

# Folien zur Vorlesung **Softwaretechnik**

## **Teil 2: UML – Unified Modeling Language** **Abschnitt 2.1: Überblick**

## Unified Modeling Language (UML)

ist eine Sammlung von mehreren grafischen Notationen.

Mit jeder dieser Notationen lassen sich bestimmte Aspekte beschreiben, die bei der Entwicklung von Softwaresystemen relevant sind.

### Anmerkungen:

- ❑ Die durch in den Notationen entstehenden Beschreibungen werden als *Diagramme* oder *Modelle* bezeichnet.
- ❑ Der Vorgang des Beschreibens wird als *Modellierung* bezeichnet.
- ❑ UML ist ab 1995 aus verschiedenen älteren grafischen Notationen entstanden und seit 1997 (Version 1.1) standardisiert
- ❑ Der aktuelle Standard ist seit 2015: Version 2.5
- ❑ Die standardisierte Notation erleichtert die Kommunikation zwischen den Entwicklern, da die zu verwendenden Elemente und deren Bedeutung festgelegt sind.
- ❑ Die Spezifikation der UML besitzt fast 800 Seiten.
  
- ❑ Die Spezifikationen der UML kann heruntergeladen werden:  
<http://www.omg.org/spec/UML/2.5/>

## Unified Modeling Language (UML)

(Fortsetzung)

UML umfasst 14 grafische Notationen, die führen zu

- ❑ 7 Arten von strukturbeschreibenden Diagrammen (Strukturdiagramme):  
**Klassendiagramm, Objektdiagramm, Paketdiagramm, Komponentendiagramm, Profildiagramm, Kompositionsstrukturdiagramm, Verteilungsdiagramm**
- ❑ – 7 Arten von verhaltensbeschreibenden Diagrammen (Verhaltensdiagrammarten):  
**Sequenzdiagramm, Aktivitätsdiagramm, Anwendungsfalldiagramm, Zustandsdiagramm, Kommunikationsdiagramm, Interaktionsübersichtsdiagramm, Zeitverlaufsdiagramm**
- ❑ Anmerkungen:
  - Für die meisten Entwicklungen wird nur ein Teil der Diagramme benötigt.
  - Für die meisten Entwicklungen wird nur ein Teil des Sprachumfangs benötigt, den eine der Notationen bereitstellt.
  - Daher wird in SWT nur ein recht überschaubarer Anteil von UML präsentiert.

## Unified Modeling Language (UML)

(Fortsetzung)

Zunächst erfolgt eine Einführung in

- ❑ Paketdiagramme
- ❑ Klassendiagramme
- ❑ Objektdiagramme
- ❑ Sequenzdiagramme

zur Vorbereitung der nachfolgenden Präsentation von Entwurfsmustern.

Später folgt eine Einführung in

- ❑ Aktivitätsdiagramme (im Rahmen der Präsentation von Testverfahren)
- ❑ Anwendungsfalldiagramme (im Rahmen der Präsentation von Ztechniken zur Anforderungsanalyse)

# Folien zur Vorlesung **Softwaretechnik**

## **Teil 2: UML – Unified Modeling Language** **Abschnitt 2.2: Klassendiagramme/Paketdiagramme**

## Paketdiagramm

Ein Paketdiagramm zeigt

- ❑ die Pakete eines Projekts
- ❑ die Abhängigkeiten zwischen diesen Paketen
- ❑ eventuell auch die Zuordnung von Klassen zu Paketen und
- ❑ eventuell die Abhängigkeiten zwischen Klassen (durch ein eingebettetes Klassendiagramm).

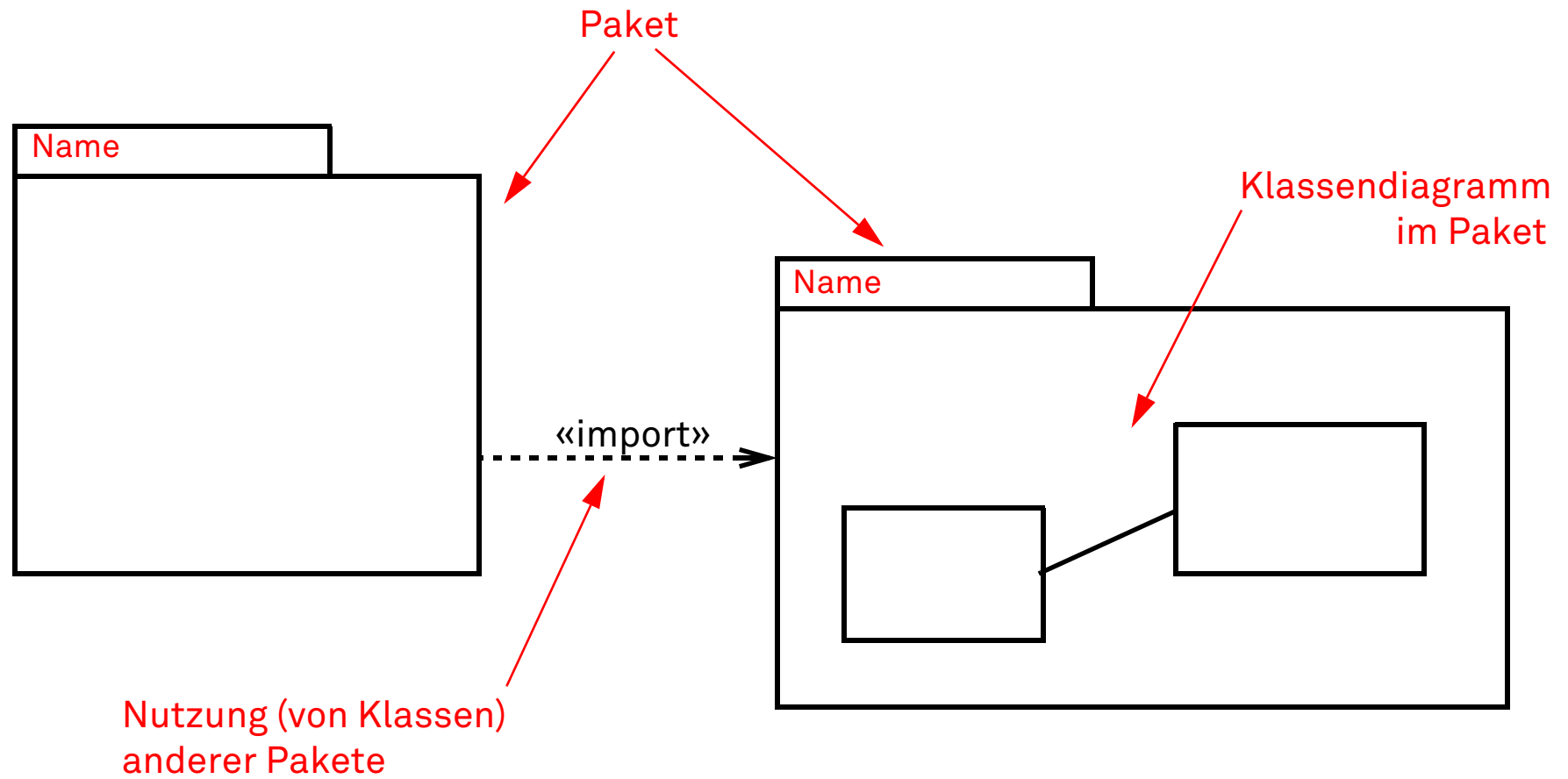
Ein Paketdiagramm schafft eine Darstellung auf einer Abstraktionsebene, die es in Java nicht gibt:

Pakete und ihre Beziehungen werden in Java nur innerhalb von Klassen durch **package**- und **import**-Anweisungen geschaffen.

## Paketdiagramm

(Fortsetzung)

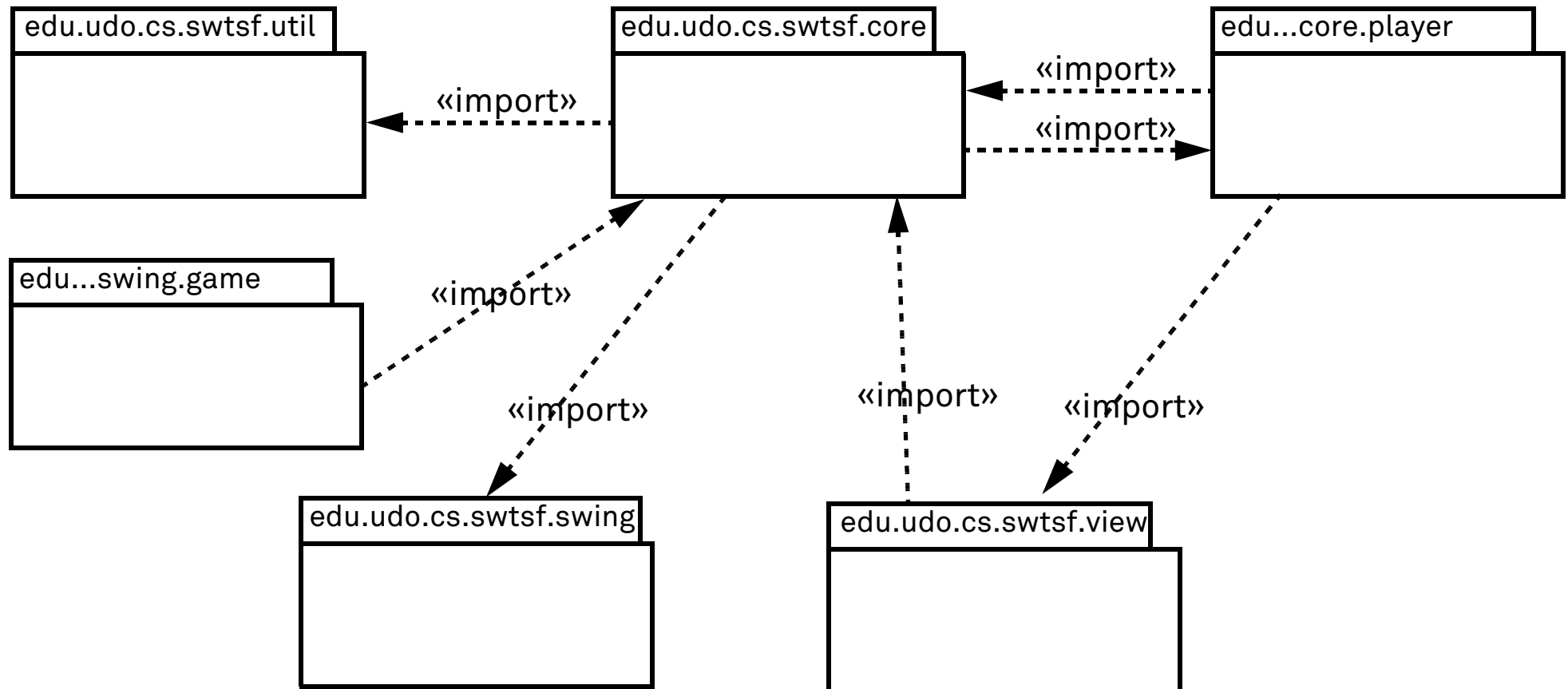
Syntax:



## Paketdiagramm

(Fortsetzung)

Beispiel *SWT-Starfighter – Framework*:

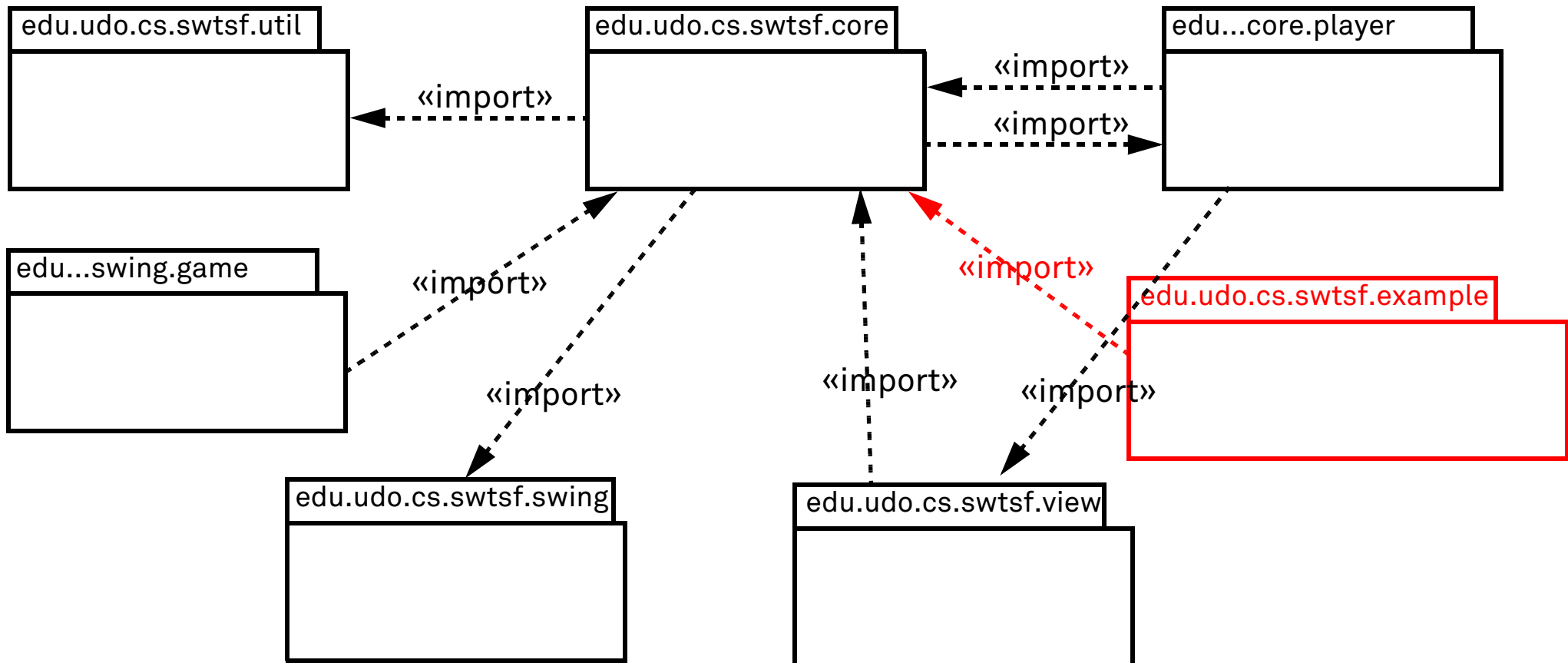




## Paketdiagramm

(Fortsetzung)

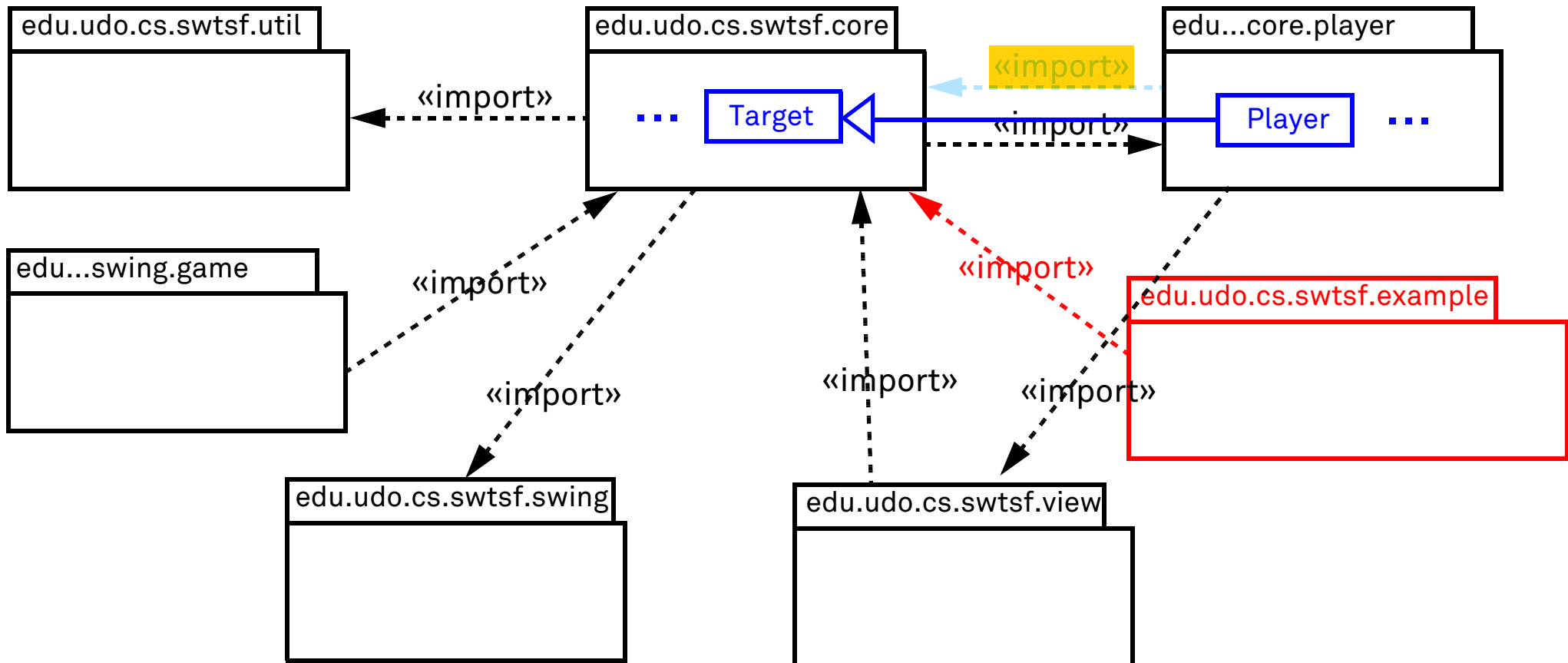
Beispiel *SWT-Starfighter* – Spiel:



## Paketdiagramm

(Fortsetzung)

Beispiel *SWT-Starfighter* – Spiel:



## Klassendiagramm

Ein Klassendiagramm zeigt

- ❑ die Klassen (eines Projekts/eines Pakets),
- ❑ die Beziehungen zwischen diesen Klassen,
- ❑ eventuell die Bestandteile einzelner Klassen.

Klassendiagramme können genutzt werden, um

- ❑ Implementierungen zu visualisieren,
- ❑ Vorgaben für die (direkte) Umsetzung in eine Implementierung zu liefern,
- ❑ Vorgaben für die Planung einer Implementierung zu liefern:  
Das Klassendiagramm veranschaulicht dann nur eine idealisierte Struktur.
- ❑ Daten, Operationen und Abhängigkeiten in objektorientierter Form zu visualisieren  
– unabhängig davon, ob überhaupt eine objektorientierte Implementierung beabsichtigt ist.

## Klasse

Beispiel einer Java-Implementierung einer Klasse:

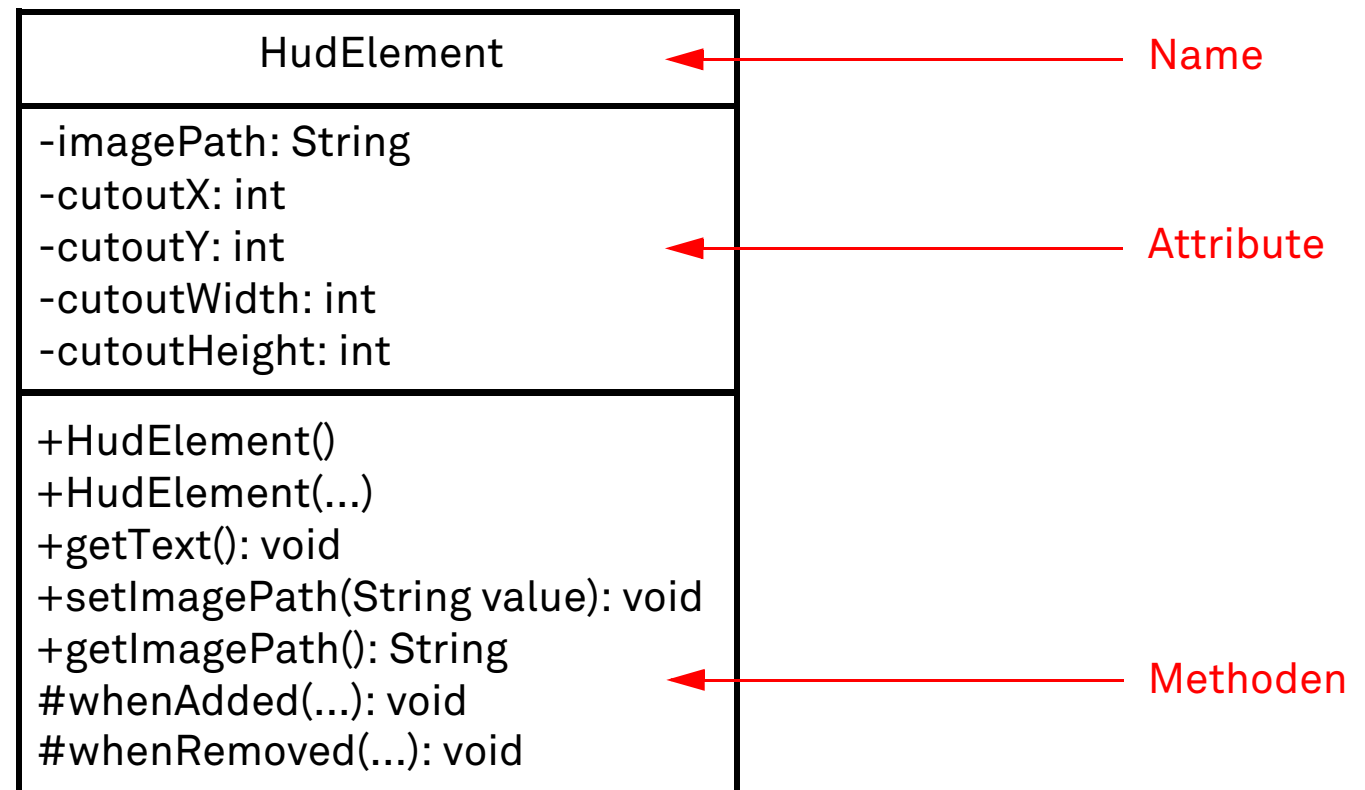
```
public class HudElement {  
    private String imagePath;  
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;  
    public HudElement() {}  
    public HudElement(HudElementOrientation orientation, String imagePath,  
        int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight) { ... }  
    public void setText(String value) { ... }  
    public String getText() { ... }  
    public void setOrientation(HudElementOrientation value) { ... }  
    public HudElementOrientation getOrientation() { ... }  
    public void setImagePath(String value) { ... }  
    public String getImagePath() { ... }  
    ...  
    protected void whenAdded(ViewManager view, Game game) {}  
    protected void whenRemoved(ViewManager view, Game game) {}  
}
```

Attribute

Konstruktoren

Methoden

## Klassendiagramm – Visualisierung einer Klasse



## Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

alternative Darstellungen:

HudElement

nur Klassenname  
ohne Details:  
um Zusammenhänge  
zwischen Klassen zu  
visualisieren

HudElement

-imagePath: String  
-cutoutX: int  
-cutoutY: int  
-cutoutWidth: int  
-cutoutHeight: int

nur mit Attributen:  
um die in einem Objekt  
abgelegten Daten zu  
visualisieren

HudElement

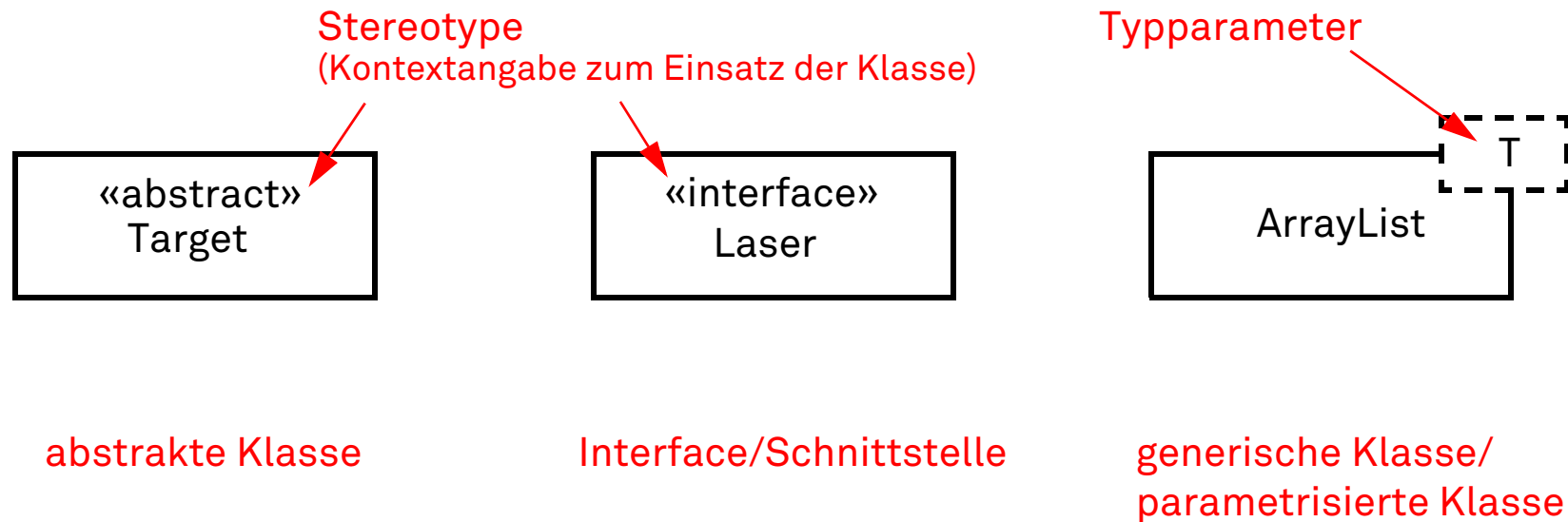
+HudElement()  
+HudElement(...)  
+getText(): void  
+setImagePath(...): void  
+getImagePath(): String  
#whenAdded(...): void  
#whenRemoved(...): void

nur mit Methoden:  
um die von einem Objekt  
angebotenen Operationen  
zu visualisieren

## Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

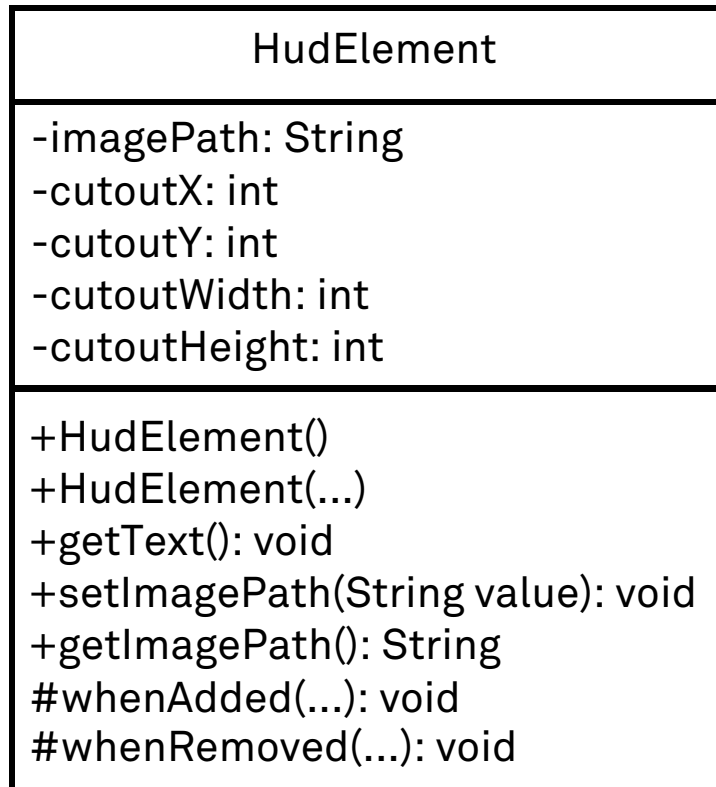
abstrakte Klasse/Interface/generische Klasse



## Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

Deklaration von Eigenschaften: Zugriffsrechte



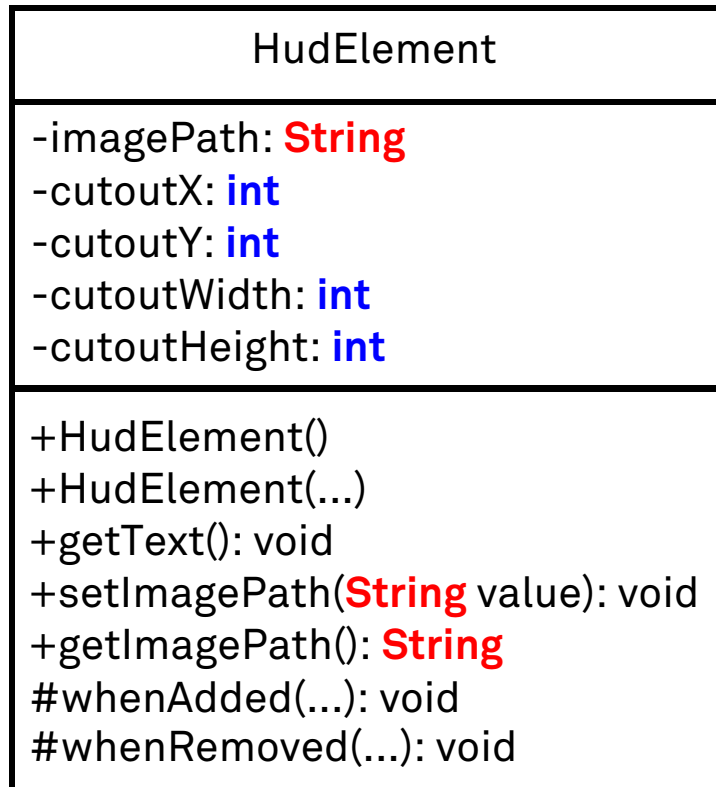
public: +  
private: -  
protected: #  
package: ~



## Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

Deklaration von Attributen: Typangaben

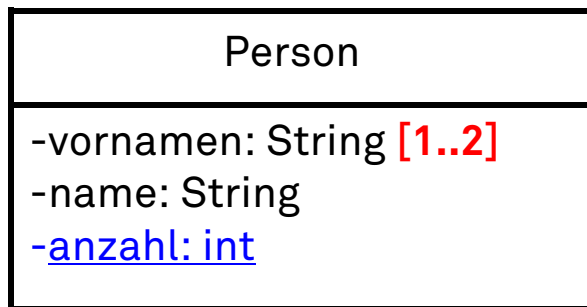


primitive Typen  
oder **Klassen**

## Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

Ergänzende Angaben zu Eigenschaften:



Multiplizität:  
"Eine Person hat 1 bis 2 Vornamen."

```
private String[] vornamen;
```

Aber in Java kann die Zahl der  
Elemente eines Feldes nur zur  
Laufzeit gesteuert werden.

statische Eigenschaft: \_\_\_

## Klassendiagramm – Generalisierung/Spezialisierung/Vererbung

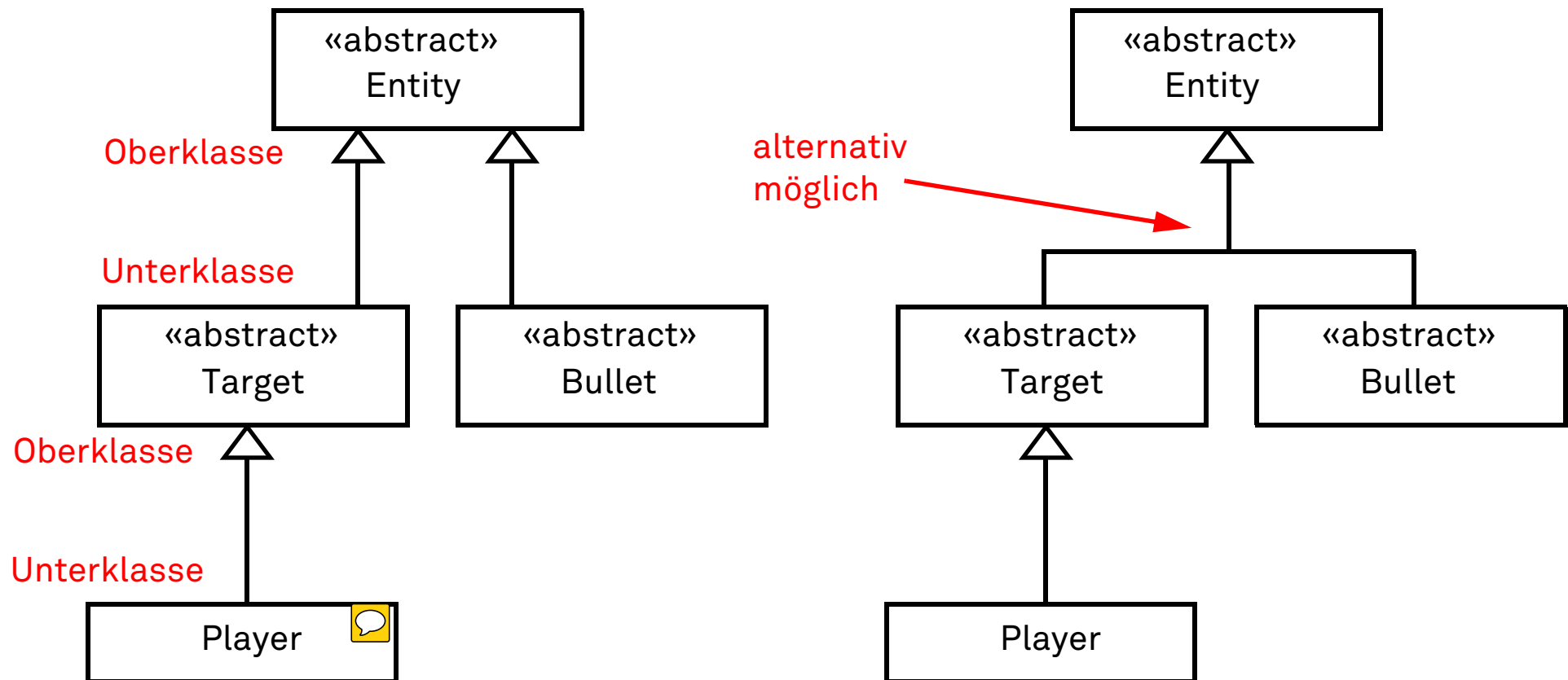
- ❑ Generalisierung/Spezialisierung beschreibt eine Beziehung zwischen Klassen.
- ❑ Eine Klasse kann (mehrere) spezialisierende Klassen besitzen.
- ❑ Jede Instanz einer spezialisierende Klasse wird auch als Instanz der allgemeineren Klasse aufgefasst.
- ❑ Generalisierung/Spezialisierung beschreibt eine **Typbeziehung** zwischen Klassen.
- ❑ Die allgemeinere Klasse heißt Ober- oder Superklasse.
- ❑ Die spezialisierende Klasse heißt Unter- oder Subklasse.
- ❑ Das Vererbungskonzept wie in Java ist eine mögliche Umsetzung des Konzepts der Generalisierung/Spezialisierung.  
Dabei besitzen alle Objekte der spezialisierenden Klasse alle Eigenschaften der allgemeineren Klasse:  
Attribute, Methoden, Beziehungen zu anderen Klassen
- ❑ Es entstehen Spezialisierungshierarchien.

