UML für Abschlussarbeiten

Überblick über wichtige UML Modelle

Prof. Dr. Jens von Pilgrim

Stand: 28.08.2023

BHT Berliner Hochschule für Technik

BHT Berliner Hochschule für Technik

Unified Modeling Language

Unified Modeling Language (UML)

Keine Angst vor TLAs (Three-Letter Acronyms)

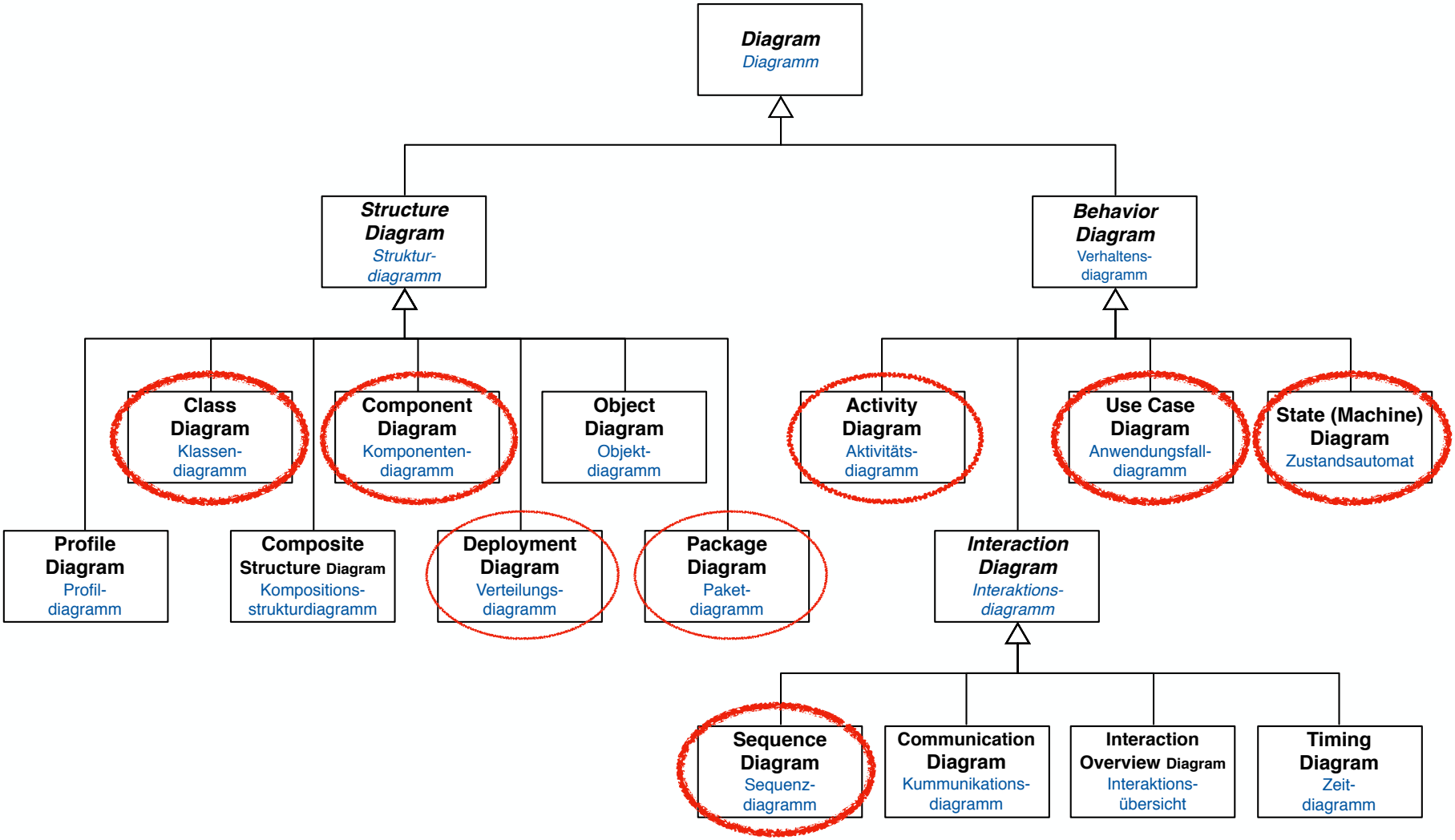
Sprache definiert

- abstrakte Syntax mittels MOF (Meta Object Facilities) – vereinfacht: Klassenmodelle
- Semantik mittels OCL (Object Constraint Language) und textuell
- graphische Notation von Elementen in Diagrammen

Erweiterbar mittels Profilen über Stereotypes (Constraints, Icons)

Spezifikationen von UML, MOF, OCL etc. unter dem Dach der OMG (Object Management Group): www.omg.org

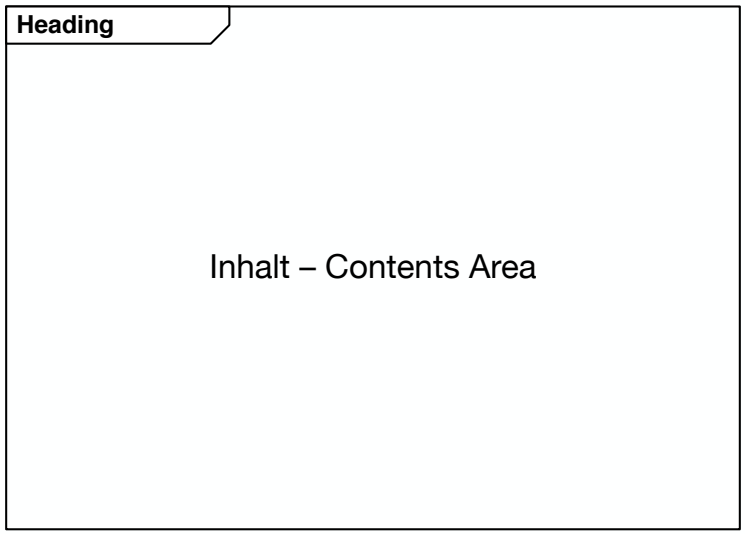
UML Diagramme



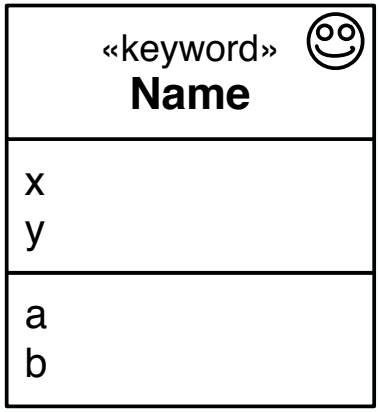
nach: The taxonomy of structure and behavior diagrams [OMG17],
Übersetzungen siehe www.oose.de/nuetzliches/uml-auf-gut-deutsch

Diagramm

- dient im Wesentlichen der Identifikation
- hier: Pragmatischer Ansatz: Kann auch weggelassen werden, wenn bspw. durch Bildunterschrift eindeutig identifizierbar und Tools den Rahmen nicht unterstützen.



UML Classifier



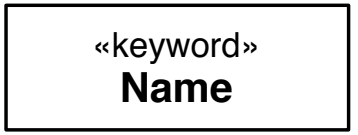
abstraktes Basiskonzept, Spezialisierungen sind

- Classes
- Interfaces
- Actors etc.

Eigenschaften (ähnlich Member in Java) werden “Features” genannt

“Classifier”-Notation

- Rechteck mit
  - Name
  - Keyword und/oder **Stereotypes** in “Guillemets” (franz. Anführungszeichen)
  - Icon, optional
- darunter ausblendbare Compartments
- alternativ: Icon-Darstellung



Name

macOS: « — ⌥Q; » — ⌥⇧Q
Windows: « — Alt + 174, » — Alt + 175

BHT Berliner Hochschule für Technik

System und Systemkontext

Kein UML, aber sehr nützlich

System

“1. combination of interacting elements organized to achieve one or more stated purposes

...” [SEV17]

Aus Modellierungssicht ist das “System” das Original, also das, was wir am Ende herstellen. Dies kann eine Kombination aus Elementen sein, i.A.

- **Software**
- Hardware

Diese können wiederum aus Elementen bestehen.

(Application) Domain (Anwendungsdomäne)

BHT Berliner Hochschule für Technik

Application Domain (Anwendungsdomäne)

- Fachlich zusammenhängender, abgegrenzter Bereich innerhalb der Realität.
- Anwendungsbereich des Systems

RE: Problemraum verstehen und Anforderungen ermitteln, die um dieses Problem lösen.

Problemraum — **Was!**

- Innerhalb der Domain treten Problemstellung technischer oder organisatorischer Art auf

Lösungsraum — **Wie!**

- Mögliche Lösungen, die das Problem einer Domain lösen.

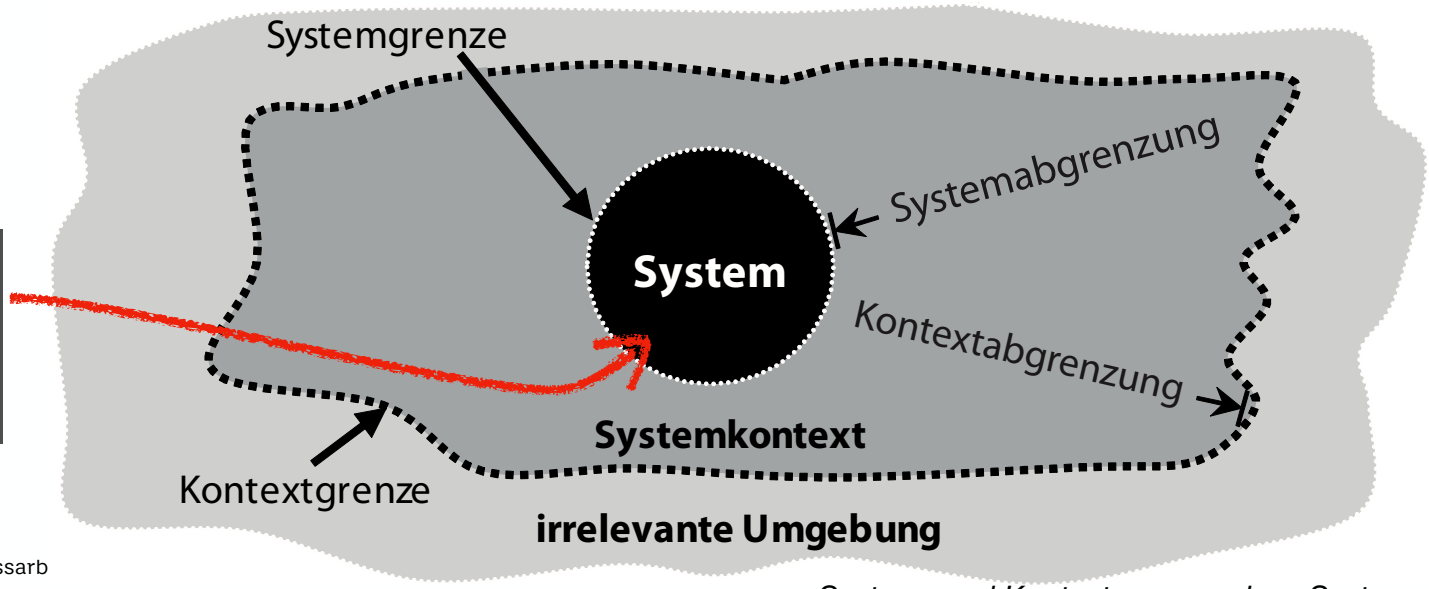
Das "wie" zu definieren ist dann Teil des Entwurfs.

von Pilgrim: UML für Abschlussarbeiten

Systemkontext

Der Systemkontext soll beantworten:

- Wer oder was interagiert (in einer Situation) mit dem System?
- Was gehört zum System, was ist außerhalb?
- Wo kann man das System abgrenzen?



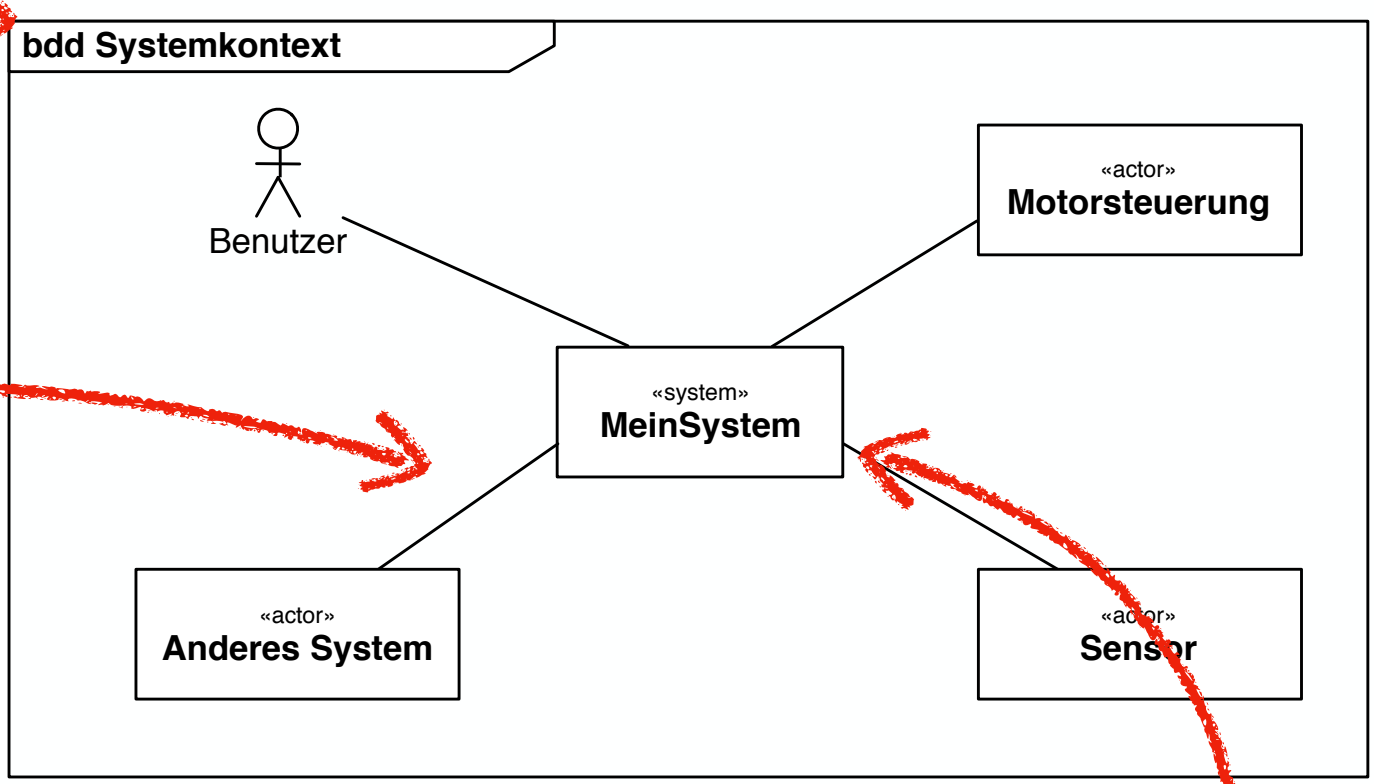
Definierter Scope (System-umfang)

von Pilgrim: UML für Abschlussarb

System- und Kontextgrenzen eines Systems, aus [PR15]

Modellierung des Systemkontexts

bdd: Block Definition Diagram, aus SysML (nicht so wichtig)



Alles außer das eine «system» sind Akteure (Actors), daher lassen wir pragmatisch das Stereotyp «actor» manchmal weg.

Kante zeigt möglichen Austausch von Signalen, Daten, aber auch Energie oder Material

Classifier mit Stereotyp «system»,

In dieser Form kein UML-Standard, sondern aus [WEI14] entnommen.

UML Profil

Die einfachste Art, UML zu erweitern und für einem bestimmten Zweck anzupassen.

ein Profil ist eine Sammlung von Stereotypen

ein Stereotyp

- hat einen Namen
- optional ein Icon
- wird auf existierende Elemente in der UML angewendet
- und legt weitere Eigenschaften (Constraints) dieser Elemente fest

der Name wird über dem Classifier-Namen in Guillemets geschrieben: «MeinStereoTyp» (wie Keywords)

macOS: « — ⌥Q; » — ⌥⇧Q
Windows: « — Alt + 174, » — Alt + 175

SysML

Erweiterung der UML speziell für System Engineering
offizielles Profil definiert von der OMG (die auch UML definiert):

- OMG Systems Modeling Language (OMG SysML), Version 1.5. formal/2017-05-01. Need- ham, MA: Object Management Group, Mai 2017. https://www.omg.org/cgi-bin/doc?formal/2017-05-01

ergänzt UML um ein paar Diagrammtypen und spezialisiert einige Elemente

UML Actor (Akteur)

Akteure kommen in der UML auch außerhalb von Use-Case-Diagrammen vor!

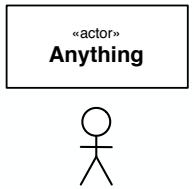
interagieren mit dem System

sind immer **außerhalb** des Systems

kann eine Person sein

kann aber auch anderes (externes) System, ein Zeitereignis, ein Sensor o.ä. sein!

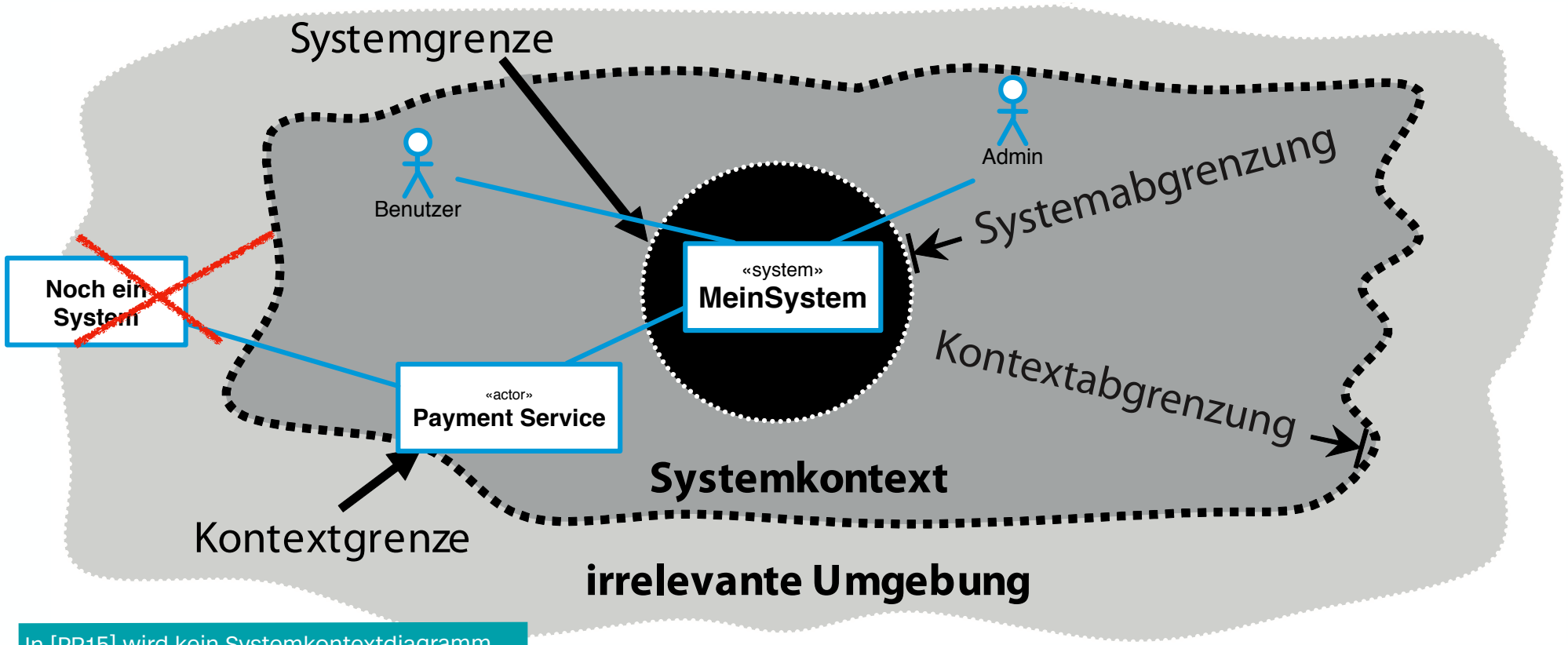
Notation:

- Classifier 

mit Keyword «actor»
- Actor-Icon Achtung: I.A. Person, kann aber auch als allgemeines Icon verwendet werden
- benutzerdefiniert, bspw. 

für Zeitereignisse.

Systemkontextdiagramm im Kontext

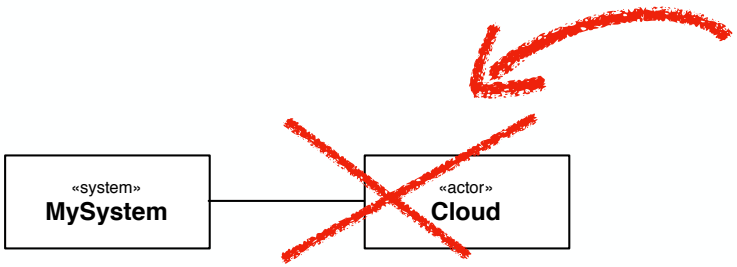


In [PR15] wird kein Systemkontextdiagramm verwendet, sondern nur eine Illustration der Begriffe (die grau-schwarzen Flächen). Im Systemkontextdiagramm werden NUR Akteure eingezeichnet. Also unbedingt zwischen Illustration [PR15] und Diagramm unterscheiden!

System- und Kontextgrenzen eines Systems, aus [PR15]

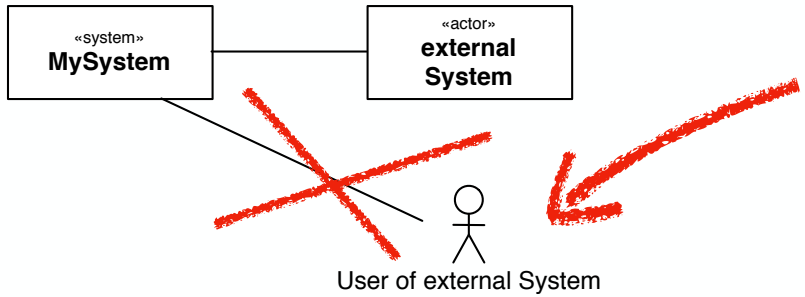
Akteur oder nicht Akteur?

Akteur bedeutet, es werden **Informationen** o.ä. zwischen System und Akteur **ausgetauscht**.
Wenn dies nicht der Fall ist: Ist der Akteur wirklich ein Akteur?



Infrastruktur ist kein Akteur!

Manche dieser Nicht-Akteure sind Infrastruktur oder werden indirekt, über andere (externe) Systeme angesteuert



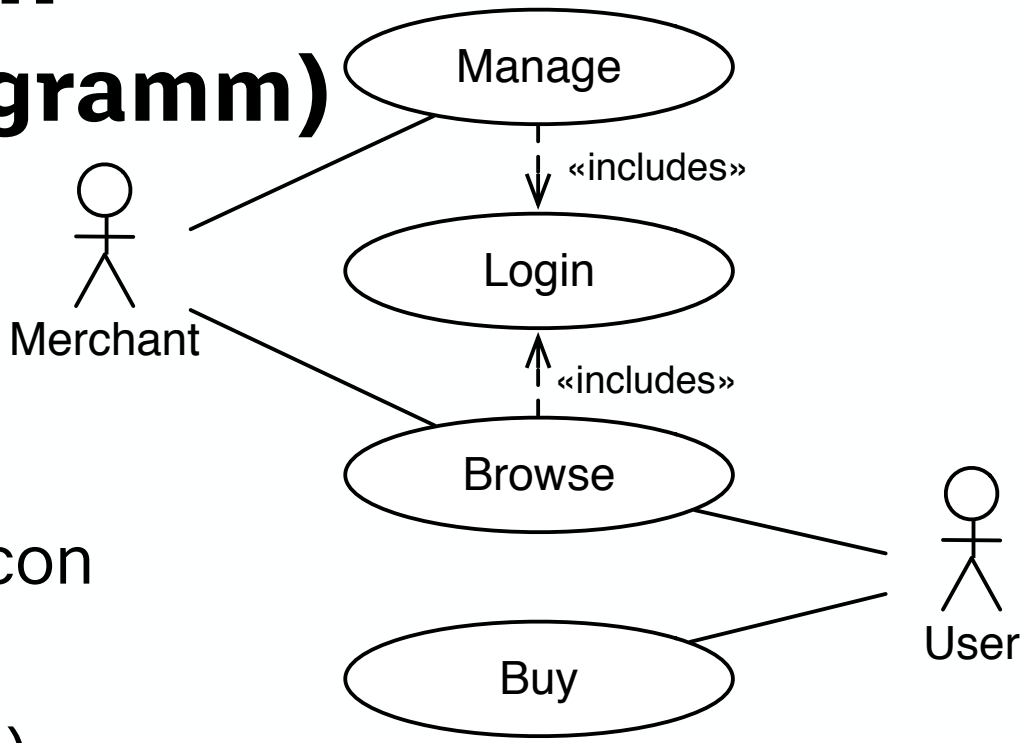
Benutzer externer Systeme haben nichts mit „unserem“ System zu tun.

von Pilgrim: UML für Abschlussarbeiten

BHT Berliner Hochschule für Technik

Funktionsmodellierung mit Anwendungsfällen

Use Case Diagram (Anwendungsfalldiagramm)



enthält

- Actors (Akteure, bekannt aus Systemkontext-diagramm), meist als Actor-Icon gezeichnet.
- Use Cases (Anwendungsfälle)
- Assoziationen zwischen Actors und Use Cases
- Beziehungen zwischen Use Cases

Use Case Diagramme sehen meist wie hier dargestellt aus: weitere Beziehungen selten.

