

3. Objektorientierte Analyse

3.4 Exkurs: Realisierung von UML-Klassen in Java

'Nothing to be alarmed about, sir, madam or thing. A perfectly routine take off.' The Doorbot then blacked out and lay in a tidy heap, while the ship accelerated at speeds far beyond its original specifications.

Douglas Adams, Starship Titanic

3. Objektorientierte Analyse

3.1 Systemanalyse

3.2 Statische Modellierung mit UML

3.3 Weitere UML-Diagramme in der Analyse

3.4 Exkurs: Realisierung von UML-Klassen mit Java

3.5 Dynamische Modellierung mit UML



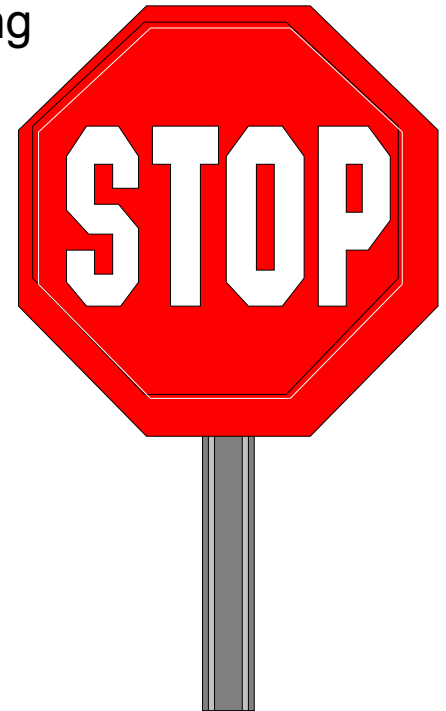
Fortsetzung



Obligatorische Literatur

Warnung: Modellierung vs. Realisierung

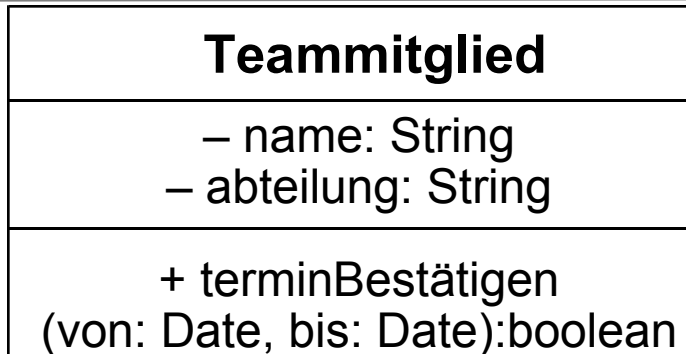
- **Zuerst** das Problem verstehen (Analyse),
dann Implementierungsdetails festlegen (Entwurf) !
 - Diese Regel gilt auch bei evolutionärer Entwicklung und eXtreme Programming !
- ▶ Klassischer Weg:
 - Für Analyse **nur UML**, noch keine Programmierung
- ▶ In dieser Vorlesung:
Java schon in der Analyse erwähnt - warum?
 - Java-Programmierung hilft beim Verständnis der abstrakten Konzepte der UML
 - **Pure fachliche Modelle** können im Sinne eines Prototyps bereits in Programmiersprache realisiert werden:
 - Keine Benutzeroberfläche, keine Datenhaltung
 - Programmiersprache als Werkzeug zum besseren Verständnis des abstrakten Modells



Hinweis: Online-Ressourcen!

- ▶ Über die Homepage der Lehrveranstaltung (bei "Vorlesungen") finden Sie die Datei
`Terminv.java`
- ▶ Diese Datei enthält eine vollständige Umsetzung des Beispiels "Terminverwaltung" in lauffähigen Java-Code.
- ▶ Empfohlene Benutzung:
 - Lesen
 - Übersetzen, Starten, Verstehen
 - Modifizieren
 - Kritisieren

Realisierung von Klassen



Kein Ergebnistyp angegeben:
Java-Pseudotyp "void"



```
class Teammitglied {  
  
    private String name;  
    private String abteilung;  
  
    public boolean terminBestaetigen (Date von, Date bis) {  
        // Methodenrumpf  
    }  
}
```

Zwei Alternativen für Datentypen in UML-Diagrammen:

- Sprachunabhängige UML-Typen
- Java-Datentypen

"Modifier" für Deklarationen in Java

- Java: Vor den Deklarationen von Attributen und Operationen oft längere Ketten von Schlüsselworten
Bsp.: `public static void main(...)`
- Sichtbarkeiten: Umsetzung der UML-Sichtbarkeiten
- Klassenattribute und -operationen: Schlüsselwort `static`

C
- a1: T1 # a2: T2 + a3: T3 <u>+a4: T4</u>
- op1() # op2() + op3() <u>+op4()</u>



```
class C {  
  
    private T1 a1;  
    protected T2 a2;  
    public T3 a3;  
    public static T4 a4;  
  
    private ... op1(...) {...}  
    protected ... op2(...) {...}  
    public ... op3(...) {...}  
    public static ... op4(...) {...}  
}
```

Klassenattribute und -operationen in Java

- ▶ Die einfache Umsetzung mit "static" ist nicht immer passend:
 - Eine Klasse "kennt" in Java nicht die Gesamtheit ihrer Instanzen.
 - Beispiele:
 - Attribut "anzahl" in Besprechungsraum muß explizit beim Erzeugen von Objekten hochgezählt werden
 - Operation "freienRaumSuchen()" nur über einer Datenstruktur realisierbar, in der alle Besprechungsraumobjekte zugänglich sind
- ▶ Praktische Lösungen:
 - *Verwaltungsobjekte*, die alle Instanzen einer Klasse *verwalten*
 - "Anmeldung" bei einer Verwaltung als Bestandteil der Konstruktoroperation
 - "Fabrikmethode" statt Konstruktor
- ▶ **Genauere Entscheidung erst im Entwurf !**

Konstruktor(operation) für Objekte

```
class Teammitglied {  
  
    private String name;  
    private String abteilung;  
  
    public Teammitglied (String n, String abt) {  
        name = n;  
        abteilung = abt;  
    }  
  
    public Teammitglied (String n) {  
        name = n;  
        abteilung = "";  
    }  
    ... }
```

```
Teammitglied m = new Teammitglied("Max Müller", "Abt. B");
```

Konstruktoren werden meist *überladen*.

Überladung

- ▶ Auswahl aus mehreren gleichnamigen Operationen nach Anzahl und Art der Parameter.
- ▶ Klassisches Beispiel: Arithmetik
 - + - Operation: (Nat,Nat)Nat, (Int,Int)Int, (Real,Real)Real
- ▶ Java-Operationen:

```
int f1 (int x, y) {...}
int f1 (int x) {...}
int x = f1(f1(1,2));
```
- ▶ Konstruktoren:

```
public Teammitglied () {...}
public Teammitglied (String n) {...}
public Teammitglied (String n, String abt) {...}
```

Default-Konstruktor

```
class Teammitglied {  
    ... // ohne Konstruktor  
}
```

```
Teammitglied m = new Teammitglied();
```

- ▶ Für jede Klasse gibt es immer mindestens einen Konstruktor.
- ▶ Wenn kein benutzerdefinierter Konstruktor vorliegt, wird ein "Default-Konstruktor" bereitgestellt.
- ▶ Der Default-Konstruktor ist nicht mehr für die Klasse verfügbar, wenn irgendein benutzerdefinierter Konstruktor definiert wird !

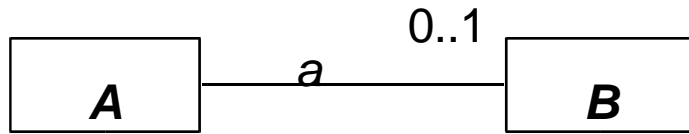
"This"

```
class Teammitglied {  
  
    private String name;  
    private String abteilung;  
  
    public Teammitglied (String name, String abteilung) {  
        this.name = name;  
        this.abteilung = abteilung;  
    }  
    ...  
};
```

```
Teammitglied m = new Teammitglied("Max Müller", "Abt. B");
```

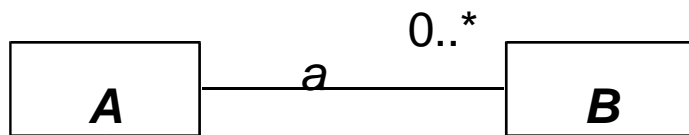
- ▶ Bezüge auf Attributwerte und Operationsaufrufe werden i.a. mit *Objekt . Name* angegeben.
- ▶ "this" bedeutet das Objekt, auf dem die Methode ausgeführt wird.
- ▶ "this" wird automatisch ergänzt, wenn die Objektangabe fehlt.

(Einseitige) Realisierung von Assoziationen



Einseitige Realisierung
(aus Sicht der Klasse A)

```
class A {
    ...
    private B a;
    ...
}
```



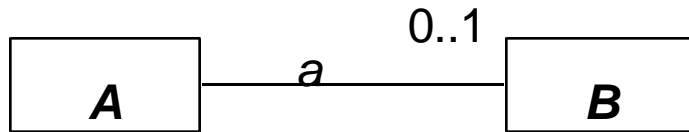
```
class A {
    ...
    private B[] a;
    ...
}
```

Annahme:

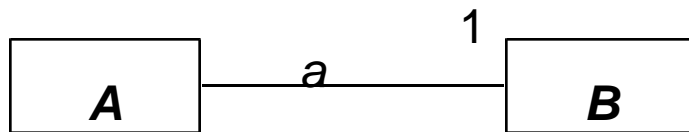
Obergrenze der Anzahl der *B*-Objekte spätestens bei erstmaliger Eintragung von Assoziationsinstanzen bekannt und relativ klein.

(Allgemeinere Realisierungen siehe später.)

Optionale und notwendige Assoziationen



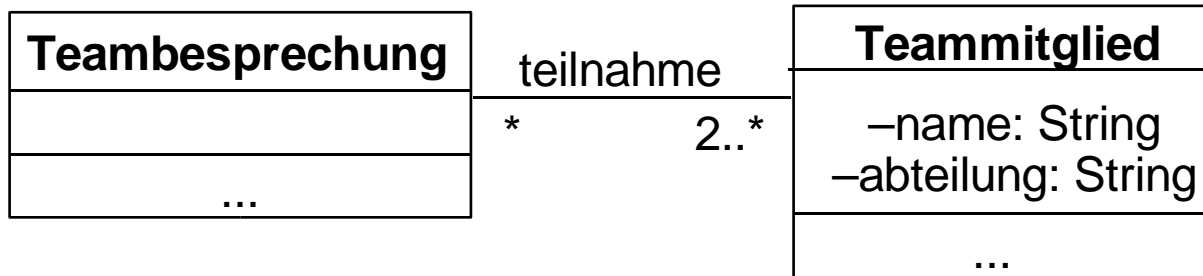
```
class A {
    ...
    private B a;
    ...
}
```



```
class A {
    ...
    private B a;
    ...
    public A (B a, ...) {
        this.a = a; ...
    }
}
```

Analog z.B. für Multiplizitäten 0..* und 1..*

Bidirektionale Assoziationen: Beispiel



```
class Teambesprechung {
    private Teammitglied[] teilnahme;
    ...
    public Teambesprechung (Teammitglied[] teilnehmer) {
        this.teilnahme = teilnehmer;
    }
}

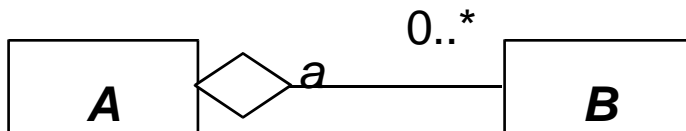
class Teammitglied {
    private String name;
    private String abteilung;
    private Teambesprechung[] teilnahme; ...
}
```