### Anmerkung:

Grundsätzlich kann eine Klasse mehrere Klassen spezialisieren, also mehrere Oberklassen besitzen. (engl. multiple inheritance)

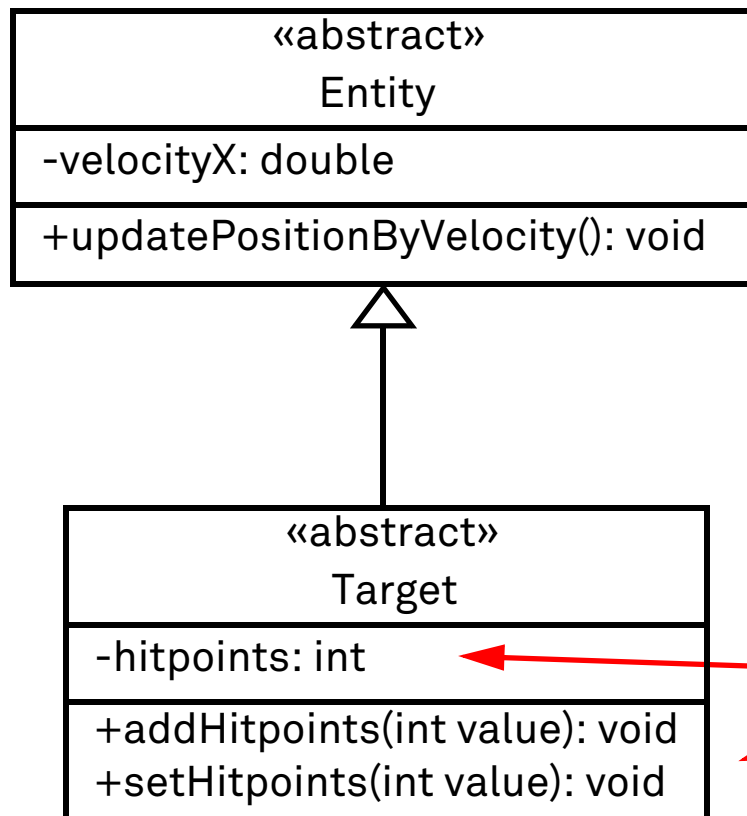
## Klassendiagramm – Generalisierung/Spezialisierung

(Fortsetzung)



## Klassendiagramm – Generalisierung/Spezialisierung

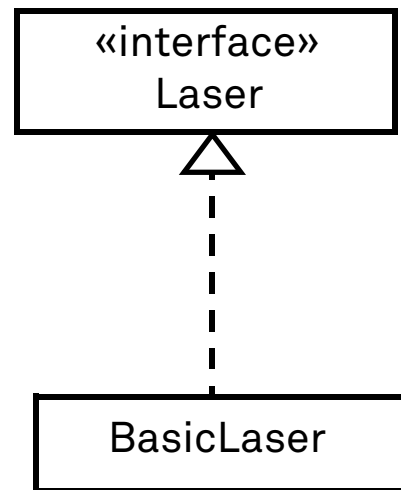
(Fortsetzung)



In Unterklassen werden nur  
zusätzliche Attribute und Methoden aufgeführt.

## Klassendiagramm – Realisierung/implementierung

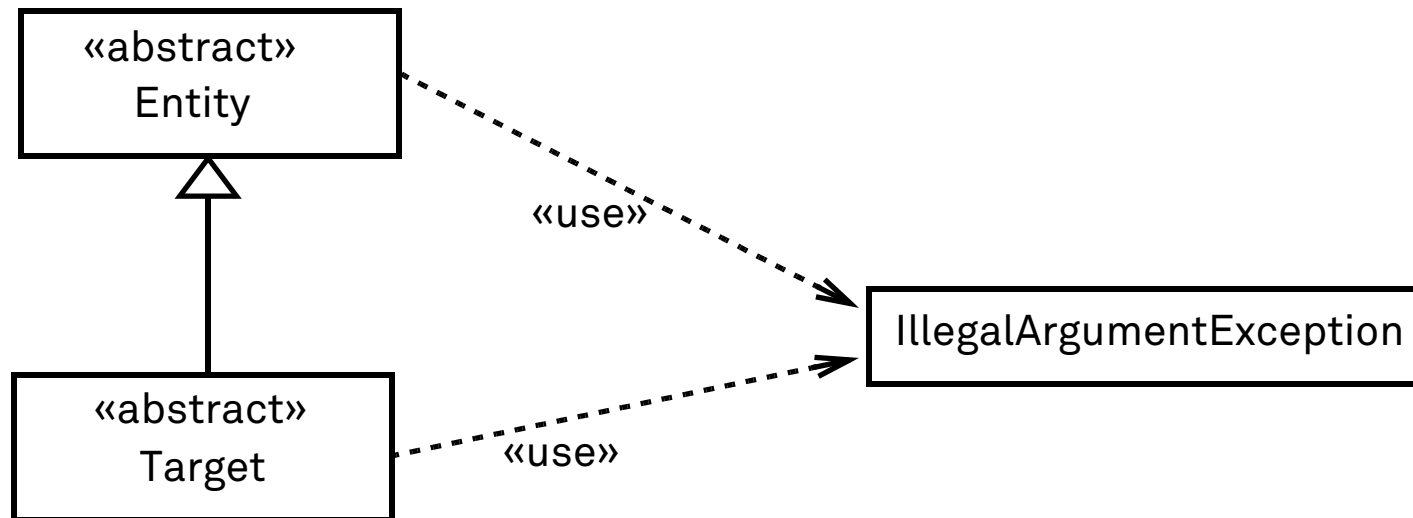
- ❑ Eine Realisierung beschreibt eine Beziehung zwischen einem Interface und einer Klasse.
- ❑ Ein Interface kann durch (mehrere) Klassen realisiert werden.
- ❑ Eine Klasse kann mehrere Schnittstellen realisieren.
- ❑ Realisierung beschreibt eine **Typbeziehung** zwischen Klassen.



## Klassendiagramm – Abhängigkeit

Eine Abhängigkeit liegt dann vor, wenn in einer Klasse *K* eine andere Klasse *U* benötigt wird um

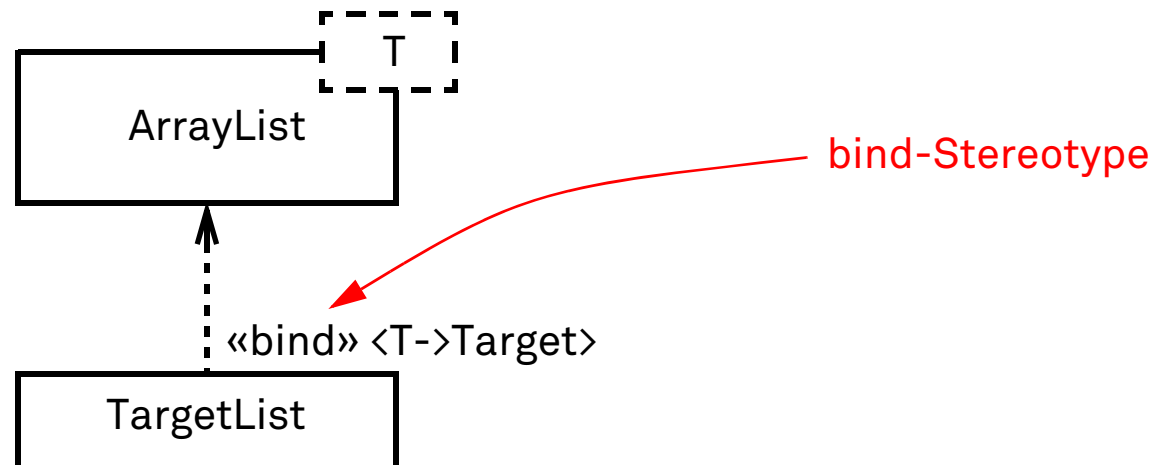
- ❑ in einer Methode von *K* über einen Parameter Zugriff auf ein Objekt von *U* zu bekommen .
- ❑ in einer Methode von *K* ein Objekt von *U* zu erzeugen.



## Klassendiagramm – Abhängigkeit

(Fortsetzung)

### Binden von Typparametern



Legt eine neue Klasse  
mit eigenem Namen an.

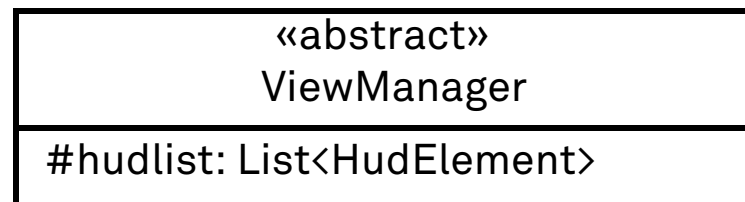


## Klassendiagramm – Attribute von Klassen

(Fortsetzung)

Attribute, der Typ eine Klasse ist, können auf zwei Arten modelliert werden:

- textuell in der Visualisierung der Klasse.



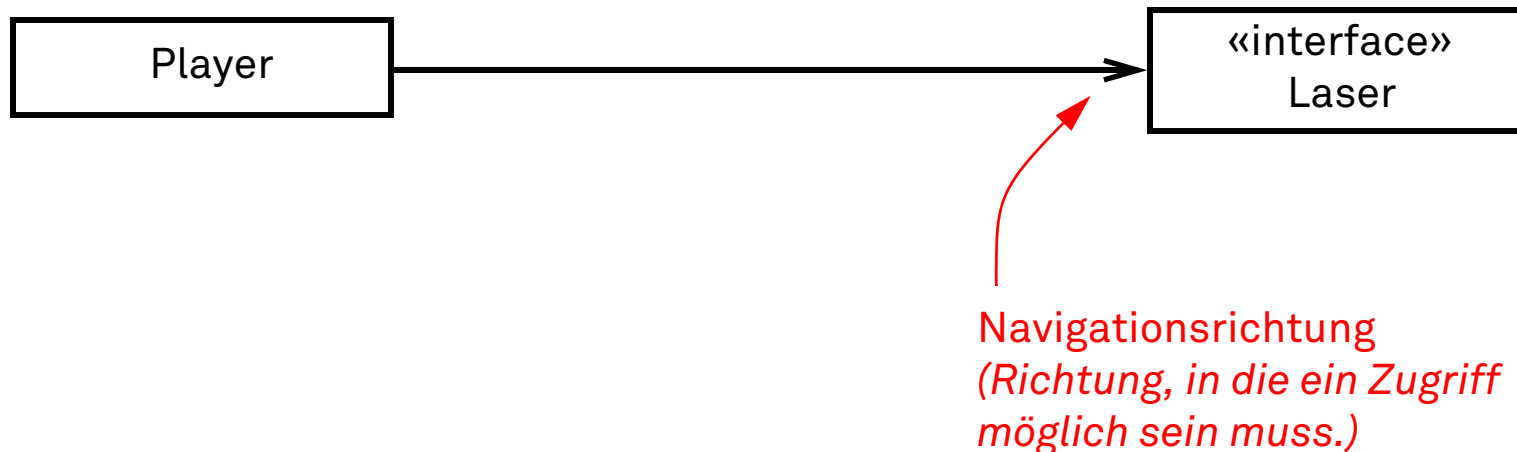
Dann können aber die Beziehungen zwischen den Klassen nicht unmittelbar in der Darstellung erkannt werden.

- in Form einer sogenannten Assoziation als grafische Verbindung zwischen Klassen.

Das ist die bevorzugte Darstellung,  
um Beziehungen zwischen Klassen sofort wahrnehmen zu können.

## Klassendiagramm – Assoziation

- ❑ Eine **A**ssoziation beschreibt eine Beziehung zwischen den Objekten von zwei Klassen:
  - ein Objekt der Klasse *K* *kennt* Objekt(e) der Klasse *A*,
  - ein Objekt der Klasse *K* *besitzt/hat* Objekt(e) der Klasse *A*.
- ❑ Im Gegensatz zu einer Abhängigkeit ist bei einer Assoziation die Beziehung zwischen den Objekten derart **dauerhaft**, dass sie über die Ausführung einer einzelnen Methode hinweg besteht.

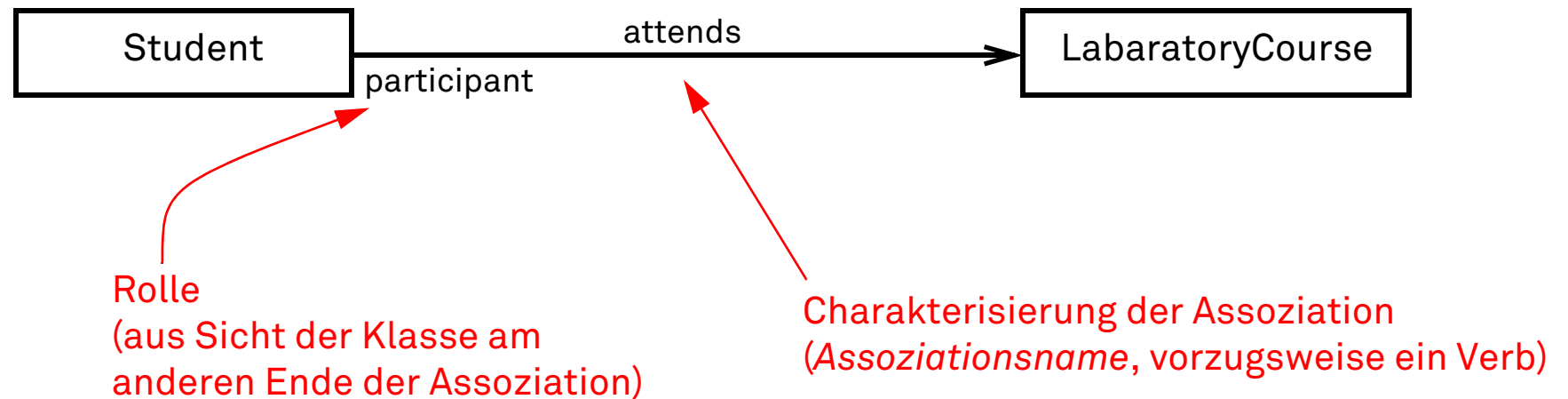


## Klassendiagramm – Assoziation

(Fortsetzung)

Präzisierungsmöglichkeiten

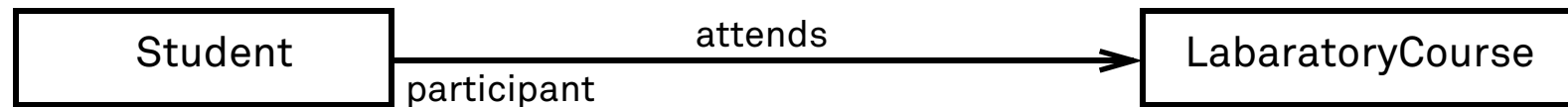
*(Beispiel nicht aus SWT-Starfighter)*



## Klassendiagramm – Assoziation

(Fortsetzung)

Vorgaben für die Umsetzung in Java  
(*Beispiel nicht aus SWT-Starfighter*)



```
public class Student {
    LaboratoryCourse myCourse;
}

public class LaboratoryCourse { }
```

## Klassendiagramm – Assoziation

(Fortsetzung)

Präzisierungsmöglichkeiten

*(Beispiel nicht aus SWT-Starfighter)*



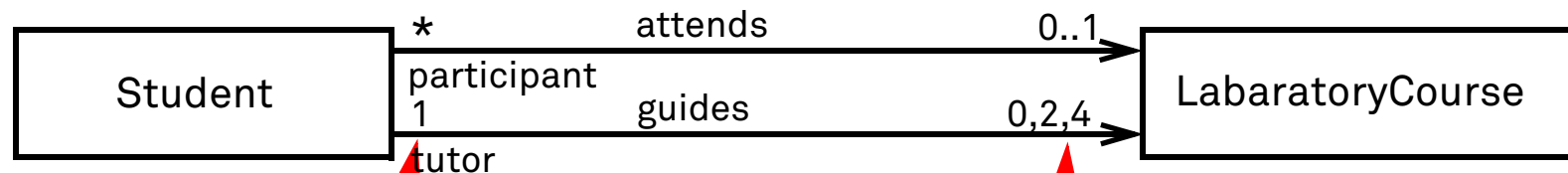
Mehrere Assoziationen zwischen den gleichen Klassen sind möglich und üblich, um verschiedene Arten von Beziehungen zu verdeutlichen.

## Klassendiagramm – Assoziation

(Fortsetzung)

Präzisierungsmöglichkeiten

(Beispiel nicht aus SWT-Starfighter)



### Multiplizität

ist eine Angabe, wie viele **unterscheidbare(!)** Objekte einer Klasse einem Objekt der am anderen Ende liegenden Klasse bekannt sein können/müssen.

- Standardwert (ohne explizite Angabe): unbestimmt
- \* = explizit beliebig viele (einschließlich 0)
- 1..\* = beliebig viele, aber mindestens 1

Aussage des Diagramms:

Ein Student kann als einer von beliebig vielen Teilnehmern an einer Übungsgruppe teilnehmen.

Ein Student kann als (dann einzige) Tutorin 2 oder 4 Übungsgruppen leiten.

## Klassendiagramm – Assoziation

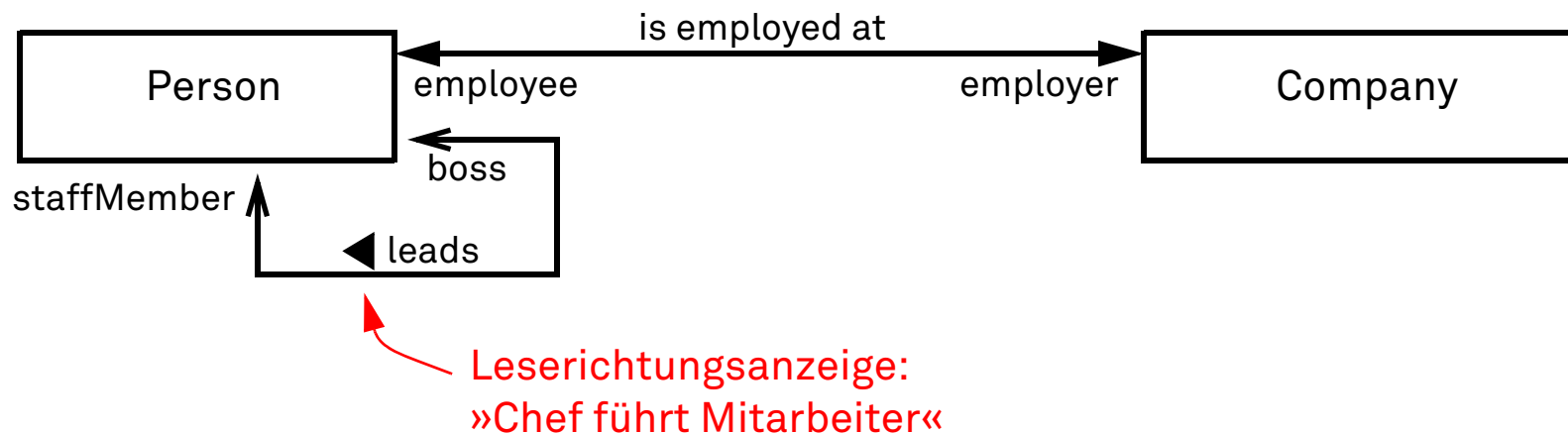
(Fortsetzung)

### Präzisierungsmöglichkeiten

(Beispiel nicht aus SWT-Starfighter)

#### Navigationsrichtung

- ❑ auch möglich: Assoziationen **ohne** Richtungspfeil mit **unbestimmter** Navigationsrichtung
  - kann insbesondere auch in beide Richtungen navigierbar sein.
- ❑ auch möglich: Assoziationen **mit beidseitigen** Richtungspfeilen
  - erzwingt Navigationsmöglichkeit in beide Richtungen.

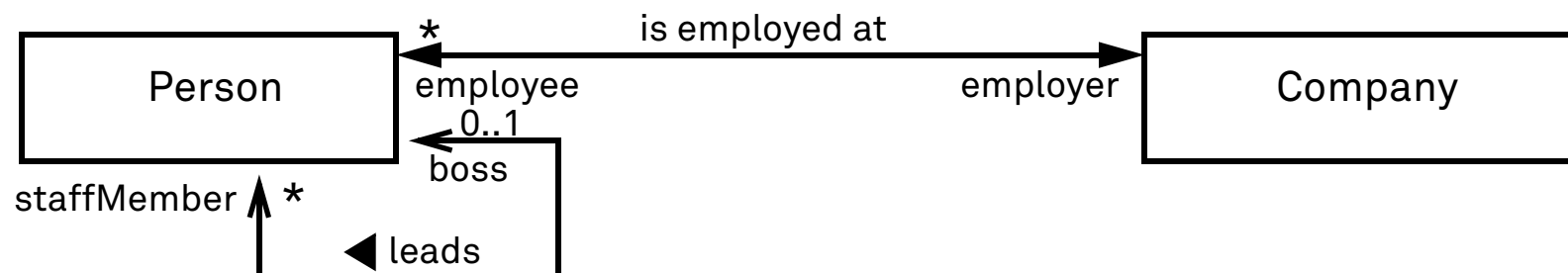


## Klassendiagramm – Assoziation

(Fortsetzung)

Präzisierungsmöglichkeiten: mit Angabe von Multiplizitäten  
(Beispiel nicht aus SWT-Starfighter)

1





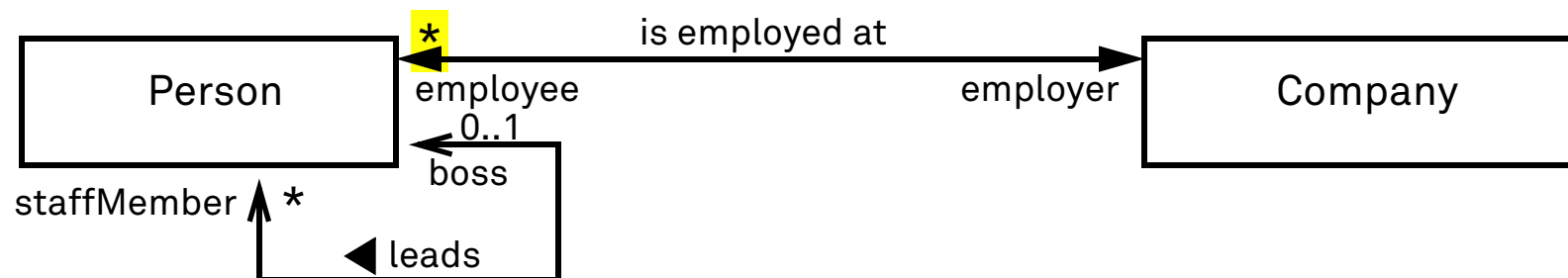
## Klassendiagramm – Assoziation

(Fortsetzung)

Vorgaben für die Umsetzung in Java  
(*Beispiel nicht aus SWT-Starfighter*)

```
public class Person {
    Company employer;
    Person boss;
    Person[] staffMember;
}
```

```
public class Company {
    Person[] employee;
}
```



## Klassendiagramm – Assoziation

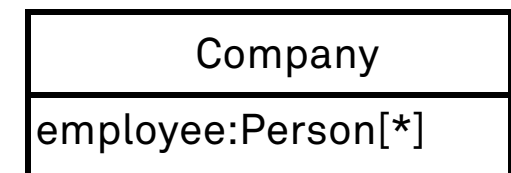
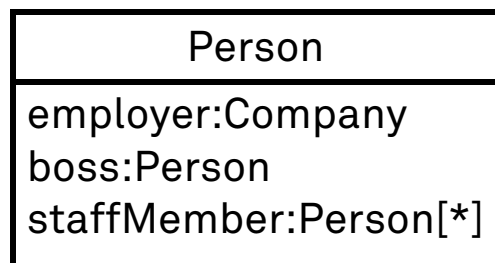
(Fortsetzung)

Vorgaben für die Umsetzung in Java  
(*Beispiel nicht aus SWT-Starfighter*)

```
public class Person {  
    Company employer;  
    Person boss;  
    Person[] staffMember;  
}
```

```
public class Company {  
    Person[] employee;  
}
```

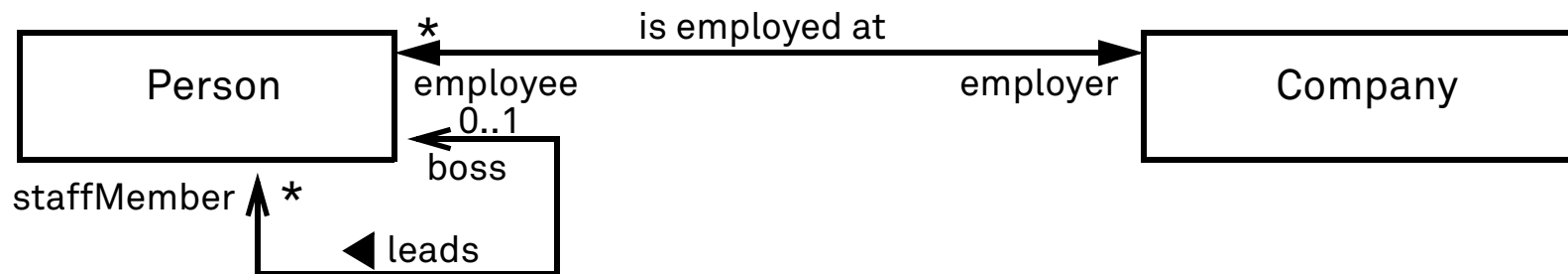
entspricht aber auch



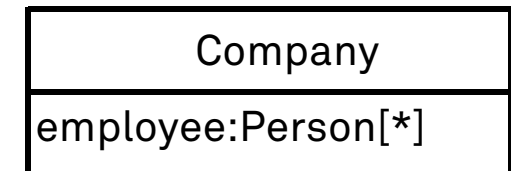
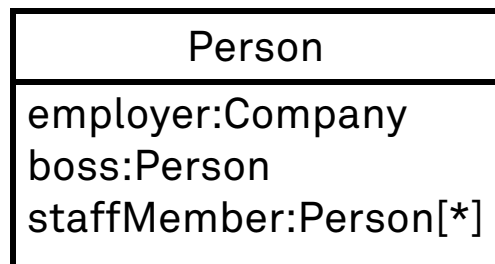
## Klassendiagramm – Assoziation

(Fortsetzung)

- Eine Assoziation entspricht einem Attribut mit komplexem Typ (= Klasse).  
(Beispiel nicht aus SWT-Starfighter)



entspricht also



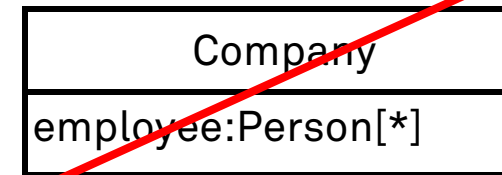
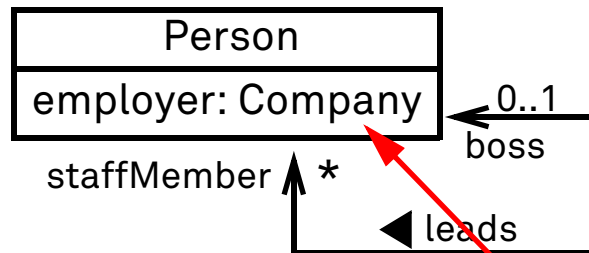
**Aber:** Die untere Darstellung enthält keine Visualisierung der Beziehungen zwischen Person- und Company-Objekten.

## Klassendiagramm – Assoziation

(Fortsetzung)

Eine Assoziation entspricht einem Attribut mit komplexem Typ (= Klasse).

*(Beispiel nicht aus SWT-Starfighter)*



Die textuelle Version ist nur dann sinnvoll,  
wenn die Klasse Company nicht im Diagramm  
vorkommen soll.

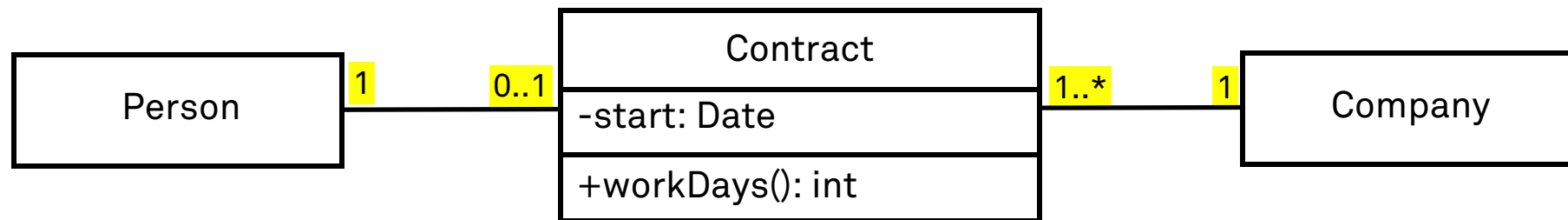
Meist ist das bei Klassen wie z.B. String der Fall,  
die zum Standardumfang der eingesetzten  
Programmiersprache gehören.

