

## Entwurfsmuster *Abstrakte Fabrik*

### Eine **Abstrakte Fabrik**

ermöglicht die Nutzung gleicher Abläufe für verschiedene Familien von Objekten.

#### Motivation:

- ❑ Ein Softwareprodukt kann mit den weitgehend gleichen Abläufen in verschiedenen Kontexten eingesetzt werden. Die gleichen Teile sollen dabei unverändert beibehalten werden.

#### Idee:

- ❑ Die Software besteht aus einem gleichbleibenden Anwendungskern und weiteren Komponenten, die in verschiedenen Varianten auftreten.
- ❑ Für eine Konfiguration werden immer nur Komponenten ausgewählt, die zusammen passen, d.h. zu einer Familie von Produkten gehören.
- ❑ Die für eine Konfiguration benötigten Komponenten werden durch eine spezielle Komponente, die Fabrik, bei Bedarf erzeugt.

Literatur: Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.214-217  
[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)

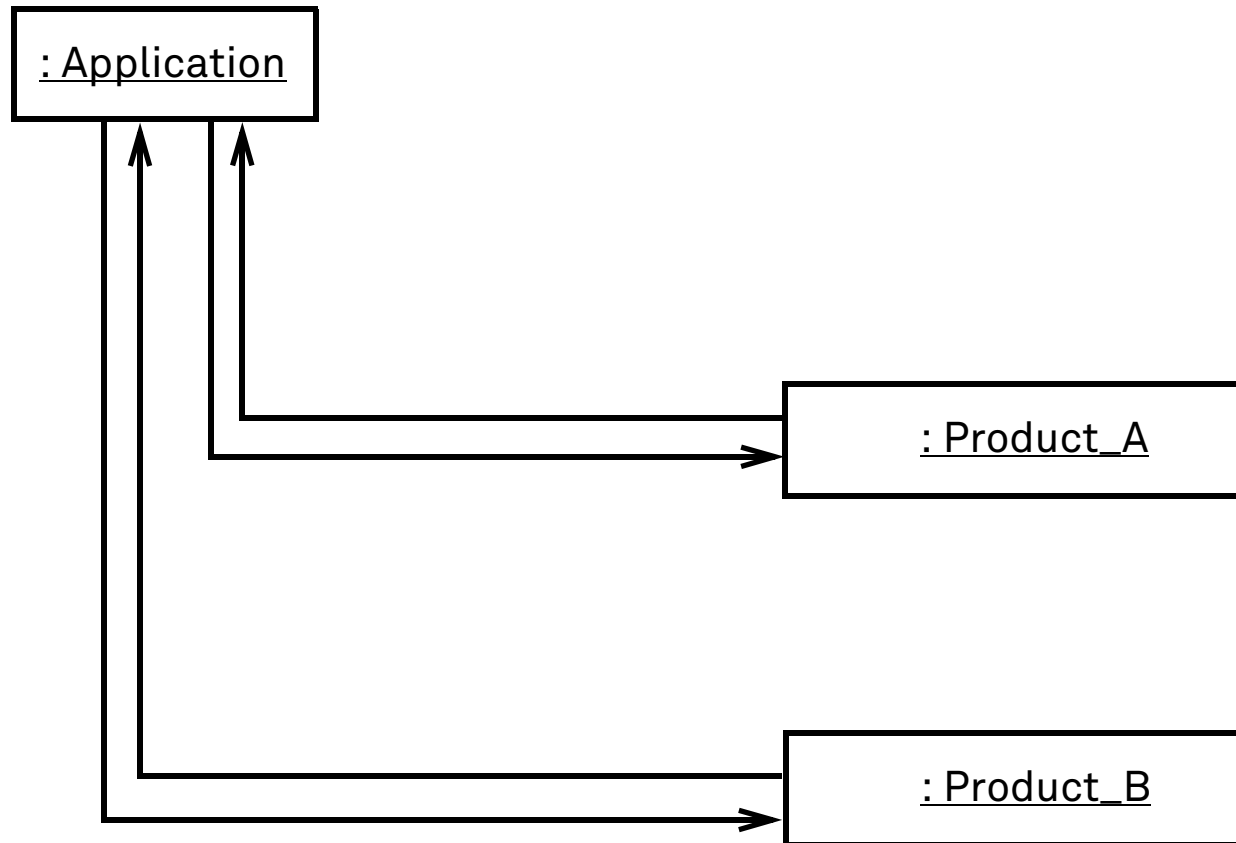
## Entwurfsmuster *Abstrakte Fabrik*

(Fortsetzung)

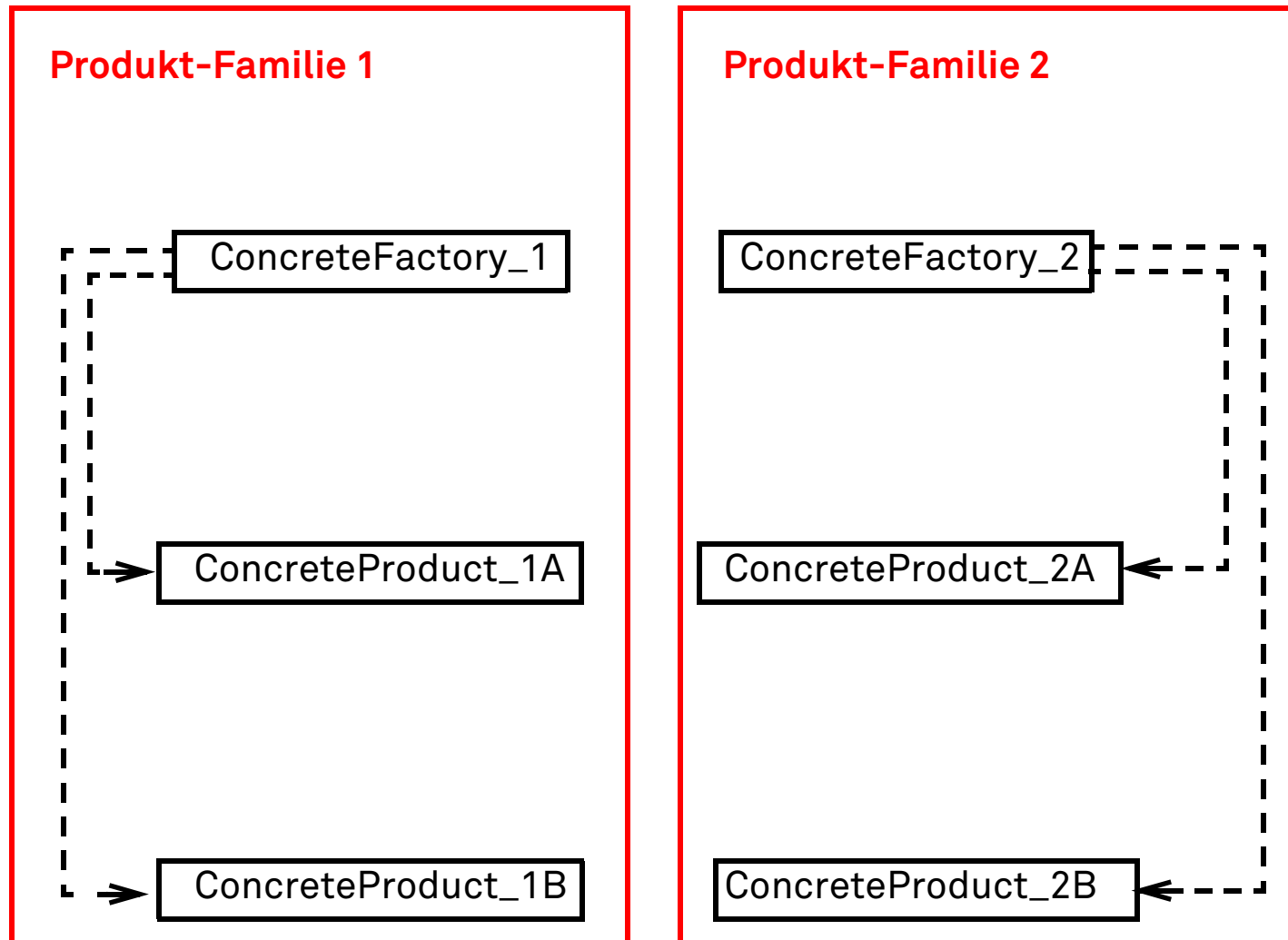
### Beispiele:

- ❑ Ziel: verschiedene Benutzeroberflächen für das gleiche Softwareprodukt  
Lösung: mehrere Familien mit Klassen für graphische Präsentationen, eine dieser Familien wird ausgewählt und die zugehörigen Objekte werden erzeugt
  
- ❑ Ziel: Einsatz eines Softwareprodukts in unterschiedlichen Anwendungsbereichen  
Lösung: mehrere Familien mit Klassen für die Benutzeroberfläche mit unterschiedlicher Präsentation, eine dieser Familien wird ausgewählt und die zugehörigen Objekte werden erzeugt
  
- ❑ Ziel: verschiedene Darstellungen von Spielelementen im *SWT-Starfighter*  
Lösung: mehrere Familien mit Klassen für die Anzeige von Spielelementen, eine dieser Familien wird ausgewählt und die zugehörigen Objekte werden erzeugt

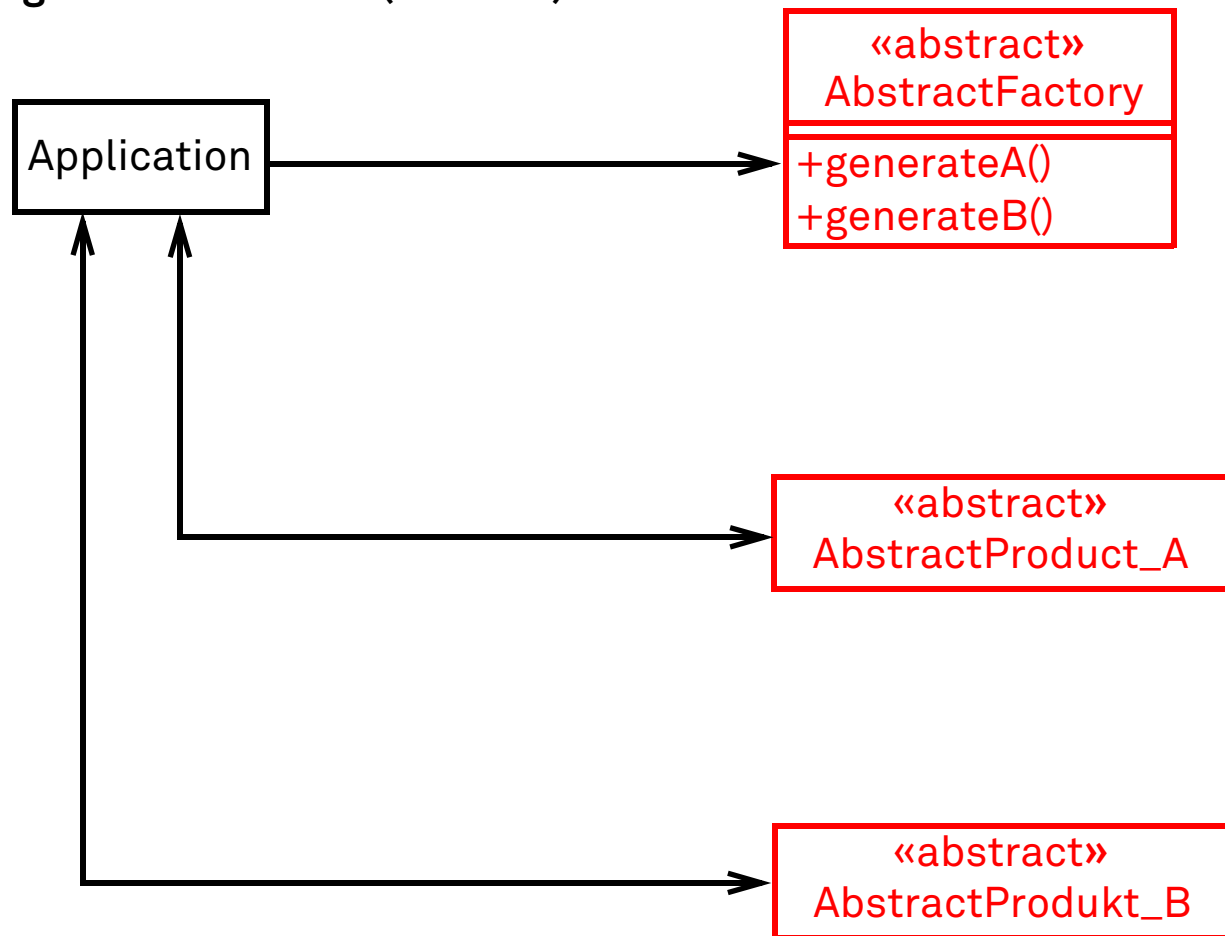
## allgemeine Struktur (Objekte der Anwendung)