Damit wird eigentlich die Multiplizität "0..*" realisiert! (Verallgemeinerung)

Bidirektionalität von Assoziationen

- ▶ Realisierung nur in einer Richtung:
 - Gibt nicht die volle Semantik des Modells wieder
 - Abhängig von Benutzung (Navigation) der Assoziation
- ▶ Realisierung durch zwei Assoziations-Variablen:
 - Siehe vorhergehendes Beispiel ("teilnahme")
 - Redundanz:
 - zusätzlicher Speicheraufwand
 - Gefahr von Inkonsistenzen
- ▶ Realisierung durch Assoziationsklasse:
 - Geeignete Datenstrukturen erforderlich
 - "Beidseitige" Abfragemöglichkeit
 - Sicherstellung der Mengeneigenschaft
- ▶ **Genauere Entscheidung erst im Entwurf !**

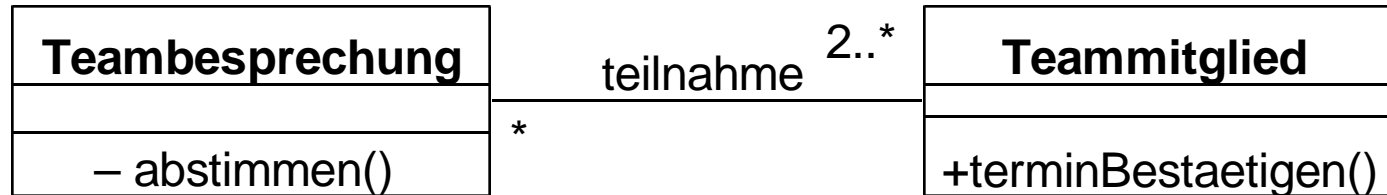
Realisierung von Aggregation



```
class A {
    ...
    private B[] a;
    ...
}
```

- ▶ Aggregationen können genau analog zu Assoziationen abgebildet werden !
- ▶ Komposition wird am einfachsten in gleicher Weise abgebildet.

Beispiel Methodenrumpf (1)

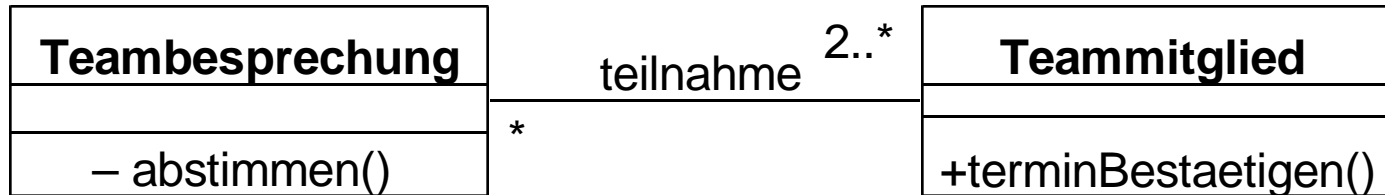


```
class Teambesprechung {

    private Teammitglied[] teilnahme; ...

    private boolean abstimmen (Hour beginn, int dauer) {
        boolean ok = true;
        for (int i=0; i<teilnahme.length; i++)
            ok = ok &&
                teilnahme[i].terminBestaetigen(beginn, dauer);
        return ok;
    }
}
```

Beispiel Methodenrumpf (2)

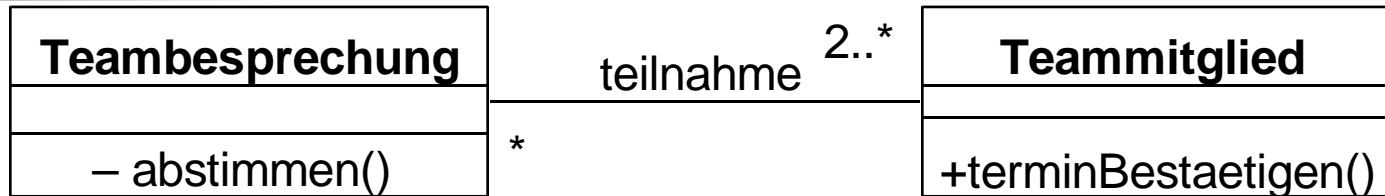


```
class Teambesprechung {

    private Teammitglied[] teilnahme; ...
    private boolean abstimmen (Hour beginn, int dauer) ...

    public Teambesprechung
    (String titel, Hour beginn, int dauer,
    Teammitglied[] teilnehmer) {
        ... titel, beginn, dauer lokal speichern
        this.teilnahme = teilnehmer;
        if (! abstimmen(beginn, dauer))
            System.out.println("Termin bitte verschieben!");
        else ...
    } ...
}
```

Beispiel Methodenrumpf (3)



```
class Teammitglied { ...  
  
    private Teambesprechung[] teilnahme;  
  
    public boolean terminBestaetigen (Hour beginn,int dauer) {  
        boolean konflikt = false;  
        int i = 0;  
        while (i < teilnahme.length && !konflikt) {  
            if (teilnahme[i].inKonflikt(beginn, dauer))  
                konflikt = true;  
            else  
                i++;  
        };  
        return !konflikt;  
    } ...  
}
```

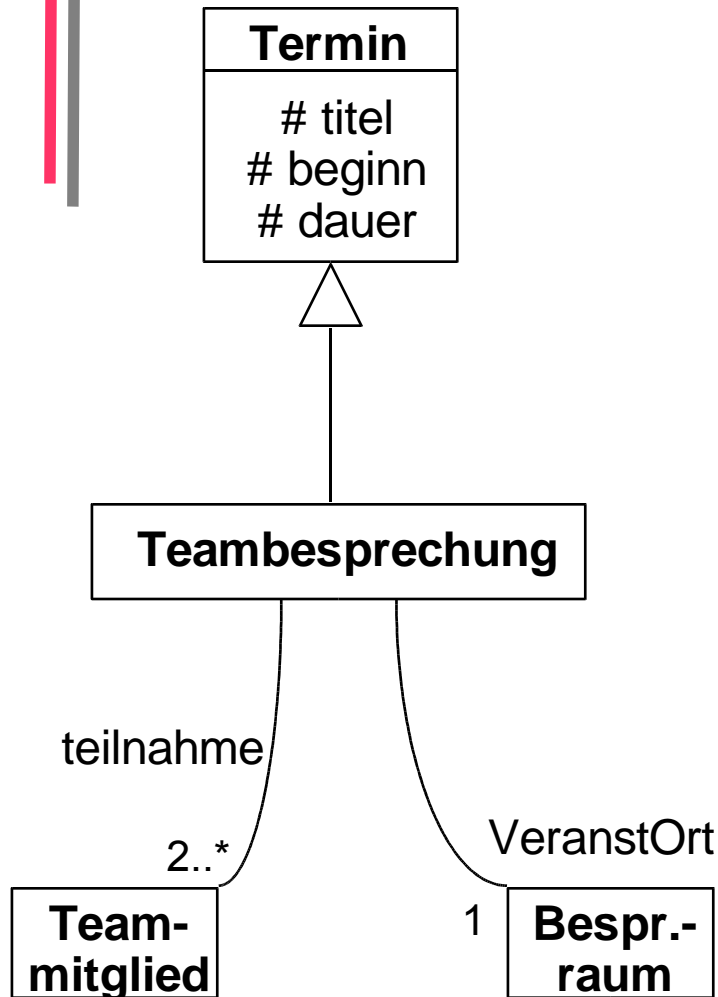
Beispiel Hauptprogramm

- ▶ Ein Programm auf höherer Ebene muss verdrahten. Dies kann das Hauptprogramm sein
- ▶ Verdrahtung bedeutet: Aufbau eines Objektnetzes, so, dass die Objekte kommunizieren können

```
class Terminv {
    public static void main (String argv[]) {
        Besprechungsraum r1 = new Besprechungsraum("R1", 10);
        Besprechungsraum r2 = new Besprechungsraum("R2", 20);
        Teammitglied mm = new Teammitglied("M. Mueller", "Abt. A");
        Teammitglied es = new Teammitglied("E. Schmidt", "Abt. B");
        Teammitglied fm = new Teammitglied("F. Maier", "Abt. B");
        Teammitglied hb = new Teammitglied("H. Bauer", "Abt. A");
        Hour t1s5 = new Hour(1,5); // Tag 1, Stunde 5
        Teammitglied[] t1B1 = {mm, es};
        Teambesprechung tb1 =
            new Teambesprechung("Bespr. 1", t1s5, 2, t1B1);
        tb1.raumFestlegen();

        ...
    }
}
```

Vererbung



```
class Termin {
    ...
    protected String titel;
    protected Hour beginn;
    protected int dauer;
    ...
};
```

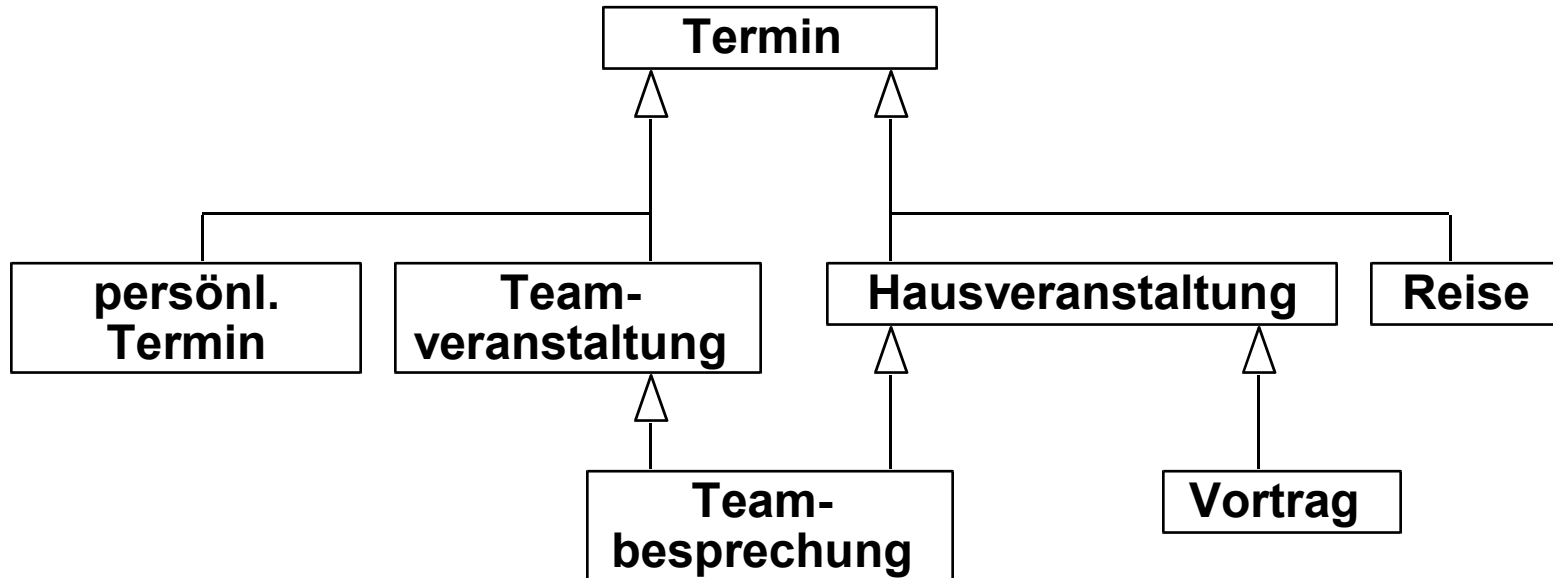
```
class Teambesprechung
    extends Termin {

    private Teammitglied[] teilnahme;
    private BesprRaum veransthOrt;

    ...
};
```

Mehrfachvererbung

- ▶ In UML ist es prinzipiell möglich, daß eine Klasse von mehreren Klassen erbt:



- ▶ Java unterstützt Mehrfachvererbung **nicht** !

```
class Teambesprechung
    extends Teamveranstaltung, Hausveranstaltung
```



Abstrakte Klassen und Operationen

```
abstract class Termin {  
    ...  
    protected String titel;  
    protected Hour beginn;  
    protected int dauer;  
    ...  
    public abstract boolean verschieben (Hour neu);  
};
```

```
class Teambesprechung  
    extends Termin {  
    ...  
    public boolean verschieben (Hour neu)    {  
        boolean ok = abstimmen(neu, dauer);  
        if (ok) {  
            beginn = neu;  
            raumFestlegen();  
        };  
        return ok;  
    };  
};
```

Jede abstrakt deklarierte Methode muß in der Unterklasse realisiert werden - sonst können keine Objekte der Unterklasse erzeugt werden!

