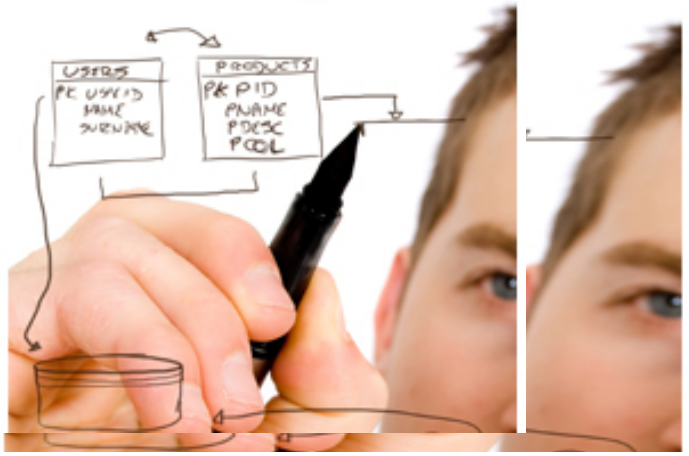
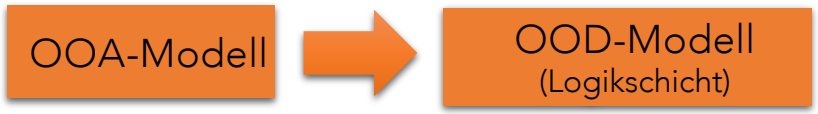
Fachhochschule
Dortmund
University of Applied Sciences and Arts

Softwaretechnik 2



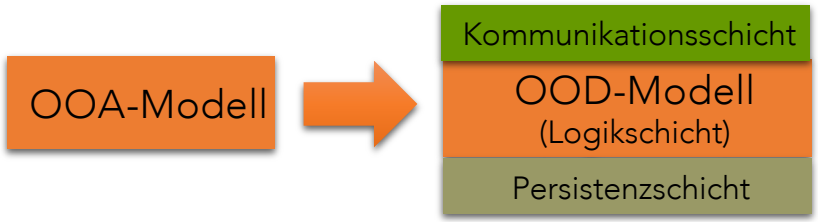
OOD Businesslogikschicht

Struktur



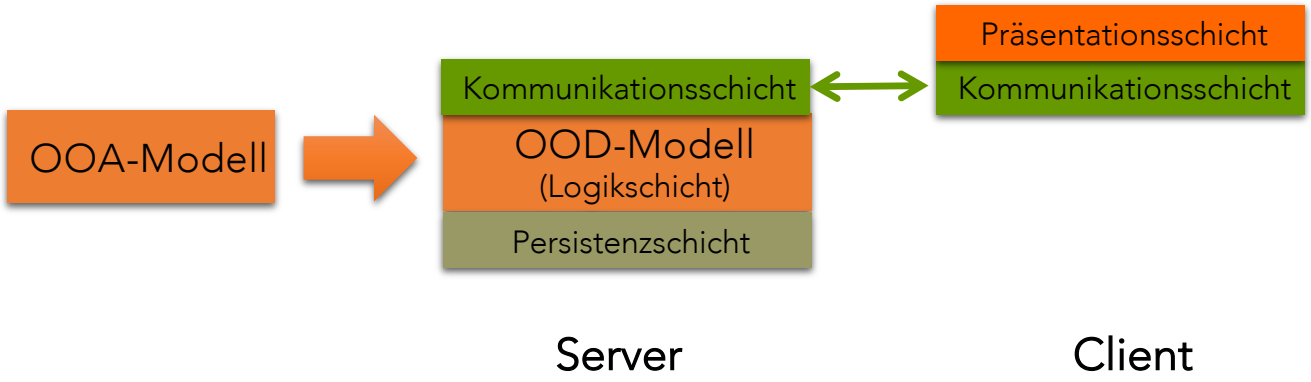
Server

Struktur

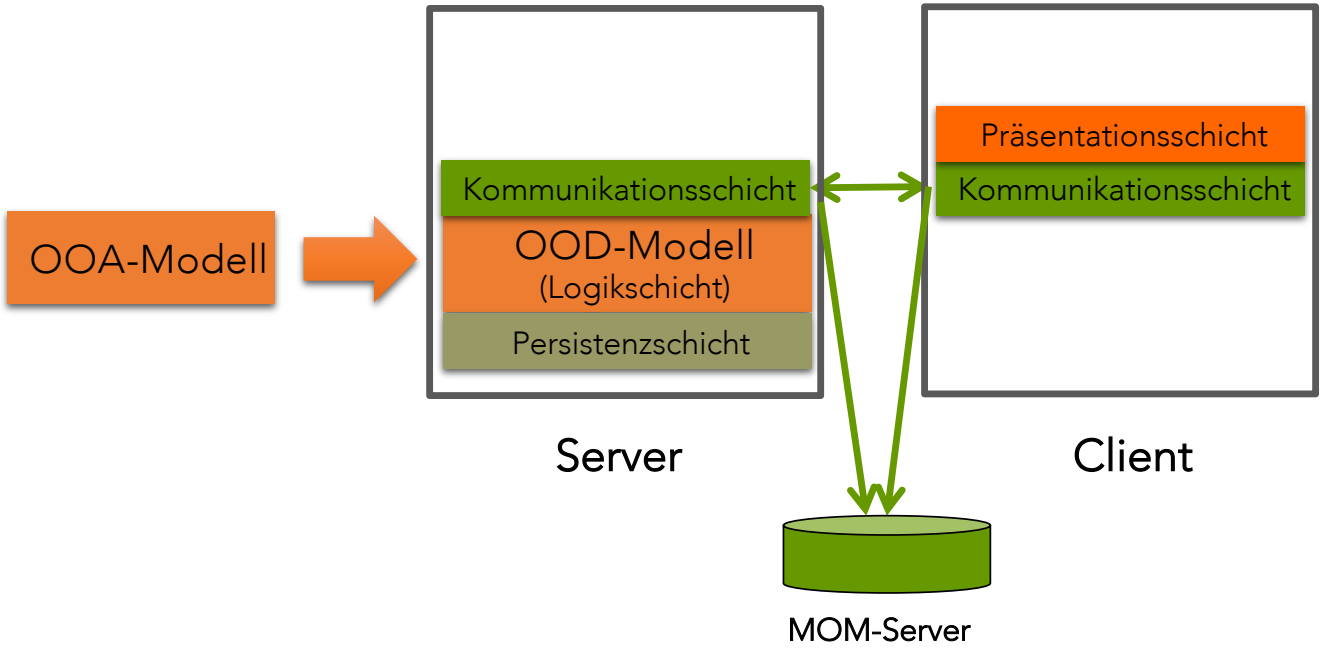


Server

Struktur



Struktur



Fachhochschule
Dortmund
University of Applied Sciences and Arts

Businesslogik

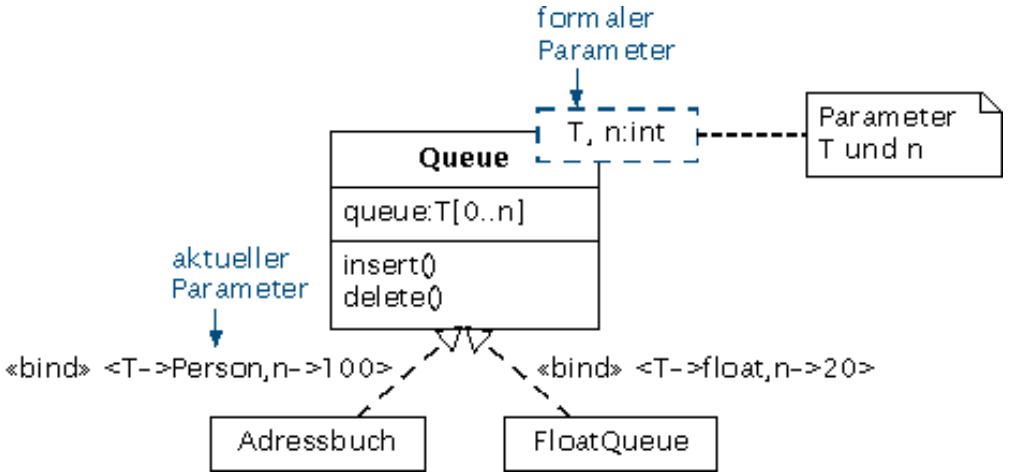
Prozess zum verfeinern statischer Modelle

