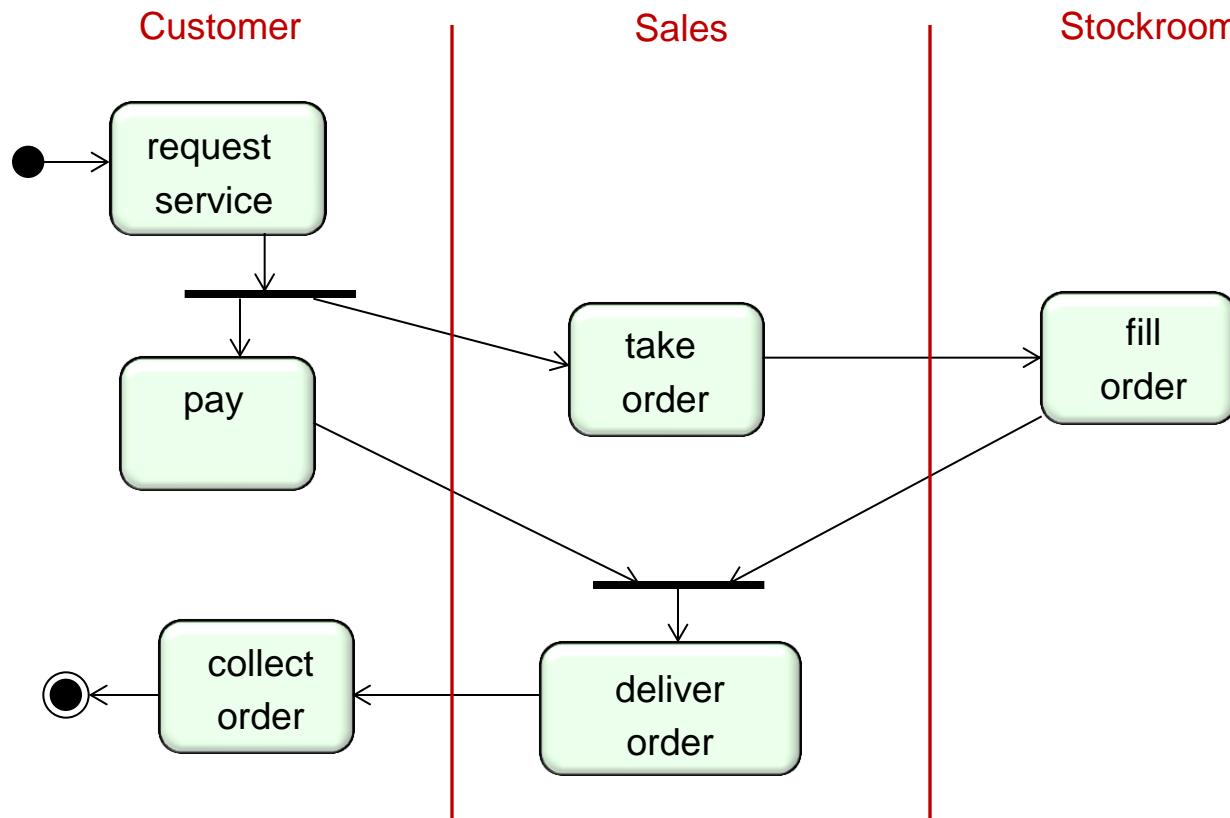


Diagramm mit 3  
Swimlanes



# Erstellung des Objektmodells

- Objekte identifizieren
- Data-Dictionary aufbauen
- Assoziationen identifizieren
- wesentliche Attribute ergänzen

**Als Grundlage dienen die Use-Cases und ihre Beschreibungen.**

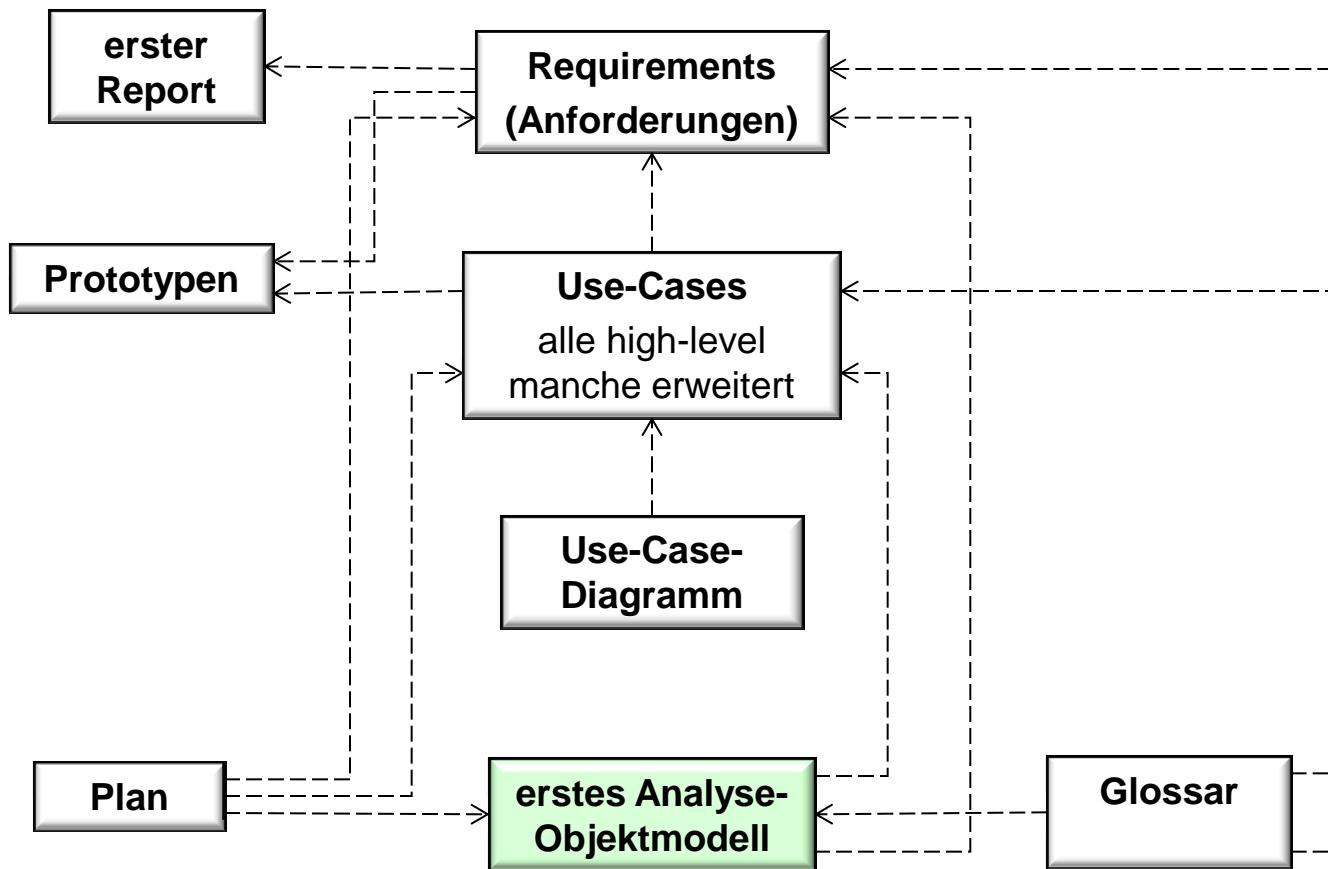
**Ziel:**

**Die Problemdomäne verstehen und die Beziehungen erarbeiten und aufzeigen.**

**Das Problem nicht die (Software-)Lösung steht im Vordergrund**



# Das (Analyse-)Objektmodell





# (Analyse-)Objektmodell

## Bisher:

- **Systemfunktionalität erfasst (das System macht „X“)**  
Rechnungssumme bestimmen, Transaktion speichern, Barzahlung abwickeln, Kreditkartenzahlung loggen, Bonität prüfen, ...
- **Systemeigenschaften erfasst (das System soll „y“ sein)**  
„easy of use“, Fehlertoleranz, Antwortzeit < 5 Sek, Plattformen Win9, ...
- **Systemgrenzen definiert (Kassensystem)**
- **High-Level-Use-Cases aufgestellt**
- **Use-Cases und Funktionalitäten bewertet  
(sehr wichtig, wichtig, weniger wichtig)**
- **Use-Case-Diagramme aufgestellt und Beziehungen erarbeitet**
- **Zentrale Use-Cases ausgearbeitet**



# Analysemodell: Objekte finden

Die Objekte, die gesucht werden, sind Konzepte der Realität

## Strategien:

1. **Via Checkliste:** Objekte (Konzepte) finden, die Instanzen vorgegebener Kategorien sind.  
**Typische Kategorien sind:** Dinge, Namen, Orte, Rollen, Prozesse, Ereignisse, Organisationen, Regeln, Kataloge, ...
  
2. **Substantive suchen:** Substantive in der Problembeschreibung lokalisieren und als Kandidaten für Konzepte betrachten.

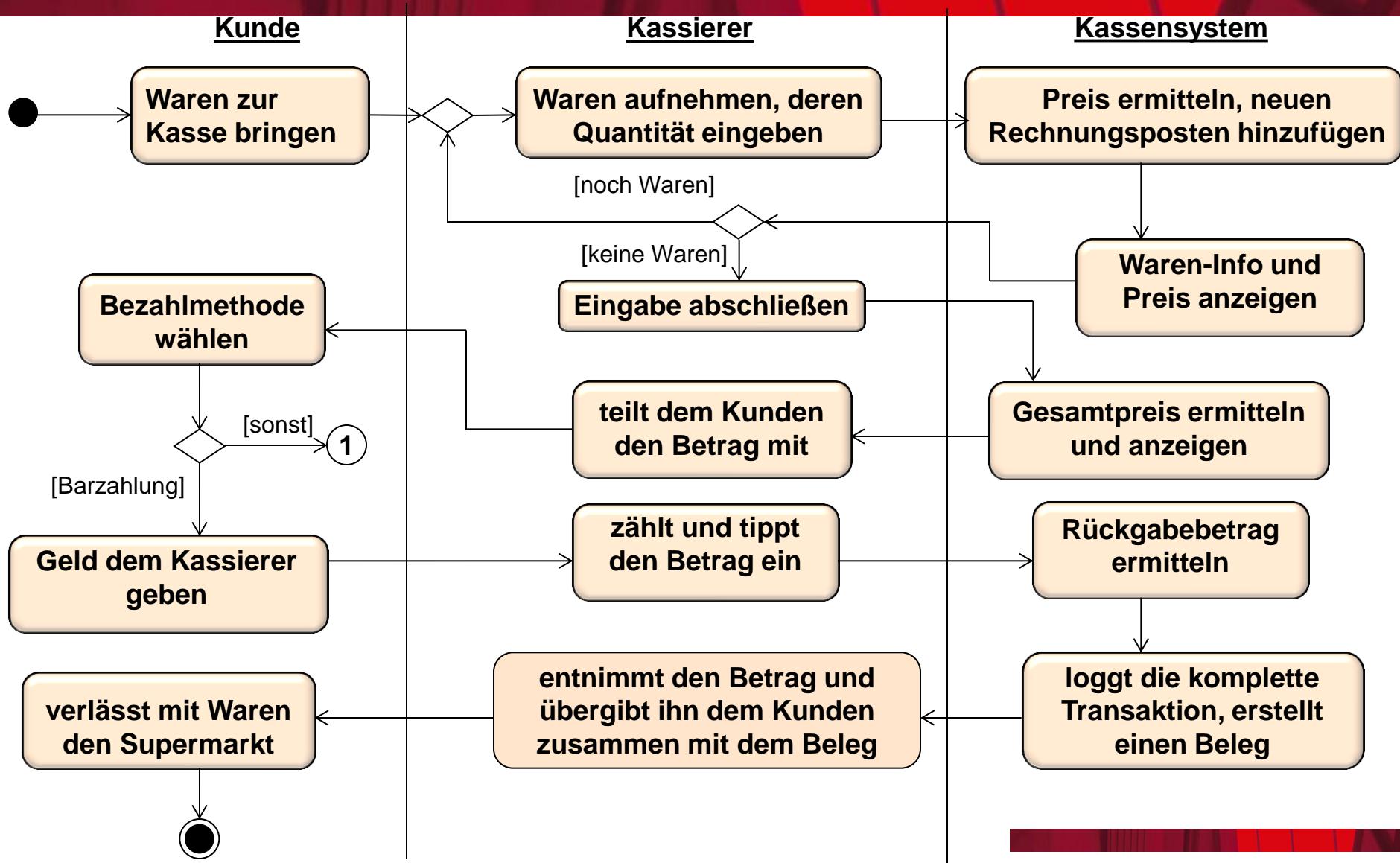
**Problem:** Nicht jedes Substantiv beschreibt ein Objekt (Konzept).

Besser, man kombiniert beide Varianten. D.h., die Objekte aus 2 über die Checkliste aus 1 als Konzepte identifizieren.

**(mehr Konzepte sind besser als zu wenige)**

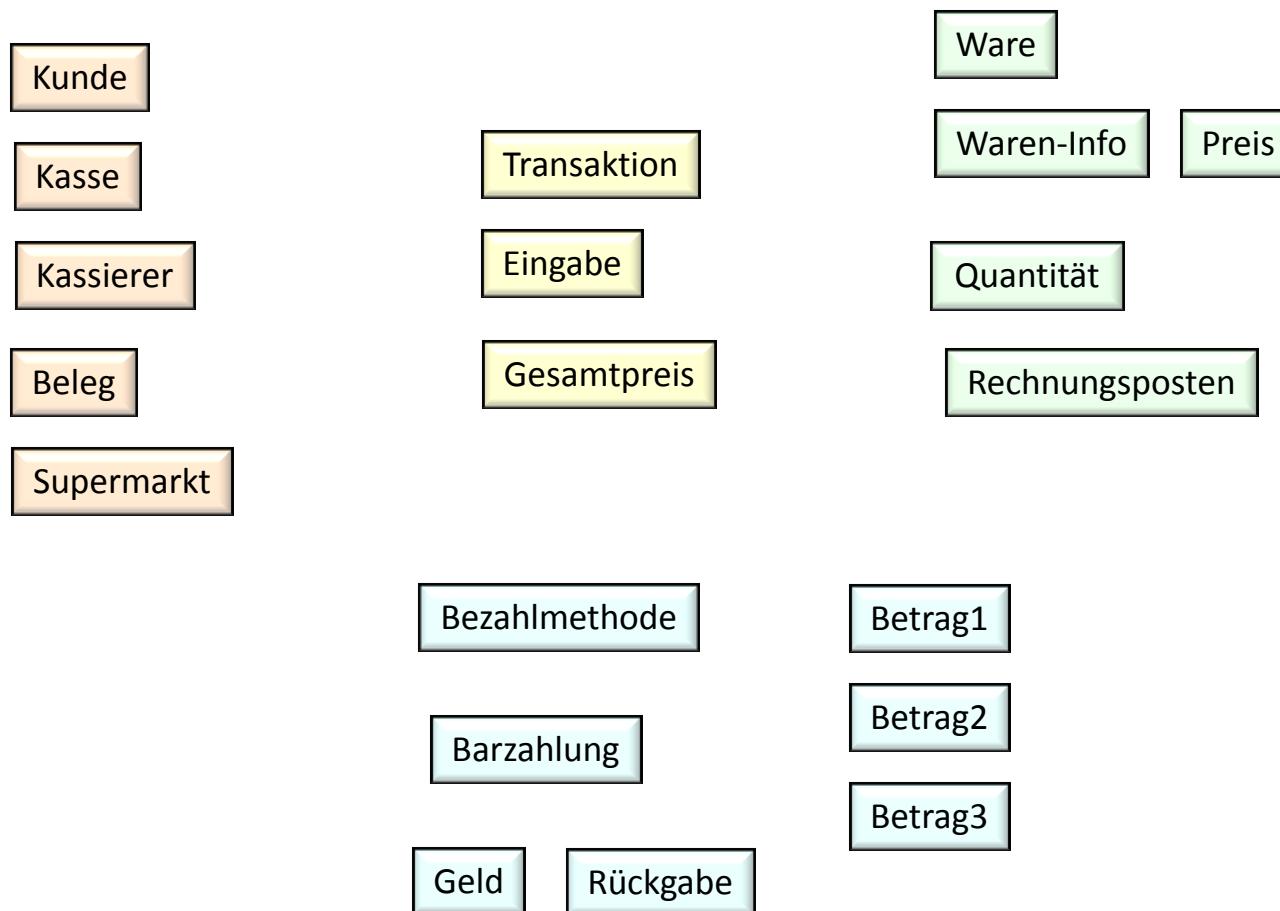


# Objekte finden: Registrierkasse



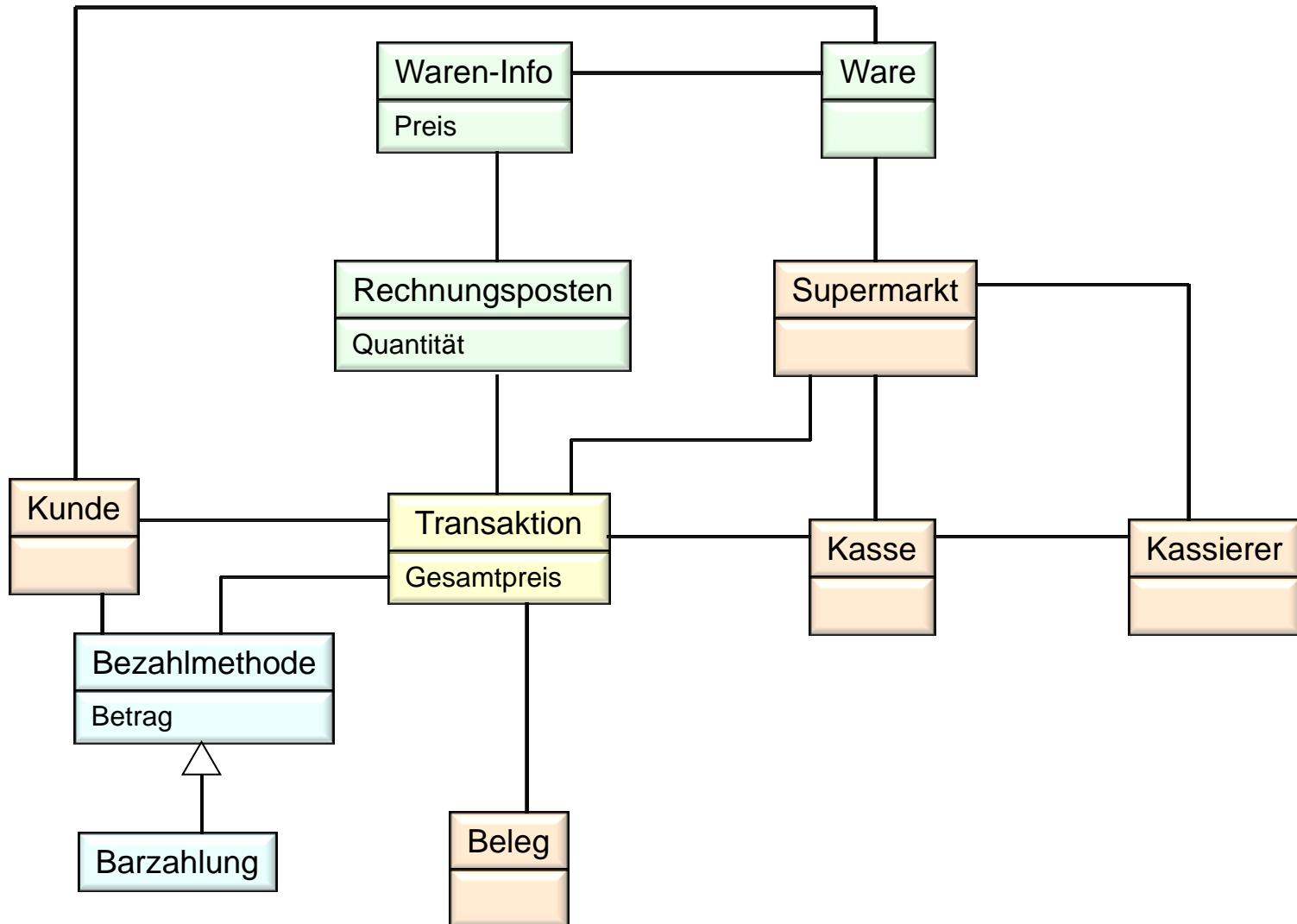


# Objekte finden: Registrierkasse



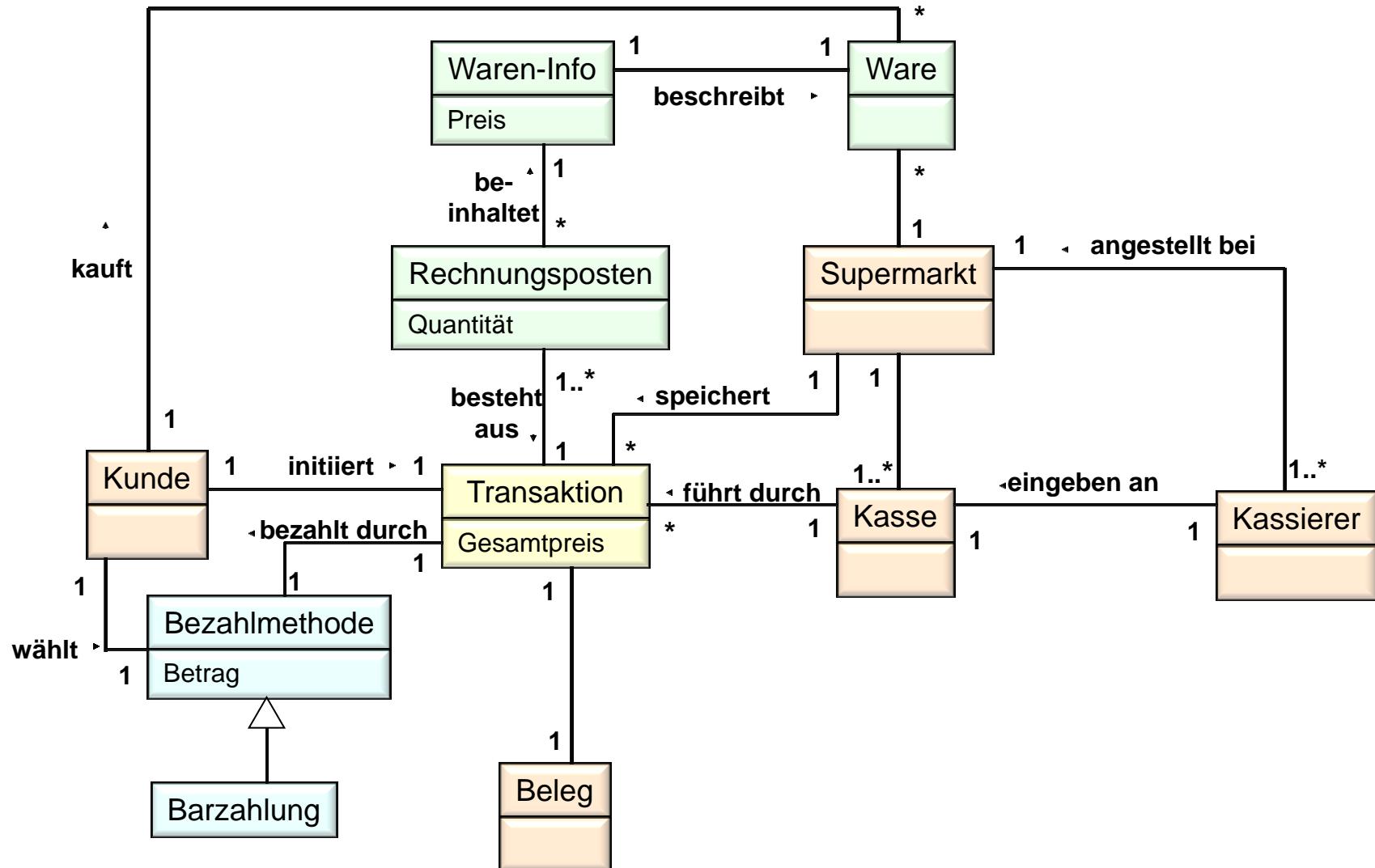


# Objekte finden: Registrierkasse





# Objekte finden: Registrierkasse





# Typische Fehler:

**Ein oft gemachter Fehler: Aus einem Konzept wird ein Attribut.**

**Faustregel:**

Dinge, die man anfassen kann, die keine Quantitäten, Texte oder Adjektive (z.B.: Farben) darstellen, sind keine Attribute.

**Beispiel:**

Flug
Flughafen

oder

Flug
Nr.:

Flughafen
Name

**Falls es unklar ist, dann wählt man immer die Alternative Objekt (Konzept) und nicht Attribut.**



# UML Klassen- und Instanzdiagramme

**Ein Klassendiagramm ist ein Graph von Classifier-Instanzen (Klassen, Datentypen, Interfaces) verbunden durch statische Beziehungen.**

**Ein Instanzdiagramm ist ein Graph von Instanzen (Objekte und Daten).**

**Eine Instanzdiagramm ist eine Instanz eines Klassendiagramms.**



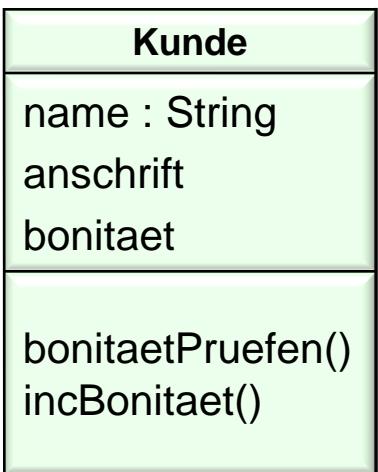
# Darstellung von Klassen

## Name Compartiment (Namensfeld):

beinhaltet den Klassennamen

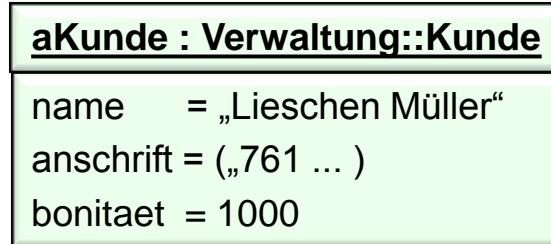
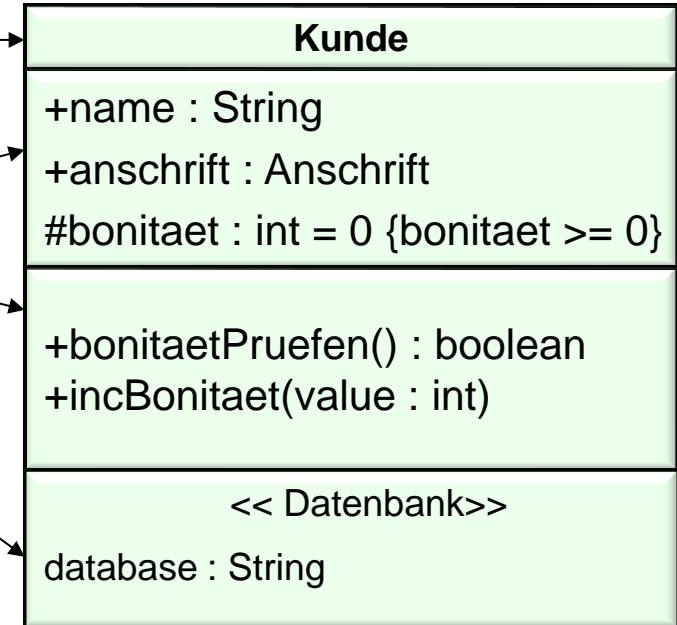
## List Compartments (Listenfeld)

für Attribute, Operationen  
und anderweitige Gruppierungen



**ohne Details**

**Analyseniveau**



**Designniveau**

**ein Objekt**



# Das Namensfeld

<<Stereotype>>



Klassenname

{Property}

- **Stereotype:** Definiert ein besonderes Modellelement (*Interface, Type, Controller,...*). Durch das Einführen eigener Stereotypen (abgeleitete Klassen existierender Klassen im Metamodell) kann die Semantik der UML erweitert und angepasst werden.
- **Property:** Charakterisiert das Beschreibungselement.
  - vordefiniert      `isAbstract = true, isLeaf = true`      (Attribut im Meta-Modell)
  - selbstdefiniert    `author = Th. Fuchß`                 (immer als Key-Value-Pair)



# Die Attribute

Attribute beschreiben die Struktur der Objekte,  
deren Bestandteile, Information, Daten, ...

```
+name : String [0..1]  
+/alter : int  
#bonitaet : int = 0 {bonitaet >= 0}
```

visibility / name : type [multiplicity] = initial-value {property}

- **visibility** + public, # protected, – private, ~ package, static
- **/** abgeleitetes Attribut, der Wert kann berechnet werden und muss nicht gespeichert werden
- **name** Identifiziert das Attribut ( *name* plus *package::class* eindeutig)
- **type** Klassenname oder Datentyp einer Programmiersprache
- **multiplicity** aus wie vielen Elementen besteht das Attribut
- **initial-value** initialer Wert bei einem neu angelegten Objekt
- **property** Constraints (Beschränkungen), die für das Attribut gelten
  - **unordered** die Elemente sind ungeordnet (Menge)
  - **ordered** die Elemente sind geordnet (Liste)



# Die Operationen

keyword  
(stereotype)

```
+bonitaetPruefen() : boolean {query}  
<>constructor>>  
Kunde (name : String, bonitaet : int)  
...
```

visibility name (parameter-list): return-type {property}

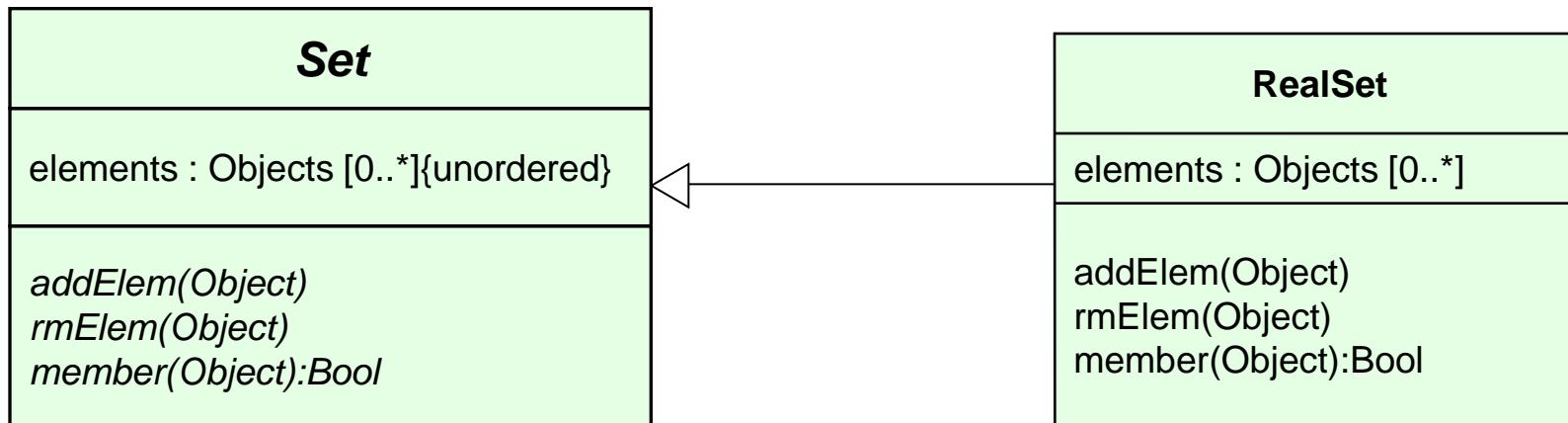
- **parameter-list**    kommaseparierte Liste formaler Parameter  
**kind name : type [multiplicity] = default-value {property}**
  - **kind**                in, out, inout
  - **default-value**    Default-Wert, falls der Parameter nicht übergeben wird
- **besondere Properties:**
  - **concurrency = ...**, **sequential**, **concurrent**, **guarded**
  - **abstract**            anstatt {abstract} kann auch die Operation kursiv dargestellt werden



# Abstrakte Klassen

**Eine abstrakte oder virtuelle Klasse ist eine Klasse, die keine realen Exemplare besitzt, sie ist bewusst unvollständig.**

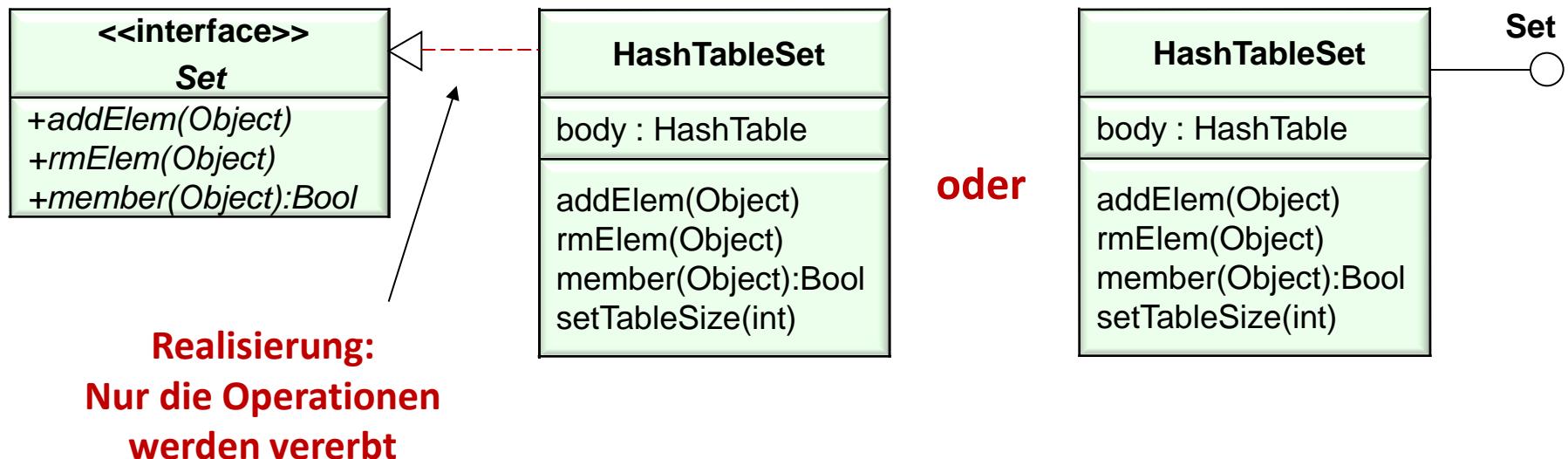
- Eine abstrakte Klasse ist immer Oberklasse.
- Eine abstrakte Klasse, die keine Unterklassen hat, ist sinnlos.





# Interfaces

Ein Interface beschreibt die externe Sichtbarkeit von Operationen einer Klasse, ohne die interne Struktur zu definieren. Üblicherweise spezifiziert ein Interface nur einen bestimmten Teil des Verhaltens einer Klasse.

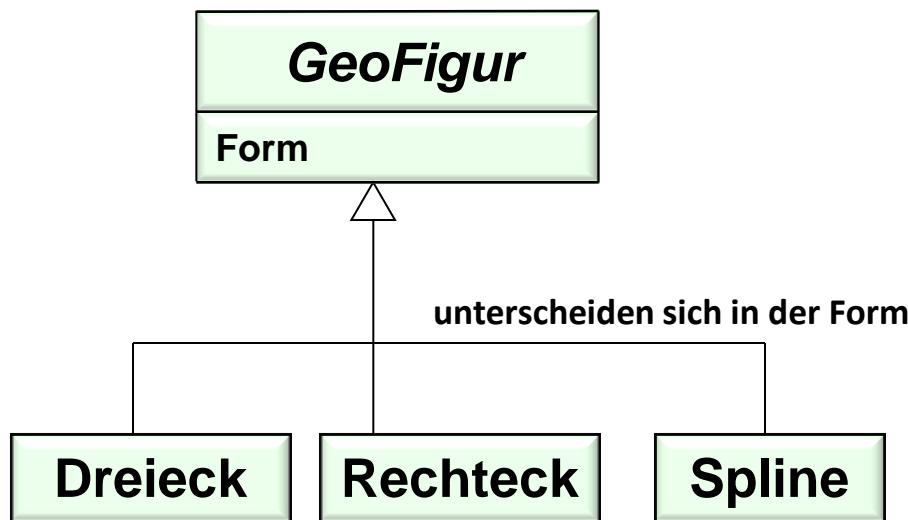


Interfaces sollten weder Attribute noch Assoziationen zu anderen Klassen haben.



# Angaben zum Generalisierungsgrund

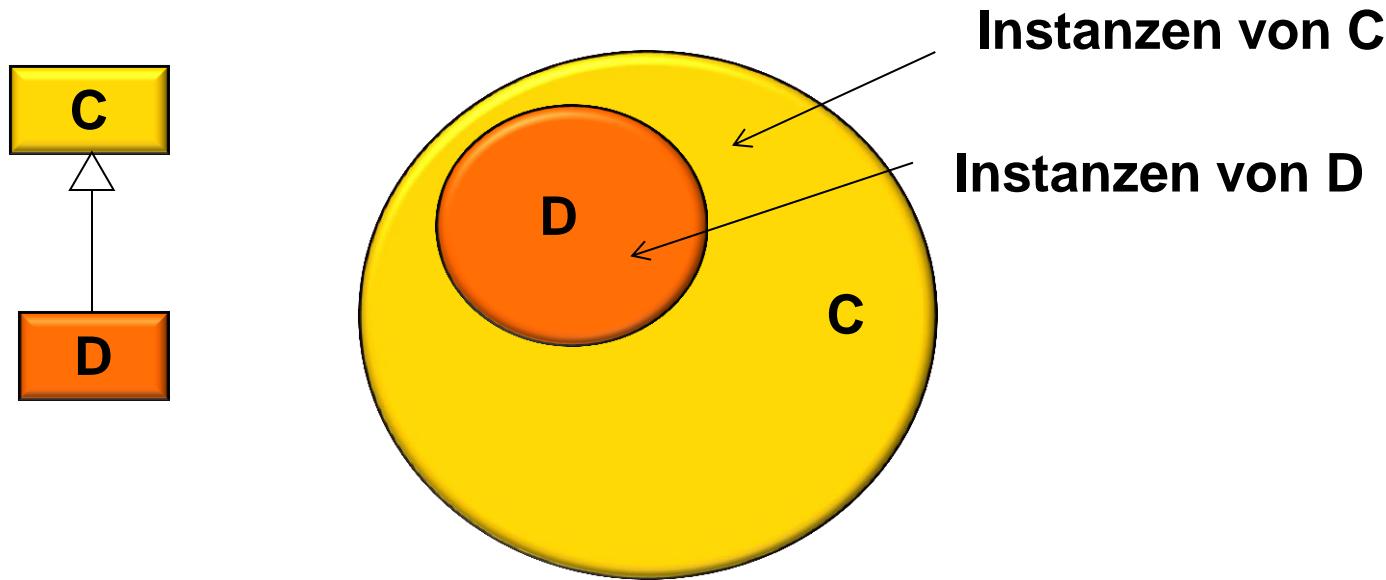
Oft gibt es abgeleitete Klassen, bei denen es unklar ist, in welchen Punkten sie sich unterscheiden. Hier hilft ein beschreibender Text am Generalisierungspfeil.





# Bedeutung

## Spezialisierungsvererbung (ist-ein-Vererbung)

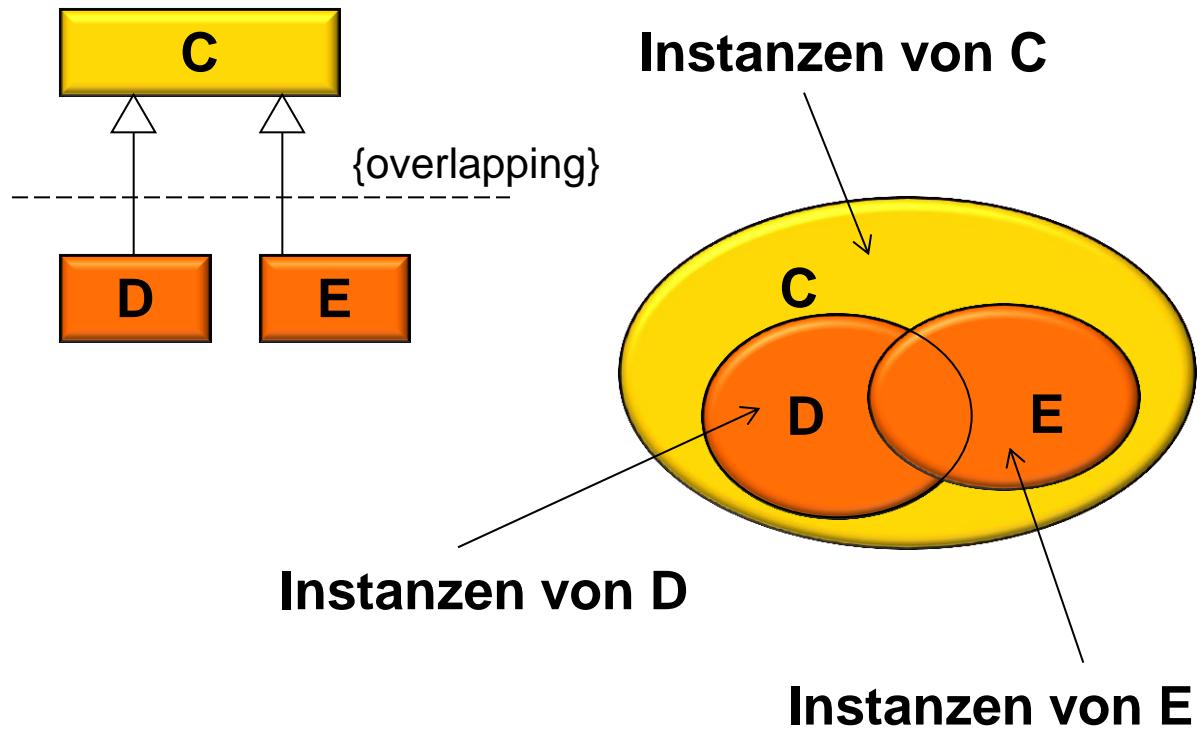


Die Instanzen der Unterklasse bilden eine Teilmenge der Instanzen der Oberklasse.



# Constraints

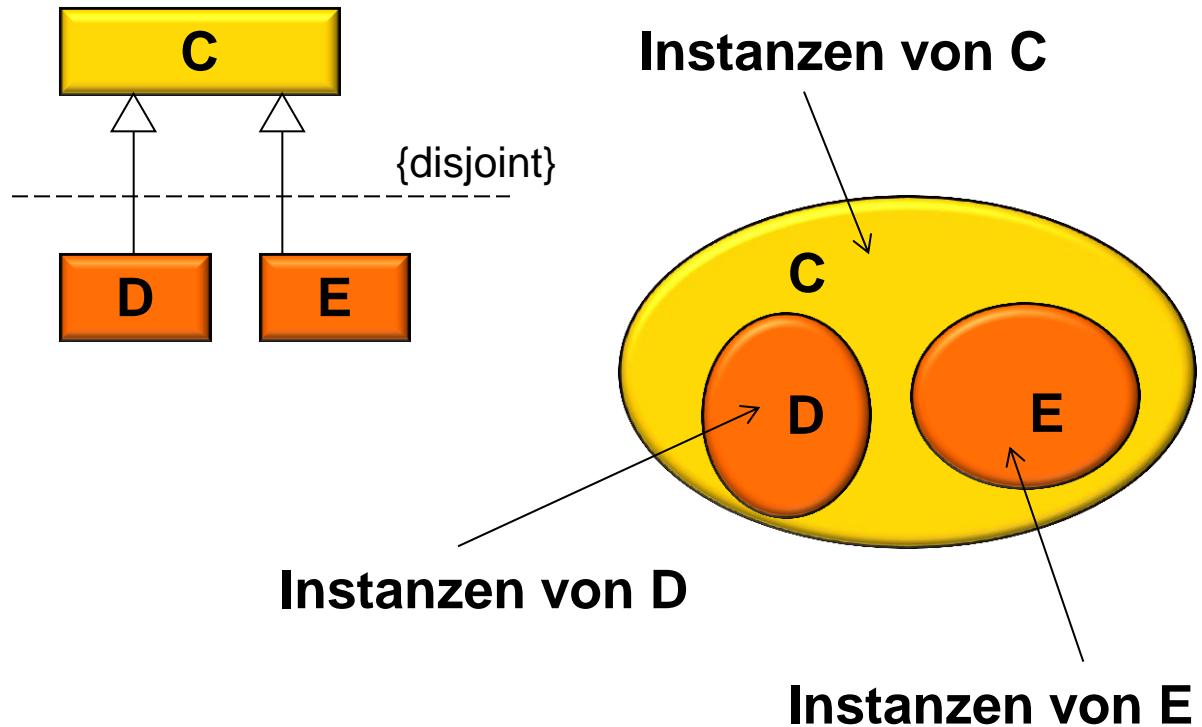
- overlapping





# Constraints

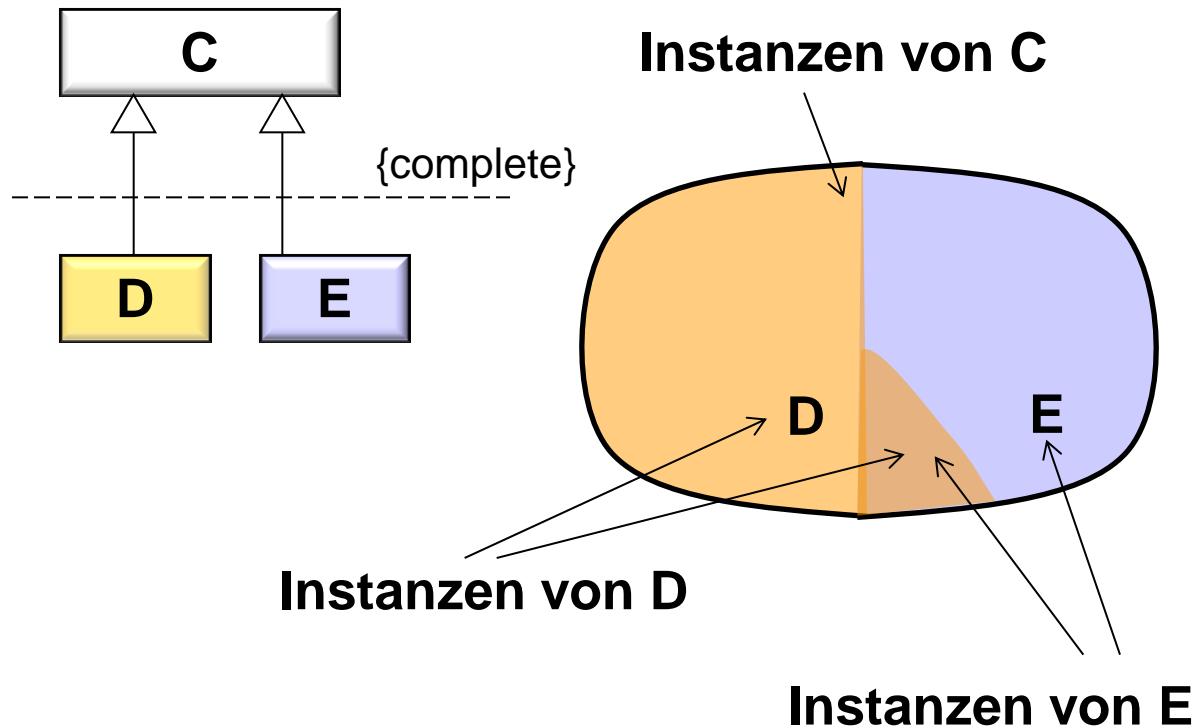
- disjoint





# Constraints

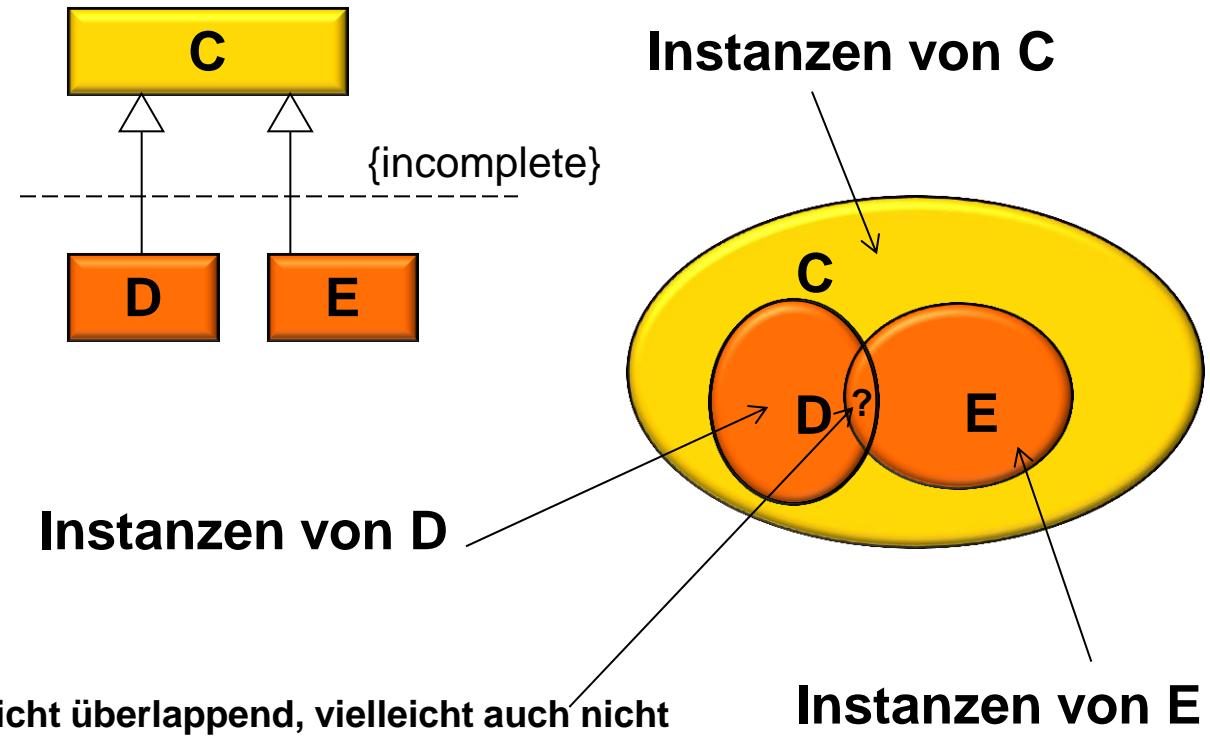
- complete





# Constraints

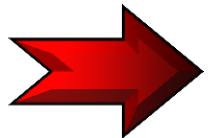
- incomplete





# Templates und Bindungen

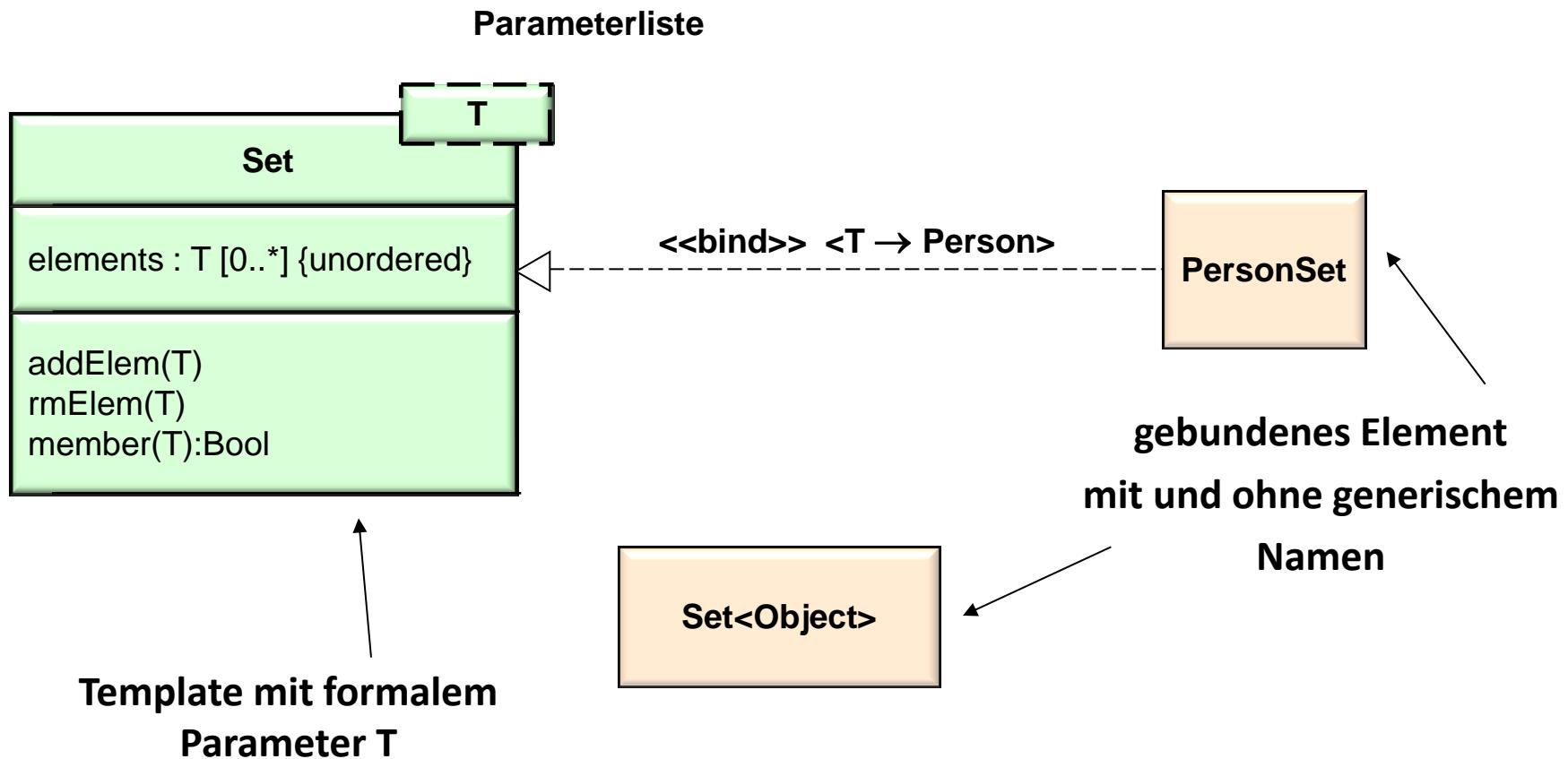
Template ist eine Bezeichnung für eine Klasse mit einem oder mehreren formalen Parametern. Damit verweist ein Template auf eine Familie von Klassen. Jede Klasse dieser Familie kann durch Substitution der Parameter bestimmt werden. Dies nennt man Bindung.



- Beziehungen (Assoziationen) sind immer gerichtet – vom Template zur Klasse, nicht umgekehrt.
- Templates können keine Superklassen wohl aber Subklassen sein. D.h., jede Klasse, die durch Bindung der Parameter eines Templates entsteht, das eine Subklasse ist, ist auch eine Subklasse der entsprechenden Superklasse.



# Beispiel





# Deklarationen

**Falls all diese Beschreibungselemente noch nicht ausreichen,  
dann können eigene Erweiterungen definiert werden.**

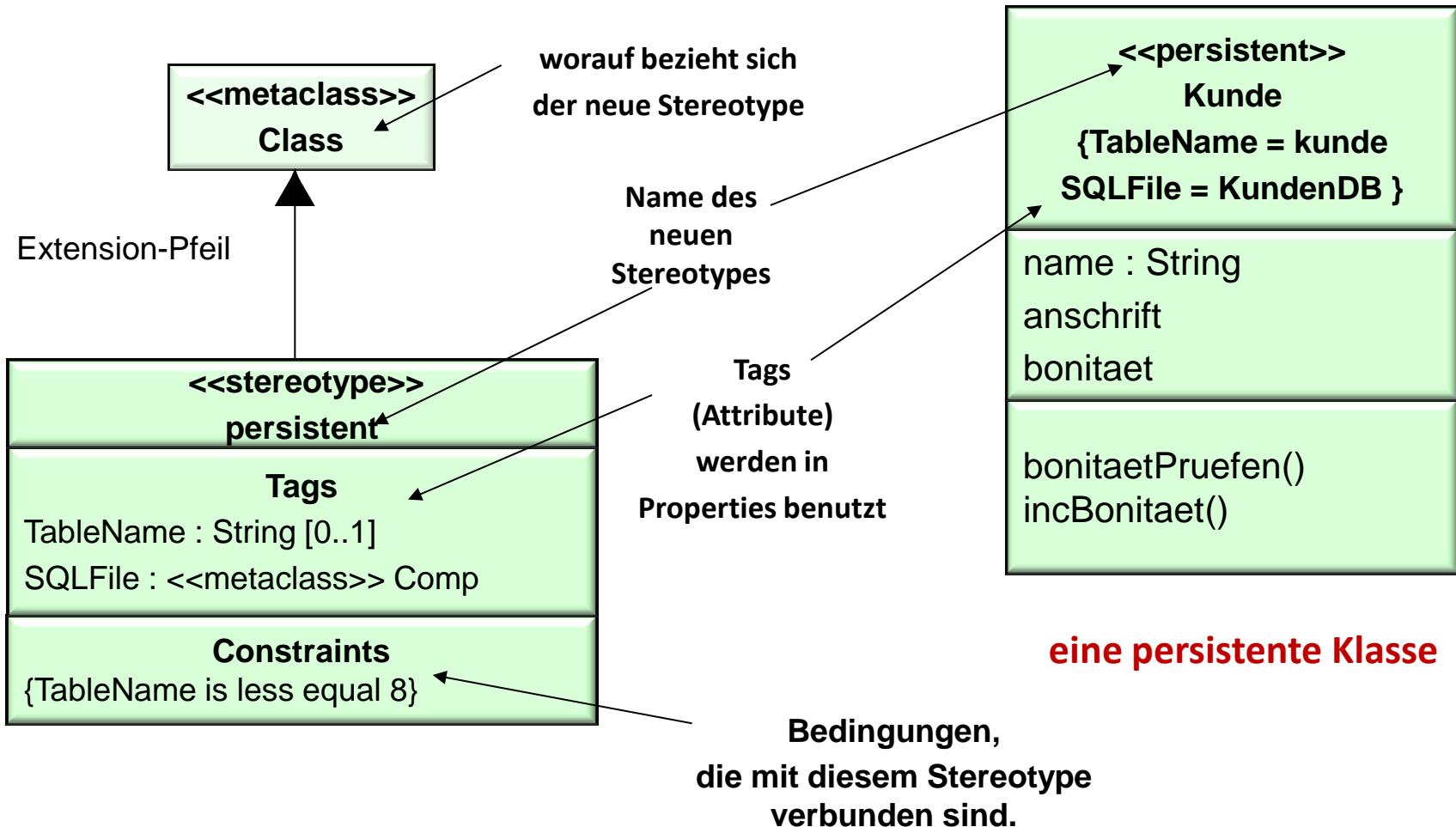
Durch **Deklaration** eines eigenen Stereotypes  
(eines Meta-Modell-Elements auf Modellebene)



Die Spezifikation kann besonderen Gegebenheiten angepasst werden.