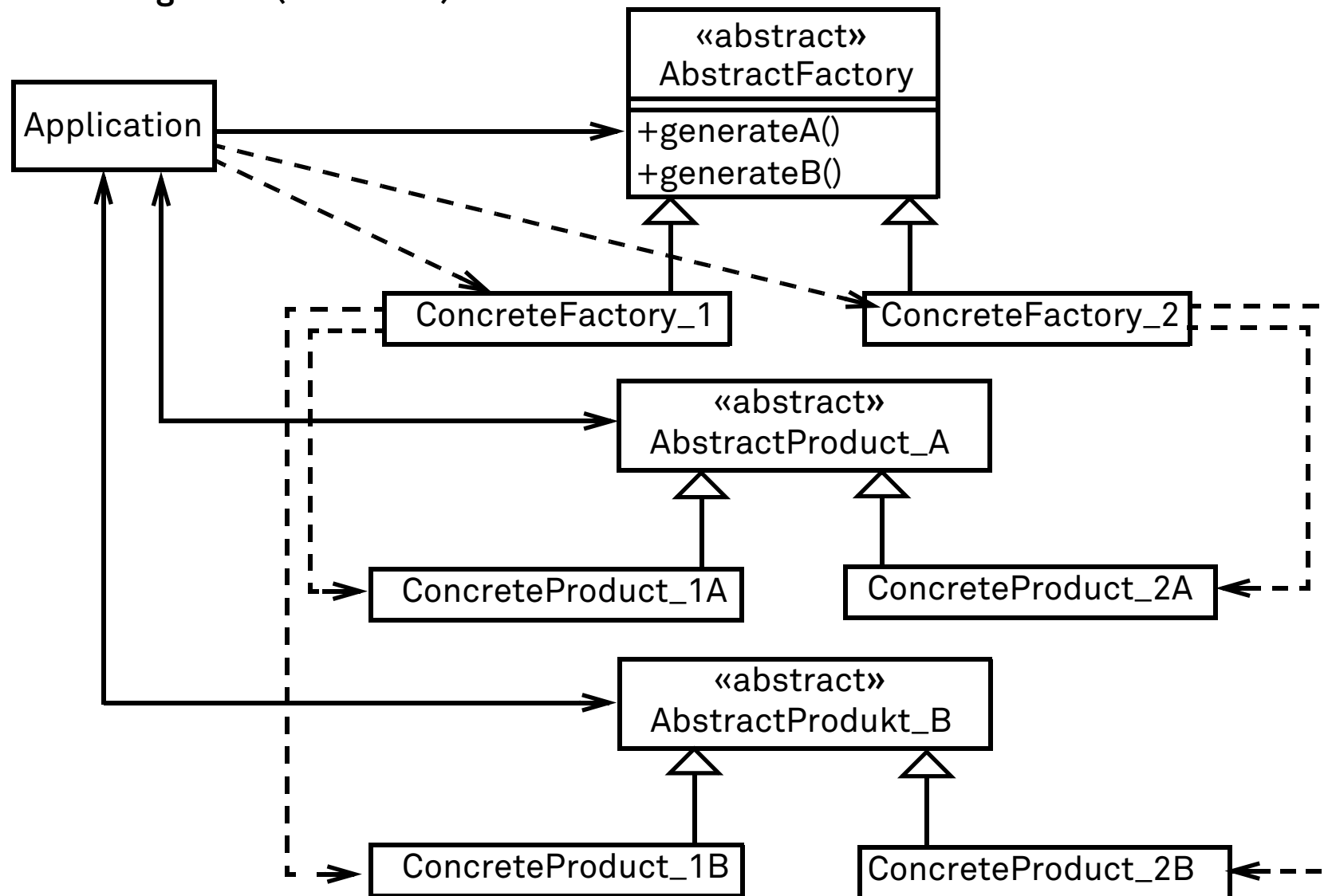
## allgemeine Struktur (Klassendiagramm Produkt-Familien)



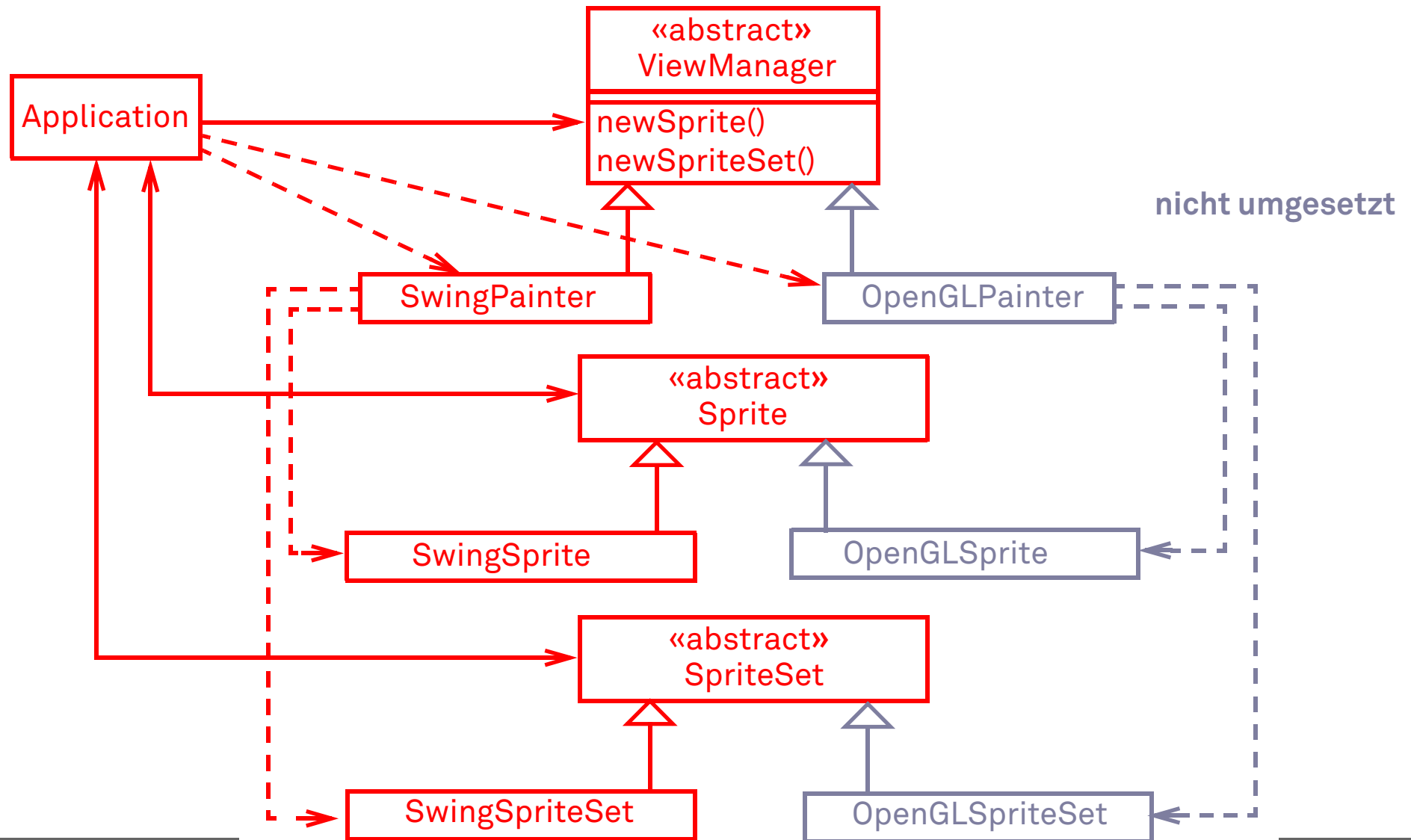
## allgemeine Struktur (Klassen)



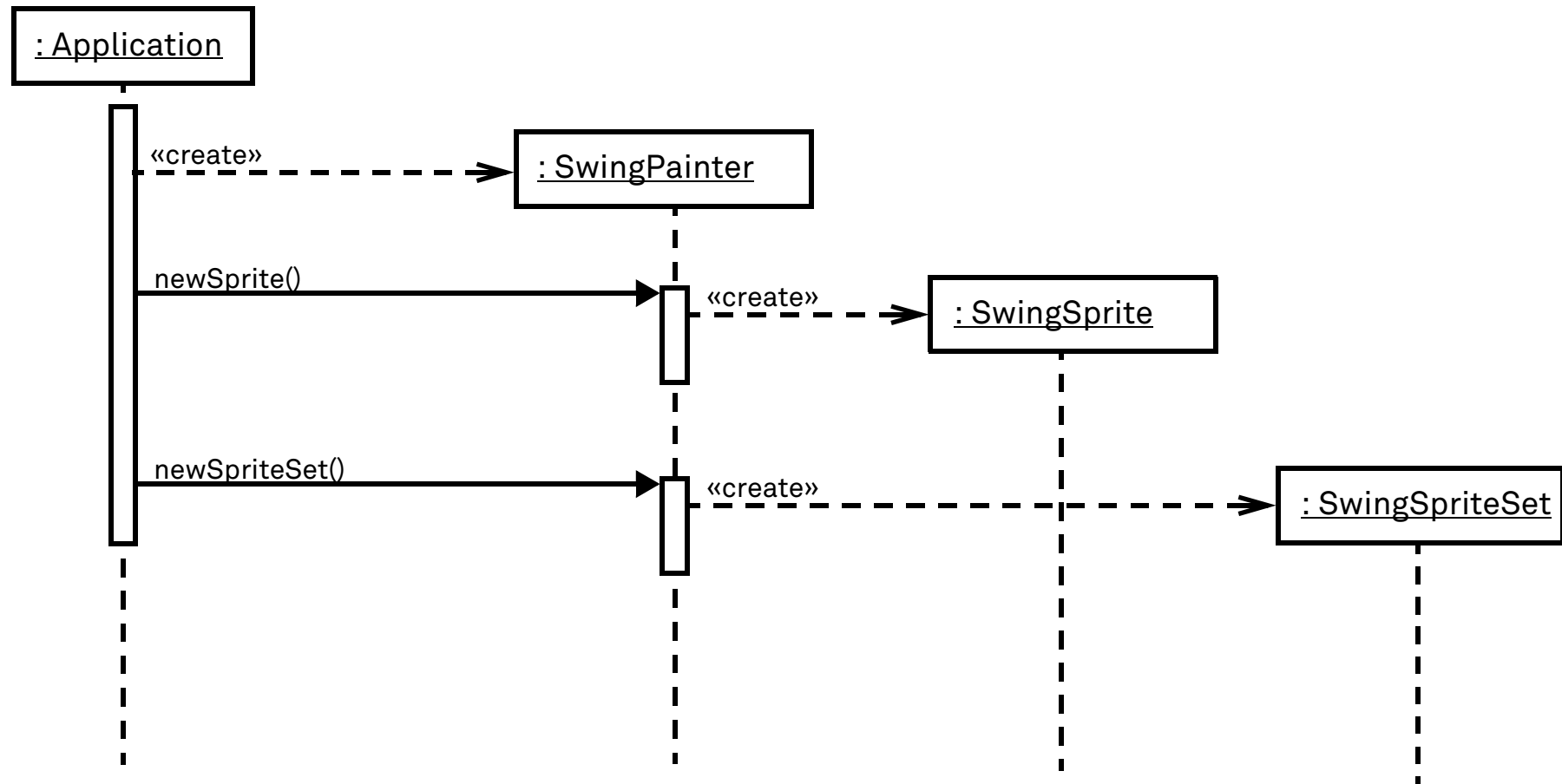
## Klassendiagramm (Übersicht)



## Klassendiagramm (Beispiel aus dem *SWT-Starfighter*)



## Sequenzdiagramm (Beispiel)





## Zusammenfassung – Entwurfsmuster *Abstrakte Fabrik*


### Vorteile:

- ❑ Das Muster Abstrakte Fabrik vereinfacht die Anpassung eines Softwareprodukts durch Austauschen von Gruppen (Familien) von Objekten.
- ❑ Die Anpassung erfolgt dynamisch zur Laufzeit.
- ❑ Weitere Produktfamilien lassen sich in dem durch die Schnittstellen gegebenen Rahmen leicht ergänzen.

### Nachteile:

- ❑ Das Vorab-Erkennen einer Situation, die durch eine abstrakte Fabrik nachhaltig unterstützt wird, ist schwer.
- ❑ Die Konstruktion einer abstrakten Fabrik ist aufwändig.  
Insbesondere muss als Vorbereitung eine geeignete Beschreibung des Umfangs der Produktfamilie erfolgen.
- ❑ Das Anlegen einer abstrakten Fabrik lohnt nur dann, wenn tatsächlich mehrere Produkte in verschiedenen Familien identifiziert werden können.

## Zusammenfassung Entwurfsmuster

	Strukturmuster	Verhaltensmuster	Erzeugungsmuster
<b>klassenbezogene Muster</b>	Klassenadapter		Fabrikmethode
<b>objektbezogene Muster</b>	Objektadapter Dekorierer Kompositum Fassade 	Strategie Mediator Beobachter Iterator Besucher	Abstrakte Fabrik Singleton

- ❑ Alle Entwurfsmuster sind aus Erfahrungen abgeleitet worden.
- ❑ Entwurfsmuster bieten geeignete Lösungsansätze für wiederkehrende Probleme.
- ❑ Einige Entwurfsmuster werden in Standardbibliotheken – siehe Java – unterstützt.
- ❑ Entwurfsmuster können flexibel auf verschiedene Weisen umgesetzt werden.
- ❑ Entwurfsmuster bieten ein gemeinsames Vokabular für Entwickler.
- ❑ Entwurfsmuster können miteinander kombiniert werden

## Zusammenfassung Entwurfsmuster

(Fortsetzung)

kritische Anmerkungen:

- ❑ Entwurfsmuster sind *Ideen* für Lösungen, aber keine fertigen Lösungen.
- ❑ Entwurfsmuster müssen dem konkreten Problem angepasst werden.
- ❑ Der Einsatz von Entwurfsmustern erfordert Erfahrung in der Gestaltung objektorientierter Software.
- ❑ Entwurfsmuster umfassen meist nur wenige Klassen, viele Entwurfsmuster sind naheliegende objektorientierte Lösungen.
- ❑ Entwurfsmuster können nur schwer im Quelltext erkannt werden.
- ❑ Kombinationen von Entwurfsmustern können noch viel schwerer im Quelltext erkannt werden.
  
- ❑ Ein sinnloser Einsatz von Entwurfsmustern macht Software nicht besser.

# Folien zur Vorlesung **Softwaretechnik**

## **Teil 4: Überprüfen von Software** **Abschnitt 4.1: Motivation**



## Überprüfen von Software

- ❑ Die folgenden Beispiele zeigen vier abgeschlossene Projekte, bei denen im Betrieb Fehler mit schwerwiegenden Folgen aufgetreten sind.
- ❑ Die Fehler hätten durch geeignete Überprüfungen der Software vor ihrem Einsatz erkannt werden können.
- ❑ Eine Möglichkeit zum Prüfen von Software ist das Testen von Software, in das in diesem Teil der Vorlesung eingeführt wird.