Das Diagramm gibt nur eine Übersicht, die eigentliche Arbeit steckt in der Beschreibung der Anwendungsfälle

von Pilgrim: UML für Abschlussarbeiten

UML Use Case (Anwendungsfall)

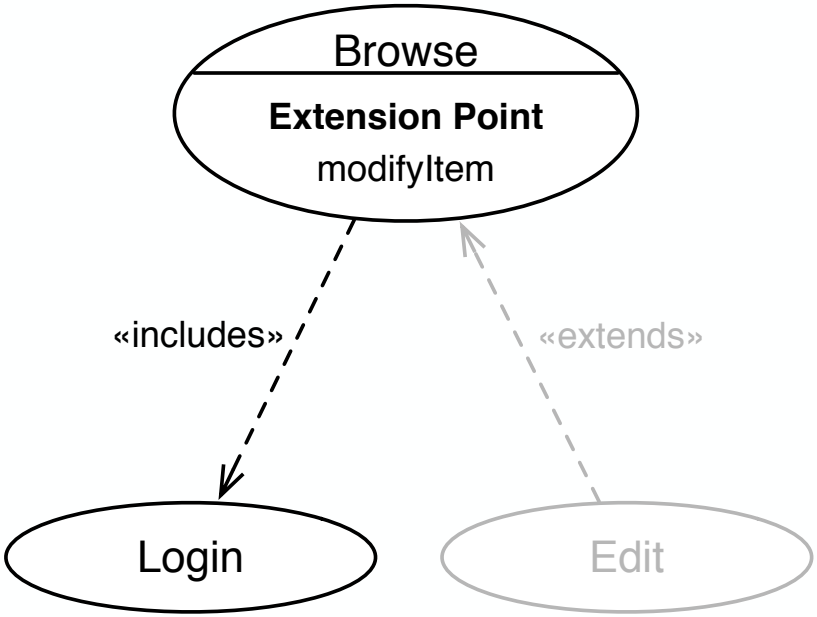
Use Case ist ein Classifier

Notation:

- **Ellipse** mit dem Namen

mögliche Beziehungen (gestrichelter Pfeil mit Keyword):

- **includes**: Der Use Case beinhaltet einen anderen Use Case
- **extends**: Der Use Case erweitert einen anderen Use Case. Dieser muss dazu **Extension Points** anbieten (selten verwendet!)



Theoretisch Generalisierung und Assoziationen zwischen Use Cases möglich, aber super selten. Auch „extends“ ist eher selten!

von Pilgrim: UML für Abschlussarbeiten

Warum das Use-Case-Diagramm?

Berliner Hochschule für Technik

Die Übersicht könnte meist auch durch eine einfache Stichpunktliste ausgedrückt werden, also:

- User und Merchant können browsen
    - dabei müssen sie sich u.U. einloggen
- User können etwas kaufen
- Merchants können Artikel einstellen

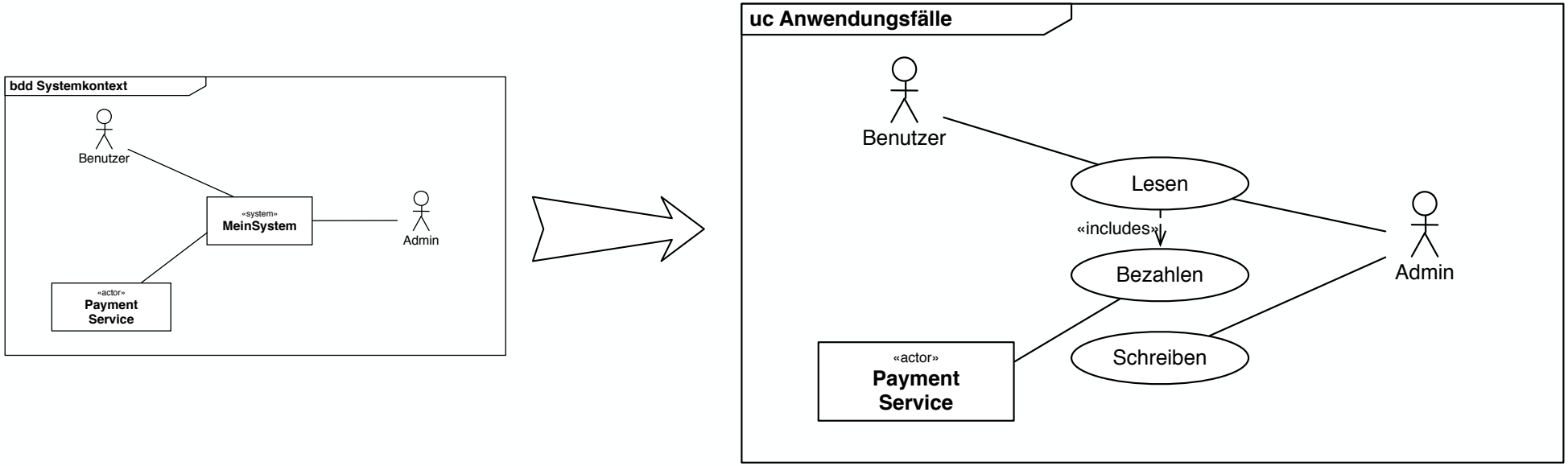
Stimmt, aber evtl. ist das Diagramm doch übersichtlicher... tatsächlich kann das Diagramm aber aus so einer Liste (aus dem Lastenheft bspw.) abgeleitet werden.

und da gibt es noch etwas....

Systemkontext–Use Cases

Das Anwendungsfalldiagramm entsteht durch Verfeinerung des Systemkontextdiagramms:

- Die Akteure werden übernommen
- Die Use Cases werden ergänzt.



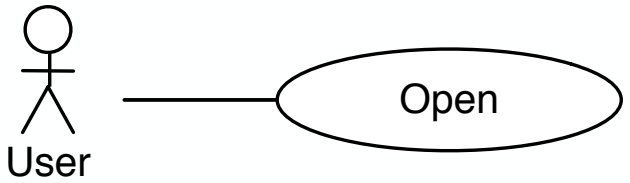
von Pilgrim: UML für Abschlussarbeiten

Use Case — Actor Beziehung

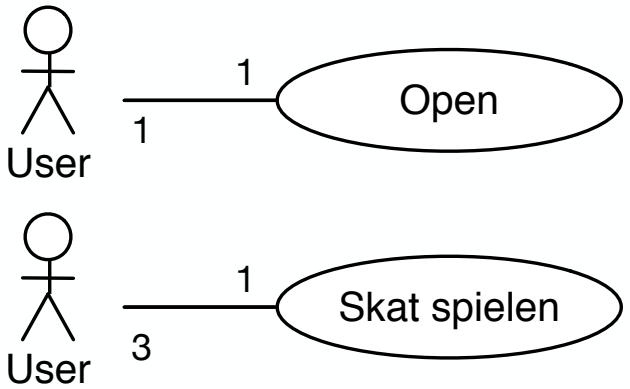
ist eine Assoziation

kann Multiplizitäten haben

- in der Regel 1—1
- wird daher meist pragmatisch weggelassen



fast immer nur so



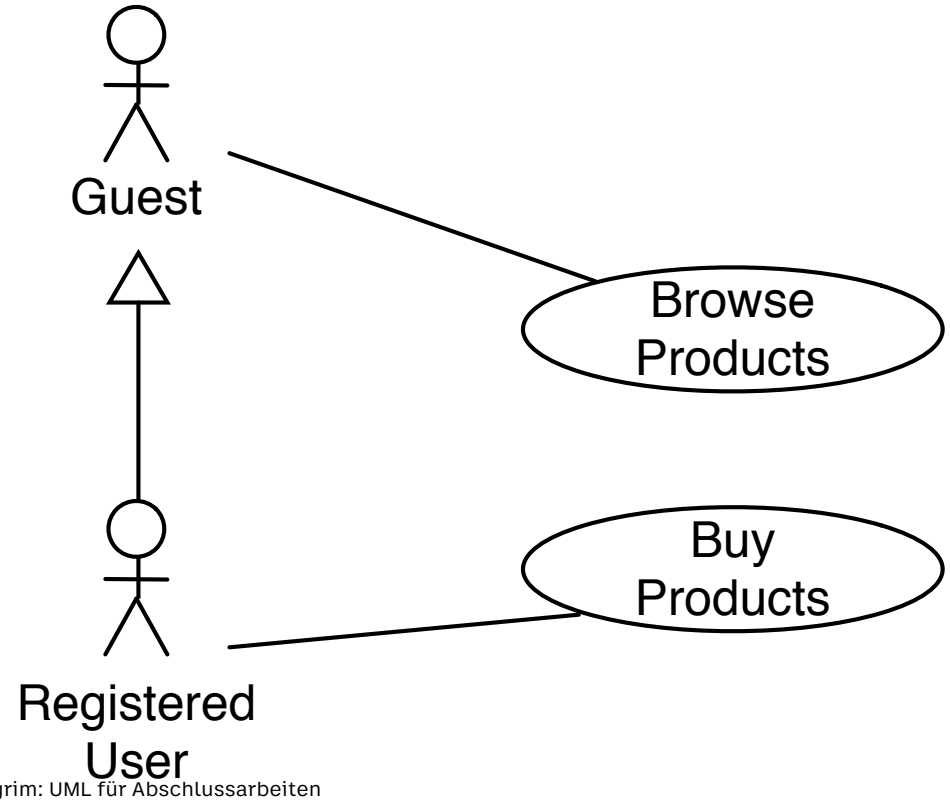
sehr selten

von Pilgrim: UML für Abschlussarbeiten

Actor-Beziehungen

Generalisierung möglich

- häufig problematisch
- falls nicht 100% sicher, besser vermeiden

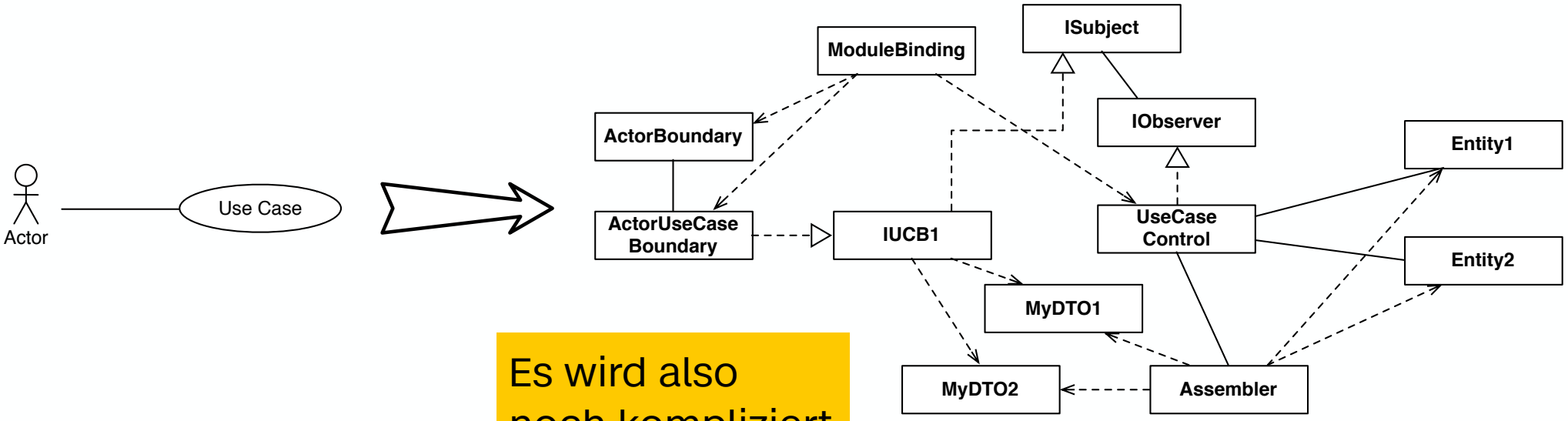


Sieht das „Browsen“ für Guest und User tatsächlich gleich aus?

Einfach ist gut!

Anwendungsfalldiagramme sollten möglichst einfach (=wenig Elemente) sein

aus einem Anwendungsfalldiagramm kann man später Klassen ableiten (Analyse) und diese im (Fein-) Entwurf noch überarbeiten.



Es wird also noch kompliziert genug!

Was ist ein Anwendungsfall?

BHT Berliner Hochschule für Technik

“A UseCase specifies a set of actions performed by its subjects, which yields an observable result that is of value for one or more Actors or other stakeholders of each subject. Each UseCase’s subject represents a system under consideration to which the UseCase applies.”

Zusammenfassende, etwas vereinfachende Übersetzung:

Ein Anwendungsfall spezifiziert eine Menge von Aktionen die von dem System ausgeführt werden und zu beobachtbaren Ergebnissen führen die für ein oder mehrere Akteure oder andere Stakeholder des Systems von Bedeutung sind.

[OMG17a, p. 639; Übers. u. Hervorh. JvP]

Granularität eines Use-Cases

Berliner Hochschule für Technik

“Menge von Aktionen”, also

- keine einzelne Aktion oder gar Programmfunktion
- v.a.: “include” ist kein Funktionsaufruf!

“beobachtbares Ergebnis”, also

- nichts was nur intern abläuft

“von Bedeutung für Akteur”, also

- Modellierung orientiert sich an Akteuren

Sollten aber auch nicht zu groß sein, vgl. atomare Requirements!

Anwendungsfall-beschreibung

“Menge von Aktionen” — die eigentliche Arbeit beim Modellieren von Use Cases steckt in der Beschreibung

kann einfach als nummerierte Liste erfolgen, sortiert nach Main Flow, Alternative Flow und Exceptional Flows (sehen wir gleich)

oder alternativ als Aktivitätsdiagramm (sehen wir später)

Textuelle Use Case Beschreibung

Berliner Hochschule für Technik

Tipp: Geben Sie die ID auch im Use-Case Diagram an!

use case <ID>: <NAME>
- actors
  - <Actor1>, ...
- precondition
  - ...
- main flow
  - 1. ...
  - 2. ...
  - ...
- alternative flow 1 <NAME>
  - at step 2. of main flow:
  - 1. ...
  - 2. ...
  - ...
- alternative flow n <NAME>
  - ...
- postcondition
  - ...
- exceptional flow 1 <NAME>
  - ...
  - [postcondition of e.f.]
- ...

end

Vorbedingung

Hauptszenario

Alternativszenario

Nachbedingung (für Main- und Alternative Flow)

Ausnahmeszenario

pre- und postcondition vgl. design-by-contract

aus [Winter05, S. 136], leicht abgewandelt

BHT Berliner Hochschule für Technik

Klassendiagramm

UML Klassendiagramm

wird in unterschiedlichen Phasen und damit auf verschiedenen Abstraktionsebenen verwendet:

- Domain-Modell im Requirements Engineering
- Architektur- und Grobentwurf
- Feinentwurf
- Code-Visualisierung (vgl. BlueJ)

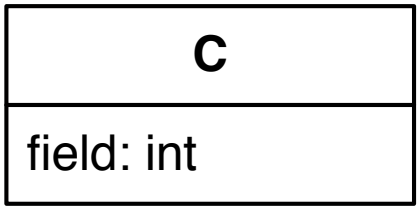
UML Class

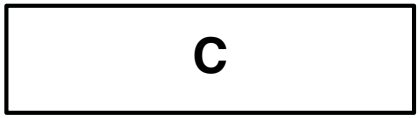
Notation:

- Name
- Compartments:
  Attribute, Operationen,
  (Receptions), interne Struktur

typisches Mapping nach Java:

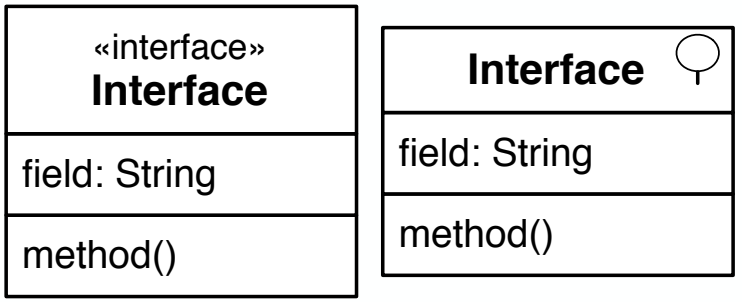
- Java “class”
- Attribute → Felder
- Operationen → Methoden





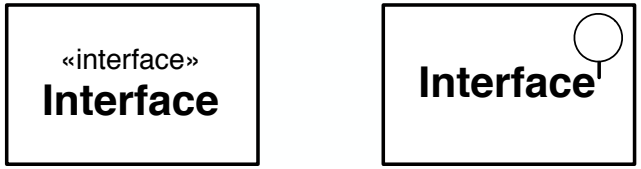

```
class C {
    int field;
    void method() {}
}
```

UML Interface



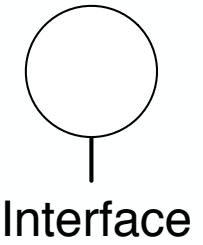
Notation:

- Keyword “interface”
- oder “Lollipop”-Icon



Semantik

- alle Features (Element) müssen public sein

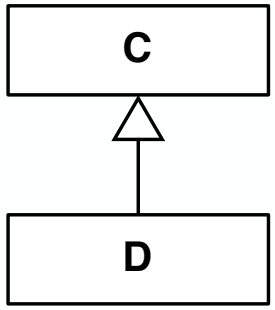


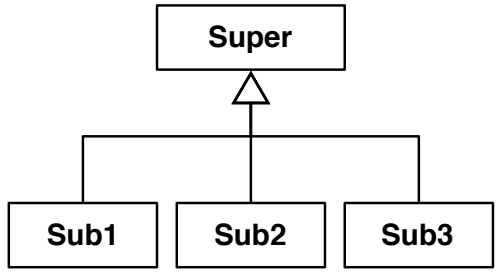
typisches Mapping nach Java:

- Java “interface”
- keine Instanz-Felder in Java!

```
interface Interface {}
```

UML Generalisierung (Generalization)





Notation

- Pfeil mit dreieckiger Pfeilspitze
- “separated” oder “shared” target style

Semantik:

- keine Zyklen erlaubt
- i.A. nur zwischen Classifiern desselben Typs erlaubt
- Mehrfachvererbung möglich

```
class D extends C {}
```

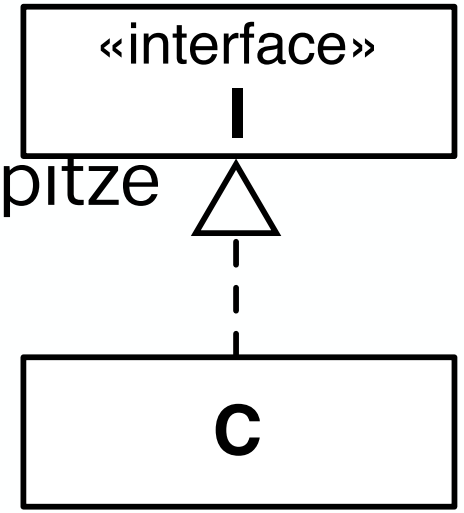
typisches Mapping nach Java:

- Vererbung (“extends”)
- Mehrfachvererbung über Interfaces und Delegation

UML Realisierung (Realization)

Notation

- gestrichelte Linie mit dreieckiger Pfeilspitze



Semantik

- zwischen Interface und Klasse (allg. “BehavioredClassifier”)

typisches Mapping nach Java:

- Implementierung (“implements”)

```
class C implements I {}
```

Assoziation, Übersicht

binäre A.: durchgängige Linie zwischen den beteiligten Classifiern

Rollen (Enden der A.)

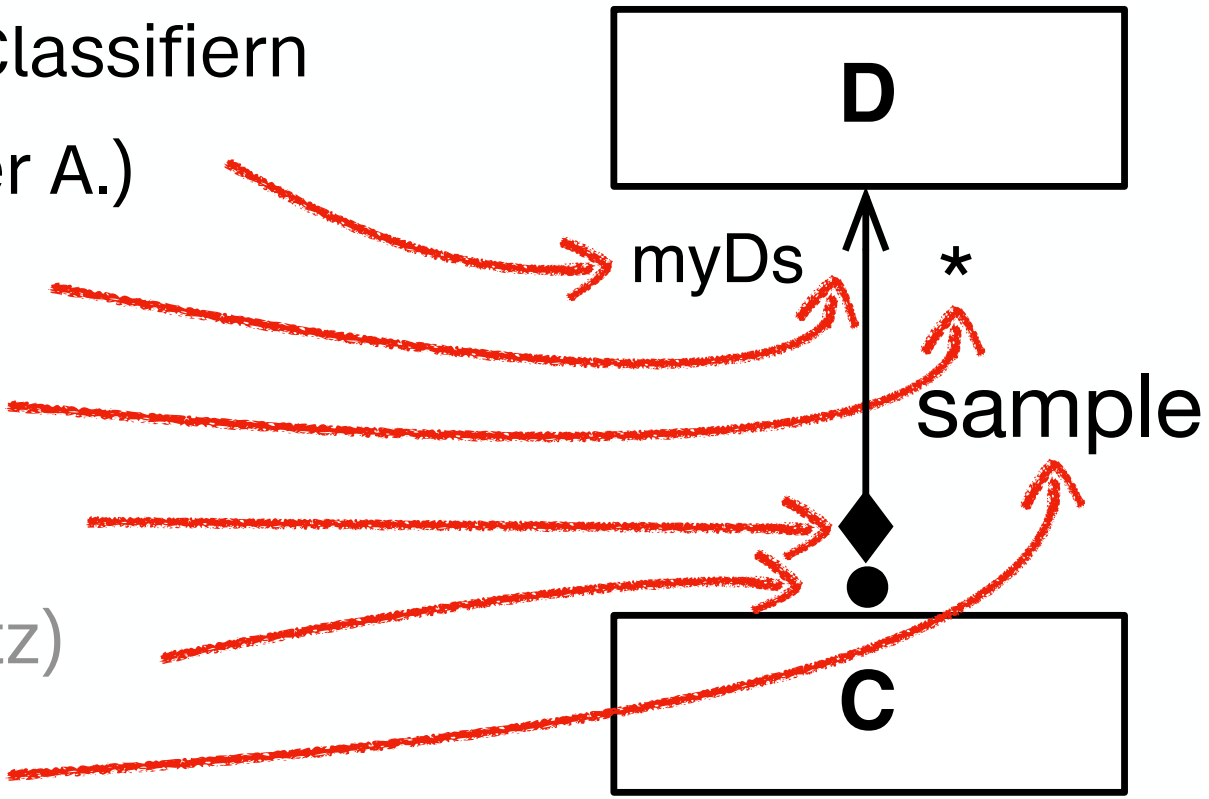
Navigierbarkeit

Multiplizität

Aggregationsart

Ownership (Besitz)

Name der A.