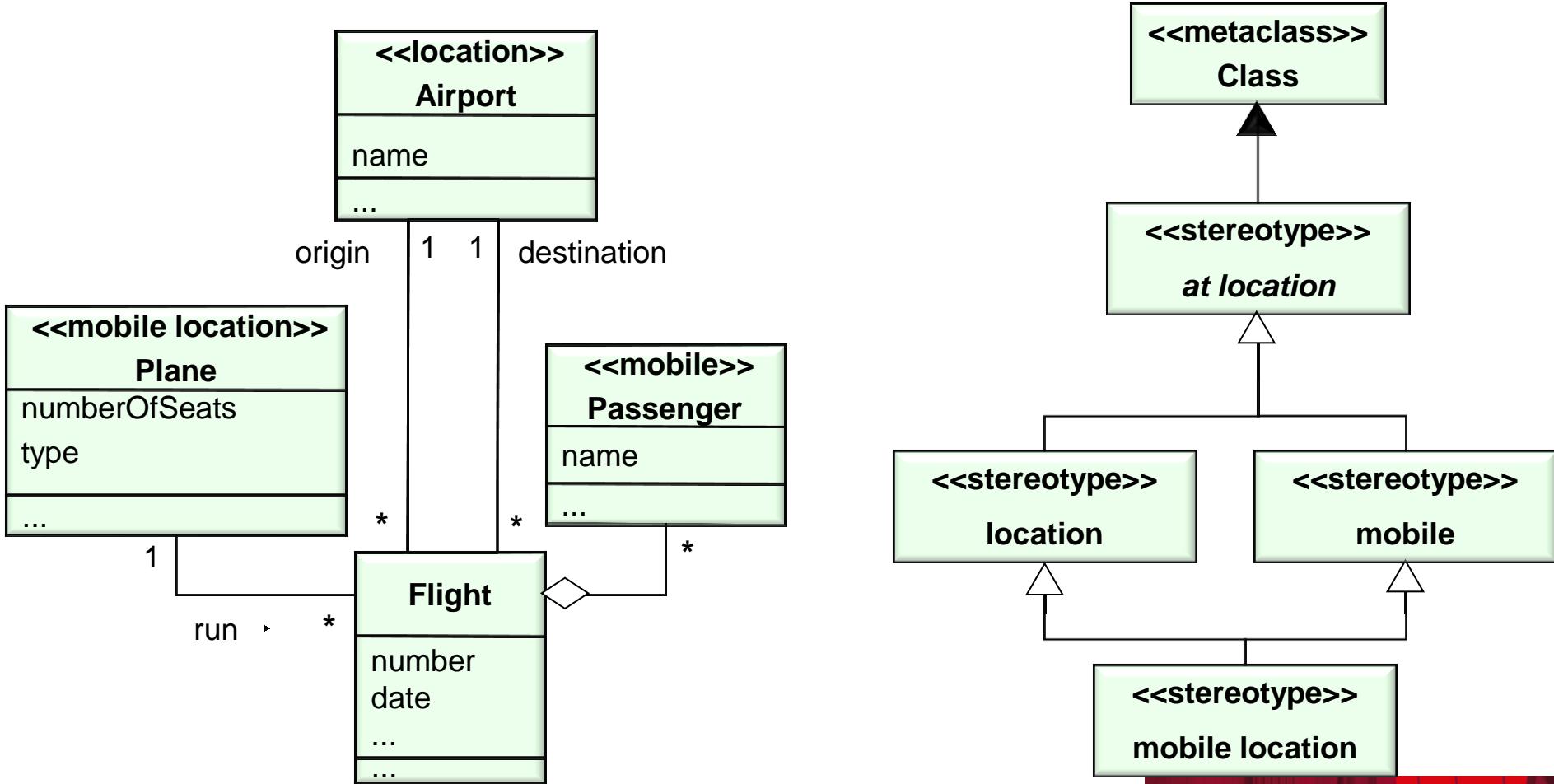
# Beispiel





# Bsp.: Modellierung mobiler Systeme

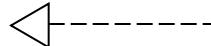
L. Baumeister, N. Koch, P. Kosiuczenko, and M. Wirsing. Extending Activity Diagrams to Model Mobile Systems. NODE 2002.





# Beziehungen zwischen Klassen

**Wie beschreibt man Assoziationen,  
Aggregationen und Kompositionen – und welche  
Beziehungen gibt es noch?**

- Bisher haben wir zwei Beziehungen betrachtet
  - Spezialisierung/Generalisierung      
  - (Spezialfall) Realisierung      



# Assoziationen

**Die allgemeinste Form der Beziehung zwischen Klassen ist die Assoziation. Sie beschreibt die Struktur einer Menge von Objektverbindungen und drückt damit die Verantwortlichkeit untereinander aus.**

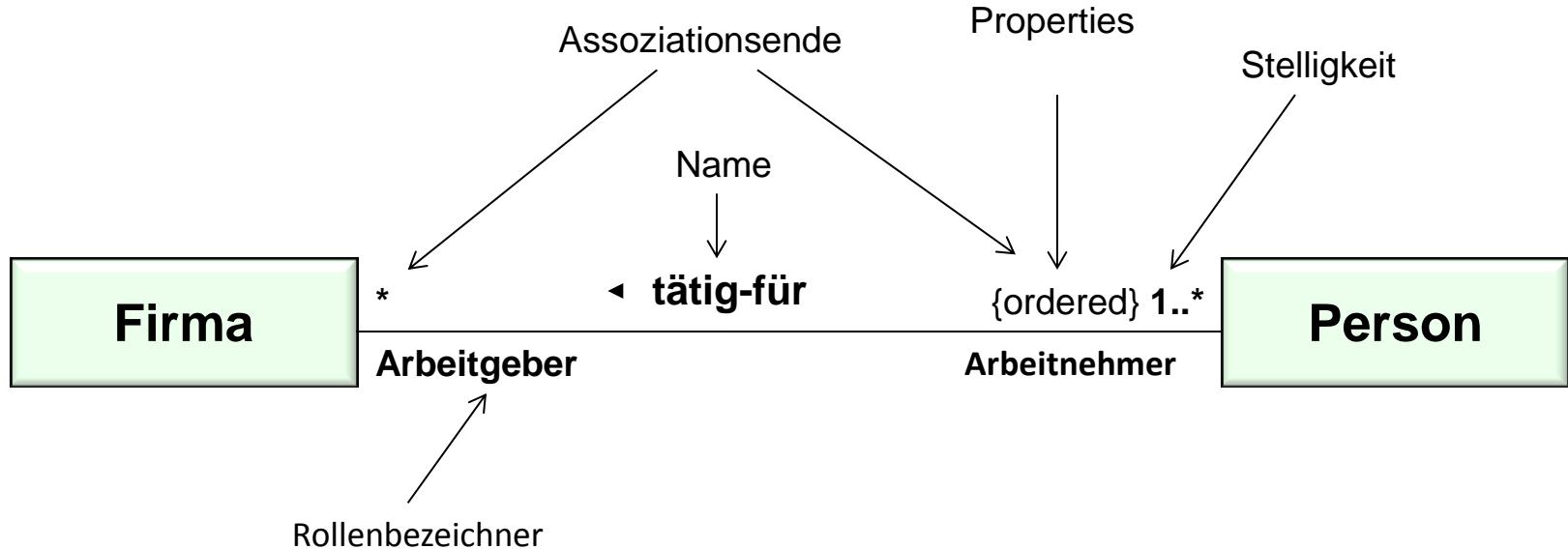
- Assoziationen sind notwendig, damit Objekte miteinander kommunizieren können.
- Assoziationen können zwischen einer, zwei, oder auch mehreren Klassen bestehen.
  - rekursive Assoziation
  - binäre Assoziation
  - n-stellige Assoziationen
- Assoziationen können sich über die gesamte Lebenszeit der beteiligten Objekte erstrecken oder auch nicht.
- An einer Assoziation können mehrere Objekte beteiligt sein.
- Eine Assoziation kann mit Attributen versehen werden.
- Eine Assoziation hat einen Namen.
- Eine Assoziation kann gerichtet sein.



# Binäre Assoziation

Die elementarste Form der Assoziation ist die  
binäre Assoziation.

Eine Beziehung zwischen den Objekten zweier Klassen.





# Wer kennt wen?

In Beziehungen kennt nicht immer jeder jeden.



Nur die Rechnung kann auf ihre Anschrift zugreifen.  
Die Anschrift selbst weiß nichts darüber,  
dass sie zu einer Rechnung in Beziehung steht.

Eine solche Assoziation heißt navigierbar.

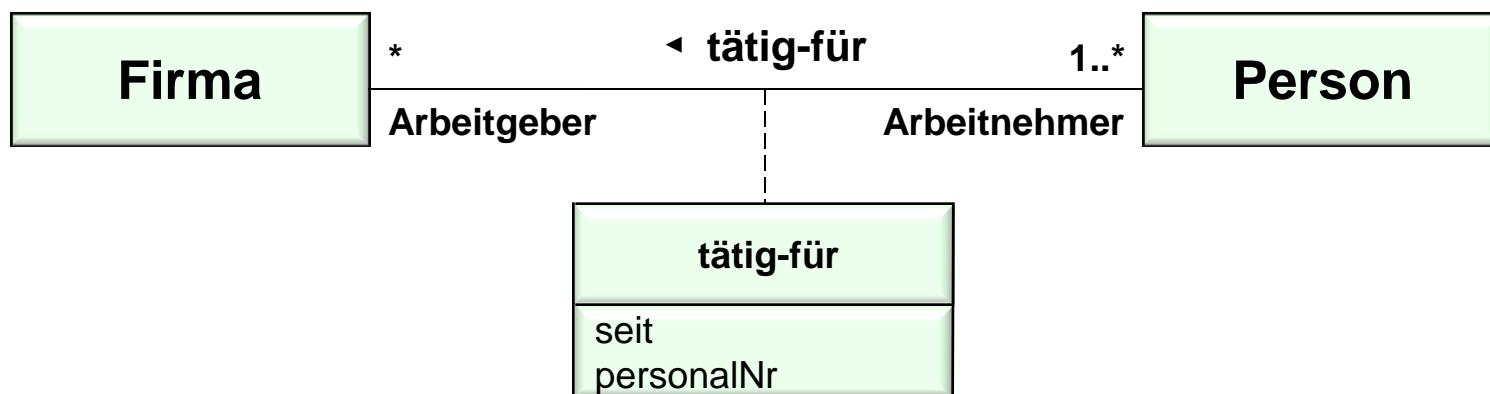


# Assoziationsklassen

Assoziationsklassen ermöglichen es, Beziehungen zwischen Klassen näher zu beschreiben; insbesondere dann, wenn Eigenschaften nicht einer beteiligten Klasse zugewiesen werden können.

Je nach Sichtweise hat man

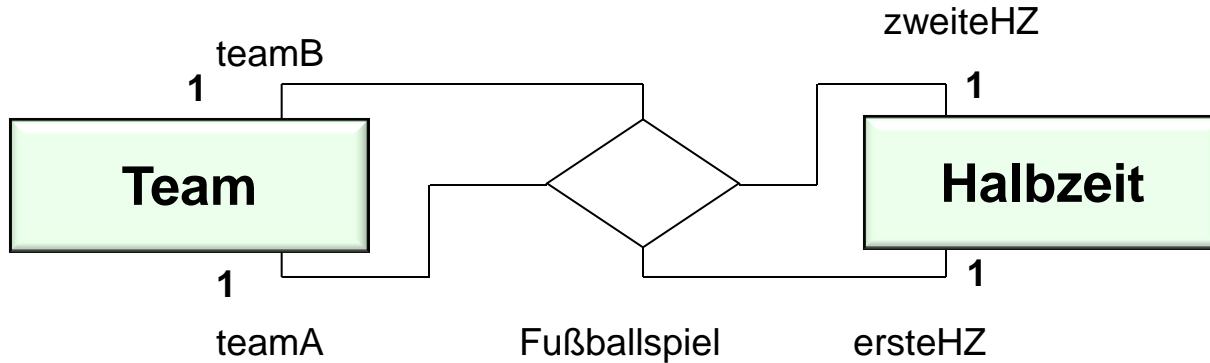
- eine Klasse mit Assoziationseigenschaften
- eine Assoziation mit Klasseneigenschaften.





# n-stellige Assoziationen

Beziehungen können auch mehrstellig sein



{self.teamA <> self.teamB and self.zweiteHZ <> self.ersteHZ}



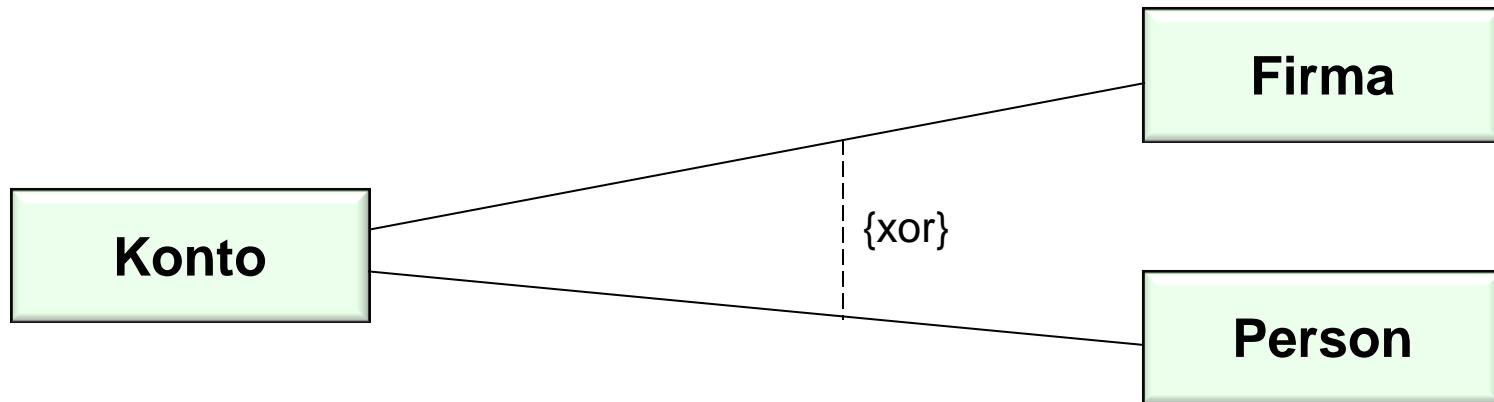
und können eingeschränkt werden (Constraints).



# Beziehungen zwischen Assoziationen

Assoziationen können zueinander in Beziehung gesetzt werden.

- über ein XOR-Constraint

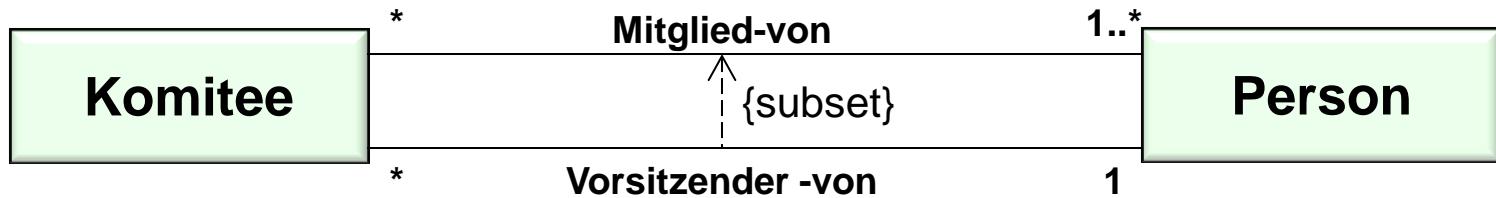


Man bezeichnet diese Form der Assoziationen auch als XOR-Assoziation.

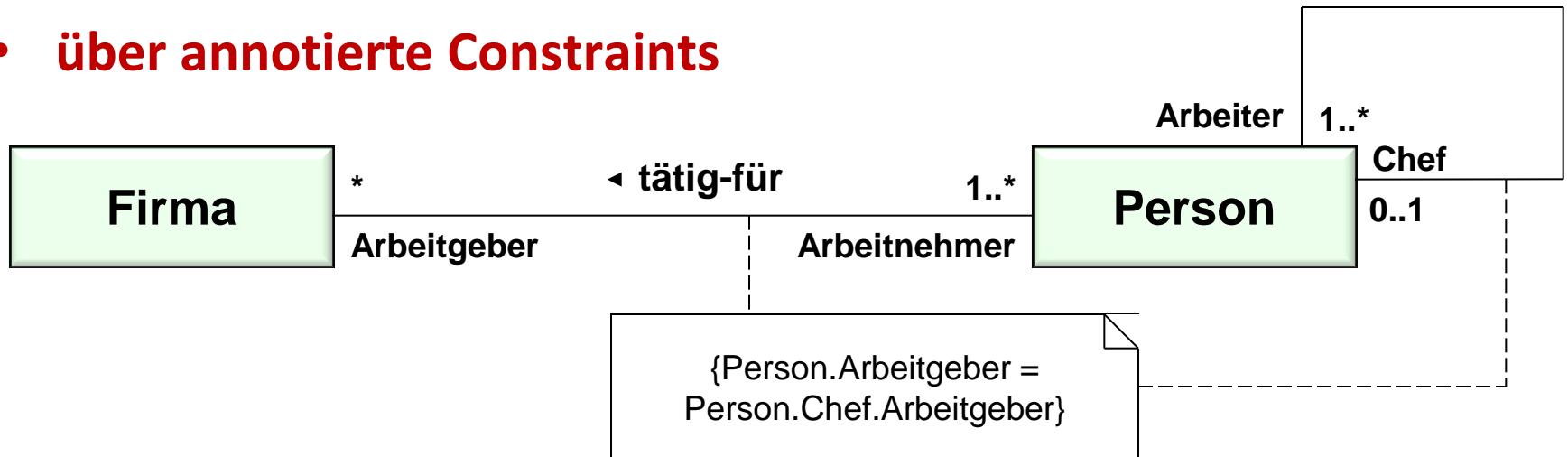


# Beziehungen zwischen Assoziationen

- über gerichtete Constraints



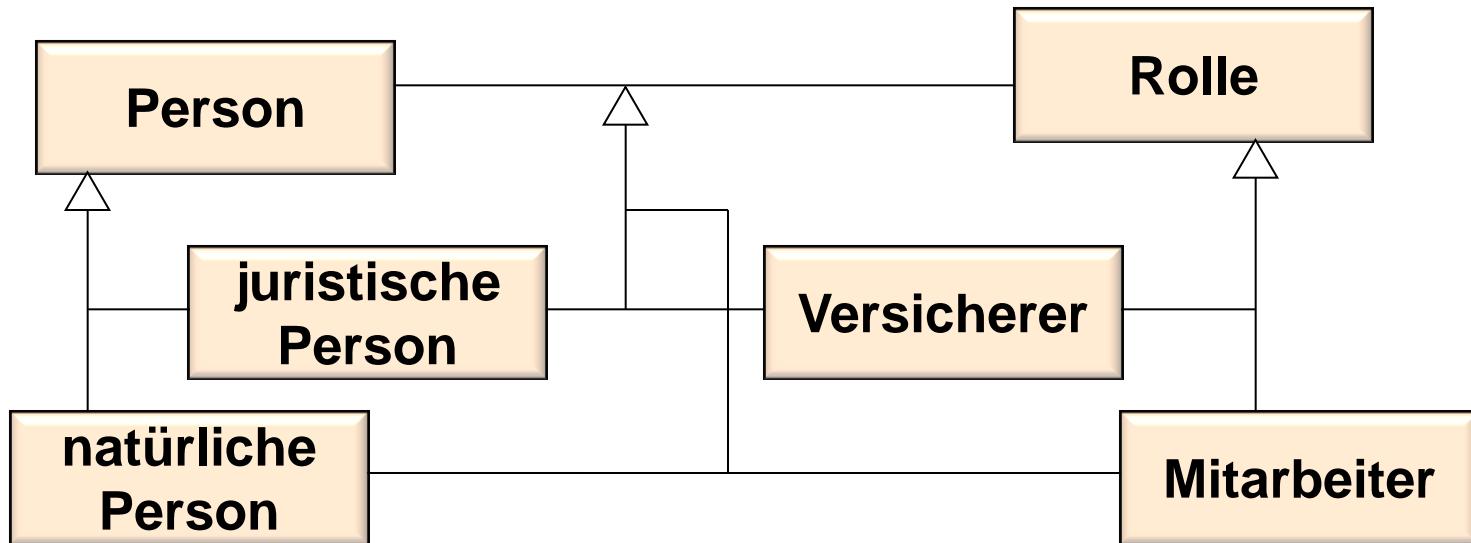
- über annotierte Constraints





# Beziehungen zwischen Assoziationen

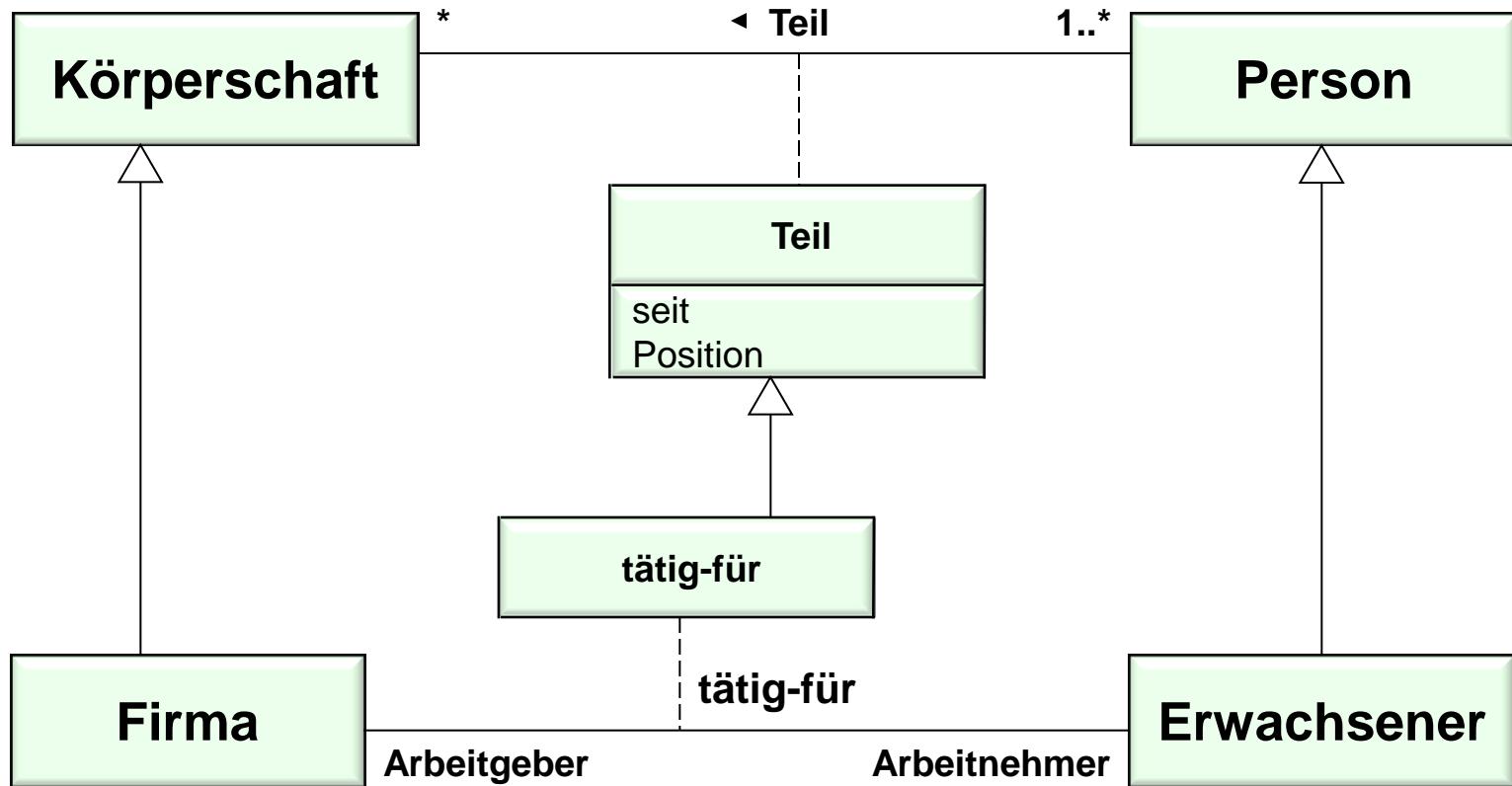
- durch Vererbung (I)





# Beziehungen zwischen Assoziationen

- durch Vererbung (II)

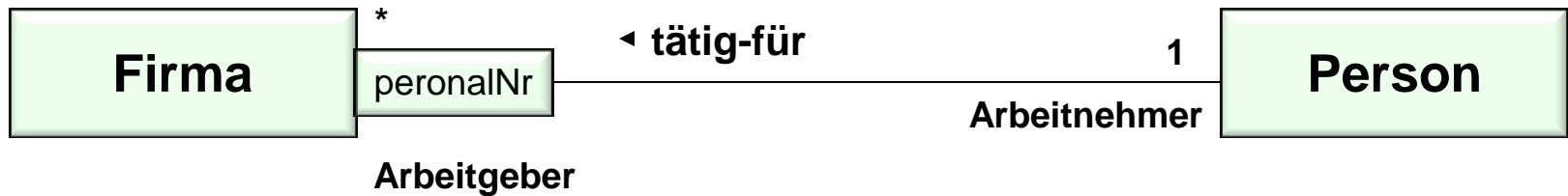




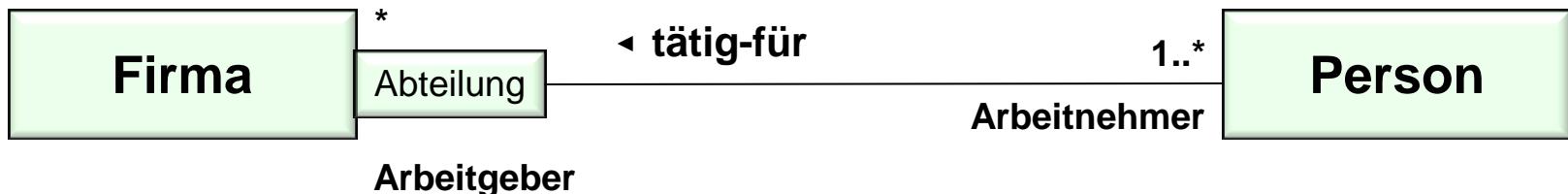
# Assoziationen werden eindeutig

Oft ist es unerwünscht, dass Assoziationen n-m-Beziehungen darstellen.  
In vielen Fällen sind die Objekte, die später an diesen Beziehungen partizipieren, sortiert oder geordnet und es kann dediziert auf sie zugegriffen werden.

Dies wird durch einen **Qualifier** modelliert



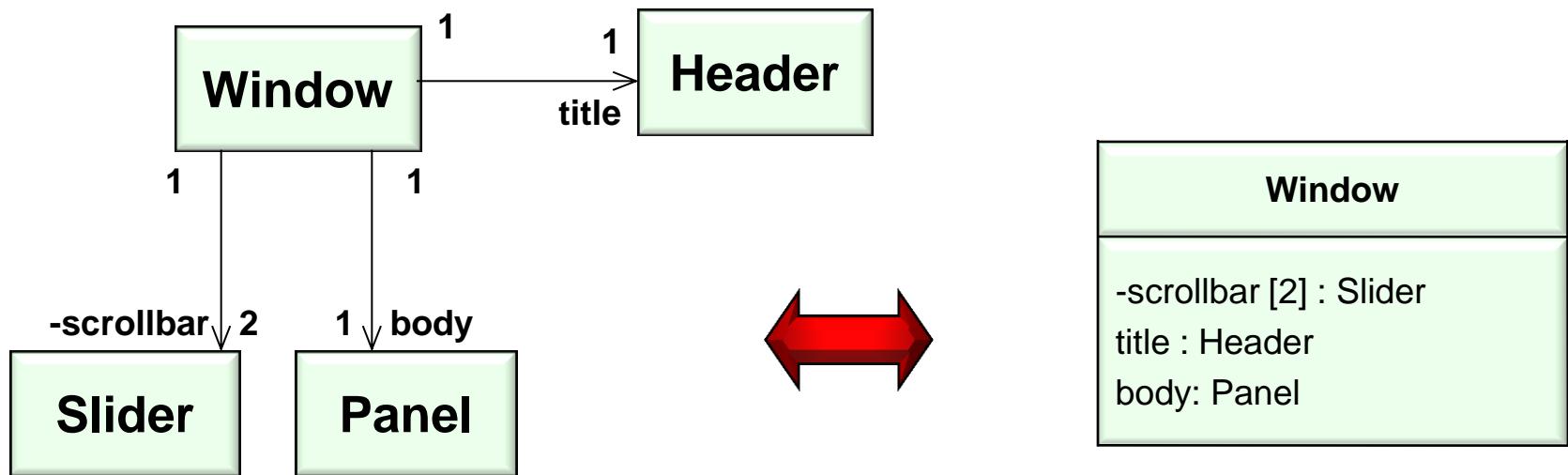
Ein Qualifier partitioniert die Mengen der an der Assoziation beteiligten Objekte.





# Assoziation und Attribut

Offensichtlich stellt jedes Attribut einer Klasse eine Assoziation dieser Klasse zur Klasse des jeweiligen Attributs dar.

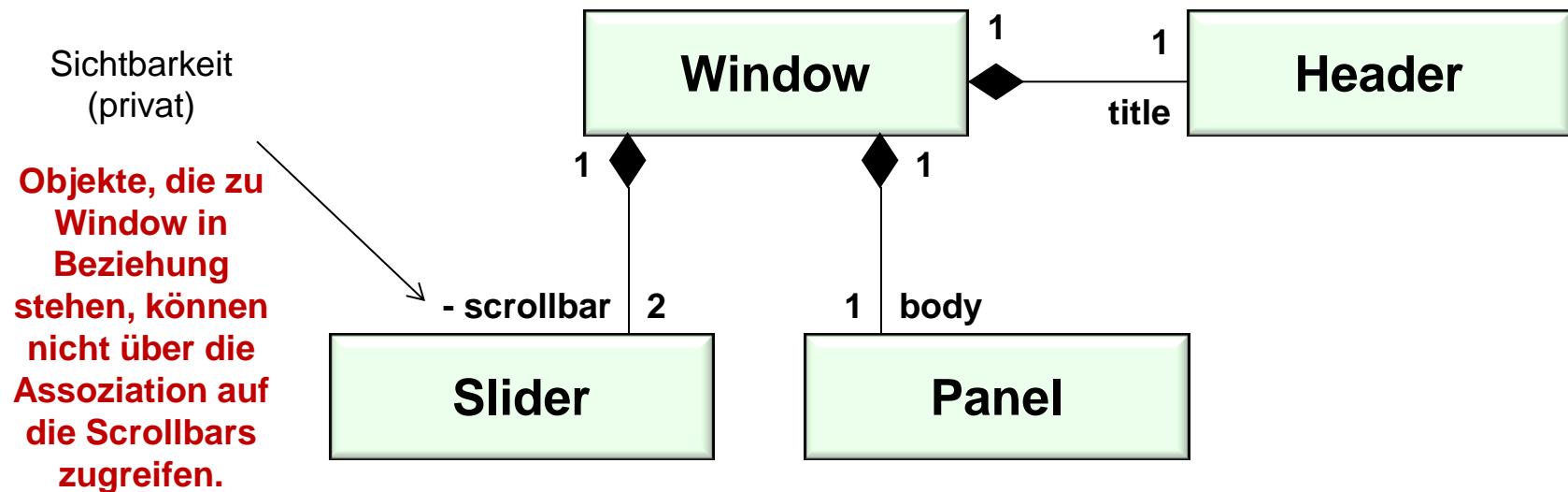




# Spezielle Assoziationen

- **Komposition**

Eine Komposition ist eine Teile-Ganzes-Beziehung, bei der die Teile existenzabhängig sind. Sie beschreibt, wie etwas Ganzes sich aus Einzelteilen zusammensetzt, und kapselt dieses.





# Komposition

- **Die Stelligkeit auf Seiten des Aggregats ist stets 1.**
  - kein Teil kann von mehr als einem Objekt existenzabhängig sein
  - kein Teil existiert ohne das Ganze
  - das Ganze ist für die Erzeugung und Zerstörung zuständig
- **Die Stelligkeit auf Seiten der Teile kann variabel sein.**
  - Teile werden erst später erzeugt
  - Teile werden schon früher zerstört



**Kein Teil existiert ohne sein Aggregat.**



# Spezielle Assoziationen

- **Aggregation**

Die Aggregation ist wie die Komposition eine Teile-Ganzes-Beziehung, jedoch in abgeschwächter Form. Die Teile sind nun nicht mehr existenzabhängig aber auch nicht gleichwertig.

Dies unterscheidet die Aggregation von der Assoziation.

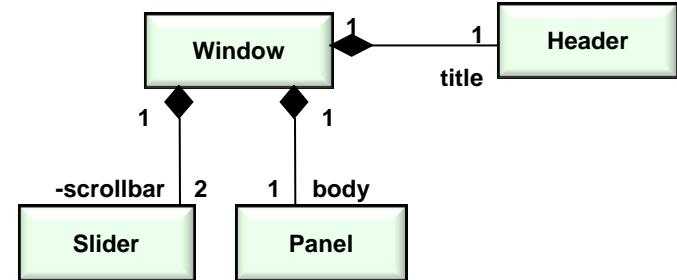
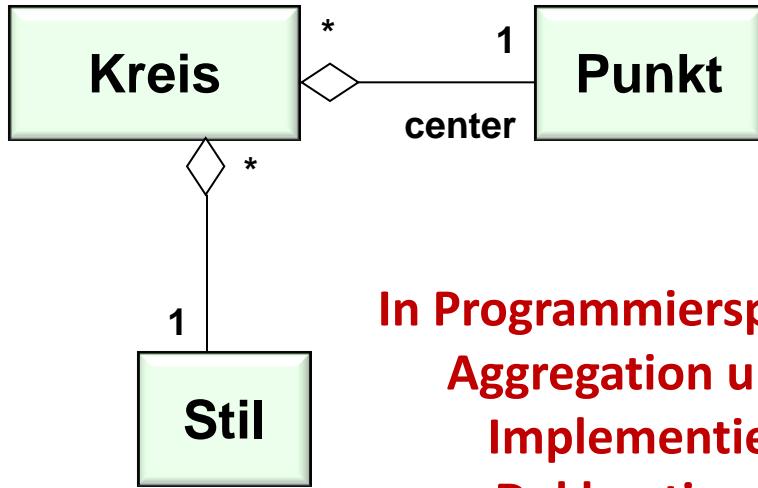
**Ein Aggregat handelt stellvertretend für seine Teile.**

- Operationen des Aggregats beziehen sich oft nur auf dessen Teile.
- Ein Aggregat nimmt Anfragen für seine Teile entgegen und delegiert.

Eine Aggregation dient der Zusammensetzung eines Objekts aus einer Menge von Einzelteilen.



# Aggregation



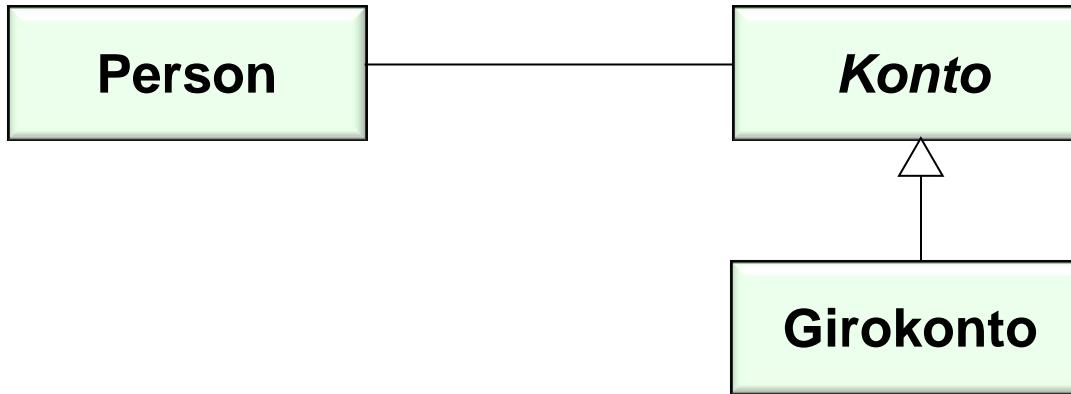
In Programmiersprachen wie C++ führt die Unterscheidung von Aggregation und Komposition zu einer unterschiedlichen Implementierung. Während die Aggregation über die Deklaration eines Zeigers realisiert wird, wird für die Komposition meistens ein Objekt deklariert.

```
class Kreis {  
    Punkt* center;  
    Stil* shape;  
    ...  
}
```

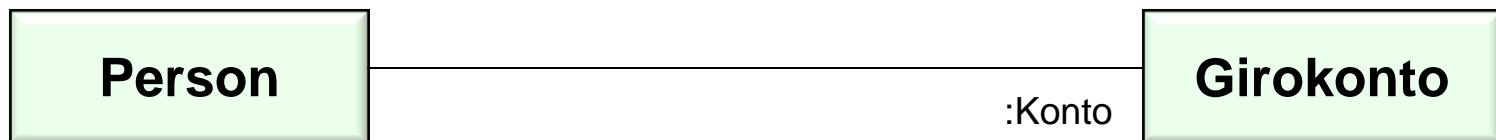
```
class Window {  
    scrollbar Slider[2];  
public:  
    Header title;  
    Panel body;  
    ...  
}
```



# Assoziationen werden vererbt



Es kann sein, dass in manchen Teilmodellen, nur Girokonten benötigt werden, um dennoch die Assoziation „Person Konto“ hervorzuheben (im Spezialfall Girokonto), versieht man die Assoziation zum Girokonto mit einem **Interface-Specifier**.



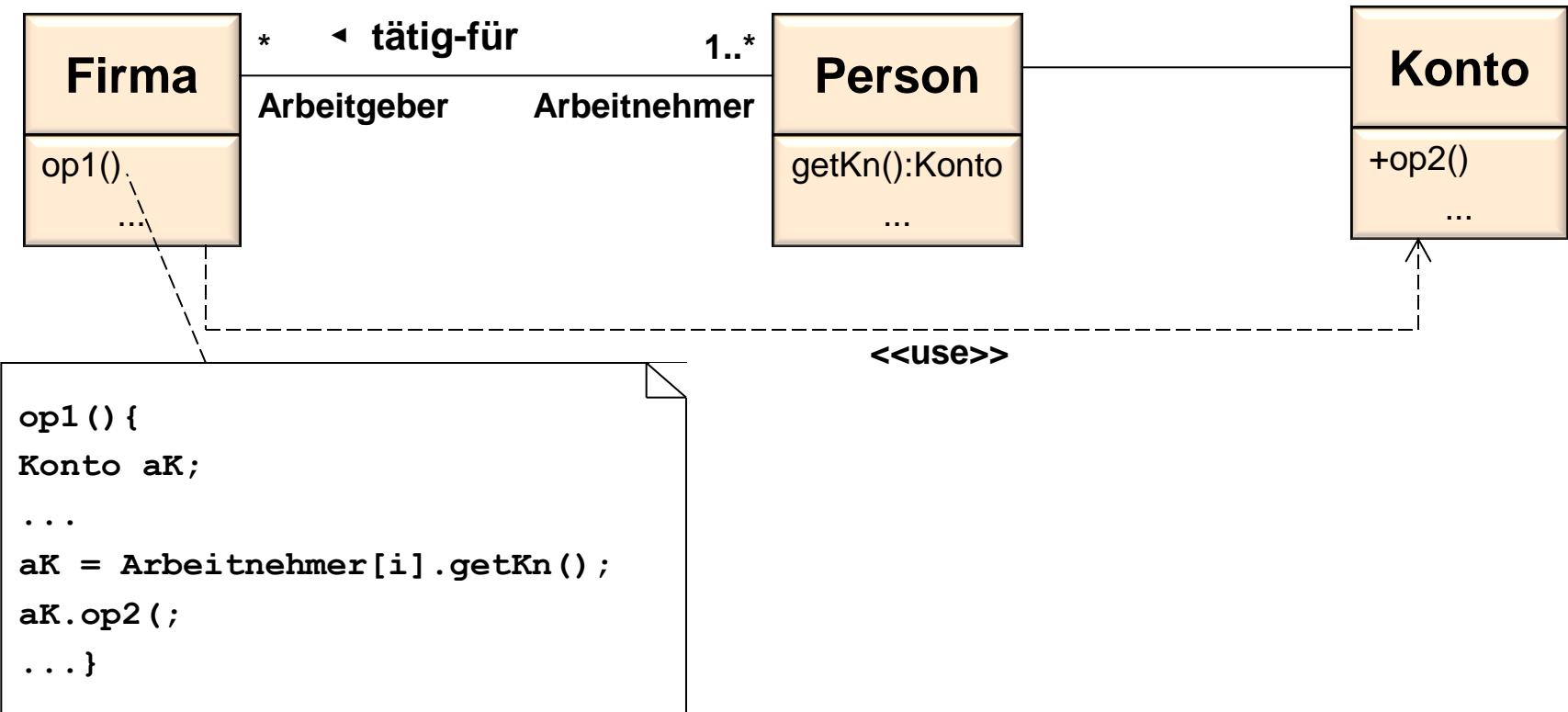
**Assoziationen können sich auf allgemeinere Klassen beziehen.**



# Abhängigkeiten (Dependencies)

Viele Beziehungen zwischen Instanzen von Klassen beruhen auf Assoziationen, ohne selbst eine Assoziation zu sein, oder sind lediglich dynamischer Art (Objekte werden übergeben).

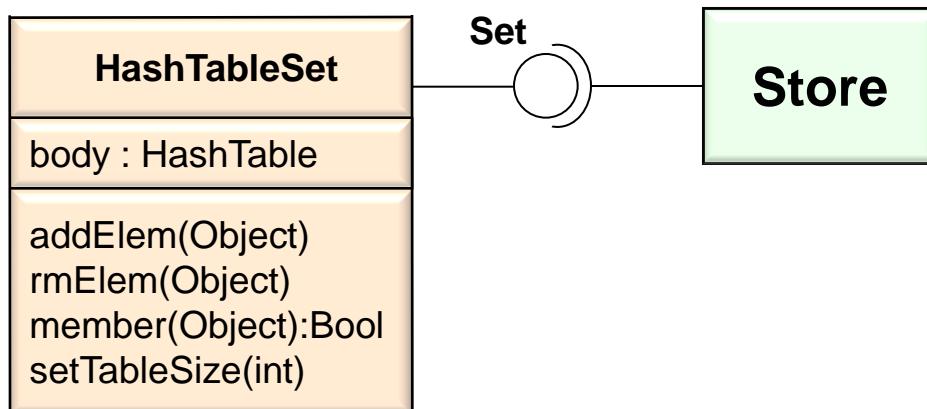
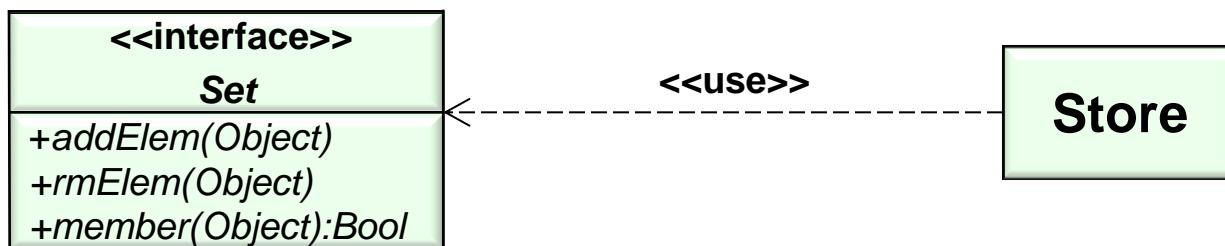
Um diese im Modell hervorzuheben, verwendet man **Dependencies**.





# Abhängigkeiten (Dependencies)

Sehr häufig findet man Abhängigkeiten bei Interfaces





# Instanzdiagramme

Instanzdiagramme zeigen die Beziehungen zwischen einzelnen Objekten. Statt Assoziationen spricht man jetzt von Links. Für Links können die gleichen Stilmittel verwendet werden wie für Assoziationen.

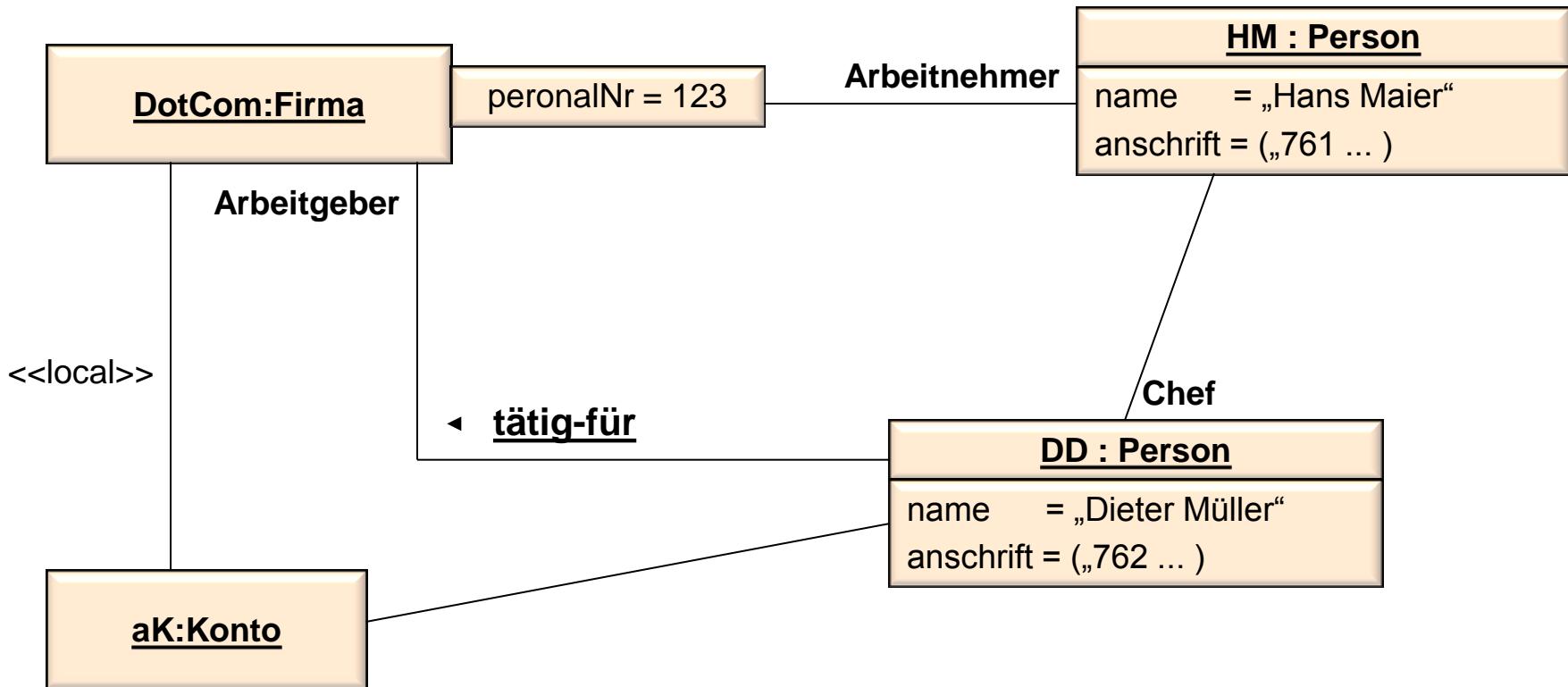
**Ausnahme:** die Multiplizität (Stelligkeit) ist immer 1

Links entstehen nicht nur aufgrund statischer Assoziation, sondern auch durch:

- Parameterübergabe (<<parameter>>)
- lokale Variablen bei Operationen (<<local>>)
- Verwendung globaler Variablen (<<global>>)
- Selbstreferenz (<<self>>)



# Beispiel



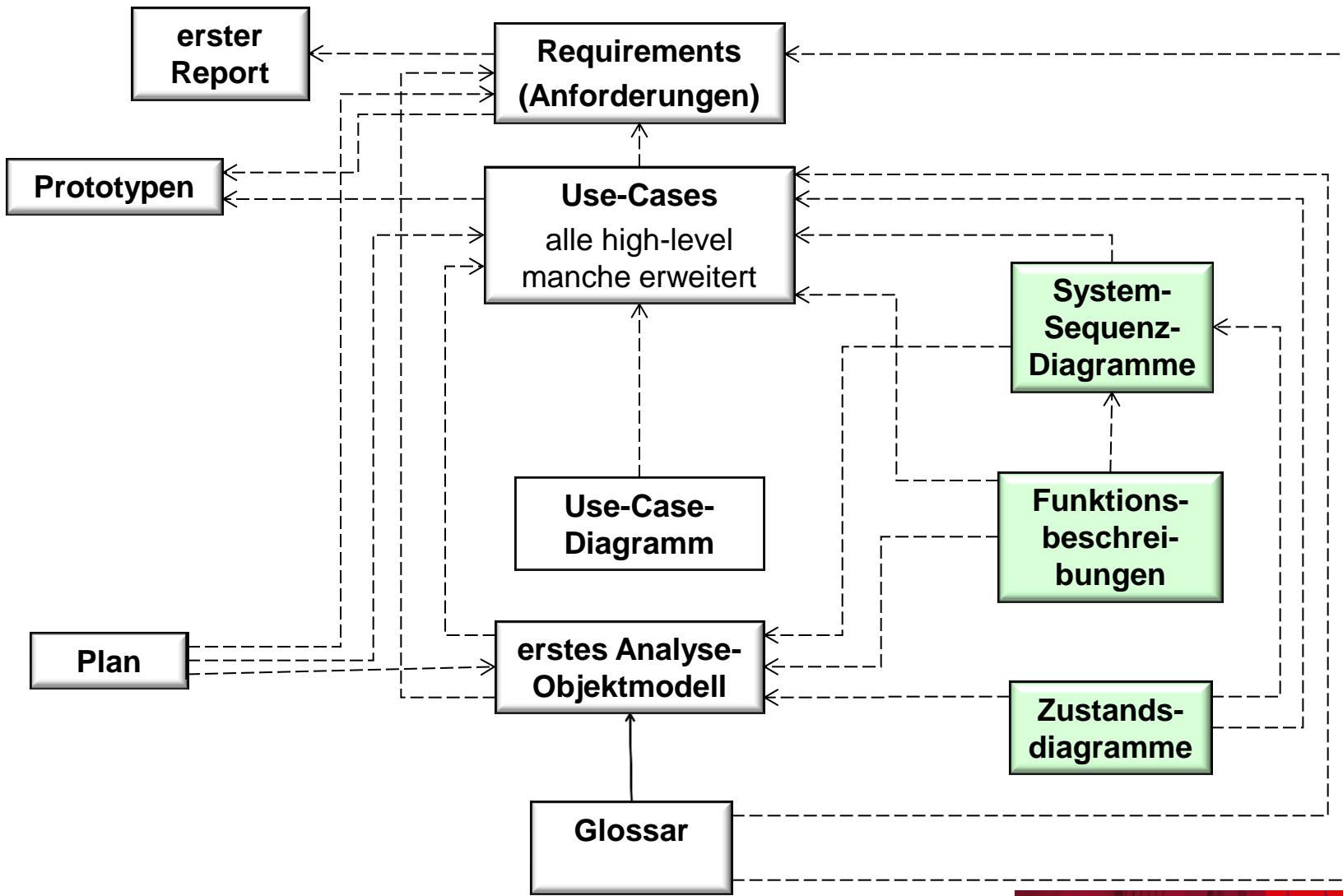


# Das dynamische Modell ergänzen

- **Sequenz-Diagramme erstellen**
- **Schlüsseloperationen definieren**
- **wichtige Parameter einfügen**
- **Schlüsseloperationen Objekten zuordnen**
- **für zentrale Objekte Zustandsübergangsdiagramme anfertigen**



# Das dynamische Modell ergänzen





# Das dynamische Modell ergänzen

- das Systemverhalten beschreiben
- die Interaktionen bestimmen, Reihenfolgen festlegen, wichtige Parameter berücksichtigen

**Bisher: Use-Case-Diagramme und Activity-Diagramme**

## Ziel:

- Sequenz-Diagramme erstellen
- Schlüsseloperationen definieren
- wichtige Parameter einfügen
- Schlüsseloperationen Objekten zuordnen
- für zentrale Objekte Zustandsübergangsdiagramme anfertigen



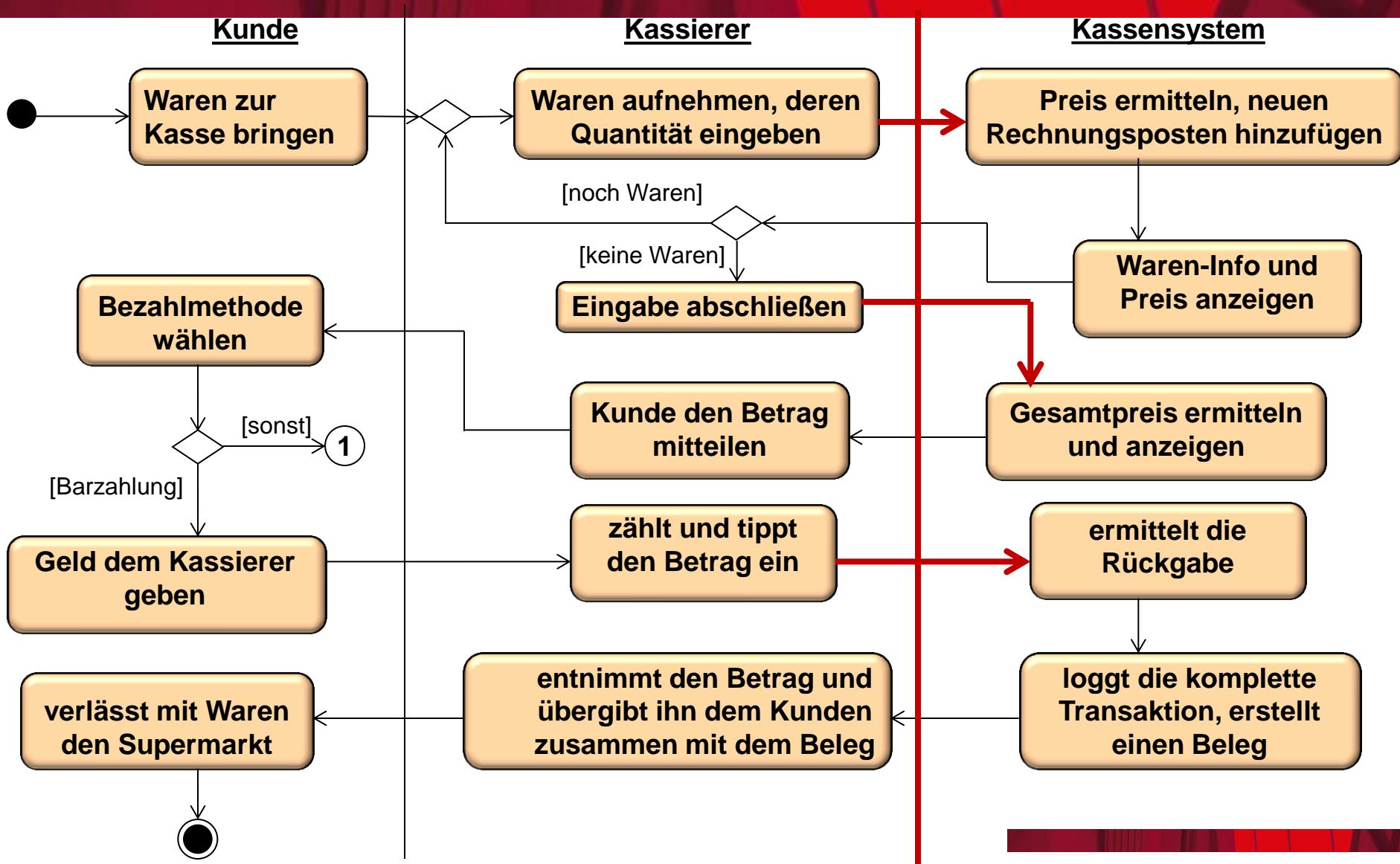
# System-Sequenz-Diagramme

- System-Ereignisse und System-Operationen sind zu identifizieren
- für jeden Use-Case sind System-Sequenz-Diagramme zu erstellen
- für jede Funktion ist eine Funktionsbeschreibung anzufertigen

Für einzelne Funktionen sind erweiterte Sequenzdiagramme zu erstellen, um Zugriffspfade zu validieren und weitere Operationen zu finden.

