

Folien zur Vorlesung **Softwaretechnik**

Fakultät für Informatik
Sommersemester 2018

Stefan Dissmann

Inhaltsverzeichnis

Teil 1: Einführung	003
1.1: Überblick	003
1.2: Ideen der objektorientierten Softwaregestaltung	024
1.3: Objektorientierte Konzepte in Java	028
Teil 2: UML – Unified Modeling Language	049
2.1: Überblick	049
2.2: Klassendiagramme/Paketdiagramme	053
2.3: Sequenzdiagramme	107
Teil 3: Entwurfsmuster	141
Teil 4: Überprüfen von Software	333
4.1: Motivation	333
4.2: Aktivitätsdiagramme	347
4.3: Überblick Testen	377
4.4: Funktionsorientierter Test (Black-Box-Test)	389
4.5: Strukturorientierter Test (White-Box-Test)	409
4.6: Testunterstützung durch JUnit	466
Teil 5: Vorgehensmodelle	482
5.1: Motivation	482
5.2: Analyse	493
5.3: Anwendungsfalldiagramme	516
5.4: Aktivitätsdiagramme (Ergänzung zu Teil 4.2)	527
5.5: Zusammenfassung Analyse	550
5.6: Übergang Analyse – Architekturentwurf	558
5.7: Übergang Analyse – Grobentwurf	575
5.8: Agile Vorgehensmodelle	601
Teil 6: Zusammenfassung	616

Folien zur Vorlesung **Softwaretechnik**

Teil 1: Einführung **Abschnitt 1.1: Überblick**

Softwaretechnik

ist die Disziplin in der Informatik, die sich beschäftigt mit

- Planung,
- Konstruktion,
- Erstellung,
- Überprüfung,
- Einführung,
- Betrieb und
- Wartung

von Softwaresystemen (Programmen).

Zielsetzungen:

- ❑ funktional geeignete Software herstellen
- ❑ qualitativ gute Software herstellen:
führt meist auch zu kostengünstigem Betrieb
- ❑ nachhaltig funktionale Eignung und Qualität sicherstellen:
führt meist auch zu kostengünstiger Wartung
- ❑ Einhalten des geplanten Entwicklungsaufwands

Zielsetzung: nachhaltig funktional geeignete Software

Die Funktionalität einer Software hängt ab von

- ❑ Anwendungsbereich und
- ❑ Anwendungsumfeld.

Eignung wird dadurch sichergestellt, dass

- ❑ Anwendungsbereich und
- ❑ Anwendungsumfeld

im Rahmen der Entwicklung analysiert und interpretiert werden.

Nachhaltigkeit wird dadurch sichergestellt,
dass Prognosen zu möglichen Änderungen an

- ❑ Anwendungsbereich und
- ❑ Anwendungsumfeld

in die Analyse einbezogen

und während der Softwarekonzeption technisch berücksichtigt werden.

Zielsetzung: nachhaltig qualitativ gute Software

Qualitätsmerkmale

sind Aspekte, die in ihrer Gesamtheit die Qualität eines Softwareprodukts ausmachen.

Qualität von Software

wird aus verschiedenen Perspektiven unterschiedlich wahrgenommen:

- andere Gewichtung der Merkmale,
- andere mögliche Ausprägungen der Merkmale,
- andere Maße zum Beschreiben und Prüfen der Merkmale

Perspektiven sind:

Benutzung, Entwicklung, Administration, Betrieb, Beschaffung, ...

Literatur: Liggesmeyer, Peter: Software-Qualität – Testen, Analysieren und Verifizieren von Software, S. 1-48

http://link.springer.com/chapter/10.1007/978-3-8274-2203-3_1

Hoffmann, Dirk W.: Software-Qualität, S. 1-26

http://link.springer.com/chapter/10.1007/978-3-540-76323-9_1

Qualitätsmerkmale (Beispiele)

- ❑ Zuverlässigkeit
= korrektes Verhalten der Software über die Zeit (Fehler je Zeiteinheit)
- ❑ Robustheit/Fehlertoleranz
= Verhalten der Software bei unerwarteten Informationen aus der Umgebung
- ❑ Performanz
= Ausführungsschnelligkeit der Software
- ❑ Effizienz
= Wirtschaftlichkeit, mit der eine Software ihre Aufgaben erfüllt
- ❑ Skalierbarkeit
= Anpassung der Software an die Betriebsrealität
- ❑ Wartbarkeit
= Anpassung der Software an zukünftige – teilweise unbekannte – Anforderungen
- ❑ Interoperabilität
= Kooperationsfähigkeit mit anderer Software
- ❑ Portabilität
= Übertragbarkeit auf verschiedene Plattformen
- ❑ Bedienbarkeit/Verständlichkeit
= Brauchbarkeit für den Anwender