## Klassendiagramm – Assoziation

(Fortsetzung)

Assoziation – Abstraktionsmöglichkeit: **Assoziationsklasse**

*(Beispiel nicht aus SWT-Starfighter)*



Die Beziehung zwischen Arbeitnehmer und Arbeitgeber ist normalerweise mit dem Vorhandensein eines Arbeitsvertrags verknüpft.

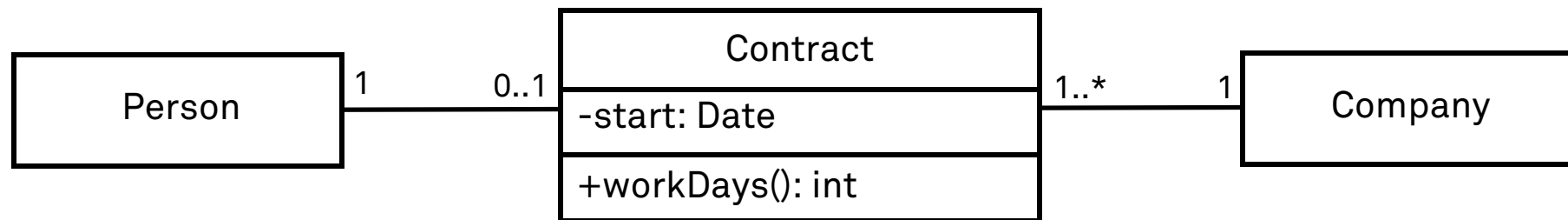
Dieser Kontext kann aber nur durch Analyse aller drei Klassen und der dazwischen stehenden Beziehungen erkannt werden.

## Klassendiagramm – Assoziation

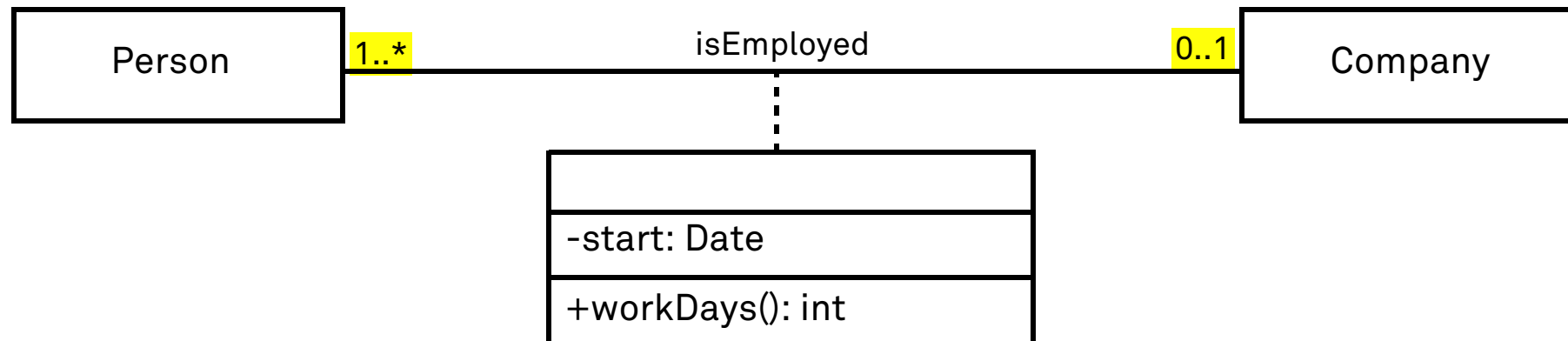
(Fortsetzung)

Assoziation – Abstraktionsmöglichkeit: **Assoziationsklasse**

*(Beispiel nicht aus SWT-Starfighter)*



**Der Kontext kann durch folgende Modellierung deutlich gemacht werden:**



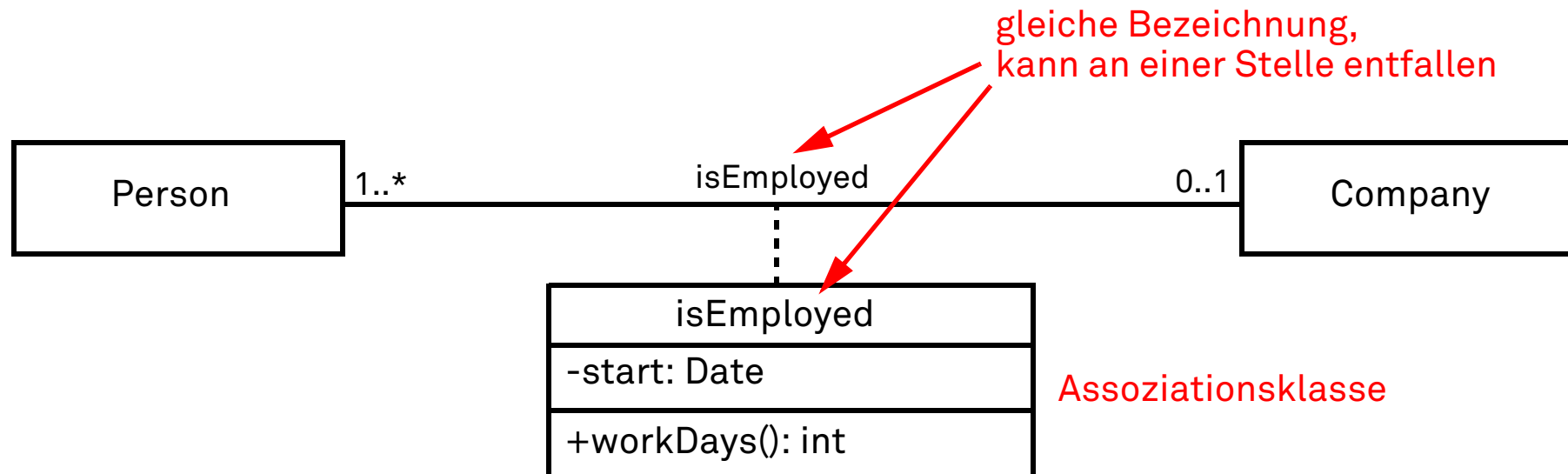
## Klassendiagramm – Assoziation

(Fortsetzung)

### Assoziation – Abstraktionsmöglichkeit: **Assoziationsklasse**

*(Beispiel nicht aus SWT-Starfighter)*

- ❑ Eine Assoziationsklasse legt Attribute und Operationen zu einer Assoziation an, die nur dann existieren, wenn eine tatsächlich Verbindung zwischen Objekten besteht.
- ❑ im Beispiel:  
Das Attribut `start` existiert nur, wenn eine Person in einer Company beschäftigt ist, also eine Verbindung zwischen einem Objekt der Klasse `Person` und einem Objekt der Klasse `Company` besteht.

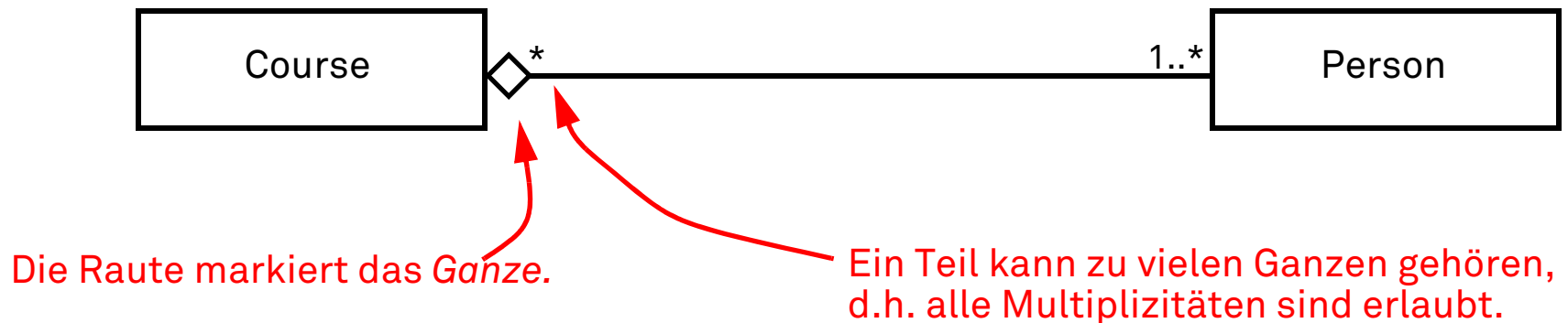


## Klassendiagramm – Aggregation

**Aggregation** ist eine spezielle Form der Assoziation:

- ❑ Die Aggregation ist eine Teile-Ganzes-Beziehung mit der Aussage: "das *Ganze* besteht aus den *Teilen*".
- ❑ **Semantische Unterschiede zur normalen Assoziation sind nicht (vor-)definiert.**
- ❑ Die Semantik darf aber selbst präzisiert werden.  
(Dieses ist nur dann sinnvoll, wenn eine Arbeitsgruppe eine einheitliche Semantik nutzt.)

(Beispiel nicht aus SWT-Starfighter)





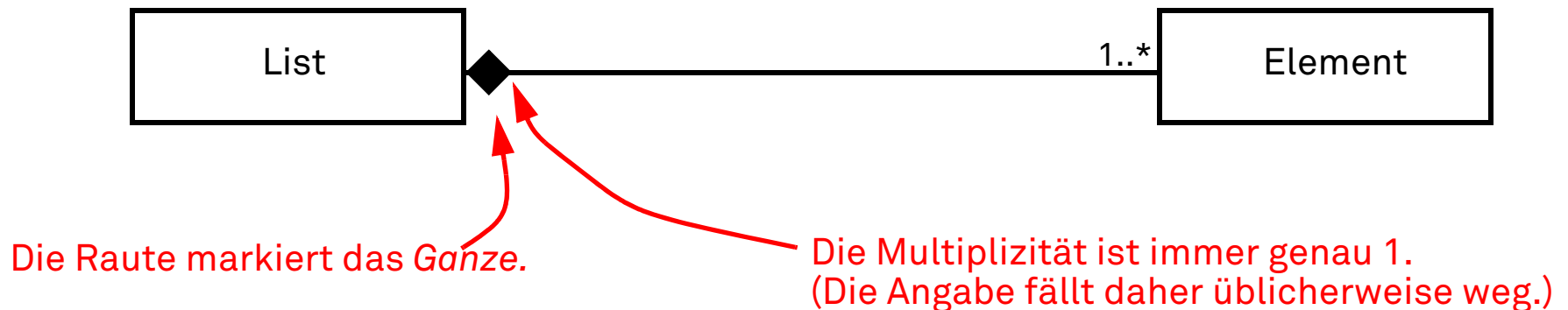
## Klassendiagramm – Komposition

**Komposition** ist eine spezielle Form der Aggregation (und damit auch der Assoziation).

Die Komposition ist auch eine Teile-Ganzes-Beziehung, aber mit zwei Bedingungen:

- ❑ Die *Teile* existieren nur gemeinsam mit dem *Ganzen*, sie sind von dem Ganzen **existentiell abhängig**.
- ❑ Jedes *Teil* gehört zu **genau nur einem** *Ganzen*.

(Beispiel nicht aus SWT-Starfighter)



## Klassendiagramm – Anmerkungen zu Assoziationen

- ❑ In den meisten Fällen reichen *normale* Assoziationen zur Modellierung aus.
- ❑ "besteht aus" (Aggregation) ist eine umgangssprachliche Beschreibung.
- ❑ Aggregation hat in UML wenig Aussagekraft.
- ❑ Aggregation kann dazu dienen, eine engere Beziehung zwischen Klassen auszudrücken.
- ❑ Komposition hat strenge Konsequenzen.
- ❑ Komposition sollte nur bei klarer *Indikation* verwendet werden.
- ❑ Komposition tritt insbesondere bei technischen Konstrukten der Implementierung auf:  
Ein Menü-Objekt existiert nur gemeinsam mit seinem Fenster-Objekt.
- ❑ Komposition tritt selten bei den konzeptionellen Konstrukten der Anwendung auf:  
Personen eines Kurses können auch ohne den Kurs existieren.

## Klassendiagramm – Anmerkungen zur Syntax

- ❑ Die Syntax für Klassendiagramme ist sehr reichhaltig.
- ❑ Hier ist nur ein Ausschnitt dargestellt worden, der in der weiteren Vorlesung verwendet wird.
- ❑ Beim Einsatz aller syntaktischen Möglichkeiten enthalten Klassendiagramme leicht **sehr viele** Modellelemente:
  - Attribute und Operationen (insbesondere mit zusätzlichen Angaben) erzeugen umfangreiche grafische Repräsentationen für Klassen.
  - Assoziations- und Rollennamen benötigen lange Assoziationskanten.
  - Bei vielen gleichzeitig gezeichneten Assoziationen kann die Zuordnung der ergänzenden Elemente eventuell visuell nicht eindeutig vorgenommen werden
    - die Lesbarkeit des Diagramms leidet.
  - Der Verlauf kreuzender Assoziationen und Spezialisierungen kann eventuell visuell gar nicht eindeutig bestimmt werden.
- ❑ Daher sollten in Klassendiagrammen nur die Aussagen ausgedrückt werden, wie in der Entwicklung unbedingt benötigt werden.

## Klassendiagramm (Wiederholung)

Klassendiagramme können genutzt werden, um

- ❑ Implementierungen zu visualisieren,
- ❑ Vorgaben für die (direkte) Umsetzung in eine Implementierung zu liefern,
- ❑ Vorgaben für die Planung einer Implementierung zu liefern:  
Das Klassendiagramm veranschaulicht dann nur eine idealisierte Struktur.
- ❑ Daten, Operationen und Abhängigkeiten in objektorientierter Form zu visualisieren  
– unabhängig davon, ob überhaupt eine objektorientierte Implementierung beabsichtigt ist.

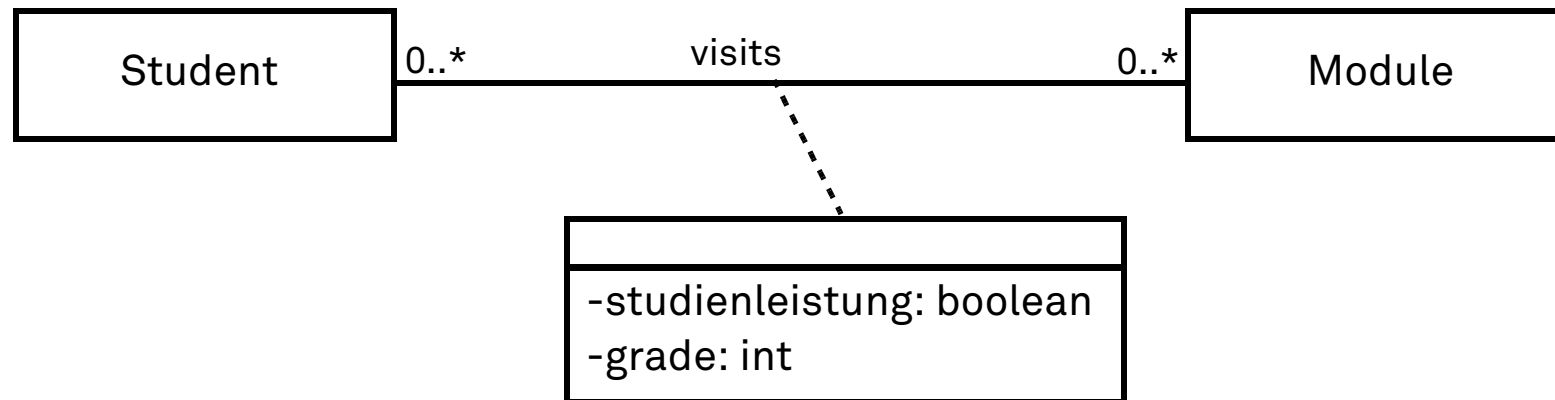
Die Modellierung kann auf unterschiedlichen Abstraktionsniveaus erfolgen und ist abhängig vom

- ❑ Zeitpunkt der Modellierung im Entwicklungsvorgang
- ❑ Zielsetzung der Modellierung  
(Diskussion/Ideenskizze, Programmiervorgabe, Programmdokumentation)
- ❑ Größe des Modells

## Klassendiagramm – Beispiel

Beschreibung von Daten, Operationen und Abhängigkeiten in objektorientierter Form  
(unabhängig von einer Implementierung)

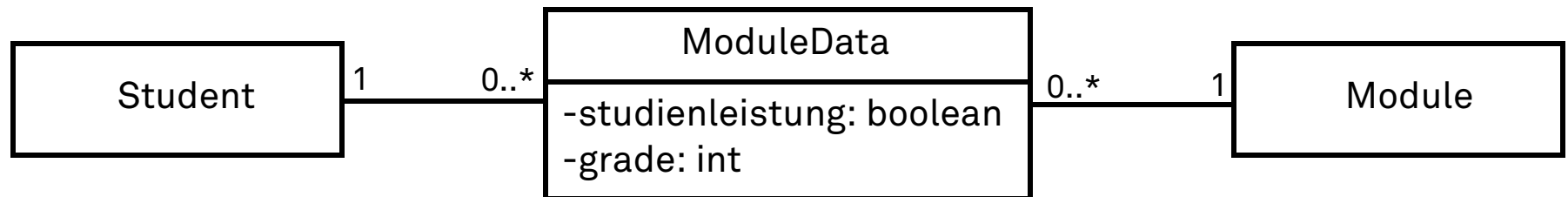
*(Beispiel nicht aus SWT-Starfighter)*



## Klassendiagramm – Beispiel

(Fortsetzung)

Vorgabe für die Planung einer Implementierung:  
Das Klassendiagramm veranschaulicht dann nur eine idealisierte Struktur  
(Beispiel nicht aus SWT-Starfighter)



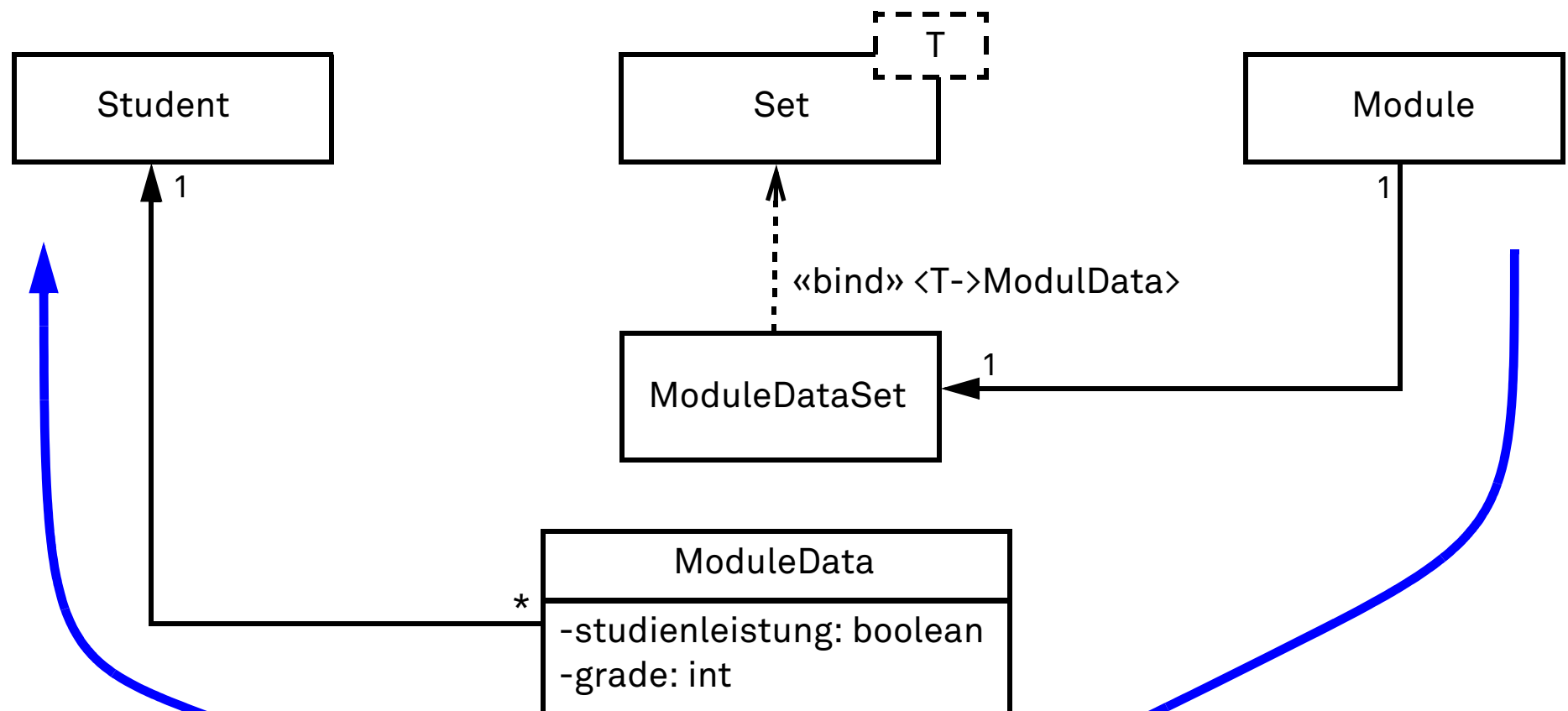
## Klassendiagramm – Beispiel

(Fortsetzung)

Vorgabe für die (direkte) Umsetzung in eine Implementierung:

Nutzung der generischen Klasse Set<T>

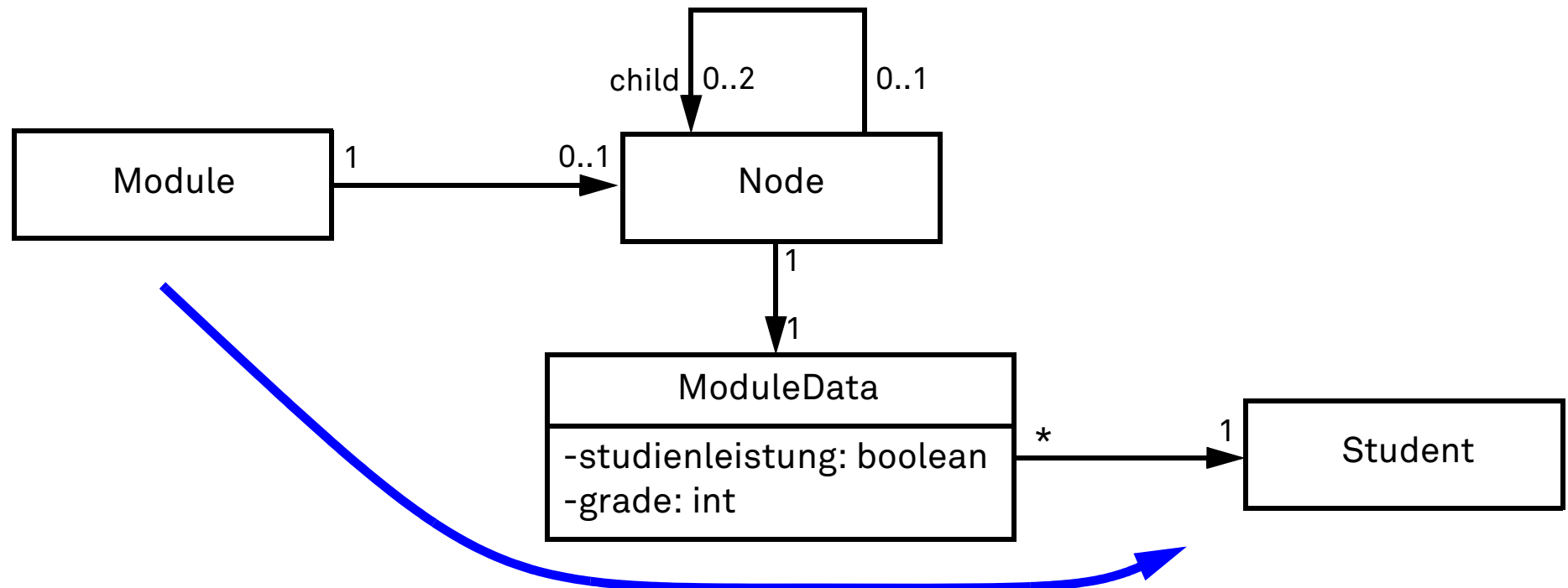
– erlaubt nur die Navigation von einem Modul zu den teilnehmenden Studierenden



## Klassendiagramm – Beispiel

(Fortsetzung)

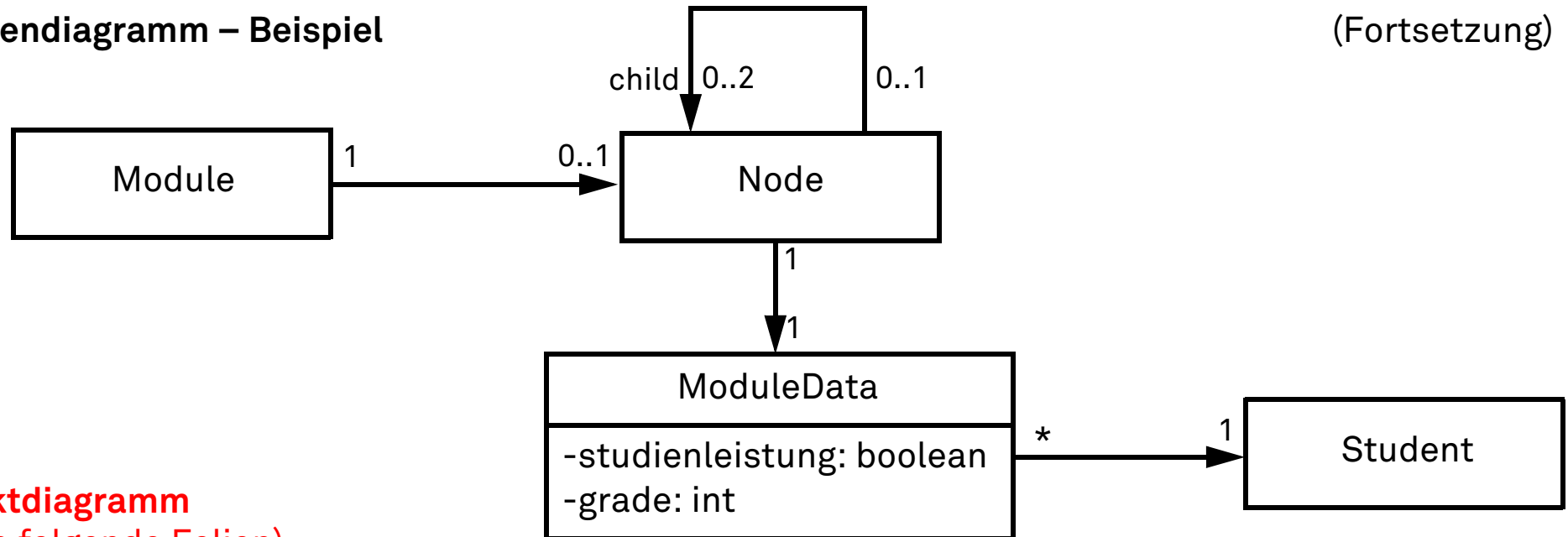
alternative Vorgabe für die (direkte) Umsetzung in eine Implementierung:  
Nutzung eines binären Suchbaums  
– erlaubt nur die Navigation von einem Modul zu den teilnehmenden Studierenden



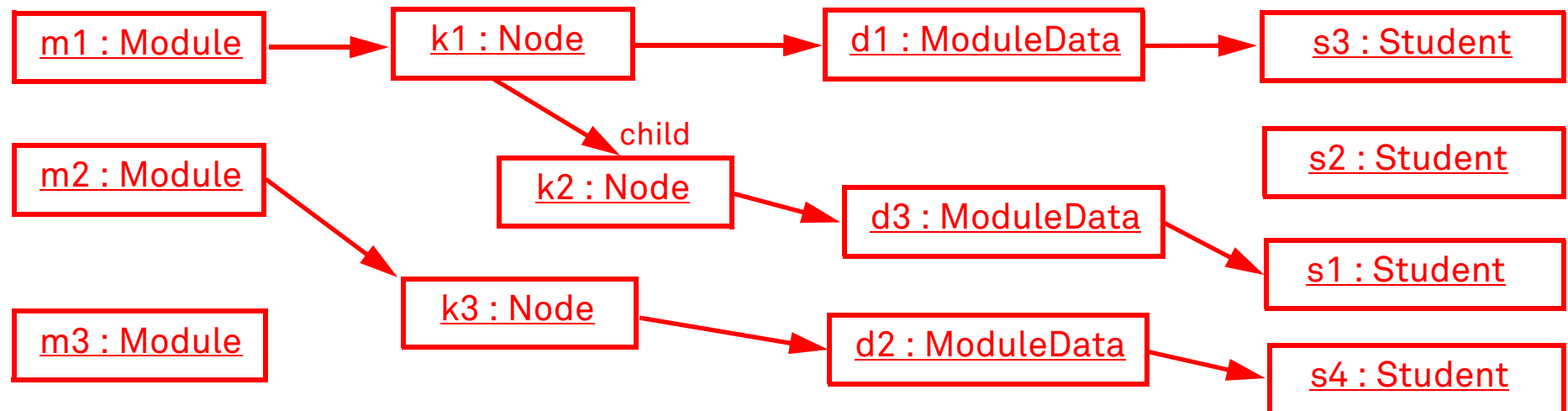


## Klassendiagramm – Beispiel

(Fortsetzung)



## Objektdiagramm (siehe folgende Folien)



## Klassendiagramm (Wiederholung)

Klassendiagramme können genutzt werden, um

- ❑ Implementierungen zu visualisieren,
- ❑ Vorgaben für die (direkte) Umsetzung in eine Implementierung zu liefern,
- ❑ Vorgaben für die Planung einer Implementierung zu liefern:  
Das Klassendiagramm veranschaulicht dann nur eine idealisierte Struktur.
- ❑ Daten, Operationen und Abhängigkeiten in objektorientierter Form zu visualisieren  
– unabhängig davon, ob überhaupt eine objektorientierte Implementierung beabsichtigt ist.

Anmerkungen:

- ❑ Implementierungsnahe Visualisierungen bestehen häufig aus vielen Klassen.
- ❑ Klassen für Datenstruktur (Set) können Vielfachbeziehungen auch ohne Nutzung von \*-Multiplizitäten realisieren.
- ❑ Rekursive Datenstrukturen können große Objektgeflechte auch ohne Nutzung von \*-Multiplizitäten aufbauen.
- ❑ daher:

**Die Modellierung muss der gewünschten Abstraktionsstufe entsprechen.**

## Klassendiagramm – Objektdiagramm

### Klassendiagramm

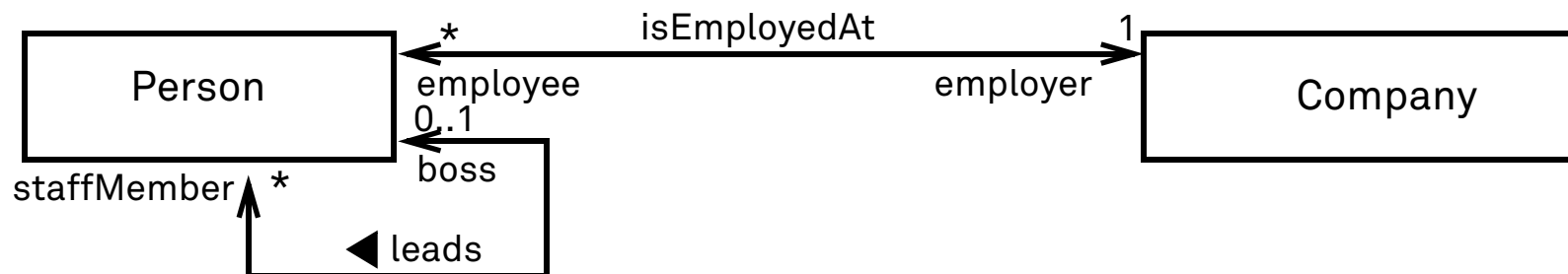
- ❑ beschreibt Beziehungen zwischen **Klassen** in Form von
  - Spezialisierung,
  - Realisierung,
  - Abhängigkeiten.
- ❑ beschreibt die **möglichen** Beziehungen zwischen **Objekten/Instanzen** der Klassen durch Assoziationen.

