



Software-Engineering

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



Inhaltsverzeichnis

- Grundlagen
- Objektorientierung



Teil1

- Entwicklungsprozesse
- Objektorientierte Analyse



Teil2

- Objektorientiertes Design



Teil3

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.**
- 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.



Literatur (Grundlagen)

- **OMG Object Management Group**
OMG Unified Modeling Language (OMG UML) Version 2.5 – Needham Ma: OMG, 2015.
(<http://www.omg.org>)
- **OMG Object Management Group**
Object Constraint Language (OMG OCL) Version 2.4 – Needham Ma: OMG, 2014.
(<http://www.omg.org>)
- Rupp,C., Queins, S. und Zengler, B.
UML 2 glasklar: Praxiswissen für die UML-Modellierung, 3. Auflage – Hanser, 2007.
- Jim Arlow, Ila Neustadt
UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design, 2. ed. – Addison-Wesley Professional, 2005.
- 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 (Grundlagen)

- 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 (Grundlagen)

- 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. 2001



Software-Engineering

Grundlagen

(Ein Prozess entsteht)

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



Was ist Software-Engineering?

Der Teil der Informatik, der sich mit der systematischen Entwicklung und Wartung softwarebasierter Anwendungen beschäftigt.

Vorlesungsziel:

Die Vermittlung von Strategien und Techniken zur strukturierten und objektorientierten Entwicklung dieser Anwendungen.



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.

„ingenieurmäßig“
Basiert auf dem Einsatz von
wissenschaftlich fundierten Methoden.



Zentrale Herausforderungen

- Dekomposition und der Umgang mit Abhängigkeiten
 - intern / extern
 - implizit / explizit
- Wiederverwendung und Redundanz
 - vertikal / horizontal
- Abstraktion, Formalisierung und Beschreibung
- Einbettung in den Kontext
 - Unternehmen
 - Ressourcen / Beschränkungen
 - rechtliche Vorgaben
 - ...

von Drachenfels H.

Um welche Probleme geht es eigentlich im Software-Engineering?
Informatik Spektrum 39(3), Springer-Verlag, Heidelberg, 2016.



Übergeordnete Ziele

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

Zur Beherrschung dieser Aufgaben ist ein systematisches Vorgehen unabdingbar.



- Standardisierte Programmiertechniken
- Standardisierte Frameworks
- Standardisierte Entwicklungsprozesse
- Standardisierte ...



Eigenschaften großer Anwendungen

- Sie haben viele unterschiedliche Benutzer.
- Sie werden von vielen „Entwicklern“ gebaut.
- Einzelne „Entwickler“ verstehen lediglich noch Teile der Anwendung.
- Die Entwicklungskosten sind immens (Zeit, Geld, ...).
- Die „Lebenserwartung“ ist beträchtlich.
Im Allgemeinen deutlich höher als zu Beginn veranschlagt.

Entwickler sind
keine Benutzer!



Verständnis für die
Problematik wird
schwierig!



Die Arbeit macht
nicht immer Spaß!



Eigenschaften großer Anwendungen

**Große
Anwendungen
beinhalten
große
Gefahren!**



Probleme mit denen man rechnen muss

- Unterschiedliche Fähigkeiten im Team, dies führt zu Personalmangel in kritischen Situationen
Lösung: Schulungen, heterogene Teams, Pair-Programming, ...
- Mangelndes Problemverständnis, dies führt zur Entwicklung „falscher“ Funktionalität und ungeeigneter Benutzerschnittstellen
Lösung: Agile Entwicklung, Prototypen, Einbindung des Kunden,...
- Unterschätzte Komplexität, dies führt zu unrealistischen Aufwandsschätzungen
Anwenderprogramme < Frameworks < Systemprogramme 
Lösung: Agile Entwicklung unter Einbindung des Kunden!
- Neue Werkzeuge und Technologie, dies führt zu ineffizienter Nutzung und Produktivitätsverlust
Lösung: Schulungen, heterogene Teams, Pair-Programming, ...



Probleme mit denen man rechnen muss

Unterschätzte Risiken führen zu mangelhafter Qualität und nicht einsetzbaren Produkten!
(Dem Scheitern des Projekts)



Risiken niemals unterschätzen!
Zeit für Unerwartetes einplanen!



Sorgfältig planen und die Aufgaben auf das Wesentliche reduzieren!



Wie können Probleme reduziert werden?

**Sorgfältiges Planen und
die Aufgaben auf das Wesentliche reduzieren,
vermeidet hohe Kosten, schlechte Qualität, Verzögerungen usw.**

- **Festgelegt werden**

Ziele, Anforderungen, Beschränkungen, Lösungsstrategien, Budget, 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 einer Anwendung muss in klar definierte 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

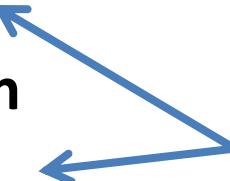
Aufgaben
des Managements:

- Kosten überwachen
- Ressourcen verteilen
- Planung
- Termine überwachen
- Risiken abschätzen

Aufgaben
der Entwickler:

- Analyse
- Design / Implementierung
- Test

**Die Experten für die Realisierung
sollte man in die Planung und
Risikoabschätzung integrieren!**





Aufgaben in der SW-Entwicklung

Diese Aufgaben müssen sich in den Übergängen zwischen den einzelnen Projektabschnitten (Meilensteine) widerspiegeln, um den Verantwortlichen Entscheidungsmöglichkeiten und Steuermechanismen zu bieten.



Übergänge planen und in Meilensteinen denken

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**

Die Entwicklung
wird transparent.



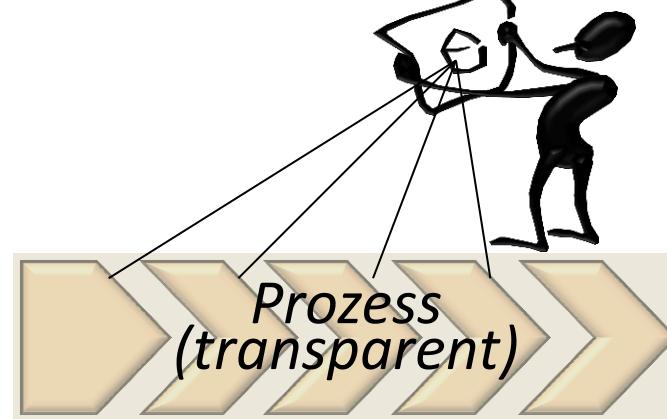
Prozessmodelle

Anforderungen
(Requirements)



Produkt
(hoffentlich)

Anforderungen
(Requirements)





Auf dem Weg zum modernen Prozess

- Codieren und Verbessern
- Wasserfallmodell
- Prototypmodell
- Spiralmodell
- Versionsmodell
- RUP
- Scrum, FDD, TDD, ...

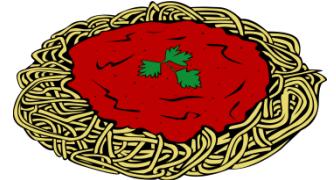


früher

Sicher?
Oftmals ist zwischen agil und
chaotisch nur schwer zu
unterscheiden!



heute





Codieren und Verbessern (Der naive Ansatz)

Das Problem löst man im Kopf, kodiert wird mit Links und als Entwickler weiß man sowieso besser, was der Kunde möchte.

Probleme löst man, wenn man ihnen begegnet!

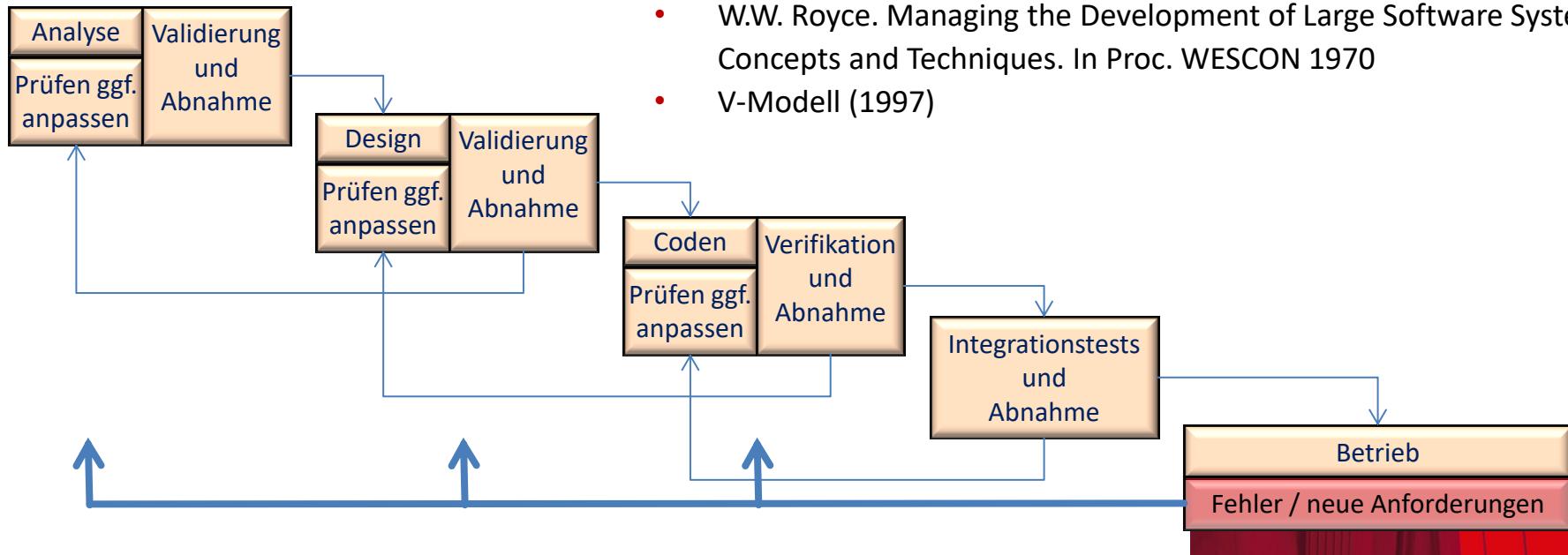
- Nach vielen Iterationen auf dem Weg zum Ziel ist der Code unbrauchbar.
Ein Design ist notwendig (nicht nur ein Neubeginn)!
- Auch genialer Code nützt nichts, wenn er Probleme löst, die niemand hat.
Requirement-Analyse ist gefragt!
- Korrekturen und Änderungen sind zeitaufwendig und damit kostspielig und bringen das Projekt nicht wirklich voran!

Andauerndes Coden weißt noch nicht auf einen echten Fortschritt hin!



Das Wasserfallmodell

Analyse, Design und Implementierung sind tolle Meilensteine mit klar erkennbaren Übergängen.





Probleme des Wasserfallmodells

- Die Aufgabe wird als Ganzes betrachtet
- Die Phasen sind starr.
- Vor jedem Übergang ist die Phase vollständig abzuschließen.
- Eine Rückkopplung ist nur im Fehlerfall möglich.
- Genaue Spezifikationen für schlecht verstandene Anforderungen gefolgt von der Implementierung großer Mengen nutzlosen Codes.

**Tolle Spezifikationen für schlecht verstandene Anforderungen,
gefolgt von der Implementierung großer Mengen nutzlosen Codes.**



Prototypen

Eine anschauliche Requirement-Analyse verringert das Risiko für Missverständnisse aus denen sich spätere Fehlentwicklungen ableiten.

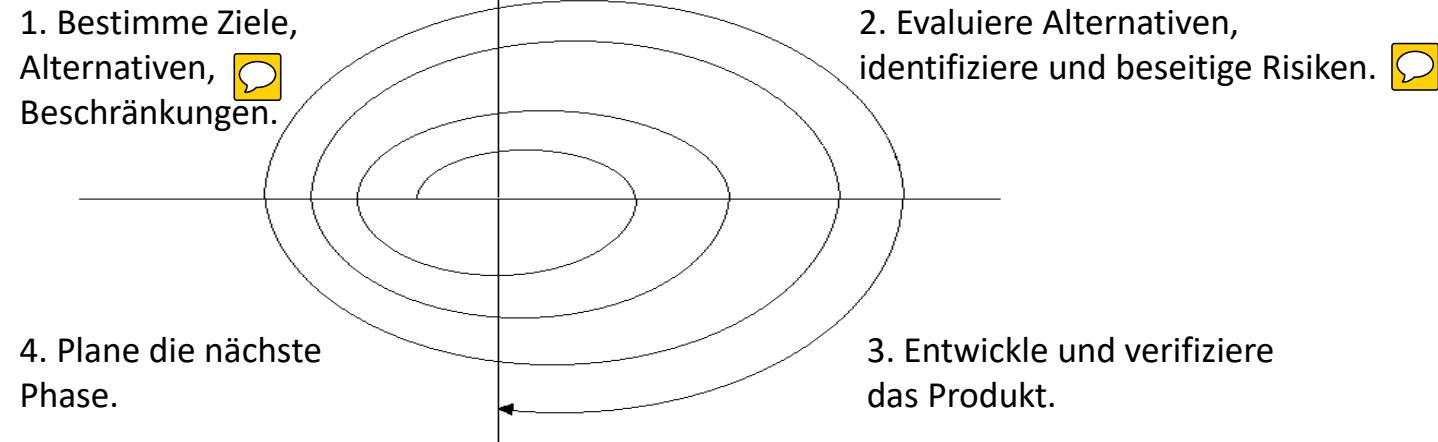




Spiralmodell

Boehm (1988) ein neuer Ansatz:
Risikomanagement
bestimmt den
Entwicklungsverlauf!

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



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



Spiralmodell

Der größte Unterschied zu den bisherigen Ansätzen liegt in der expliziten Betrachtung des **Risikos** als entscheidende und treibende Kraft bei der Entwicklung der Software.

Risiken sind dabei Entscheidungen, die auf unvollständigen und falschen Annahmen und Informationen beruhen.



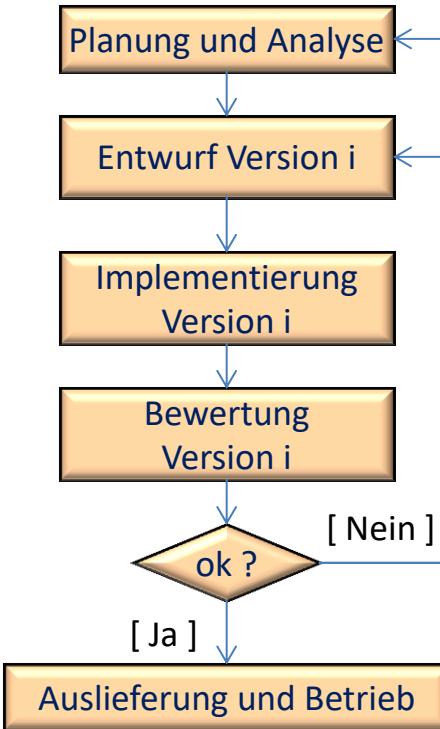
Informationen sammeln, aufbereiten, bewerten und somit die Unsicherheiten beseitigen!

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.



Offene Fragen:

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



Vorteile des Versionsmodelle

- **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)



Übergang in die Moderne

Offensichtlich zerfällt in Versionsmodellen das bisherige Gefüge und die damit verbundene zeitliche Abfolge von Analyse, Design, Implementierung, und Inbetriebnahme. Damit wird Platz geschaffen für neue problemspezifische Phaseneinteilungen (Sprints, Arbeitspakete, Features).

Analyse und Design findet weiterhin statt, jedoch deutlich problem- und zielorientierter.

Das Teilprojekt steht im Vordergrund



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 Voraussetzungen.

→ 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)

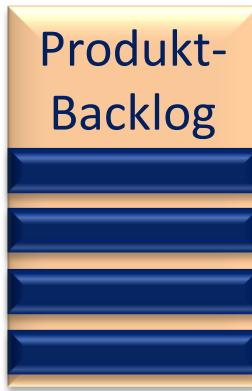


Agile Software-Entwicklung (SCRUM)

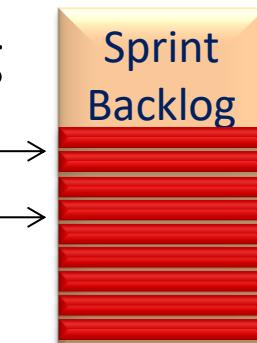
Schwaber, K. and Sutherland, J.

The Scrum Guide: The Definitive Guide to Scrum – Scrum.org, 2011

Requirements



Grooming



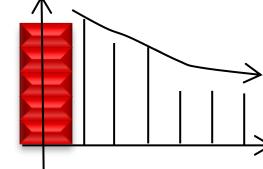
Arbeitspakete
(Tasks)

(1 Tag)



Sprint (1 Monat)

Burndown Chart





Software-Engineering

Grundlagen

(Objektorientierung eine durchgängige Sichtweise)

Prof. Dr. Thomas Fuchß

Hochschule Karlsruhe – Technik und Wirtschaft

Fakultät für Informatik und Wirtschaftsinformatik



Objektorientierte Software-Entwicklung

Moderne Software entwickelt man:

- **agil,**
- **iterativ und**
- **architekturorientiert,**

Die Architektur ist ein Bauplan für die Organisation des Systems, eine Beschreibung der Komponenten und Subsysteme und ihrer Beziehungen und Berücksichtigung der zentralen Anforderungen.

Auch die Erstellung der Architektur erfolgt agil:

- 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.



Objektorientierte Software-Entwicklung

Objektorientierung ist ein durchgängiges Stilmittel. Es bietet ähnlich Konzepte auf den unterschiedlichsten Ebenen.

- Klassen, Methoden und Attribute in der Programmierung
- Klassen, Operationen und Assoziationen in der Modellierung
- Komponenten und Konnektoren in der Architektur.

Ein langer Weg, bis zu diesem Punkt.



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

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

Objekte teilen sich „gemeinsame“ Eigenschaften.

Die Zugriffssteuerung erfolgt über super und self.

- **Abstrakte Datentypen (ADTs) (Eiffel** Bertrand Meyer 1985)

Objekte kapseln ihren internen Zustand.

Der Zugriff erfolgt über eine definierte Schnittstelle.

```
feature {NONE}
aPrivateCounter: 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.

Vererbung 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:** 
 - Die Ergebnisse der Analyse (**Problemraum**) lassen sich in die Implementierung (**Lösungsraum**) übertragen.

**Achtung: Eine gute Simulation
ist noch keine gute Lösung!**



Methodische Durchgängigkeit

Am Anfang steht die Erfassung der Anforderungen (**Problemraum**).

Ein Dialog:

Entwickler: Was ist Ihnen wichtig?

Anwender: Unsere Mitglieder?

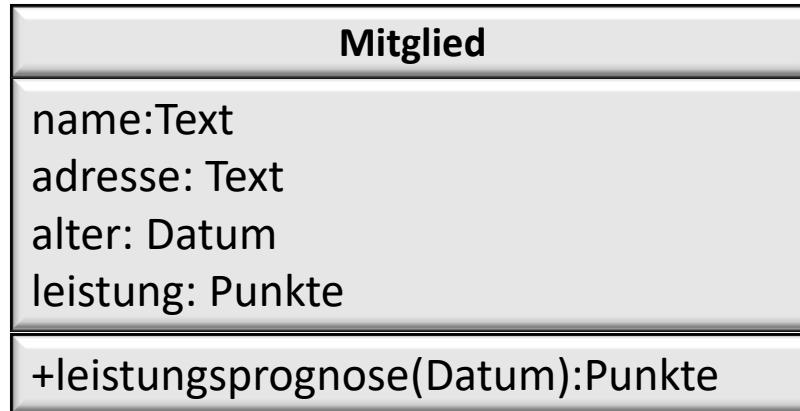
Entwickler: Welche Merkmale zeichnen Ihre Mitglieder aus?

Anwender: Unsere Mitglieder sind nicht anonym. Wir kennen uns beim Namen, wir wissen Alter und Adresse. Die Leistungsfähigkeit eines jeden wird akribisch überwacht. Die muss stimmen. D.h., sie muss jederzeit über das System abrufbar sein!



Methodische Durchgängigkeit

Im Analyse- und Designmodell (UML)
erscheint das Mitglied als Klasse.





Methodische Durchgängigkeit

In der Implementierung gibt es immer noch
die Klasse Mitglied. (**Lösungsraum**)

```
class Mitglied
{
    private String      name;
    private Anschrift  adresse;
    private Date        alter;
    ...
    public int leistungsprognose(Date datum)
    { ...
    }
    ...
}
```



Essentials

- **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.**



Software-Engineering

Grundlagen

(Objektorientierung, Begriffe und ihre Bedeutung)

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

Methodische Durchgängigkeit erfordert eine gemeinsames
Verständnis!



Was sind Objekte?

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

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

Nicht alle diese Dinge existieren physisch. Aber alle haben Eigenschaften, oder setzen sich aus anderen 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:

? Realität ?



Archery (PSF): Aus dem Archiv von Pearson Scott Foresman, gestiftet der Wikimedia Foundation, 2007 (public domain)
[http://commons.wikimedia.org/wiki/File:Archery_\(PSF\).png](http://commons.wikimedia.org/wiki/File:Archery_(PSF).png)



[DM BoV2015 www.dbsv1959.de]

- 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 Mitglied ist, das über einen Namen, eine Adresse und eine gewisse Fähigkeit verfügt.

Mitglied

name:Text

adresse: Anschrift

alter: Datum

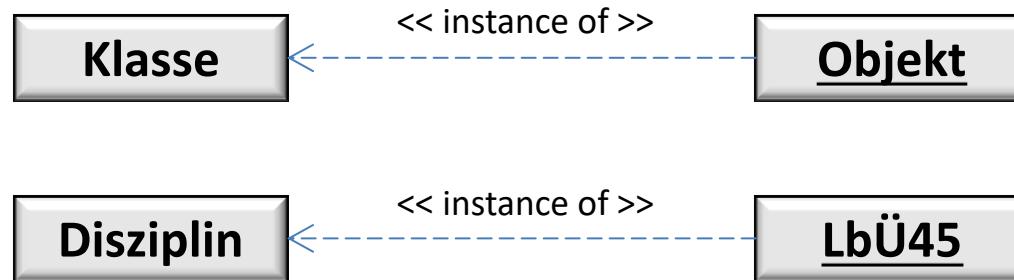
leistung: Punkte

+leistungsprognose ():Punkte



Klassen und Klassifikation

Klassen beschreiben die Struktur und das Verhalten vieler gleichartiger Objekte. Objekte bezeichnet man auch als Instanzen einer Klasse. Die systematische Zusammenfassung vieler gleichartiger Objekte zu einer Klasse bezeichnet man als Klassifikation (oder Taxonomie).





Attribute, Operationen, ...

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

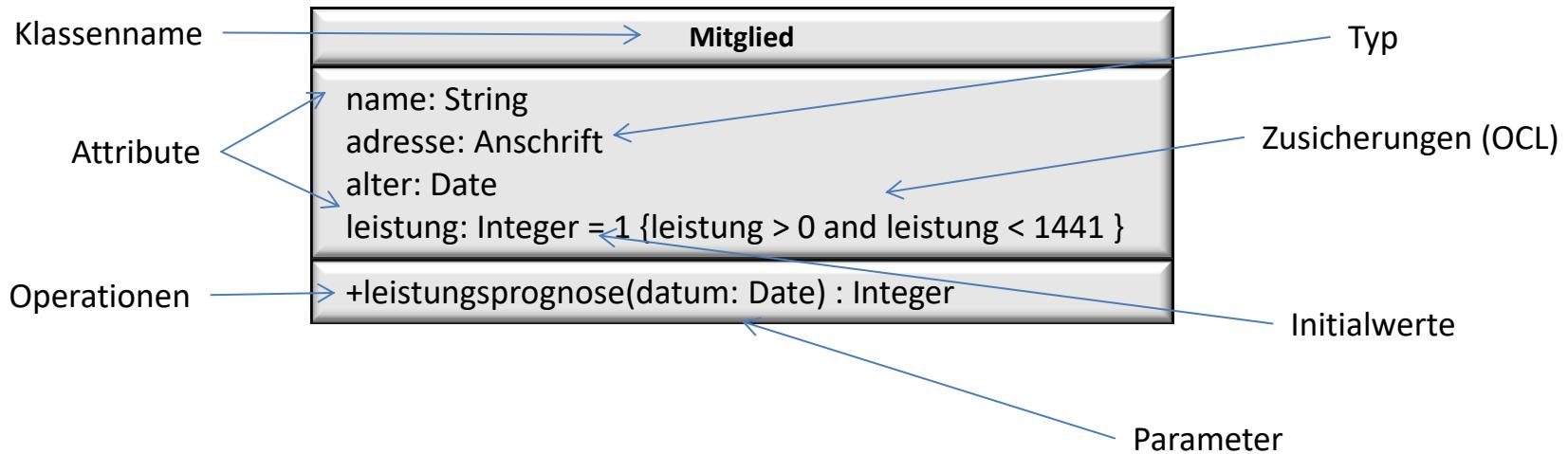
- Attribute
 - beschreiben die Struktur von Objekten.
 - sind Bestandteile, Informationen, Daten.
- Operationen
 - beschreiben das Verhalten von Objekten.
- Zusicherungen
 - sind Regeln und Zwänge (Constraints), die die Objekte erfüllen müssen.
- Beziehungen
 - sind Relationen zwischen Klassen, hat-ein, ist-ein, ist-Teil-von, ...

**Eigenschaften sind
mehr als Attribute!**



Beispiel

Klassen sind Einheiten aus Attributen, Operationen und Zusicherungen.
Objektorientierte Ansätze (Programmiersprachen) zeichnen sich somit dadurch aus, dass sie Daten und Funktionen als eine Einheit begreifen.





Objektidentität

Die Existenz eines Objektes ist unabhängig von den konkreten Attributwerten

- Zwei Objekte mit gleichen Attributwerten sind zwei Objekte!
- Objekte können nicht über die Werte ihrer Attribute identifiziert werden.

Das gleiche Objekt ist nicht dasselbe Objekt!



Ein Vorteil und Fluch objektorientierter Sprachen.



Beispiel

```
String vergleiche (String x, String y)
{
    return (x==y) ? "derselbe Text" : "nicht derselbe Text";
    return (x.equals(y)) ? "der gleiche Text" : "nicht der gleiche Text".
}
```

- Was ist richtig?
- Wann braucht man Identität und wann nur Gleichheit?



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!

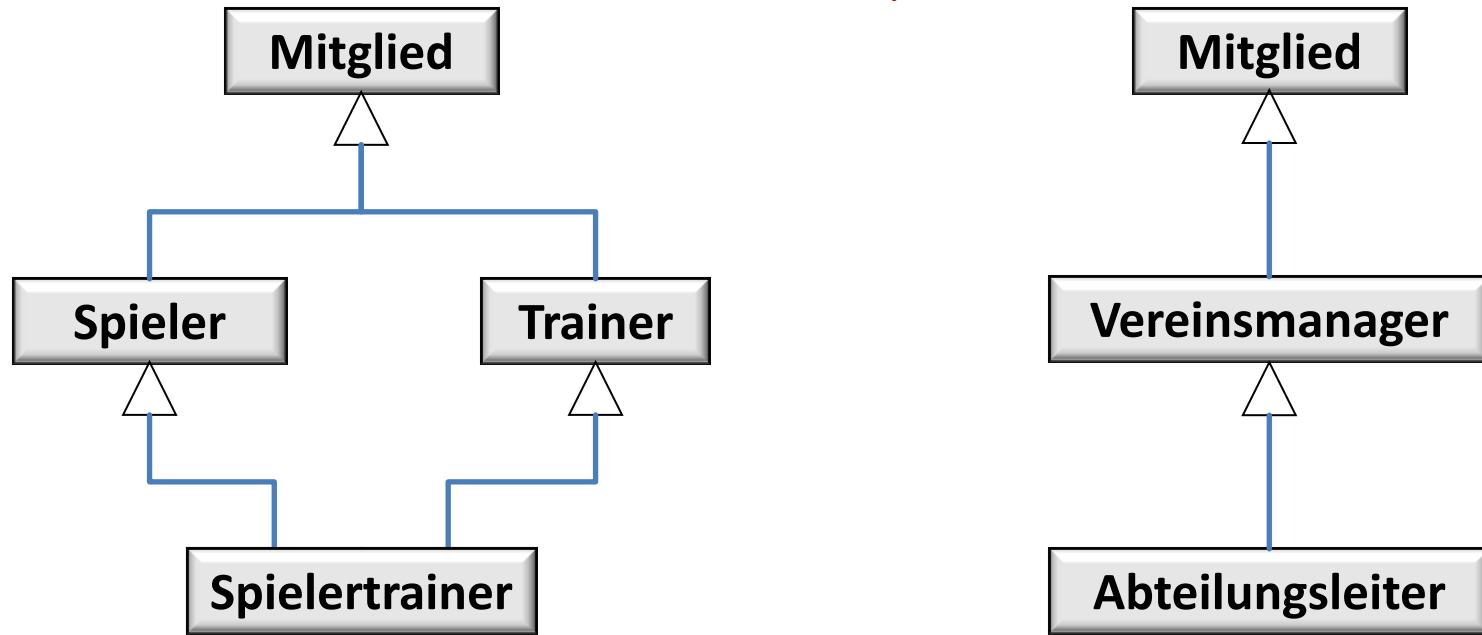
- **Das Substitutionsprinzip**

Instanzen (Objekte) von Unterklassen sind immer auch Instanzen ihrer Oberklasse und können als solche betrachtet und verwendet werden.



Vererbung

Oberklasse, Superclass

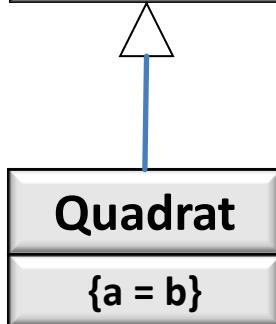
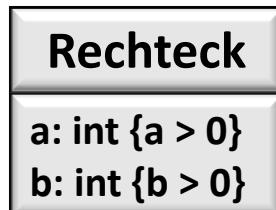


Unterklasse, Subclass

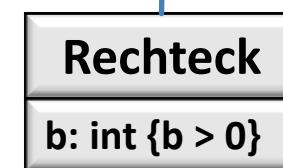
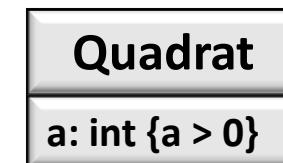


Vererbung: Beispiel

Was ist besser?



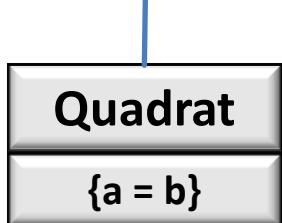
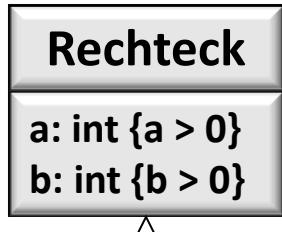
?





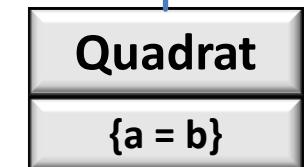
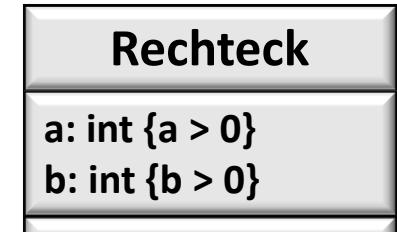
Vererbung: Beispiel

sicher?



Quadrat q = new Quadrat(3);
q.**stretch**(1,2);

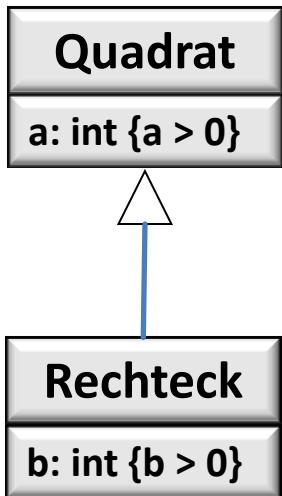
?





Vererbung: Beispiel

sicher?



Quadrat q;

Rechteck r = new Rechteck(3,4);

q = r;

?



Vererbung: Beispiel

oder besser ohne Vererbung?

Rechteck

a: int {a > 0}

b: int {b > 0}

+istQuadrat():boolean

```
public boolean istQuadrat()
{
    return (a==b);
}
```



UML Klassen- und Instanzdiagramme

OMG Object Management Group

OMG Unified Modeling Language (OMG UML) Version 2.5 – Needham Ma: OMG, 2015.
(<http://www.omg.org>)

- 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.



Die Semantik der UML

Eine Beschreibungssprache macht nur Sinn, wenn die zur Verfügung stehenden Elemente eine (wohl-) definierte Bedeutung haben.

- Die UML verfügt über ein Meta-Modell in dem alle Beschreibungselemente verankert sind.
- Im Sinne einer methodische Durchgängigkeit ist dieses Meta-Modell in UML-Notation beschrieben.



**Neue Beschreibungsmöglichkeiten erfordern lediglich
eine Erweiterung des bestehenden Meta-Modells!**



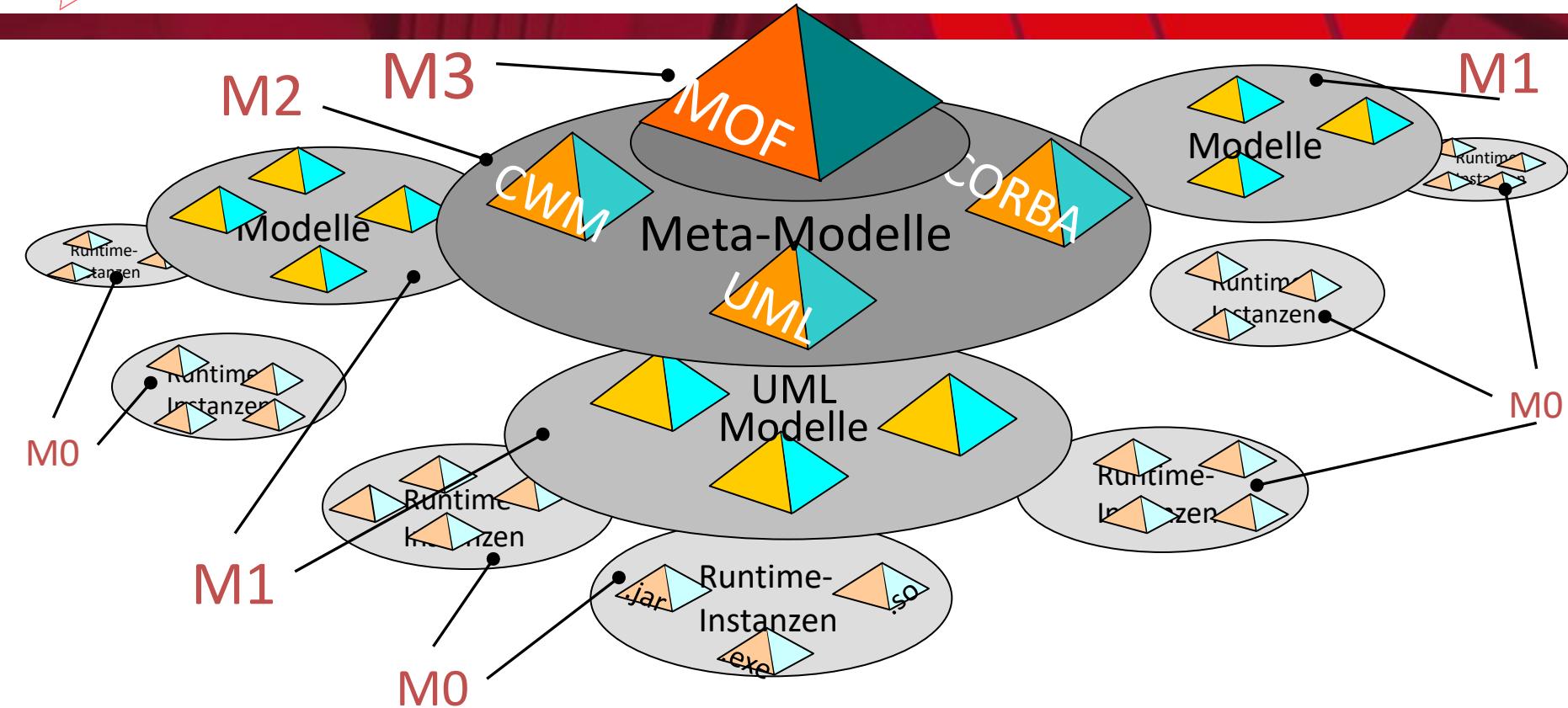
Das Meta-Modell (MOF)

OMG Object Management Group
OMG Meta Object Facility (MOF)
Core Specification Version 2.5.1 – Needham Ma: OMG, 2016.
(<http://www.omg.org>)

| Schicht | Beschreibung | Beispiel |
|---|---|--|
| M3 Meta-Meta-Modell | Die Infrastruktur: Ein Meta-Meta-Modell definiert eine Sprache, um Meta-Modelle zu erstellen. | MetaKlasse, MetaAttribut, MetaOperation, ... |
| M2 Meta-Modell (abstrakte Syntax) | Eine Instanz eines Meta-Meta-Modells: Ein Meta-Modell definiert eine Sprache (etwa UML), um Modelle zu erstellen. | Klasse, Attribut, Operation, Assoziation, ... |
| M1 Modell | Eine Instanz eines Meta-Modells: Ein Diagramm mit Klassen, Attributen, ... | Mitglied, Team, Vorstand, Name, Anschrift, ... |
| M0 reale Objekte (Ziel) | Etwas „greifbares“ wie Programme (Quellcode), Daten in Datenbanken, ... | „Mustermann“, 1234, ... |

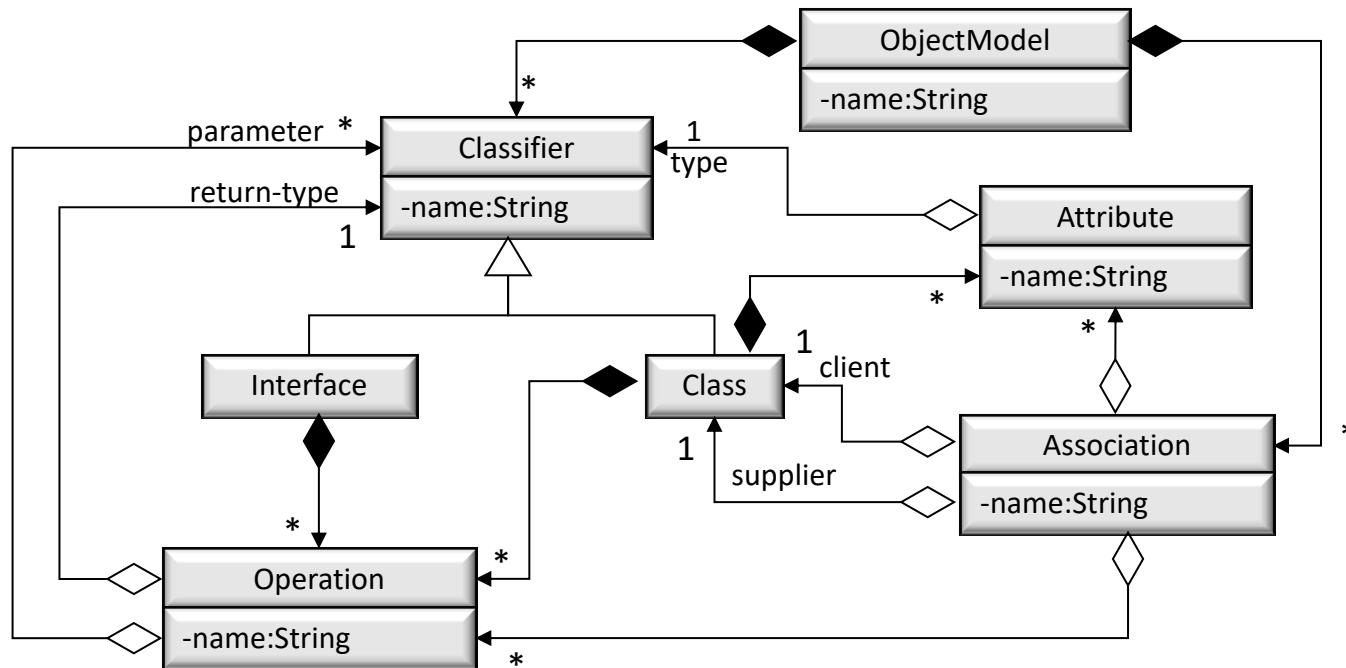


Die (semantischen) Hierarchien der UML





Beispiel: Ein einfaches Meta-Modell



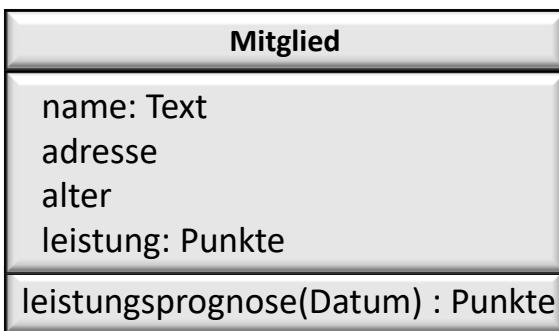


Darstellung von Klassen

Name Compartment (Namensfeld),
beinhaltet den Klassennamen

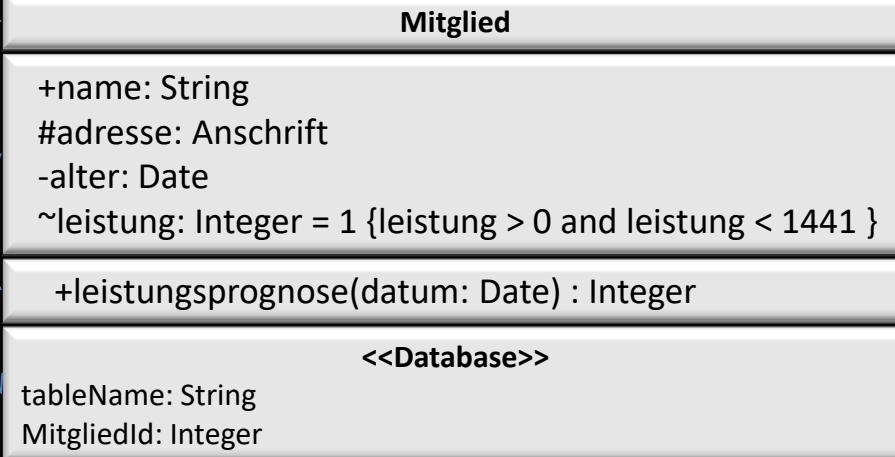
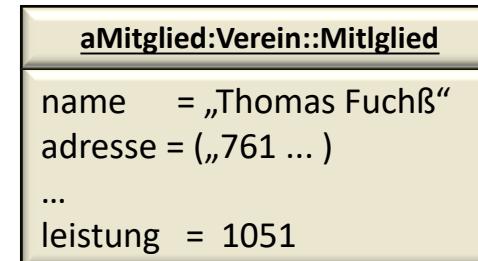
List Compartments (Listenfeld)
für Attribute, Operationen
und anderweitige Gruppierungen

Analyseniveau



Mitglied

ohne Details



Designniveau

ein Objekt



Das Namensfeld



- **Stereotype:** definiert ein besonderes Modellelement wie Interface, Type, Controller,... Durch das Einführen eigener Stereotypen, **Unterklassen existierender Klassen des UML Metamodells**, kann die Ausdrucksmöglichkeit der UML erweitert und angepasst werden.
 - **Property:** charakterisiert das Beschreibungselement.
 - vordefiniert isAbstract = true, isLeaf = true (**Attribut im Meta-Modell**)
 - selbstdefiniert author = Th. Fuchs (immer als Key-value Pair)



Die Attribute

Attribute beschreiben die Struktur!

```
+name: String [2]  
#adresse: Anschrift  
-alter: Date  
~/leistung: Integer = 1 {leistung > 0 and leistung < 1441 }
```

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)