Verfeinerung des OOA-Modells zum OOD-Modell

- Klassenstruktur verfeinern
  - Hinzufügen von Container-Klassen (→ **Exkurs: Parametrisierte Klasse**)
  - Verwendung von Schnittstellen für eine lockere Bindung
  - Zerlegen komplexer Klassen
  - Zusammenfassen von Klassen mit starker Interaktion
  - Hinzufügen von Klassen zum Modellieren von Zwischenergebnissen
  - Transformation von Assoziationsklassen in »normale« Klassen

Exkurs: Parametrisierte Klasse

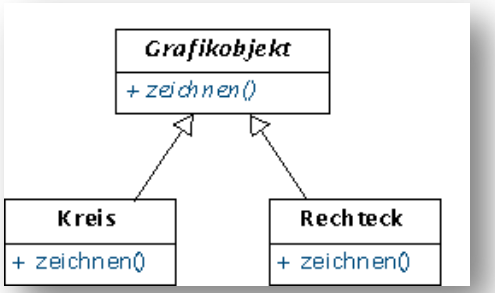
- Beispiel: Parametrisierte Klasse **Queue**
- Besitzt die üblichen Operationen **insert()** und **delete()**
- Parameter **T** beschreibt einen Typ
- Parameter **n** gibt maximale Größe an
- Bildet die Vorlage für die Klassen **Adressbuch** und **FloatQueue**

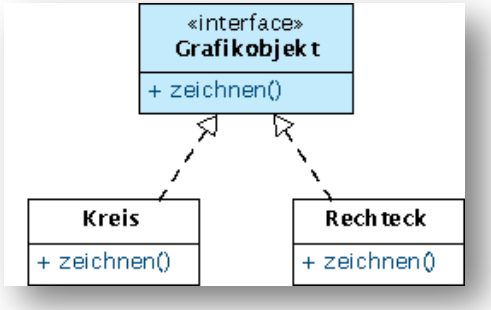


Operationen verfeinern

Abstrakte Operation

- Besteht nur aus der Signatur
  - Name der Operation
  - Namen und Typen aller Parameter
  - Ergebnistyp
- Besitzt keine Implementierung
- Definiert gemeinsame Schnittstelle für Unterklassen





Operationen weiter verfeinern (1/2)

Komplexe Operationen in einfachere, interne Operationen zerlegen

- Wird der Algorithmus zu umfangreich, dann muss eine komplexe Operation in einfachere Operationen zerlegt werden.
- Ggf. müssen für diese Operationen neue Klassen identifiziert werden.

Transformieren einfacher Lebenszyklen in Algorithmen

- Besitzt die Klasse einen Zustandsautomaten, so ist eine auszuführende Operation von dem jeweiligen Objektzustand abhängig.

⇒ Der Algorithmus muss entsprechende Abfragen enthalten.

Operationen weiter verfeinern (2/2)

Transformieren komplexer Lebenszyklen mittels Zustandsmuster

- Besitzt die Klasse einen Zustandsautomaten, so ist eine auszuführende Operation von dem jeweiligen Objektzustand abhängig.

⇒ Das Zustandsmuster (→ Entwurfsmuster) ist anzuwenden.

Assoziationen verfeinern

Assoziationen verfeinern

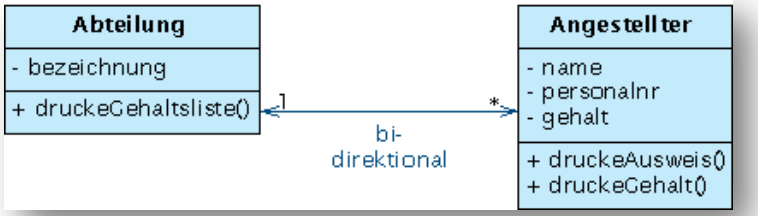
- Prüfen, welche Assoziationen unidirektional modelliert werden können
- Navigierbarkeit für alle Assoziationen spezifizieren
- many-Assoziationen durch Container realisieren
- {ordered} auf geordnete Container abbilden
- (Aggregation und Komposition wie einfache Assoziation entwerfen)
- (Bei Komposition Delegation der Funktionalität prüfen)
- Zugriffspfade optimieren

Assoziationen verfeinern

Implementierung in Java

Als Beispiel wird die birektionale Assoziation aus folgender Abbildung realisiert:

- Basisoperationen add(), remove() und get() müssen implementiert werden




```java
import java.util.Vector;

   public class Abteilung {
       private Vector<Angestellter> angestellte
                   = new Vector<Angestellter>();

       public void addAngestellten(Angestellter angestellter){
               angestellte.addElement(angestellter);
           }

       public void removeAngestellten(Angestellter angestellter){
               angestellter.removeAbteilung();
       }

       public Angestellter getAngestellterByIndex(int index) {
           return(angestellte.get(index));
       }

       public Vector<Angestellter> getAngestellte( ) {
           return(angestellte);
       }
   }
```

```java
public class Angestellter {
    private Abteilung arbeitetIn;

    public void setAbteilung(Abteilung abteilung){
        this.arbeitetIn = abteilung;
    }

    public void removeAbteilung(){
        this.arbeitetIn = null;
    }

    public Abteilung getAbteilung() {
        return(this.arbeitetIn);
    }
}
```

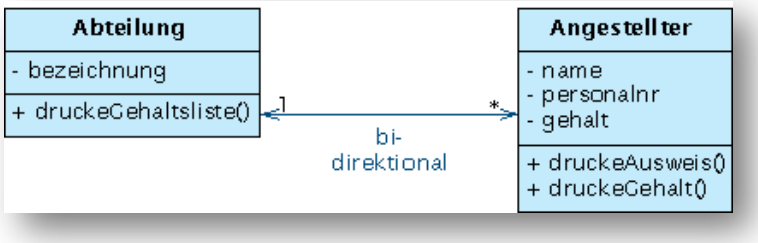
Fachhochschule Dortmund
University of Applied Sciences and Arts

Assoziationen verfeinern

Implementierung in Java

Als Beispiel wird die birektionale Assoziation aus folgender Abbildung realisiert:

- Basisoperationen add(), remove() und get() müssen implementiert werden

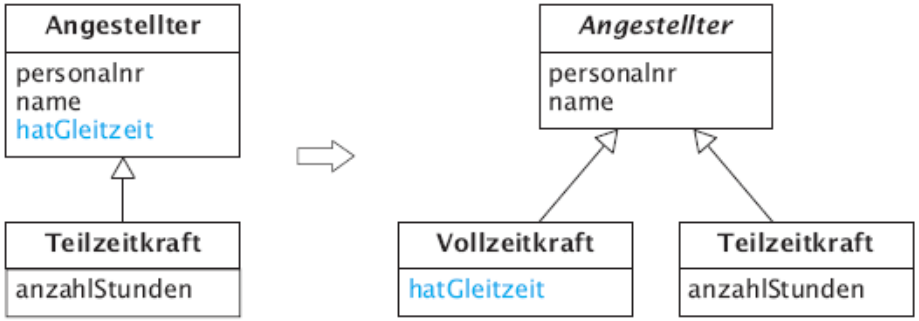



```java
import java.util.Vector;

public class Abteilung {
    private Vector<Angestellter> angestellte
                = new Vector<Angestellter>();

    public void addAngestellten(Angestellter angestellter){
        if (!angestellte.contains(angestellter)){
            angestellte.addElement(angestellter);
            angestellter.setAbteilung(this);
        }
    }

    public void removeAngestellten(Angestellter angestellter){
        if (angestellte.removeElement(angestellter)){
            angestellter.removeAbteilung();
        }
    }

    public Angestellter getAngestellterByIndex(int index) {
        return(angestellte.get(index));
    }

    public Vector<Angestellter> getAngestellte( ) {
        return(angestellte);
    }
}
```

```java
public class Angestellter {
    private Abteilung arbeitetIn;

    public void setAbteilung(Abteilung abteilung){
        this.arbeitetIn = abteilung;
        if (!abteilung.getAngestellte().contains(this)){
            abteilung.getAngestellte().addElement(this);
        }
    }

    public void removeAbteilung(){
        if (this.arbeitetIn.getAngestellte().contains(this)) {
            this.arbeitetIn.getAngestellte().removeElement(this);
        }
        this.arbeitetIn = null;
    }

    public Abteilung getAbteilung() {
        return(this.arbeitetIn);
    }
}
```

