

# Kapitel 4.1 – Abbildung von UML-Modellen auf Code

SWT I – Sommersemester 2021

Walter F. Tichy, Christopher Gerking, Tobias Hey



# Literatur

- Diese Vorlesung orientiert sich an Abschnitt 10.4.2 aus

B. Bruegge, A.H. Dutoit, **Object-Oriented Software Engineering: Using UML, Patterns and Java**, Pearson Prentice Hall, 2004.

- **Lesen!**

# Kapitel 4.1.1 – Abbildung von Klassendiagrammen

SWT I – Sommersemester 2021

Walter F. Tichy, Christopher Gerking, Tobias Hey



# Abbildung von Klassen: OO-Sprachen

- Bei OO-Sprachen wird jede UML-Klasse auf eine Klasse in der Programmiersprache abgebildet (einschließlich Attributen und Methoden).

```
class C {    /* Attribute */  
           /* Methoden  */ };
```

# Abbildung von Klassen: Nicht-OO-Sprachen (1)

- Falls keine OO-Sprache zur Verfügung steht, wird eine Klasse auf einen Verbund (record, structure) abgebildet; dieser enthält aber nur die Attribute. Für die Sprache C:

```
struct C { int a1; /* Attribute */ };  
struct C c1, c2, c3; /* Instanzen */
```

- Der Zugriff auf das Attribut **a1** der Instanz **c3** geschieht folgendermaßen:  
**c3.a1**

# Abbildung von Klassen: Nicht-OO-Sprachen (2)

- Alternativ mit Typdefinition in C:

```
typedef struct { int a1; /* Attribute */ } C;  
C c1, c2, c3; /* Instanzen */
```

- Alternativ kann der Speicher für eine Instanz auch zur Laufzeit angefordert werden:

```
C * c4; /* Referenz! */  
c4 = (C*) malloc(sizeof(C));
```

- `malloc()` liefert `void*` (einen untypisierten Zeiger) zurück, daher die Typumwandlung (engl. *cast*) nicht vergessen!
- Bei Referenzen geschieht der Attributzugriff mit Operator `->`:  
`c4->a1`

# Abbildung von Klassen: Nicht-OO-Sprachen (3)

- Methoden werden auf Unterprogramme abgebildet, die den Verbundtyp als zusätzlichen Referenz-Parameter enthalten. Methode  
`m(parameter) { ...attribut... }`  
wird zu freistehender Funktion  
`m(C * objekt, parameter) { ...objekt->attribut... }`
- Der Aufruf `c4.m(parameter)`  
wird zu `m(c4, parameter)`
- Vererbung wird simuliert, in dem die Attribute der Oberklasse(n) dem Verbund hinzugefügt werden.
- Polymorphie kann man, falls nötig, über Funktionszeiger und Typumwandlung des ersten Parameters simulieren.