```
+leistungsprognose(datum: Date = toDay): Integer {concurrency = sequential }  
    <<constructor>>  
+ Mitglied(in name: String [2])
```

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

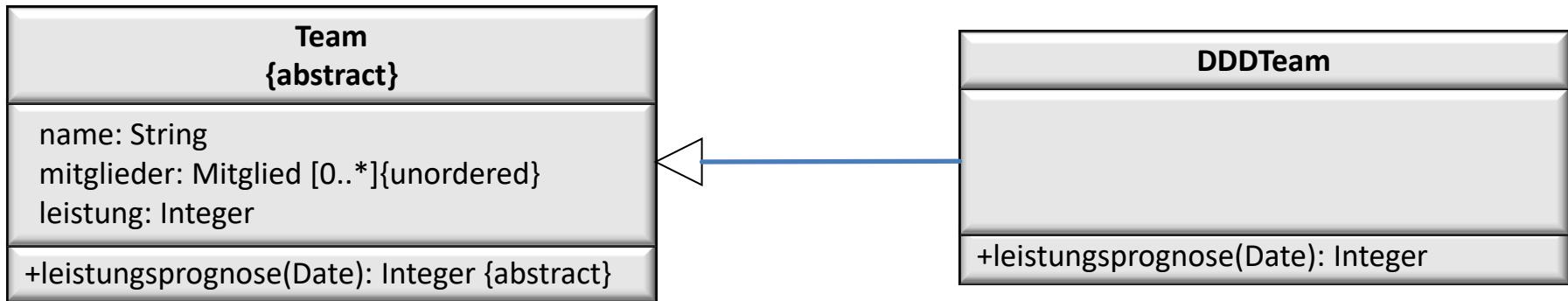
- **parameter-list** kommaseparierte Liste formaler Parameter
kind name : type [multiplicity] = default-value {property}
 - **kind** in, out, inout
 - **default-value** Default-Wert, falls der Parameter nicht übergeben wird
- **besondere Properties:**
 - **concurrency = ..., sequential, concurrent, guarded**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



Abstrakte Klassen

Eine abstrakte Klasse ist eine Klasse, die keine "direkten" Instanzen 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.

Realisierung: nur die Operationen werden vererbt!

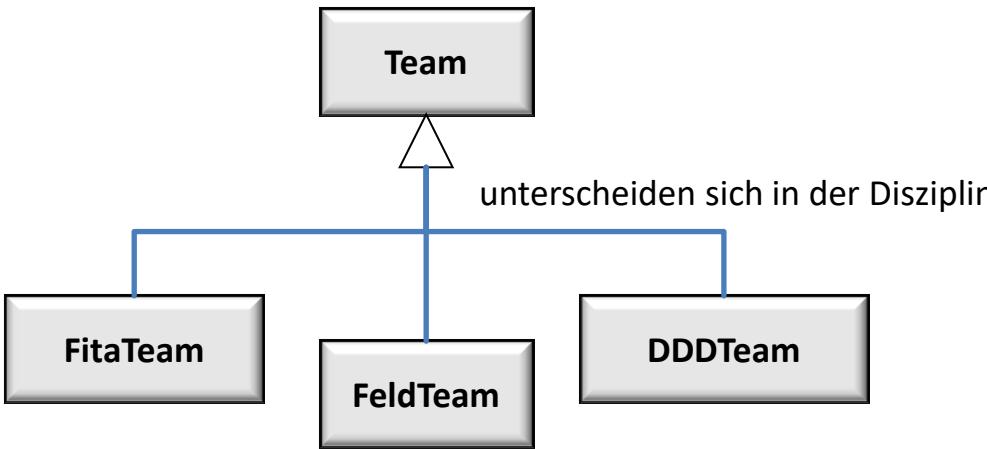


Interfaces sollten weder Attribute noch Assoziationen zu anderen Klassen haben.



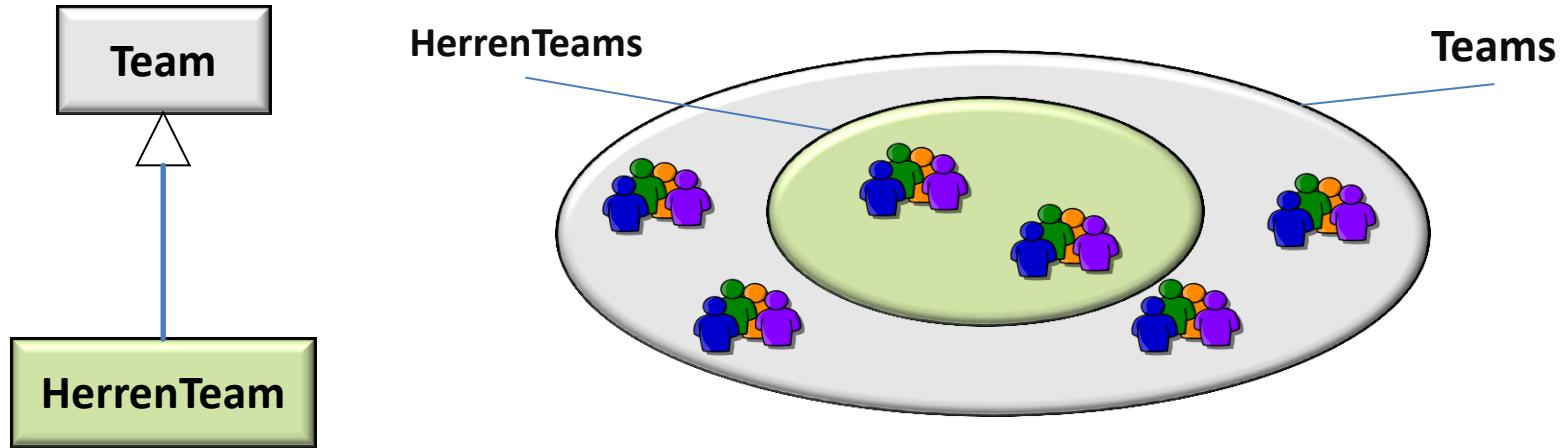
Angaben zum Generalisierungsgrund

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





Semantik der Vererbung (Modellierungsebene)



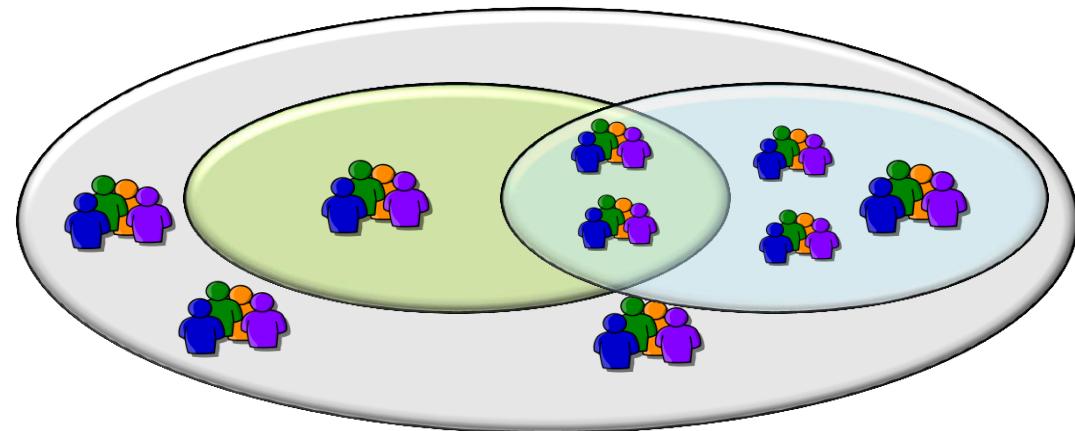
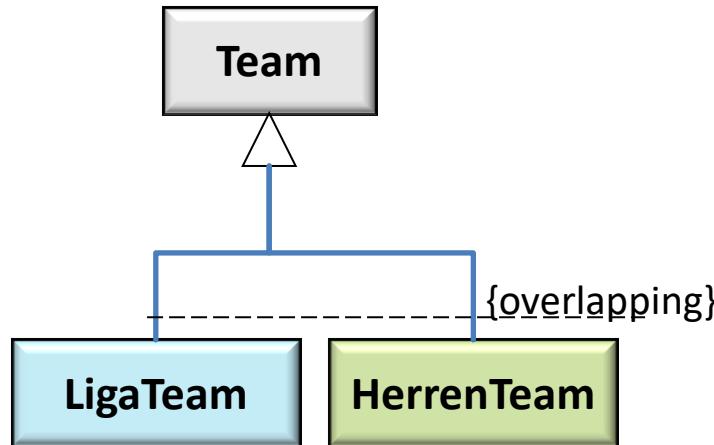
Spezialisierungsvererbung (Ist-ein-Vererbung)

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



Besondere Beziehungen

- overlapping

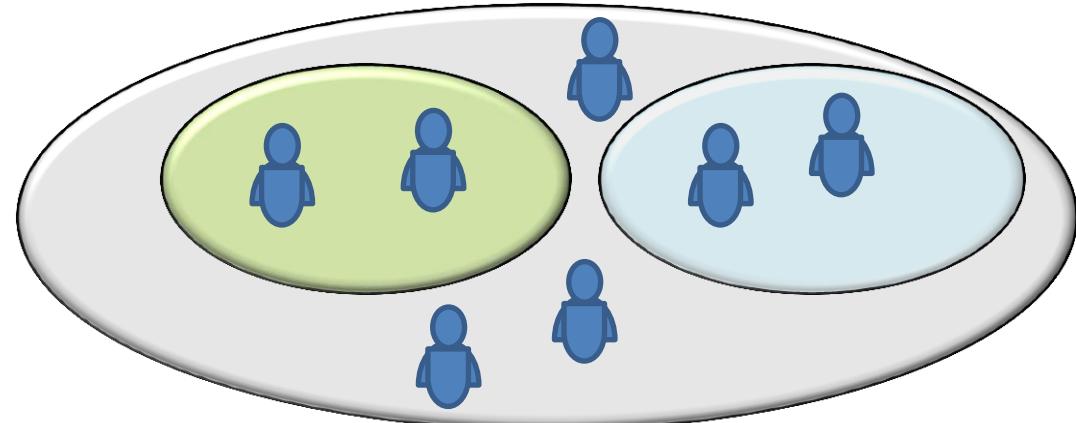
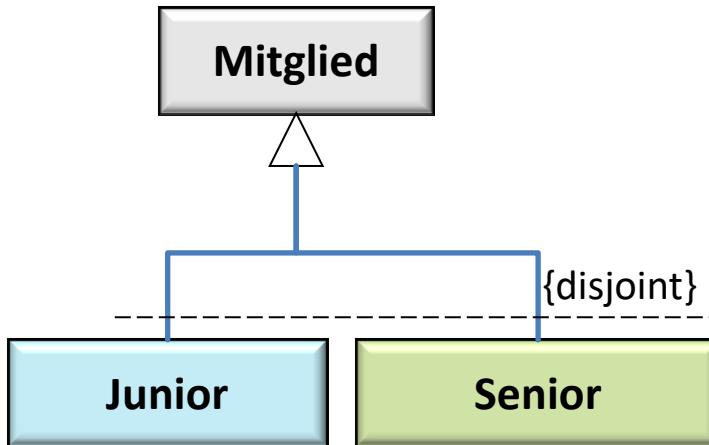


Es gibt Instanzen, die sich in der Schnittmenge befinden. Eine Mehrfach-Vererbung könnte Sinn ergeben! (HerrenLigaTeam)



Besondere Beziehungen

- disjoint

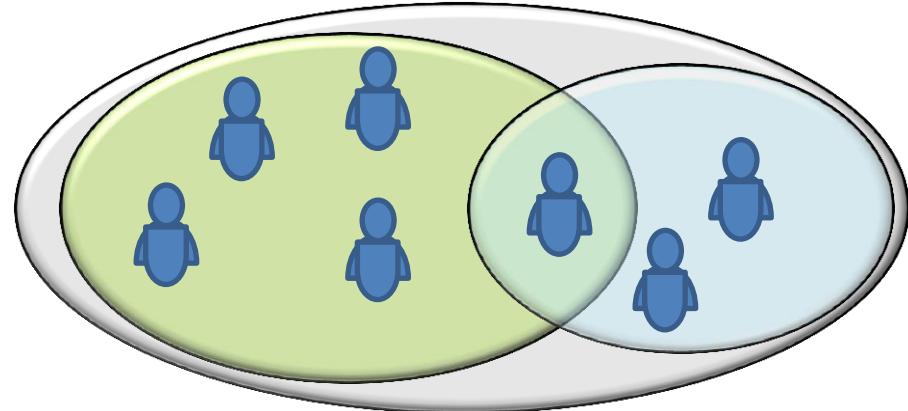
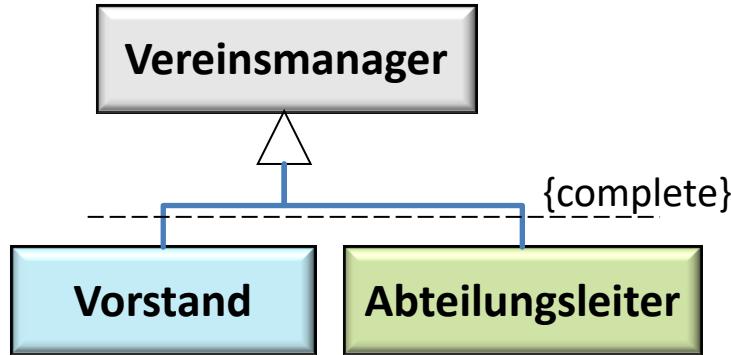


Es gibt keine Instanzen, die sich in der Schnittmenge befinden. Eine Mehrfach-Vererbung wäre sinnlos
(JuniorSenior)



Besondere Beziehungen

- complete

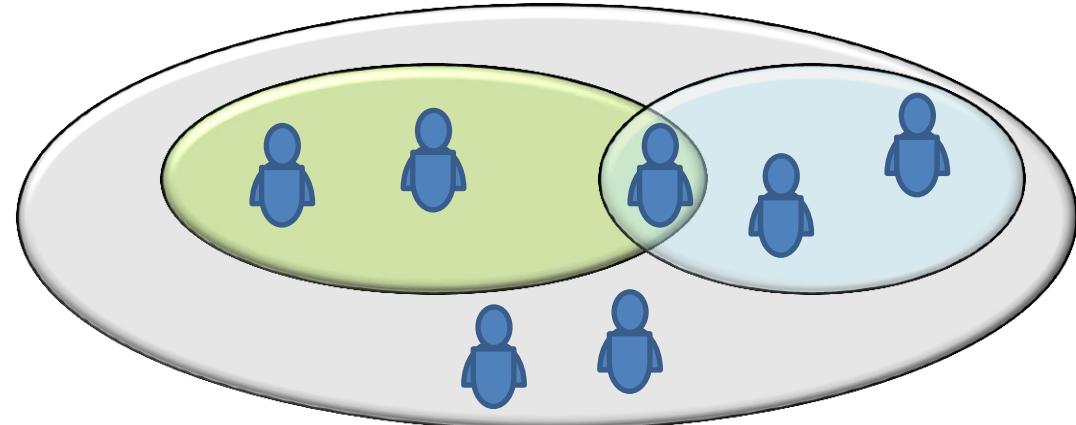
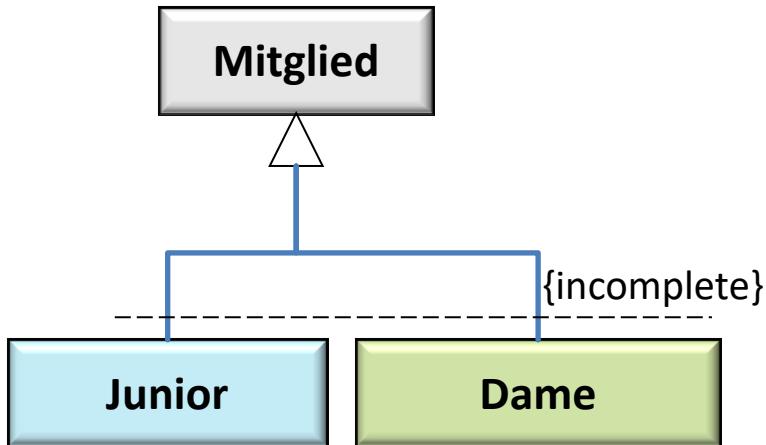


Es gibt keine Instanzen außerhalb. Zusätzliche Spezialisierungen der Vereinsmanager, die sich von den angegebenen Klassen abgrenzen sind sinnlos.
(GruppenLeiterohneVorstandsaufgabe)



Besondere Beziehungen

- incomplete



Es gibt noch Instanzen außerhalb. Eine zusätzliche Spezialisierungen der Mitglieder ist sinnvoll und angebracht. (Herr, Senior, ...)



Templates und Bindungen

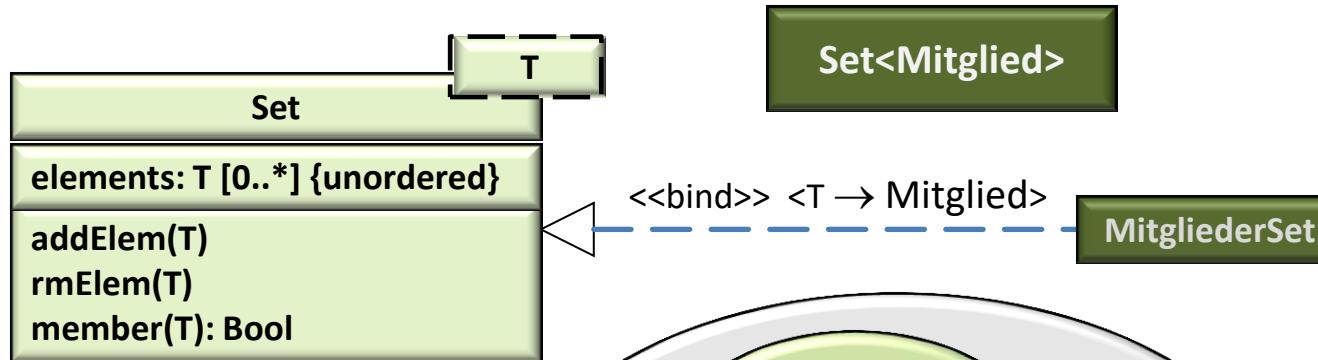
Grenzen der "Ist-ein-Vererbung"

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 (**Binding**) bestimmt werden.

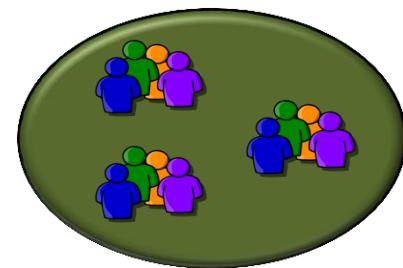
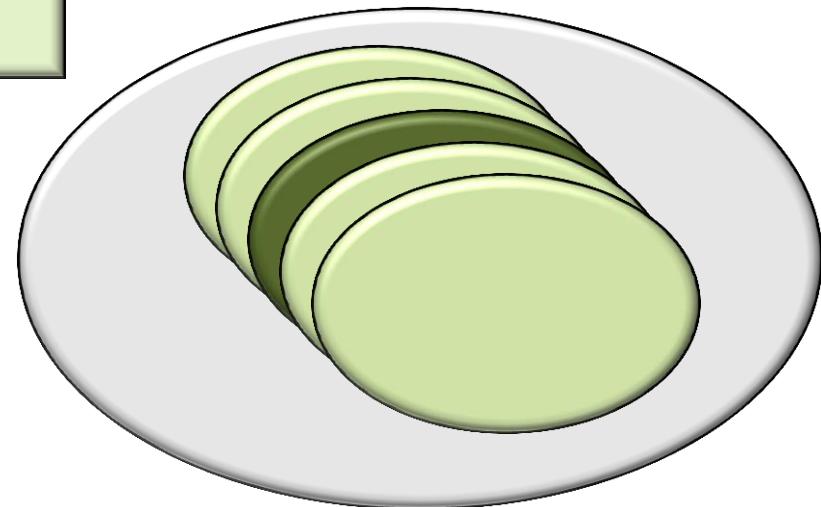
- Beziehungen (Assoziationen) sind immer gerichtet – vom Template zur Klasse, nicht umgekehrt.
- Templates können keine Oberklassen wohl aber Unterklassen sein. D.h., jede Klasse, die durch Bindung der Parameter eines Templates entsteht, das eine Subklasse ist, ist auch eine Subklasse der entsprechenden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



Templates



Eine Familie von
Klassen





Eigene 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!

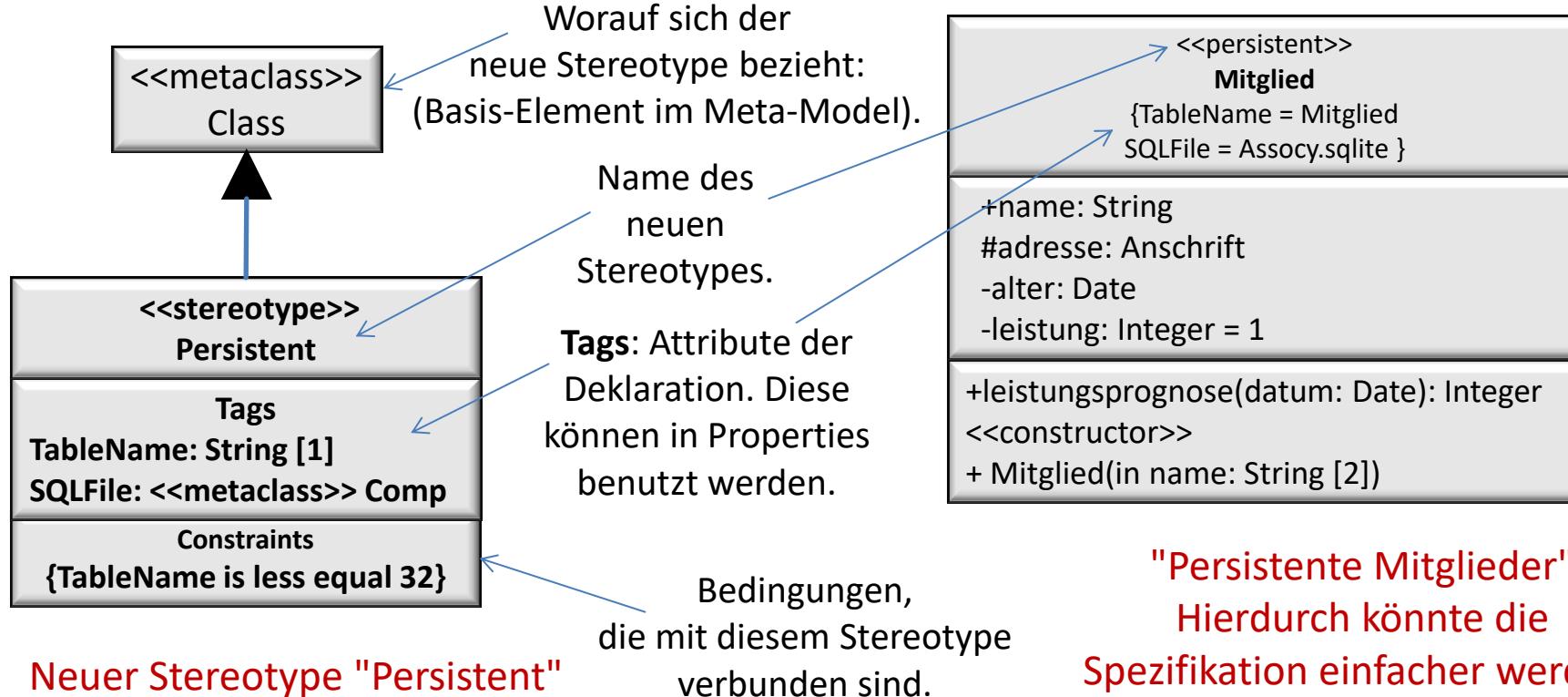


Die Beschreibungssprache kann besonderen Gegebenheiten (Domänen) angepasst werden.

Achtung: Spezialwissen entsteht!



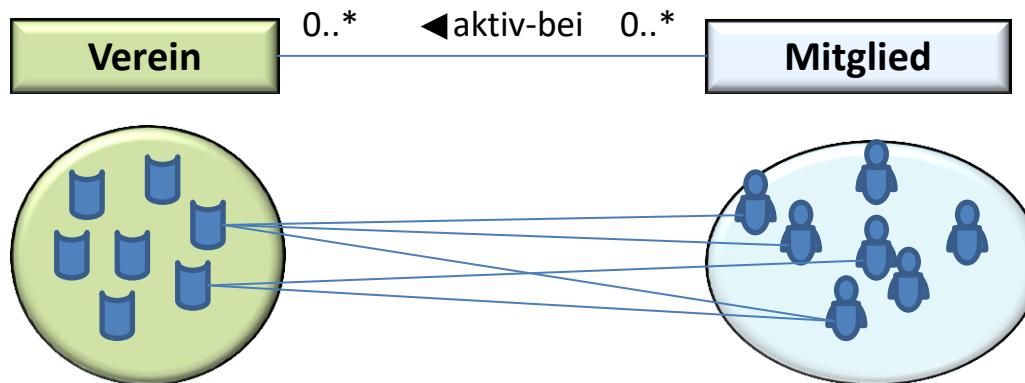
Eigene Deklarationen





Beziehungen zwischen Klassen

Die allgemeinste Form der Beziehung zwischen Klassen ist die Assoziation. Sie kann als Relation zwischen den Instanzen der beteiligten Klassen interpretiert werden.



Achtung: Auch
Relationen sind
Mengen!



Assoziationen

- 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 gerichtet sein.
- **Eine Assoziation hat einen Namen.**
- **Eine Assoziation kann mit Attributen versehen werden.**

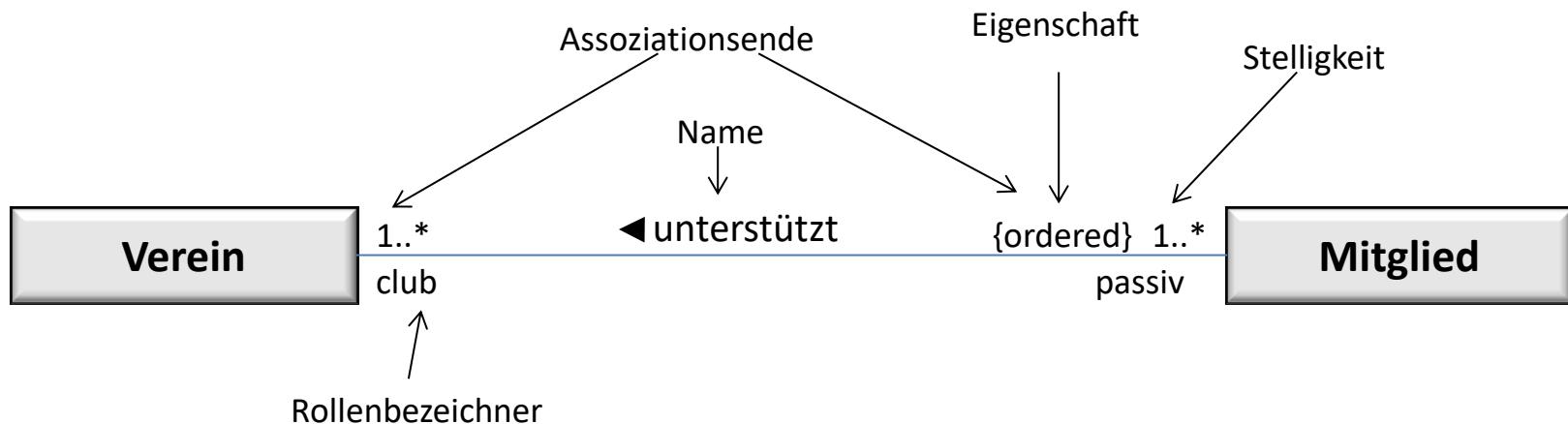
**Assoziationen besitzen
Klasseneigenschaften.**



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 das Mitglied kann auf die ihm zugeordneten Disziplinen zugreifen. Die Disziplinen selbst wissen nicht, welche Mitglieder sie ausüben.

Eine solche Assoziation bezeichnet man als navigierbar.

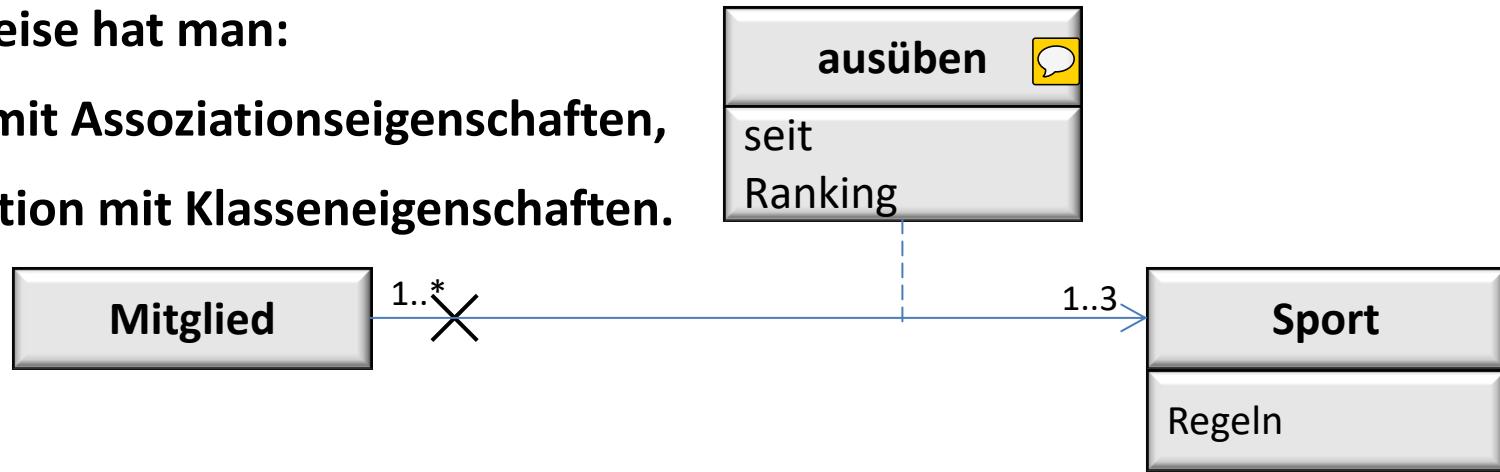


Assoziationen besitzen Klasseneigenschaften

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:

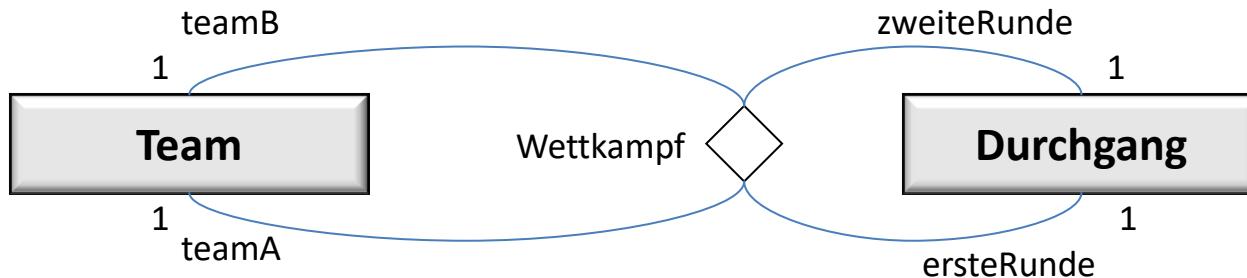
- eine Klasse mit Assoziationseigenschaften,
- eine Assoziation mit Klasseneigenschaften.





Mehrstellige Assoziationen

Beziehungen können auch mehrstellig sein



{self.teamA self.teamB and self.zweiteRunde <> self.ersteRunde}

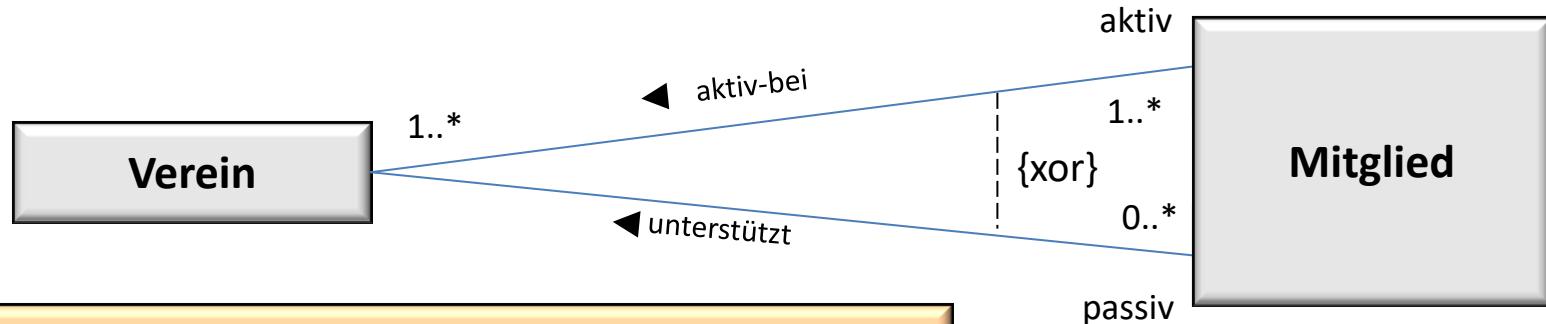
Beziehungen können eingeschränkt werden (Constraints).



Beziehungen zwischen Assoziationen

Assoziationen können zueinander in Beziehung gesetzt werden.

- über ein XOR-Constraint

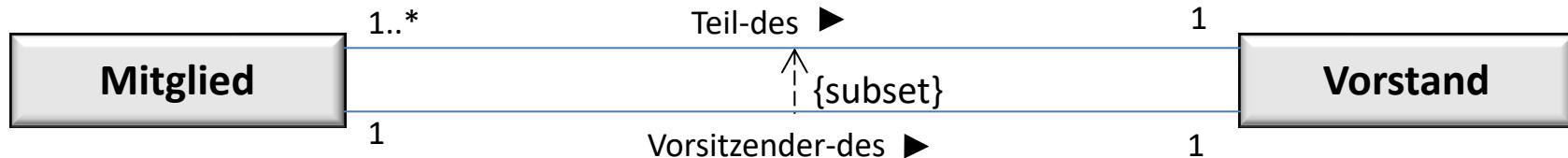


Eine Instanz der Klasse **Mitglied** ist entweder ein aktives oder ein passives Mitglied, aber niemals beides.

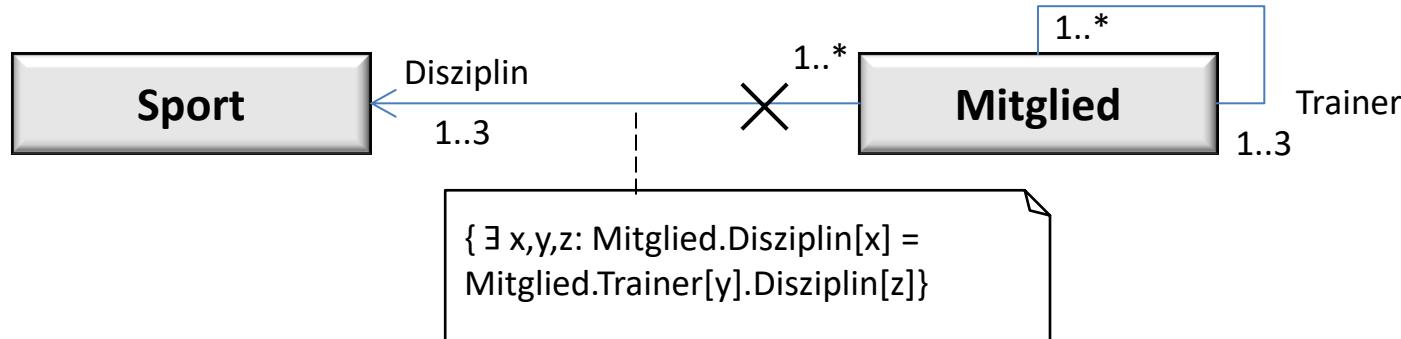


Beziehungen zwischen Assoziationen

- über gerichtete Constraints



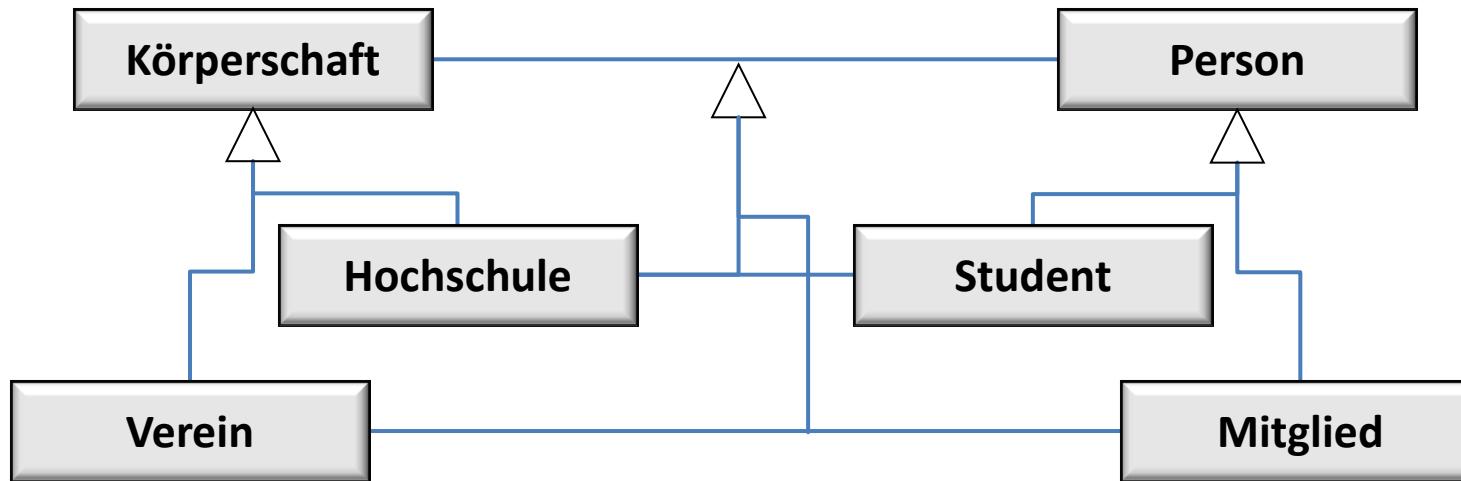
- über annotierte Constraints





Beziehungen zwischen Assoziationen

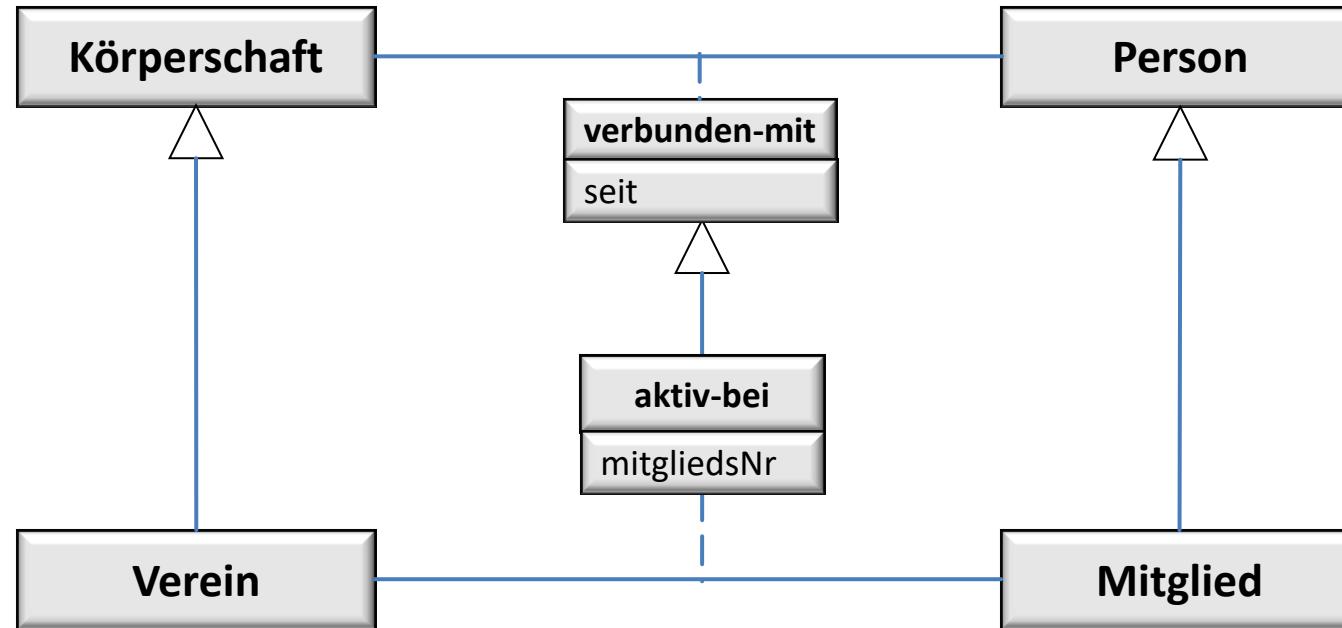
- durch Vererbung





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.

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

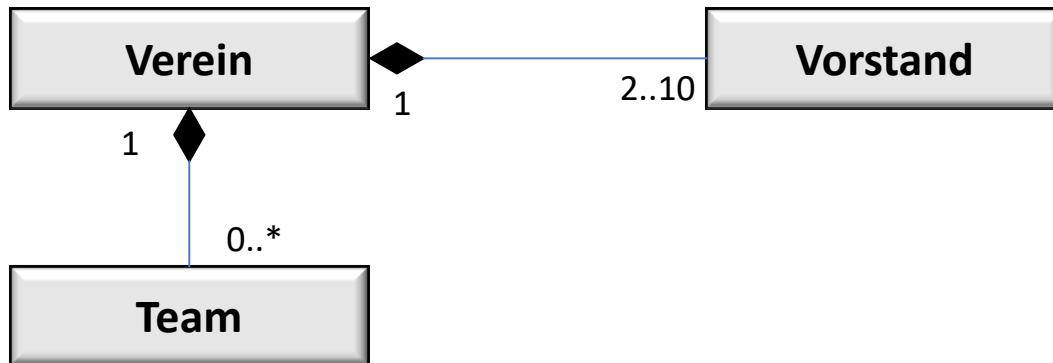




Spezielle Assoziationen

- 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.

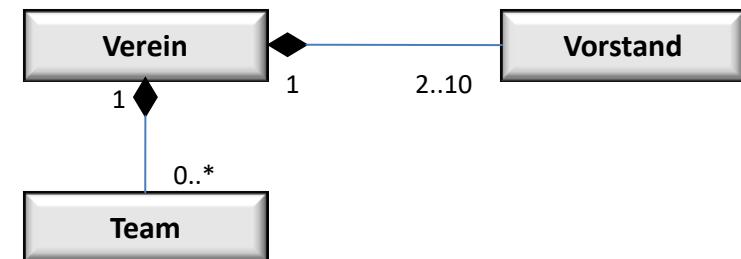




Komposition

- **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.**
 - Die Teile werden erst später erzeugt.
 - Die 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.





Aggregation

- 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.



Der Zugriff auf die Mitglieder, die in einem Team starten, erfolgt stets über das Team-Objekt.



Assoziation und Attribut

Attribute einer Klasse können immer auch als nicht navigierbare Aggregationen (Kompositionen) verstanden werden.

Mitglied

-name: String
#adresse: Anschrift
-alter: Date
-leistung: Integer

+leistungsprognose (datum: Date): Integer

Mitglied

-name: String
-alter: Date
-leistung: Integer

+leistungsprog...

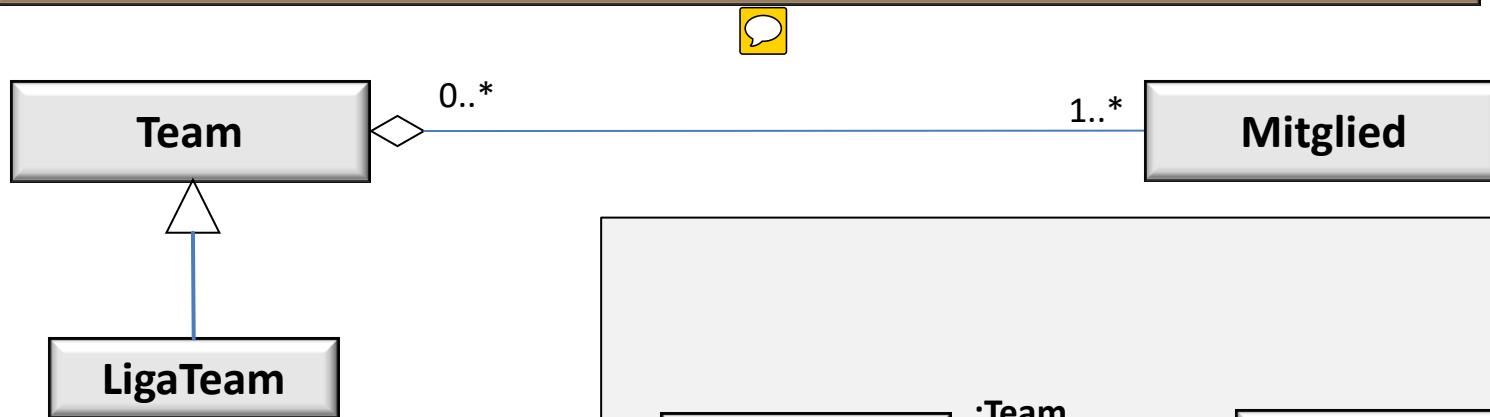
#adresse
1

Anschrift

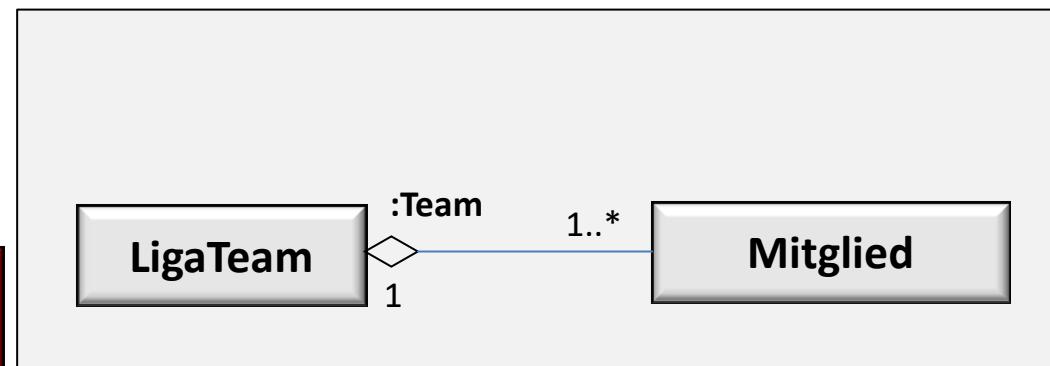


Auch Assoziationen werden vererbt

Übersicht entsteht durch Fokussierung und das Weglassen von unwichtigen Details. Dies führt zu Problemen bei der Darstellung von abgeleiteten Details. **Interface-Specifier** lösen dieses Problem.



Keine neue Aggregation, sondern die vererbte Aggregation der Team-Klasse!

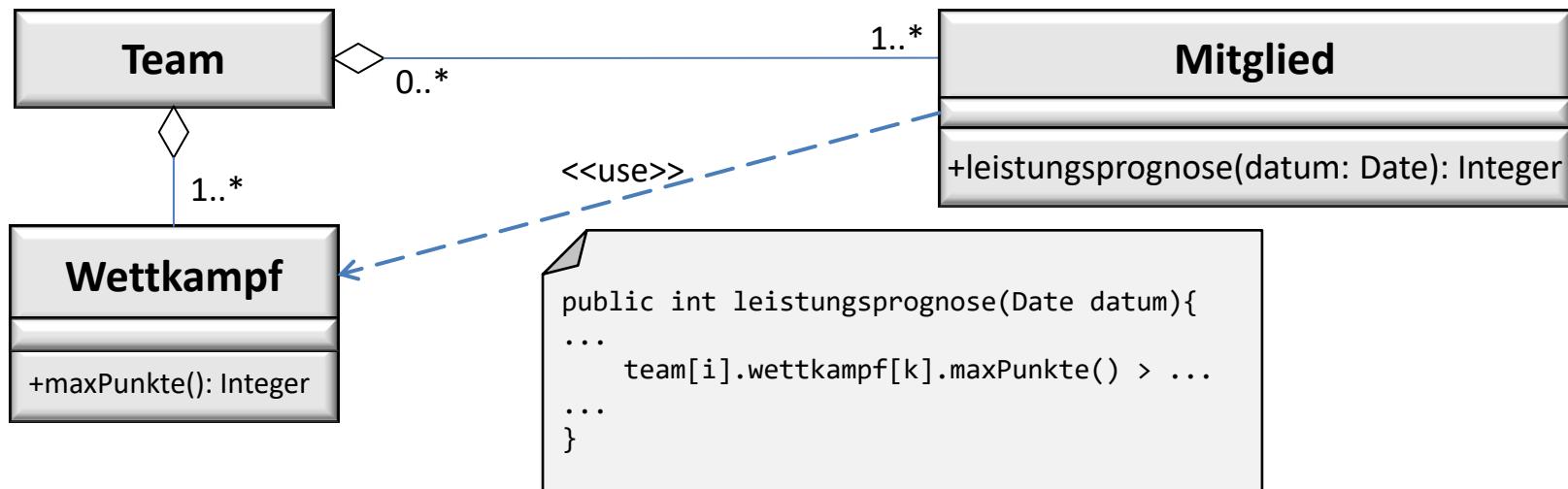




Abhängigkeiten (Dependencies)

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

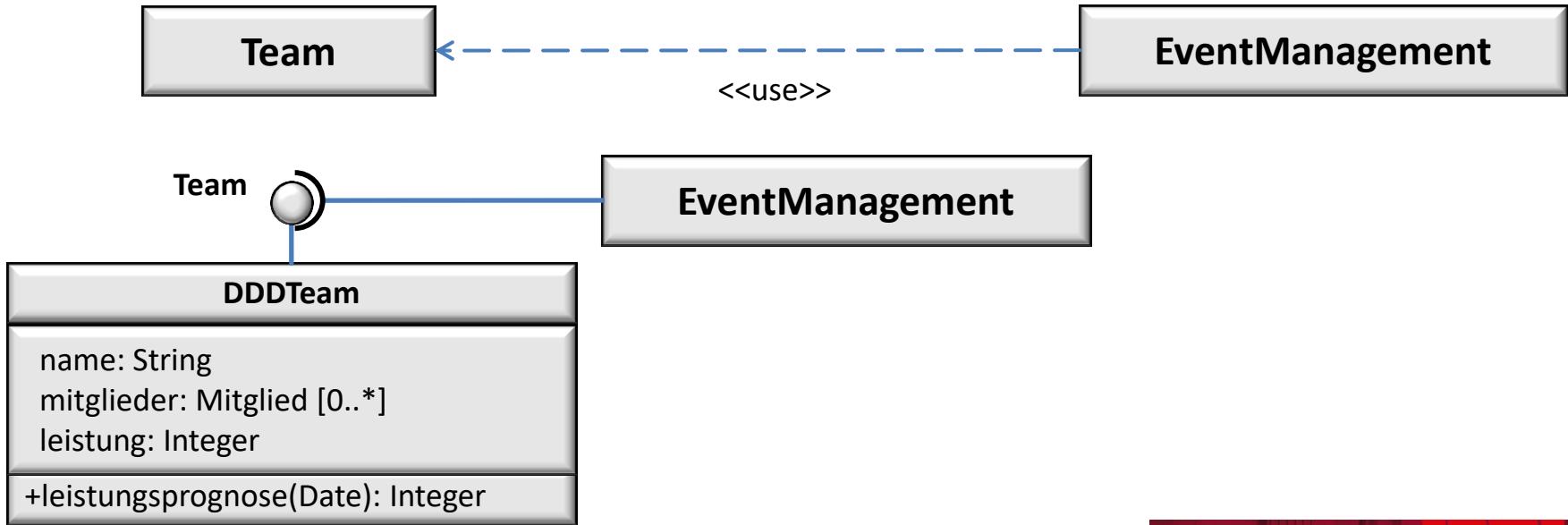
Um dies im Modell hervorzuheben, verwendet man **Dependencies**.





Abhängigkeiten (Dependencies)

Sehr häufig findet man Abhängigkeiten bei Interfaces.





Software-Engineering Entwicklungsprozesse

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



Prozessmodelle

Alle modernen Prozessmodelle sind Weiterentwicklungen des Versions- und Spiralmodells.

- Rumbaugh et. al. (OMT)
- The Unified Software Development Process (RUP)
(Jacobson, Booch, Rumbaugh)
- SCRUM (Schwaber, Sutherland)
- FDD,
- TDD,
- ...





Die Aufgabe des Prozessmodells

Ein Prozessmodell definiert im weitesten Sinne die zu erbringenden Aufgaben und deren zeitliche Abfolge, um ein gestecktes Ziel zu erreichen. Ein Prozessmodell definiert einen Rahmen, der eine effiziente Entwicklung ermöglicht. Es reduziert die Risiken und erhöht die Vorhersehbarkeit.

Bei der Auswahl oder besser Konfiguration des Prozessmodells sind viele Faktoren zu beachten!

- Technologien und Werkzeuge (UML, ...)
- Personen (Entwickler, Architekt, Projektverantwortliche, ...)
- Unternehmen (kleines Startups, großer Konzern, ...)
- Die Art der Aufgabe (kleine APP, großes ERP-System, ...)



Anforderungen an ein Prozessmodell

- **einfach und nachvollziehbar**

Nur was man versteht, setzt man um!

- **definiert und dokumentiert**

Nur was fixiert und verständlich beschrieben ist, kann eingehalten werden!

- **messbar**

Nur was messbar ist, kann überprüft und bewertet werden!

- **flexibel**

Nur was flexibel und adaptierbar ist, kann agil sein!

- **stabil** 

Nur was Stabilität besitzt, bleibt verlässlich!



Ziele, die mit einem Prozess verbunden sind

- Projekte innerhalb der veranschlagten Zeit und zum veranschlagten Preis abschließen.
- Produktivität, Qualität und Zufriedenheit steigern.
- Kosten und Aufwand für Pflege und Wartung senken.



Der richtige Prozess ist gefragt und dessen Umsetzung muss beherrscht werden!



CMMI: Capability Maturity Model Integration

CMMI Product Team: CMMI for Development, Version 1.3
TECHNICAL REPORT CMU/SEI-2010-TR-033 ,
Software Engineering Institute, Carnegie Mellon University, 2010

Ein Stufenmodell, das die Fähigkeiten eines Unternehmens definiert Projekte erfolgreich umzusetzen.

- Der voraussichtliche Projekterfolg wird vor der Projektvergabe messbar.
- Die voraussichtlichen Erfolgsaussichten werden kontinuierlich verbessert.



CMMI: Levels

| Level | <i>Capability Levels</i> | <i>Maturity Levels</i> |
|-------|--------------------------|------------------------|
| L0 | Incomplete | |
| L1 | Performed | Initial |
| L2 | Managed | Managed |
| L3 | Defined | Defined |
| L4 | | Quantitatively Managed |
| L5 | | Optimizing |



Capability Levels (Fähigkeitsgrade)

- **Incomplete:**
Prozesse werden nicht oder nur teilweise durchgeführt.
- **Performed:**
Prozesse existieren und werden individuell befolgt.
- **Managed:**
Prozesse existieren und ihre Durchführung wird überwacht und gesteuert.
- **Defined:**
Es existieren überwachte Prozesse und diese sind auf die jeweiligen Aufgaben präzise zugeschnitten und formal fixiert.



Maturity Levels (Reifegrade)

- **Initial:**

Die Arbeitsweise im Unternehmen ist chaotisch und unorganisiert. Projekte stehen und fallen mit einzelnen Mitarbeitern.

**Unternehmen überschätzen sich,
Erfolge sind nicht wiederholbar.**



**Termine werden selten eingehalten
und die Kosten laufen aus dem Ruder!**



Maturity Levels (Reifegrade)

- **Managed:**

Arbeitsschritte und Aufwand werden geplant und überwacht, und Ressourcen werden in ausreichendem Umfang zur Verfügung gestellt.



Auch in Stresssituationen werden die definierten Prozesse eingehalten.



Maturity Levels (Reifegrade)

- **Defined**

Unterschiedlichste Arbeitsschritte sind normiert und werden unternehmensweit eingehalten.

Auch der Vorgang der Anpassung eines Prozesses auf die Anforderungen eines spezifischen Projekts ist definiert.



Prozesse sind definiert, werden überwacht und sind generisch!



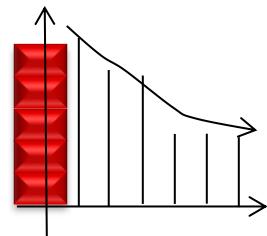
Maturity Levels (Reifegrad)

- **Quantitatively Managed**

Es existieren nicht nur Meilensteine, sondern quantitativ messbare Vorgaben, die auf die Bedürfnisse der Projektbeteiligten zugeschnitten sind.

Diese Vorgaben werden durchgängig überwacht und statistisch ausgewertet!

Der Prozesserfolg (Leistung) wird vorhersehbar!





Maturity Levels (Reifegrad)

- **Optimizing**

Es existieren generische Prozesse, die quantitativ überwacht werden.

Die dadurch gewonnenen Ergebnisse werden reflektorisch zur Verbesserung der etablierten Prozesse eingesetzt.

Eine permanente Verbesserung der Unternehmensleistung!





Ein Prozessmodell

Oftmals sind Prozessmodelle ein Glaubenskampf. Es gibt die Verfechter der einen und die Verfechter der anderen Lehrmeinung. In der Praxis findet man viele, die behaupten, das eine zu tun und sie tun doch das andere.

Nicht immer ist die reine Lehre gefragt, sondern vielmehr Best Practice.
Unser Ziel: eine Methode, die uns von den Requirements bis zur Implementierung führt.
Vergleichbar einem Expeditionshandbuch für Neulinge. Wir wollen Herausforderungen
meistern und bewältigen, wissen aber sehr wohl, dass wir keine Freikletterer sind.

Als Basis hierfür dient
der Unified Process!



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. 

Der Unified Process ist
anpassungsfähig und
individualisierbar!



Der Unified Process ist ein Framework

Er muss angepasst werden an:

- die Größe der zu erstellenden Anwendung,
- das Einsatzgebiet der zu erstellenden Anwendung,
- die Fähigkeiten und Erfahrungen der Entwickler,
- die Unternehmensstruktur.



- **Wie viele Iterationen sind notwendig?**
- **Wann ist die Architektur abgeschlossen?**
- ...

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



Die Notwendigkeit der Iteration

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



Man zerlegt die Aufgabe in Mini-Projekte! (Iterationen oder Sprints)

- 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.

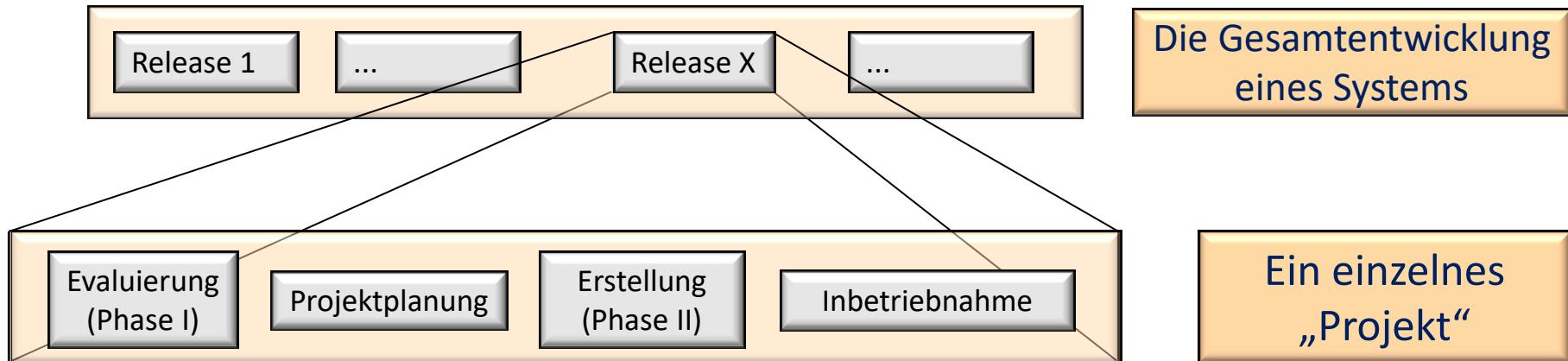
Eine Iterationen muss geplant sein und einem definierten Ablauf folgen.
Schritte, die das Projekt dem Ziel nicht näher bringen, müssen unterbleiben.



Ein Iterativer Entwicklungsprozess

Mit dem Unified Process in vier Schritten zum Erfolg!

(Nach Craig Larman „ Applying UML and Patterns“)

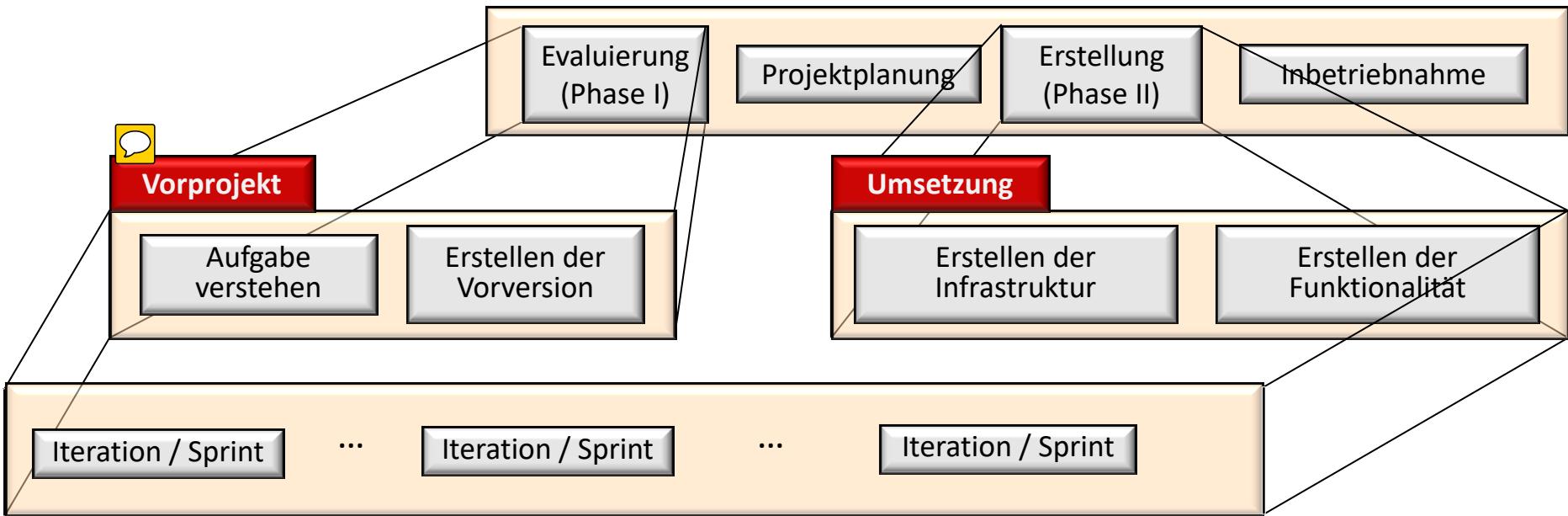


Larman, Craig.

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.



Die Hauptaufgaben im Projekt



Die einzelnen Aufgaben werden in Iterationen bzw. Sprints organisiert!



Die zentralen Aufgaben der Phase I

- **Aufgabe verstehen**

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

Die zu realisierende Kernfunktionalität wird strukturiert und die prinzipielle Architektur wird festlegt.

- **Erstellen der Vorversion**

Die ersten Entwicklungsarbeiten werden durchgeführt (1-3 Monate bzw. 1-2 Iterationen). Ziel ist es, zentrale Aspekte der Kernfunktionalität umzusetzen und Architekturgesichtspunkte zu fixieren.

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



Die zentralen Aufgaben der Phase II

- **Erstellen der Infrastruktur** 

In den ersten Iterationen der Erstellungsphase wird dann die verbliebene Kernfunktionalität umgesetzt. Dies ist verbunden mit der Konsolidierung der Systemarchitektur. Komponenten und Schichten (die Architektur) werden verfeinert. Neue Anforderungen / Requirements werden ggf. erfasst und in die Entwicklung integriert.

Die Architektur reift.

- **Erstellen der Funktionalität**

Die späteren Iterationen konzentrieren sich im Allgemeinen auf die Vervollständigung der Funktionalität. Dies beeinflusst die Systemarchitektur nur unwesentlich. Gegebenenfalls werden noch Schnittstellen erweitert.

Neue Anforderungen sollten nicht mehr hinzu kommen!



Gedanken zu SCRUM

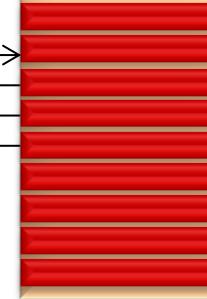
Requirements

Planen und Erarbeiten

Erstellen der Vorversion



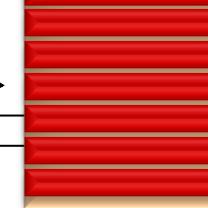
Sprint Backlog



Grooming

Arbeitspakete (Tasks)

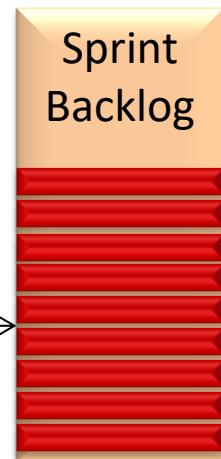
Sprint Backlog



Sprint Backlog



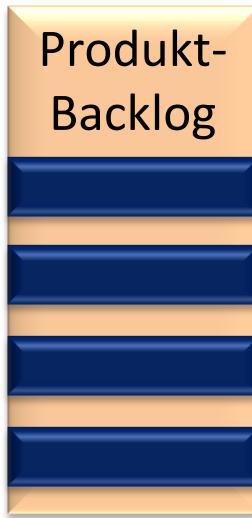
Phase II: Umsetzung





Gedanken zu SCRUM

Requirements



Phase I: Vorprojekt

Planen und Erarbeiten

Erstellen der Vorversion

Ziel:

den initialen Produkt-Backlog definieren, die zu erwartende Anzahl von Sprints festlegen und die Aufgaben grob verteilen!

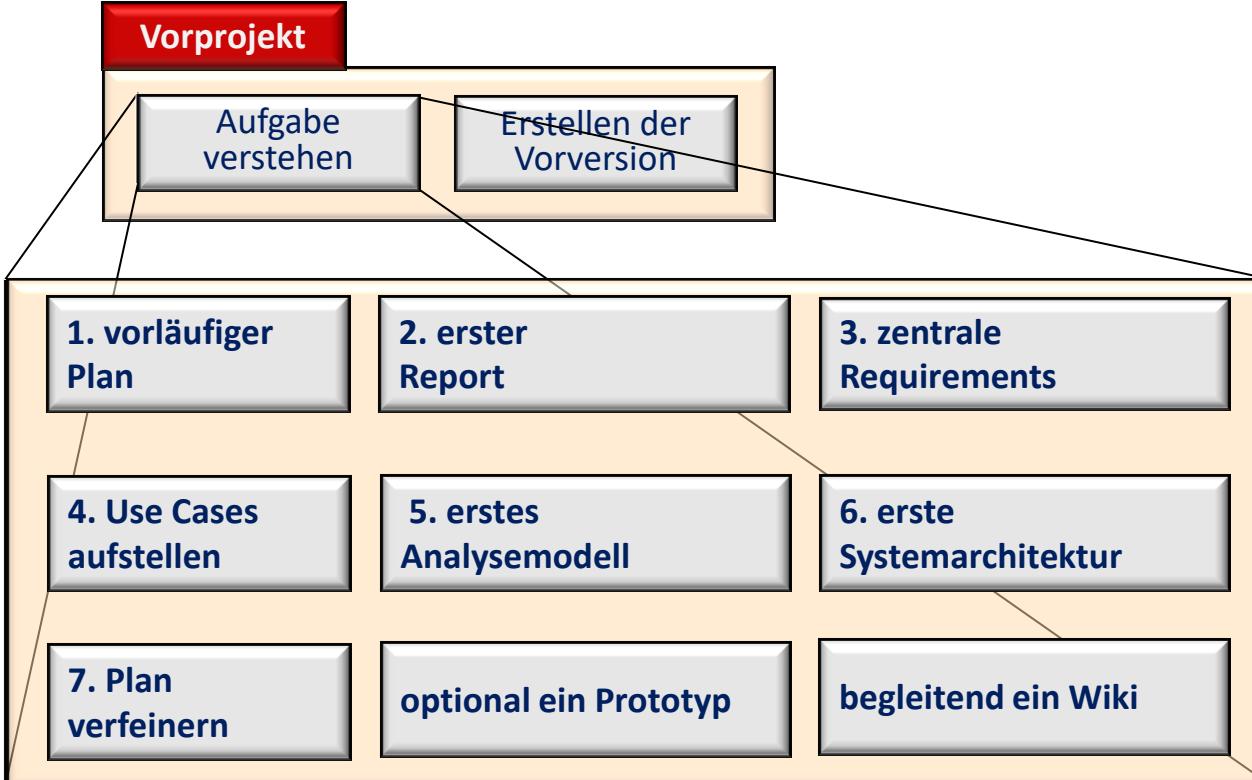
Sprint
Backlog

Sprint
Backlog

Sprint
Backlog



„Aufgabe verstehen“ im Detail



Aufwand und Umfang steigt mit der Größe des Projektes!



„Aufgabe verstehen“ im Detail

- **vorläufiger Plan: (Ist die Aufgabe zu bewältigen?)**

Geschätzt werden Zeit, zu erwartenden Aufwände und Kosten, der notwendige Personal- und Materialbedarf, usw.

- **erster Report: (Ist die Aufgabe erfolgversprechend?)** 

Er beinhaltet die Motivation, mögliche Alternativen, eruiert das Marktpotential und definiert die eigene Zielsetzung, ...

- **zentrale Requirements: (das Lastenheft)**

Meist eine deklarative Auflistung der Anforderungen aus Kundensicht. Diese lassen sich leicht in Workshops zusammen mit dem Auftraggeber erarbeiten.



„Aufgabe verstehen“ im Detail

- **Use Cases aufstellen: (Aufgabe strukturieren)**

Zentrale Abläufe („Geschäftsvorfälle“) werden in Form von Szenarien beschrieben und zueinander in Beziehung gesetzt, um die Anforderungen des Auftraggebers in großen Einheiten zu erfassen und zu strukturieren.

- **erstes Analysemodell: (Problemraum definieren)**

Neben den Use Cases wird die Problemdomäne erfasst, deren zentrale Zusammenhänge offengelegt. Strukturen werden erkennbar und die Anforderungen des Auftraggebers werden greifbar.

- **erste Systemarchitektur: (die zu erwartende Struktur der Lösung)**

Ergibt sich direkt aus dem Analysemodell und den nicht-funktionalen Requirements des Auftraggebers.



„Aufgabe verstehen“ im Detail

- **Plan verfeinern: (Ziel der Vorversion definieren)** 

Festgelegt wird der Teil der Kernfunktionalität, der zuerst realisiert wird. Was ist Gegenstand der Vorversion und wie viele Iterationen sind hierfür geplant.

- **optional ein Prototyp: (Risiken minimieren)**

Ein Prototyp kann helfen, offene Fragen in Bezug auf Wünsche und Vorstellung des Auftraggebers zu klären. Er schafft Planungssicherheit und reduziert das Risiko einer späteren Fehlentwicklung.

- **begleitend ein Wiki: (Entscheidungen werden nachvollziehbar und auffindbar)**

Alles vom Requirement bis hin zu Klassen und einzelnen Funktionen wird in einem glossarartigen Wiki gelistet und definiert. Dies schafft Klarheit und reduziert das Risiko für Missverständnisse.



Der endgültige Projektplan



Nach der Evaluierungsphase sollte Planungssicherheit bestehen.
Die Ergebnisse der Evaluierungsphase (insbesondere die erstellte Vorversion) können mit den Auftraggebern abgestimmt werden.

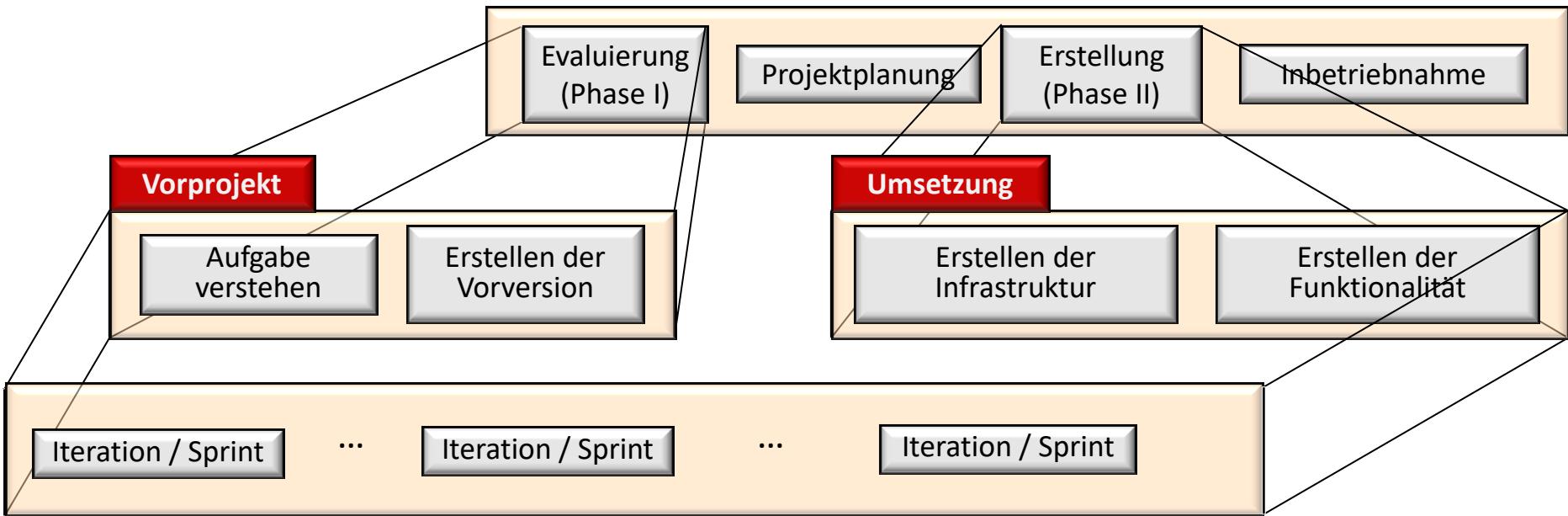
**Jetzt fällt die Entscheidung über Realisierung oder nicht!
(Kosten und Zeitverlauf lassen sich bestimmen!)**

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

- die Entwicklungsschritte mit den Meilensteine werden definiert (Netzplan),
- **die Anzahl der zu erwartenden Iterationen und deren Umfang werden festgelegt,**
- die Zahlungspläne werden ausgehandelt,
- **Personalverteilung und die Teams werden aufgestellt (Organisationsstruktur),**
- externe Kräfte, Zulieferer und Qualitätssicherung werden gebucht,
- Hardwarebeschaffungen werden geplant, ...



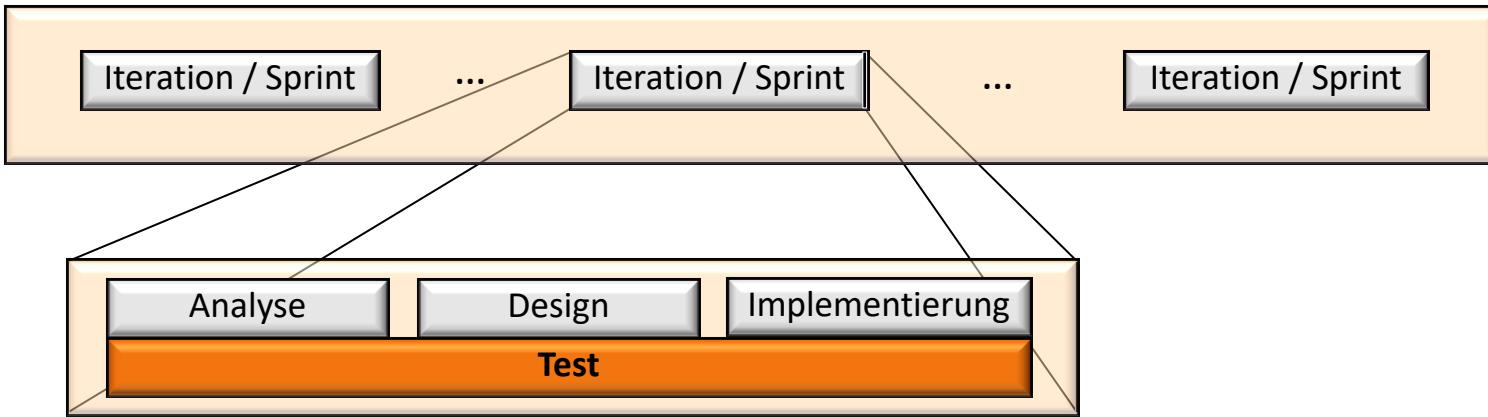
Die Hauptaufgaben im Projekt



Die einzelnen Aufgaben werden in Iterationen bzw. Sprints organisiert!



Jede Iteration ist ein Mini-Projekt



Testen gehört zu den wichtigsten Aufgaben und wird oftmals vernachlässigt!



Der Umfang einer Iteration

Die Entscheidung darüber, welche Anforderungen innerhalb eines Entwicklungszyklus realisiert werden, sollte immer in Zusammenarbeit mit dem **Entwicklungsteam** erfolgen.

Die Entwickler sind diejenigen, die die Ergebnisse liefern müssen!





Der Dauer einer Iteration

Ist sie zu kurz, wird es schwierig, eine sinnvolle Menge an Funktionalität umzusetzen. Ist sie zu lang, läuft man Gefahr die Komplexität falsch einzuschätzen und das Feedback auf die ersten Ergebnisse entfällt.

Faustregel:

Eine Iteration sollte nicht kürzer als 2 Wochen und nicht länger als 2 Monate sein.

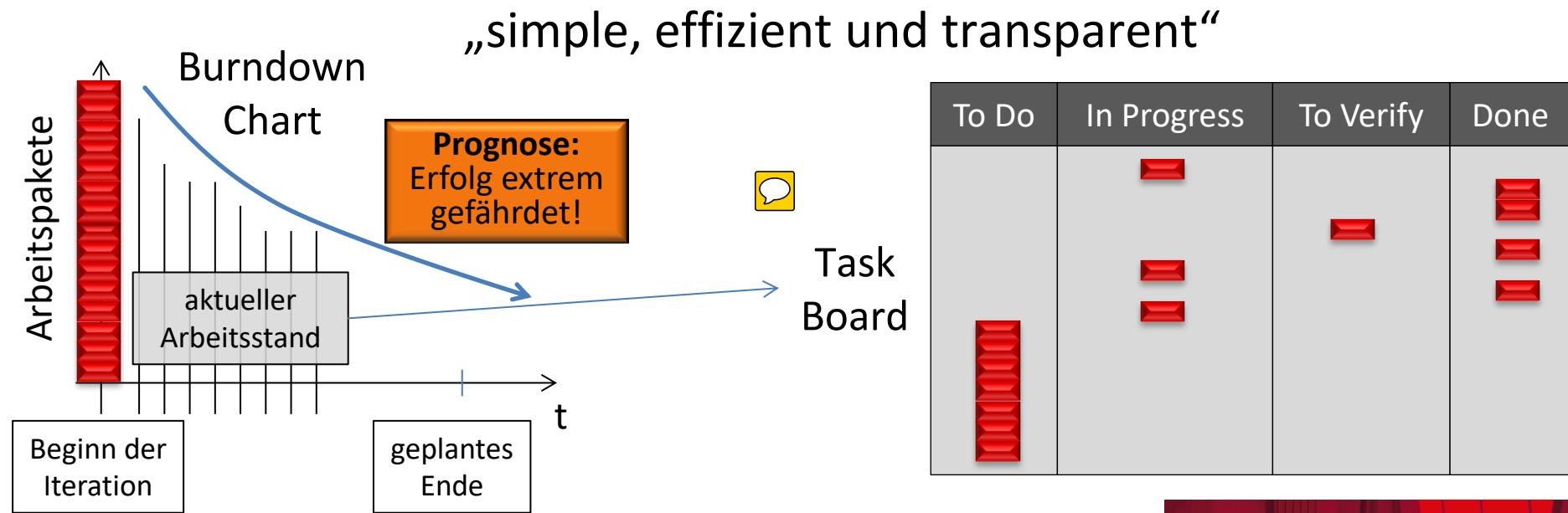
Faktoren, die den Erfolg gefährden:

- Einführung neuer Technologien,
- Personalprobleme (Mangel an Experten),
- Zusammenarbeit im Team (Koordination und Kommunikation frisst Zeit),
- Unterschätzen der Komplexität – insbesondere in frühen Phasen. Die Erarbeitung der Architektur ist meist aufwendiger als gedacht.



Feste Zeitvorgaben

Feste Zeitvorgaben und ein definierter Umfang sind unumgänglich. Mit ihnen steht und fällt das Projektmanagement.





Die Schritte innerhalb einer Iteration

- **Analyse (Problemraum)**

Requirements (Anforderungen) erfassen, strukturieren, verstehen, aufeinander abstimmen, ...

- **Design (Lösungsraum)**

Entscheidungen treffen, sowohl im Großen als auch im Kleinen, und die Umsetzung definieren.

- **Implementierung**

Die getroffenen Designentscheidungen werden in Code verwandeln, **nicht die Analyseergebnisse!**

Iterationen erfordern, dass Dokumente und Modelle stets synchronisiert werden, um sie konsistent zu halten.



Testen

- **Akzeptanz- und Systemtests: (Im Fokus das Gesamtsystem)**

Validiert werden die Anforderungen und die daraus abgeleiteten Entscheidungen.
Überprüft werden nicht-funktionale Anforderungen.

- **Systemintegrations-/Komponentenintegrationstest: (Im Fokus Teilsysteme)**

Verifiziert werden die getroffenen Architekturentscheidungen und wie die einzelnen Bausteine des Systems zusammenarbeiten. Typischerweise erfolgt dies hierarchisch. Kleine, einzeln getestete Komponenten bilden große Komponenten auf der nächsthöheren Testebene.

- **Modultests: (Im Fokus Units – „kleine“ Komponenten)**

Verifiziert wird die korrekte Implementierung der zu realisierenden Operationen.
Man unterscheidet zwischen White-box- und Black-box-Tests. Je nachdem, ob man den Tests Wissen über die Implementierung zugrunde legt oder nicht.



Software-Engineering Analyse

(angelehnt an Craig Larman „Applying UML and Patterns“)

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



Analyse

Analyse bedeutet, die gestellte Aufgabe verstehen und so aufzubereiten, dass ein korrekter Entwurf möglich wird. Hierzu wird ein Analysemmodell erstellt. Je komplexer die Aufgabe ist, desto aufwendiger und wichtiger wird die Analyse.

- **Erstellen des Analyse-Use-Case-Modells**

Dies dient der Beschreibung und Strukturierung der zu realisierenden Funktionalität unter Berücksichtigung der gegebenen Anforderungen. 

- **Erstellen des Analyse-Objektmodells**

Dies dient der Beschreibung der Problemdomäne.

- Klassen finden und organisieren – wer steht mit wem in Beziehung.
- Verantwortlichkeiten und Beziehungen aufzeigen – wer ist für was verantwortlich.
- Zeitliche Veränderungen herausarbeiten – wie verändern sich Objekte (Zustände).

- **Erstellen des funktionalen Modells**

Dies dient der Beschreibung der zu erwartenden Schnittstellen.

- Welche Operationen sind erforderlich?
- Wie darf man sich die Interaktion zwischen den beteiligten Systemteilen und Anwendern vorstellen?



Requirements und das Use-Case-Modell



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

Probleme verstehen, bedeutet Anforderungen erkennen, analysieren und bewerten.

- Die Aufgaben müssen beschrieben werden: **Um was geht es!**
- Die Anwender müssen bestimmt werden: **Wer nutzt das System, wen nutzt das System!**
- Der Umfang muss bestimmt werden: **Was kann das System!**
- Die Eigenschaften müssen bestimmt werden: **Was zeichnet das System aus!**
- Grenzen müssen definiert werden: **Was ist Teil des Systems und was nicht!**

Anforderungen sind so zu formalisieren, dass sie unzweideutig sind.



Beispiel: Mitgliederverwaltung

- **Aufgabenbeschreibung:** Entwicklung einer einfachen aber erweiterbaren Mitgliederverwaltung für kleine und mittlere Vereine. In der ersten Version als simple Desktop-Anwendung, die bei Erfolg als Software as a Service in die Cloud migriert werden kann.
- **Anwender:** Pilotkunde ist der „SV Hoch und Weit“, dessen bisherige auf Excel basierende Lösung an ihre Grenzen gestoßen ist.
- **Ziele:** Marktanalysen haben gezeigt, dass der Sektor „Vereinssoftware“ boomt und weiterhin ein enormes Wachstumspotential verspricht, aber gerade im Bereich individualisierbarer Software für kleine und mittlere Vereine unversorgt ist. Die Software soll diese Nische besetzen und dem Kunden folgende zentralen Merkmale bieten:
 - schnelle Erfassung neuer Mitgliedern,
 - einfaches Ändern bestehender Mitgliedsdaten,
 - leichte Anbindung an die Beitragserfassung,
 - ...



schöner,
schneller, weiter



Anforderungen (Requirements)

Problem verstehen, bedeutet Anforderungen erkennen, bewerten und aufbereiten!

- Systemfunktionalität erfassen (Was kann das System!)
- Systemeigenschaften erfassen (Was zeichnet das System aus!)
- Systemgrenzen definieren (Was ist Teil des Systems und was nicht!)

**Das Fundament einer
erfolgreichen Entwicklung!**



funktionale Anforderungen

Was soll das System können?

Ein Softwaresystem wird erstellt, um Aufgaben für seine Anwender (Menschen, Systeme, ...) zu erledigen. D.h., um ein erfolgreiches System zu entwickeln, ist es notwendig, dass diese Aufgaben (funktionale Anforderungen) erfasst und verstanden werden.



Funktionale Anforderungen beschreiben das „Was“ einer Lösung.



funktionale Anforderungen

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

- **typische Kategorien:**
 - **sichtbar:** Die Funktionalität wird vom Anwender erwartet, ein Fehlen im Produkt wäre fatal!
 - **versteckt:** Die Funktionalität wird von Anwender nicht wahrgenommen, sie ist für das Produkt jedoch essentiell.
 - **optional:** Die Funktionalität wird vom Anwender nicht erwartet, ein Fehlen wäre unkritisch.