Generalisierungsstruktur verfeinern (1/2)

- Halten Sie Generalisierungshierarchien flach → Maximal sechs Ebenen
- Oberklassen einer Generalisierungshierarchie sollen abstrakt sein



Generalisierungsstruktur verfeinern (2/2)

- Gemeinsamkeiten so hoch wie möglich in Generalisierungshierarchien einordnen
  - Bei einer Änderung nur eine Klasse betroffen
  - Voraussetzung: sinnvolle Generalisierungsstruktur
- Gemeinsame Attribute als Datentyp spezifizieren und komplexes Attribut in jede Unterklasse einfügen
- Gemeinsame Operationen durch Oberklasse realisieren
  - Wenn zusätzlich gemeinsame Attribute existieren, sollten diese ebenfalls in der Oberklasse spezifiziert und vererbt werden

Polymorphismus und Wiederverwendung

Maximierung des Polymorphismus → leichte Erweiterbarkeit

- Alle Operationen von Unterklassen sind so hoch wie irgend möglich in der Generalisierungshierarchie einzuordnen
- Die Namen von Operationen sind so zu wählen, dass man immer einen einzigen Namen für konzeptionell gleiche Operationen verwendet, z.B. print() oder put().
- Die Signaturen der Operationen sind so allgemein wie möglich zu spezifizieren, um die spätere Wartbarkeit zu unterstützen. Dazu ist zu überlegen, welche Änderungen evt. an dem System vorgenommen werden könnten.
- Es kann auch sinnvoll sein Generalisierungsstrukturen zu komprimieren. Dadurch wird dann ein Teil der Semantik, die im statischen Modell spezifiziert ist, in das dynamische Modell übernommen.

Wiederverwendung existierender Klassen

Fachhochschule
Dortmund
University of Applied Sciences and Arts

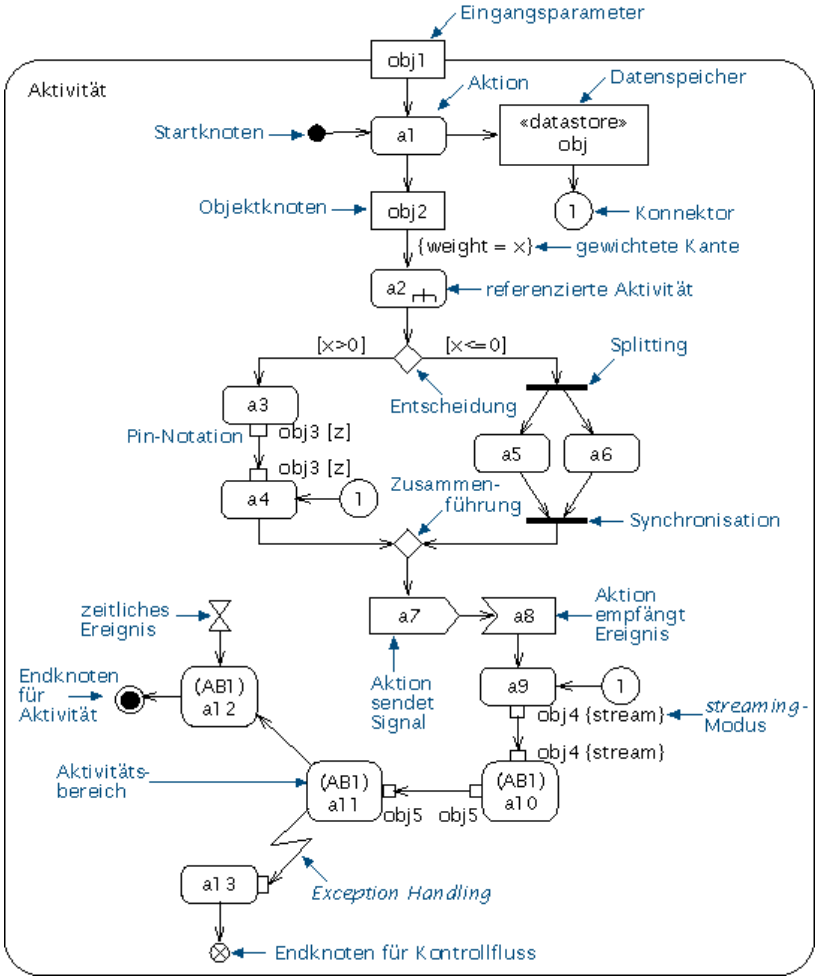
Aktivitätsdiagramme

Aktivitäten

- Beschreiben wie in der Analyse die Ausführung von Verarbeitungsschritten (→ Aktionen).
- In der Analyse werden sie zur Modellierung von Workflows oder für die Spezifikation von Use-Cases eingesetzt.
- Im (Fein-)Entwurf werden Aktivitätsdiagramme zur Beschreibung **komplexer Operationen** eingesetzt.

→ Kurze Wiederholung

Notation für Aktivitätsdiagramm

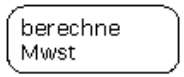


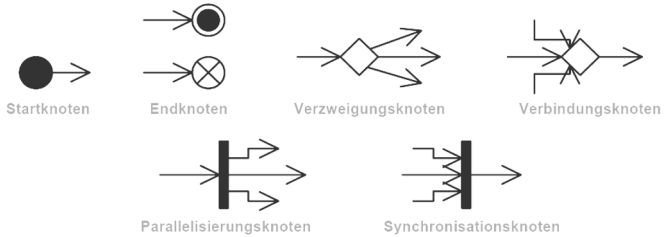
Aktivitätsdiagramm - Knoten

- Es lassen sich Aktionsknoten, Kontrollknoten, Objektknoten und Strukturknoten unterscheiden

Aktionsknoten

Elementare Aktion
Aufruf einer Aktivität

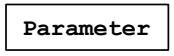




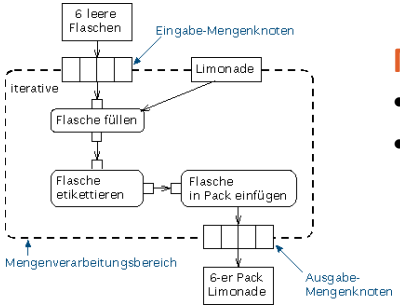
Kontrollknoten

- Entscheidung und Zusammenführung
- Splitting und Synchronisation
- Start- und Endknoten

Objektknoten



- Weiterreichung von Daten von einer Aktion zur nächsten
- Häufig mit dem Namen der Klasse benannt



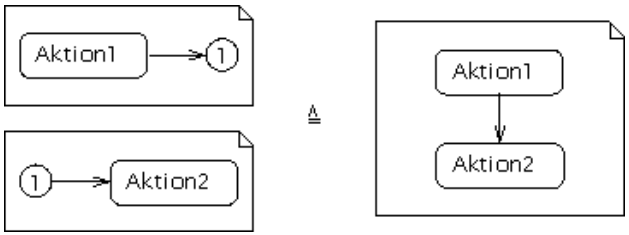
Mengenverarbeitungsbereich

- Verarbeitung von Objektmengen
- Darstellung:
  - Gestricheltes Rechteck mit abgerundeten Ecken und
  - Links oben Schlüsselwort «iterative»