## Überprüfen von Software – Beispiele für Probleme

### Therac 25 - Bestrahlungsgerät

- ❑ Ziel: Tumorbekämpfung durch Röntgenstrahlen
- ❑ Zeitraum: 1985 – 1987
- ❑ Probleme:
  - verschiedene, gleichzeitig ablaufende Prozesse mit unterschiedlichen Prioritäten:
    - Bestrahlungssteuerung mit hoher Priorität
    - Bedienprozess mit niedriger Priorität
    - Konsequenz: Korrektur von Eingabewerten wird verzögert übernommen
  - technisch formulierte Fehlermeldungen, unverständlich für Bedienpersonal
- ❑ Folge: 6 überstrahlte Patienten

## Überprüfen von Software – Beispiele für Probleme

(Fortsetzung)

### Mars Climate Orbiter

- ❑ Ziel: Mars-Beobachtung durch Satellit
- ❑ Zeitraum: 12/1998 – 9/1999
- ❑ Probleme:
  - Sonnensegel versetzen Satellit in Rotation
  - ständiges Gegensteuern notwendig, aber
    - Satellit rechnet mit metrischer Maßeinheit: Newton
    - Bodenstation rechnet mit amerikanischer Maßeinheit: pound
- ❑ Folge: Satellit verglüht in der Mars-Atmosphäre
  
- ❑ Schaden: 165.000.000 US\$  
Folgen: Image-Verlust der NASA  
Teil eines Mars-Programms nicht durchführbar und nicht nachholbar

## Überprüfen von Software – Beispiele für Probleme

(Fortsetzung)

### Versagen der Patriot-Abwehrrakete

- ❑ Ziel: Abfangen irakischer Rakete im Golf-Krieg
- ❑ Zeit: 1991
- ❑ Probleme:
  - interne 1/10-s-Uhr
  - in Software umgerechnet in 1-s-Zählung mit Division durch 10
  - Betriebszeit über 100 Stunden
  - fortlaufende Rundungsfehler summieren sich zu einer Abweichung von der Realzeit um 0,34 s
  - Anfluggeschwindigkeit: 1700 m/s
- ❑ Folge: 28 Tote, 100 Verletzte



## Überprüfen von Software – Beispiele für Probleme

(Fortsetzung)

### Ariane 5, Flug 501 (Erststart)

- ❑ Ziel: Transport von Satelliten in Erdumlaufbahn
- ❑ Zeit: 1996
- ❑ Probleme:
  - Software von Ariane 4 übernommen
  - unnötige Kalibrierung während des Starts dauert zu lange für die viel schneller beschleunigende neue, stärkere Rakete
  - Overflow in 16 bit-Register führt zu Abschaltung des Navigationssystems
  - Diagnosemeldungen werden von der Steuerung als Flugdaten interpretiert
  - Steuerung berechnet daraus eine nicht kontrollierbare Flugbahn
- ❑ Folge: Selbstzerstörung nach 42 s
- ❑ Schaden: 1.700.000.000 €  
Folge: Image-Verlust der ESA

## Analyse der Beispiele

- ❑ Komplexe Softwareprojekte sind schwer zu beherrschen.
- ❑ Komplexe Softwareprojekte können an vergleichsweise einfachen Details scheitern.
- ❑ Beschreibungen von Anforderungen sind in der Praxis nicht so präzise und widerspruchsfrei, wie man das aus theoretischer Sicht erwarten würde.
- ❑ Beschreibungen von Anforderungen orientieren sich an der während ihrer Erhebung zugrunde gelegten Situation.
- ❑ Die Szenarien, in denen die Software getestet wird, müssen den in der Realität auftretenden Gegebenheiten entsprechen. Das Bestimmen dieser Szenarien ist nicht trivial.
- ❑ Nicht jede Software kann unter realen Bedingungen getestet werden.

## Organisation: Prozessmodelle/Vorgehensmodelle

### allgemeiner Ablauf von Projekten:



## Organisation: Prozessmodelle/Vorgehensmodelle

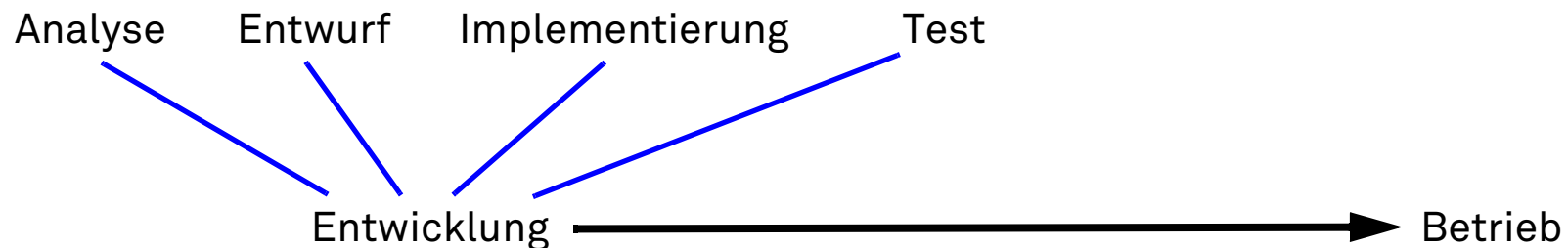
(Fortsetzung)

### allgemeiner Ablauf von Projekten:



### bei Software:

- (Re-)Produktion von Softwareprodukten ist sehr einfach.
- (Software-)Entwicklung beinhaltet auch die »Erstellung« des Endprodukts.
- Terminologie für die Softwareentwicklung:



## Organisation: Prozessmodelle/Vorgehensmodelle

(Fortsetzung)

intuitiver Ansatz: **Wasserfall-Modell** (Royce 1970)

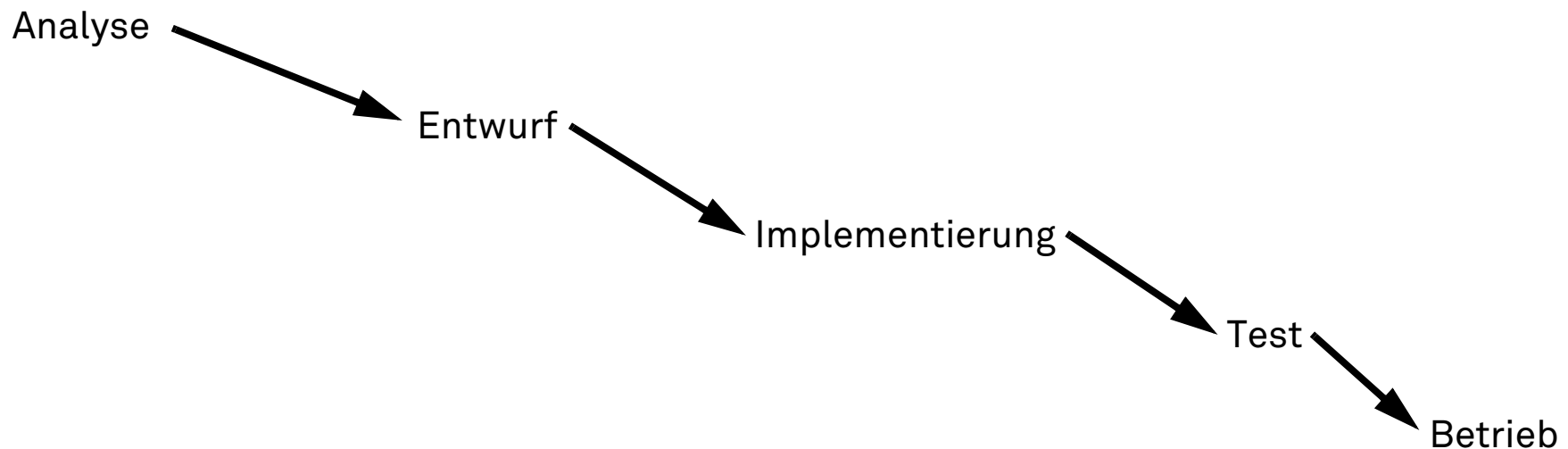


Literatur: Brandt-Pook, Hans; Kollmeier, Rainer: Softwareentwicklung kompakt und verständlich – Wie Softwaresysteme entstehen, S. 1-42  
<http://www.springerlink.com/content/r66585/#section=77668&page=1&locus=0>

## Organisation: Prozessmodelle/Vorgehensmodelle

(Fortsetzung)

intuitiver Ansatz: **Wasserfall-Modell** (Royce 1970)



Beschreibung:

- Alle Tätigkeiten einer Phase werden abgeschlossen, bevor die nächste Phase beginnt.
- Das Softwareprodukt wird in seiner Gesamtheit vollständig weiterentwickelt.
- Es gibt also keine Notwendigkeit für einen Rücksprung in eine frühere Phase.

## Organisation: Prozessmodelle/Vorgehensmodelle

(Fortsetzung)

intuitiver Ansatz: **Wasserfall-Modell** (Royce 1970)

### Vorteile:

- ❑ Die Abläufe sind einfach zu planen.
- ❑ Die Abläufe sind einfach zu überwachen.
- ❑ Das Vorgehen ist ausreichend für kleinere Projekte mit überschaubarer Dauer.

### Nachteile:

- ❑ Das Vorgehen ist unflexibel bei geänderten oder neu auftretenden Anforderungen.
- ❑ Beim Erkennen von Fehlern ist eine Überarbeitung der Ergebnisse vorangehender Phasen nicht vorgesehen.
- ❑ Das Vorgehen ist daher für größere Projekte nicht anwendbar.

## Organisation: Prozessmodelle/Vorgehensmodelle

(Fortsetzung)

intuitiver Ansatz: **Wasserfall-Modell** (Royce 1970)

Vorteile:

- ❑ Die Abläufe sind einfach zu planen.
- ❑ Die Abläufe sind einfach zu überwachen.
- ❑ Das Vorgehen ist ausreichend für kleinere Projekte mit überschaubarer Dauer.

⇒ **Softwarepraktikum**

Nachteile:

- ❑ Das Vorgehen ist unflexibel bei geänderten oder neu auftretenden Anforderungen.
- ❑ Beim Erkennen von Fehlern ist eine Überarbeitung der Ergebnisse vorangehender Phasen nicht vorgesehen.
- ❑ Das Vorgehen ist daher für größere Projekte nicht anwendbar.



## Organisation: Prozessmodelle/Vorgehensmodelle

(Fortsetzung)

Verbesserungen des Wasserfall-Modells:

- ❑ Die Rückkehr in frühere Phasen wird erlaubt.
- ❑ Ein unterschiedlicher Entwicklungsfortschritt für Teile des Projekts wird vorgesehen.
- ❑ Eine geplante schrittweise Vervollständigung des Projekts wird vorgesehen.

⇒ **Alle Verbesserungen führen zu mehr Aufwand für die Projektplanung, Projektsteuerung und Projektüberwachung.**

(Prozessmodelle/Vorgehensmodelle werden in einem später folgenden Teil der Vorlesung noch einmal betrachtet.)

# Folien zur Vorlesung **Softwaretechnik**

## **Abschnitt 4.2: Aktivitätsdiagramme**

## Planung/Visualisierung von Algorithmen

(Fortsetzung)

Vorteile der graphischen Planung von Algorithmen:

- ❑ Die modellierten Abläufe können leicht nachvollzogen werden.
- ❑ Die graphische Darstellungsform unterstützt Gruppenarbeit und Diskussionen.
- ❑ Graphen können schrittweise erweitert werden:  
Zuerst werden Knoten angelegt, die dann geeignet verbunden werden.
- ❑ Fehlende Verbindungen können im Graph unmittelbar erkannt werden.
- ❑ Ein Graph kann formal analysiert werden.

Nachteile der graphischen Darstellung von Algorithmen:

- ❑ Umfangreiche Abläufe können wegen ihres Umfangs nur schwer überblickt werden.
- ❑ *Zyklen* können nur schwer erkannt werden.
- ❑ Abläufe in komplexen Graphen können nur schwer nachvollzogen werden.
- ❑ Die Ableitung von Programmcode aus der graphischen Visualisierung eines Algorithmus ist ein komplexer Vorgang, da unterschiedliche Formen der Umsetzung möglich sind.

## Aktivitätsdiagramm

- ❑ Ein Aktivitätsdiagramm spezifiziert
  - eine Menge von potentiellen Abläufen,
  - die sich unter bestimmten Bedingungen ergeben können.
- ❑ Das zentrale Element der Modellierung ist die **Aktion**.
- ❑ Alle anderen Elemente dienen dazu, Aktionen geeignet zu verknüpfen.
- ❑ Dient ein Aktivitätsdiagramm der **Planung** eines Algorithmus, so enthalten die Aktionen
  - meist keinen Programmcode
  - sondern abstrakt formulierte Handlungsanweisungen.
- ❑ Dient ein Aktivitätsdiagramm der Visualisierung eines Algorithmus, so enthält die einzelne Aktion eine *nicht-unterbrechbare* Folge von Anweisungen.

## Beispiel (*irgendeine* Java-Methode)

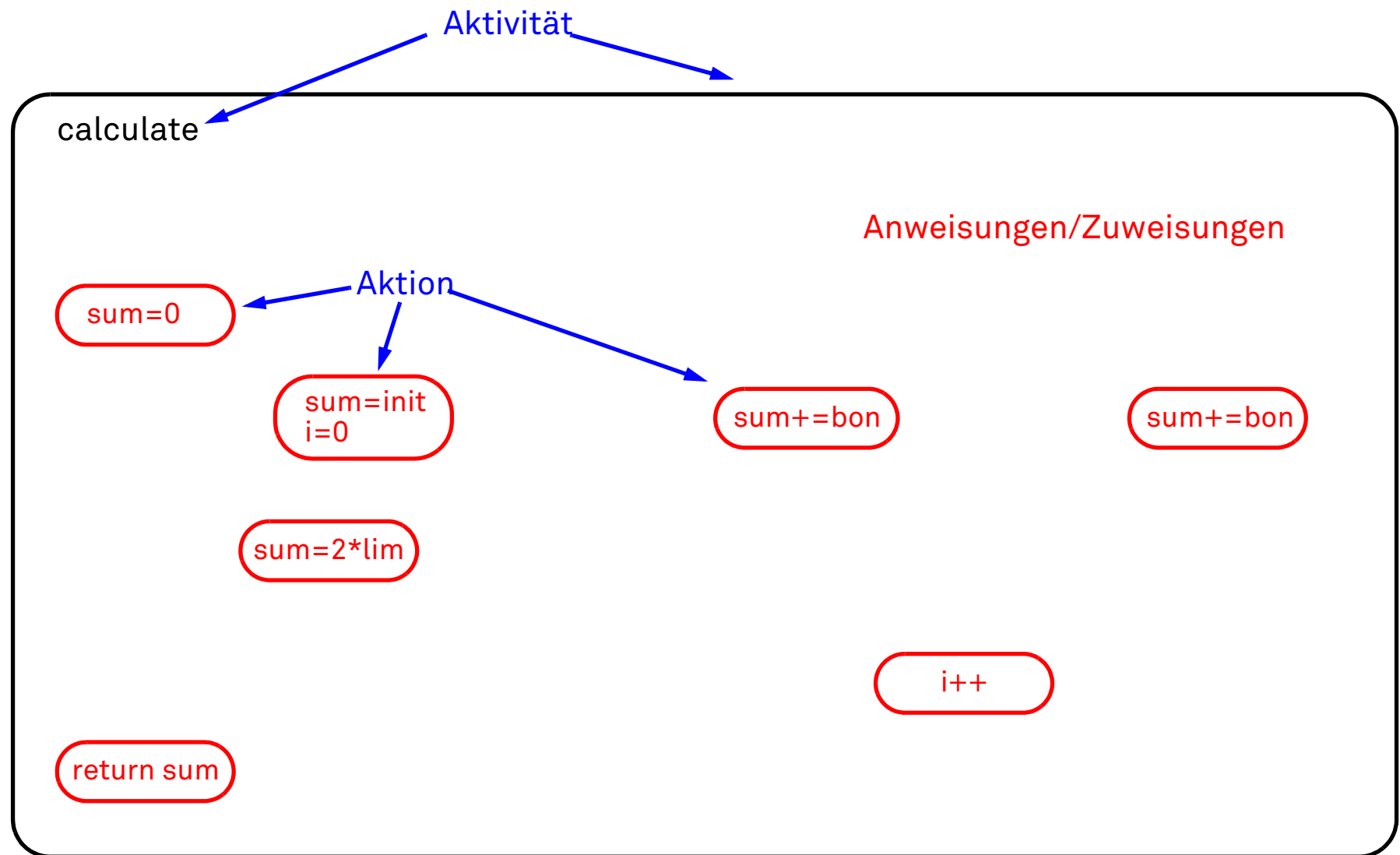
```
int calculate ( int end, int init, int lim, int bon )
{
    int sum = 0;
    if (end > 0)
    {
        sum = init;
        for (int i=0; i < end; i++)
        {
            sum += bon;
            if (sum > lim)
            {
                sum += bon;
            }
        }
        if (sum > 2*lim)
        {
            sum = 2*lim;
        }
    }
    return sum;
}
```