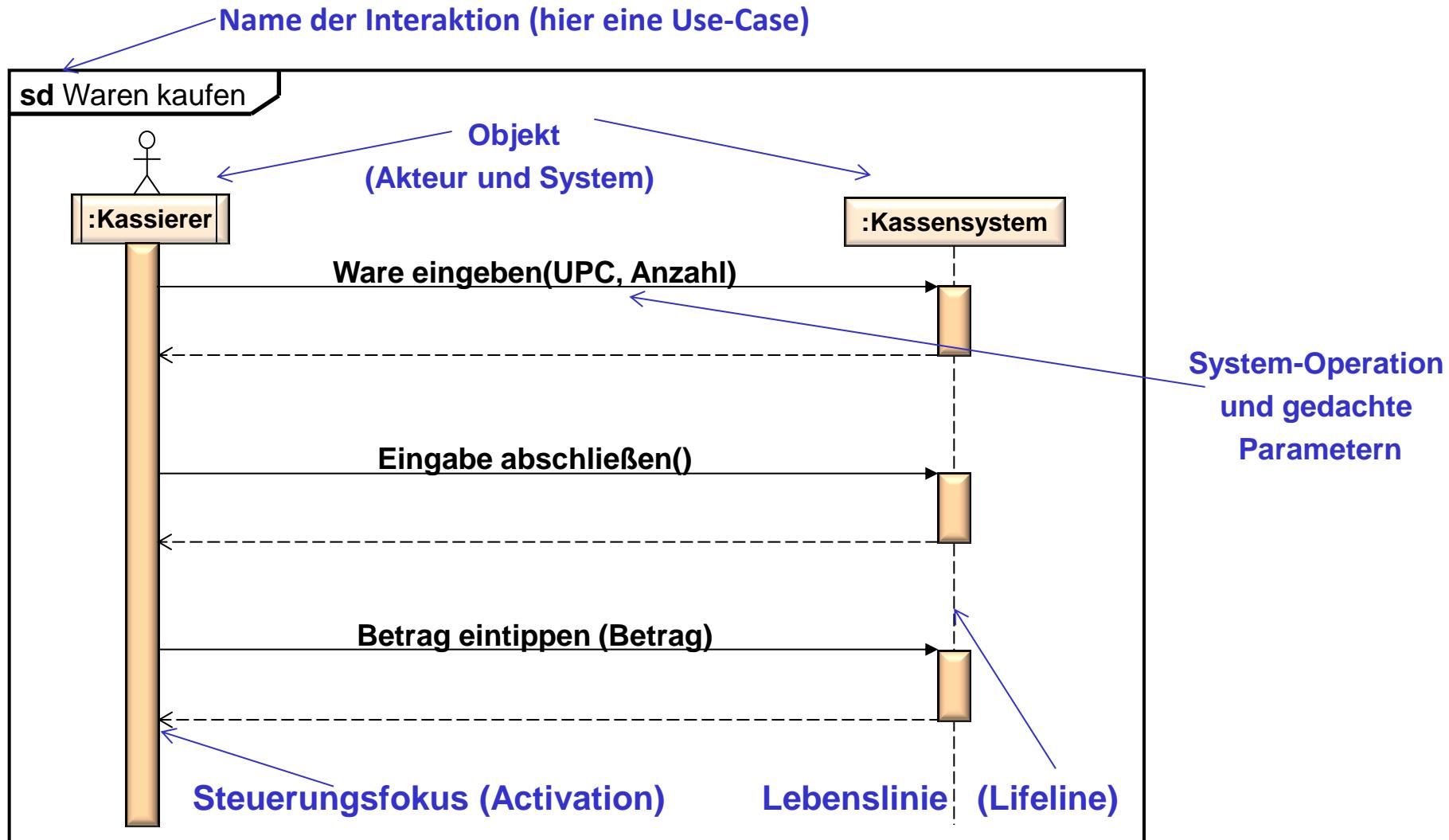


# System-Sequenz-Diagramme





# System-Sequenz-Diagramme



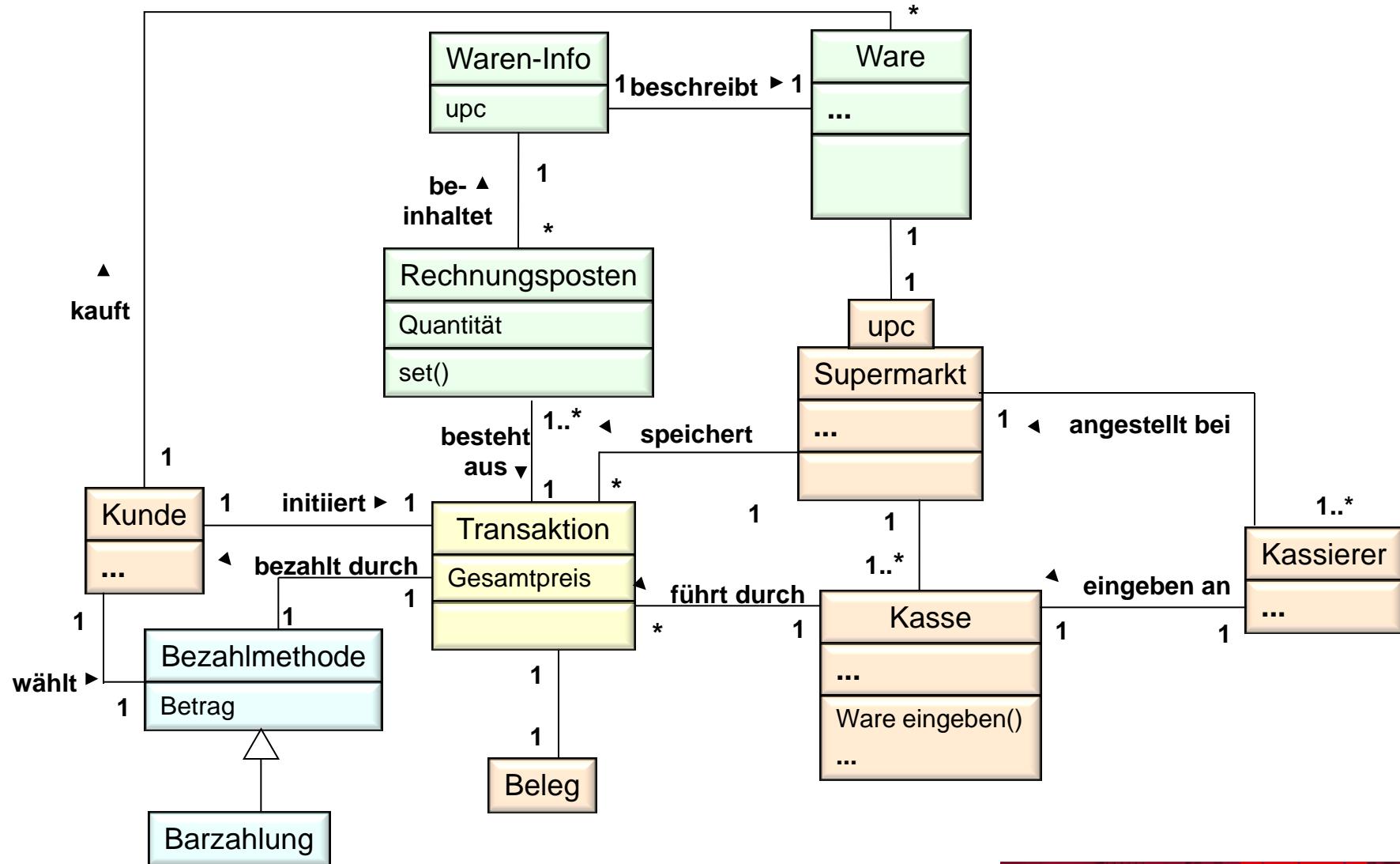


# Funktionsbeschreibung

Name:	<b>Ware eingeben (UPC:Code , Anzahl:Countable)</b>
Verantwortlichkeit:	registriert den Kauf der Ware und fügt ihn der Transaktion hinzu; zeigt Preis und Beschreibung an.
Referenzen:	Use-Case: Ware kaufen Funktionen: F1.1 – F5.6
Bemerkungen:	.....
Ausnahmen:	Falls UPC unbekannt ist, ist ein Fehler zu melden.
Vorbedingungen:	Kassierer ist angemeldet, Kassensystem ist initialisiert,...
Nachbedingungen:	<b>GGf.</b> (Beim ersten Aufruf) wurde eine neue Transaktion erzeugt, ein Rechnungsposten wurde erzeugt und hinzugefügt, usw...

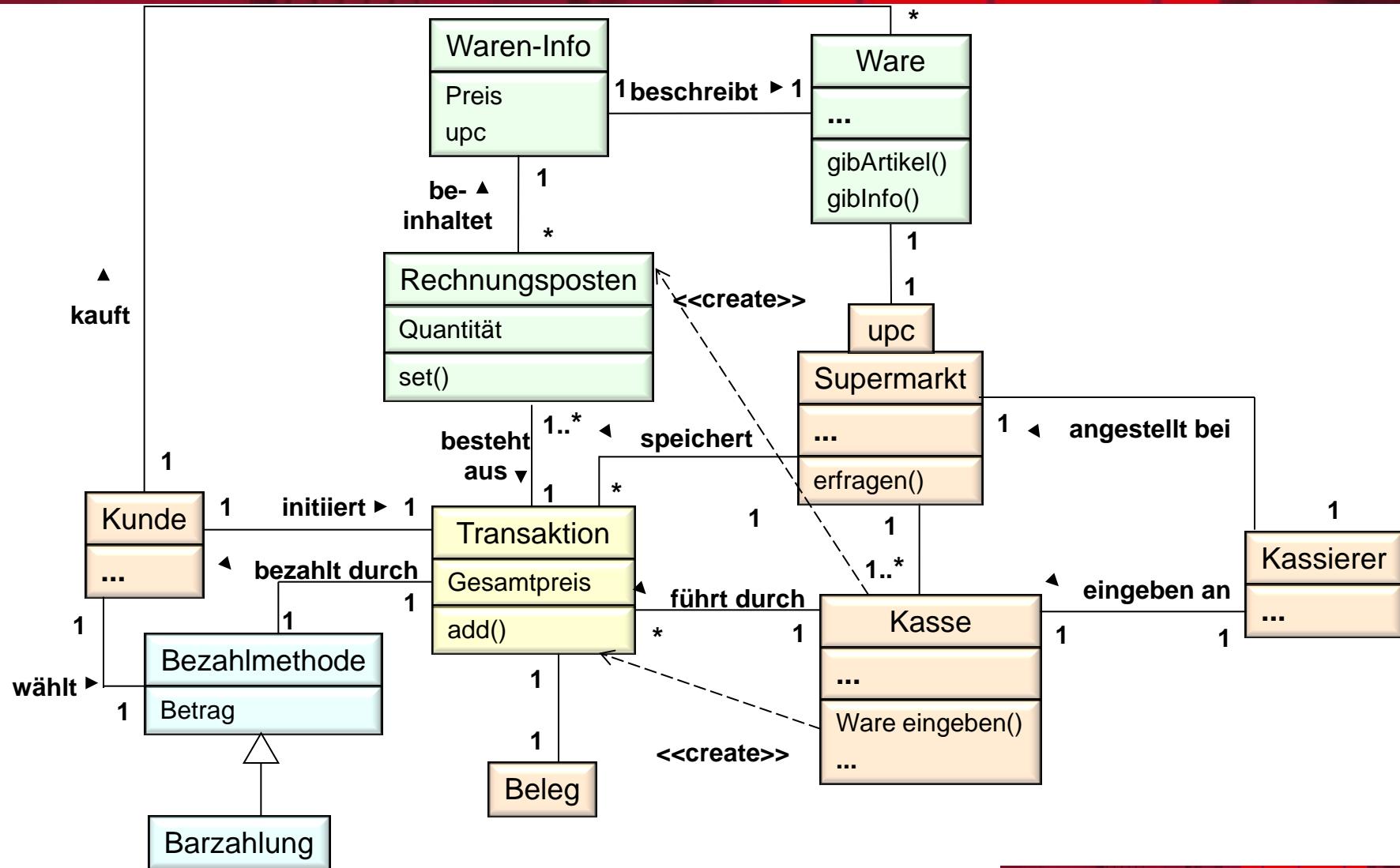


# Operationen ergänzen





# Operationen ergänzen (späte Phase)





# Analyse Zusammenfassung

- Anforderungen
- Use-Case-Diagramme (plus Beschreibung) und Activity-Diagramme
- Objektmodell und Data-Dictionary
- (System-)Sequenz-Diagramme plus Funktionsbeschreibungen

**beschreiben:**

- **Was sind die zentralen Abläufe?**
- **Was sind die zentralen Konzepte und Objekte?**
- **Was sind die zentralen Ereignisse und Operationen?**
- **Was tun die zentralen Operationen?**



# Analyse Zusammenfassung

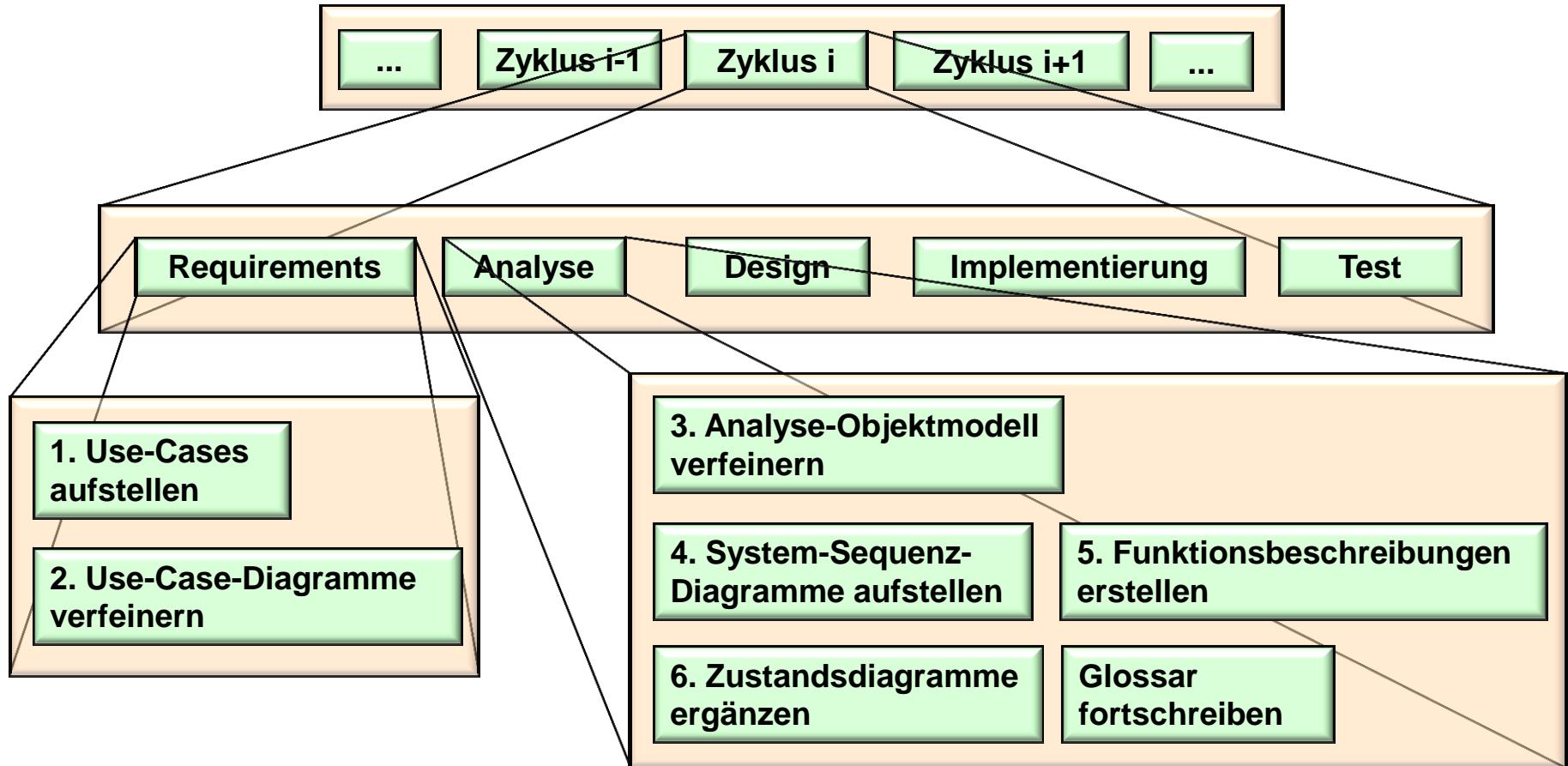
oder kurz:

**Welche Schnittstellen hat das System?**

- **Was sind die zusammenhängenden funktionalen Einheiten?**
- **Welche Schnittstellen gibt es?**
- **Welche Operationen werden an den Schnittstellen gebraucht?**
- **Was ist deren Aufgabe?**
- **Welche Konzepte sind relevant und wie hängen sie voneinander ab?**



# Die iterative Entwicklung (Zyklen)





Hochschule Karlsruhe  
Technik und Wirtschaft  
**UNIVERSITY OF APPLIED SCIENCES**

## **Software-Engineering**

# **Objektorientiertes Design (OOD nach Craig Larman)**

Prof. Dr. Thomas Fuchß  
Hochschule Karlsruhe – Technik und Wirtschaft  
Fakultät für Informatik und Wirtschaftsinformatik





# Phase I: Erstellung des Analysemodells

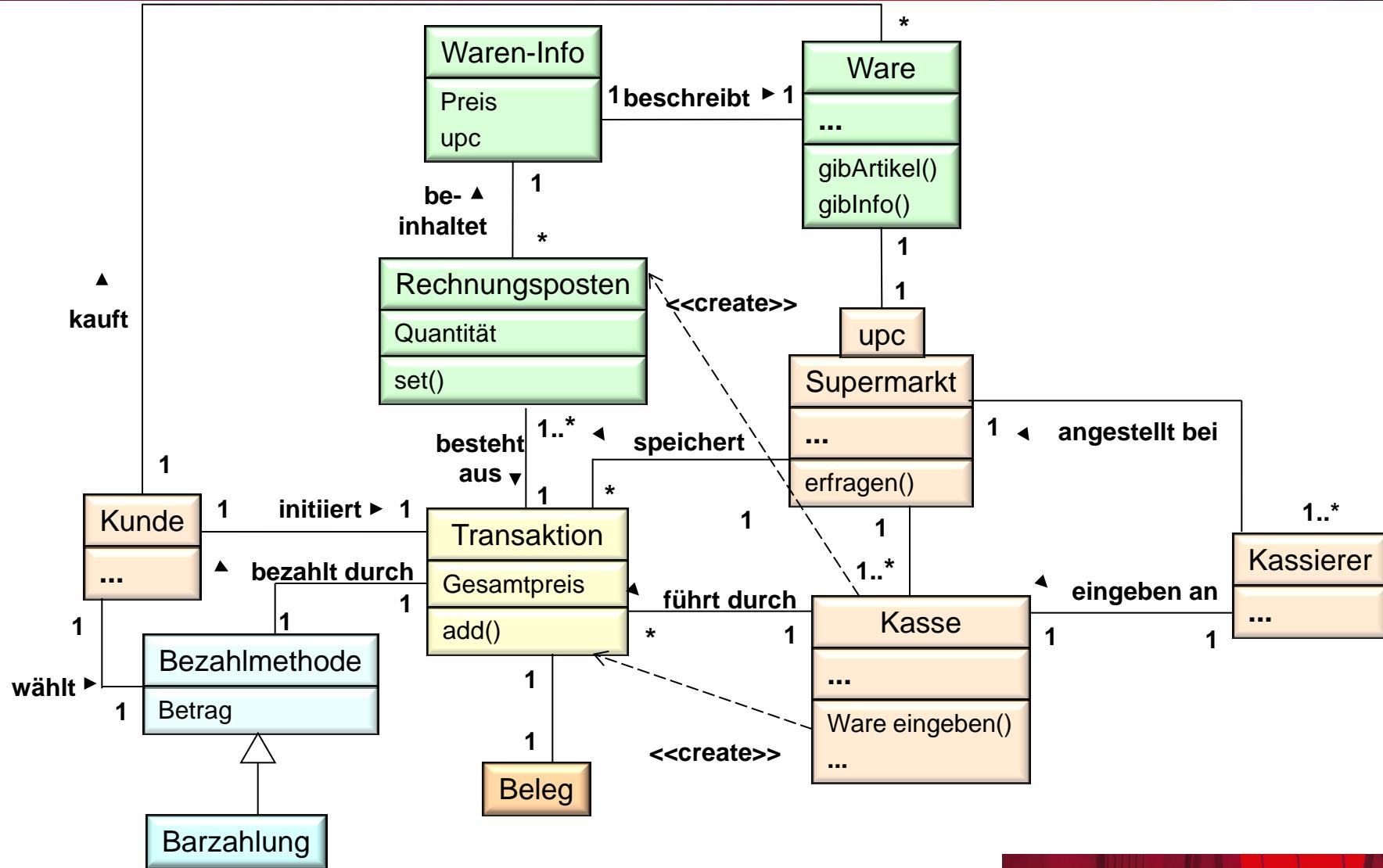
- **Das Objektmodell, die statische Struktur der realen Welt**
  - Objekte finden und organisieren
  - Verantwortlichkeiten und Beziehungen aufzeigen
  - im Data Dictionary/Wiki beschreiben
- **Das dynamisch funktionale Modell**
  - das Systemverhalten beschreiben
  - die Interaktionen bestimmen, Reihenfolgen festlegen, wichtige Parameter berücksichtigen
  - Zustandsübergangsverhalten zentraler Objekte beschreiben

**beschrieben durch:**

- Use-Case-Diagramme (plus Beschreibung) und Activity-Diagramme
- Objektmodell und Data Dictionary
- (System-)Sequenz-Diagramme plus Funktionsbeschreibungen
- falls nötig ergänzt um Zustandsdiagramme



# Bisheriges Objektmodell (Analyse)





# Der Übergang zwischen Analyse und Design

(nach Craig Larman: Applying UML and patterns)

- **Zu realisierende Use-Cases festlegen**

abhängig davon:

- Bedienkonzept und
- Mockups erstellen

- **Die logische Architektur entwerfen**

- **Erste Iteration:** Systemarchitektur festlegen
- **Kommende Iterationen:** Architektur erweitern, anpassen,...



# Systemarchitektur festlegen (Systementwurf)

Ziel ist es, die globale Systemstruktur auszuwählen!

- System in Teilsysteme zerlegen
- Strategien für die Datenhaltung festlegen
- Ansatz zur Implementierung der Ablauflogik festlegen
  - explizite / implizite Zustandsmaschine
  - use-case-spezifische Repräsentanten oder Systemrepräsentanten
- Parallelitäten bestimmen
- Teilsysteme entsprechenden Maschinen / Prozessoren / Prozessen / Threads zuweisen
- Globale Ressourcen festlegen und Zugriff organisieren
- Grenzbedingungen erfassen und dokumentieren
- Kompromissprioritäten festlegen



# Teilsysteme organisieren

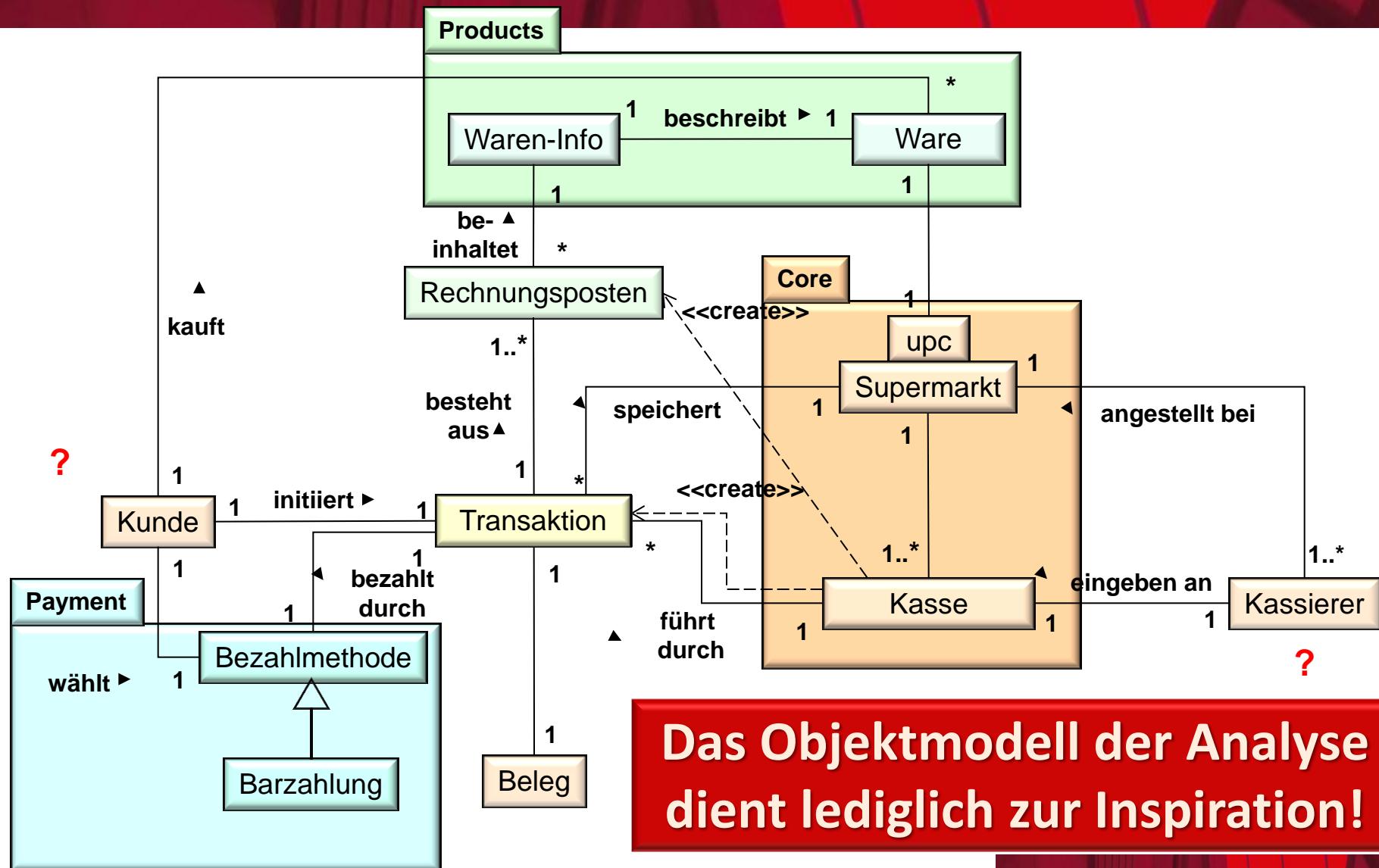
## einige Richtlinien

**Klassen werden wie folgt zusammengefasst:**

- Klassen, die demselben Aufgabenbereich zuzuordnen sind
- Klassen, die in derselben Typ hierarchie liegen
- Klassen, die in denselben Use-Cases vorkommen
- Klassen, die eine enge Beziehung aufweisen
- ...



# Objektmodell (Grundlage der Domäne)





# Packages

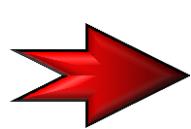
**Um Klassen zu gruppieren werden Packages verwendet. Packages selbst können wiederum Subpackages beinhalten.**

**Packages bilden eine echte Baumstruktur  
(kein Element gehört zu mehreren Packages)**

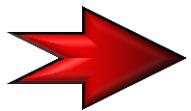


# Packages

Die Strukturierung eines Systems in Packages ist nicht willkürlich. Sie dient der Gruppierung und Abschottung zusammengehörender Elementen.



Packages erzeugen Übersichtlichkeit und verdeutlichen Abhängigkeiten  
(innerhalb eines Packages sind alle Namen disjunkt)

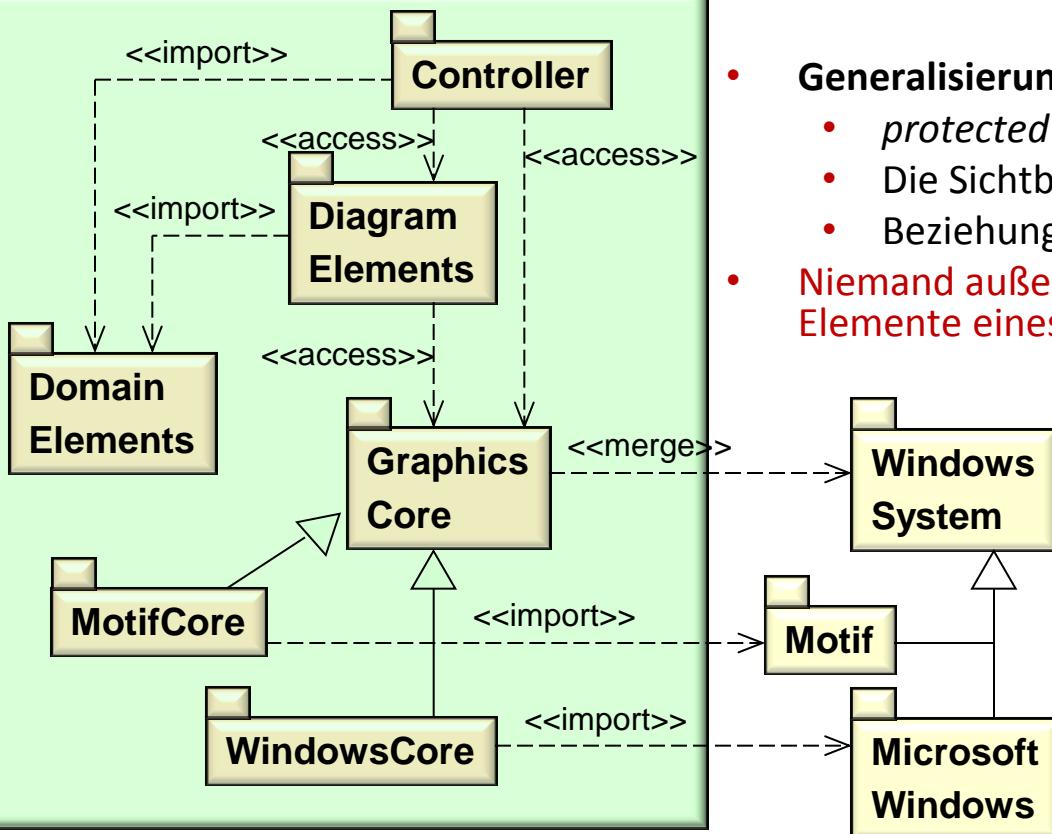


Packages sollten **nie** zyklisch voneinander abhängen



# Beispiel: Packages eines Editors

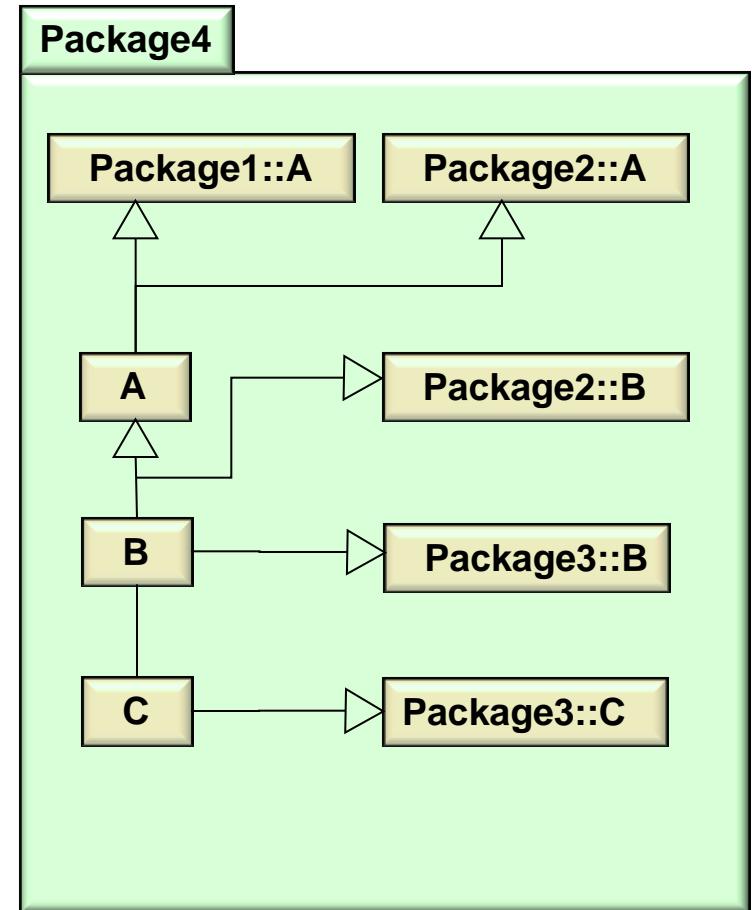
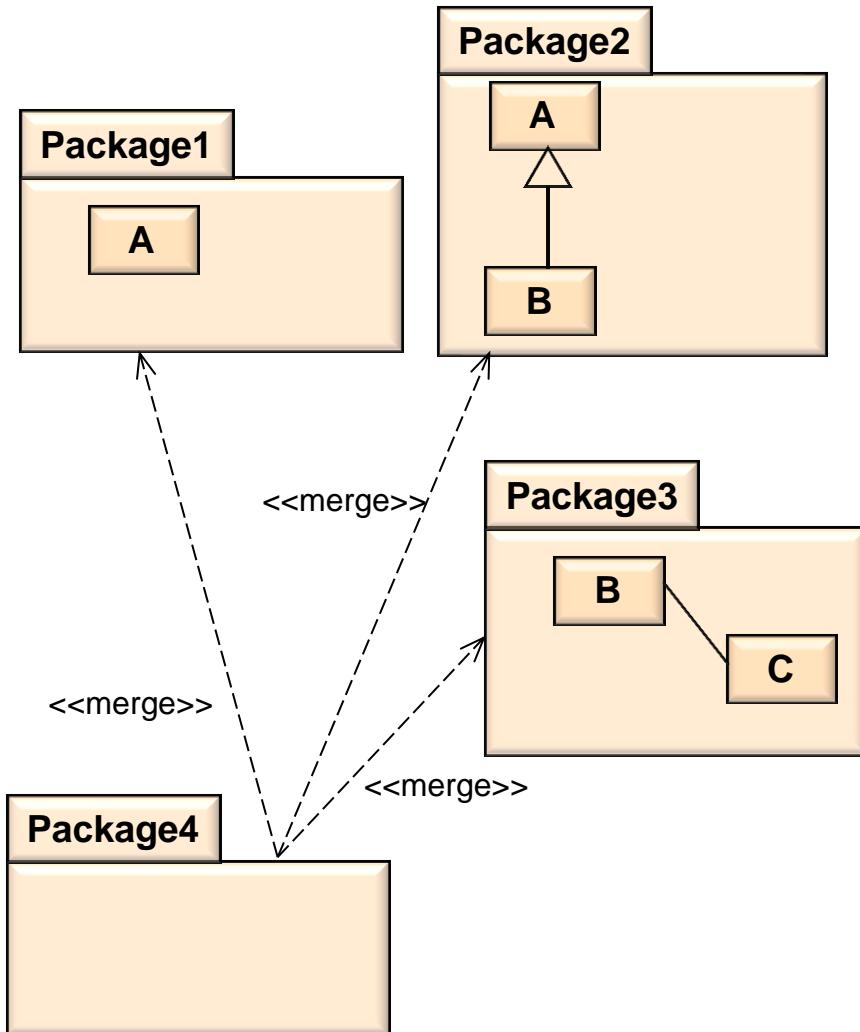
## Editor



- **access public** Elemente werden benutzt, der Namensraum bleibt jedoch getrennt.
- **import** alle *public* Elemente werden importiert, d.h. können benutzt werden.
- **merge** alle Elemente werden neu angelegt als Generalisierung der ursprünglichen.
- **Generalisierung**
  - *protected* und *public* Elemente stehen zur Verfügung.
  - Die Sichtbarkeit bleibt erhalten.
  - Beziehungen des Packages werden übernommen.
- Niemand außerhalb des Packages hat Zugriff auf **private** Elemente eines Packages



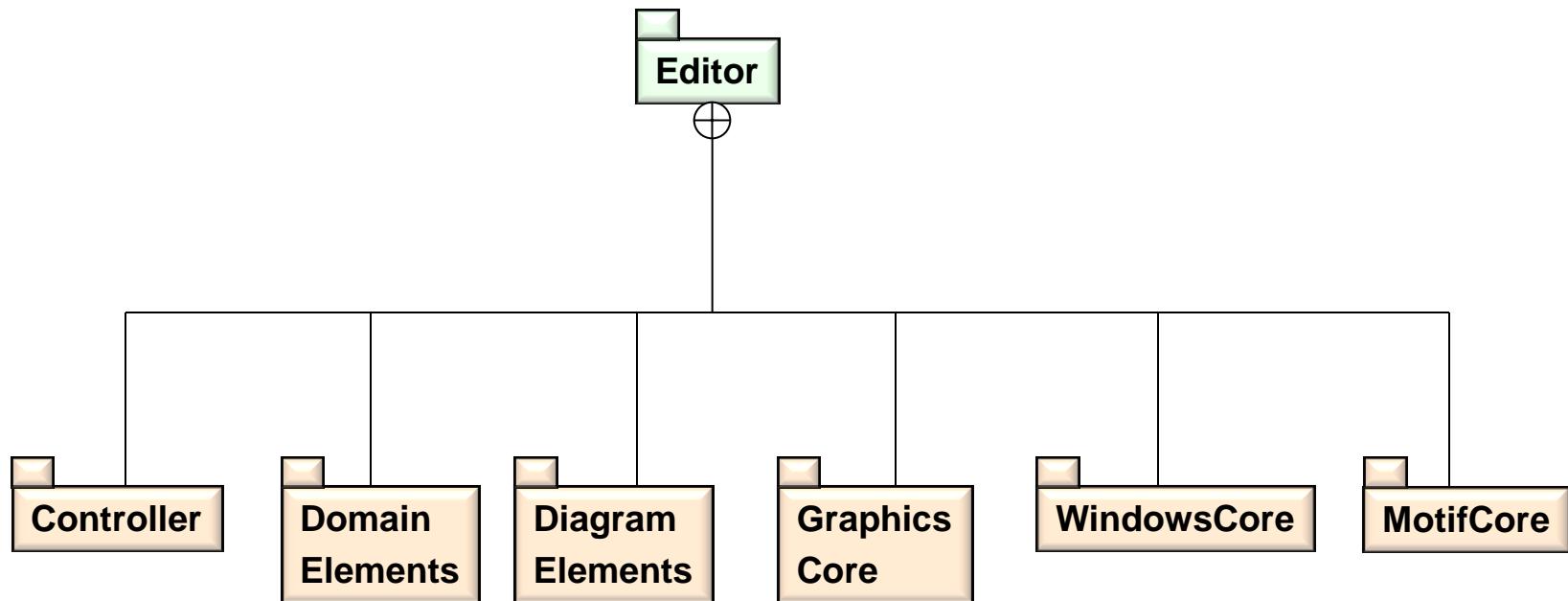
# Die Bedeutung von „merge“





# Alternative Darstellung

## Baumstruktur





# Implementierungsdiagramme

Implementierungsdiagramme dienen der Darstellung von Implementierungsaspekten:

- **Source-Code-Verteilung auf Bibliotheksebene (.jar, .o, .dll, ...)**
- **Laufzeitverteilung auf Rechner, Prozessoren u.ä.**

- **Komponentendiagramme (Component Diagrams)**

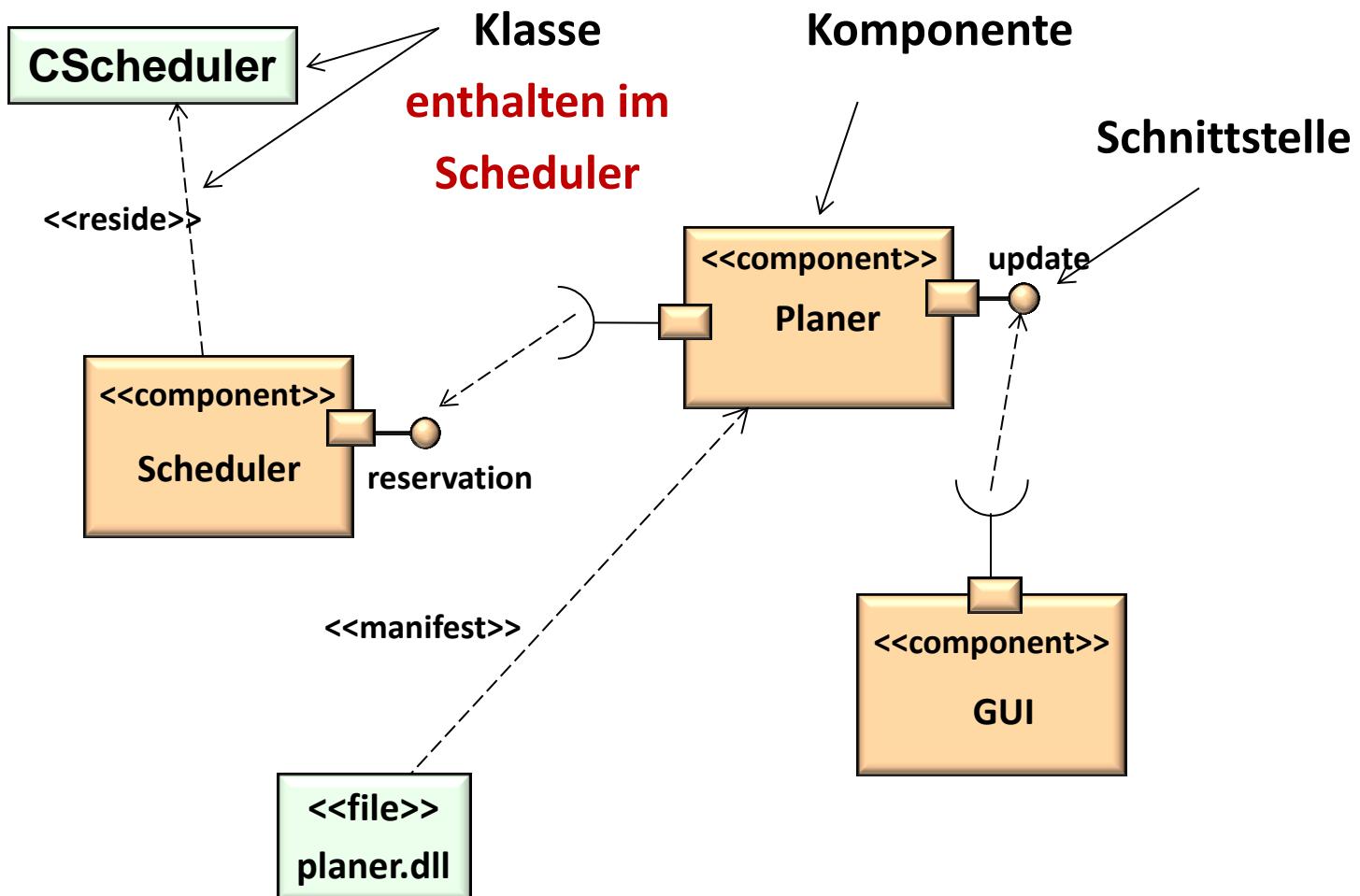
Stellen die Abhängigkeiten zwischen Softwarekomponenten dar (Soure-Code, Binaries, Executables) zur Compile-Zeit, zur Link-Zeit, zur Laufzeit.

- **Verteilungsdiagramme (Deployment Diagrams)**

Sie zeigen die Laufzeitkonfiguration, die Verteilung auf zur Laufzeit physisch vorhandene Objekte (Rechner, Prozessoren, ...) und welche Kommunikationsbeziehungen bestehen.

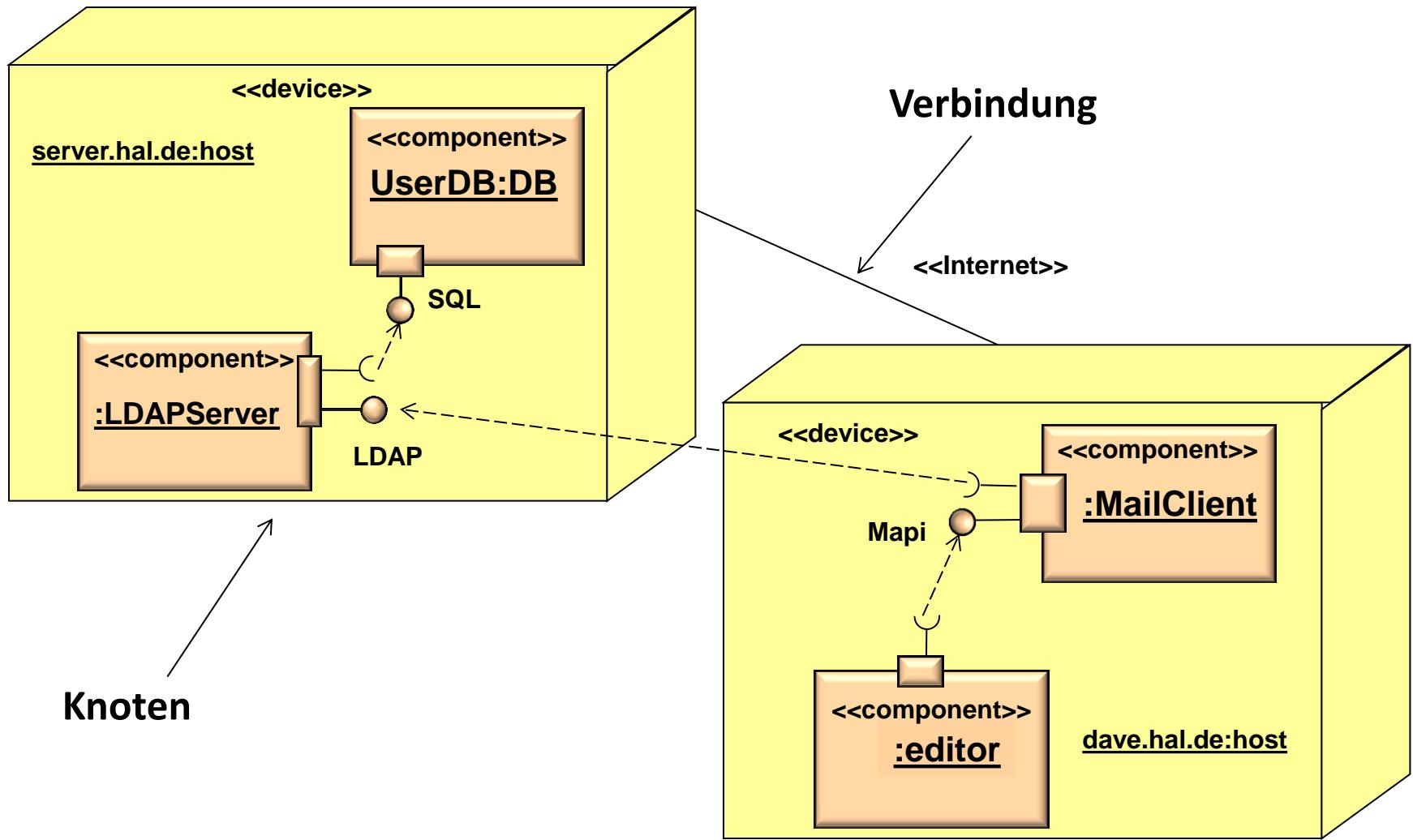


# Komponentendiagramme





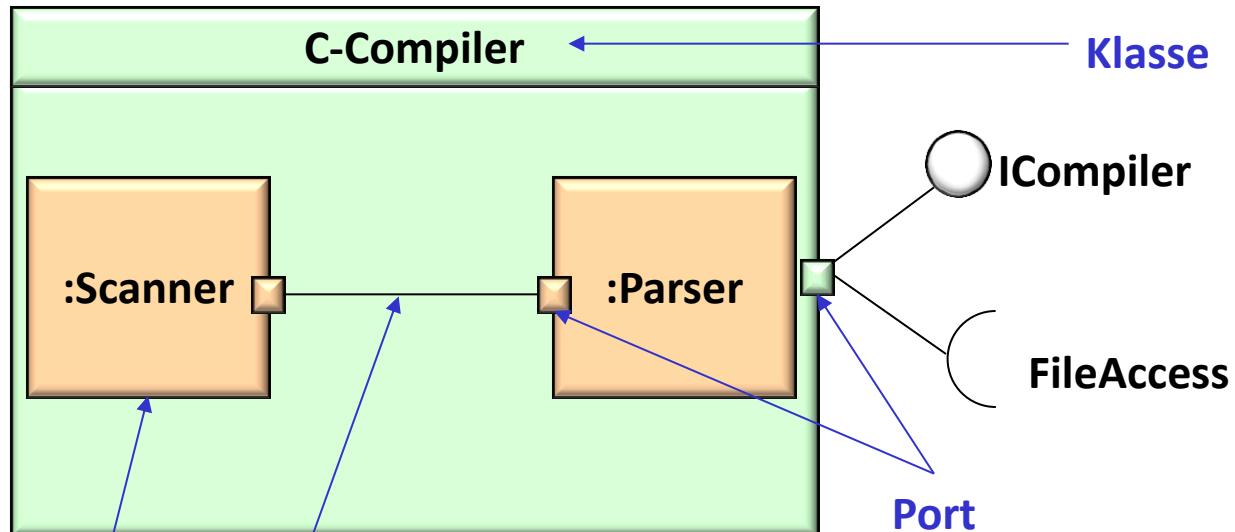
# Verteilungsdiagramme





# Kompositionssstrukturdiagramme

- dienen der Darstellung der Zusammenhänge komplexer Systemarchitekturen
- heben die Stellen hervor, an denen Interaktionen mit Systemteilen stattfinden
- dokumentieren Designentscheidungen



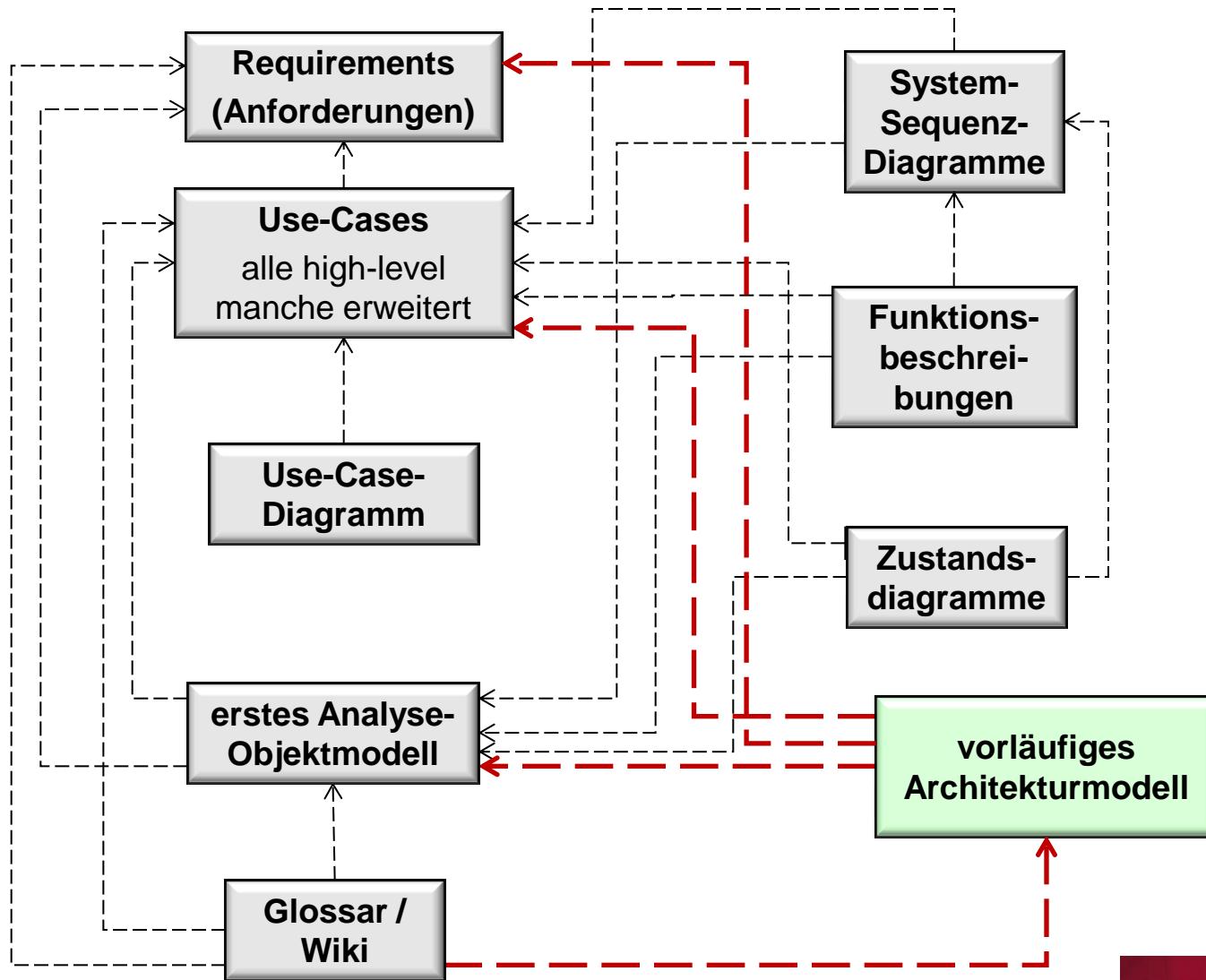
**Connector** veranschaulicht die Kommunikationsbeziehung zwischen Parts

definierte Ausprägung einer Klasse – hier ein Scanner im Kontext des C-Compilers

dedizierter Interaktionspunkt mit der Umgebung, beschrieben durch die Menge der zur Verfügung gestellten und benötigten Operationen.



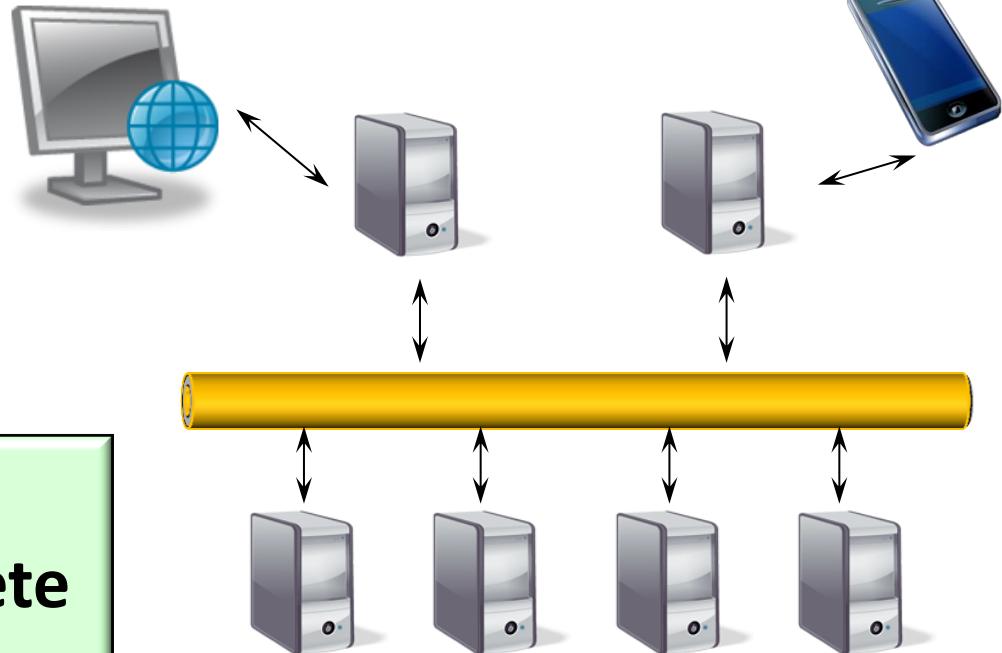
# Die nächsten Designschritte





# Die nächsten Designschritte

- Die logische Architektur entwerfen



**Tiers, Schichten,  
Komponenten und Pakete  
entwerfen, strukturieren,  
verteilen und zuordnen.**



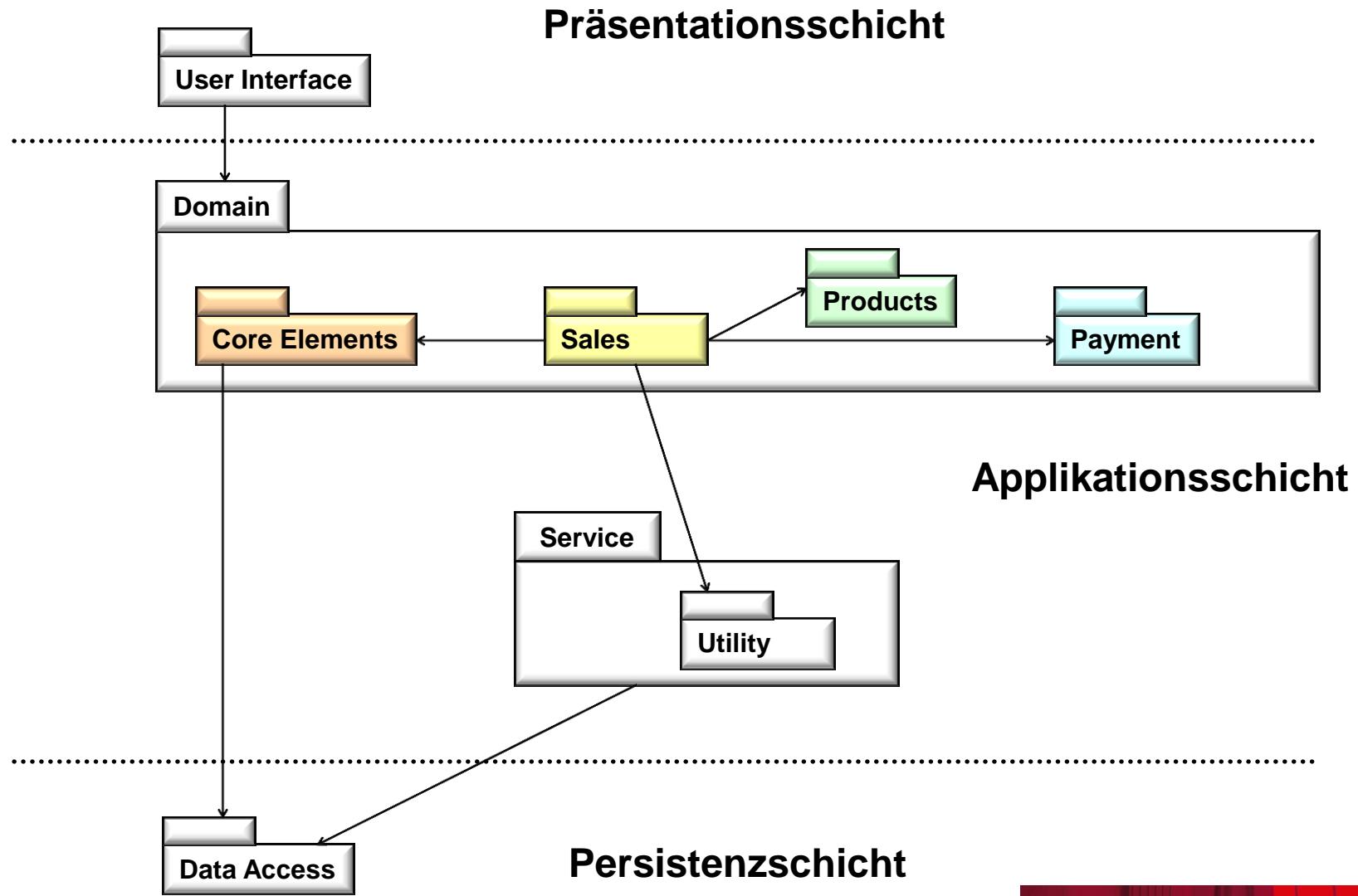
# Das Drei-Schichtenmodell

**Folgende Zerlegung ist typisch für fast alle Applikationen!  
Und zwar unabhängig davon, ob es sich um ein Client-Server- oder lediglich eine Desktop-Anwendung handelt.**

1. **Präsentationsschicht:** ermöglicht den Zugriff von außerhalb über Fenster, Dialoge, Web-Seiten, ...
2. **Applikationsschicht:** realisiert die eigentliche Lösung, man spricht von der Business-, bzw. Geschäftslogik. Sie wird im Allgemeinen weiter aufgeteilt und strukturiert.
  - **domänenspezifische Teile, Utilities, ...**
3. **Persistenzschicht:** realisiert die Anbindung an Drittsysteme, wie Datenbanken, Dateisysteme, Netzwerkdienste, Mail-Systeme usw.



# Die Architektur des Kassensystems





# Erster Schritt: Die Spielregeln

1. Jede Schicht hat ihre eigenen Aufgaben. Sie übernimmt nicht die Aufgaben einer anderen Schicht. Es gibt eine klare Hierarchie.
2. Höhere Schichten nutzen tiefere, **niemals** umgekehrt.