Multiplizität

0..1 — “optional”

1 — muss gesetzt sein

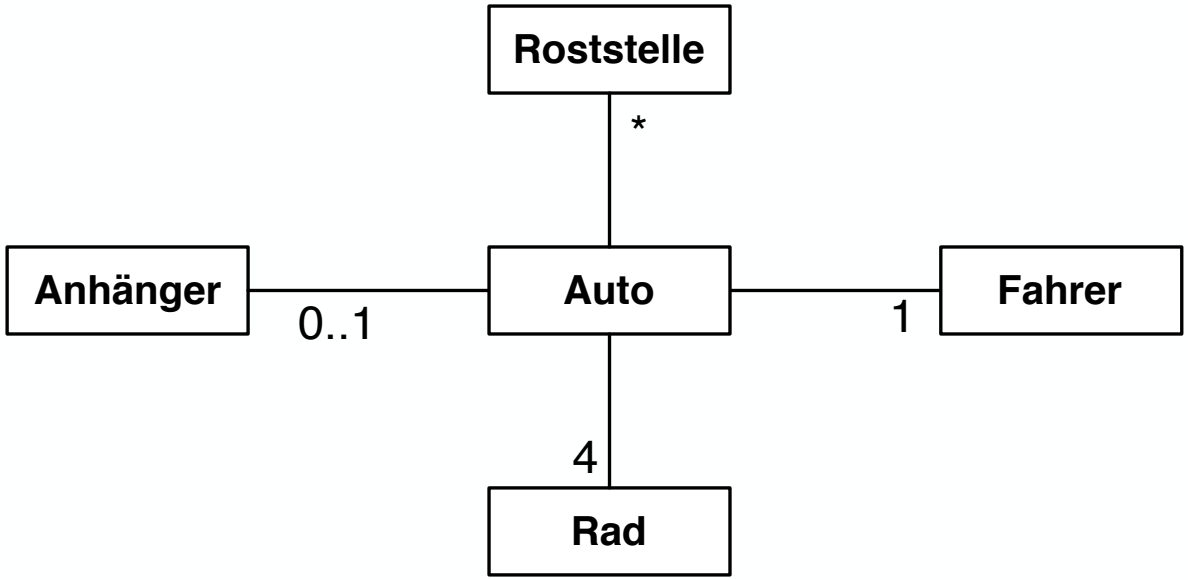
0..n oder * — beliebig viele

1..n — mindestens 1

4 — genau 4

nicht angegeben:

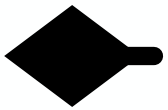
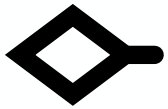
- undefiniert
- nicht anwendbar

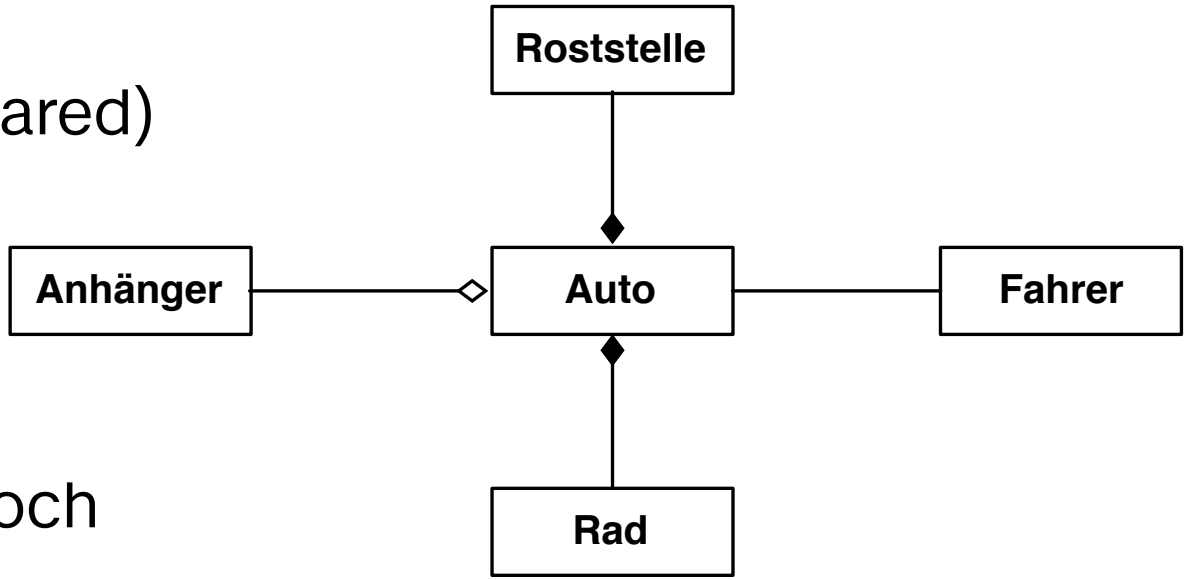


Aggregationsart

"Shared" gar nicht verwenden.

„Composite" sehr vorsichtig verwenden (im Zweifel: nicht verwenden)

-  Komposition (composite)
  - Teil (Part) in maximal einem Objekt enthalten
  - wenn Objekt gelöscht wird, muss auch Teil gelöscht werden
-  Aggregation (shared)
  - Semantik nicht genau definiert
- keine
  - weder Komposition noch Aggregation
  - oder einfach nicht definiert

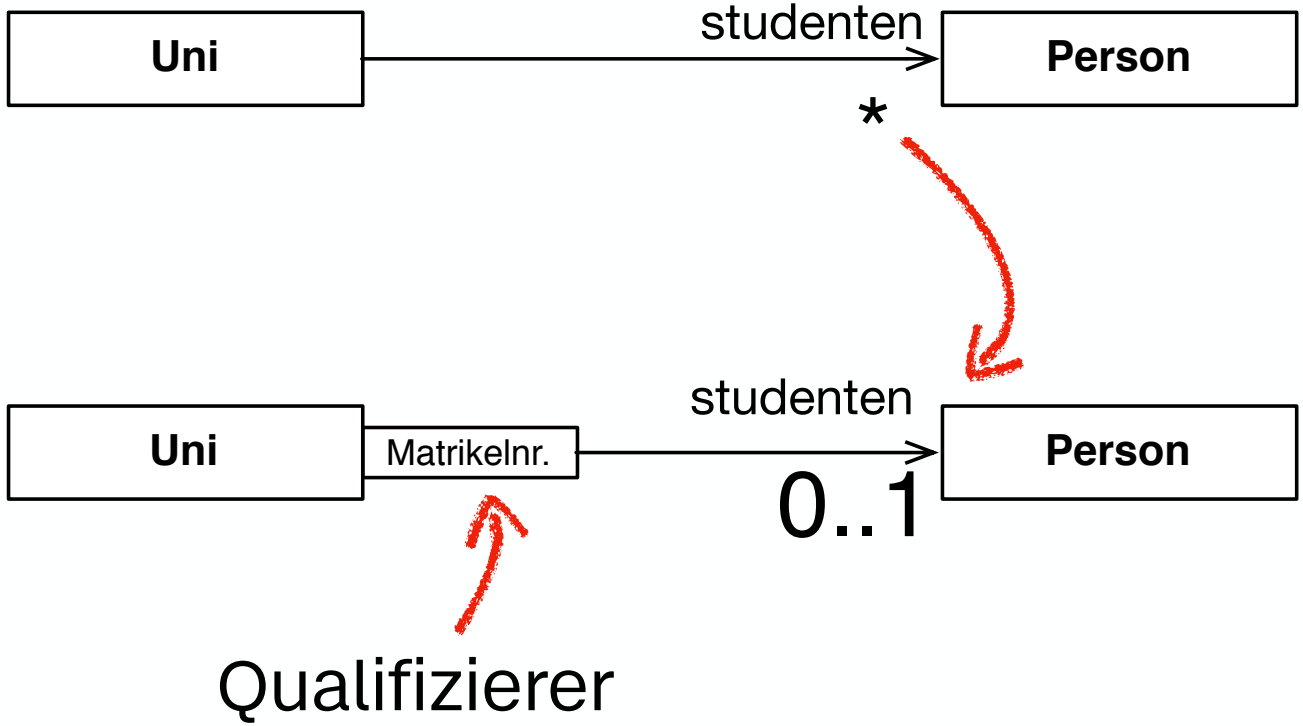


Qualifizierte Assoziation

Qualifizierer an einem Ende, definiert eine Partition

Multiplizität gilt dann pro Partition!

Mulitiplizität i.A. 0..1



Assoziation, typisches Mapping nach Java

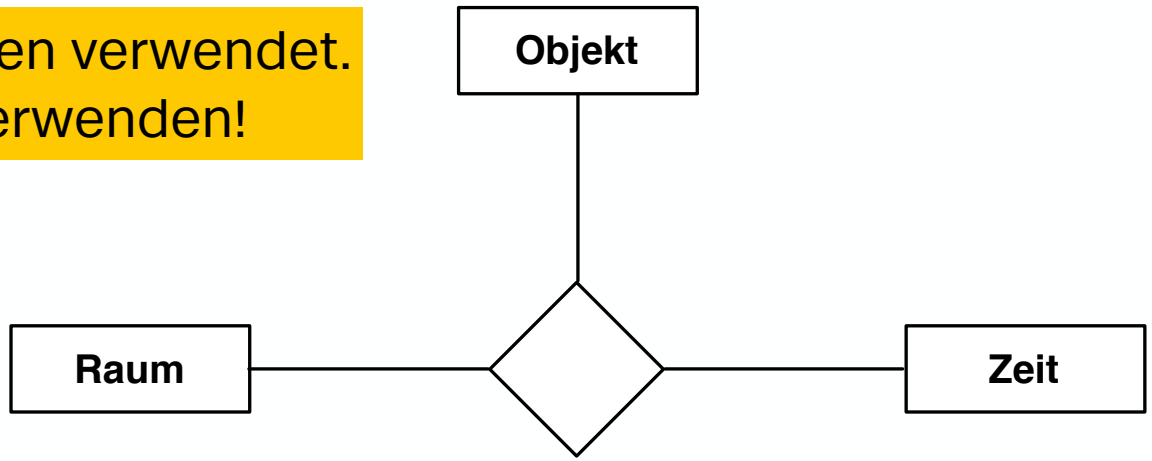
- Felder
- Multiplizitäten
  - 0..1: „normales“ Feld
  - 1: häufig im Konstruktor gesetzt, u.U. final
  - 0..n/1..n → i.a. durch Collections (Set, List)
- Composition: durch spezielle Logik der Getter/Setter zur Sicherstellung, dass Element in nur einem Objekt enthalten ist
- Rollen: Namen der Enden (Rollen) sind die Namen der Felder
- Qualifizierte Assoziationen → Map

n-näre Assoziation

Notation:

- Diamand (größer als Aggregation)

Wird generell sehr selten verwendet.
Am besten gar nicht verwenden!

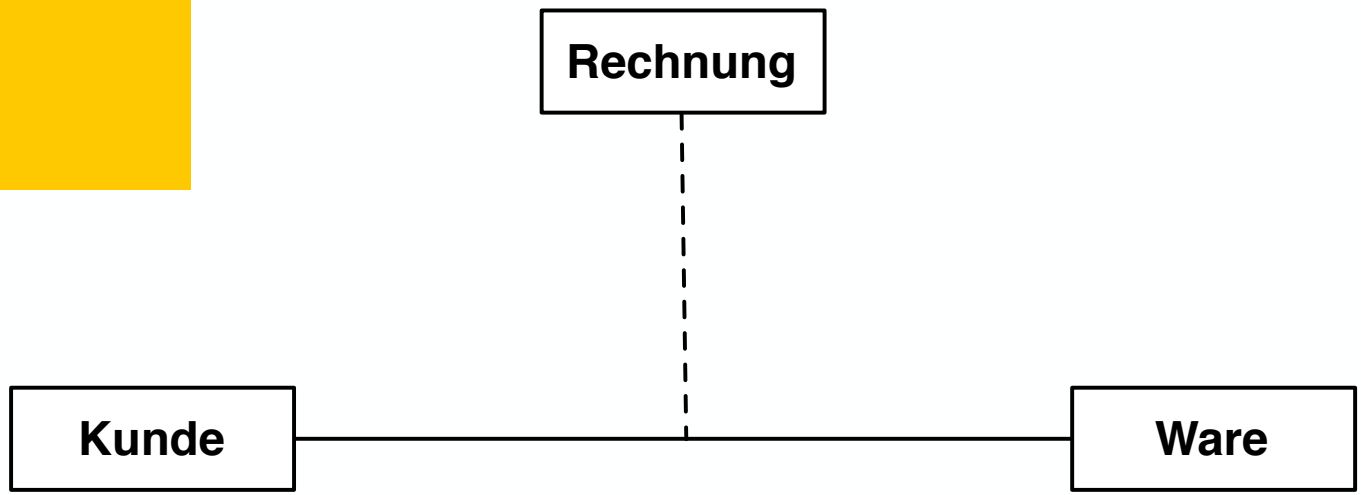


Assoziationsklasse

Assoziation mit Klasseneigenschaften bzw.

Klasse mit Assoziationseigenschaften

Wird selten verwendet, i.A. vermeiden.



Assoziationen: Rollen und Multiplizitäten

Berliner Hochschule für Technik

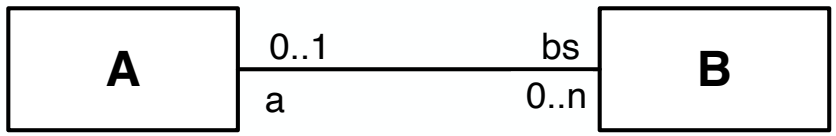
Assoziation: Multiplizitäten und Rollen auf der richtigen Seite der Assoziation eintragen!

- Wenn Multiplizitäten eingezeichnet werden, unbedingt auf beiden Seiten wenn die Assoziation bidirektional ist
- wie in Java: Attribute und Rollen (die ja quasi Attribute sind) klein schreiben

Die Rolle (mit Multiplizität) wird bzw. ist Attribut der Klassen auf der jeweils anderen Seite!

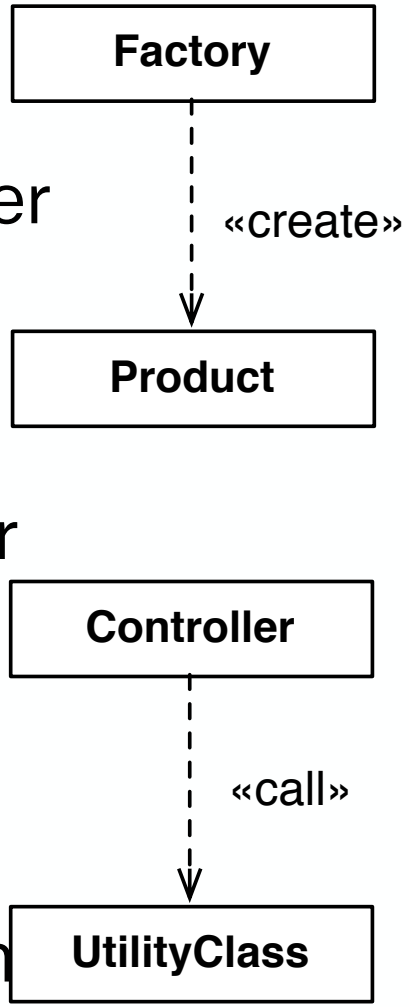
```
class A { List<B> bs; }
class B { A a; }
```

Welche Rolle spielt eine Klasse in einer Assoziation? Das steht direkt bei der Klasse, die die Rolle spielt! Oft heißt die Rolle wie die Klasse.



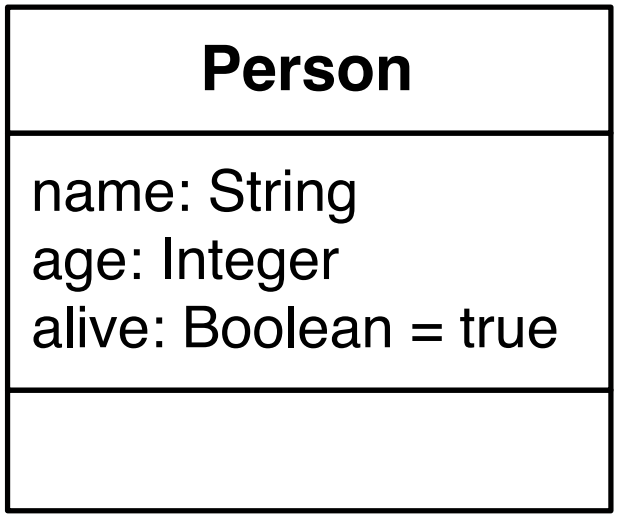
Dependency

- Notation:
  - gestrichelte Linie von Client zum Supplier
  - Pfeilspitze beim Supplier
- Semantik:
  - Client ist nicht vollständig ohne Supplier
  - mit Stereotyp weiter qualifiziert, häufig: «use», «create», «call»
- typisches Mapping nach Java
  - nicht explizit, ergibt sich aus Verwendun
  - sichtbar als “import”-Statement



Properties / Attribute

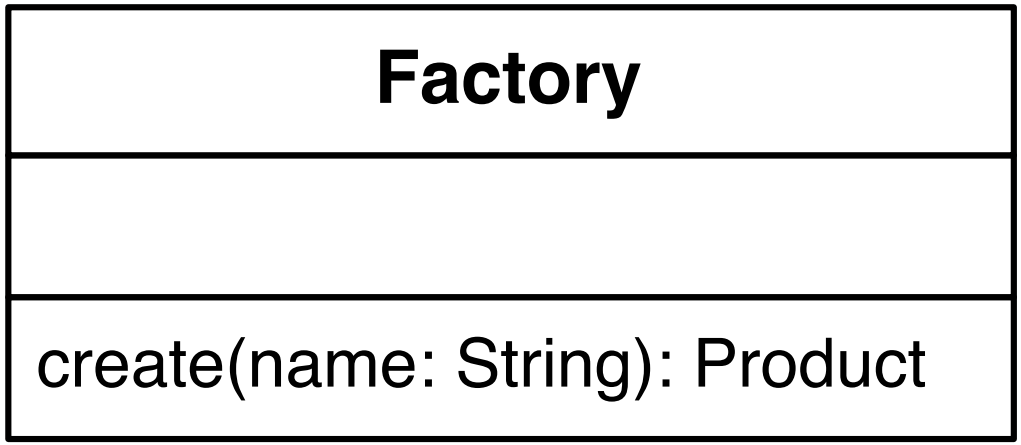
- Attribute eines Classifiers, entsprechen Feldern in Java
- stehen im Attribut-Compartment
- (unidirektionale) Assoziationen können als Attribute dargestellt werden
- weitere Angaben (je nach Detailgrad):
  - Typ
  - Multiplizität in eckigen Klammern
  - Default-Value



Primitive Typen in UML:
Boolean, Integer, Real, String, UnlimitedNatural
Sie können aber auch einfach Java-Typen verwenden.

Operationen

- entsprechen Methoden in Java
- Syntax ähnlich wie in Java
- keine Implementierung!
- stehen im Operations-Compartment



Visibility (Sichtbarkeit)

- Sichtbarkeit von Classifiern (Klassen, Interfaces), **Attributen**, **Assoziationen** und **Operationen**
- Arten:
  - public (“+”): überall sichtbar, wie in Java
  - private (“-“): nur innerhalb eines Classifiers bzw. besitzenden Namespaces, ähnlich wie in Java
  - protected (“#”): sichtbar für Elemente mit Generalisierungsbeziehung; wie in Java, beinhaltet aber nicht die package-Visibility
  - package (“~”): sichtbar für Elemente im gleichen Package, ähnlich wie in Java

Weitere Eigenschaften von Elementen

- **abgeleitet**: Attribute und Assoziationen können als abgeleitet (berechnet) markiert werden mittels vorangestellten Schrägstrich “/“.
- **statische** Attribute, Assoziationen und Operationen werden unterstrichen dargestellt
- **abstrakte** Classifiers (insbesondere Klasse) werden kursiv dargestellt oder mit “{abstract}” annotiert.

Beschränkungen (Constraints)

Berliner Hochschule für Technik

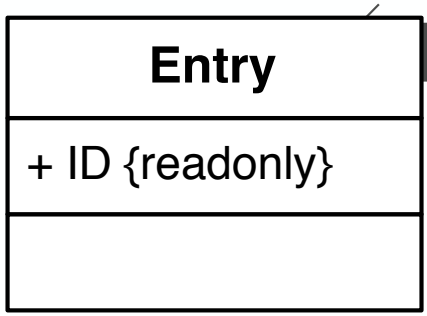
Constraints und einige weitere Eigenschaften von Attributen, Assoziationen und Operationen werden in geschweiften Klammern „{..}“ hinter den Elementen dargestellt.

Constraints können in OCL oder einer anderen Sprache (bspw. Java) formuliert werden.

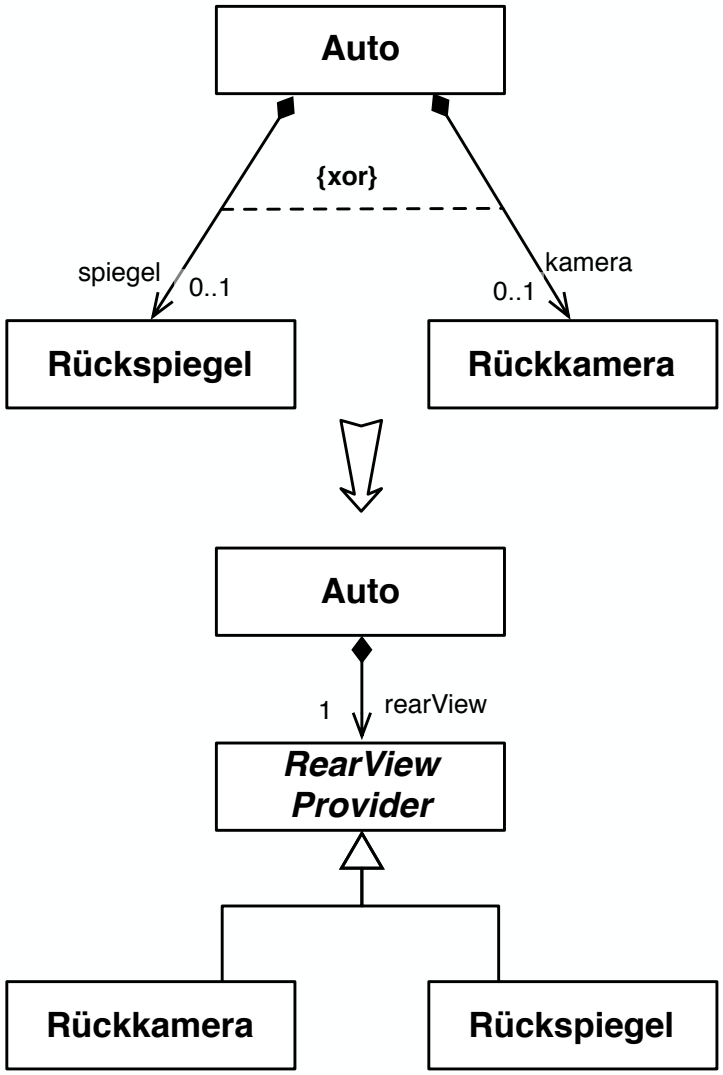
Einige vordefinierte Eigenschaften:

- Properties/Assoziationen: readOnly, union, subsets ..., ordered, unordered, unique, sequence, id
- Operationen: query, ordered, unordered, unique

Constraints Beispiele



- Häufig bei Attributen
  - praktisch auch, um spezielle Eigenschaften einer Sprache abzubilden, wie final in Java
- Sehr nützlich bei Assoziationen, v.a. {xor}
- Insgesamt sollte man es nicht übertreiben! Tipp: Häufig findet sich eine Art der Modellierung, die das Constraint automatisch garantiert.
  Bspw. über Basisklassen.