- **Gruppierungen entstehen durch Zuordnungen im Anwendungsbereich:**
 - **Basisfunktionalität (Mitglieder verwalten)**
 - Eine Adresse ändern
 - Eine Willkommens-Mail versenden
 - ...
 - **Meldungen an den Verband**
 - ...



nichtfunktionale Anforderungen

Was soll das System können?

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

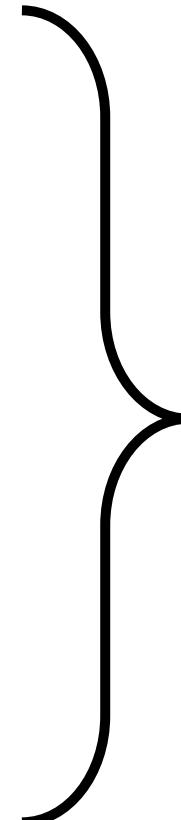


**Nichtfunktionale Anforderungen (Constraints)
beschreiben das „Wie“ einer Lösung.**



nichtfunktionale Anforderungen

- **Technische Constraints**
 - Plattformen,
 - Programmiersprachen,
 - ...
- **Unternehmerische Constraints**
 - Unabhängigkeit,
 - Strategische Ausrichtung,
 - Kosten,
 - ...
- **Qualitätsspezifische Constraints**
 - Skalierbarkeit,
 - Zuverlässigkeit,
 - Leistungsfähigkeit,
 - Sicherheit,
 - ...



Unterschiedliche Concerns
verschiedener Stakeholder

- **Management**
 - Kosten, Gewinn, ...
- **Vertrieb**
 - Time to Market, ...
- **Anwender**
 - Usability, Performance, ...
- **Entwickler**
 - ...



Anforderungen (Requirements)

Anforderungen lassen sich leicht tabellarisch erfassen!

F1: Basisfunktionen

| <u>Ref.#</u> | Funktion | <u>Kategorie</u> |
|--------------|--------------------------------------|------------------|
| F1.1 | Adresse ändern: Die ... | sichtbar |
| ... | | |
| F1.5 | Mail-Versand protokollieren: Der ... | versteckt |

A1: Sicherheitsanforderungen (Datenschutz)

| <u>Ref.#</u> | Eigenschaft | <u>Kategorie</u> |
|--------------|---|------------------|
| A1.1 | Zugriff auf Mitgliederdaten: Der ... authentisiert ... | want |
| ... | | |
| A1.7 | Versand von Mitgliederlisten: Der ... verschlüsselt ... | must |



Die Analyse beginnt (Schritt 1)

Problem verstehen, bedeutet Anforderungen erkennen, bewerten **und aufbereiten!**

- 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 Anforderung an das System – egal ob funktional oder nichtfunktional – sollte mindestens einem Use Case zugeordnet werden können!



Die Rolle der Use Cases

Anwendungen bzw. Systeme werden erstellt, um dedizierte Bedürfnisse (Anforderungen) von Anwendern und Kunden zu erfüllen. Nur wenn diese Anforderungen (Requirements) erfasst und verstanden werden, kann eine Entwicklung erfolgreich sein.

Zum Aufbereiten und verstehen von Anforderung verwendet man Use Cases.

Damit bilden Use Cases die Grundlage für:

- die Analyse,
- die Architektur,
- das Design,
- die Implementierung,
- die Tests,
- die Evaluation, ...

Für alles!



Use Cases

Use Cases sind grobe Beschreibungen des Ablaufs aus Sicht der beteiligten Akteure.

Aufbau eines Use Case:

- **Name:** „Verb ...“
- **Akteure:** Nutzer, Fremdsysteme,...
- **Systemgrenze:** Das Systems oder die verantwortliche Komponente
- **Prio:** 1, 2 oder 3
- **Requirements:** Referenzen auf die zu erfüllende funktionalen und nicht-funktionalen Anforderungen; F1.1,...
- **Beschreibung**

Beschreibung bei high-level Use Cases:
Wenige Sätze beschreiben übersichtlich die zentralen Abläufe.

Beschreibung bei erweiterten Use Cases:
Detaillierte Darstellung der Abläufe unter Berücksichtigung der beteiligten Akteure.



Use Cases

Use Cases helfen das Projekt zu strukturieren und zu organisieren.

Es gilt folgendes Prinzip:

Zuerst werden die Use Cases realisiert, die die zentralen Strukturen der Anwendung (die Architektur) beeinflussen, oder als besonders kritisch eingestuft und damit mit hohen Risiken für die Entwicklung verbunden sind.

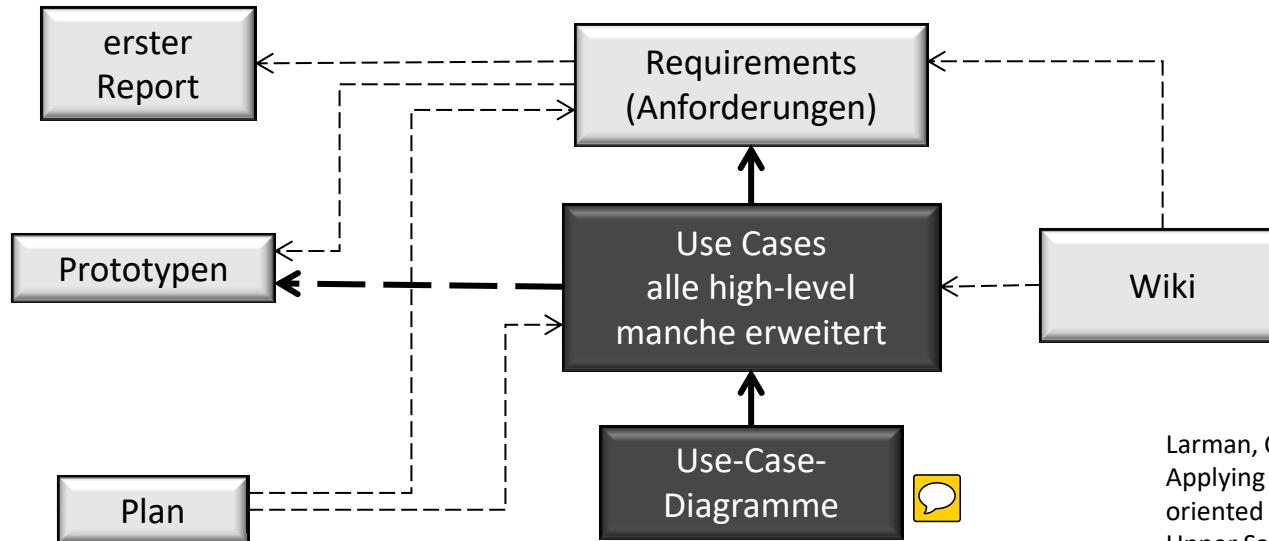
Der moderne Entwicklungsprozess ist:

- use-case-driven und
- architekturorientiert.



Requirements analysieren

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



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.



Beispiel

- **Name:** Mitglieder verwalten
- **Akteure:** Vereinsmanager, Mail Client
- **Systemgrenze:** MyClub
- **Prio:** 1 (hoch)
- **Referenz:** F1.1, F 1.2, F 2.1, ...
- **Vorbedingung:** MyClub ist korrekt installiert und der Vereinsmanager hat sich bereits authentisiert.
- **Beschreibung:** Der Vereinsmanager hat sich entschlossen, die anstehenden Mitgliedsänderungen durchzuführen. Er kann nun entweder ein neues Mitglied anlegen, oder aus der Liste aller Mitglieder ein Mitglied auswählen, dass er bearbeiten möchte. Etwa seine Adresse ändern, Fehler im Namen korrigieren oder auch die Abteilungszugehörigkeit auf den neuesten Stand bringen. Auch ein Austritt aus dem Verein sollte möglich sein. Liegt ein Abteilungsaustritt oder -beitritt vor, werden der entsprechende Abteilungsleiter und das Mitglied per Mail informiert. Ist ein Mitglied keiner Abteilung mehr zugeordnet, wird es als passiv geführt.
- **Offene Punkte:** Soll ein Mitglied per Mail über jede Änderungen oder nur bestimmte informiert werden?

High-level Use Cases erzeugen schnell ein Problemverständnis.



Typische Fehler

Ein Use Case ist weder eine einzelne Aktion noch ein einzelner Schritt.

Keine Use-Cases sind:

Den Status einer Mitglieds von passiv auf aktiv setzen.

Eine Mail an den Abteilungsleiter versenden.

Use Cases sind zusammenhängende Einheiten aus Aktionen und Schritten, die als Einheit im Anwendungsumfeld sinnvoll sind.



Wie findet man Use Cases

• Über 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 Ereignisse

- Man bestimmt die externen Ereignisse auf die ein System reagieren muss.
- Die Ereignisse werden dann zu Akteuren und Use Cases in Beziehung gesetzt.

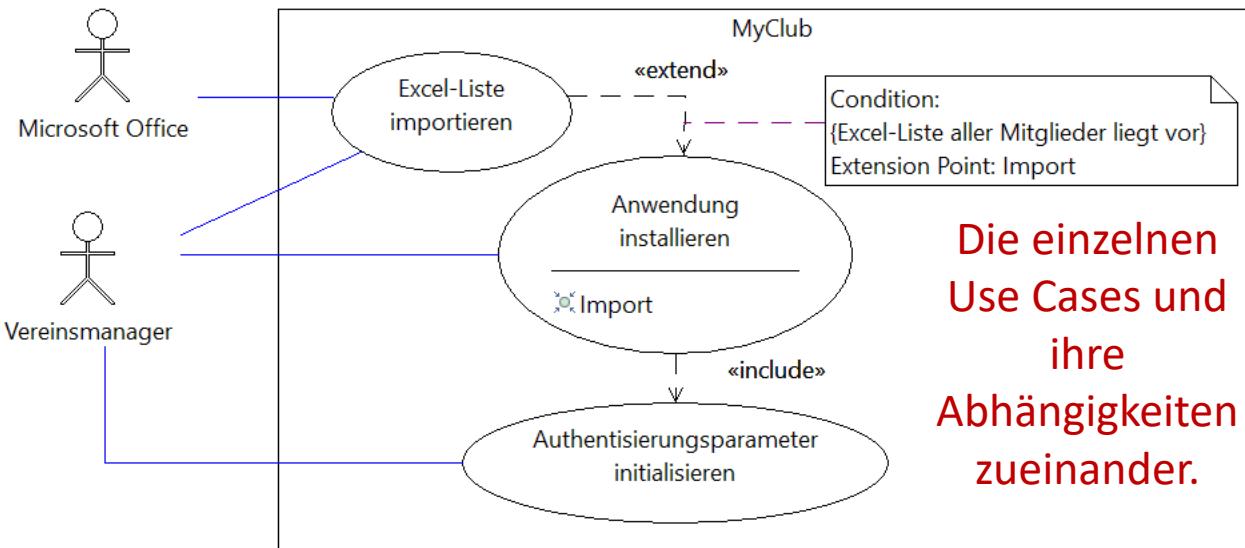
Wie reagieren System und Akteure auf die Ereignisse, welche Vorgänge lösen sie aus?



Use Cases und die UML

Die Akteure,
die mit dem
System
interagieren.

Use-Case-Diagramme helfen bei der Strukturierung
und erzeugen Übersicht in der Masse der Use Cases?



Die einzelnen
Use Cases und
ihre
Abhängigkeiten
zueinander.

Das System,
das die Use Cases
realisiert und die
Abgrenzung zur
Umgebung.



Akteure

Akteure sind nicht Teil des Systems, das betrachtet wird. Sie sind aber an der Durchführung des Use Case beteiligt oder direkt davon betroffen. 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 kein Anwendungsfall – niemand braucht ihn.



Beziehungen bei Akteuren

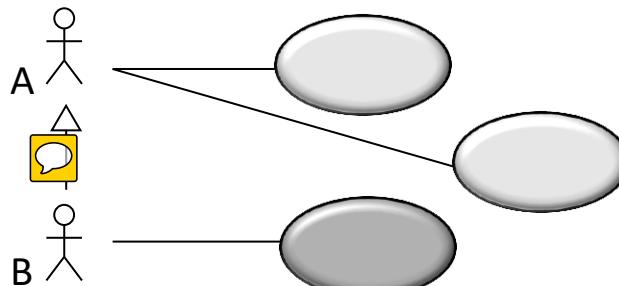
- **Assoziation (Akteur <-> Use Case)**

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



- **Generalisierung**

Eine Instanz des Akteurs „B“ kann mit allen Use Cases kommunizieren mit denen auch „A“ kommunizieren kann plus die, an denen „B“ beteiligt ist.

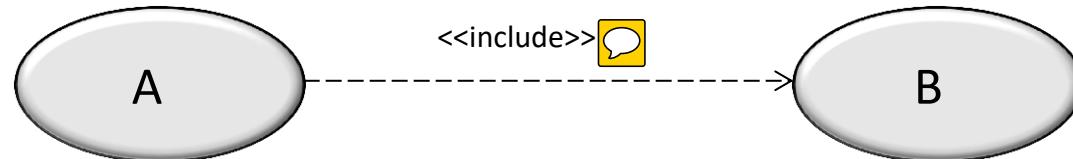




Beziehungen zwischen Use Cases

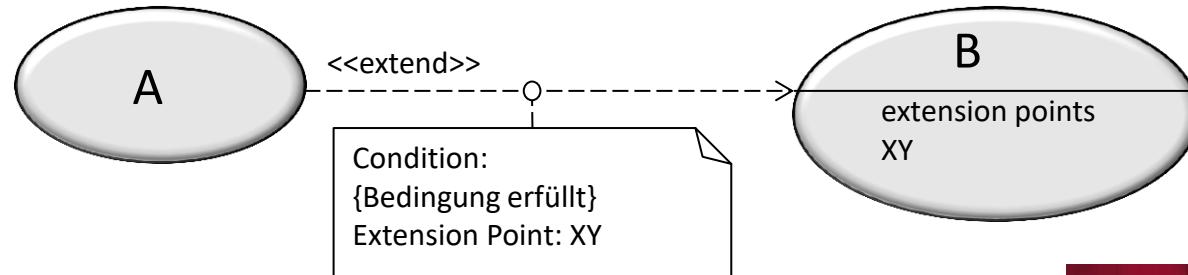
- **Beinhalten (<<include>>)**

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



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

Die Aktionen des Use Cases „A“ kommen im Use Case „B“ hinzu, falls die entsprechende Bedingung am Extension Point erfüllt ist.

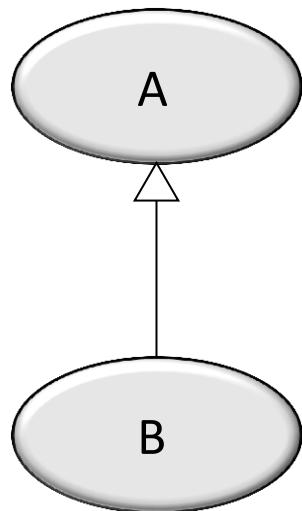




Beziehungen zwischen Use Cases

- **Generalisierung**

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



Die Generalisierung 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 auch alle „Beziehungen“ von A.



Faustregel

- **include**

Man verwendet **include**, wenn sich in mehreren Anwendungsfällen große, eigenständige Abläufe wiederholen und man dies vermeiden möchte.

- **extend**

Man verwendet **extend**, wenn in einem Use Case Variationen auftreten, die als eigenständige Abläufe verstanden werden können und die Punkte an denen Variationen im Basis-Use-Case auftreten greifbar und offensichtlich sind.

- **Generalisierung**

Man setzt auf **Generalisierung**, wenn man hervorheben möchte, dass es sich bei der Spezialisierung um eine Variante eines allgemeineren Anwendungsfalls handelt.



Typische Fehler

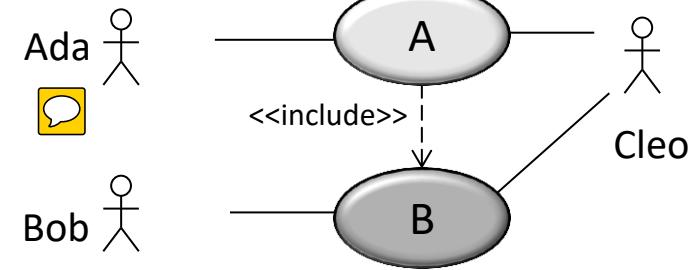
- **Zeitliche Abfolgen**

Weder **extend** noch **include** beziehen sich auf die zeitliche Abfolge von Use Cases. Müssen im Rahmen eines komplexen Workflows unterschiedliche Use Cases in einer definierten Reihenfolge abgearbeitet werden, dann kann dies **nicht** im Use-Case-Diagramm dargestellt werden.

- **Verzicht auf Assoziationen**

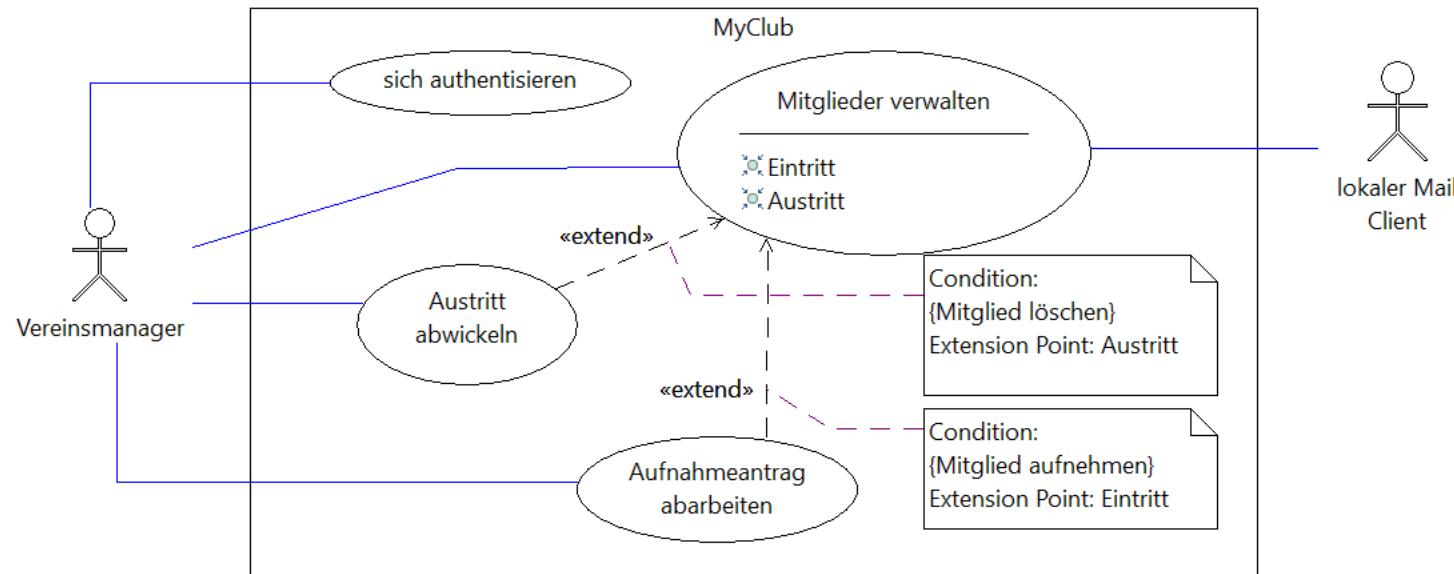
Weder **extend** noch **include** stellen Vererbungsrelationen zwischen Use Cases dar. Insofern werden auch keine Beziehungen zu beteiligten Akteuren vererbt. Ein Weglassen von Assoziationen ist nicht zulässig.

Ada ist nicht am Use Case B beteiligt und Bob nicht am Use Case A. Lediglich Cleo ist an beiden Use Cases beteiligt.





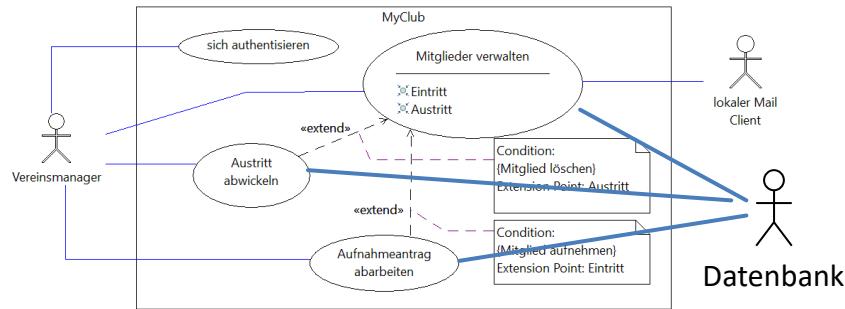
Beispiel: MyClub





Beispiel: MyClub (offene Fragen)

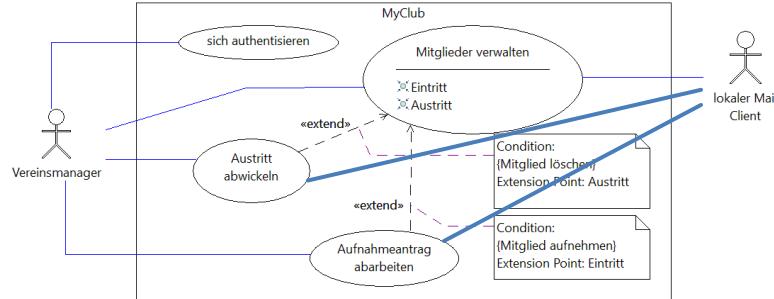
- Weitere Akteure?



Eher nein:

Dass Daten zu speichern sind, steht außer Frage. Ob hierfür eine Datenbank benötigt wird, wird sich zeigen.

- Weitere Assoziationen?



Eher ja:

Das Versenden von Mails ist vorgesehen und warum sollte dies beim Austritt oder Beitreitt unterbleiben?



Use Cases verstehen (Schritt 2)

Nachdem Requirements erfasst und die Anwendungsfälle identifiziert und strukturiert wurden, müssen die Anwendungsfälle aufgearbeitet und verstanden werden. Hierzu verwendet man Szenarien. Diese sind Beschreibungen der Interaktionen zwischen den Akteuren und dem System in einer mehr oder weniger konkreten Situation.

- Je komplexer der Use Case, desto mehr Szenarien werden gebraucht.
- Je einfacher der Use Case, desto allgemeiner kann das Szenario ausfallen.

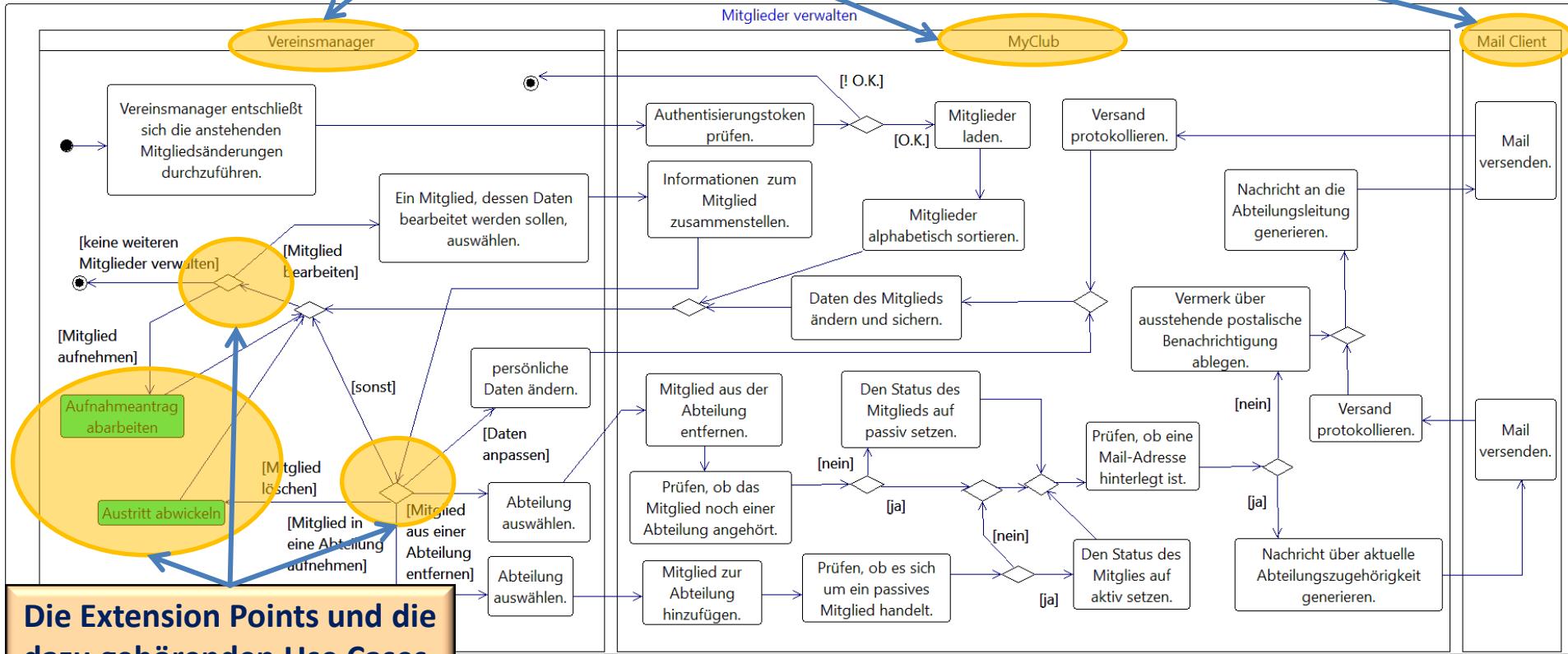
Zur Beschreibung einzelner Szenarien nutzt man Interaktionsdiagramme, Zustandsdiagramme oder Aktivitätsdiagramme – manchmal auch nur Text.



Activity-Diag

Die Abgrenzung zwischen System und den beteiligten Akteuren.

Mitglieder verwalten“



Die Extension Points und die dazu gehörenden Use Cases.



Activity-Diagramme und die UML

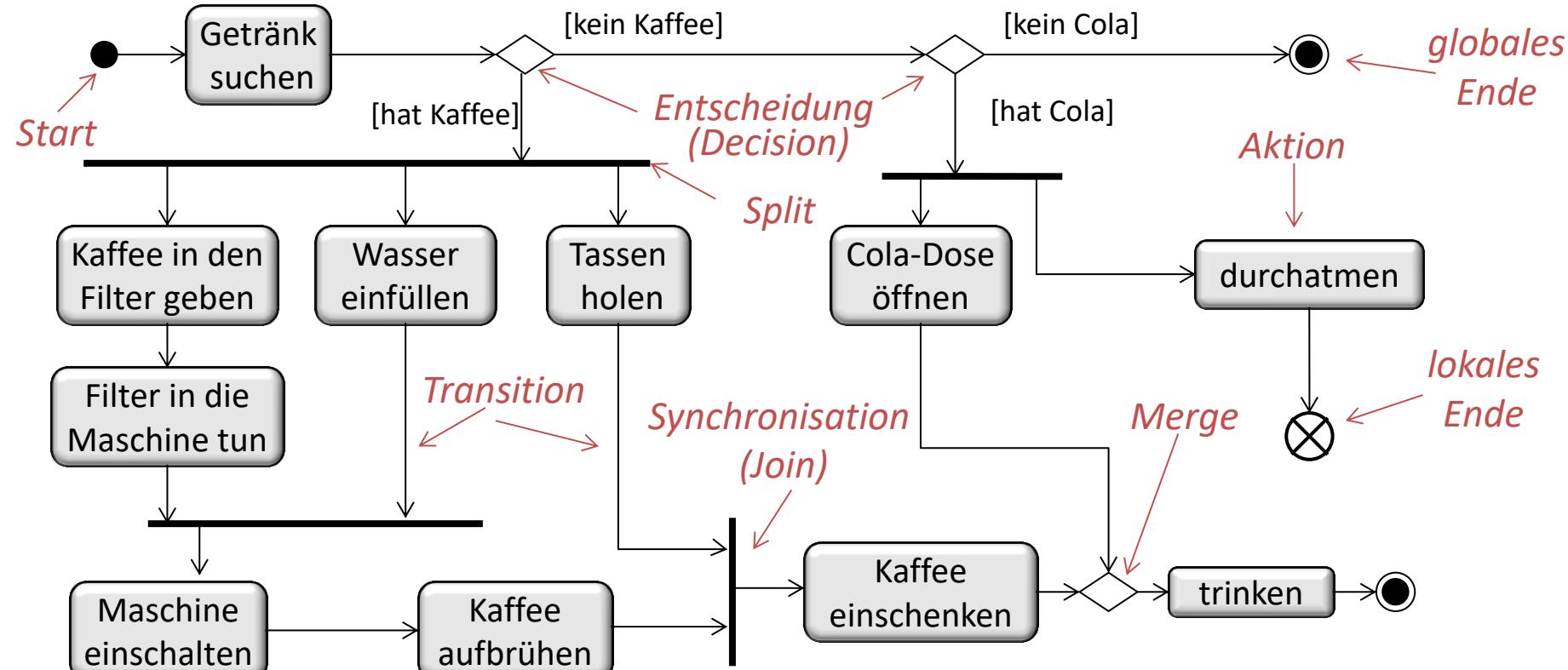
Ein Aktivitätsdiagramm ist ein Diagramm von zusammenhängenden Aktionen beliebiger Granularität. Ein Übergang (Transition) ist nicht mit einem Ereignis, sondern mit dem Abschluss der entsprechenden Aktion verbunden.

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

Sie helfen Klassen, Operationen, Anwendungsfälle, ... zu verstehen.



Activity-Diagramme und die UML



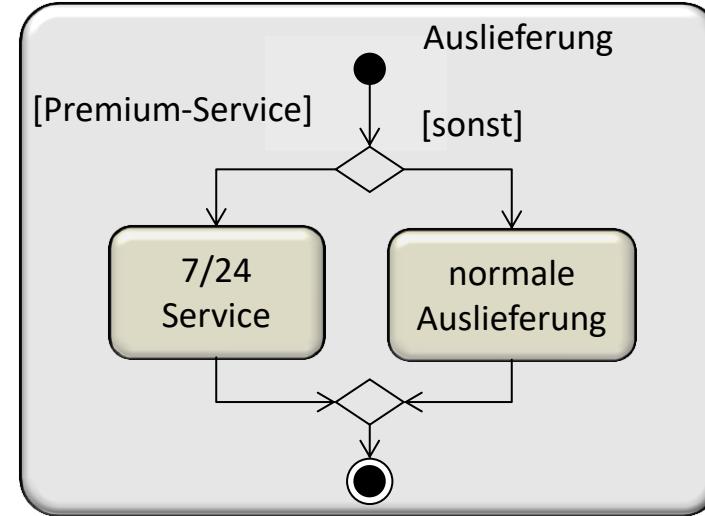
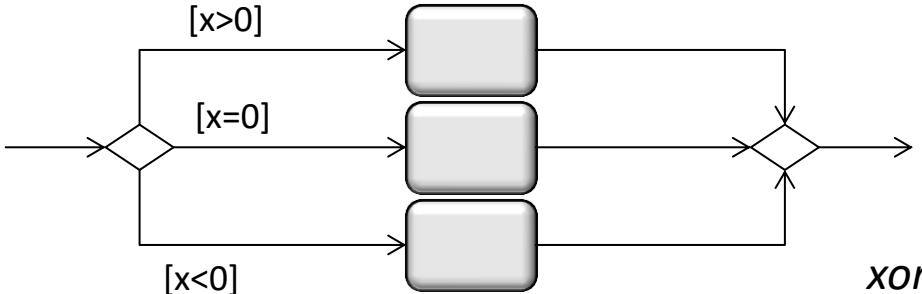


Activity-Diagramme und die UML

- Geschachtelte Aktionen



- Entscheidung und Merge

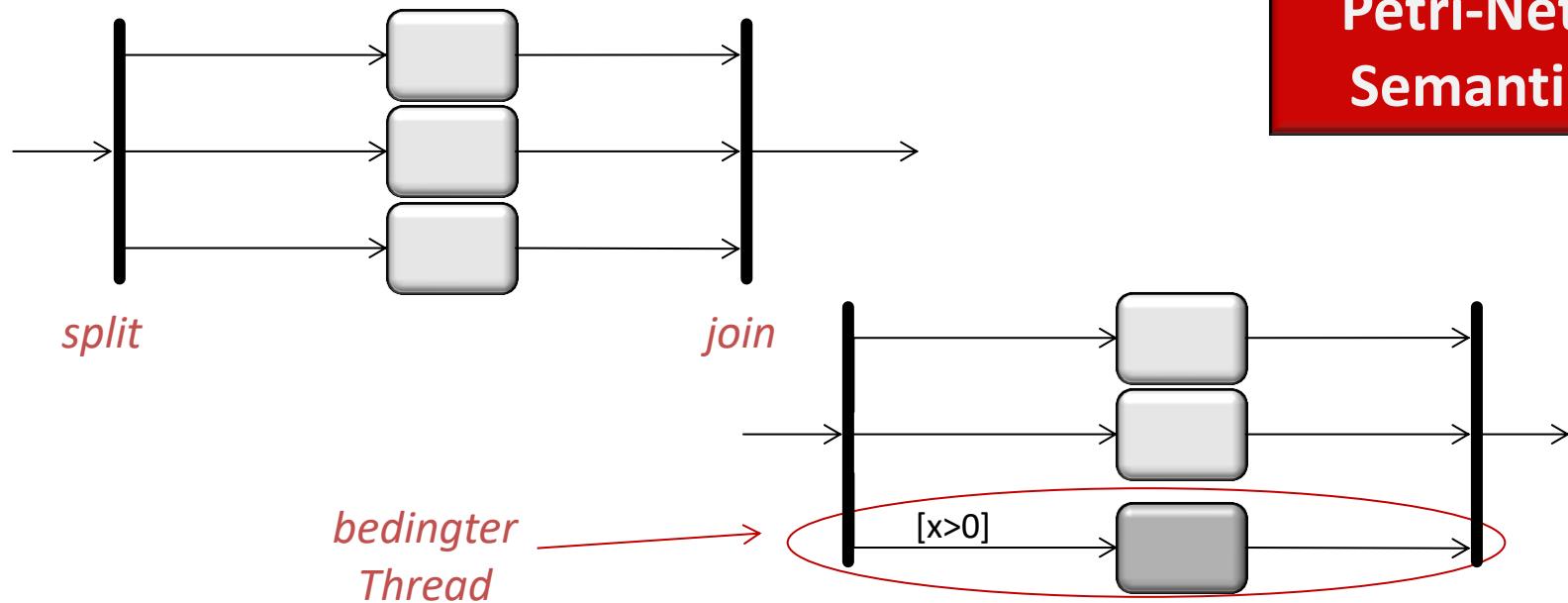


Petri-Netz
Semantik



Activity-Diagramme und die UML

- Split und Join



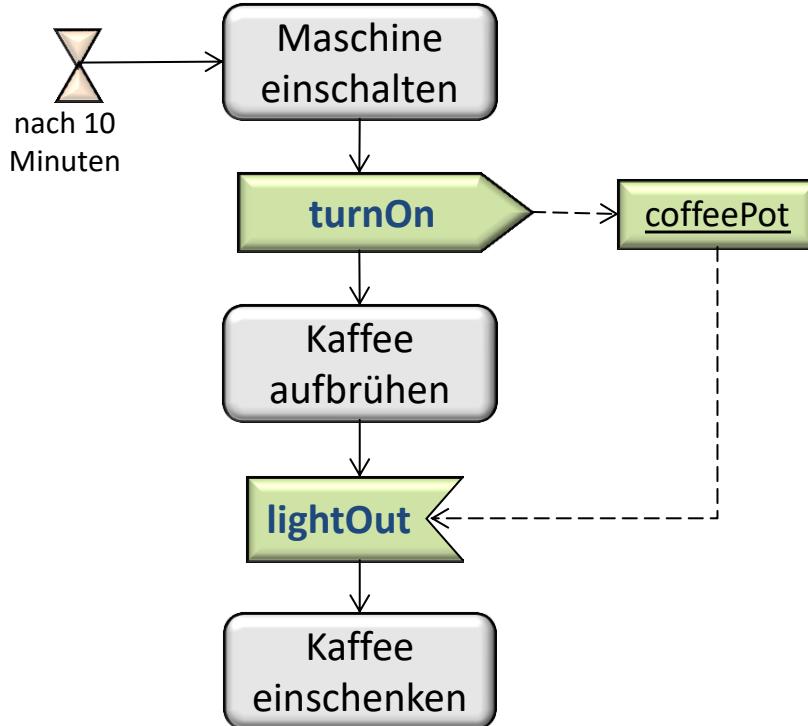
Petri-Netz
Semantik



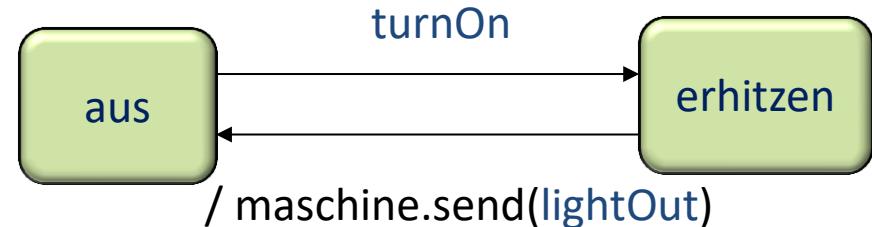
Activity-Diagramme und die UML

- Signale senden und empfangen

Timer



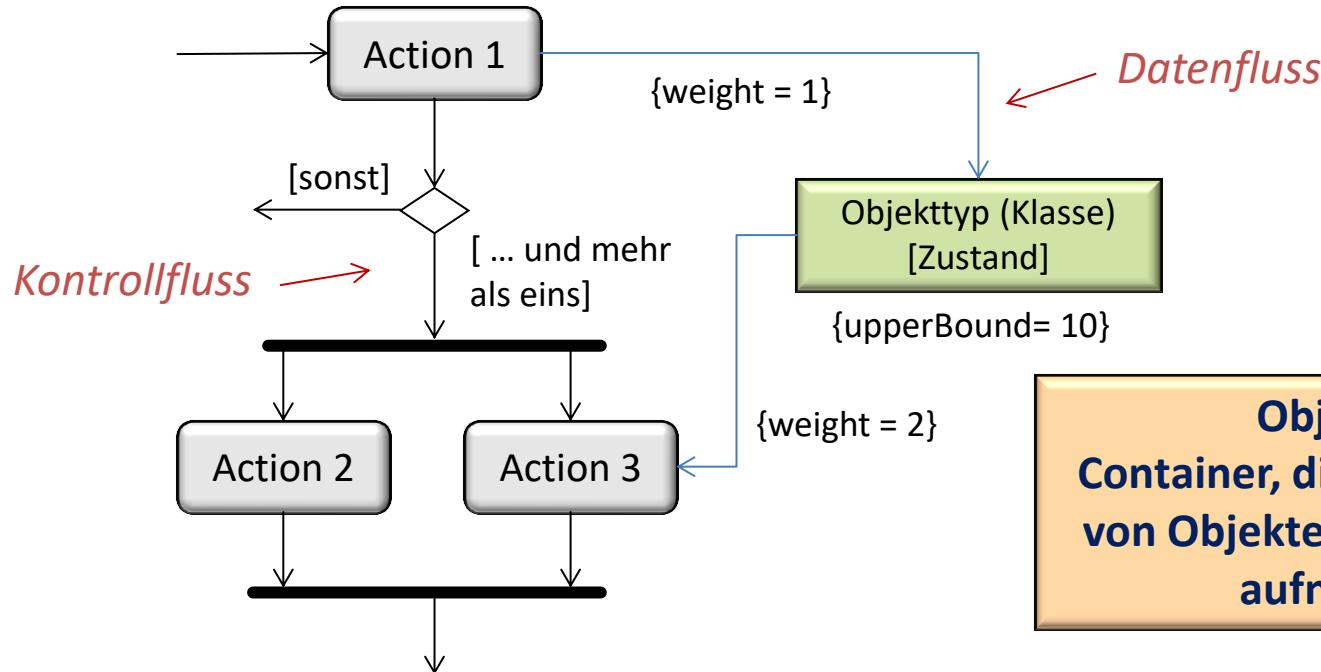
zugehöriges
Zustandsübergangsdiagramm
des „coffeePot“





Activity-Diagramme und die UML

- Objektknoten dienen der Modellierung des Datenflusses

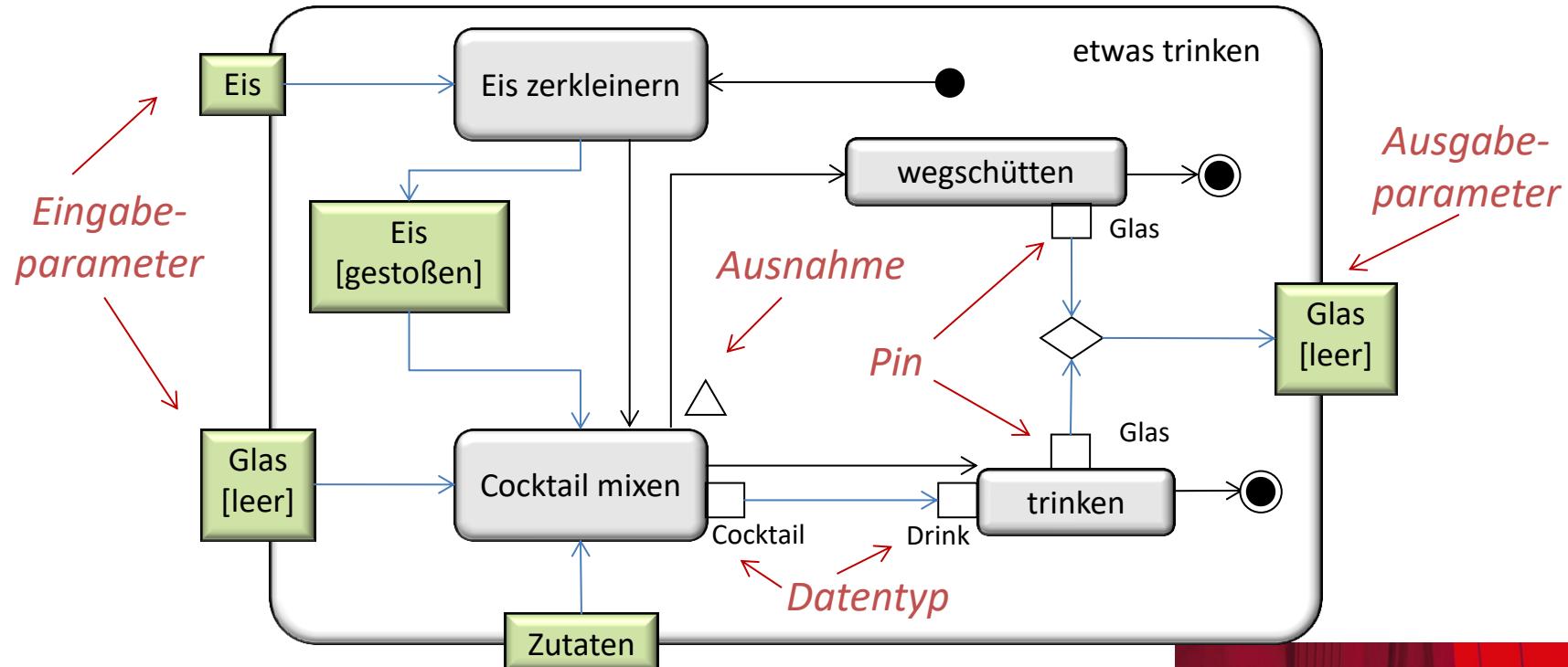


Objektknoten sind Container, die eine begrenzte Anzahl von Objekten eines definierten Typs aufnehmen können.



Activity-Diagramme und die UML

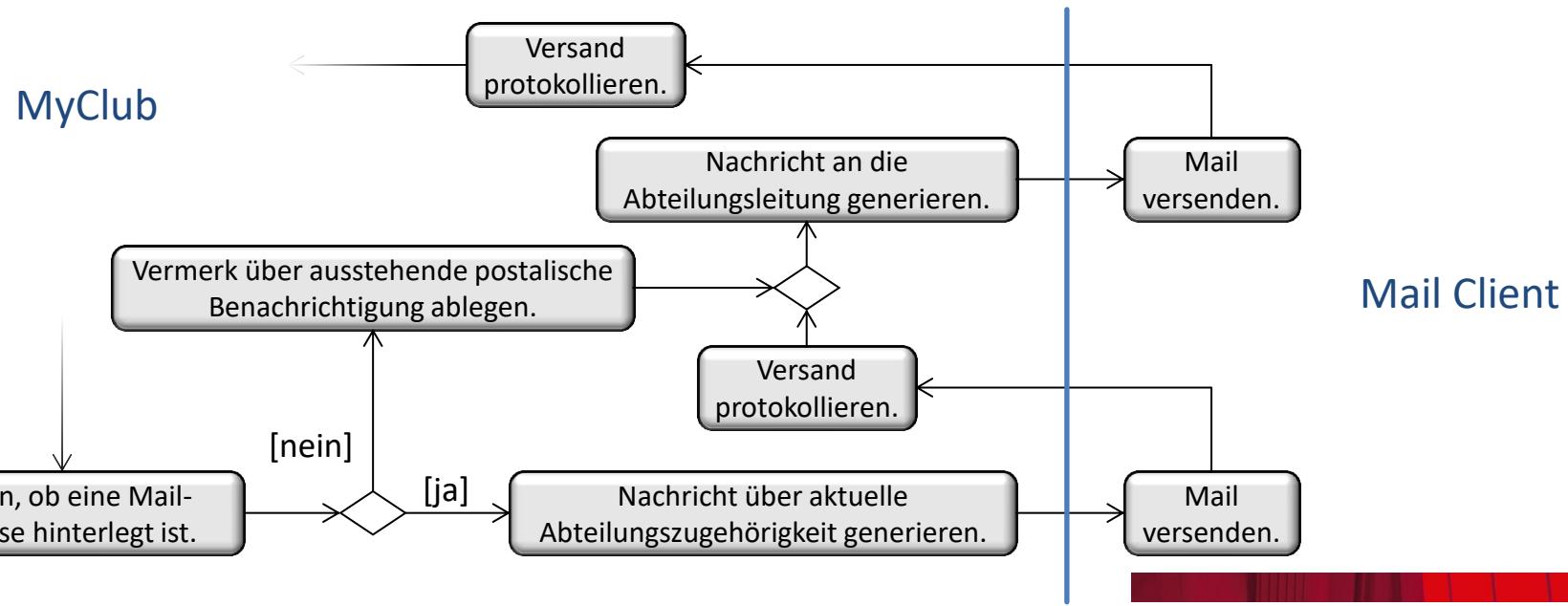
- Parameter und Pin: fortgeschrittene Modellierung des Datenflusses





Activity-Diagramme und die UML

- **Swimlanes:** Aktivitäten können in Verantwortlichkeitsbereiche (Swimlanes) aufgeteilt werden. Diese sind typischerweise organisatorische Einheiten in Businessmodellen oder Akteure und Systeme in Use-Case-Modellen.





Die Analyse abschließen (Schritt 2)

**Erkenntnisse
gewinnen!**

- Objekte identifizieren und deren Beziehungen aufzeigen.
- Systemoperationen bzw. Interaktionen identifizieren und formal definieren.

Als Grundlage dienen die Use Cases und ihre Beschreibungen.

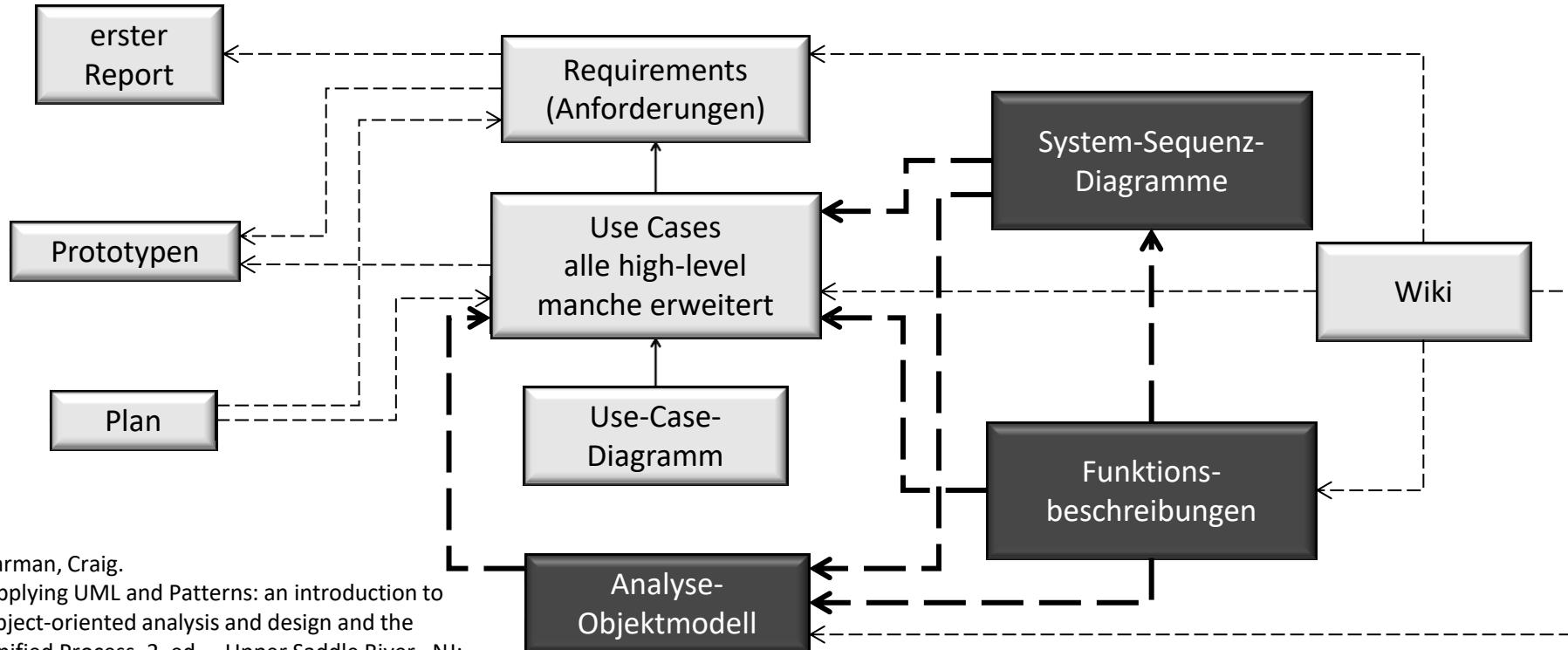
Ziel:

Die Problemdomäne verstehen!

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



Die Analyse abschließen



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 Analyse-Objektmodell

Gesucht sind die Strukturen und Zusammenhänge in der Problemdomäne!

Strategien:

1. **Via Checkliste:** Objekte (Konzepte) finden, die Instanzen vorgegebener Kategorien sind.
Typische Kategorien sind: Dinge, Rollen, Prozesse, Ereignisse, Organisationen, ...

2. **Substantive suchen:** Substantive in den Use-Case-Beschreibungen und den Requirements lokalisieren und als Kandidaten für Konzepte und Entitäten der Problemdomäne betrachten.

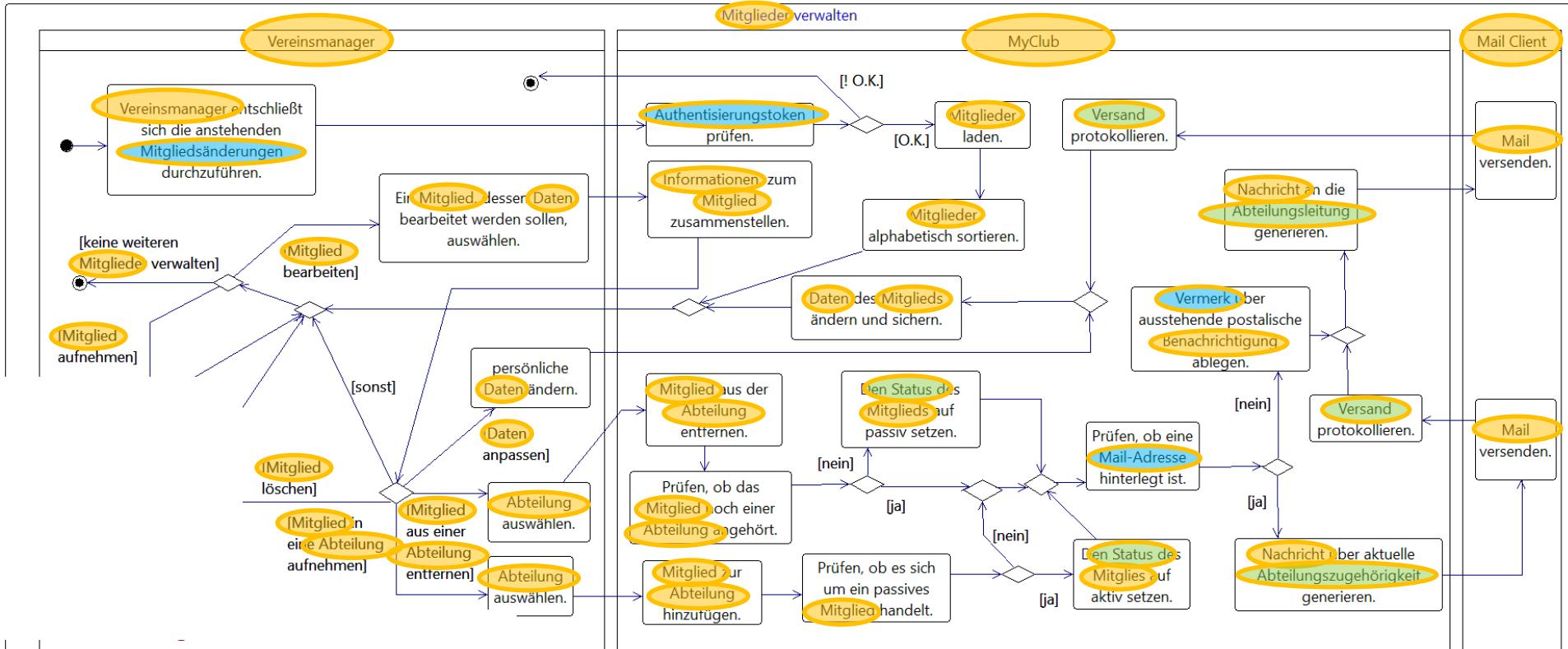
Achtung:

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!)

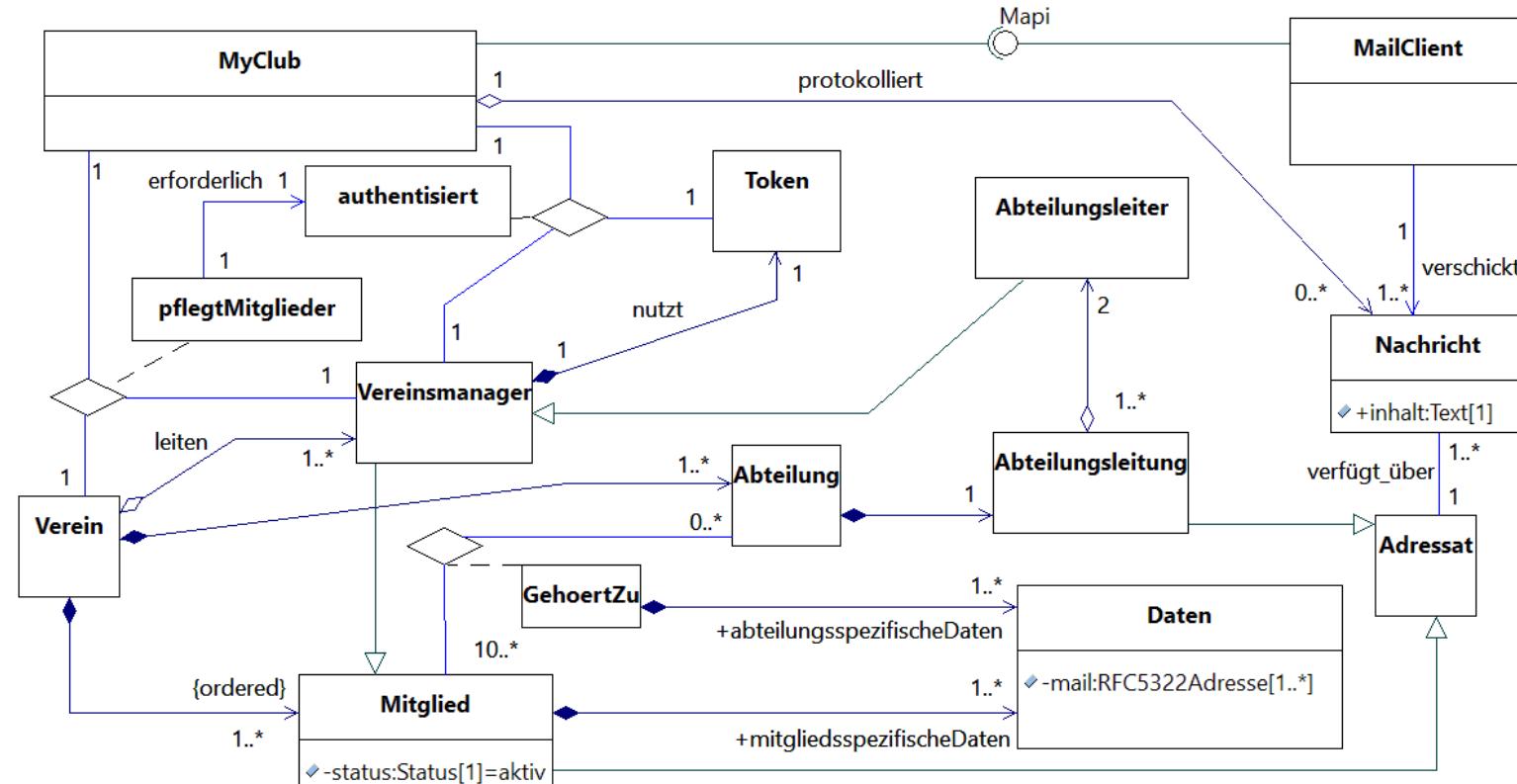


Objekte finden im Use Case „Mitglieder verwalten“





Objekte finden im Use Case „Mitglieder verwalten“



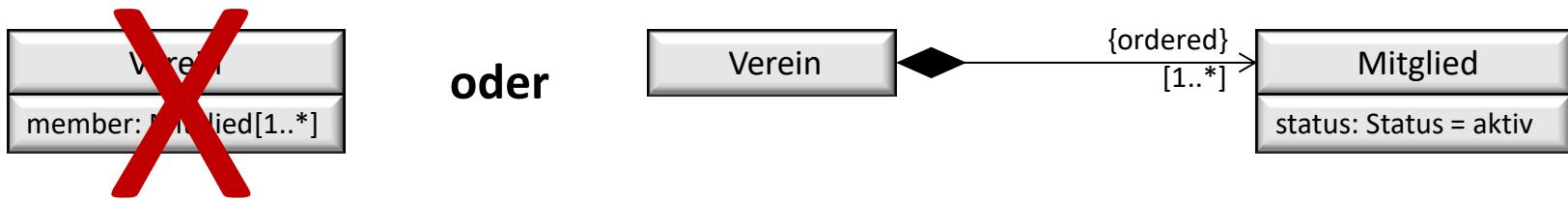


Typischer Fehler: Attribut statt Konzept

Faustregel:

Dinge, die man anfassen kann, die keine Quantitäten, Texte oder **Adjektive** (z.B.: Farben) darstellen, sind keine Attribute, sondern eigenständige Konzepte bzw. Entitäten des Problemraums.

Beispiel:



Falls man sich unsicher ist, dann wählt man immer die Alternative Objekt (Konzept) und nicht Attribut.



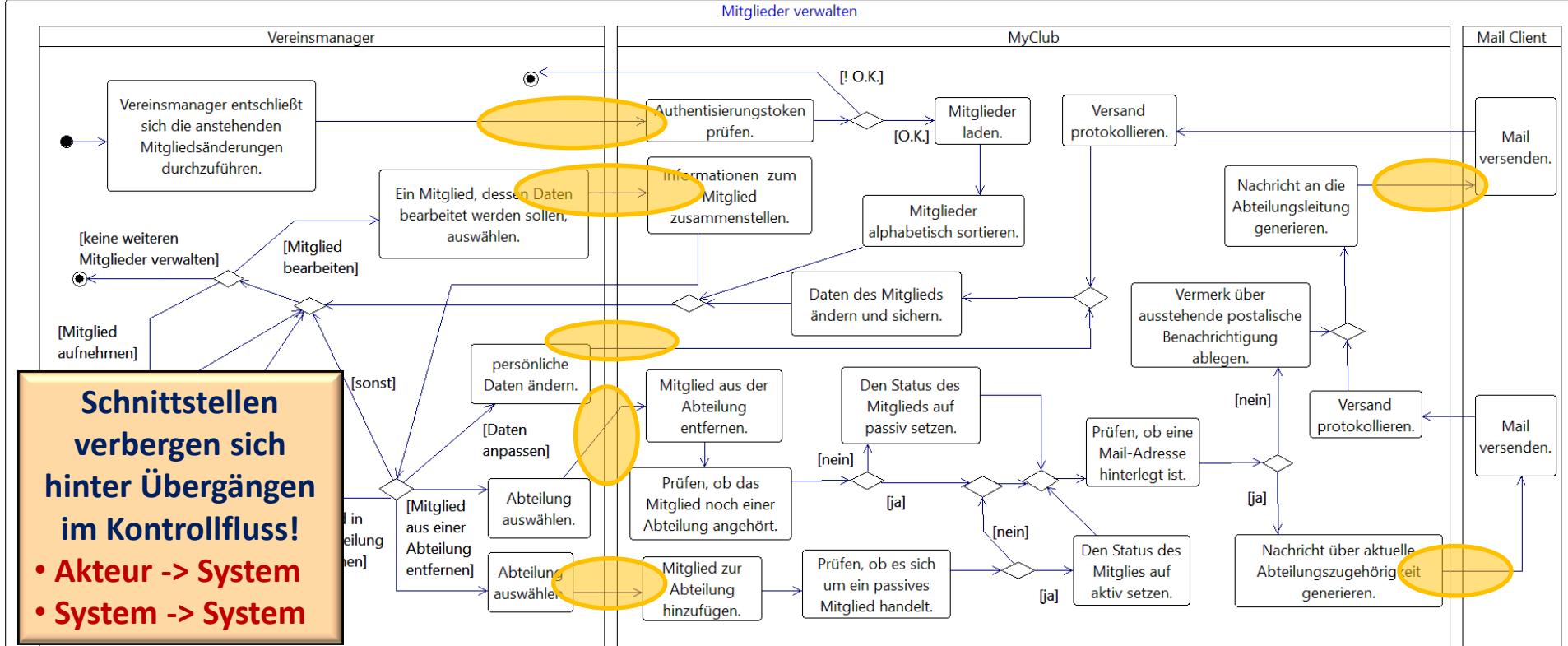
Das dynamische Modell ergänzen

- System-Sequenz-Diagramme erstellen
- Schlüsseloperationen definieren
- Schlüsseloperationen Objekten zuordnen
- (Für zentrale Objekte Zustandsübergangsdiagramme anfertigen)

Ziel:
Identifizierung der erforderlichen Schnittstellen für die Durchführung eines Use Cases.

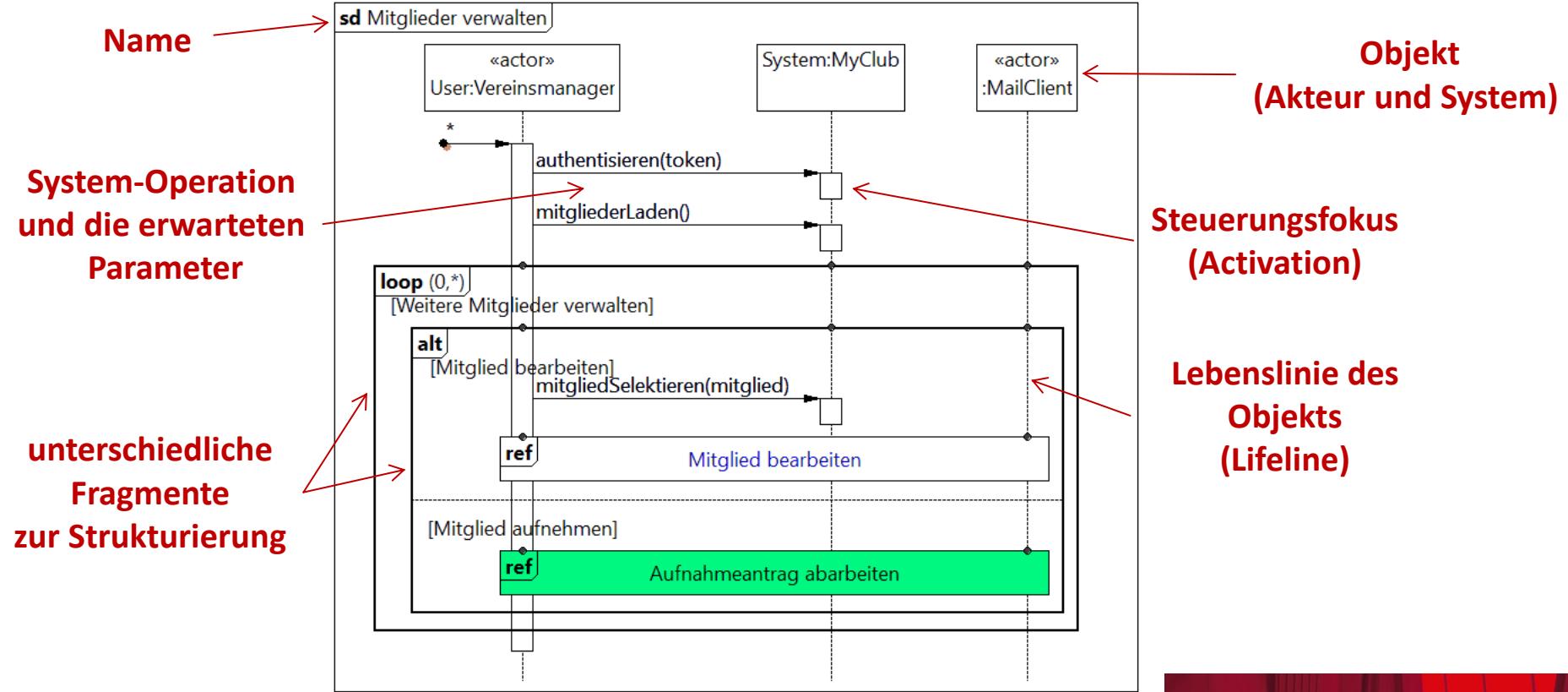


Schnittstellen identifizieren



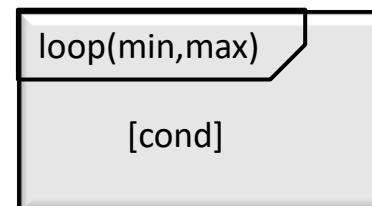
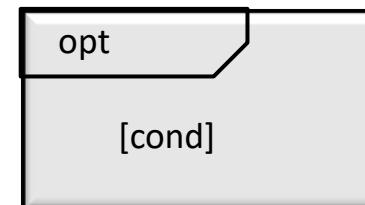
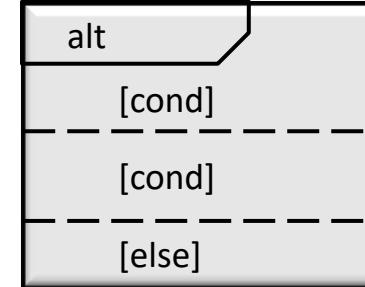
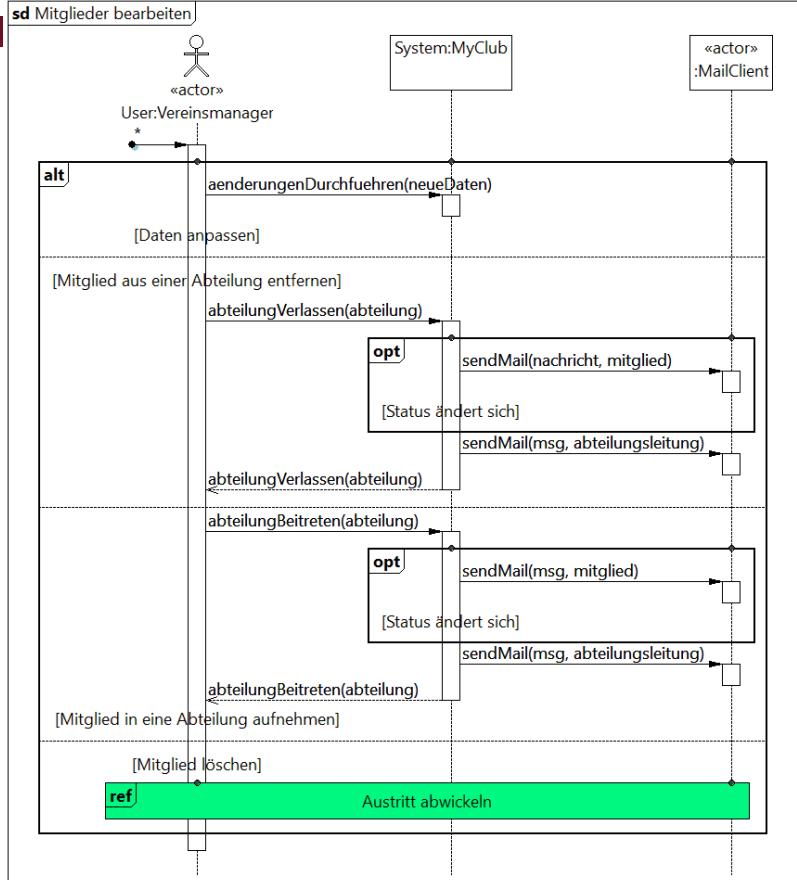


System-Sequenz-Diagramme





Typische Fragmente



- Zur Darstellung alternativer Abläufe

- Zur Darstellung optionaler Abläufe

- Zur Darstellung iterativer Abläufe

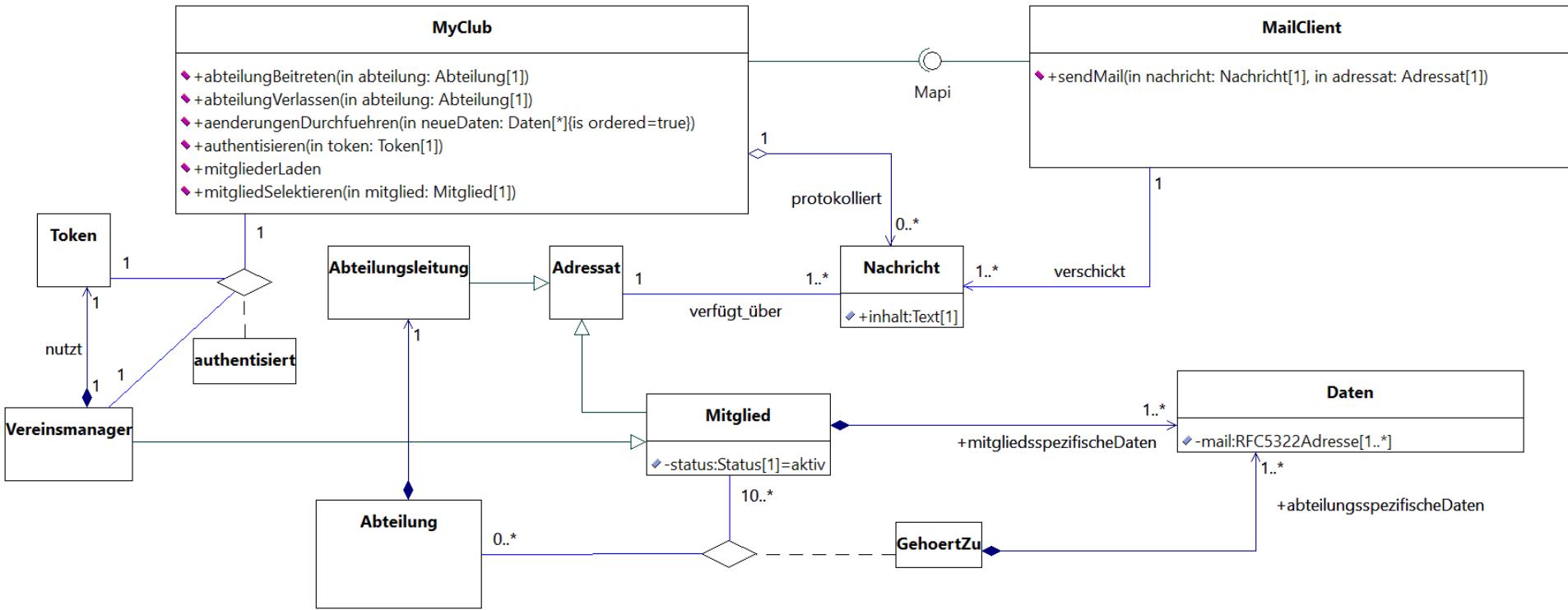


Operationen beschreiben

| | |
|---------------------|--|
| Name: | abteilungVerlassen(abteilung: Abteilung) |
| Verantwortlichkeit: | Entfernt ein zuvor ausgewähltes Mitglied aus der angegebenen Abteilung und informiert die Abteilungsleitung per Mail. Gehört das Mitglied danach keiner Abteilung mehr an, wird sein Status auf „passiv“ gesetzt. Liegt eine Statusänderung vor, wird das Mitglied elektronisch angeschrieben, bzw. ein Benachrichtigungsvermerk hinterlegt. |
| Referenzen: | Use Case: Mitglieder verwalten Funktionen: F1.5, ..., A1.1, ... |
| Bemerkungen: | Was geschieht mit Abteilungen, die nach einem Austritt über weniger als 10 Mitglieder verfügen? Wie geht das System mit einem hinterlegten Benachrichtigungsvermerk um? |
| Ausnahmen: | Falls das Mitglied nicht der Abteilung angehört, ist ein Fehler zu melden. |
| Vorbedingungen: | Ein Mitglied ist ausgewählt und der Vereinsmanager hat sich korrekt authentisiert. |
| Nachbedingungen: | Das Mitglied wurde aus der Abteilung entfernt. Die verschickten Mails wurden protokolliert, ggf. wurde ein Benachrichtigungsvermerk hinterlegt. |



Operationen ergänzen





Zusammenfassung

- **Die frühen Iterationen**

- Nachdem die Requirements erfasst wurden, werden die Systemgrenzen und die Akteure identifiziert.
- Alle zentralen Use Cases werden erfasst und priorisiert.
- Use-Case-Diagramme werden erstellt und Beziehungen modelliert.
- **Kritischsten und essentiellsten Use Cases werden zuerst ausgearbeitet. Sie definieren das Problem.**

- **In späteren Iterationen**

- Die Use Cases bearbeiten, die gemäß der Planung für diesen Entwicklungsschritt vorgesehen sind.
- **Ggf. werden neue Anforderungen aufgenommen und in die Entwicklung integriert.**



Requirement
Engineering

Man folgt der Priorisierung und stimmt das Vorgehen mit dem Auftraggeber ab.



Zusammenfassung

Die Analyse beantwortet
folgende Fragen:

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



Zusammenfassung

oder kurz:

Welche Komponenten und Schnittstellen müssen realisiert werden?

- 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?



Software-Engineering Design

(angelehnt an Craig Larman „Applying UML and Patterns“)

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



1. Der Übergang zwischen Analyse und Design

- Zu realisierende Use Cases festlegen

abhängig davon:

- Bedienkonzept und
- Mockups erstellen



Nur die Use Cases kommen in Betracht,
die bereits analysiert wurden!
(aufgearbeitet und verstanden)

- Die Architektur entwerfen
(Komponenten, Konnektoren, Module, ...)

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



Architekturentscheidungen

- 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
- ...

Ziel ist es, die
übergreifende Struktur
des Systems
festzulegen!



Was ist Architektur (3+1 View Model)

Architektur bedeutet Organisation

Clements, P. et. al., editors.
Documenting Software Architectures:
Views and Beyond (2nd Edition).
Addison-Wesley, 2010

Organisation des
Gesamtsystems

Component and
Connector View

Organisation
des Betriebs

Use
Cases

Module
View

Allocation
View

Organisation der
Implementierung

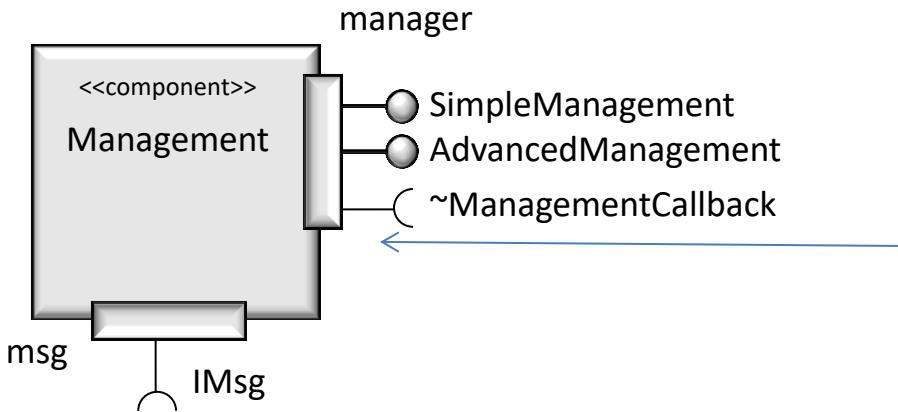
Organisation der
Funktionalität



Organisation des Gesamtsystems

- **Komponentendiagramme**

- Aufgaben werden gelöst indem sie in kleine, beherrschbare Teile zerlegt werden.
- Jede Komponente übernimmt eine dedizierte und überschaubare Teilaufgabe.
- Die Zusammenarbeit der Komponenten löst das ursprüngliche Problem.



Port:

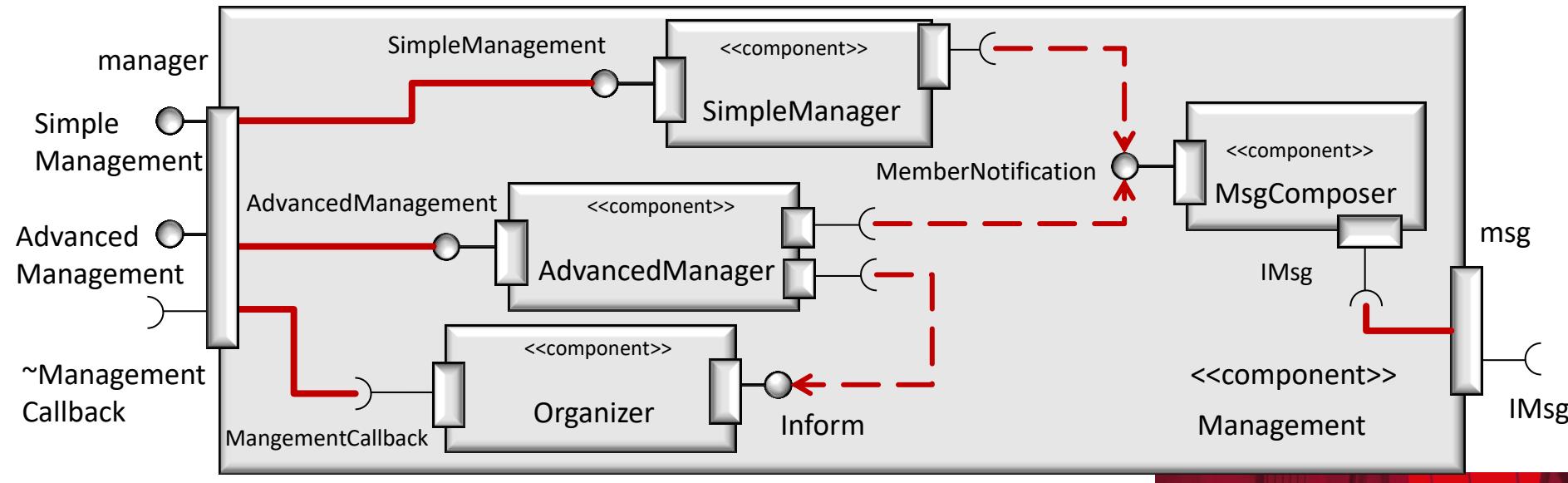
Ein Port ist ein dedizierter Interaktionspunkt mit der Umgebung. Er ist beschrieben durch die Menge der zur Verfügung gestellten und benötigten Interfaces.



Konnektoren verbinden Komponenten

- **Konnektoren**

- Assembly Connector: verbinden Komponenten über Schnittstellen.
- Delegation Connector: verbinden externe Schnittstellen mit den (Sub-) Komponenten die sie realisieren oder benötigen.





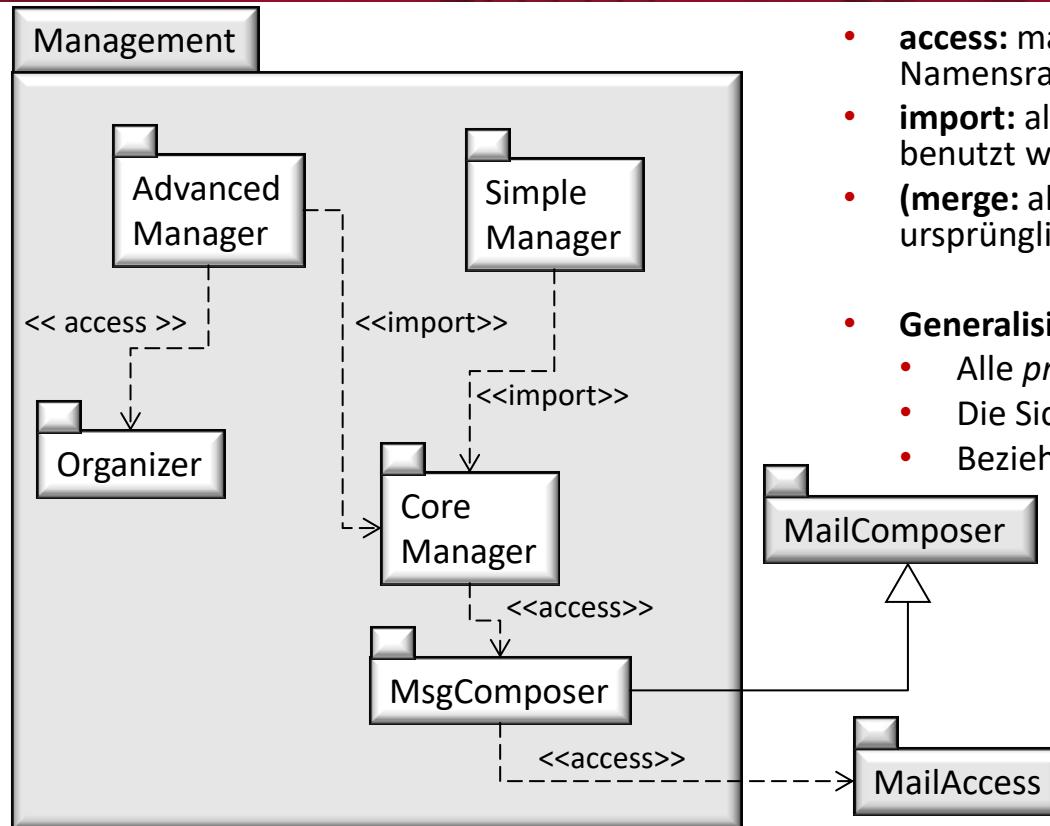
Organisation der Implementierung

Um Klassen (die Strukturen der Implementierung) zu strukturieren verwendet man Packages.

- Packages bilden eine echte Baumstruktur. Kein Element (Klasse, ...) gehört zu mehreren Packages.
- Packages erzeugen Übersichtlichkeit und verdeutlichen Abhängigkeiten.
- Packages sollten **nie** zyklisch voneinander abhängen.



Organisation der Implementierung

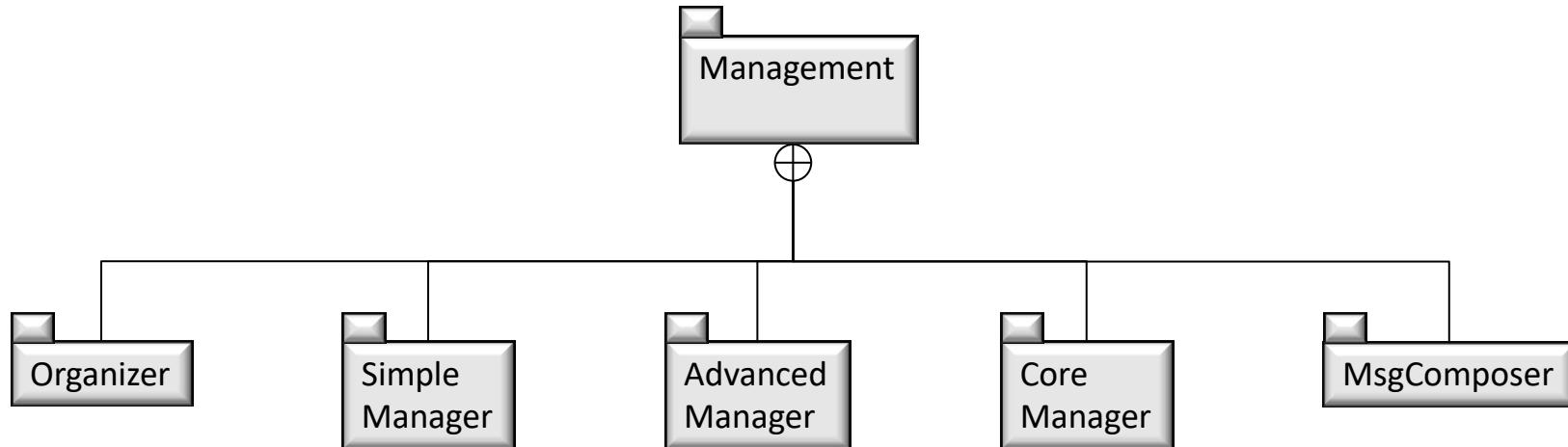


- **access:** manche *public* Elemente werden benutzt, der Namensraum bleibt jedoch getrennt.
- **import:** alle *public* Elemente werden importiert, d.h. können benutzt werden.
- **(merge:** alle Elemente werden als Generalisierung der ursprünglichen neu angelegt.)
- **Generalisierung:**
 - Alle *protected* 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.



Alternative Darstellung



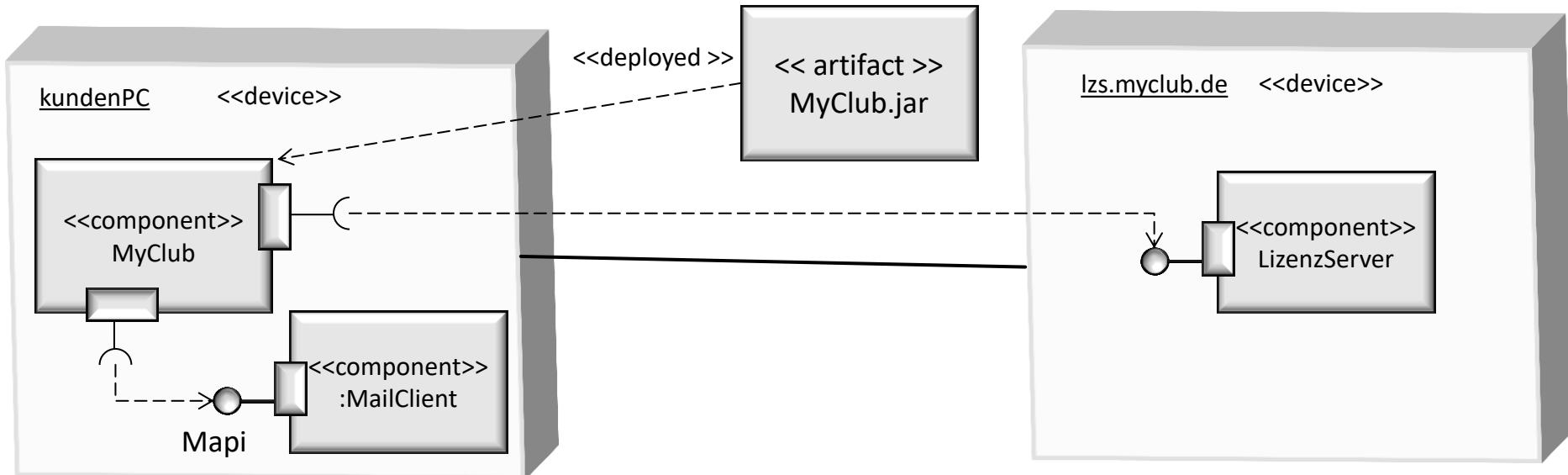
Packages bilden eine echte
Baumstruktur!



Organisation des Betriebs

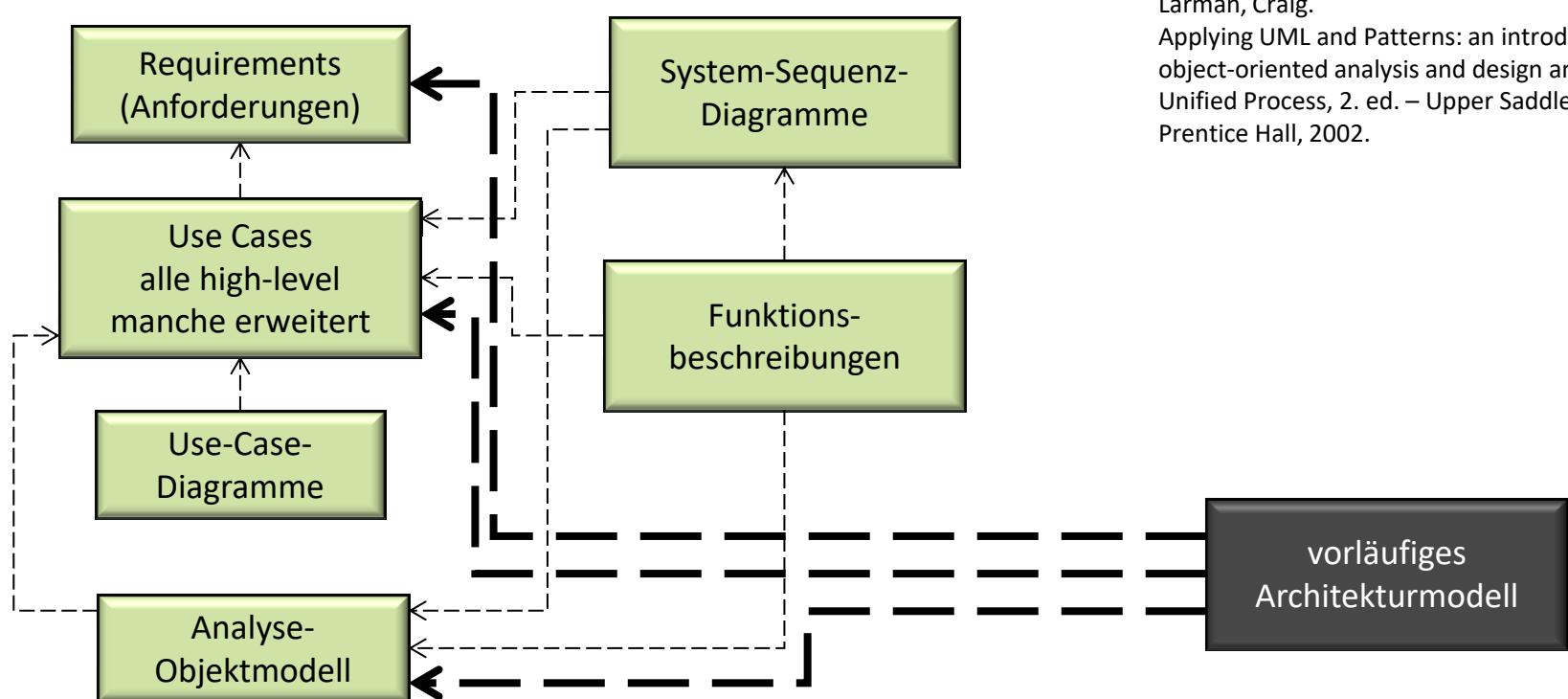
Verteilungsdiagramme (Deployment Diagrams)

zeigen die Laufzeitkonfiguration, die Verteilung der Komponenten auf die physisch vorhandenen Knoten (Tiers, Rechner, Prozessoren, ...) und deren Kommunikationsbeziehungen.





Die ersten Designschritte



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.

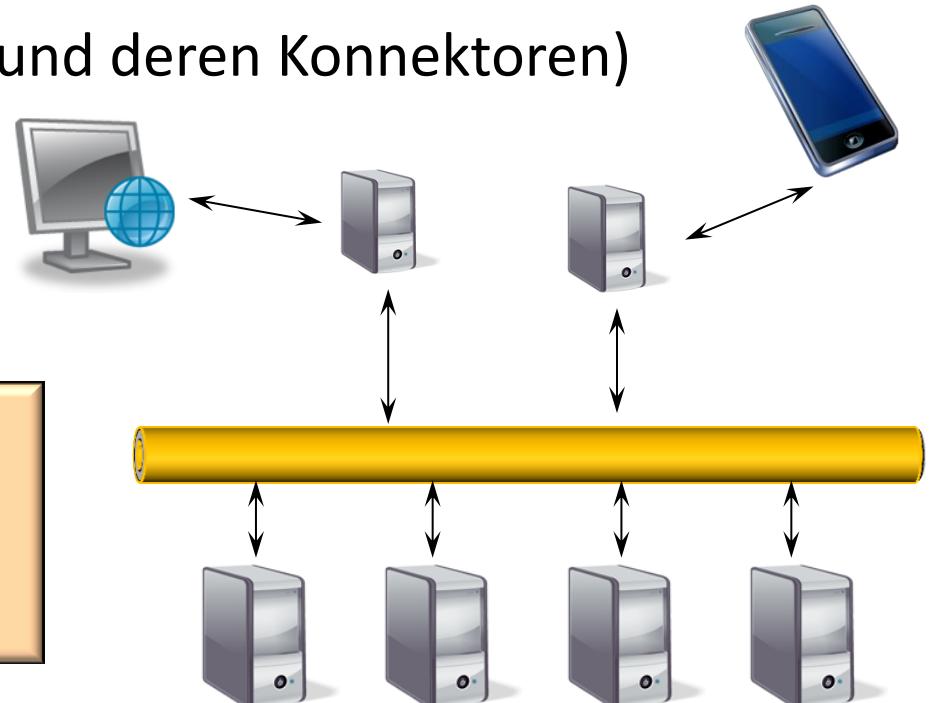


Die ersten Designschritte

- Die primäre Architektur entwerfen
(übergeordnete Komponenten und deren Konnektoren)



Tiers, Schichten und die zentralen
Komponenten entwerfen,
strukturieren, verteilen und
zuordnen.

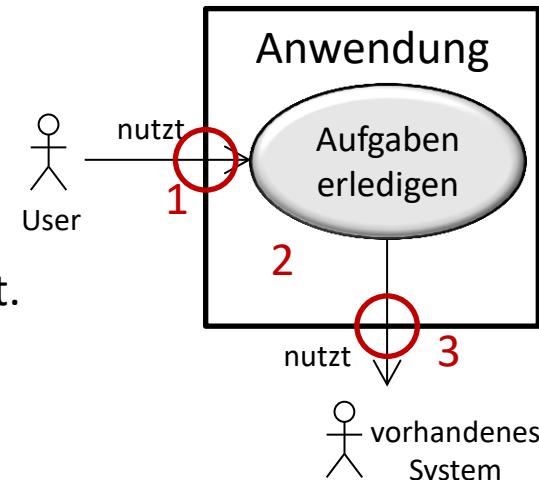




Das Drei-Schichtenmodell

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

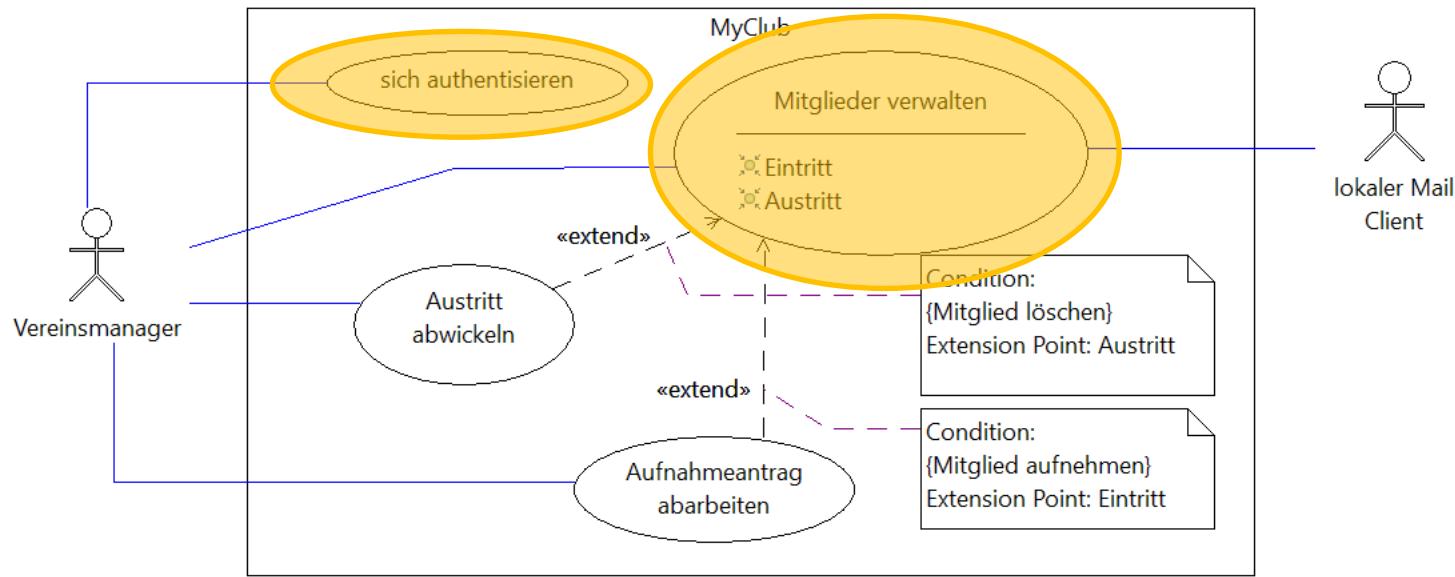




MyClub: Use Cases festlegen

Use Cases für die erste Iteration:

- „Manage Members“ (zentral für die Anwendung)
- „Authenticate User“ (komplex und es fehlt „uns“ an Erfahrung)

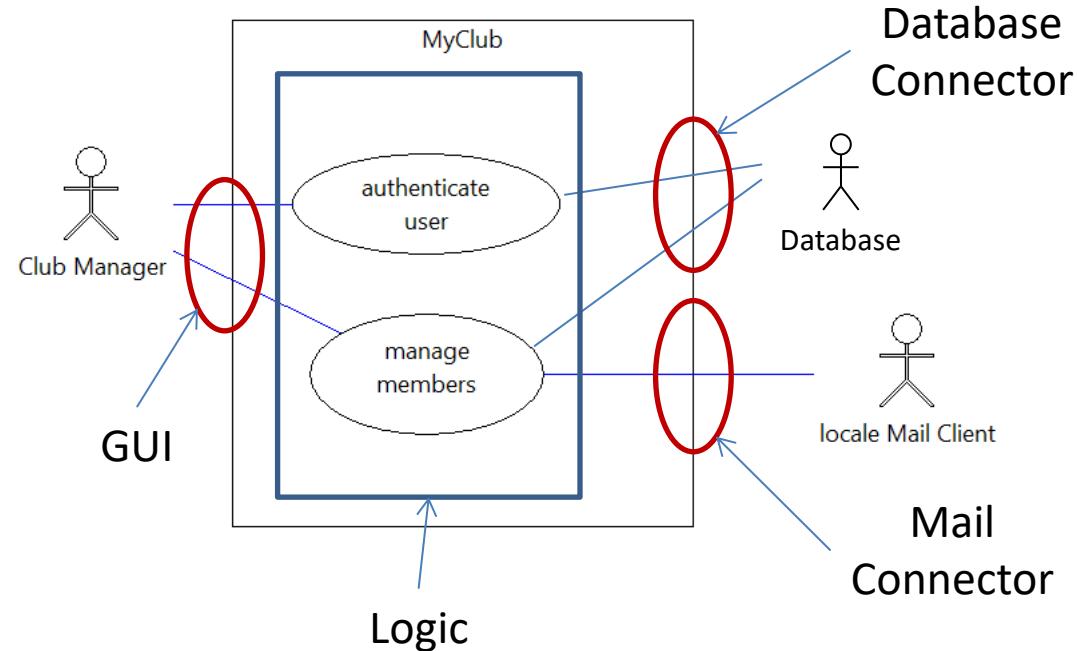




MyClub: Vom Use Case zur Architektur

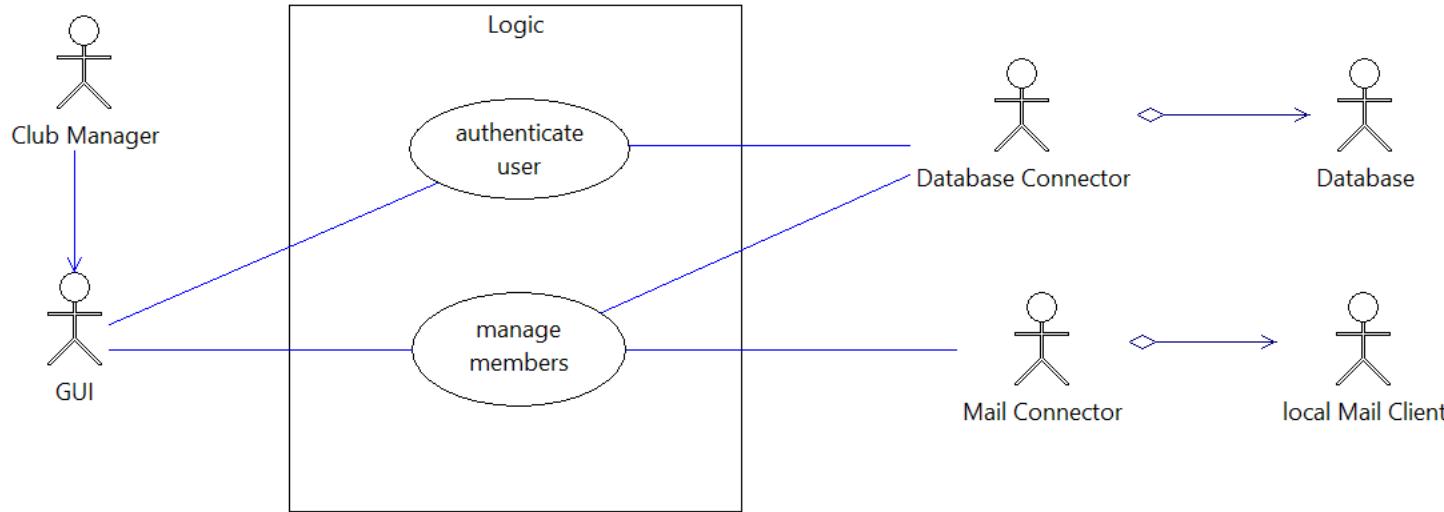
Zur Speicherung der Daten verwenden wir eine lokale Datenbank!

- Präsentationsschicht
 - GUI
- Datenzugriffs- und Persistenzschicht
 - Database Connector
 - Mail Connector
- Applikationsschicht
 - Logic





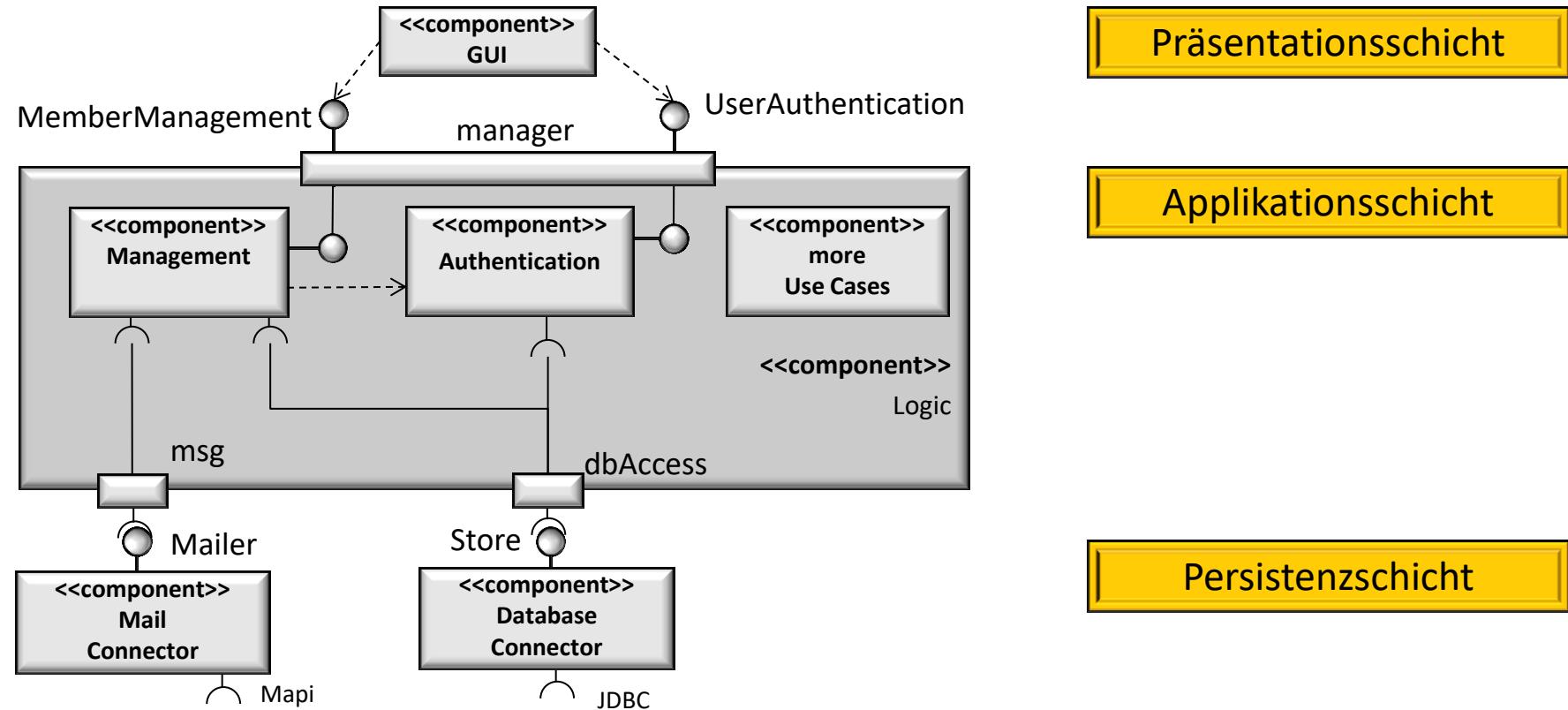
MyClub: Vom Use Case zur Architektur



Für jeden Use Case und jeden Akteur eine Komponente!



MyClub: Die Architektur





Architektur: Schichten organisieren

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.

Wird in einer Schicht ein Objekt genutzt, dann muss die dazu gehörende Klasse in dieser Schicht definiert sein, oder das Objekt stammt aus einer tieferen Schicht.



Beispiel

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

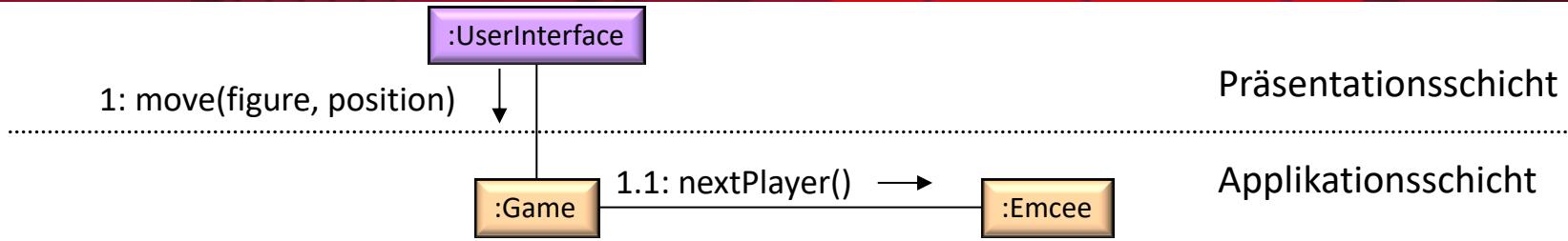
Welche Schicht überwacht die Spielregeln und trifft die entsprechenden Entscheidungen?

- die Präsentationsschicht? ✗
- die Applikationsschicht? ✓
- die Persistenzschicht? ✗

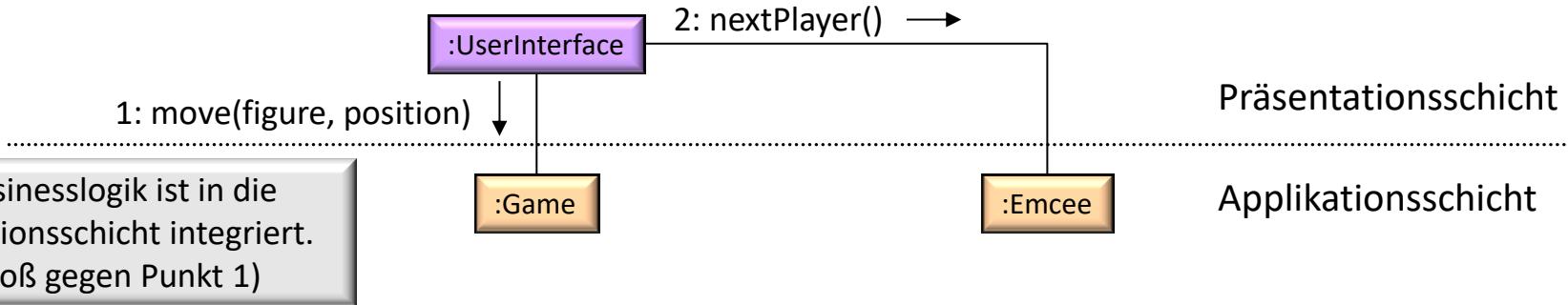


Prinzipien der Benutzerschnittstelle

Variante 1



Variante 2

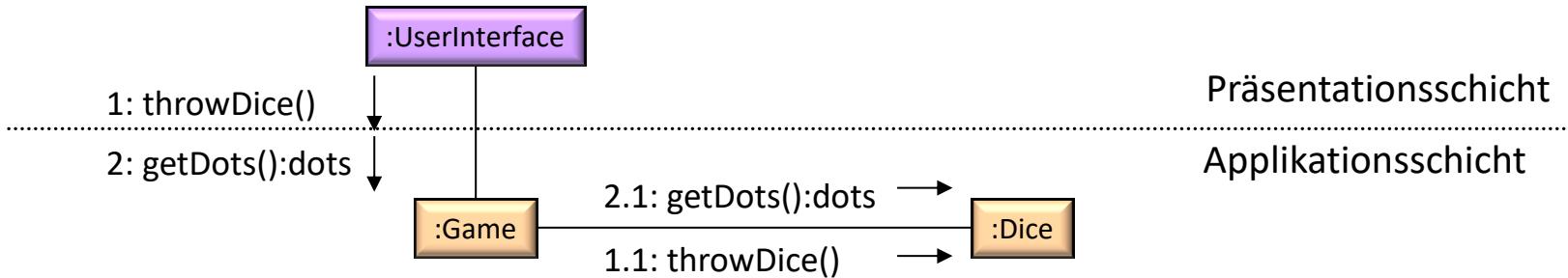


**Die Präsentationsschicht ist nicht für die Bearbeitung der Systemoperation verantwortlich,
sondern lediglich für das Anstoßen der Systemoperationen.**

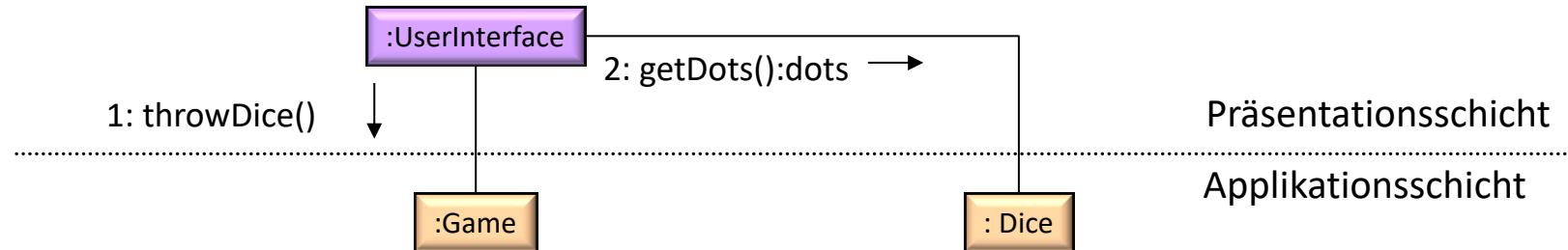


Visualisierung eines Resultats

Variante 1



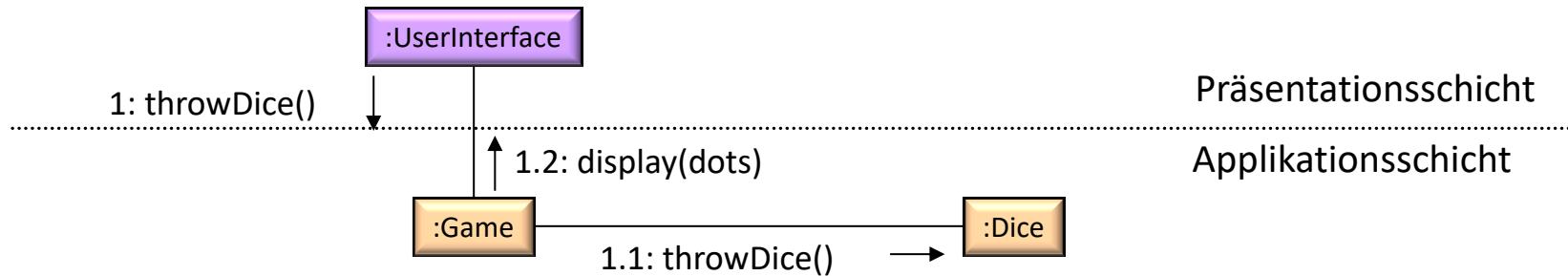
Variante 2



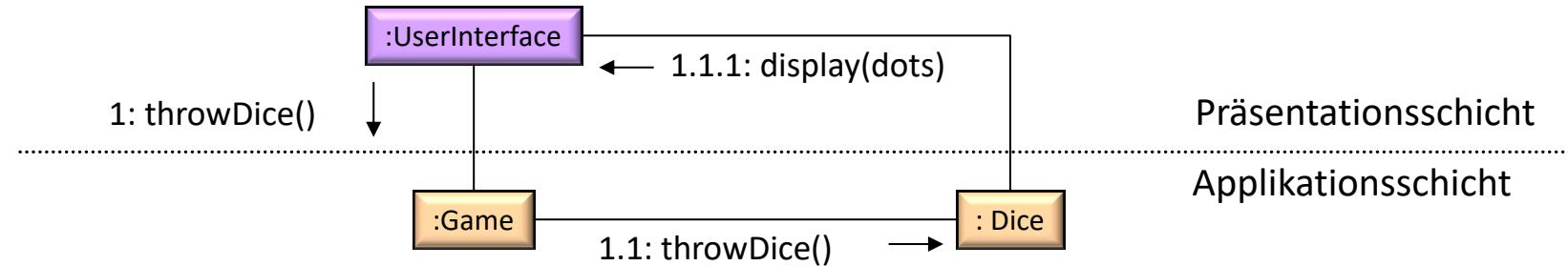


Visualisierung eines Resultats

Variante 3



Variante 4





Visualisierung eines Resultats

Die Varianten 1 und 2 sind Polling-Lösungen.

**Polling ist in der Regel ineffizient und lastintensiv und damit
ungeeignet zur Darstellung von spontanen Änderungen.**

- **Etwa bei der Überwachung von Systemgrößen, oder**
- **bei der Animation von Abläufen.**

Die Varianten 3 und 4 sind Push-from-below-Lösungen.

Diese sind von der Idee her besser.

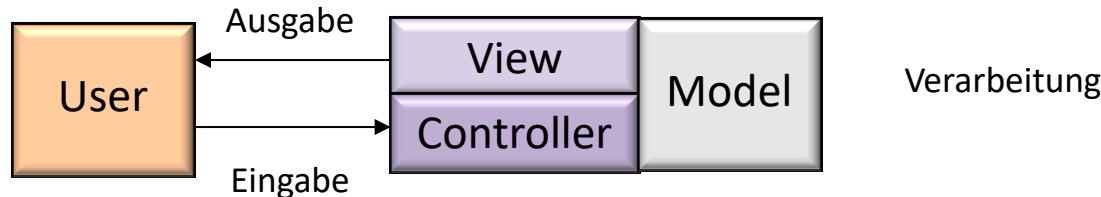
Es muss jedoch für eine Entkopplung der Ebenen gesorgt werden, denn sonst liegt ein Verstoß von Punkt 2 vor und eine tiefere Schicht kennt die höhere Schicht.



Model-View-Controller

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

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





Model-View-Controller

Probleme und Missverständnisse

- Was ist eine View?



- Was ist ein Modell?

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



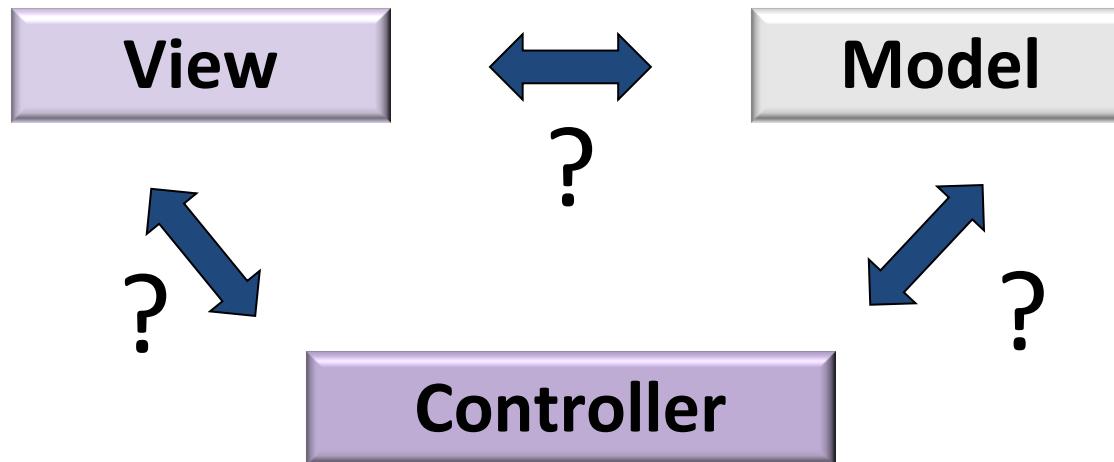
- Was ist ein Controller?





Model-View-Controller

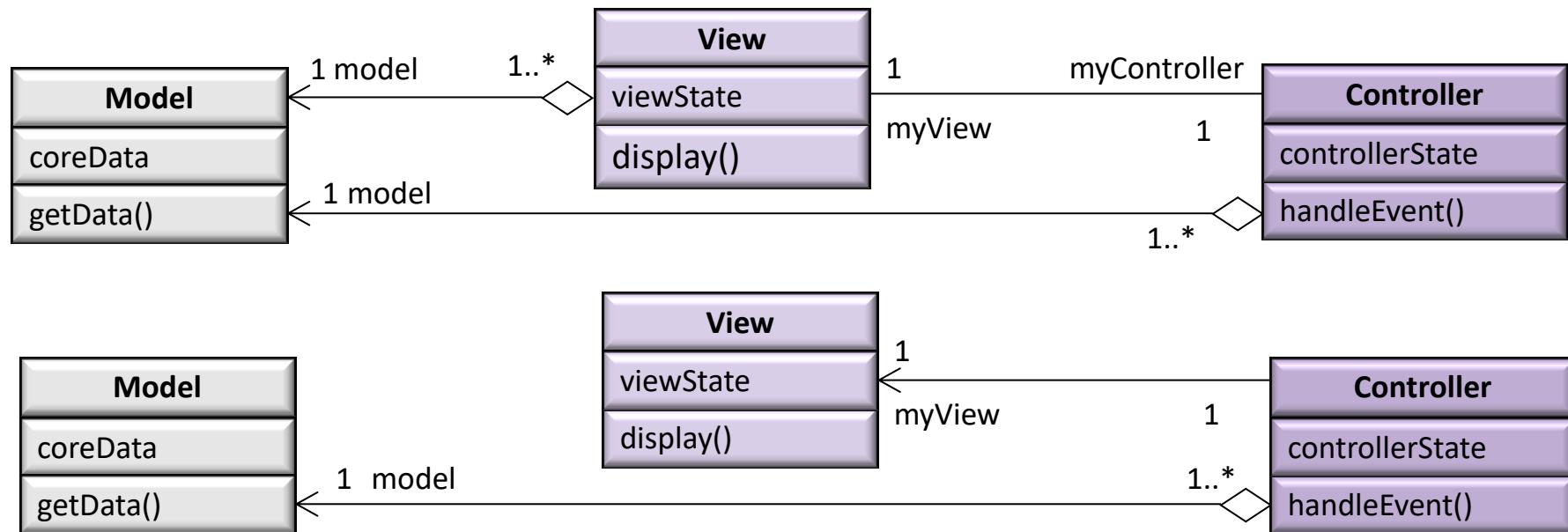
Wer kennt wen?





Model-View-Controller

Wer kennt wen?





Model-View-Controller

Was gehört zu welcher Schicht?

Model

Präsentationsschicht

View

.....

Applikationsschicht

Controller



Model-View-Controller

Schlussfolgerung:

- Ein Controller verfügt über keine Logik im Sinne domänenspezifischen Wissens.
Der Controller übernimmt nicht die Ablaufsteuerung im Use Case.
- Das Modell beinhaltet nicht nur Daten, sondern auch das domänenspezifische Wissen.
Im Modell sind Daten und Logik vereint.



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.

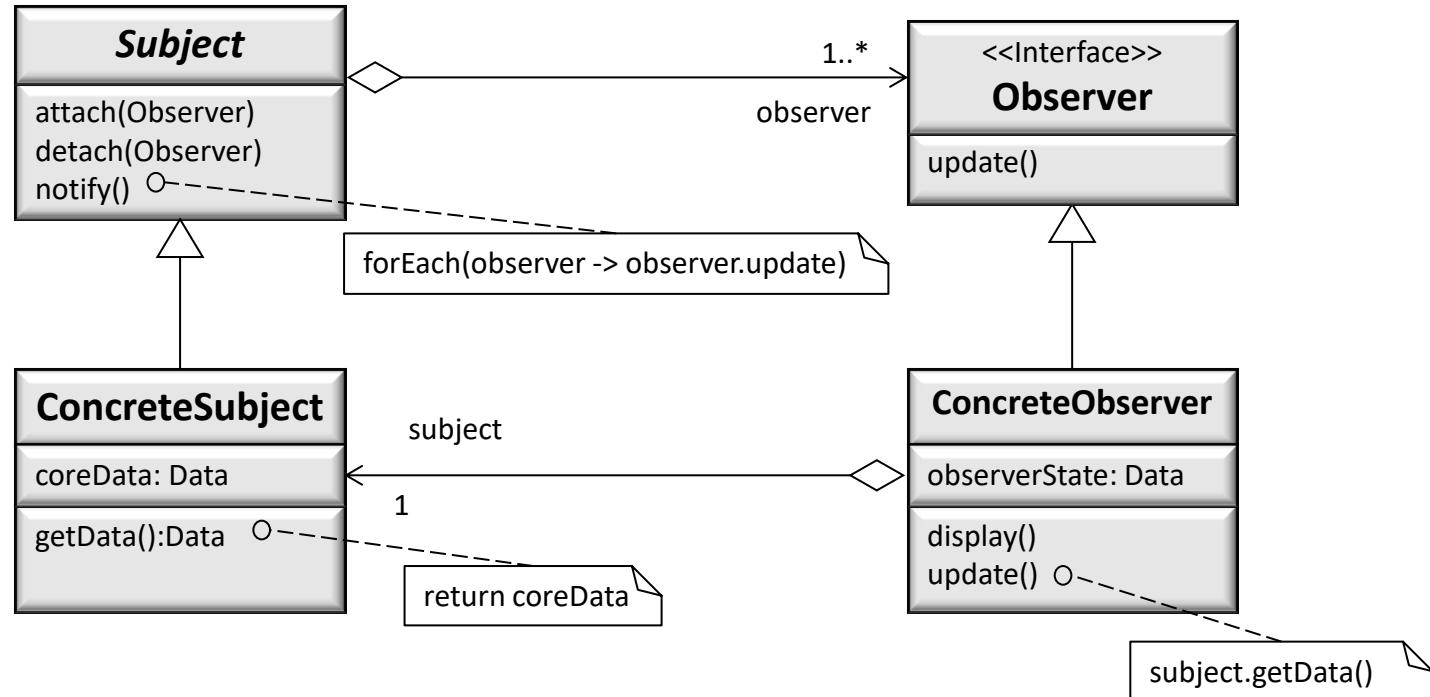
The **Views** render response to the user with the help of the model object.

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>



Zentrale Idee: Indirekte Kommunikation (Observer-Pattern)

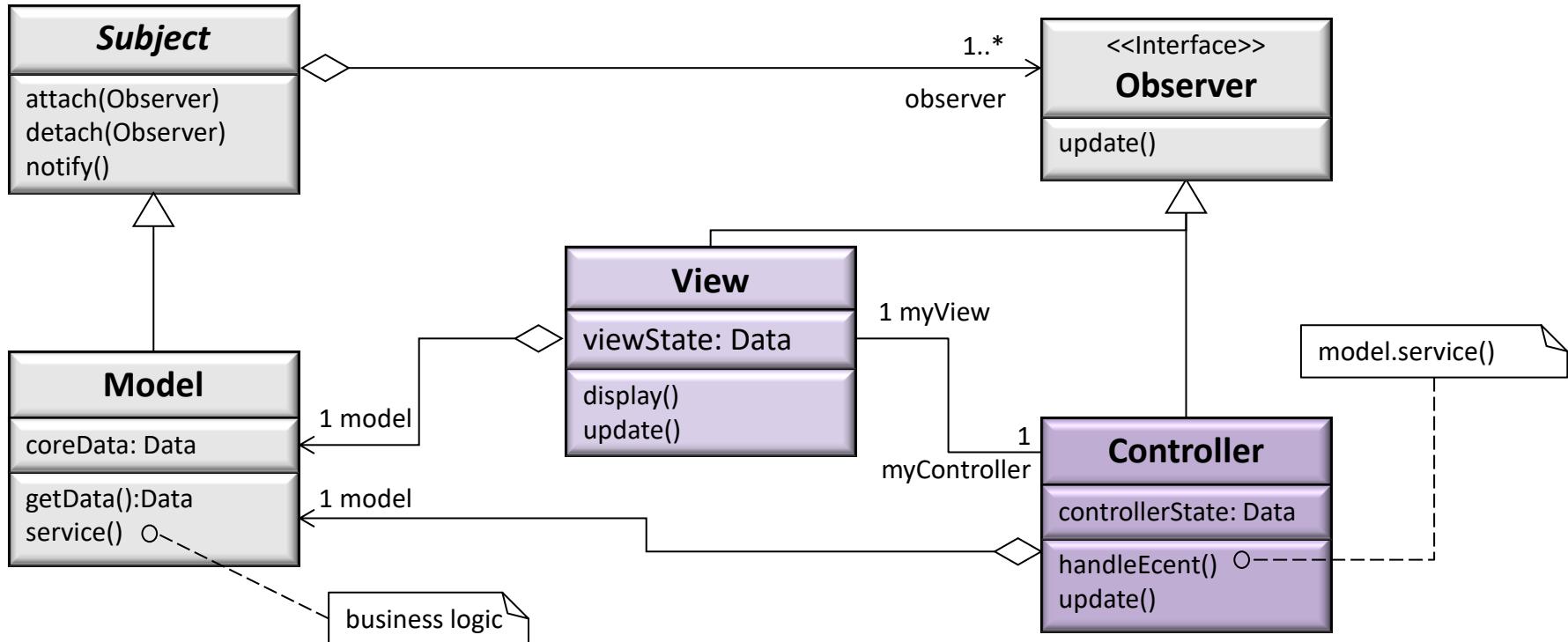


Gamma, Erich et.al.

Design patterns : elements of reusable object-oriented software – Reading, Mass.: Addison-Wesley, 1995.



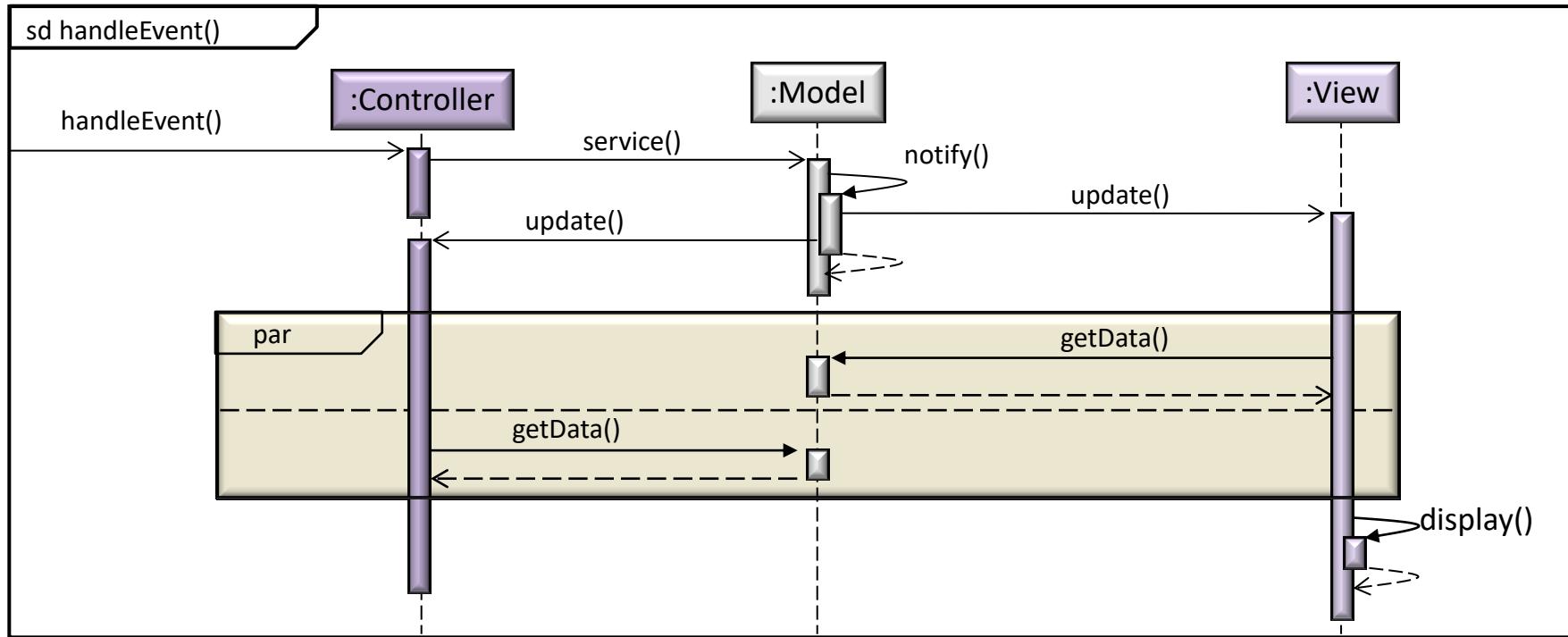
Der Observer in MVC





Dynamische Aspekte im Model-View-Controller-Pattern

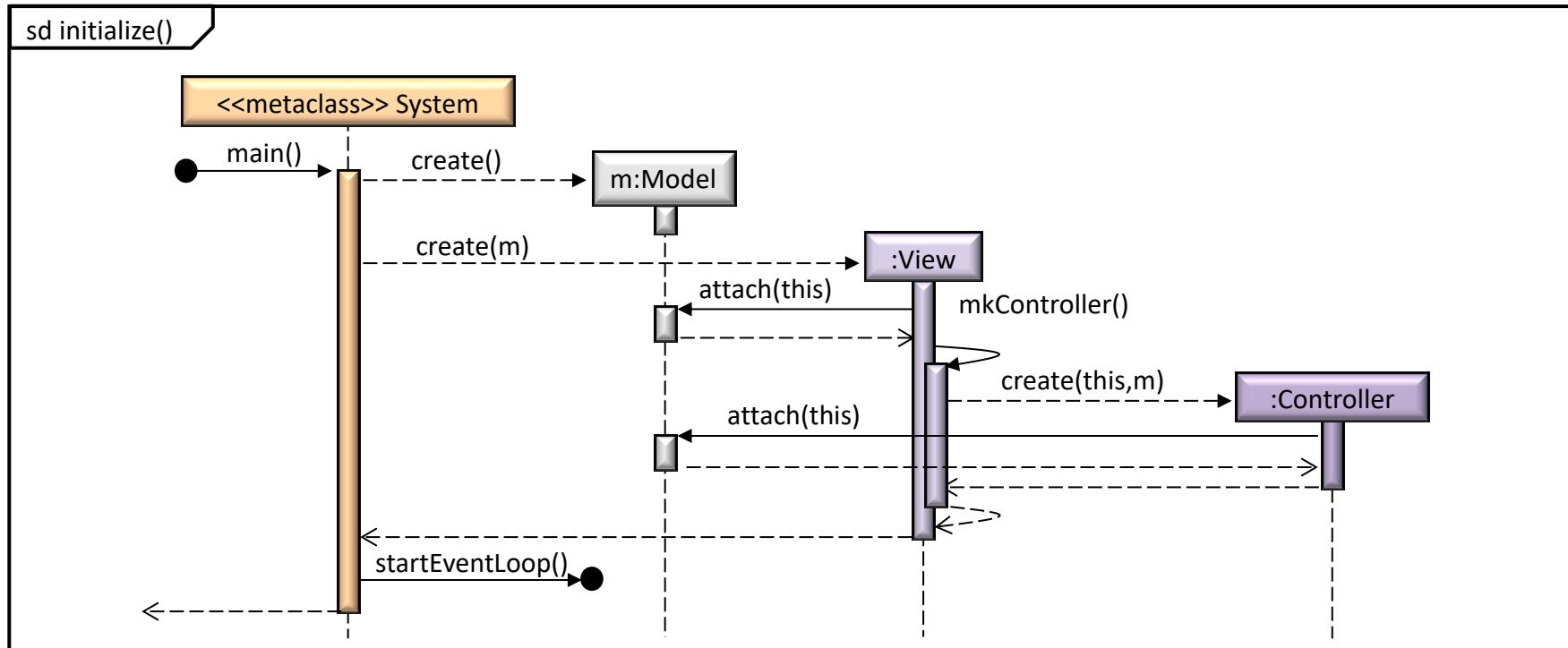
- Benutzereingaben werden propagiert





Dynamische Aspekte im Model-View-Controller-Pattern

- Die Initialisierung des Systems





MVC – 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 erklärt 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 anfordert, die Realisierung der Update-Funktion. Zusätzliche Parameter in der Update-Funktion ermöglichen eine Optimierung.

- **Ist überhaupt eine Aktualisierung notwendig?**
- **Sollte die Aktualisierung verzögert werden, da noch weitere Updates kommen?**



MVC – 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 die entsprechenden System-Funktionen ab (model.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 über Fabrikmethoden zu organisieren.

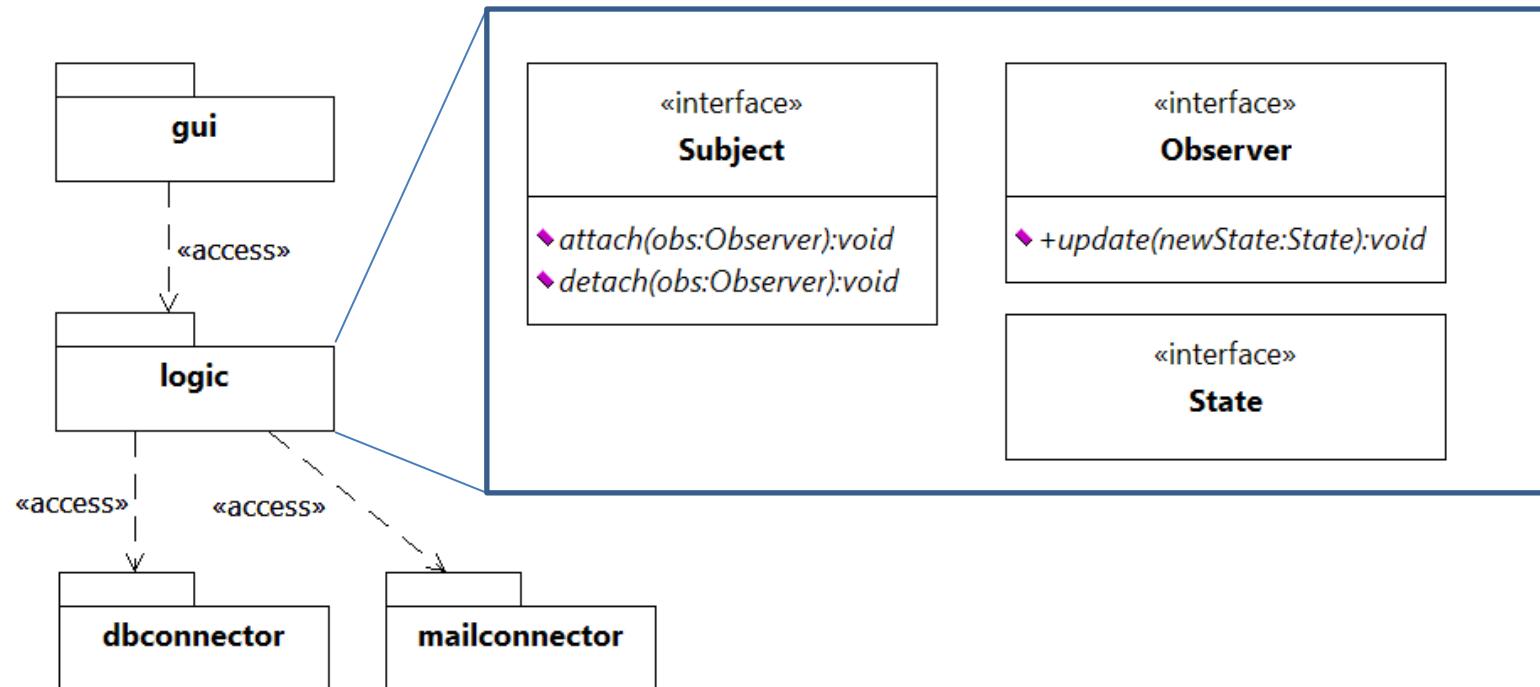
6. Initialisierung der gesamten MVC-Struktur

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.



MyClub: Strukturen der Implementierung

- Die ersten Interfaces gemäß MVC





MyClub: Strukturen der Implementierung

- **Die ersten Interfaces gemäß MVC**

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

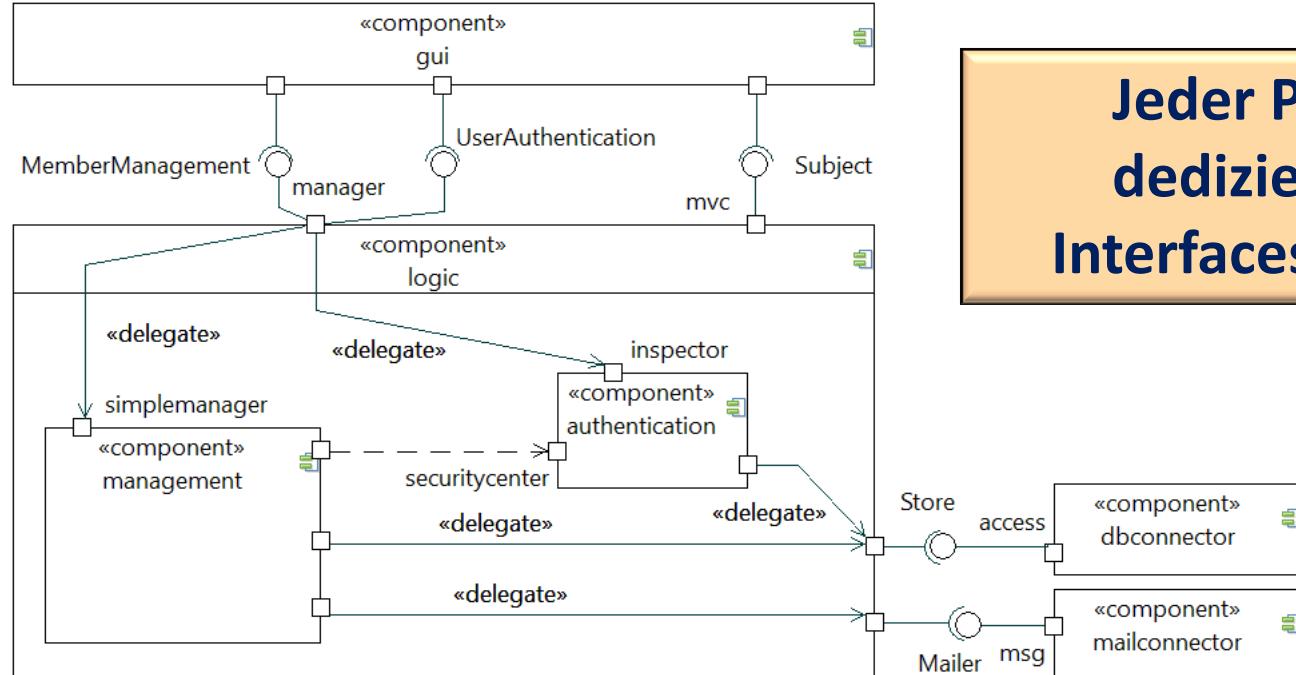
```
public interface Observer {  
  