**Jedes Package kann genau einer Schicht (Layer) zugeordnet werden. D.h., alle Klassen, die innerhalb eines Packages benutzt werden, werden in diesem auch definiert, oder sie gehören zu einer tieferen Schicht.**



# Beispiel:

**Im Rahmen der Realisierung eines Spiels muss sichergestellt werden, dass die durchgeführten Aktionen regelkonform sind.**

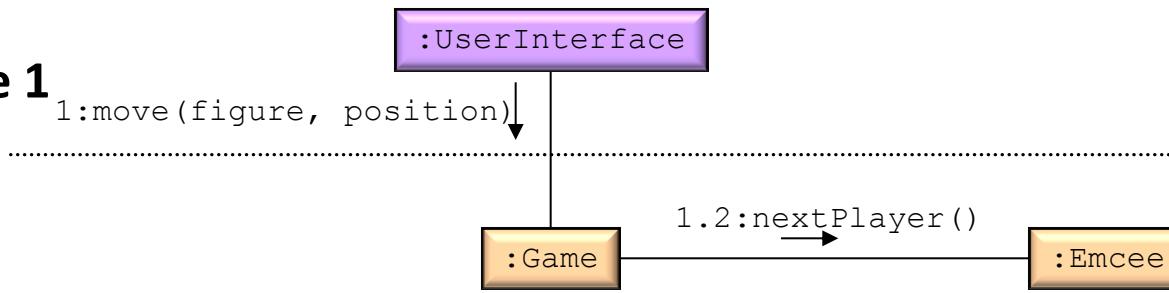
**Welche Schicht überwacht die Spielregeln und trifft die entsprechenden Entscheidungen?**

- die Präsentationsschicht ? **X**
- die Applikationsschicht? **✓**
- die Persistenzschicht? **X**



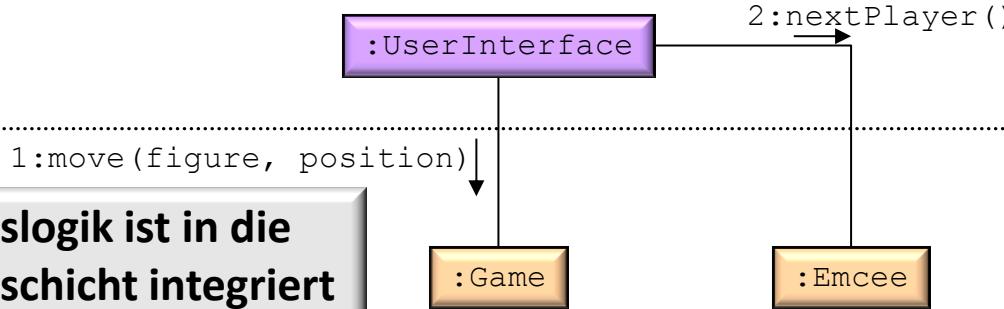
# Prinzipien der Benutzerschnittstelle

## Variante 1



Präsentationsschicht  
Applikationsschicht

## Variante 2



Präsentationsschicht  
Applikationsschicht

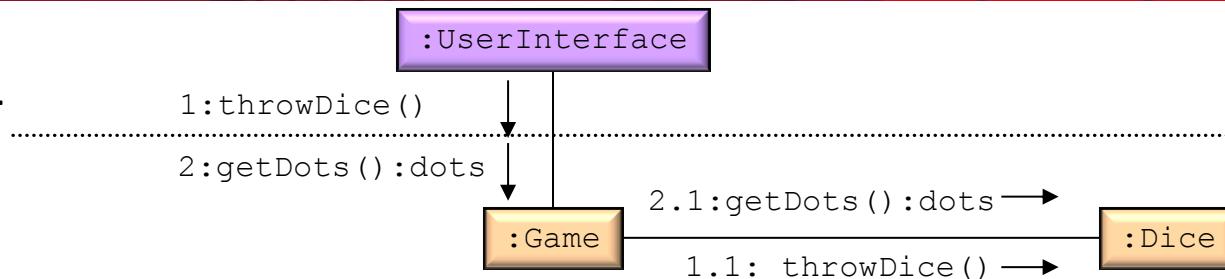
Die Businesslogik ist in die  
Präsentationsschicht integriert  
(Verstoß gegen Punkt 1)

Die Präsentationsschicht ist **nicht** für die Bearbeitung der  
Systemoperation verantwortlich,  
**sondern lediglich für das Anstoßen der Systemoperationen.**



# Visualisierung von Resultaten

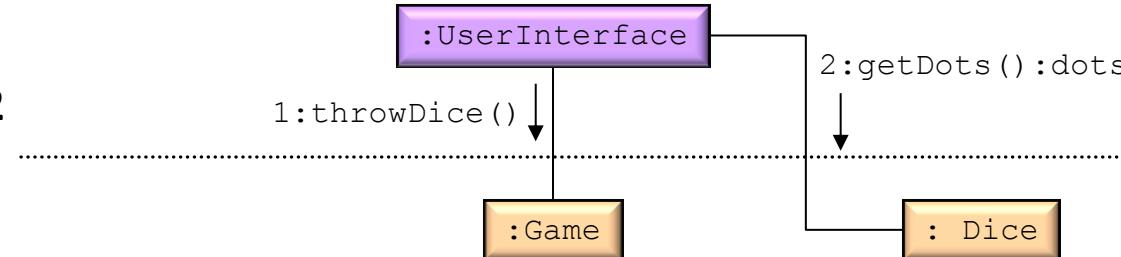
## Variante 1



Präsentationsschicht

Applikationsschicht

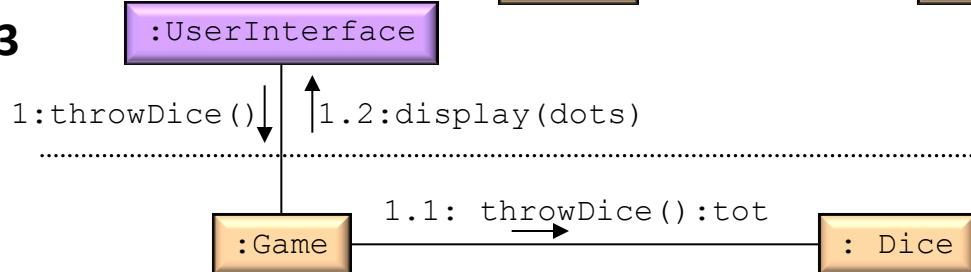
## Variante 2



Präsentationsschicht

Applikationsschicht

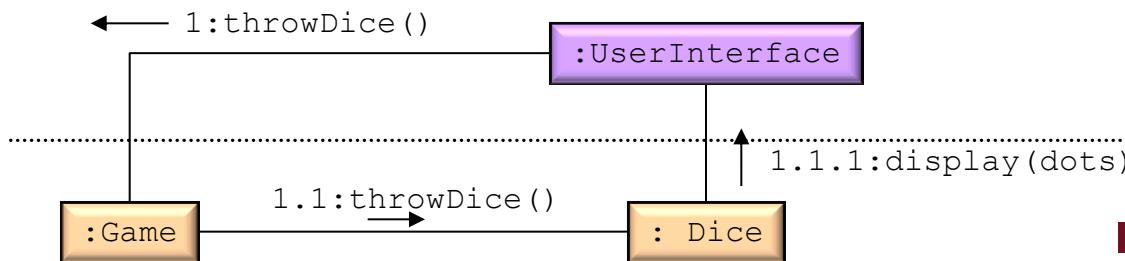
## Variante 3



Präsentationsschicht

Applikationsschicht

## Variante 4



Präsentationsschicht

Applikationsschicht



# Probleme:

**Die Varianten 1 und 2 sind Polling-Lösungen**

- diese sind oft zu lastintensiv und ineffizient
- ungeeignet zur Darstellung von spontanen Änderungen
  - etwa bei Monitoring oder
  - bei Animationen

**Die Varianten 3 und 4 sind Push-from-below-Lösungen.  
Diese sind von der Idee her besser.**

**Es muss jedoch für eine Entkopplung der Ebenen gesorgt werden,  
denn sonst liegt ein Verstoß von Punkt 2 vor:**

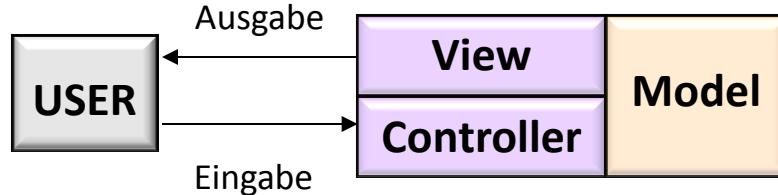
**Eine tiefere Schicht nutzt eine höhere.**



# Lösung: Model-View-Controller

Buschmann, Frank (Pattern-Oriented Software Architecture) Band: 1 A System of Patterns – John Wiley & Sons. 1996

**Das Model-View-Controller Muster unterteilt eine interaktive Anwendung in drei Komponenten.**

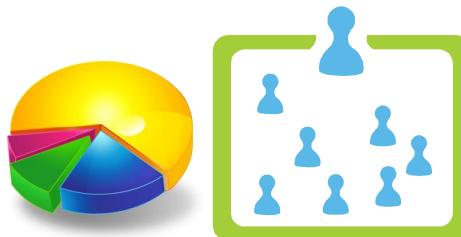




# Lösung: Model-View-Controller

## Probleme und Missverständnisse

- Was ist eine View?
- Was ist ein Model?
- Was ist ein Controller?



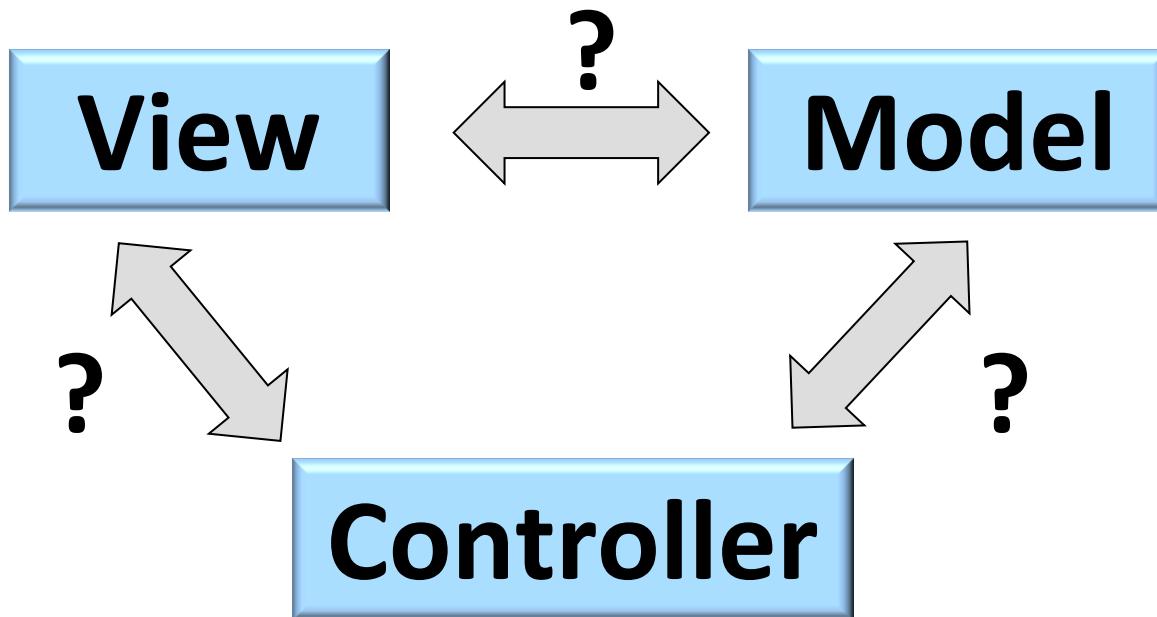
x	y	z
1	2	3
12	24	56
123	245	678





# Lösung: Model-View-Controller

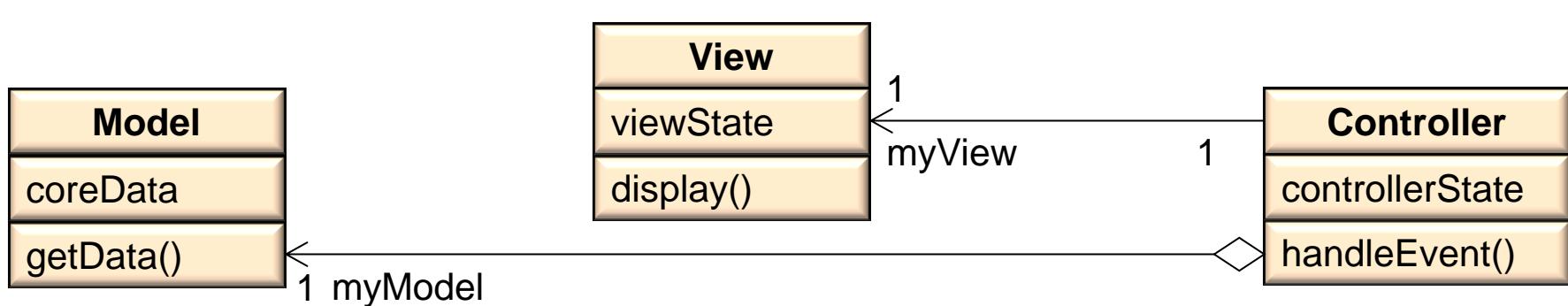
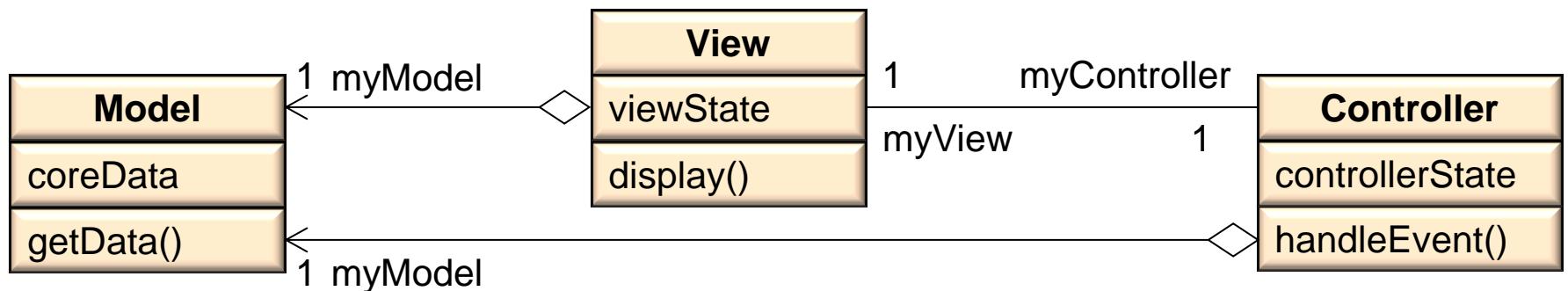
## Wer kennt wen?





# Lösung: Model-View-Controller

## Wer kennt wen?





# Lösung: Model-View-Controller

## Wer gehört zu welcher Schicht?

View

Controller

Präsentationsschicht

Model

Applikationsschicht



# Lösung: Model-View-Controller

## Schlussfolgerung:

Ein Controller verfügt über keine Logik im Sinne domänenspezifischen Wissens.

Der Controller übernimmt nicht die Ablaufsteuerung im Use-Case.

Das Model beinhaltet mehr als nur Daten, sondern auch das domänenspezifische Wissen.

Im Model sind Daten und Logik vereint.



# Lösung: Model-View-Controller

**“Spring MVC helps in building flexible and loosely coupled web applications.**

The Model-view-controller design pattern helps in **separating** the **business logic**, **presentation logic** and **navigation logic**.

**Models** are responsible for encapsulating the application data.

The **Views** render response to the user with the help of the model object.

**Controllers** are responsible for receiving the request from the user and calling the back-end services.”

[M. Muthuraman. Spring MVC Framework Tutorial. Dzone, 2012.

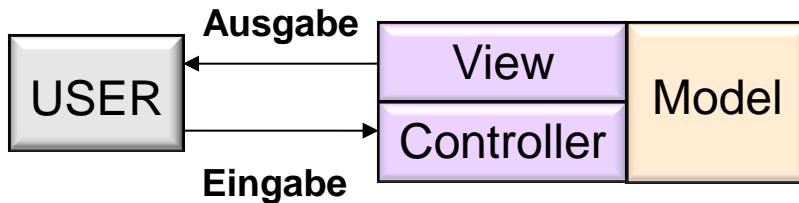
<http://www.dzone.com/tutorials/java/spring/spring-mvc-tutorial-1.html>]



# Lösung: Model-View-Controller

Buschmann, Frank (Pattern-Oriented Software Architecture) Band: 1 A System of Patterns – John Wiley & Sons. 1996

**Das Model-View-Controller Muster unterteilt eine interaktive Anwendung in drei Komponenten.**



- **Model** Daten und ihre Verarbeitung, die Kernfunktionalität
  - **View** Präsentation der Daten
  - **Controller** Benutzereingaben
- } **Bedienschnittstelle  
(Präsentationsschicht)**

**Das MVC-Muster sorgt für eine Entkoppelung der verschiedenen Teile der Applikation. Dies bedeutet mehr Flexibilität.**

**Ein Benachrichtigungsmechanismus über Änderungen sichert die Konsistenz**



# Model-View-Controller (IDEE)

**Die Grundidee ist die Aufspaltung der Anwendung in drei Bereiche:  
Verarbeitung, Ausgabe und Eingabe**

- **Verarbeitungskomponente (Model):**

Sie ist unabhängig von den Anzeige und Eingabemöglichkeiten und stellt die Kernfunktionalität bereit

- **Ausgabekomponente (View):**

Sie präsentiert dem Anwender die Daten. Diese erhält sie vom Model. (Die Anzahl der Views kann variieren)

- **Eingabekomponente (Controller):**

Für jede Ansicht gibt es in der Regel eine eigene Eingabekomponente. Sie empfängt die Eingaben (Mausbewegung, Klicks, Tasteneingaben) und wandelt sie in Dienstanforderungen für das Model (oder die Ansicht) um. Der Anwender interagiert ausschließlich über die Kontrollkomponenten.

**Dies ermöglicht die Definition von mehreren Ansichten mit entsprechenden Controllern für ein und dasselbe Modell.**



# Wie werden Änderungen propagiert?

Wenn ein Anwender das Modell mit Hilfe eines Controllers ändert, dann sollten alle Views über diese Änderung informiert werden.

- **Observer-Pattern:**

Im Observer-Pattern wird eine 1-zu-n-Beziehung zwischen Objekten definiert, so dass die Änderungen des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

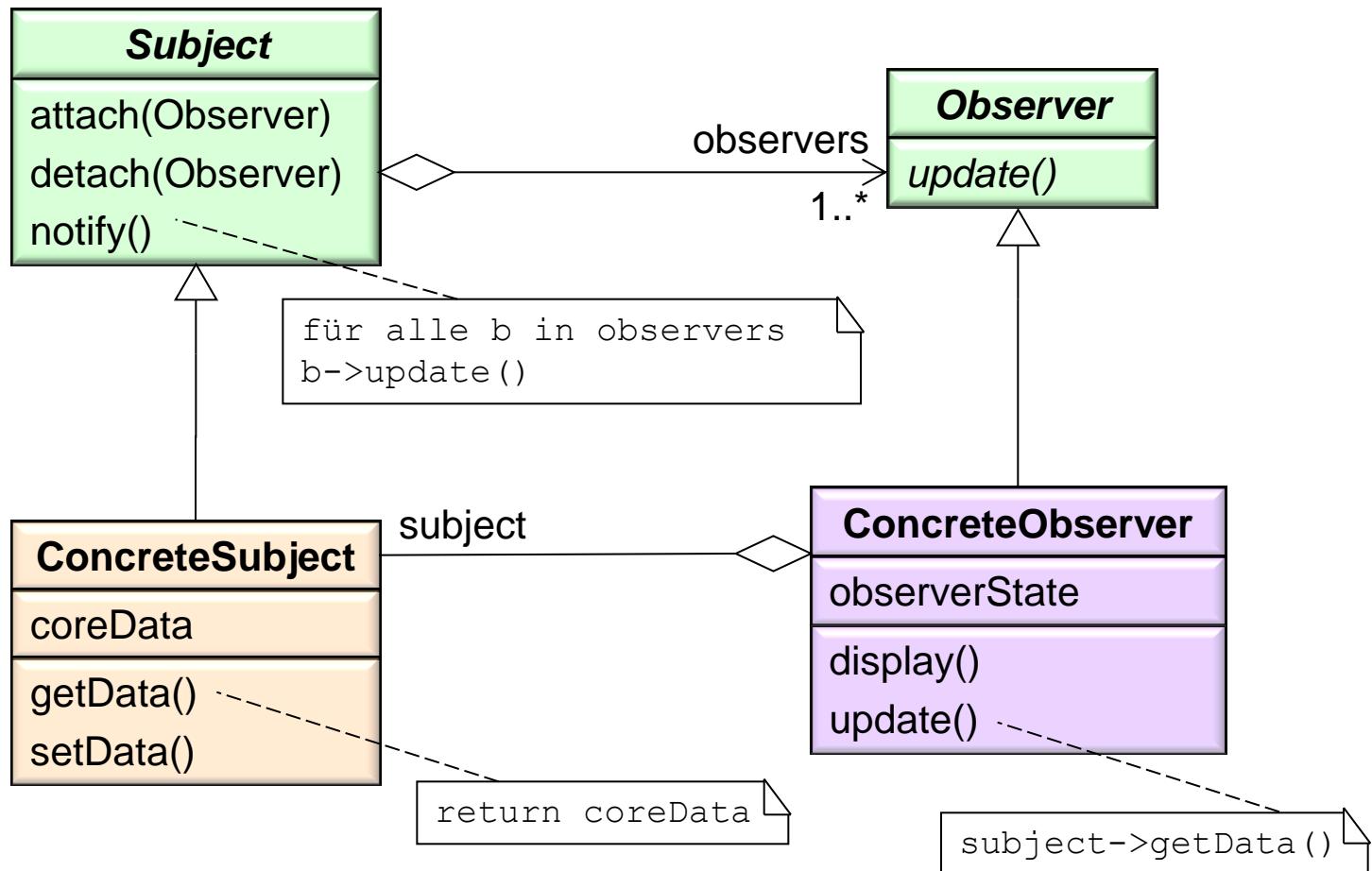
- **Anwendbarkeit:**

- Wenn eine Abstraktion zwei Aspekte besitzt, von denen der eine vom anderen abhängt. (Model, View)
- Wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt.
- Wenn ein Objekt andere Objekte informieren soll, ohne Annahmen darüber zu treffen, wer diese Objekte sind.

Dies passt ideal auf das Szenario Model-View-Controller

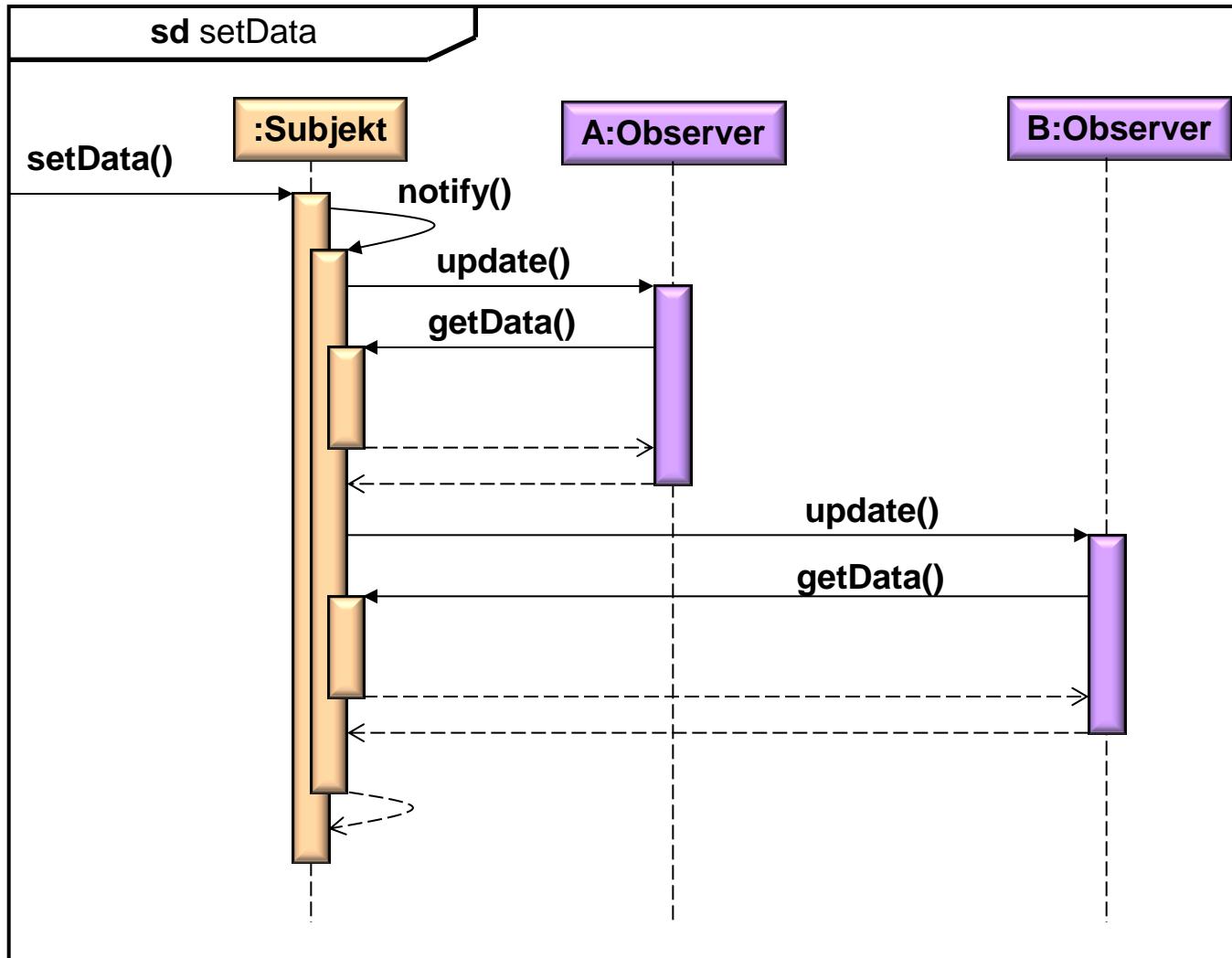


# Struktur des Observer-Patterns



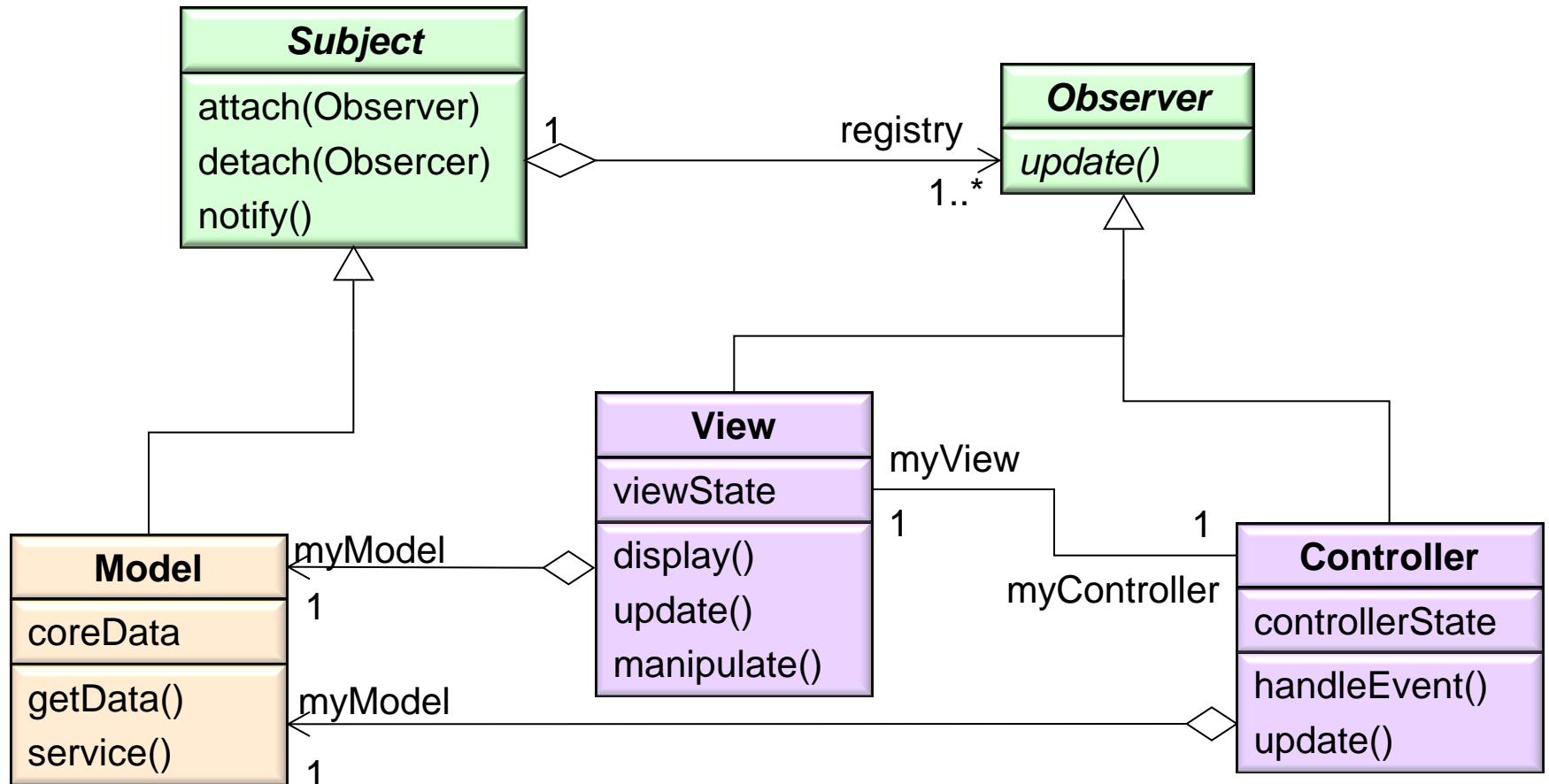


# Dynamische Aspekte im Observer-Pattern





# Struktur des Observer-Patterns im MVC-Pattern





# Struktur (Zusammenfassung)

- **Model (Verarbeitungskomponente):**

Verwaltet die zentralen Daten (*coreData*) und bietet Operationen an, welche die für die Anwendung spezifischen Verarbeitungen vornehmen (*service()*). Die Eingabekomponente (*controller*) rufen diese Funktionen auf. Des Weiteren stellt es Funktionen bereit, die den Zugriff auf die Daten ermöglichen (*getData()*). Funktionalität zum Verwalten und Benachrichtigen der Beobachter erbt es von der Oberklasse Subjekt.

- **View (Ausgabekomponente):**

Präsentieren dem Anwender die Informationen (*display()*). Als Beobachter registrieren sie sich beim Modell (*attach()*) und werden über Änderungen via *update()* informiert. Ermitteln dann die relevanten Daten (*getData()*) und stellen die Änderung dar.

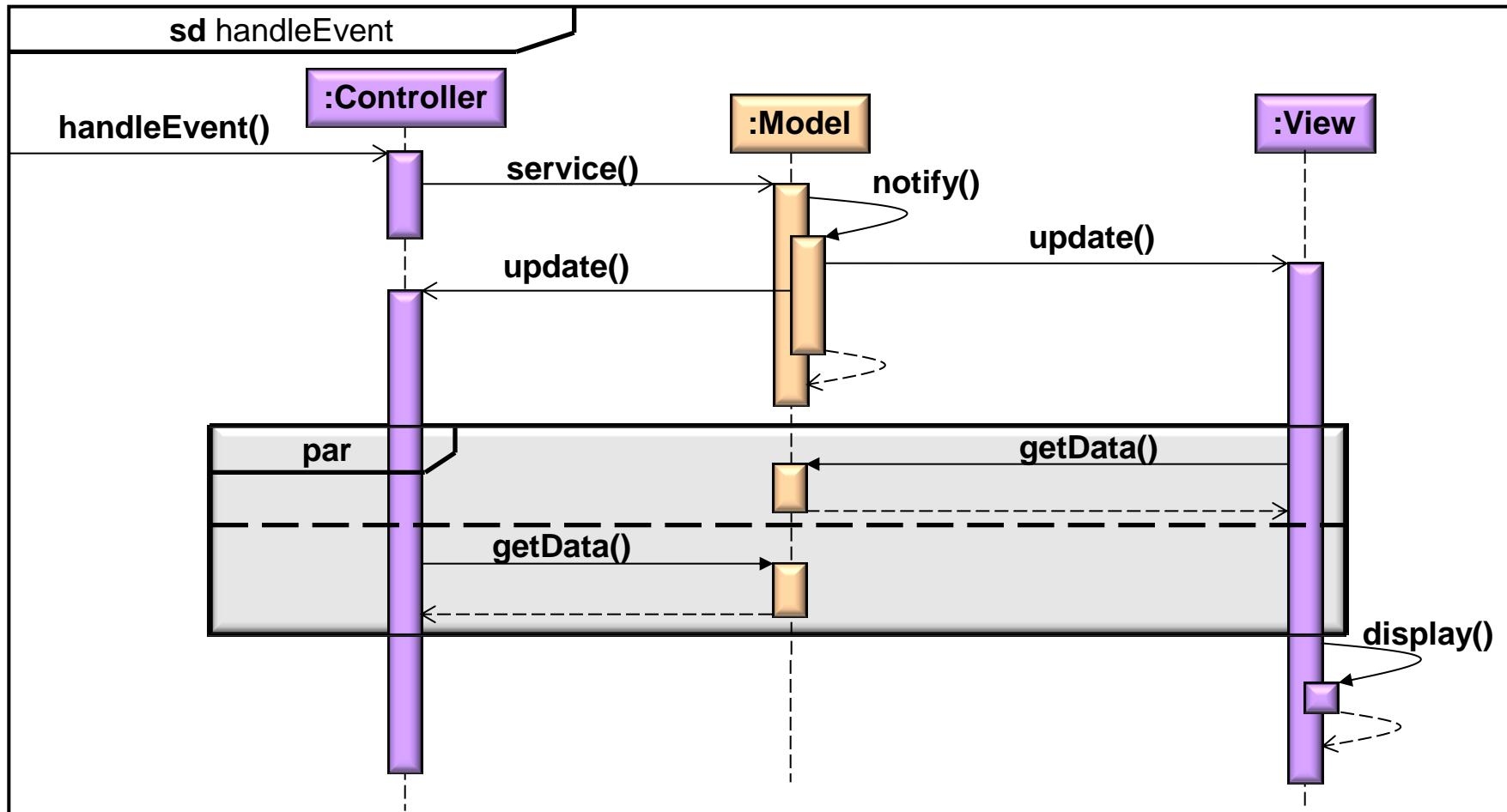
- **Controller (Eingabekomponente):**

Akzeptiert die Bedieneingaben (*handleEvent*), und leitet Sie entweder ans Modell weiter (*service()*) oder manipuliert die Ansicht (*manipulate()*). Als Beobachter des Modells (*update()*, *getData()*) reagiert er durch enable/disable von Steuermöglichkeiten oder Menüeinträgen.



# Dynamische Aspekte im Model-View-Controller-Pattern

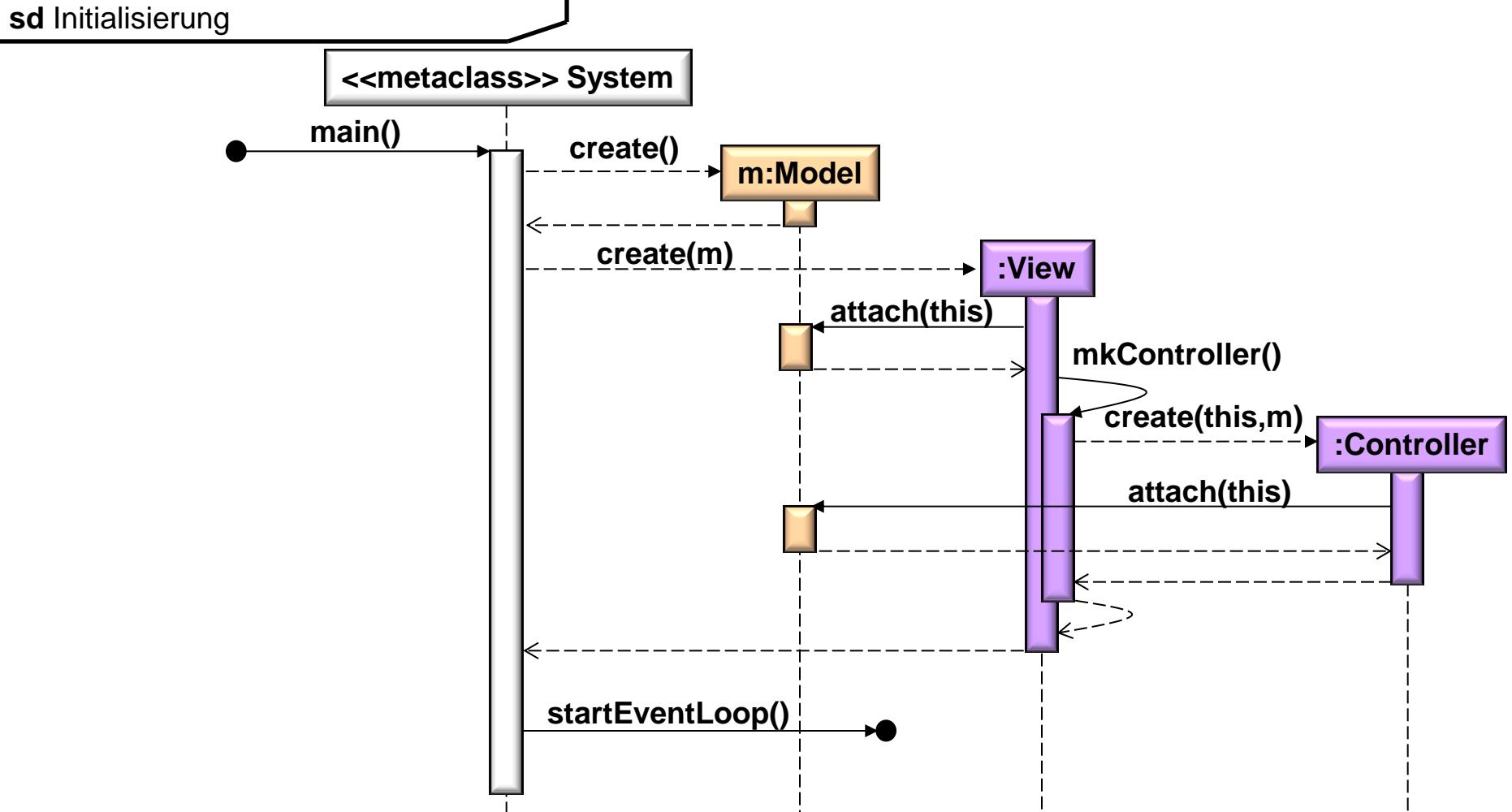
- Benutzereingaben werden propagiert





# Dynamische Aspekte im Model-View-Controller-Pattern

- Das System wird initialisiert





# Implementierung

## 1. Abtrennen der Mensch-Maschine-Schnittstelle

Die Kernfunktionalität wird im Modell realisiert. In der Oberfläche werden lediglich die Funktionen angestoßen und die Ergebnisse dargestellt. (Vollständiges Design ohne GUI). Danach wird geklärt welche Funktionalität dem Benutzer angeboten wird und wie die Schnittstellenkomponenten im Modell aussehen.

## 2. Benachrichtigungsmechanismus implementieren (Beobachtermuster umsetzen)

Container sowie An- und Abmeldemöglichkeiten für die Beobachter. Eventuell ist es angebracht die Beobachtung zu individualisieren. Nicht jeder wird über alles informiert.

## 3. Views entwerfen und implementieren

Dies beinhaltet neben der Display-Funktion, die die Inhalte anfordern die Realisierung der Update-Funktion.

Zusätzliche Parameter in der Update-Funktion ermöglichen Optimierungen:

- ist überhaupt eine Aktualisierung notwendig
- sollte die Aktualisierung verzögert werden, da noch weitere Updates kommen.



# Implementierung

## 4. Die Controller werden entworfen und implementiert

Für jede View werden die Bedienmöglichkeiten festgelegt, die der Anwender ausführen kann. Ein Controller erhält und interpretiert die Eingaben und bildet sie auf entsprechende System-Funktionen ab (service()). Die Initialisierung des Controllers verbindet ihn mit Modell und View.

## 5. Die Beziehungen zwischen View und Controller

Jede View erzeugt ihren Controller. Für eine Hierarchie von Views ist es angebracht die Erzeugung der Controller z. B. über Fabrikmethoden zu organisieren.

## 6. Initialisierung der gesamten MVC-Struktur

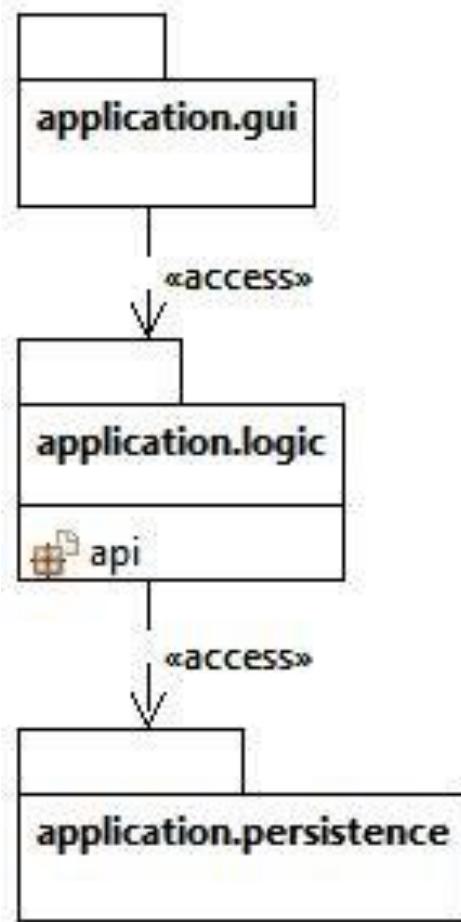
Zuerst wird das Modell erzeugt, danach die Views, diese registrieren sich beim Modell und erzeugen ihre Controller. Ist diese Initialisierung abgeschlossen, wird die Hauptschleife gestartet.

**Eine Initialisierungsroutine meldet die Views an und erzeugt die Controller.**



# Struktur (Implementierung)

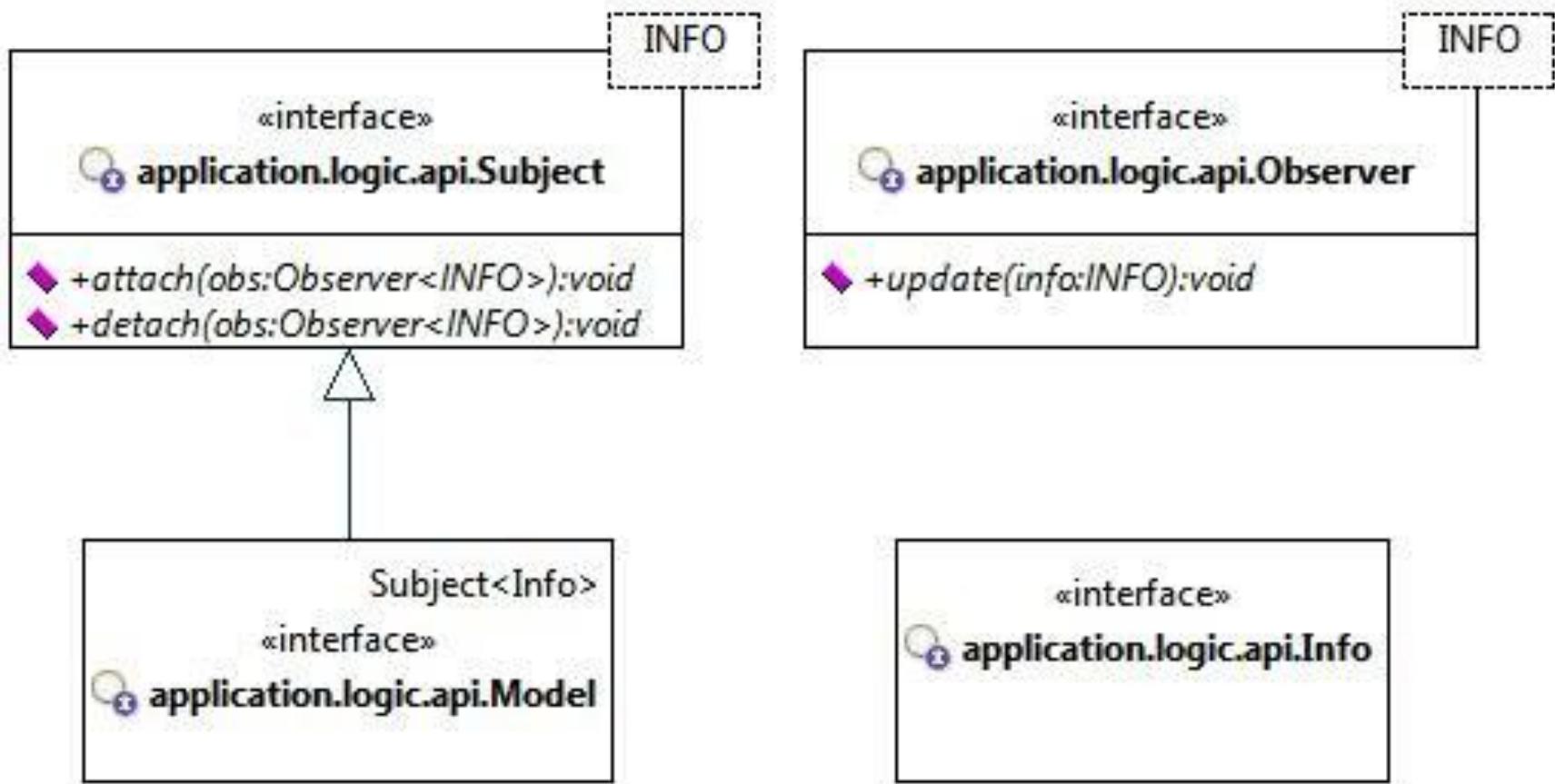
- **Erste Architektur**





# Struktur (Implementierung)

- **Interfaces (logic.api)**





# Struktur (Implementierung)

- **Interfaces (logic.api)**

```
public interface Subject<INFO> {  
  
    public void attach(Observer<INFO> obs);  
    public void detach(Observer<INFO> obs);  
}
```

```
public interface Observer<INFO> {  
  
    public void update(INFO info);  
}
```

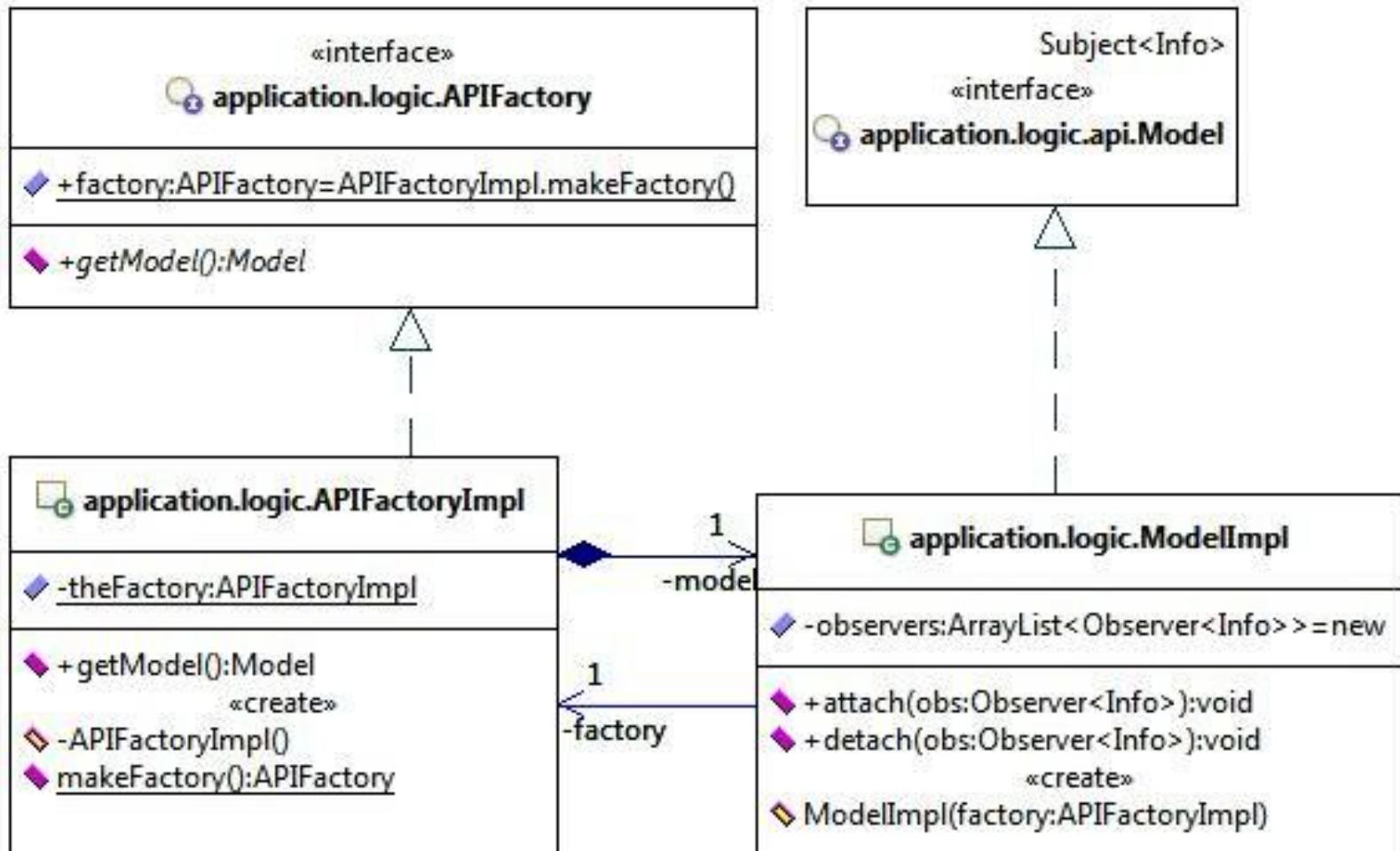
```
public interface Model extends Subject<Info> {}
```

```
public interface Info {}
```



# Struktur (Implementierung)

- Schnittstelle zur Logik (logic)





# Struktur (Implementierung)

- **Schnittstelle zur Logik (logic)**

```
import application.logic.api.Model;

class APIFactoryImpl implements APIFactory {
    private static APIFactoryImpl theFactory;
    private ModelImpl model;

    private APIFactoryImpl() {}

    static APIFactory makeFactory() {
        if (APIFactoryImpl.theFactory == null)
            APIFactoryImpl.theFactory = new APIFactoryImpl();
        return APIFactoryImpl.theFactory;
    }

    public Model getModel() {
        if (this.model == null)
            this.model = new ModelImpl(this);
        return model;
    }
}
```

```
import application.logic.api.Model;

public interface APIFactory {
    public APIFactory factory =
        APIFactoryImpl.makeFactory();

    public Model getModel();
}
```



# Struktur (Implementierung)

- **Schnittstelle zur Logik (logic)**

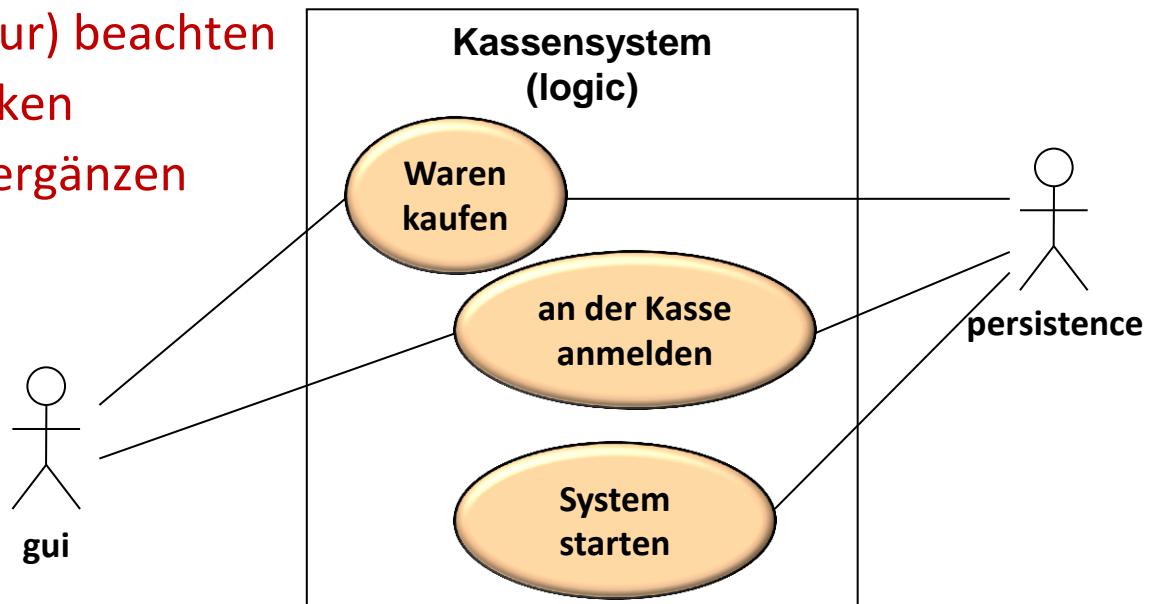
```
class ModelImpl implements Model {  
  
    private ArrayList<Observer<Info>> observers = new ArrayList<Observer<Info>>();  
    private Info currentInfo;  
    private APIFactoryImpl factory;  
  
    ModelImpl(APIFactoryImpl factory) {  
        this.factory = factory;  
    }  
  
    public void attach(Observer<Info> obs) {  
        this.observers.add(obs);  
        obs.update(this.currentInfo);  
    }  
  
    public void detach(Observer<Info> obs) {  
        this.observers.remove(obs);  
    }  
}
```



# Die nächsten Designschritte

(nach Craig Larman: Applying UML and patterns)

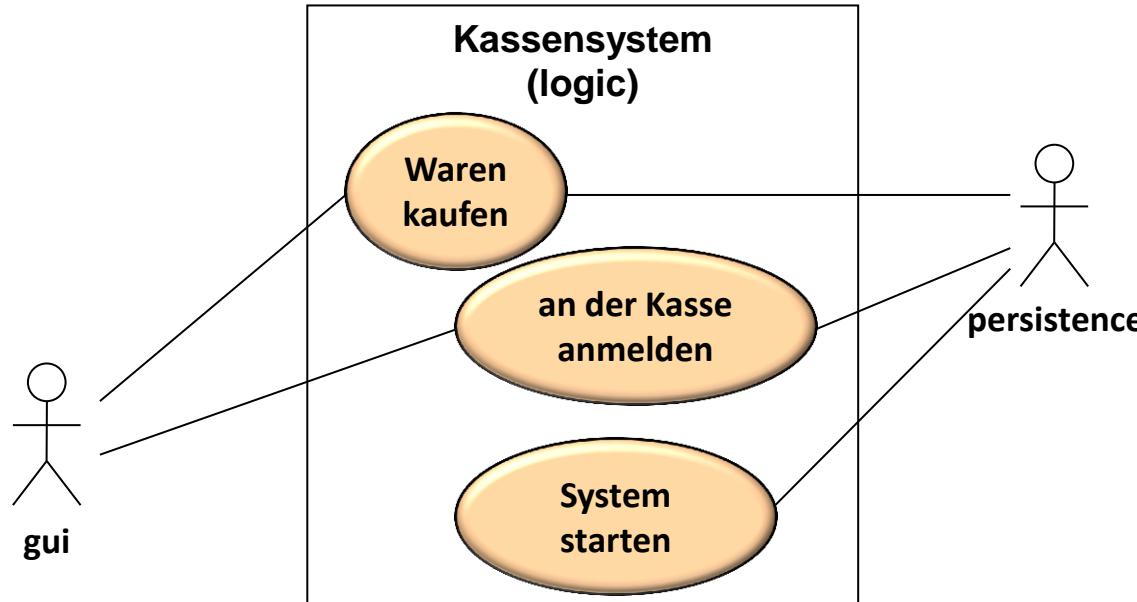
- Für die umzusetzenden Use-Cases werden auf Basis der Analyseergebnisse Mockups erstellt
- Aus den Use-Cases der Analyse werden konkrete (System-) Use-Cases als Grundlage für das Design
  - Input-Output-Terminologie verwenden
  - Subsysteme (Architektur) beachten
  - über Akteure nachdenken
  - technische Use-Cases ergänzen
  - ...