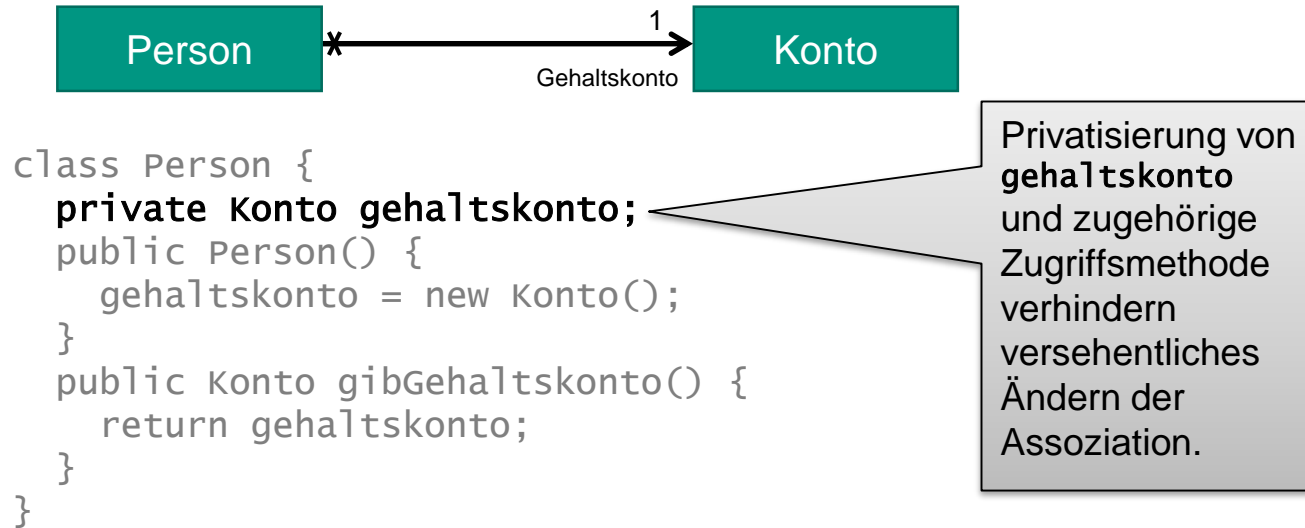
# Abbildung von Assoziationen

- Selbst OO-Sprachen bieten keine Assoziationen, nur Referenzen. Letztere benutzt man, um die verschiedenen Arten von Assoziationen (uni-/bidirektional, Kardinalitäten) zu implementieren.



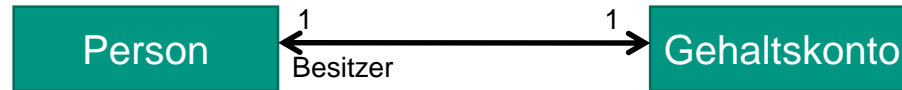
# Unidirektionale Eins-zu-Eins-Assoziation

- Benutze in der Klasse, in der zur anderen navigiert wird, eine Instanzvariable, die eine Referenz zur anderen Klasse enthält.



# Bidirektionale Eins-zu-Eins-Assoziation

- Benutze in beiden Klassen eine Instanzvariable, die eine Referenz zur anderen enthält.



- Achtung: echte 1:1-Beziehung bedeutet gegenseitige Abhängigkeit!

# Bidirektionale Eins-zu-Eins Assoziation

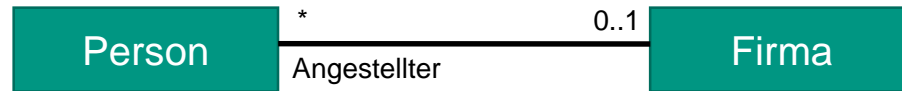
```
class Person {  
    private Konto gehaltskonto;  
    public Person() {  
        gehaltskonto = new Konto(this);  
    }  
    public Konto gibGehaltskonto() {  
        return gehaltskonto;  
    }  
}
```

Hier Lösung für initiale Konsistenz: Sorge dafür, dass keine null-Werte bei der Initialisierung entstehen können

```
class Konto {  
    private Person besitzer;  
    public Konto(Person besitzer) {  
        this.besitzer = besitzer;  
    }  
    public Person gibBesitzer() {  
        return besitzer;  
    }  
}
```

# 1:N-Assoziationen

- Vielfache Referenzen müssen durch mengenwertige Instanzvariablen ausgedrückt werden.



# 1:N-Assoziationen

```
class Person {  
    private Firma firma;  
    public Person() {}  
    public void setzeFirma(Firma f) {  
        if (f == firma) return;  
        firma = f;  
        firma.stelleEin(this);  
    }  
    public Firma gibFirma() { return firma; }  
}
```

Endlosschleife  
verhindern

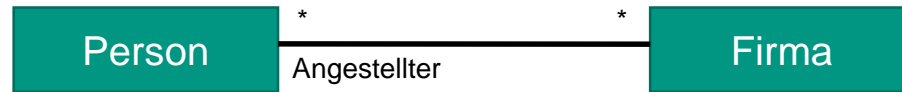
```
class Firma {  
    private Collection<Person> angestellte = new ArrayList<Person>();  
    public Firma() {}  
    public void stelleEin(Person p) {  
        if (angestellte.contains(p)) return;  
        angestellte.add(p);  
        p.setzeFirma(this);  
    }  
}
```

Referenz  
hinzufügen

Dieser Code hat noch einen Fehler! Welchen?