    void update(State newState);  
}
```

```
public interface State {  
  
}
```



MyClub: Ports als Zugriffspunkte

- Der Zugriff auf das Modell: Ein Port ist ein dedizierter Interaktionspunkt mit der Umgebung, beschrieben durch die zur Verfügung gestellten und benötigten Interfaces.

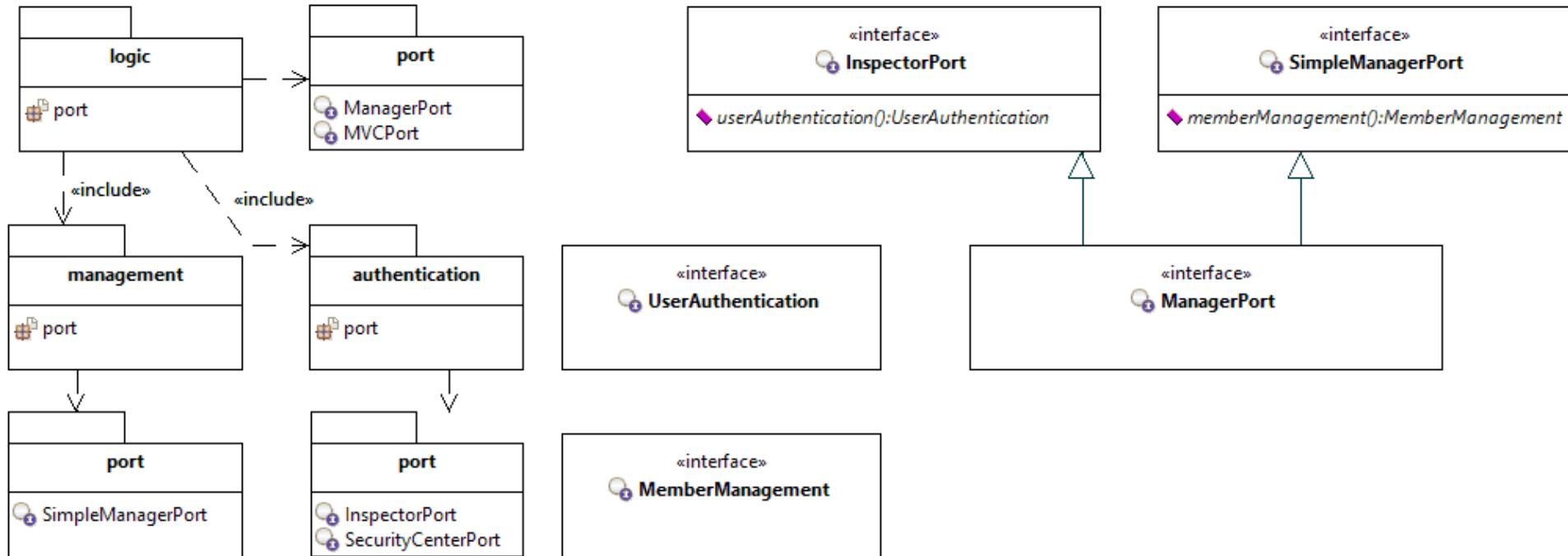


Jeder Port stellt eine dedizierte Menge an Interfaces zur Verfügung!



MyClub: Ports als Zugriffspunkte

- Der Zugriff auf das Modell (der Manager-Port)





MyClub: Ports als Zugriffspunkte

- Der Zugriff auf das Modell (der Manager-Port)

```
import myClub.authentication.port.InspectorPort;  
import myClub.management.port.SimpleManagerPort;  
  
public interface ManagerPort extends SimpleManagerPort, InspectorPort {  
}
```

```
public interface InspectorPort {  
  
    UserAuthentication userAuthentication();  
}
```

```
public interface UserAuthentication {  
  
}
```

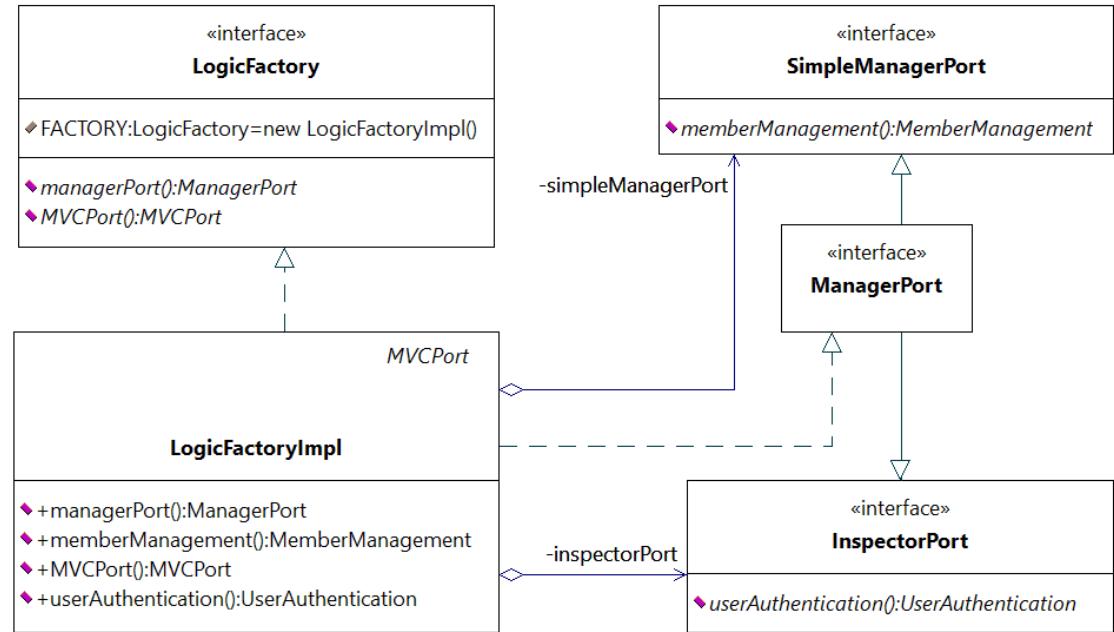
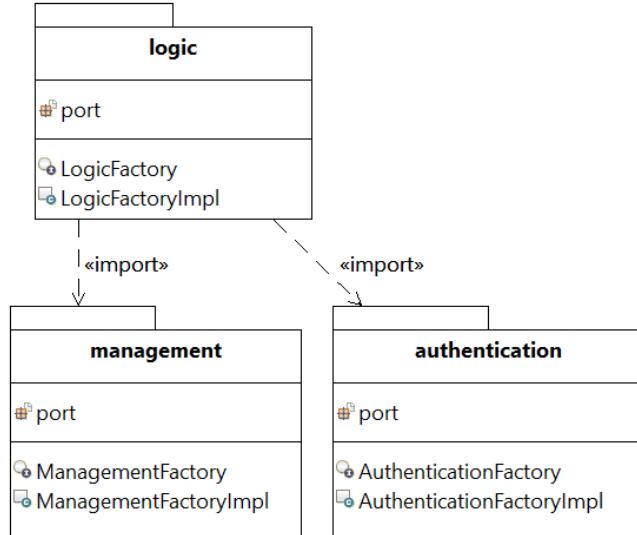
```
public interface SimpleManagerPort {  
  
    MemberManagement memberManagement();  
}
```

```
public interface MemberManagement {  
  
}
```



MyClub: Ports als Zugriffspunkte

- Der Zugriff auf das Modell (der Manager-Port)



Interfaces müssen
bereitgestellt werden.



MyClub: Ports als Zugriffspunkte

- Der Zugriff auf das Modell (Delegation des Manager-Ports)

```
public interface LogicFactory {  
  
    LogicFactory FACTORY =  
        new LogicFactoryImpl();  
  
    MVCPort MVCPort();  
  
    ManagerPort managerPort();  
}
```

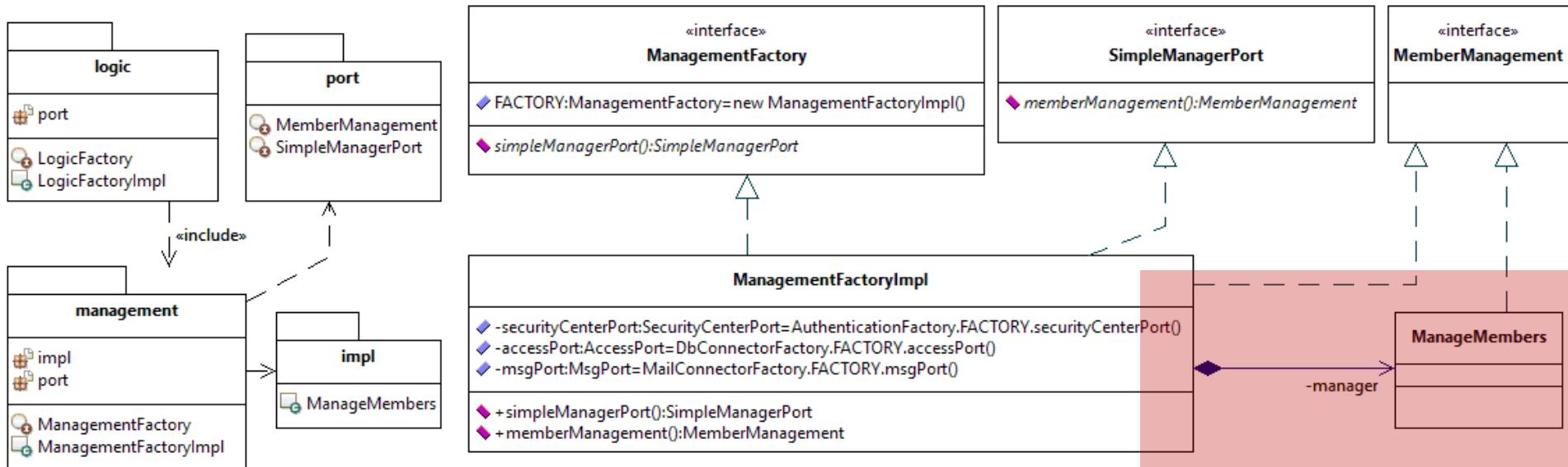
Die richtige Sichtbarkeit
versteckt die
Implementierung!