### Objektdiagramm

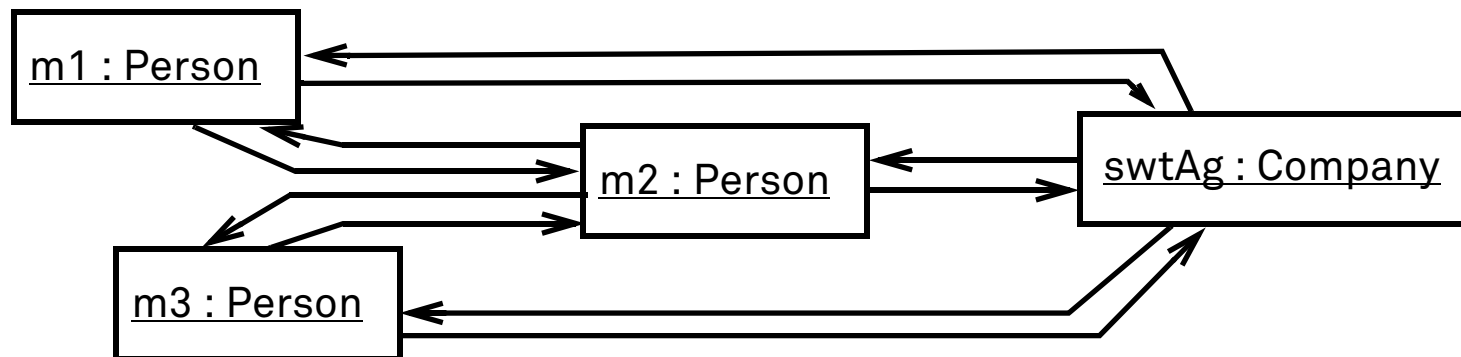
- ❑ zeigt **eine konkrete** Konfiguration von Objekten, die die Vorgaben des zugehörigen Klassendiagramms einhält.
- ❑ enthält ausschließlich Objekte.
- ❑ enthält nur die zwischen diesen Objekten bestehenden Verbindungen.
- ❑ ist i.d.R. nur ein Beispiel aus vielen möglichen Konfigurationen.
- ❑ dient zur Verdeutlichung der Aussagen des zugehörigen Klassendiagramms.
- ❑ kann **als Folge von mehreren** Objektdiagrammen Änderungen an Objekten und ihren Verbindungen visualisieren.

## Objektdiagramm

Klassendiagramm: Beziehungen zwischen Klassen/Typen



## Objektdiagramm: Beziehungen zwischen Objekten der Klassen

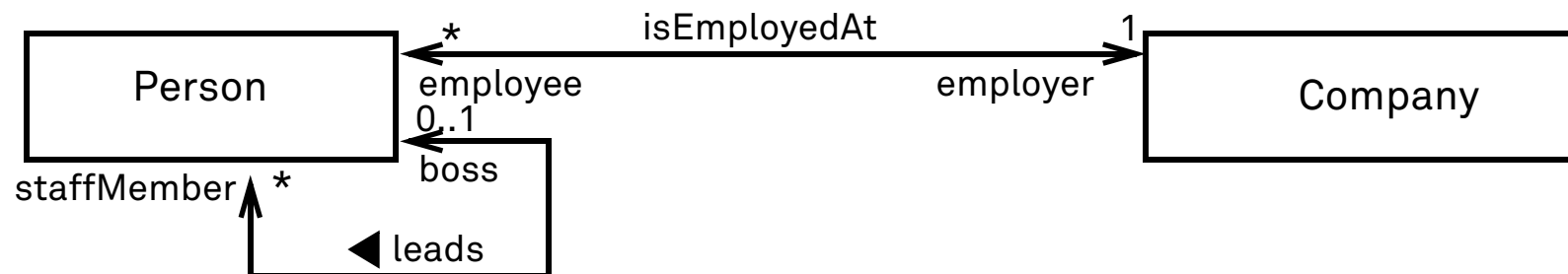


Literatur: Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 49-51  
[http://link.springer.com/chapter/10.1007/3-540-30950-0\\_4](http://link.springer.com/chapter/10.1007/3-540-30950-0_4)

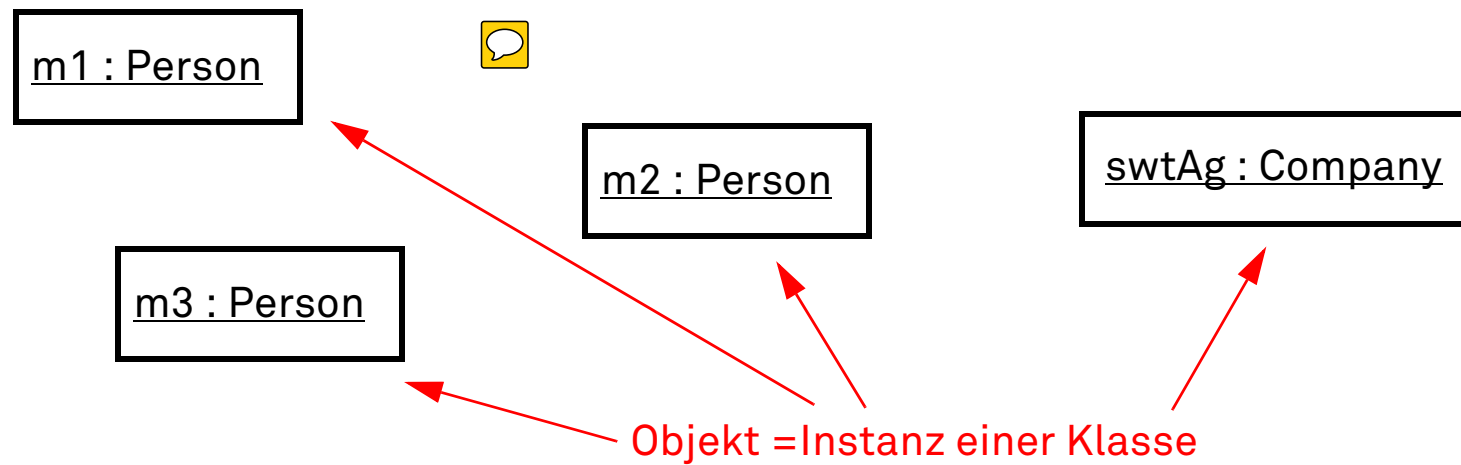
## Objektdiagramm

(Fortsetzung)

Klassendiagramm:



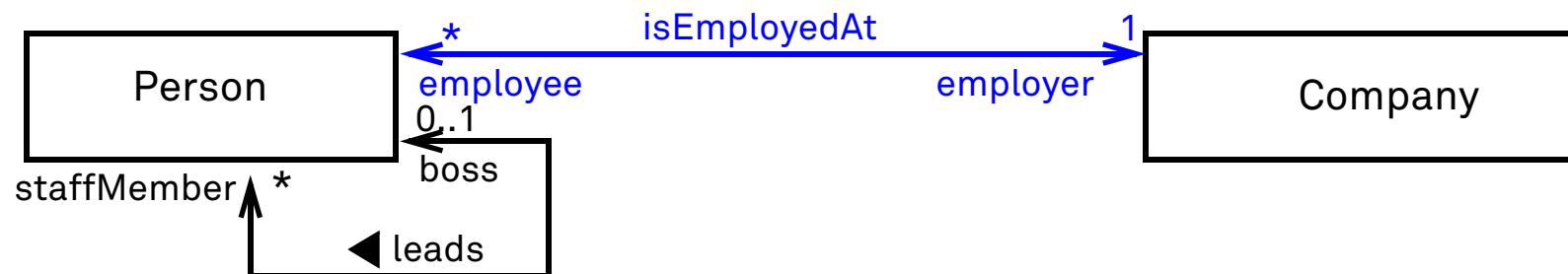
Objektdiagramm:



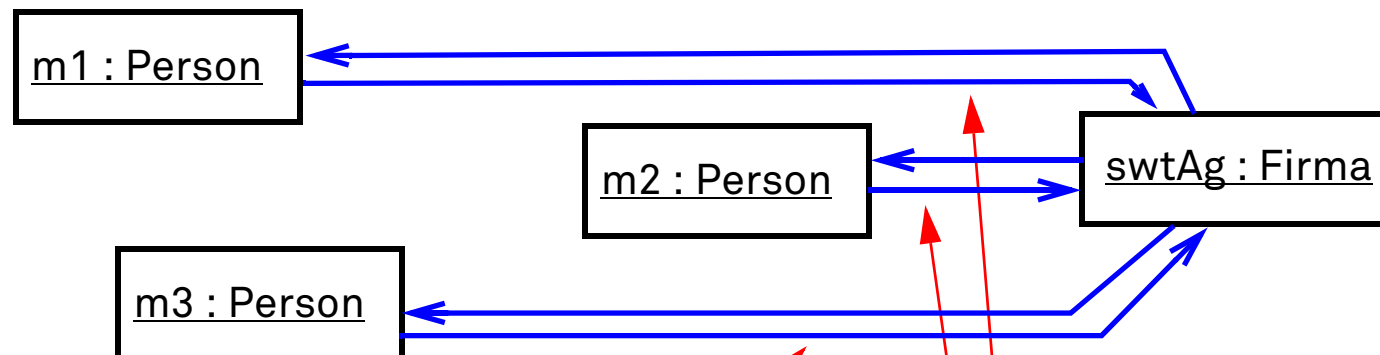
## Objektdiagramm

(Fortsetzung)

Klassendiagramm:



Objektdiagramm:

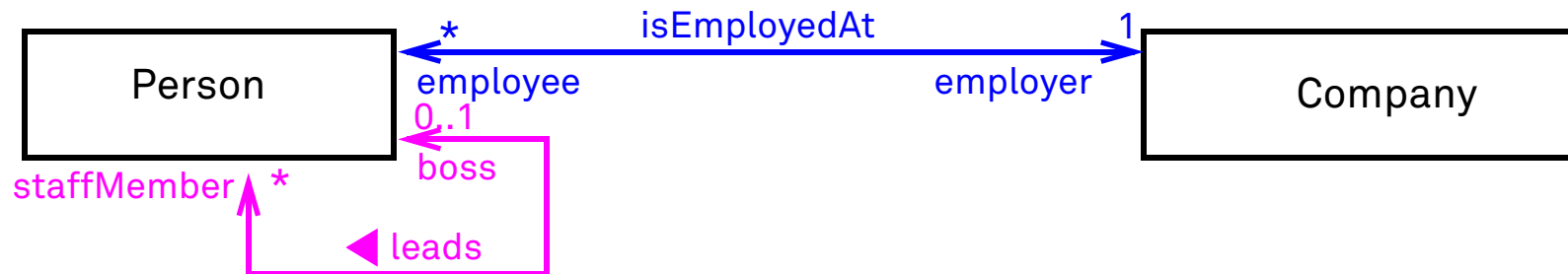


Link = konkrete Verbindung gemäß einer Assoziation

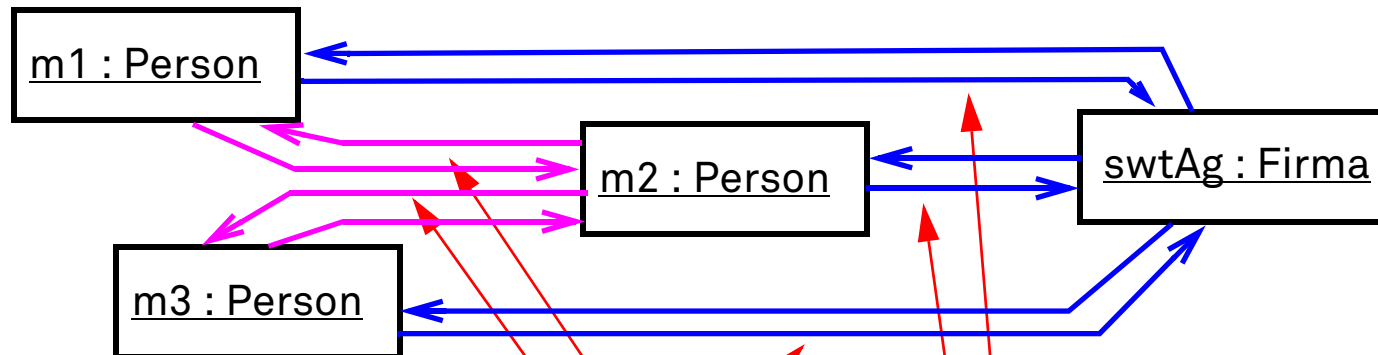
## Objektdiagramm

(Fortsetzung)

Klassendiagramm:



Objektdiagramm:



Link = konkrete Verbindung gemäß einer Assoziation

## Objektdiagramm

(Fortsetzung)

### Ergänzung: Benennung von Objekten

p : Person

Objekt mit dem Namen **p** ist Instanz der Klasse **Person**

: Person

Objekt mit unbekanntem Namen ist Instanz der Klasse **Person**

x

Objekt mit dem Namen **x** ist Instanz einer unbekannten Klasse

.

Objekt mit unbekanntem Namen und unbekannter Klasse

.

ein weiteres Objekt mit unbekanntem Namen und unbekannter Klasse



## Objektdiagramm

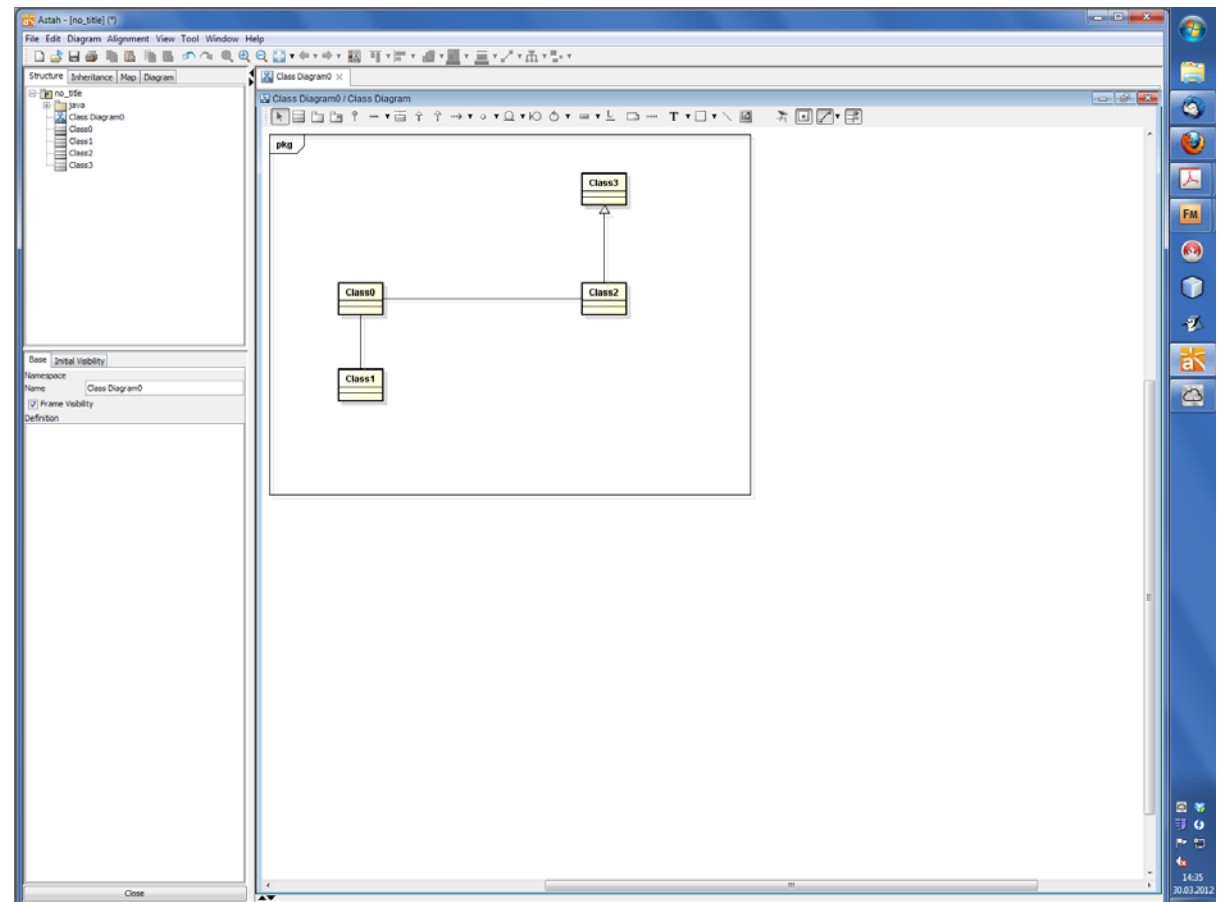
(Fortsetzung)

### Zusammenfassung

- ❑ Objektdiagramme enthalten ausschließlich Objektsymbole und Links (zwischen den Objektsymbolen)
- ❑ Falls für ein Objekt eine Klasse angegeben wird, müssen von dieser auch Objekte erzeugt werden dürfen. Abstrakte Klassen können nicht zu Objekten führen.
- ❑ Zwei Objekte dürfen nur durch einen Link verbunden sein, wenn es eine passende Assoziation im Klassendiagramm gibt.
- ❑ Die Anzahl der abgehenden und eingehenden Links muss die Vorgaben der Multiplizitäten der Assoziationen des Klassendiagramms einhalten.

## Erstellung von UML-Diagrammen

- ❑ Bei ernsthafter Softwareentwicklung werden spezielle UML-Editoren benutzt.
- ❑ Im Softwarepraktikum wird der Editor Astah (astah.net) eingesetzt.



# Folien zur Vorlesung **Softwaretechnik**

## **Teil 2: UML – Unified Modeling Language** **Abschnitt 2.3: Sequenzdiagramme**

## Spezifikation von Klassen

- ❑ Eine Klasse vereinigt Attribute und Methoden.
- ❑ Die Wertebelegung der Attribute bildet den (formalen) Zustand eines Objekts.
- ❑ Die Aufrufe von Methoden ändern die Werte der Attribute und damit den Zustand.

## Spezifikation durch Klassendiagramm

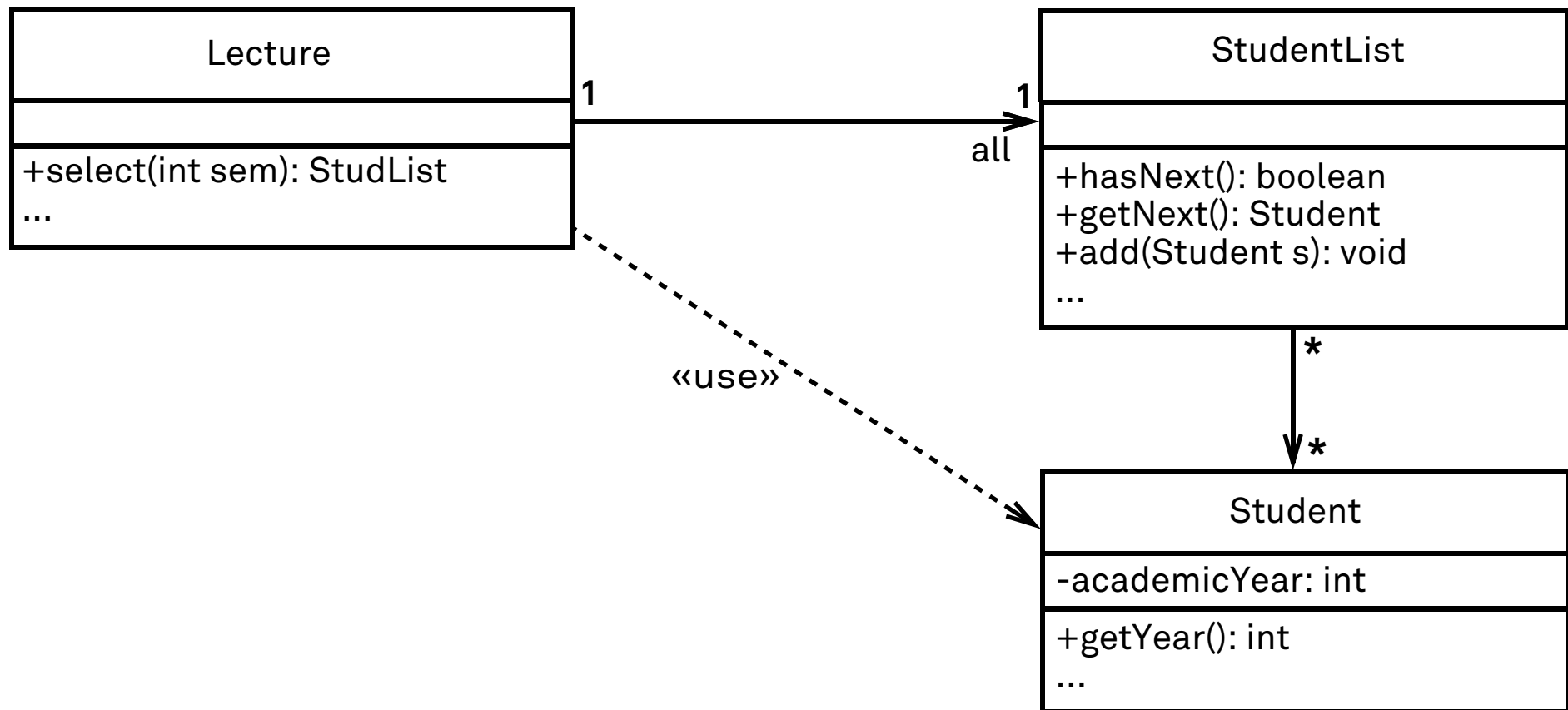
- ❑ mehrere Objekte (verschiedener) Klassen arbeiten zusammen durch Aufrufe ihrer Methoden.
- ❑ **Beispiel (*nicht aus SWT-Starfighter*):**  
Erstellen einer Liste der SWT-Teilnehmer, die im zweiten Semester sind:
  - Studierenden-Objekt aus der Liste aller Teilnehmer auswählen
  - Studierenden-Objekt überprüfen
  - Einfügen des Objekts in die gewünschte Liste, falls Semesterzahl den Wert 2 hat.

**jetzt: Sequenzdiagramm,**  
mit dem sich die Abfolge und Abhängigkeiten zwischen den Aufrufen verschiedener Objekte veranschaulichen lassen

Literatur: Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 79-93  
[http://link.springer.com/chapter/10.1007/3-540-30950-0\\_5](http://link.springer.com/chapter/10.1007/3-540-30950-0_5)

## Beispiel

### Klassendiagramm



## Beispiel

(Fortsetzung)

Methode der Klasse Lecture:

- ❑ `select(int sem): StudentList`  
erstellt die gewünschte Liste der Studierenden des 2. Semesters

Methoden der Klasse StudentList:

- ❑ `hasNext(): boolean`  
zeigt an, ob die Liste noch unbearbeitete Student-Objekte enthält
- ❑ `getNext(): Student`  
liefert das nächste Student-Objekt, falls der Aufruf von `hasNext` den Wert `true` liefert
- ❑ `add(Student s): void`  
fügt ein Student-Objekt in die Liste ein

Methode der Klasse Student:

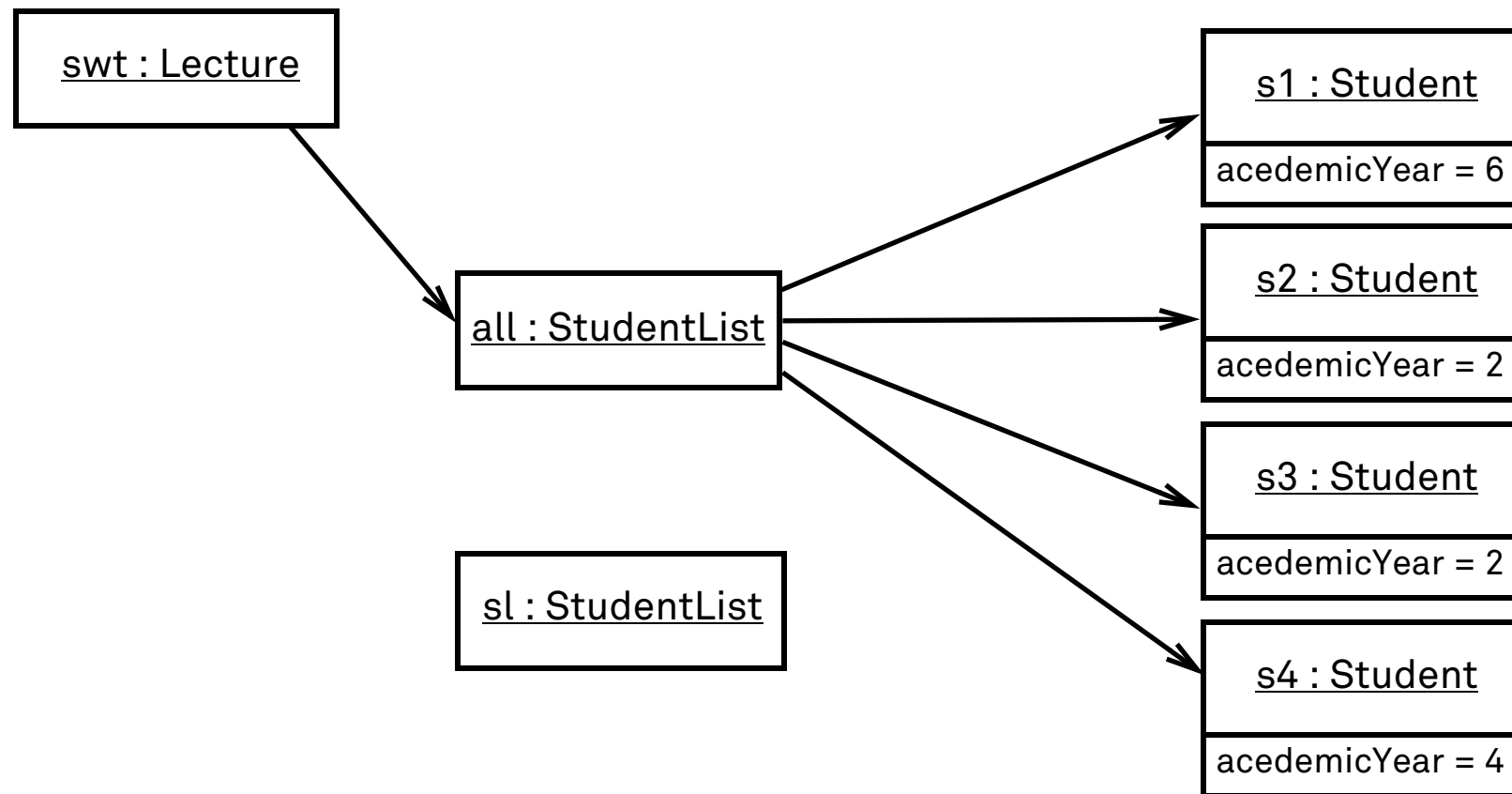
- ❑ `getYear(): int`  
liefert die Semesterzahl aus dem Attribut des Student-Objekts

**==> Der Ablauf muss eine bestimmte Abfolge von Methodenaufrufen einhalten.**

## Beispiel

(Fortsetzung)

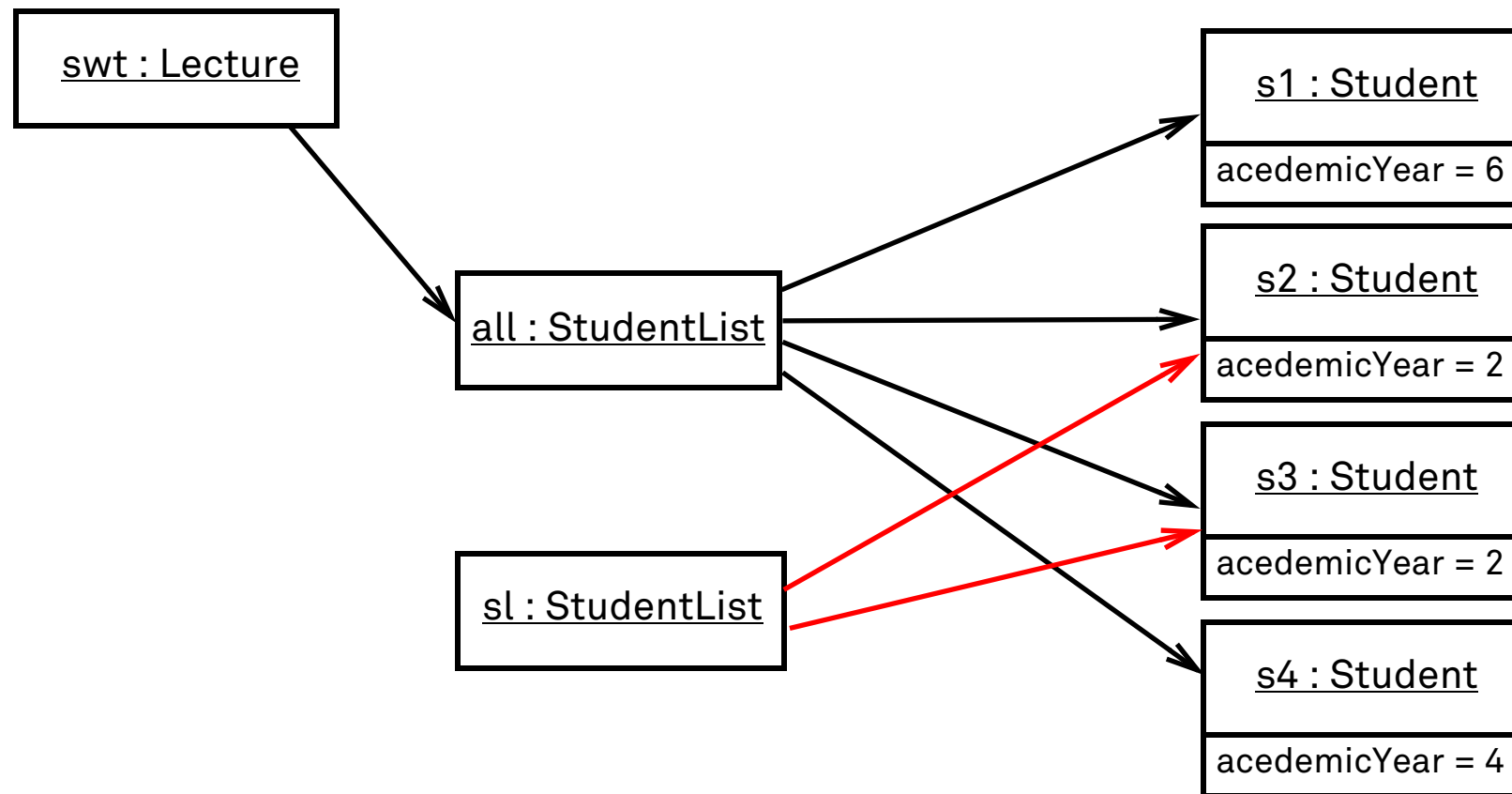
Objektdiagramm (vor dem Aufruf der Methode `select`)



## Beispiel

(Fortsetzung)

Objektdiagramm (**nach** dem Aufruf der Methode `select`)





## Überblick

Hilfsmittel zur Beschreibung der Reihenfolge von Methodenaufrufen:

### Sequenzdiagramm

- ❑ Es zeigt den Ablauf der Kommunikation zwischen Objekten.
- ❑ Es zeigt die zeitliche Folge der Kommunikationsschritte.
- ❑ Es zeigt ein endliches Beispielszenario,  
also meist nur einen zeitlich begrenzten Ausschnitt aus der Kommunikation.
- ❑ Es zeigt nur den Aufruf von Methoden an.