# M:N-Assoziation

- Bei N:M-Assoziationen müssen zwei mengenwertige Attribute verwendet werden.



```
class Person {
    private Collection<Firma> arbeitgeber;
    ...
}
class Firma {
    private Collection<Person> angestellte;
    ...
}
```

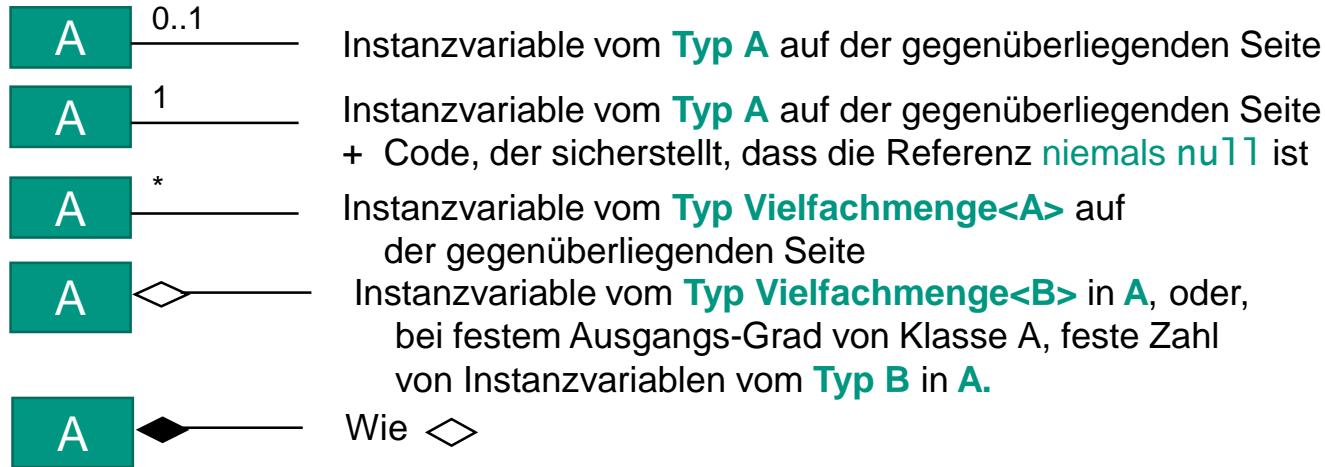
# M:N-Assoziation

```
class Person {  
    ...  
    public void fügehinzuArbeitgeber(Firma f) {  
        if (arbeitgeber.contains(f)) return;  
        arbeitgeber.add(f);  
        f.stelleEin(this);  
    }  
    ...  
}
```

Endlosschleife  
verhindern

```
class Firma {  
    ...  
    public void stelleEin(Person p) {  
        if (angestellte.contains(p)) return;  
        angestellte.add(p);  
        p.fügehinzuArbeitgeber(this);  
    }  
    ...  
}
```

# Zusammenfassung



Grundsätzlich gilt für den Assoziationen manipulierenden Code:

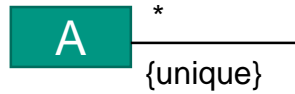
- Assoziationen als „Ganzes“ betrachten
- Immer beide Seiten anpassen
- „Transaktional“ denken
- Synchronisierung in nebenläufigen Programmen beachten (hier weggelassen!)



# Sonderfälle



Instanzvariable vom **Typ Liste<A>** (mit Reihenfolge) auf der Gegenseite, also z.B. ArrayList<A> oder Vector<A>



Instanzvariable vom **Typ Menge<A>** auf der Gegenseite, also z.B. HashSet<A> oder TreeSet<A>



Instanzvariable vom **Typ Abbildung<Objekt, B>** in **A**, also z.B. HashMap<Object, B>  
+ Methoden für Zugriff über den Qualifizierer