Anmerkungen:

- ❑ Unterscheidung möglich:
 - *interne* Qualitätsmerkmale beziehen sich auf die Implementierung der Software
 - *externe* Qualitätsmerkmale beziehen sich auf die Nutzung der Software

- ❑ Qualitätsmerkmale können sich gegenseitig fördern oder behindern:
 - Skalierbarkeit kann zu besserer Effizienz führen.
 - Portabilität kann die Interoperabilität einschränken.

Probleme bei der Erstellung von Software

**sind häufig eine Folge von
Konkurrenzsituationen zwischen**

- ❑ funktionaler Eignung der Software,
- ❑ guter Qualität der Software und
- ❑ kostengünstiger Erstellung von Software,
die führt zu
 - fehlender Definition von Anforderungen
 - fehlender Dokumentation von Anforderungen und Entscheidungen
 - fehlender Absicherung der Qualität der Umsetzung

**Die folgenden beiden Beispiele zeigen öffentlich gewordene Misserfolge
bei der Erstellung von Softwaresystemen.**

Probleme bei der Erstellung von Software – Beispiele

FBI Virtual Case File

- ❑ Ziel: einheitliche elektronische Speicherung aller Fallakten des FBI
- ❑ Laufzeit: 2000 – 2005, dann Projekt erfolglos abgebrochen
- ❑ Probleme:
 - ständige Änderungen der funktionalen (und qualitativen) Anforderungen
 - ständiger Personalwechsel im Projektmanagement
 - Softwareentwicklung durch Mitarbeiter ohne ausreichende technische Kompetenz
- ❑ Schaden: 170.000.000 US\$

fiscus-Software

- ❑ Ziel: bundeseinheitliche Software für Lohn-/Einkommenssteuererhebung
- ❑ Laufzeit: 1993 – 2005, dann Projekt erfolglos abgebrochen
- ❑ Probleme:
 - wechselnde politische Vorgaben
 - wechselnde technische Vorgaben
 - erzwungene Weiter- und Wiederverwendung vorhandener Software
 - mehrfacher Neustart des Projekts
- ❑ Schaden: 1.000.000.000 €

Aufbau der Vorlesung

Konzept zur Vermittlung der Inhalte in dieser Vorlesung:

- ❑ anknüpfen an die bereits bekannten Aspekte der Softwareentwicklung:
objektorientierte Programmierung mit Java
- ❑ ausgehend von der Programmierung
 - Präsentation von (allgemein nützlichen) Programmierkonzepten
 - Einführen von abstrakteren Beschreibungsformen
 - Nutzung einer umfangreichen Beispielsoftware zur Veranschaulichung

Vorteile dieses Konzepts:

- ❑ bessere Verständlichkeit der vorgestellten Konzepte während der Vermittlung
- ❑ bessere Möglichkeiten, um Beispiele zu formulieren

Konsequenz:

- ❑ Aufbau der Vorlesung folgt nicht dem Ablauf einer Softwareentwicklung

Inhalte in der Vorlesung

- ❑ Ideen der objektorientierten Softwaregestaltung
- ❑ Objektorientierte Konzepte in Java
- ❑ UML-Notation
- ❑ Technische Gestaltung von Software
- ❑ Testen von Software
- ❑ Vorgehensweisen zur Softwareentwicklung

Beispielsoftware *SWT-Starfighter*

- ❑ Arcade-Spiel
- ❑ Spieler (=Raumschiff) muss sich gegen Monster verteidigen
- ❑ zweidimensionale Grafik
- ❑ Steuerung über Tasten

- ❑ vollständig implementiert in Java
- ❑ vollständiger Programmtext ist verfügbar
- ❑ Entwickler ist SWT-Tutor: Dominic Starzinski

- ❑ Vorteile für die Veranstaltung
 - **ein** durchgängiges Beispiel
 - Änderungen im Programmtext können selbst ausprobiert werden
 - visuelle Ausrichtung des Spiels hilft, die Wirkung von Änderungen zu beobachten
 - die Projektaufgaben zum Erwerb der Studienleistung werden als Erweiterungen des Spiels definiert

kurze Präsentation

Beispielsoftware *SWT-Starfighter*

(Fortsetzung)