```
class LogicFactoryImpl implements LogicFactory, ManagerPort, MVCPort {  
  
    private SimpleManagerPort simpleManagerPort =  
        ManagementFactory.FACTORY.simpleManagerPort();  
    private InspectorPort inspectorPort =  
        AuthenticationFactory.FACTORY.inspectorPort();  
  
    public MVCPort MVCPort() {return this;}  
  
    public ManagerPort managerPort() {return this;}  
  
    public MemberManagement memberManagement() {  
        return this.simpleManagerPort.memberManagement();}  
  
    public UserAuthentication userAuthentication() {  
        return this.inspectorPort.userAuthentication();}  
}
```



MyClub: Ports als Zugriffspunkte

- Der Zugriff auf das Modell (die Realisierung des Interface MemberManagement)



Der Use Case ist isoliert.



MyClub: Ports als Zugriffspunkte

- Der Zugriff auf das Modell (die Realisierung des Interface MemberManagement)

```
public interface ManagementFactory {  
    SimpleManagerPort simpleManagerPort();
```

```
ManagementFactory FACTORY = new  
    ManagementFactoryImpl();  
  
}
```

```
class ManagementFactoryImpl implements ManagementFactory,  
    SimpleManagerPort, MemberManagement {  
  
    private SecurityCenterPort securityCenterPort = AuthenticationFactory.  
    private AccessPort accessPort = DbConnectorFactory.FACTORY.accessPort  
    private MsgPort msgPort = MailConnectorFactory.FACTORY.msgPort();  
  
    public SimpleManagerPort simpleManagerPort() {return this;}  
  
    private ManageMembers manager;  
  
    public MemberManagement memberManagement() {  
        if (this.manager == null)  
            this.manager = new ManageMembers(/*...*/);  
        return this; }  
}
```

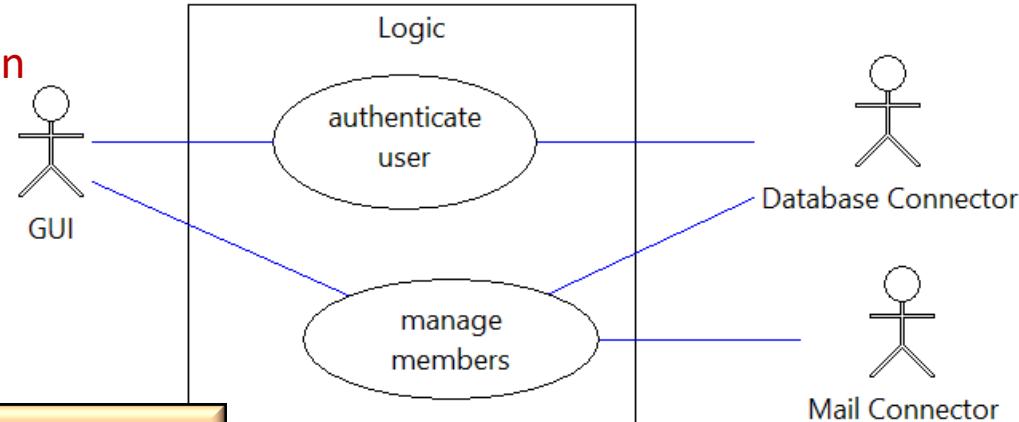
Die Erzeugung erfolgt bei
Bedarf (lazy)!



Die nächsten Designschritte

- Für die umzusetzenden Use Cases werden auf Basis der Analyseergebnisse Mockups erstellt.
- Aus den Use Cases der Analyse werden im Design konkrete System-Use-Cases.

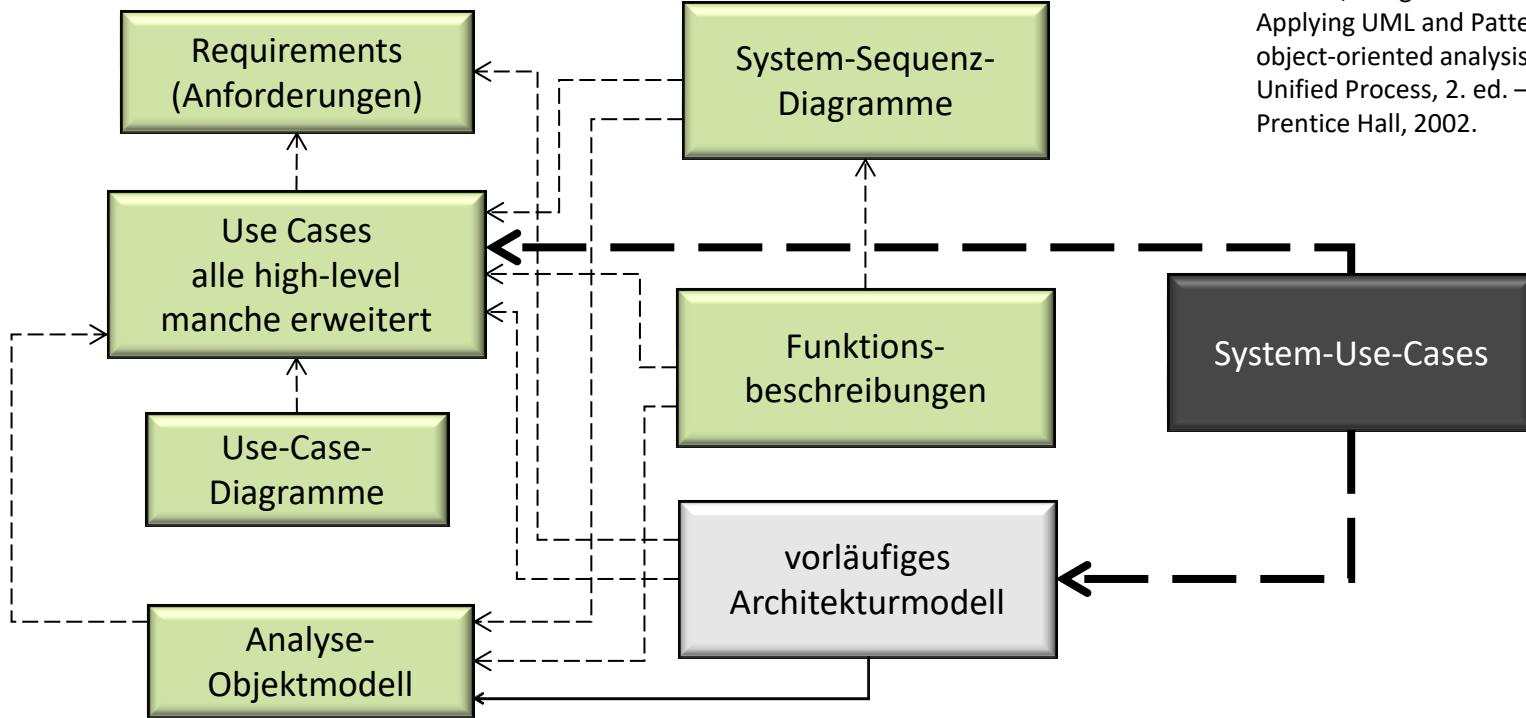
- Input-Output-Terminologie verwenden
- Subsysteme (Architektur) beachten
- über Akteure nachdenken
- technische Use Cases ergänzen
- ...



Im Rahmen der Designphase werden die Use Cases iterativ entworfen und realisiert.



Die nächsten Designschritte



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.



MyClub: Mockups erstellen

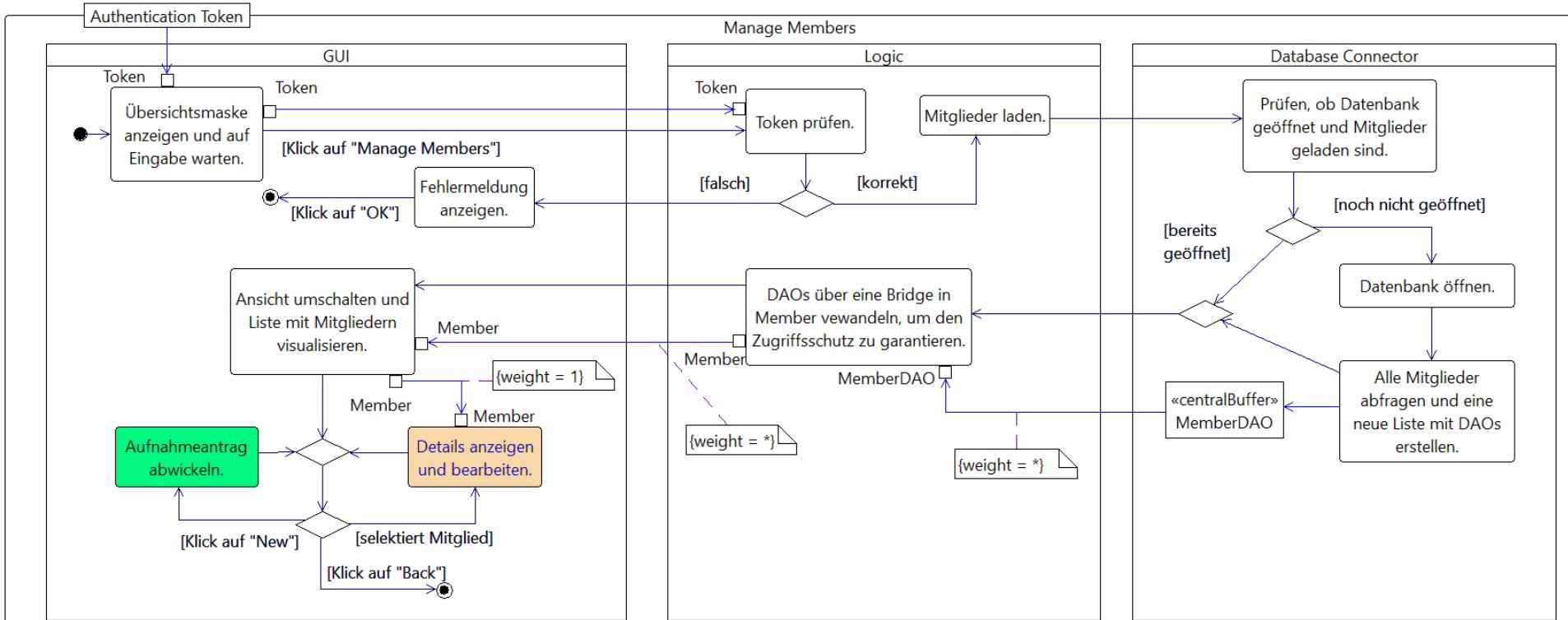
The diagram illustrates the creation of user interface mockups for a MyClub application, featuring four distinct screens:

- Login Screen:** A window titled "MyClub" contains a "Login" button and a "Welcome to MyClub" message. Below these are fields for "Username" (Thomas Fuchs) and "Password" (represented by asterisks). A "Login" button is located at the bottom.
- Manage Members Screen:** A window titled "MyClub: Manage Members" features a "search" input field, a "New" button, and a "Back" button. A list of items including "Berry Backlog", "Use Case", "Sema Phore", "Rup Scrum", "Analy de Sign", and "Ada Thread" is displayed. This screen has its own "Back" button.
- Member Details Screen:** A window titled "MyClub: Maage Member" shows a "Berry Backlog" section. It includes tabs for "Member" and "Department". Below this are fields for "Last": "Backlog", "First": "Berry", "Address": "Swimelane 2", "City": "Join City", "Email": "berry.backlog@swe.club", "Birthday": "10.11.1899", "Gender": "male", and a "Save" button.
- Departments Selection Dialog:** A separate window titled "Departments" prompts the user to "Please choose a department:" with options "Field", "Forest", and "Meadow". It includes "OK" and "Cancel" buttons.

A connection line links the "Berry Backlog" section of the Member Details screen to the "Berry Backlog" section of the Manage Members screen. Another connection line links the "Berry Backlog" section of the Member Details screen to the "Berry Backlog" section of the Error dialog. The Error dialog displays a "Value Exception" with the reason "S\$!S Is not a name!" and an "OK" button.

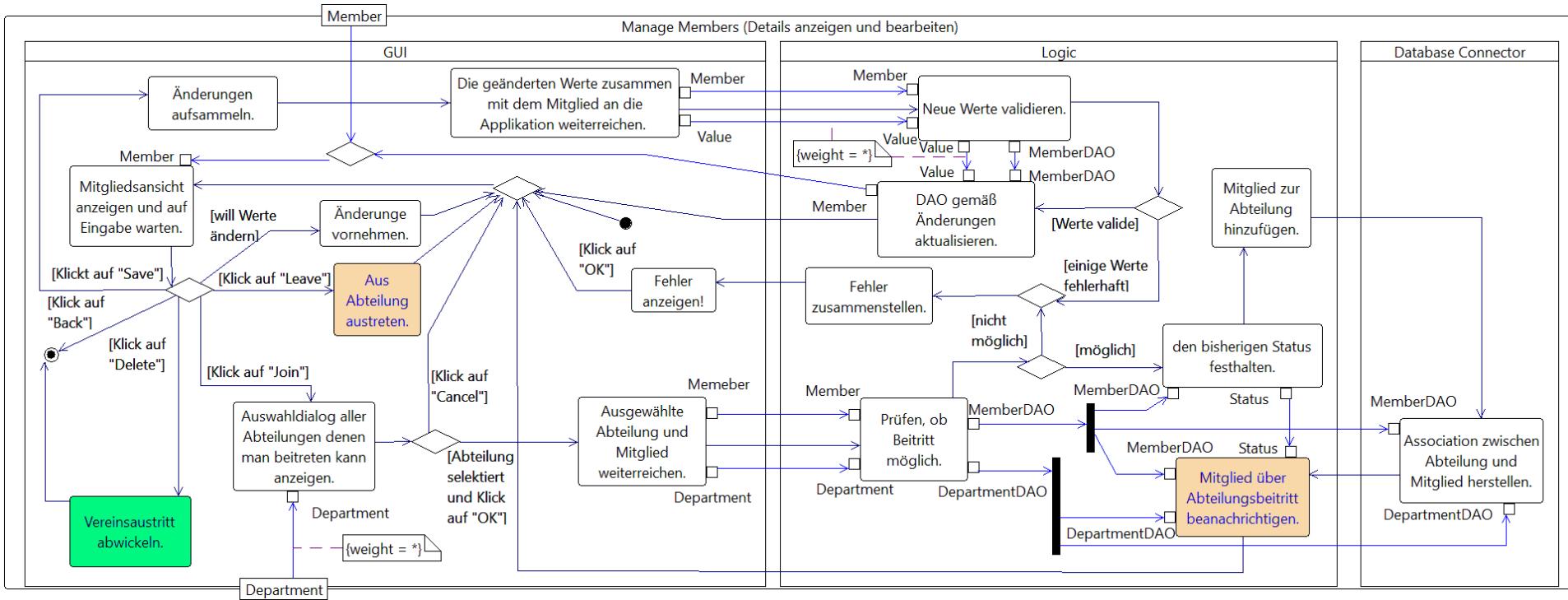


MyClub: Use Case „Manage Members“



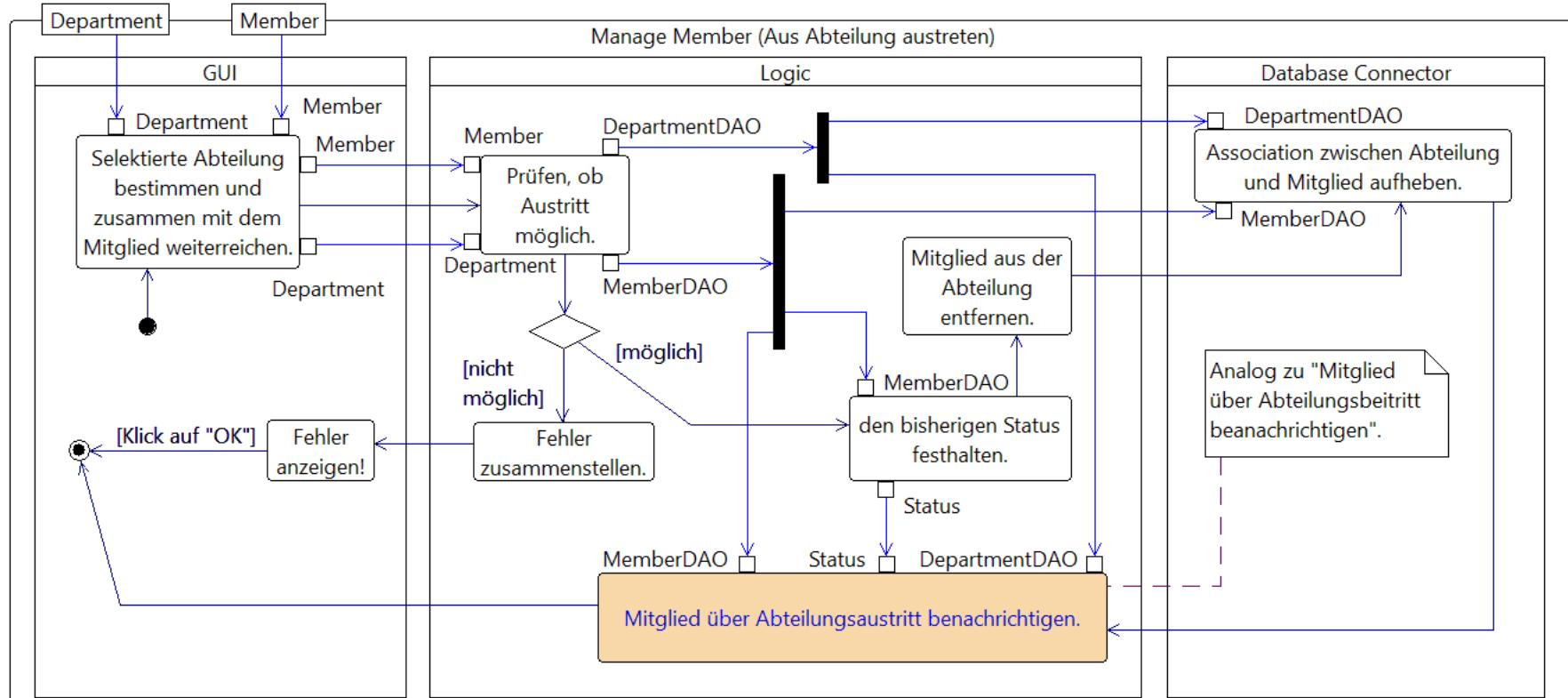


MyClub: Use Case „Manage Members“



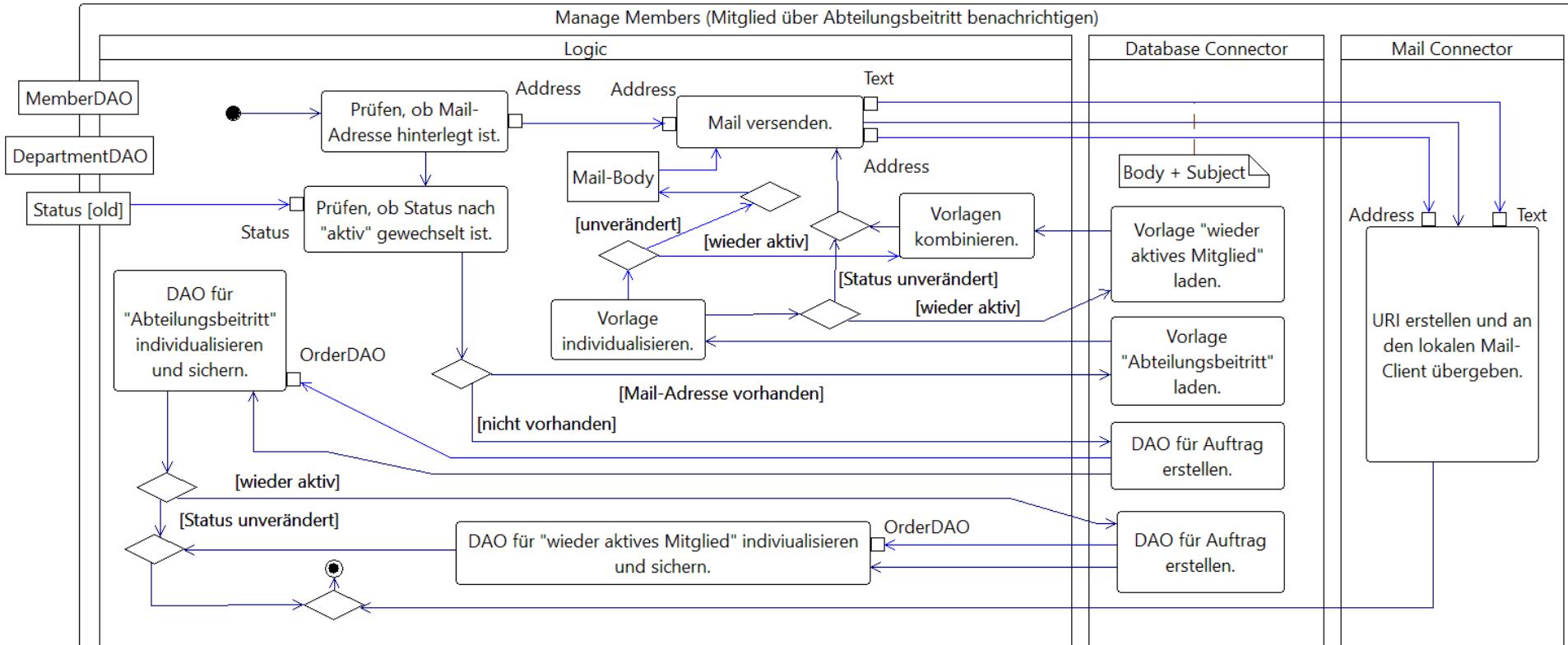


MyClub: Use Case „Manage Members“





MyClub: Use Case „Manage Members“





Schnittstellen definieren

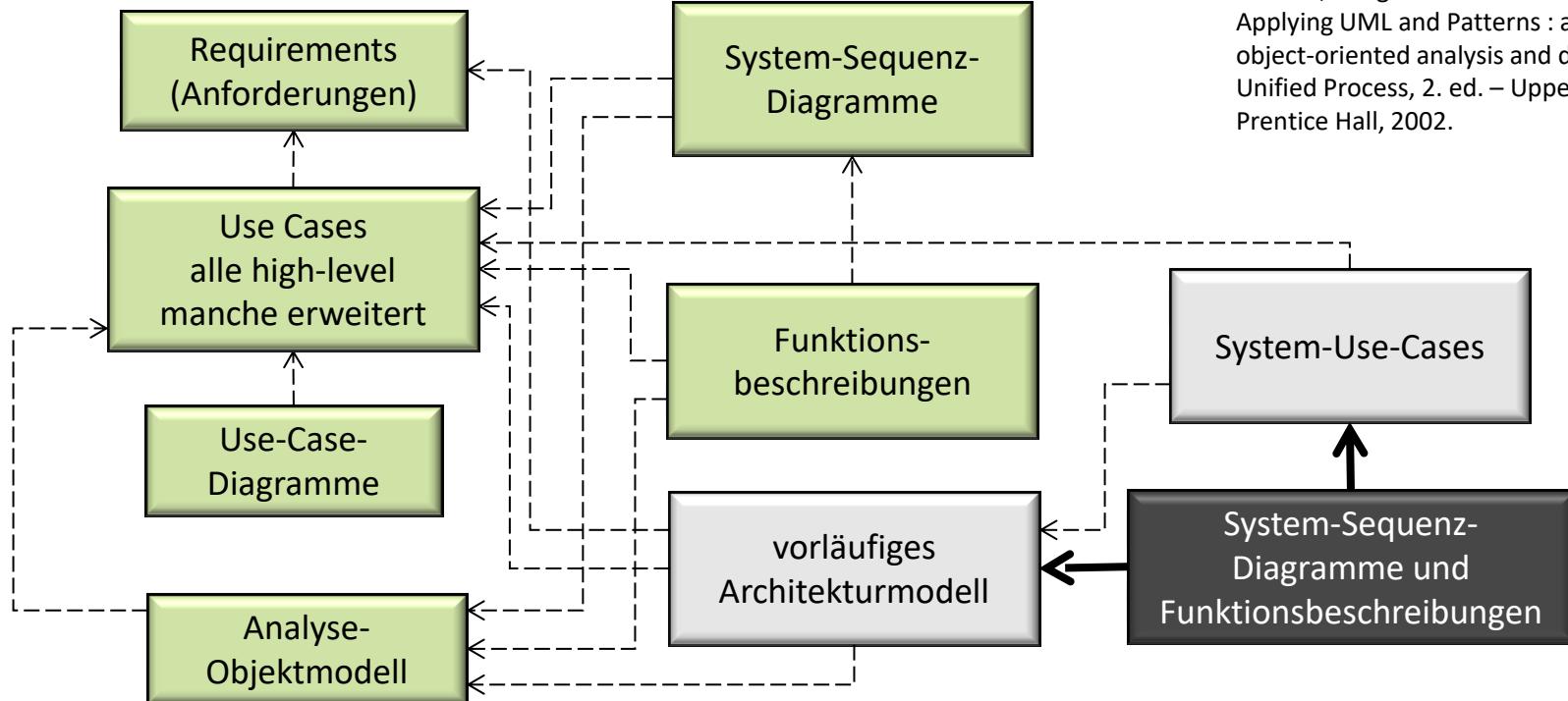
**System-Ereignisse und System-Operationen
sind zu präzisieren.**

- Für jeden Use Case müssen die System-Sequenz-Diagramme 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.**



Die nächsten Designschritte

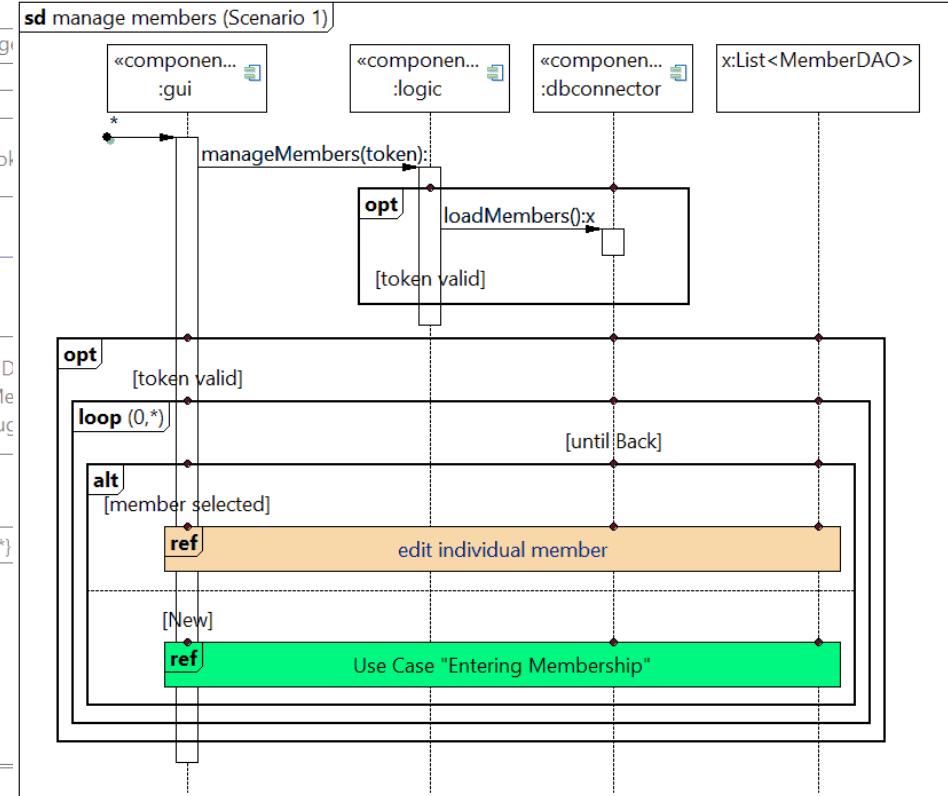
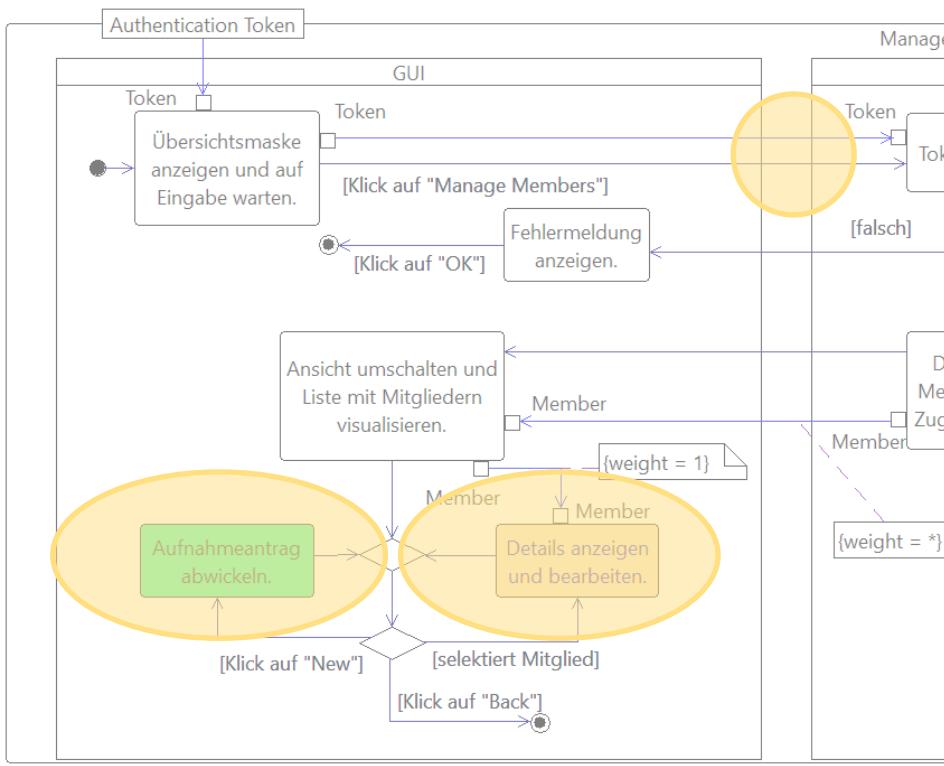


Larman, Craig.

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.

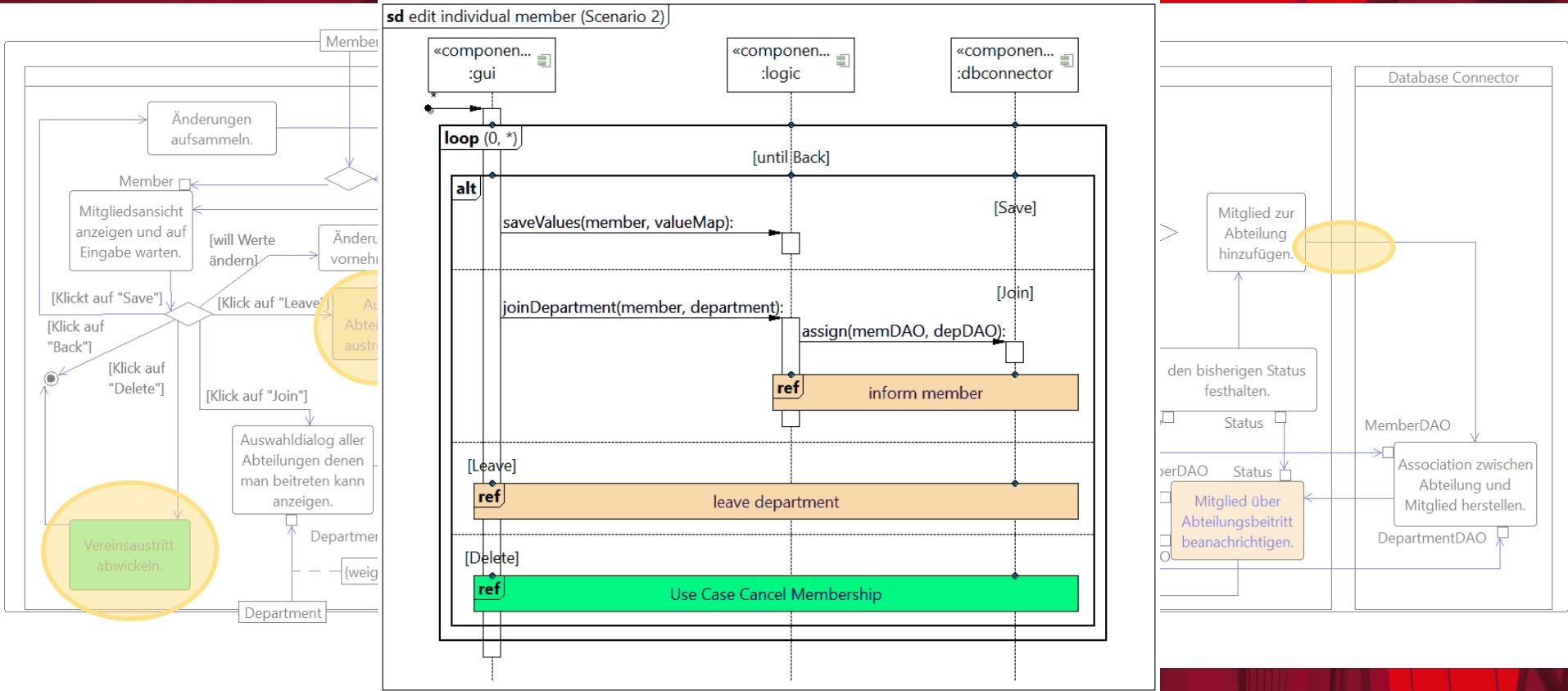


MyClub: System-Operationen für „Manage Members“



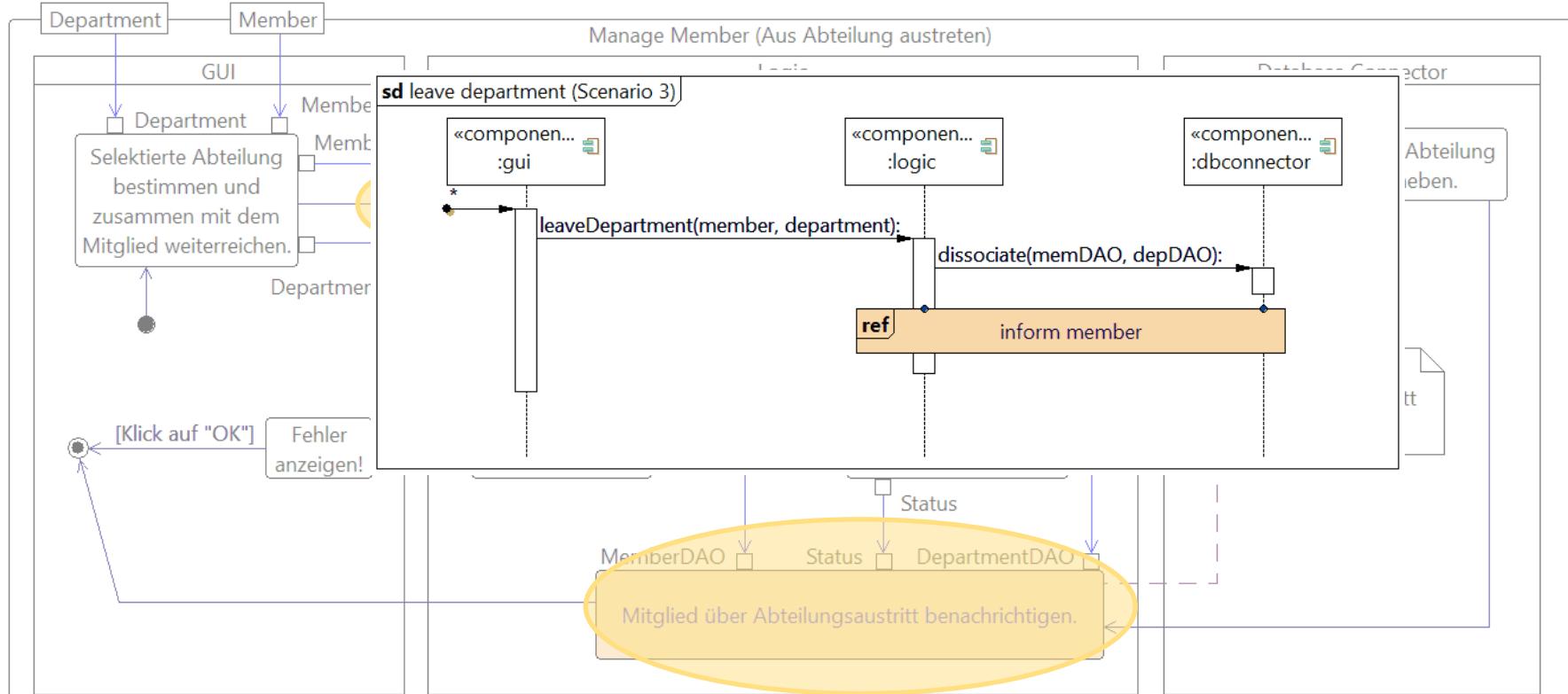


MyClub: System-Operationen für „Manage Members“



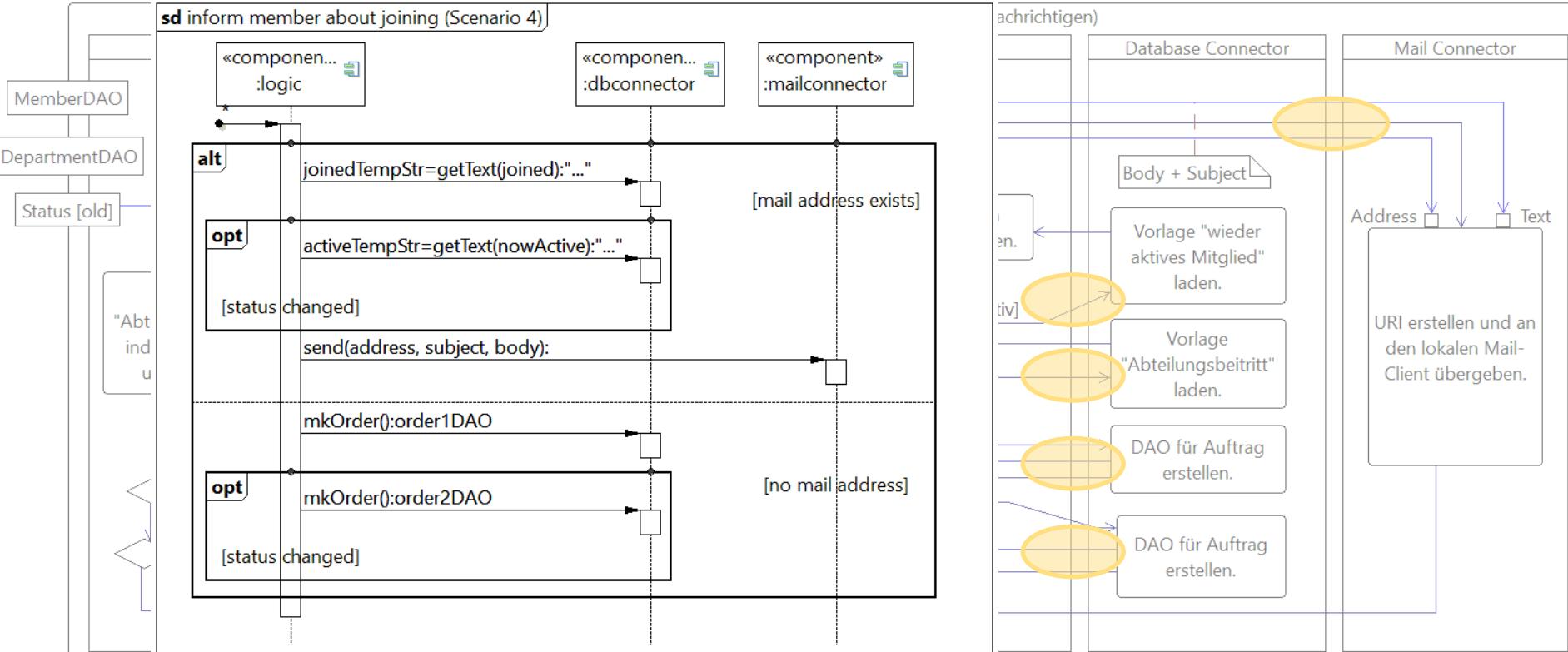


MyClub: System-Operationen für „Manage Members“



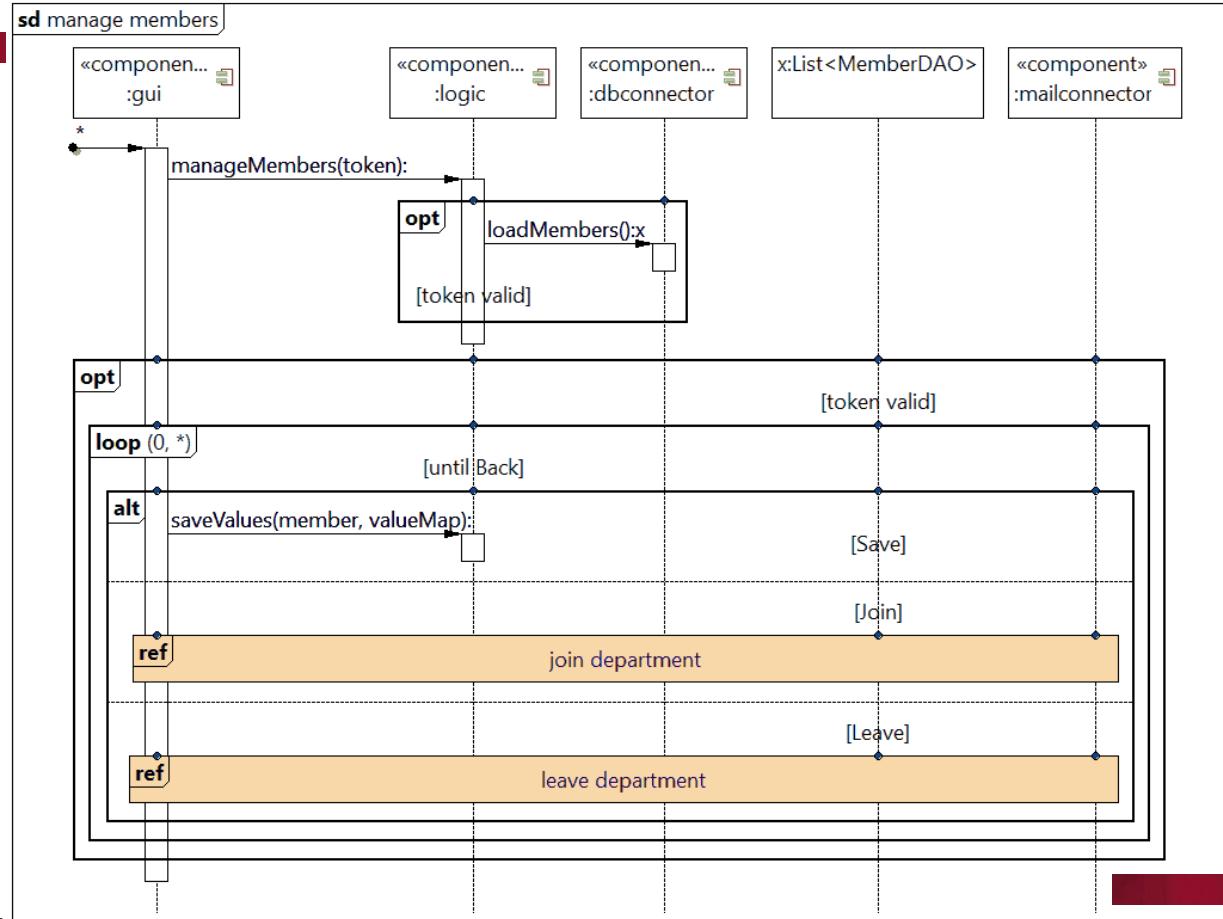


System-Operationen im Use Case „Manage Members“



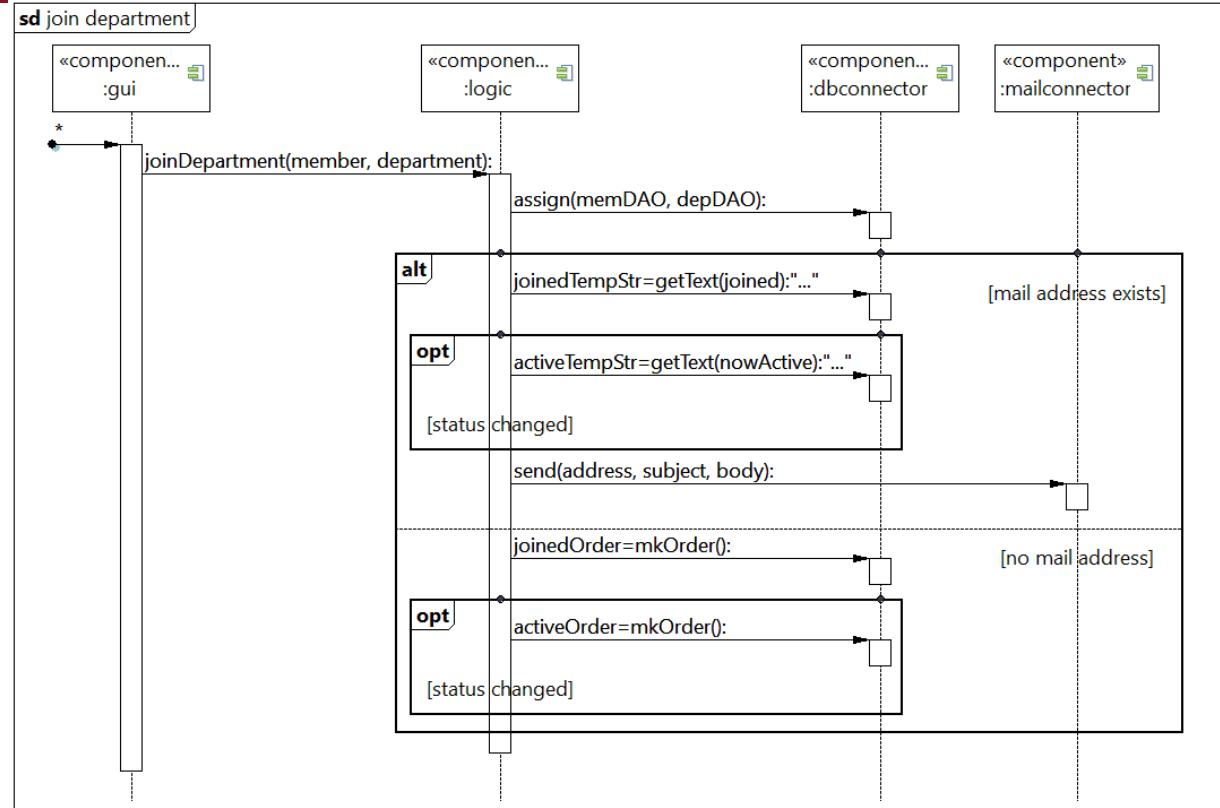


System-Operationen im Use Case „Manage Members“



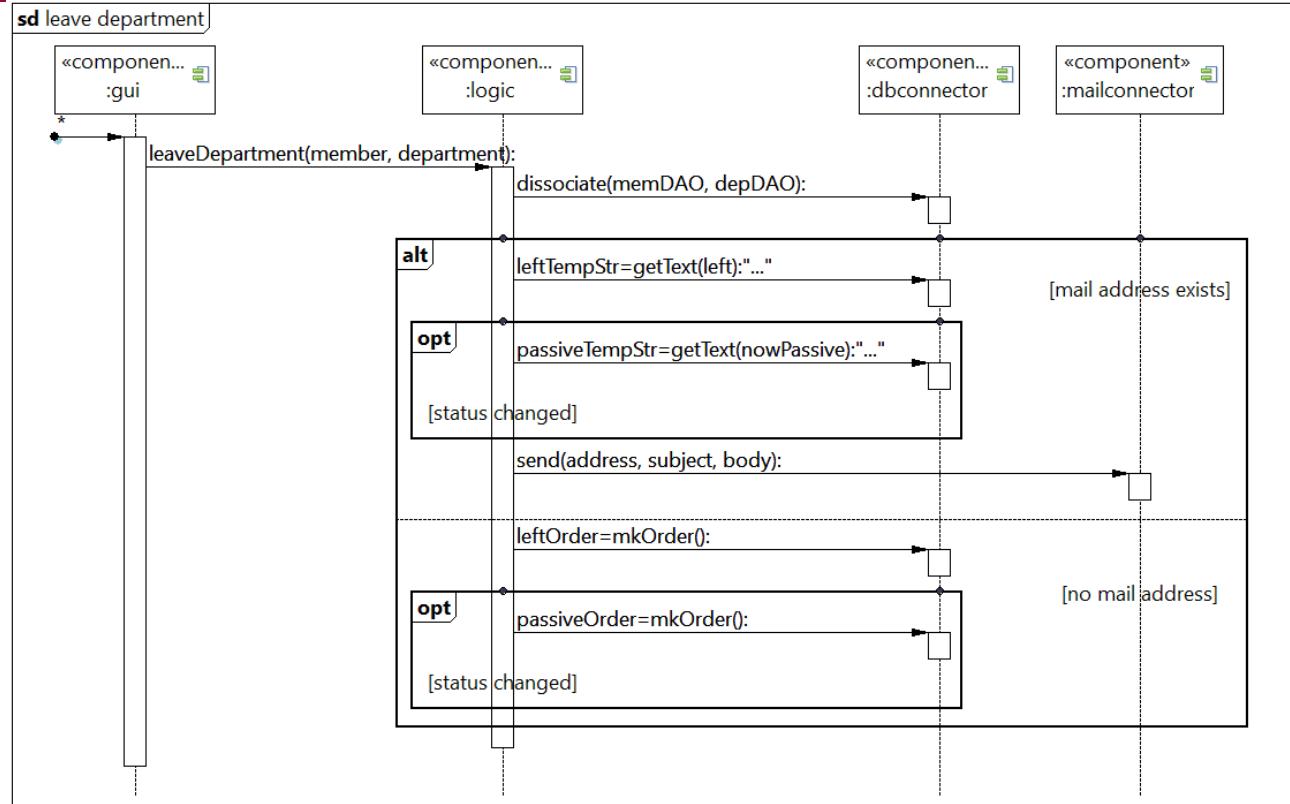


System-Operationen im Use Case „Manage Members“





MyClub: System-Operationen für „Manage Members“



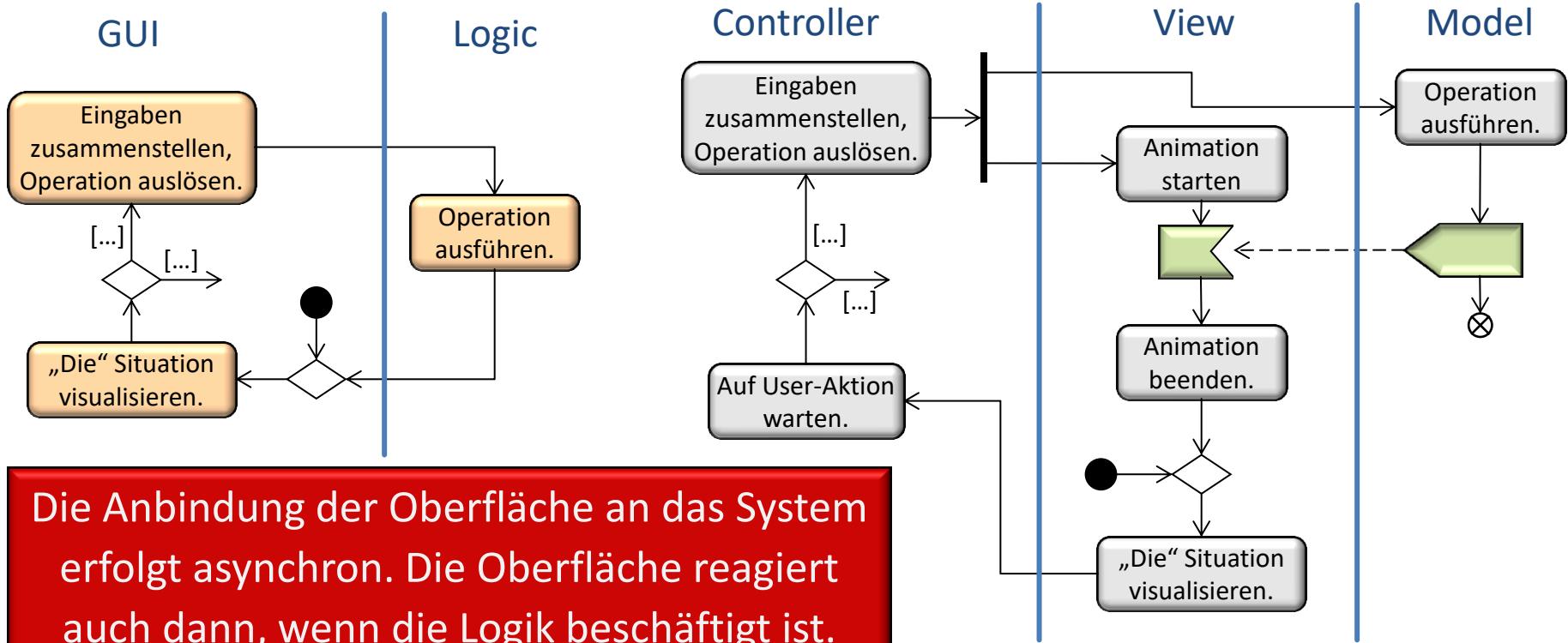


Operationen beschreiben

| | |
|---------------------|--|
| Name: | manageMembers(token: Token) |
| Verantwortlichkeit: | Nach der Validierung des übergebenen Authentisierungstokens wird der Use Case „Manage Members“ initialisiert und die Informationen zu den Mitgliedern aus der Datenbank geladen und für den Zugriff aufbereitet. |
| Referenzen: | Use Case: Manage Members Funktionen: F1.5, ..., A1.1, ... |
| Bemerkungen: | Falls die Validierung fehlgeschlagen ist, dürfen weder Mitglieder aus der Datenbank geladen werden noch darf eine andere Funktion ausgeführt werden, mit deren Hilfe Mitgliederdaten bearbeitet werden können. |
| Ausnahmen: | Das Token kann nicht validiert werden, die Datenbank ist nicht erreichbar. |
| Vorbedingungen: | Das System ist initialisiert. |
| Nachbedingungen: | Mitglieder können bearbeitet werden, oder die Validierung ist fehlgeschlagen und muss wiederholt werden. |



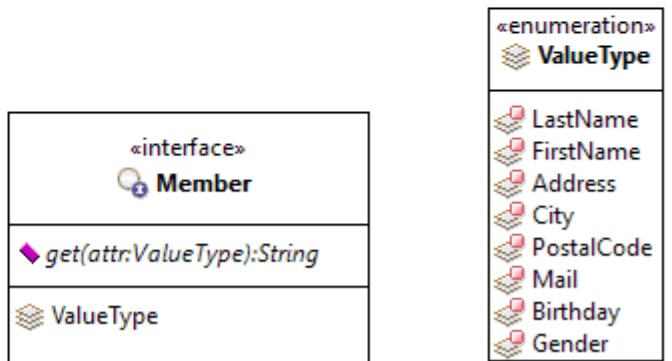
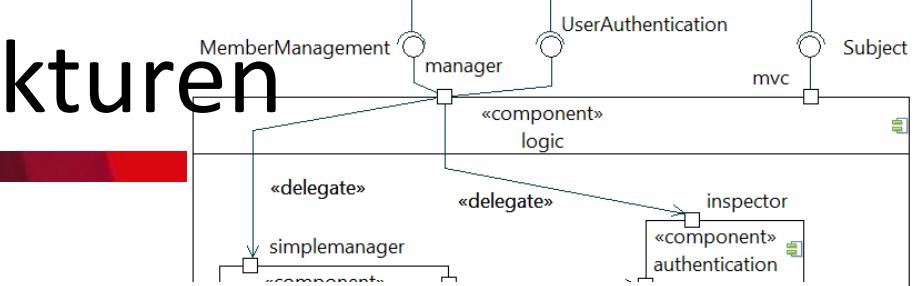
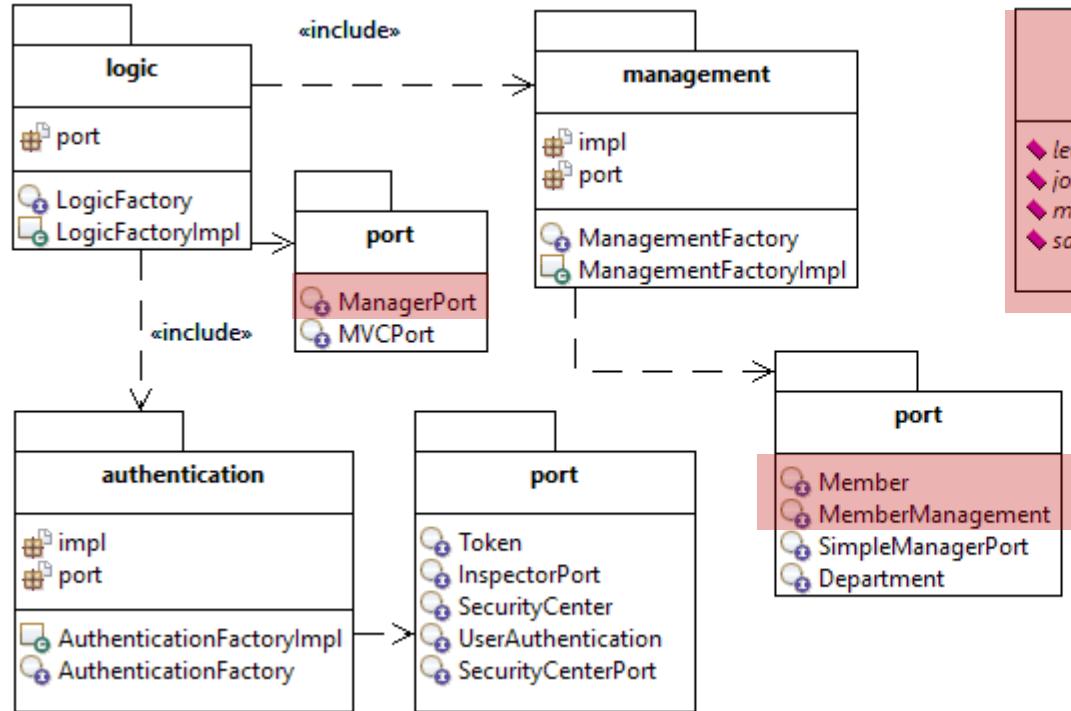
System-Operation sind „void“





MyClub: Neue Strukturen

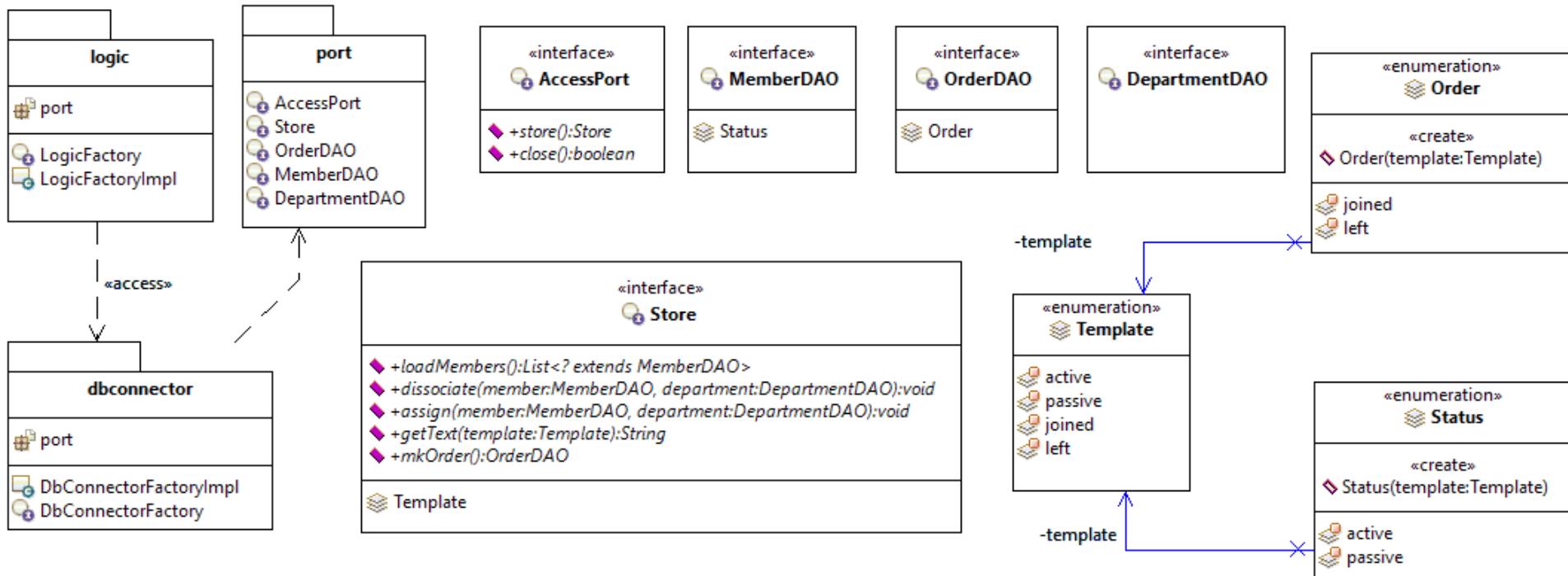
- neue Interfaces am Port „Manager“





MyClub: Neue Strukturen

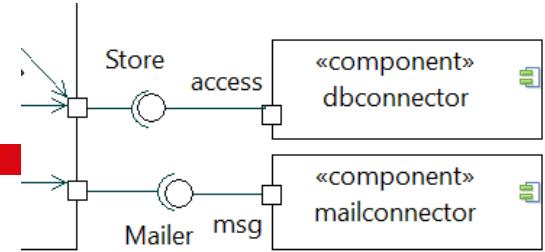
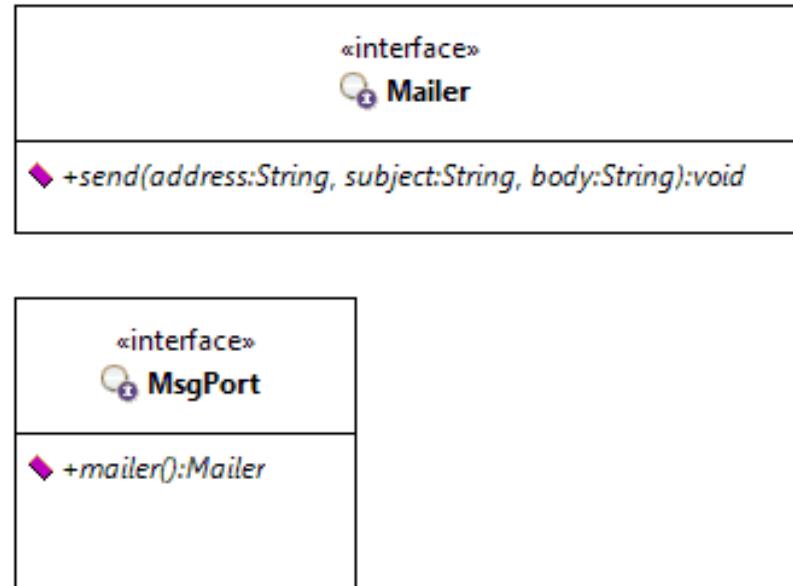
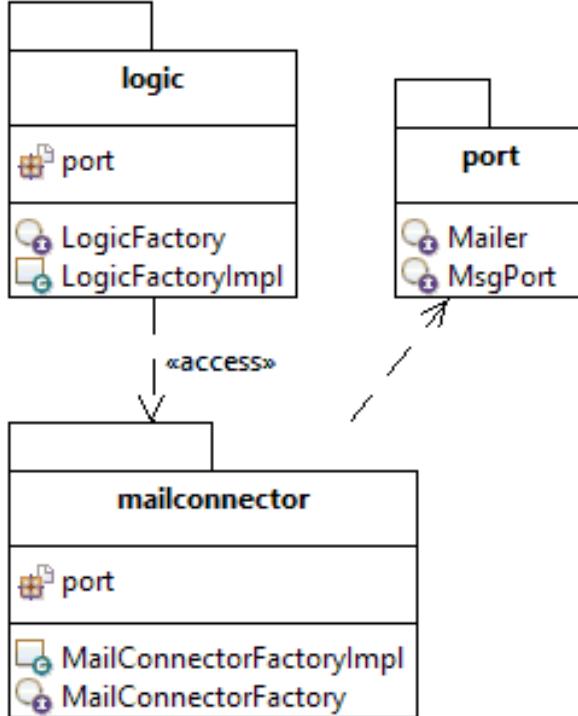
- Interfaces am Access-Port des DbConnectors





MyClub: Neue Strukturen

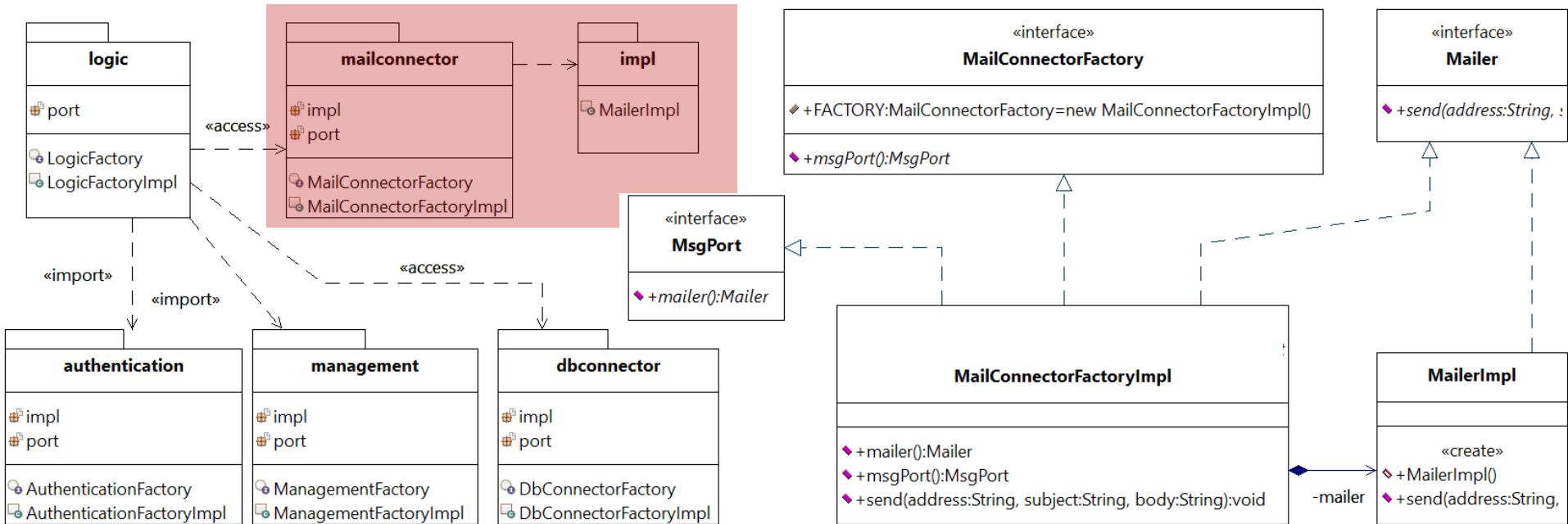
- Interfaces am Port „msg“





MyClub: Neue Strukturen

- Factories erzeugen Interfaces und dienen als Fassade! (vergl. Folie 231)





MyClub: Neue Strukturen

- Die Realisierung erfolgt nach Schema „F“

```
class MailConnectorFactoryImpl implements MailConnectorFactory, MsgPort, Mailer {  
  
    private MailerImpl mailer;  
  
    public synchronized Mailer mailer() {  
        if (this.mailer == null)  
            this.mailer = new MailerImpl();  
        return this;  
    }  
  
    public synchronized MsgPort msgPort() {return this;}  
  
    public synchronized void send(String address, String subject, String body) {  
        this.mailer.send(address, subject, body);}  
}
```

```
public interface MailConnectorFactory {  
  
    MailConnectorFactory FACTORY =  
        new MailConnectorFactoryImpl();  
  