Konstruktor bei Vererbung

```
class Termin {  
    ...  
    protected String titel;  
    protected Hour beginn;  
    protected int dauer;  
    ...  
    public Termin  
        (String t, Hour b, Dauer d) {  
        titel = t; beginn = b; dauer = d; };  
};
```

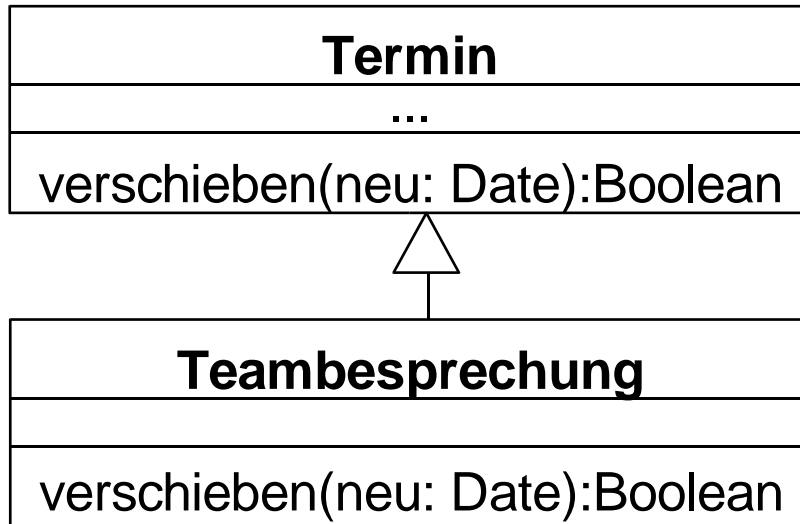
"super"
ermöglicht Aufruf
von Methoden, die
in der Oberklasse
definiert sind.

Spezielle Syntax
für Konstruktoren!

"super" immer als
erste Anweisung!

```
class Teambesprechung extends Termin {  
    ...  
    private Teammitglied[] teilnahme;  
    ...  
    public Teambesprechung (String t, Hour b, Dauer d,  
        Teammitglied[] tn) {  
        super(t, b, d);  
        teilnahme = tn; ... };  
};
```

"super" in überschriebenen Operationen



```
class Termin {
    boolean verschieben
        (neu: Date) {
        ...z.B. Aufzeichnung
            in Log-datei
        }
    ...
}
```

```
class Teambesprechung
    extends Termin

    boolean verschieben
        (neu: Date) {
        ...
        super.verschieben (neu)
        }
    ...
}
```

"super"
ermöglicht auch den Zugriff auf
überschriebene Versionen von
Operationen, wie sie in einer
Oberklasse definiert sind.

Andere Syntax als bei Konstruktoren!

Die Klasse "Object"

- `java.lang.Object`: allgemeine Eigenschaften aller Objekte.
 - Jede Klasse ist Unterklasse von `Object` ("extends `Object`").
 - Diese Vererbung ist *implizit* (d.h. man kann "extends `Object`" weglassen).
 - Dadurch werden Probleme mit Mehrfachvererbung vermieden.

```
class Object {  
    public boolean equals (Object obj);  
    public int hashCode();  
    public String toString(); ...  
}
```

- Jede Klasse kann die Standard-Operationen überdefinieren:
 - `equals`: Objektgleichheit (Standard: Referenzgleichheit)
 - `hashCode`: Zahlcodierung
 - `toString`: Textdarstellung, z.B. für `println()`

Tests und Szenarien

- ▶ Hauptprogramm definiert Testabläufe:

```
Teammitglied[] t1B1 = {mm, es};
Teambesprechung tb1 =
    new Teambesprechung("Bespr. 1", t1s5, 2, t1B1);
tb1.raumFestlegen();
Teammitglied[] teilnTb2 = {mm, fm};
Teambesprechung tb2 =
    new Teambesprechung("Bespr. 2", t2s3, 2, teilnTb2);
tb2.raumFestlegen();
Teammitglied[] teilnTb3 = {es, fm};
Teambesprechung tb3 =
    new Teambesprechung("Bespr. 3", t2s4, 2, teilnTb3);
tb3.raumFestlegen();
```

- ▶ Testablauf = Szenario!
- ▶ Zugehöriger Anwendungsfall: "Teambesprechung organisieren"

Löschen von Objekten

- ▶ In Java gibt es **keine** Operation zum Löschen von Objekten (keine "Destruktoren").
 - Andere objektorientierte Programmiersprachen kennen Destruktoren (z.B. C++)
- ▶ Speicherfreigabe in Java:
 - Sobald keine Referenz mehr auf ein Objekt besteht, *kann* es gelöscht werden.
 - Der konkrete Zeitpunkt der Löschung wird vom Java-Laufzeitsystem festgelegt ("garbage collection").
 - Aktionen, die vor dem Löschen ausgeführt werden müssen:
`protected void finalize()` (über)definieren
- ▶ Fortgeschrittene Technik zur Objektverwaltung (z.B. für Caches):
 - "schwache" Referenzen (verhindern Freigabe nicht)
 - erst ab Java 1.2 (`java.lang.ref`)

Ausnahmebehandlung in Java

- ▶ **Ausnahme (*Exception*):**
 - Objekt einer Unterklasse von `java.lang.Exception`
 - Vordefiniert oder und selbstdefiniert
- ▶ Ausnahme **auslösen** (*to **throw** an exception*)
 - Erzeugen eines `Exception`-Objekts
 - Löst Suche nach Behandlung aus
- ▶ Ausnahme **abfangen** und **behandeln** (*to **catch** and **handle** an exception*)
 - Aktionen zur weiteren Fortsetzung des Programms bestimmen
- ▶ Ausnahme **deklarieren**
 - Angabe, daß eine Methode außer dem normalen Ergebnis auch eine Ausnahme auslösen kann (Java: **throws**)
 - Beispiel aus `java.io.InputStream`:

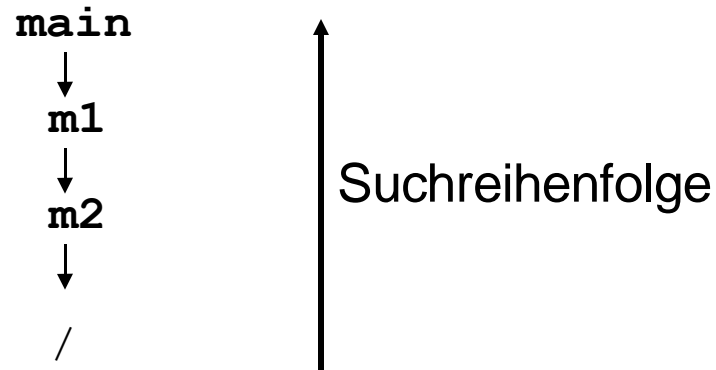
```
public int read() throws IOException;
```

Java-Syntax für Ausnahmebehandlung

```
class TotalDiv {
    public static int tDiv (int x, int y) {
        try {
            return (x / y);
        }
        catch (ArithmeticException e) {
            return 0;
        }
    }
}
```

Suche nach Ausnahmebehandlung

- Suche nach Abfangklausel (**catch**) entlang der (dynamischen) Aufrufhierarchie:



- Bei mehreren Abfangklauseln an der gleichen Stelle der Hierarchie gilt die zuerst definierte Klausel:

```
try { }
catch (xException e)
catch (yException e)
```

↓ Suchreihenfolge

Definition neuer Ausnahmen

```
class TestException extends Exception {
    public TestException () {
        super ();
    }
}

class SpecialAdd {
    public static int sAdd (int x, int y)
        throws TestException {
        if (y == 0)
            throw new TestException ();
        else
            return x + y;
        }
}
```

Benutzung von vordefinierten Ausnahmen möglich und empfehlenswert !

Deklaration und Propagation von Ausnahmen

- ▶ Wer eine Methode aufruft, die eine Ausnahme auslösen kann, muß
 - entweder die Ausnahme abfangen
 - oder die Ausnahme weitergeben (*propagieren*)
- ▶ Propagation in Java: Deklarationspflicht mittels **throws** (außer bei Error und RuntimeException)

```
public static void main (String[] argv) {  
    System.out.println (SpecialAdd.sAdd (3, 0) );  
}
```

Java-Compiler: Exception TestException must be caught, or it must be declared in the throws clause of this method.

Regeln zum Umgang mit Ausnahmen

- ▶ Ausnahmebehandlung niemals zur Behandlung normaler (d.h. häufig auftretender) Programmsituationen einsetzen
- ▶ Ausnahmebehandlung nicht zu komplex gestalten
- ▶ Auf keinen Fall Ausnahmen “abwürgen”, z.B. durch triviale Ausnahmebehandlung
- ▶ Ausnahmen zu propagieren ist keine Schande, sondern erhöht die Flexibilität des entwickelten Codes.