# Qualifizierte Assoziation

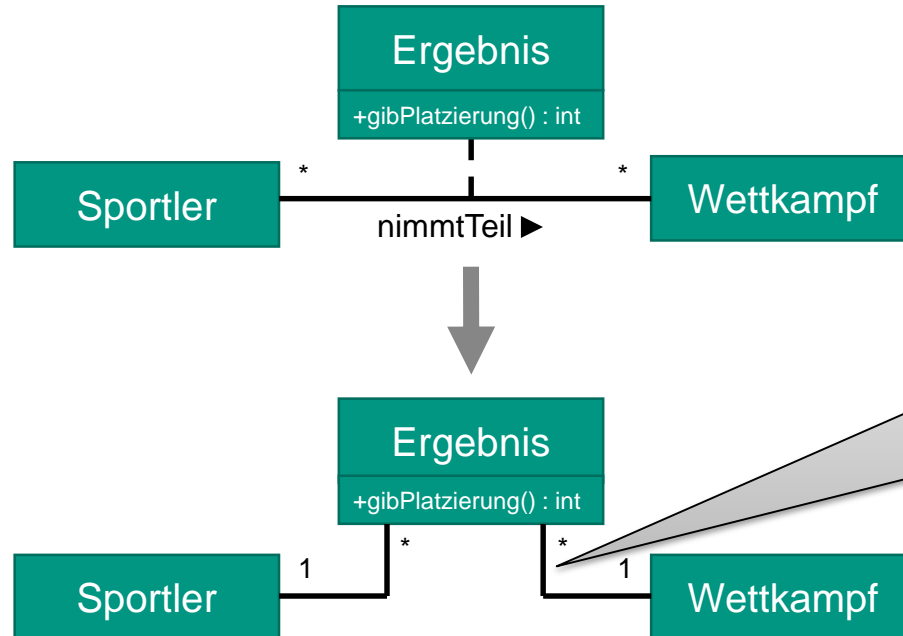


```
class Bibliothek {
    private Map<String,Buch> bücher;
    public void fügehinzuBuch(String signatur, Buch b) {
        if (!bücher.containsKey(signatur)) {
            bücher.put(signatur,b);
            b.fügehinzuBibliothek(signatur,this);
        }
    }
    public Buch gibBuch(String signatur) {
        return bücher.get(signatur);
    }
    ...
}
```

Zugriff über  
Qualifizierer

# Assoziationsklassen

## ■ Realisierung durch Modelltransformation:



Code muss sicherstellen, dass sich die Realisierung der *zwei Assoziationen* so verhalten, als würde es sich um *eine Assoziation* mit zusätzlichen Attributen handeln

# Kapitel 4.1.2 – Abbildung und Implementierung von Zustandsautomaten

SWT I – Sommersemester 2021

Walter F. Tichy, Christopher Gerking, Tobias Hey



# Speicherung des Zustands eines Objektes

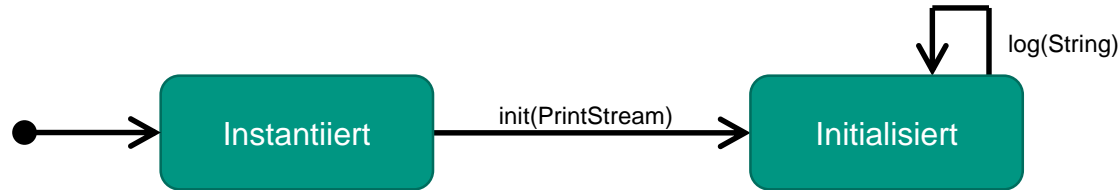
## ■ Implizite Speicherung

- Der Zustand des Objektes kann aus den Attributwerten eines Exemplars „berechnet“ werden
- Keine dedizierten Instanzvariablen nötig, der Zustand muss aber jedes Mal neu berechnet werden
- Zustandsübergangsfunktion ist implizit

## ■ Explizite Speicherung

- Der Zustand eines Objektes wird in dedizierten Instanzvariablen gespeichert und kann daher einfach gelesen und neu gesetzt werden
- Die Zustandsübergangsfunktion muss ebenfalls explizit angegeben werden

# Beispiel für implizite Speicherung des Zustands



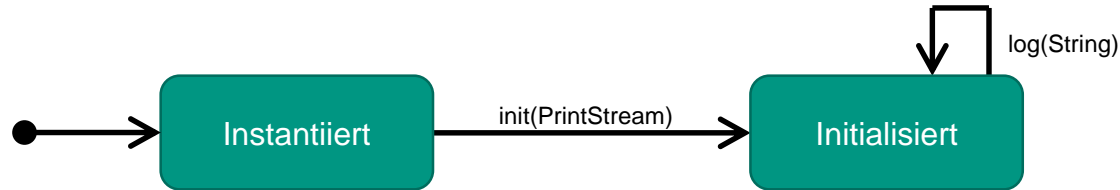
```

public class Logger {
    private PrintStream log;
    public void init(PrintStream dst) {
        log = dst;
    }
    public void log(String msg) {
        if (log == null) throw new IllegalStateException();
        log.append(msg);
    }
}
    
```