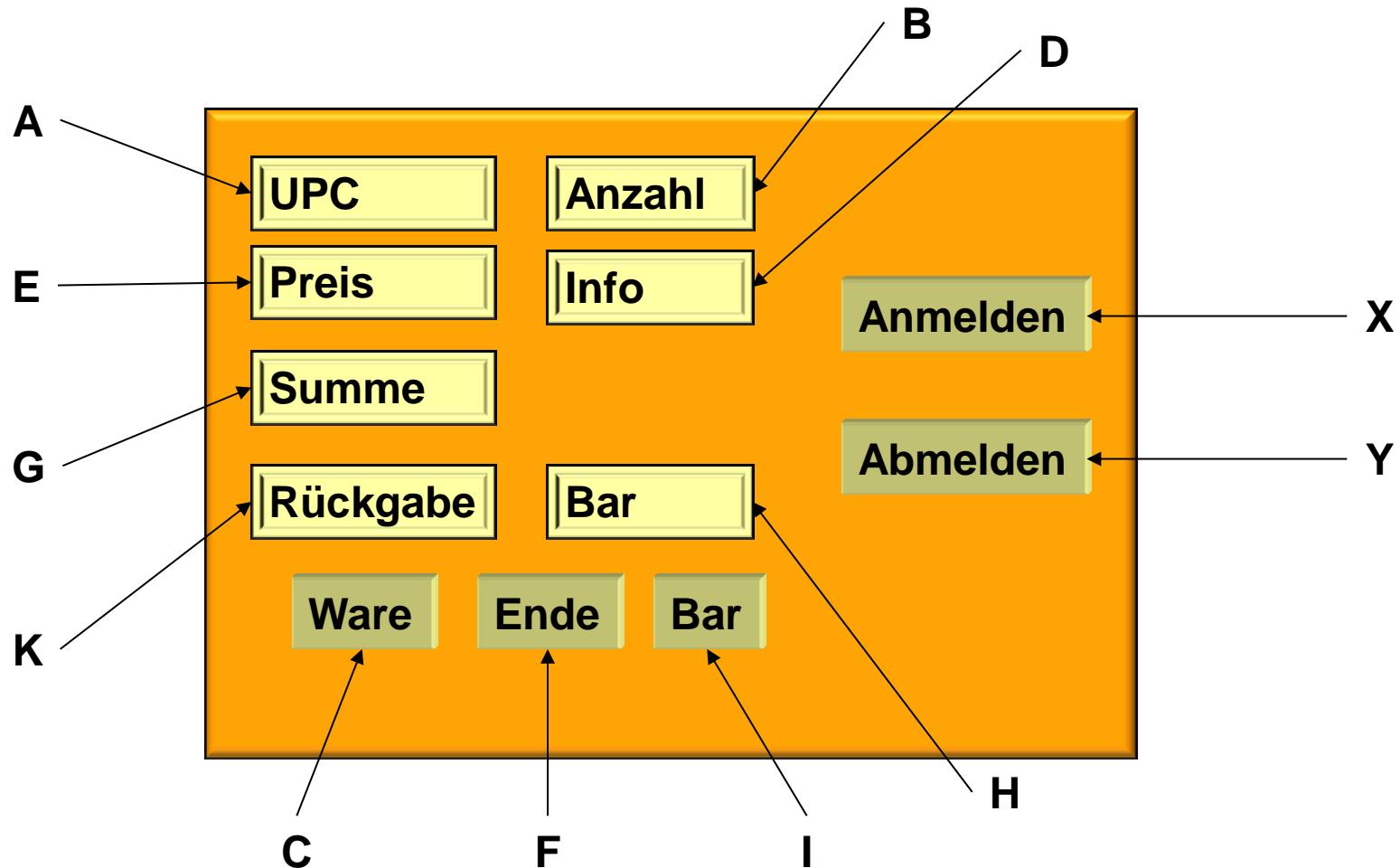
# Die nächsten Designschritte

- Im Rahmen der Designphase werden die Use-Case iterativ entworfen und realisiert
- Design und Implementierung gehen Hand in Hand



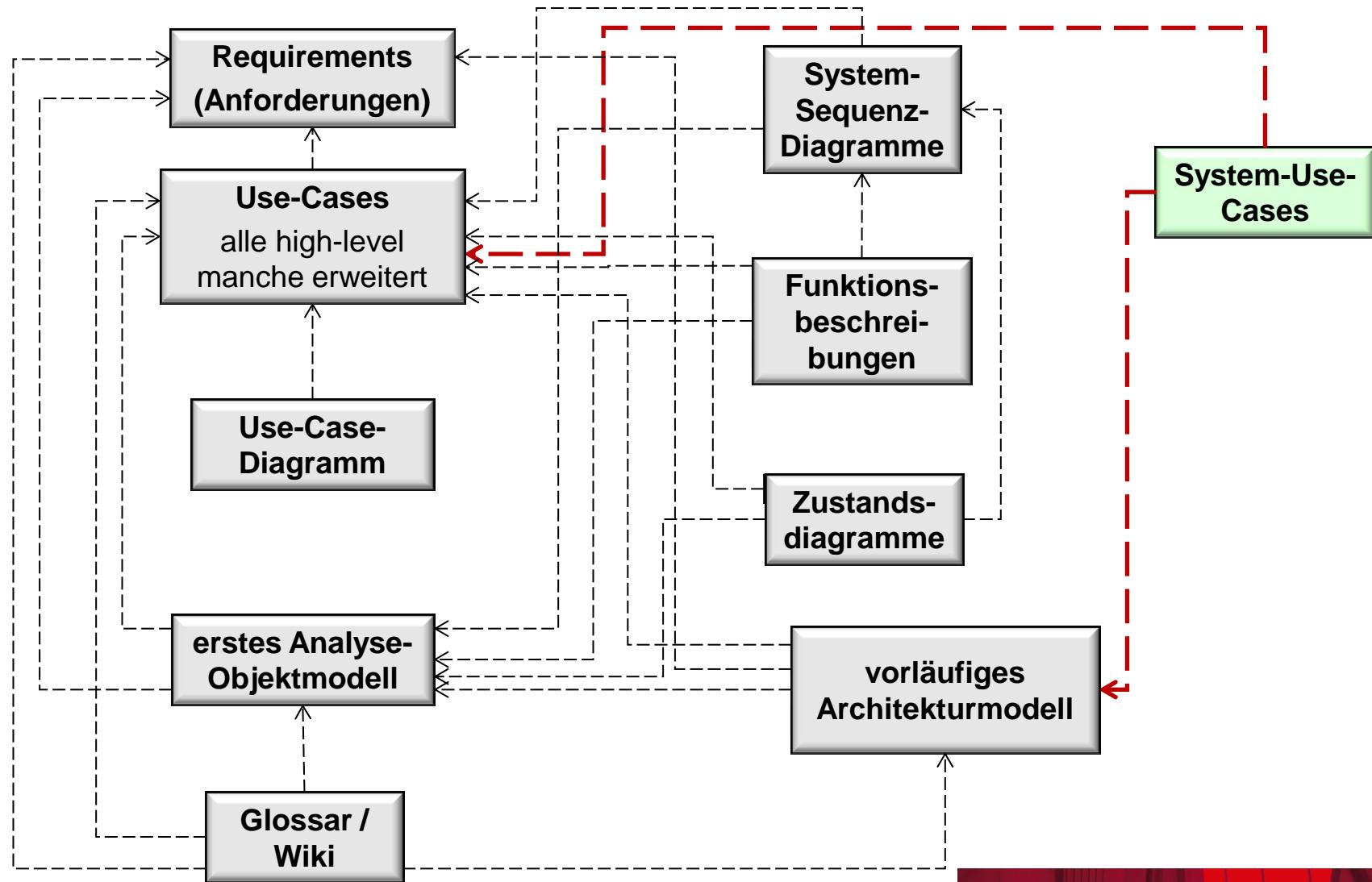


# Mockups erstellen





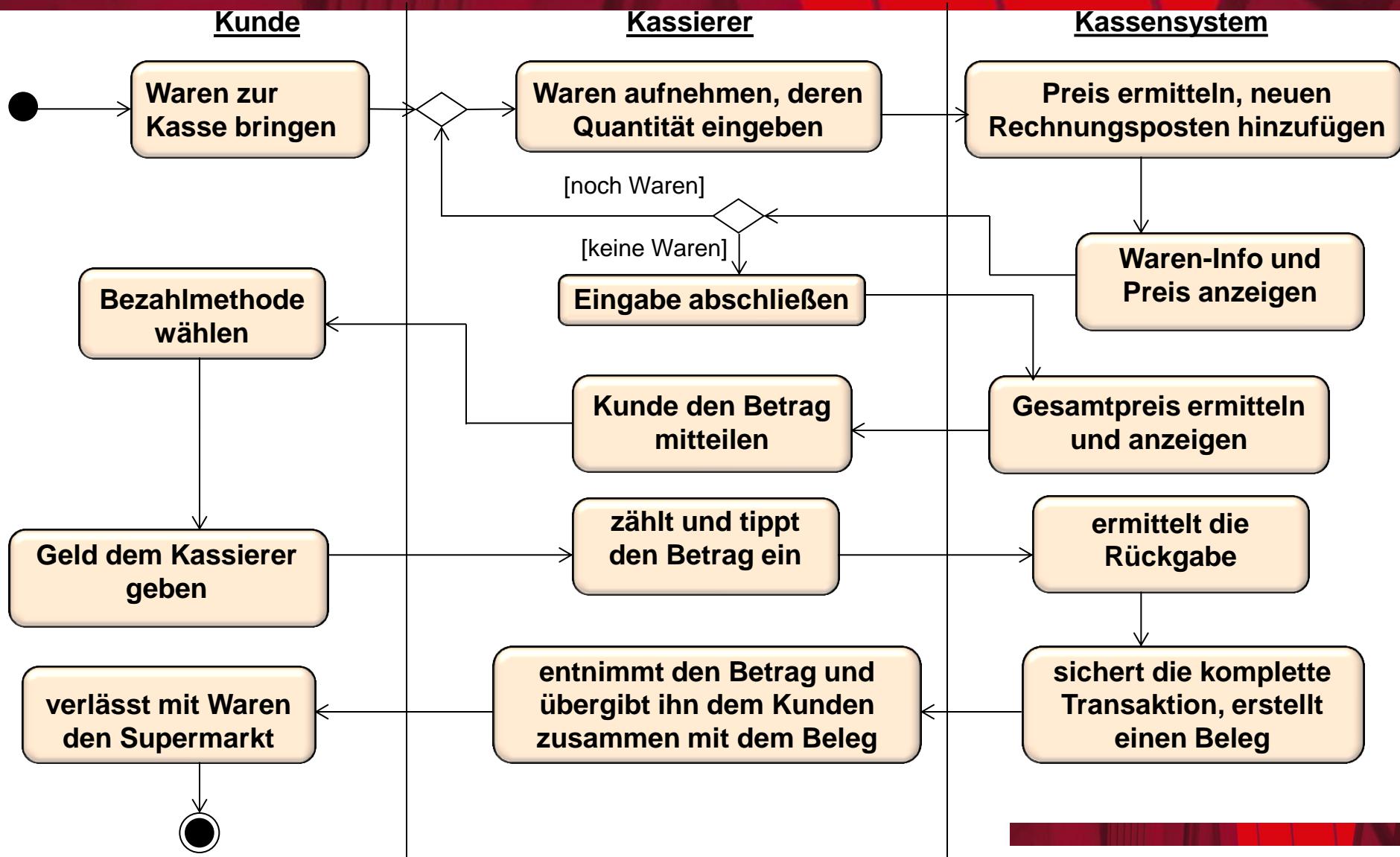
# Die nächsten Designschritte





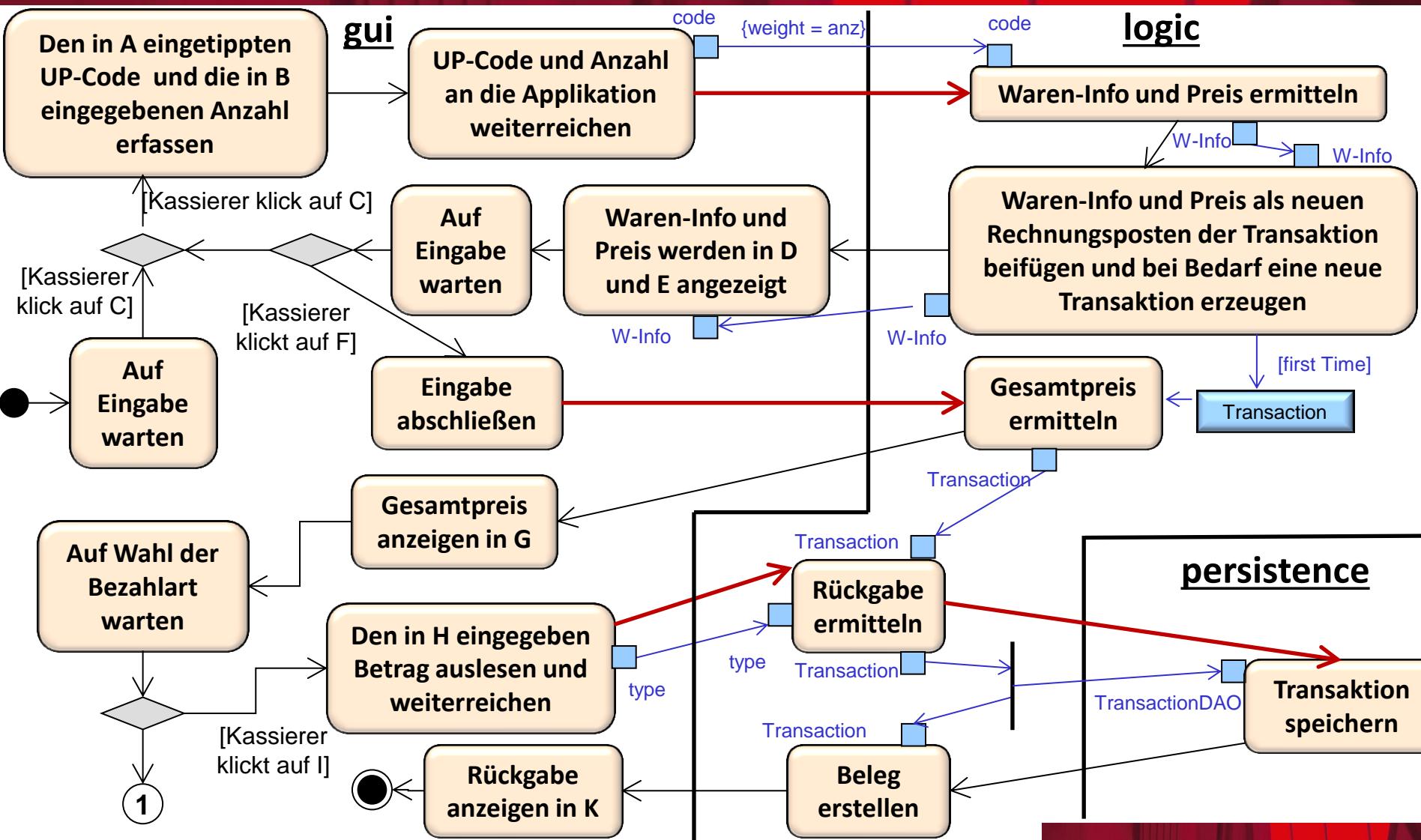
# Beispiel

# Registrierkasse (Analyse)



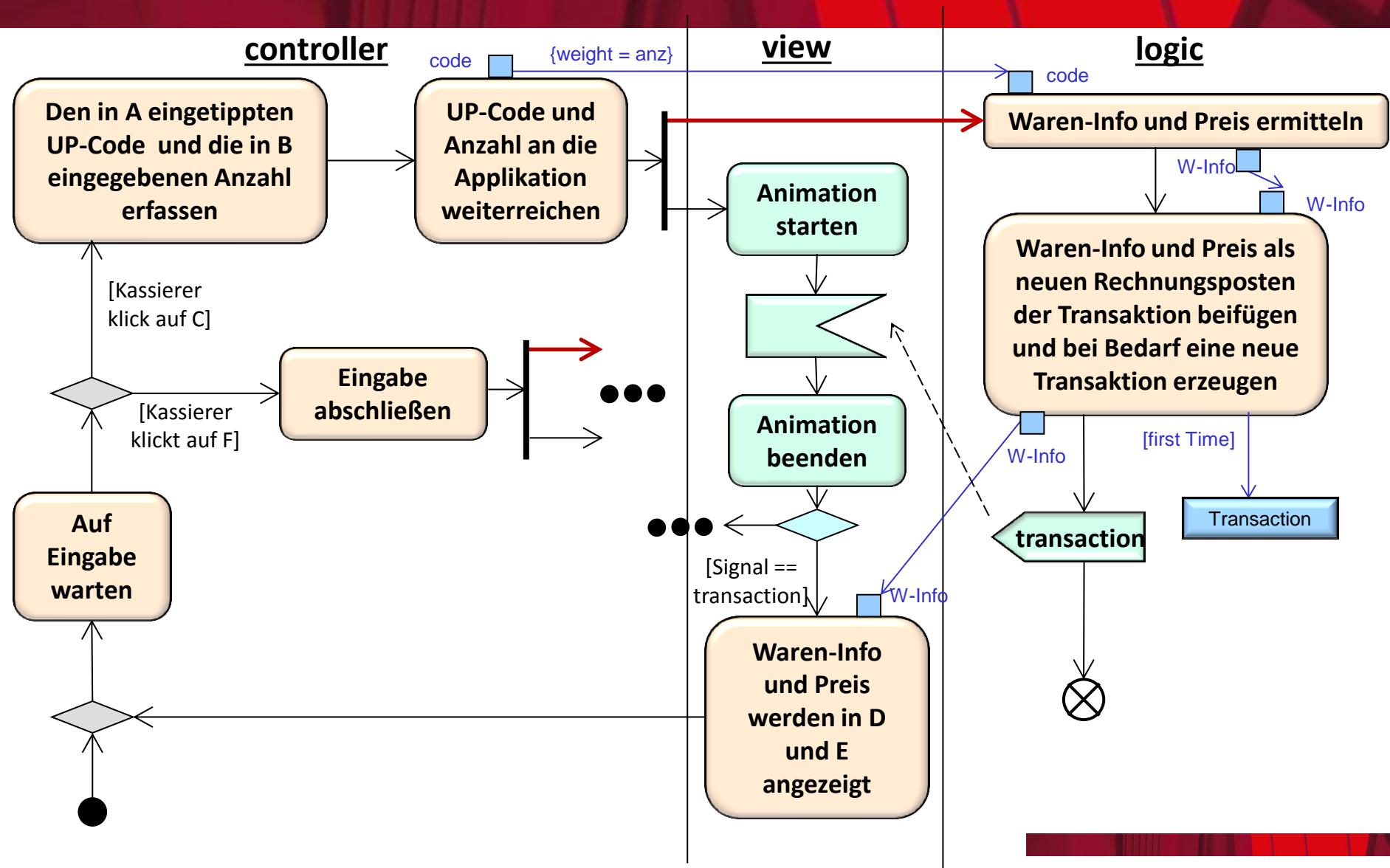


# Der Use-Case „Waren kaufen“





# MVC bereits im Activity-Diagramm





# System-Sequenz-Diagramme

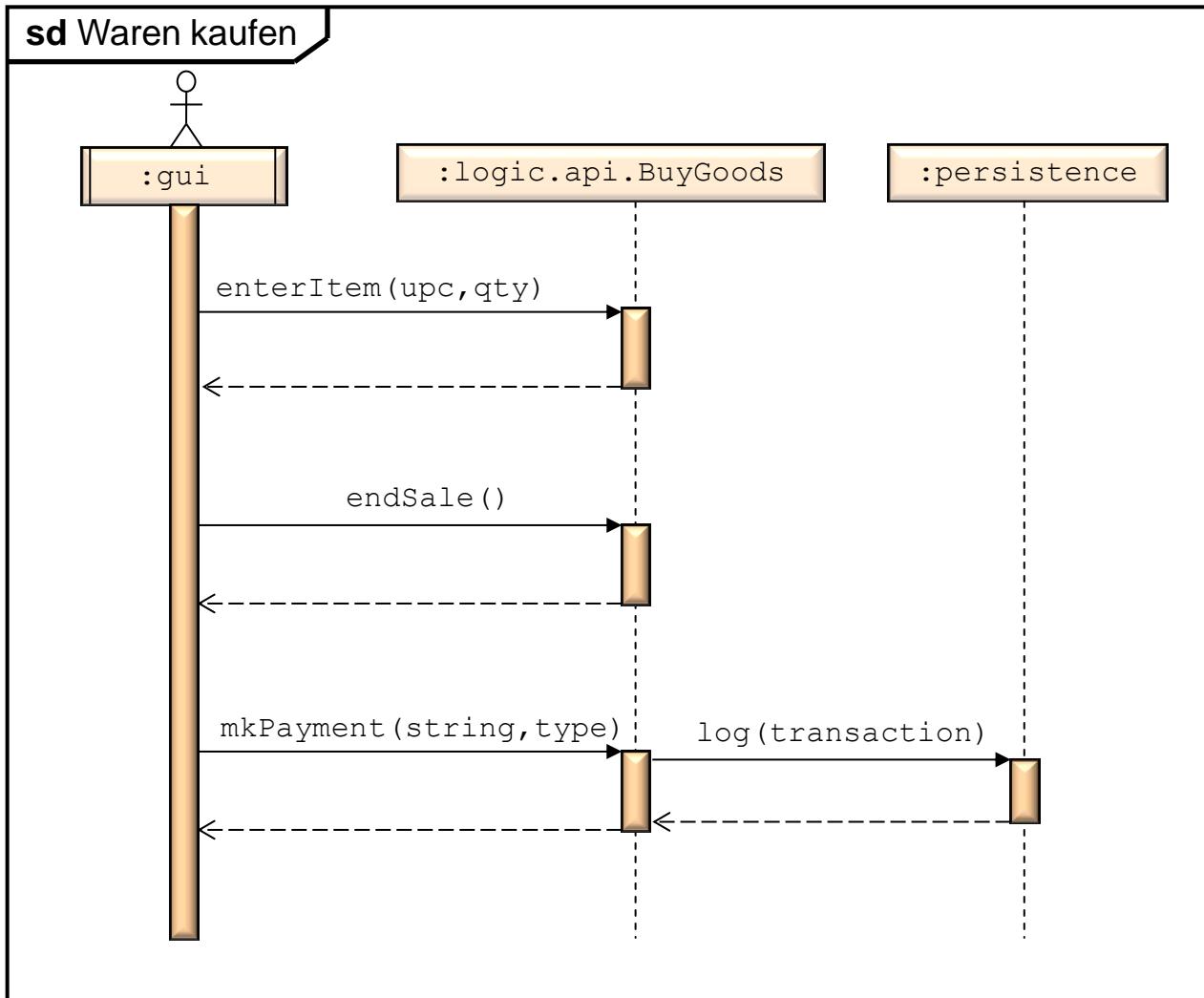
**System-Ereignisse und System-Operationen sind zu präzisieren**

- für jeden Use-Case muss das System-Sequenz-Diagramm überarbeitet werden  
**(Parameter, Ausnahmen, ggf. Rückgabewerte sind festzulegen)**
- für jede System-Operation muss die Funktionsbeschreibung überarbeitet werden

**Die Schnittstelle  
(die zu realisierende Aufgabe)  
wird festgelegt.**



# System-Sequenz-Diagramme





# Funktionsbeschreibung

**Name:**

`enterItem(upc: Integer, qty: Integer)`

**Verantwortlichkeit:**

registriert den Kauf der Ware und fügt ihn der Transaktion hinzu; zeigt Preis und Beschreibung an

**Referenzen:**

Use-Case: Ware kaufen

Funktionen:  $F_i - F_j$

**Bemerkungen:**

.....

**Ausnahmen:**

falls UPC unbekannt, ist ein Fehler zu melden

**Vorbedingungen:**

UPCs aller Waren sind bekannt, ...

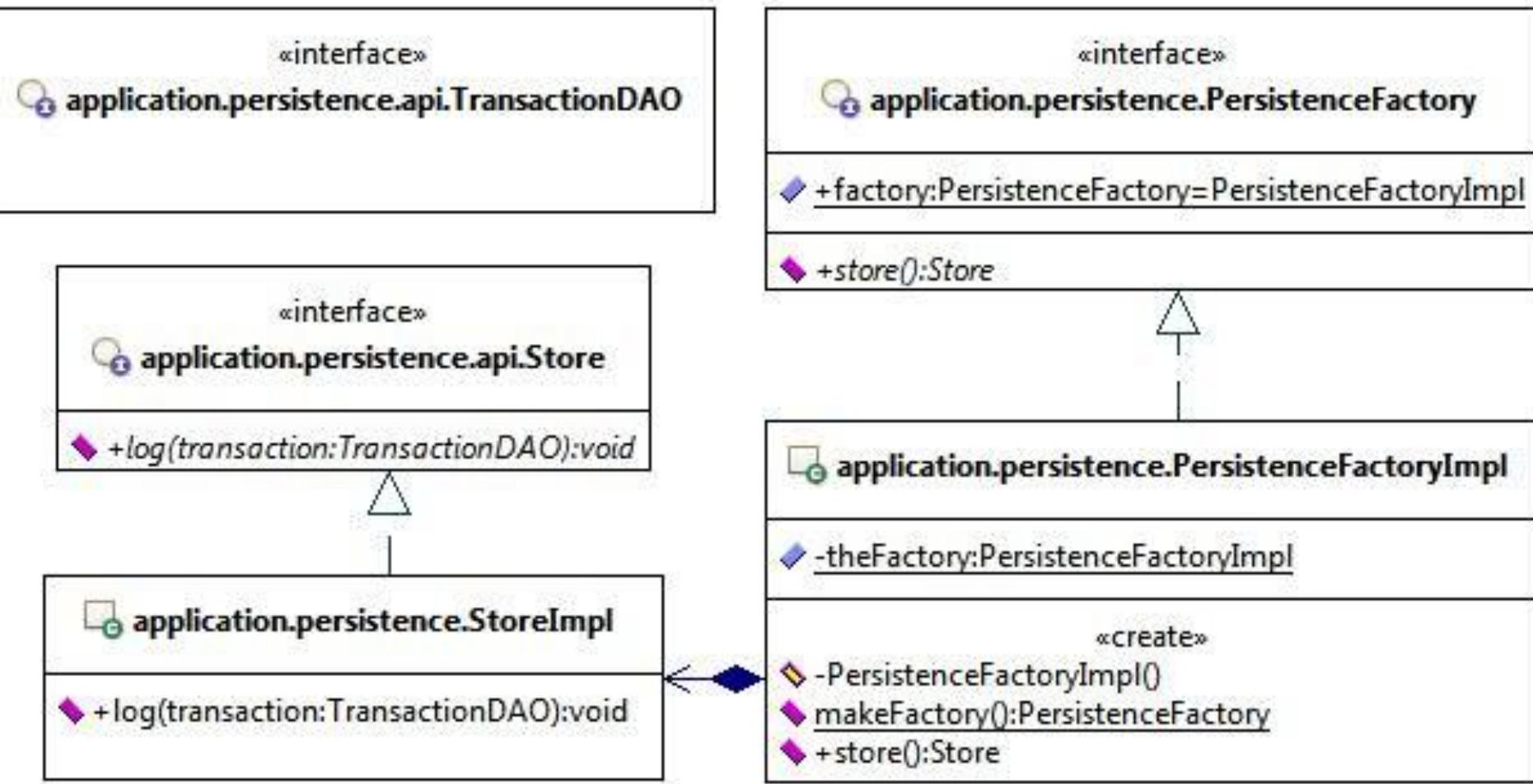
**Nachbedingungen:**

beim ersten Aufruf wird eine neue Transaktion erzeugt  
ein Rechnungsposten wird erzeugt, usw...



# Strukturänderungen

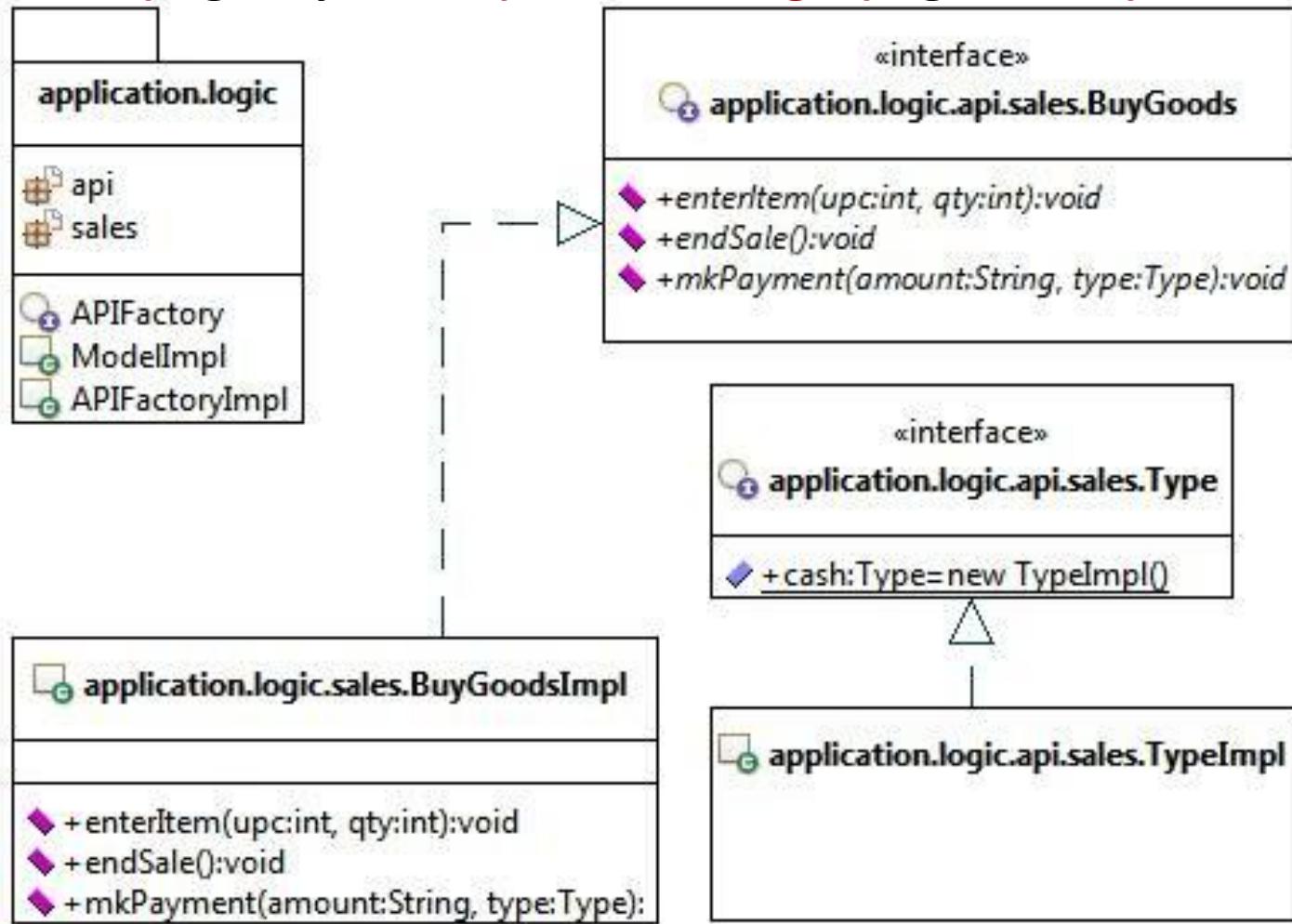
- **Interfaces (persistence und persistence.api)**





# Struktur (Implementierung)

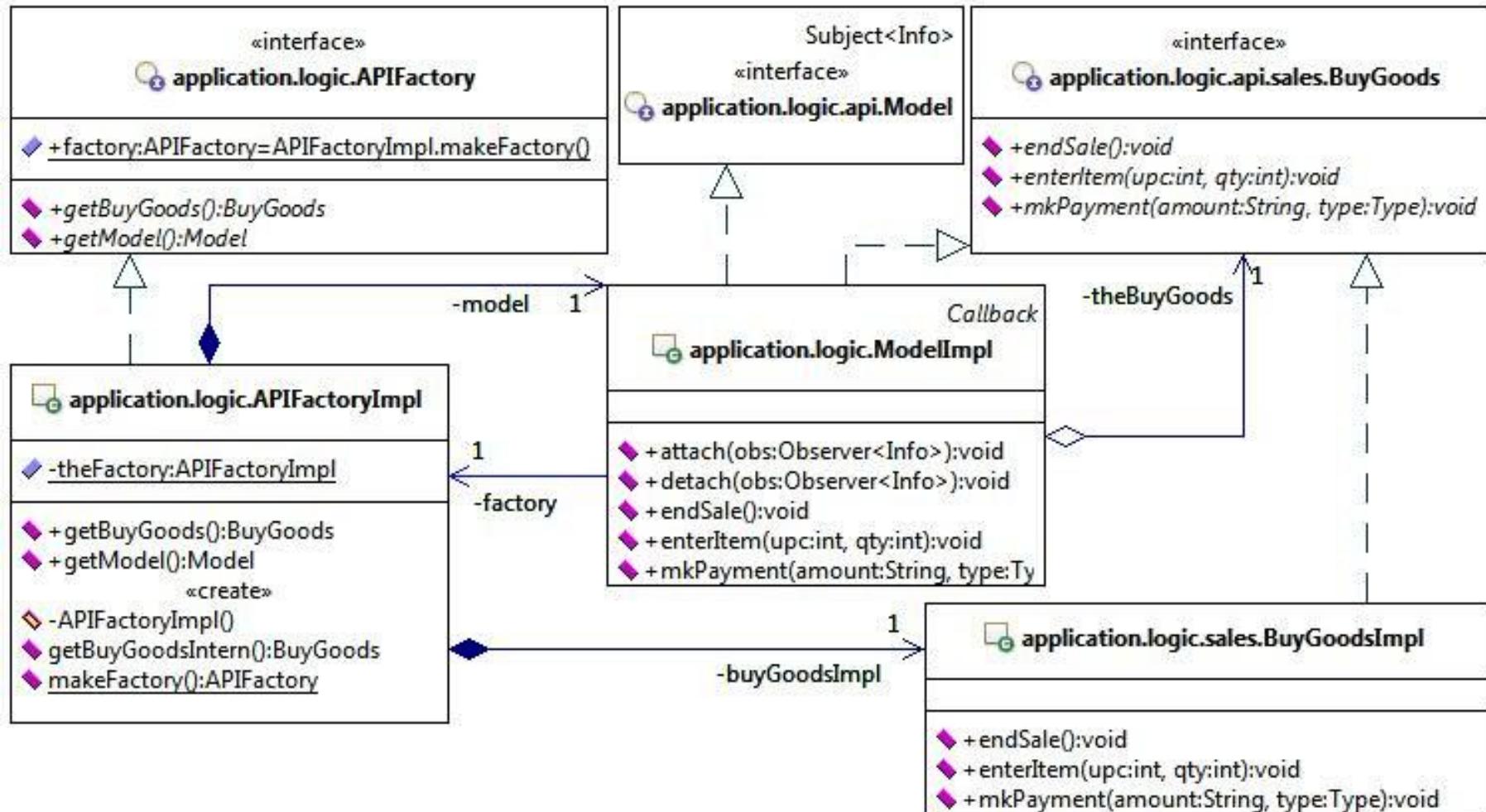
- Interface (logic.api.sales) und Package (logic.sales)





# Struktur (Implementierung)

- **Schnittstelle zur Logik (logic)**





# Struktur (Implementierung)

- **Schnittstelle zur Logik (logic)**

```
class APIFactoryImpl implements APIFactory {  
    private BuyGoodsImpl buyGoodsImpl;  
    //...  
  
    private ModelImpl getModelImpl() {  
        if (this.modelImpl == null)  
            this.modelImpl = new ModelImpl(this);  
        return modelImpl; }  
  
    public Model getModel() {return this.getModelImpl();}  
    public BuyGoods getBuyGoods() {return this.getModelImpl();}  
  
    BuyGoods getBuyGoodsImpl() {  
        if (this.buyGoodsImpl == null)  
            this.buyGoodsImpl = new BuyGoodsImpl();  
        return modelImpl; }  
}
```



# Struktur (Implementierung)

- **Schnittstelle zur Logik (logic)**

```
class ModelImpl implements Model, BuyGoods {  
  
    private BuyGoods theBuyGoods;  
    /*  
     * ...  
    */  
  
    public void enterItem(int upc, int qty) {  
        this.theBuyGoods.enterItem(upc, qty);}  
  
    public void endSale() {  
        this.theBuyGoods.endSale();}  
  
    public void mkPayment(String amount, Type type){  
        this.theBuyGoods.mkPayment(amount, type);}  
}
```

Noch unklar, wann und wer dies initialisiert!

Die Factory stellt es jedoch zur Verfügung.



# Eine Alternative

Als Alternative zu Activity-Diagrammen können bei der Beschreibung von Use-Cases auch Zustandsdiagramme verwendet werden

- das System als Objekt

Prinzipiell dienen Sie der Modellierung und Analyse von Objekten mit komplexem Verhalten.

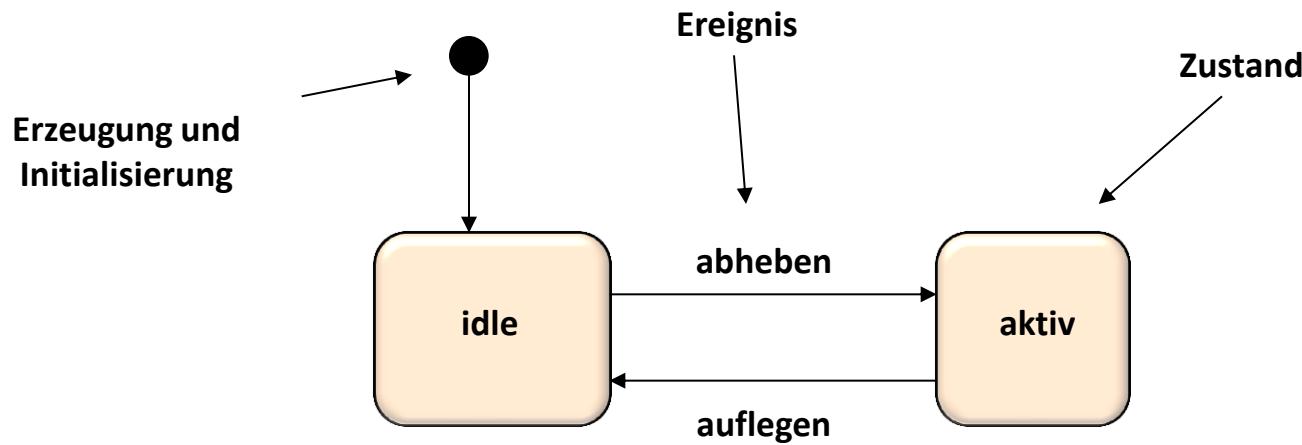
- falls unterschiedliche Reaktionen auftreten bei gleichen Ereignissen
- zur Beschreibung legaler Eingaben und Eingabeabfolgen



# Zustandsdiagramme (State-Diagram) UML

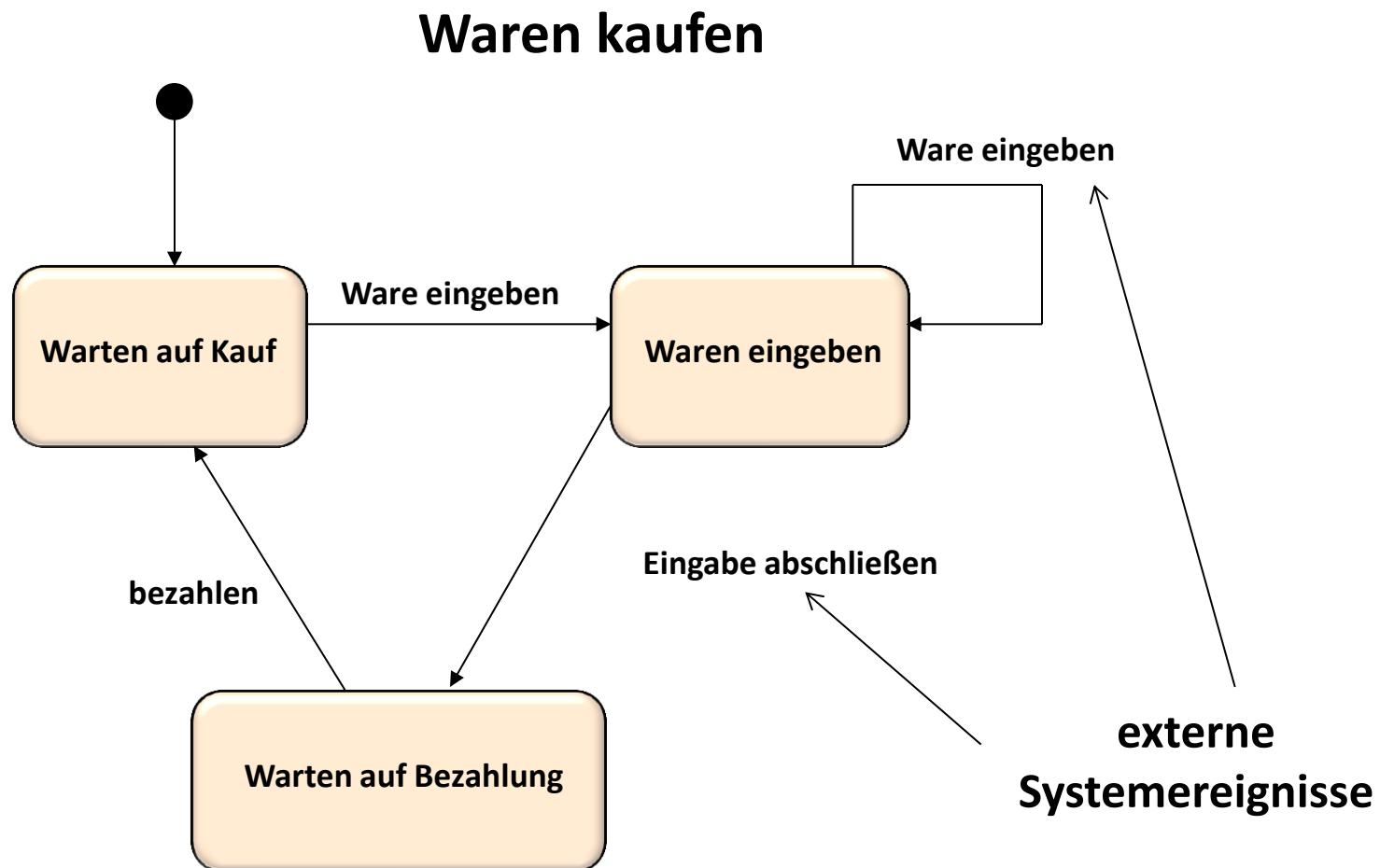
Ein Zustandsdiagramm veranschaulicht die Zustände eines Objekts, d.h. das Verhalten beim Reagieren auf Ereignisse. Ein Zustand repräsentiert eine bestimmte Belegung bestimmter Attribute.

Bsp.: Telefon





# Bsp.: Zustandsdiagramm bei Use-Cases





# Typische Objekte die über Zustandsdiagramme modelliert werden

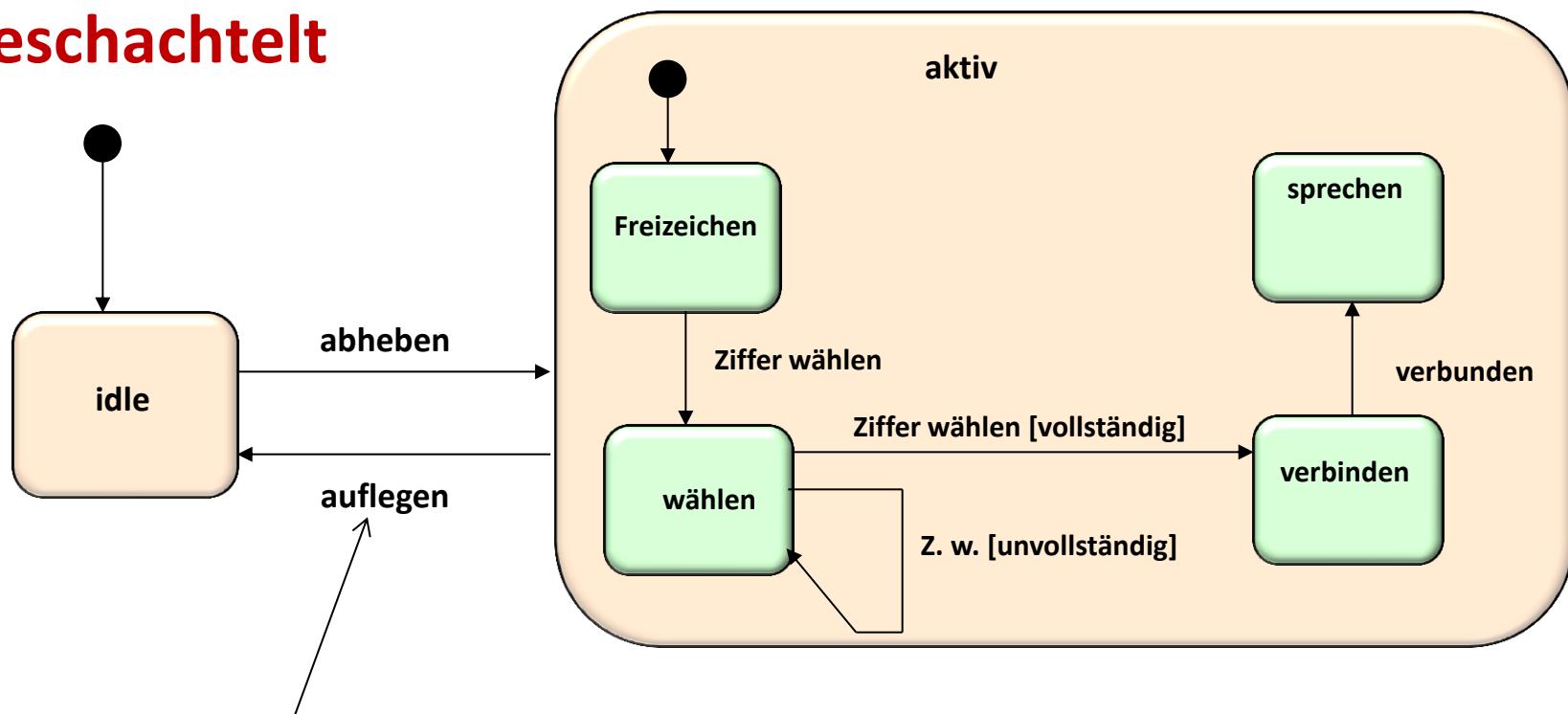
- Windows (Klicks und Eingaben)
- komplexe Transaktionen
- Geräte
- Mutators (Objekte, die häufig ihre Rollen wechseln)
- Gesamt- oder Teilsystem
- **Workflows**



# Komplexe Zustände I

Oft ist es erforderlich, Zustände verschiedener Granularität zu modellieren. Hierzu können Unterzustände eingeführt werden.

- **geschachtelt**

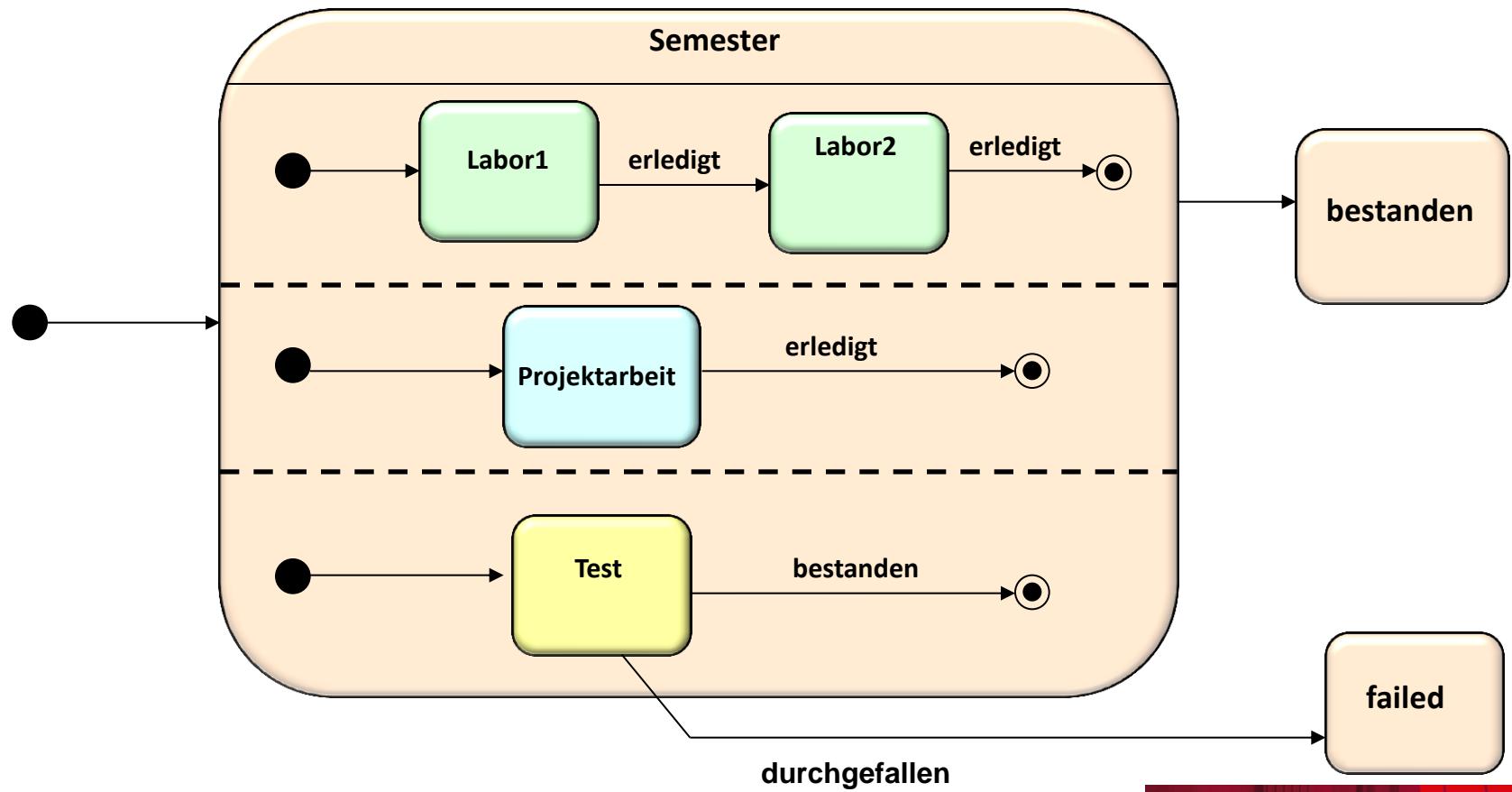


Übergang existiert für alle Unterzustände  
(Freizeichen, wählen, verbinden, sprechen)



# Komplexe Zustände II

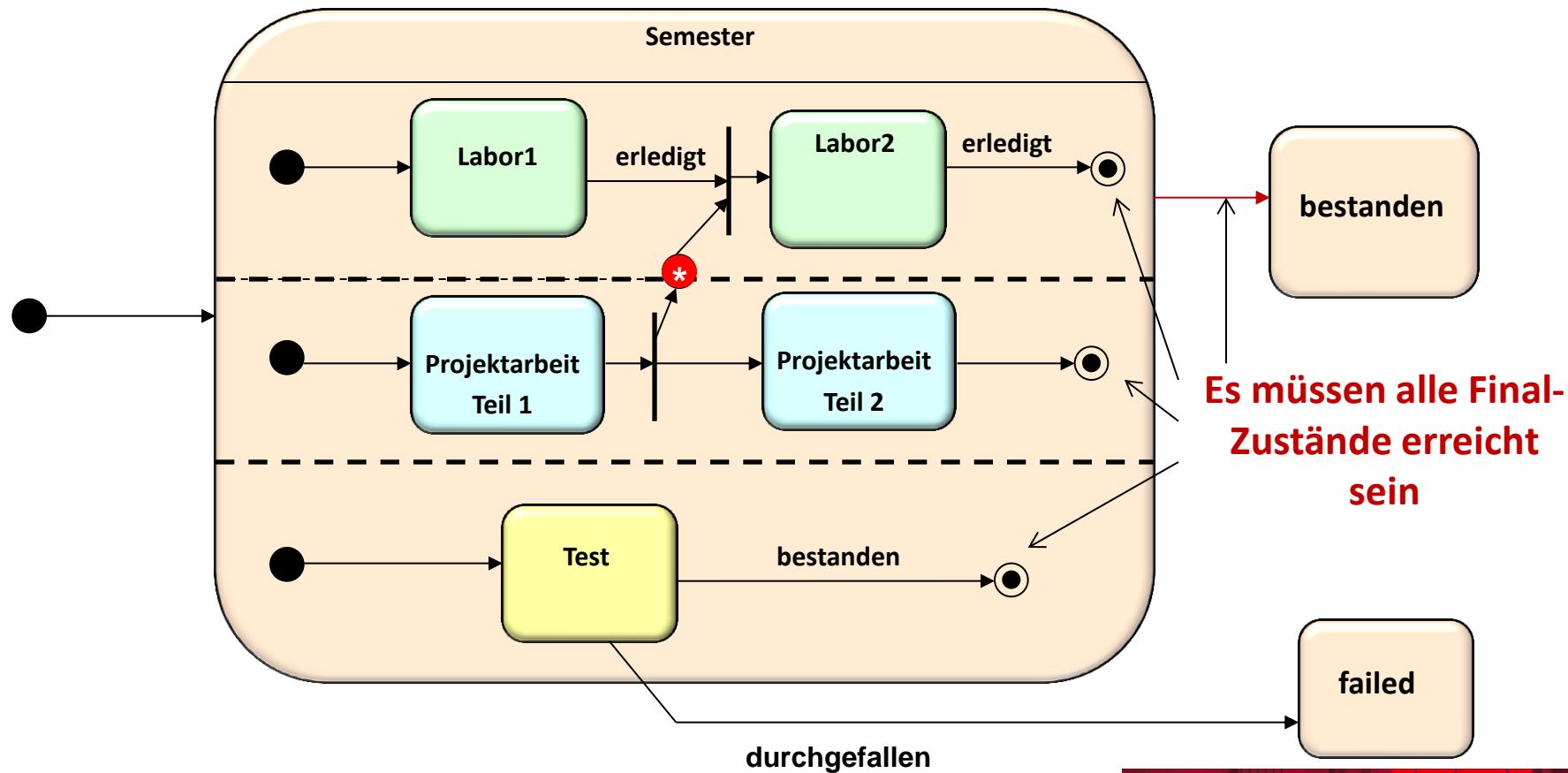
- **Nebenläufigkeit**





# Komplexe Zustände III

- **Synchronisierung:** Labor2 beginnt erst, nachdem Teil 1 der Projektarbeit abgeschlossen ist.





# Aufbau von Transitionen



*trigger ::= trigger-name (parameters)      Methodenaufruf   Signal   Zeitereignis (after())*

*parameters ::= parameter*

*I   parameter, parameters*

*parameter ::= parameter-name : type-expression*

*guard-condition ::= bool expr*

*action-expression ::= pseudo-Code, verwendet Attribute  
und Methoden der entsprechenden Klassen*

**Bsp.:** `right-mouse-down(location) [location in window] /  
object:=pick-object(location);  
object.highlight`



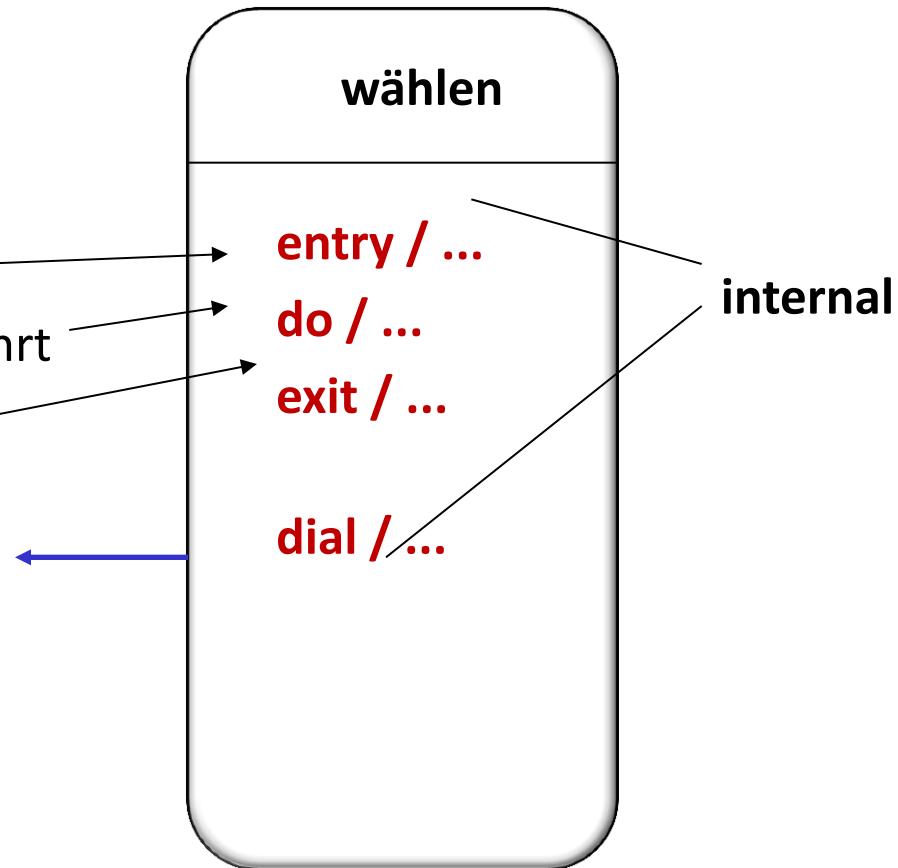
# Besondere Transitionen

## vordefiniert:

- wird beim Betreten ausgeführt
- wird nach dem Betreten ausgeführt
- wird beim Verlassen ausgeführt

## completion:

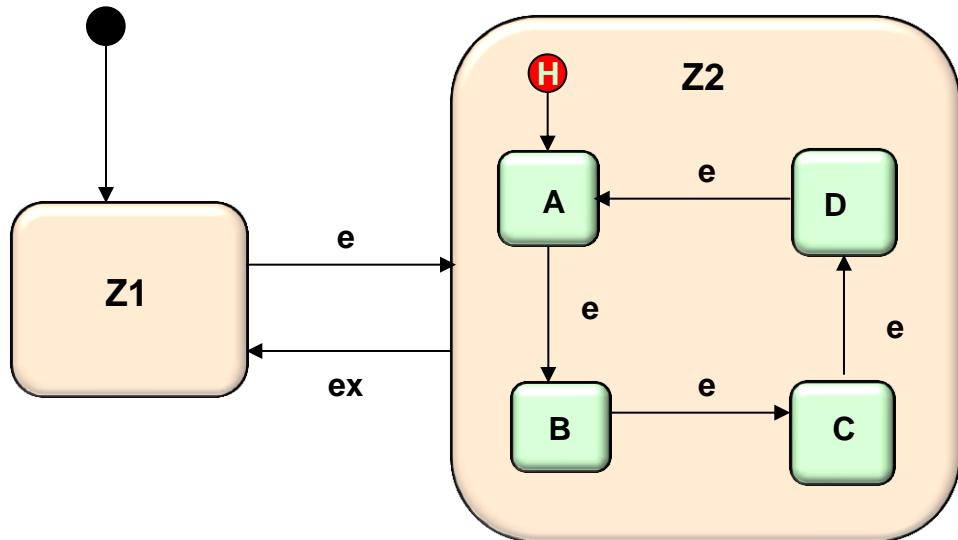
wird beim Beenden ausgeführt





# History-Zustände

History-Zustände merken sich Unterzustände:



Ein Ablauf: **Z1**  $\xrightarrow{e}$  **A**  $\xrightarrow{e}$  **B**  $\xrightarrow{e}$  **C**  $\xrightarrow{ex}$  **Z1**  $\xrightarrow{e}$  **C**  $\xrightarrow{e}$  **D**  $\xrightarrow{ex}$  **Z1**



# Protokoll-Automaten

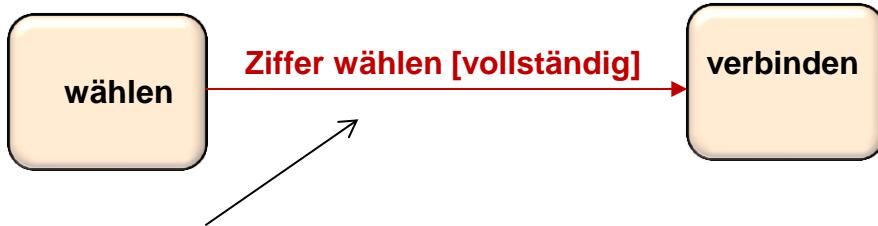
Neben der Beschreibung des Verhaltens eines Objekts können Zustandsautomaten auch dazu eingesetzt werden, Protokolle zu beschreiben, die durch ein **Objekt** oder **System** realisiert werden.



- Dies ermöglicht die Modellierung der Schnittstelle.
- Welche Operationen sind wann zulässig.