## Beispiel (irgendeine Java-Methode)

(Fortsetzung)

```
int calculate ( int end, int init, int lim, int bon )
{
    int sum = 0;
    if (end > 0)
    {
        sum = init;
        for (int i=0; i < end; i++)
        {
            sum += bon;
            if (sum > lim)
            {
                sum += bon;
            }
        }
        if (sum > 2*lim)
        {
            sum = 2*lim;
        }
    }
    return sum;
}
```

Anweisungen/Zuweisungen



## Beispiel (*irgendeine* Java-Methode)

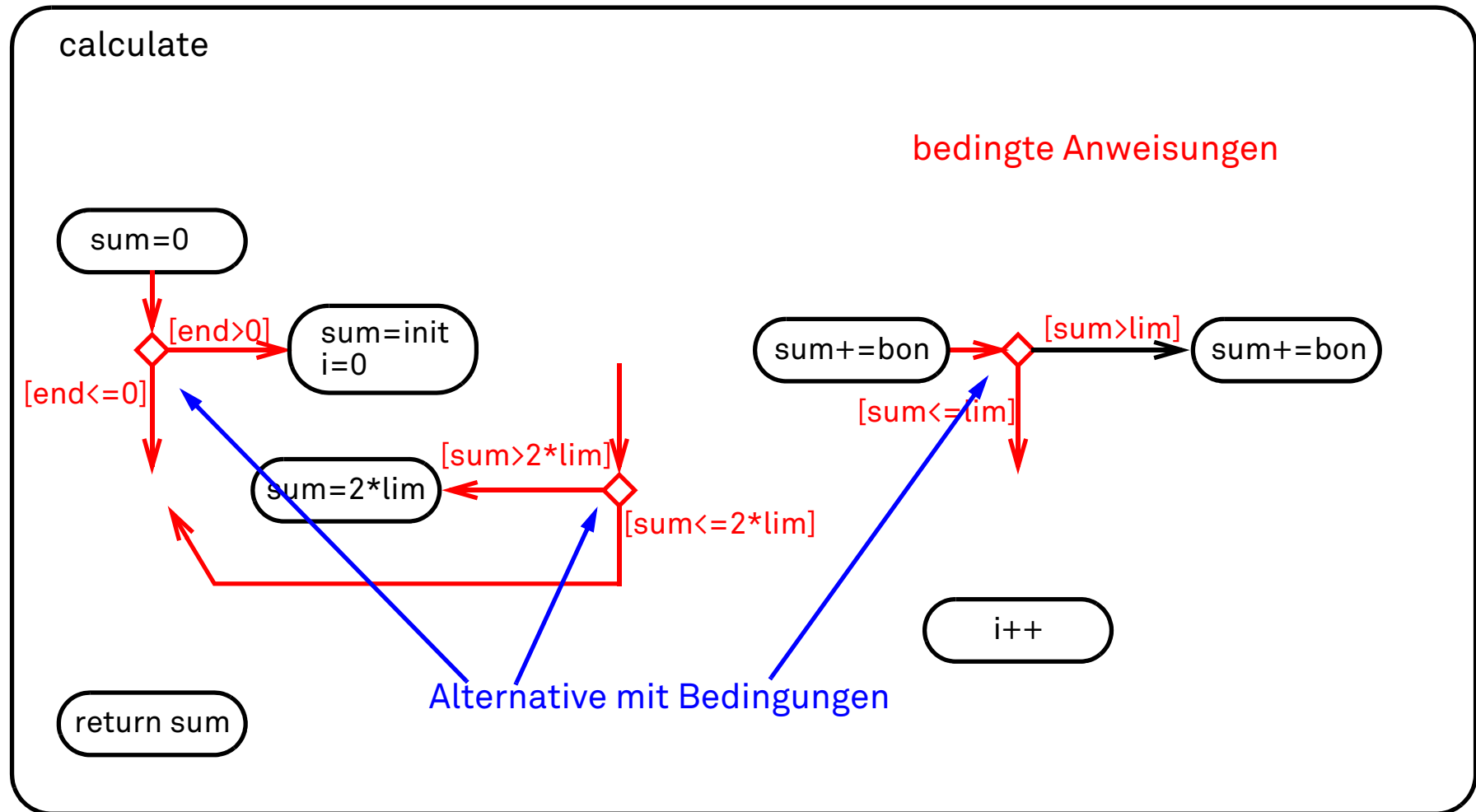
```
int calculate ( int end, int init, int lim, int bon )
{
    int sum = 0;
    if (end > 0)
    {
        sum = init;
        for (int i=0; i < end; i++)
        {
            sum += bon;
            if (sum > lim)
            {
                sum += bon;
            }
        }
        if (sum > 2*lim)
        {
            sum = 2*lim;
        }
    }
    return sum;
}
```

bedingte Anweisungen



## Beispiel (irgendeine Java-Methode)

(Fortsetzung)



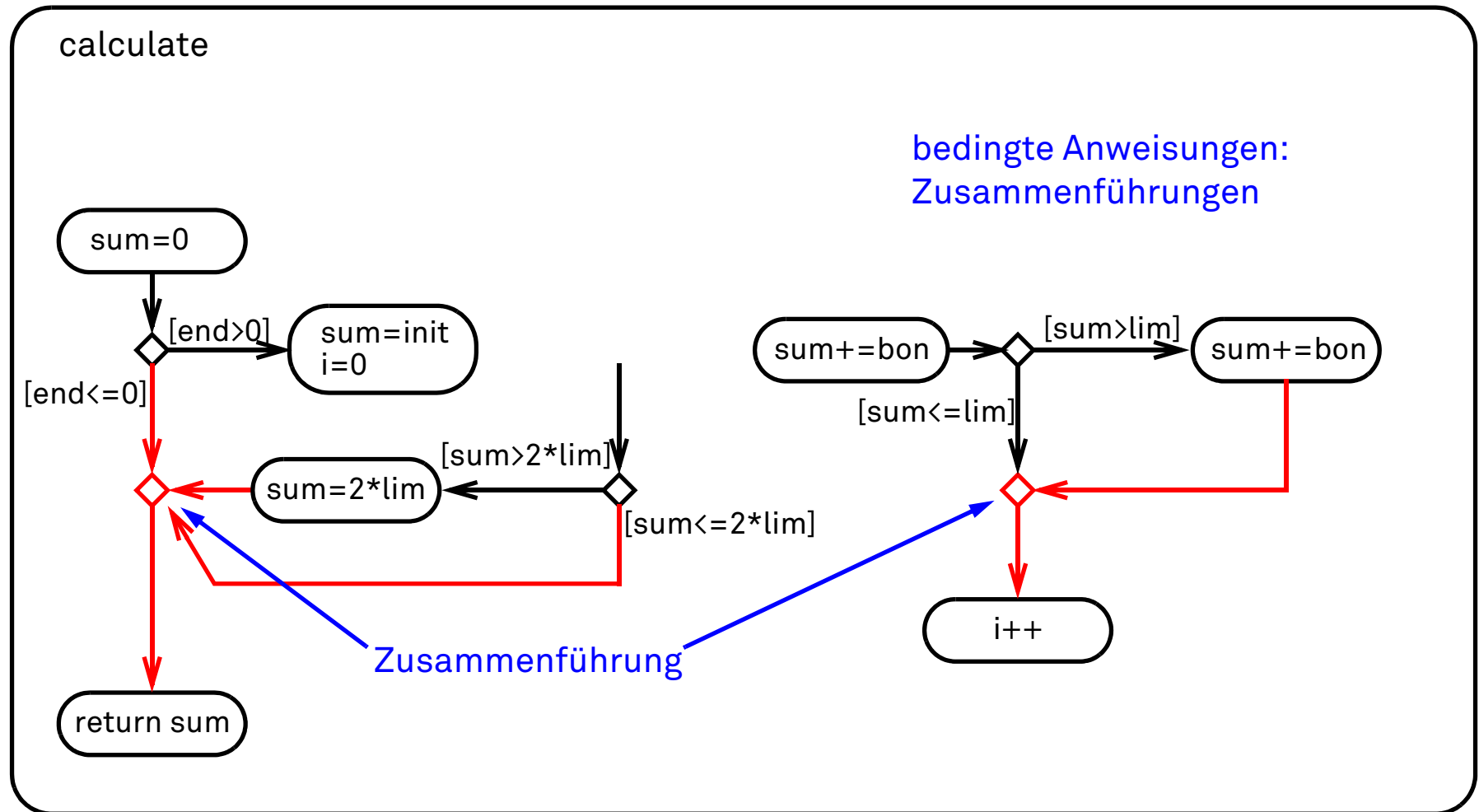
## Beispiel (*irgendeine* Java-Methode)

```
int calculate ( int end, int init, int lim, int bon )
{
    int sum = 0;
    if (end > 0)
    {
        sum = init;
        for (int i=0; i < end; i++)
        {
            sum += bon;
            if (sum > lim)
            {
                sum += bon;
            }
        }
        if (sum > 2*lim)
        {
            sum = 2*lim;
        }
    }
    return sum;
}
```

bedingte Anweisungen:  
Zusammenführungen

## Beispiel (irgendeine Java-Methode)

(Fortsetzung)



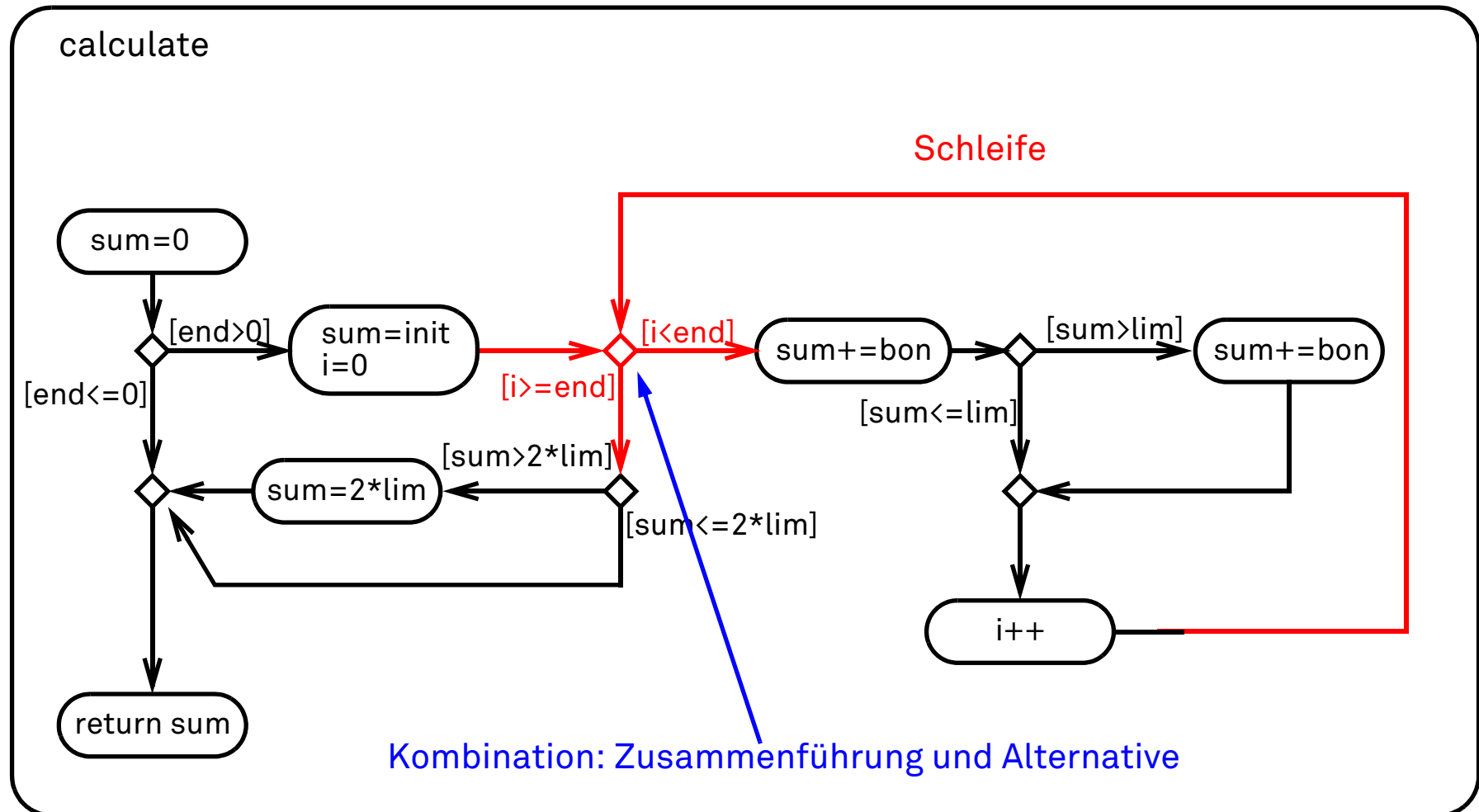
## Beispiel (*irgendeine* Java-Methode)

```
int calculate ( int end, int init, int lim, int bon )
{
    int sum = 0;
    if (end > 0)
    {
        sum = init;
        for (int i=0; i < end; i++)
        {
            sum += bon;
            if (sum > lim)
            {
                sum += bon;
            }
        }
        if (sum > 2*lim)
        {
            sum = 2*lim;
        }
    }
    return sum;
}
```