„Berechnung“ des aktuellen Zustands aus den Attributwerten

aus `java.lang`

# Beispiel für explizite Speicherung des Zustands



```

public class Logger {
    private enum Zustand { instanziiert, initialisiert };
    private Zustand zustand = Zustand.instanziiert;
    private PrintStream log;
    public void init(PrintStream dst) {
        log = dst;
        zustand = Zustand.initialisiert;
    }
    public void log(String msg) {
        if (zustand != Zustand.initialisiert)
            throw new IllegalStateException();
        log.append(msg);
    }
}
    
```

Explizite Speicherung  
des Zustands

Zustand kann direkt  
ausgelesen werden

# Vergleich implizite/explicite Speicherung des Zustands

- Die implizite Speicherung spart **Speicherplatz**, die explizite (potentiell) **Rechenzeit**.
- Die implizite Speicherung ist potentiell **komplizierter** (trickreicher), die explizite **umfangreicher**.
- Die explizite Speicherung ist immer **offensichtlich** und wird daher bei Änderungen wahrscheinlicher berücksichtigt.
- Die implizite Speicherung ist **nicht immer möglich**, die explizite schon.



# Alternativen der Implementierung expliziter Speicherung der Zustände

## ■ Eingebettet

- Jede Methode „kennt“ den kompletten Automaten
  - Sie führt ihre Aufgabe kontextsensitiv (= dem aktuellen Zustand entsprechend) durch und
  - Nimmt die Zustandsübergänge selbst vor
  - Implementiert als eine große Fallunterscheidung pro Methode (switch- oder if-Anweisungen)
- Vorteil: kompakter, schneller

## ■ Ausgelagert

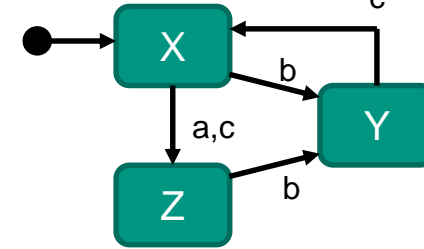
- Die Methode läuft im aktuellen Zustand, der ein Objekt ist.
- Der Code zur Reaktion und Änderung der Zustände befindet sich in dedizierten (ggf. automatisch erzeugten) Klassen
- Die Äste der Fallunterscheidung sind auf gleichnamige Methoden verteilt (a la Strategie-Muster)
- Vorteil: flexibler, bei komplexen Automaten übersichtlicher

# Beispiel für eingebettete explizite Speicherung

```

public class XYZ {
    private enum Zustand{ X, Y, Z };
    private Zustand zustand = Zustand.X;
    public void a() {
        if (zustand == Zustand.Y)
            throw new IllegalStateException();
        if (zustand == Zustand.Z)
            throw new IllegalStateException();
        // ... mach was ...
        zustand = Zustand.Z;
    }
    public void b() {
        if (zustand == Zustand.Y) throw new IllegalStateException();
        // ... mach was ...
        zustand = Zustand.Y;
    }
    public void c() {
        if (zustand == Zustand.Z) throw new IllegalStateException();
        // ... mach was ...
        if (zustand == Zustand.X) zustand = Zustand.Z;
        if (zustand == Zustand.Y) zustand = Zustand.X;
    }
}

```



Nächster  
Zustand: Z

Die Methode a() darf  
im Zustand Y und Z  
nicht aufgerufen werden

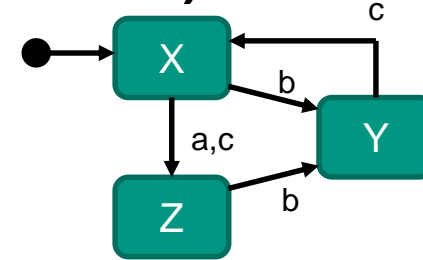
Was ist, wenn hier etwas passieren  
soll, das vom gegenwärtigen  
Zustand abhängig ist?