    MsgPort msgPort();  
}
```

Der Rest
analog!



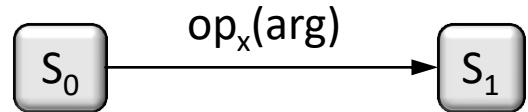
Vor- und Nachbedingungen

| | |
|---------------------|-------------------------------|
| Name: | manageMembers(token: ...) |
| Verantwortlichkeit: | Nach der Validierung des ... |
| Referenzen: | Use Case: Manage ... |
| Bemerkungen: | Falls die Validierung ... |
| Ausnahmen: | Das Token kann nicht ... |
| Vorbedingungen: | Das System ist initialisiert. |
| Nachbedingungen: | Mitglieder können ... |

Folie 249

Jede Operation hat definierte
Vor- und Nachbedingungen.

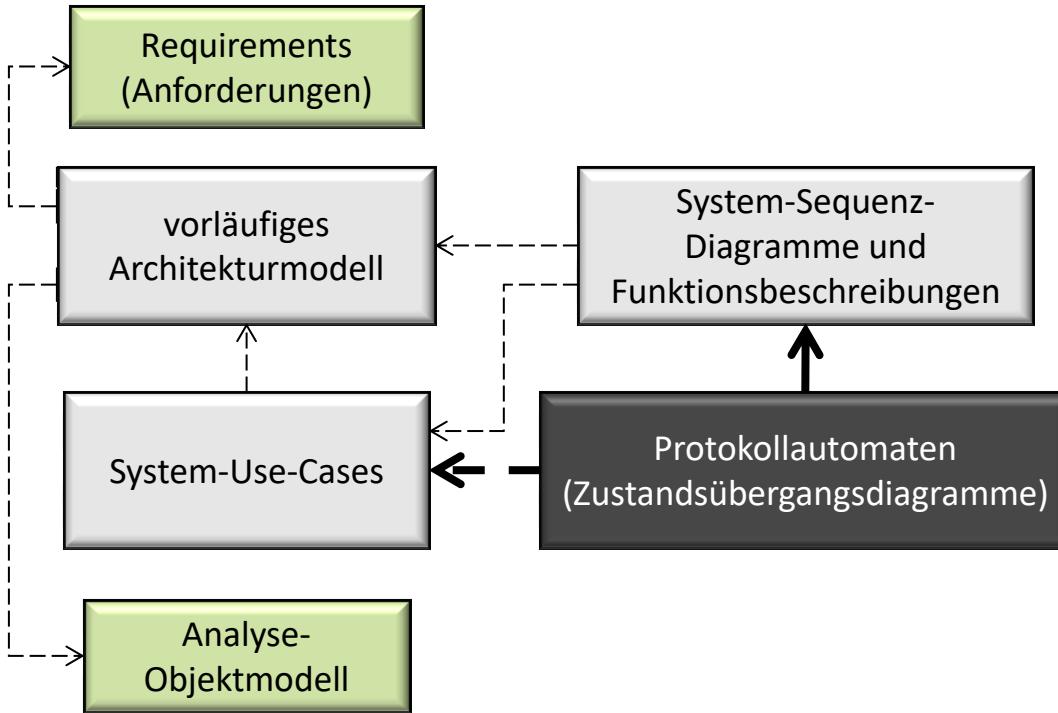
- Vorbedingung:**
die Menge der Zustände, in denen ein Aufruf zulässig ist.
- Nachbedingung:**
die Menge der Zustände, aus denen einer nach Beenden der Operation angenommen wird.



Werden Zustände mithilfe eines „Protokollautomaten“ explizit umgesetzt, bleibt ein System überschaubar. Man weiß stets, wo man sich befindet!



Die nächsten Designschritte



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.

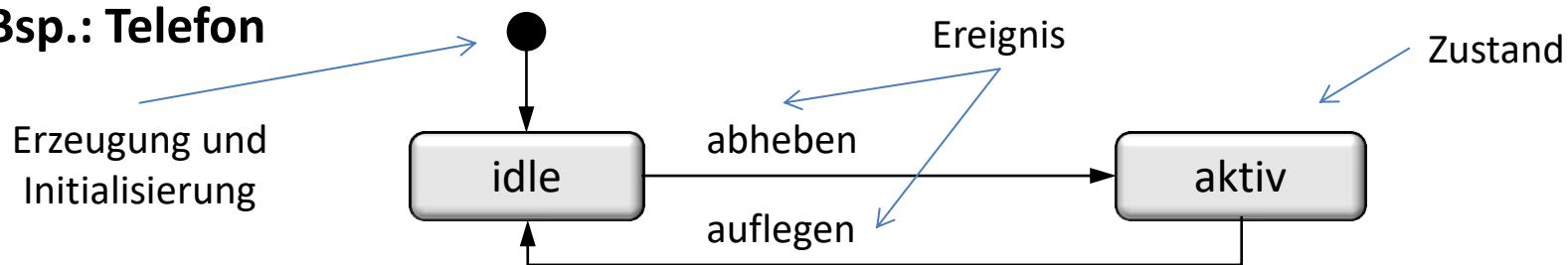


Zustandsdiagramme (State Diagrams)

Ein Zustandsdiagramm veranschaulicht die Zustände von Objekten, Komponenten und ganzen Systemen. D.h., die Veränderungen, die durch innere oder äußere Ereignisse hervorrufen werden.

Ein Zustand repräsentiert dabei typischerweise eine bestimmte Belegung bestimmter Attribute.

Bsp.: Telefon

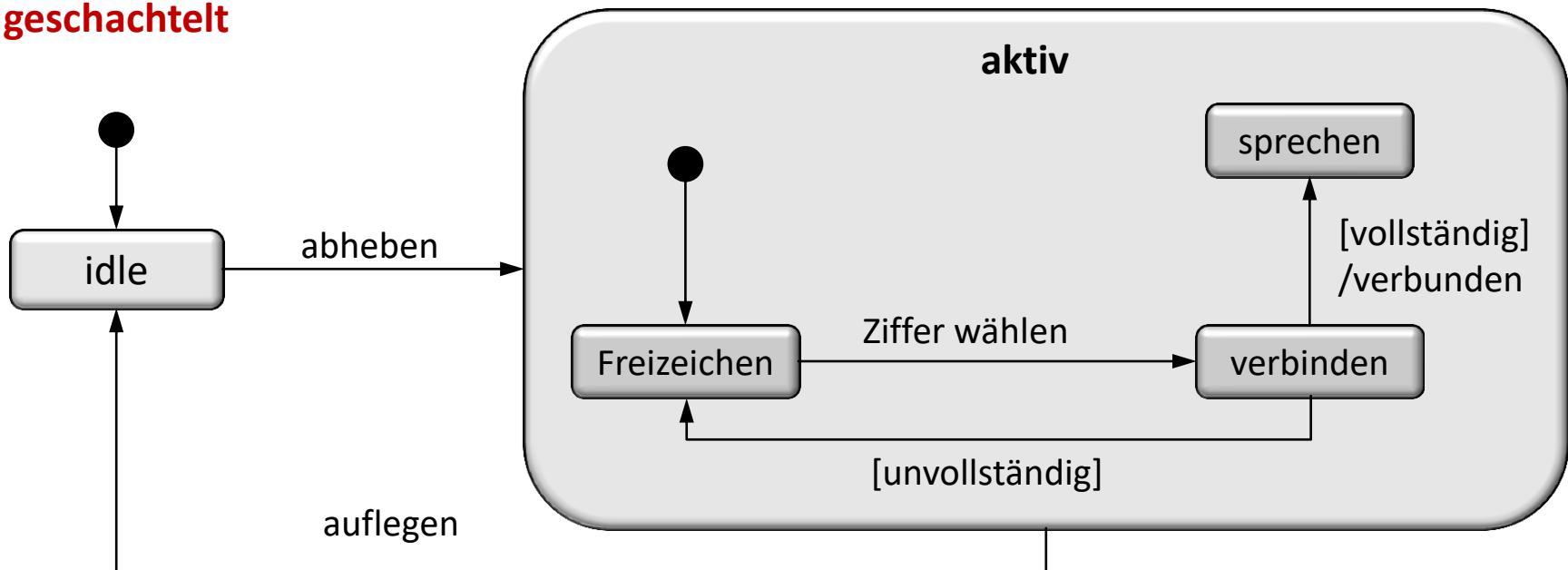




Komplexe Zustände

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

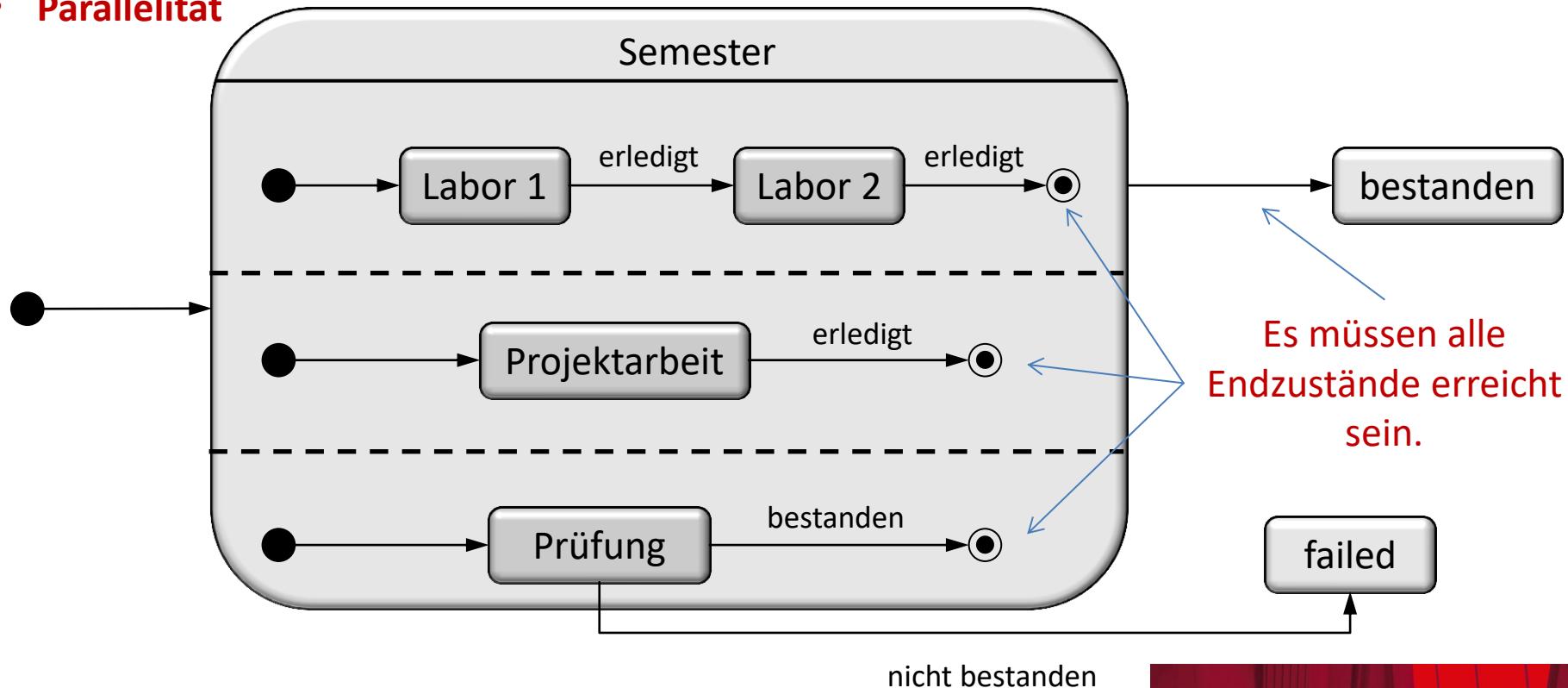
- **geschachtelt**





Komplexe Zustände

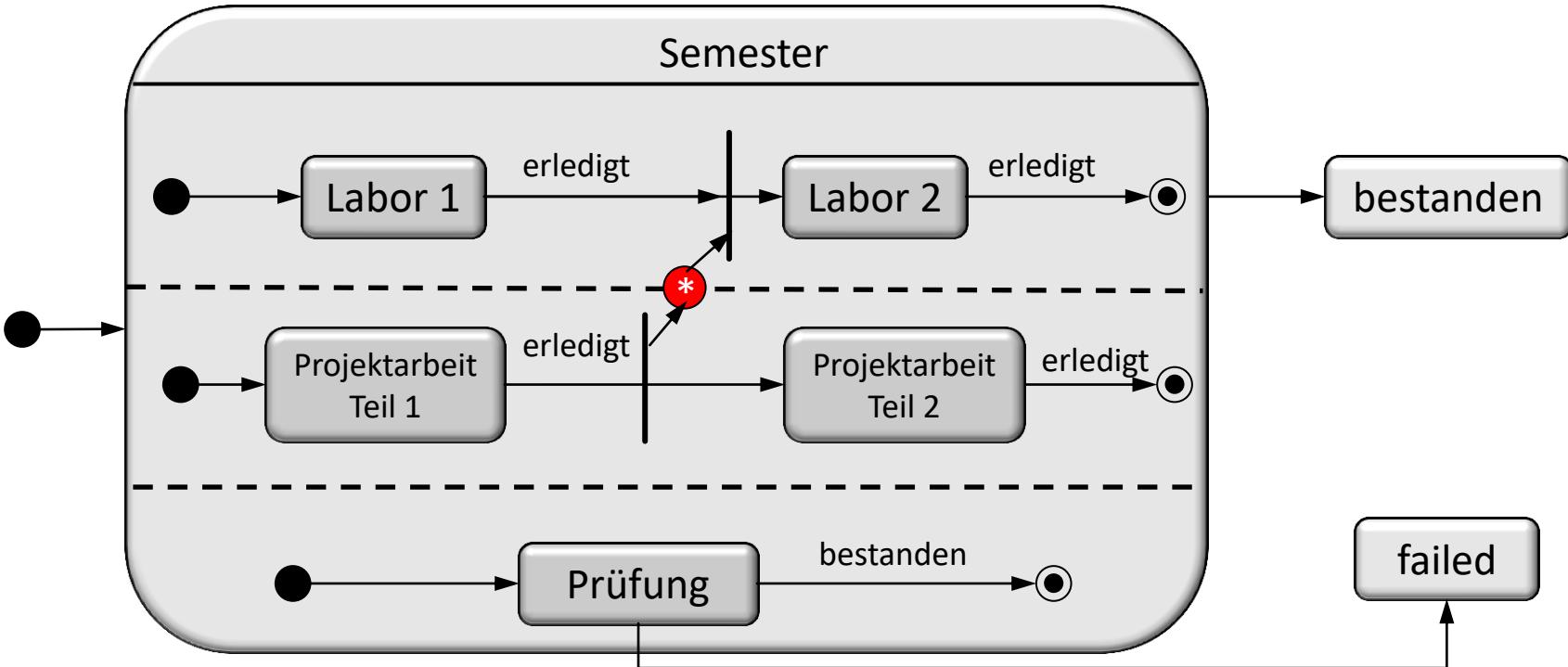
- **Parallelität**





Komplexe Zustände

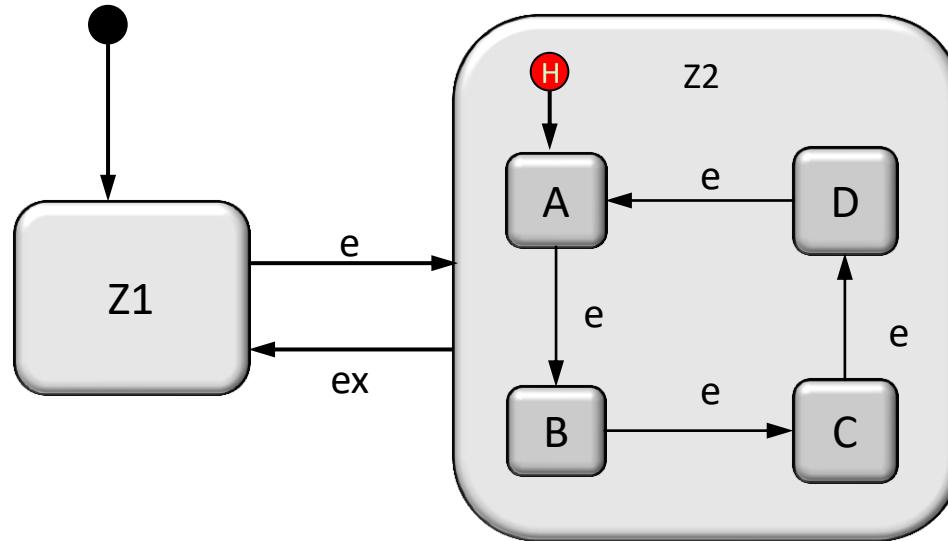
- **Synchronisierung:** Labor 2 beginnt erst, nachdem Teil 1 der Projektarbeit abgeschlossen ist.





Komplexe Zustände

- **History-Zustände** verfügen über ein Gedächtnis.



Ein Ablauf: $Z_1 \xrightarrow{e} A \xrightarrow{e} B \xrightarrow{e} C \xrightarrow{ex} Z_1 \xrightarrow{e} C \xrightarrow{e} D \xrightarrow{ex} Z_1$



Interne Übergänge

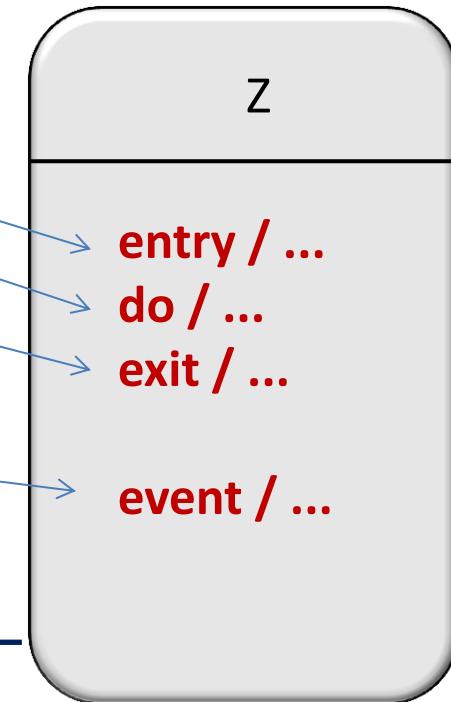
vordefinierte interne Übergänge:

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

selbst definierte interne Übergänge:

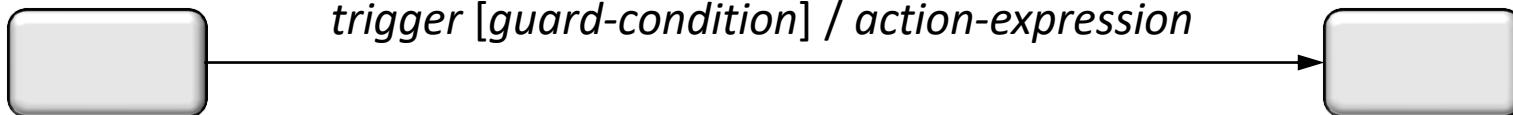
- werden beim Eintreffen des entsprechenden Events ausgeführt
(der Zustand wird dabei nicht verlassen)

completion:
wird beim Beenden ausgeführt





Aufbau von Transitionen



| | | | | |
|--------------------------|---|-----------------------|--------|------------------------|
| <i>trigger</i> | $::= \text{trigger-name}(\text{parameters})$ | Methodenaufruf | Signal | Zeitereignis (after()) |
| <i>parameters</i> | $::= \text{parameter}$ | | | |
| | | parameter, parameters | | |
| <i>parameter</i> | $::= \text{parameter-name} : \text{type-expression}$ | | | |
| <i>guard-condition</i> | $::= \text{bool expr}$ | | | |
| <i>action-expression</i> | $::= \text{pseudo-Code, verwendet Attribute und Methoden der entsprechenden Klassen}$ | | | |

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

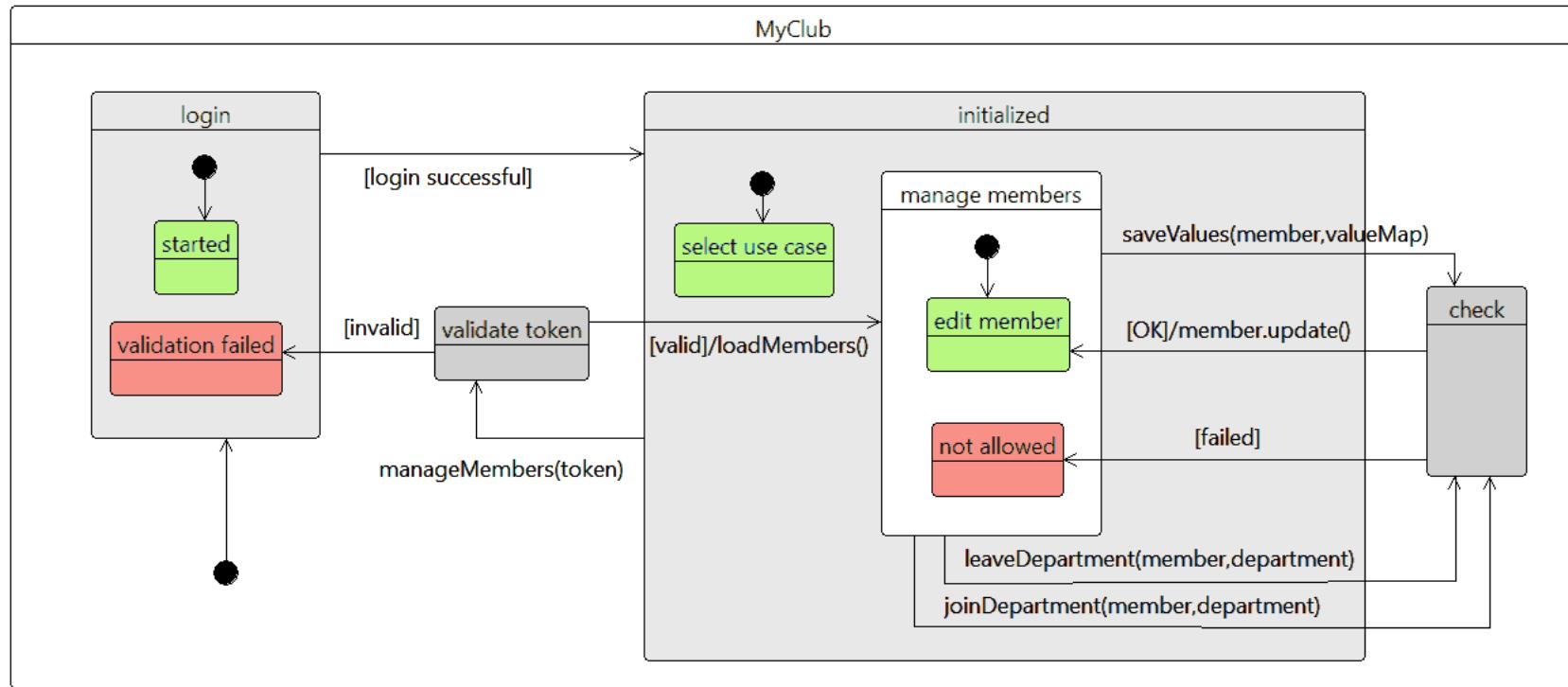


Protokoll-Automaten

- **Zustände:**
Situationen, in denen der Aufruf einer System-Operation erlaubt ist.
- **Trigger:**
Aufruf einer System-Operation im zu modellierenden Protokoll.
- **Guard-Condition:**
Bedingungen, die sich auf Parameter der System-Operation und besondere Aspekte des Zustands beziehen.
- **Action-Expression:**
Ereignisse, die die System-Operation charakterisieren. Im Allgemeinen dargestellt in **Pseudo-Code**.



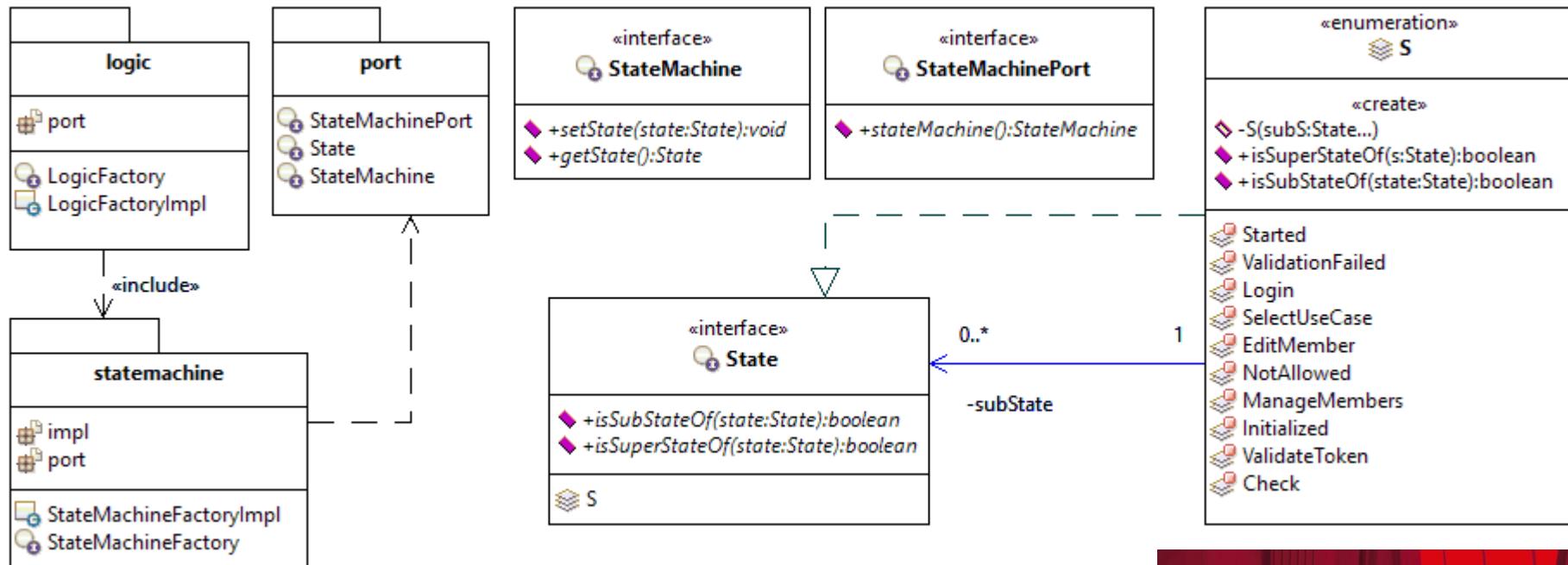
MyClub: Use Case “Manage Members”





MyClub: Zustände einführen

Werden Zustände mithilfe eines „Protokollautomaten“ explizit umgesetzt, bleibt ein System überschaubar. Man weiß stets, wo man sich befindet!



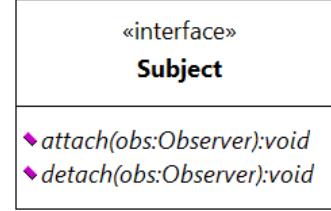
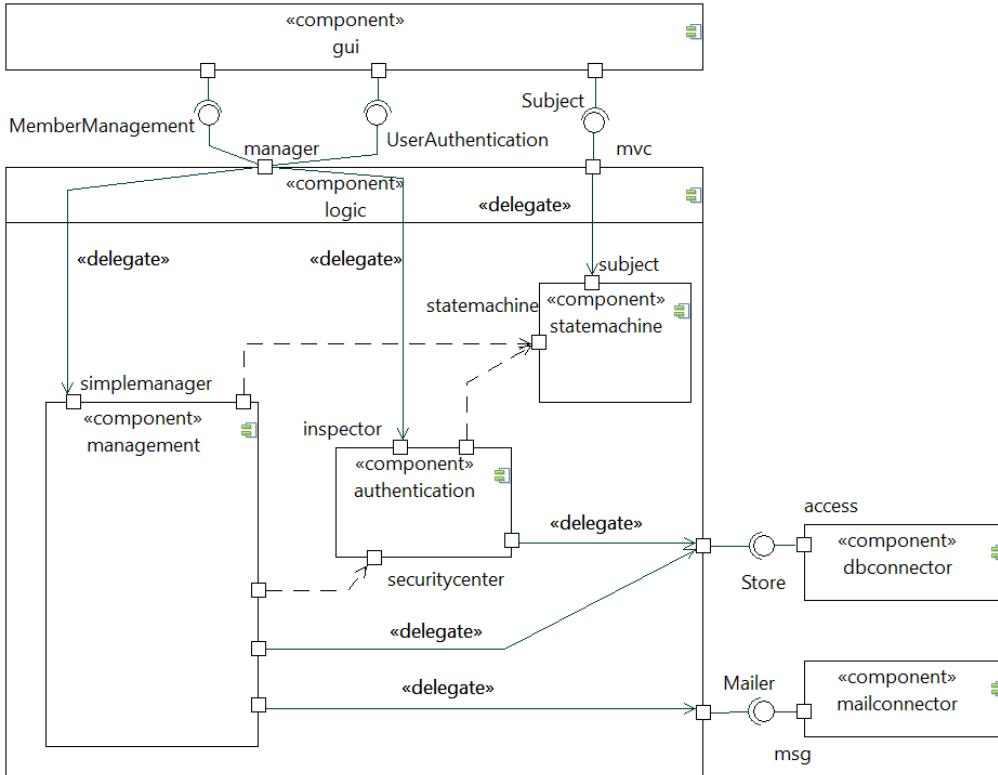


MyClub: Zustände einführen

```
public interface State {  
    public boolean isSubStateOf(State state);  
    public boolean isSuperStateOf(State state);  
  
    public enum S implements State {  
        Started, ValidationFailed, Login(Started, ValidationFailed), //  
        SelectUseCase, EditMember, NotAllowed, ManageMembers(EditMember, NotAllowed), //  
        Initialized(SelectUseCase, ManageMembers), ValidateToken, Check;  
  
        private List<State> subStates;  
        private S(State... subS) {this.subStates = new ArrayList<>(Arrays.asList(subS));}  
  
        public boolean isSuperStateOf(State s) {  
            boolean result = s == null || this == s; // self contained  
            for (State state : this.subStates) // or  
                result |= state.isSuperStateOf(s); // contained in a substate!  
            return result;}  
  
        public boolean isSubStateOf(State state) {return state == null ? false : state.isSuperStateOf(this);} } }
```



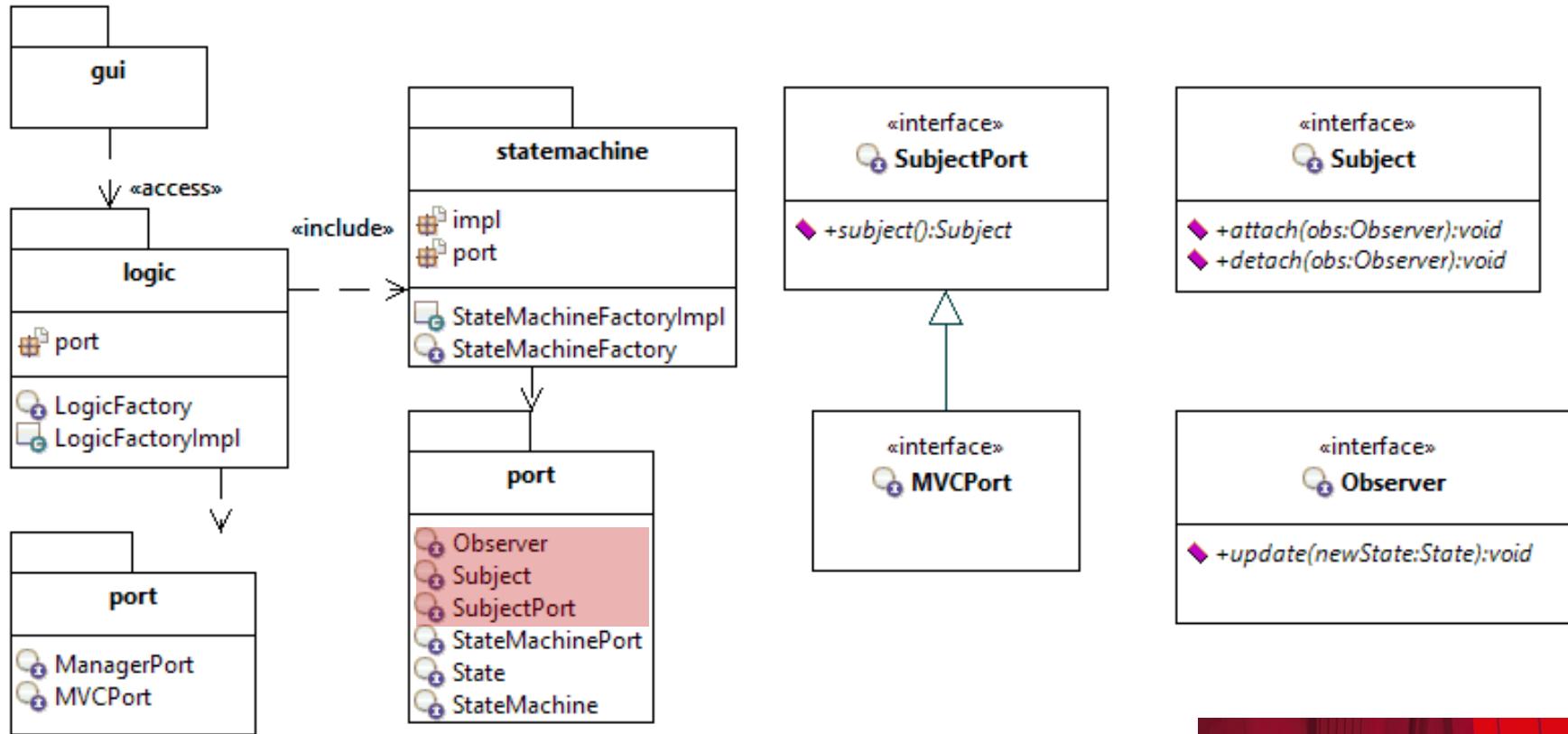
Der MVC-Port nimmt Gestalt an!



Wenn Views und Controller (Observer) die Zustände der Anwendung überwachen, und eine explizite Zustandsmaschine vorhanden ist, dann macht es Sinn, dass diese die Implementierung des „Subjekts“ übernimmt.

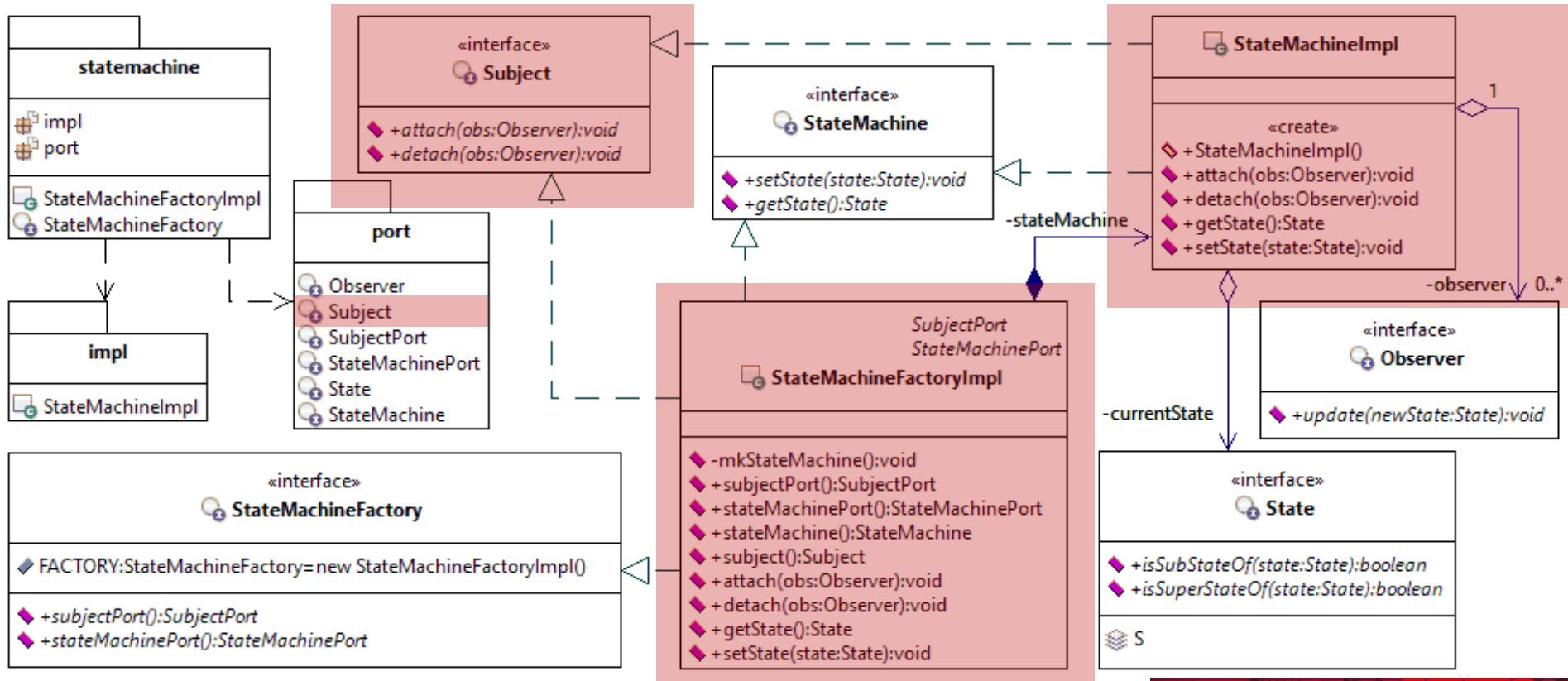


MyClub: Der MVC-Port nimmt Gestalt an!





MyClub: Die Implementierung des Subjekts





MyClub: Die Implementierung des Subjekts

```
class StateMachineFactoryImpl impl. StateMachineFactory,SubjectPort,StateMachinePort,StateMachine,Subject{  
  
    private StateMachineImpl stateMachine;  
  
    private void mkStateMachine(){ if (stateMachine == null) stateMachine = new StateMachineImpl();}  
  
    public SubjectPort subjectPort(){ return this;}  
    public StateMachinePort stateMachinePort(){ return this;}  
  
    public synchronized StateMachine stateMachine() {this.mkStateMachine(); return this;}  
  
    public synchronized Subject subject() {  
        this.mkStateMachine(); return this;}  
  
    public synchronized void attach(Observer obs) {this.stateMachine.attach(obs);}  
    public synchronized void detach(Observer obs) {this.stateMachine.detach(obs);}  
    public synchronized State getState() {return this.stateMachine.getState();}  
    public synchronized void setState(State state) {this.stateMachine.setState(state);} } }
```

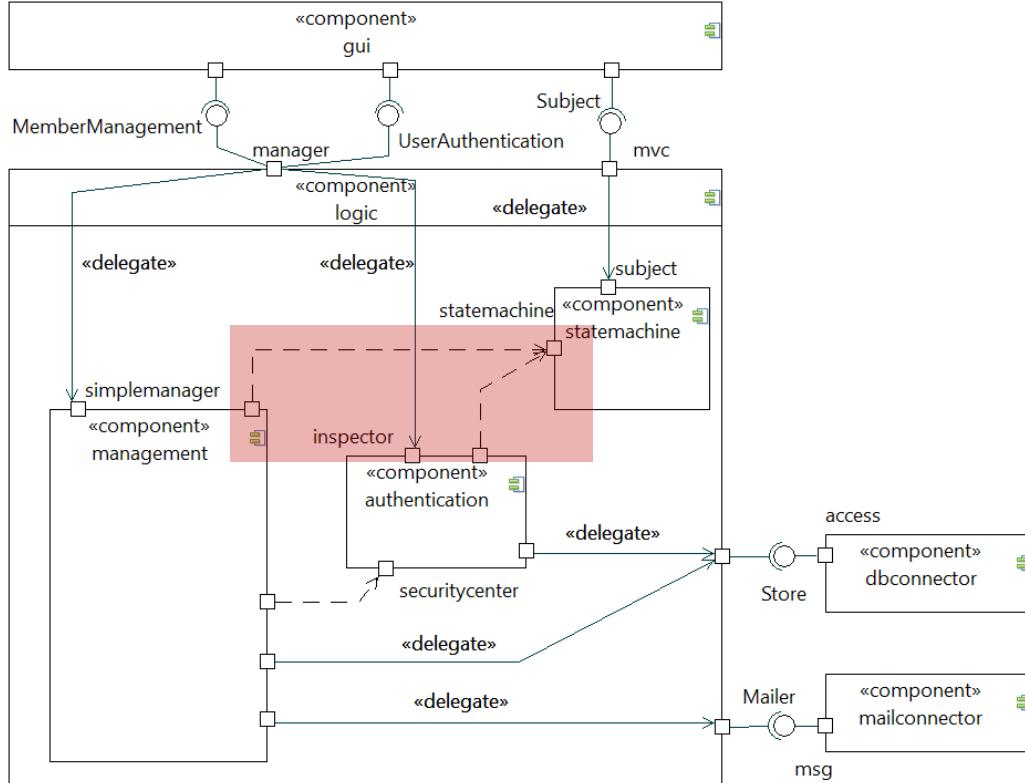


MyClub: Die Implementierung des Subjekts

```
public class StateMachineImpl implements StateMachine, Subject {  
  
    private List<Observer> observers = new ArrayList<>();  
    private State currentState;  
  
    public StateMachineImpl(){this.currentState = State.S.Login;}  
  
    public void attach(Observer obs) {this.observers.add(obs); obs.update(currentState);}  
  
    public void detach(Observer obs) {this.observers.remove(obs);}  
  
    public State getState() {return this.currentState;}  
  
    public void setState(State state) {  
        currentState = state;  
        observers.forEach(obs -> obs.update(state));  
    }  
}
```



Zustände werden beachtet



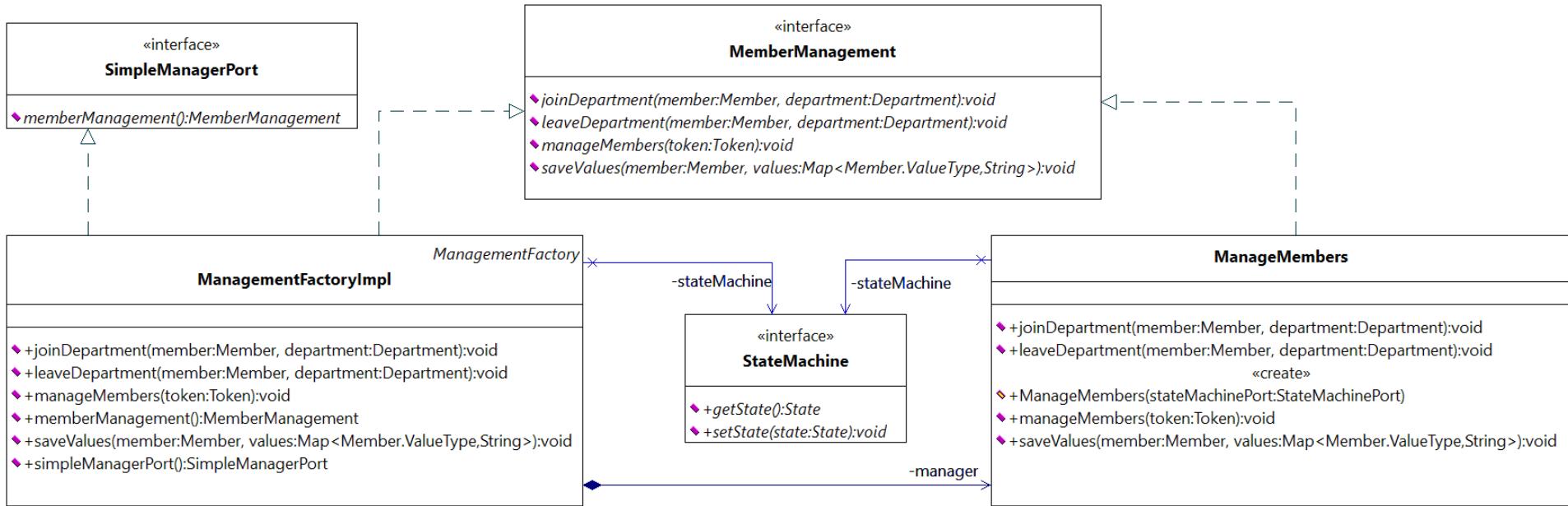
| | |
|---------------------|-------------------------------|
| Name: | manageMembers(token: ...) |
| Verantwortlichkeit: | Nach der Validierung des ... |
| Referenzen: | Use Case: Manage ... |
| Bemerkungen: | Falls die Validierung ... |
| Ausnahmen: | Das Token kann nicht ... |
| Vorbedingungen: | Das System ist initialisiert. |
| Nachbedingungen: | Mitglieder können ... |

Folie 249

Jede Operation hat
definierte Vor- und
Nachbedingungen.



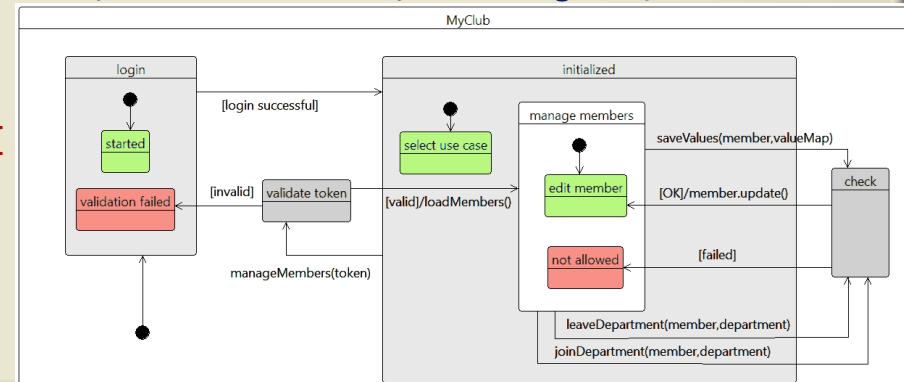
MyClub: Zustände werden beachtet





MyClub: Zustände werden beachtet

```
class ManagementFactoryImpl implements ManagementFactory, SimpleManagerPort, MemberManagement {  
  
    private StateMachinePort stateMachinePort = StateMachineFactory.FACTORY.stateMachinePort();  
    /* siehe Folie 232 */  
    private StateMachine stateMachine;  
    private ManageMembers manager;  
  
    public synchronized MemberManagement memberManagement() {  
        if (this.manager == null) {  
            this.stateMachine = this.stateMachinePort.stateMachine();  
            this.manager = new ManageMembers(this.stateMachinePort, this.accessPort, this.msgPort, this.securityPort);  
        }  
        return this;  
    }  
  