Schleife

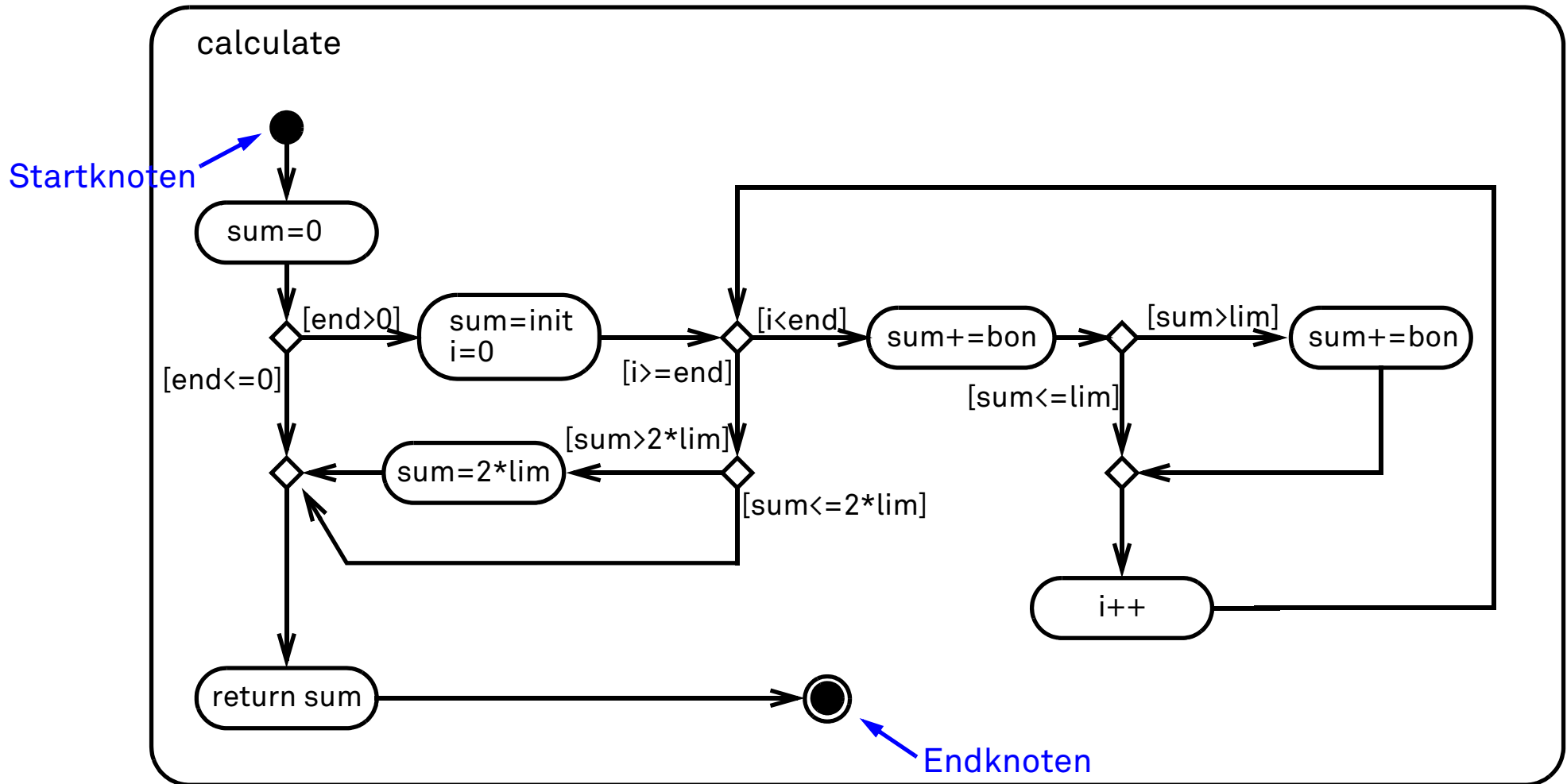
## Beispiel (irgendeine Java-Methode)

(Fortsetzung)



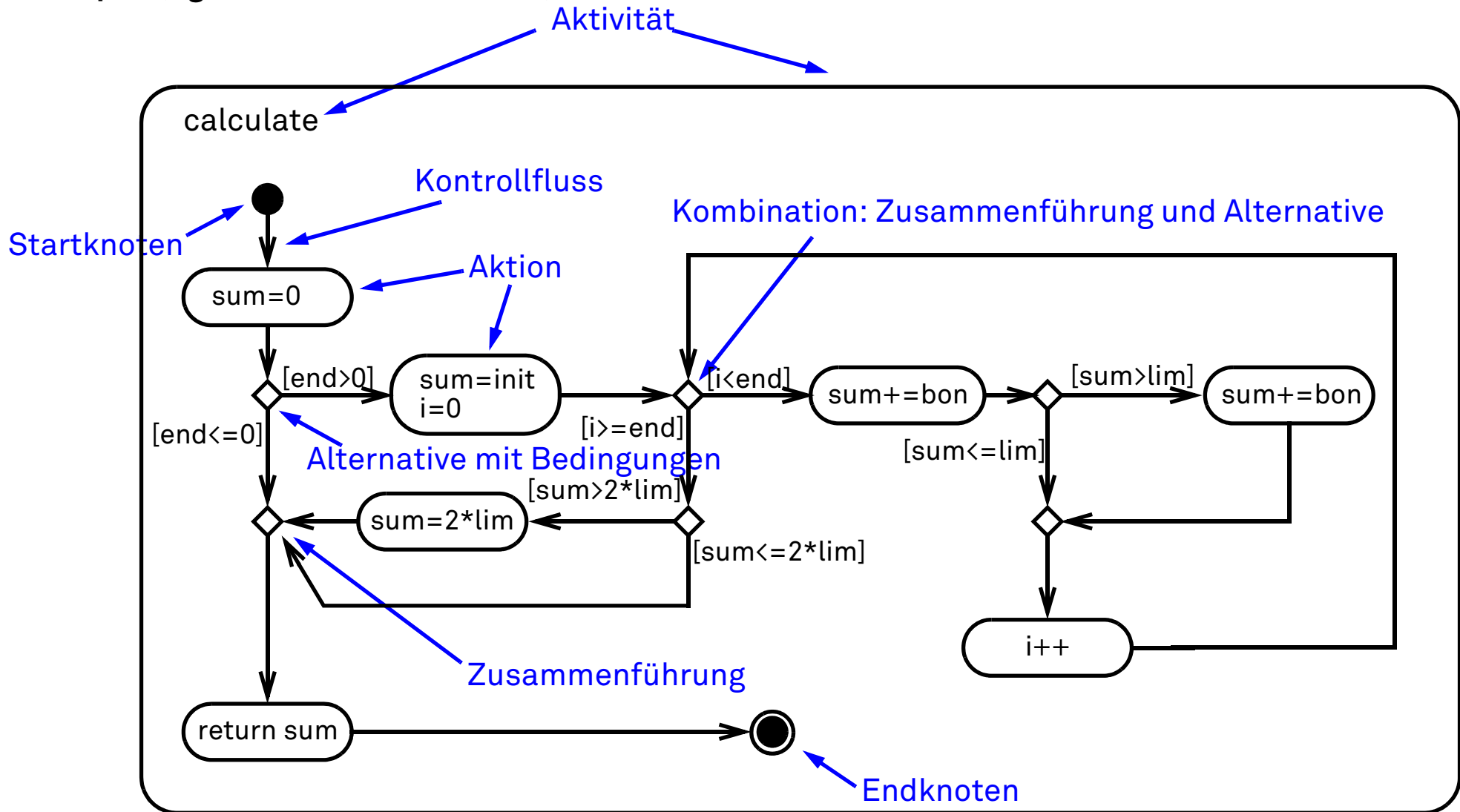
## Beispiel (irgendeine Java-Methode)

(Fortsetzung)



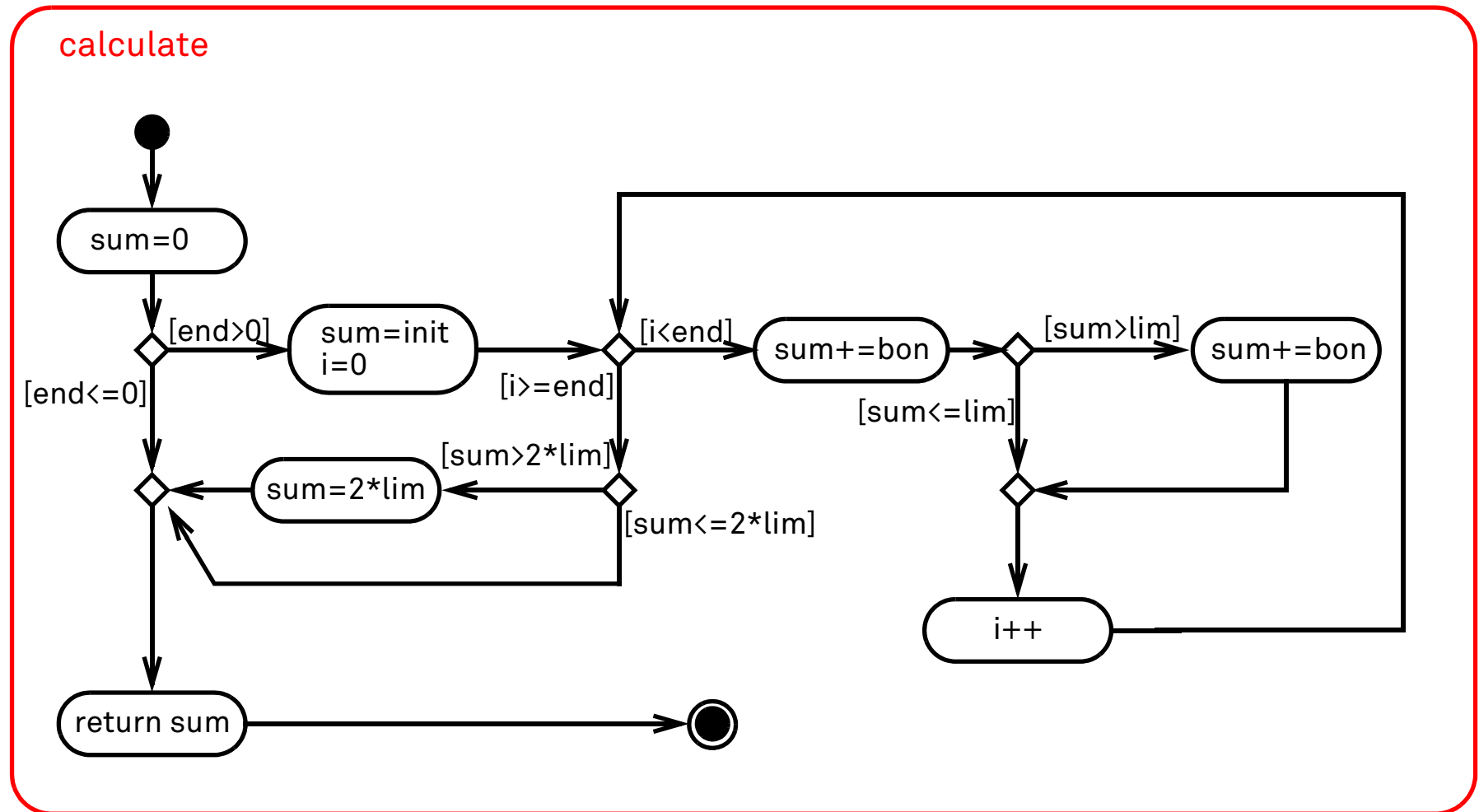
## Beispiel (irgendeine Java-Methode)

(Fortsetzung)



## Aktivität

= Spezifikation eines Verhaltens als koordinierte Folge der Ausführung von Aktionen






## Anmerkungen zu Aktivitätsdiagrammen

- Eine Aktivität kann
  - Eingangsparameter besitzen,
  - Ausgangsparameter besitzen.





- Eine Aktivität kann in anderen Aktivitäten als Aktion verwendet werden.  
(Kennzeichnung durch )

Parameter können veranschaulicht werden.

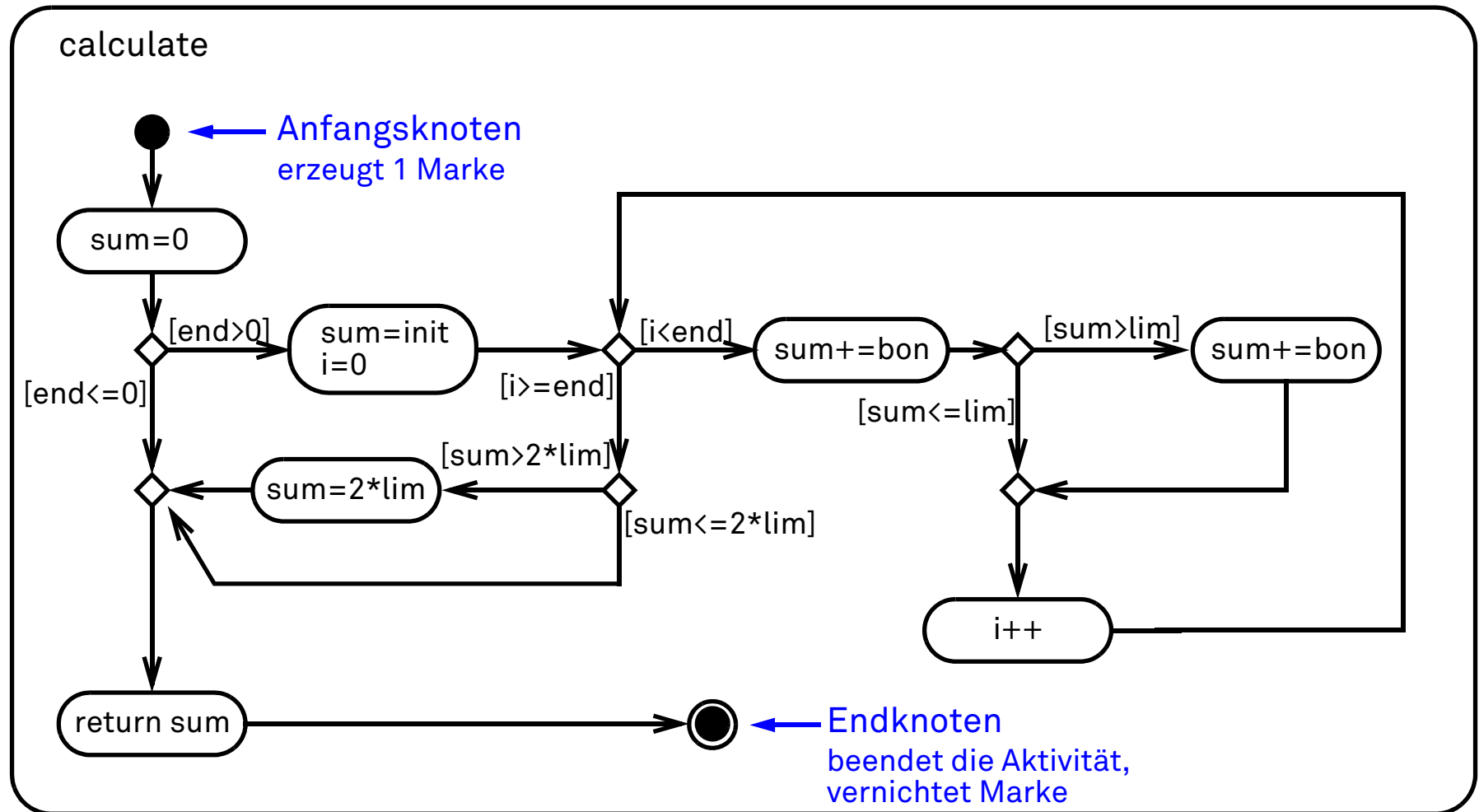


## Anmerkungen zu Aktivitätsdiagrammen

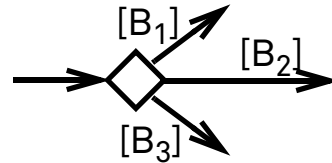
(Fortsetzung)