Der nächste Zustand ist  
vom vorhergehenden  
abhängig

Der „mach was“-Teil  
muss also auch noch  
in die if-Blöcke rein!

# Beispiel für eingebettete explizite Speicherung (diesmal ordentlich) (alternativ auch mit switch)

```
public void a() {  
    if (zustand==Zustand.X) {  
        // ... mach was ...  
        zustand = Zustand.Z; return;  
    }  
    if (zustand==Zustand.Y)  
        throw new IllegalStateException();  
    if (zustand==Zustand.Z)  
        throw new IllegalStateException();  
}  
public void b() {  
    if (zustand==Zustand.X) {  
        // ... mach was ...  
        zustand = Zustand.Y; return;  
    }  
    if (zustand==Zustand.Y)  
        throw new IllegalStateException();  
    if (zustand==Zustand.Z) {  
        // ... mach was anderes ...  
        zustand = Zustand.Y; return;  
    }  
}
```



```
public void c() {  
    if (zustand==Zustand.X) {  
        // ... mach was ...  
        Zustand = Zustand.Z; return;  
    }  
    if (zustand==Zustand.Y) {  
        // ... mach was anderes ...  
        Zustand = Zustand.X; return;  
    }  
    if (zustand==Zustand.Z)  
        throw new IllegalStateException();  
}
```

... und die entry- und exit-Aktionen müssen da eigentlich auch noch überall rein!

# Ausgelagerte explizite Speicherung (engl. *state pattern*)

## ■ Idee:

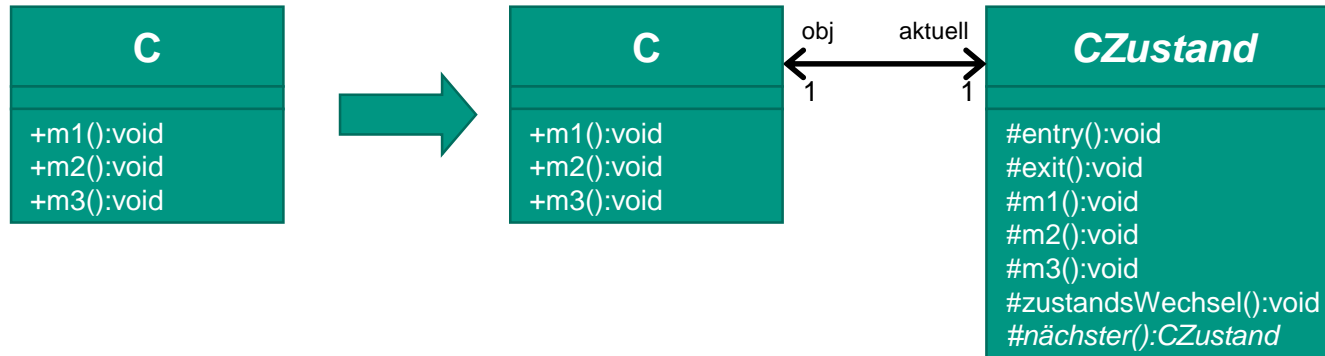
- Das eigentliche Objekt weiß nicht was genau in welchem Zustand zu tun ist.
- Es kennt nur seinen Zustand
- Und **delegiert** das, was zu tun ist, wenn eine Botschaft eintrifft, an den jeweiligen Zustand.

## ■ Vorteil:

- Die Kontextsensitivität (= Zustandsabhängigkeit) der Methoden braucht nicht mehr explizit verwaltet zu werden, statt dessen wird dynamische Polymorphie verwendet
- Die Implementierungsarbeit wird auf verschiedene Klassen (= verschiedene Dateien in Java) aufgeteilt
  - Bessere Parallelisierbarkeit der Implementierungsarbeit
  - „separation of concerns“