Anforderungen an zukünftige (Weiter-)Entwicklung:

- ☐ weitere Monster
- ☐ anderes Verhalten von Monstern
- ☐ weitere Anzeigen
- ☐ (etwas) anderer Spielablauf
- ☐ andere Grafiken

- ☐ ähnliche Spiele mit anderem Kontext

**=> Die notwendigen Änderungen müssen bereits bei der Entwicklung
vorbereitet werden:**

Wartbarkeit!

Beispielsoftware *SWT-Starfighter*

(Fortsetzung)

Vorbereitung auf Änderungen/Erweiterungen:

The diagram consists of two ovals. The left oval is black and contains the text 'allgemeines Framework für Arcade-Spiele'. The right oval is red and contains the text 'konkrete Ergänzungen für SWT-Starfighter'. There are no lines connecting the two ovals.

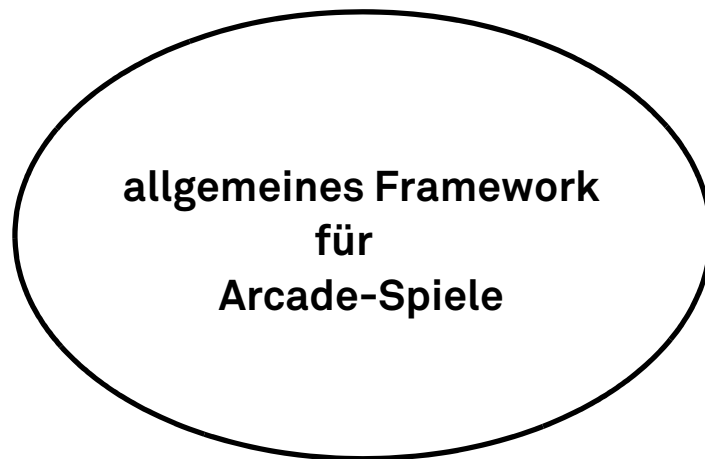
**allgemeines Framework
für
Arcade-Spiele**

**konkrete Ergänzungen
für
*SWT-Starfighter***

Beispielsoftware *SWT-Starfighter*

(Fortsetzung)

Vorbereitung auf Änderungen/Erweiterungen:



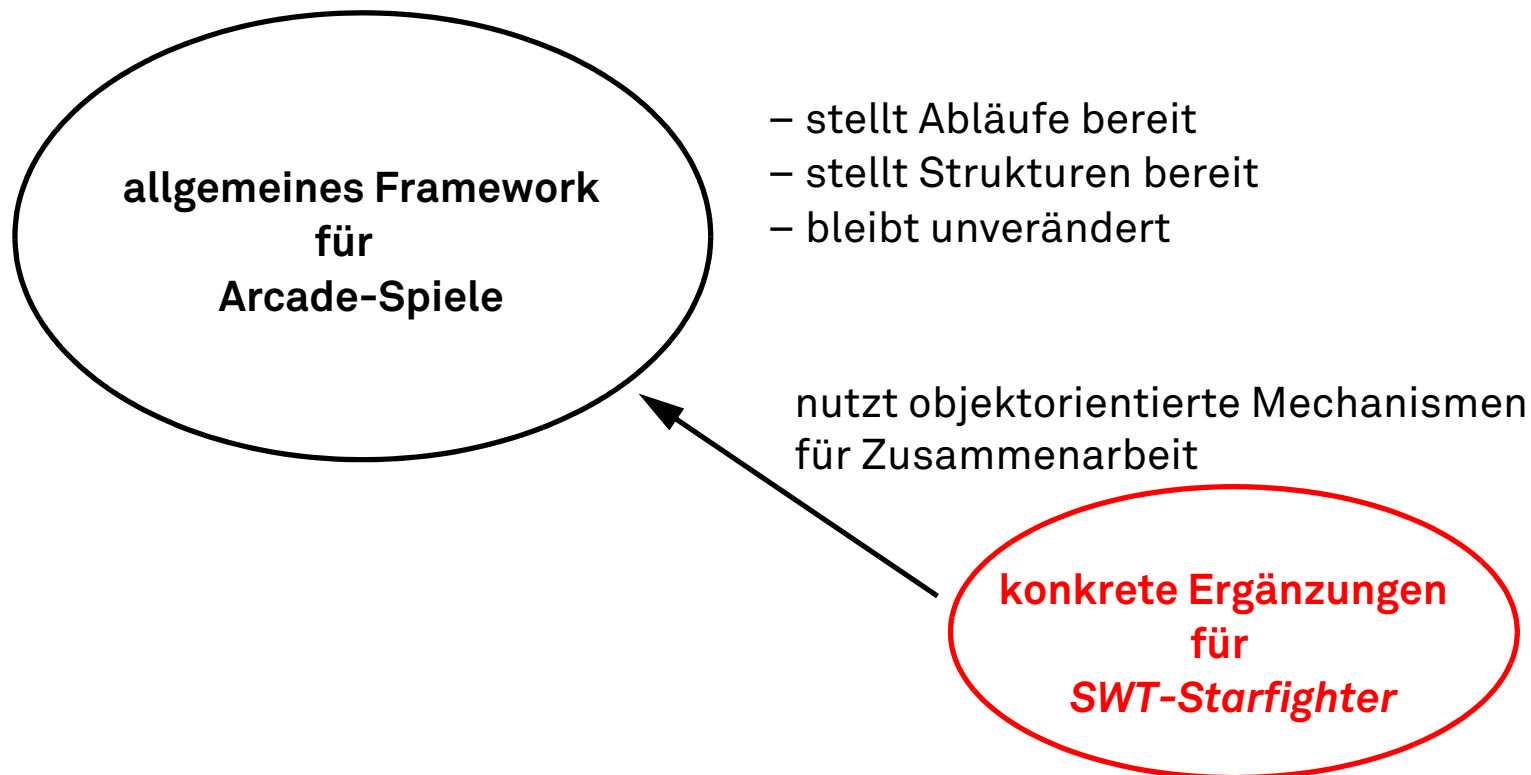
- stellt Abläufe bereit
- stellt Strukturen bereit
- bleibt unverändert