Namespaces

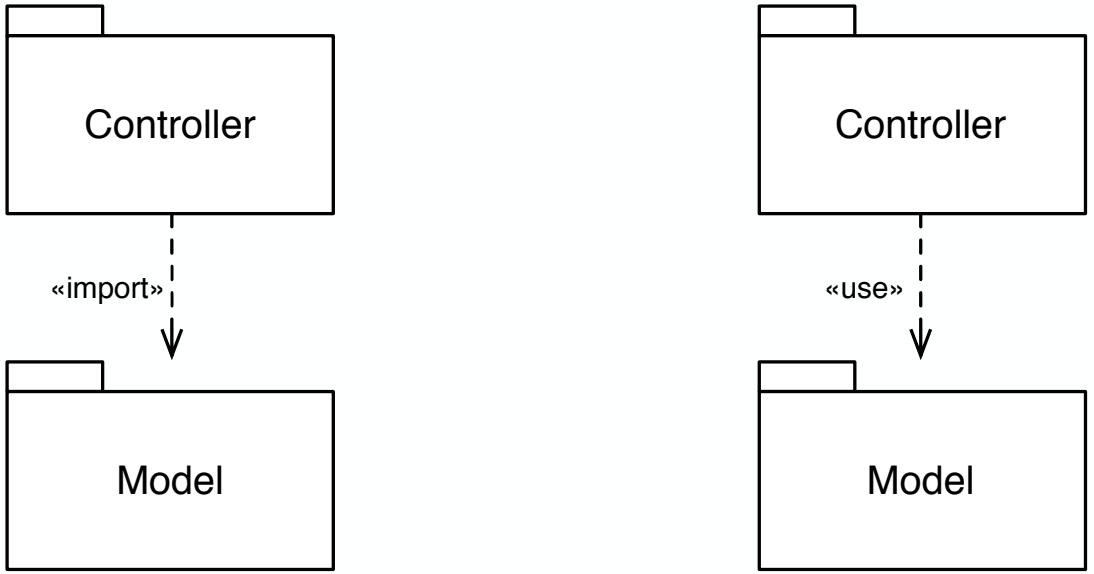
- Ein „Namesspace“ ist einfach ein Element, dass andere Elemente, seine Member, enthält
- enthaltene Elemente werden über den Namen identifiziert
- typische Namespaces:
  - Packages
  - aber auch allgemein: Classifiers
- qualified name (qualifizierter Name):
  - qualifiedName ::= container.qualifiedName ’::’ name
  - bspw. „my::package::sub::MyClass::NestedClass“
  - pragmatisch: häufig Trennung durch „.“ wie in Java

Notation von Namespaces

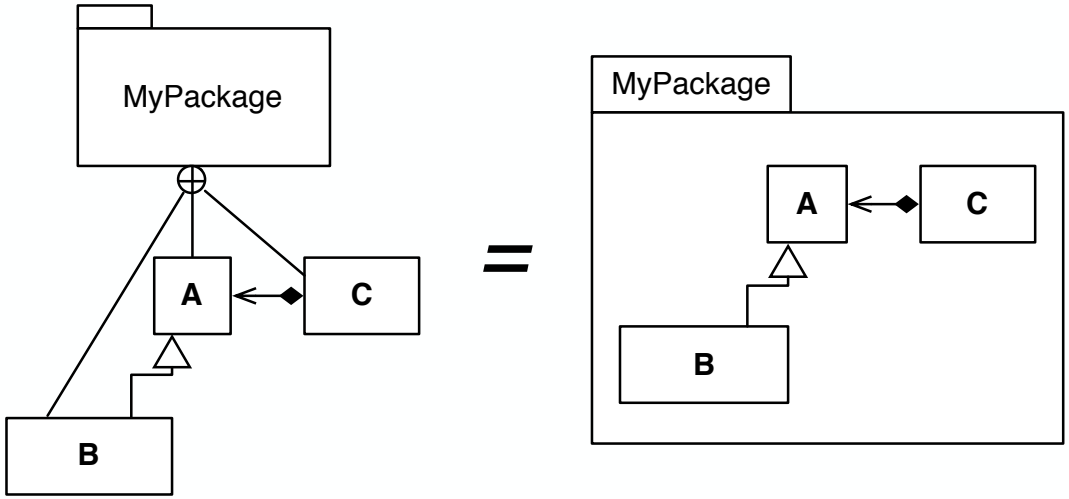
- keine allgemeingültige Notation
- Member werden entweder
  - in einem speziellen Compartment gezeichnet oder
  - mittels „circle-plus“-Notation zugeordnet

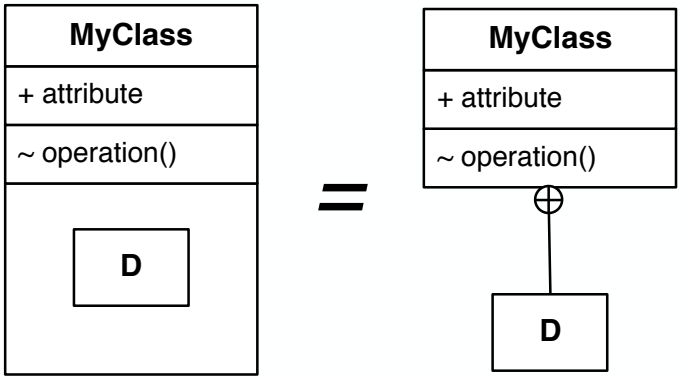
Packages

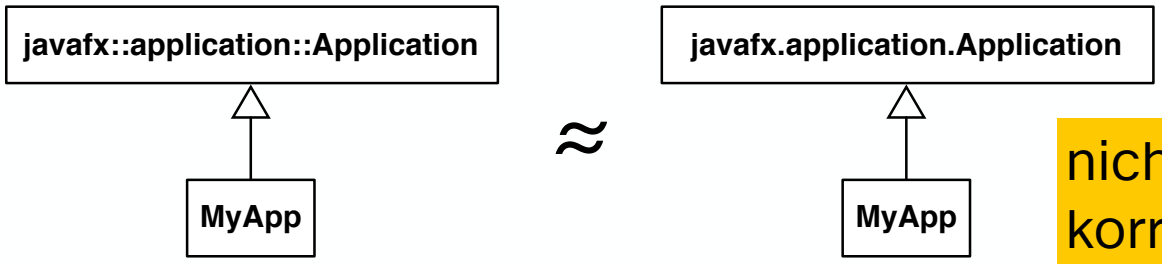
- Package ist Namespace für seine Member
- ist selbst ein Classifier
- kann andere Packages importieren
  - ähnlich wie in Java, nur auf Paketebene



Notationsbeispiele





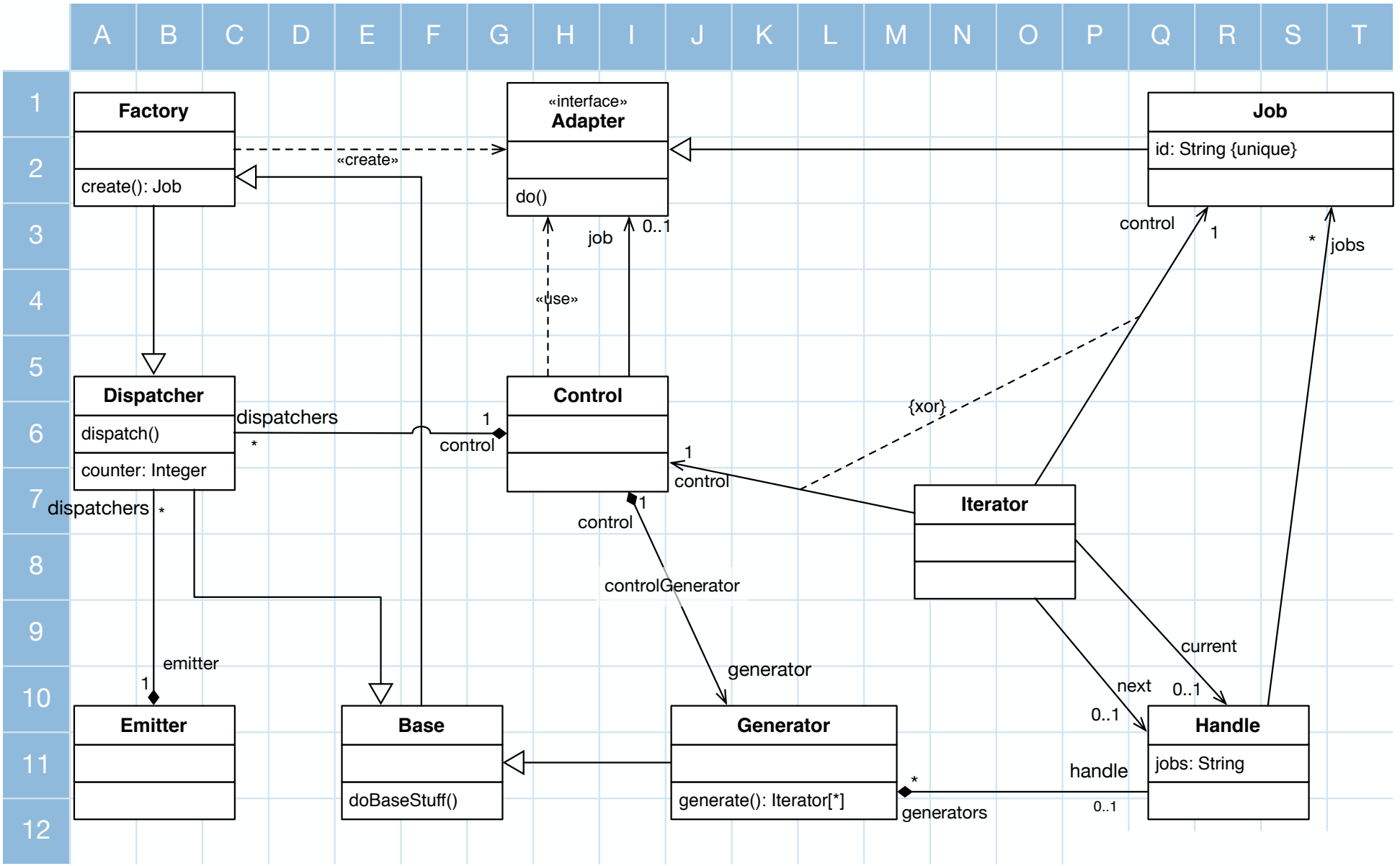


nicht 100% korrekt, aber pragmatisch

Übung

Wer findet die Fehler?

Zeit	Aktion
10 min	*Jeder einzelne erarbeitet still eine Lösung*
10 min	*Lösung und Diskussion*

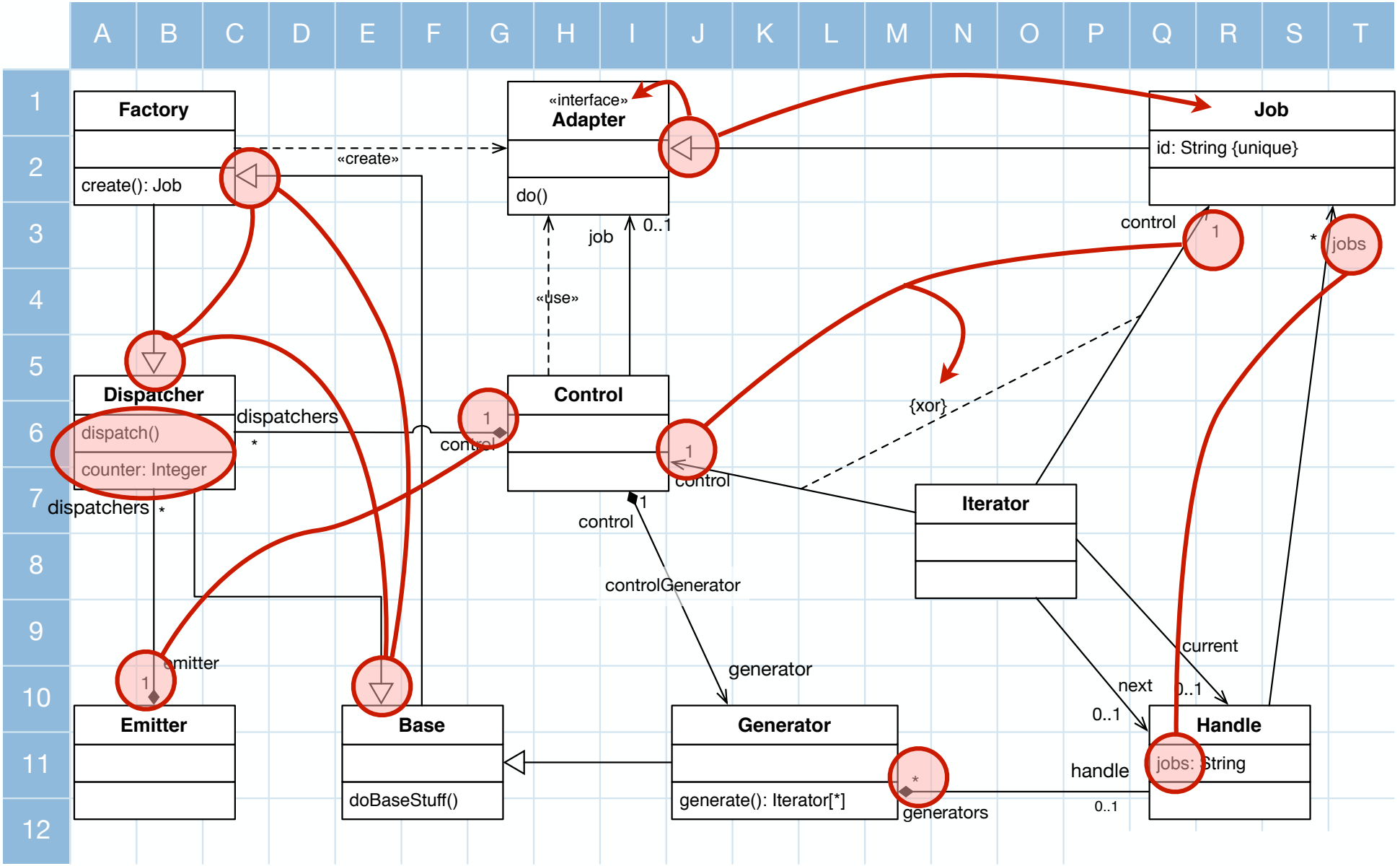


7 Fehler

Es gibt noch viele andere „Ungereimtheiten“, die aber auf fachlicher Ebene angesiedelt sind! Diese sind hier nicht markiert.

Berliner Hochschule für Technik

Lösung



Domänenklassenmodell

- mittels UML-Klassendiagramm und zusätzlicher textueller Spezifikation
- Jedes Element sollte ein Pendant in der Problemwelt (Domain) besitzen.
  - In den Anwendungsfällen sollte auf diese Elemente Bezug genommen werden!
  - v.a. bei abstrakten Klassen und Generalisierung prüfen
  - Klassen sollten Attribute besitzen
  - Ausnahmen sollten begründet werden
- bei größeren System sollten Pakete verwendet werden

Tipp: Im Grunde ist das Domänenmodell das, was man auch als ER-Modell erstellen könnte zur Speicherung in der Datenbank

Detailgrad im Domänenmodell

Operationen werden noch nicht spezifiziert

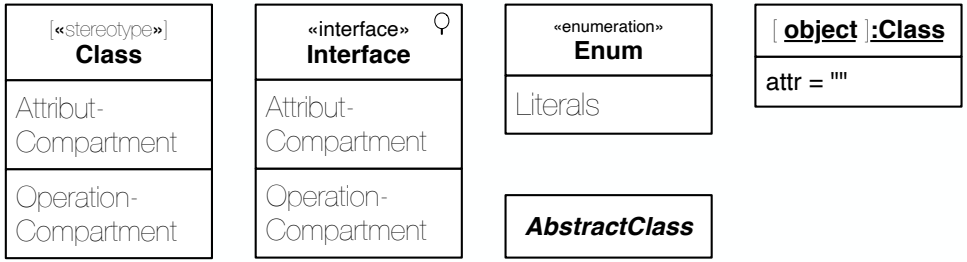
Sichtbarkeit ist überall „public“, kann daher weggelassen werden

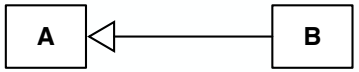
Assoziationen:

- angeben:
  - Rollennamen und Multiplizitäten auf der Seite, wo dies aus der Domäne heraus klar wird (nichts ausdenken)
  - Aggregationsart (composite, none) soweit bekannt, im Zweifel nur none.
- Navigierbarkeit sollte weggelassen werden, außer sie ist sehr wichtig

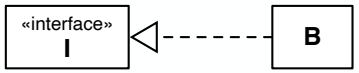
abgeleitete Attribute und Assoziationen entsprechend markieren

Cheat Sheet Class Diagram

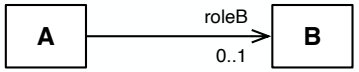




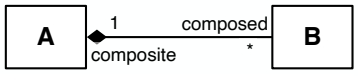
Generalization: B extends A



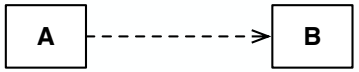
Realization: B implements I



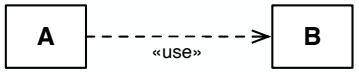
Association: unidirectional,
Java: class A { B roleB; }



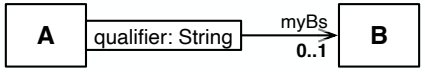
Composed Association: bidirectional,
A is composed of B
(e.g., B is destroyed when A is destroyed)



Dependency: A depends on B



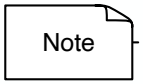
Dependency with stereotype, standard stereotypes:
use, **instantiate**, **create**, **call**, **responsibility**



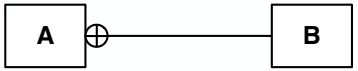
Association with Qualifier:
Java: class A { Map<String, B> myBs; }

attribute ::= [visibility] [derived] name [**:** type] [multiplicity]
[**=** default] [propmodifiers] [**{** constraints **}**]
operation ::= [visibility] name [**(** parameters **)**] [**:** type] [multiplicity]
[opmodifiers] [**{** constraints **}**]
visibility ::= **+** (public) | **-** (private) | **#** (protected) | **~** (package)
multiplicity ::= [lower **..**] upper | ***
upper, lower ::= number
contraint ::= *OCL expression, Java expression, pseudo code*
propModifiers ::= **readOnly** | **union** | **subsets** name
| **ordered** | **unordered**
| **unique** | **sequence** | **id**
derived ::= **/**
opModifiers ::= **query** | **ordered** | **unordered** | **id**
parameter ::= name **:** type
name ::= *string*
type ::= *fully qualified or simple type name, basic types:*
Boolean, **Integer**, **Real**, **String**
default ::= *value expression*

- Lists (plural) are comma separated
- static attributes / operations are undelined
- role (association end) similar to attribute



Comment, can be attached to arbitrary elements



Package-relation (e.g., B inner class of A)

BHT Berliner Hochschule für Technik

Verhaltensmodellierung mit State Machines

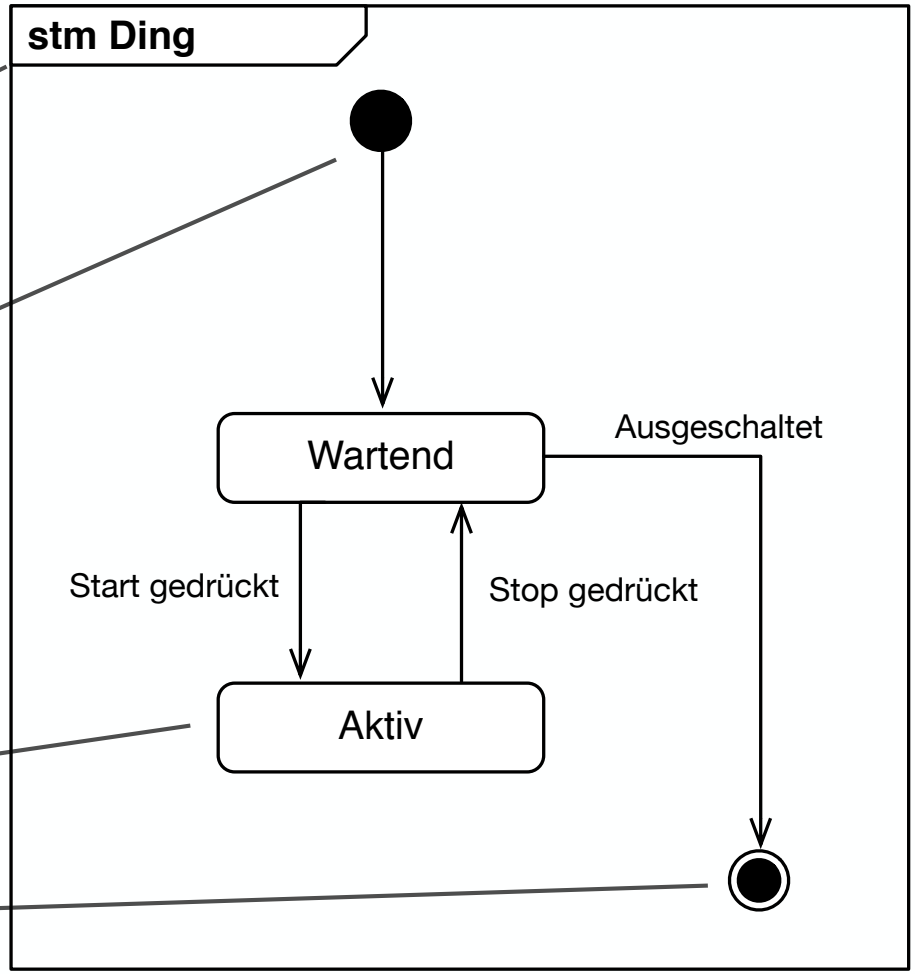
State Machine Diagrams (Zustandsdiagramme)

Berliner Hochschule für Technik

- beschreiben den Zustand eines Classifiers, also
  - eines ganzen Systems
  - eines Use Cases — und sind damit ähnlich wie Aktivitätsdiagramme zur Verhaltensbeschreibung von Use Cases geeignet
  - einer Klasse (insbesondere eine Kontrollklasse aus der Analyse)
- sind eine Erweiterung finiter Automaten
- können zusammengesetzt werden

UML State Machine Diagram

- besteht im Wesentlichen aus
  - Diagrammrahmen mit Kürzel „stm“
  - Knoten:
    - Startknoten (und andere „Pseudo-states“)
    - State (Zustand)
    - Endknoten
  - Kanten: Transitionen



Vorsicht: In dieser Variante kaum von Aktivitäten zu unterscheiden!

Konzeptionelle Annahmen

- State Machine befindet sich zu einem gegebenen Zeitpunkt in genau einem Zustand.
- Zeitdauer der Transitionen nicht definiert!
- Es gibt eine endliche (finite) Anzahl von Zuständen, daher sprechen wir auch von „finite state machines“ (FSM).
- **Event-Driven**:

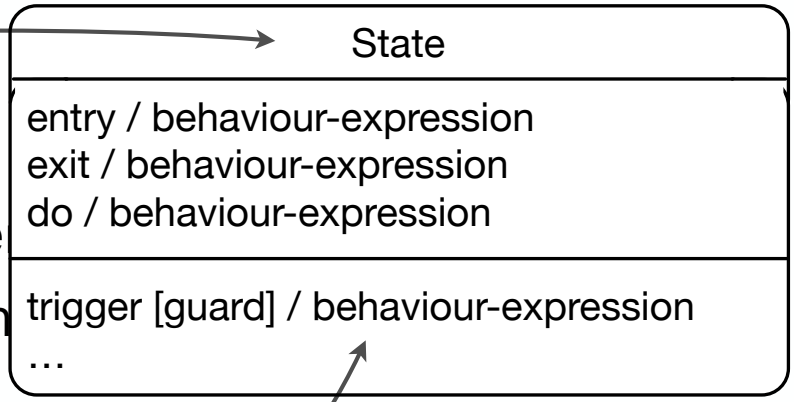
  Aktueller Zustand + Event → Neuer Zustand

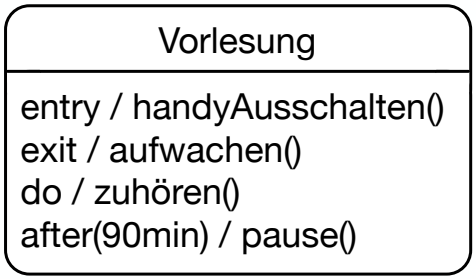
  - Events lösen sogenannte Trigger aus

Compartements

Tipp zu den Namen: Es handelt sich um Zustände, nicht um Aktionen! In Englisch durch Gerund („ing“-Form)

- **Name**
- **Internal Behaviors** (optional):
  - entry: Eintrittsverhalten, wird ausgeführt, we
  - exit: Austrittsverhalten, wird ausgeführt, wen
  - do: Zustandsverhalten, wird während des Zustands ausgeführt. Startet nach Entry-Behavior, wird beendet wenn von selbst fertig oder Zustand verlassen wird.
- **Internal Transitions** (optional):
  - (internes) Verhalten, das aufgrund von Triggern ausgeführt wird.
  - Dabei wird der Zustand nicht verlassen, insbesondere werden entry- und exit-Verhalten nicht ausgeführt!

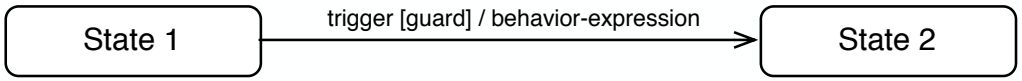




Internal Compartments werden meist ohne weitere Trennung dargestellt.

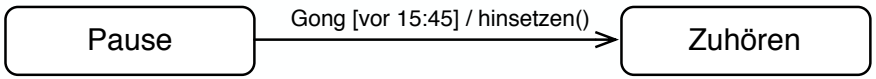
Transitionen

„A Transition represents an arc between exactly one source Vertex and exactly one Target vertex (the source and targets may be the same Vertex).“ [OMG17a]



Beschriftung:
[<trigger> (‚’ <trigger>)* [‘[‘ <guard> ’]’] [‘/’ <behavior-expression>]]

- Trigger: Werden durch Event ausgelöst
- Guard: boolscher Ausdruck (Bedingung)
- Behavior-Expression: Verhalten während der Transformation



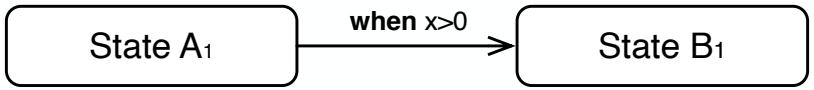
Trigger und Events

„A Trigger specifies a specific point at which an Event occurrence may trigger an effect [..].“ [OMG17a, p. 300]

Event	Beschreibung	Notation am Trigger
CallEvent	*Anfrage, dass eine Operation (des Classifiers) aufgerufen werden soll.*	doSomething() →
SignalEvent	*Empfang eines (asynchronen) Signals*	someSignal →
ChangeEvent	*Tritt auf, wenn eine **Bedingung true wird.***	**when** x>0 →
TimeEvent	*Erreichen einer definierten Zeit*	**at** 15:45 → **after** 90 min →
AnyReceiveEvent	*Beliebieges Call- oder SignalEvent*	**all** →

Problem mit Bedingungen bei Change Events

Das Problem: Ein Change Event tritt nur auf, wenn sich eine Bedingung **ändert**.