# Protokoll-Automaten



*transition ::= trigger [guard-condition] / action-expression*

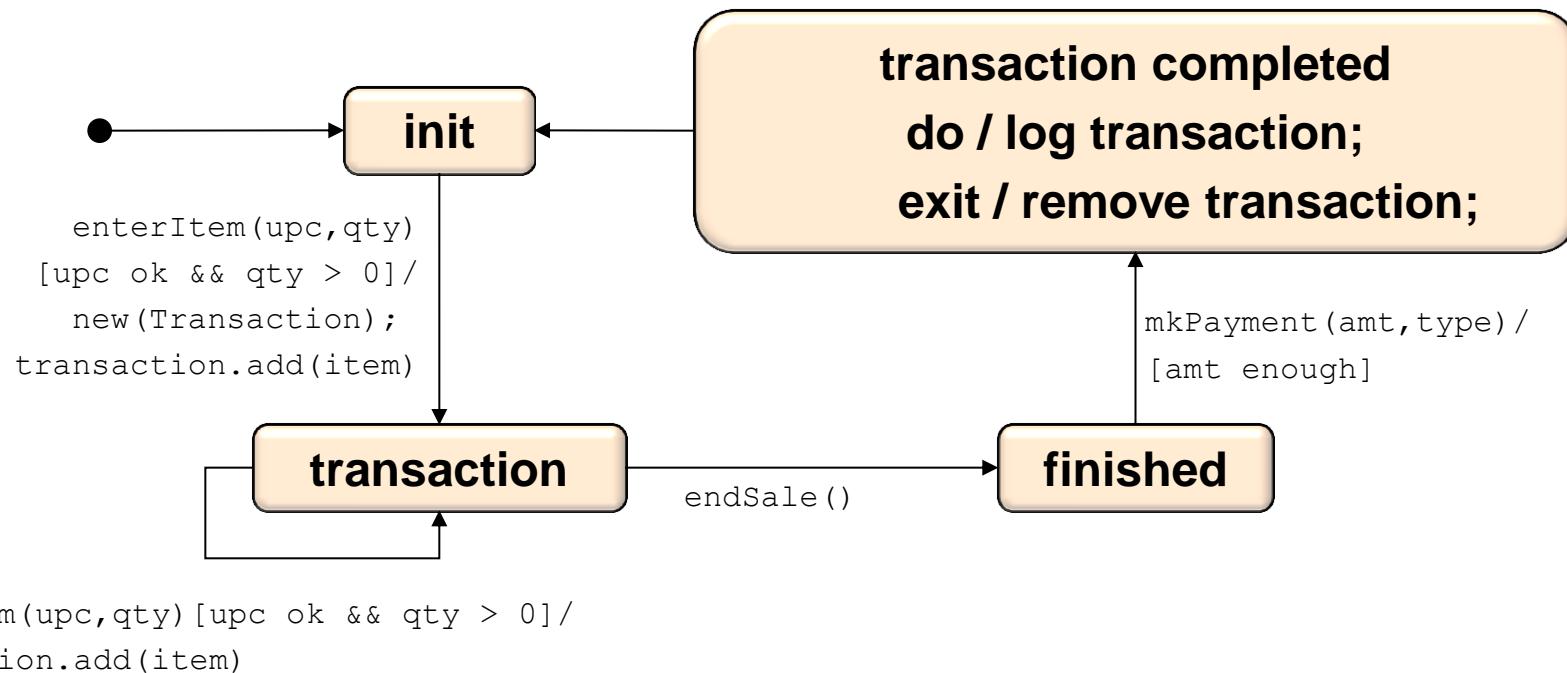
- **Zustand –**  
beschreibt eine Situation, in der der Aufruf einer System-Operation erlaubt ist.

**Diese steht in der Regel direkt mit einem Use-Case in Verbindung.**

- **Trigger –**  
stellt den Aufruf einer System-Operation dar.
- **Guard-Condition –**  
Bedingungen, die sich auf Parameter der System-Operation und besondere Aspekte des Zustands beziehen.
- **Action-Expression –**  
Ereignisse, die den Zustandsübergang charakterisieren. (**Pseudo-Code**)



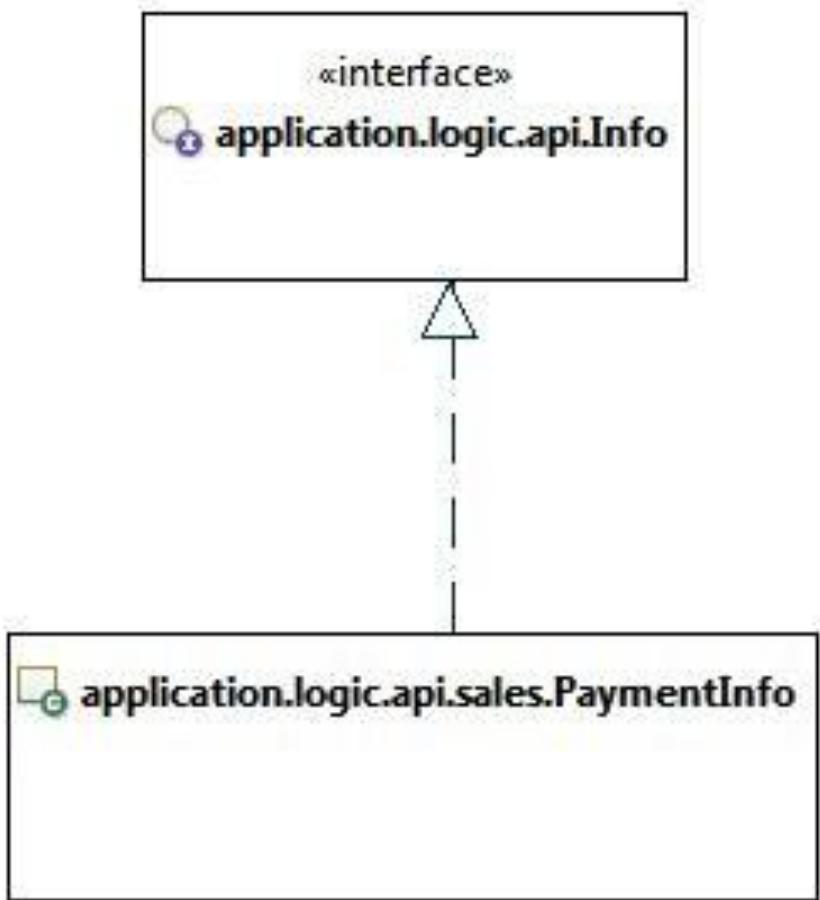
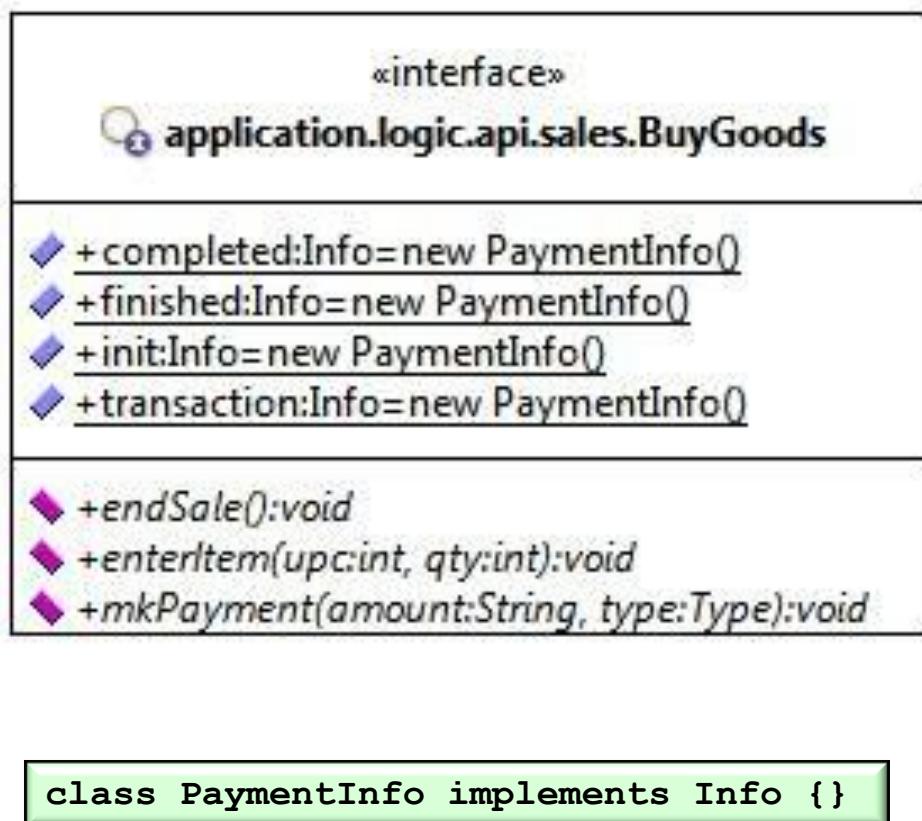
# Beispiel





# Struktur (Implementierung)

- **Interface (logic.api)**

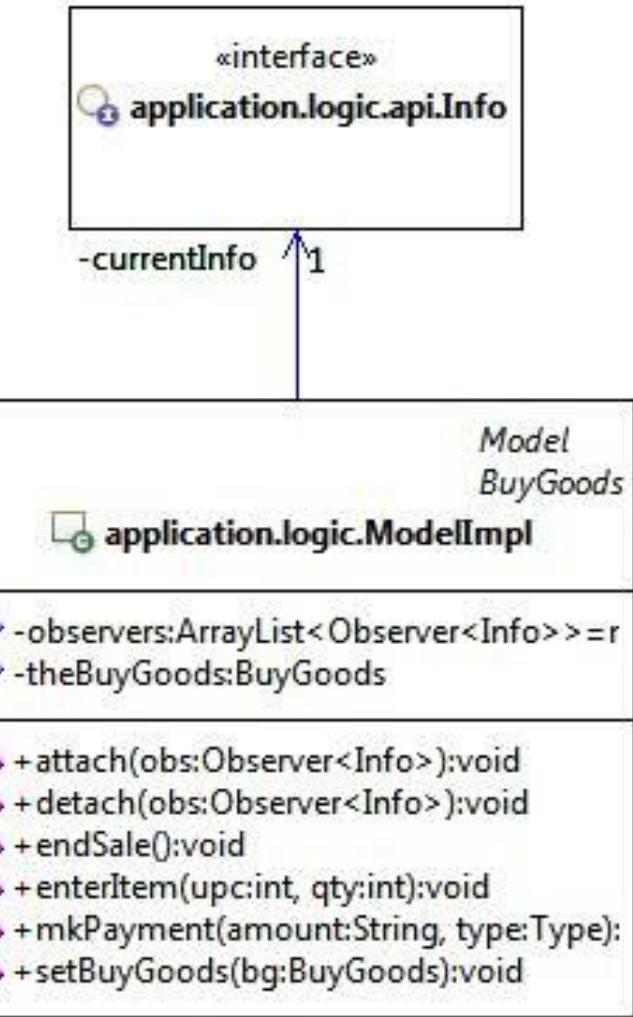




# Struktur (Implementierung)

- **Schnittstelle zur Logik (logic)**

```
class ModelImpl implements Model, BuyGoods {  
  
    private Info currentInfo;  
/*  
...  
*/  
    public void enterItem(int upc, int qty) {  
        if (this.currentInfo == BuyGoods.init ||  
            this.currentInfo == BuyGoods.transaction)  
            this.theBuyGoods.enterItem(upc, qty);}  
  
    public void endSale() {  
        if (this.currentInfo == BuyGoods.transaction)  
            this.theBuyGoods.endSale();}  
  
    public void mkPayment(String amount, Type type) {  
        if (this.currentInfo == BuyGoods.finished)  
            this.theBuyGoods.mkPayment(amount, type);}  
    }  
}
```





Larman, Craig. Applying UML and patterns : an introduction to object-oriented analysis and design and the Unified Process, 2. ed. – Upper Saddle River, NJ : Prentice Hall, 2002.

# **Das Herzstück des Designs**



# Verantwortlichkeiten zuweisen

- **Verantwortlichkeiten, etwas zu tun**
  - etwas selbst tun
  - jemanden auffordern, etwas zu tun (delegieren)
- **Verantwortlichkeiten, etwas zu wissen**
  - private Informationen kennen
  - abgeleitete oder berechnete Information kennen
  - zugeordnete Objekte kennen

**Die Verantwortlichkeiten werden den Objekten  
während des Designs zugeordnet.**



# Einige Prinzipien

- **Experte:**  
Verantwortlichkeit dem Informationsexperten geben; der Klasse, die alle notwendigen Informationen hat.
- **Erzeuger:**  
Eine Klasse ist verantwortlich für die Erzeugung einer anderen, wenn sie diese Klasse beinhaltet, oder alle Informationen für die Initialisierung kennt.
- **Lose Kopplung:**  
Verantwortlichkeiten so zuweisen, dass die Kopplung nicht überhand nimmt.
- **Hohe Kohäsion:**  
Verantwortlichkeiten so zuweisen, dass nicht einer alles macht.
- **Steuerung:**  
Die Verantwortlichkeit der Steuerung obliegt einer Klasse (Steuereinheit), die
  - das gesamte System repräsentiert,
  - die gesamte Organisation repräsentiert,
  - einen Akteur der realen Welt modelliert,
  - einen Use-Case repräsentiert,



# Das Erstellen von Interaktionsdiagrammen

- Für jede Systemoperation wird ein separates Diagramm erstellt.

**Ein Diagramm, in dem diese Operation die Startoperation ist.**
- Falls das Diagramm zu komplex wird, wird es in kleinere aufgespaltet
- Man verwendet die Funktionsbeschreibungen und Use-Cases als Ausgangspunkt. **Ziel ist es, jede Funktion als System interagierender Objekte zu beschreiben**



# Interaktionsdiagramme (UML)

**Interaktionsdiagramme beschreiben dynamische Modellsachverhalte, d.h. wie Gruppen von (i.a.) Objekten zusammenarbeiten.**

Ein Interaktionsdiagramm erfasst üblicherweise das Verhalten eines einzelnen Anwendungsfalls. Das Diagramm zeigt eine Anzahl von exemplarischen kommunikationsfähigen Instanzen und die Nachrichten, die zwischen ihnen innerhalb des Anwendungsfalls ausgetauscht werden.

- **Kommunikationsdiagramme** (Communication Diagram)  
Diagramme, die auf Objektdiagrammen basieren und die Interaktionen in räumlich verteilter Sicht darstellen. Die Verbindung der *Objekte* steht im Vordergrund.
- **Sequenzdiagramme** (Sequenz Diagram)  
Diagramme, die den zeitlichen Verlauf der Interaktion in den Vordergrund stellen und die Objektlebenszeit betonen.



# Sequenzdiagramme (UML)

**Sequenzdiagramm zeigt im wesentlichen den gleichen Sachverhalt wie ein Kommunikationsdiagramm, jedoch aus einer anderen Perspektive.**

**Beim Sequenzdiagramm steht der zeitliche Verlauf der Nachrichten im Vordergrund. Objekte erhalten Lebenslinien, und die Zeit vergeht von oben nach unten.**

**Für Realzeitanwendungen, kann eine Metrik eingeführt werden.**



# Sequenzdiagramme

Name der Interaktion (hier eine Operation mit Parameter)

sd reserve(ord:Order)

Attribut – bindet  
den Rückgabewert

reserve(ord)

```
reserve(Order ord) {  
    Item item;  
    Product prod;  
    int qty;  
    ...  
    item = ord.getItem();  
    prod = item.getProduct();  
    qty = item.getQuantity();  
    store.reserve(prod,qty);  
}
```

:Reservation

ord:Order

x:Item

store

getItem()

item = getItem(): x

getProduct()

prod = getProduct()

getQuantity()

qty = getQuantity()

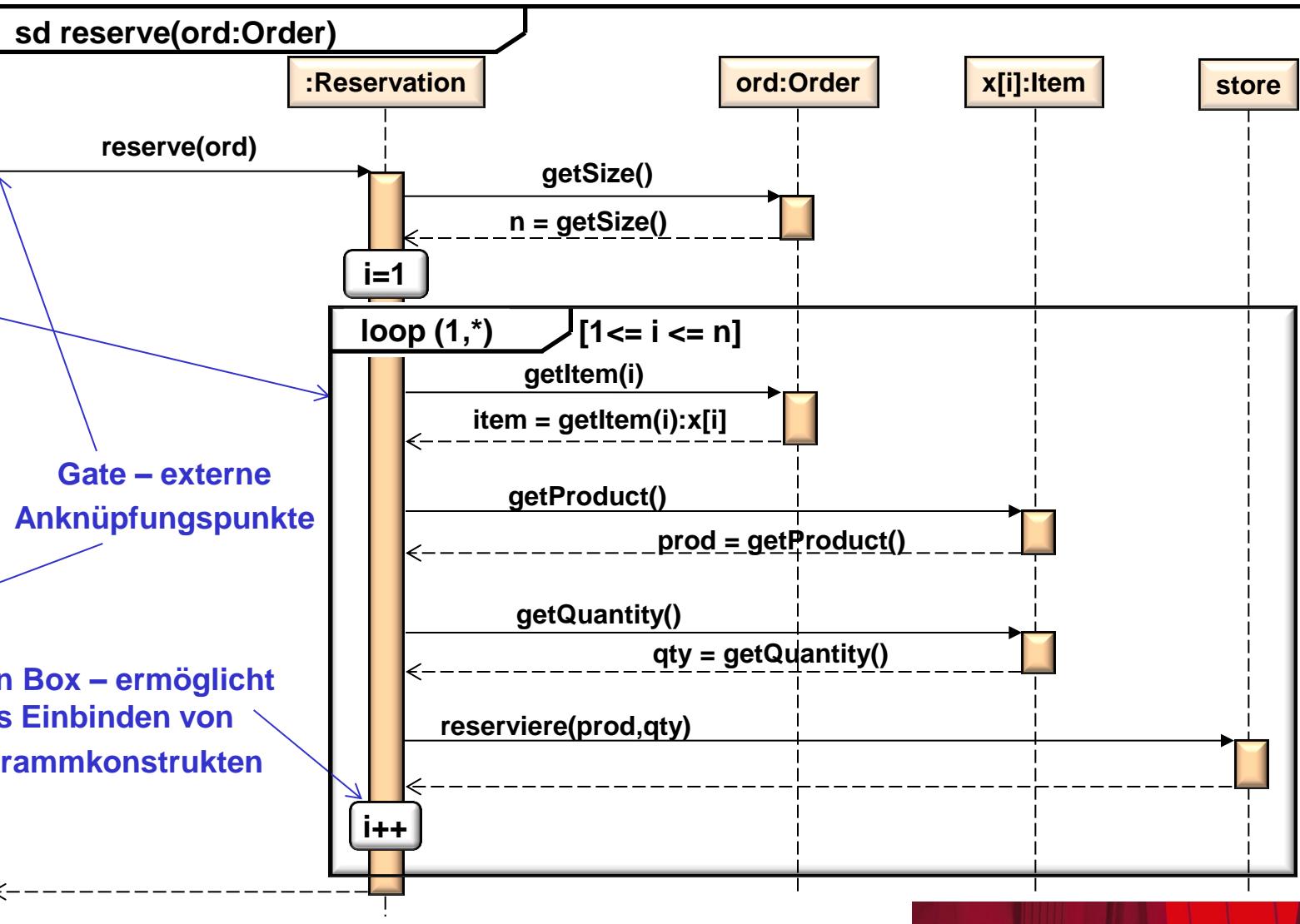
reserve(prod,qty)

Objekt



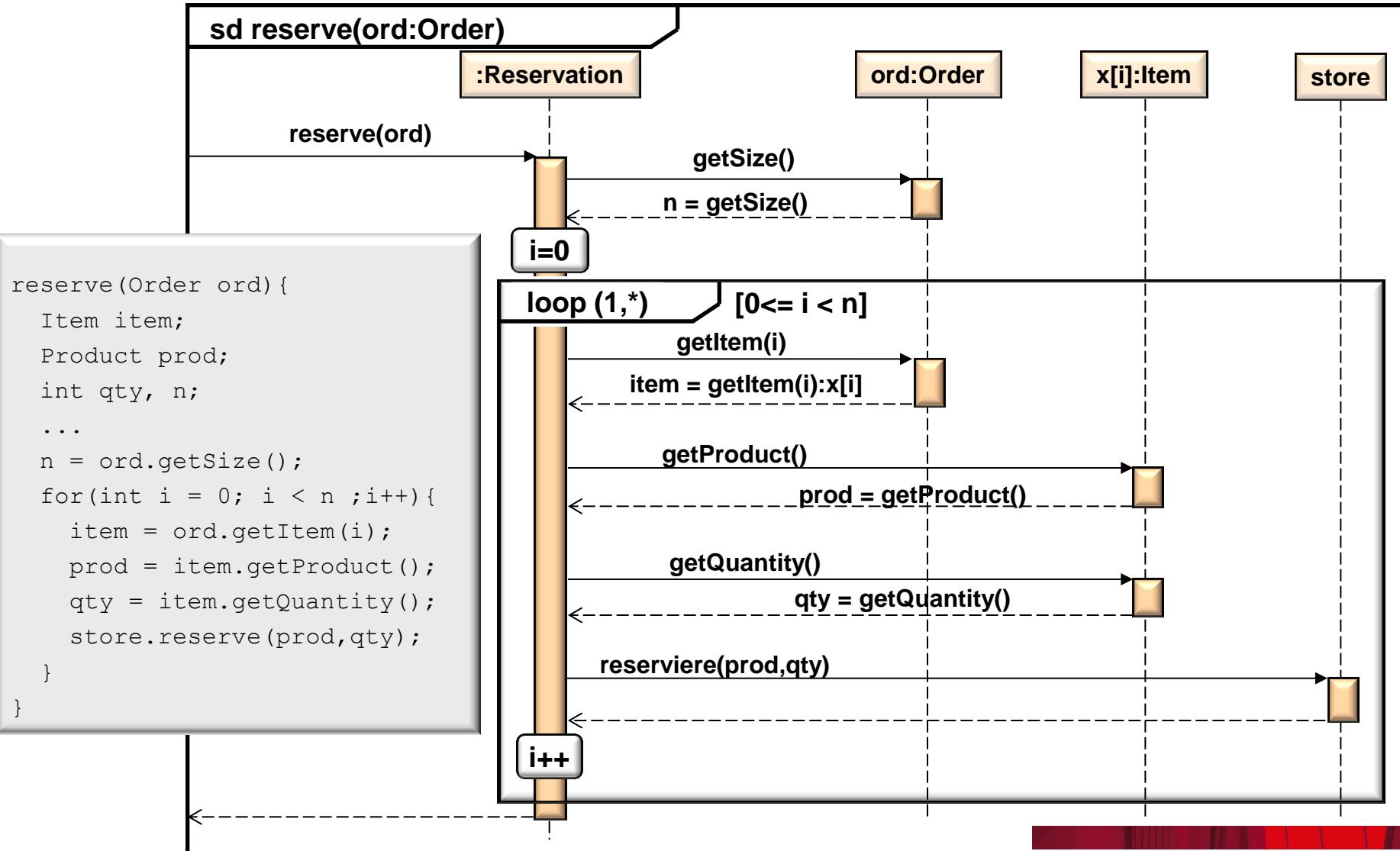
# Sequenzdiagramme

Fragmente –  
dient der  
Strukturierung



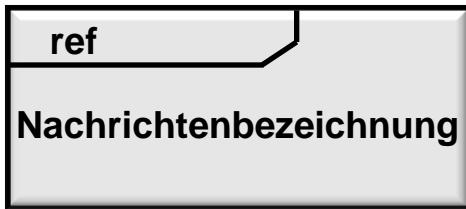


# Sequenzdiagramme

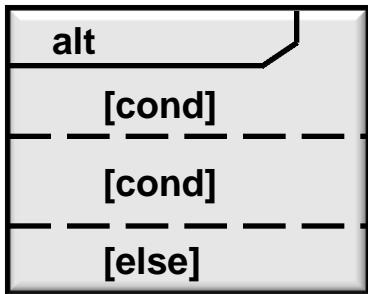




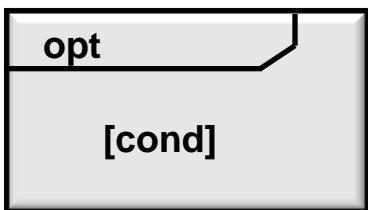
# Kombinierte Fragmente



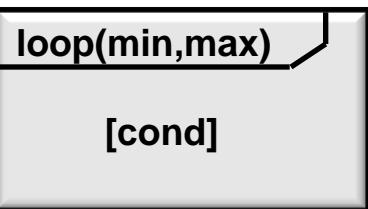
- **Allgemeine Referenz auf ein Fragment**  
Die Bindung von Werten erfolgt über die Nachrichtenbezeichnung



- **Zur Darstellung alternativer Abläufe**



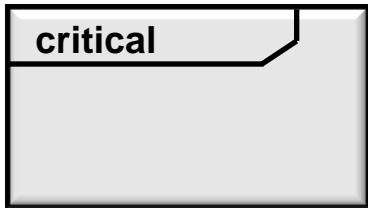
- **Zur Darstellung optionaler Abläufe**



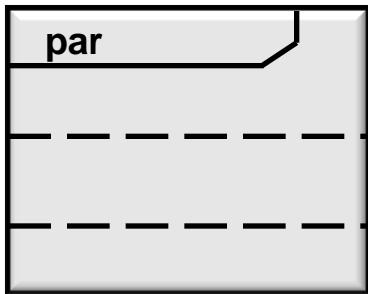
- **Zur Darstellung iterativer Abläufe**



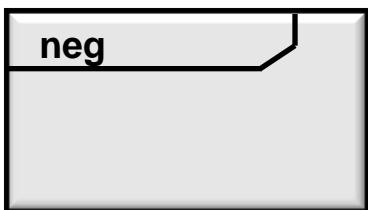
# Kombinierte Fragmente



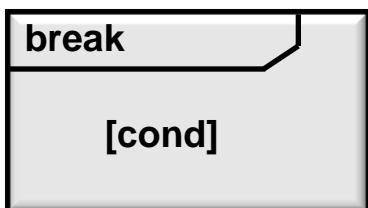
- Zur Darstellung kritischer Abschnitte



- Zur Darstellung paralleler Abläufe



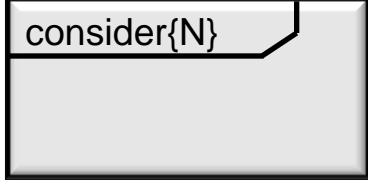
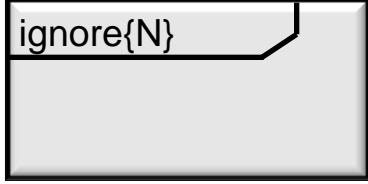
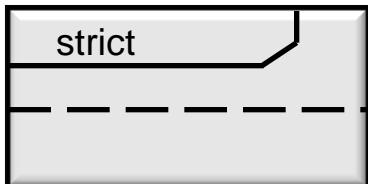
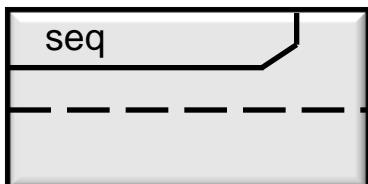
- Zur Darstellung ungültiger Abläufe



- Zur Darstellung von Ausnahmebehandlungen



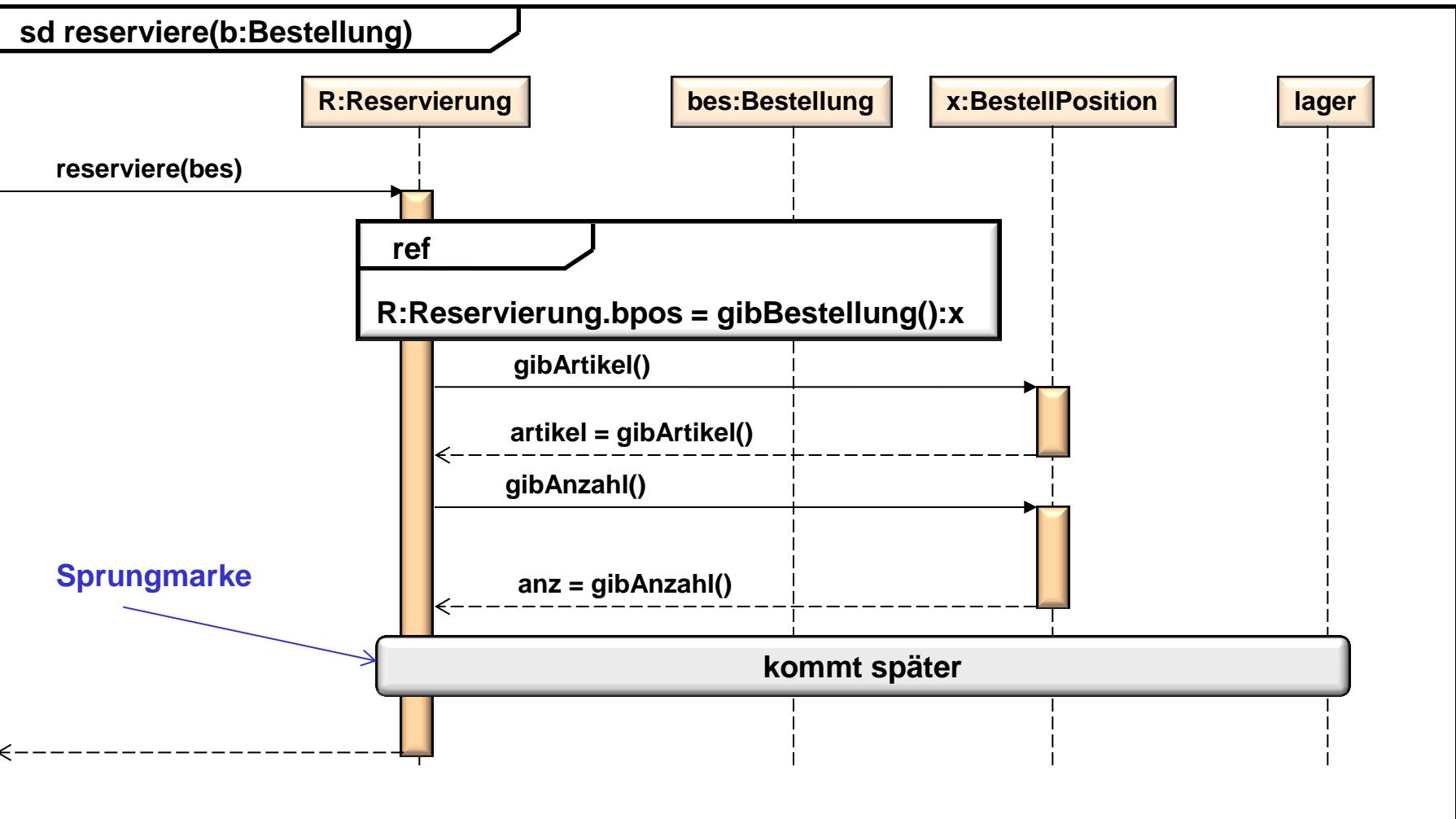
# Kombinierte Fragmente



- **Der beschriebene Ablauf ist unter allen Umständen bindend, die Beschreibung ist vollständig**
- **Zur Abgrenzung reihenfolgetreuer Abläufe**
- **Zur Hervorhebung der strikten Einhaltung der zeitlichen Abfolge auch über Grenzen von Lebenslinien hinaus.**
- **Zur Darstellung von Nachrichten, die für den aktuellen Sachverhalt irrelevant aber technisch notwendig sind**
- **Zur Hervorhebung besonders bedeutender Nachrichten**

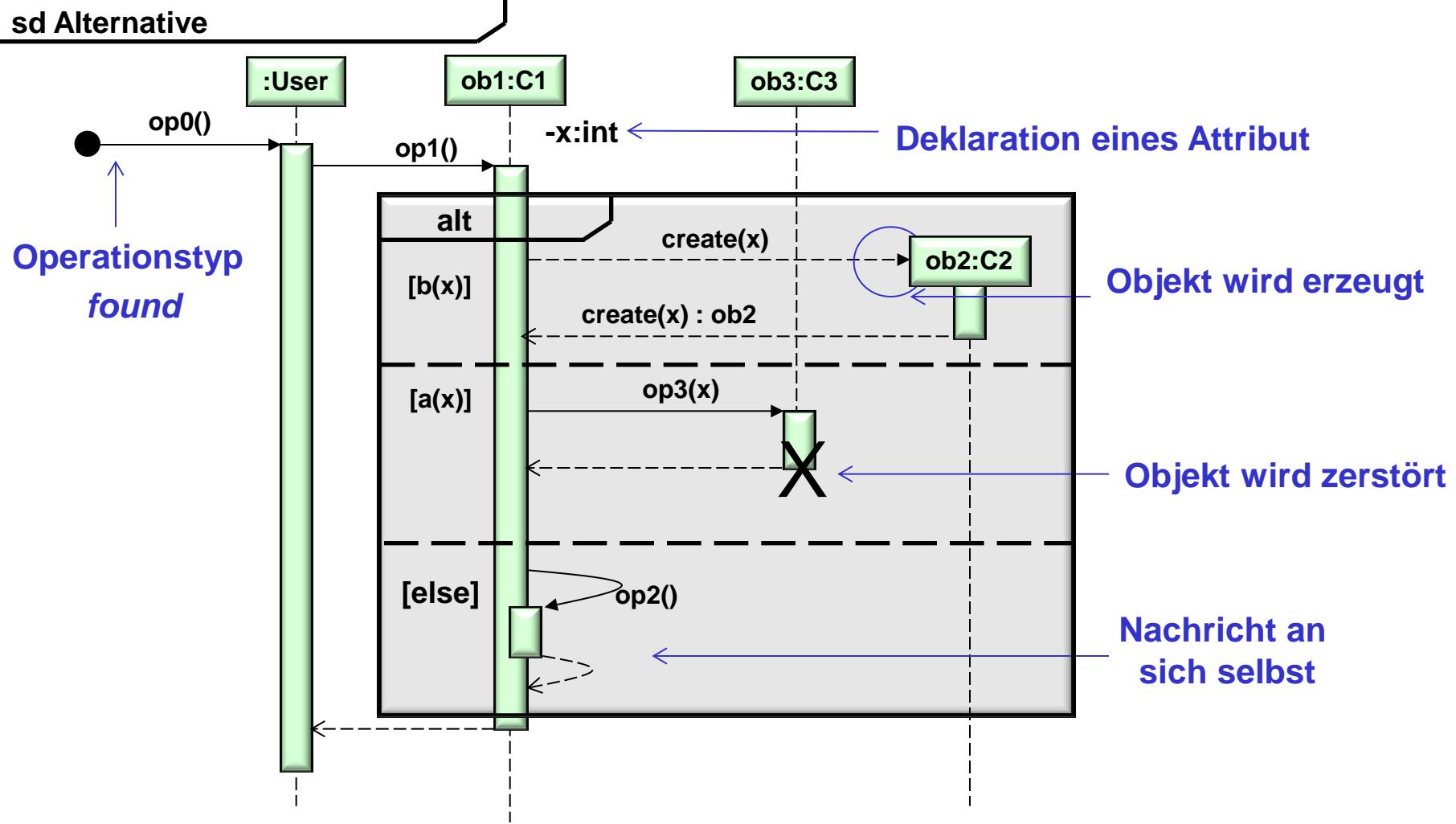


# Kombinierte Fragmente (Referenz)



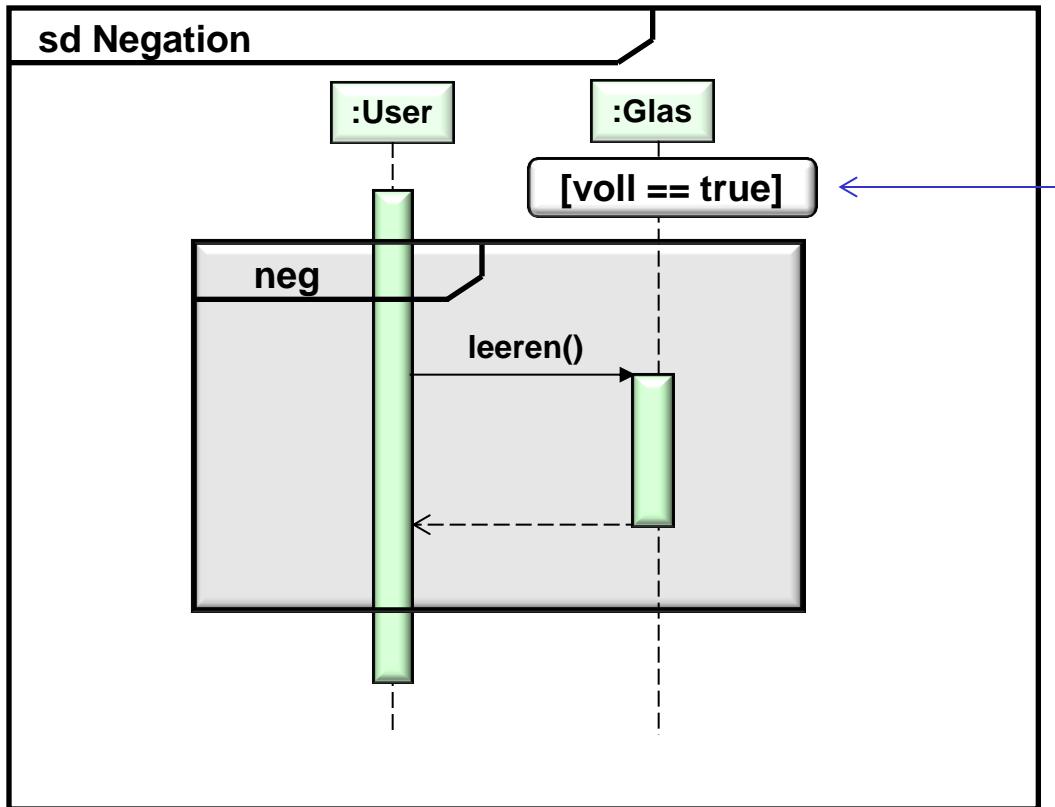


# Kombinierte Fragmente (Alternative)



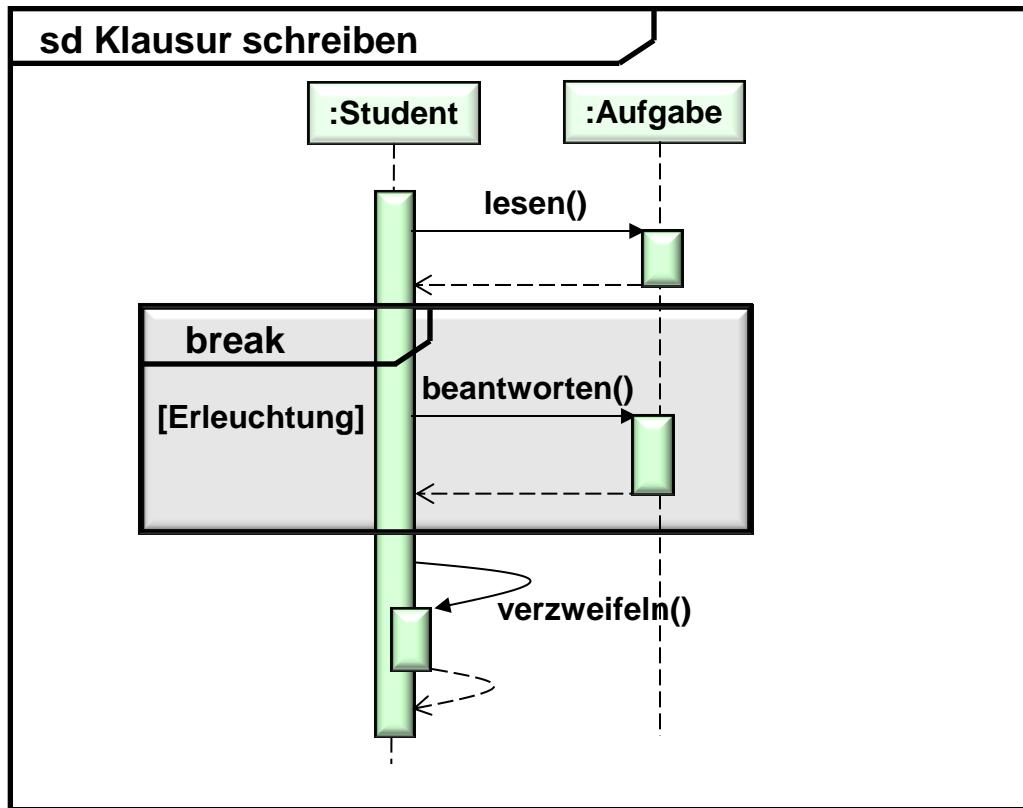


# Kombinierte Fragmente (Negation)





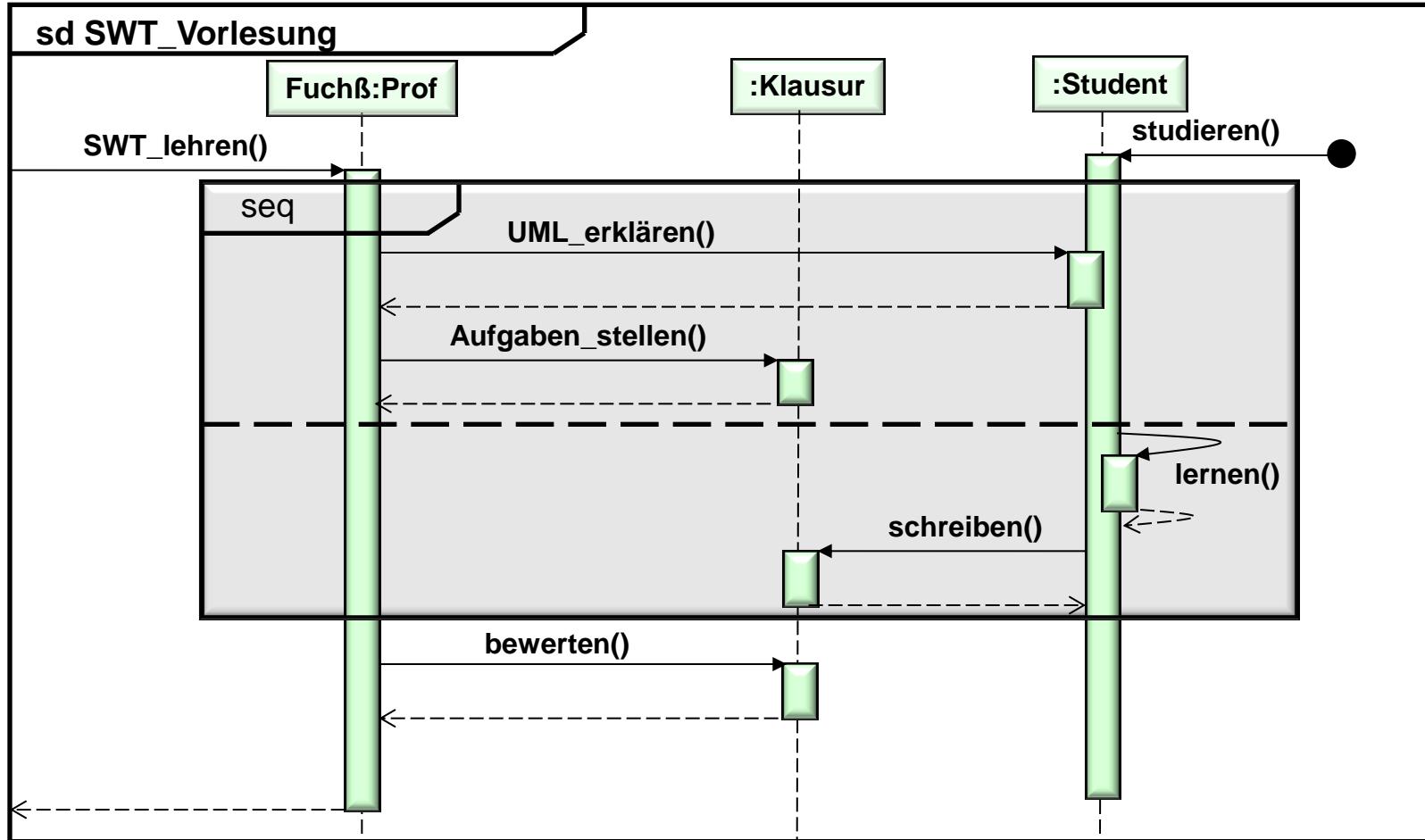
# Kombinierte Fragmente (Abbruch)





# Kombinierte Fragmente (Ordnung)

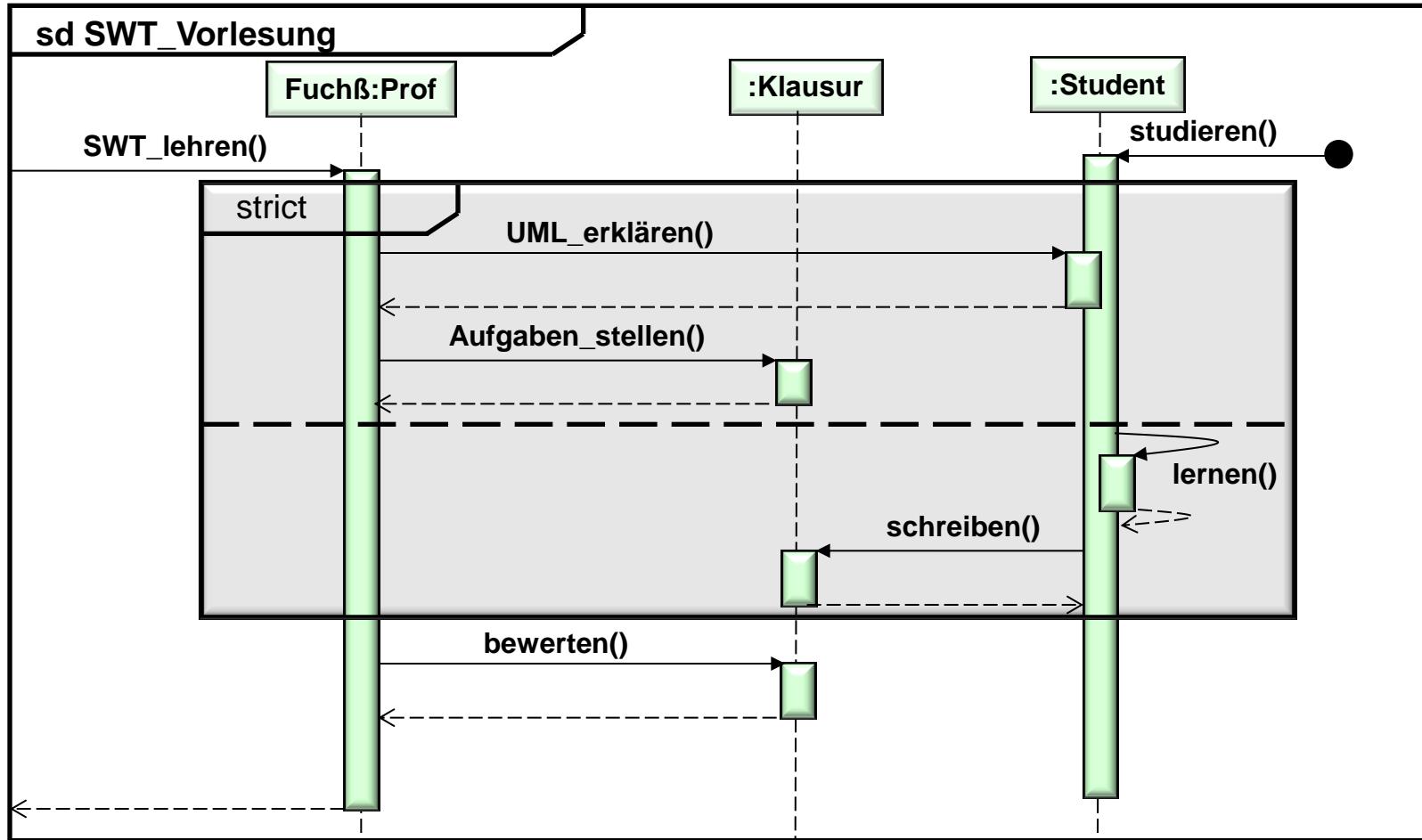
Es gibt keine zeitliche Abhangigkeit zwischen dem Lernen und dem Stellen der Aufgaben.





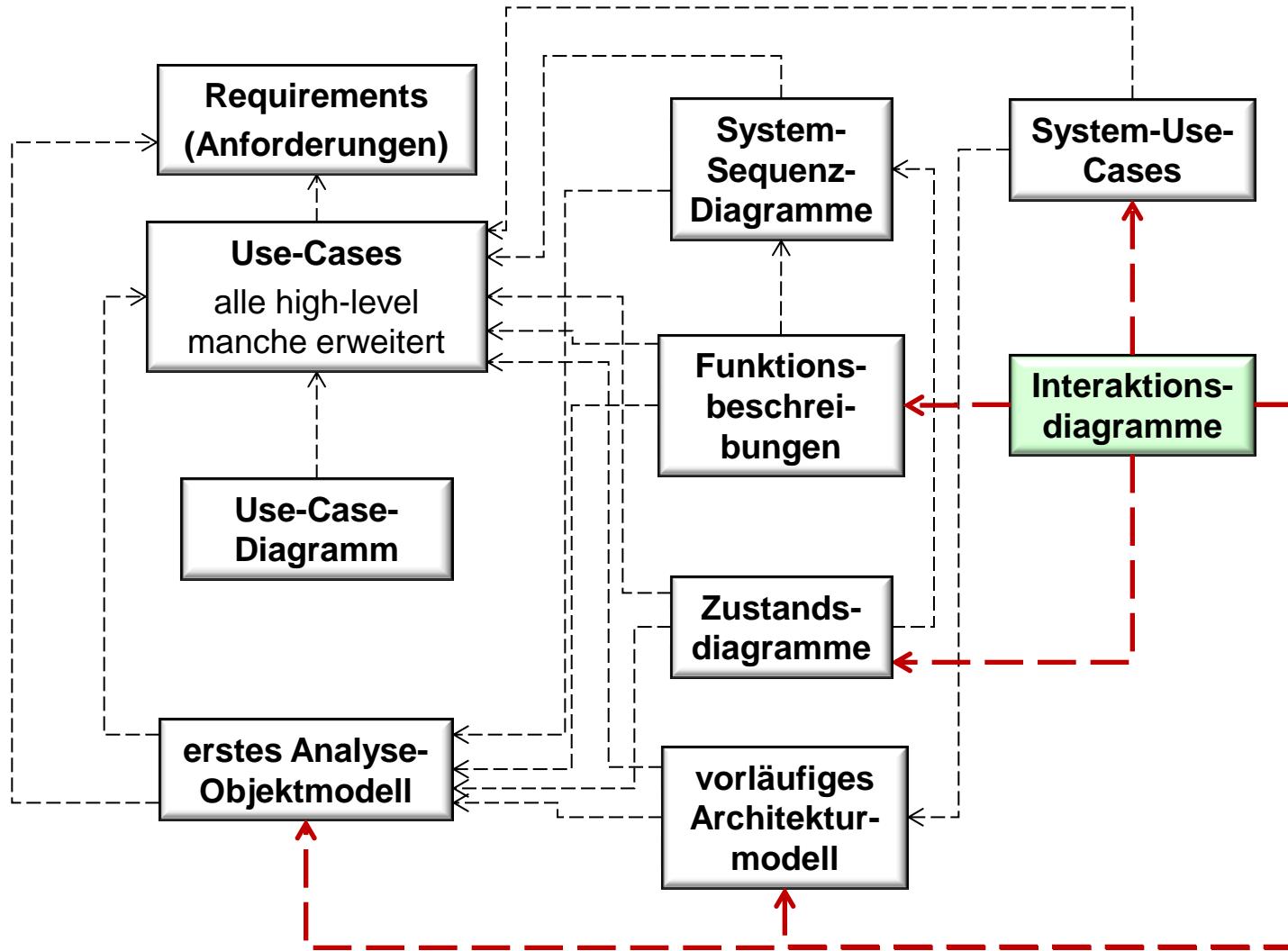
# Kombinierte Fragmente (Sequenz)

## Ohne Idealismus



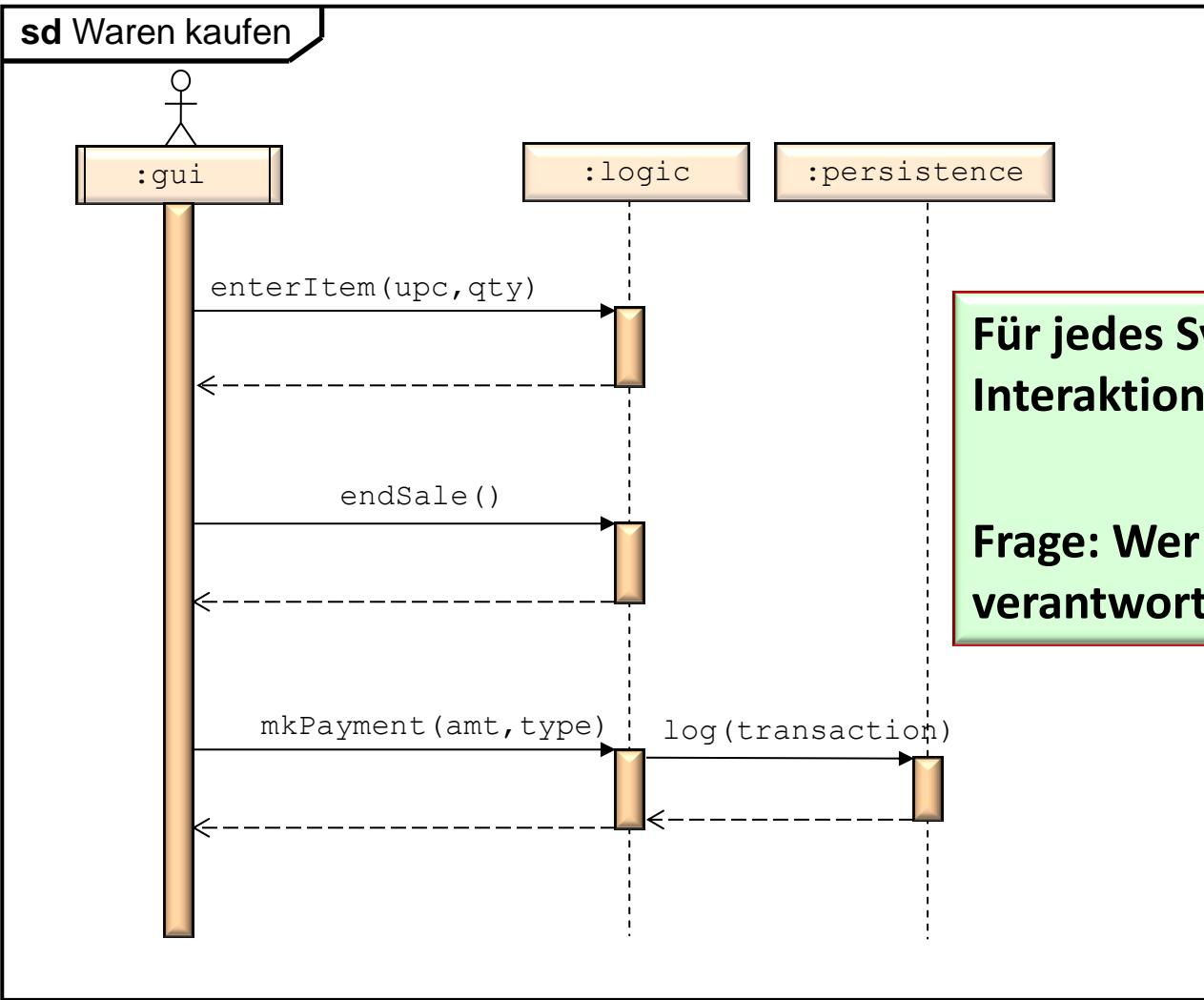


# Die zentralen Designschritte





# System-Sequenz-Diagramm



Für jedes Systemereignis wird ein Interaktionsdiagramm erstellt.

Frage: Wer ist für diese Operationen verantwortlich?



# Funktionsbeschreibung

**Name:**

`enterItem(upc: Integer, qty: Integer)`

**Verantwortlichkeit:**

registriert den Kauf der Ware und fügt ihn der Transaktion hinzu; zeigt Preis und Beschreibung an

**Referenzen:**

Use-Case: Ware kaufen

Funktionen:  $F_i - F_j$

**Bemerkungen:**

.....

**Ausnahmen:**

falls UPC unbekannt, ist ein Fehler zu melden

**Vorbedingungen:**

UPC ist bekannt, ...

**Nachbedingungen:**

beim ersten Aufruf wird eine neue Transaktion erzeugt  
ein Rechnungsposten wird erzeugt, usw...