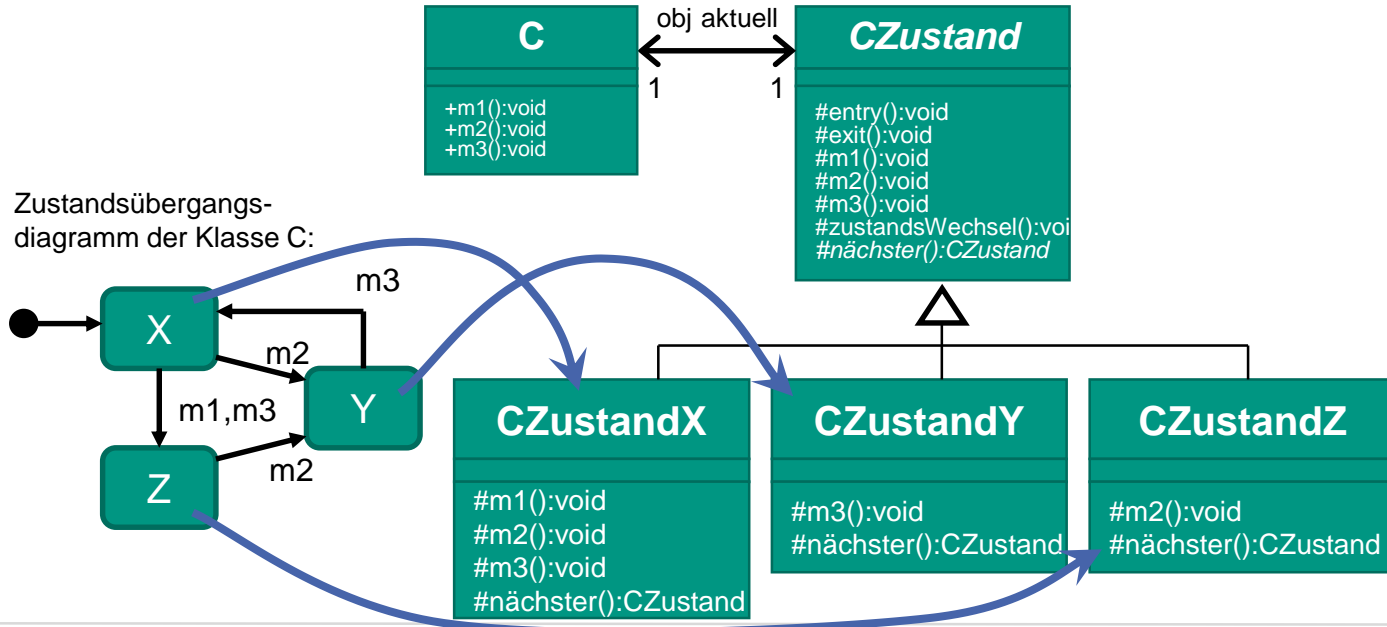
# Vorgehensweise

- Zu jeder Klasse **C**, für deren Verhalten ein endlicher Automat implementiert werden soll, erzeuge eine abstrakte Oberklasse **CZustand**, die für alle Zustände von **C** steht:



# Vorgehensweise

- Jeder Zustand, den die Klasse C einnehmen kann wird durch einen eigenen Typ repräsentiert



# Vorgehensweise: Delegation

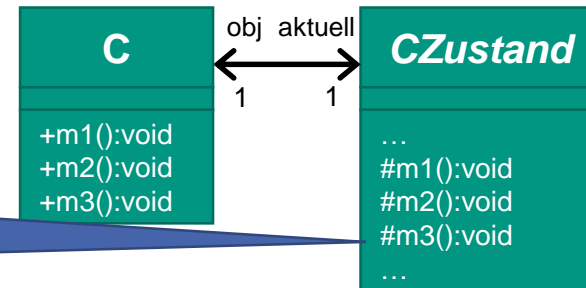
- Die Implementierung der Methoden `m1`, `m2` und `m3` in `C` wird delegiert:

```
class C {
    ...
    protected CZustand aktuell;
    ...
    public void m1() {
        aktuell.m1(); // delegiere
        // Zustandsänderung
        // fehlt noch
    }
    ...
}
```

Aufruf funktioniert nur wenn  
protected auch package-private ist  
(z.B. wie in Java) und C im  
gleichen Paket wie CZustand liegt

Alternative: Zustände  
sind innere Klassen von  
C

In UML sind protected Elemente  
nur aus der Klasse selbst und  
Unterklassen zugreifbar



# Implementierung der Methoden der abstrakten Klasse `CZustand`

- Standardverhalten: „Tue nichts beim `entry`- und `exit`-Ereignis“
  - ein konkreter Zustand braucht nur da Implementierungen angeben, wo er das auch will.

```
protected void entry() { //leer }  
protected void exit()  { //leer }
```