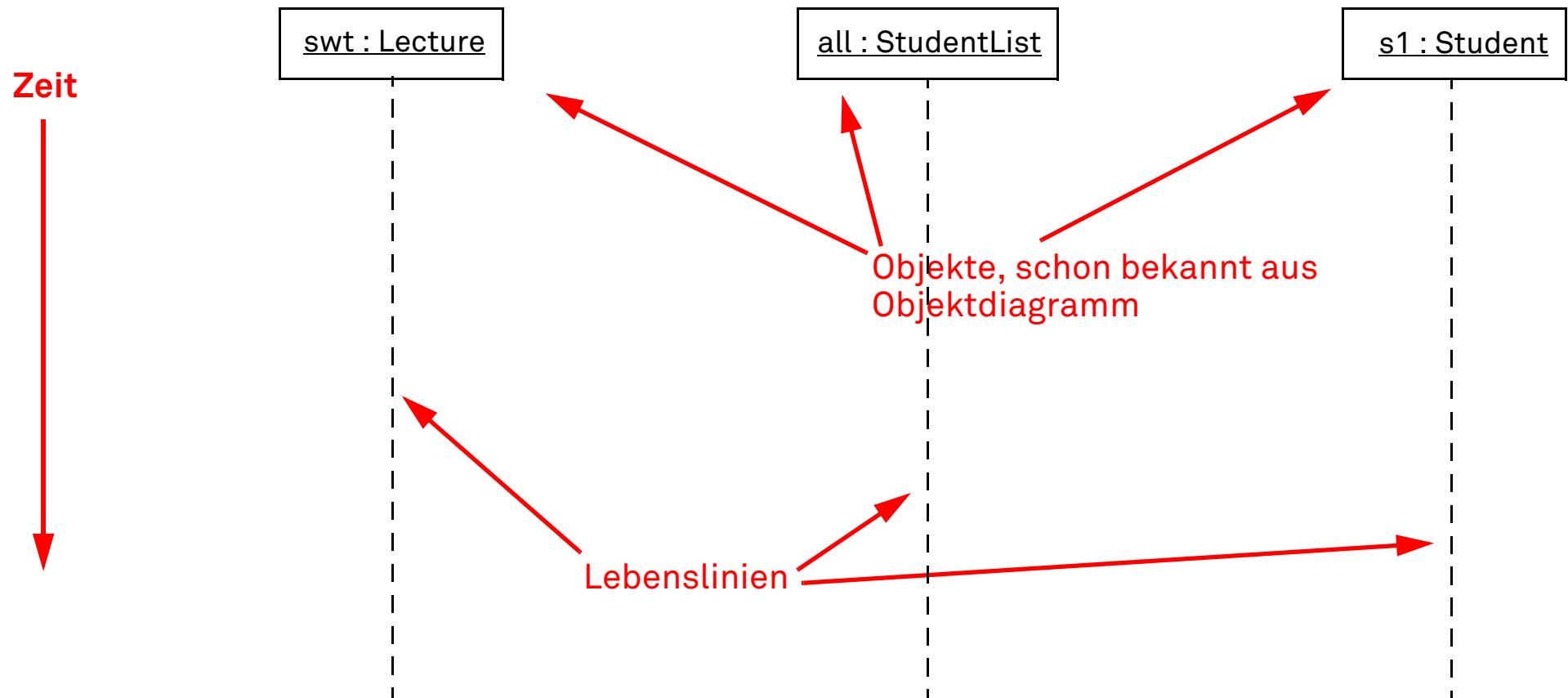
Man spricht dabei auch allgemein von Nachrichtenaustausch:

- Der *Aufruf einer Methode*  $m$  für ein Objekt  $o$  wird als  
das *Senden einer Nachricht* an  $o$  aufgefasst, da die Methode bestimmt,  
was  $o$  tun soll.
- Das *Beenden* von  $m$  (und eventuell das Zurückgeben eines Wertes)  
werden als *Antwort* von  $o$  auf die Nachricht verstanden.

## Aufbau eines Sequenzdiagramms

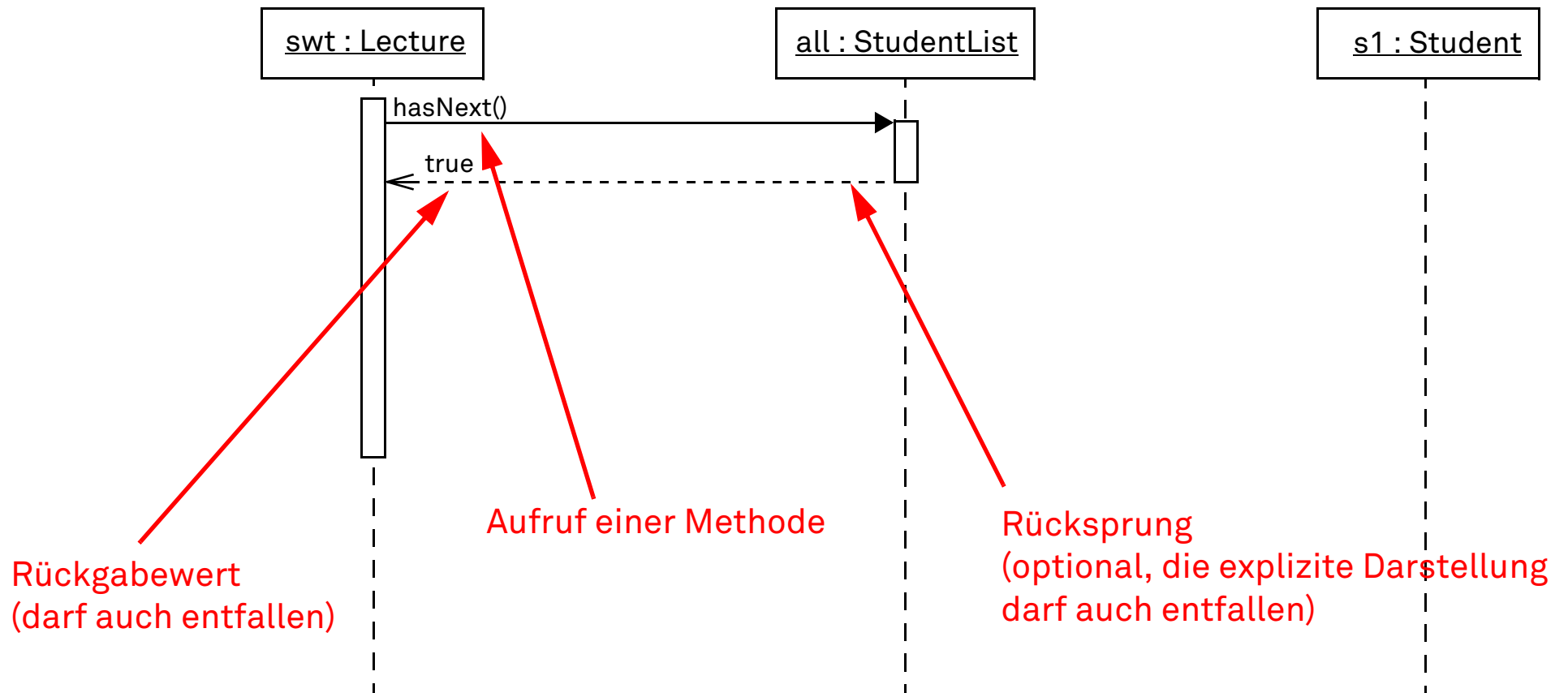
- ❑ Im Sequenzdiagramm werden dargestellt:
  - die handelnden **Objekte**,
  - deren Existenz als **senkrechte** Lebenslinie,
  - Methodenaufrufe als **waagerechte** Verbindungen zwischen Lebenslinien,
  - Phasen der Kontrolle als Blöcke auf Lebenslinien,
  - die Zeit durch die implizit nach unten verlaufende Zeitachse.
  
- ❑ Sichtbar wird dadurch
  - welche Objekte an Methodenaufrufen beteiligt sind,
  - von welcher Methode ein Aufruf ausgeht und welche Methode aufgerufen wird,
  - welche Methode den Ablauf über mehrere Aufrufe hinweg kontrolliert,
  - wie der Kommunikationsablauf insgesamt aufgebaut ist.

## Beispiel – Sequenzdiagramm



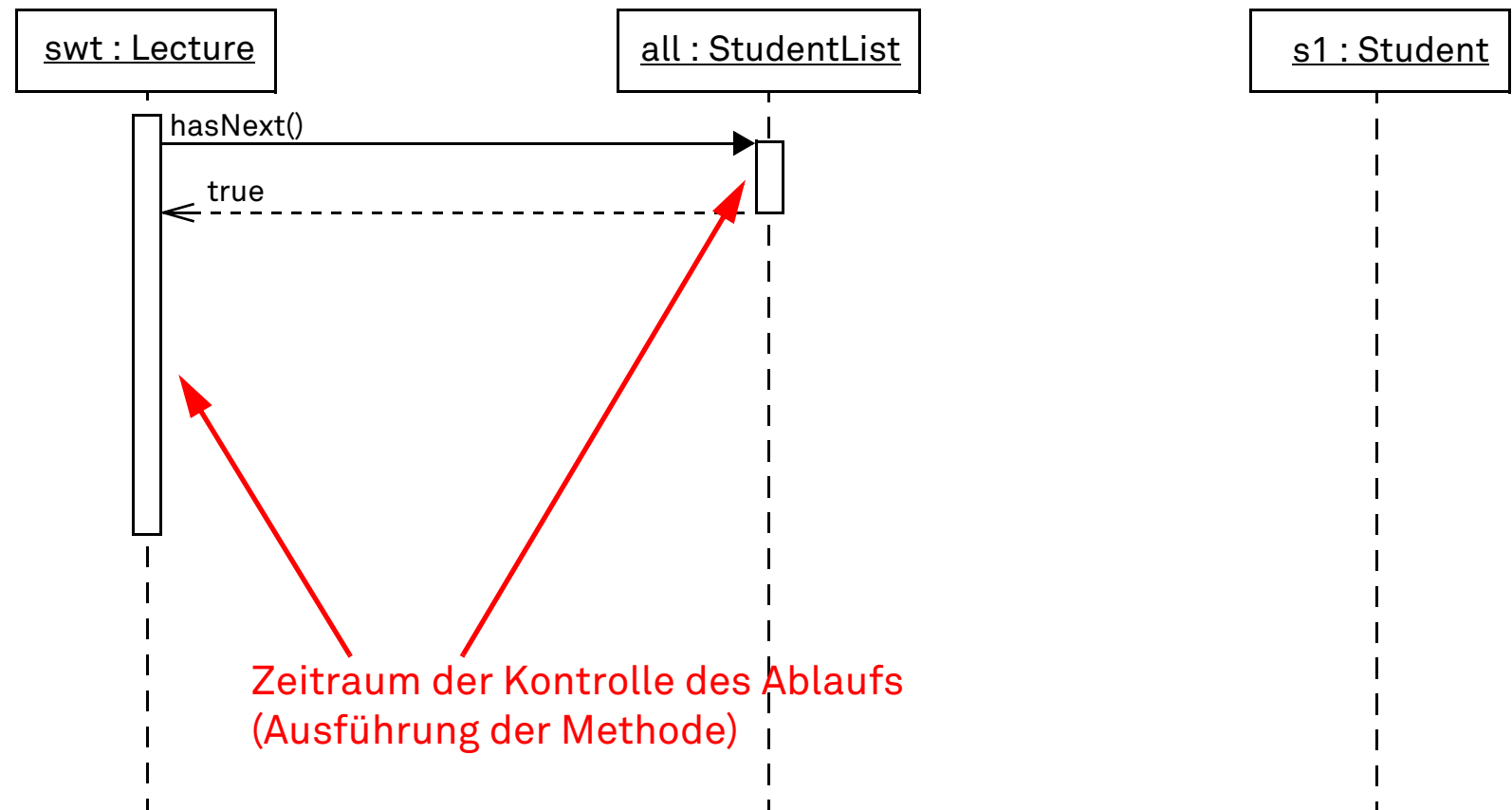
## Beispiel – Sequenzdiagramm

(Fortsetzung)



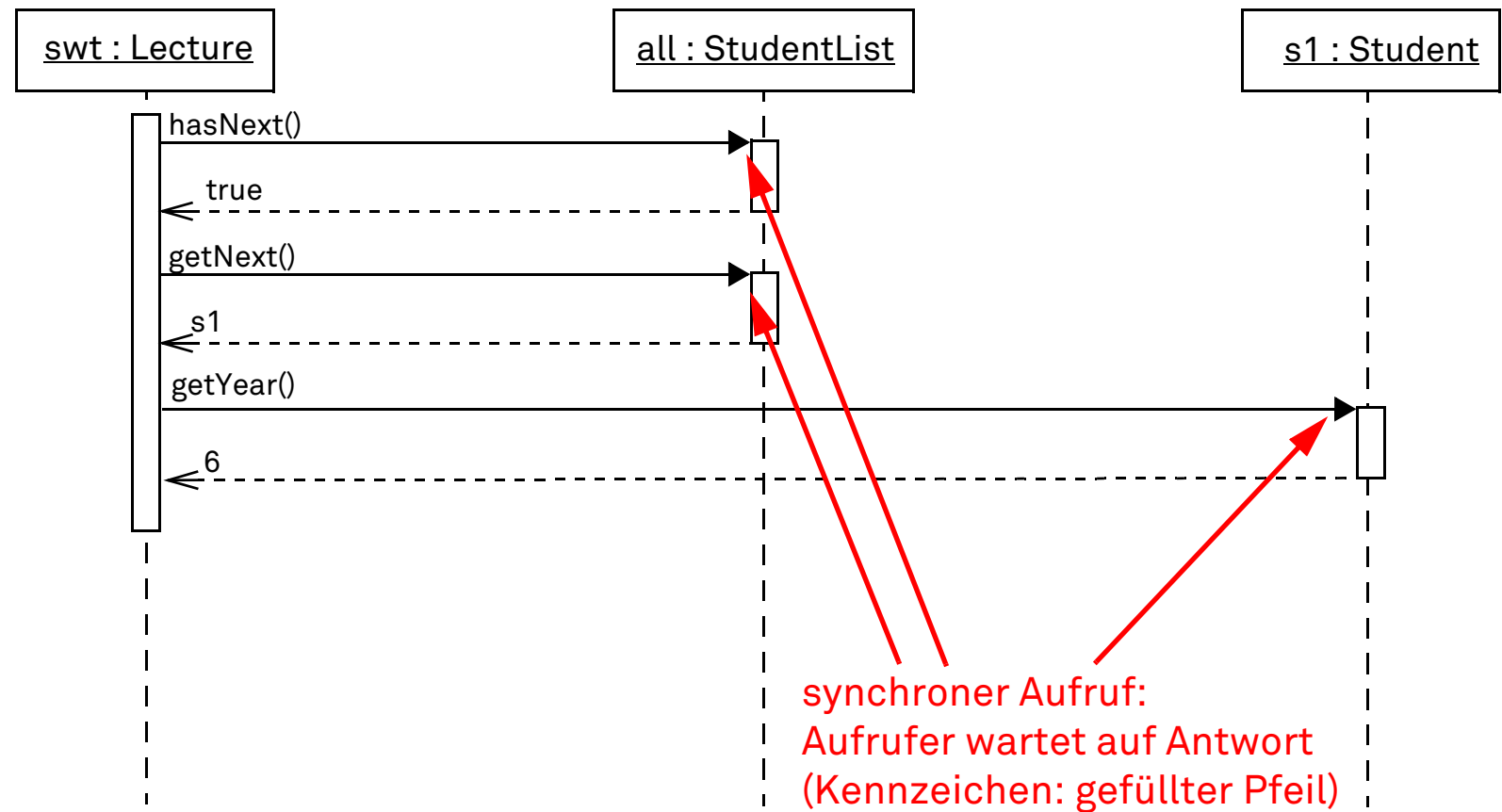
## Beispiel – Sequenzdiagramm

(Fortsetzung)



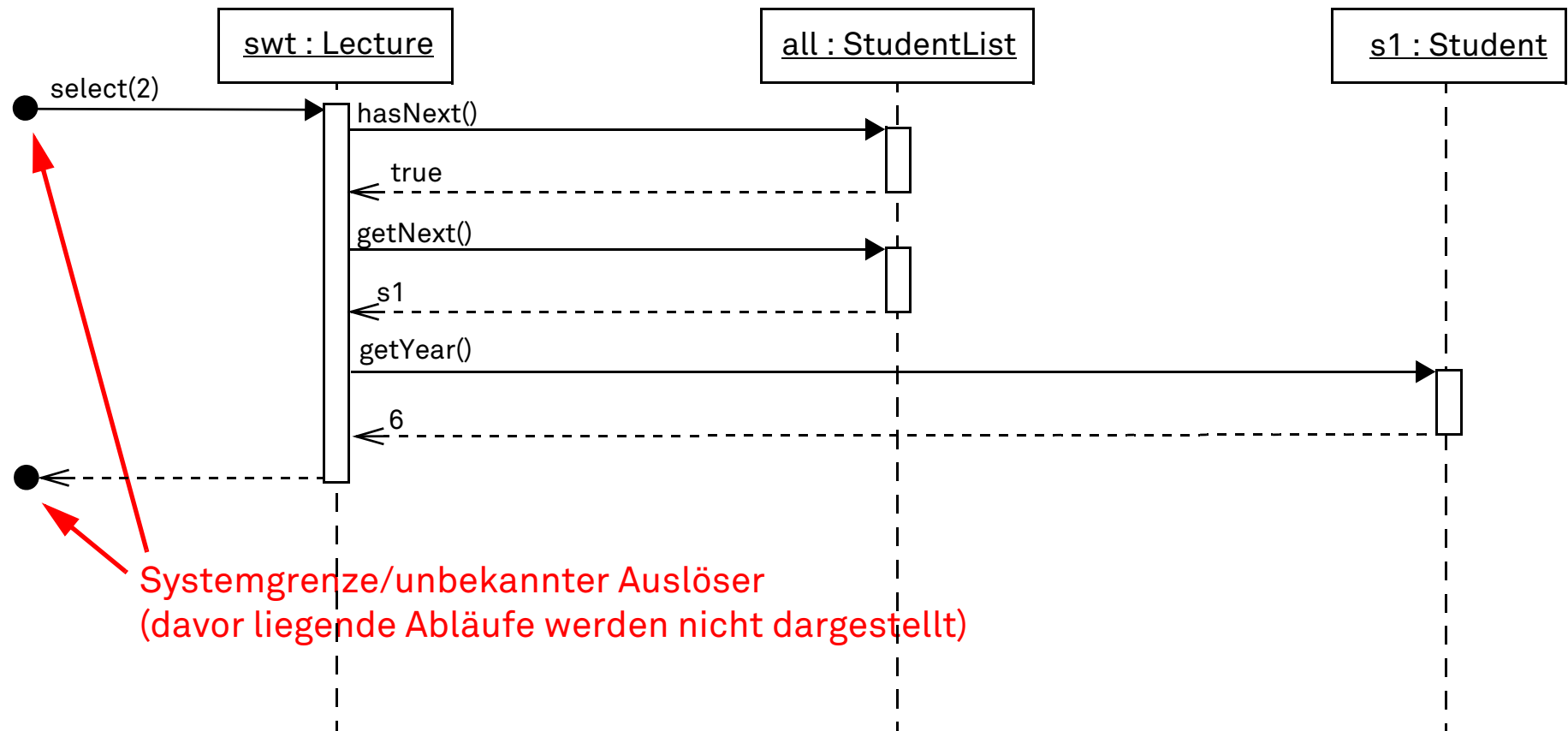
## Beispiel – Sequenzdiagramm

(Fortsetzung)



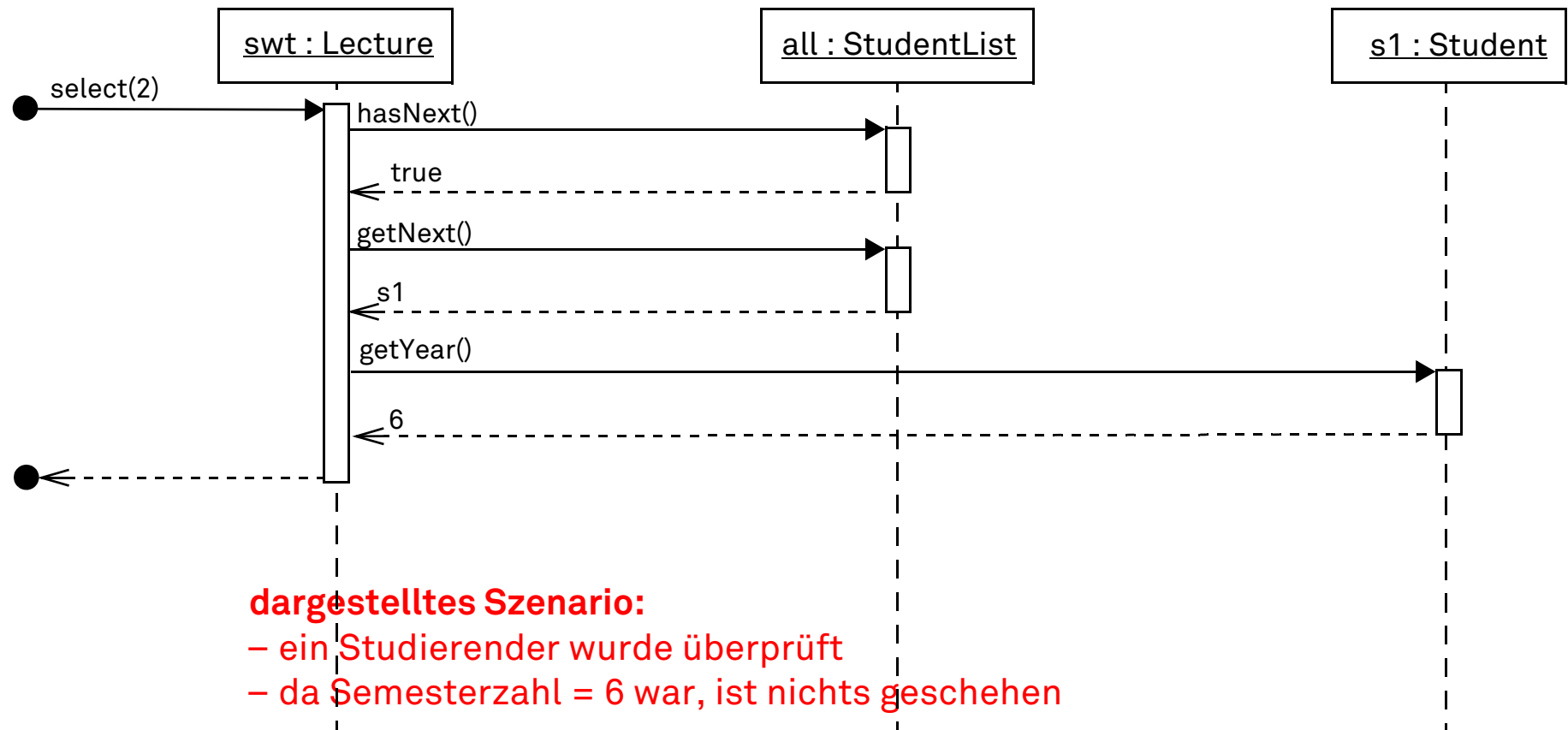
## Beispiel – Sequenzdiagramm

(Fortsetzung)



## Beispiel – Sequenzdiagramm

(Fortsetzung)

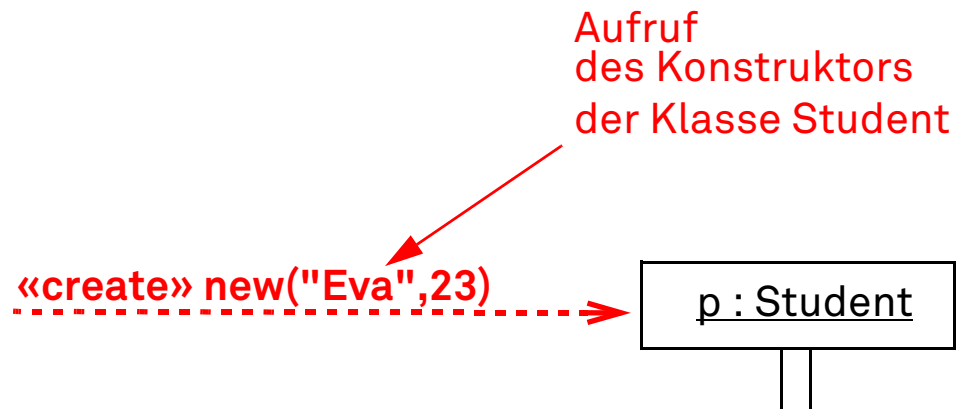




## Sequenzdiagramm – Syntax

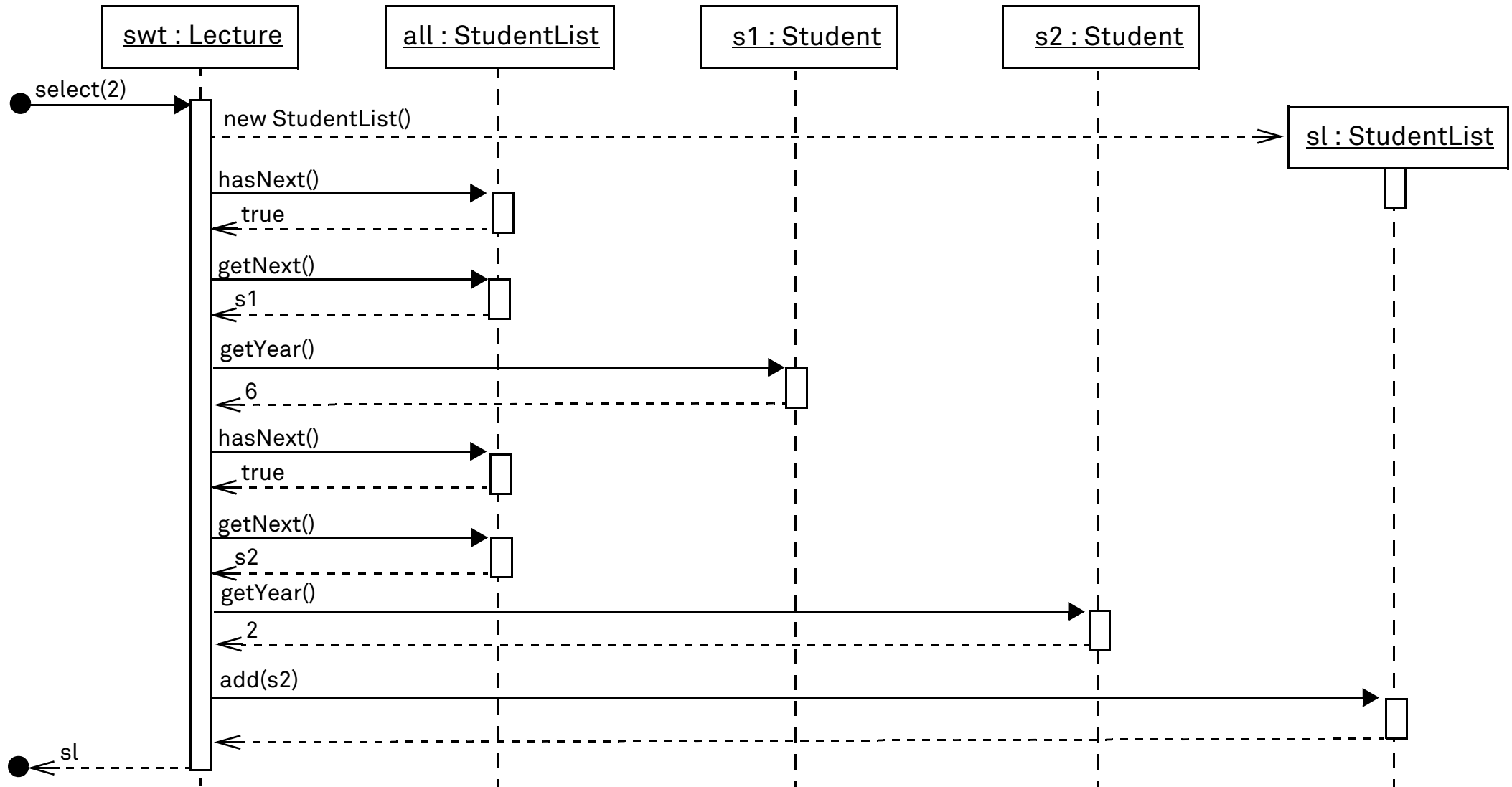
(Fortsetzung)

### Erzeugen von Objekten



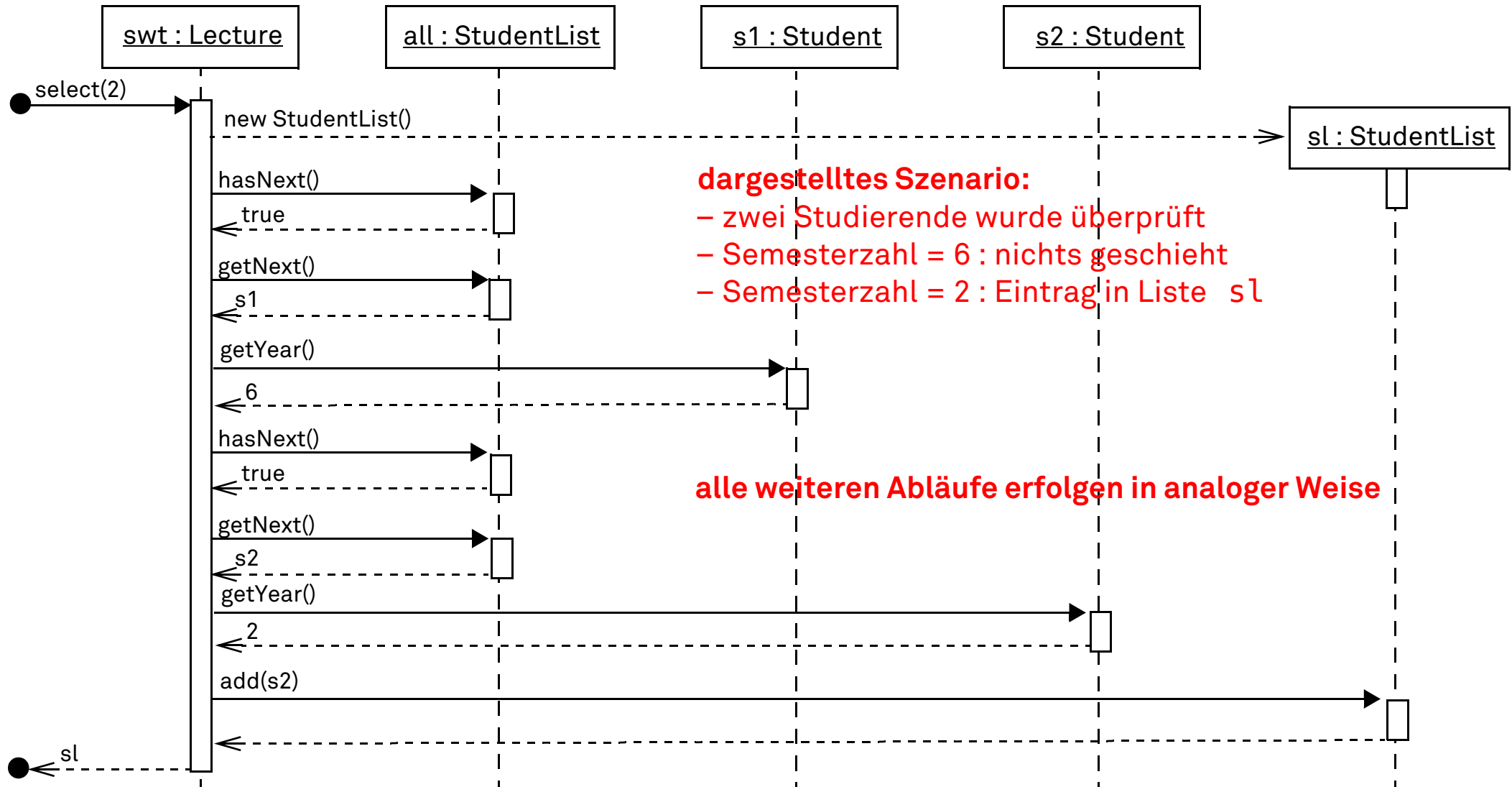
## Beispiel – Sequenzdiagramm

(Fortsetzung)