Wiederholung: Ablauf eines Java-Programms

```
class Counter {  
    private int ct;  
    public void inc() { ct++; }  
    public void dec() { ct--; }  
    public void reset() { ct = 0; }  
    public void set(int s) { ct = s; }  
}
```

```
...  
Counter c1 = new Counter();  
c1.reset();  
Counter c2 = new Counter();  
c2.set(3);  
c1.inc();  
c2.dec();  
c1.inc();  
c2.dec();
```

- ▶ Visualisierung des Ablaufs:
 - UML-Sequenzdiagramm
 - Folge von UML-Objektdiagrammen

3. Objektorientierte Analyse

3.1 Systemanalyse

3.2 Statische Modellierung mit UML

3.3 Weitere UML-Diagramme in der Analyse

3.4 Realisierung von UML-Klassen mit Java

3.5 Dynamische Modellierung mit UML



Fortsetzung



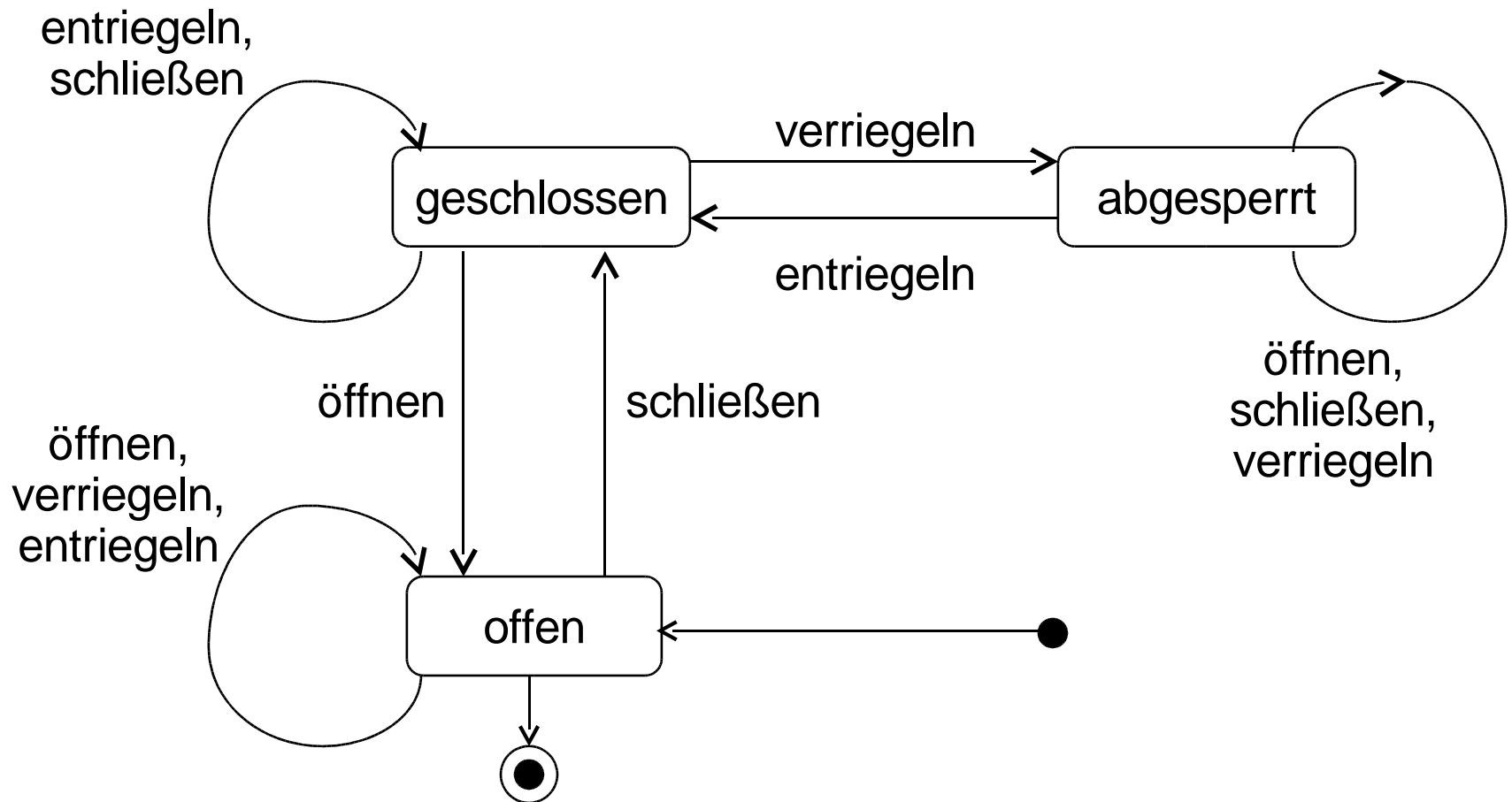
3. Objektorientierte Analyse

3.5 Dynamische Modellierung mit UML

Sed fugit interea, fugit inreparabile tempus.

Vergil, 70-16 v. Chr.

Beispiel: Zustandsmodell einer Tür



Zustandsmodelle und endliche Automaten

Theoretische Informatik, Automatentheorie:

Ein endlicher Zustandsautomat über einem Alphabet A von Ereignissen ist ein Tupel, bestehend aus:

- einer Menge S von Zuständen
- einer (partiellen) Übergangsfunktion $\delta : S \times A \rightarrow S$
- einem Startzustand $s_0 \in S$
- einer Menge von Endzuständen $S_f \subseteq S$



δ (geschlossen, verriegeln) = abgesperrt

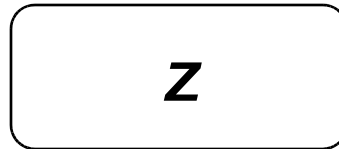
Semantik eines Zustandsmodells

- ▶ Die Semantik eines Zustandsmodells ist definiert als Menge von Sequenzen:
 - in der Theoretischen Informatik:
Menge von "akzeptierten Wörtern"
(über Grundalphabet von Ereignissen)
 - in der Softwaretechnik:
Menge von zulässigen *Ereignisfolgen*
- ▶ Wichtige Verallgemeinerung: "Automaten mit Ausgabe"
 - *Mealy-Automaten*: Ausgabe bei Übergang
 - Softwaretechnik: *Aktion* bei Übergang
 - *Moore-Automaten*: Ausgabe bei Erreichen eines Zustands
 - Softwaretechnik: *Eintrittsaktion (entry action)*

UML-Zustandsmodelle

- ▶ **Definition:** Ein *Zustand* ist eine Eigenschaft eines Systems, die über einen begrenzten Zeitraum besteht.

- ▶ **Notation:**



- ▶ Was ist ein "System"?
 - Technisch: Ein Objekt oder eine Gruppe von Objekten
 - Praktisch:
 - Eigenschaft eines komplexen Softwaresystems
 - Eigenschaft eines Arbeitsprozesses
 - Eigenschaft eines Produkts eines Arbeitsprozesses
 - Eigenschaft eines einzelnen Objekts (im Extremfall)

Ereignisse

- ▶ **Definition** Ein **Ereignis** ist ein Geschehen von vernachlässigbarer Zeitdauer, das auf das betrachtete System Auswirkungen hat.
- ▶ Eine **Ereignisklasse** wird durch ihren Namen und evtl. weitere Parameter beschrieben.
- ▶ Eine **Ereignisinstanz** ist ein Objekt einer Ereignisklasse, das unter Umständen durch konkrete Werte für **Parameter** der Ereignisklasse näher beschrieben wird.

Sprachgebrauch: Ereignis = Ereignisklasse = Ereignisinstanz

- ▶ **Notation:**
 E
 $E (P1, \dots, Pn)$
 $E (P1 : T1, \dots, Pn : Tn)$

Arten von Ereignissen

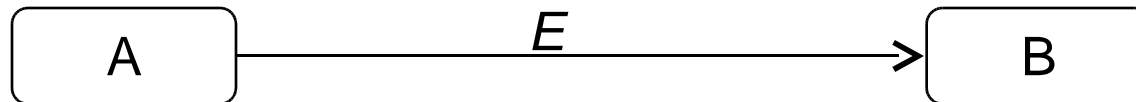
- ▶ Allgemeingültige Arten von Ereignissen:
 - Empfang einer Nachricht von außen
 - Ablauf einer Zeitbedingung (*time-out*)
 - Veränderung einer (überwachten) Bedingung (*change event*)
 - Beendigung einer Folge von Aktionen ("namenloses" Ereignis)
- ▶ Spezielle Arten von Ereignissen für einzelne Objekte:
 - Eintreffen einer Nachricht bei einem Objekt
 - » Aufruf einer Operation (Methode)
 - » Signal (ohne zugehörige Methode, wird nur im Zustandsdiagramm behandelt)
 - Erzeugen oder Löschen des Objekts
 - Ereignisse können auch explizit durch Objekte repräsentiert werden. Dann erzeugt ein externes Ereignis ein Ereignisobjekt im Programm
- ▶ Detaillierungsgrad in der Analysephase:
 - Ereignisse allgemeingültiger Art
 - Textuelle Beschreibungen

Zustandsübergänge nach Ereignissen

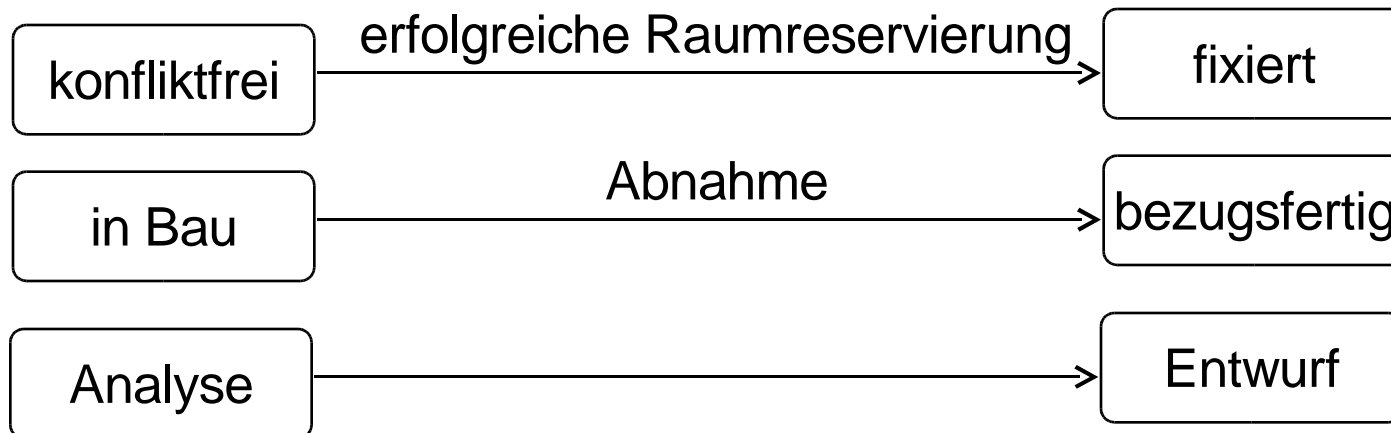
▶ **Definition:**

Ein **Zustandsübergang** von Zustand *A* nach Zustand *B* mit Ereignisnamen *E* besagt, daß im Zustand *A* bei Auftreten eines *E*-Ereignisses der neue Zustand *B* angenommen wird.

▶ **Notation:**



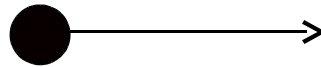
▶ **Beispiele:**



Start- und Endzustand

- ▶ Jedes Zustandsdiagramm sollte einen eindeutigen Startzustand haben. Der Startzustand ist ein "Pseudo-Zustand".

- ▶ **Notation:**

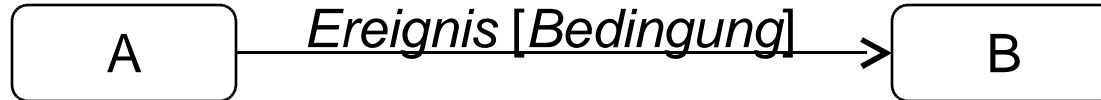


- ▶ Ein Zustandsdiagramm kann einen oder mehrere Endzustände haben. Die Angabe mindestens eines Endzustands ist wünschenswert.

- ▶ **Notation:** ("*bull's eye*")



Bedingte Zustandsübergänge



- ▶ **Definition** Eine **Bedingung** (*guard*) ist eine Boolesche Bedingung, die zusätzlich bei Auftreten des *E*-Ereignisses erfüllt sein muß, damit der beschriebene Übergang eintritt.
- ▶ **Notation:** Eine Bedingung wird in der Analysephase meist noch textuell beschrieben. In formalerer Beschreibung (v.a. im Entwurf) kann eine Bedingung folgende Informationen verwenden:
 - Parameterwerte des Ereignisses
 - Attributwerte und Assoziationsinstanzen (Links) der Objekte
 - ggf. Navigation über Links zu anderen Objekten
- ▶ **Beispiel:**



Aktionen bei Zustandsübergängen



- ▶ **Definition** Eine *Aktion* ist die Beschreibung einer ausführbaren Anweisung, wobei angenommen wird, daß die Dauer der Ausführung vernachlässigbar ist. Aktionen sind nicht unterbrechbar.
Eine Aktion kann auch eine Folge von Einzelaktionen sein.
- ▶ Typische Arten von Aktionen:
 - Lokale Änderung eines Attributwerts
 - Versenden einer Nachricht an ein anderes Objekt (bzw. eine Klasse)
 - Erzeugen oder Löschen eines Objekts
 - Rückgabe eines Ergebniswertes für eine früher empfangene Nachricht
- ▶ Eine Aktion wird in der Analysephase meist textuell beschrieben. In formalerer Beschreibung werden Aktions Sprachen verwendet, die ähnlich zu Programmiersprachen (und ausführbar) sind.