Aktivitätsdiagramm

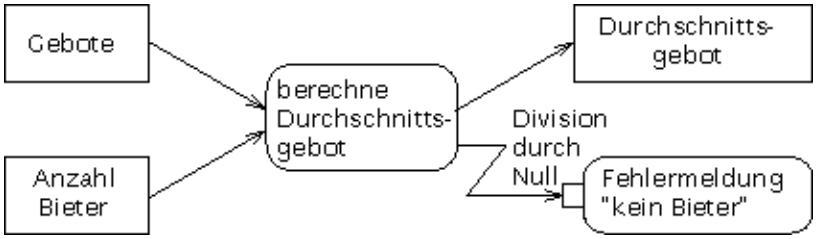
Konnektor

- Ermöglicht es, eine Kante zu unterbrechen und an beliebiger Stelle fortzuführen



Exception Handler

- Element, das spezifiziert, welche Anweisungen auszuführen sind, falls ein bestimmter Fehler auftritt



Aktivitätsdiagramm

Beispiel: Fibonacci-Folge

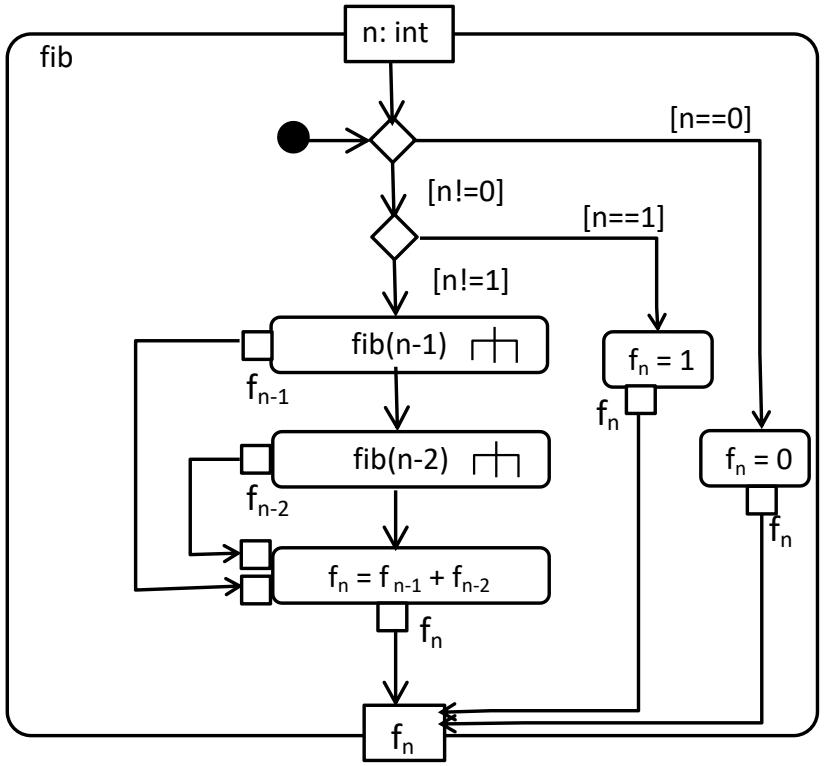
$$f_n = \begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad \text{,für alle } n \geq 2 \end{cases}$$

n	f_n
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34

- Für die beiden ersten Zahlen werden die Werte null und eins vorgegeben.
- Jede weitere Zahl ist die Summe ihrer beiden Vorgänger.

Aktivitätsdiagramm

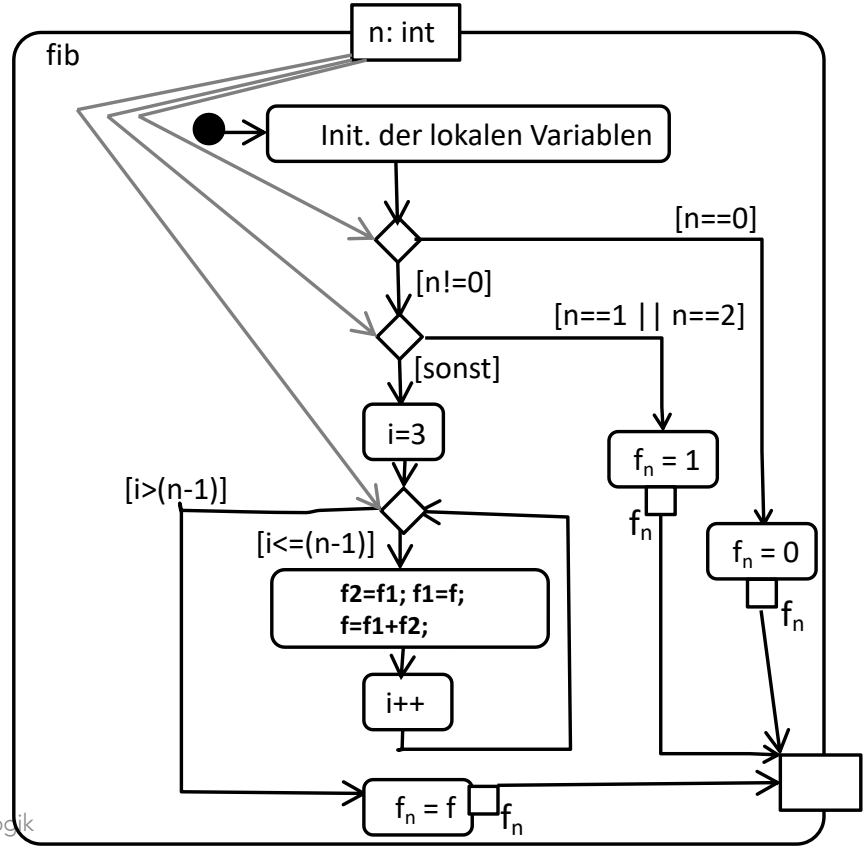
Beispiel: Fibonacci-Folge (rekursiv)




```java
public static int fib(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}
```

Aktivitätsdiagramm

Beispiel: Fibonacci-Folge (iterativ)



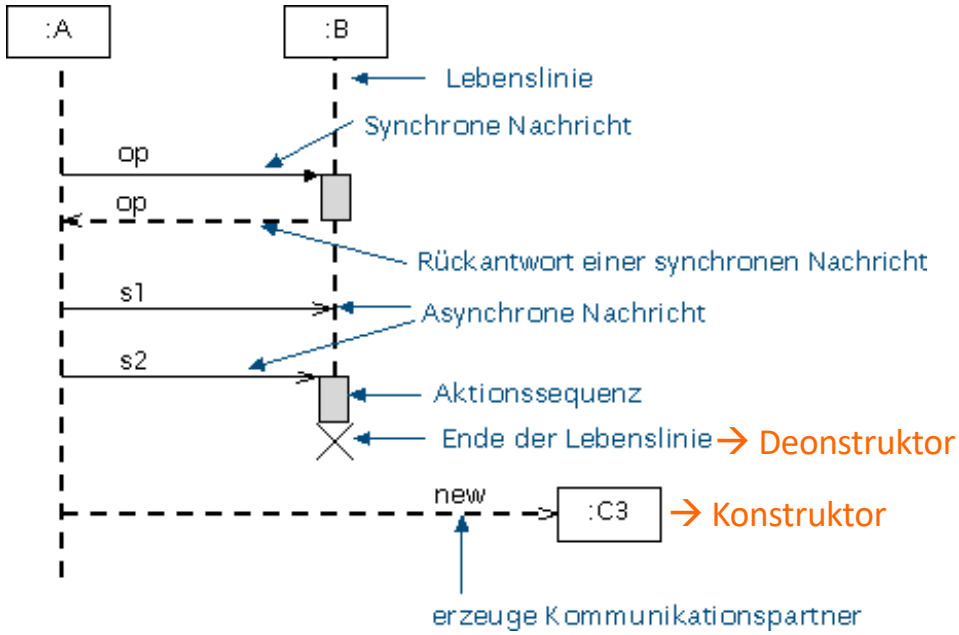

```java
public static int fib(int n) {
    int f2=0;
    int f1=1;
    int f=2;

    if (n==0) return 0;
    if (n == 1 || n == 2)  return 1;

    for (int i=3; i<=(n-1); i++){
        f2=f1;
        f1=f;
        f=f1+f2;
    }
    return f;
}
```