Beispielsoftware SWT-Starfighter

(Fortsetzung)

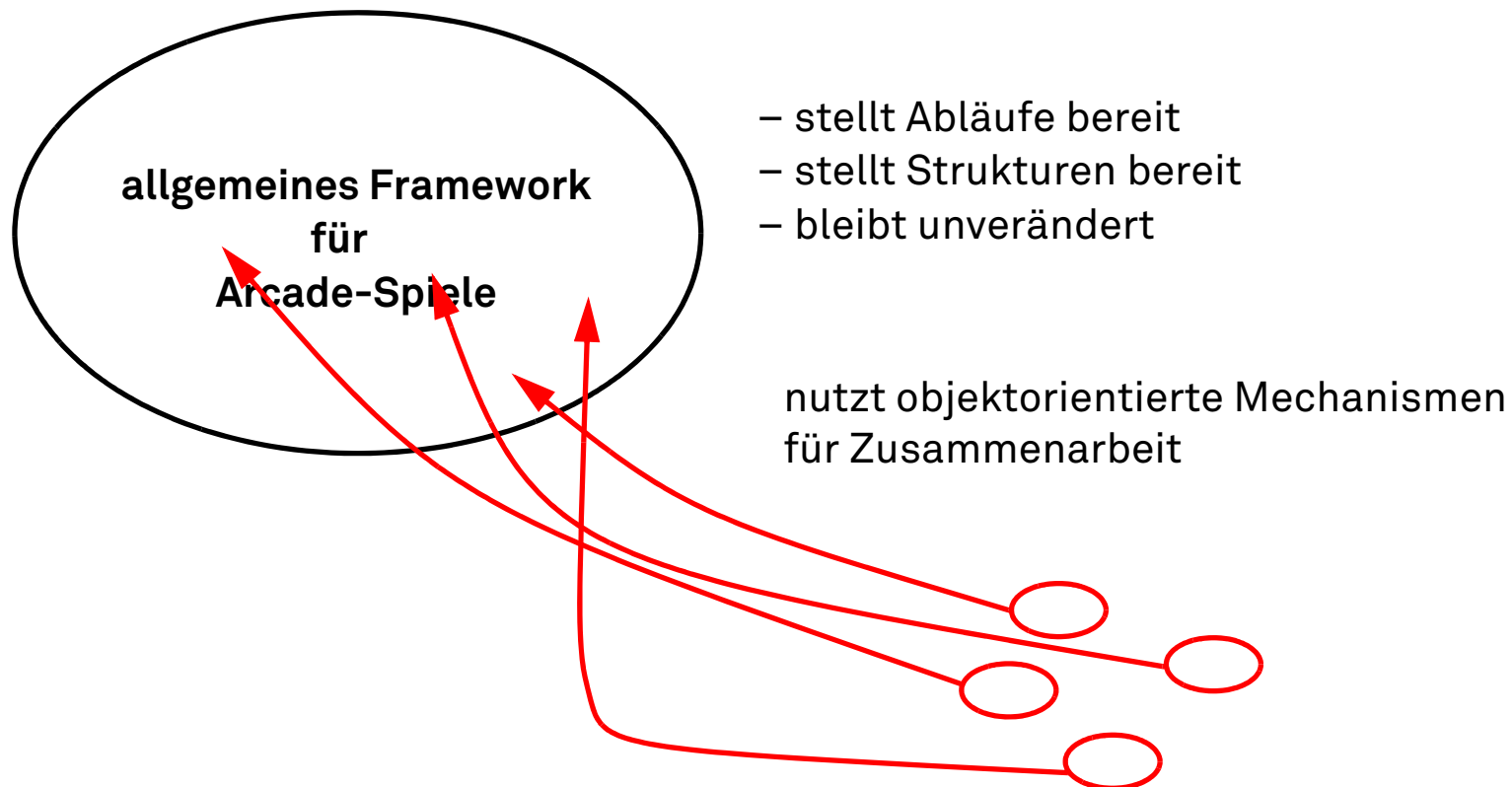
Vorbereitung auf Änderungen/Erweiterungen:



Beispielsoftware SWT-Starfighter

(Fortsetzung)

Vorbereitung auf Änderungen/Erweiterungen:



Beispielsoftware *SWT-Starfighter*

(Fortsetzung)

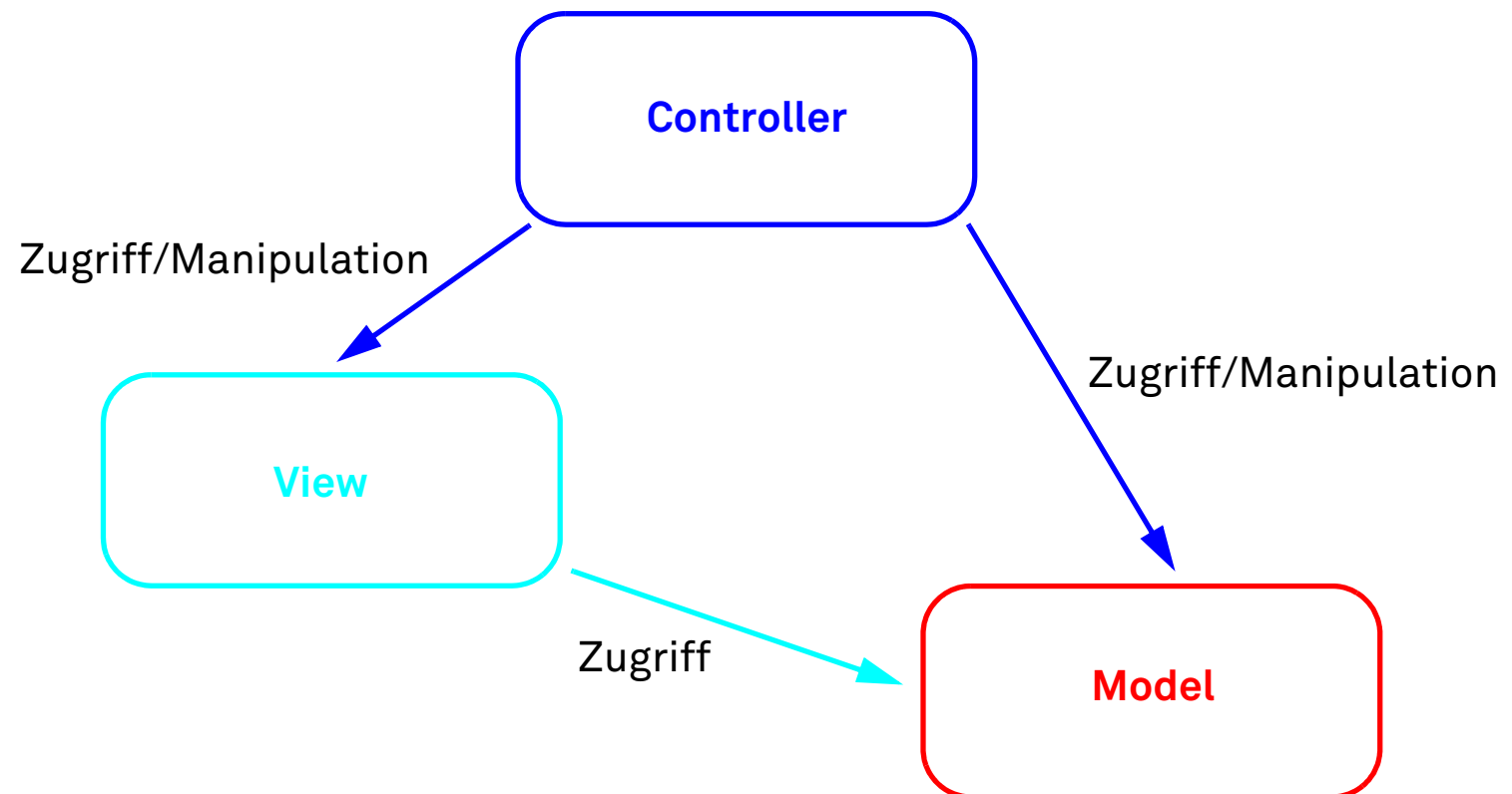
weitere Vorbereitung auf Änderungen/Erweiterung:

Architekturstil ***Model – View – Controller (MVC)***

- ❑ getrennte Behandlung der drei Aufgabenbereiche
 - Model (Datenmodell):
unabhängig von Präsentation und Abläufen
 - View (Präsentation/Benutzungsschnittstelle):
visualisiert Programmzustände, verarbeitet aber keine Daten,
reagiert auf die Änderung von Daten
 - Controller (Steuerung der Abläufe):
verbindet Präsentation und Datenmodell durch Ausführung von Aktionen
- ❑ Die technische Ausgestaltung von MVC kann sehr unterschiedlich erfolgen.

Architekturstil Model - View - Controller (MVC)

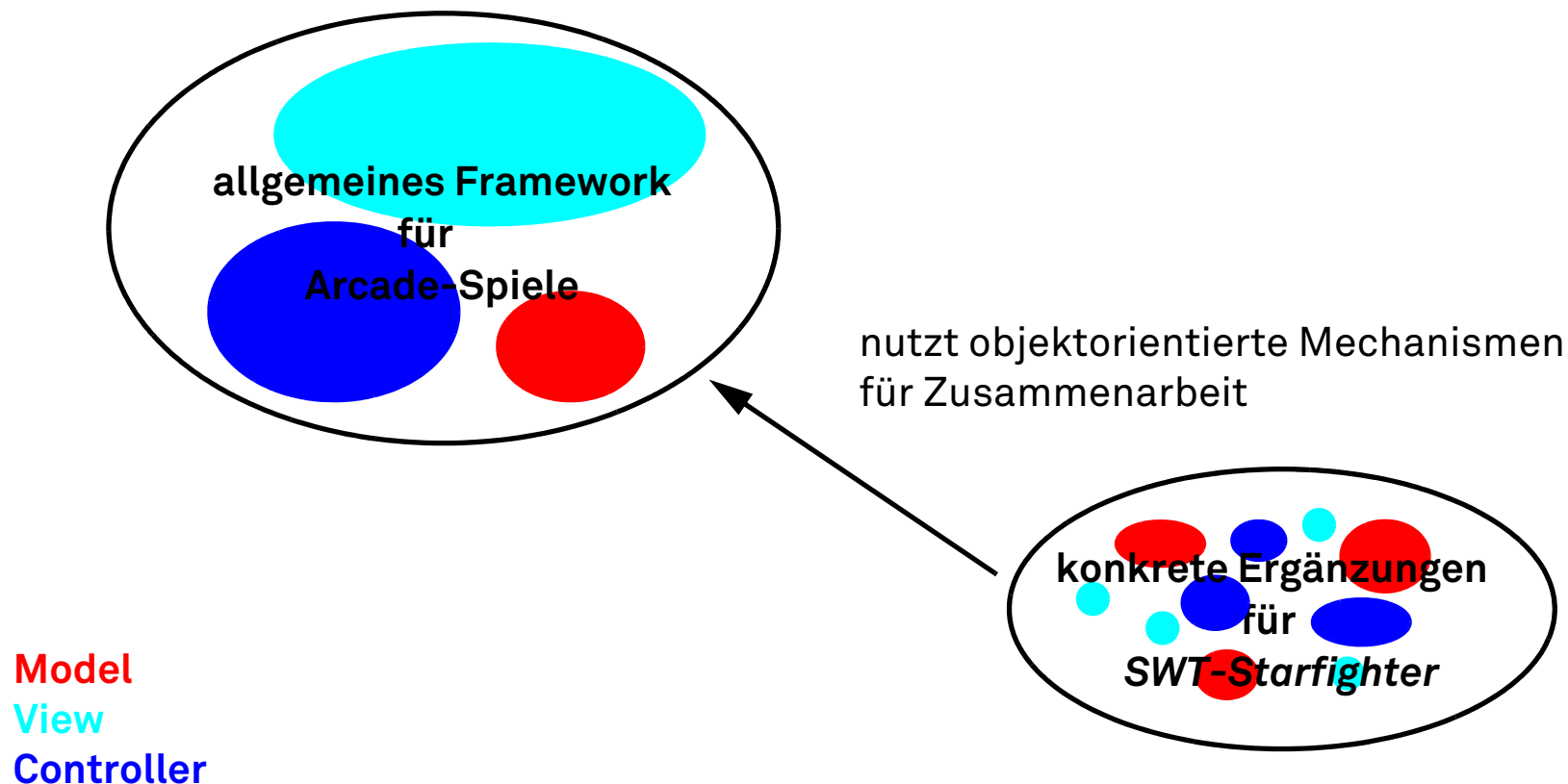
Grundlegende Idee der Zusammenarbeit



Beispielsoftware SWT-Starfighter

(Fortsetzung)