Verwendung von UML-Zustandsmodellen

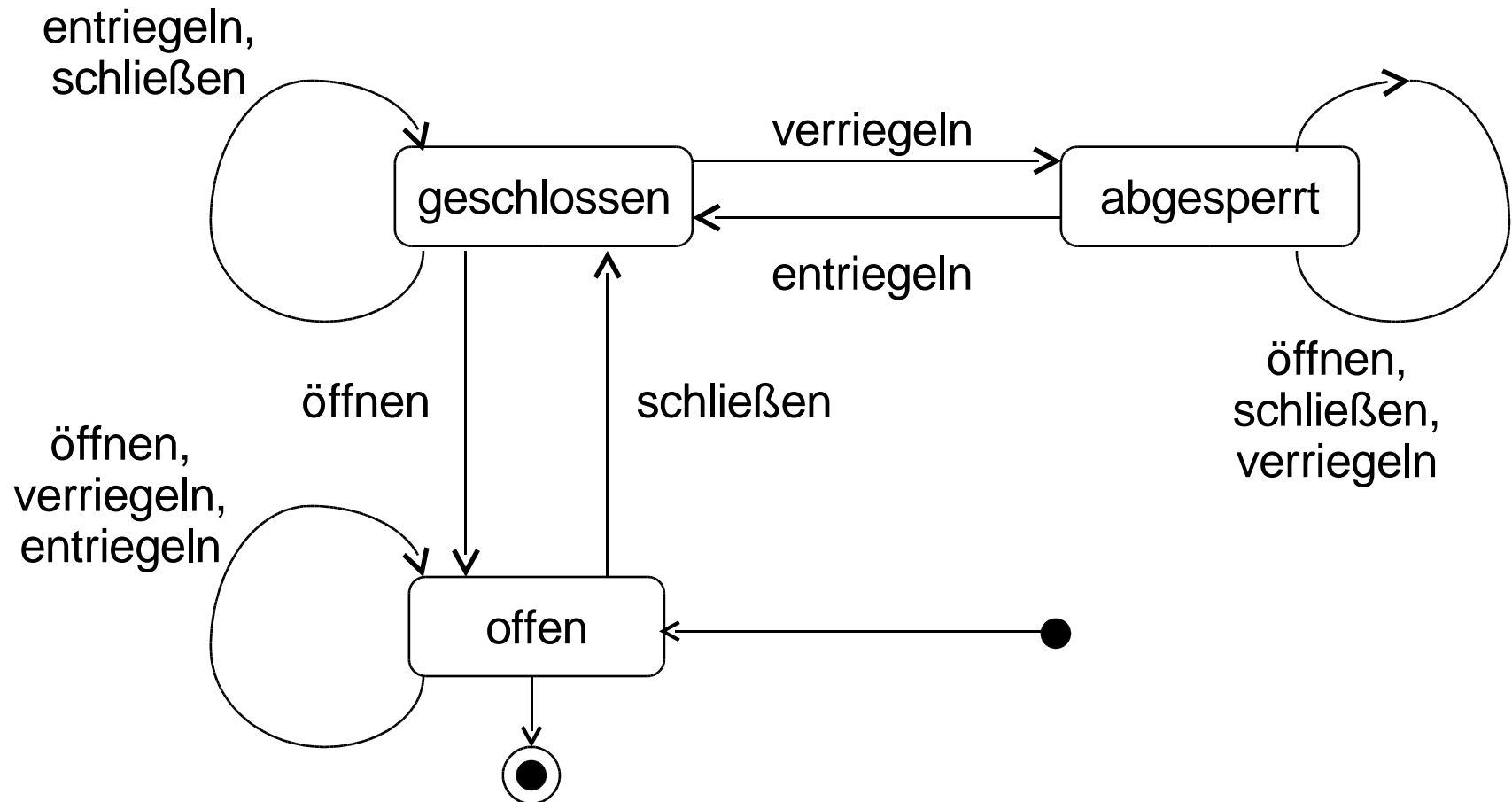
Steuerungs-Maschinen:

- ▶ Beschreiben die *Steuerung* eines Systems der realen Welt
- ▶ Für steuernde Systeme, eingebettete Systeme etc.
- ▶ Ereignisse sind Signale der Umgebung oder anderer Systemteile
- ▶ Reaktion in gegebenem Zustand auf ein bestimmtes Signal:
 - neuer Zustand
 - **ausgelöste Aktion** (wie im Zustandsmodell spezifiziert)
- ▶ Ähnlich zum Konzept von Mealy-Automaten (Transduktor)
- ▶ Zustandsmodelle definieren die *Reaktion* des gesteuerten Systems auf mögliche Ereignisse

Protokoll-Maschinen (Prüfmaschinen für Objekt-Lebenszyklen):

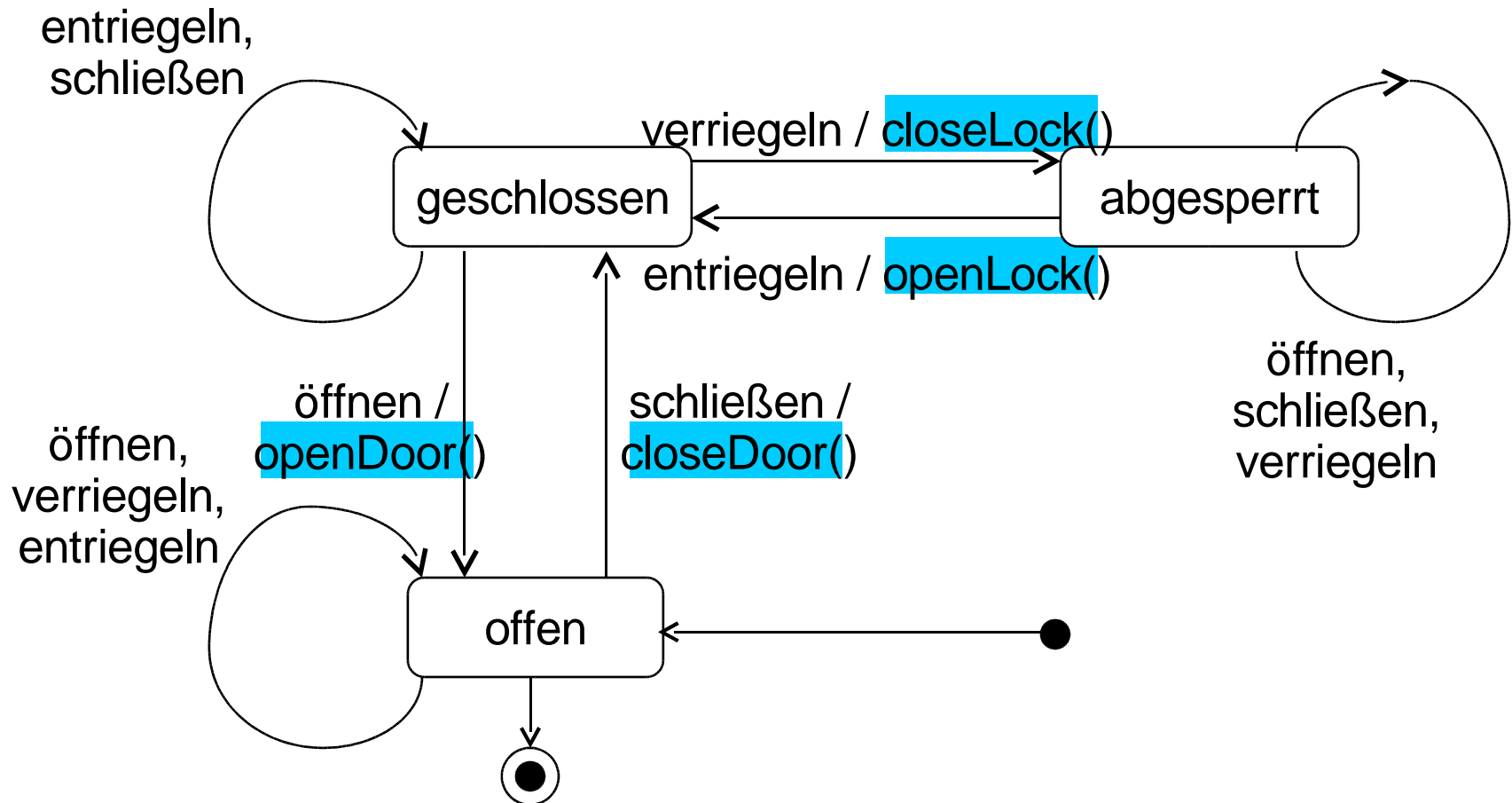
- ▶ Zum *Überprüfen* der Konsistenz eines Systems
- ▶ Für Informationssysteme, Datenbankanwendungen etc.
- ▶ Ereignisse sind Operationsaufrufe
- ▶ Reaktion in gegebenem Zustand auf bestimmten Aufruf:
 - durch Methodenrumpf gegeben (komplex)
 - **Keine Aktionen** im Zustandsmodell!
- ▶ Ähnlich zum Moore-Automaten (Akzeptor)
- ▶ Zustandsmodelle definieren zulässige *Reihenfolgen* von Aufrufen
- ▶ Protokollmaschinen sind “checker”, d.h. *Prüfer*, ob das System bzw. das Objekt einem Zustandsmodell folgt

Beispiel: Protokollmaschine für eine Tür



Eine Protokollmaschine kontrolliert, ob eine Tür einer Zustandsmaschine folgt (beobachtend, prüfend). Frage: Was passiert mit der Protokollmaschine, wenn jemand in die abgeschlossene Tür einbricht?

Beispiel: Steuerungsmaschine für eine Tür einer Behindertentoilette



Eine Türsteuerung empfängt die Signale des Türbenutzers und steuert Servo-Motoren an

Unterschied

- ▶ Steuerungsmaschinen müssen das Wissen über das gesteuerte System *vollständig* repräsentieren (ansonsten gerät das System ausser Kontrolle)
- ▶ Protokollmaschinen können ein *partiell*es Wissen über das geprüfte System repräsentieren (der Rest des Verhaltens wird *nicht* abgeprüft)

Implementierung von Steuerungsmaschinen

- ▶ Zustand wird als Integer-Variable repräsentiert [1..n]
- ▶ Methoden schalten Zustand fort, indem sie Fallanalyse betreiben
 - In jedem Zustand wird ein eingehendes Ereignis untersucht, welchem *Fall* es entspricht, und entsprechend der Zustand fortgeschaltet.