Fachhochschule
Dortmund
University of Applied Sciences and Arts

Sequenzdiagramme

Sequenzdiagramme

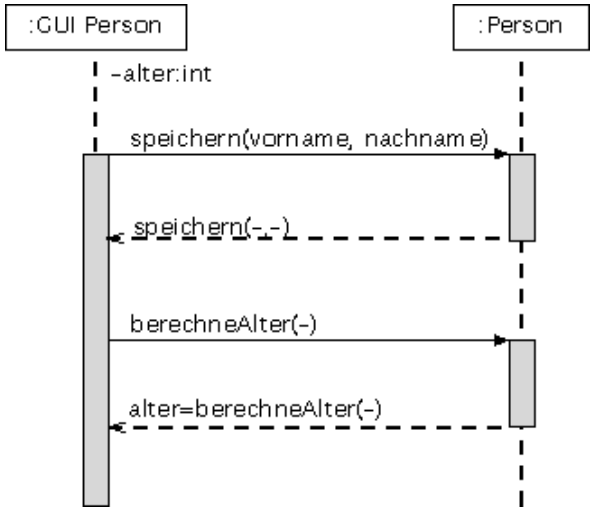
- Beschreibt Interaktion von Operationen
- Im (Fein-)Entwurf werden Sequenzdiagramme eingesetzt, um die Reihenfolge der Operationsaufrufe (über verschiedene Objekte/Schichten hinweg) deutlich zu machen



Sequenzdiagramme

Modellierung der Parameterübergabe

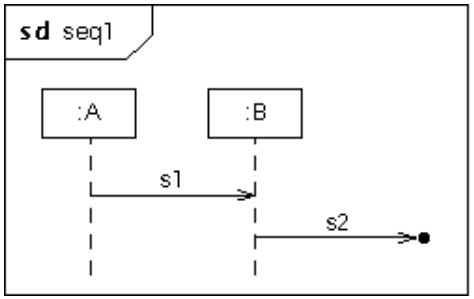
- Klammern mit Variablen werden nur angegeben, wenn durch den Aufruf Daten übergeben werden
- Bei Antwortnachrichten wird der Name der ursprünglichen Aufrufnachricht wiederholt

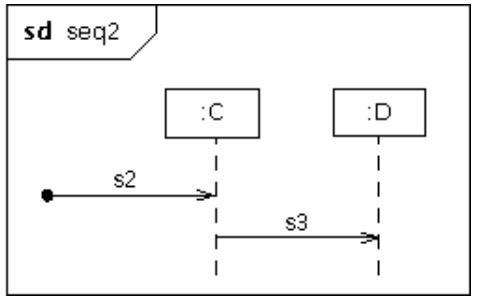


Sequenzdiagramme

Verlorene und gefundene Nachrichten

- Zur Verteilung umfangreicher Abläufe auf mehrere Sequenzdiagramme
- Verlorene Nachricht: Nachricht erreicht nicht ihr Ziel
- Gefundene Nachricht: Das Senden der Nachricht liegt außerhalb der Spezifikation
- Darstellung: Pfeilspitze zeigt auf einen Punkt bzw. geht von einem Punkt aus





Sequenzdiagramme

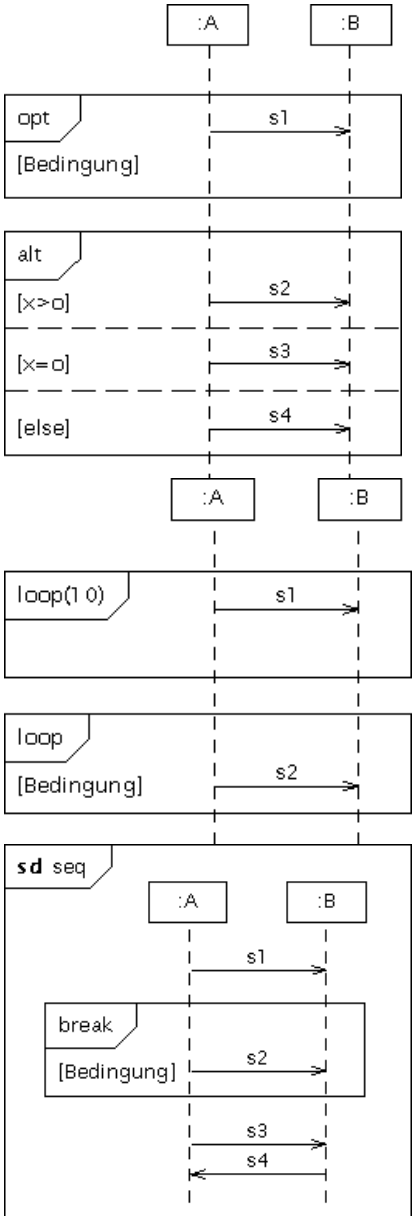
Kombinierte Fragmente im Sequenzdiagramm

- Ermöglichen es, Sequenzdiagramme präzise zu spezifizieren
- Darstellung: Rechteckiger Rahmen, in dem links oben der Interaktionsoperator eingetragen wird

Sequenzdiagramme

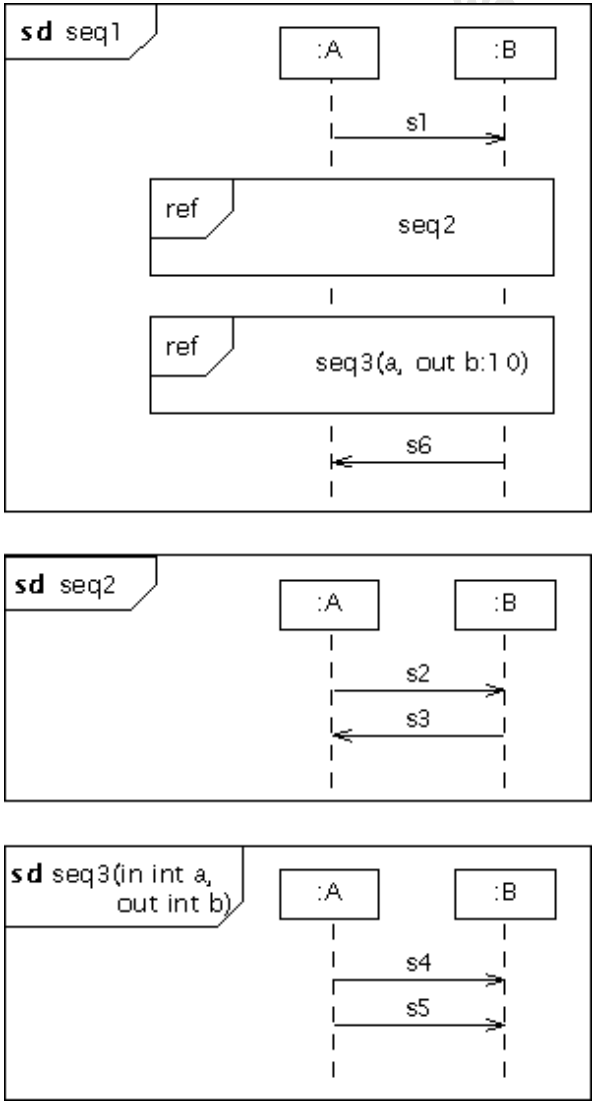
Interaktionsoperationen

- opt: bedingte Anweisung(en) (if-then)
- alt: alternative Abläufe (if-then-else, switch)
- loop: Schleife (for, while-do, do-while)
  - loop (min,max): ganzzahlige Werte, die die Mindest- und Höchstzahl von Wiederholungen angeben
  - loop (min,*): min als Untergrenze, unendlich als Obergrenze
  - loop (min): Anzahl der Iterationen
  - loop: Schleife wird 0 bis unendlich Mal ausgeführt
  - loop (var, Bed, Inc/Dec): vgl. for-Schleife
- break: Ausnahmebehandlungen (exception, goto)