D.h. wir können über ein Change Event nicht einfach einen Art „Verzweigung“ aufgrund eine Bedingung formulieren:

- wenn die Bedingung beim Betreten von State A1 bereits erfüllt ist, wird das Event niemals ausgelöst!

Completion Event und Completion Transition

Berliner Hochschule für Technik

Eine Transition ohne Trigger wird implizit getriggert über das sogenannte Completion Event.

Sie wird daher als Completion Transition bezeichnet.

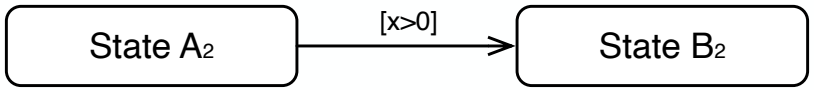
Das Completion Event wird ausgelöst, nachdem ein State betreten wurde und alle entry- und do-Behaviours beendet wurden.

Eine Completion Transition kann auch einen Guard haben!

Leider ist die UML-Spec nicht ganz klar bzgl. der Completion Transition. Sie wird in der Notation nicht beschrieben und die Beschreibung der Trigger-Notation läßt einen Guard ohne (expliziten) Trigger eigentlich nicht zu. Sachdienliche Hinweise werden dankbar entgegengenommen!

Change Event vs. Completion Transition mit Guard

Ein „Workaround“ für das Problem mit den Change Events ist nun die Verwendung einer Completion Transition mit einem Guard.

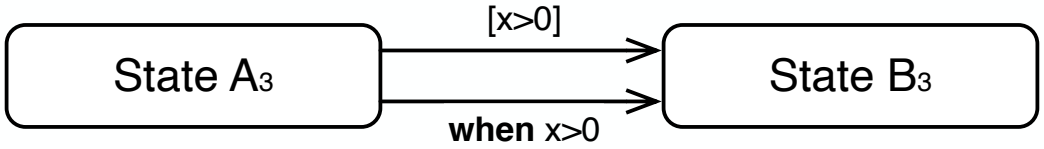


Allerdings: Hier haben wir ein Problem, wenn x beim betreten kleiner oder gleich 0 ist...

Warnung: Diese Folie ist „meine“ Interpretation zur Lösung eines in der Praxis leider häufiger auftretenden Problems...

Change Event vs. Completion Transition mit

Eine vollständige Lösung würde in der Verwendung von zwei Transition bestehen:

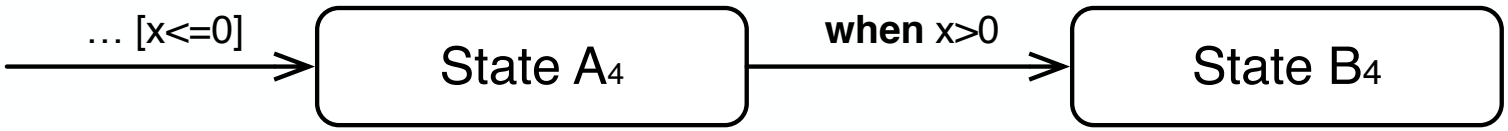


Das ist aber nun wirklich nicht schön...

Radikale Lösung: Konsistenz prüfen

Die vermutlich ordentlichste Lösung besteht allerdings darin, bei Entwurf der State Machine dafür zu sorgen, dass solche Fälle nicht eintreten können.

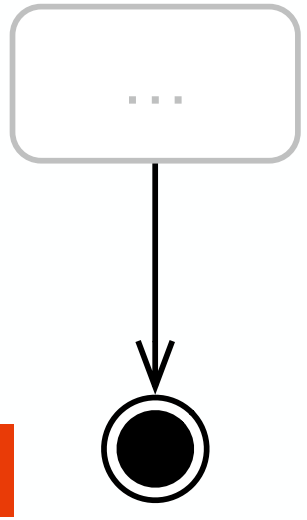
Dies ist die aus theoretischer Sicht beste Lösung.



Wir werden später sehen, wie wir sinnvoll Tests für State Machines erstellen können. Diese können uns dann beim Entwurf der State Machine, ggf. mit einem Simulator, helfen!

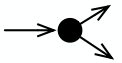
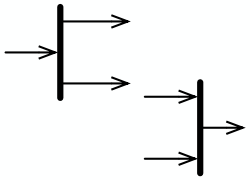
Endknoten (final state)

- spezieller State
- darf keine ausgehenden Transitionen haben
- darf kein Verhalten haben (entry, do, internal triggers)
- kann 0-n mal vorkommen



Achtung: Kein Pseudozustand!

Pseudo States (Pseudozustände)

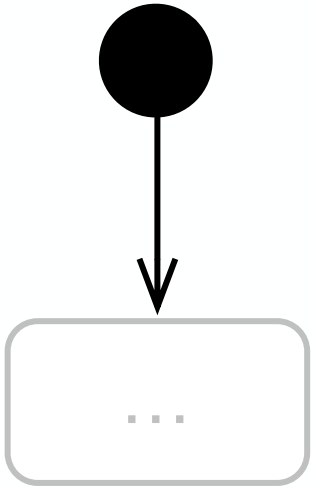
- im Wesentlichen zur Verbesserung der Darstellung
- System kann niemals in einem „pseudo state“ sein
- u.a. folgende Arten:
  - initial: Startzustand 
  - junction: Kreuzung 
  - choice: Entscheidung 
  - fork: Gabelung 
  - join: Vereinigung
  - entry: Eintrittspunkt 
  - exit: Austrittspunkt 

v.a. in Kombination mit Regions interessant, lassen wir (erstmal) weg

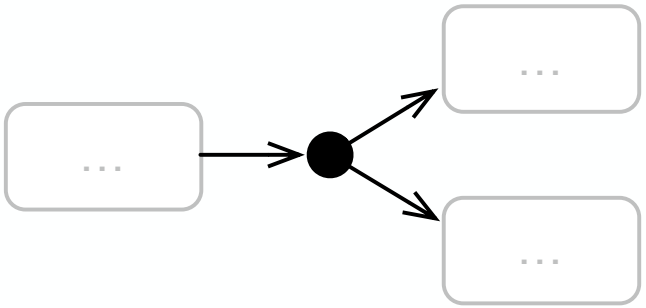
- je nach Art unterschiedliche Trigger, Guards und Transitionen erlaubt

Startzustand (initial state)

- jede State Machine muss genau einen Startzustand haben
- darf keine eingehenden Transitionen haben
- muss genau eine ausgehende Transition haben
  - diese darf keinen Trigger oder Guard haben



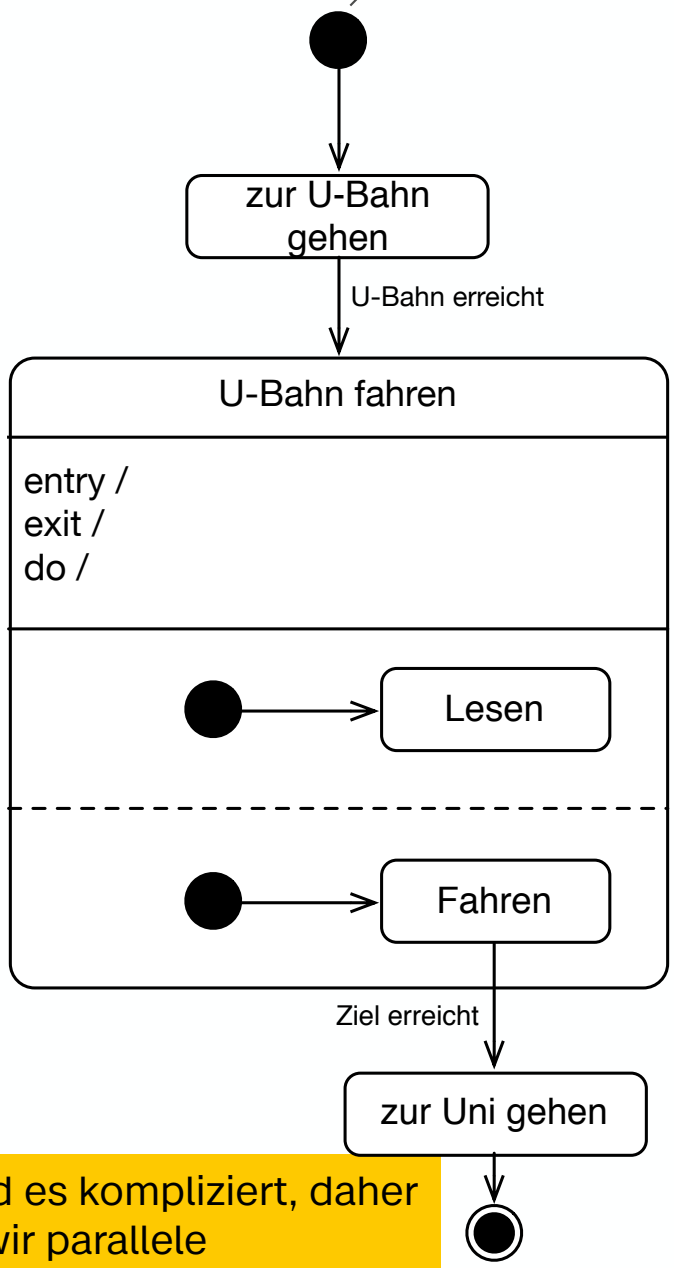
Junction



- mindestens je eine Eingangs- und Ausgangstransition
- **Junction**:
  - hat keine eigene Semantik
  - fasst Transitionen zusammen, die als Kombination der eingehenden- und ausgehenden Transitionen (mit deren Knoten, Trigger und Guards) berechnet werden können; dabei dürfen keine Widersprüche auftreten

Composition State

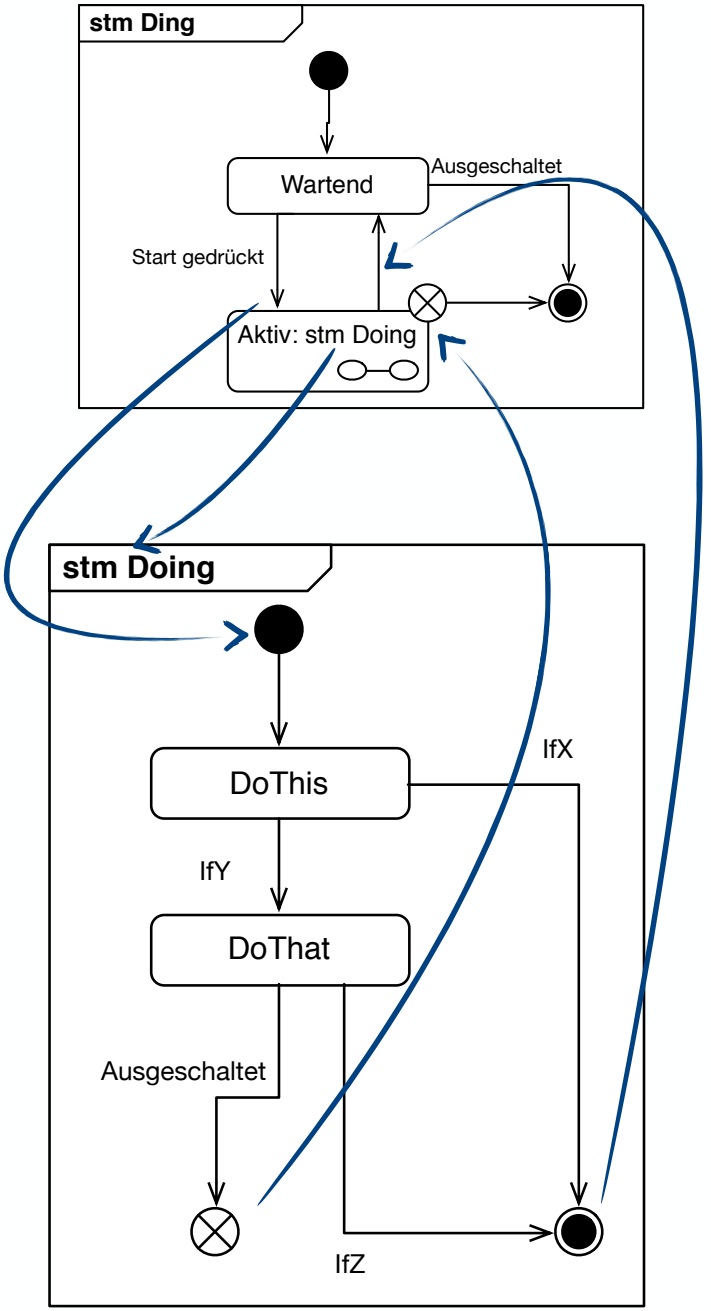
- State kann auf mehrere State Machines, sogenannte Regions, aufgeteilt werden.
- States der Regions dürfen nicht miteinander verbunden werden
- Regions werden parallel ausgeführt
- State wird verlassen, wenn alle Regions verlassen wurden oder eine Transition eines Sub-States explizit nach außen geht.
- beim Betreten/Verlassen kommen fork und join ins Spiel



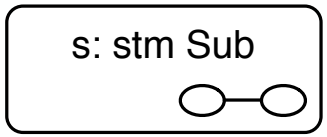
Hier wird es kompliziert, daher lassen wir parallele Verarbeitung außen vor. Diese Folie also nur als Teaser, was theoretisch möglich wäre.

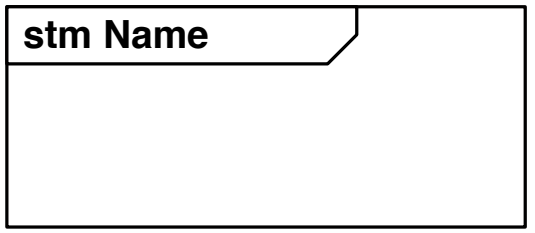
Submachine State

- wie normaler State, Name-Compartment enthält Name der referenzierten State Machine:
  <state-name> ':' <name-of-referenced-StateMachine>
- Betreten und Verlassen der Submachine über „Standard"-Knoten:
  - initial state beim Betreten des Submachine State
  - Submachine State kann verlassen werden, wenn Submachine den final state erreicht hat
- oder über Connection Point References:
  - entry: beim Betreten des Submachine States wird die ref. SM dort gestartet
  - exit: beim Verlassen der referenzierten SM wird der Submachine State dort ebenfalls verlassen

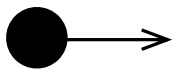


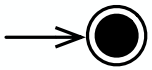
Cheat Sheet UML State Machine

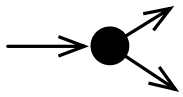


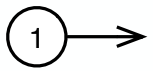


trigger ::= signalEvent | changeEvent
| timeEvent | callEvent
| anyReceiveEvent
signalEvent ::= name of signal
changeEvent ::= **when** *expression*
timeEvent ::= **at** *time* | **after** *duration*
anyReceiveEvent ::= **all** (else)
guard ::= expression
behavior-expression ::= call expression
(call) expression ::= OCL expression,
Java expression, function call

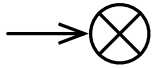

Initial Node


Final Node


Junction



Entry


Exit

BHT Berliner Hochschule für Technik

Verhaltensmodellierung mit Aktivitätsmodellen

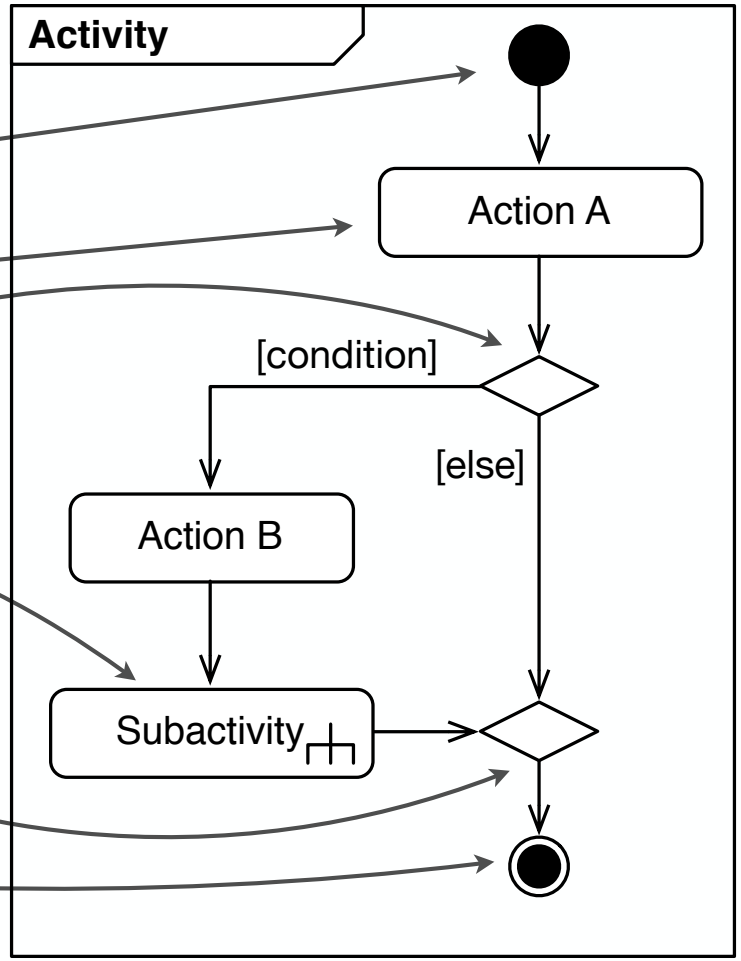
Aktivitätsdiagramm

- Statt textueller Notation der Flows/Szenarien können wir den Ablauf modellieren
- Vorteil:
  - man „sieht“ den Ablauf
  - fehlende Flows können schnell erkannt werden
- Nachteil:
  - häufig schon sehr detailliert
  - nicht trivial zu modellieren
- Ein „Szenario“ (oder Flow) entspricht dann einem Pfad vom Startknoten zu einem Endknoten

UML Activity Diagram

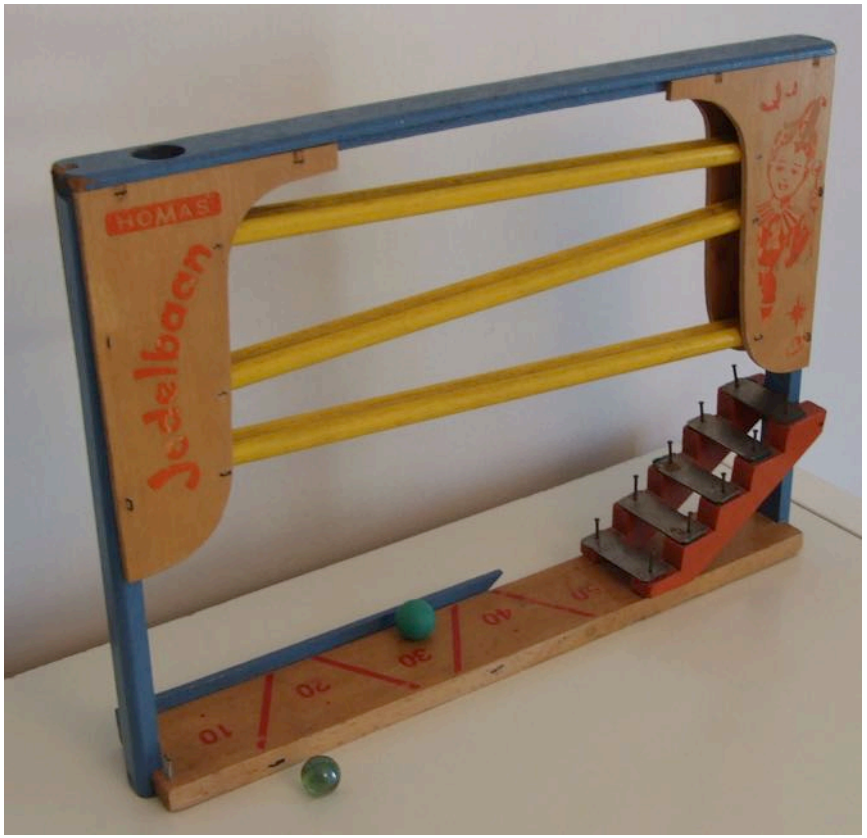
besteht im Wesentlichen aus

- Knoten:
  - Startknoten
  - Aktionen
  - Entscheidungsknoten
  - Subaktivitäten
  - Zusammenführungsknoten
  - Endknoten
- Kanten (Activity Edges)



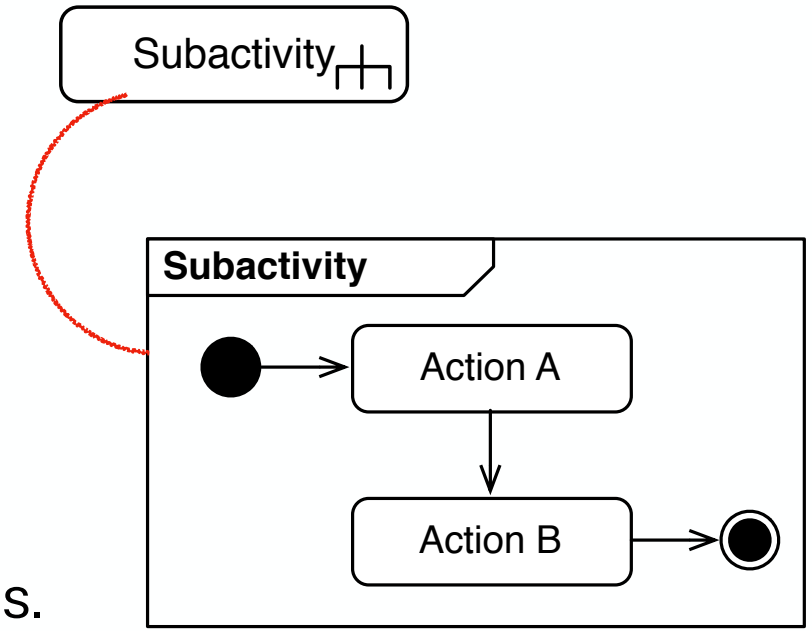
Tokenkonzept

- „Ausführung“ der Aktivität wird über Tokens modelliert
- Tokens werden nur implizit modelliert
- wurde von Petri-Netzen übernommen
- gut zur Modellierung paralleler Prozesse!
- Tokens werden von einer Aktivität oder Aktion zur nächsten gereicht.
- Tokens müssen notfalls bei Action/Subaktivität warten (siehe nächste Folien)



Aktionen und (Sub-)Aktivitäten

- Eine Aktion (Action) kann eine beliebige Anweisung sein. Sie wird mit abgerundeten Kanten gezeichnet.
- Größere „Aktionen“ sollten als Aktivität modelliert werden. Sie können dann als „Sub-Aktivität“ in anderen Aktivitäten wie Aktion verwendet werden. Man erkennt sie am „Gabel“-Symbol. Der Name der Subaktivität ist der Name des Diagramms.

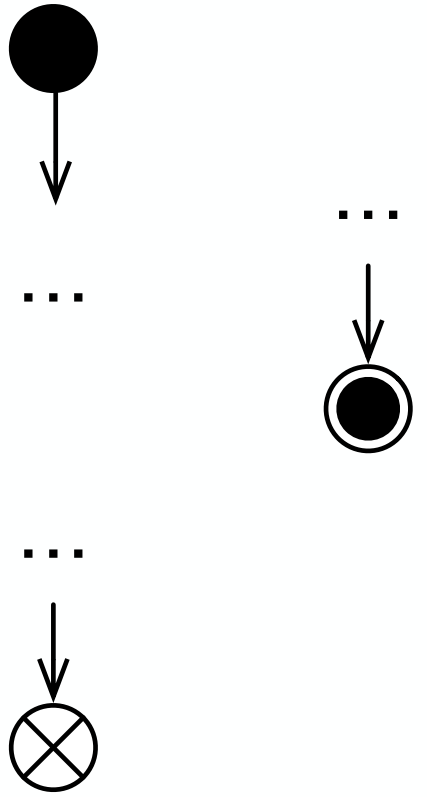


- Eine Aktion startet, wenn sie ein Token erreicht. Sie **nimmt das Token dann auf und reicht es, falls möglich, nach Beendigung der Aktion, weiter. Falls nicht möglich, bleibt das Token bei der Action/ Subaktivität.**

Wenn eine Aktion „läuft“ (also ein Token aufgenommen hat), kann sie i.A. keine weiteren Tokens aufnehmen. Dies ist erst möglich, wenn sie das aktuelle Token weitergegeben hat.