Beispiel: Code zur Steuerung einer Tür (1)

```
class Tuer {
```

"final" bei Attributen: unveränderlich

```
    // Konstante
```

```
    private static final int Z_offen = 0;
    private static final int Z_geschlossen = 1;
    private static final int Z_abgesperrt = 2;
```

```
    // Zustandsvariable
```

```
    private int zustand = Z_offen;
```

```
    public void oeffnen() {
```

```
        switch (zustand) {
```

```
            case Z_offen:
```

```
                break;
```

```
            case Z_geschlossen:
```

```
                zustand = Z_offen;
```

```
                System.out.println("Klack");
```

```
                break;
```

```
            case Z_abgesperrt:
```

```
                break;
```

```
        }
```

```
    }
```

"final" bei Methoden : nicht überschreibbar

Beispiel: Code zur Steuerung einer Tür (2)

```
public void schliessen() {
    switch (zustand) {
        case Z_offen:
            z̄ustand = Z_geschlossen;
            System.out.println("Klick");
            break;
        case Z_geschlossen:
            b̄reak;
        case Z_abgesperrt:
            break;
    }
}

public void verriegeln() {
    switch (zustand) {
        case Z_offen:
            b̄reak;
        case Z_geschlossen:
            z̄ustand = Z_abgesperrt;
            System.out.println("Knirsch");
            break;
        case Z_abgesperrt:
            break;
    }
}
```

Beispiel: Code zur Steuerung einer Tür (3)

```
public void entriegeln() {
    switch (zustand) {
        case Z_offen:
            break;
        case Z_geschlossen:
            break;
        case Z_abgesperrt:
            zustand = Z_geschlossen;
            System.out.println("Knirsch");
            break;
    }
}

class TuerZustaende {
    public static void main(String[] args) {
        Tuer t1 = new Tuer();
        t1.oeffnen();
        t1.schliessen();
        t1.verriegeln();
        t1.entriegeln();
        t1.oeffnen();
        t1.schliessen();
    }
}
```

Steuerungsmaschinen: Zusammenfassung

- ▶ Anwendungsgebiet: Gerätesteuerungen
 - Mikrowelle, Stoppuhr, Thermostat, ...
 - Große Bedeutung z.B. in Automobil- und Luftfahrtindustrie
 - Problem: Verhalten des gesteuerten Geräts muss *regulär* sein, d.h. die Zustandsmenge muss einer reguläre Sprache entsprechen
- ▶ Codegenerierung prinzipiell möglich
 - bei genau definierter Aktionssprache (Aus den Aktionen muss Code generiert werden)
 - Prinzip: Zustandsvariable und Fallunterscheidungs-Kaskaden
- ▶ Praktische Aspekte:
 - Kommunikation: Nachrichten empfangen/versenden
 - Nebenläufigkeit
 - Reaktivität (Akzeptieren von Nachrichten zu beliebigem Zeitpunkt)
 - Realzeitaspekte

Verwendung von UML-Zustandsmodellen

Steuerungs-Maschinen:

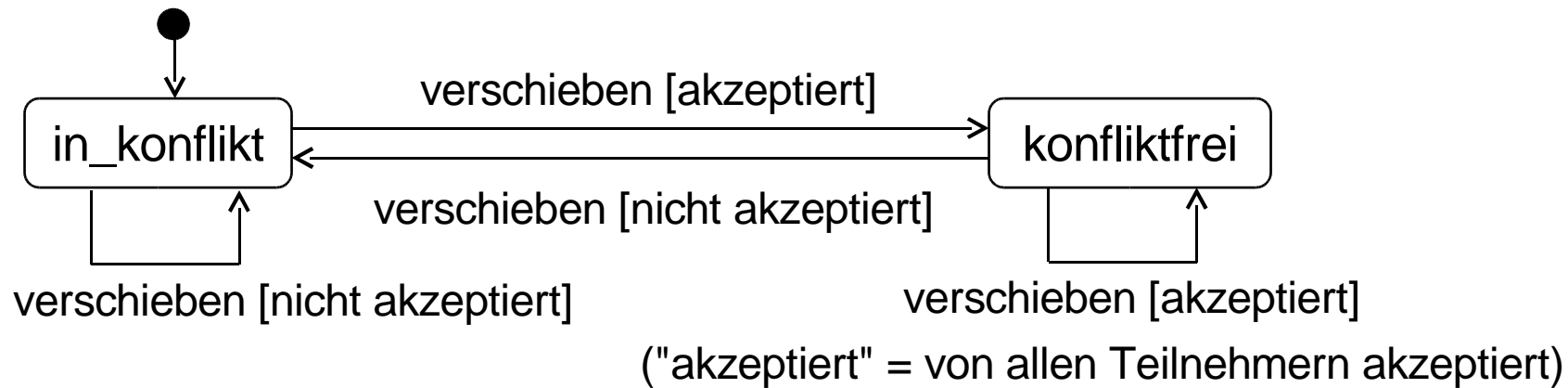
- ▶ Beschreiben die *Steuerung* eines Systems der realen Welt
- ▶ Für steuernde Systeme, eingebettete Systeme etc.
- ▶ Ereignisse sind Signale der Umgebung oder anderer Systemteile
- ▶ Reaktion in gegebenem Zustand auf ein bestimmtes Signal:
 - neuer Zustand
 - **ausgelöste Aktion** (wie im Zustandsmodell spezifiziert)
- ▶ Ähnlich zum Konzept von Mealy-Automaten (Transduktor)
- ▶ Zustandsmodelle definieren die *Reaktion* des gesteuerten Systems auf mögliche Ereignisse

Protokoll-Maschinen (Prüfmaschinen für Objekt-Lebenszyklen):

- ▶ Zum *Überprüfen* der Konsistenz eines Systems
- ▶ Für Informationssysteme, Datenbankanwendungen etc.
- ▶ Ereignisse sind Operationsaufrufe
- ▶ Reaktion in gegebenem Zustand auf bestimmten Aufruf:
 - durch Methodenrumpf gegeben (komplex)
 - **Keine Aktionen** im Zustandsmodell!
- ▶ Ähnlich zum Moore-Automaten (Akzeptor)
- ▶ Zustandsmodelle definieren zulässige *Reihenfolgen* von Aufrufen
- ▶ Protokollmaschinen sind “checker”, d.h. *Prüfer*, ob das System bzw. das Objekt einem Zustandsmodell folgt

Beispiel: Protokollmaschine

- ▶ Folgende Protokollmaschine definiert die zulässigen Aufrufreihenfolgen der Klasse Terminverschiebung:



- Begriff "Protokoll":
 - Kommunikationstechnologie
 - Regelwerk für Nachrichtenaustausch
- Protokollmaschinen in der Softwarespezifikation:
 - **zusätzliche** abstrakte Sicht auf komplexen Code (partielles Wissen)
 - Hilfsmittel zur Einhaltung von Aufrufreihenfolgen

Code für Protokollmaschine (Prinzip)

Variante 1: Impliziter Zustand

```
public Teambesprechung
    (String titel, Hour beginn, int dauer,
     Teammitglied[] teilnehmer) {
    super(titel, beginn, dauer);
    this.teilnahme = teilnehmer;
    if (! abstimmen(beginn, dauer)) {
        System.out.println("Termin bitte verschieben!");
    }
    else {
        for (int i=0; i<teilnahme.length; i++)
            teilnahme[i].teilnahmeSetzen(this);
    }
}
```

Zustandswechsel

Zustand in_konflikt

Zustand konfliktfrei

Information über Zustand jederzeit berechenbar

- hier aus den Werten der Assoziationen und den Datumsangaben

Zustandsinformation gibt zusätzliches Modell, nicht direkt im Code wiederzufinden

Code für Protokollmaschine (Prinzip)

Variante 2: Expliziter Zustand

```
public Teambesprechung
    (String titel, Hour beginn, int dauer,
     Teammitglied[] teilnehmer) {
    int zustand = Z_nicht_abgestimmt;
    super(titel, beginn, dauer);
    this.teilnahme = teilnehmer;
    if (! abstimmen(beginn, dauer)) {
        System.out.println("Termin bitte verschieben!");
        zustand = Z_in_konflikt;
    }
    else {
        for (int i=0; i<teilnahme.length; i++)
            teilnahme[i].teilnahmeSetzen(this);
        zustand = Z_konfliktfrei;
    }
}
```

Explizites
Zustandsattribut

Information über Zustand explizit angegeben
Ablauflogik kann den Zustandswert benutzen (muß aber nicht!)
Keine Aktionen

Protokoll-Maschinen: Zusammenfassung

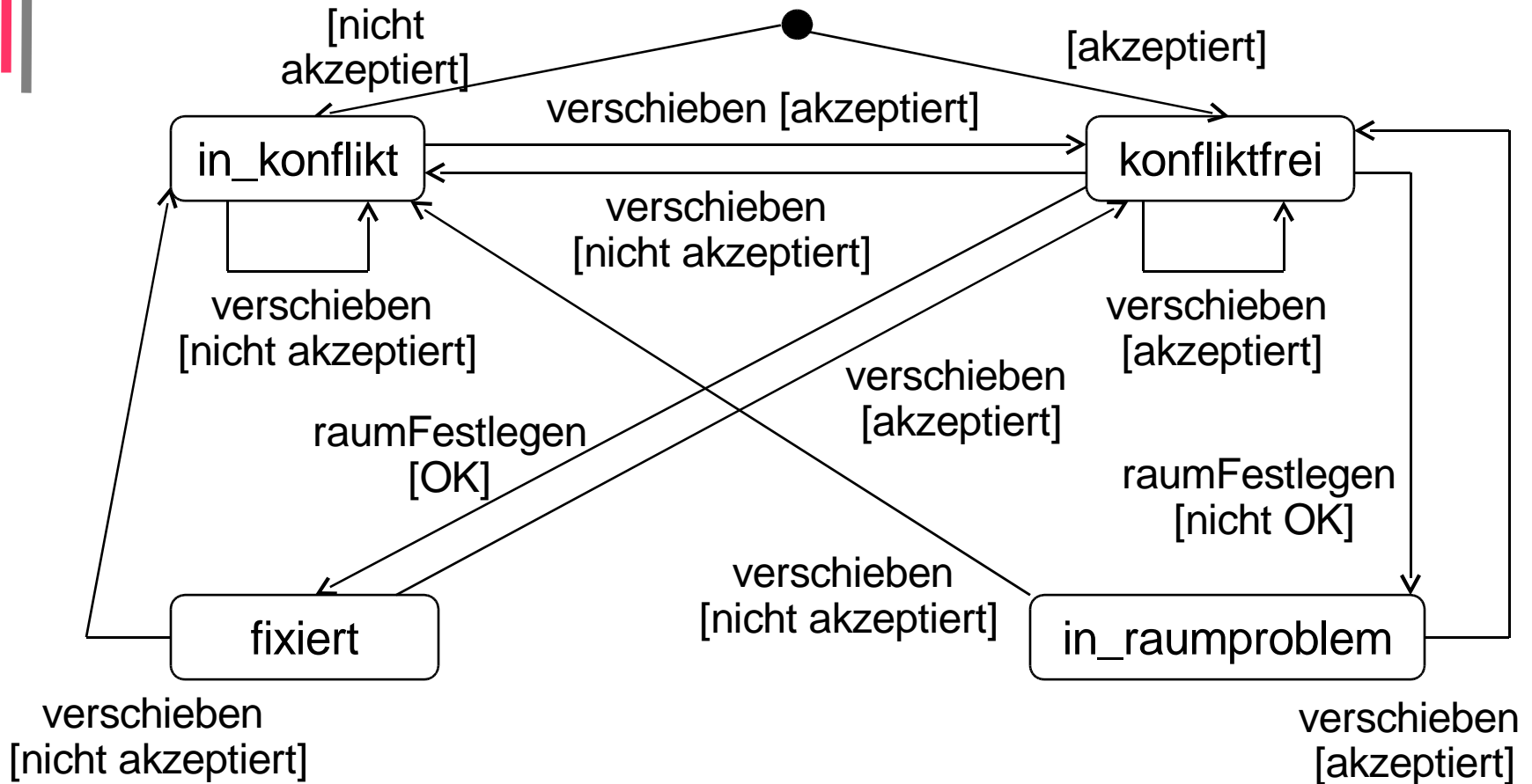
- ▶ Anwendungsgebiet: beliebig
- ▶ Codegenerierung aus Zustandsmodell nicht möglich:
 - Zustandsmodell liefert höchstens Information für Teilaspekte des Codes (zulässige Reihenfolgen)
 - Großteil des Codes unabhängig vom Zustandsmodell
 - Modellierung auch für Code möglich, der *keine* expliziten Zustandskonzepte enthält
 - Aber: Testcode, Prüfcode ist wohl generierbar! (genau wie bei Steuerungsmaschine)
- ▶ Praktische Aspekte:
 - Relevant in der Analyse zur Darstellung von Geschäftsregeln
 - Nützlich für die Implementierung von Klassen mit komplexen Regeln für die Aufrufreihenfolge
 - Hilfreich zur Ableitung von Status-Informationen für Benutzungsschnittstellen
 - Hilfreich zum Definieren sinnvoller Testfälle für Klassen

Vervollständigung von Übergängen

- ▶ Was passiert, wenn **kein** Übergang im aktuellen Zustand für das aktuelle Ereignis angegeben ist?
- ▶ Möglichkeiten:
 - Unzulässig
 - Fehlermeldung (Fehlerzustand)
 - Ausnahmebehandlung
 - Zustand unverändert (impliziter "Schleifen"-Übergang)
 - Warteschlange für Ereignisse
 - Unterspezifikation ("wird später festgelegt")
- ▶ Achtung: Ein vollständiges Zustandsmodell (totale Übergangsfunktion) ist meist sehr umfangreich und unübersichtlich!