## Beispiel – Sequenzdiagramm

(Fortsetzung)



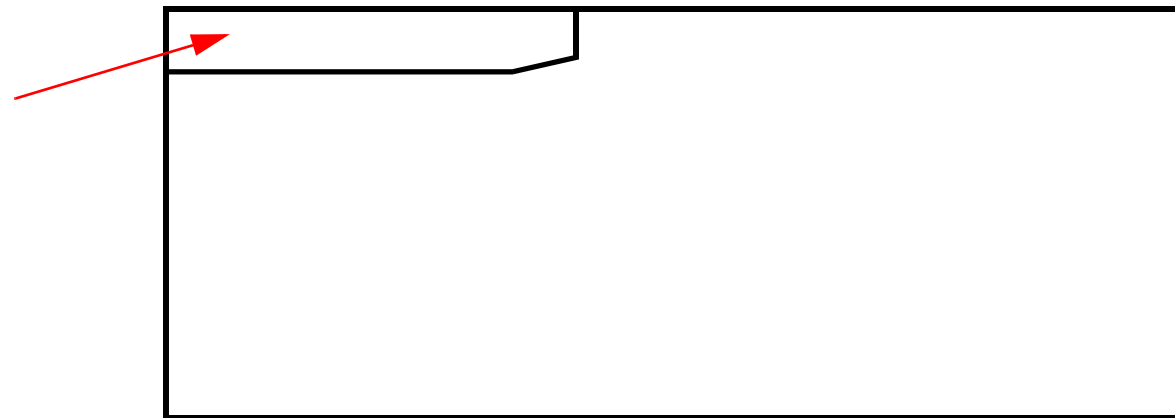
## Interaktionsfragment

Das Sequenzdiagramm für das Beispiel  
zeigt Ablauf mit zwei Studierenden

Möglich sind auch verallgemeinerte Darstellungen mit

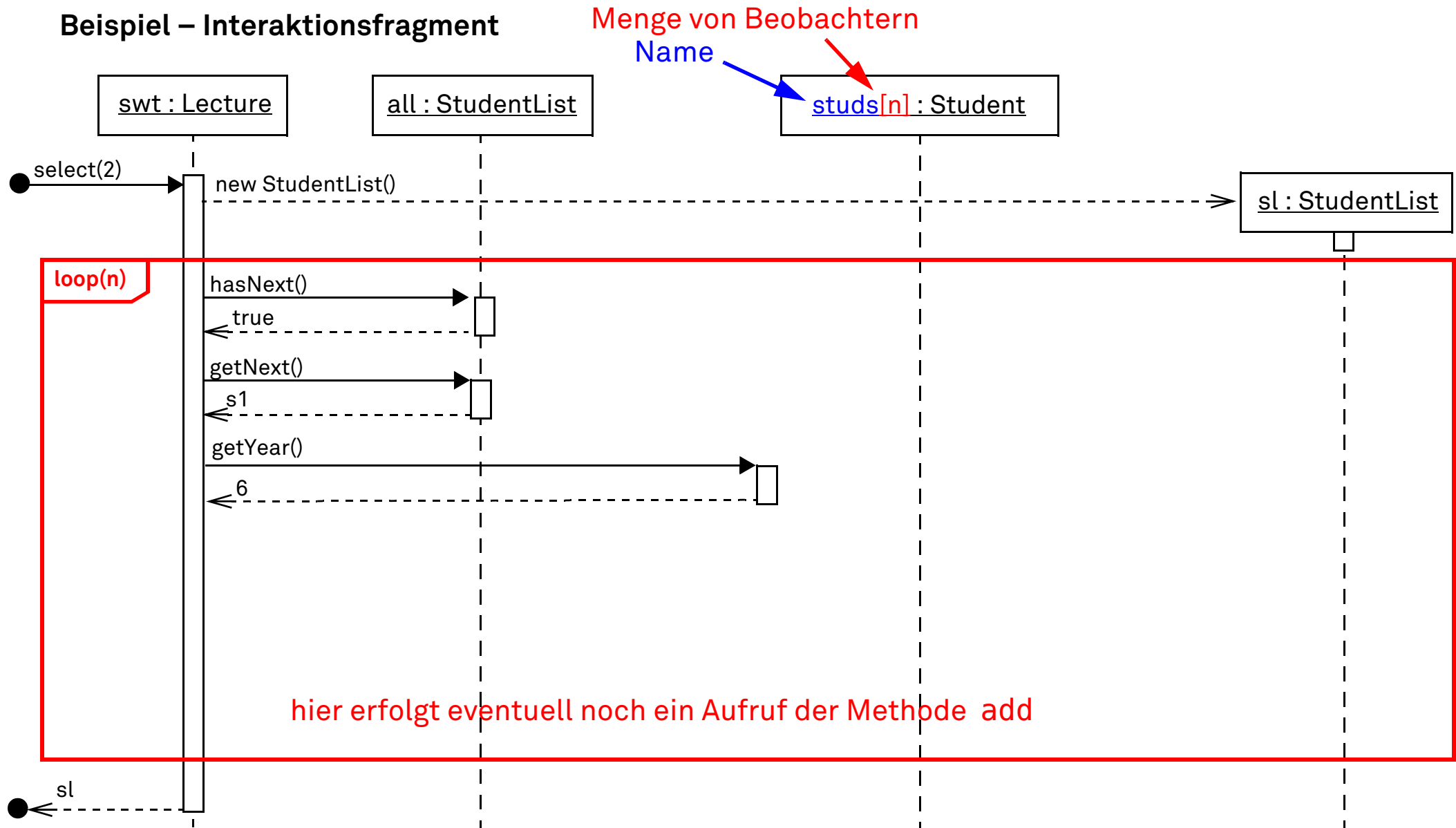
- ❑ Mengen von Objekten und
- ❑ Interaktionsfragmenten: Ablaufsequenzen, die abhängig von Bedingungen (wiederholt) ausgeführt werden können.

Eintrag für einen  
Interaktionsoperator

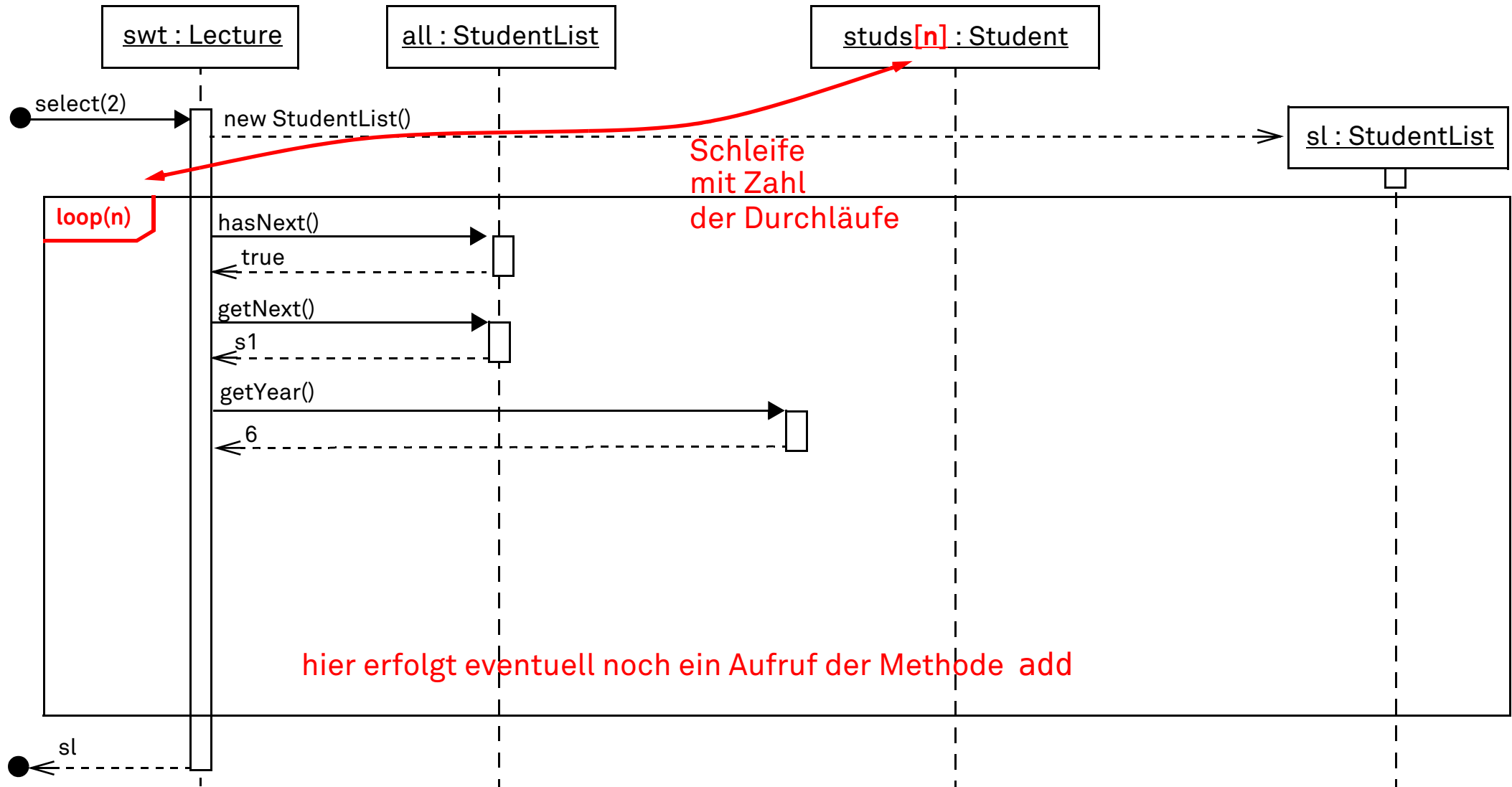


Interaktions-Fragment

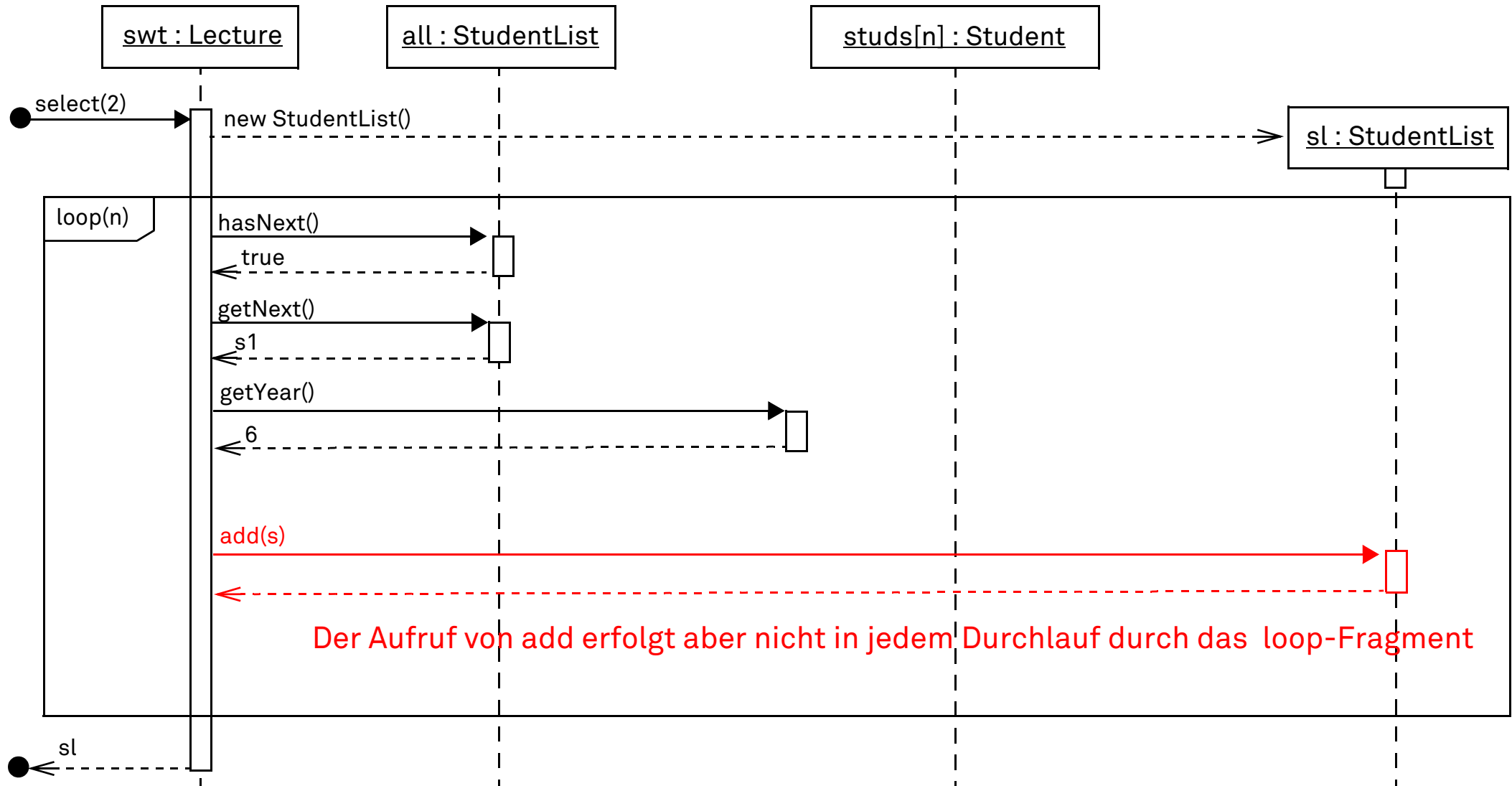
## Beispiel – Interaktionsfragment



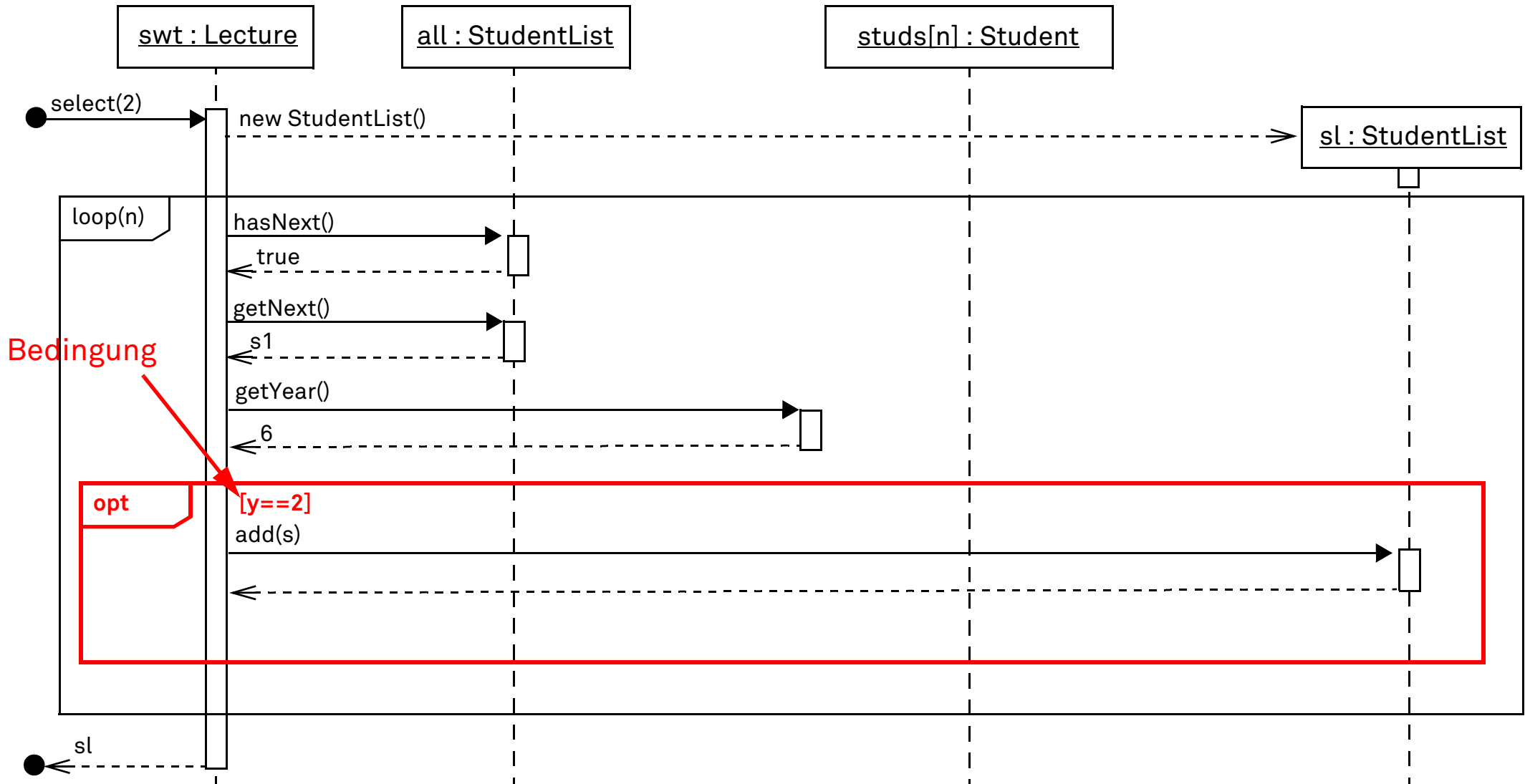
## Beispiel – Interaktionsfragment



## Beispiel – Interaktionsfragment

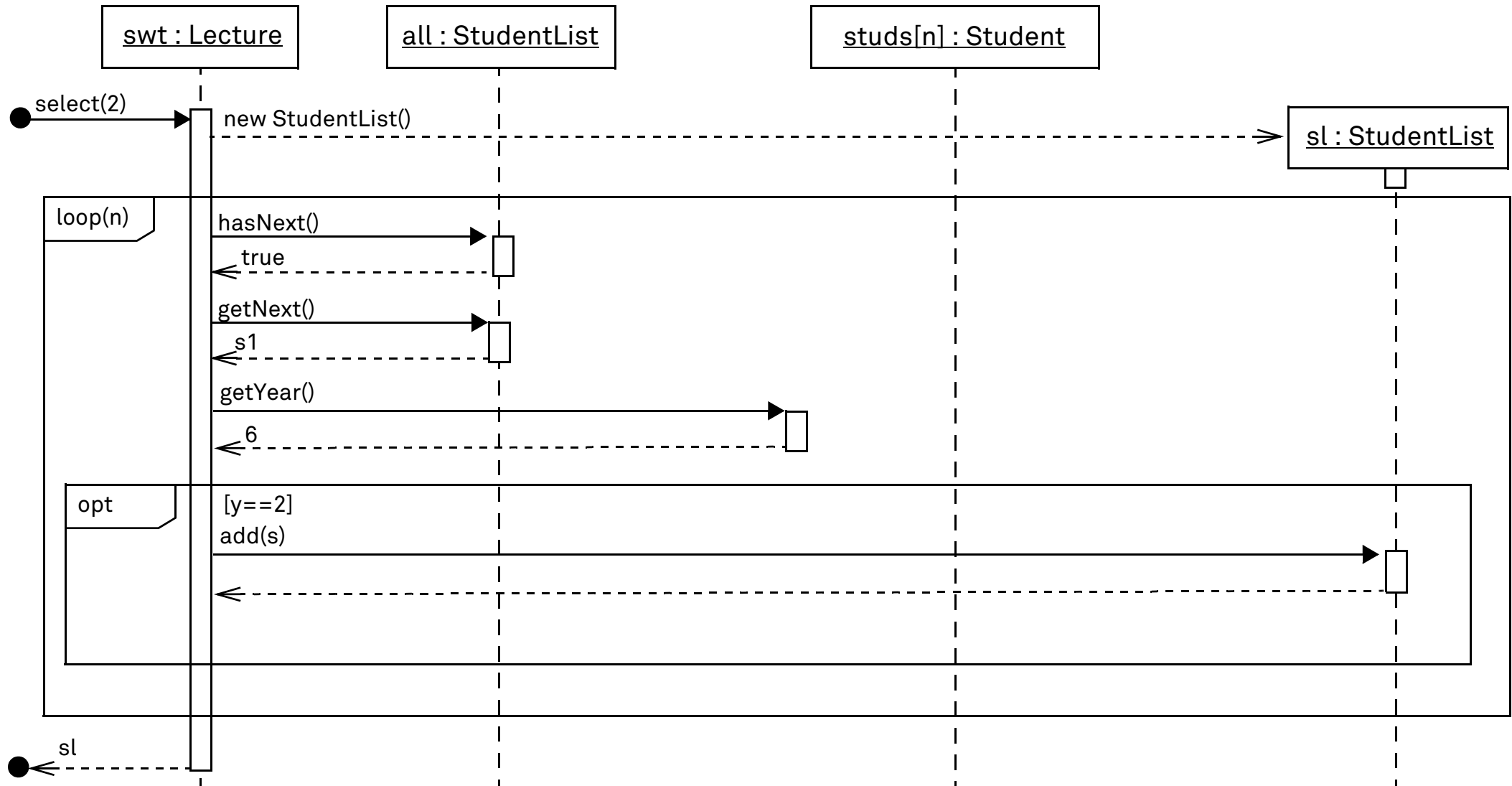


## Beispiel – Interaktionsfragment





## Beispiel – Interaktionsfragment

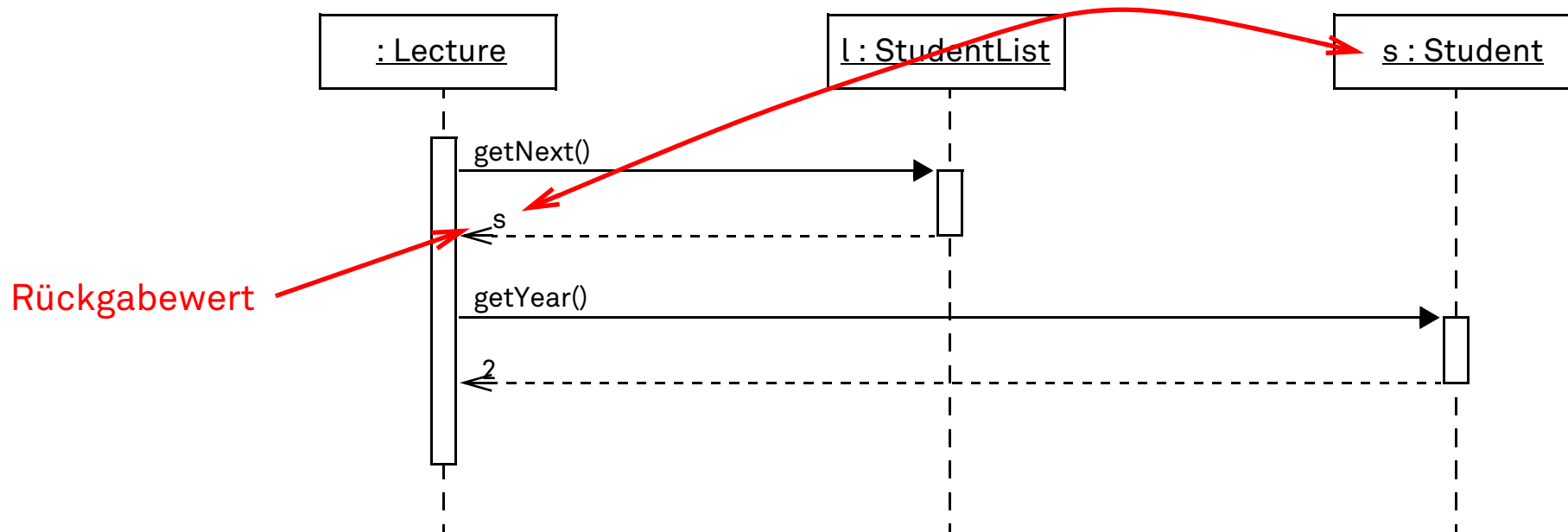


## Sequenzdiagramm – Syntax

(Fortsetzung)

Aufruffolgen – von links nach rechts

`l.getNext().getYear();`

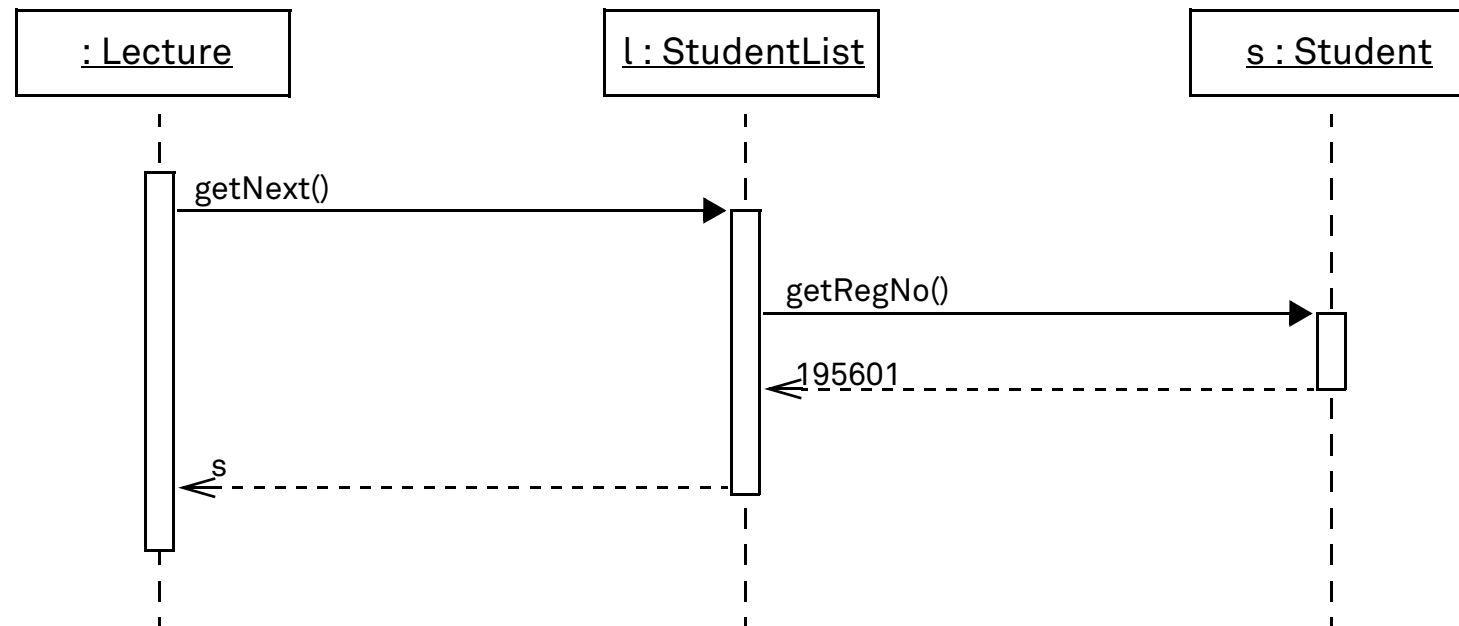


## Sequenzdiagramm – Syntax

(Fortsetzung)

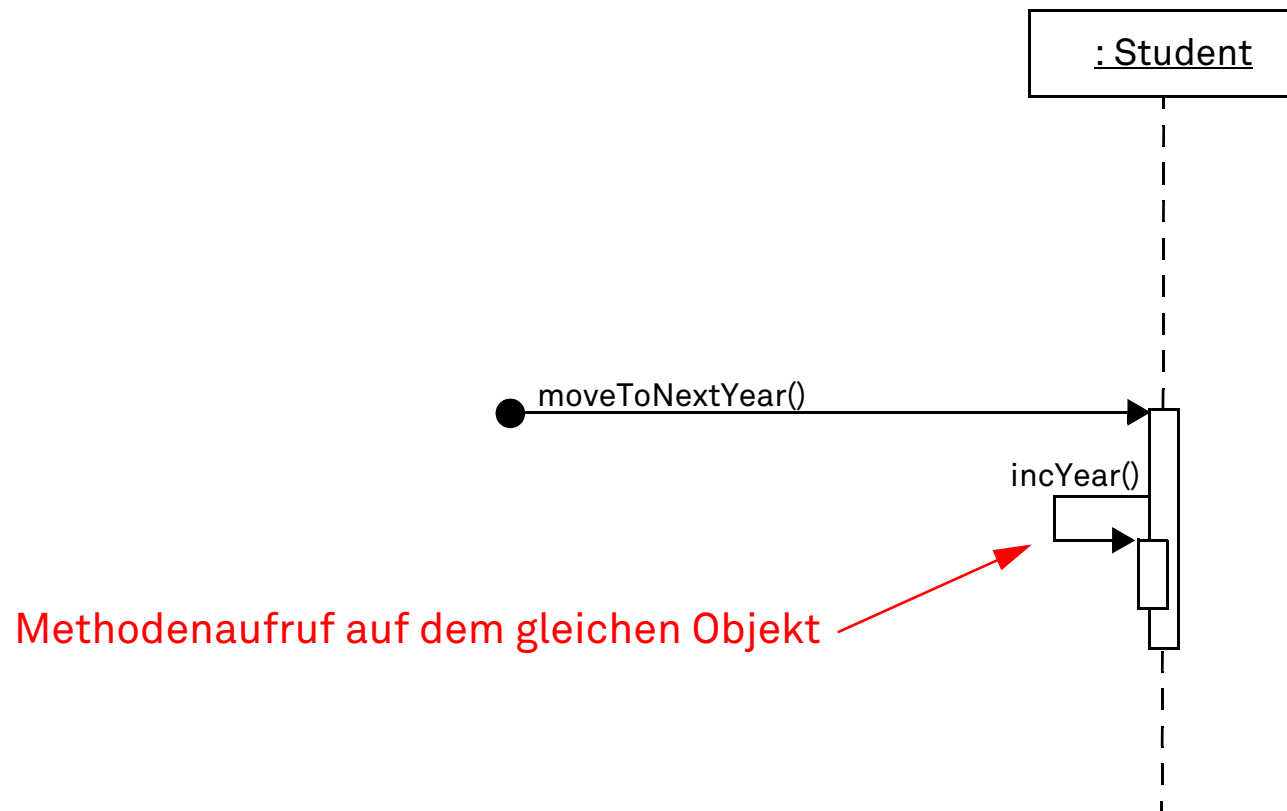
### Aufrufschachtelungen

`l.getNext();`  
und `s.getRegNo()` wird **in** `getNext()` aufgerufen!