# Die Steuereinheit für enterItem()

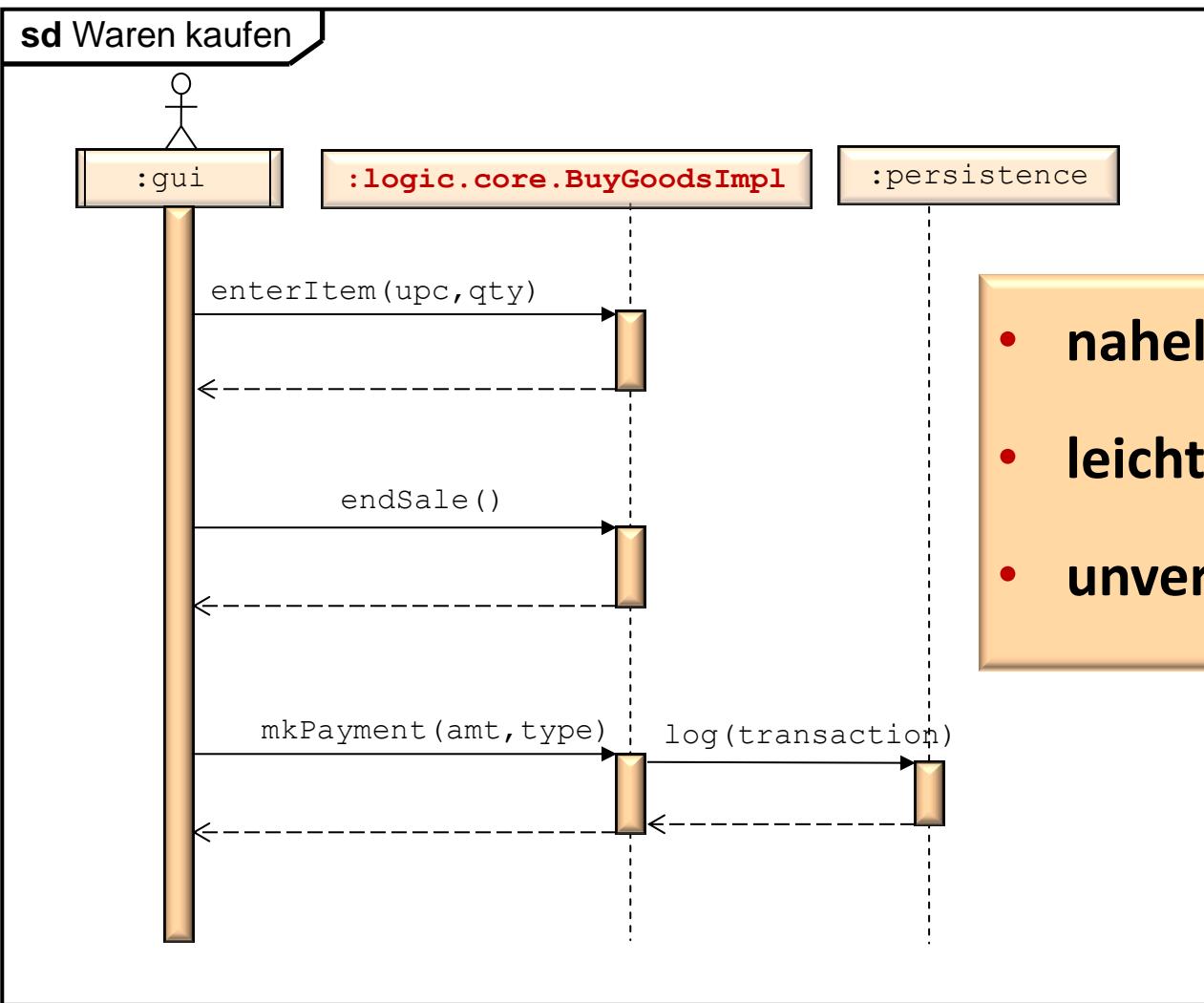
Gemäß dem Steuerungsprinzip gibt es folgende Alternativen:

- Die **Kasse** repräsentiert das Gesamtsystem  
(oder eine neue Klasse **Kassensystem**)
- Der **Supermarkt** repräsentiert die Organisation
- Der **Kassierer** repräsentiert einen Akteur
- **BuyGoods** ein Use-Case-Controller

Eine Steuereinheit für alle Operationen eines Use-Cases



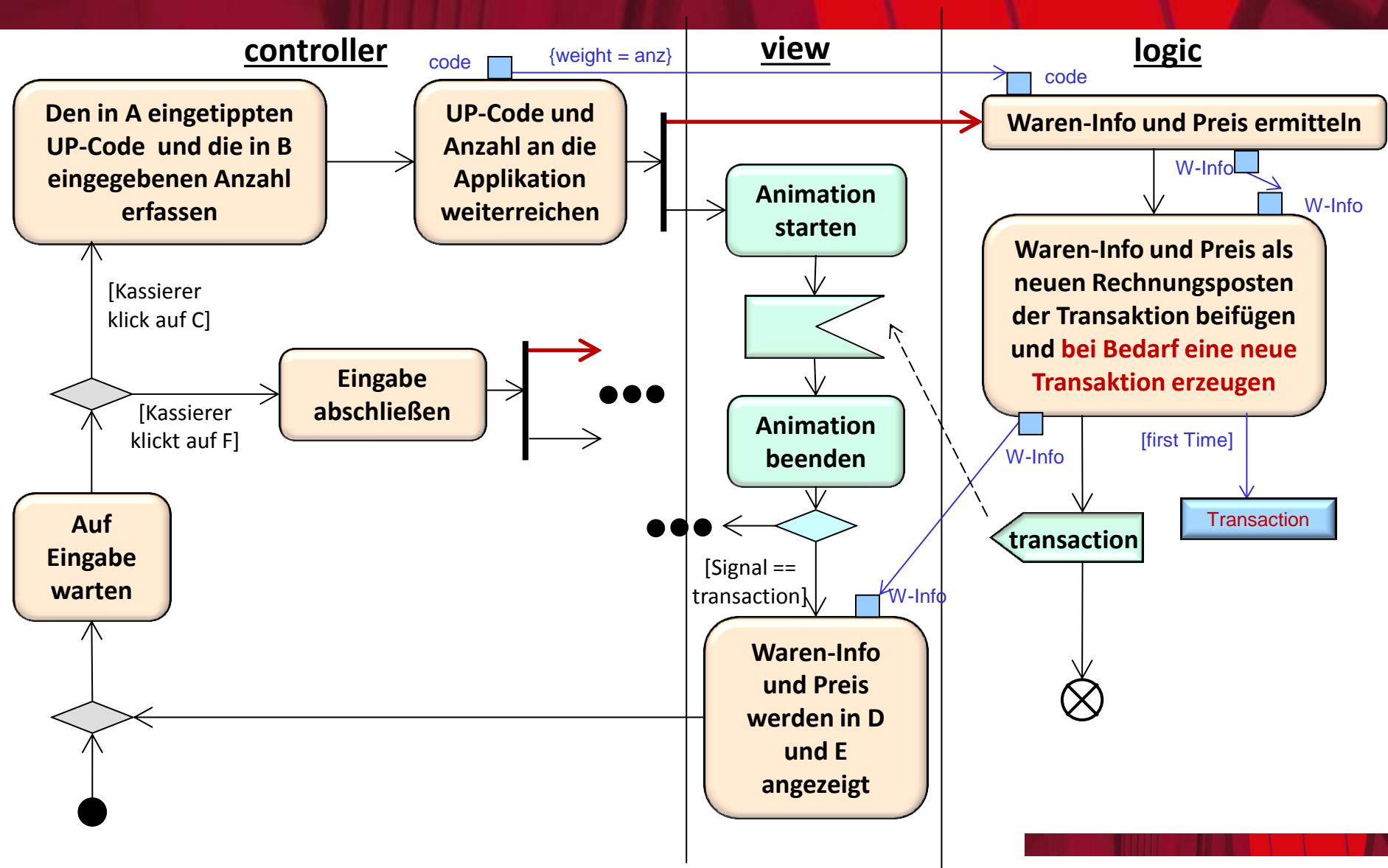
# Use-Case-Controller



- naheliegend
- leicht zu identifizieren
- unverwechselbar



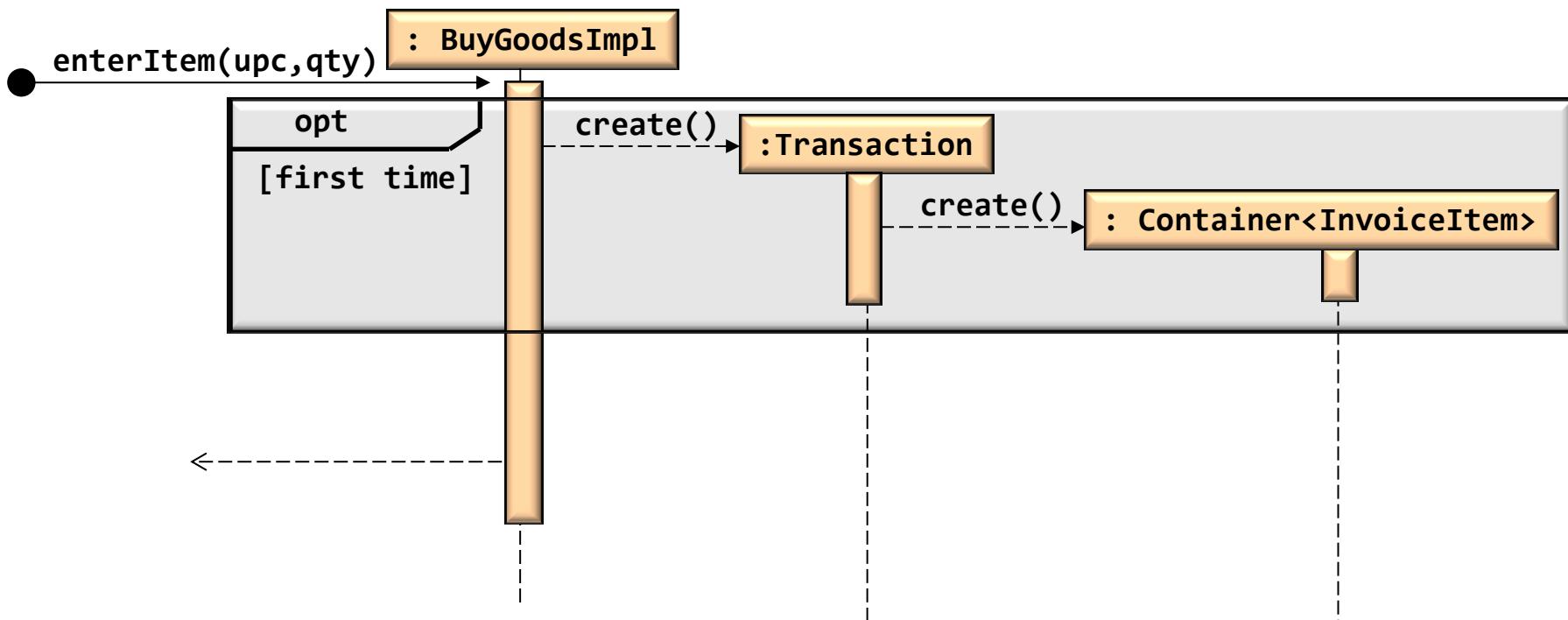
# Waren kaufen „EnterItem“





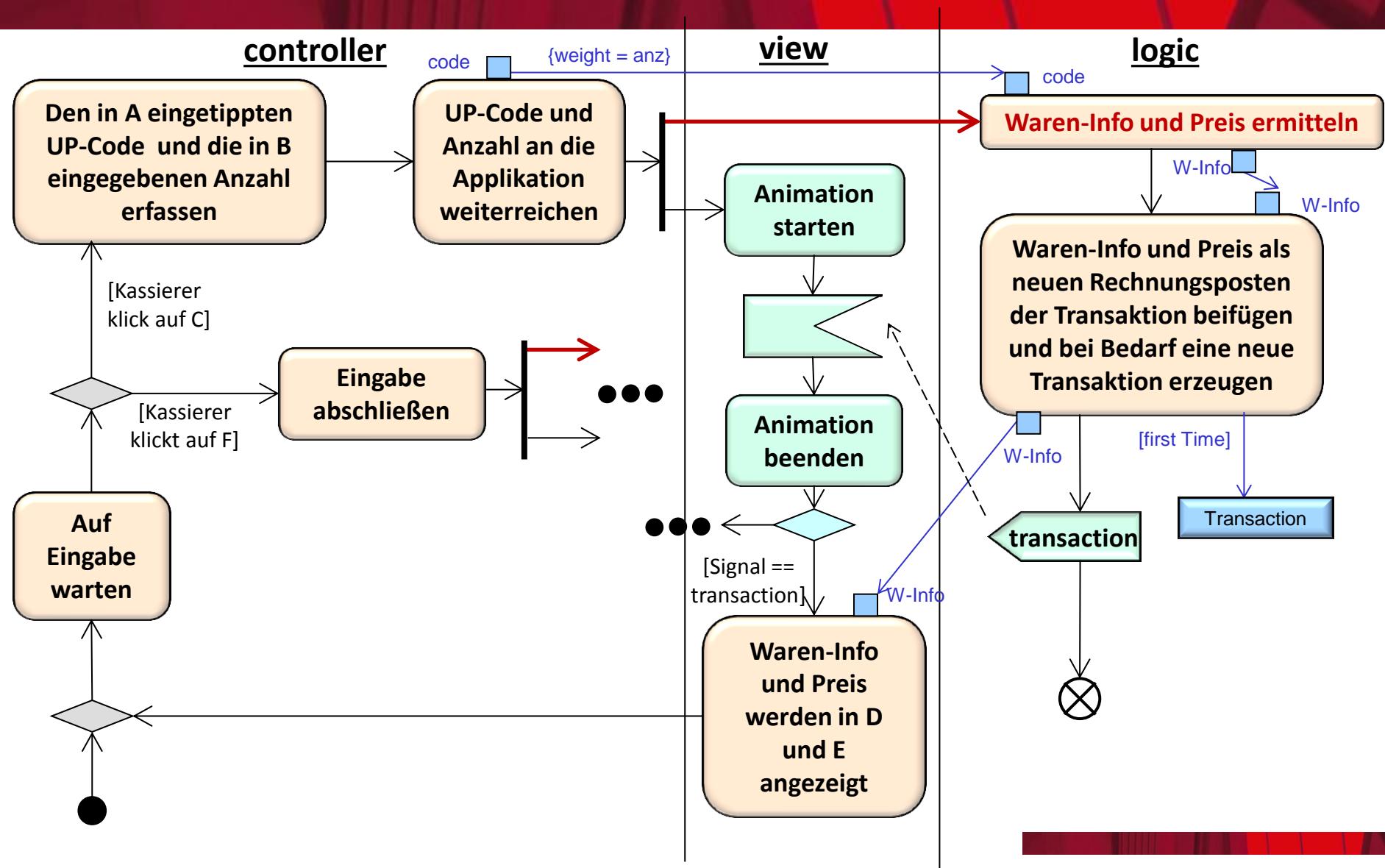
# enterItem() Sequenz-Diagramm (I)

```
sd enterItem(upc: Integer,qty: Integer)
```





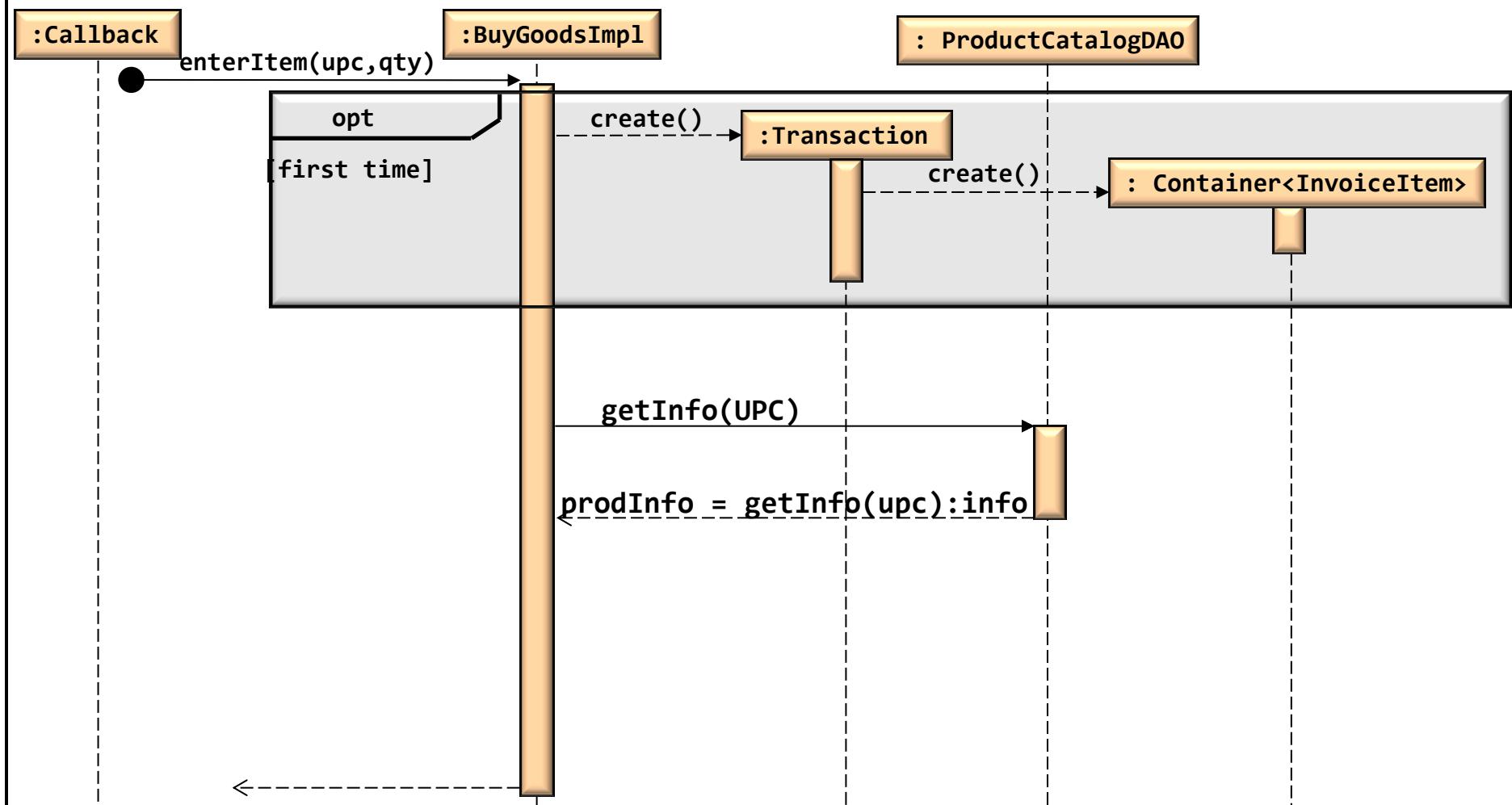
# Waren kaufen „EnterItem“





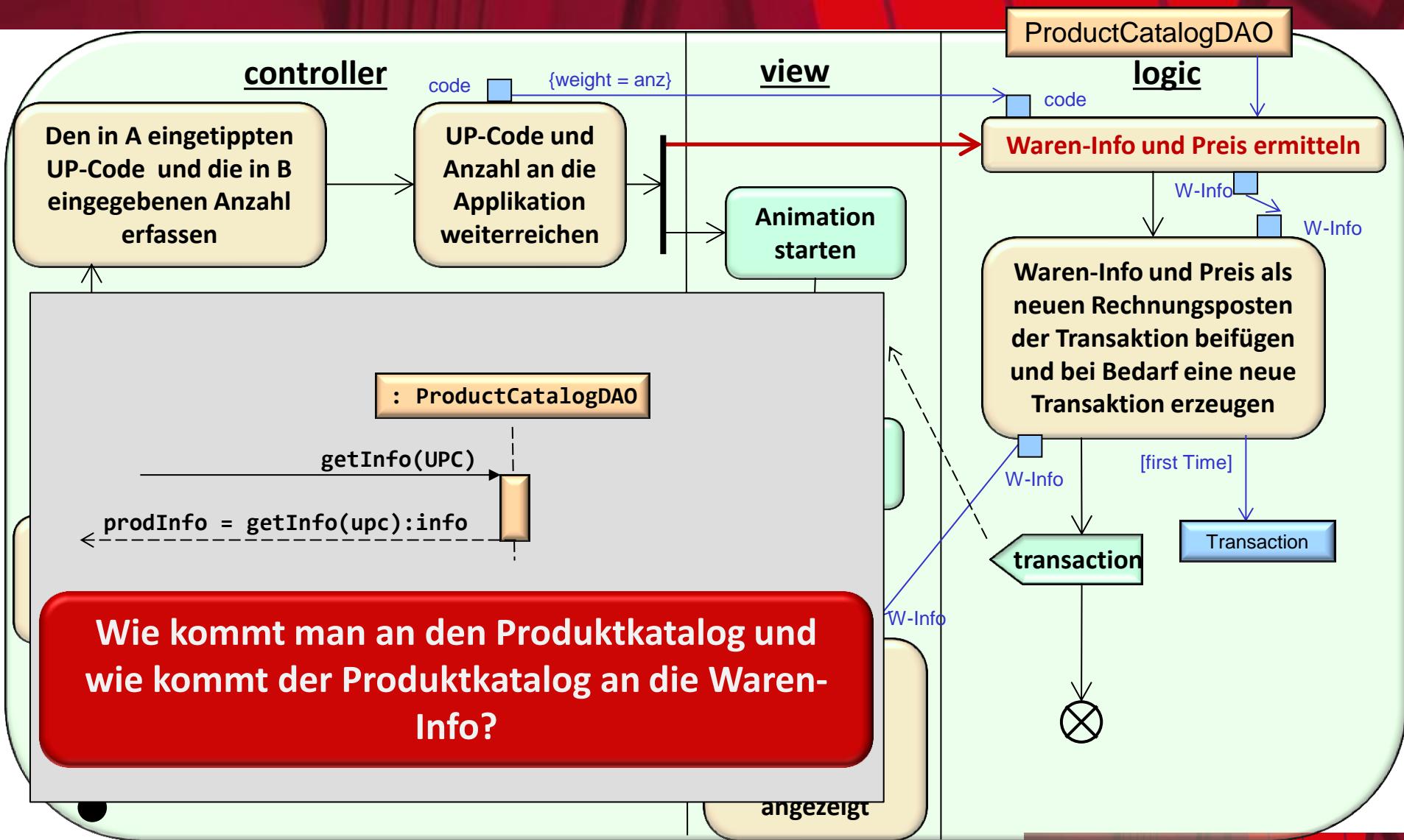
# enterItem() Sequenz-Diagramm (II)

```
sd enterItem(upc: Integer,qty: Integer)
```



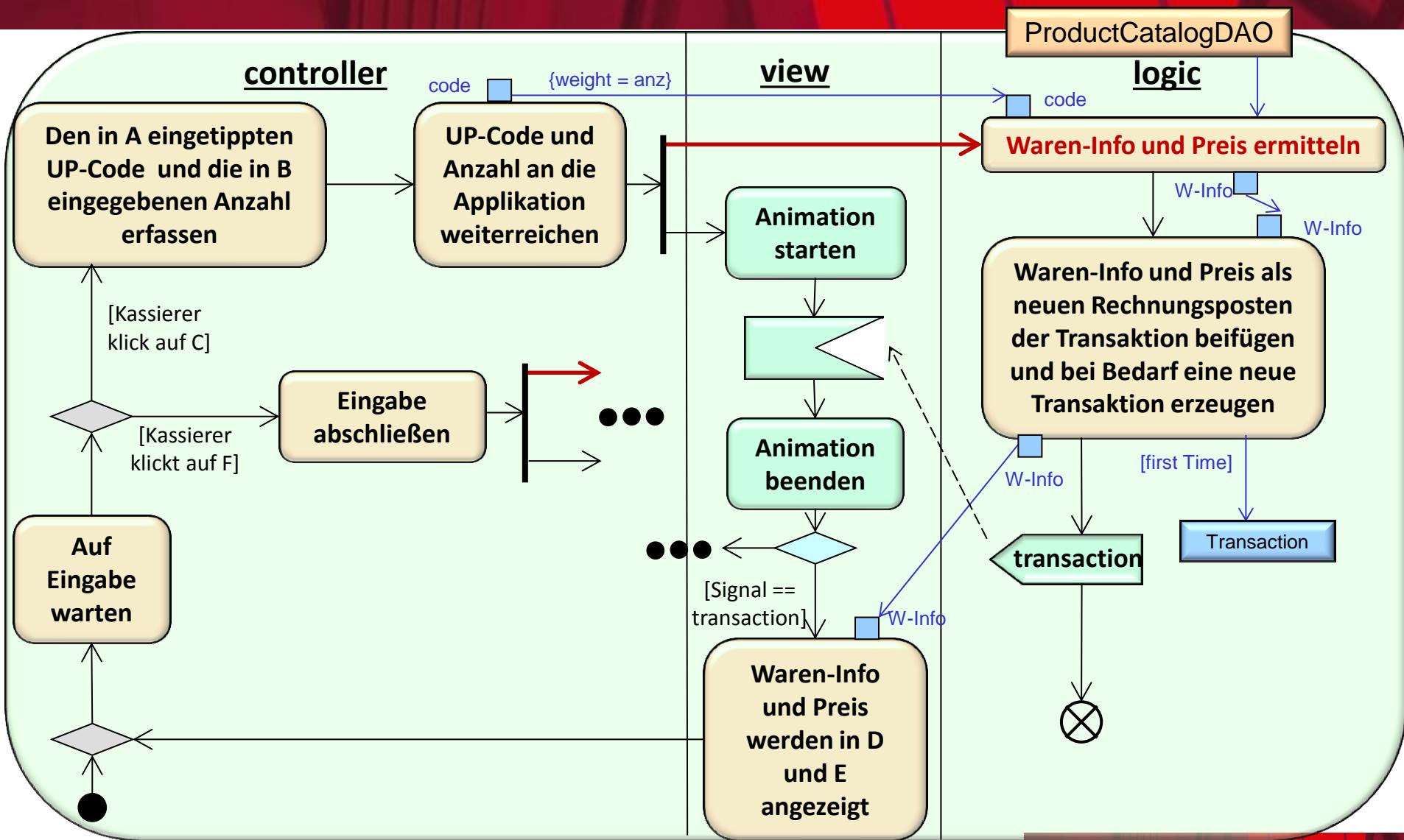


# Waren kaufen „EnterItem“





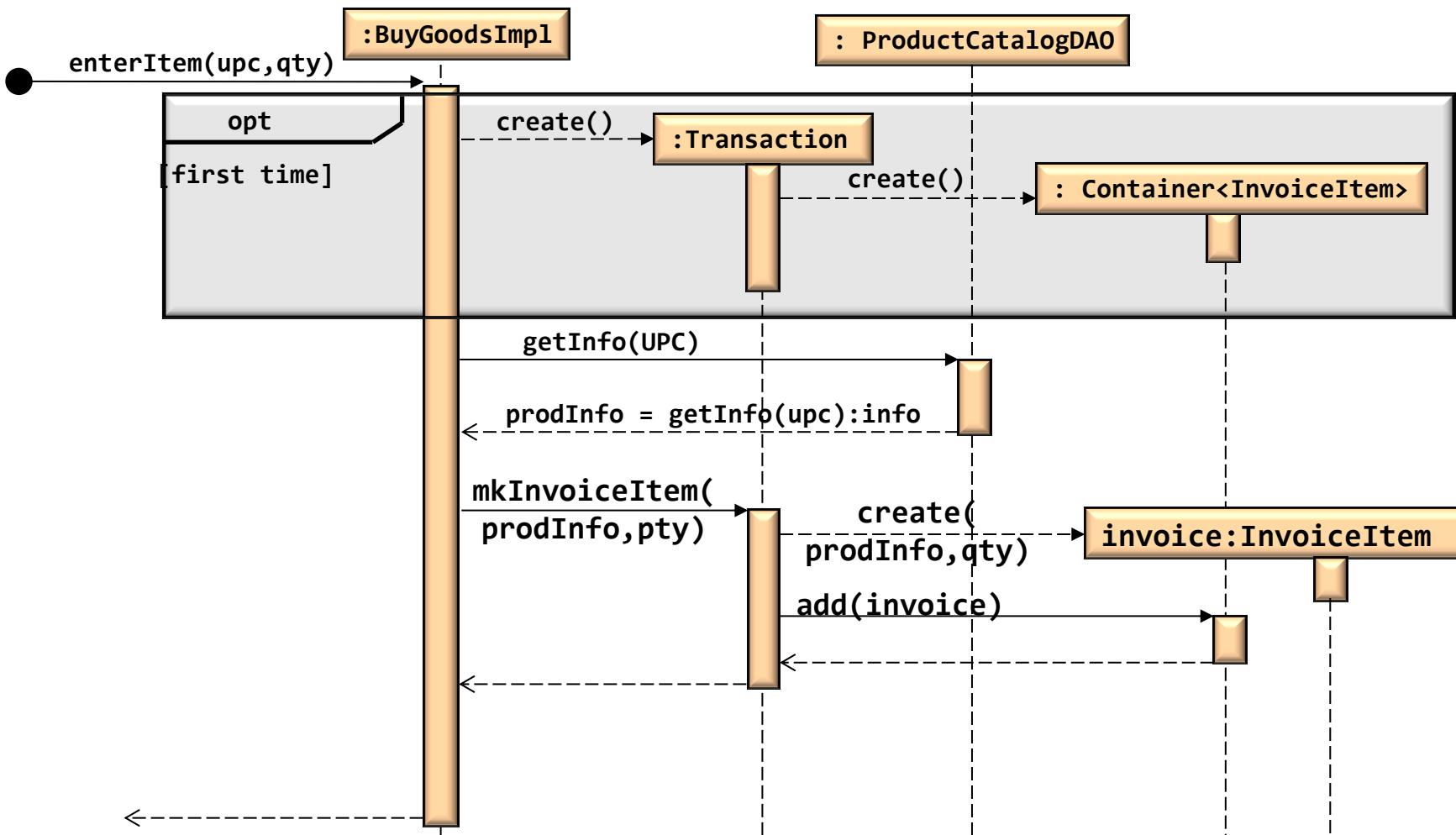
# Waren kaufen „EnterItem“





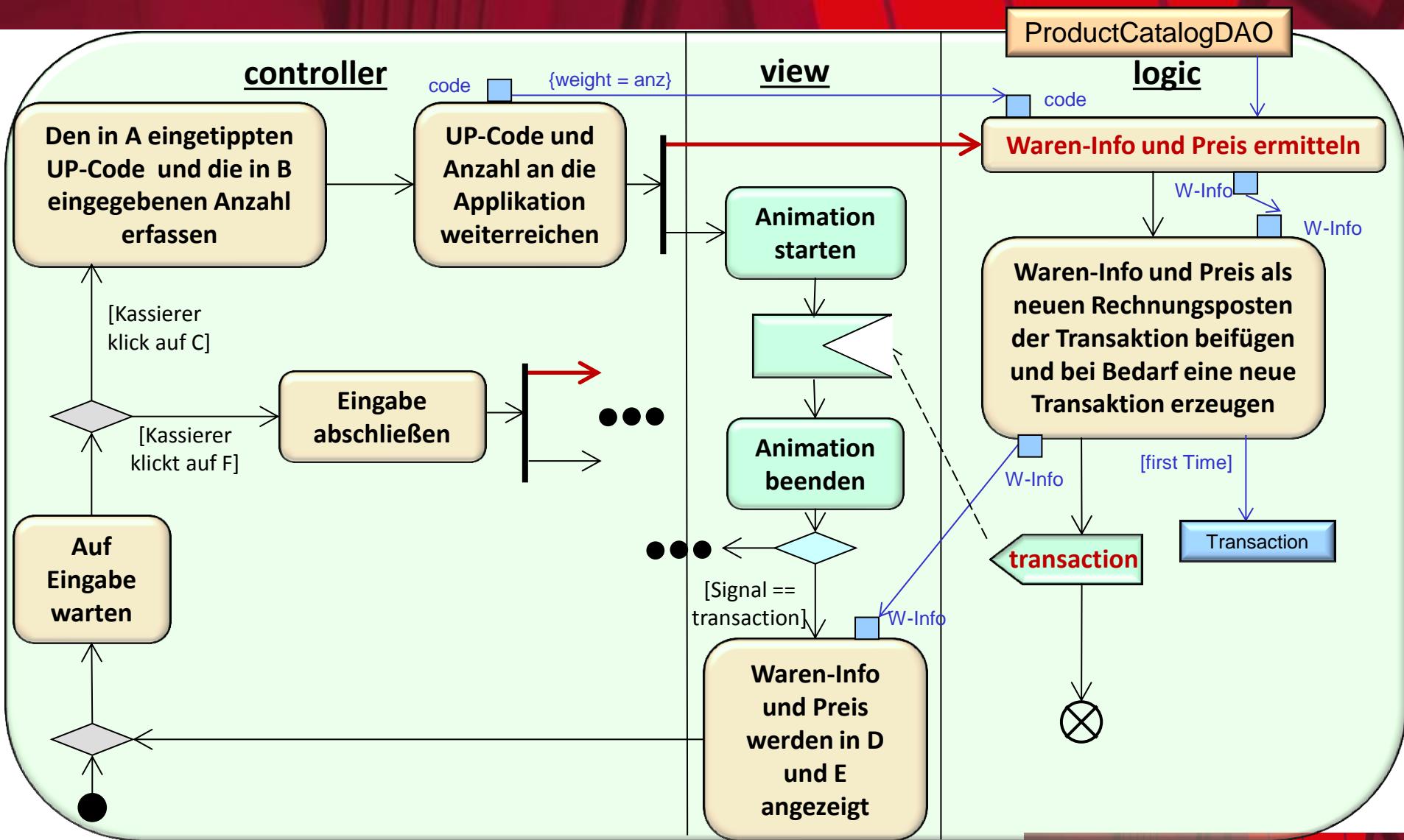
# enterItem() Sequenz-Diagramm (III)

```
sd enterItem(upc: Integer,qty: Integer)
```





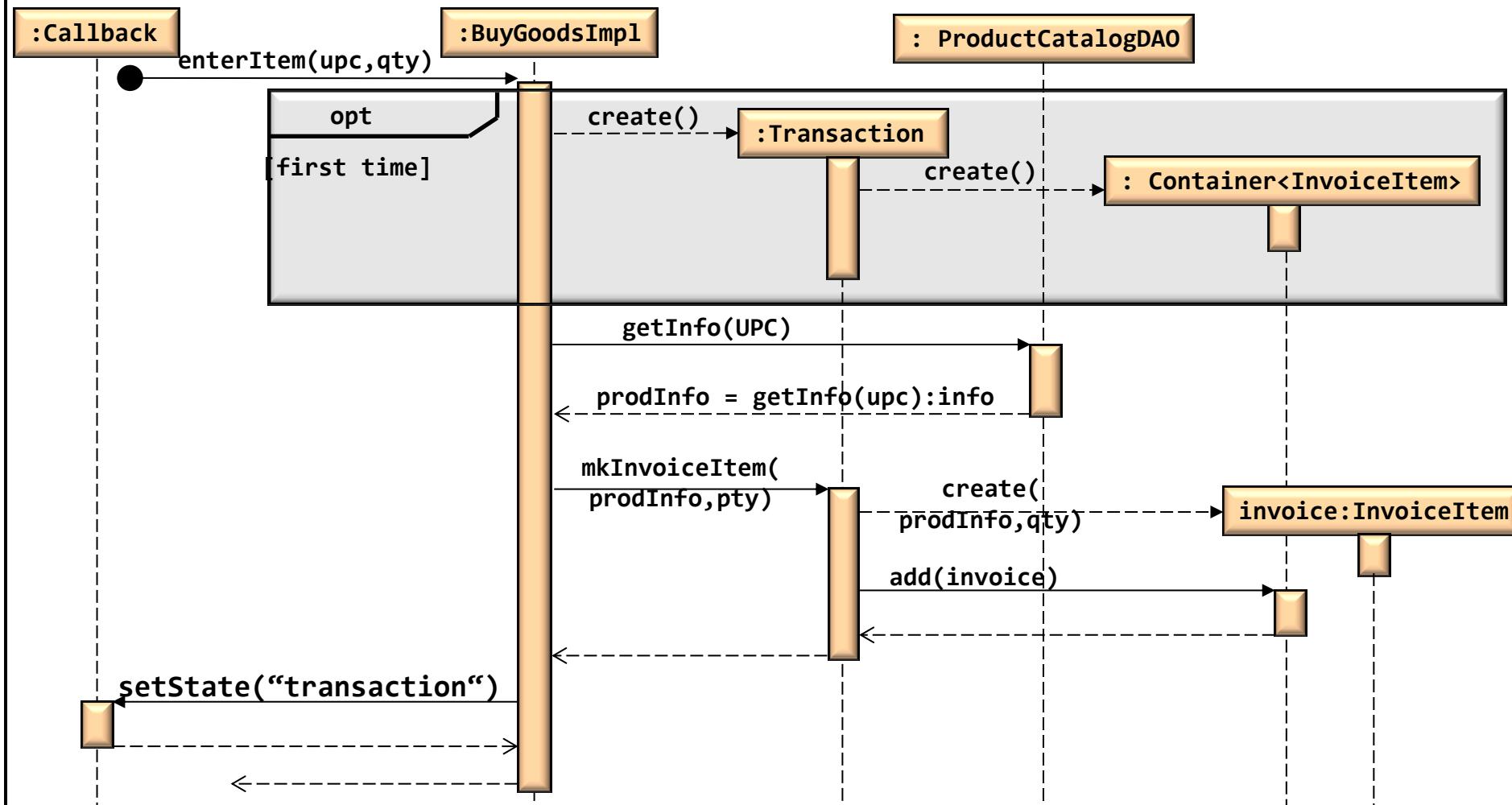
# Waren kaufen „EnterItem“





# enterItem() Sequenz-Diagramm (IV)

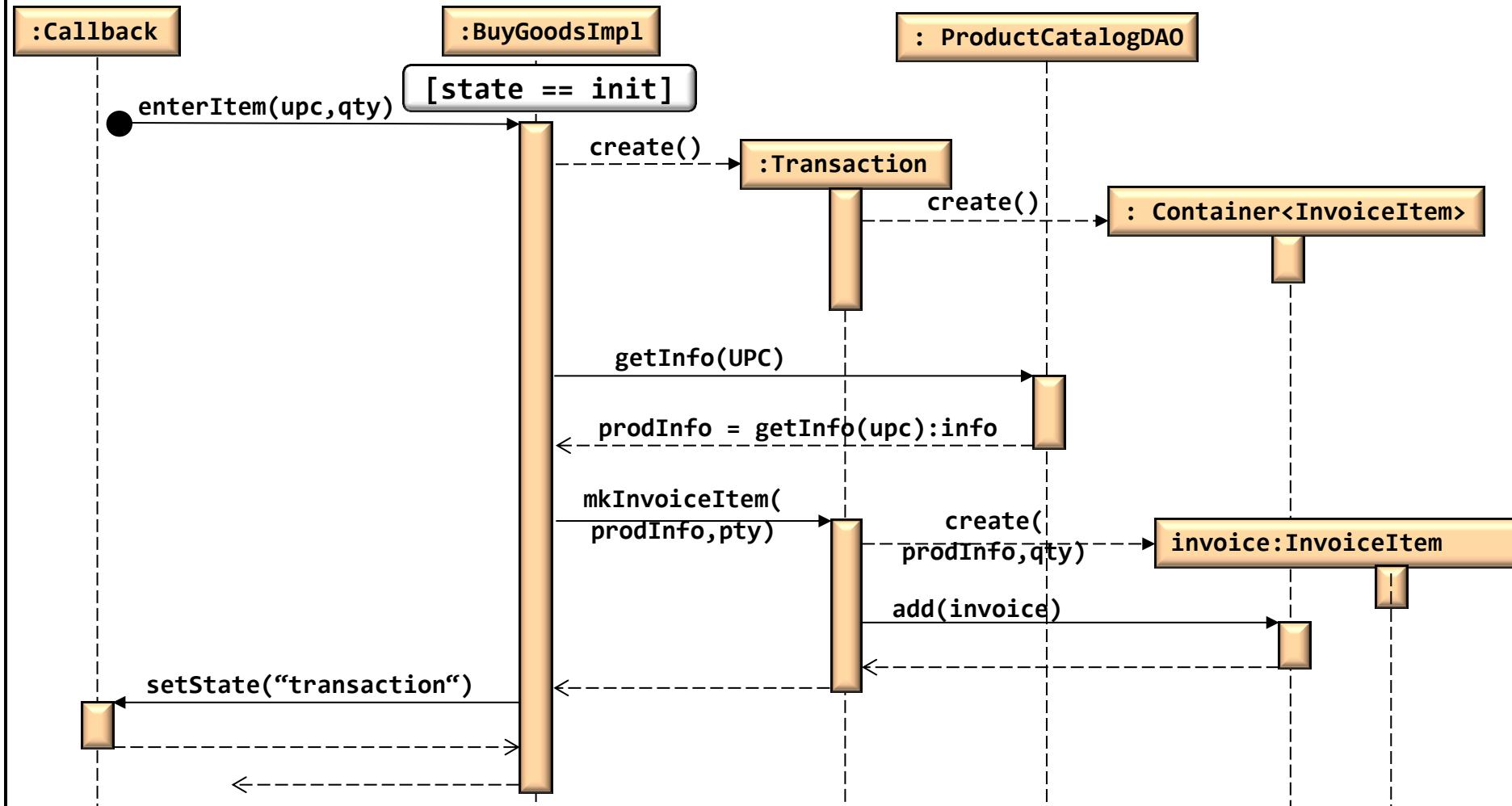
```
sd enterItem(upc: Integer,qty: Integer)
```





# Zustände erzeugen Übersicht

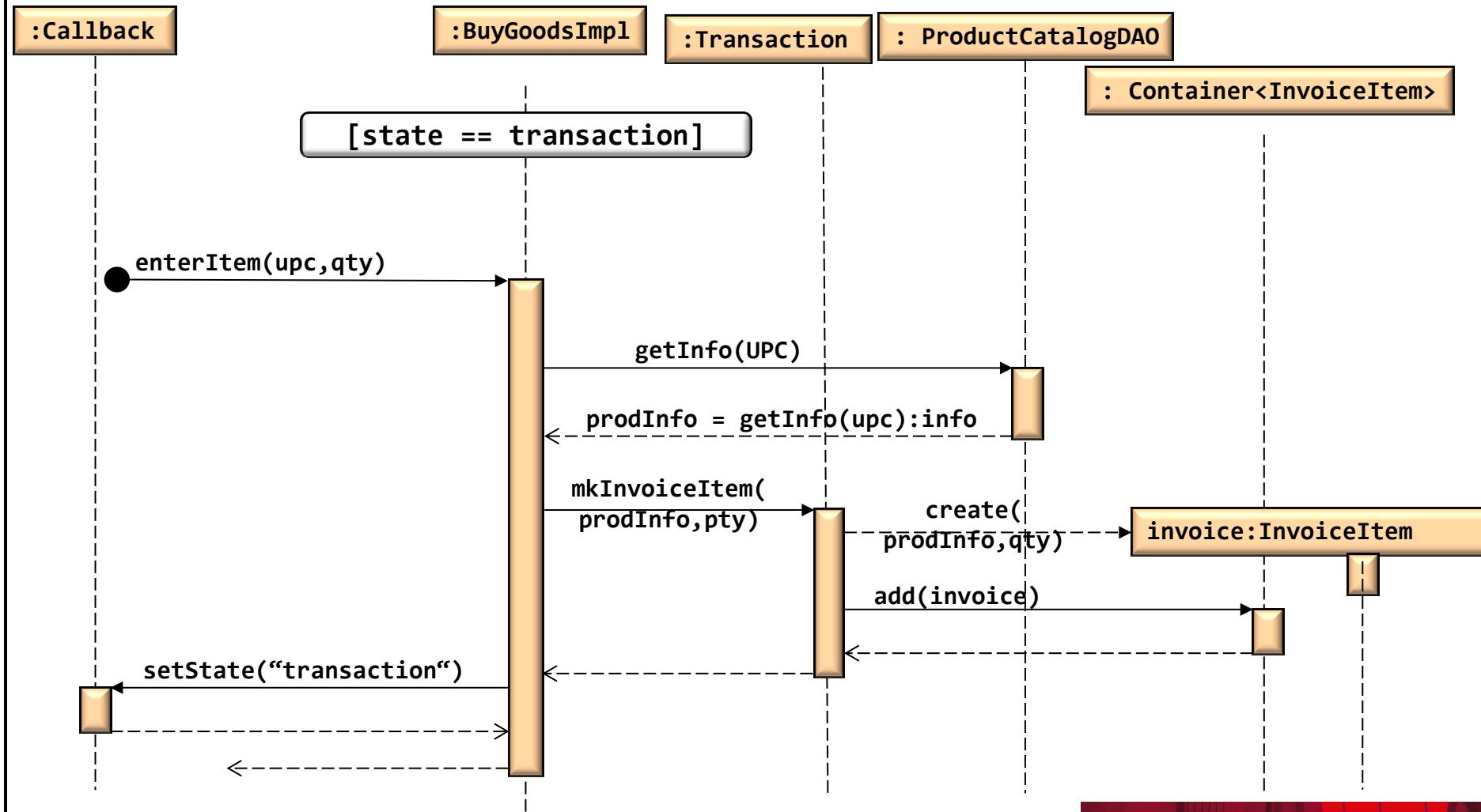
```
sd enterItem(upc: Integer,qty: Integer)
```





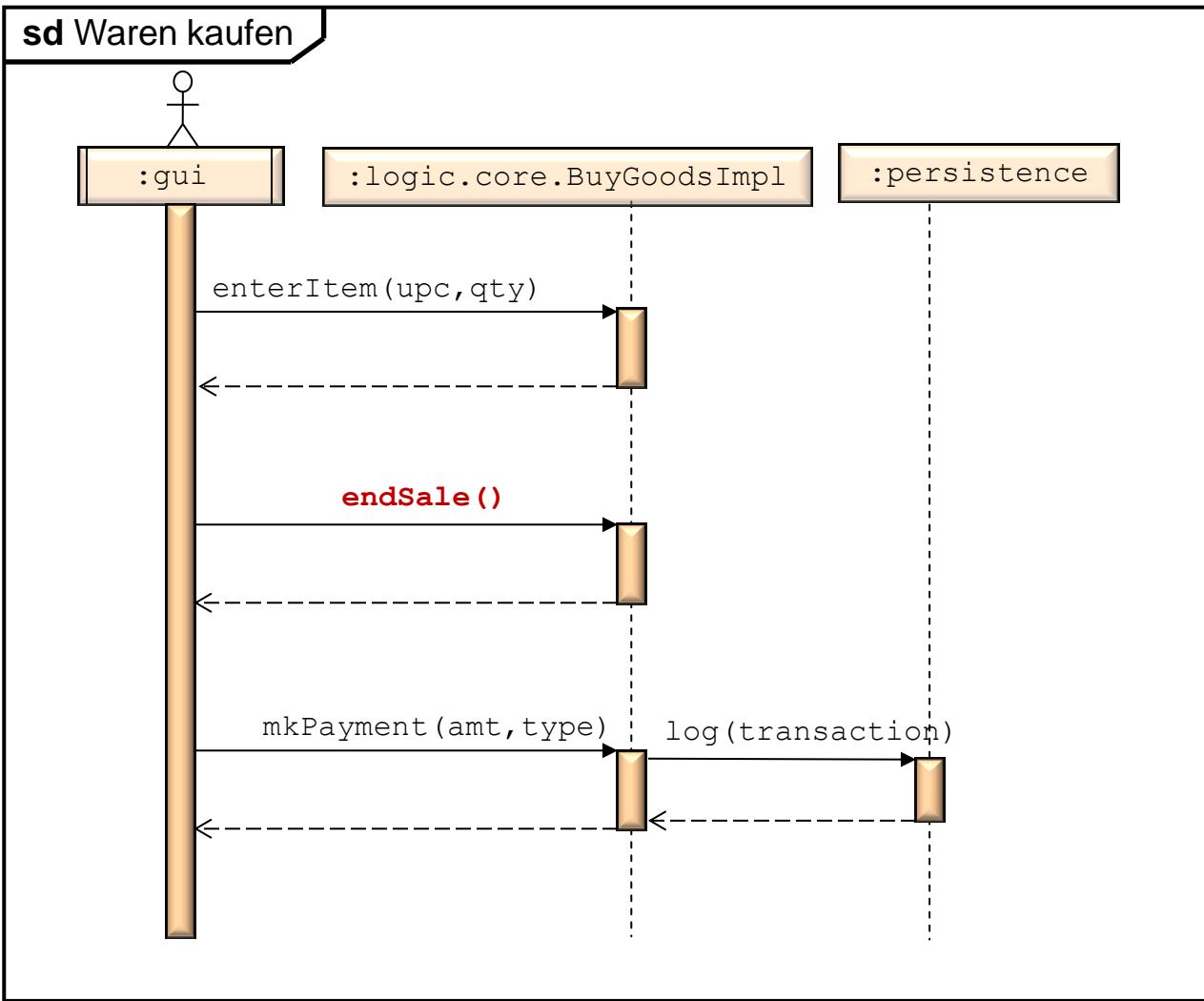
# Zustände erzeugen Übersicht

```
sd enterItem(upc: Integer,qty: Integer)
```



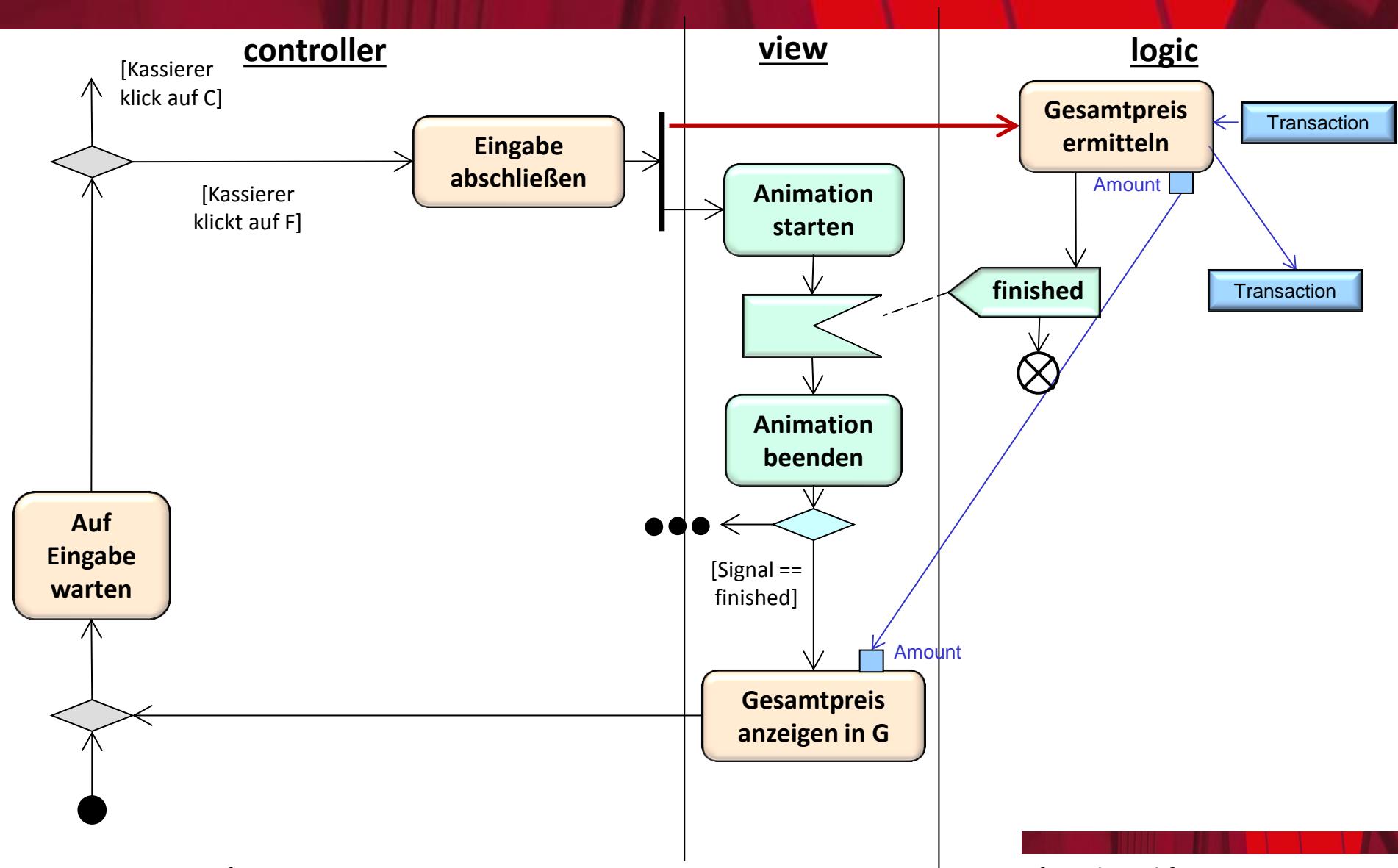


# Nächster Schritt „EndSale“



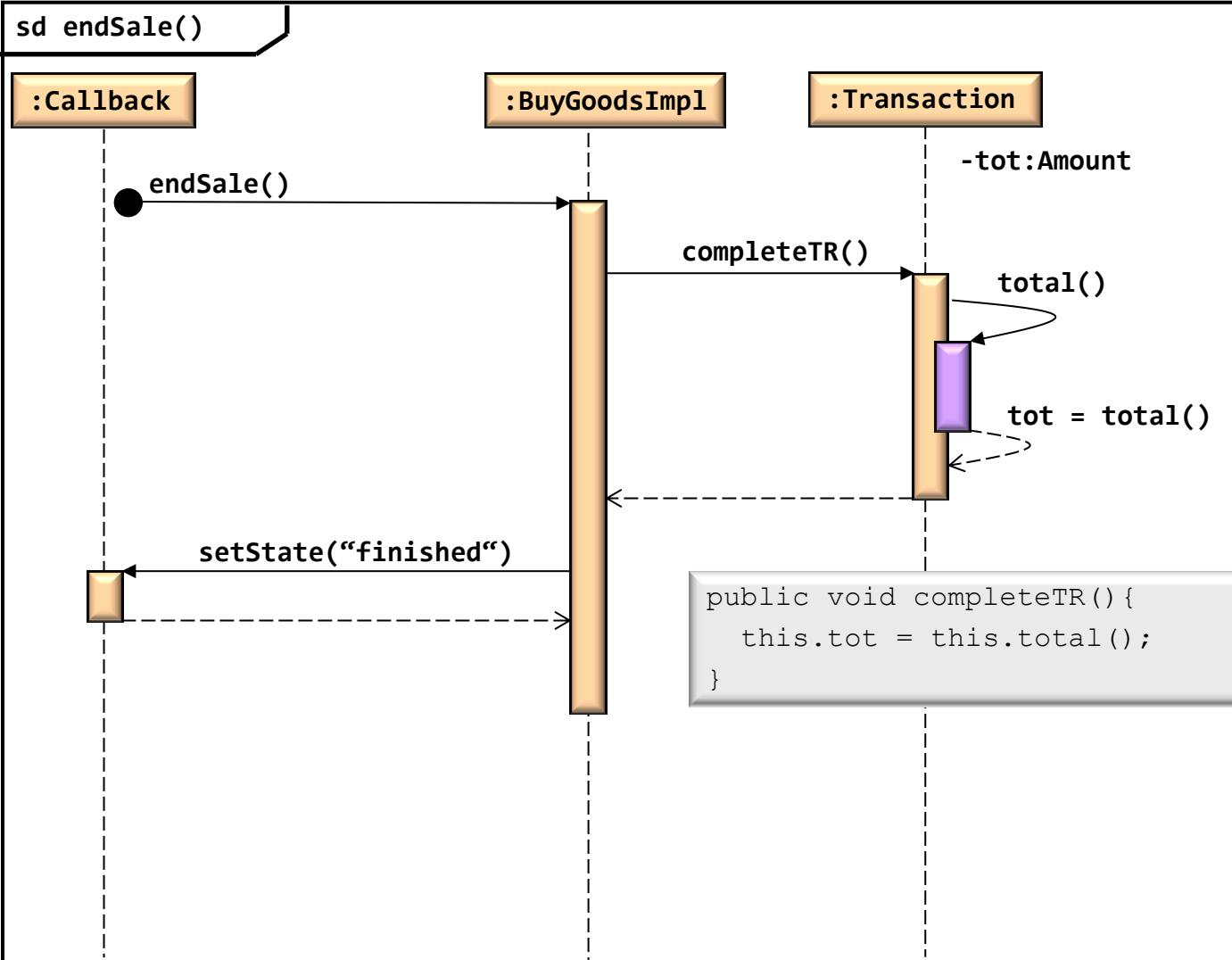


# Eingabe abschließen „EndSale“



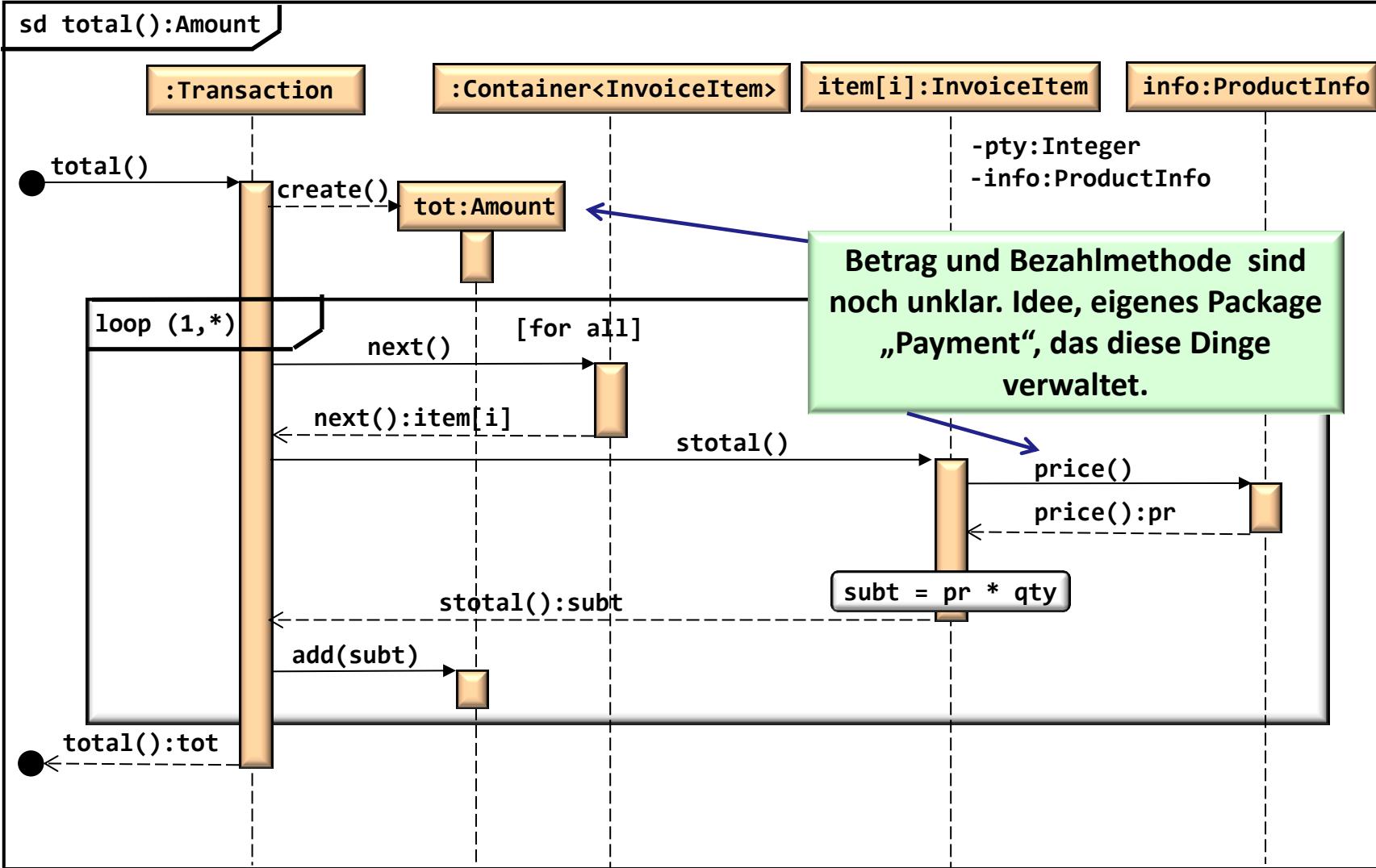


# endSale () Sequenz-Diagramm





# total() Sequenz-Diagramm





# total () Implementierung

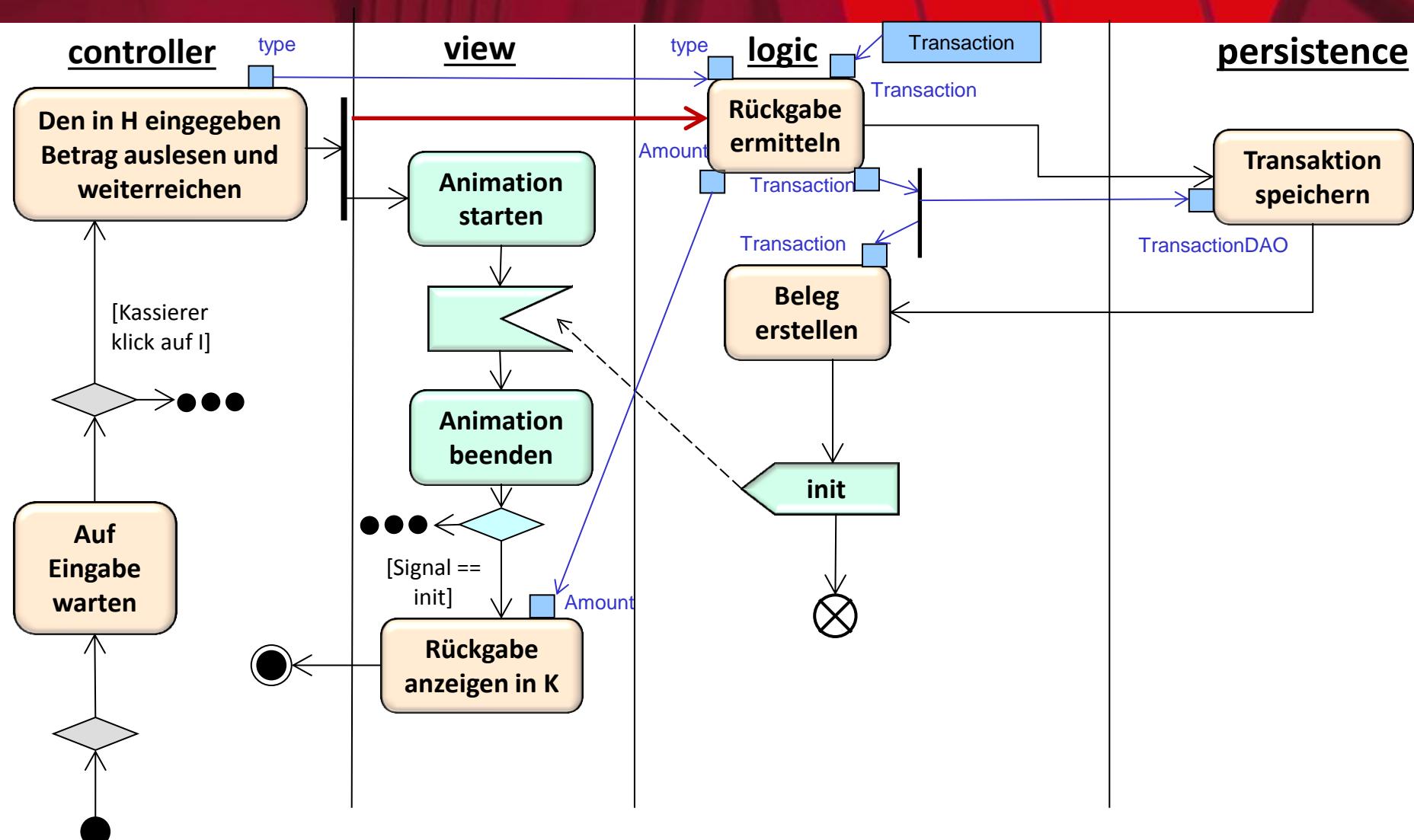
```
private Amount total() {  
    Amount tot =  
        PaymentFactory.paymentFactory.mkAmount();  
    for (InvoiceItem item: invoice)  
        tot.add(item.stotal());  
    return tot;  
}
```