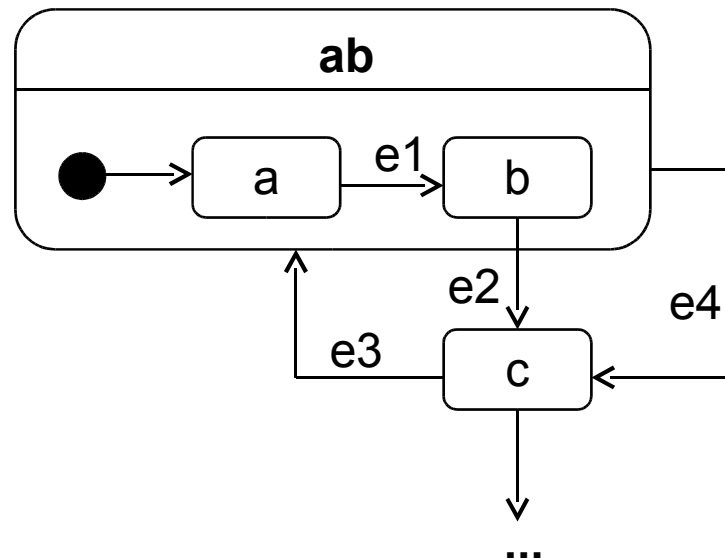
Objektlebenszyklus (Protokollmaschine)

Zustände von Objekten der Klasse "Teambesprechung":

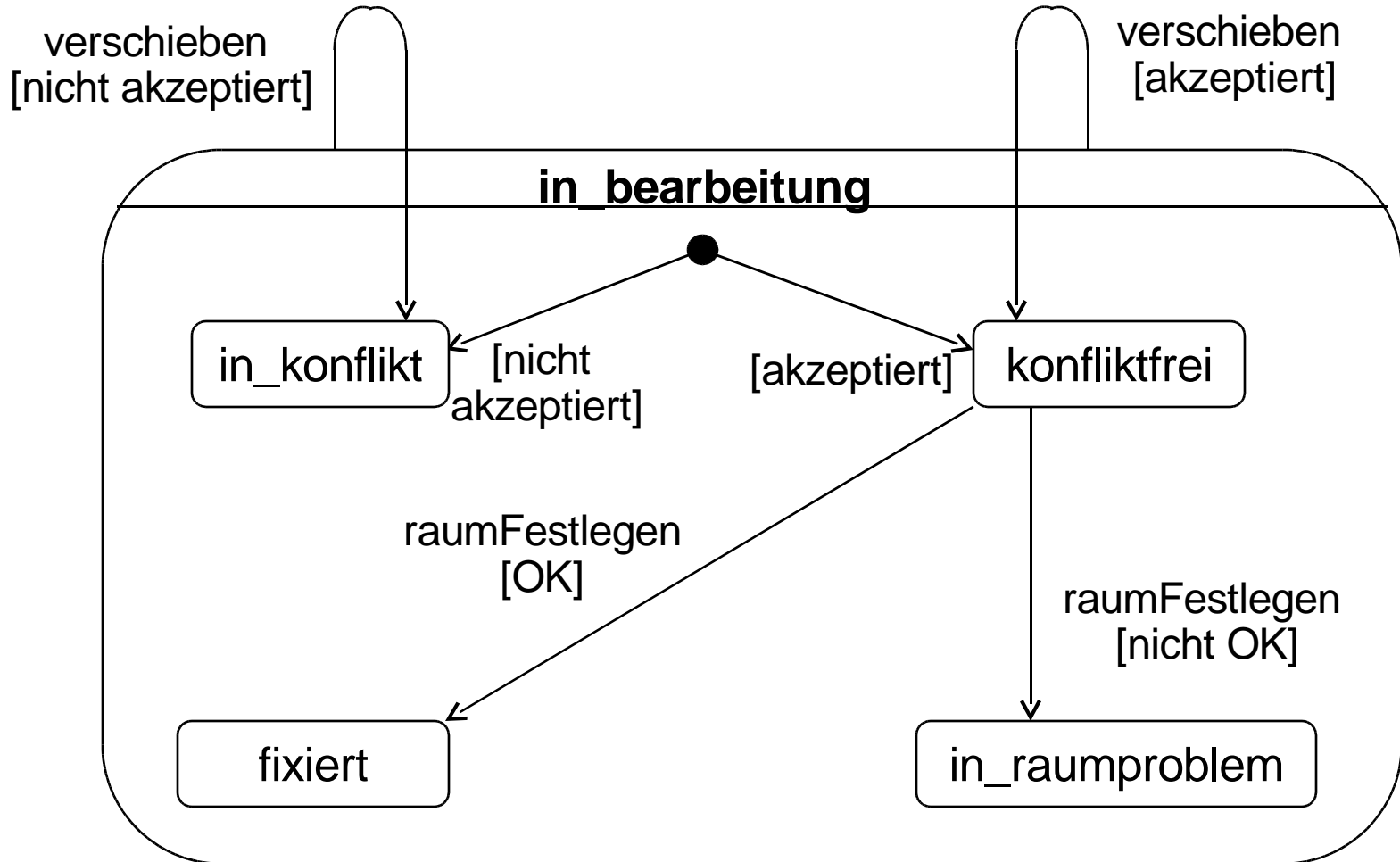


Ober- und Unterzustände

- ▶ Um eine ganze Gruppe von Zuständen einheitlich zu behandeln, können **Oberzustände** eingeführt werden.
 - Ein Zustand in den Oberzustand ist ein Übergang in den Startzustand des enthaltenen Zustandsdiagramms.
 - Ein Zustand aus dem Oberzustand gilt für alle Zustände des enthaltenen Zustandsdiagramms (*Vererbung* von Übergangsverhalten).

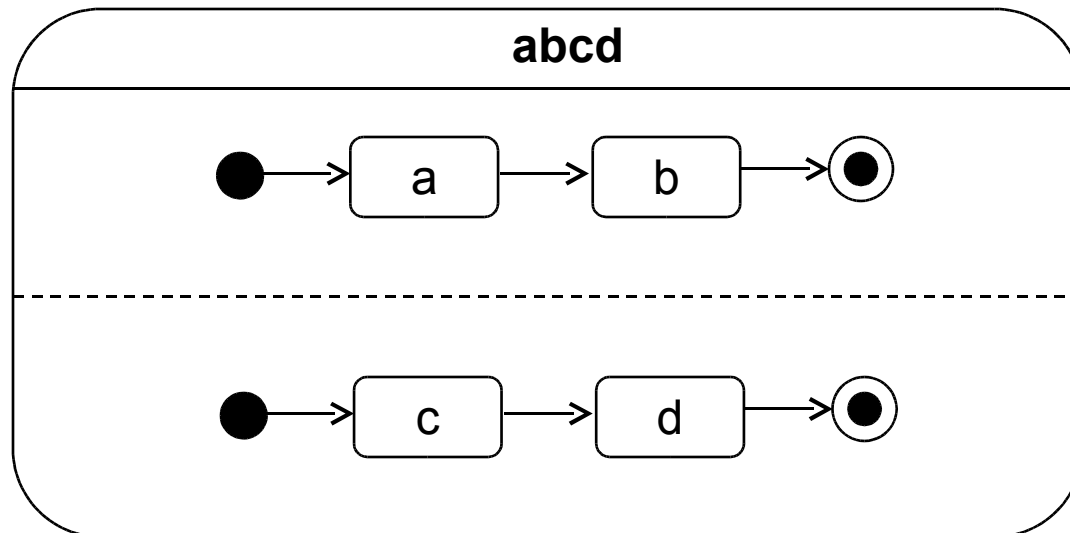


Zustandshierarchie



Nebenläufige Teilzustände

- ▶ Um voneinander zeitlich unabhängige Vorgänge einfach darzustellen, kann ein Zustand in nebenläufige Teilbereiche zerlegt werden (getrennte "Schwimmbahnen").
 - Ein Zustand des Oberzustands ist ein Tupel von Zuständen der Teilbereiche (Schwimmbahnen).