Fachhochschule Dortmund
University of Applied Sciences and Arts

Sequenzdiagramme



Interaktionsreferenz

- Referenz auf ein anderes Sequenzdiagramm oder anderes Interaktionsdiagramm
- Häufig wiederkehrende Abläufe können durch ein eigenständiges Diagramm beschrieben und an beliebigen Stellen eingefügt werden
- Parameter können übergeben werden

Bedeutung

- Sequenzdiagramme sind im Entwurf sehr wichtig, weil die das dynamische Verhalten transparent machen.
- Für die Praxis ist die richtige Auswahl der wichtigen Ausschnitte wichtig

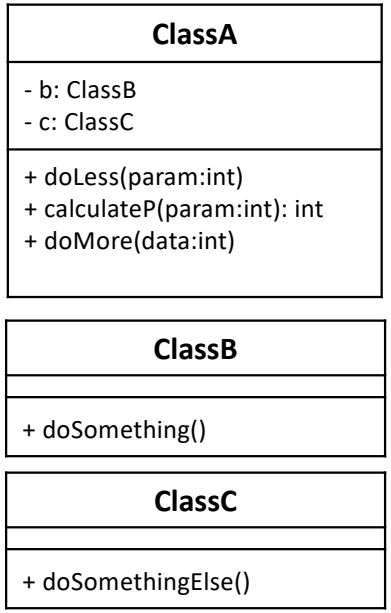
Sequenzdiagramme

Beispiel

```
public void doLess(int param)
{
  b.work(param);
}
```

```
public int calculateP(int param)
{
  int p = 2 * param;
  return p;
}
```

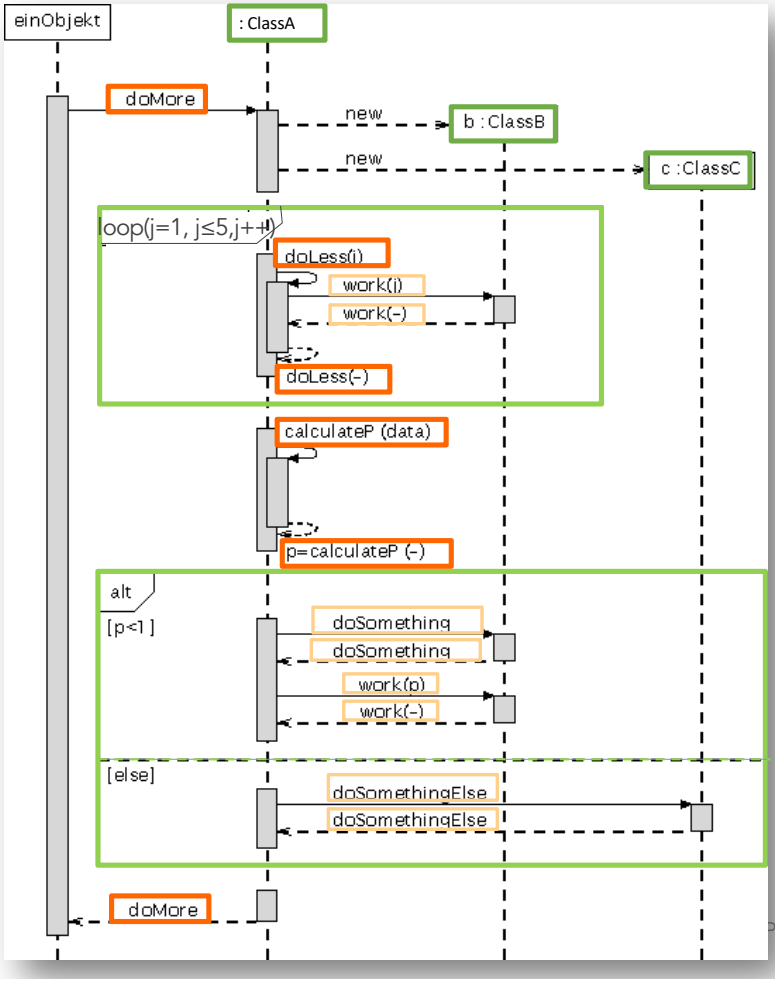
```
public void doMore(int data)
{
  b = new ClassB();
  c = new ClassC();
  for (int j = 1; j <= 5; j++)
    doLess (j);
  int p = calculateP (data);
  int (p < 1)
  {
    b.doSomething();
    b.work(p);
  }
  else
    c.doSomethingElse();
}
```



Sequenzdiagramme

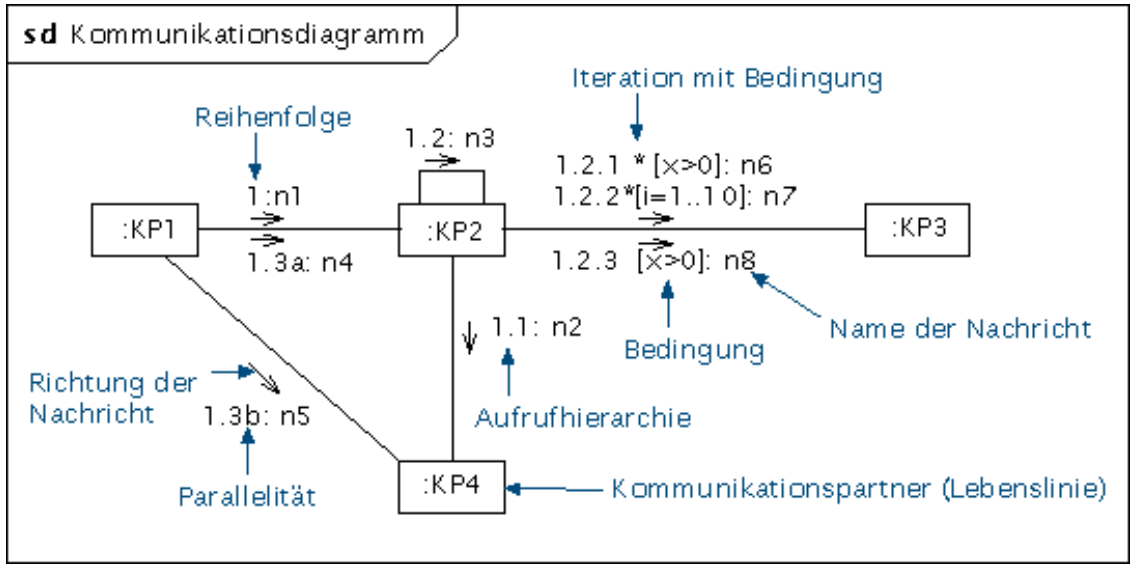
Beispiel

```
public void doLess(int param)
{
  b.work(param);
}

public int calculateP(int param)
{
  int p = 2 * param;
  return p;
}

public void doMore(int data)
{
  b = new ClassB();
  c = new ClassC();
  for (int j = 1; j <= 5; j++)
    doLess (j);
  int p = calculateP (data);
  int (p < 1)
  {
    b.doSomething();
    b.work(p);
  }
  else
    c.doSomethingElse();
}
```



Kommunikationsdiagramme

- Beschreibt wie das Sequenzdiagramm die Kommunikation zwischen Lebenslinien
- Reihenfolge der Nachrichten → Nummerierungsschema
- Bedingungen und Iterationen können einfach spezifiziert werden



Kommunikationsdiagramme

- Nachricht kann wie beim Sequenzdiagramm mit Parametern angetragen werden
- Pfeil gibt die Richtung der Nachricht an
- Hierarchische Nummern geben die Reihenfolge und Verschachtelung an