- ❑ Der Kontrollfluss einer Aktivität wird durch (*fiktive*) *Marken* (engl. *Token*) bestimmt:
  - Eine Marke wandert über die Kontrollfluss-Kanten durch die Aktivität.
  - Eine Marke verweilt in den Aktionen.
  - Eine Marke wechselt **zeitlos** zur nächsten Aktion, sobald dieses möglich ist.
- ❑ Eine Aktivität benötigt mindestens eine Marke, um ausgeführt zu werden.
- ❑ Jeder Startknoten erzeugt eine Marke. 
- ❑ Ein Eingangsparameter überführt eine Marke in die Aktivität.
- ❑ Die Ausführung einer Aktivität endet, wenn keine Marke mehr vorhanden ist.
- ❑ Wird ein *Endknoten* der Aktivität von einer Marke erreicht,  so wird die Marke vernichtet.
- ❑ Ein Ausgangsparameter überführt eine Marke aus der Aktivität in die umgebende Aktivität.

## bekanntes Beispiel

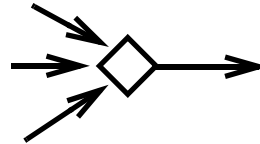


## Verzweigungsknoten

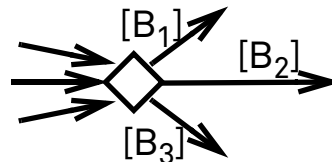


- ❑ Ein Verzweigungsknoten spaltet **eine** Eingangskante in **mehrere alternative** Ausgangskanten auf.
- ❑ Eine Bedingung  $B_i$  an einer Ausgangskante formuliert die Anforderungen, die das Passieren dieser Kante erlauben.
- ❑ Die Bedingungen der Ausgangskanten **müssen sich gegenseitig** ausschließen.
- ❑ Die Bedingungen aller Ausgangskanten **müssen alle möglichen** Fälle abdecken.
- ❑ An einem Entscheidungsknoten kann daher immer **eindeutig** entschieden werden, auf welcher Ausgangskante ein Ablauf fortgesetzt wird.
- ❑ Die Bedingung **[else]** ist erlaubt und vereinfacht das Einhalten der Regeln.
- ❑ Wirkungsweise:  
Auf der Eingangskante trifft immer genau eine Marke ein, die den Verzweigungsknoten genau auf der eindeutig bestimmten Ausgangskante verläßt, deren Bedingung  $B_i$  wahr ist.
- ❑ Durch einen Verzweigungsknoten werden keine Marken erzeugt oder vernichtet.

## Verbindungsknoten



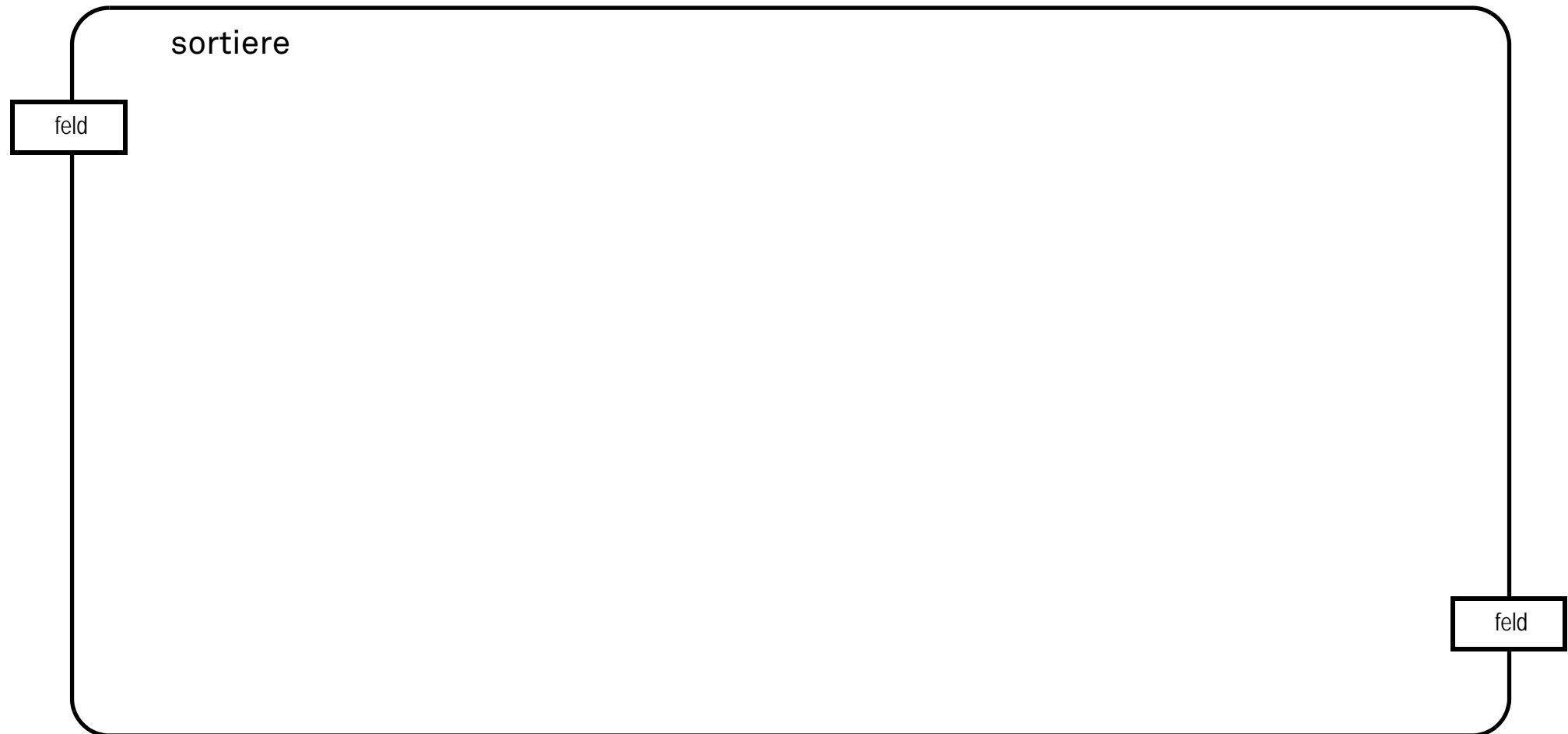
- ❑ Ein Verbindungsknoten führt mehrere Eingangskanten zu einer Ausgangskante zusammen.
- ❑ Die auf einer der Eingangskanten ankommende Marke wird auf die **einzigste** Ausgangskante weitergeleitet.
- ❑ Durch einen Verbindungsknoten werden keine Marken erzeugt oder vernichtet.
- ❑ Die Kombination aus Verzweigungs- und Verbindungsknoten ist möglich und muss die Eigenschaften beider Knotentypen besitzen.



## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

Aufgabe: Sortieren eines Feldes

(noch unvollständig; **kein gültiges Aktivitätsdiagramm**, da Aktionen fehlen)

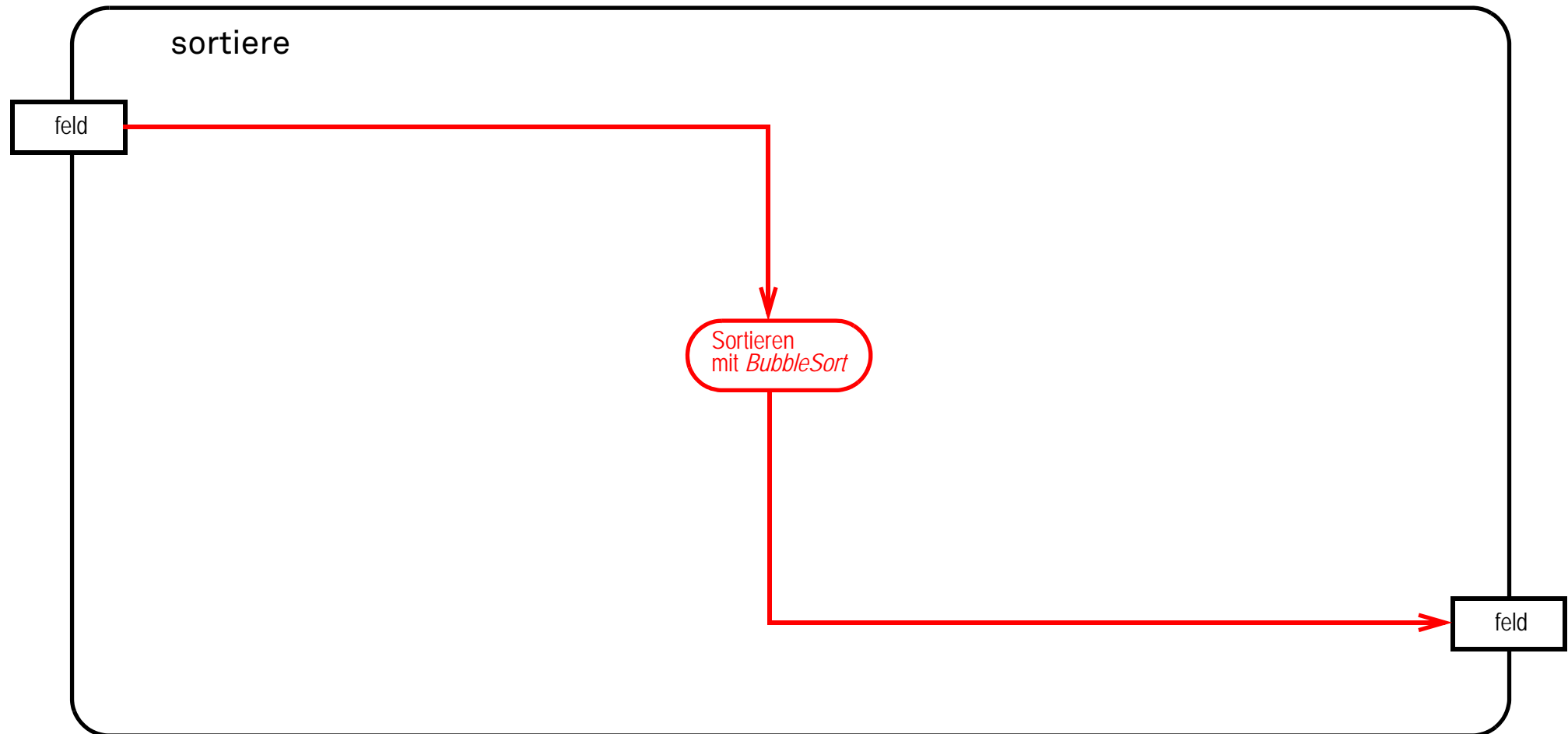


## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(**gültiges Aktivitätsdiagramm**, sehr abstrakt)

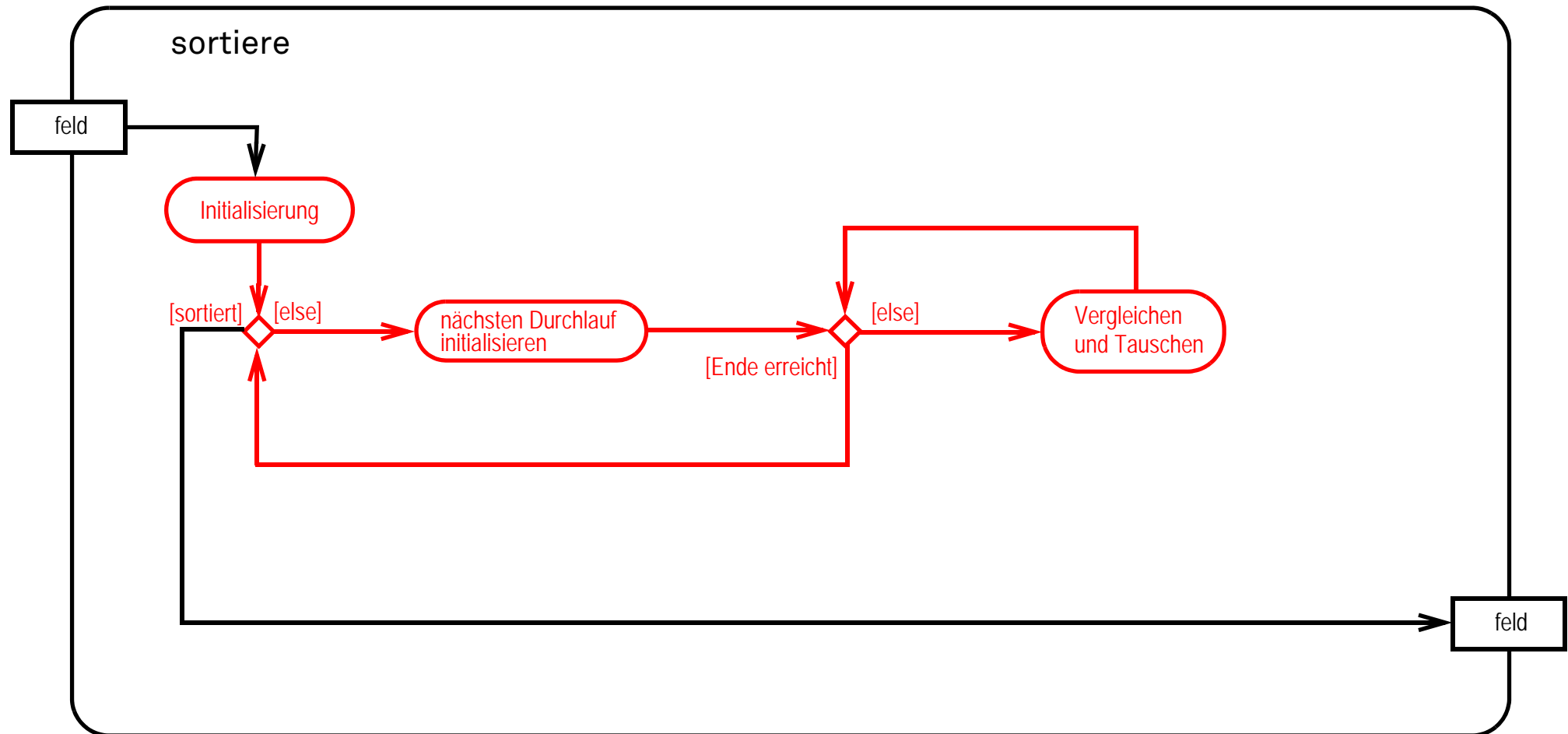


## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(Aktivitätsdiagramm, abstrakt)