Kontrollknoten (1/2)

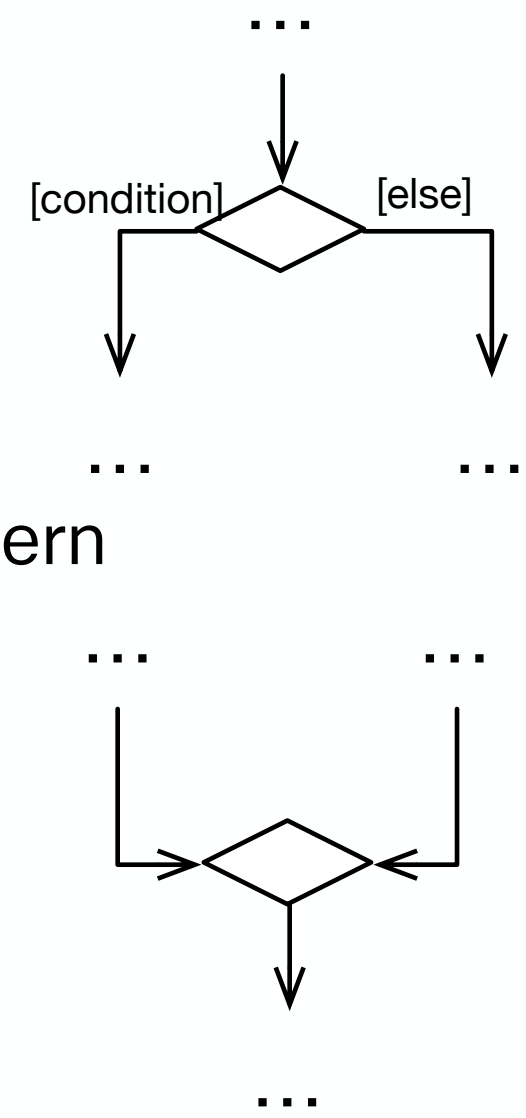
- **Initial node** (Startknoten):
  Startet Aktivität und bringt
  Token ins Spiel
- **Activity final node**
  (Aktivitätsendknoten):
  Beendet Aktivität und löscht alle Tokens
- **Flow final node**
  (Endknoten für Kontrollfluss):
  Terminiert nur den Fluss durch
  Vernichtung des Tokens, andere Tokens
  bleiben unberührt.



Tokens „verweilen“ nicht auf Kontrollknoten!

Kontrollknoten (2/2)

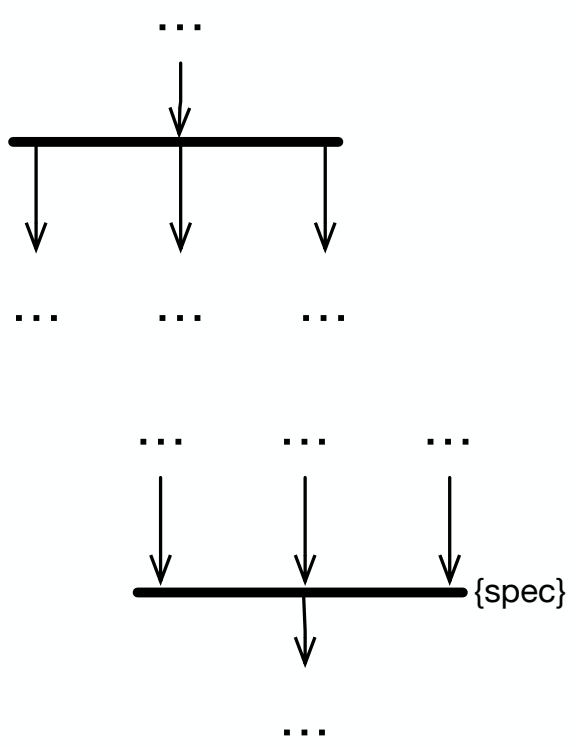
- **Decision node**
  (Entscheidungsknoten):
  Token muss den Knoten auf genaue einer Kante verlassen, die Kanten tragen Bedingungen in eckigen Klammern



- **Merge node**: Führt Flüsse zusammen, Tokens werden nicht verschmolzen!

Tokens „verweilen“ nicht auf Kontrollknoten! Entscheidung ist „zeitlos“

Kontrollknoten für parallele Ausführung



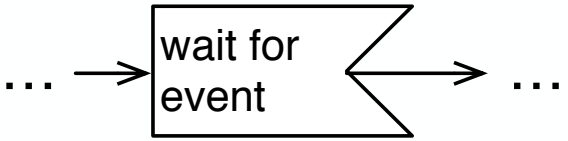
- **Fork Node** (Parallelisierungsknoten): vervielfacht Tokens entsprechen der ausgehenden Kanten, Kanten können Bedingungen haben
- **Join Node** (Synchronisationsknoten): verschmilzt eingehende Tokens entsprechend der JoinSpec (Synchronisationsspezifikation):
  - „and“: Standard, wenn nichts angegeben. Knoten „schaltet“ erst, wenn auf allen eingehenden Kanten Tokens anliegen. **Hier können ausnahmsweise Tokens gesammelt werden!**
  - „or“, „xor“, sonstige Bedingungen: Tokens werden mit „true“ interpretiert und Bedingung entsprechend ausgewertet.
- implizite Modellierung: mehrere Kanten, die aus oder in Aktionen/Subaktivitäten gehen, werden wie Fork bzw. and Join Node behandelt

Join und Fork bitte immer explizit notieren!

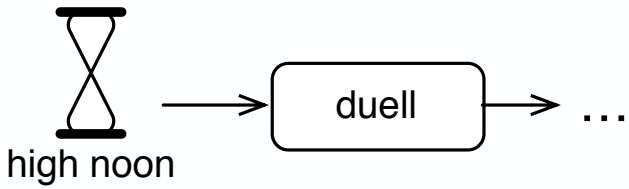
Signale und Ereignisse

- **Signalversand** (SendSignalAction):
  Ein Signal wird verschickt,
  der Fluss geht direkt weiter.

- **Ereignisempfang** (Accept Event Action):
  Ablauf wartet, bis ein spezielles
  Ereignis eintritt. Dann wird an der
  Action ein Token erzeugt ähnlich
  wie beim Startknoten.
  - Kann auch eine Aktivität starten



- **Zeitereignis** (Time Event):
  Token wird zu einem beschriebenen
  Zeitpunkt („jede Stunde“, „alle 5 Minuten“,
  „Mitternacht“) erzeugt.

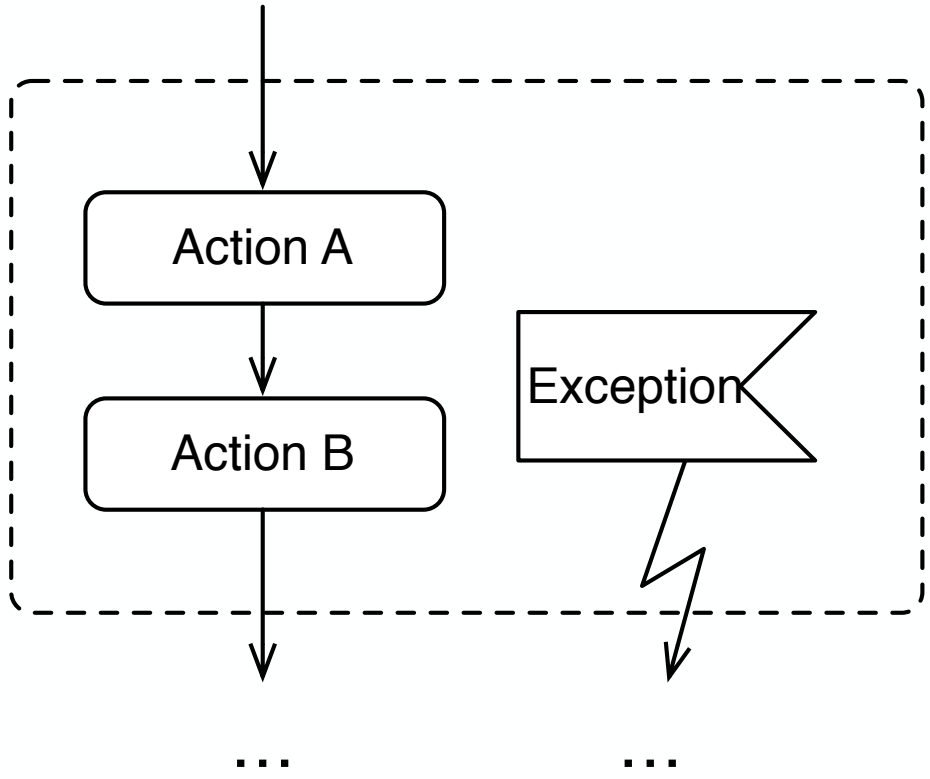


- Ereignisse und Signale korrespondieren häufig miteinander.

Unterbrechungsbereiche

Pattern: Ein Ereignis führt zu einer Unterbrechungskante

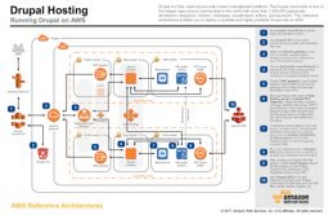
- **Unterbrechungsbereich**:
  Bereich mit Aktivitäten etc., der mit **speziellen Kanten** so verlassen werden kann, dass alle anderen Tokens im Bereich vernichtet werden.
- Notation:
  - **Unterbrechungsbereich**: gestricheltes Rechteck mit abgerundeten Ecken
  - **Unterbrechungskante**: Kante mit Zick-Zack-Muster



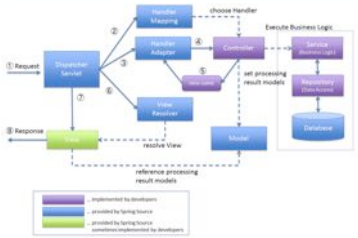
BHT Berliner Hochschule für Technik

Dokumentation von Softwarearchitekturen

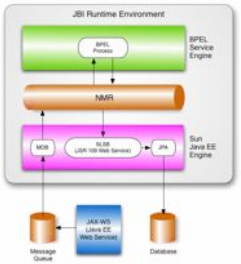
Don’t try this at home



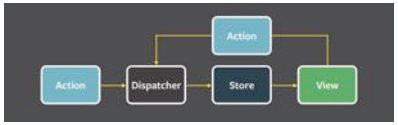
https://github.com/aws-samples/aws-refarch-drupal



https://terasolunaorg.github.io/guideline/1.0.1.RELEASE/en/Overview/SpringMVCOverview.html



https://docs.oracle.com/cd/E19509-01/821-0237/javaeeseug_intro/index.html



https://facebook.github.io/flux/docs/in-depth-overview.html

„We all have seen many books and articles in which a single diagram attempts to capture the gist of a system architecture. But when you look carefully at the diagram’s boxes and arrows, it becomes clear that the **authors are struggling to represent more in one diagram than is practical**. Do the boxes represent running programs? Chunks of source code? Physical computers? Or merely logical groupings of functionality? Do the arrows represent compilation dependencies? Control flows? Dataflows? Usually the answer is that they represent a bit of everything.“
[Kru95]

Architecture View — Architektursichten

„work product expressing the architecture of a system from the perspective of specific system concerns“

„Arbeitsergebnis, das die Architektur eines Systems aus der Perspektive eines spezifischen Anliegens (am System) darstellt.
“

[SEV17, Übers. JvP]

Wessen Anliegen? Ein Anliegen eines Stakeholders!

Exkurs

Viewpoint — Standpunkt

„specification of the conventions for constructing and using a view“

„Spezifikation der Konventionen zur Erstellung einer Sicht.“

[SEV17, Übers. JvP]

Quasi das Metamodell einer View bzw. Sicht.
Wir reden einfach nur von Sichten.

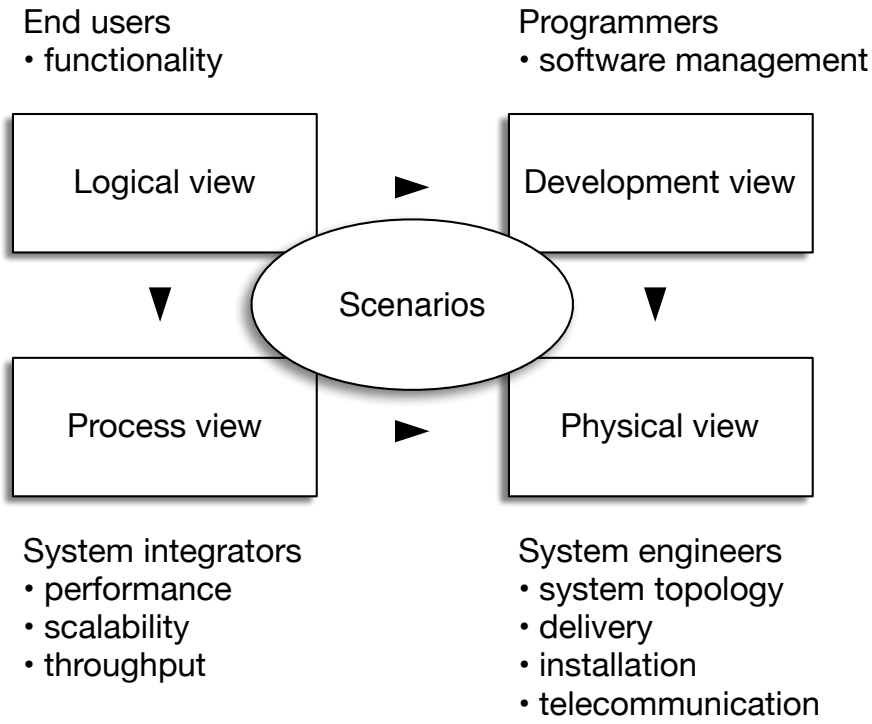
Exkurs

4+1 View Model

Kruchten stellt in [Kru95] folgende Sichten vor:

1. **Logische Sicht** (logical view):
   Klassen- oder ER-Modell
   (UML class diagram)
2. **Prozesssicht** (process view):
   dynamische Aspekte wie
   Nebenläufigkeiten und
   Synchronizsationaspekte
   (UML activity diagram, state machine)
3. **Verteilungssicht** (physical view):
   Verteilung auf Hardware
   (UML deployment diagram)
4. **Entwicklersicht** (development view): Komponenten und Pakete
   (UML component and package diagrams)

+ 1 = **Szenarien:**
   ausgewählte Szenarien von Use Cases



nach [Kru95]

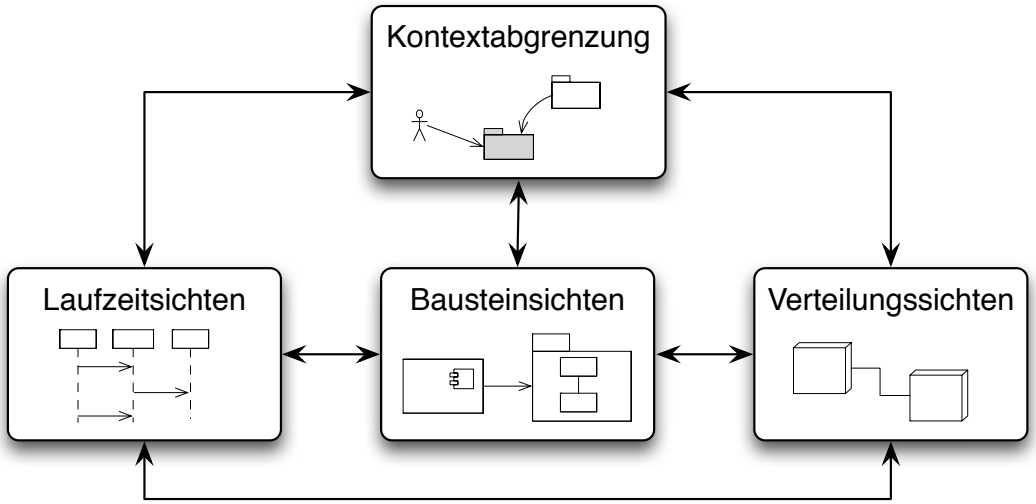
Sichten nach iSAQB

International Software Architecture Qualification Board e.V.
https://www.isaqb.org/

Kontextabgrenzung:
vgl. *Systemkontextdiagramm*

Laufzeitsicht:
Dynamische Sicht, Ablauf von U
(vgl. „process view“
bei [Kru95])

Bausteinsicht:
Komponenten- und Top-Level
Klassendiagramme
(vgl. „development view“ bei [Kru95])



[Sta18, S. 157], ähnlich [GKRS17, S. 102]

Verteilungssicht:
Verteilung auf Hardware, *Deployment-Diagramme* (vgl. „physical view“ bei [Kru95])

BHT Berliner Hochschule für Technik

Bausteinsicht mit Component Diagrams

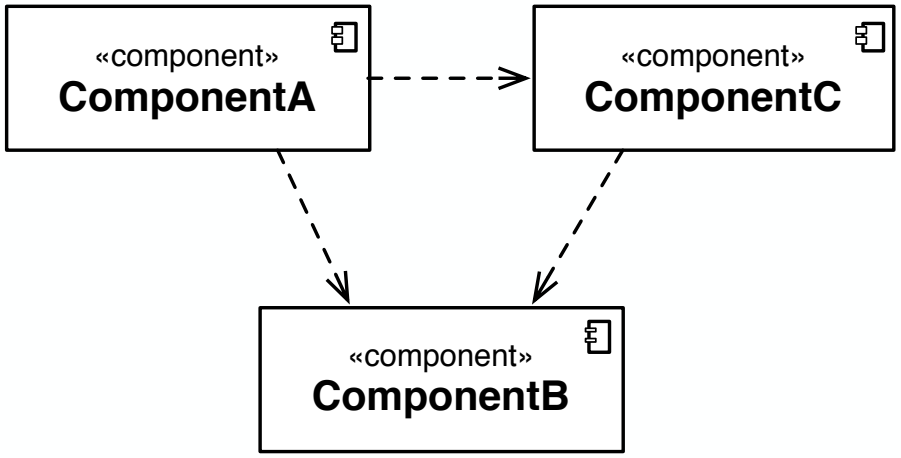
UML Component

- ist eine Spezialisierung von Class
  - Keyword «component» und/oder Icon
- sind **die** Architekturelemente:
  - es gibt nur diese Elemente (quasi die Knoten des Architekturgraphen)
- werden ja nach Architektur unterschiedliche bezeichnet (Schicht, Plugin, Bundle, Baustein etc.)
- Werden für 3 Sichten verwendet:
  - **Abhängigkeiten**
  - **Externe Sicht** mit Fokus auf Kommunikation
  - **Interne Sicht** mit Fokus auf interne Umsetzung

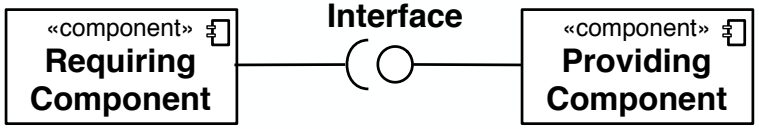
Component Diagram: Abhängigkeitssicht

Component Diagrams werden verwendet,

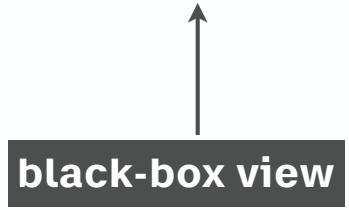
- um die Abhängigkeiten zwischen Components darzustellen
- insbesondere, um die Richtung der Abhängigkeiten zu erkennen
- dies sind (i.d.R.) Compile-Time-Abhängigkeiten



UML Component, externe Sicht

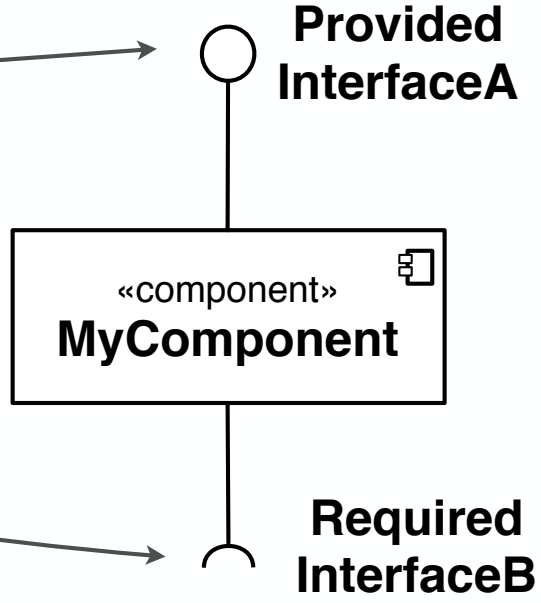


Hier sehen wir nicht mehr die Richtung der Abhängigkeiten, sondern wie die Components miteinander kommunizieren!



Kommunikation nach außen über **Interfaces**:

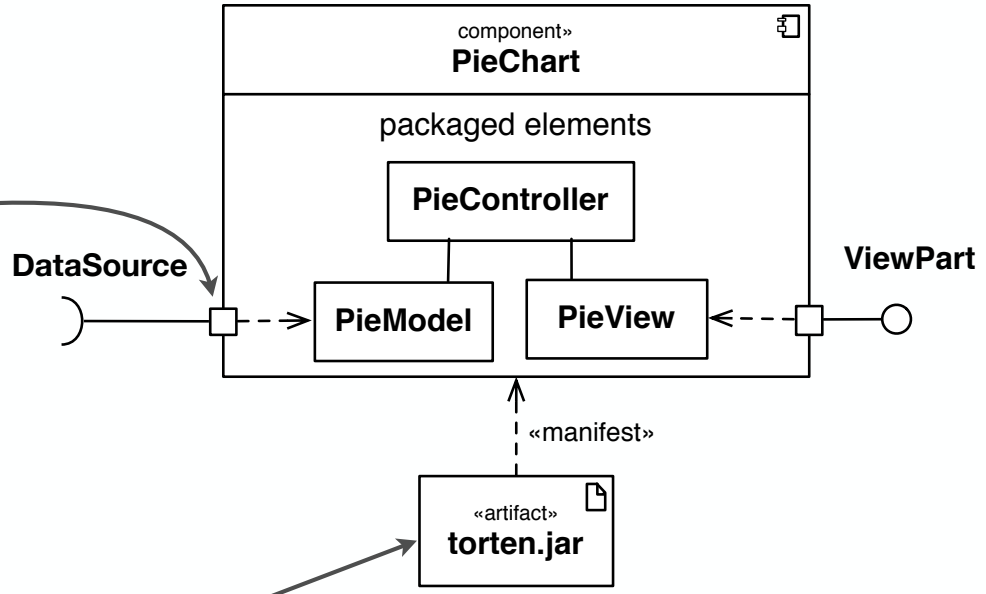
- bietet (**provides**) Interfaces an (**Lollipops**)
- benötigt (**requires**) Interfaces (**Sockets**)



UML Component, interne Sicht (1)

white-box view

- kann wie ein Paket Elemente (v.a. Components oder Classes) enthalten
  - diese können für die externen Interfaces benötigt werden, dies wird über Dependencies an Ports visualisiert
- Ports (Rechtecke) verbinden extern sichtbare Interfaces mit interner Umsetzung
  - ein Port pro Interface oder auch für mehre Interfaces (dann: komplexer Port)
- Artefakte (artifacts) können Komponente „manifestieren“

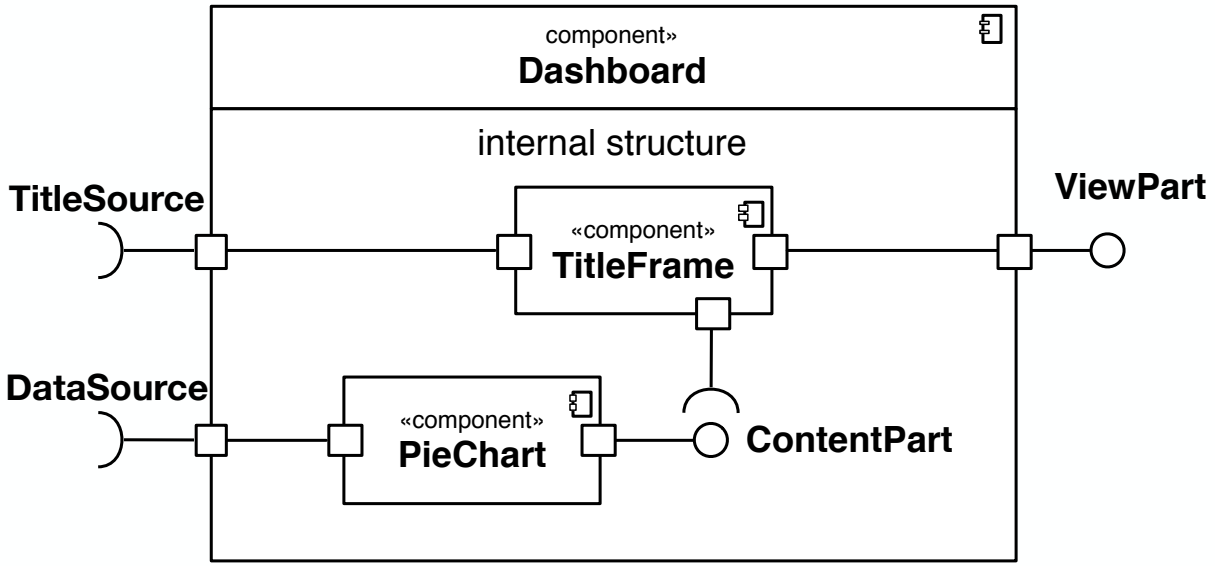


Geist (=Komponente) erscheint (manifestiert sich) aus der Flasche (=Artefakt)