- Iteration: *[Bedingung]: nachricht
- Bedingte Ausführung: [Bedingung] nachricht
- Sequenz von Nachrichten: 1:nachricht1 und 2:nachricht2
- Schachtelung von Nachrichten: 1:nachricht1 und 1.1:nachricht2
- Parallele Ausführung: 2a:nachricht1 und 2b:nachricht2

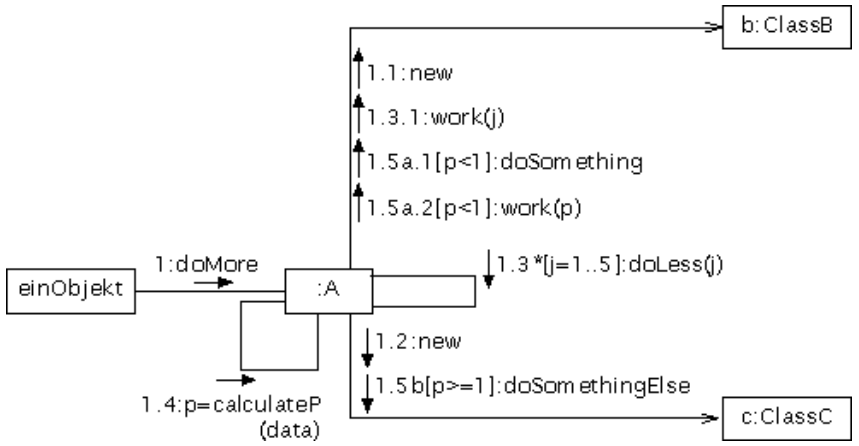
Kommunikationsdiagramme

Beispiel

```
public void doLess(int param)
{
  b.work(param);
}
```

```
public int calculateP(int param)
{
  int p = 2 * param;
  return p;
}
```

```
public void doMore(int data)
{
  b = new ClassB();
  c = new ClassC();
  for (int j = 1; j <= 5; j++)
    doLess (j);
  int p = calculateP (data);
  int (p < 1)
  {
    b.doSomething();
    b.work(p);
  }
  else
    c.doSomethingElse();
}
```



Fachhochschule
Dortmund
University of Applied Sciences and Arts

Zustandsautomaten

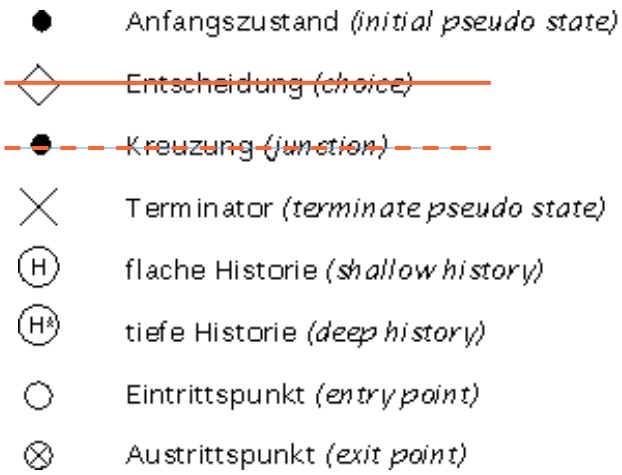
Zustandsautomaten

- **Zwei Arten**
  - **Verhaltenszustandsautomat**
  - Protokollzustandsautomat

- **(Verhaltens-)Zustandsautomaten im Design**
  - Beschreiben das **Verhalten von Objekten** einer Klasse.

Verhaltenszustandsautomaten

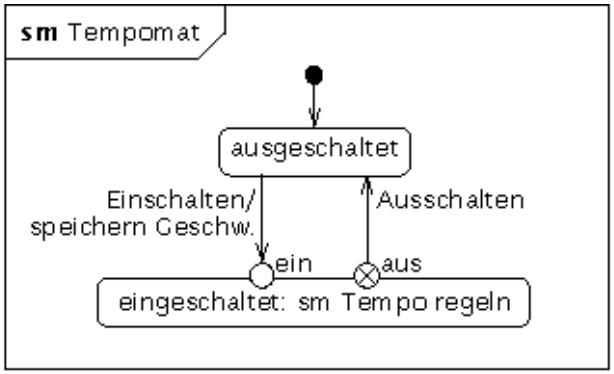
- **Pseudozustände**
  dienen dazu, eine bestimmte Ablauflogik in den Zustandsautomaten einzufügen

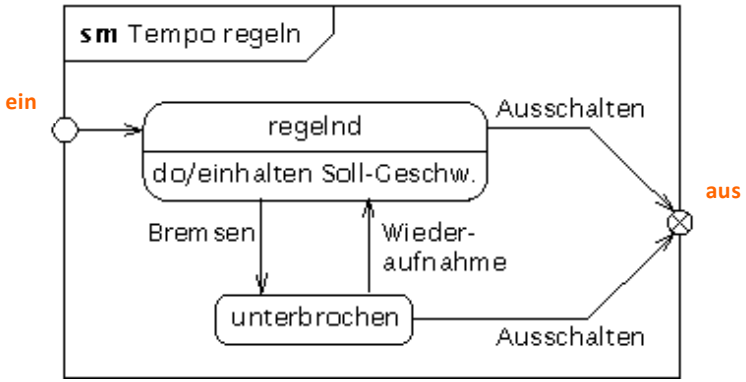


Verhaltenszustandsautomaten

Unterzustandsautomat

- Zustandsdiagramm kann einen Unterzustandsautomatenzustand enthalten, der durch den Unterzustandsautomaten verfeinert wird
- Ein- und Austrittspunkte werden jeweils benannt