# Implementierung der Methoden der abstrakten Klasse `CZustand`

- Standardverhalten: „In diesem Zustand ist das Aufrufen dieser Methode unzulässig“
  - ein konkreter Zustand braucht nur da Implementierungen angeben, wo er das auch will.

```
protected void m1() {  
    throw new IllegalStateException();  
}  
protected void m2() {  
    throw new IllegalStateException();  
}  
protected void m3() {  
    throw new IllegalStateException();  
}
```

# Implementierung der Methoden der abstrakten Klasse `CZustand`

- Die abstrakte Methode `nächster()` ist eine *Einschubmethode*, die angibt, welcher Zustand der nächste Zustand wäre. Sie nimmt keinen Zustandswechsel vor und wird als Einschubmethode auch nicht von außen aufgerufen!
- Die Methode ist *abstrakt*, so dass jeder Zustand seine eigene Implementierung angeben muss.

```
protected abstract CZustand nächster();
```

# Implementierung der Methoden der abstrakten Klasse `CZustand`

- Die Methode `zustandswechsel()` ist eine Schablonenmethode, die ggf. einen Zustandswechsel vornimmt. Sie muss nach jedem Aufruf einer Methode von `c` aufgerufen werden:

```
protected final void zustandswechsel() {  
    CZustand n = nächster();  
    this.exit();  
    n.entry();  
    obj.aktuell = n;  
}
```



Aufruf der  
Einschubmethode

# Vorgehensweise

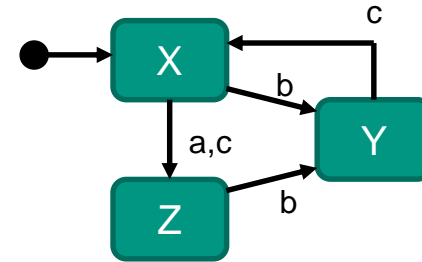
- Die Implementierung der Methoden `m1`, `m2` und `m3` in `C` sieht also folgendermaßen aus:

```
class C {  
    ...  
    protected CZustand aktuell;  
    ...  
    public void m1() {  
        aktuell.m1();  
        aktuell.zustandswechsel();  
    }  
    ...  
}
```

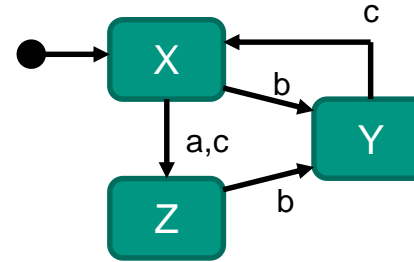
# Beispiel

- Sei für die Klasse `M` unten stehender Automat definiert.
- Dann ist

```
class M {  
    protected MZustand aktuell = new MZustandX(this);  
    public void a() {  
        aktuell.a();  
        aktuell.zustandswechsel();  
    }  
    public void b() {  
        aktuell.b();  
        aktuell.zustandswechsel();  
    }  
    public void c() {  
        aktuell.c();  
        aktuell.zustandswechsel();  
    }  
}
```



# Beispiel



```
class MZustand {
    protected M obj;
    protected MZustand(M m) { obj = m; }
    ... // wie beschrieben
}
class MZustandX extends MZustand {
    private MZustand z;
    protected MZustandX(M m) { super(m); }
    protected void a() {
        ... // mach was
        z = new MZustandZ(obj);
    }
    protected void b() {
        ... // mach was
        z = new MZustandY(obj);
    }
    protected void c() {
        ... // mach was
        z = new MZustandZ(obj);
    }
    protected MZustand nächster() {
        return z;
    }
}
```

```
class MZustandY extends MZustand {
    prot. MZustandY(M m) { super(m); }
    protected void c() {
        ... // mach was
    }
    protected MZustand nächster() {
        return new MZustandX(obj);
    }
}
class MZustandZ extends MZustand {
    prot. MZustandZ(M m) { super(m); }
    protected void b() {
        ... // mach was
    }
    protected MZustand nächster() {
        return new MZustandY(obj);
    }
}
```

Hier mit new. Es könnten auch Einzelstücke verwendet werden.