## Beispiel – Sequenzdiagramm

(Fortsetzung)

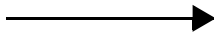



## synchrone/asynchrone Nachrichten

<b>bisher: synchrone Aufrufe</b>

(1) die aufrufende Methode gibt temporär Kontrolle ab
(2) die aufrufende Methode wartet auf die Rückkehr der Kontrolle
(3) die aufrufende Methode übernimmt die Kontrolle nach Rücksprung
<i>Konsequenz 1:</i> Aktivität der aufrufenden Methode ist länger als die der aufgerufenen Methode
<i>Konsequenz 2:</i> der Rücksprungpfeil kann entfallen, da Ablauf klar definiert ist

## synchrone/asynchrone Nachrichten

(Fortsetzung)

bisher: synchroner Aufruf	asynchroner Aufruf
	
(1) die aufrufende Methode gibt temporär Kontrolle ab	(1) die aufrufende Methode kann nach dem Aufruf mit seiner Bearbeitung fortfahren
(2) die aufrufende Methode wartet auf die Rückkehr der Kontrolle	(2) die aufrufende Methode <b>wartet nicht</b> auf eine Antwort
(3) die aufrufende Methode übernimmt die Kontrolle nach Rücksprung	(3) aufrufende und aufgerufene Methode agieren <b>nebenläufig</b>
<i>Konsequenz 1:</i> Aktivität der aufrufenden Methode dauert länger als die der aufgerufenen Methode	<i>Konsequenz 1:</i> auf Aktivitätsbalken kann auch verzichtet werden
<i>Konsequenz 2:</i> der Rücksprungpfeil kann entfallen, da Ablauf klar definiert ist	<i>Konsequenz 2:</i> es gibt <b>keine</b> Rücksprungpfeile, aber eventuell Aufrufe in umgekehrter Richtung

## synchrone/asynchrone Nachrichten

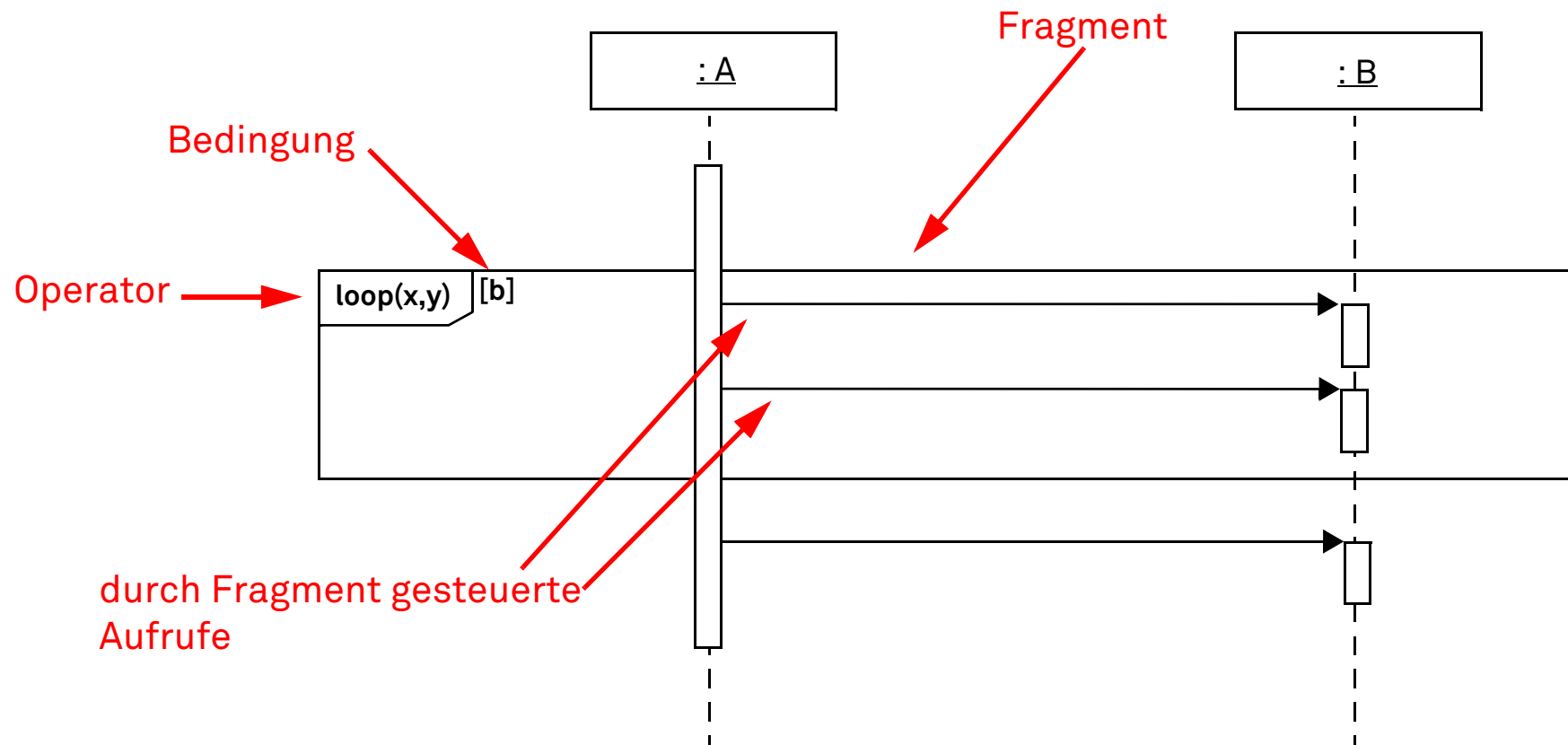
(Fortsetzung)

bisher: synchroner Aufruf	asynchroner Aufruf
(1) die aufrufende Methode gibt temporär Kontrolle ab	(1) die aufrufende Methode kann nach dem Aufruf mit seiner Bearbeitung fortfahren
(2) die aufrufende Methode wartet auf die Rückkehr der Kontrolle	(2) die aufrufende Methode wartet nicht auf eine Antwort
(3) die aufrufende Methode übernimmt die Kontrolle nach Rücksprung	(3) aufrufende und aufgerufene Methode agieren nebenläufig
<i>Konsequenz 1:</i> Aktivität der aufrufenden Methode dauert länger als die der aufgerufenen Methode	<i>Konsequenz 1:</i> auf Aktivitätsbalken kann auch verzichtet werden
<i>Konsequenz 2:</i> der Rücksprungpfeil kann entfallen, da Ablauf klar definiert ist	<i>Konsequenz 2:</i> es gibt keine Rücksprungpfeile, aber eventuell Aufrufe in umgekehrter Richtung

Asynchrone Aufrufe werden hier nicht weiter betrachtet,  
da Methodenaufrufe in Java synchron arbeiten.

## Interaktionsfragemente

### Ergänzungen zum loop-Operator

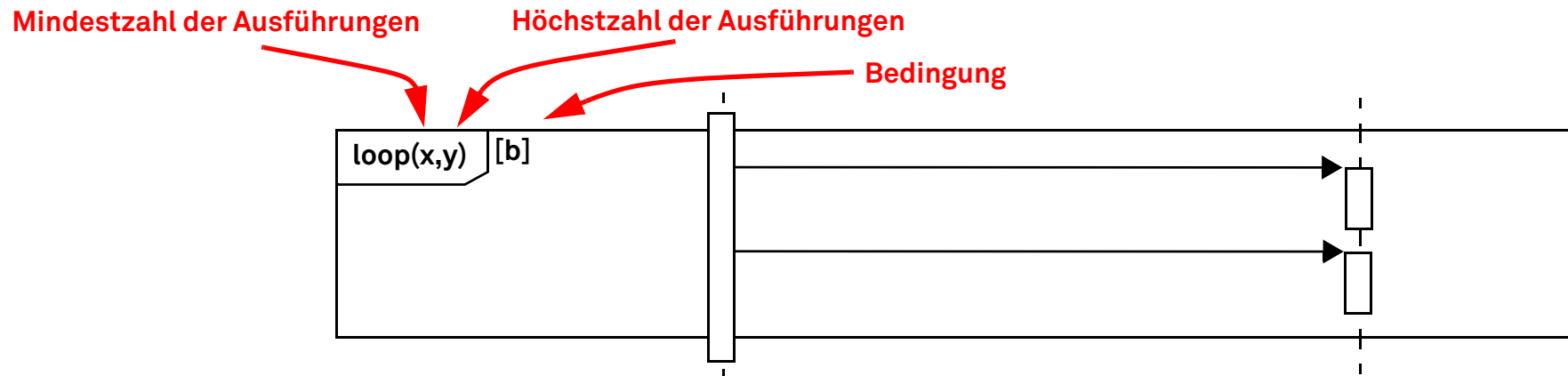




## Interaktionsfragemente

(Fortsetzung)

### Ergänzungen zum loop-Operator



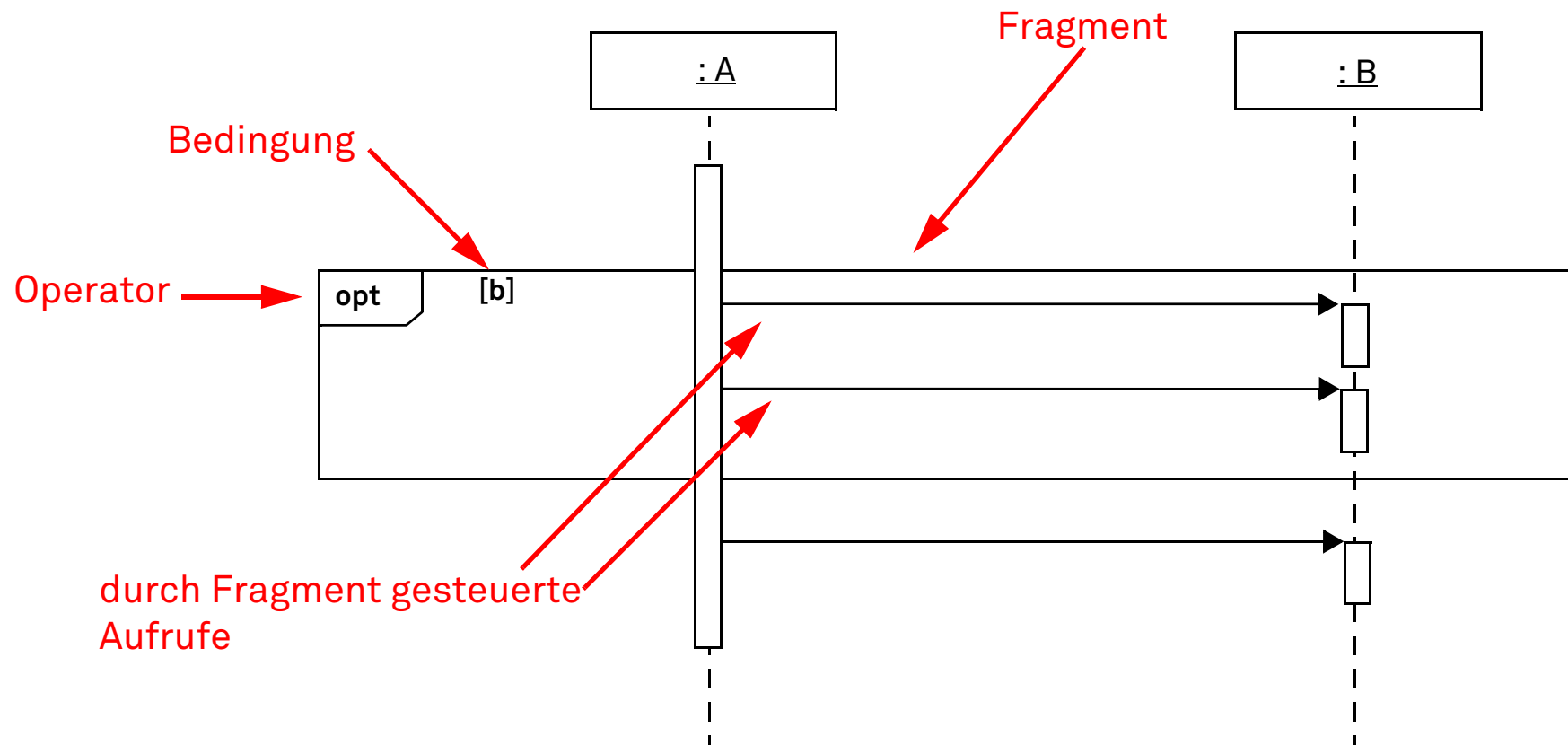
- ❑ `x` muss kleiner gleich `y` sein.
- ❑ Das **loop**-Fragment wird auf jeden Fall `x`-mal ausgeführt (– unabhängig von `b`).
- ❑ Das **loop**-Fragment wird höchstens `y`-mal ausgeführt – aber nur solange `b` gilt.
- ❑ **loop(z) entspricht loop(z,z)**, das **loop**-Fragment wird **genau z**-mal ausgeführt.
- ❑ **loop entspricht loop(0,\*)**, das loop-Fragment wird nicht oder beliebig oft ausgeführt .
- ❑ Die Bedingung `b` kann entfallen,  
keine Bedingung ist gleichwertig mit der Bedingung **[true]**.

## Interaktionsfragemente

(Fortsetzung)

### opt-Operator

Die Aufrufe im Fragment werden nur ausgeführt, wenn die Bedingung **b** wahr ist.  
(**opt**ionaler Ablauf)



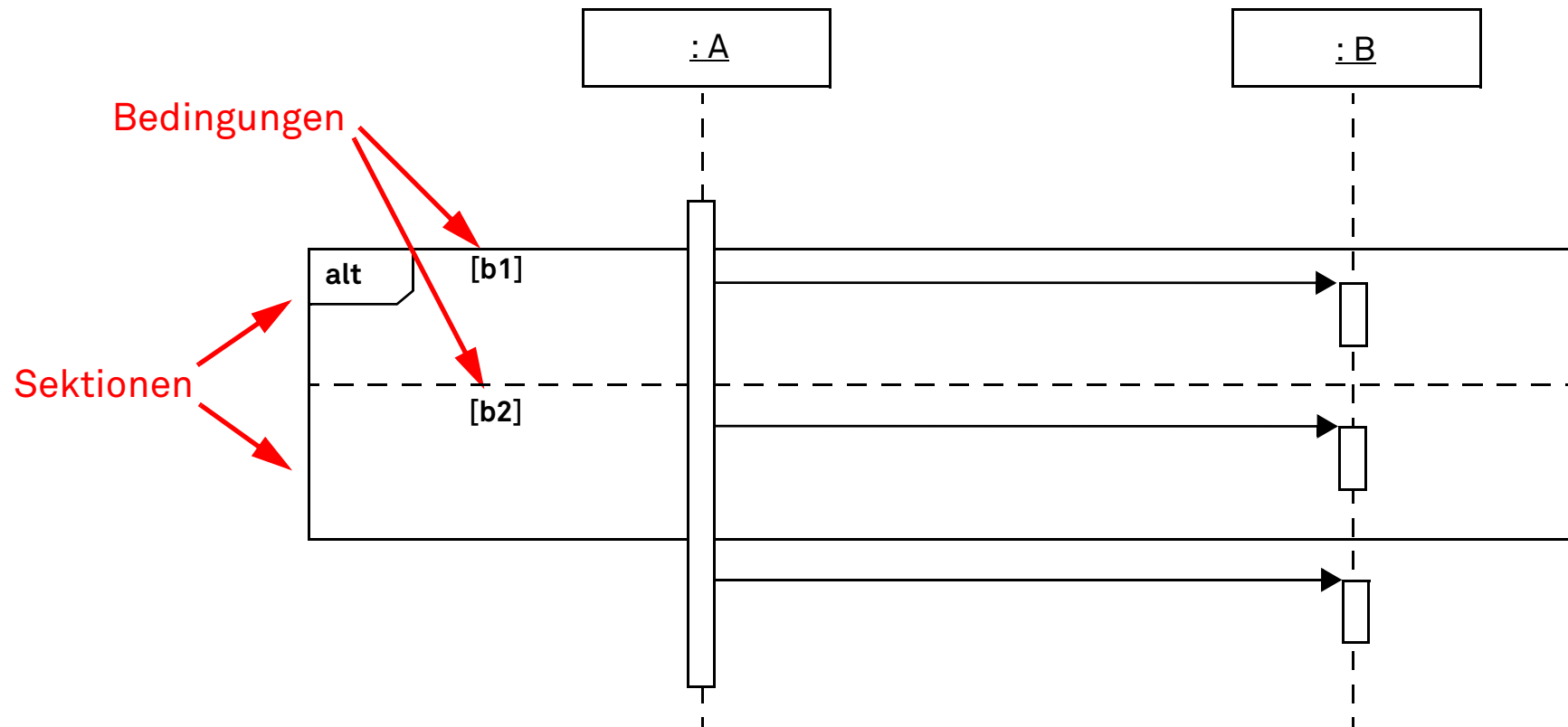
## Interaktionsfragemente

(Fortsetzung)

### alt-Operator

Die Aufrufe einer Sektion werden nur ausgeführt, wenn die Bedingung der Sektion wahr ist. Entweder b1 oder b2 muss true sein.

(alternative Abläufe)



## Zusammenfassung

- ❑ Sequenzdiagramme zeigen
  - die handelnden **Objekte**,
  - die zwischen ihnen stattfindenden Methodenaufrufe,
  - Phasen der Kontrolle als Aktivitätsbalken auf Lebenslinien, entlang der Zeitachse.
- ❑ Sequenzdiagramme zeigen in der Regel nur beispielhafte Szenarios.
- ❑ Der visuell erfassbare Umfang von Sequenzdiagrammen ist schnell erreicht.
- ❑ Interaktionsfragmente können die Aussagen verallgemeinern.  
schaffen allerdings durch fehlende Ausdrucksmöglichkeiten für Details nicht unbedingt mehr Klarheit.
- ❑ Sequenzdiagramme sind gut geeignet, um  
die Initiative von Objekten/Methoden in eng begrenzten Abläufen zu verdeutlichen.

# Folien zur Vorlesung **Softwaretechnik**

## **Teil 3: Entwurfsmuster** **Abschnitt 2.3: Überblick**

## Entwurfsmuster

bilden den Einstieg in die Techniken zum Entwerfen von Softwaresystemen

Entwurfsmuster in der Softwareentwicklung  
wurden erstmals vorgestellt in:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:  
Design Patterns – Elements of Reusable Object-Oriented Software,  
Addison-Wesley, 1995, ISBN 0201633612

*auch auf deutsch:* Entwurfsmuster, Addison-Wesley, 2004, ISBN 3827321999

Anmerkung:

Das Buch ist aufgrund der vielen Vorwärtsreferenzen etwas schwer verständlich.

- Literatur:   Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.209-253  
              [http://link.springer.com/chapter/10.1007/978-3-8348-9174-7\\_8](http://link.springer.com/chapter/10.1007/978-3-8348-9174-7_8)
- Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 203-231  
              [http://link.springer.com/chapter/10.1007/3-540-30950-0\\_12](http://link.springer.com/chapter/10.1007/3-540-30950-0_12)
- Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung  
              <http://link.springer.com/book/10.1007/978-3-8274-2526-3>

## Idee der Entwurfsmuster

- ❑ Bei der Gestaltung von Software treten immer wieder gleichartige Probleme auf.  
*Beispiel: Bei der Änderung eines Wertes müssen mehrere Fenster aktualisiert werden.  
(z.B. mehrere Instanzen des Filemanagers des Betriebssystems)*
- ❑ Für gleichartige Probleme gibt es immer auch gleichartige Lösungsansätze.
- ❑ Für eines der gleichartigen Probleme wird eine gute Lösung erarbeitet.
- ❑ Diese Lösung wird zu einem Lösungsansatz verallgemeinert und als sogenanntes **Entwurfsmuster** beschrieben.  
(für das o.a. Beispiel: Entwurfsmuster *Beobachter*)
- ❑ Während der Entwicklung einer konkreten Software wird das Entwurfsmuster dann in den Kontext der konkreten Aufgabenstellung übertragen und für diese passend umgesetzt.

## Vorteile beim Einsatz von Entwurfsmustern

- ❑ Die Entwicklungszeit wird verkürzt,  
da keine Lösung erfunden werden muss,  
sondern ein bekannter Lösungsansatz verwendet wird.
- ❑ Die Qualität der Software wird verbessert,  
da ein erprobter, geeigneter Lösungsansatz verwendet wird.
- ❑ Die Verständlichkeit der Software wird verbessert,  
da der Lösungsansatz vorab dokumentiert wurde und  
so vielen Entwicklern bekannt ist.
- ❑ Es entsteht eine Art *Normung* der erstellten Software,  
die auch die Diskussion unter den Entwicklern vereinfacht.



## Arten von Entwurfsmustern

Klassifizierung anhand der betrachteten Strukturierungseinheit:

- ❑ Klassen und ihre Beziehungen auf **Typ**-Ebene:

klassenbezogene Muster

- ❑ Objekte und ihre Beziehungen bei der **Ausführung**:

objektbezogene Muster

(In beiden Fällen wird das Muster selbst durch eine Struktur von Klassen beschrieben!)

Klassifizierung anhand des durch das Entwurfsmuster gelösten Problems:

- ❑ **strukturelle** Verbindung von Klassen oder Objekten:
- ❑ **Interaktion** zwischen Objekten:
- ❑ **Erzeugung** von Objekten:

Strukturmuster

Verhaltensmuster

Erzeugungsmuster

## Arten von Entwurfsmustern

(Fortsetzung)

	Strukturmuster	Verhaltensmuster	Erzeugungsmuster
<b>klassenbezogene Muster</b>	Klassenadapter	Interpreter Template	Fabrik
<b>objektbezogene Muster</b>	Objektadapter Dekorierer Kompositum Fassade Brücke Fliegengewicht Stellvertreter	Strategie Mediator Beobachter Verantwortlichkeitskette Kommando Iterator Erinnerer Zustand Besucher	Abstrakte Fabrik Singleton Erbauer Prototyp

- aus DAP 1 bekanntes Muster, das in SWT noch einmal betrachtet wird
- in SWT vorgestellte Muster

## Beschreibung von Entwurfsmustern

Angaben zur Zielsetzung:

- ☐ Welches Problem soll mit dem Muster gelöst werden?
- ☐ Was soll durch den Einsatz des Musters erreicht werden?
- ☐ Was macht das Muster?

Angaben zum Anwendungsbereich:

- ☐ Für welche Situationen passt das Muster?

Beschreibung von Struktur und Verhalten:

- ☐ Darstellung der Struktur durch **UML-Klassendiagramm**
- ☐ Verdeutlichung der Struktur an **UML-Objektdiagramm**
- ☐ Darstellung des Verhaltens durch **UML-Sequenzdiagramm**

Beispiele für die programmtechnische Umsetzung:

- ☐ Implementierung in Java
- ☐ Beispiele aus dem SWT-Starfighter

## Entwurfsmuster *Iterator*

### Ein **Iterator**

erlaubt den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts (Aggregat, z.B. Liste oder Baum), ohne dabei dessen zugrundeliegende Struktur aufzudecken.

#### Beispiele:

- ❑ sequentieller Durchlauf durch eine Liste
- ❑ sequentieller Durchlauf durch einen binären Baum

### Dabei sollen folgende Anforderungen erfüllt werden:

- ❑ Der Durchlauf soll unabhängig von der konkreten Struktur des Aggregats immer gleich erfolgen.
- ❑ Das Aggregat soll zum gleichen Zeitpunkt an mehreren Durchläufe mitwirken können.
- ❑ Es sollen verschiedene Arten von Durchläufen bereitgestellt werden können, ohne die Aggregat-Klasse aufzublähen.

Literatur: Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 48-52  
<http://www.springerlink.com/content/t38726/#section=660020&page=6&locus=71>

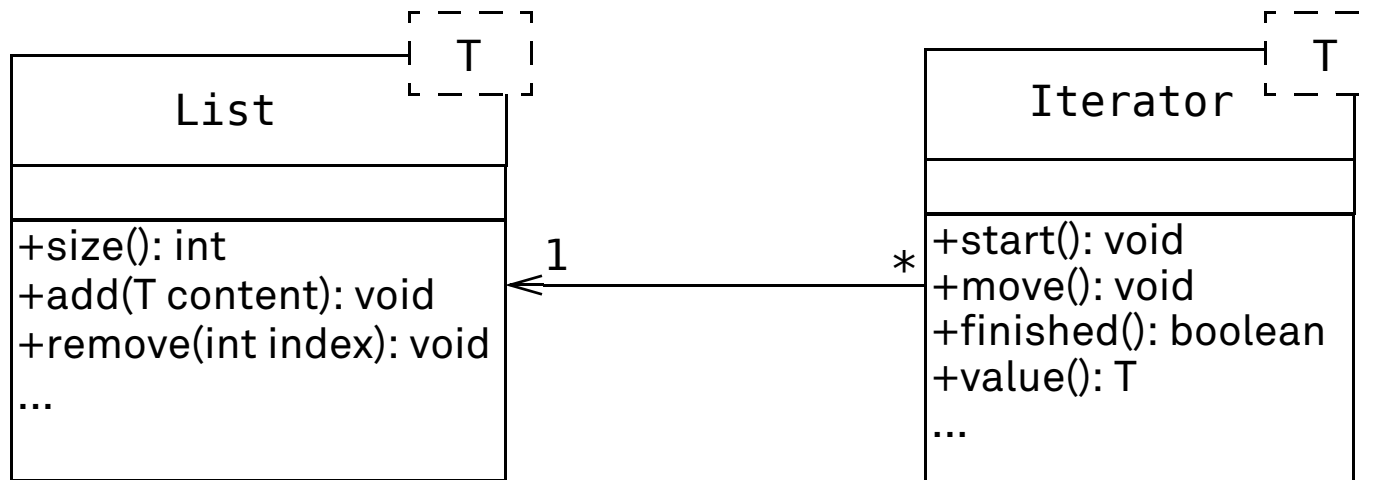
## Entwurfsmuster *Iterator*

(Fortsetzung)

Idee:

Zuständigkeit für den Durchlauf liegt nicht beim Aggregat-Objekt, sondern wird von einem **Iterator**-Objekt übernommen.

Beispiel Liste: allgemeine Struktur



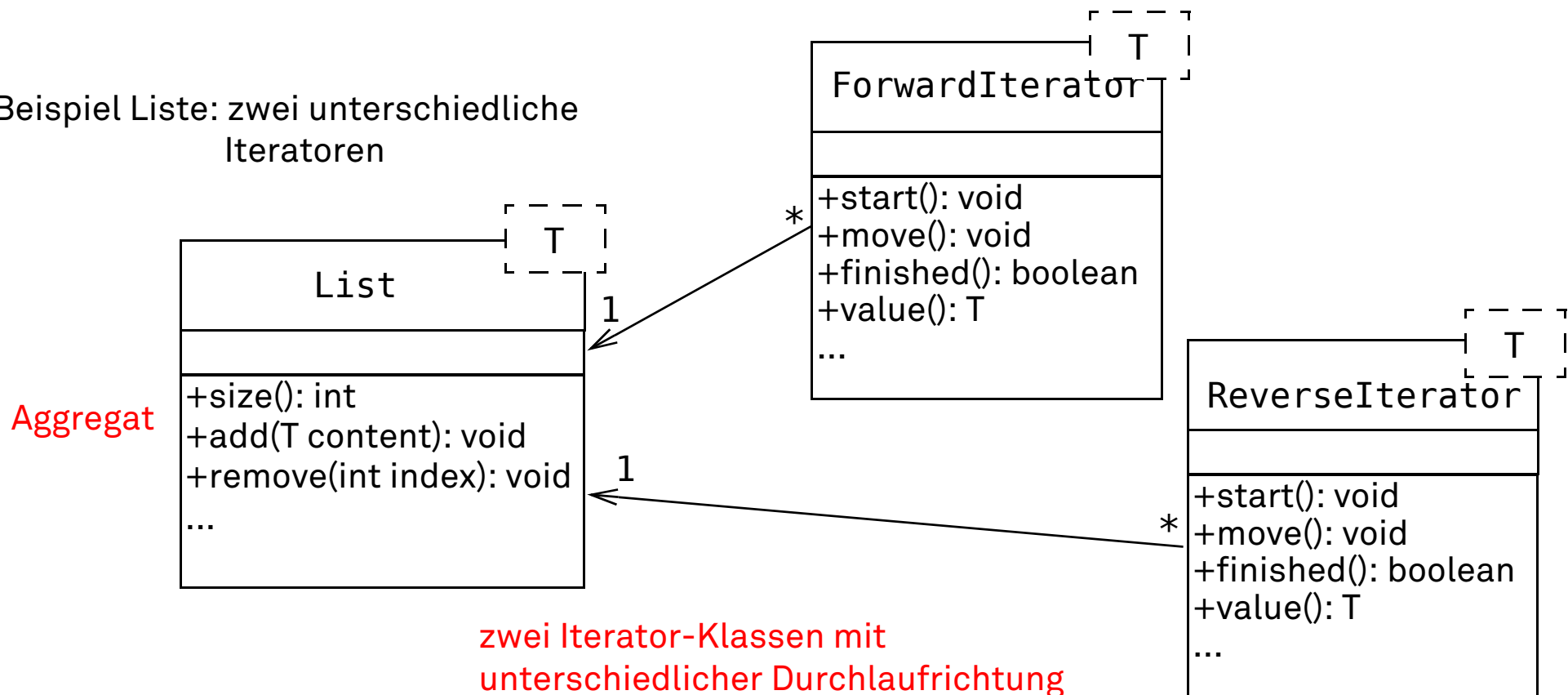
## Entwurfsmuster *Iterator*

(Fortsetzung)

Idee:

Zuständigkeit für den Durchlauf liegt nicht beim Aggregat-Objekt, sondern wird von einem **Iterator**-Objekt übernommen.

Beispiel Liste: zwei unterschiedliche Iteratoren



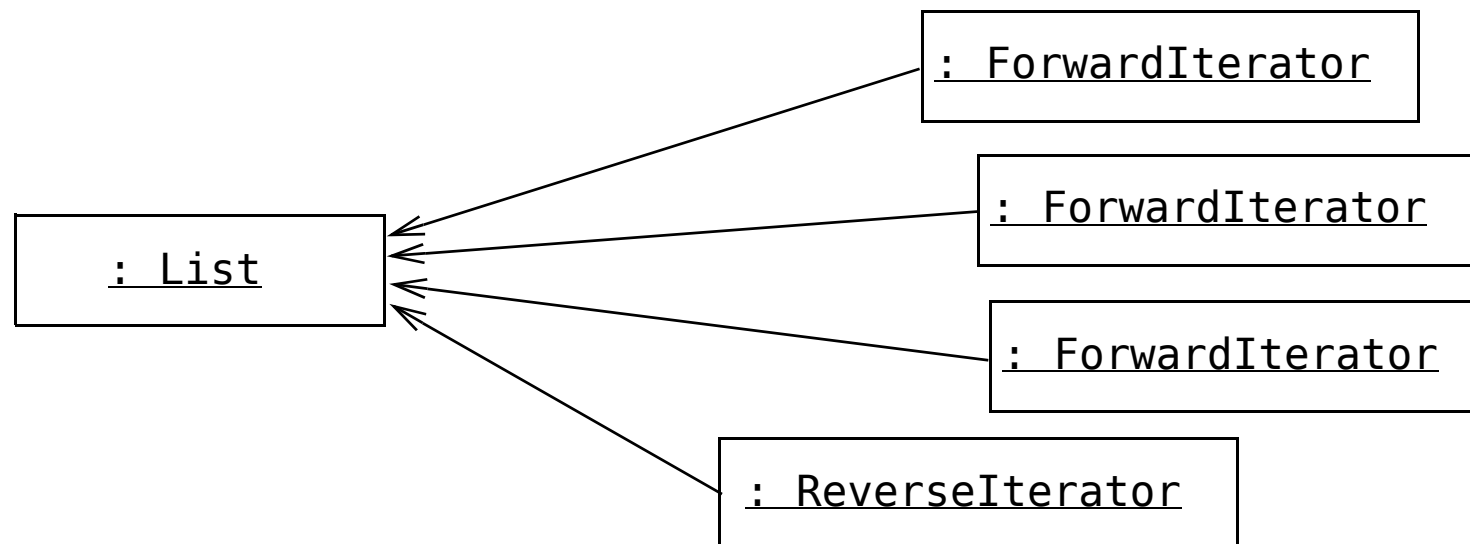
## Entwurfsmuster *Iterator*

(Fortsetzung)

Idee:

Zuständigkeit für den Durchlauf liegt nicht beim Aggregat-Objekt, sondern wird von einem **Iterator**-Objekt übernommen.