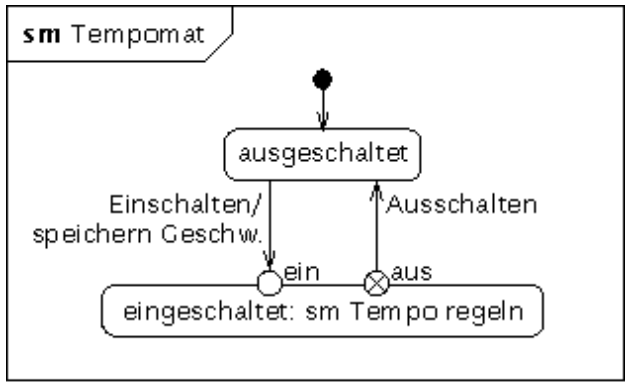


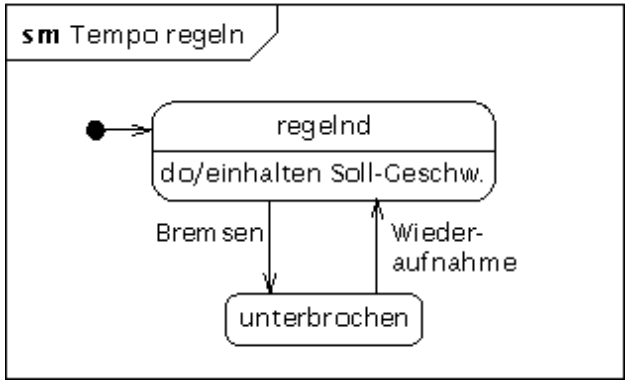


Verhaltenszustandsautomaten

Unterzustandsautomat

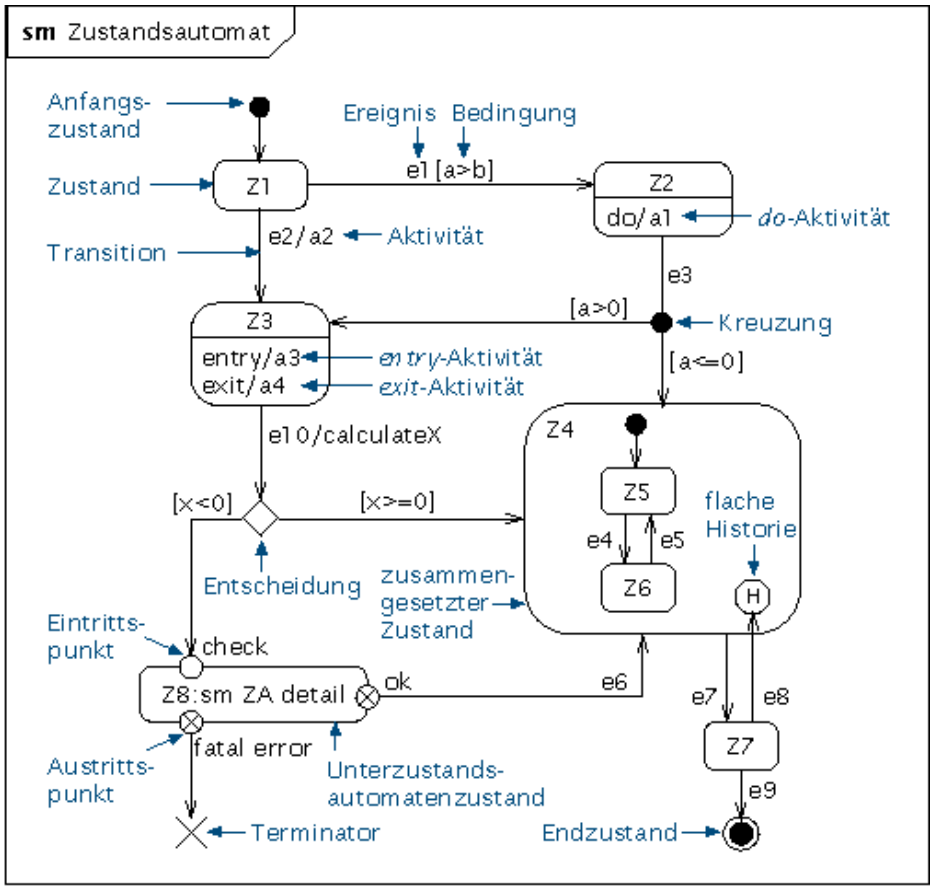
- Kein Eintrittspunkt, wenn U durch seinen Anfangszustand betreten wird
- Kein Austrittspunkt, wenn Verarbeitung beendet oder Austritt durch eine Gruppentransition

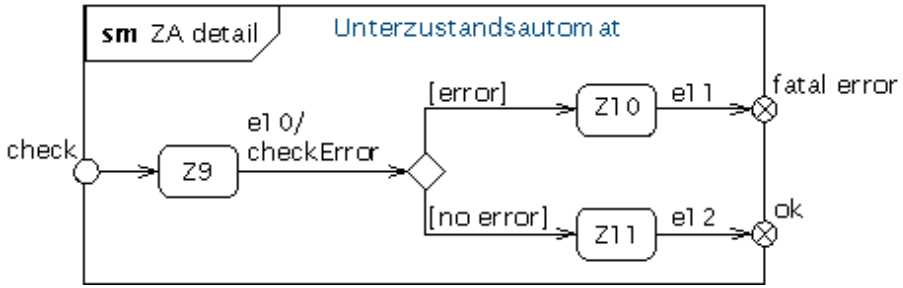




Verhaltenszustandsautomaten

Notation des Verhaltenzustandsautomaten





Realisierung von Verhaltenszustandsautomaten

Für einfache Automaten

- Jede Klasse erhält im Entwurf ein private-Attribut ***classState***, in dem der aktuelle Zustand gespeichert wird
- Jede Operation muss diesen Zustand abfragen
- Ist mit der Operation ein Zustandswechsel verbunden, dann muss sie das Zustandsattribut aktualisieren
- Alternativ kann jede Klasse, die einen Objekt-Lebenszyklus besitzt, eine Operation zur Verfügung stellen, die eintreffende Ereignisse interpretiert und ggf. eine entsprechende Verarbeitung auslöst

Für komplexe Automaten

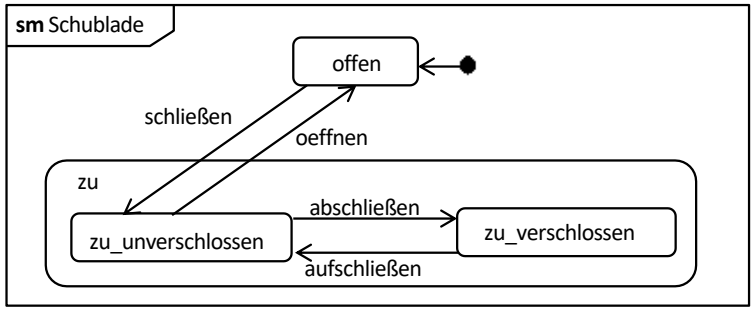
- Zustandmuster (→ Entwurfmuster-Vorlesung)

Realisierung von Verhaltenszustandsautomaten

Java-Implementierung einfacher Verhaltenszustandsautomaten

- Zustand wird durch einen Aufzählungstyp realisiert

```
class Schublade
{
  enum SchubladeZustand {offen, zuUnverschlossen, zuVerschlossen};
  private SchubladeZustand classState;
  public void oeffnen()
  {
    if (classState == zuUnverschlossen)
    {
      classState = offen;
      ...
    }
  }
}
```



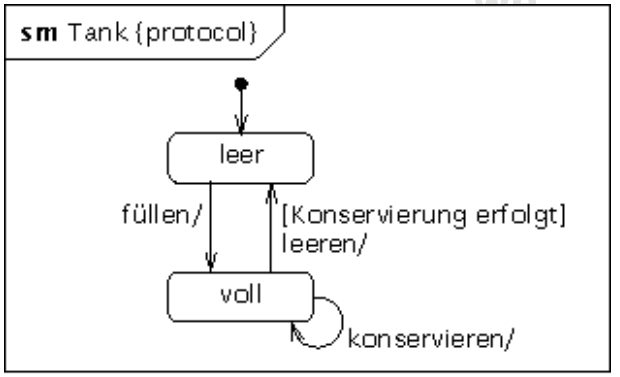
Zustandsautomaten

- **Zwei Arten**
  - Verhaltenszustandsautomat
  - **Protokollzustandsautomat**

- **Protokollzustandsautomaten (im Design)**
  - Beschreiben **Folgen zulässiger Operationsaufrufe** auf Objekten einer Klasse/Schnittstelle.

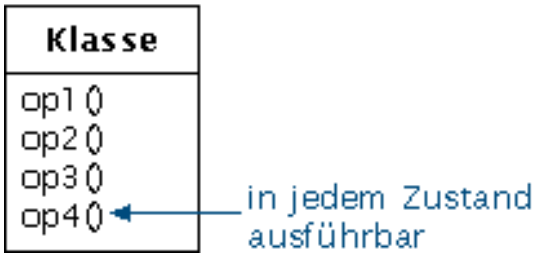
Fachhochschule Dortmund
University of Applied Sciences and Arts

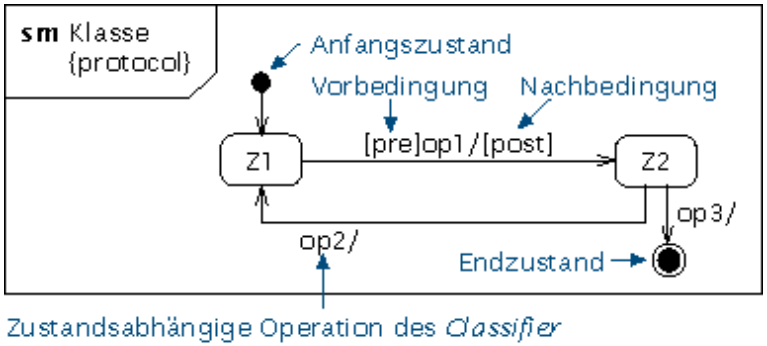
Protokollzustandsautomat



Protokollzustandsautomat

- Dient im Entwurf dazu, das externe Verhalten von Klassen und Schnittstellen zu spezifizieren
- Enthält alle Operationen, die nur in bestimmten Zuständen ausgeführt werden
- Vorbedingung und Nachbedingung





Fachhochschule
Dortmund
University of Applied Sciences and Arts

Businesslogik