Sub-Component, interne Sicht (2)

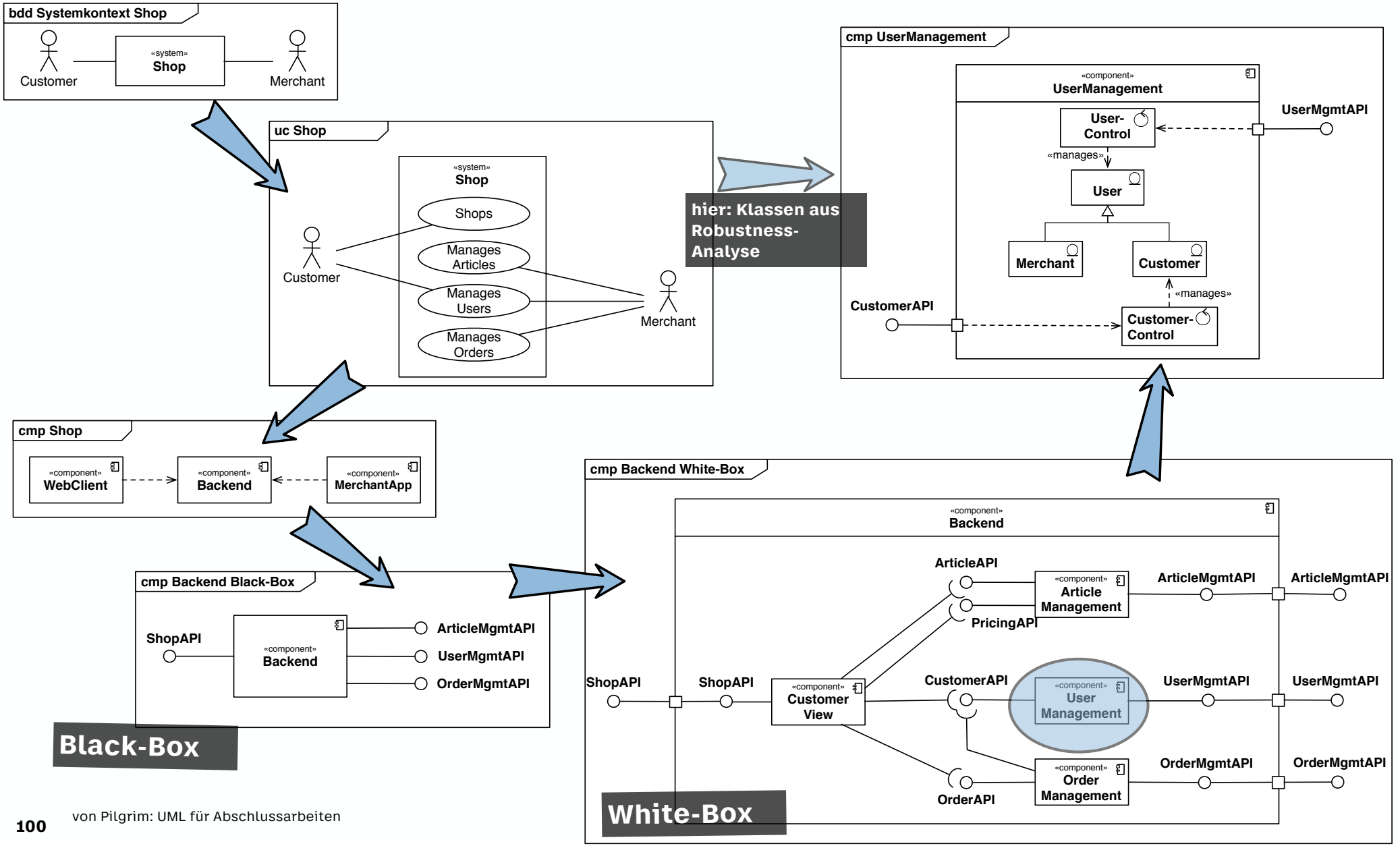
Komponenten können andere Komponenten (Sub-Components) enthalten

- diese können die externen Interfaces bereitstellen, dies wird über Connectors (Linien) an die Ports visualisiert



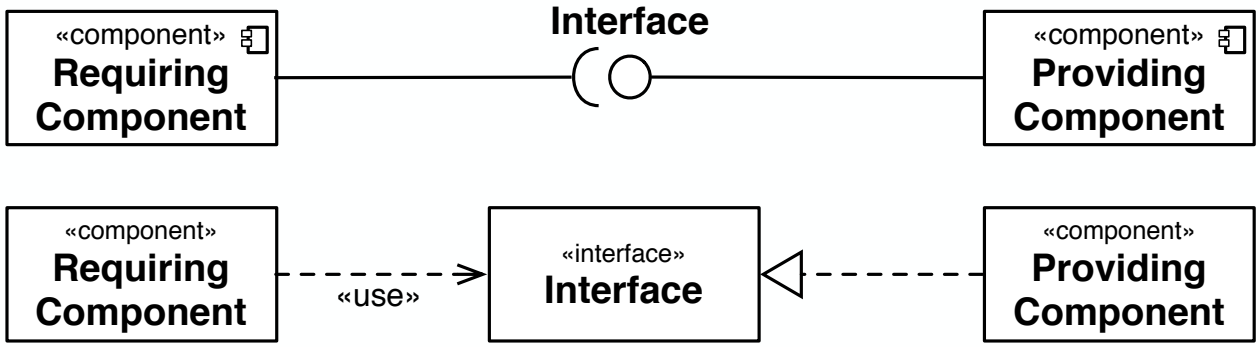
Pragmatisch: Ob „packaged elements“ oder „internal structure“ spielt i.A. keine Rolle, daher lassen wir das einfach weg.

Komponenten-Diagramm im Prozess

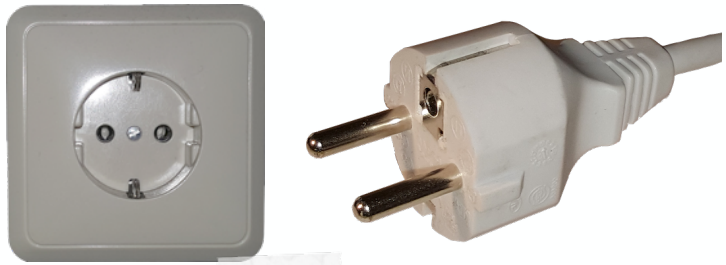


von Pilgrim: UML für Abschlussarbeiten

Interfaces und Komponenten



Socket heißt Steckdose!



Framework

Hollywood Principle:
Don't call us, we call you!

```
connect(Interface: i)
```

```
fw.connect(this)
```

User of Framework

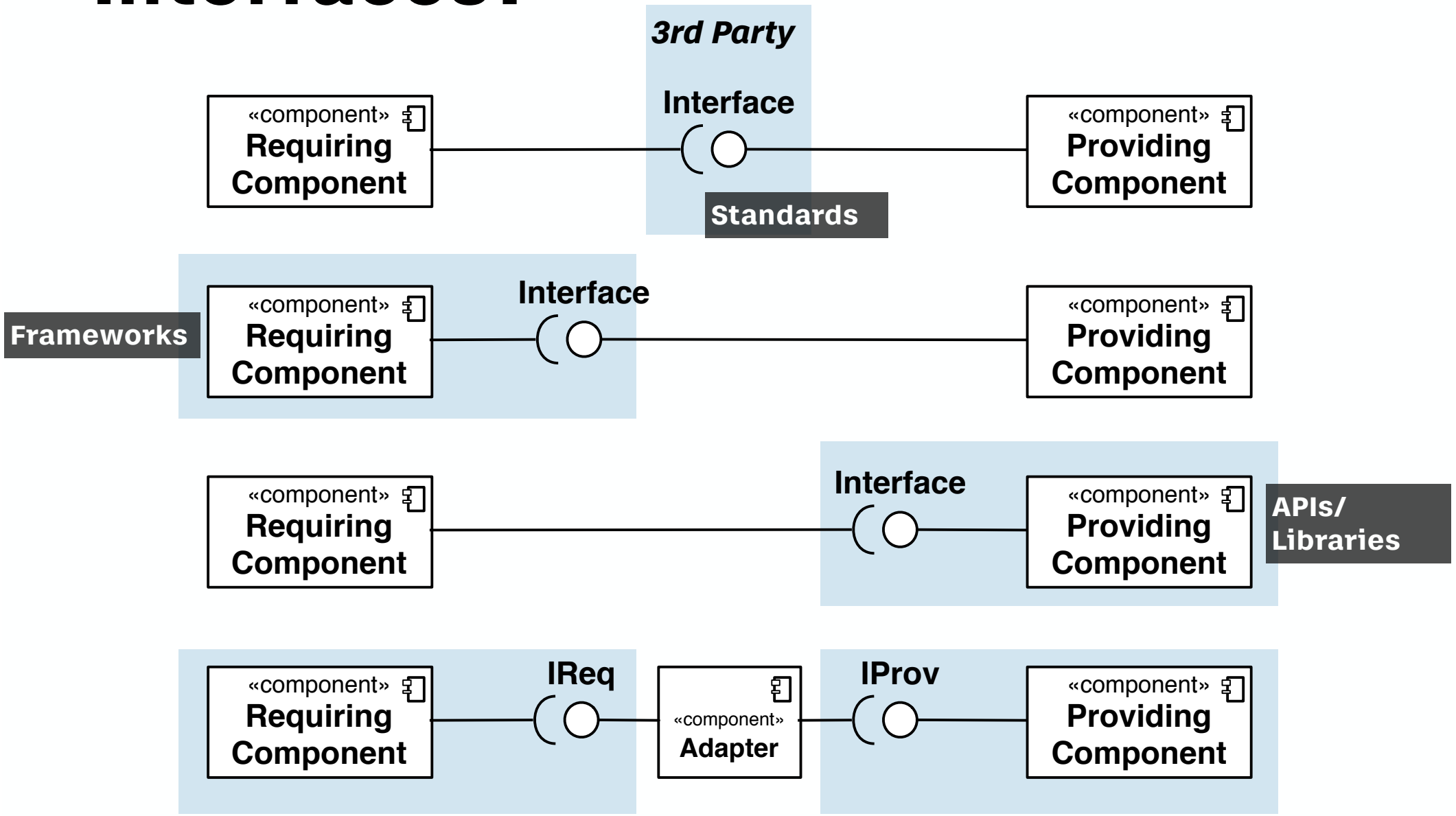
User of API (or Library)

```
api.foo()
api.bar()
```

```
API {
    foo()
    bar()
}
```

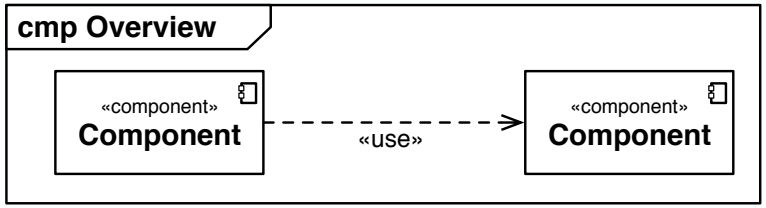
Application Programming Interface

Wem gehören die Interfaces?

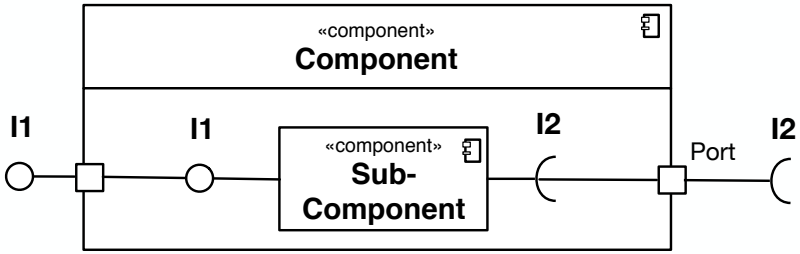


von Pilgrim: UML für Abschlussarbeiten

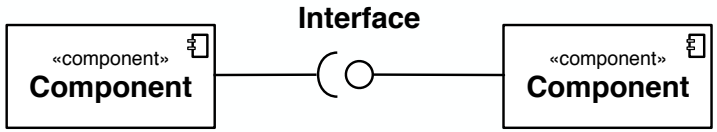
Cheat Sheet UML
Components



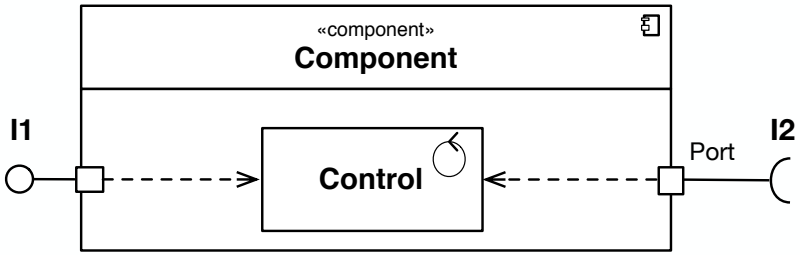
Dependencies instead of
Interfaces in Overview



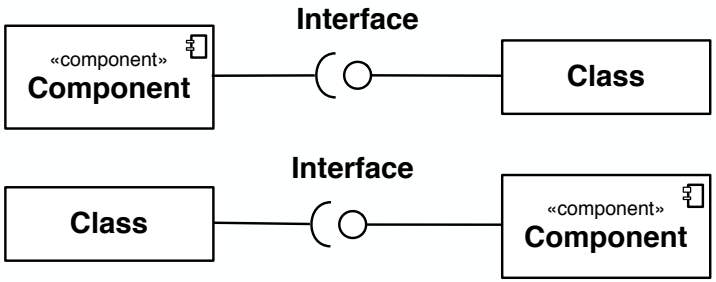
Port is associated to Interfaces provided or
required by Sub-Components



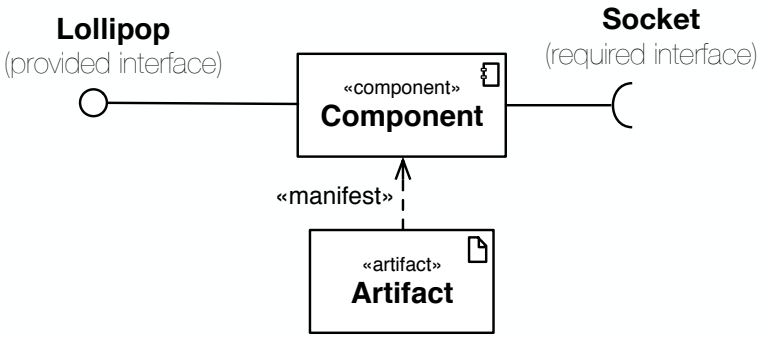
Components are connected via Interfaces



Port depends on Classifiers in
Component



Class can implement or use an interface

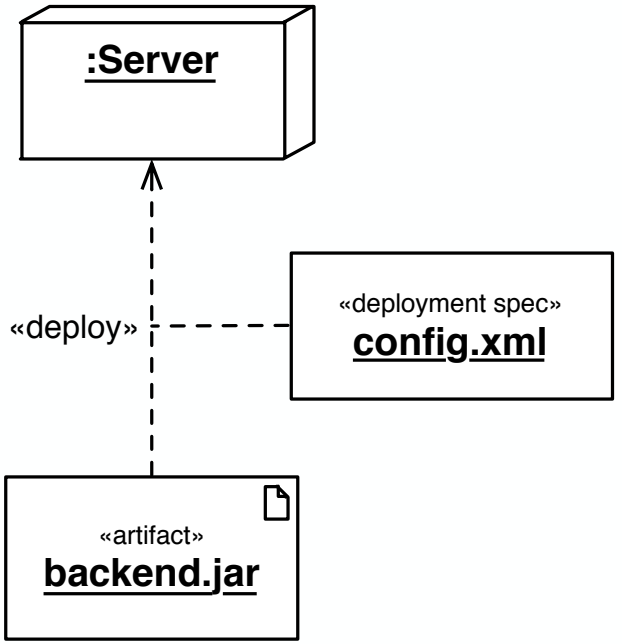


BHT Berliner Hochschule für Technik

Verteilungssicht mit Deployment Diagrams

UML Deployment

- DeploymentTarget, i.d.R. **Node**
  - dargestellt als Quader
- auf dieses können **Artefakte** («artifact») „deployed“ werden
- **Deployment** ist eine spezielle Abhängigkeit (mit Keyword «deploy»)
  - Darstellung entweder durch gestrichelte Pfeile oder der Artefakte innerhalb des Nodes
  - kann über DeploymentSpecification, Keyword «deployment spec», konfiguriert werden; Darstellung durch gestrichelte Linie zwischen Spec und Deployment-Pfeil

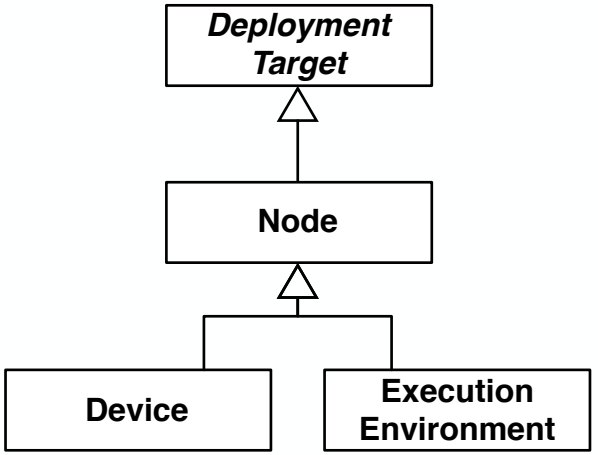


Beziehung zwischen Komponenten und Deployments über „manifestierte“ Artefakte

- Namen von konkreten Node-Instanzen, Artefakten und Deployment Specs sind unterstrichen, wenn konkrete Exemplare bezeichnet werden (wie bei Objekten)

Arten von Nodes

- **Node**: allgemeine Resource auf der Artefakte zur Ausführung installiert (deployed) werden können, bspw. „Server“ (kann virtuell sein oder auch nicht)
- **Device** «device»: Physischer Rechner, bspw. „iPhone“
- **Execution Environment** «executionEnvironment»: Ausführungsumgebung für spezifische Arten von Komponenten, bspw. „Webcontainer“ (Tomcat) oder Docker



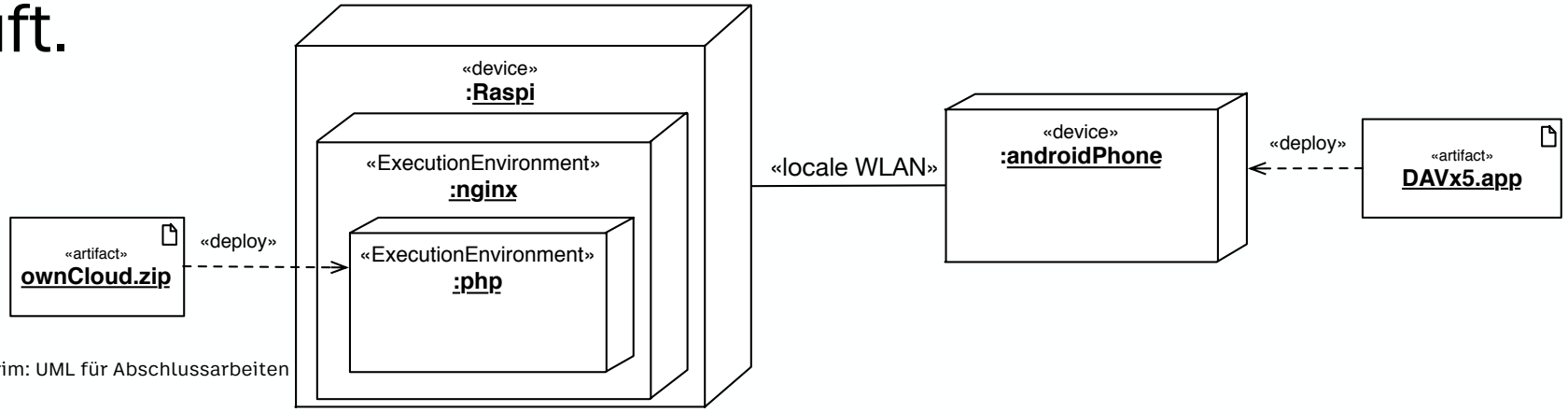
Stereotypisierung möglich, i.d.R. aber aus dem Namen ersichtlich

Beziehungen zwischen Nodes

Kommunikation mittels **Communication Path**

- eine spezielle Assoziation, entsprechend visualisiert (i.d.R. ohne Rollen)
- kann mit Stereotypen zur Art der Kommunikation versehen werden, bspw. «WLAN» oder «Funk»

Nodes können andere Nodes enthalten (nesting), wenn bspw. ein **Execution Environment** auf einem **Device** läuft.



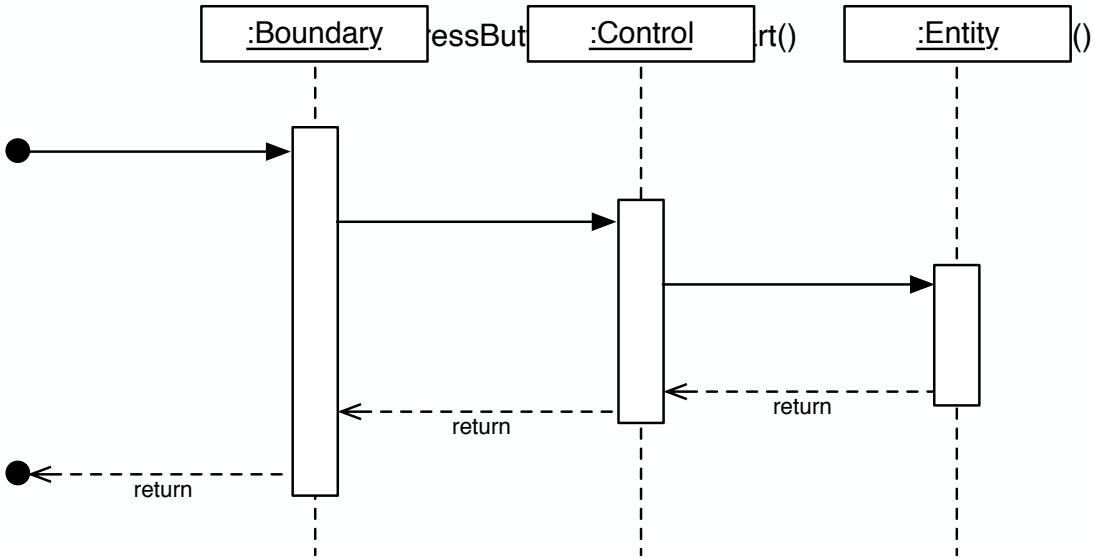
BHT Berliner Hochschule für Technik

Laufzeitsicht mit Sequence Diagrams

Laufzeitsicht

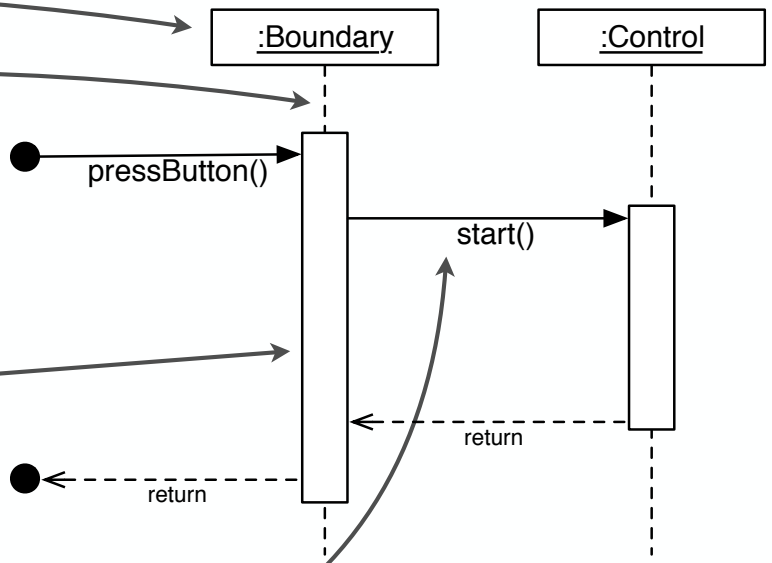
- kann u.a. dargestellt werden mit
  - UML Activity Diagram
  - UML **Sequence Diagram**
  - UML Communication Diagram
  - evtl. auch Pseudo-Code
- zeigt **Kommunikation** bzw. „**dynamisches Zusammenwirken von Bausteinen**“ [GKRS17]
  - v.a. **Life Cycle** mit Systemstart (Initialisierung)
  - Ablauf von Szenarien im System