    public synchronized void manageMembers(Token token){  
        if (!this.stateMachine.getState()  
            .isSubStateOf(State.S.Initialized))  
            return;  
        this.manager.manageMembers(token);  
    }  
}
```





Die Struktur steht,
doch noch fehlt
der Inhalt!

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.



Die zentralen 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:**

Die Verantwortlichkeiten sind so zuweisen, dass die Kopplung nicht überhandnimmt.

- **Hohe Kohäsion:**

Die Verantwortlichkeiten sind so zuzuweisen, dass nicht einer alles macht.

- **Management:**

Die Verantwortlichkeit, das System zu organisieren und zu verwalten obliegt einer Klasse, die

- das gesamte System repräsentiert,
- die gesamte Organisation repräsentiert,
- einen Akteur der realen Welt modelliert,
- **den entsprechenden Use Case repräsentiert.**

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 Erstellen von Interaktionsdiagrammen

- Für jede Systemoperation wird ein eigenes Interaktionsdiagramm erstellt.

**Ein Interaktionsdiagramm in dem diese Operation
die „Startoperation“ ist.**

- Falls das Diagramm zu komplex wird, wird es in kleinere aufgespalten.
- Man verwendet die Funktionsbeschreibungen und Use Cases als Ausgangspunkt.

Ziel ist es, jede Funktion durch eine
Menge interagierender Objekte zu
beschreiben.



Interaktionsdiagramme

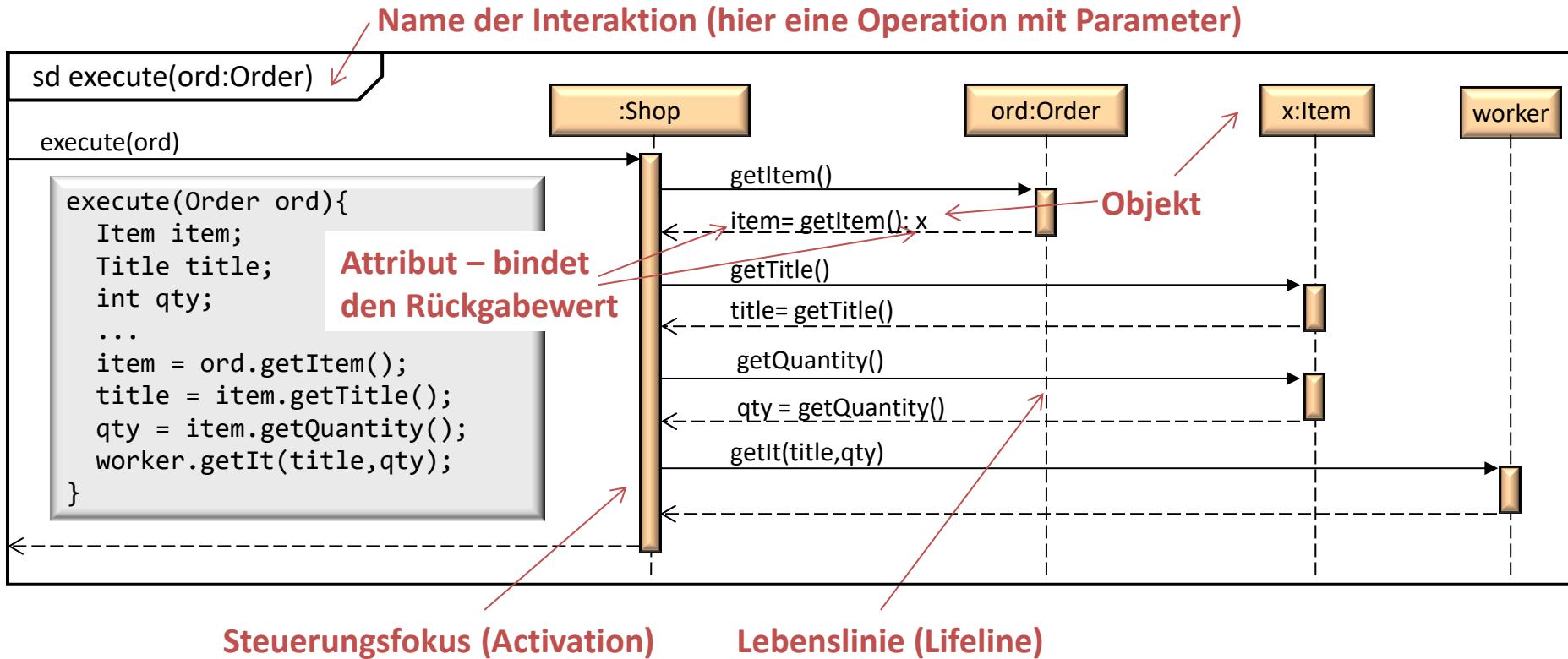
Interaktionsdiagramme beschreiben dynamische Modellsachverhalte, d.h. wie Gruppen von (i.a.) Objekten zusammenarbeiten.

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** (Sequence Diagram)
Diagramme, die den zeitlichen Verlauf der Interaktion in den Vordergrund stellen und die Objektlebenszeit betonen.

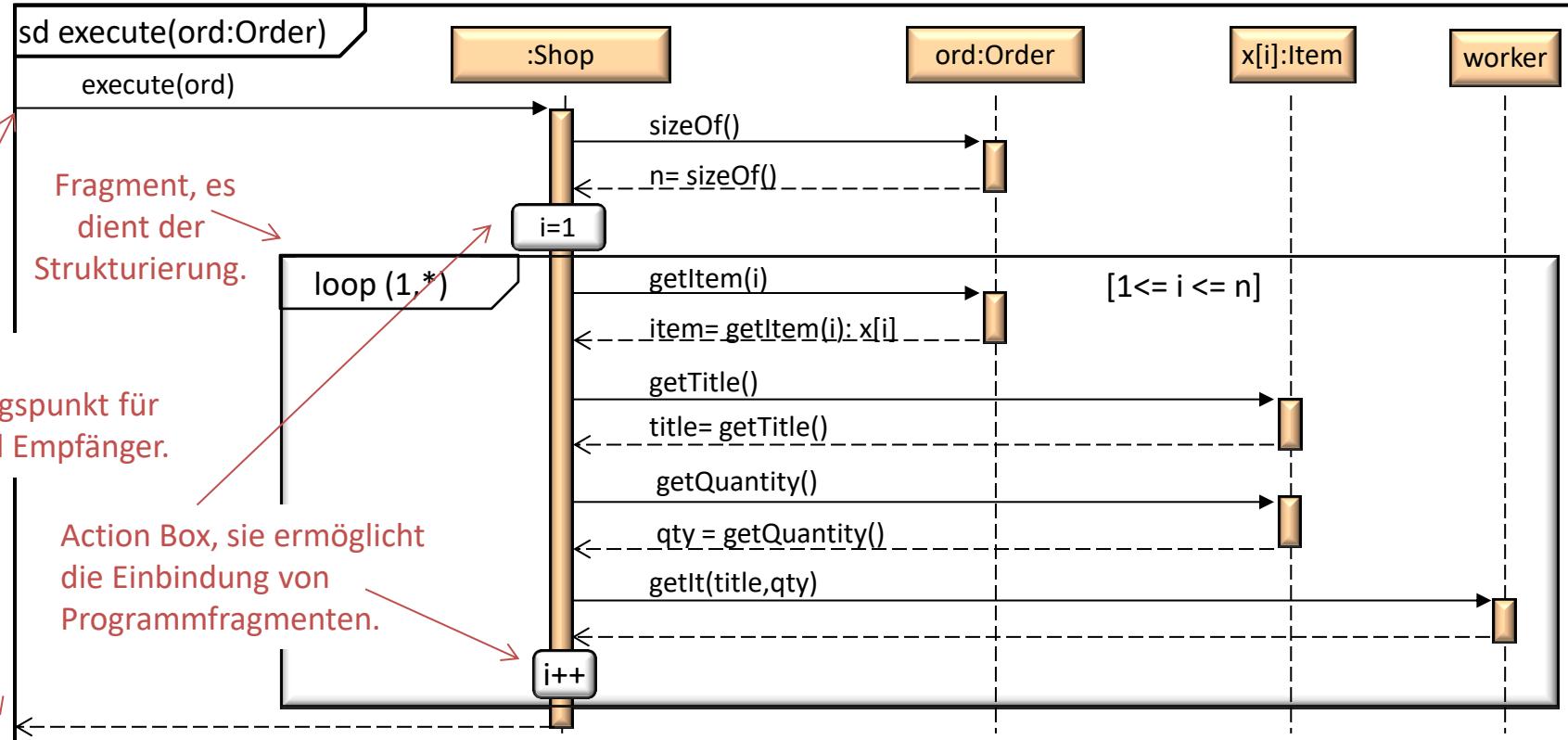


Sequenzdiagramme



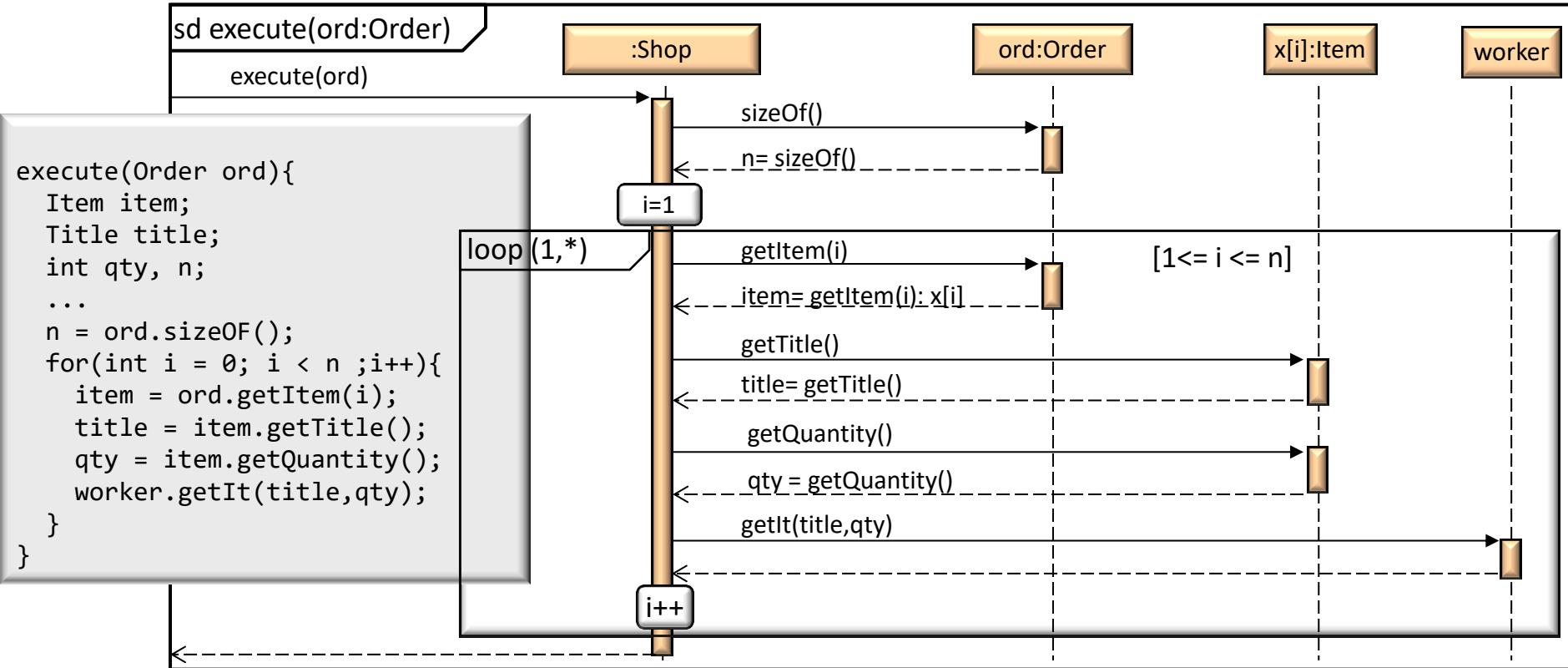


Sequenzdiagramme



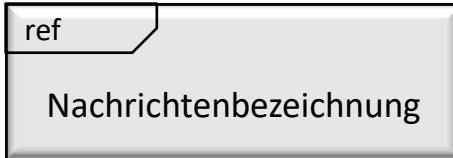


Sequenzdiagramme

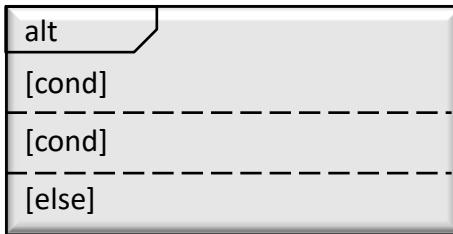




diverse Fragmente



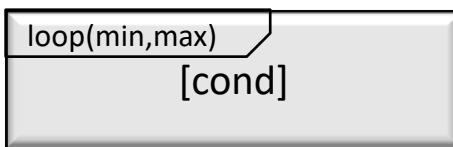
- **Allgemeine Referenz auf ein Fragment
(Die Bindung erfolgt über die Nachrichtenbezeichnung.)**



- **Darstellung alternativer Abläufe**



- **Darstellung optionaler Abläufe**



- **Darstellung iterativer Abläufe**



diverse Fragmente

critical

- Darstellung kritischer Abschnitte

par

- Darstellung paralleler Abläufe

neg

- Darstellung ungültiger Abläufe

break

[cond]

- Darstellung von Ausnahmebehandlungen



diverse Fragmente

assert

seq

strict

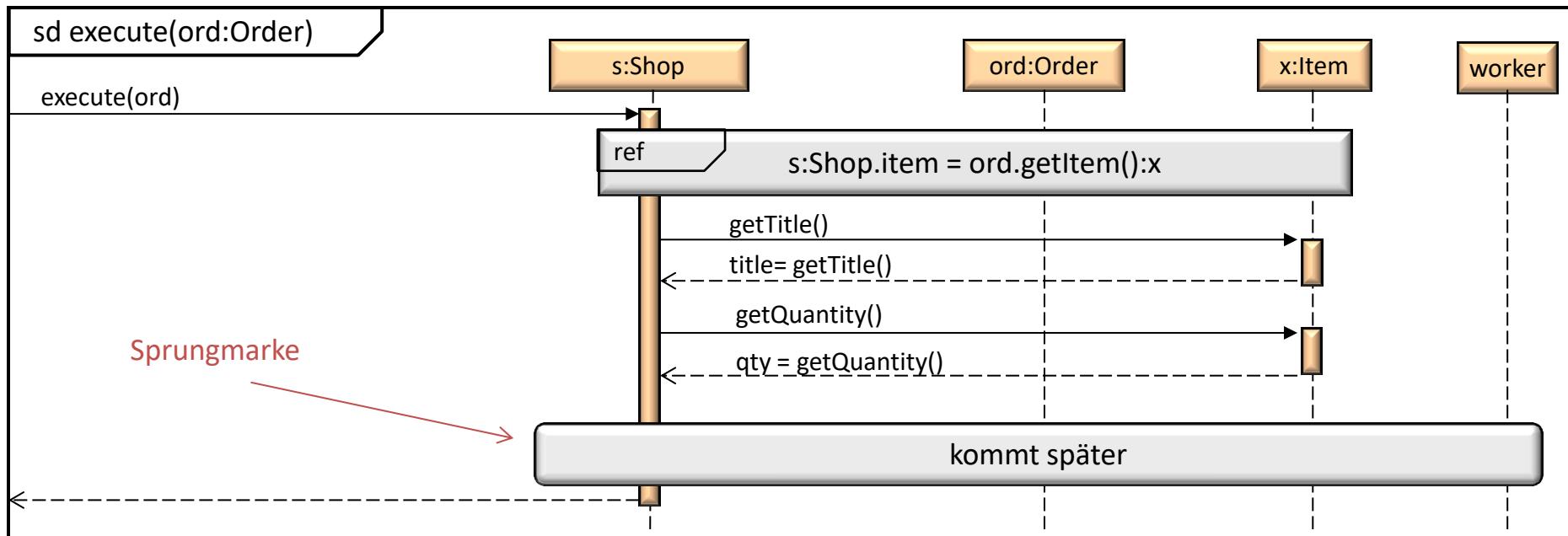
ignore{N}

consider{N}

- **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.**

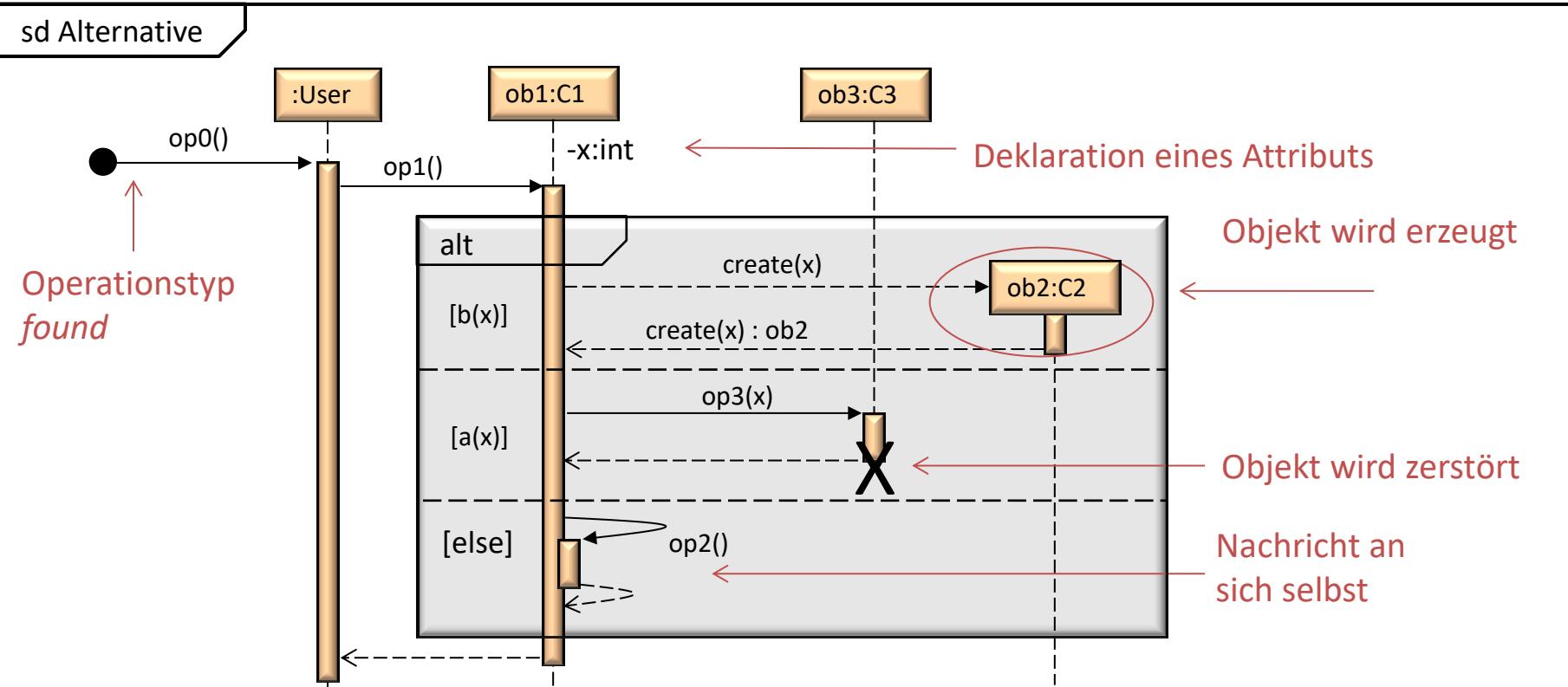


Fragment (Referenz)



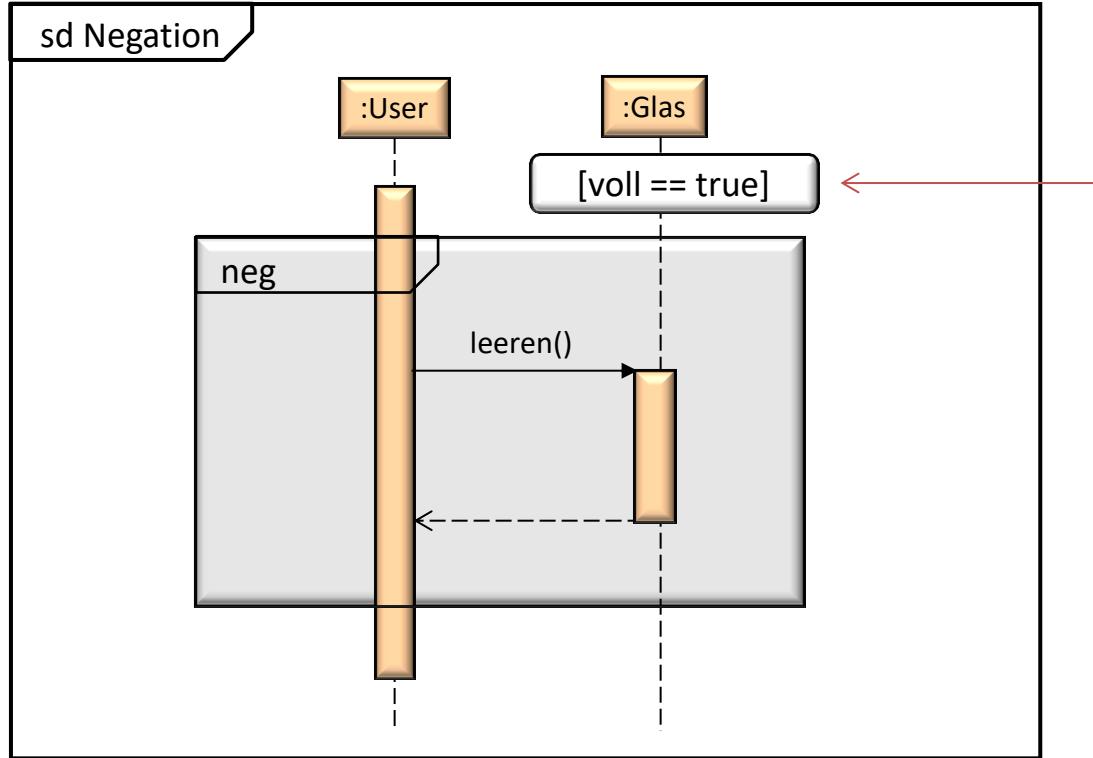


Fragment (Alternative)





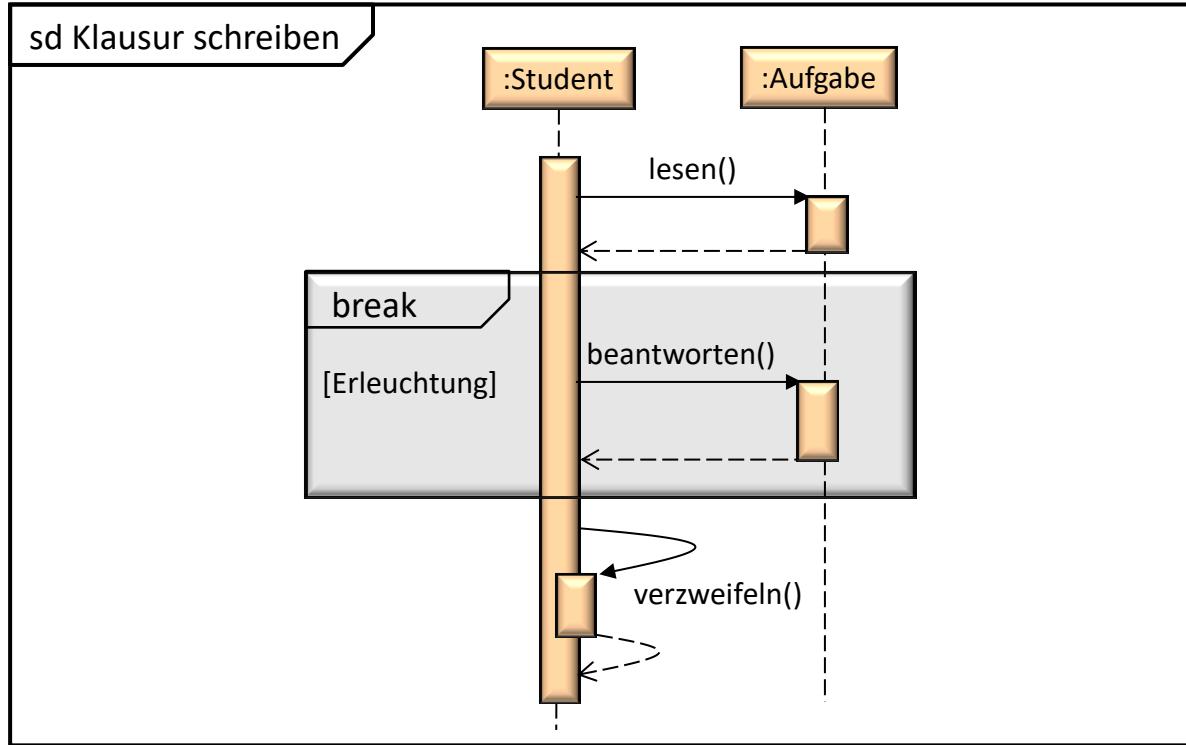
Fragment (Negation)



Zustandsinformation, sie gilt auf der ganzen Lebenslinie.

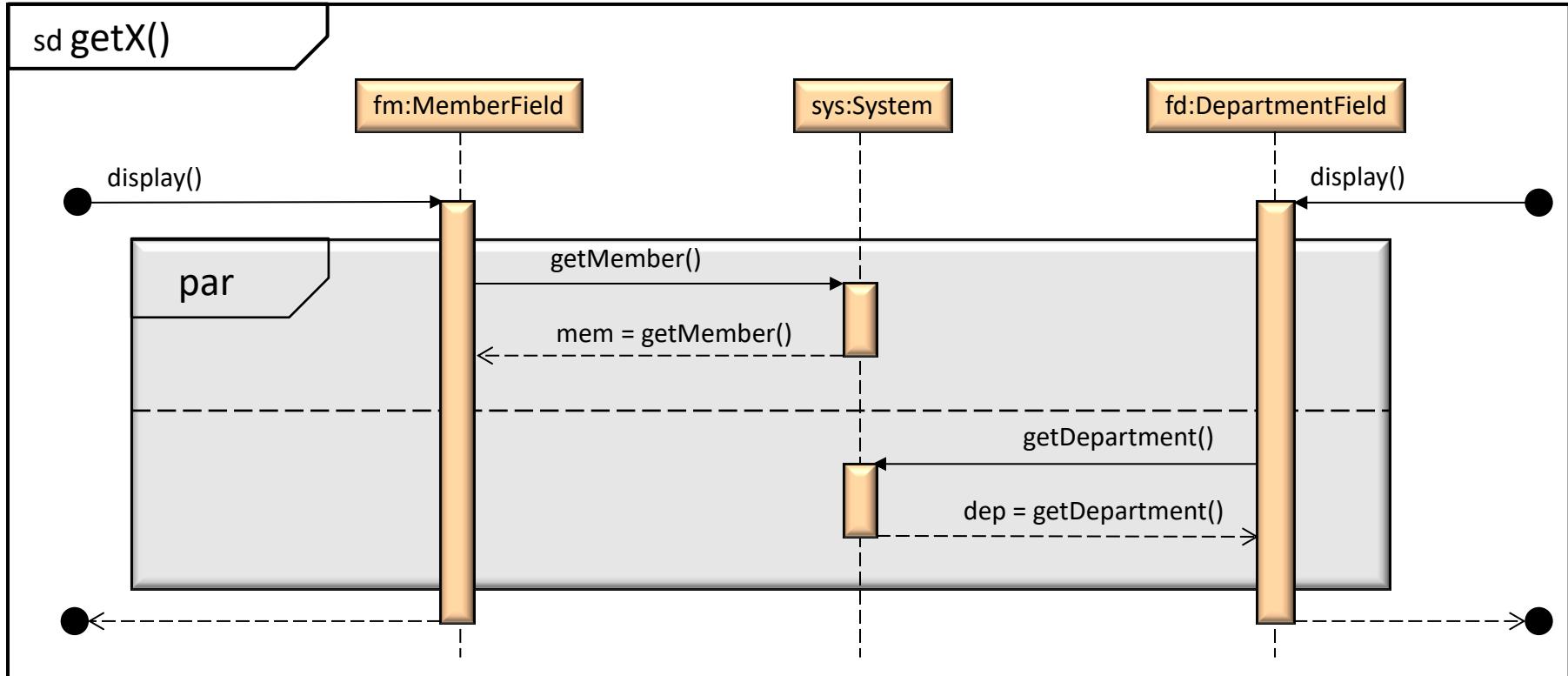


Fragment (Abbruch)



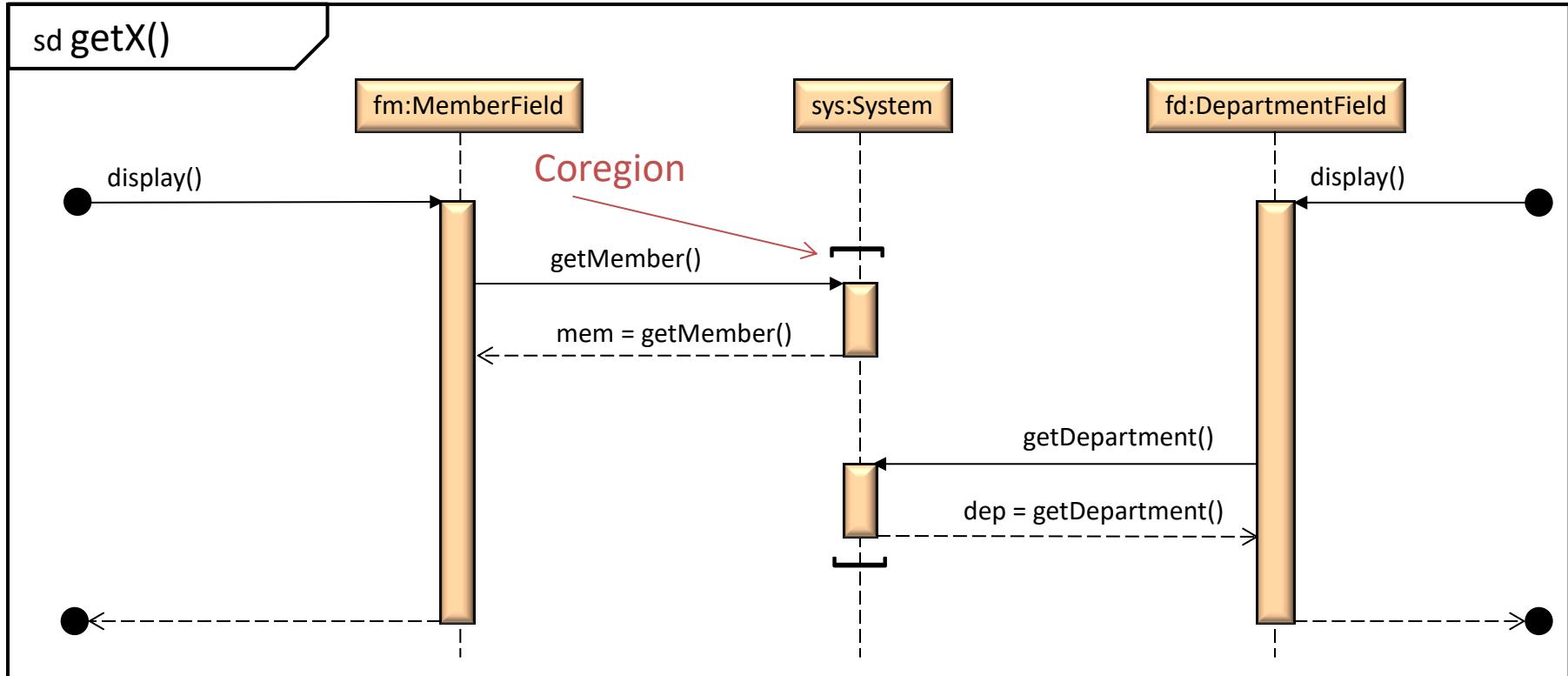


Fragment (Parallel)





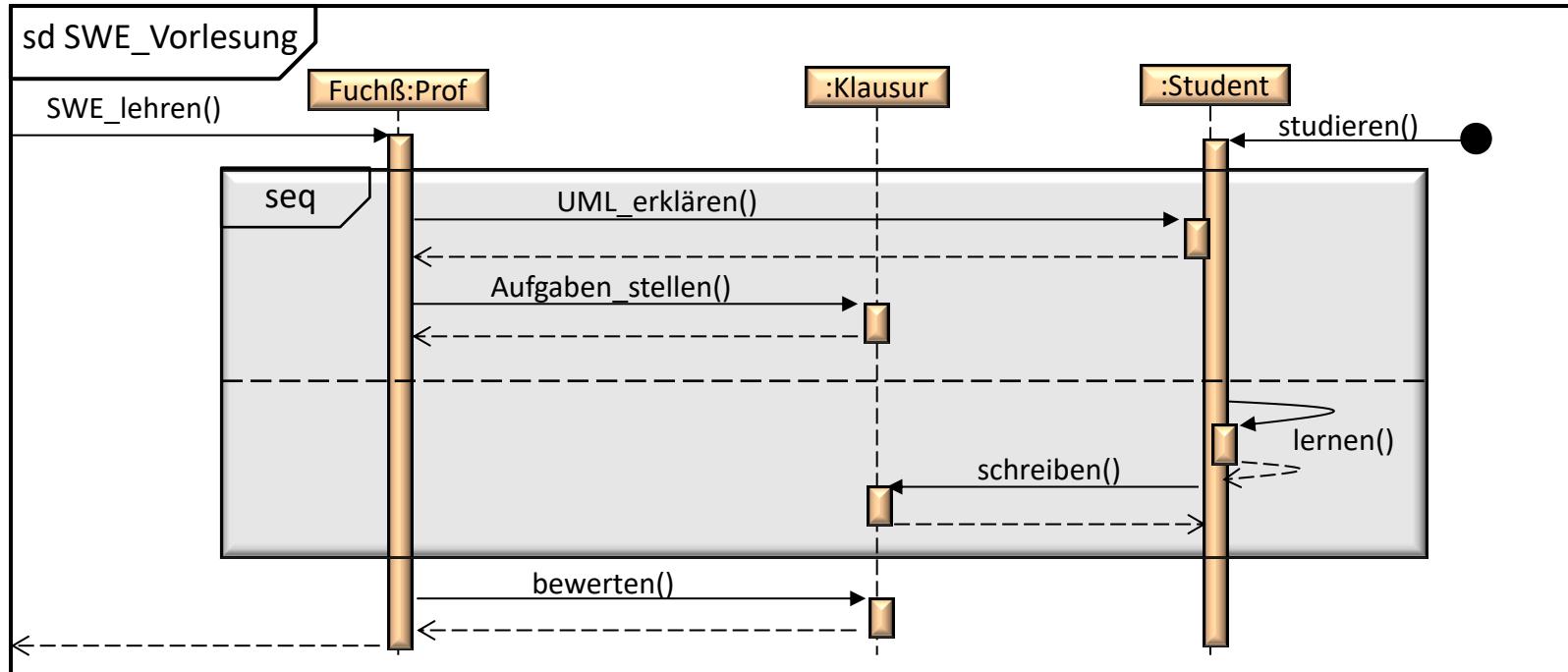
Fragment (Parallel)





Fragment (Ordnung)

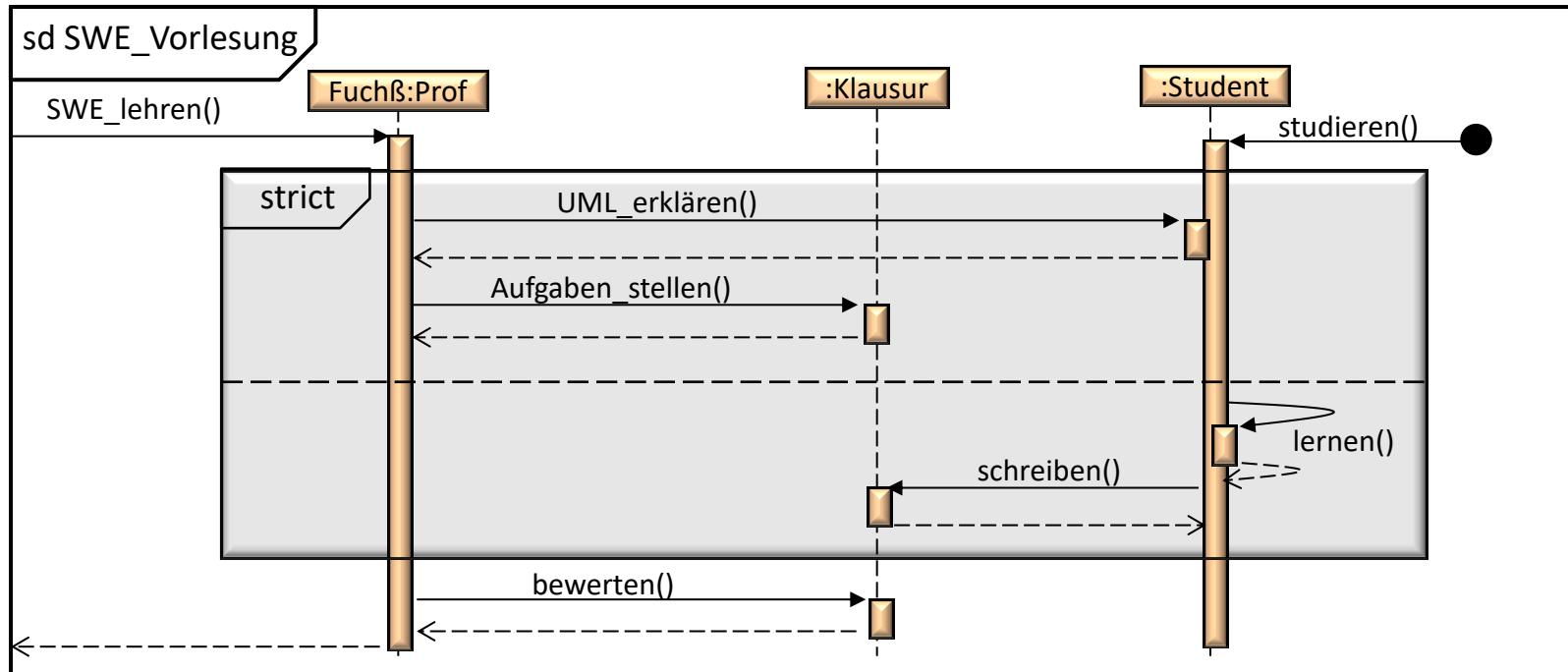
Es gibt keine zeitliche Abhangigkeit zwischen dem Lernen und dem Stellen der Aufgaben.





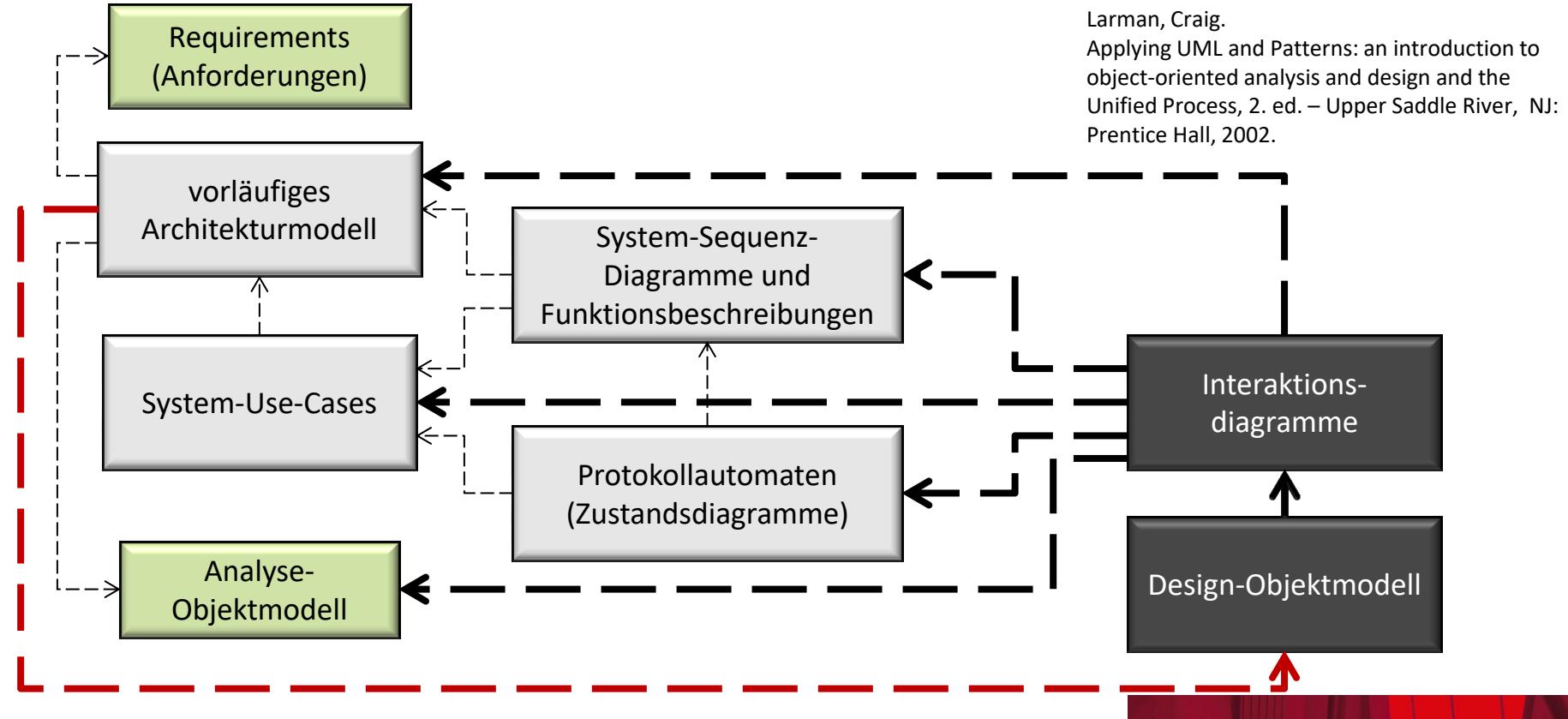
Fragment (Sequenz)

ohne Idealismus



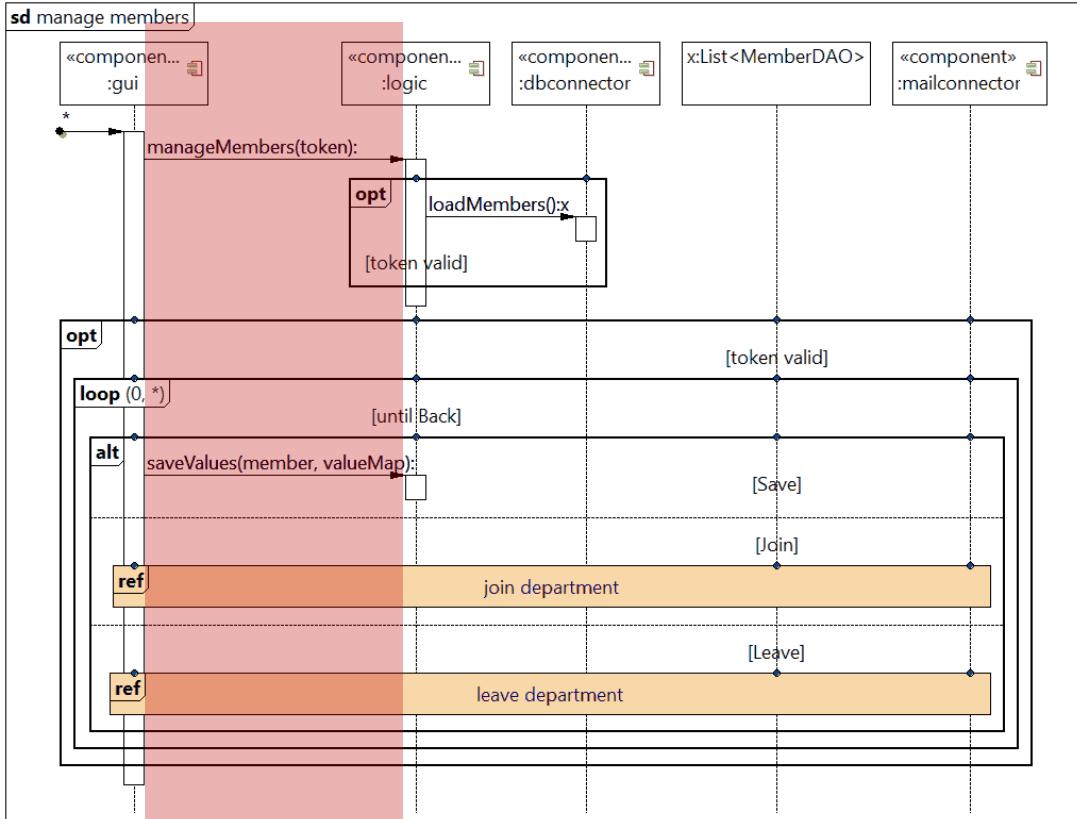


Die nächsten Designschritte





Ausgangspunkt: System-Sequenz-Diagramm



Für jede Systemoperation wird ein Interaktionsdiagramm erstellt und ein „Verantwortlicher“ definiert!

Dies erfolgt top-down von Komponente zu Komponente!



Verantwortliche finden

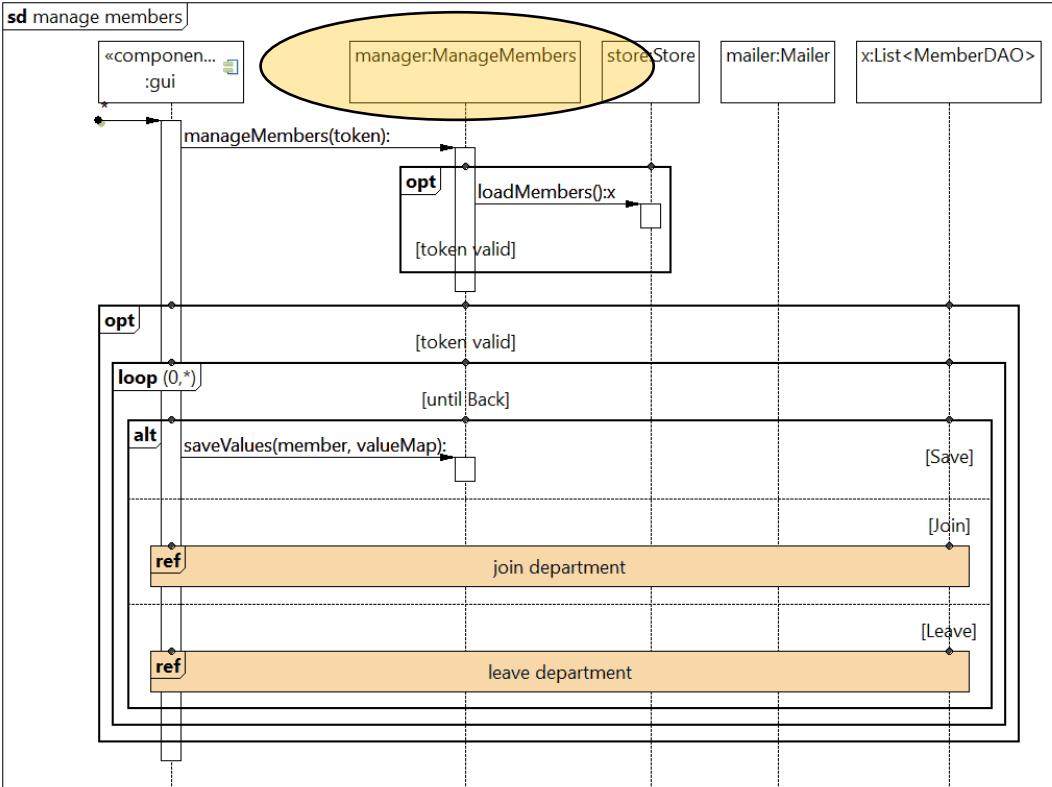
Gemäß dem Managementprinzip gibt es folgende Alternativen:

- MyClub repräsentiert das gesamte System,
- Verein repräsentiert die Organisation,
- Vereinsmanager repräsentiert den zentralen Akteur,
- ManagerImpl repräsentiert den Port,
- ManageMembers **repräsentiert den Use Case, um den es geht!**

**Ein Manager für alle Operationen
eines Use Cases.**



Use-Case-Manager



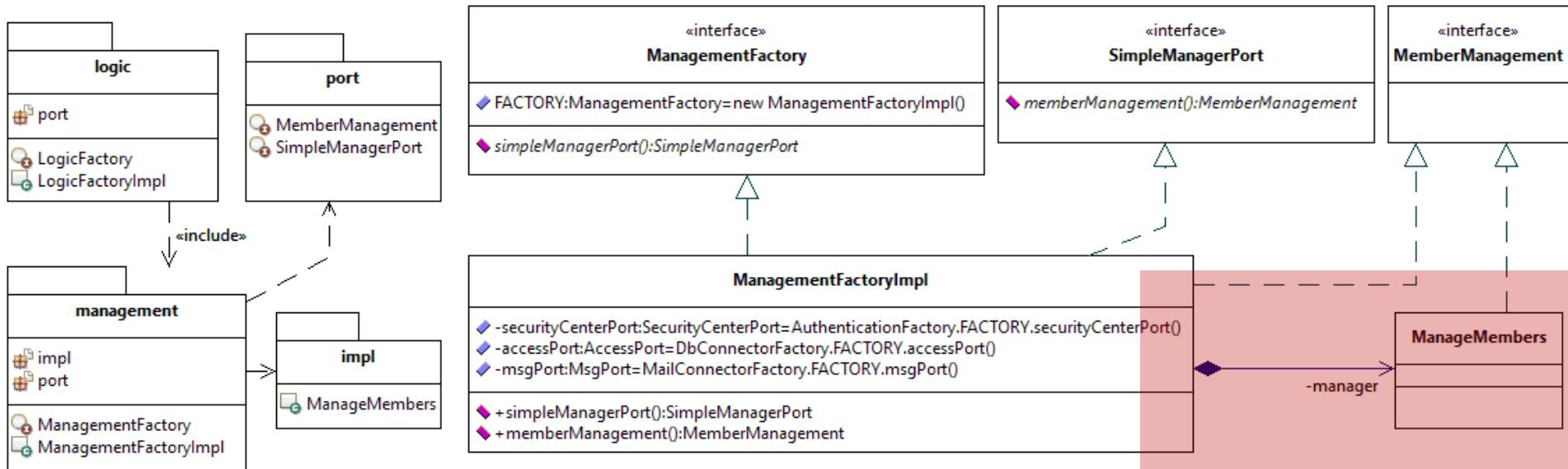
ManageMembers

- naheliegend,
- leicht zu identifizieren,
- unverwechselbar!



Ports als Zugriffspunkte (Folie 231)

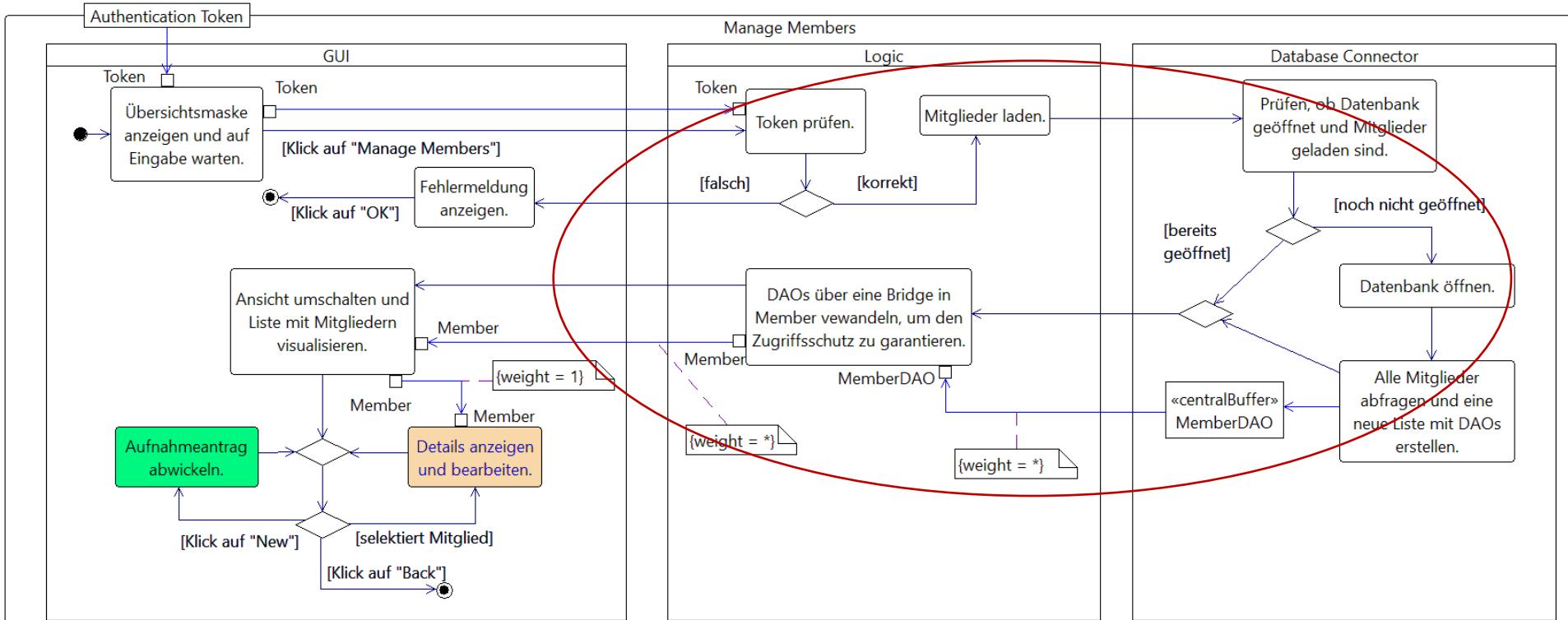
- Der Zugriff auf das Modell (die Realisierung des Interface MemberManagement)



Der Use Case ist isoliert.

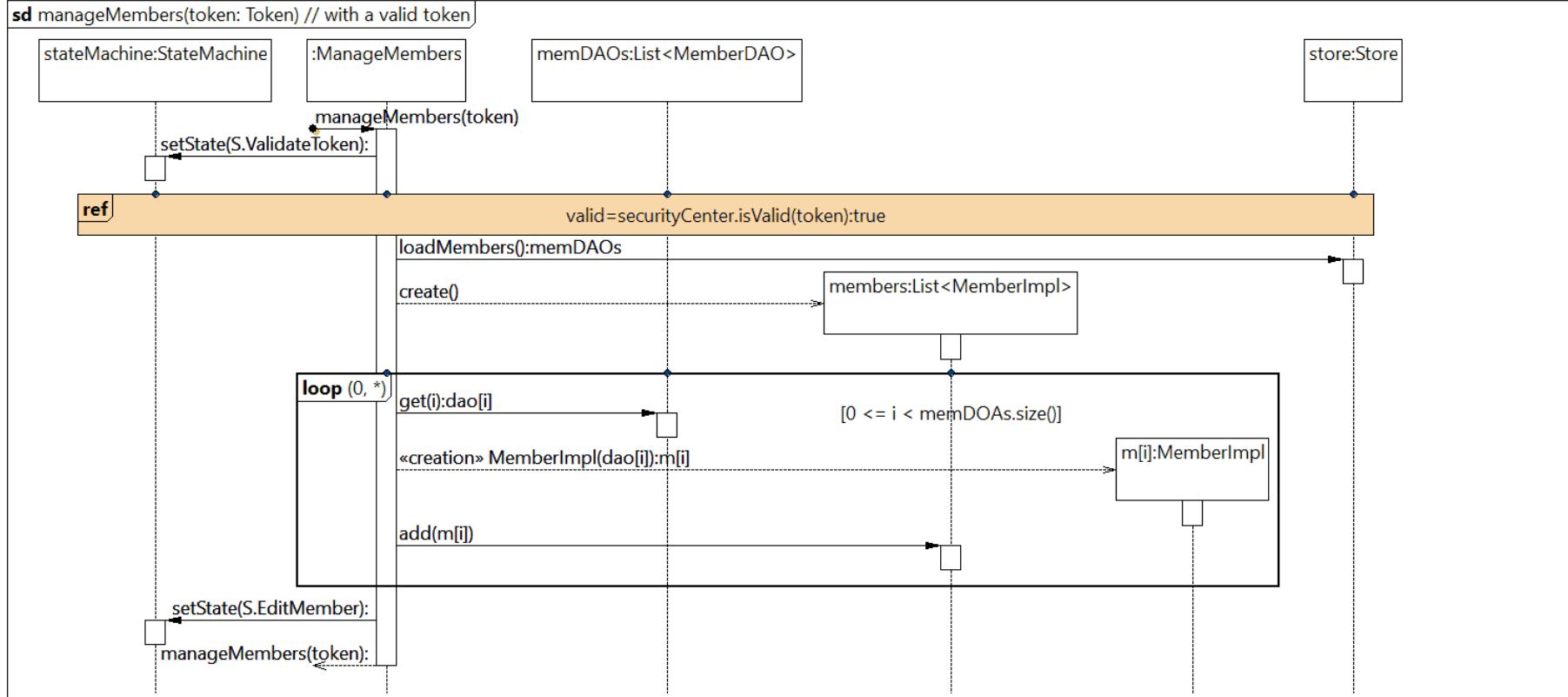


MyClub: Use Case „Manage Members Teil 1“



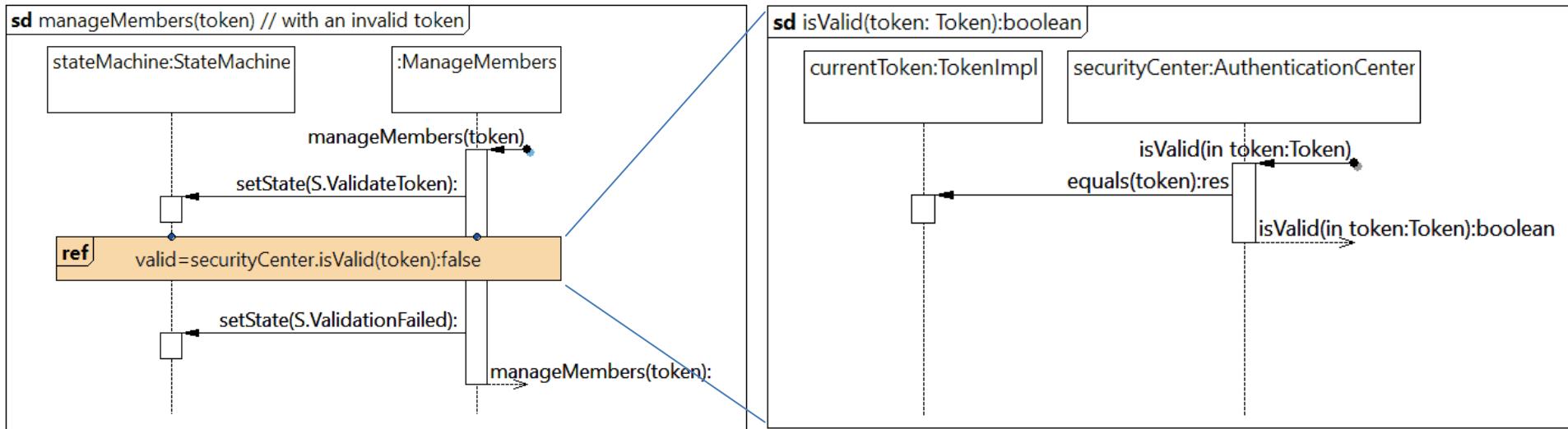


MyClub: Manage Members „manageMembers()“



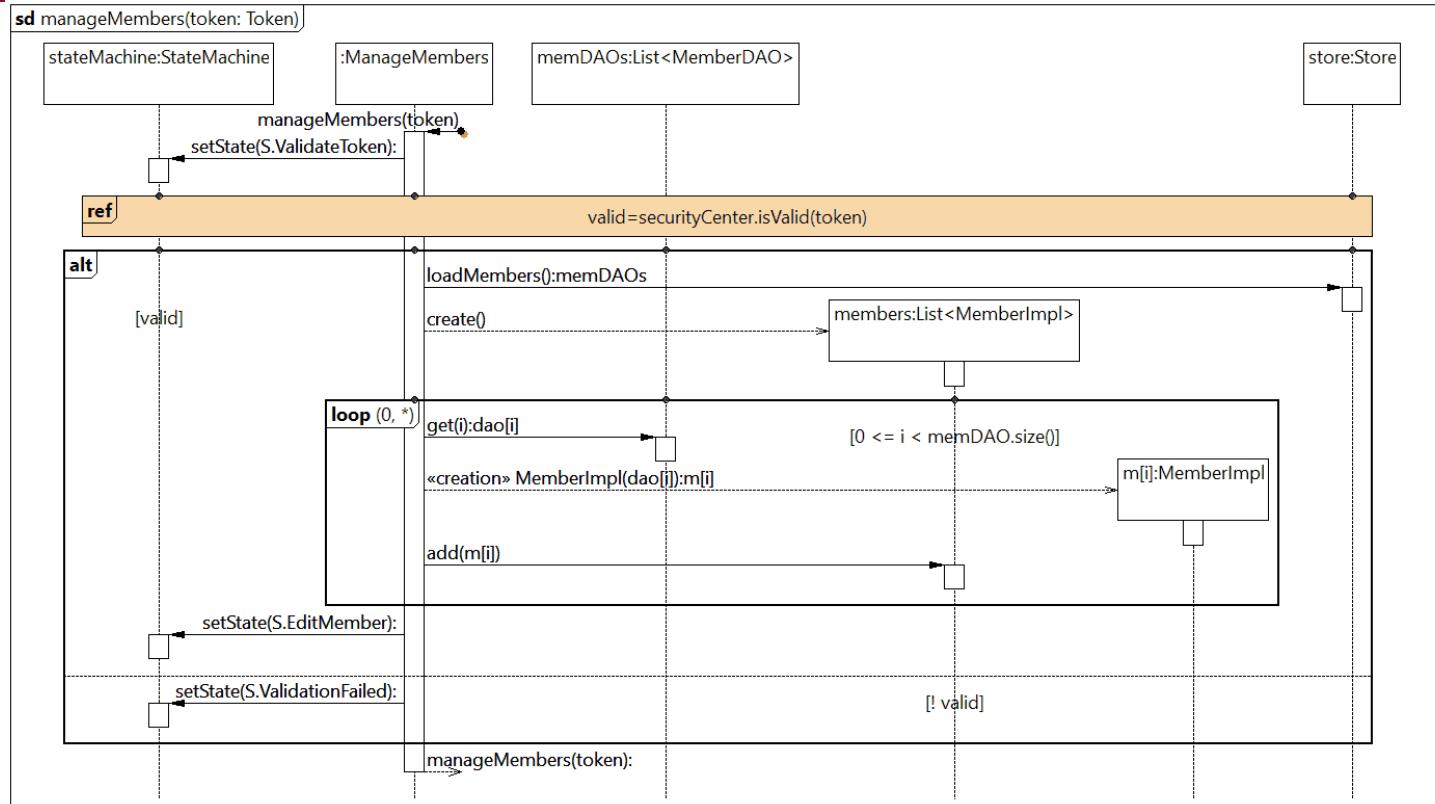


MyClub: Manage Members „manageMembers()“





MyClub: Manage Members „manageMembers()“





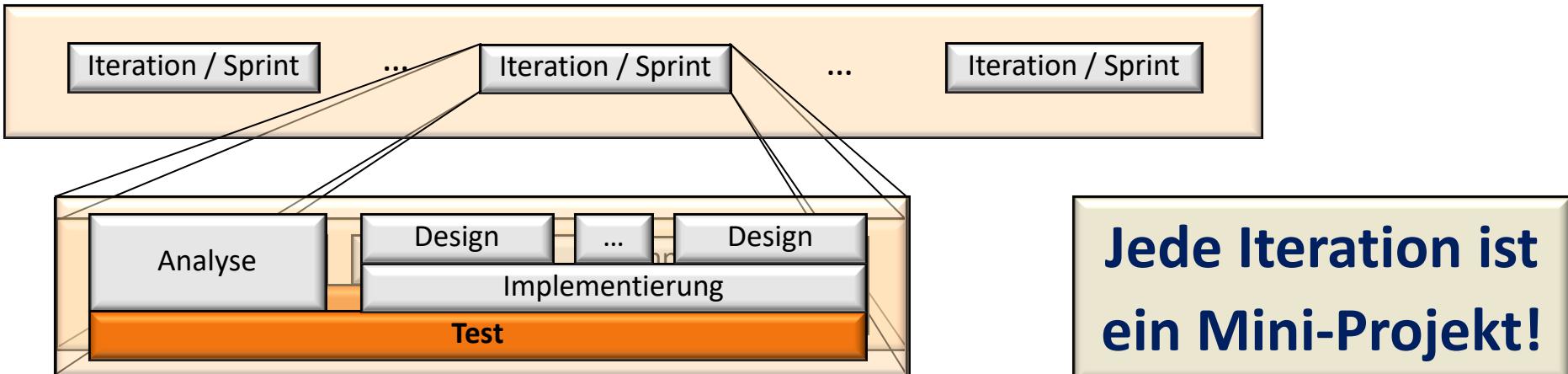
Die nächsten Schritte

- **Den Use Case fertigstellen**
 - Designs für die verbleibenden System-Operationen erstellen
 - gemäß der Architektur von Komponente zu Komponente fortschreiten
- **Den zweiten (für die aktuelle Iteration) geplanten Use Case angehen**
 - Activity-Diagramme erstellen
 - neue System-Operationen finden
 - Mockups anpassen
 - Zustände anpassen
 - Architektur anpassen
 - System-Operationen designen
- **Gemäß Design die Funktionalität implemetieren!**

Dies ist nicht
unbedingt eine
gute Idee!



Gedanken zur Implementierung



Design und Implementierung sind zwei kaum trennbare Aufgaben!

Dann sollte man sie vielleicht auch nicht trennen!



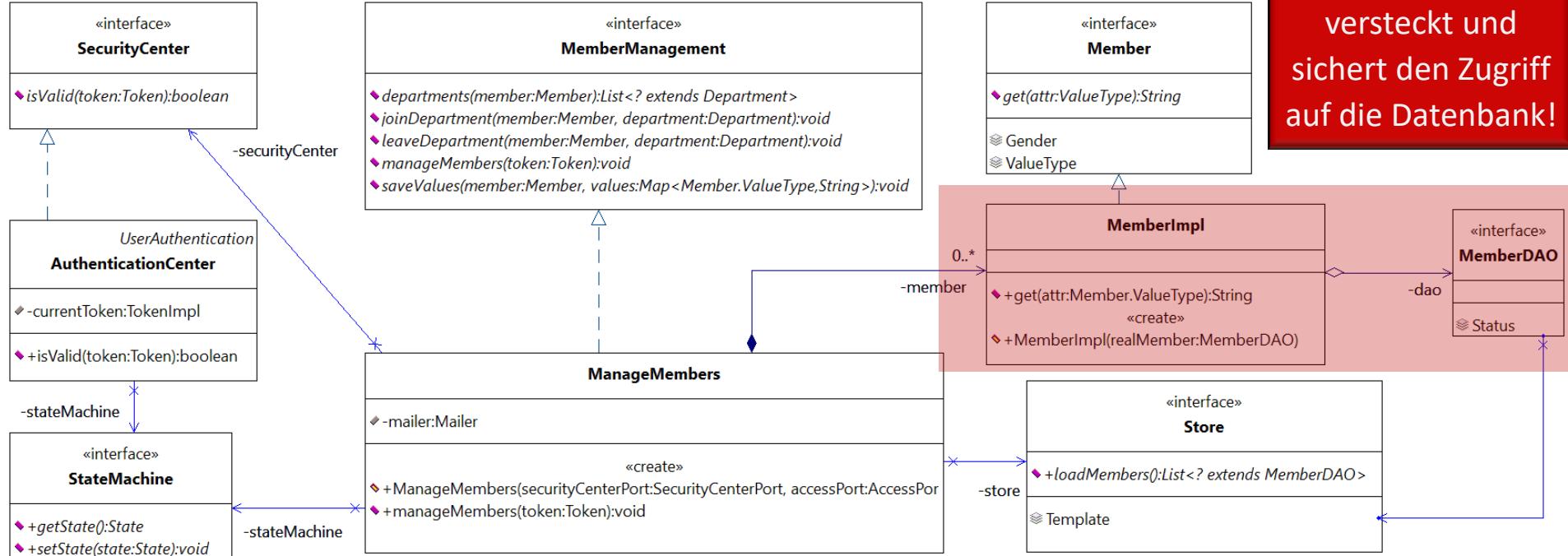
Die nächsten Schritte

- **gemäß Design die bisherige Funktionalität implementieren**
 - Objektmodelle entwerfen
 - Erfahrungen sammeln
 - Implementierungen testen
 - Fehler korrigieren
- **den Use Case fertigstellen**
 - Designs für die verbleibenden System-Operationen erstellen
 - gemäß der Architektur von Komponente zu Komponente fortschreiten
- unter Einbeziehung des Kunden das Ergebnis validieren
- entweder mit dem Design des zweiten, noch offenen Use Case fortfahren, oder die nächsten Iteration beginnen und die nächsten Use Cases analysieren.

Schnell Fehler finden und korrigieren ist wichtiger als ein vollständiges Design!



MyClub: Manage Members – die ersten Objekte





MyClub: Manage Members – die ersten Objekte

```
public class ManageMembers implements MemberManagement {  
  
    private Store store;  
    private Mailer mailer;  
    private StateMachine stateMachine;  
    private SecurityCenter securityCenter;  
  
    private List<MemberImpl> members;  
  
    public ManageMembers(SecurityCenterPort securityCenterPort,...){  
        this.securityCenter = securityCenterPort.securityCenter();  
        this.store = accessPort.store();  
        this.mailer = msgPort.mailer();  
        this.stateMachine = stateMachinePort.stateMachine();  
    }  
}
```

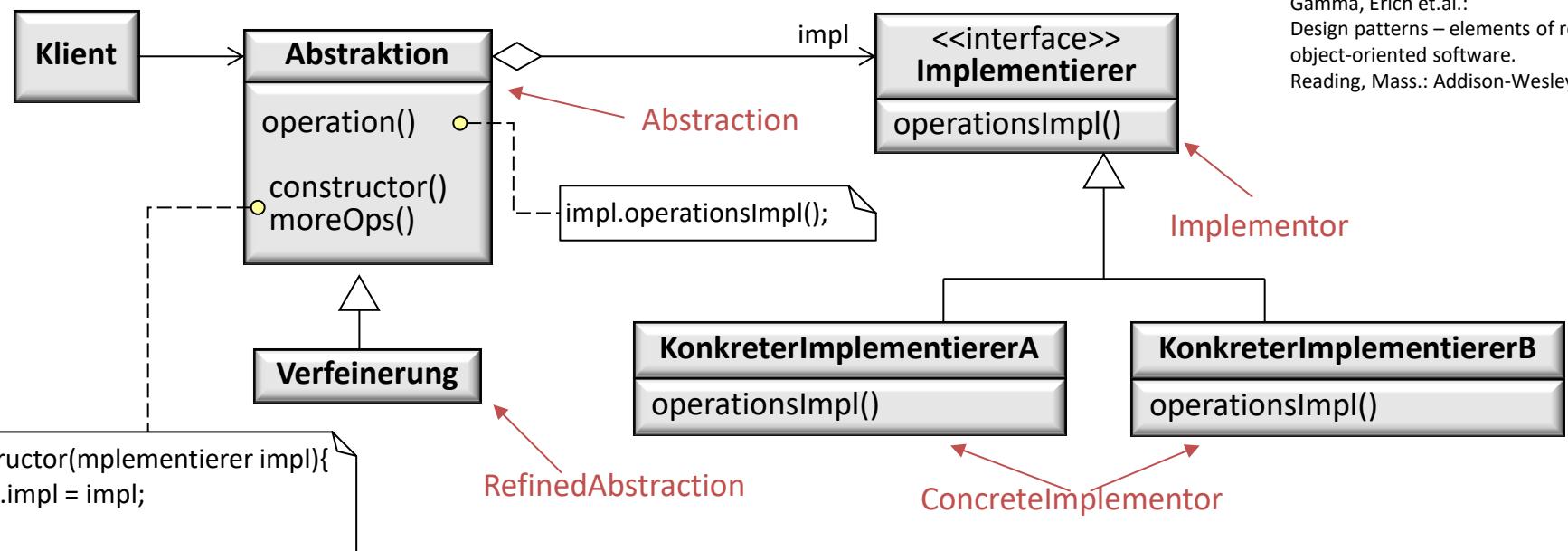
```
public void manageMembers(Token token) {  
    this.stateMachine.setState(State.S.ValidateToken);  
    if (!this.securityCenter.isValid(token)) {  
        this.stateMachine.setState(State.S.ValidationFailed);  
        return;  
    }  
    this.members = new LinkedList<>();  
    this.store.loadMembers()  
        .forEach(dao ->this.members.add(new MemberImpl(dao)));  
    this.stateMachine.setState(State.S.EditMember);  
}
```

Die Bridge wird
aufgebaut.



Das Strukturmuster der Brücke

Eine Brücke (Bridge) entkoppelt Abstraktion und Implementierung. Dies ermöglicht es Implementierungen unabhängig von der Abstraktion zu variieren. Die Abstraktion definiert und implementiert das Interface, während die eigentliche Implementierung ausgelagert wird. Alle Operationen rufen entsprechende Methoden in der Implementierung auf.



Gamma, Erich et.al.:
Design patterns – elements of reusable
object-oriented software.
Reading, Mass.: Addison-Wesley, 1995



Anwendbarkeit

- **Man verwendet das Brückenmuster, wenn:**
 - man eine permanente Bindung zwischen Abstraktion und Implementierung vermeiden möchte,
 - sowohl Abstraktion als auch die Implementierung unabhängig variiert werden sollen,
 - die Änderung der Implementierung keinen Einfluss auf den Klient haben soll,
 - die Implementierung vollständig versteckt werden soll.

```
Abstraktion x = new Abstraktion(new KonkreteImplA());  
((KonkreteImplA) x).operationsImpl();
```

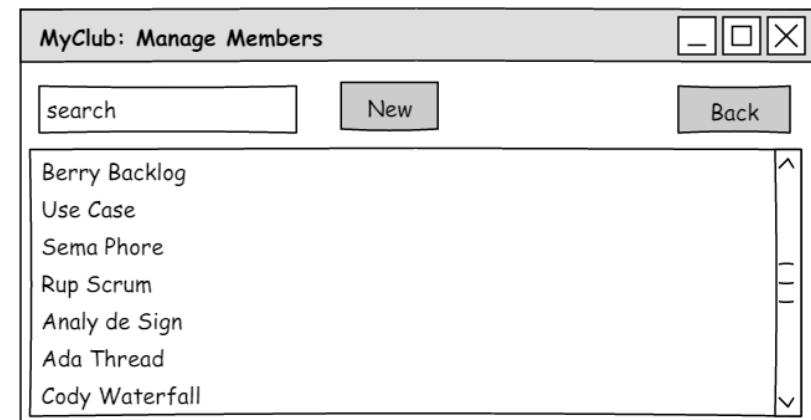
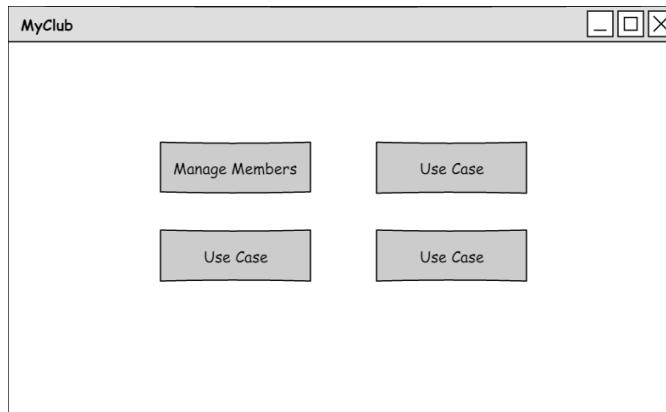
Ein Zugriff auf die Implementierung ist nichtmehr so einfach möglich!



Views und Controller implementieren

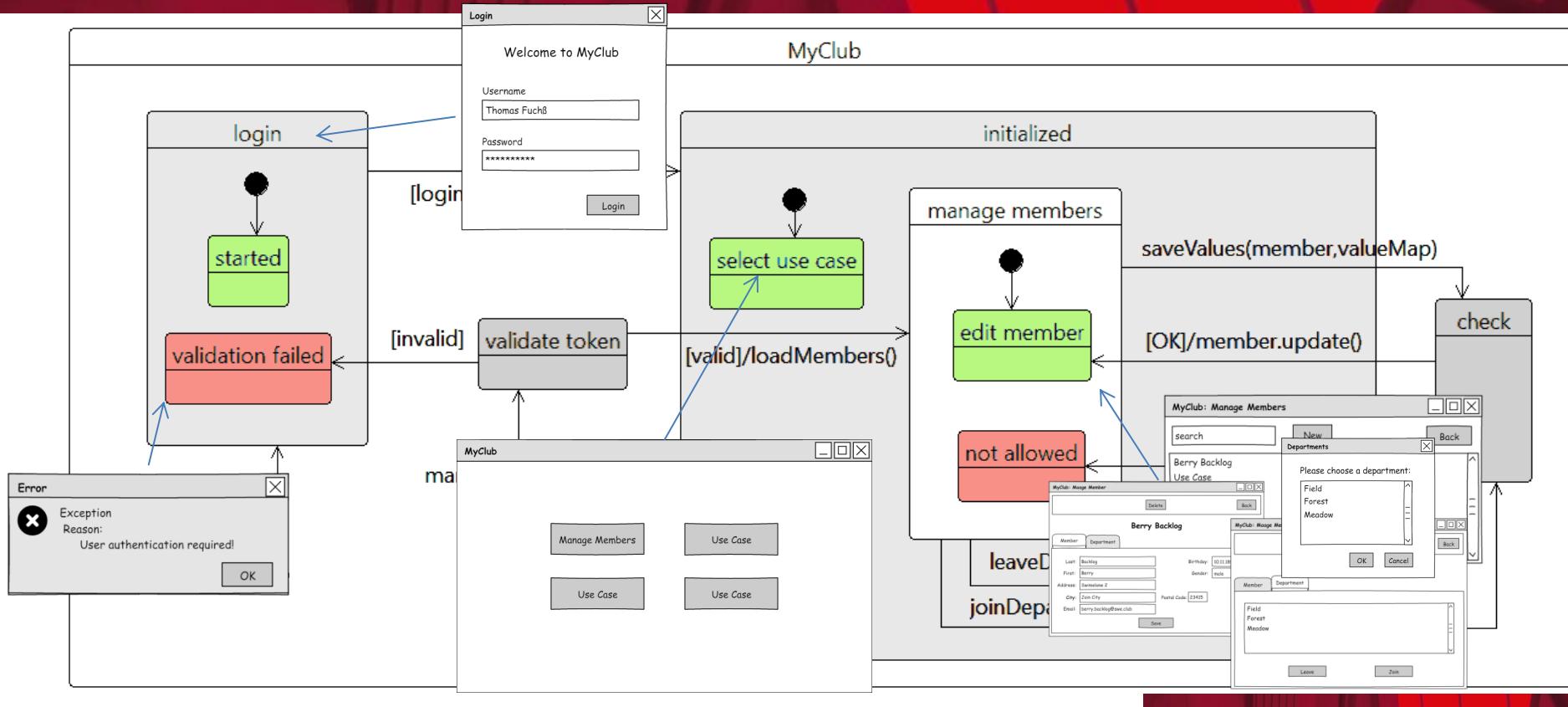
Auch bei der Oberfläche orientiert man sich an den Use Cases.

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



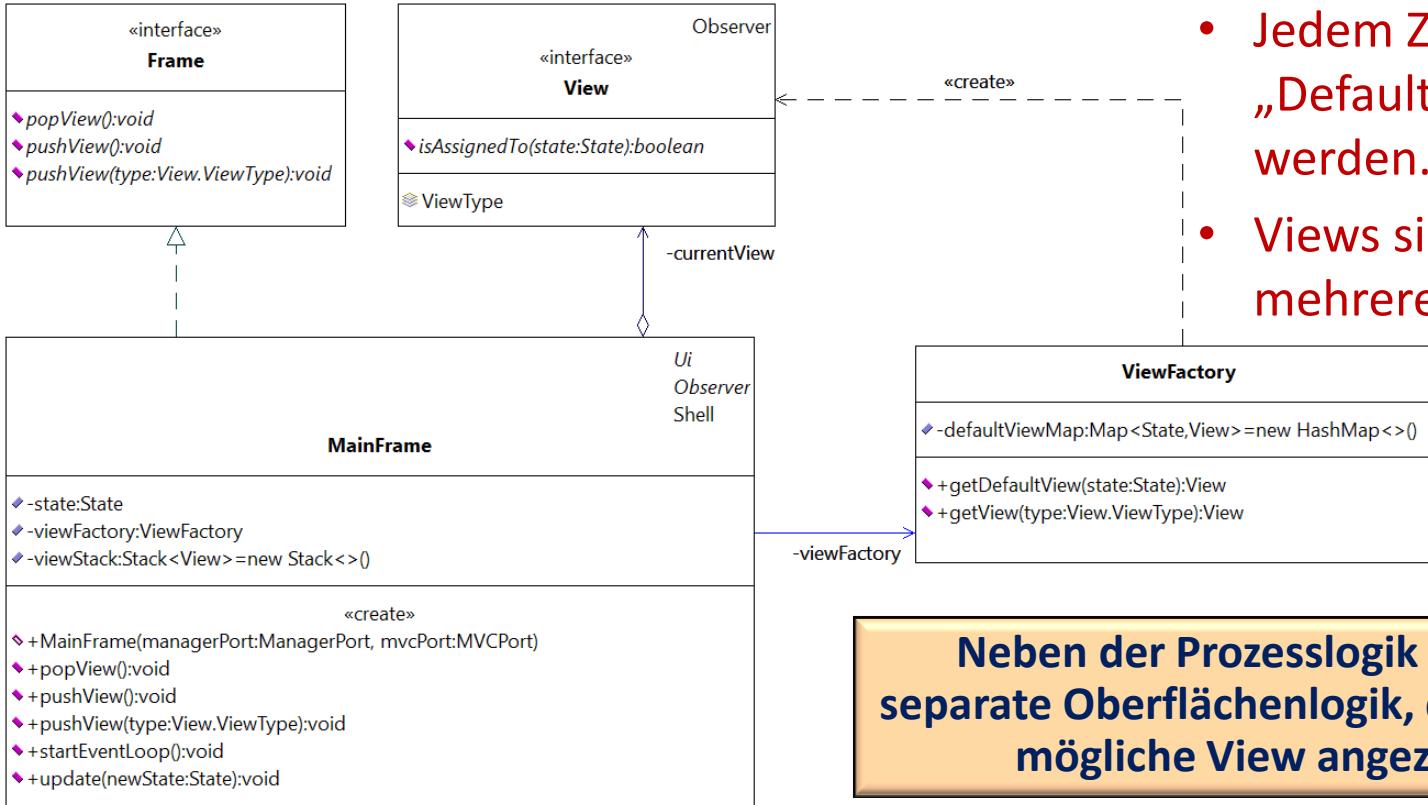


Zustände organisieren die GUI





Zustände organisieren die GUI

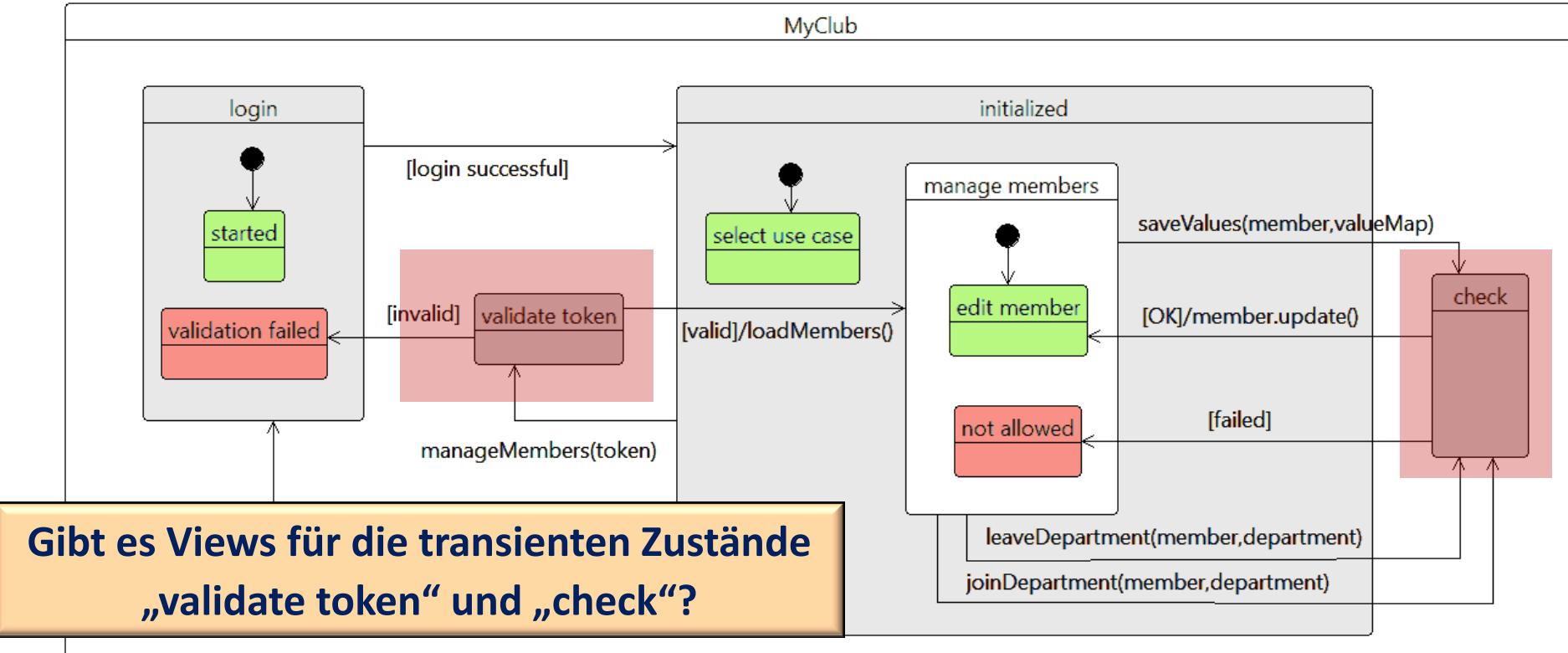


- Jedem Zustand sollte ein „Default-View“ zugewiesen werden.
- Views sind eventuell in mehreren Zuständen sinnvoll.

Neben der Prozesslogik gibt es oftmals eine separate Oberflächenlogik, die entscheidet, welche mögliche View angezeigt werden soll.



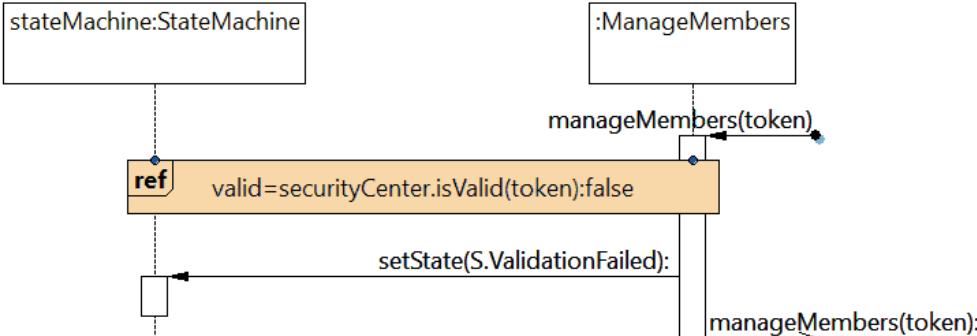
Zustände organisieren die GUI





Nicht jeder Zustand ist relevant.

```
sd manageMembers(token) // with an invalid token
```



```
public void manageMembers(Token token) {
    this.stateMachine.setState(State.S.ValidateToken);
    if (!this.securityCenter.isValid(token)) {
        this.stateMachine.setState(State.S.ValidationFailed);
        return;
    }
    this.members = new LinkedList<>();
    this.store.loadMembers().forEach(dao ->this.members.add(new MemberImpl(dao)));
    this.stateMachine.setState(State.S.EditMember);
}
```

```
public enum S implements State {
    Started, ValidationFailed,
    Login(Started, ValidationFailed),
    SelectUseCase, EditMember, NotAllowed,
    ManageMembers(EditMember, NotAllowed),
    Initialized(SelectUseCase, ManageMembers);
    ValidateToken, Check;
    private List<State> subStates;

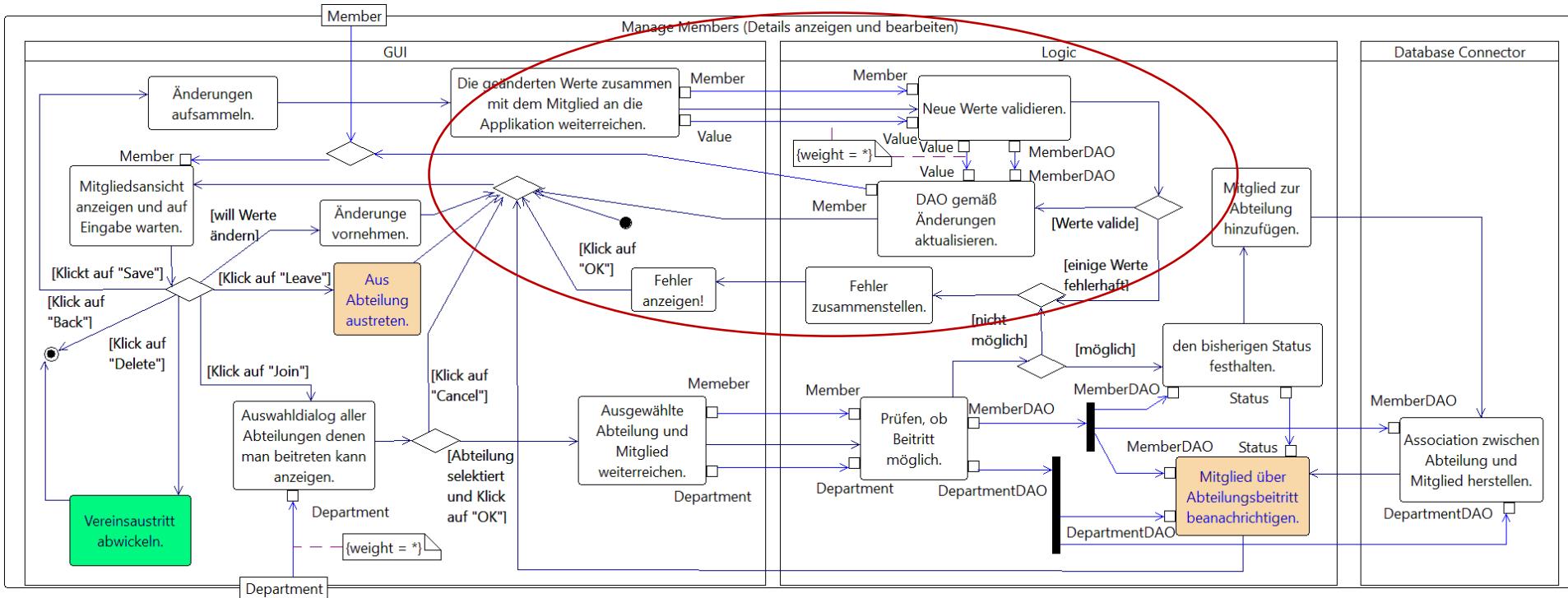
    private S(State... subS) {
        this.subStates =
            new ArrayList<>(Arrays.asList(subS));
    }
}
```

Was keiner sieht, das braucht man

nicht!

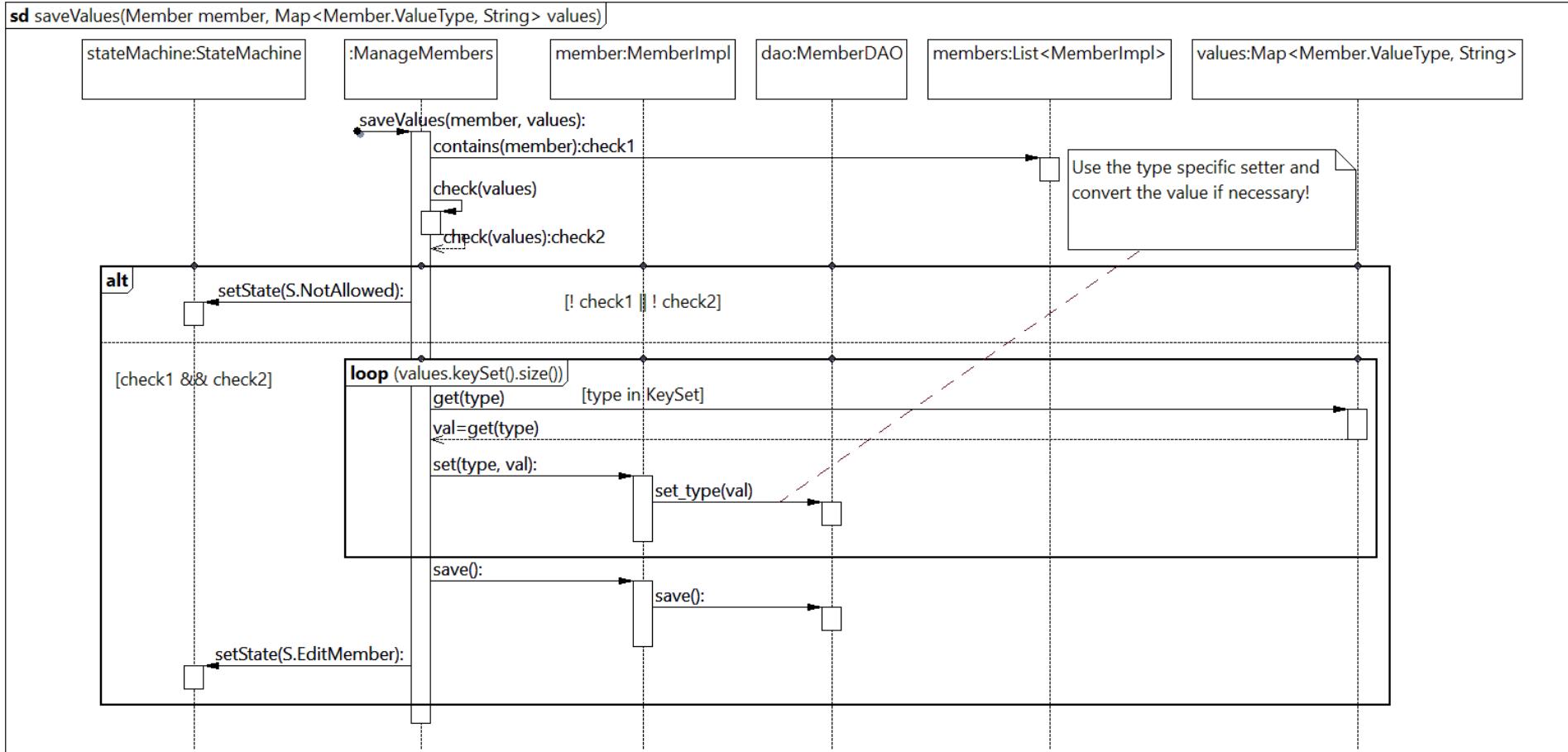


MyClub: Use Case „Manage Members“ (Schritt 2)





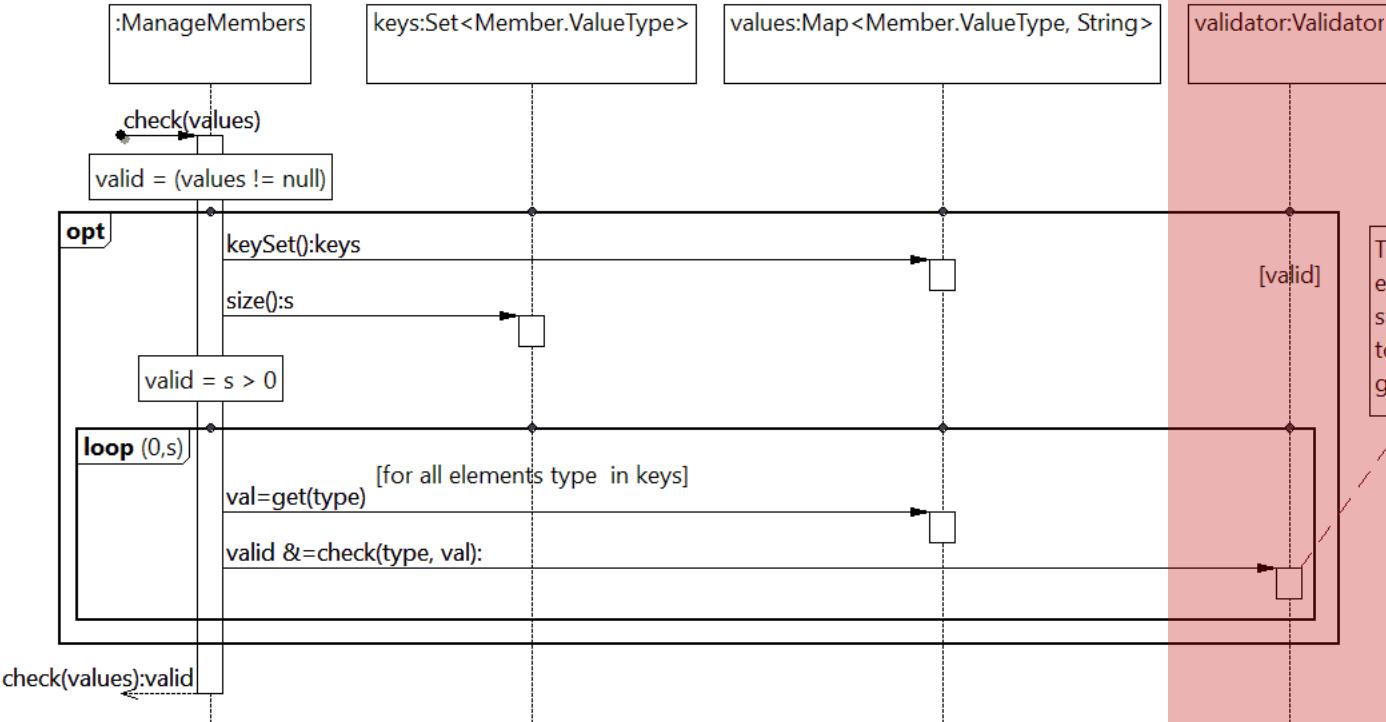
MyClub: Manage Members „saveValues()“





MyClub: Manage Members „saveValues()“

sd check(Map<Member.ValueType, String> values)



Aufgaben werden delegiert und neue Objekte werden gebraucht!



MyClub: Manage Members „saveValues()“

```
class ManagementFactoryImpl implements ManagementFactory, SimpleManagerPort, MemberManagement {  
    /* ... */  
  
    public synchronized void saveValues(Member member, Map<Member.ValueType, String> values) {  
        if (!this.stateMachine.getState().isSubStateOf(State.S.ManageMembers))  
            return;  
        this.manager.saveValues(member, values);  
    }  
}
```

Fassade erweitern!