Betrag und Bezahlmethode sind noch unklar. Idee, eigenes Package „Payment“, das diese Dinge verwaltet.

```
public Amount stotal() {  
    Amount pr =  
        PaymentFactory.paymentFactory.mkAmount(this.info.price());  
    pr.mult(this.qty);  
    return pr;  
}
```



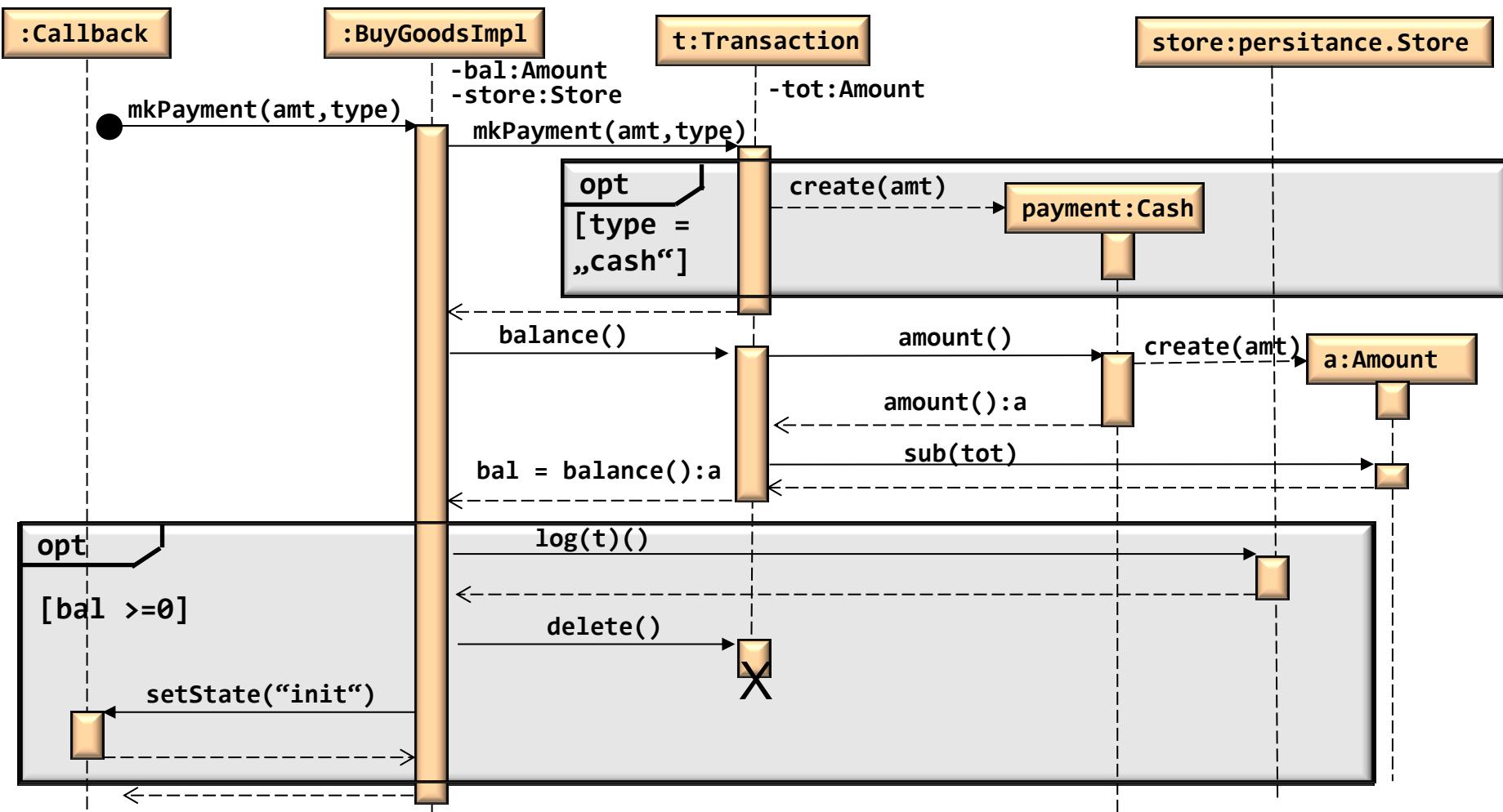
# Betrag eintippen „mkPayment“





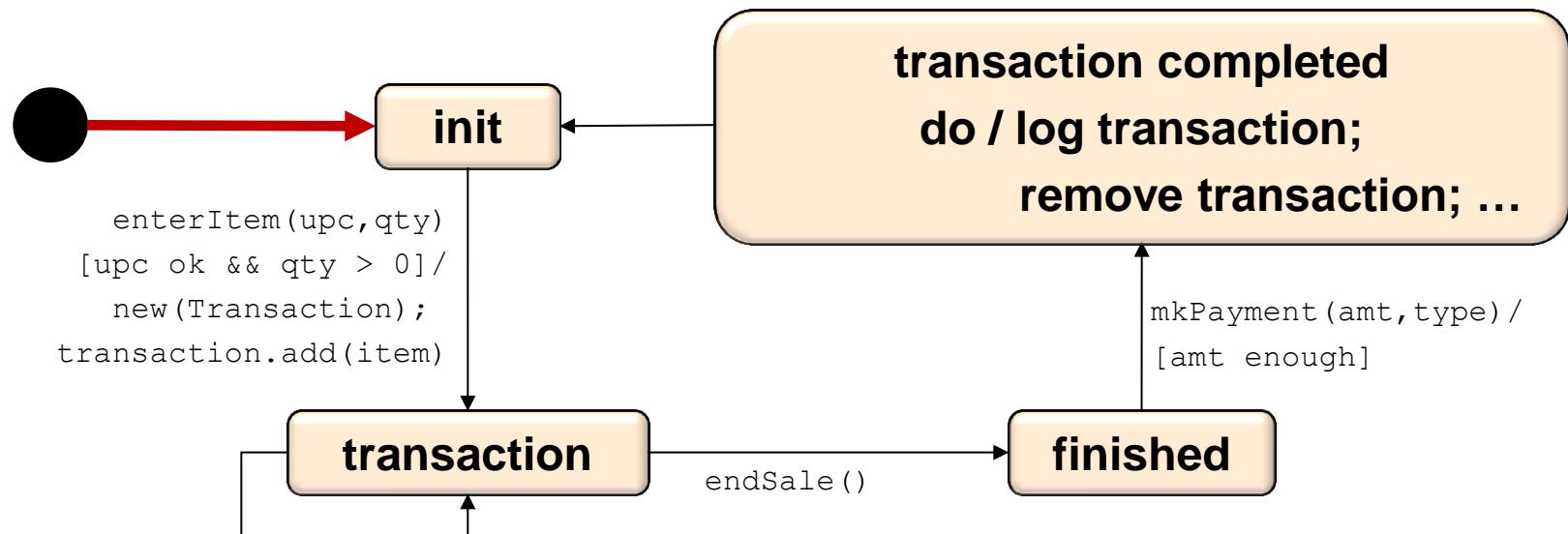
# mkPayment() Sequenz-Diagramm

sd mkPayment(amt:String, type: String)





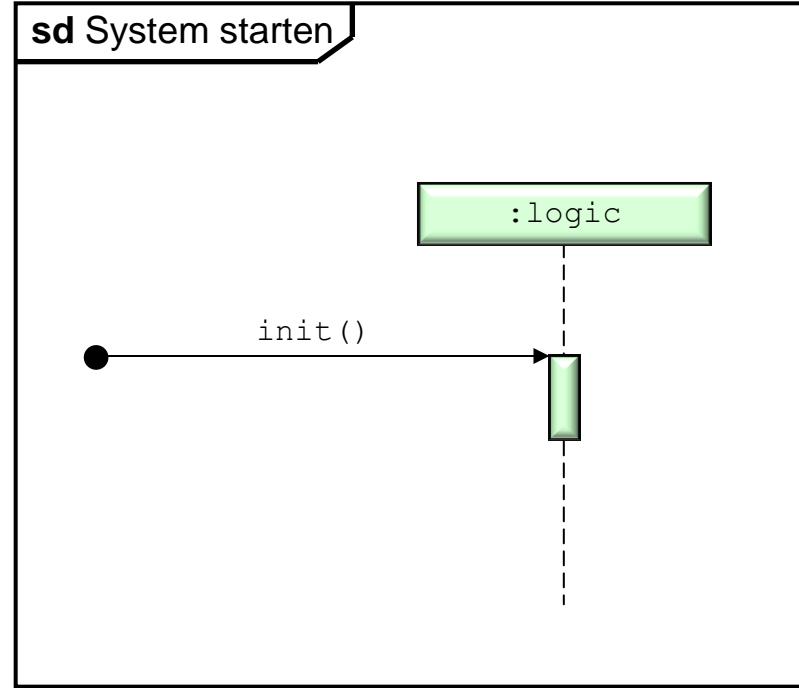
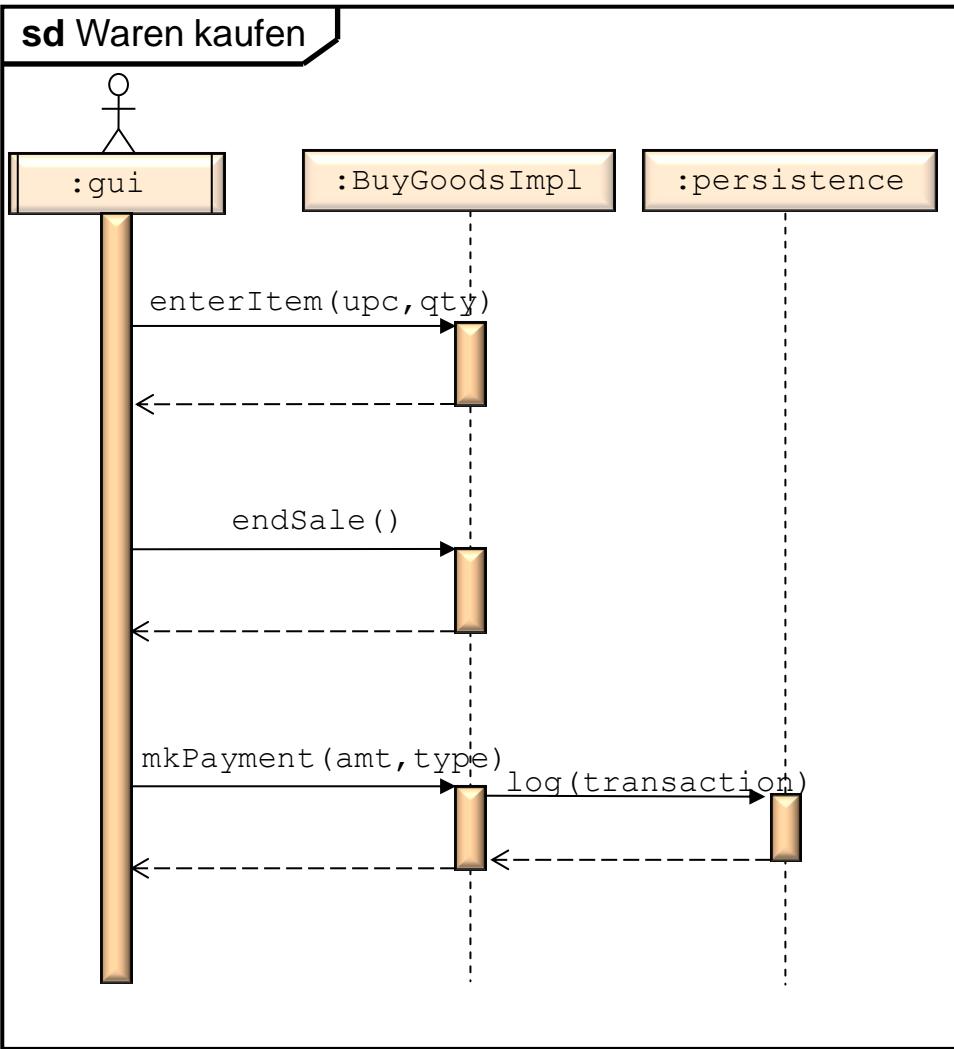
# Haben wir nicht etwas vergessen?



`enterItem(upc,qty) [upc ok && qty > 0] /  
transaction.add(item)`



# System-Sequenz-Diagramm



Neben den identifizierten Operationen benötigt man auch eine Operation für die Initialisierung des Systems.



# Die Systeminitialisierung

**Im Allgemeinen wird diese Funktion zum Schluss designed,  
wenn alle relevanten Informationen vorliegen.**

- **Erster Schritt:**

Das Designobjekt auswählen, das für den Start der Initialisierung und für die Erzeugung der restlichen Domain-Objekte verantwortlich ist.

- **Zweiter Schritt:**

Interaktionsdiagramme erstellen

- In einem Diagramm wird eine `create()` Nachricht an dieses Objekt geschickt.
- (optional) Falls das initiale Objekt auch die Steuerung übernimmt, wird in einem weiteren Diagramm eine `run()` Nachricht an dieses Objekt geschickt.



# Welche Klasse ist geeignet?

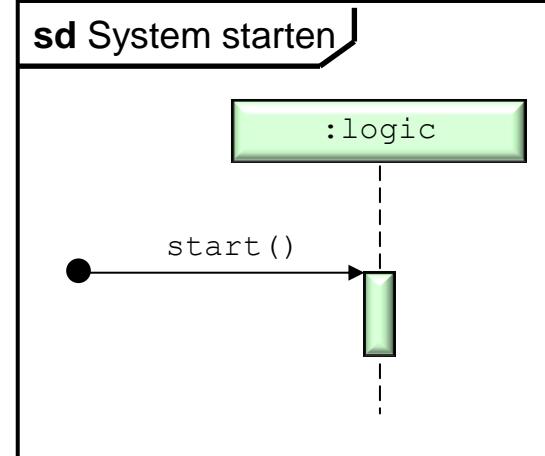
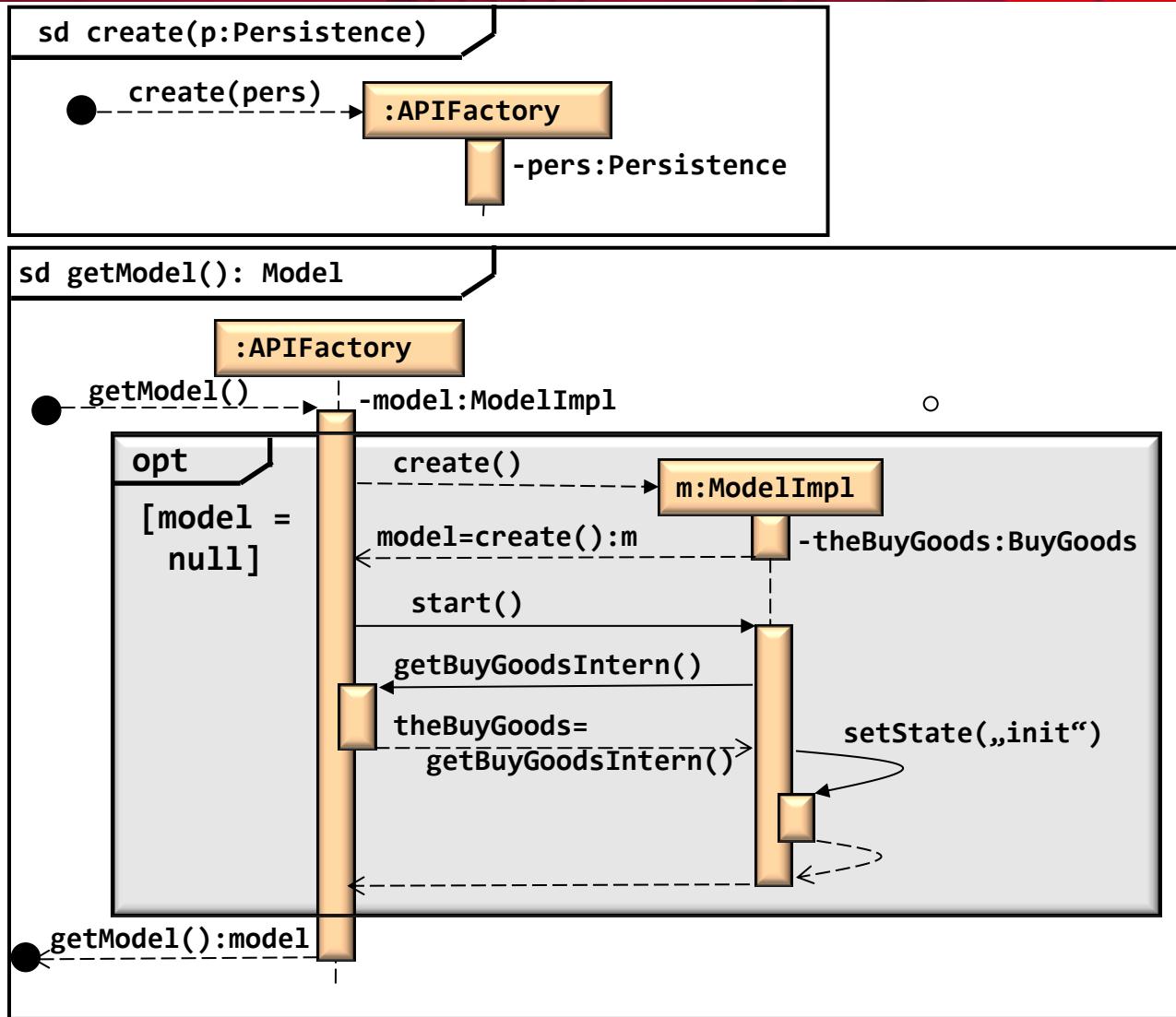
## Vorgehen:

- man wählt eine Klasse, die die komplette Organisationsstruktur repräsentiert oder
- man wählt eine Klasse, die das komplette System (Logik) repräsentiert.
- Man wählt eine Klasse, gemäß einem Erzeugungsmuster
  - Die komplette Organisationsstruktur wird repräsentiert durch:  
**Supermarkt**
  - Das komplette System wird repräsentiert durch:  
**Kasse** oder **Kassensystem**
  - Ein Klasse aus einem Erzeugungsmuster wäre:

**APIFactory**



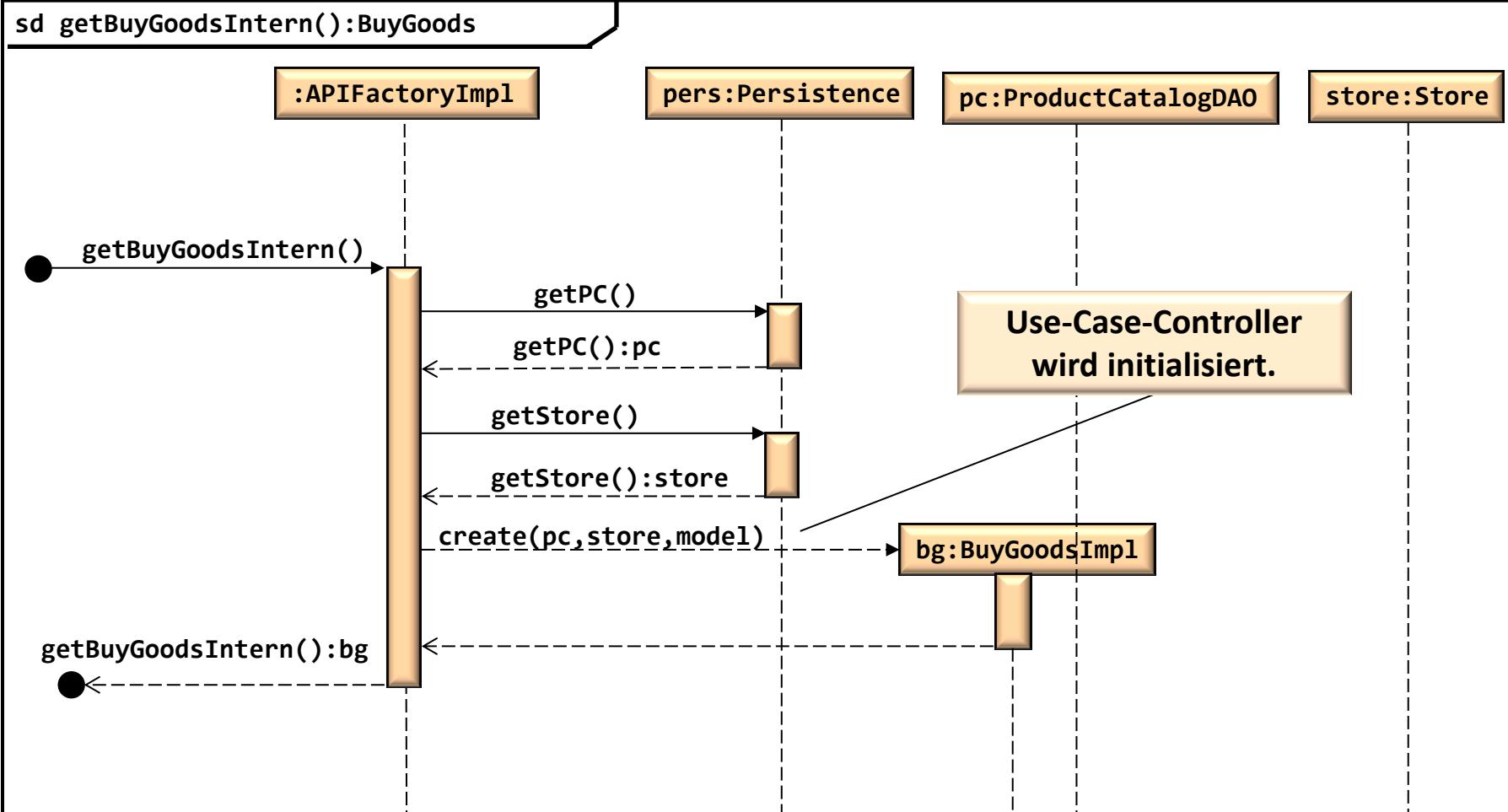
# start() Sequenz-Diagramm



Für `getBuyGoods`  
identischer Ablauf



# Zustände erzeugen Übersicht





# Realisierung (lazy) nach Bedarf

```
class ModelImpl implements Model, BuyGoods,Callback {  
    ModelImpl(APIFactoryImpl factory){ this.factory = factory; }  
  
    void start(){  
        this.theBuyGoods = this.factory.getBuyGoodsIntern();  
        this.setState(BuyGoods.init);}  
}
```

```
class APIFactoryImpl implements APIFactory {  
    public Model getModel() {return this.getModelImpl();}  
    public BuyGoods getBuyGoods() {return this.getModelImpl();}  
  
    private ModelImpl getModelImpl() {  
        if (this.modelImpl == null){  
            this.modelImpl = new ModelImpl(this);  
            this.modelImpl.start();}  
        return this.modelImpl;}  
  
    BuyGoods getBuyGoodsIntern() {  
        if (this.buyGoodsImpl == null)  
            buyGoodsImpl = new BuyGoodsImpl(pF.productCatalog(),pF.store(),modelImpl);  
        return this.buyGoodsImpl;}
```



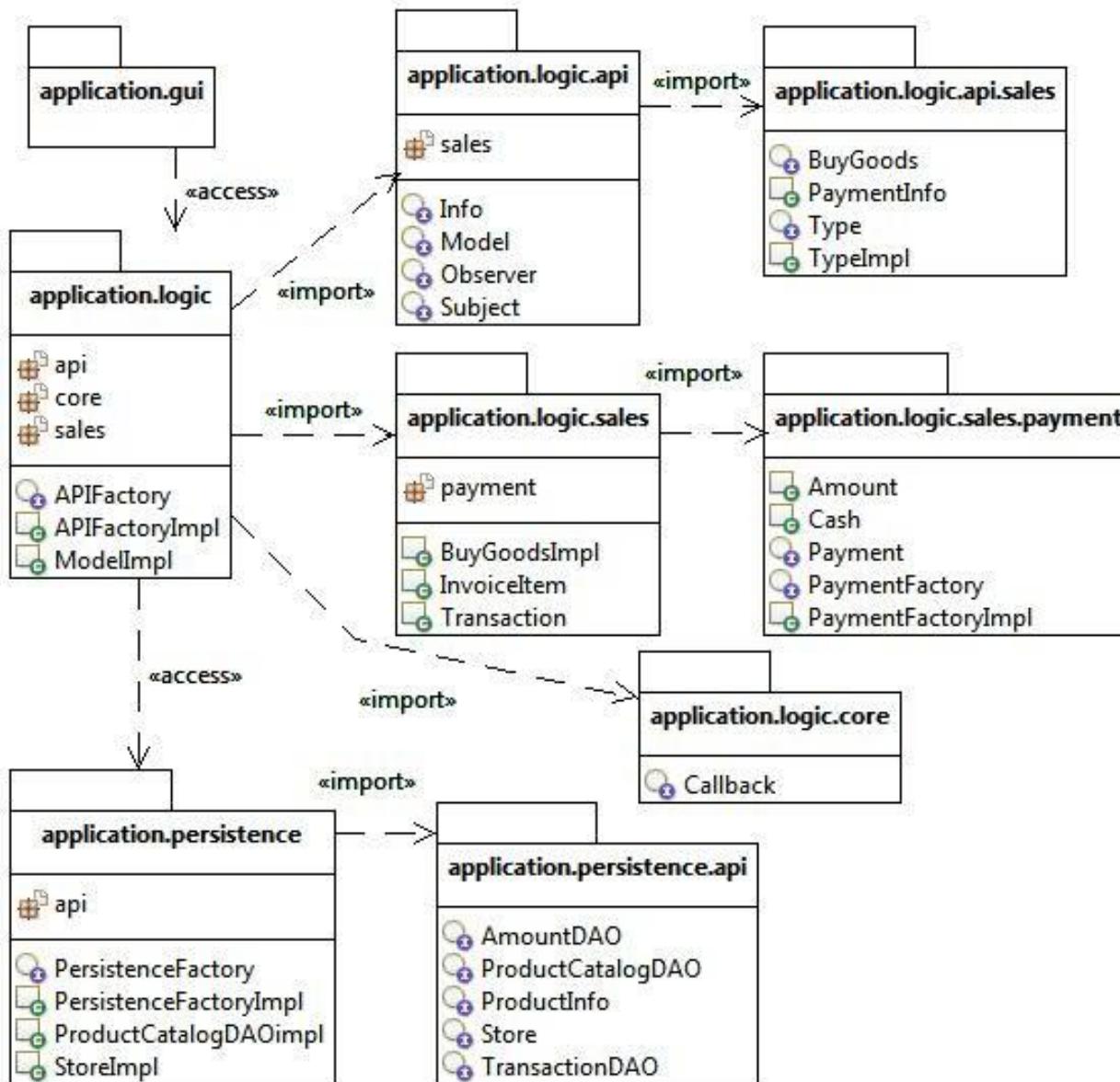
# Das Klassenmodell erstellen

- Neu identifizierte Klassen, Attribute und Methoden werden sukzessive ins Klassenmodell übertragen.
- Das Klassenmodell sowie die Implementierung entstehen parallel zum Design.
- Das Design wird sukzessive um bereits analysierte Use-Cases ergänzt.
- Die Umsetzung neuer Use-Cases beeinflusst stets das bisherige Design.

**Wirken sich Änderungen an bereits bestehenden Implementierungen auf das Design aus, werden sie ins Design übernommen.**

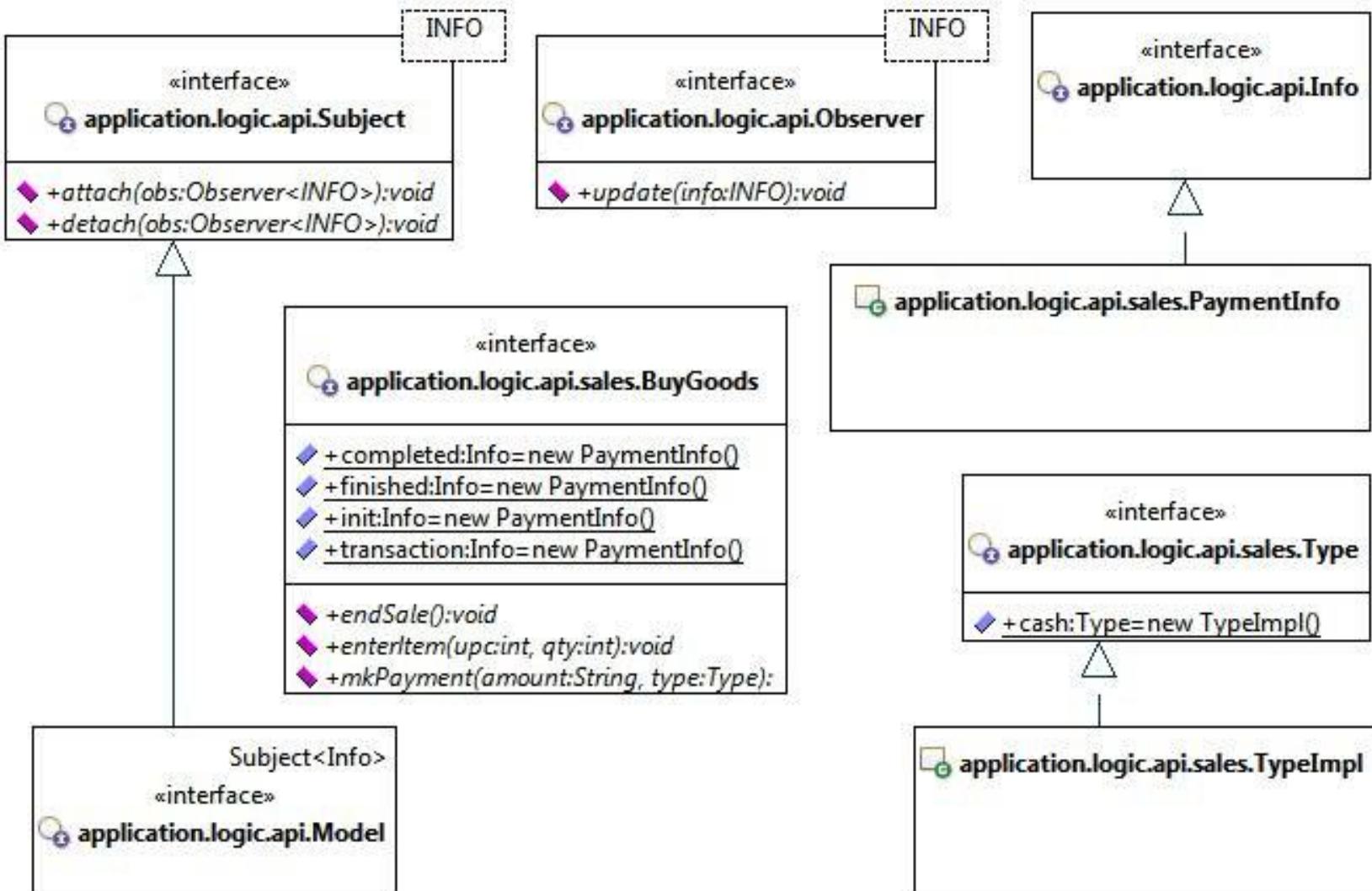


# Das Klassenmodell erstellen



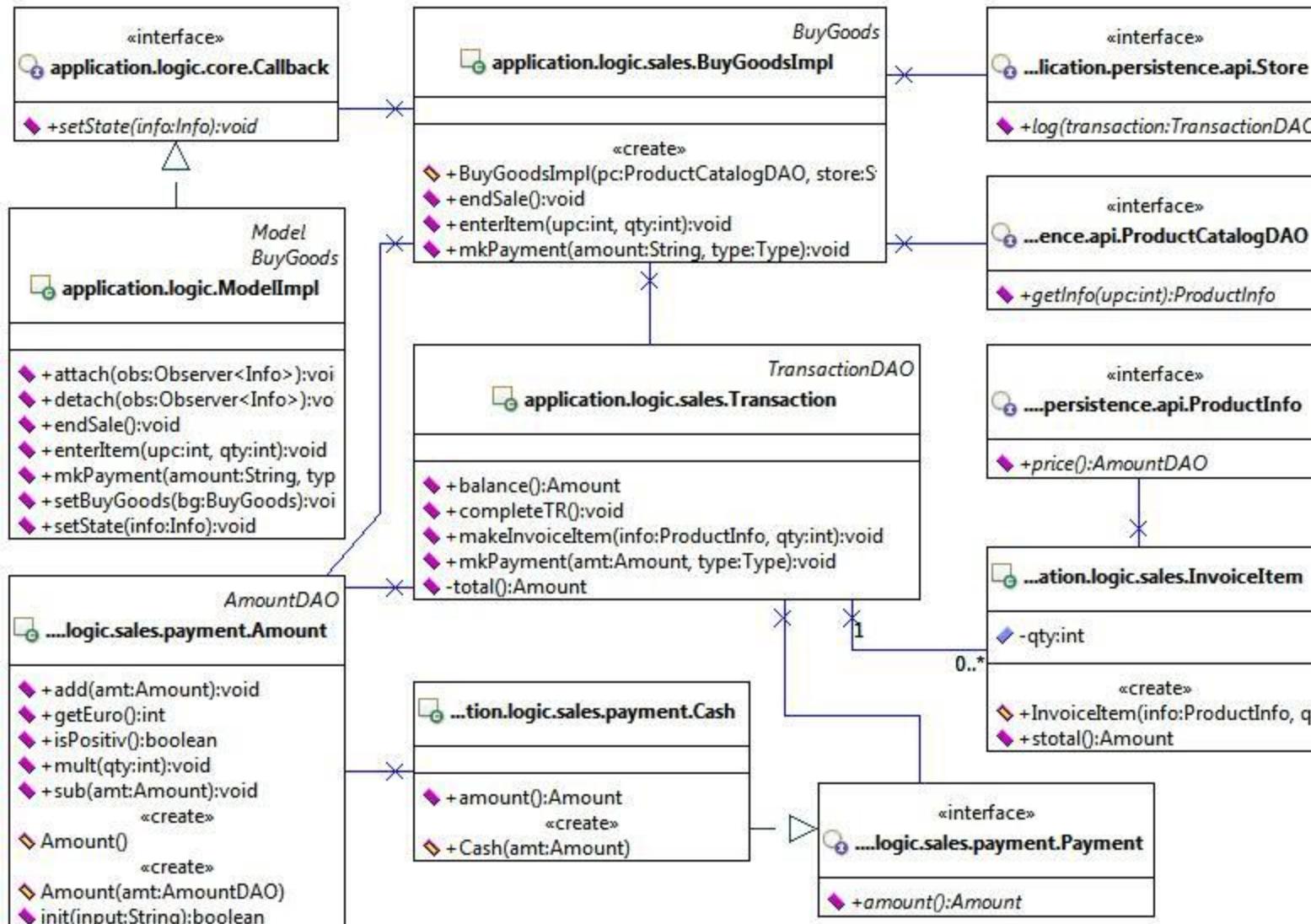


# Das Klassenmodell (die Schnittstelle API)





# Das Klassenmodell (die Logik Sales)



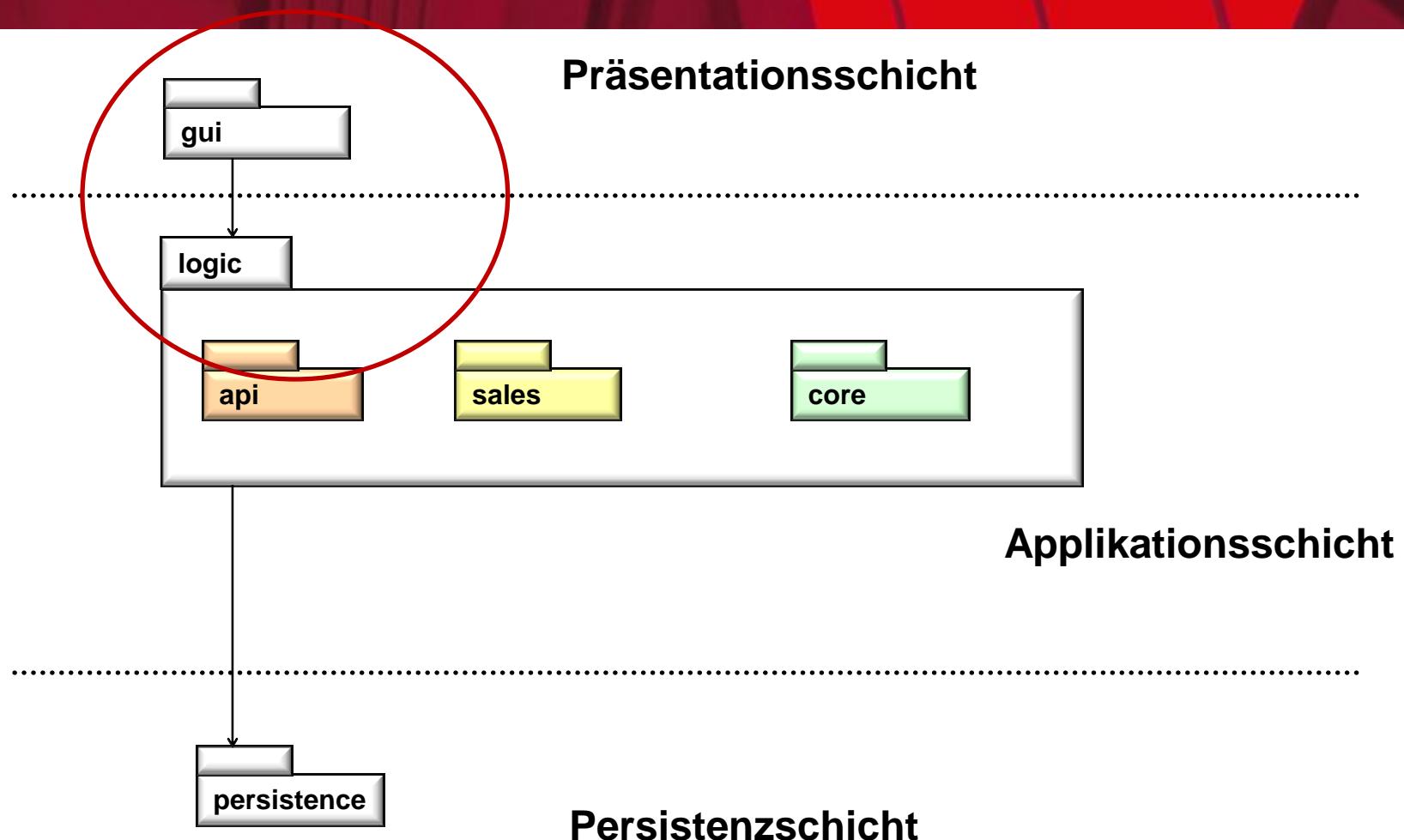


# Design abschließen

- Anbindung der Benutzerschnittstellen
- Anbindung der Speicherverwaltung
- (Einbettung in ein umfassendes Gesamtsystem)
- endgültige Aufteilung in Subsysteme
- weitere Diagramme



# Anbindung der Oberfläche





# Implementierung

## 1. Abtrennen der Mensch-Maschine-Schnittstelle

Die Kernfunktionalität wird im Modell realisiert. ...

erledigt!

## 2. Benachrichtigungsmechanismus implementieren

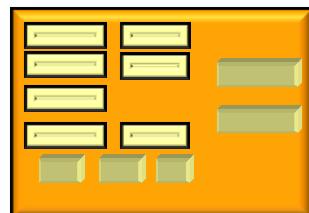
Beobachtermuster umsetzen

```
class ModelImpl implements Model, BuyGoods, Callback {  
    //...  
    public void setState(Info info) {  
        this.currentInfo = info;  
        for (Observer<Info> obs: this.observers)  
            obs.update(this.currentInfo);  
    }  
}
```

erledigt!

## 3. Views entwerfen und implementieren

Mockups vorhanden, Implementierung fehlt

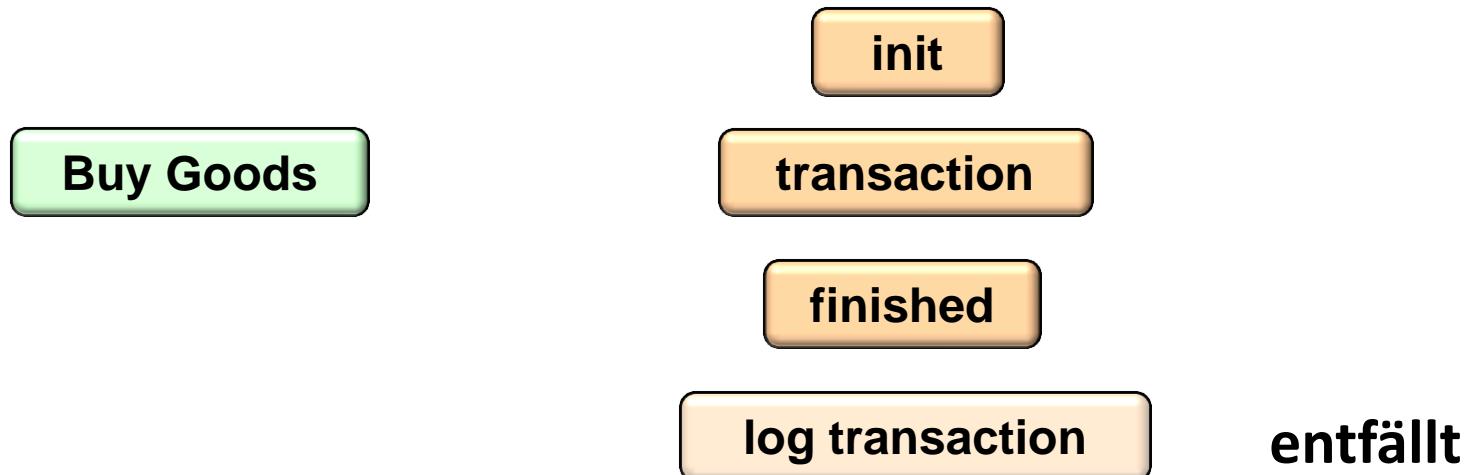




# Views entwerfen und implementieren

Auch bei der Oberfläche orientiert man sich an den Use-Cases.

- Ein Hauptfenster (ggf. einen Splash Screen)
- Für jeden Use-Case mindestens eine View (ggf. weitere Views für differenzierte Zustände)
- Views für Wizards nach Bedarf





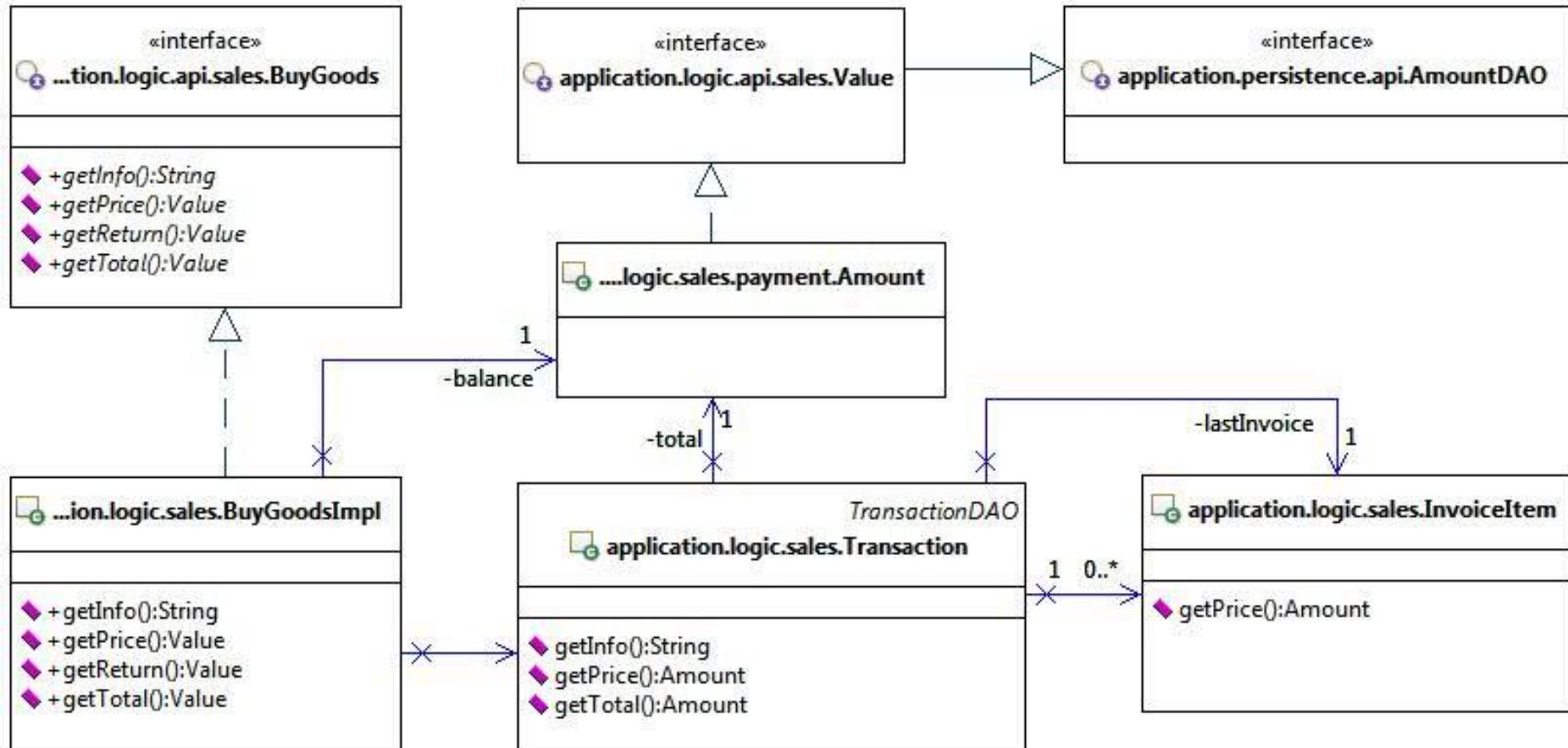
# Getter ergänzen

**Daten, die dargestellt werden sollen,  
müssen aus dem Modell ausgelesen  
werden.**

- entsprechende Operationen deklarieren
- Aufrufe zu den Verantwortlichen weiterreichen.
- ggf. weitere Datentypen vereinbaren.

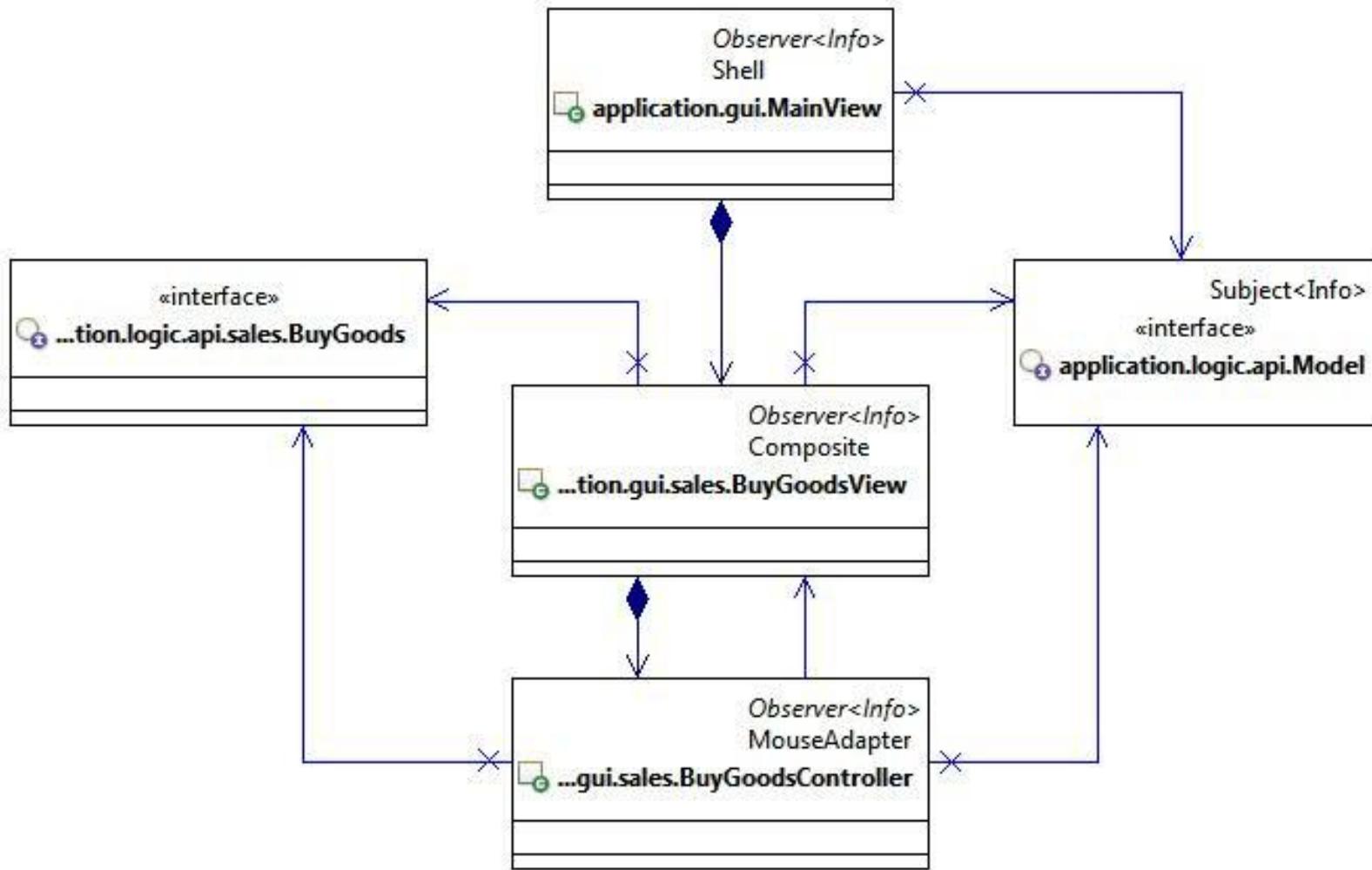


# Getter ergänzen





# Struktur





# Implementierung

## 4. Die Controller werden entworfen und implementiert

Für jede View werden die Bedienmöglichkeiten festgelegt.

erledigt!

## 5. Die Beziehungen zwischen View und Controller

Jede View erzeugt ihren Controller.

erledigt!

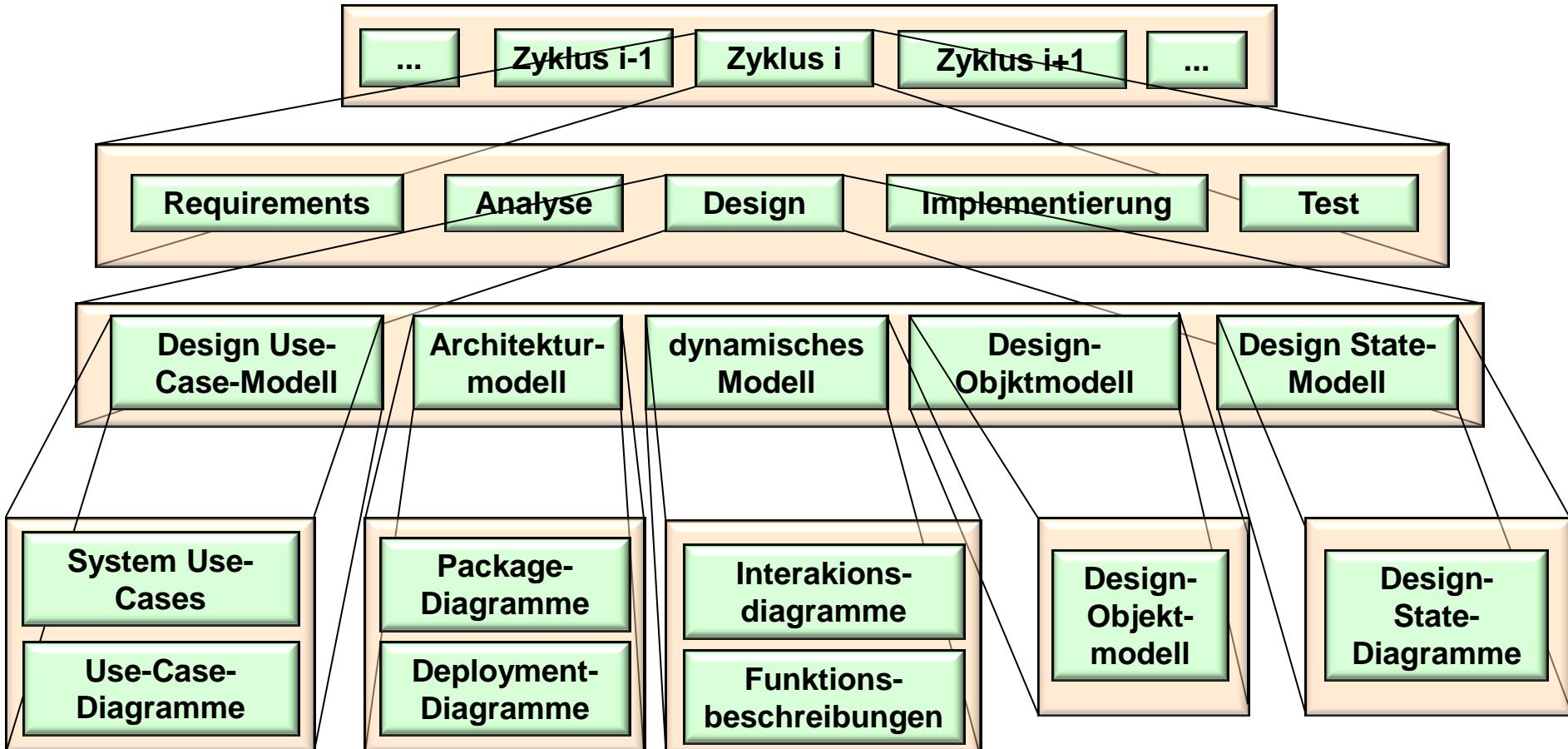
## 6. Initialisierung der gesamten MVC-Struktur

Zuerst wird das Modell erzeugt, danach die Views, diese registrieren sich beim Modell und erzeugen ihre Controller. Ist diese Initialisierung abgeschlossen, wird die Hauptschleife gestartet.

```
public class Main {  
    public static void main(String[] args) {  
        MainView gui = new MainView();  
        gui.startEventLoop();  
    }  
  
    public MainView() {  
        super(Display.getDefault(), SWT.SHELL_TRIM);  
        setLayout(new StackLayout());  
        this.defaultComposite = new Composite(this, SWT.NONE);  
        this.buyGoodsComposite = new BuyGoodsView(this, SWT.NONE);  
        this.createContents();  
        new MainController(this).  
            addControl(this, SWT.Dispose, MainController.CMD_DISPOSE);  
        APIFactory.factory.getModel().attach(this);  
    }  
}
```



# Das Design im Überblick





# Design Zusammenfassung

## Ziel:

Nach der Definition der zentralen Strategien, im Rahmen des Systementwurfs, wurden in der Designphase die Analysemodelle in Designmodelle überführt.

Dies umfasste:

- **Systementwurf (oftmals das Bindeglied zwischen Analyse und Design)**
  - Aufbrechen des Systems in Teilsysteme
  - Strategien für Steuerung und Datenhaltung
- **Objektentwurf (Design)**
  - Anreichern des Analyse-Modells
  - Umsetzung der Strategien des Systementwurfs
  - Implementierungsentscheidungen treffen
  - Algorithmen entwerfen



# Design Zusammenfassung

oder kurz:

## Wie wird die Schnittstelle realisiert?

- Architektur: Das System organisieren und die Zugriffsmöglichkeiten definieren.
- Bedienkonzept: Das Erscheinungsbild der Schnittstelle präzisieren.
- Verantwortlichkeiten zuweisen (der eigentliche Entwurf)
  - Verantwortlichkeiten, etwas zu tun (selbst tun oder delegieren)
  - Verantwortlichkeiten, etwas zu wissen (speichern, berechnen, assoziieren)
- Reorganisation der Objekte

Die Operationen der Schnittstelle entwerfen.



# Vom Design zum Code

**Die Programmierung ist nie trivial. Das Ergebnis der Design-Phase ist in der Regel ein unvollständiger erster Schritt auf dem Weg zur vollständigen Implementierung.**

**Ein gutes Design ist jedoch die Grundlage für eine tragfähige und skalierbare Realisierung, wenn es die Kernelemente umfasst und beschreibt.**



# Vom Design zum Code

**Implementieren bedeutet Code erstellen für die spezifizierten Klassen und Methoden.**

- Assoziationen und Aggregationen werden üblicherweise über Referenzen realisiert wohingegen für Kompositionen meist echte Objekte deklariert werden.
- Rollennamen werden zu Namen für Variablen.
- Quantitäten werden realisiert über Arrays, Hash-Tabellen, Sets u.ä. (im Allgemeinen sind diese implementierungsnahe Klassen nicht Bestandteil des Designs)
- Operationen werden zu Methoden des Objekts an das sie als Nachricht geschickt wurden. Operationen an Container, werden auf Methoden der entsprechenden Container-Klasse abgebildet.
- Create- und Destroy-Nachrichten werden gemäß der Programmiersprache umgesetzt.

**Design und Implementierung gehen Hand in Hand. Neue Methoden, Attribute und Klassen werden im Design ergänzt. Änderung im Design werden in der Implementierung nachgezogen.**



# Die Implementierungsreihenfolge

**Schichten werden in der Regel von unten nach oben implementiert (und vollständig getestet).**

Hierbei gilt es, die Architektur zu berücksichtigen.  
Idealerweise werden Packages und Komponenten so  
organisiert, dass die Verbindungen minimal sind, und  
eine parallele Realisierung möglich wird.

**Der Domain-Layer kommt vor dem Service-Layer, d.h. man implementiert problem- und risikoorientiert.**



# Die Implementierung im Überblick