Sequenzdiagram



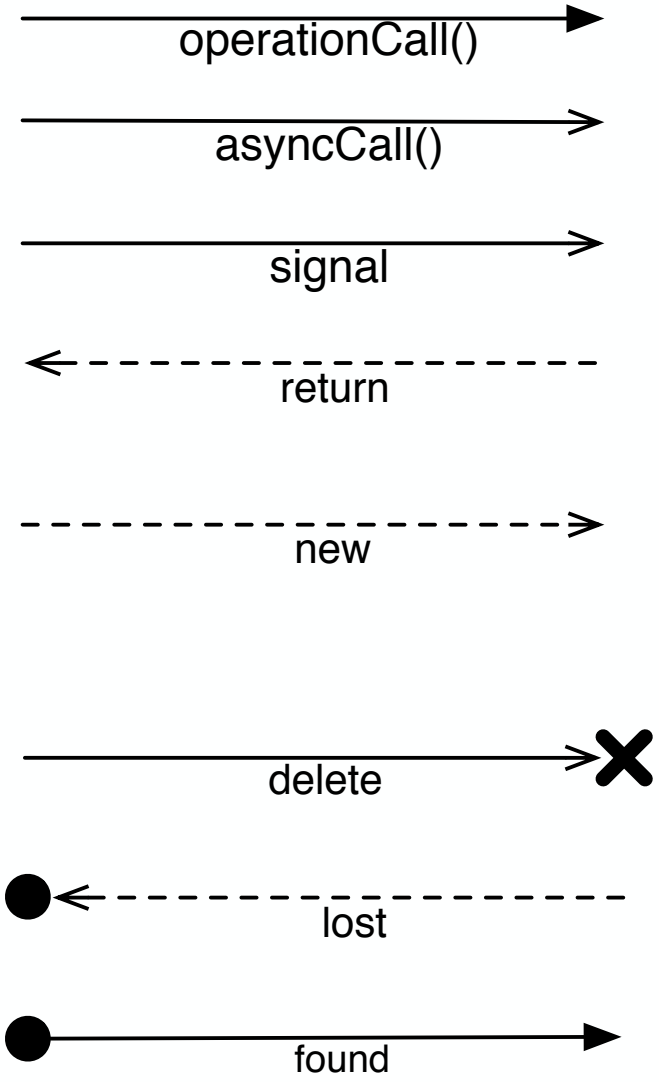
UML Sequence Diagram

- **vertikal**: zeitlicher Verlauf von oben nach unten
- **horizontal**: Objekte
  - Lebenslinien (lifelines)
- Execution Specification (Aktivitätsbereich):
  - Zeitdauer der Ausführung (einer Aktion oder eines Verhaltens)
  - Balken auf Lebenslinie
  - kann weggelassen werden (wenn Dauer nicht von Interesse), macht Diagramm aber i.d.R. besser lesbar
- **Nachrichten** (messages)
  - oft einfach Methodenaufrufe
  - Pfeile zwischen Lebenlinien bzw. Execution Specifications



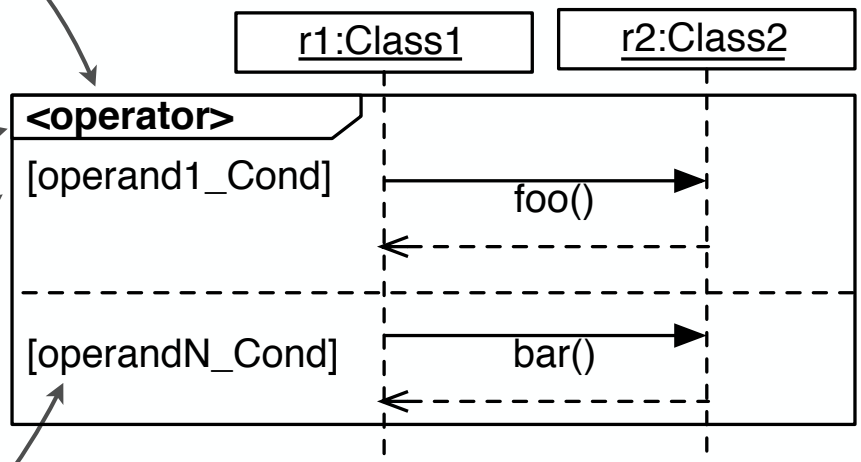
Sorten von Messages (messageSort)

- **synchCall**: Pfeil mit ausgefüllter Pfeilspitze
  - typischer synchroner Methodenaufruf (etwa in Java)
- **asynchCall**: Pfeil mit offener Pfeilspitze
  - asynchroner Methodenaufruf, kehrt sofort zurück!
  - immer, wenn die Nachricht ein Signal (und kein Operationsaufruf) ist
  - in Java über Runnables
- **reply**: gestrichelter Pfeil mit offener Spitze
  - Rückgabe einer synchronen Methode
  - „return“ in Java
  - beendet i.A. den Execution-Specification-Balken
- **create**: Objekterzeugung: gestrichelter Pfeil mit offener Spitze
  - Erzeugung eines Objekts
  - „new“ in Java
  - „Erzeugt“ neues Objekt mit eigener Lebenslinie
- **delete**: Objektzerstörung: endet in DestructionOccurrenceSpecification
  - Zerstörung eines Objekts
  - Aufruf des Destruktors in C++
- **lost**: endet in schwarzem Kreis (anstelle von Lebenslinie)
  - Sender bekannt, Empfänger außerhalb der Spezifikation
  - endet in DestructionOccurrenceSpecification
- **found**: startet in schwarzem Kreis (anstelle von Lebenslinie)
  - Empfänger bekannt, Sender außerhalb der Spezifikation



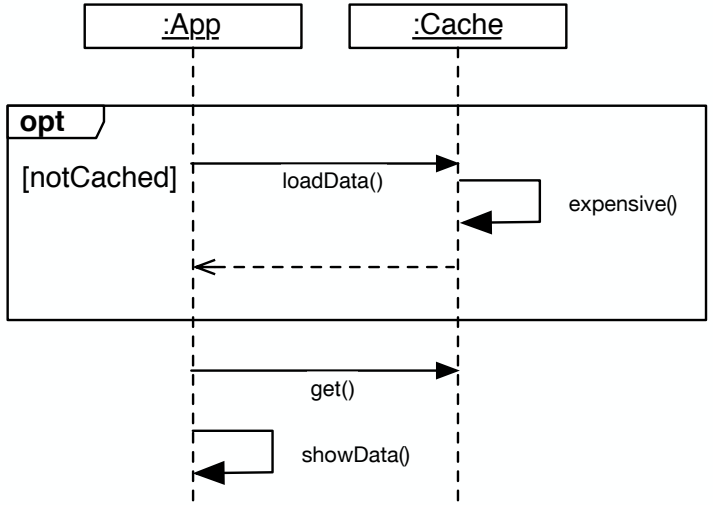
Combined Fragments

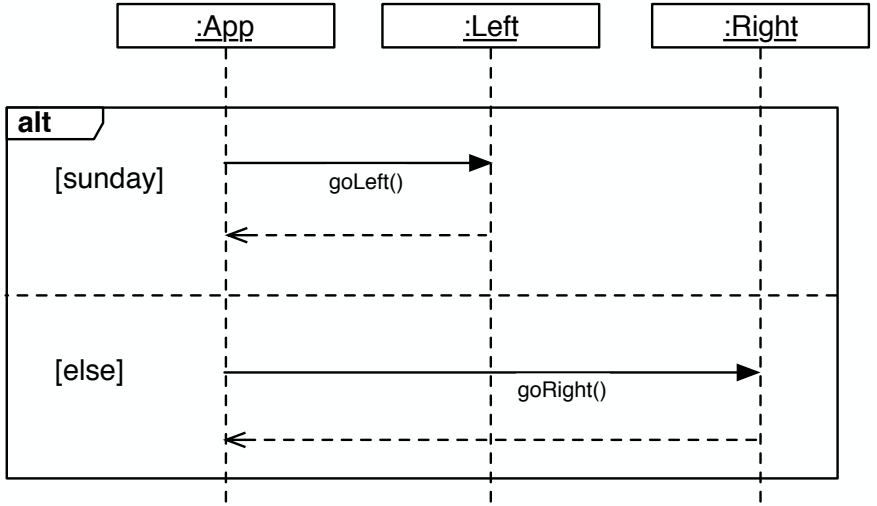
- Rahmen ähnlich Diagrammrahmen innerhalb dessen speziellen Regeln im Diagramm gelten
- können geschachtelt werden
- **Interaktionsoperator**: im Fünfeck rechts-oben
  - legt genaue Semantik des Fragments fest
- **Interaktionsoperanden**: vertikale Bereiche im Rahmen, mit gestrichelter Linie getrennt
  - **Interaktionsbedingung** rechts-oben im Operanden (bswp. [x=1] oder [else]



Combined Fragments: Option und Alternative

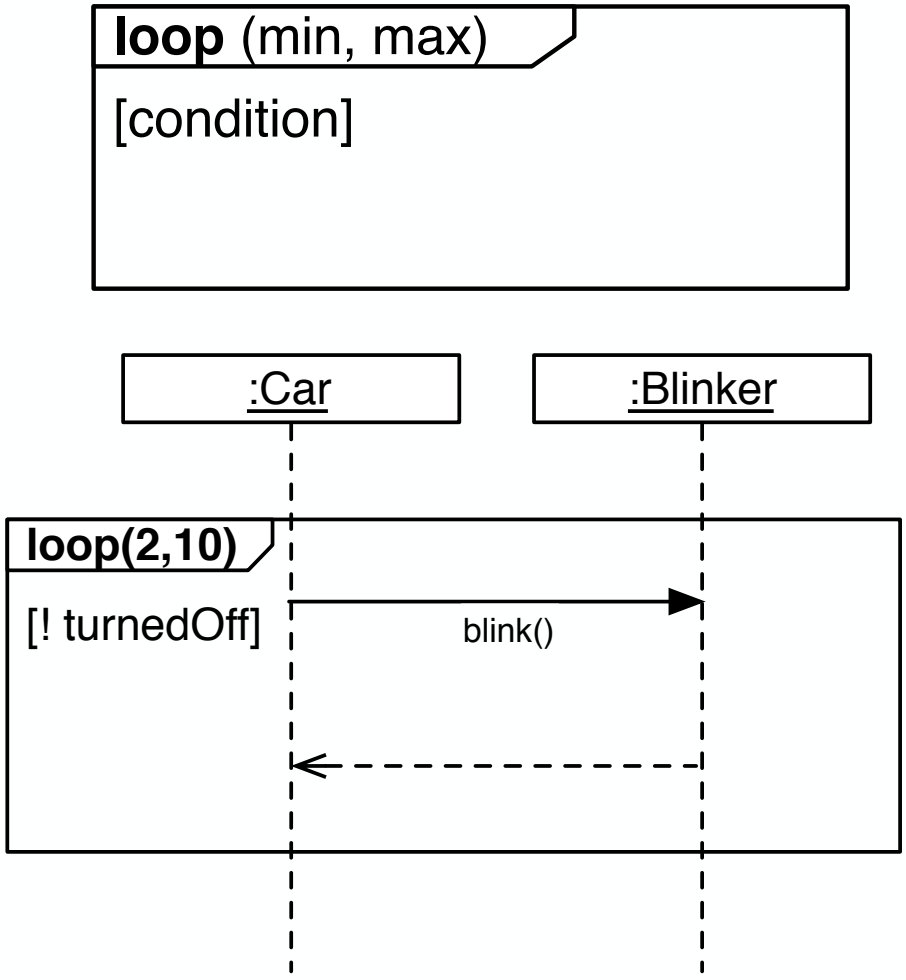
- **optional**, Operator: **opt**
  - wird ausgeführt, wenn Bedingung wahr
  - nur 1 Operand
  - wie „if...“ in Java
- **alternativ**, Operator: **alt**
  - jeweiliger Operand wird ausgeführt, wenn Bedingung wahr
  - mind. 2 Operanden
  - wie „if.. [else if]* ... else“ in Java





Combined Fragments: Schleife (Loop)

- **Schleife**, Operator: **loop**
  - wird mindestens **min**-mal ausgeführt, falls min angegeben
  - wird maximal **max**-mal ausgeführt, falls max angegeben
  - **zusätzlich wird Bedingung** (condition) geprüft




```
for (let loop=0;
      (loop<=2 || !turnedOff) && loop<10,
      loop++) {
       blink()
}
```

Referenz (Interaction Use)

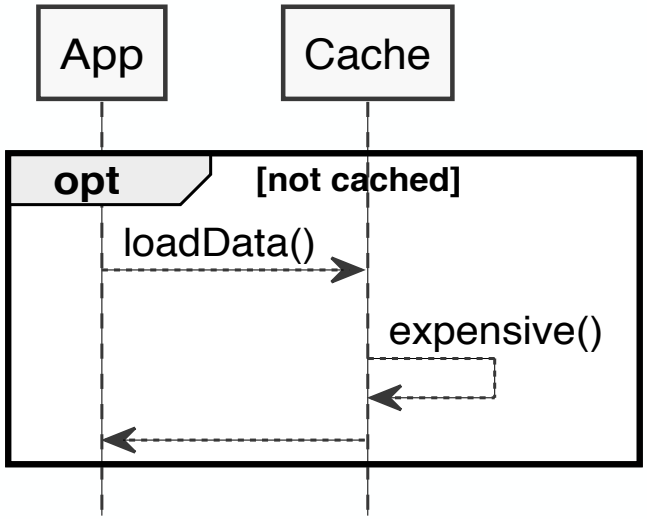
- Bereich, der ein anderes Sequenzdiagramm referenziert
  - Referenz erfolgt über den Namen
- das referenzierte Diagramm muss theoretisch nahtlos eingefügt werden können
- Anwendung:
  - Vereinfachung von großen Diagrammen
  - Wiederverwendung

PlantUML für Sequence-Diagramme

PlantUML Website: http://plantuml.com/

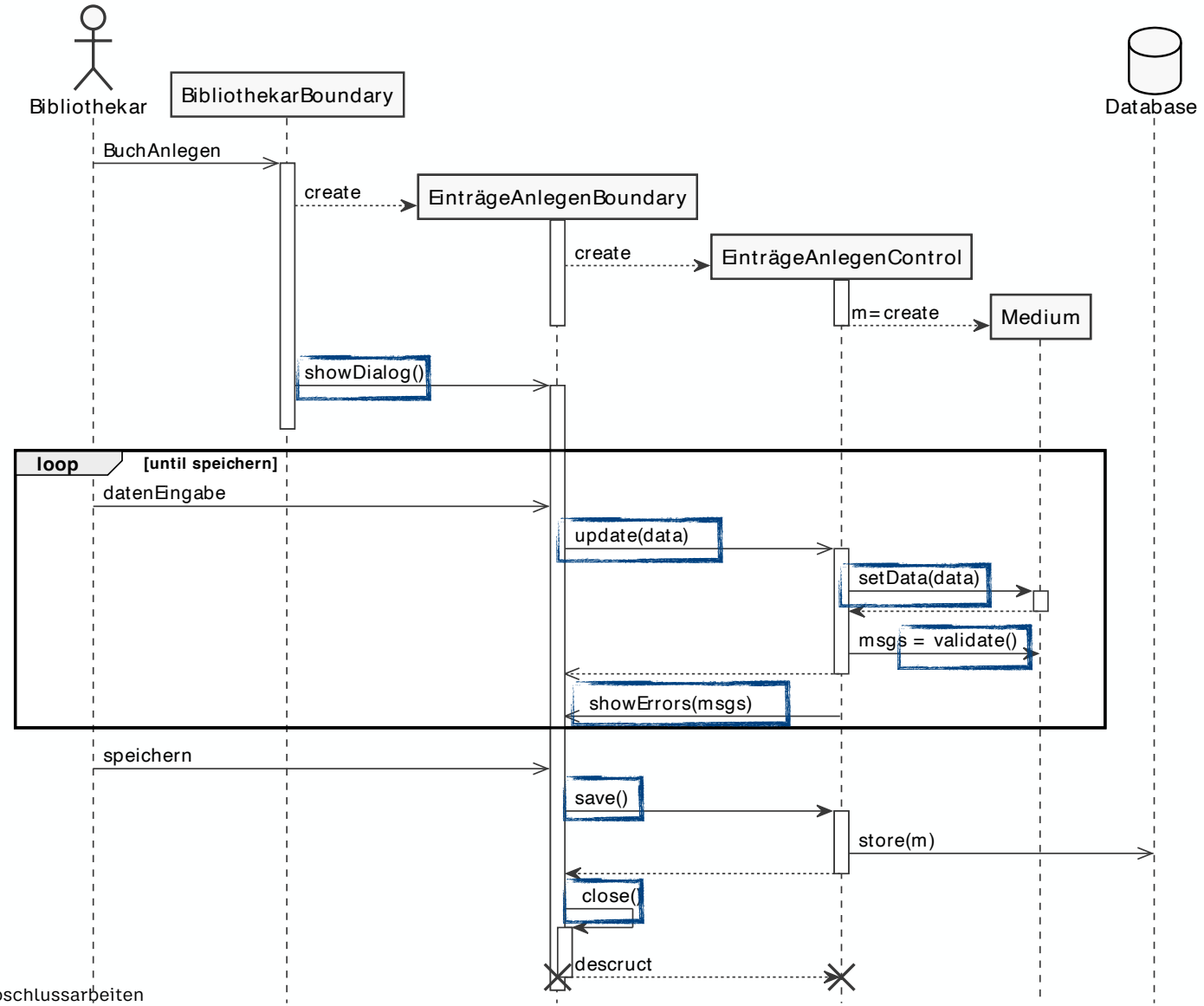
für Sequence-Diagramme empfehlenswert

```
@startuml
skinparam monochrome true
skinparam shadowing false
hide footbox

opt not cached
App --> Cache: loadData()
Cache --> Cache: expensive()
App <-- Cache:
end

@enduml
```



java -jar plantuml.jar -tsvg sample.txt

Auskleidung der Robustness-Analyseklassen



von Pilgrim: UML für Abschlussarbeiten

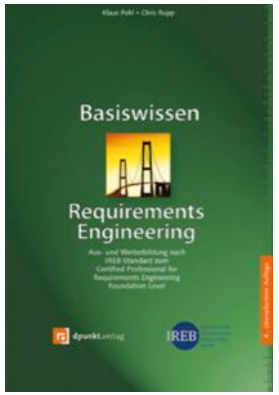
PlantUML Source Code

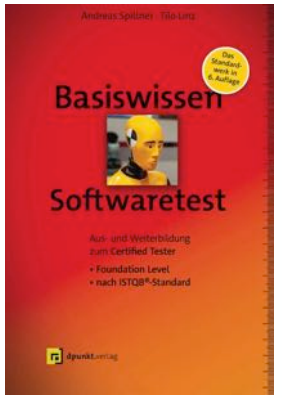
```
@startuml
skinparam monochrome true
skinparam shadowing false
hide footbox

actor Bibliothekar
Bibliothekar ->> BibliothekarBoundary : BuchAnlegen
activate BibliothekarBoundary
BibliothekarBoundary --> EinträgeAnlegenBoundary ** : create
activate EinträgeAnlegenBoundary
EinträgeAnlegenBoundary --> EinträgeAnlegenControl **: create
activate EinträgeAnlegenControl
EinträgeAnlegenControl --> Medium ** : m=create
deactivate EinträgeAnlegenControl
deactivate EinträgeAnlegenBoundary

BibliothekarBoundary ->> EinträgeAnlegenBoundary ++ : showDialog()
deactivate BibliothekarBoundary

loop until speichern
Bibliothekar ->> EinträgeAnlegenBoundary: datenEingabe
EinträgeAnlegenBoundary ->> EinträgeAnlegenControl ++ : update(data)
EinträgeAnlegenControl -> Medium ++: setData(data)
return
EinträgeAnlegenControl -> Medium: msgs = validate()
return
EinträgeAnlegenBoundary <<- EinträgeAnlegenControl: showErrors(msgs)
end

Bibliothekar ->> EinträgeAnlegenBoundary: speichern
EinträgeAnlegenBoundary -> EinträgeAnlegenControl ++ : save()
database Database
EinträgeAnlegenControl ->> Database: store(m)
EinträgeAnlegenBoundary <-- EinträgeAnlegenControl

deactivate EinträgeAnlegenControl
EinträgeAnlegenBoundary -> EinträgeAnlegenBoundary ++: close()
EinträgeAnlegenBoundary --> EinträgeAnlegenControl: descruct
destroy EinträgeAnlegenControl
destroy EinträgeAnlegenBoundary
@enduml
```

Weitere Hinweise

Literaturhinweise

- Pohl, Klaus; Rupp, Chris:
  Basiswissen Requirements Engineering.
  dpunkt, 2015
- Gharbi, Mahbouba; Koschel, Arne; Rausch, Andreas; Starke, Gernot:
  Basiswissen für Softwarearchitekten.
  dpunkt, 2017
- Spillner, Andreas; Linz, Tilo:
  Basiswissen Softwaretest.
  dpunkt, 2019





Literatur

[GKRS17] Gharbi, Mahbouba ; Koschel, Arne ; Rausch, Andreas ; Starke, Gernot: Basiswissen für Softwarearchitekten. 3. Auflage, dpunkt, 2017. – ISBN 978–3–86490–499–8

[ISO11] ISO/IEC/IEEE: ISO/IEC/IEEE Systems and software engineering – Architecture description / IEEE. Version: Dec 2011. http://dx.doi.org/10.1109/IEEESTD.2011.6129467. 2011 ((Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)). – International Standard. – 1–46 S.

[iSA16] iSAQB: iSAQB Glossary of Software Architecture Terminology. Version:2016. https: //www.isaqb.org/wp-content/uploads/2016/12/Glossary_of_Software_Architecture_ Terminology_2016_12_22.pdf

[Kru95] Kruchten, Philippe: The 4+1 View Model of architecture. In: IEEE Software 12 (1995), Nov, Nr. 6, S. 42–50. http://dx.doi.org/10.1109/52.469759. – DOI 10.1109/52.469759. – ISSN 0740–7459

[WHM+18] Weilkiens, Tim ; Huwaldt, Alexander ; Mottok, Jürgen ; Roth, Stephan ; Willert, Andreas: Modellbasierte Softwareentwicklung für eingebettete Systeme verstehen und anwenden. 1. dpunkt.verlag, 2018. – ISBN 978–3–96088–593–1

[OMG17a] OMG: OMG Unified Modeling Language (OMG UML), Version 2.5.1. formal/17-12-05. Needham, MA: Object Management Group, Dezember 2017. https://www.omg.org/spec/UML/2.5.1/

[OMG17b] OMG: OMG Systems Modeling Language (OMG SysML), Version 1.5. formal/2017-05-01. Need- ham, MA: Object Management Group, Mai 2017. https://www.omg.org/cgi-bin/doc?formal/2017-05-01

[OMG18] OMG ; Object Management Group (Hrsg.): Interface Definition Language (IDL), Version 4.2. formal/18-01-05. Needham, MA: Object Management Group, März 2018. https://www.omg. org/spec/IDL/4.2/

[PR15] Pohl, Klaus ; Rupp, Chris: Basiswissen Requirements Engineering. 4. Auflage. dpunkt.verlag, 2015. ISBN 978–3–86491–673–1

[RQ12] Rupp, Christine; Queins, Stefan: UML 2 glasklar. Praxiswissen für die UML-Modellierung und Zertifizierung. 4., überarb. u. erw. Aufl. Hanser Fachbuchverlag, 2012. – ISBN 978–3–446–43057–0

[SEV17] Systems and software engineering — Vocabulary / ISO/IEC/IEEE. Version:September 2017. https://www.iso.org/standard/71952.html (ISO/IEC/IEEE 24765:2017)

[Som16] Ian Sommerville. Software Engineering. Pearson, 10 edition, 2016

[Sta18] Starke, Gernot: Effektive Softwarearchitekturen. Ein praktischer Leitfaden. 8. Auflage, Hanser, 2018. – ISBN 978–3–446–45420–0

[Szy02] Szyperski, Clemens: Component Software: Beyond Object-Oriented Programming. 2nd. Addison- Wesley, 2002. – ISBN 0–201–74572–0

[WHM+18] Weilkiens, Tim ; Huwaldt, Alexander ; Mottok, Jürgen ; Roth, Stephan ; Willert, Andreas: Modellbasierte Softwareentwicklung für eingebettete Systeme verstehen und anwenden. 1. dpunkt.verlag, 2018. – ISBN 978–3–96088–593–1

[Win05] Winter, Mario: Methodische objektorientierte Softwareentwicklung. Eine Integration klassischer und moderner Entwicklungskonzepte. dpunkt.verlag https://www.dpunkt.de/buecher/ 10656/9783898642736-methodische-objektorientierte-softwareentwicklung.html. – ISBN 3–89864–273–9 (Kapitel Anwendungsfalldiagramm kostenlos verfügbar!)