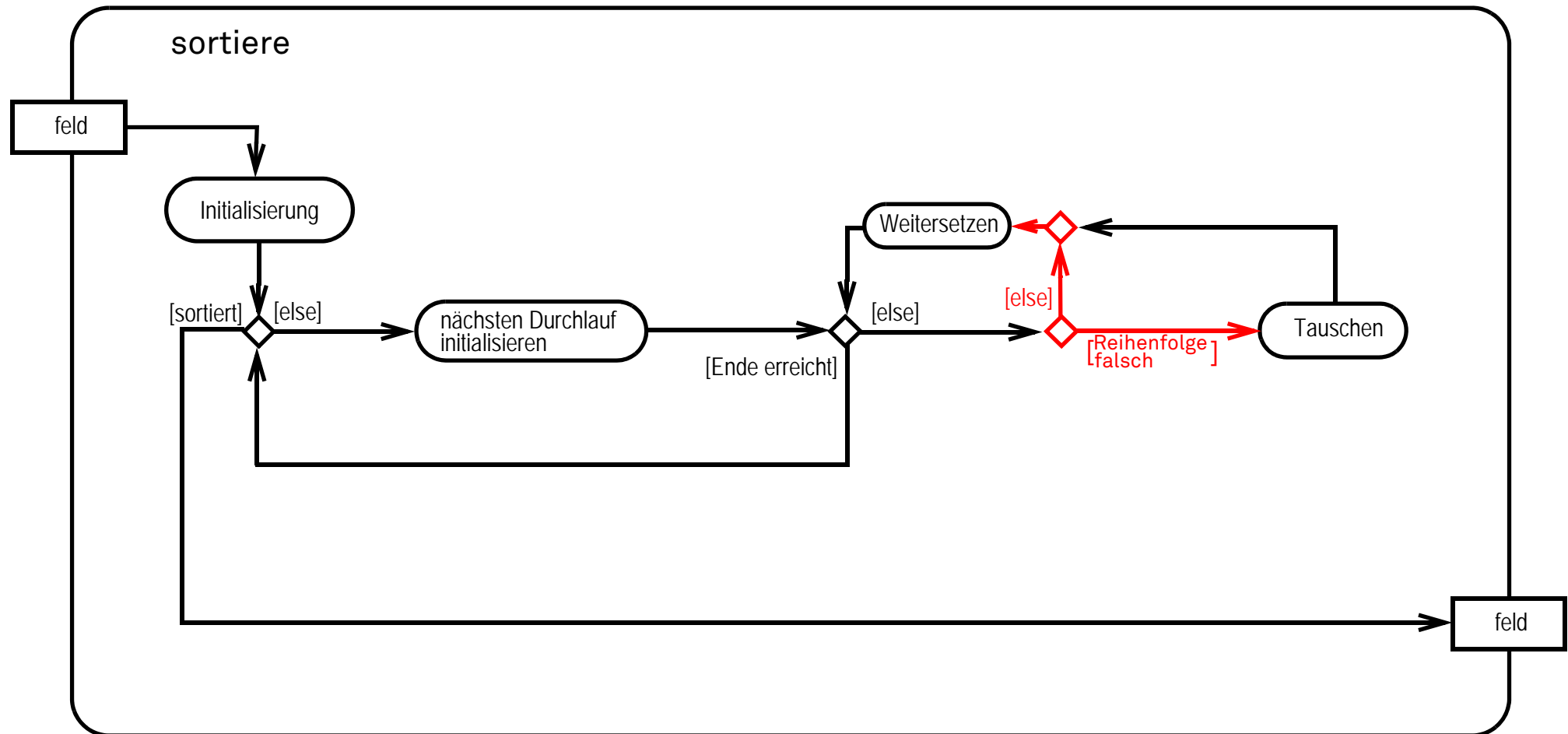


## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(gültiges Aktivitätsdiagramm, Ablauf genau beschrieben)

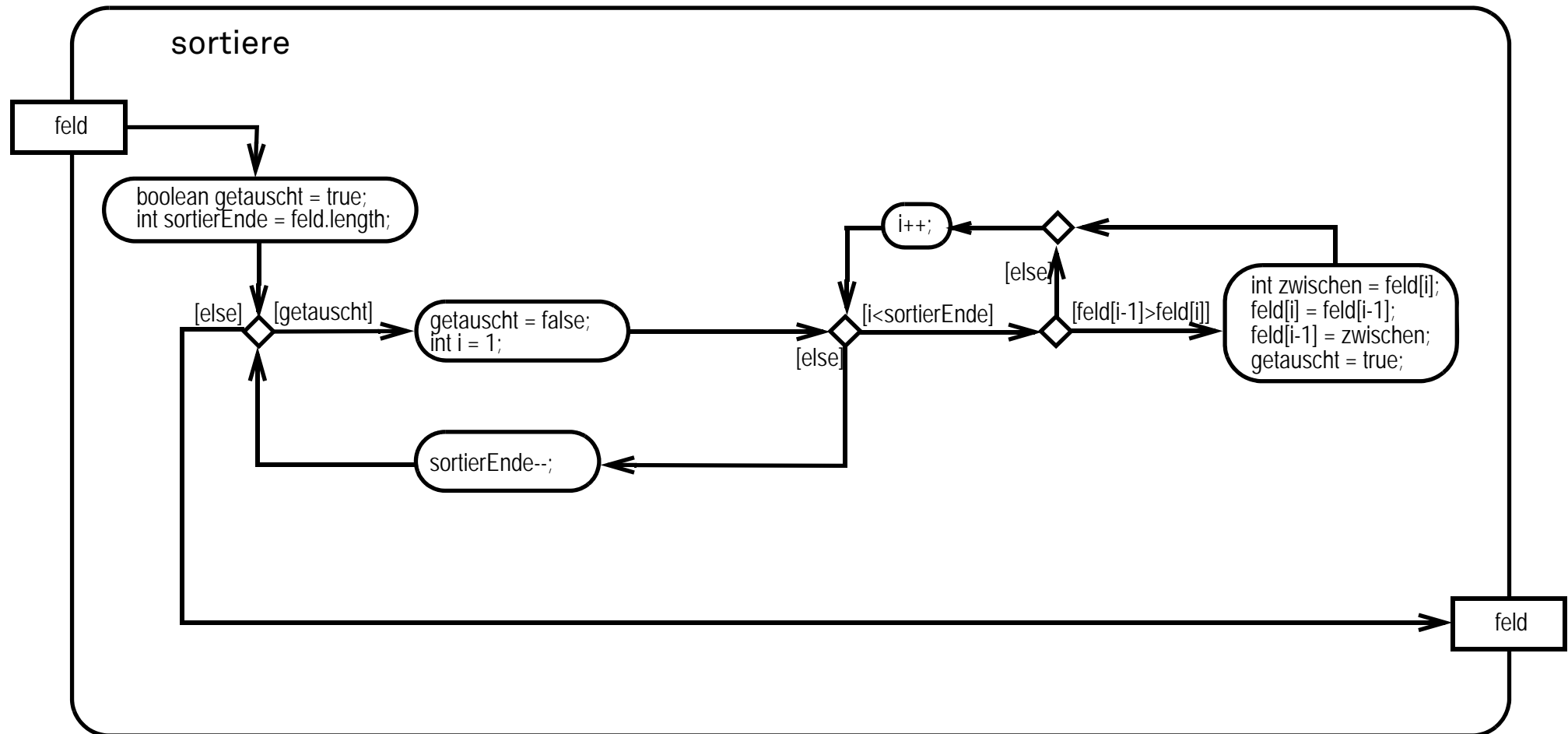


## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(**gültiges Aktivitätsdiagramm**, Aufgabe von Aktionen als Java-Code)

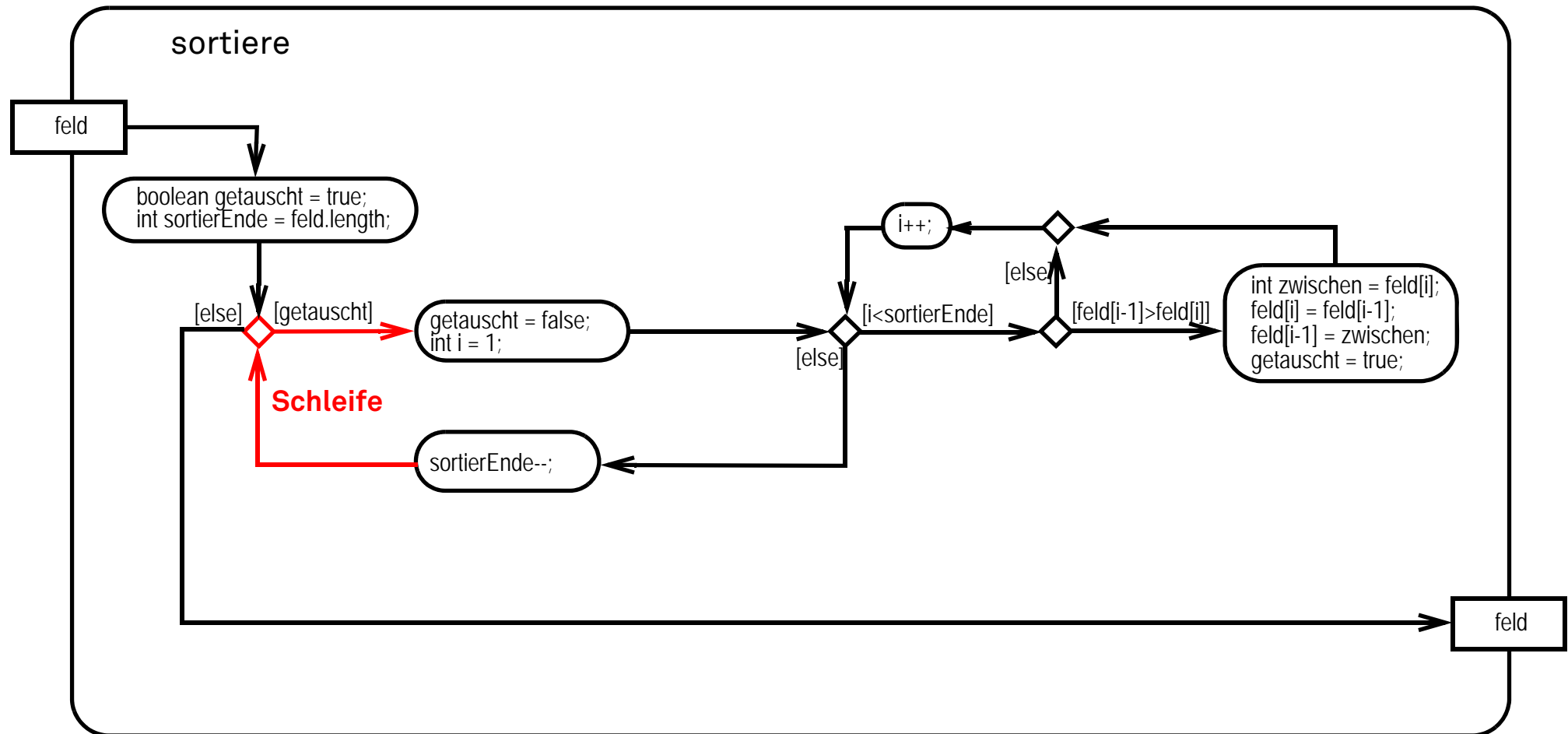


## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(**gültiges Aktivitätsdiagramm**, Aufgabe von Aktionen als Java-Code)

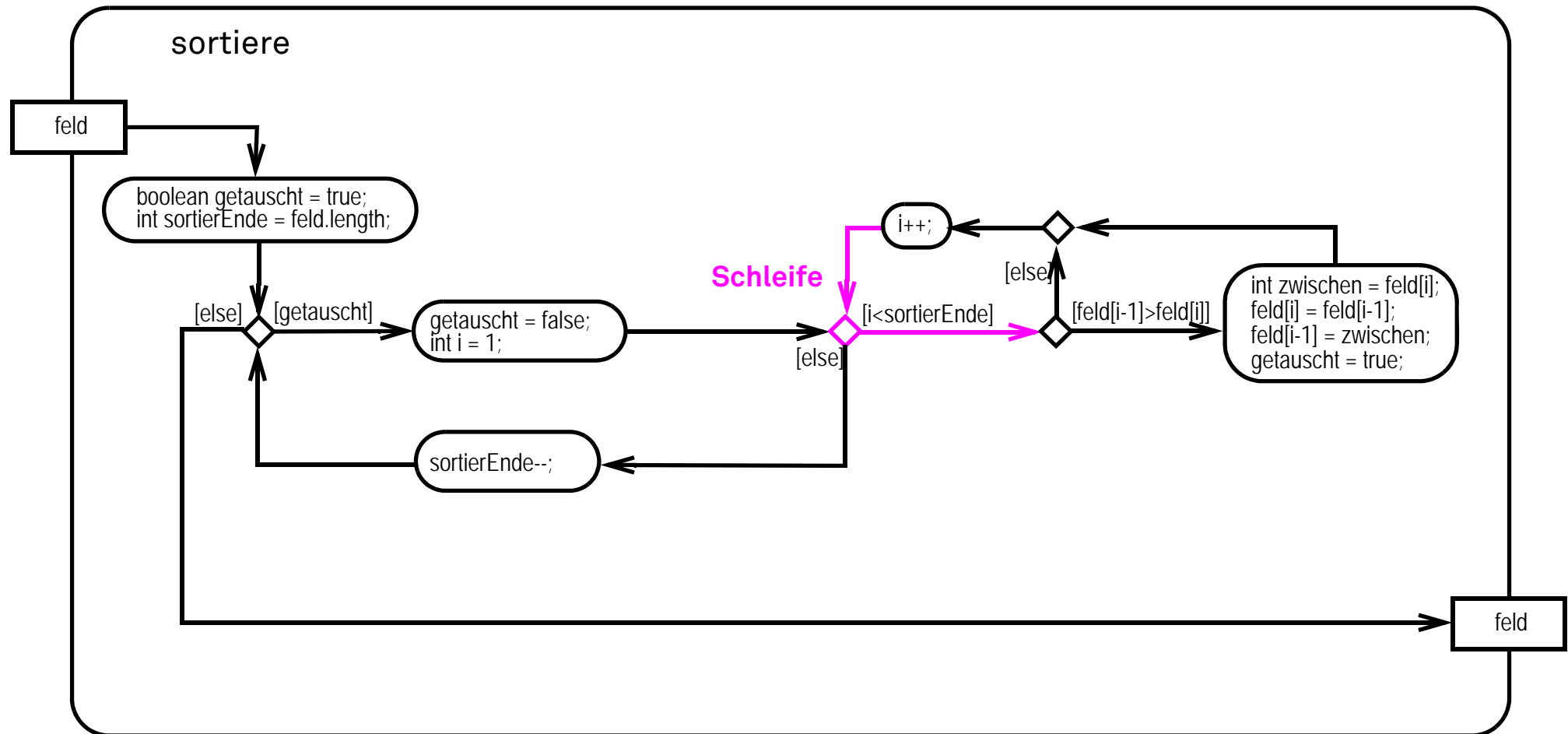


## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(**gültiges Aktivitätsdiagramm**, Aufgabe von Aktionen als Java-Code)