Beispiel Liste: Objektdiagramm



mehrere Iterator-Objekte laufen über die gleiche Liste

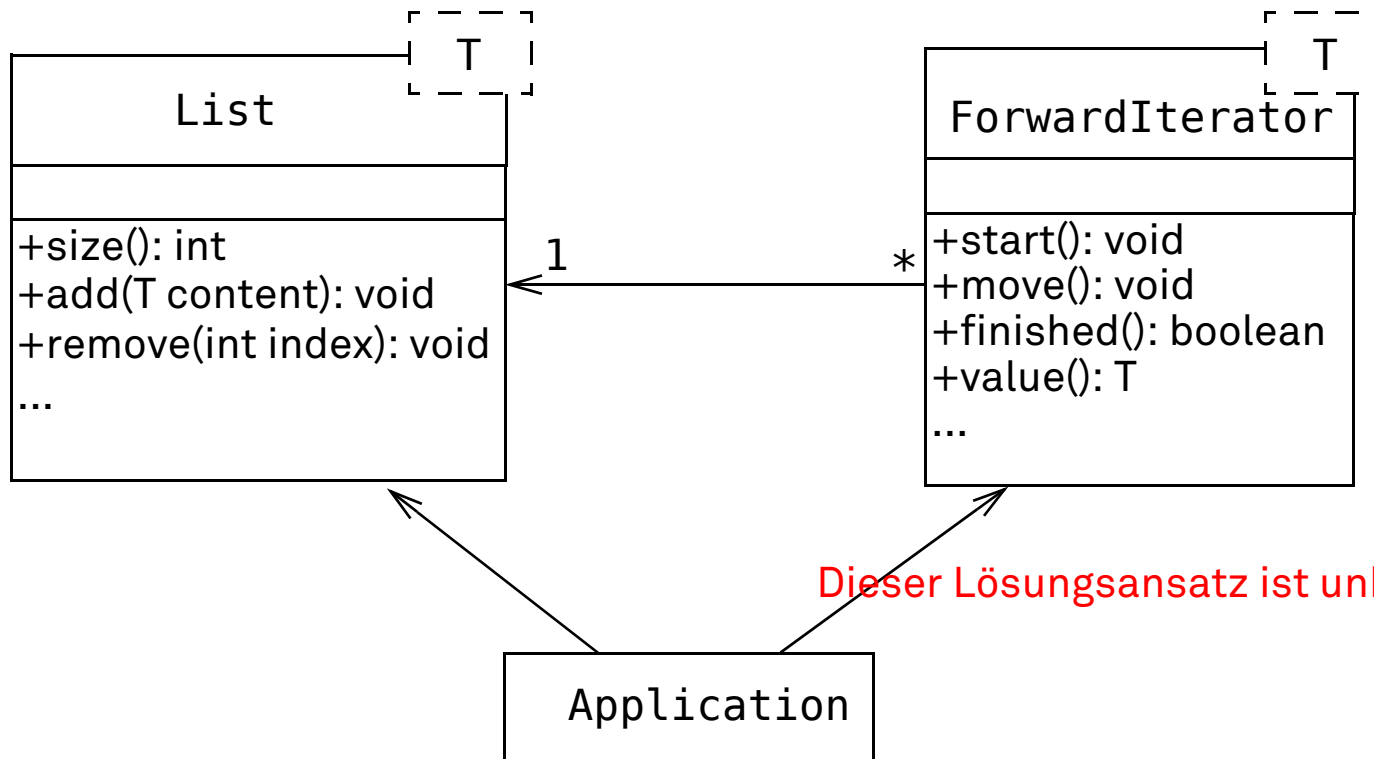
## Entwurfsmuster *Iterator*

(Fortsetzung)

Die Anwendung muss dafür sorgen, das Aggregat und Iterator miteinander verbinden.

aber:

- ❑ Die Anwendung müsste wissen, um welche Datenstruktur es sich beim Aggregat handelt.
- ❑ Der Iterator könnte nur auf öffentliche Methoden des Aggregats zugreifen.





## Entwurfsmuster *Iterator*

(Fortsetzung)

Die Anwendung muss dafür sorgen, das Aggregat und Iterator miteinander verbinden.

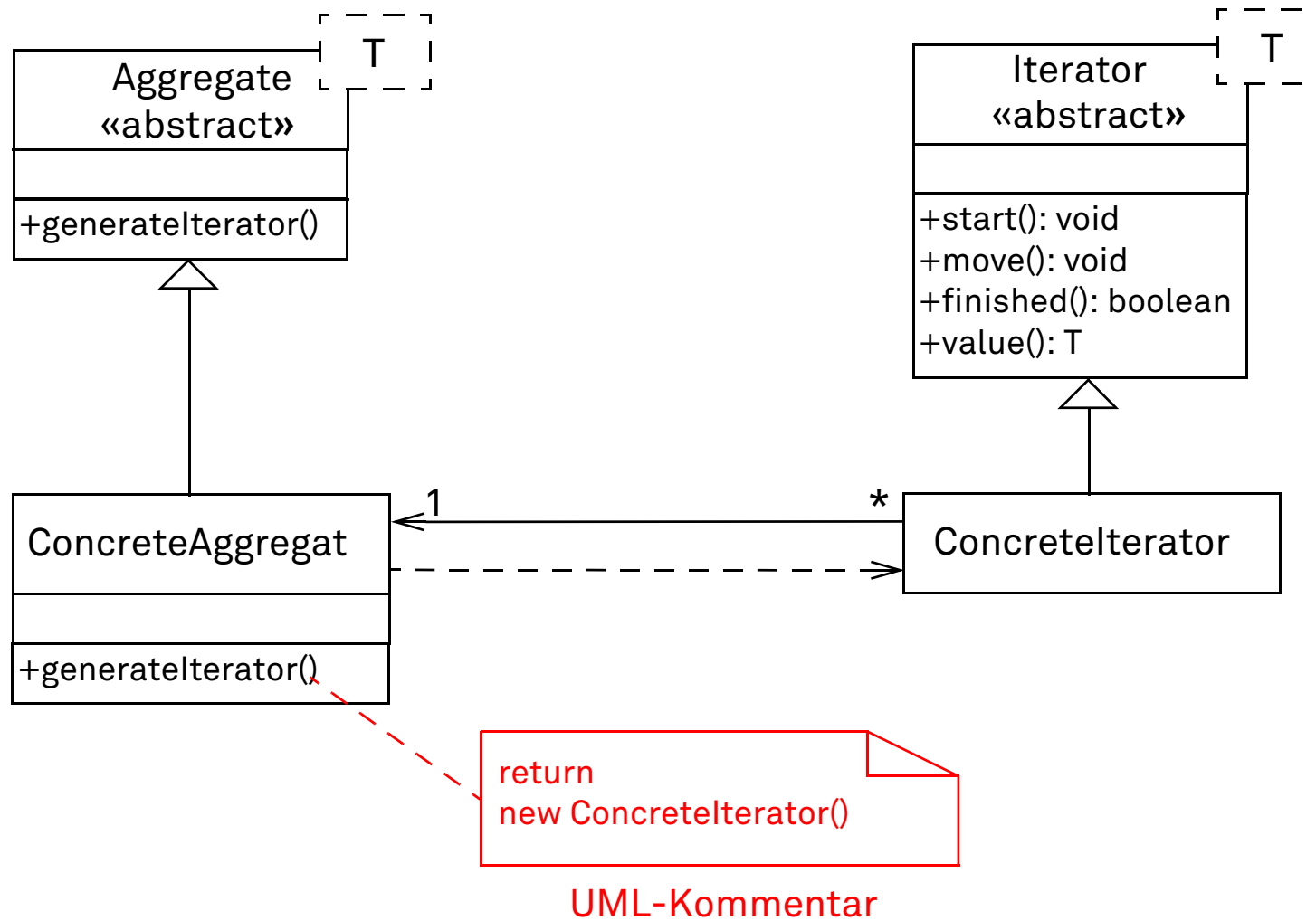
aber:

- ❑ Die Anwendung müsste wissen, um welche Datenstruktur es sich beim Aggregat handelt.
- ❑ Der Iterator könnte nur auf öffentliche Methoden des Aggregats zugreifen.

Daher folgende Verbesserungen:

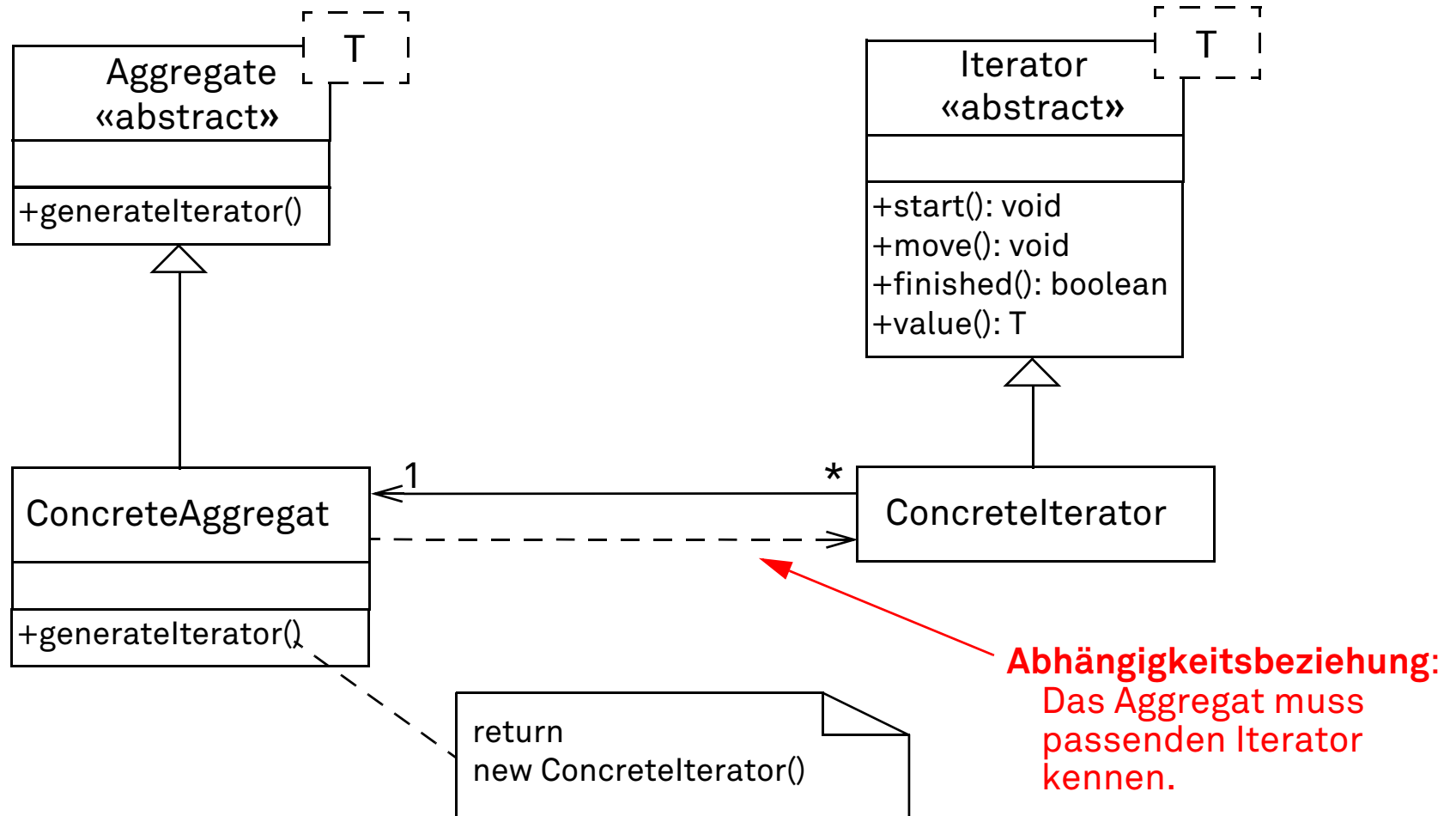
- ❑ Das Aggregat implementiert eine allgemeine Schnittstelle, die das Aggregat als *iterierbar* kennzeichnet.
- ❑ Der Iterator implementiert eine einheitliche Schnittstelle für Iteratoren, so dass alle Iteratoren gleich genutzt werden können.
- ❑ Jedes iterierbare Aggregat besitzt einen *eigenen* Iterator, der vom Aggregat bereitgestellt wird.
- ❑ Damit sind Aggregat und Iterator eng miteinander verzahnt. Der Iterator kann einen bevorzugten Zugriff auf das Aggregat erhalten und so eine (effiziente) Implementierung vornehmen.
- ❑ Die Anwendung erhält vom Aggregat auf Anforderung einen passenden Iterator, ohne dafür Details des Aggregats oder des Iterators kennen zu müssen.

## Klassendiagramm *Iterator* (allgemeine Darstellung)



## Klassendiagramm *Iterator* (allgemeine Darstellung)

(Fortsetzung)



## Eintwurfsmuster *Iterator* – Bewertung

### Vorteile:

- ❑ Die Implementierung der dem Aggregat zugrunde liegenden Datenstruktur bleibt verborgen.
- ❑ Aggregat-Objekte können durch Objekte anderer Klassen ersetzt werden, ohne die Anwendung ändern zu müssen, da der Zugriff über den Iterator unverändert bleibt.

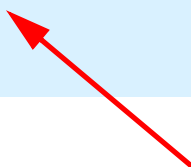
### Nachteile:

- ❑ Iteratoren erfordern zusätzlichen Programmieraufwand.
- ❑ Iteratoren müssen eventuell über Änderungen am Aggregat informiert werden. Dann muss das Aggregat seine Iteratoren kennen.

## Entwurfsmuster *Iterator*: Umsetzung in Java

- ❑ In Java wird das Iterator-Muster durch vorgegebene Interfaces und Klassen vorbereitet.
- ❑ Alle in Java-Standardbibliotheken bereitgestellten Aggregate unterstützen das Iterator-Muster.
- ❑ Ein Sprachkonstrukt (for-Schleife) unterstützt das Iterator-Muster.

```
interface Iterable<T> {  
    Iterator<T> iterator();  
}
```



Das Interface fordert nur,  
dass es eine Methode gibt,  
die einen Iterator zurückgibt.

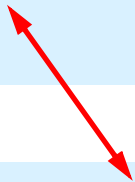
## Entwurfsmuster *Iterator*: Umsetzung in Java

(Fortsetzung)

- ❑ In Java wird das Iterator-Muster durch vorgegebene Interfaces und Klassen vorbereitet.
- ❑ Alle in Java-Standardbibliotheken bereitgestellten Aggregate unterstützen das Iterator-Muster.
- ❑ Ein Sprachkonstrukt (for-Schleife) unterstützt das Iterator-Muster.

```
interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

```
interface Iterator<E> {  
    boolean hasNext();  
    E next();  
}
```



Java-Iteratoren ermöglichen nur genau einen Durchlauf. Ein explizites Initialisieren ist daher nicht notwendig.

## Entwurfsmuster *Iterator*: Umsetzung in Java

(Fortsetzung)

- ❑ In Java wird das Iterator-Muster durch vorgegebene Interfaces und Klassen vorbereitet.
- ❑ Alle in Java-Standardbibliotheken bereitgestellten Aggregate unterstützen das Iterator-Muster.
- ❑ Ein Sprachkonstrukt (for-Schleife) unterstützt das Iterator-Muster.

```
interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

```
interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

optionale Methode:  
muss keine Wirkung haben.

## Entwurfsmuster *Iterator*: Umsetzung in Java

(Fortsetzung)

- ❑ Das **interface** `Iterable<T>`  
wird u.a. von folgenden Klassen implementiert:  
`ArrayList<E>`, `LinkedList<E>`, `PriorityQueue<E>`,  
`Stack<E>`, `HashSet<E>`, `TreeSet<E>`
- ❑ Jede dieser Klassen stellt über seine `iterator()`-Methode  
ein **passendes** `Iterator<E>`-Objekt bereit.
- ❑ Einige Klassen stellen zusätzliche Iteratoren bereit, z.B.:

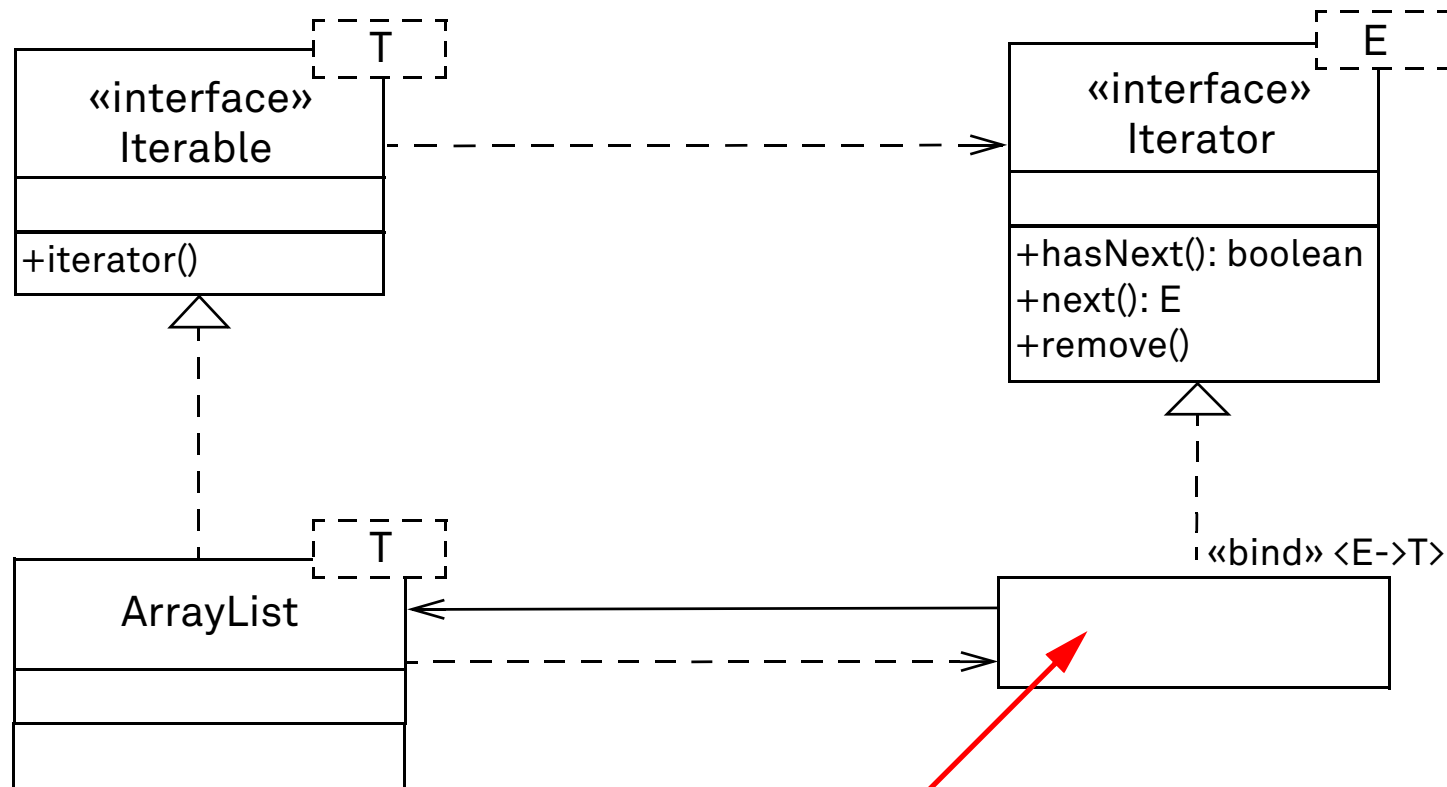
```
public ListIterator<E> listIterator()  
    (ein ListIterator besitzt zusätzliche Methoden, wie z.B. previous())
```

```
public ListIterator<E> listIterator(int index)  
    (index ist die Startposition des Durchlaufs)
```



## Entwurfsmuster *Iterator*: Umsetzung in Java

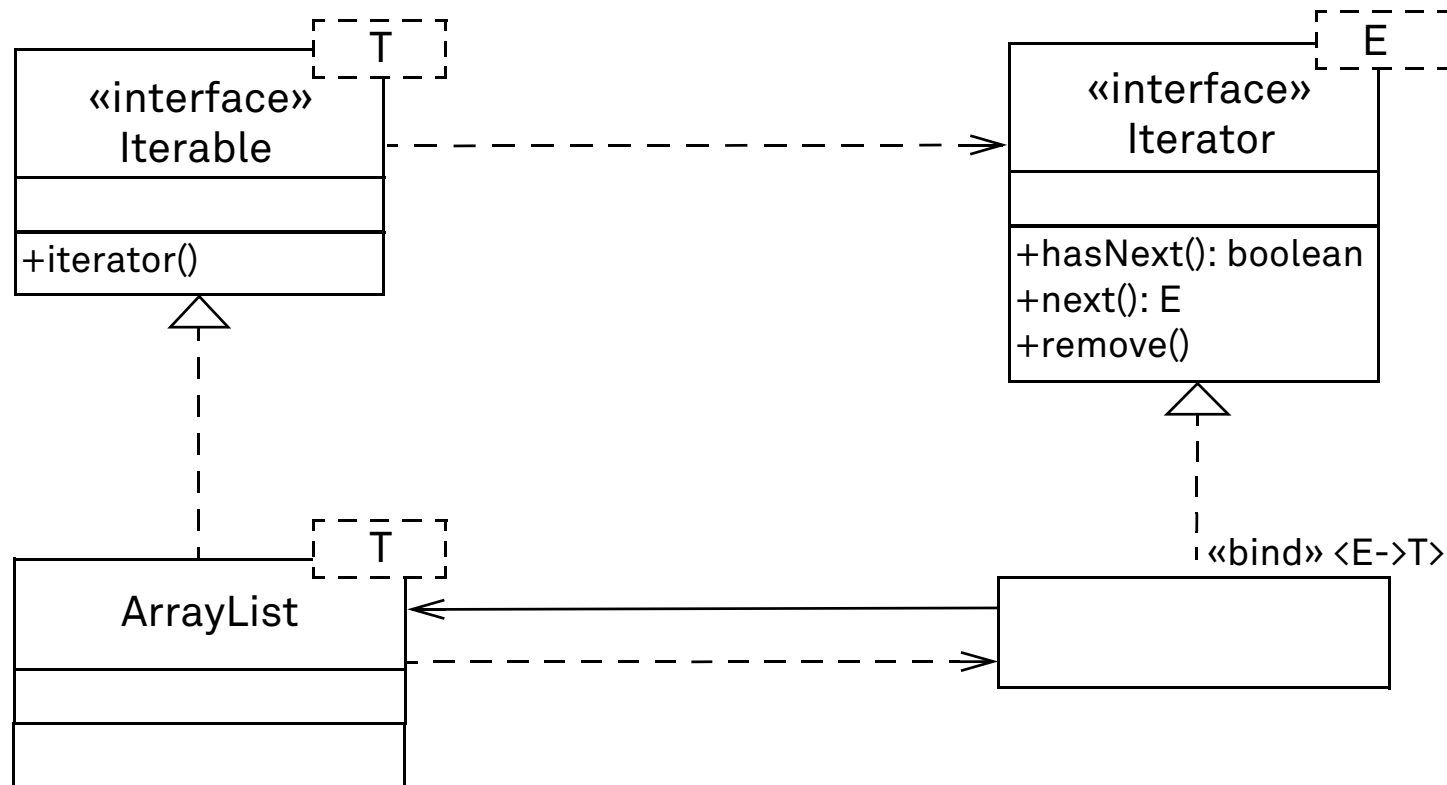
(Fortsetzung)



**Name unbekannt:**  
zur Benutzung reicht das Wissen,  
dass das Objekt das Interface **Iterator** implementiert

## Entwurfsmuster *Iterator*: Umsetzung in Java

(Fortsetzung)



## Iterator in Java: Beispiel für die Anwendung

allgemeine Methode zum Suchen auf beliebigen Strukturen, die Iterable implementieren:

```
<T> boolean check(Iterable<T> structure, T value) {  
    Iterator<T> it = structure.iterator();  
    while (it.hasNext()) {  
        if (value.equals(it.next())) { return true; }  
    }  
    return false;  
}
```

← liefert Iterator-Objekt

← prüft, ob der Iterator  
das Ende von structure  
erreicht hat

← liefert das nächste Element  
aus structure

## Iterator in Java: Beispiel für die Anwendung

(Fortsetzung)


allgemeine Methode zum Suchen auf beliebigen Strukturen, die Iterable implementieren:

```
<T> boolean check(Iterable<T> structure, T value) {  
    Iterator<T> it = structure.iterator();  
    while (it.hasNext()) {  
        if (value.equals(it.next())) { return true; }  
    }  
    return false;  
}
```

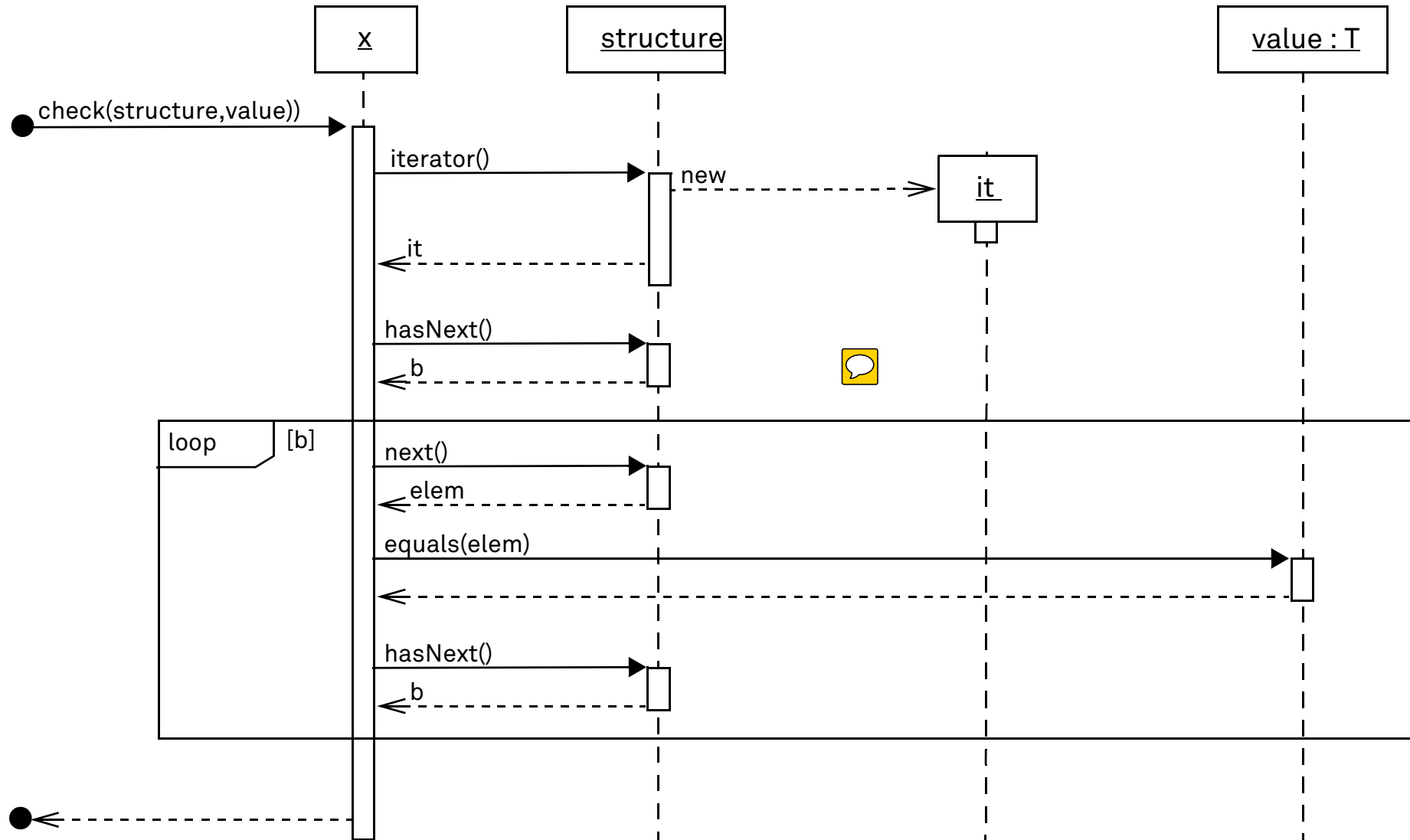
oder noch einfacher:

```
<T> boolean check(Iterable<T> structure, T value) {  
    for (T elem : structure) {  
        if (value.equals(elem)) { return true; }  
    }  
    return false;  
}
```

for-each-Schleife:  
nutzt Iterator-Muster  
aus Java



## Sequenzdiagramm zum Beispiel



## Beispiel für die Nutzung eines Iterators

in der Klasse `edu.udo.cs.swtsf.swing.game.SwingPainter`

```
public class SwingPainter extends ViewManager {  
    private void paintHUD(Graphics2D graphics) {  
        for (HudElement element : getHudElements()) {  
            ...  
        }  
    }  
    ...  
}
```



Vereinbarung in der Klasse `edu.udo.cs.swtsf.view.ViewManager`

```
public abstract class ViewManager {  
    public List<HudElement> getHudElements() {  
        return Collections.unmodifiableList(hudList);  
    }  
    ...  
}
```

## Beispielimplementierung für einen Iterator

statische innere Klasse im Interface `edu.udo.cs.swtsf.core.player.Laser`

```
public static class LaserIterator implements Iterator<Laser> {  
    private Laser current;  
    public LaserIterator(Laser start) {  
        current = start;  
    }  
    public boolean hasNext() {  
        return current != null;  
    }  
    public Laser next() {  
        if (!hasNext()) {  
            throw new NoSuchElementException();  
        }  
        Laser result = current;  
        current = current.getDecorated();  
        return result;  
    }  
}
```

## Beispielimplementierung für einen Iterator

im Interface `edu.udo.cs.swtsf.core.player.Laser`

```
public interface Laser extends Iterable<Laser> {  
    public abstract Laser getDecorated();  
    public default Iterator<Laser> iterator() {  
        return new LaserIterator(this);  
    }  
    public static class LaserIterator implements Iterator<Laser> {  
        ...  
    }  
}
```

### Anmerkungen:

- ❑ Der Iterator ist innerhalb des Interfaces vollständig implementiert.
- ❑ Eine implementierende Klasse muss lediglich die Methode `getDecorated()` implementieren.



## Iterator in Java: Beispiel für die Anwendung

(Fortsetzung)

Die Beispiele zeigen die Vorteile des Iterator-Musters:

- ❑ Die der Implementierung des Aggregats zugrundeliegende Datenstruktur hat keine Bedeutung für das Durchlaufen des Aggregats mit einem Iterator-Objekt.
- ❑ Verschiedene Aggregate können gleich behandelt werden, solange sie selbst das Interface `Iterable` implementieren.
- ❑ Aggregate können ausgetauscht werden, ohne die Anwendung zu ändern, da der Zugriff über den Iterator unverändert bleibt.
- ❑ Das Iterator-Objekt liegt außerhalb der zu iterierten Datenstruktur, so dass von außen über einen Methodenaufruf (z.B. `next()`) das Fortschreiten des Iterators bestimmt werden kann.
- ❑ Implementierungsmöglichkeiten:
  - Der Durchlauf, findet im Aggregat statt.
  - Es wird eine Kopie der Inhalte der Datenstruktur des Aggregats für den Iterator angelegt und beim Durchlauf wird diese Kopie benutzt.
- ❑ Problematisch bei der Implementierung eines Iterators ist immer:  
Wie wirken sich Änderungen in der durchlaufenen Datenstruktur auf den Iterator aus?