```
public void saveValues(Member member, Map<Member.ValueType, String> values) {  
    if (!this.check(member) || !this.check(values)) {  
        this.stateMachine.setState(State.S.NotAllowed);  
        return;  
    }  
    values.keySet().forEach(type -> this.currentMember.set(type, values.get(type)));  
    this.currentMember.save();  
    this.stateMachine.setState(State.S.EditMember);  
}
```

Implementierung nachziehen!



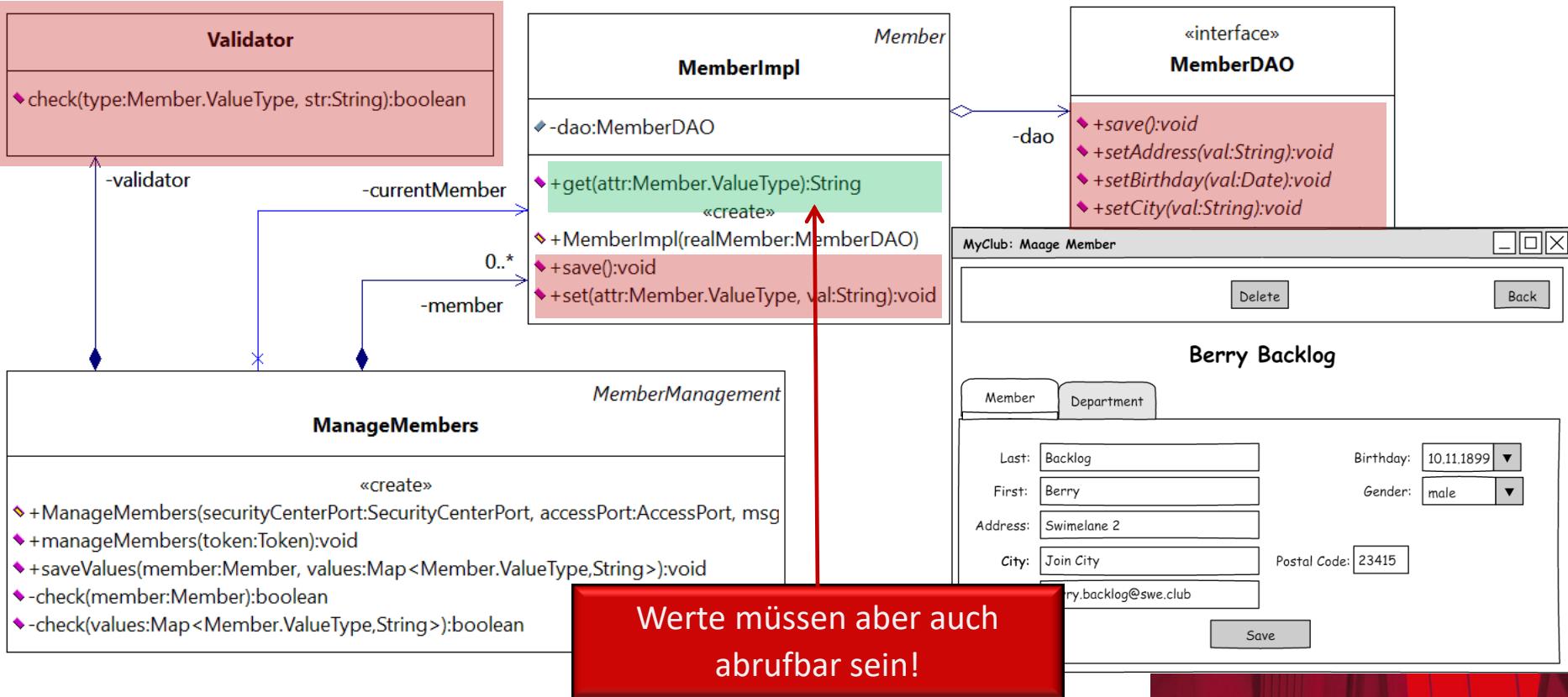
MyClub: Manage Members „saveValues()“

```
private boolean check(Map<Member.ValueType, String> values) {  
    boolean valid = (values != null);  
    if (valid) {  
        valid = values.keySet().size() > 0;  
        for (Member.ValueType type : values.keySet())  
            valid &= this.validator.check(type, values.get(type));  
    }  
    return valid;  
}  
  
private boolean check(Member member) {  
    return (this.currentMember = this.members.contains(member) ? (MemberImpl) member : null) != null;  
}
```

Aufgaben werden
delegiert und neue
Objekte werden
gebraucht!



MyClub: Neue Objekte und Operationen





Anbindung der Speicherverwaltung

Eine simple Idee: Ein OR-Mapper,

- der Tabellen einer Datenbank auf Klassen und Zeilen auf Objekte abbildet,
- der die erzeugten Objekte selbständig verwaltet,
- der für die gängigsten Datenbanksysteme frei verfügbar ist,
- der auf Standards setzt, wie etwa die Java Persistence API,
- ...

Typischerweise verwalten Frameworks wie Hibernate die erzeugten Objekte, im Rahmen eines Session-Kontextes, selbst. Um Probleme mit Locks und wechselseitigem Zugriff zu vermeiden, sollte sichergestellt sein, dass auf die Datenbank nicht aus unterschiedlichen Threads heraus zugegriffen wird.



Anbindung der Speicherverwaltung

Lösung:

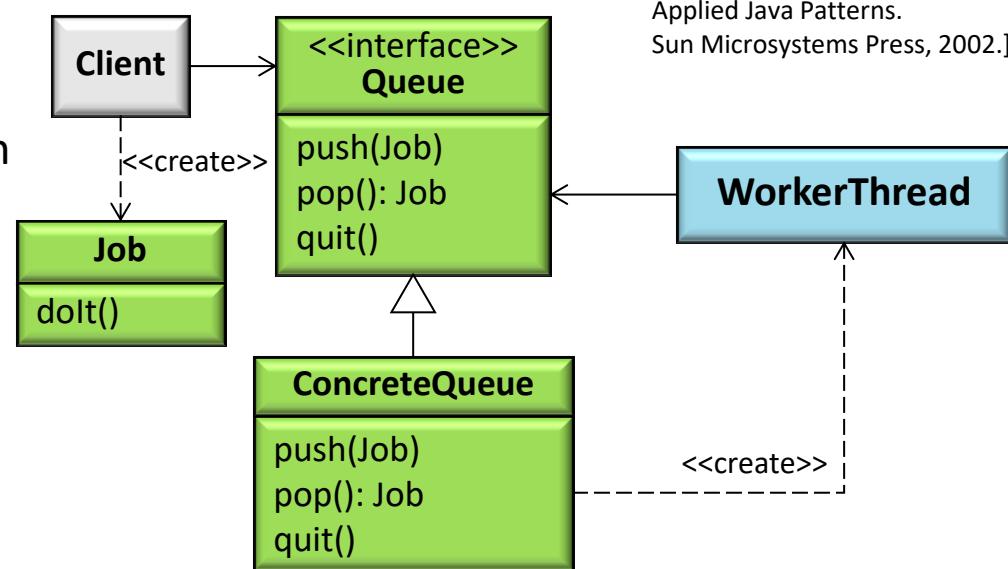
- Eine Worker-Queue, die die Datenbankanfragen als Jobs abarbeitet.
- DAO-Objekte, die Persistence Entities im Sinne der **Java Persistence API** darstellen und sich selbst speichern können.
- Ein generischer Loader, der beliebige DAO-Objekte laden kann.
- Eine zentrale Management-Klasse (Store/StoreImpl), die komplexe Aufgaben übernimmt.



Das Worker-Thread-Muster

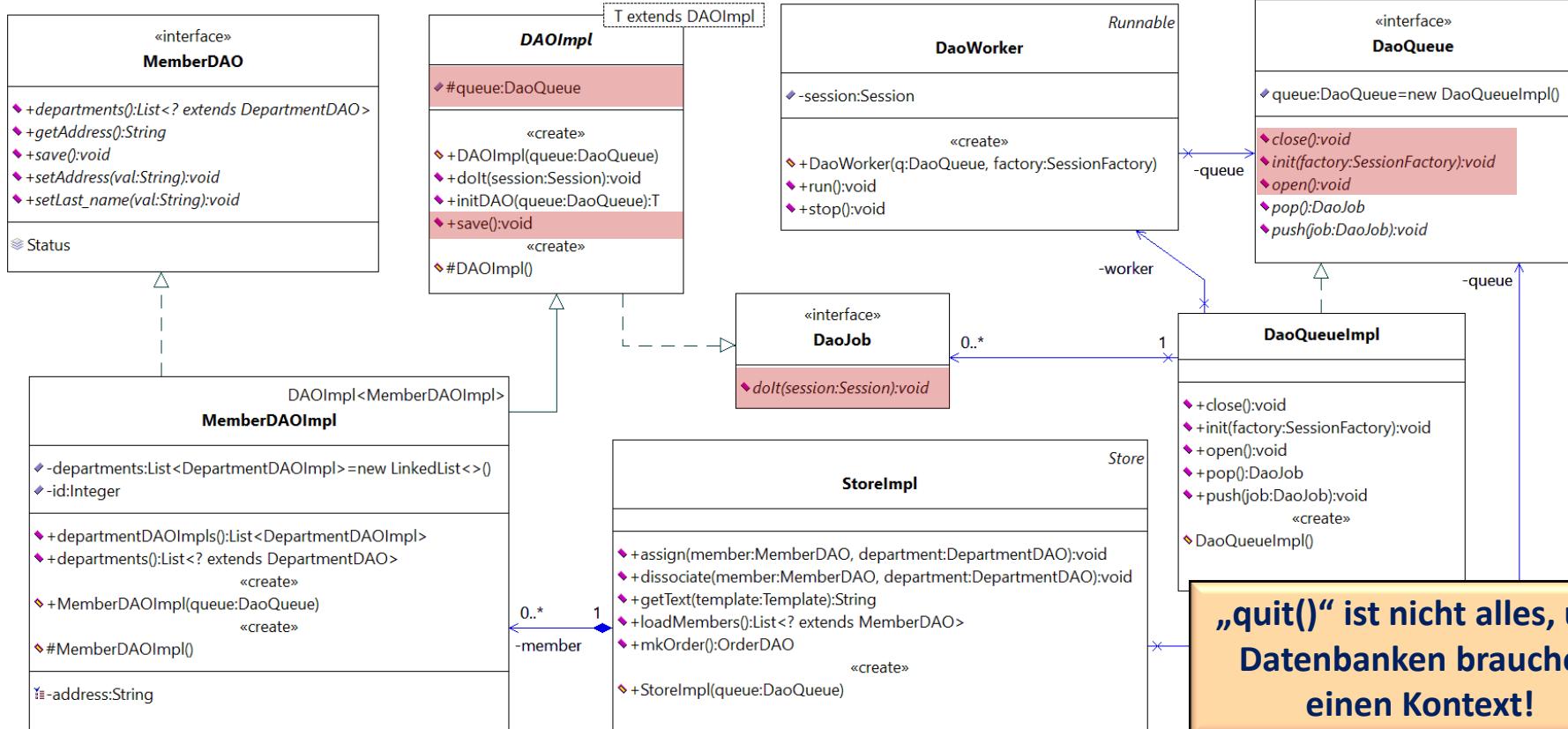
Das Muster sorgt für eine Leistungssteigerung durch die Entkopplung und Parallelisierung von Auftraggeber und Auftragnehmer und garantiert eine einheitliche Ausführungsumgebung.

- Der Klient ist der Auftraggeber.
- Jobs sind die Aufgaben, die parallel zu den übrigen Aufgaben den Auftraggebers abgearbeitet werden sollen.
- Der Thread ist der Auftragnehmer, die definierte Ausführungsumgebung, in der die Aufgaben nacheinander abgearbeitet werden.





MyClub: Anbindung der Speicherverwaltung





MyClub: Worker-Queue

```
public class DaoQueueImpl implements DaoQueue {  
  
    private SessionFactory sessionFactory;  
    private Semaphore read = new Semaphore(0); ←  
    private Queue<DaoJob> jobList = new LinkedList<>();  
  
    private DaoWorker worker;  
    private Thread thread;  
  
    public void init(SessionFactory factory) {  
        this.sessionFactory = factory;  
    }  
  
    public void close() {  
        if (this.worker != null) this.worker.stop();  
        this.worker = null;  
    }  
  
    public void open() {  
        if (this.worker != null) return;  
        this.worker = new DaoWorker(this, this.sessionFactory);  
        this.thread = new Thread(this.worker);  
        this.thread.start();  
    }
```

Worker und Queue folgen dem
Producer-Consumer-Prinzip!

```
public void push(DaoJob job) {  
    synchronized (this.jobList) {this.jobList.add(job);}  
    this.read.release();}  
  
public DaoJob pop() {  
    this.read.acquireUninterruptibly();  
    synchronized (this.jobList){return this.jobList.poll();}}
```



MyClub: Worker-Queue

```
public class DaoWorker implements Runnable {  
  
    private DaoQueue queue;  
    private boolean running = true;  
    private Semaphore stopped = new Semaphore(0);  
    private Session session;  
  
    public DaoWorker(DaoQueue q, SessionFactory factory) {  
        this.queue = q;  
        this.session = factory.openSession();  
  
    public void stop() {  
        this.queue.push(null);  
        this.stopped.acquireUninterruptibly();  
        this.session.close(); this.session = null;}  
}
```

Anhalten erfolgt synchron und schließt die aktuelle Datenbanksitzung.

```
public void run() {  
    DaoJob job = null;  
    while (this.running) {  
        job = this.queue.pop();  
        if (job == null)  
            break;  
        try{job.doIt(this.session);}  
        catch(Exception e){  
            e.printStackTrace();  
        }  
    }  
    this.stopped.release();  
}
```



MyClub: Worker-Queue

Worker-Queues sind universell!

- Die gleiche Art von Queue kann für die Oberfläche verwendet werden, um Anfragen asynchron an das Modell zu richten.
- Werden mehrere Queues verwendet, lassen sich priorisierte Warteschlangen realisieren.
- Werden mehrere Worker (Worker-Pools / Thread-Pools) eingesetzt, dann kann gezielt parallelisiert werden, wobei der Overhead einer stetigen Thread-Erzeugung vermieden wird.



MyClub: Selbstsichernde DAOs

```
public abstract class DAOImpl<T extends DAOImpl<T>> implements DaoJob {  
  
protected DaoQueue queue;  
  
public DAOImpl(DaoQueue queue) {  
    this.queue = queue;  
    this.save();}  
  
protected DAOImpl() {}  
  
public T initDAO(DaoQueue queue) {  
    this.queue = queue;  
    return (T) this;}  
  
public void doIt(Session session) {  
    Transaction tx = session.beginTransaction();  
    session.saveOrUpdate(this);  
    session.flush();  
    tx.commit();}
```

```
public void save() {  
    this.queue.push(this);  
}
```

Man ist sein
eigener Klient!



Ein generischer Loader

```
public class Loader<T extends DAOImpl<T>> implements DaoJob {  
  
    private Semaphore done = new Semaphore(0);  
    private Class<T> clazz;  
    private BiFunction<CriteriaBuilder, Root<T>, Predicate> predicate;  
    private DaoQueue queue;  
    private List<? extends T> result;  
  
    public Loader(  
        Class<T> clazz,  
        BiFunction<CriteriaBuilder, Root<T>, Predicate> predicate,  
        DaoQueue queue) {  
        this.predicate = predicate;  
        this.queue = queue;  
        this.clazz = clazz;  
        this.queue.push(this);  
  
        public void doIt(Session session) {  
            try { if (this.predicate == null) this.all(session);  
                  else this.some(session);  
                  this.result.forEach(dao -> dao.initDAO(this.queue));}  
            finally {this.done.release();}}  
    }  
}
```

Der Auftrag wird im Konstruktor erteilt.

Die SQL-Query als Prädikat!

```
public List<? extends T> result() {  
    try {this.done.acquireUninterruptibly();  
         return this.result;}  
    finally {this.done.release();}}
```

Der Nachteil von Parallelität; auf Ergebnisse muss man warten.



Ein generischer Loader

Hibernate ist nicht immer
einfach!

```
private void all(Session session) {  
    this.result = session.createQuery("from "+clazz.getName(), clazz).list();  
}  
  
private void some(Session session) {  
    EntityManager em = session.getEntityManagerFactory().createEntityManager();  
    CriteriaBuilder builder = em.getCriteriaBuilder();  
    CriteriaQuery<T> cquery = builder.createQuery(this.clazz);  
    Root<T> root = cquery.from(this.clazz);  
    cquery.where(this.predicate.apply(builder, root));  
    TypedQuery<T> query = em.createQuery(cquery);  
    this.result = query.getResultList();  
    this.result.forEach(dao -> dao.initDAO(this.queue));}  
}
```



MyClub: Individuelle DAOs und ein Manager

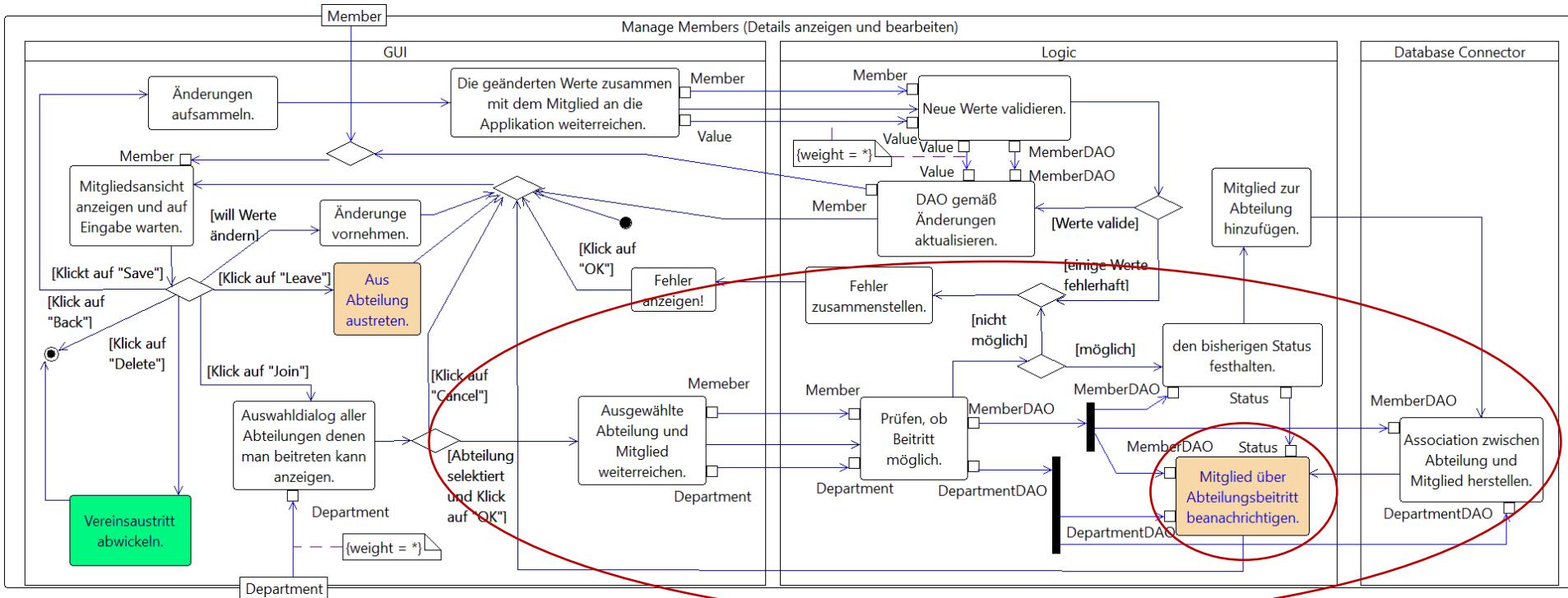
```
@Entity  
@Table(name = "member")  
public class MemberDAOImpl extends DAOImpl<MemberDAOImpl> implements MemberDAO {  
  
    protected MemberDAOImpl() {super();}  
    public MemberDAOImpl(DaoQueue queue) {super(queue);}  
  
    @Id  
    @GeneratedValue(  
        strategy = GenerationType.AUTO)  
    private Integer id;  
    private String address;  
    /*...*/  
  
    @Override  
    public String getAddress() {  
        return this.address;  
    }  
}
```

```
public class StoreImpl implements Store {  
  
    private List<MemberDAOImpl> allMembers;  
    private List<DepartmentDAOImpl> allDepartments;  
    private DaoQueue queue;  
  
    public StoreImpl(DaoQueue queue) {  
        this.queue = queue;  
        this.queue.open();  
        this.allMembers  
            = new LinkedList<>(  
                new Loader<>(MemberDAOImpl.class, null, queue).result());  
        this.allDepartments  
            = new LinkedList<>(  
                new Loader<>(DepartmentDAOImpl.class, null, queue).result());  
    }
```

Jeder hat seine
Aufgaben!

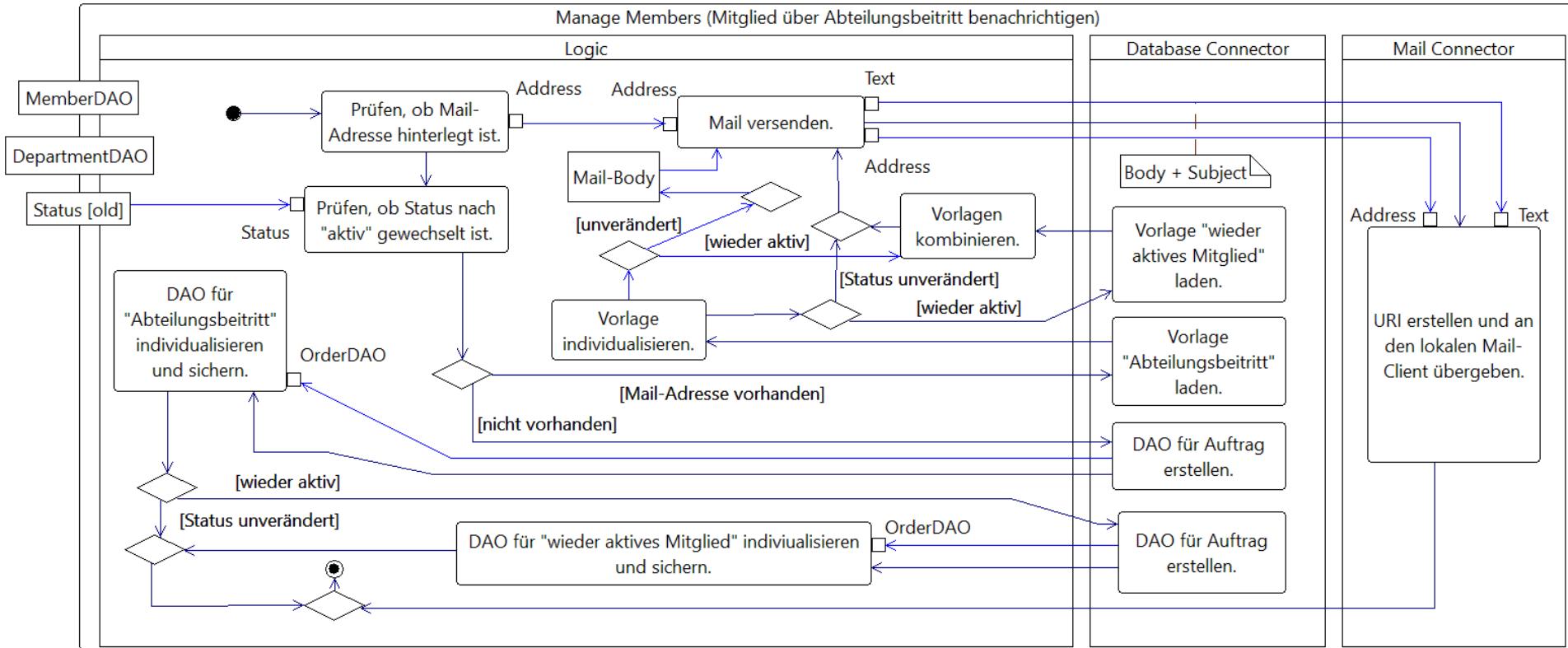


MyClub: Use Case „Manage Members“ (Schritt 3)





MyClub: Use Case „Manage Members“ (Schritt 3)

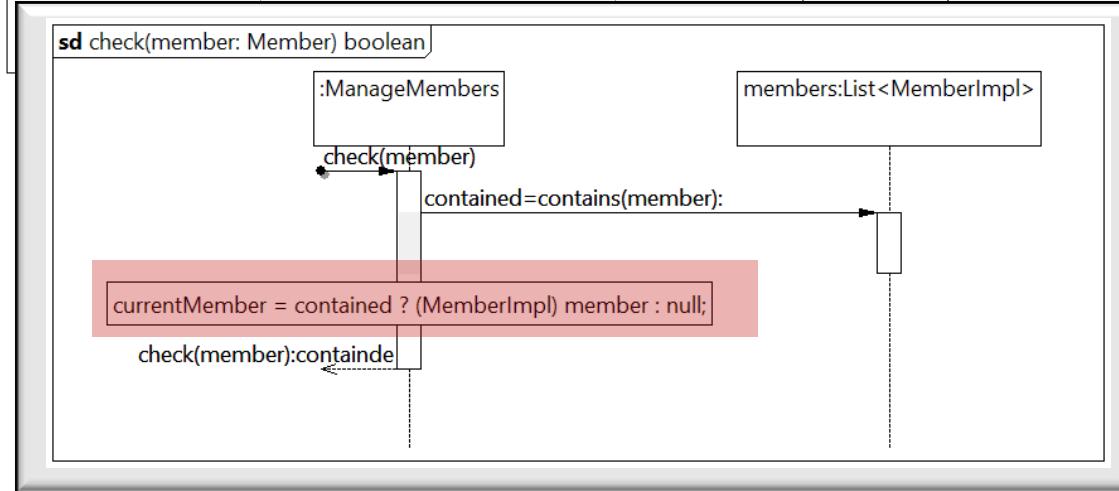
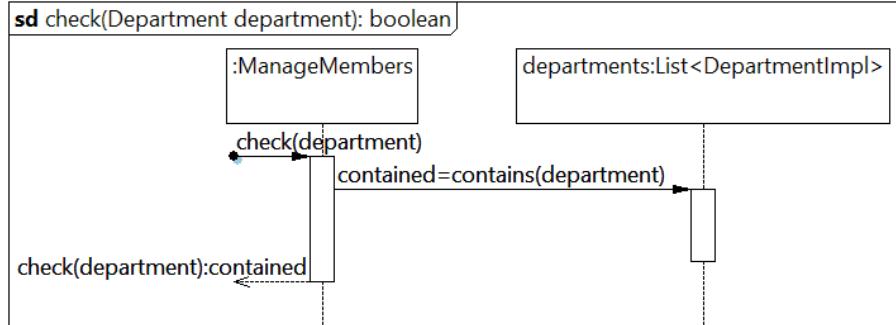
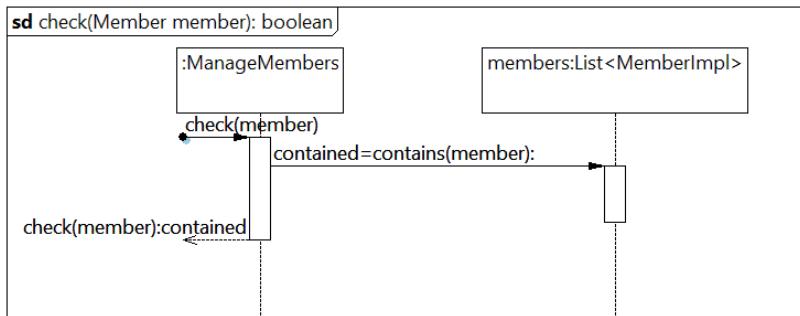


MyClub: Manage Members „joinDepartment()“





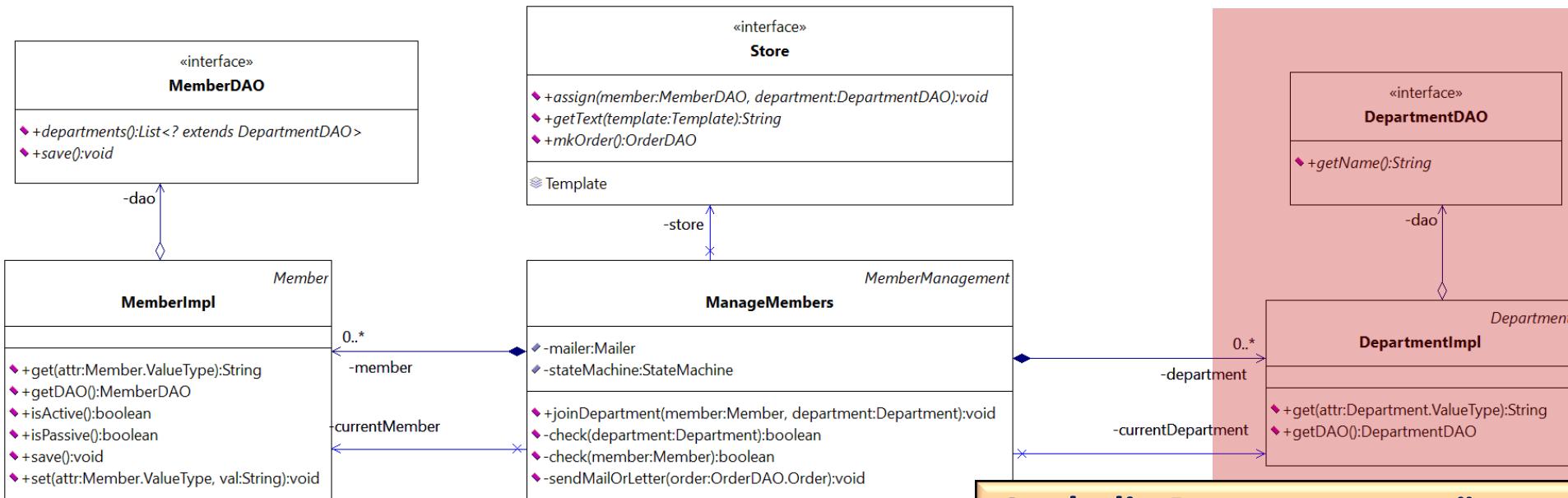
MyClub: Manage Members „joinDepartment .check()“



**Merken und Konvertieren
erleichtert den Zugriff!**



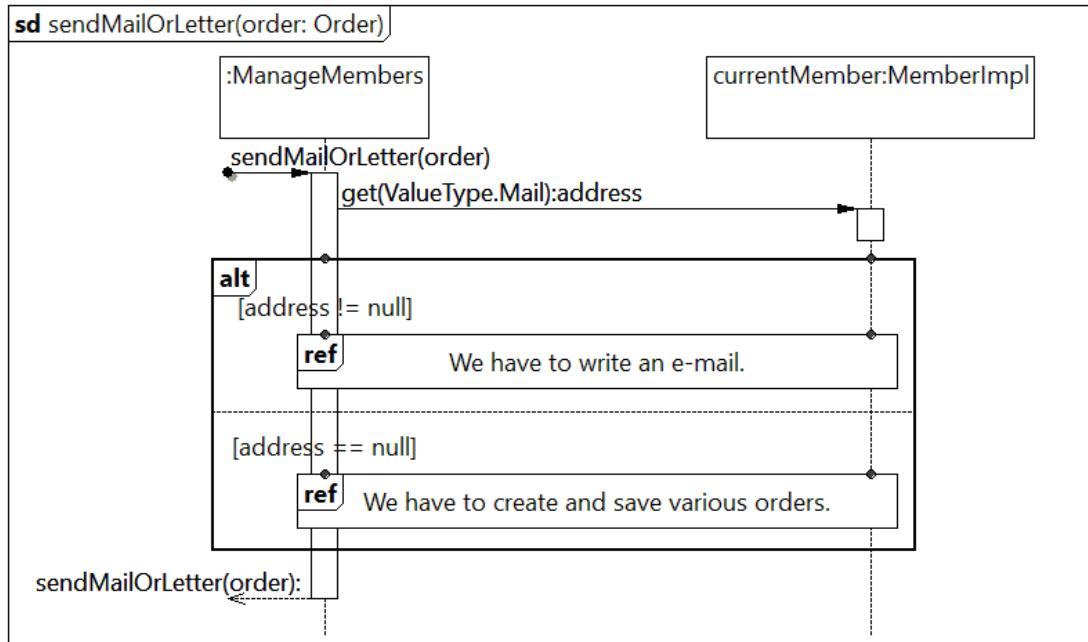
MyClub: Manage Members „joinDepartment()“



Auch die Departments müssen verwaltet werden. Die nächste Brücke muss her!



MyClub: Manage Members „joinDepartment.sendMailOrLetter()“



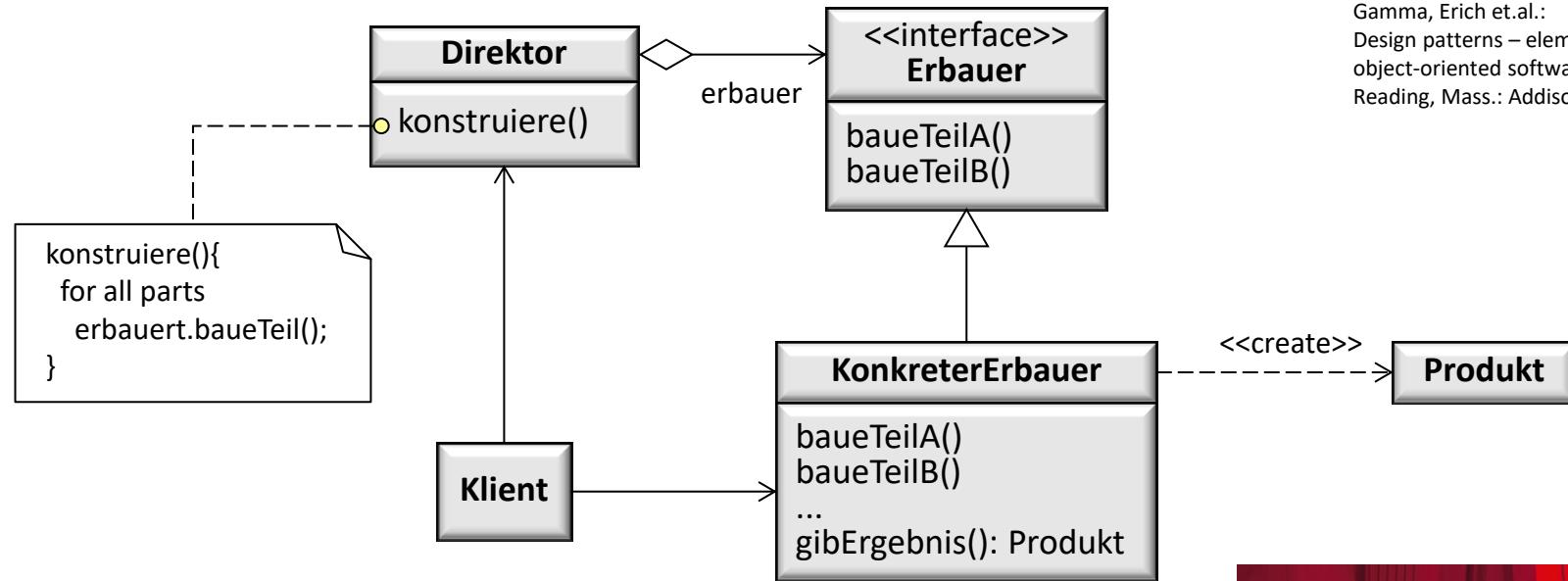
Offensichtlich fast identische Aufgaben mit leicht unterschiedlichem Ergebnis:

- eine Mail die versendet werden muss
- Benachrichtigungen (Briefe), die in der Datenbank gespeichert werden.



Das Erzeugungsmuster „Erbauer“

Der Erbauer trennt die Konstruktion eines komplexen Objekts von seiner Repräsentation. Damit kann derselbe Konstruktionsprozess für viele (auch unterschiedliche) Objekte verwendet werden.





Anwendbarkeit

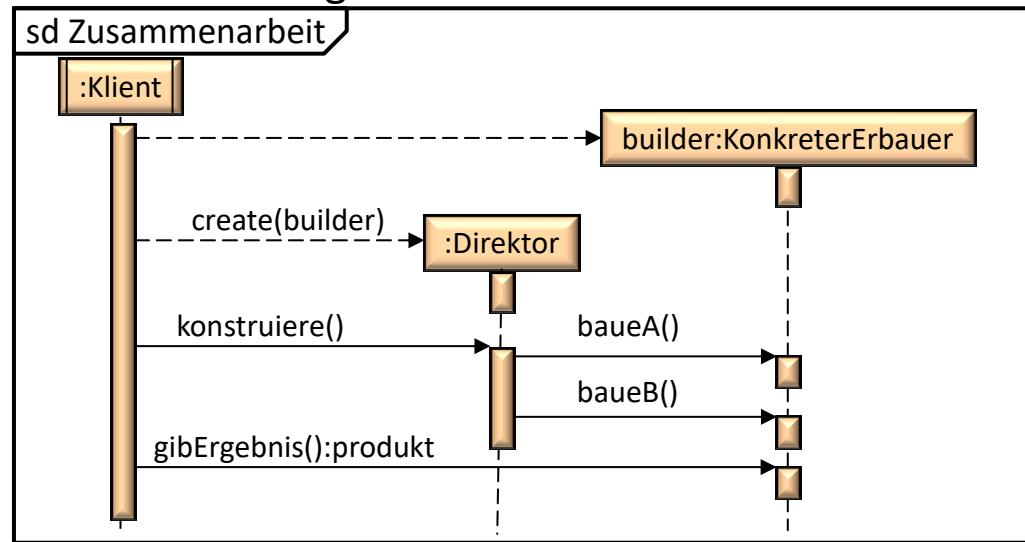
- **Man verwendet das Erbauermuster, wenn:**
 - das Objekt aus vielen unterschiedlichen Einzelteilen besteht.
 - der Algorithmus zur Erzeugung eines komplexen Objekts unabhängig vom Objekt sein soll. z.B., wenn das Objekt aus unterschiedlichsten Einzelteilen zusammen gesetzt ist.
 - der Konstruktionsprozess Varianten erlaubt.

Im Allgemeinen benötigt man keine abstrakten Produkt-Klassen, da die Produkte oft völlig disjunkt sind (komplexe Produkte).



Erbauer: Zusammenarbeit

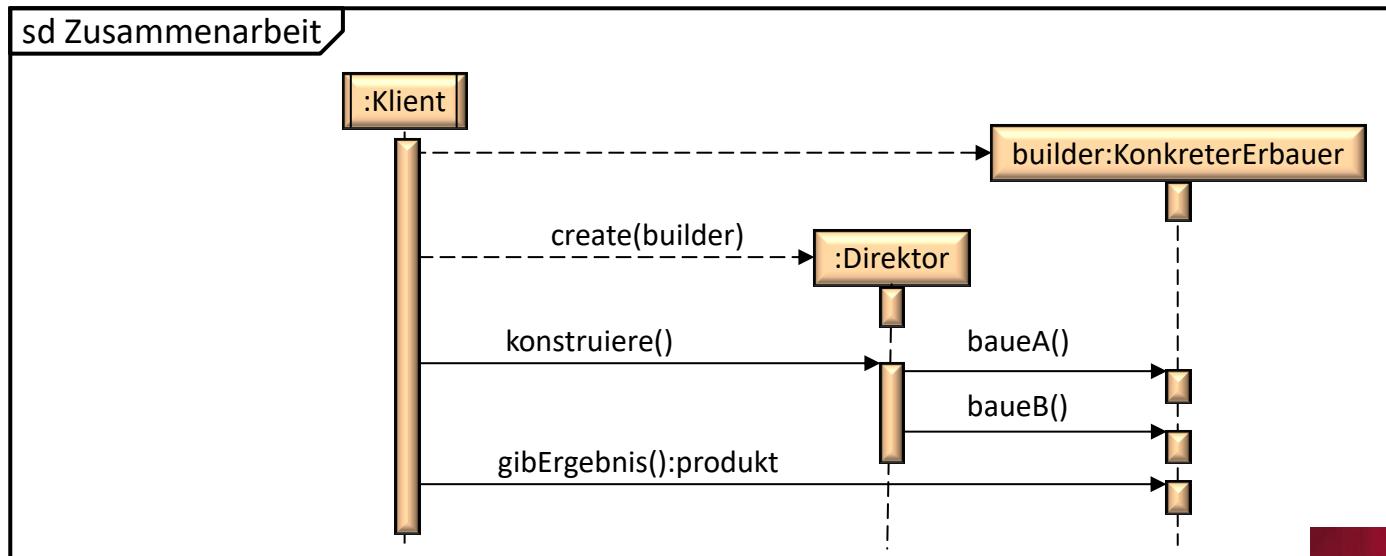
- Der Klient erzeugt den Direktor und konfiguriert ihn mit dem konkreten Erbauer.
- Der Direktor organisiert den Erstellungsprozess und delegiert ihn an den Erbauer.
- Der Erbauer handelt die Requests ab und erstellt das Produkt nach Anweisung.
- Der Klient holt sich das fertige Produkt vom Erbauer.





Erbauer: Zusammenarbeit

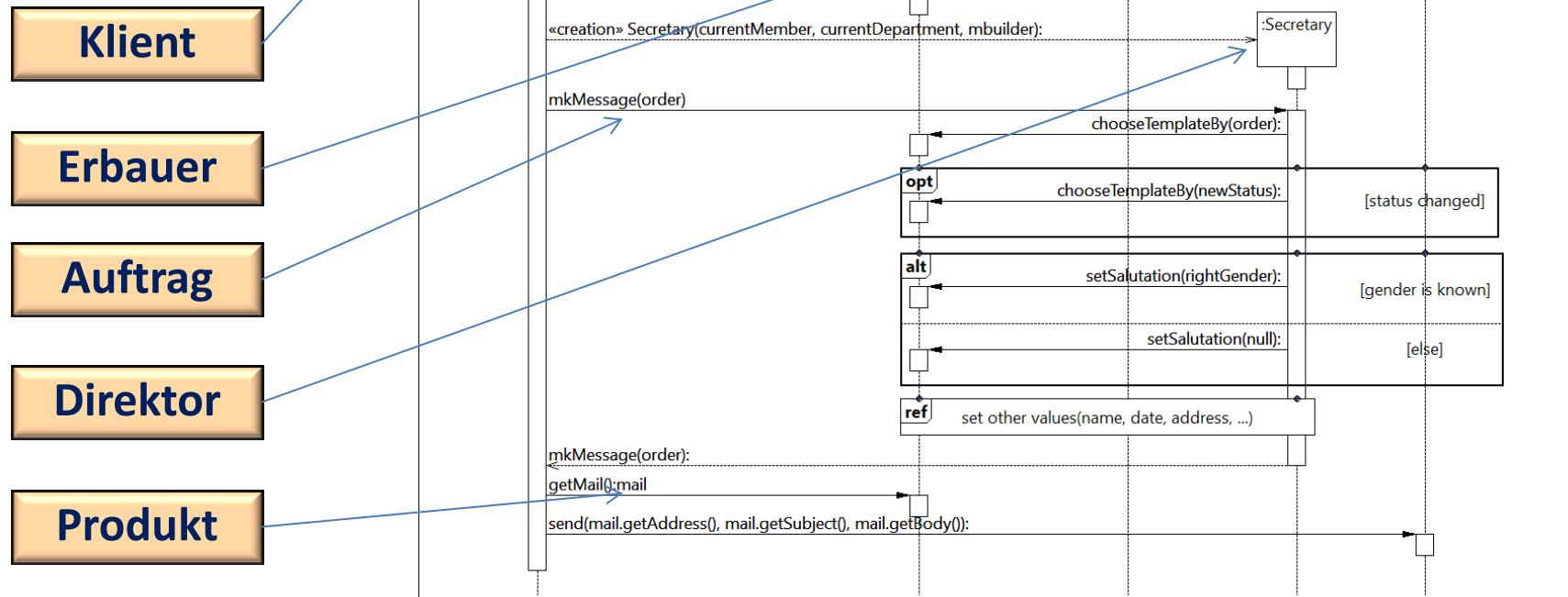
- Der Klient erzeugt den Direktor und konfiguriert ihn mit dem konkreten Erbauer.
- Der Direktor organisiert den Erstellungsprozess und delegiert ihn an den Erbauer.
- Der Erbauer handelt die Requests ab und erstellt das Produkt nach Anweisung.
- Der Klient holt sich das fertige Produkt vom Erbauer.





MyClub: Manage Members „joinDepartment.sendMailOrLetter()“

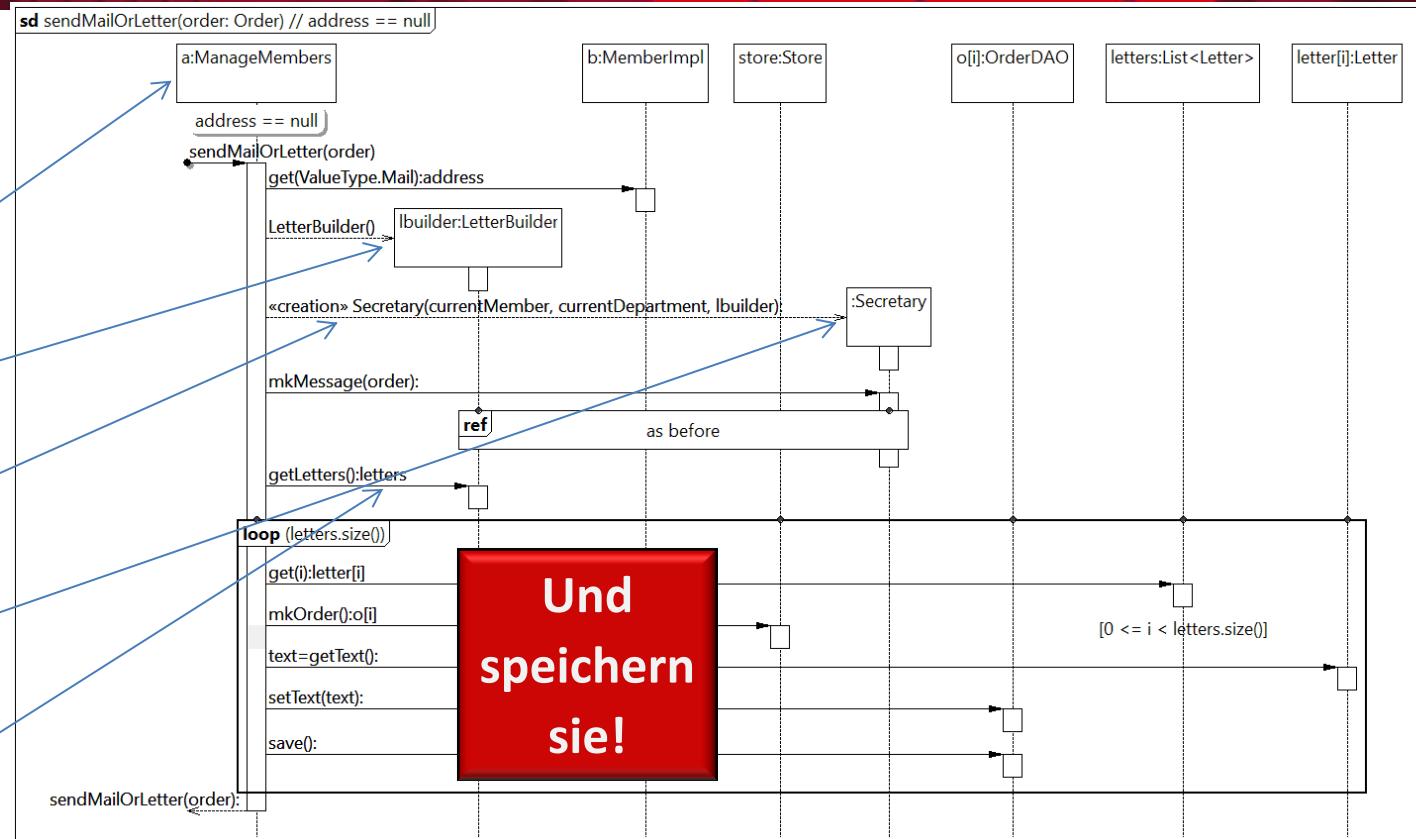
- Mail-Adresse liegt vor:
Wir bauen ein E-Mail!





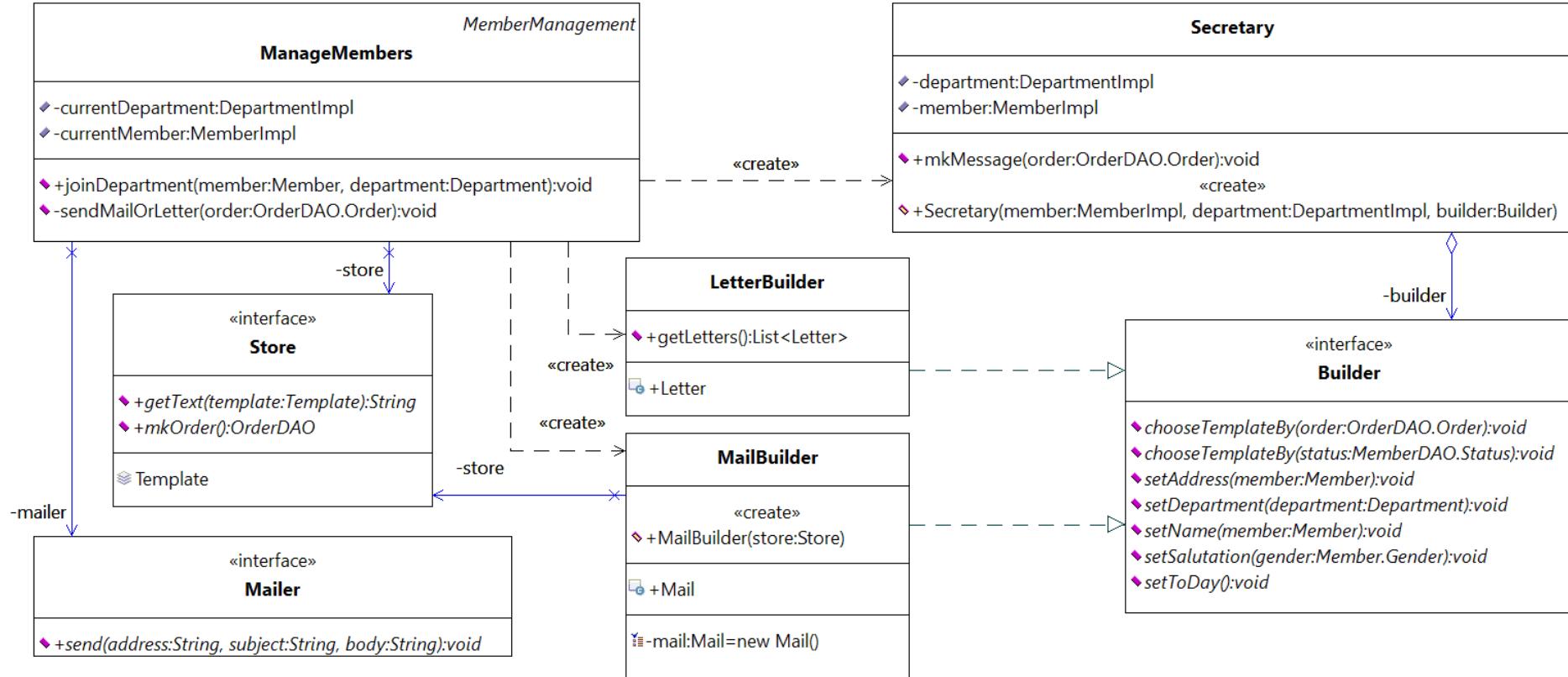
MyClub: Manage Members „joinDepartment.sendMailOrLetter()“

- Mail-Adress fehlt:
Wir bauen Briefe!





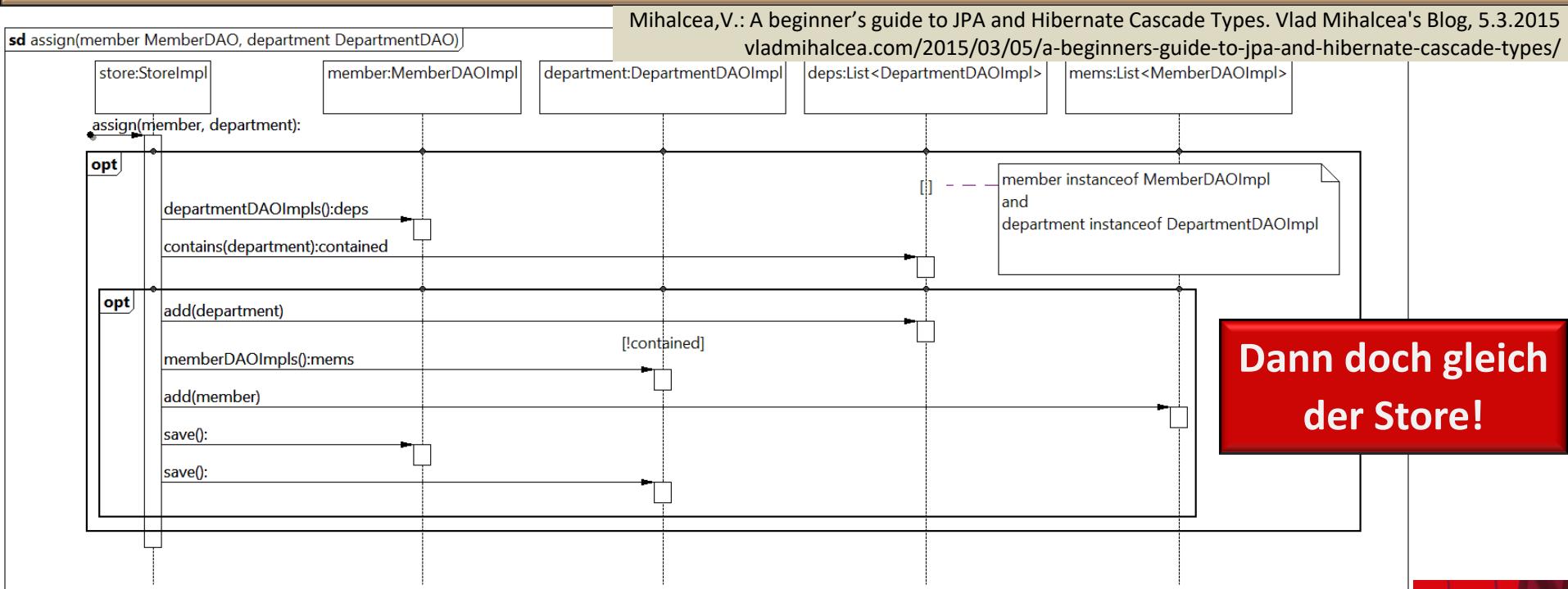
MyClub: Manage Members „joinDepartment.sendMailOrLetter()“





MyClub: Manage Members „joinDepartment.assign()“

“The many-to-many relationship is tricky because each side of this association plays both the Parent and the Child role. Still, we can identify one side from where we'd like to propagate the entity state changes.”





MyClub: Manage Members „joinDepartment.assign()“

```
@Override
public void assign(MemberDAO member, DepartmentDAO department) {
    MemberDAOImpl realMember =
        allMembers.contains(member) ? (MemberDAOImpl) member : null;
    DepartmentDAOImpl realDepartment =
        allDepartments.contains(department) ? (DepartmentDAOImpl) department : null;

    if (realMember != null
        && realDepartment != null
        && !realMember.departmentDAOI
            .contains(realDepartment))

        realMember.departmentDAOImpls()
            .add(realDepartment);
        realDepartment.memberDAOImpls()
            .add(realMember);
        realMember.save();
        realDepartment.save();
    }
}

@Entity
@Table(name = "member")
public class MemberDAOImpl extends ...

@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(name = "Member_x_Departments",
           joinColumns = @JoinColumn(name = "member_id"),
           inverseJoinColumns = @JoinColumn(name = "department_id"))

private List<DepartmentDAOImpl> departments = new LinkedList<>();

public List<DepartmentDAOImpl> departmentDAOImpls() {
    return this.departments;}
```



Was noch fehlt!

- „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
 - **Die Oberfläche bauen!**
- Tests: Nur was getestet wurde, kann abgenommen werden.
 - Systemintegrations-/Komponentenintegrationstest: (**Im Fokus Teilsysteme**)
 - Modultests: (**Im Fokus Units – „kleine“ Komponenten**)

Andere
Vorlesung

MyClub: Maage Member

Berry Backlog

Last: Backlog Birthday: 10.11.1899

First: Berry Gender: male

Address: Swimelane 2

City: Join City

Email: berry.backlog@swe.club

Postal Code: 23415

Save

Testen gehört zu den wichtigsten Aufgaben und wird oftmals vernachlässigt!



Was noch fehlt!

- Alles „zusammenschrauben“

```
public class Main {  
    public static void main(String[] args) {  
        init();  
  
        GuiFactory.FACTORY.uiPort().ui().startEventLoop();  
  
        DbConnectorFactory.FACTORY.accessPort().close();  
    }  
  
    private static void init() {  
        // We simulate the authentication  
        Token token = new TokenImpl();  
        AuthenticationFactory.FACTORY.securityCenterPort().securityCenter().setToken(token);  
        DataStore.data().setToken(token);  
        // and jump to the required state.  
        StateMachineFactory.FACTORY.stateMachinePort().stateMachine().setState(State.S.SelectUseCase);  
    }  
}
```

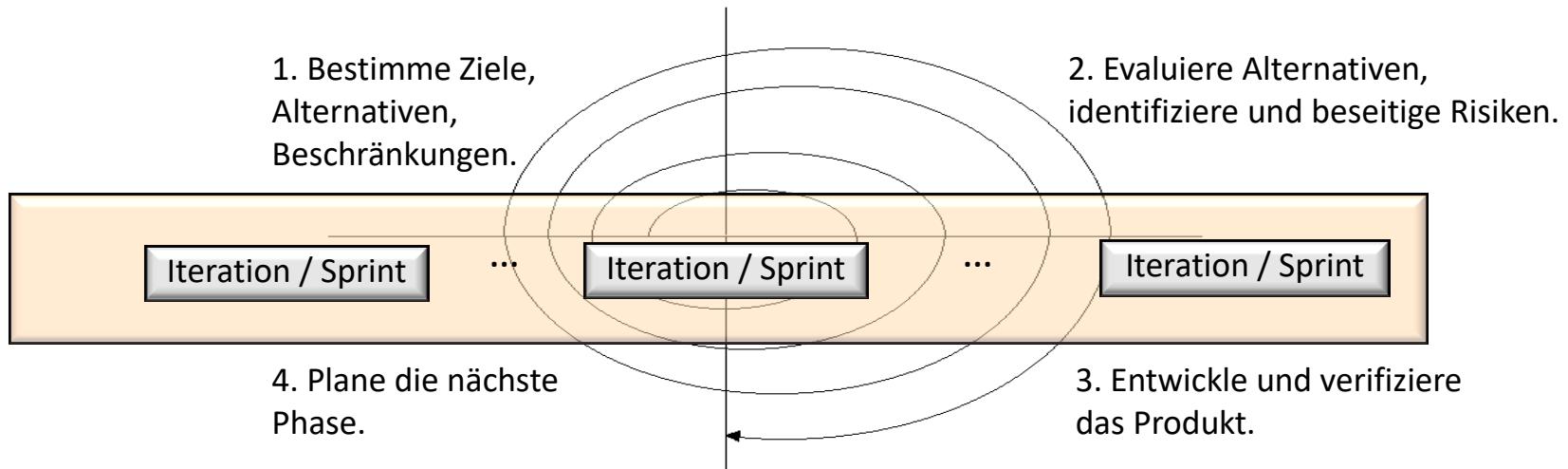
Initialisieren nach Bedarf,
starten

und zum Schluss die Datenbank
schließen!



Was jetzt?

Weiter mit Use Case 2 „Authenticate User“

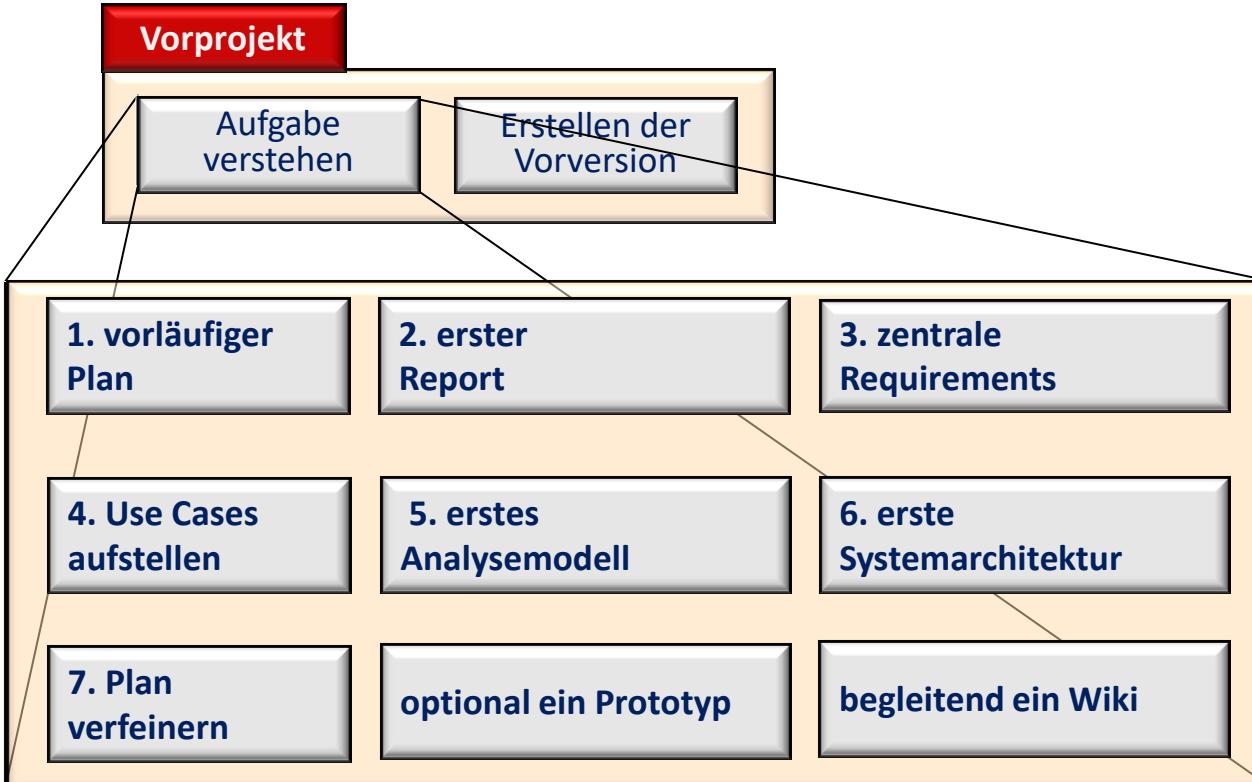


und danach zum nächsten Sprint!

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



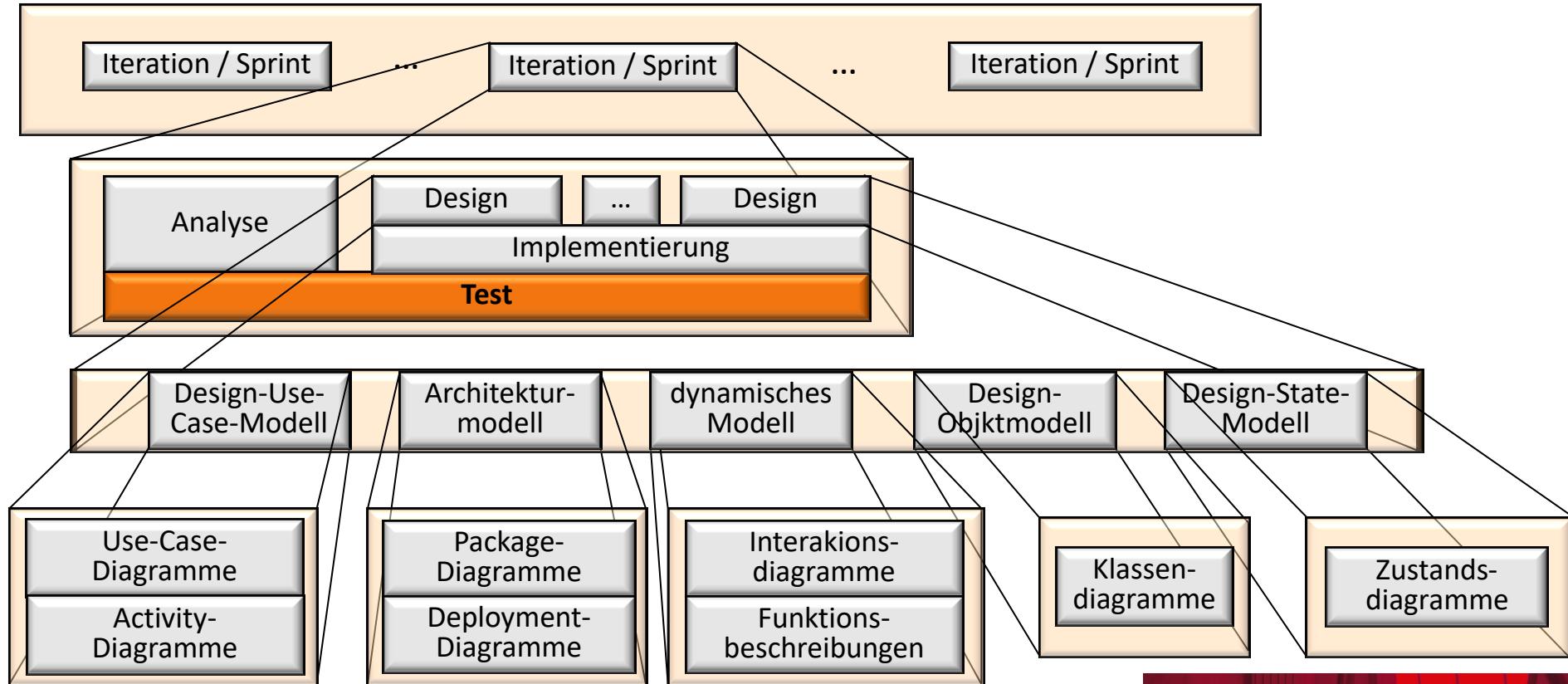
Design im Überblick



Aufwand und Umfang steigt mit der Größe des Projektes!



Das Design im Überblick





Design Zusammenfassung

Nach der Definition der zentralen Strategien, im Rahmen des Systementwurfs, werden in der Design- und Entwicklungsphase die Entscheidungen für die Realisierung getroffen und umgesetzt.

- **System-/Architekturentwurf (das Bindeglied zwischen Analyse und Design)**
 - Aufbrechen des Systems in Teilsysteme
 - Strategien für Steuerung und Datenhaltung finden
- **Objektentwurf (das funktionale Design)**
 - Umsetzung der Strategien des Systementwurfs
 - Implementierungsentscheidungen treffen
 - Verantwortlichkeiten zuweisen
 - 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)

Die Operationen der Schnittstelle entwerfen.