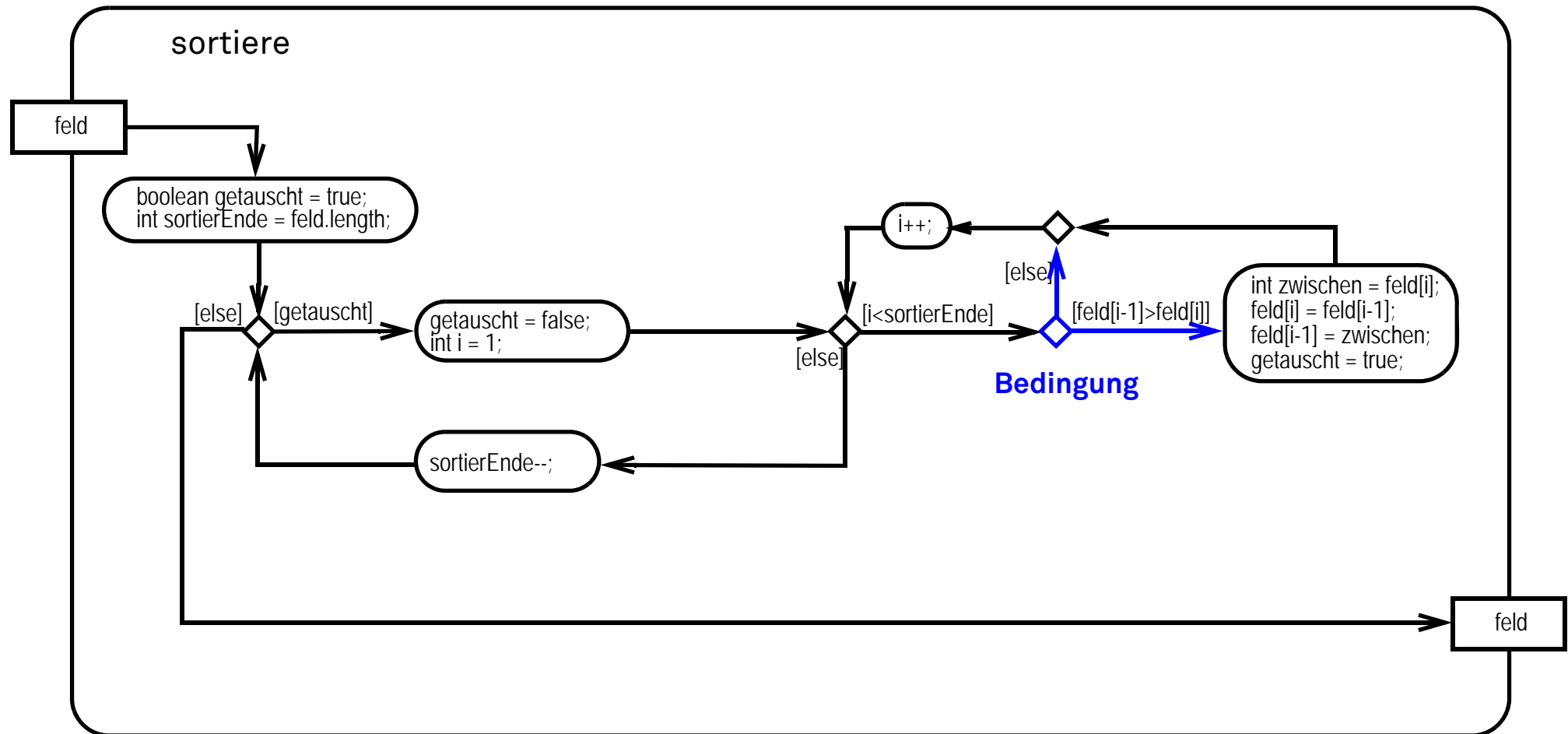


## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(gültiges Aktivitätsdiagramm, Aufgabe von Aktionen als Java-Code)



## Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Umsetzung in Java-Code)

```
int[] sortieren(int[] feld) {  
    boolean getauscht = true;  
    int sortierEnde = feld.length;  
    while (getauscht) {  
        getauscht = false;  
        for (int i=1; i<sortierEnde; i++) {  
            if (feld[i-1]>feld[i]) {  
                int zwischen = feld[i];  
                feld[i] = feld[i-1];  
                feld[i-1] = zwischen;  
                getauscht = true;  
            }  
        }  
        sortierEnde--;  
    }  
    return feld;  
}
```

## Zusammenfassung

### Aktivitätsdiagramme

dienen zur Beschreibung von Abläufen als Folgen von Aktionen,

### Vorteile der Nutzung von Aktivitätsdiagrammen:

- ❑ Es sind unterschiedliche Stufen von Detaillierung und Abstraktion möglich.
- ❑ Die visuelle Überprüfung von Abläufen ist möglich.
- ❑ Die reduzierte Syntax der Diagramme erlaubt eine einfache Übertragung in eine formalisierte (und damit analysierbare) Beschreibung.
- ❑ Auch Java-Methoden lassen sich als Aktivitätsdiagramm modellieren.

### Grenzen der Visualisierung:

- ❑ Es werden i.d.R. nur Abläufe dargestellt.  
Objektorientierte Konstrukte lassen sich nur ungenau darstellen.
- ❑ Der Umgang mit umfangreichen Diagrammen ist schwierig.

# Folien zur Vorlesung **Softwaretechnik**

## **Abschnitt 4.3: Überblick Testen**



## Testen – Überblick

- ❑ Testen ist  
das Ausführen einer Programmkomponente,
  - um nachzuweisen, dass sich diese Programmkomponente wie gewünscht verhält,
  - und**
  - um Fehler aufzudecken und zu lokalisieren.
  
- ❑ "demonstratives" Testen  
= Tester steht der Software positiv gegenüber und  
versucht, die Einsetzbarkeit zu zeigen:  
typisches Verhalten von Entwicklern
  
- ❑ "destruktives" Testen  
= Tester steht Software neutral oder negativ gegenüber und  
versucht, Fehler zu provozieren:  
**wünschenswertes** Verhalten für Tester

## Testen – Überblick

(Fortsetzung)

### Testen

- ❑ kann **vor** der Implementierung vorbereitet werden,
- ❑ erfolgt schon **während** der Implementierung,
- ❑ muss immer auch **nach** Abschluss der Implementierung erfolgen.

### Testen

- ❑ benötigt in der Regel **25–40%** des Entwicklungsaufwands,
- ❑ kann bei kritischen Systemen noch erheblich umfangreicher sein,
- ❑ ist also eine sehr aufwändige Tätigkeit während der Entwicklung!

### Beobachtungen:

- ❑ Implementierungen enthalten fast immer Fehler.
- ❑ Fehler verursachen Schäden: finanziell, technisch, lebensbedrohlich.
- ❑ Software sollte möglichst fehlerfrei in Betrieb gehen.

**=> Testen ist immer notwendig**

## Testen während der Implementierung

- ❑ **Programmtest**  
= Test einzelner Methoden einer Klasse, erfordert:
  - **Treiber (Driver)**, um die Methoden mit Parametern aufzurufen
  - **Platzhalter (Stub)**, um aufgerufene Methoden zu simulieren,  
die noch nicht implementiert oder noch nicht getestet sind.
  
- ❑ **Klassentest**  
= Test aller Methoden einer Klasse, erfordert:  
**Treiber (Driver)** und **Platzhalter (Stub)**,  
um aufgerufene Methoden anderer Klassen zu simulieren.
  
- ❑ Verminderung des Testaufwands ist möglich durch **bottom-up**-Vorgehen:  
zuerst Methoden testen, die ohne Platzhalter auskommen,  
dann Methoden testen, die ausschließlich bereits getestete  
Methoden nutzen  
  
**=> schrittweises Vorgehen,  
das möglicherweise ganz ohne Platzhalter auskommen kann**

## Testen nach Abschluss der Implementierung

- ❑ Komponententest
  - = Überprüfung der Zusammenarbeit von Klassen
    - auch Überprüfung der Korrektheit von Vererbungshierarchien:  
Sichtbarkeit von Methoden, korrektes Verwenden von Polymorphie
- ❑ Integrationstest
  - = Testen mit schrittweisem Zusammenfügen der Komponenten
    - Big-Bang-Integration (führt zu Problemen bei der Fehlerlokalisierung)
    - strukturierte Integration: Bottom-Up, Top-Down, Outside-In
  - zusätzliche Formen von Tests:
    - Akzeptanztest (mit Anwendern, um die Benutzbarkeit zu prüfen)
    - Belastungstest (um die Leistungsfähigkeit zu prüfen)
    - Kompatibilitätstests
    - Installationstests
- ❑ Abnahmetest
  - = Testen durch Auftraggeber (formaler Schritt beim Übergang zum Auftraggeber)

## Programmtest/Klassentest – Probleme und Techniken

Beispiel:

```
class Product {  
    private int att;  
    public Product(int p) { ... }  
    public int calculate(int p1, int p2) { ... }  
}
```

## Programmtest/Klassentest – Probleme und Techniken

(Fortsetzung)

Beispiel:

```
class Product {  
    private int att;  
    public Product(int p) { ... }  
    public int calculate(int p1, int p2) { ... }  
}
```

**Mit welchem Aufruf sollte getestet werden?**

```
Product obj = new Product(7);  
obj.calculate(1,3);  
obj.calculate(-11,-8);  
obj.calculate(0,0);
```



## Programmtest/Klassentest – Probleme und Techniken

(Fortsetzung)

Beispiel:

```
class Product {  
    private int att;  
    public Product(int p) { ... }  
    public int calculate(int p1, int p2) { ... }  
}
```

Voraussetzungen für das Testen sind:

- ❑ Es muss Wissen über die Aufgaben der zu testenden Methode vorhanden sein.
- ❑ Es muss Wissen über das Verhalten der zu testenden Methode vorhanden sein mit
  - Bedeutung der Parameterwerte für die Ausführung
  - Bedeutung der Attributwerte für die Ausführung
  - Bedeutung von anderen Objekten für die Ausführung,  
= Bedeutung des Systemzustands für die Ausführung.

## Programmtest/Klassentest – Probleme und Techniken

(Fortsetzung)

Beispiel:

```
class Product {  
    private int att;  
    public Product(int p) { ... }  
    public int calculate(int p1, int p2) { ... }  
}
```

Voraussetzung für das Testen ist das Vorliegen

- ❑ einer funktionalen Spezifikation
- ❑ oder mindestens einer Wertetabelle,  
die Kombinationen von Parameter- und Zustandswerten  
die zugehörigen Ergebnisse zuordnet,
- ❑ oder einer Implementierung mit aussagekräftigen Bezeichnern,  
einer geeigneten Typisierung und hilfreichen Kommentaren.



## Testfall

### Testfall

= Situationsbeschreibung, die die Überprüfung einer spezifizierten Eigenschaft des Testobjekts ermöglicht

### Beschreibung eines Testfalls umfasst:

- ❑ Vorbedingungen: Systemzustand, der vor dem Testen hergestellt werden muss
- ❑ Eingaben: z.B. Parameterwerte für die zu testende Methode
- ❑ Handlungsfolge bei der Durchführung des Tests:  
z.B. eine notwendige Folge von Methodenaufrufen
- ❑ die erwarteten Ausgaben/Reaktionen auf die Ausführung (Sollwerte)
- ❑ Nachbedingungen: erwarteter Systemzustand nach dem Testen

### Zielsetzungen von Testfällen:

- ❑ Positiv-Test = Test mit gültigen Vorbedingungen und Eingaben
- ❑ Negativ-Test (Robustheitstest)  
= Test mit ungültigen Vorbedingungen oder ungültigen Eingaben

## Anzahl der auszuführenden Testfälle im Rahmen einer Softwareentwicklung

- ❑ erschöpfender Test:  
Ausführen aller möglichen Testfälle, zeigt die vollständige Korrektheit  
(– ist aber nur in wenigen Fällen bei einer endlichen Zahl von Systemzuständen/Eingaben möglich)
- ❑ idealer Test:  
Ausführen von genau so vielen Testfällen, dass alle enthaltenen Fehler gefunden werden  
(– ist aber unmöglich, da die Fehler vorher bekannt sein müssten)
- ❑ Stichprobentest:  
Ausführen einer endlichen Zahl von Testfällen  
(– realistische Alternative, die aber nie die Korrektheit zeigen kann)

==> Testen führt also (nur) zum Aufdecken von Fehlern,  
schafft Vertrauen in die Korrektheit,  
aber beweist nur in seltenen Fällen die Abwesenheit von Fehlern

## Auswahl geeigneter Testfälle

Ziel:

- ❑ mit möglichst wenigen Testfällen (kleine Stichprobe = wenig Aufwand)
- ❑ möglichst viele Fehler finden bzw. ausschließen und
- ❑ zugleich ein möglichst großes Vertrauen in die Software gewinnen

Annahme dabei ist:

alle weiteren – nicht ausgeführten – Testfälle  
würden mit hoher Wahrscheinlichkeit  
keine weiteren Fehler aufdecken

Wie kann so eine Menge von Testfällen bestimmt werden?