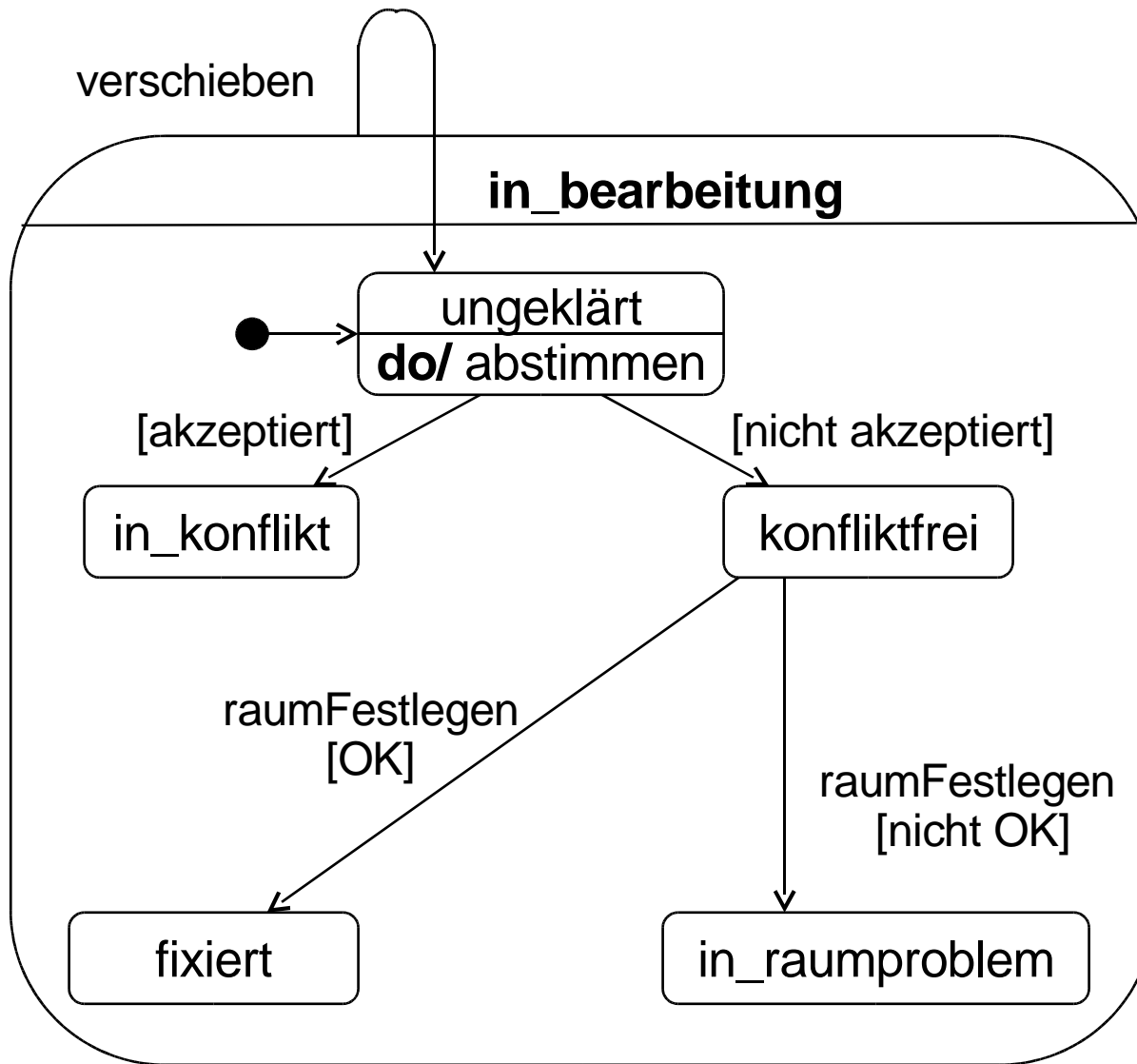
Interne Übergänge

- ▶ **Definition:** Ein *interner Übergang* eines Zustands *S* beschreibt einen Übergang, der stattfindet, während das Objekt im Zustand *S* ist.
- ▶ Es gibt folgende Fälle von internen Übergängen:
 - Eintrittsübergang (*entry transition*)
 - Austrittsübergang (*exit transition*)
 - **Fortlaufende Aktivität** (*do transition*)
 - Unterdiagrammaufruf (*include transition*)
 - Reaktion auf benanntes Ereignis
- ▶ **Notation:**



label = **entry**, **exit**, **do**, **include** oder *Ereignisname*

Aktivitäten

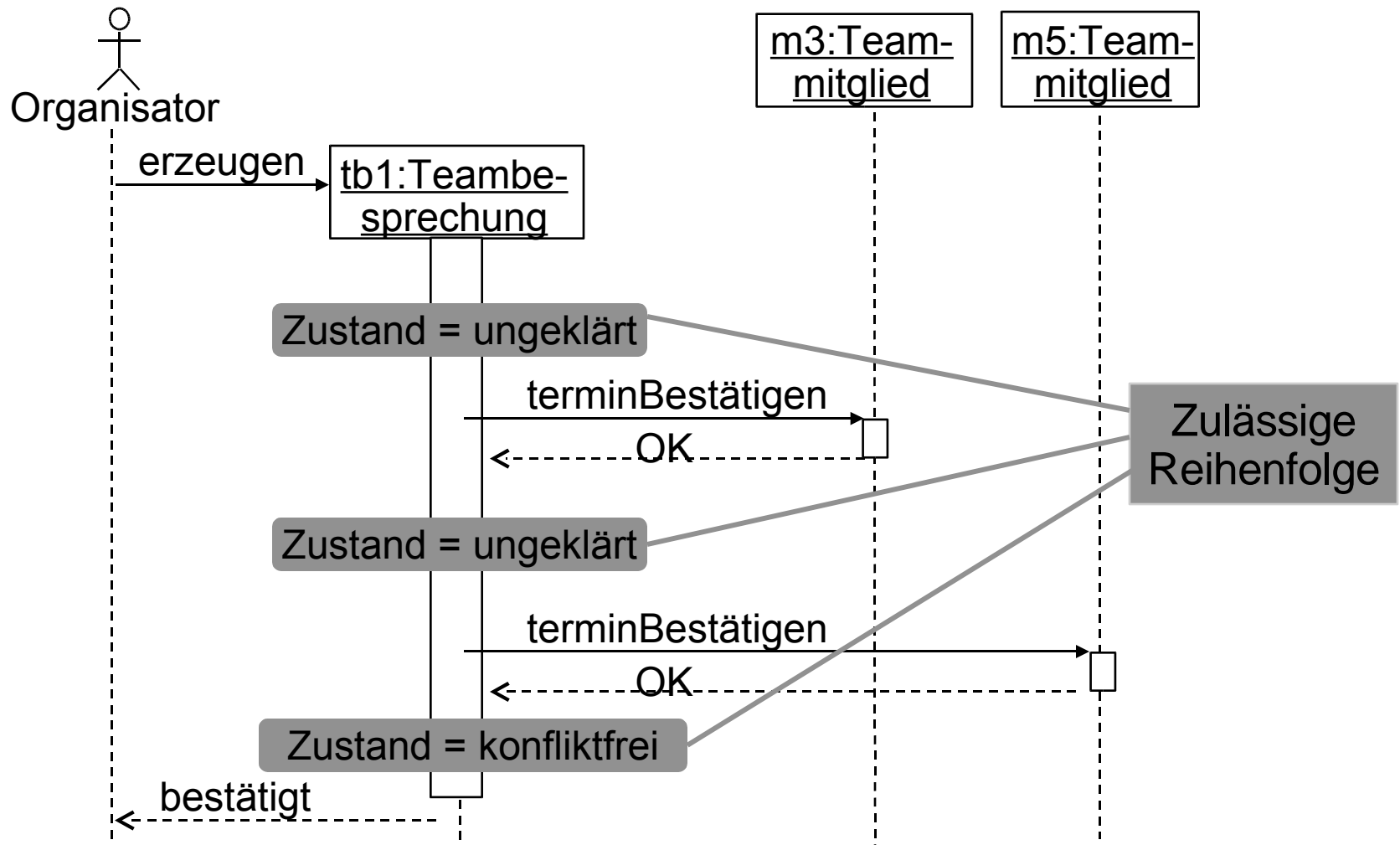


Aktivität
"abstimmen"
=
Abstimmung
mit den
Teammitgliedern
per Operation
"terminBestätigen"

(Modellierbar als
Unterdiagramm *UD*,
d.h. **include UD**
anstelle von **do**)

Zusammenhang:

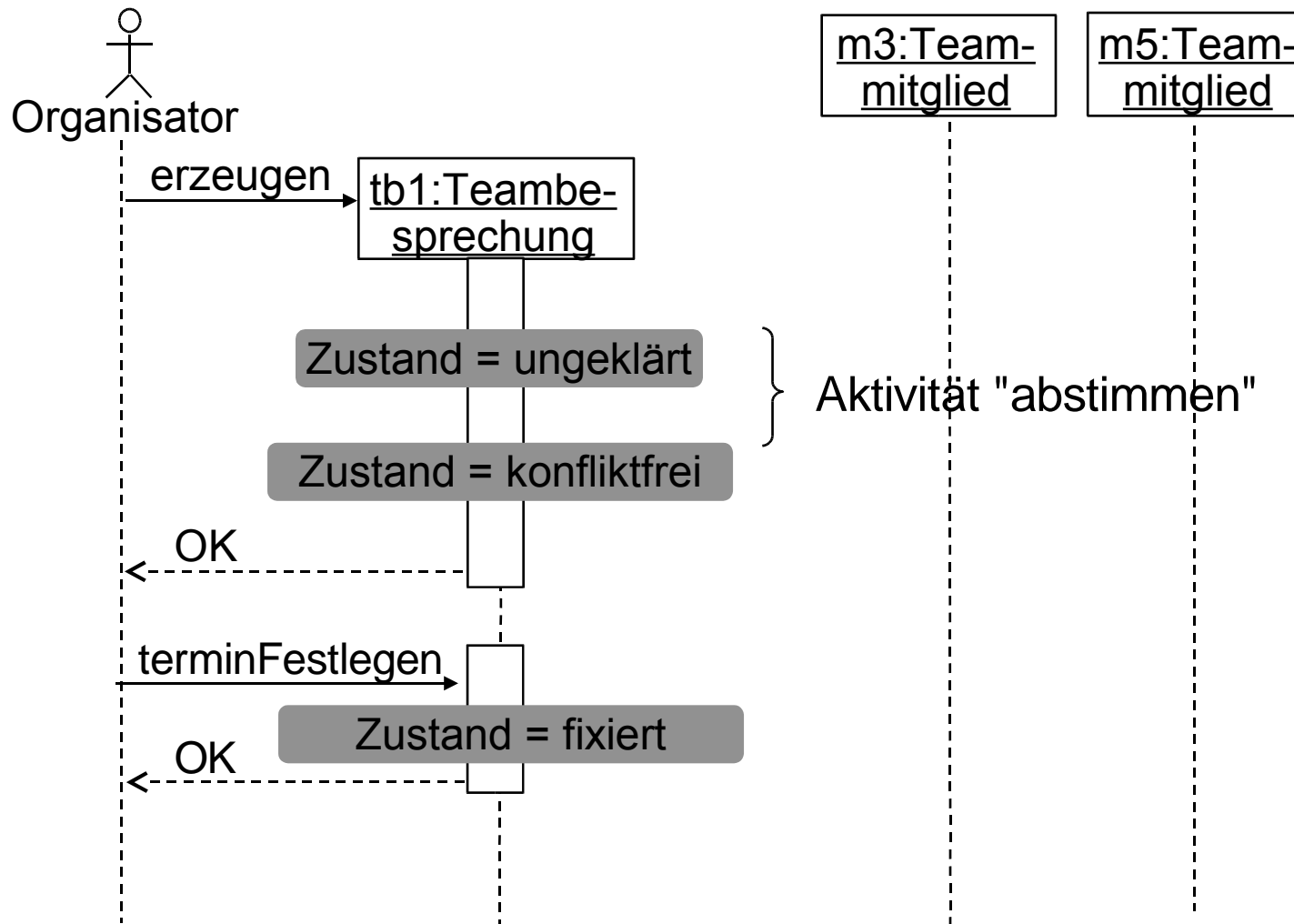
Zustandsdiagramm - Sequenzdiagramm (1)



Jede in den Szenarien auftretende Reihenfolge von Aufrufen muß mit dem Zustandsmodell verträglich sein.

Zusammenhang:

Zustandsdiagramm – Sequenzdiagramm (2)



Sequenzdiagramme und Zustandsdiagramme existieren in verschiedenen Abstraktionsstufen.

Zustandsmodellierung: Zusammenfassung

	Analysephase	Entwurfsphase
<p>Zeitbezogene Anwendungen (Echtzeit, Embedded)</p>	<p>Steuerungsmaschinen</p> <p>Skizzen der Steuerung für Teilsysteme und Gesamtsystem</p>	<p>Detaillierte Angaben, evtl. automatische Codegenerierung</p>
	<p>Große Bedeutung</p>	
<p>Datenbezogene Anwendungen (Informationssysteme, DB-Anwendungen)</p>	<p>Protokollmaschinen</p> <p>Lebenszyklen für zentrale Geschäftsobjekte</p>	<p>Genaue Spezifikation von Aufrufreihenfolge (wenn sinnvoll)</p>
	<p>Relativ geringe Bedeutung</p>	

Textuelle Umschreibungen

Präzise Spezifikation (Parameter etc.)



The End