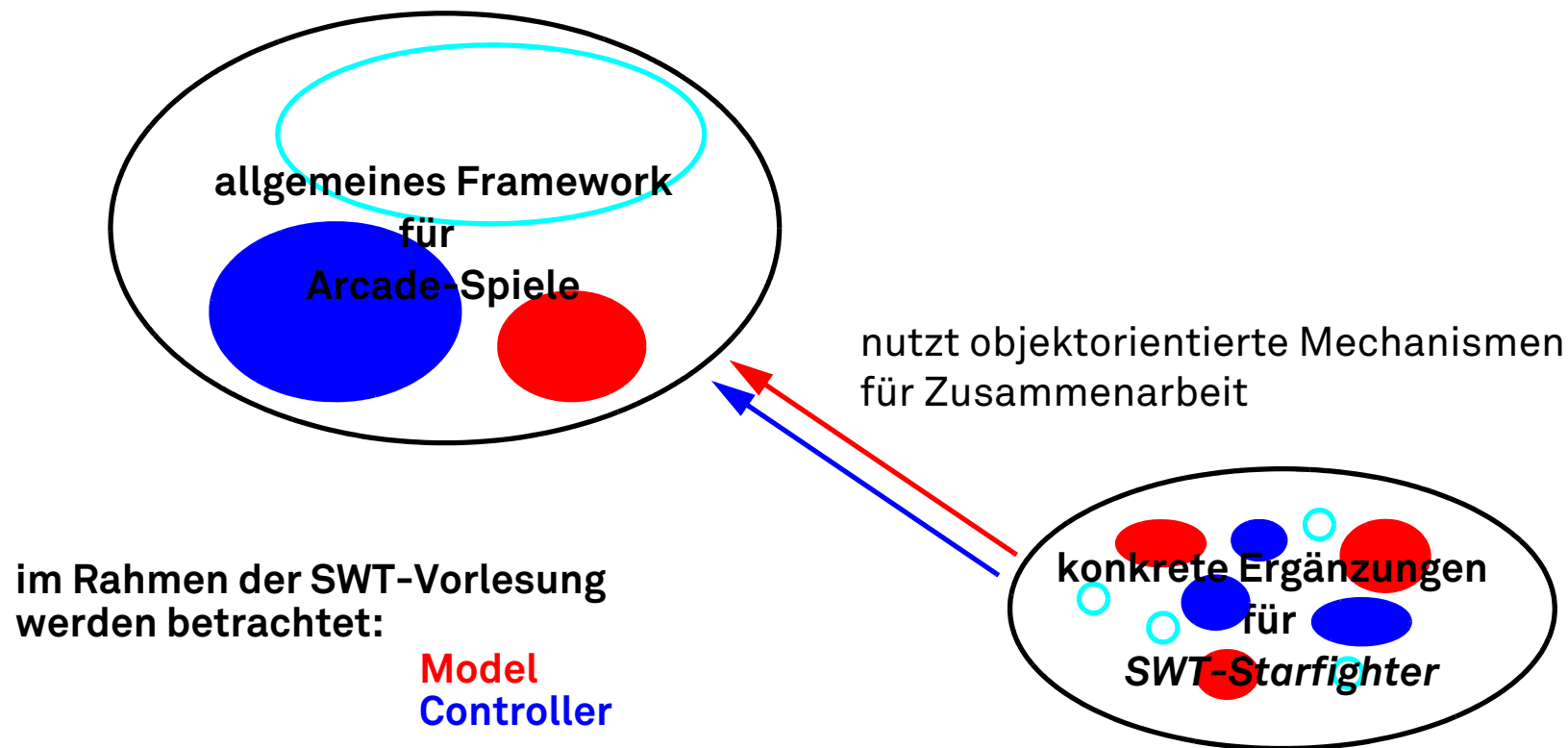
Framework und MVC



Beispielsoftware SWT-Starfighter

(Fortsetzung)

Framework und MVC



Folien zur Vorlesung **Softwaretechnik**

Abschnitt 1.2: Ideen der objektorientierten Softwaregestaltung

Objektorientierte Systeme

Die Arbeitsweise objektorientierter Systeme orientiert sich an der realen Welt:

Lösen einer Aufgabenstellung durch
Zusammenarbeit von handelnden Entitäten (Objekten),
basierend auf Kommunikation (über definierte Nachrichten) mit
Einschränkungen, die durch Regeln (objektorientierte Prinzipien) gegeben sind:

- ❑ **Kapselung**
Jedes Objekt grenzt sich ab und verletzt auch nicht die Grenzen anderer Objekte.
- ❑ **Lokalität**
Daten und zugehörige Handlungen sind in einem Objekt zusammengefasst.
- ❑ **Geheimhaltung**
Jedes Objekt stellt nach außen nur die für seine Aufgabebewältigung notwendigen Informationen bereit.
- ❑ **Autonomie**
Jedes Objekt entscheidet selbst über die von ihm ausgeführten Handlungen.

Objektorientierte Systeme

(Fortsetzung)

Konsequenzen aus der technischen Umsetzung objektorientierter Systeme:

- ❑ konzeptionelle Vollständigkeit
Jedes (korrekt implementierte) Objekt kann alle geforderten Aufgaben erfüllen.
- ❑ wohldefinierte Leistung
Für jedes Objekt ist sein Leistungsumfang bekannt.
- ❑ wohldefinierte Kommunikation
Für jedes Objekt ist bekannt, in welcher Weise seine Leistungen abgerufen werden können.
- ❑ Teamfähigkeit
Alle Objekte arbeiten immer kooperativ zusammen.

Objektorientierte Systeme

(Fortsetzung)

Anmerkungen:

- ❑ Objektorientierte Programmiersprachen stellen Konzepte bereit, um objektorientierte Systeme implementieren zu können.
- ❑ Objektorientierte Programmiersprachen stellen darüber hinaus weitere Konzepte bereit, um die Konstruktion objektorientierter Systeme zu vereinfachen.
- ❑ Art und Umfang dieser Konzepte unterscheiden sich bei den verschiedenen objektorientierten Programmiersprachen.

Im Rahmen dieser SWT-Vorlesung wird aufgrund der Vorkenntnisse der teilnehmenden Studierenden nur **Java** als Programmiersprache betrachtet.

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 1.3: Objektorientierte Konzepte in Java

Paket

- ❑ wesentliches Ziel:
Schaffen einer konzeptionellen Struktur innerhalb einer großen Anzahl von Klassen.
- ❑ weitere Ziele:
Schaffen eines Namensraums, so dass in einem Projekt mehrere Klassen mit gleichem Namen möglich sind.
- ❑ Syntax:
 - Zuordnung zu einem Paket: **package** edu.udo.cs.swtsf.core;
 - Einbeziehen eines Pakets: **import** edu.udo.cs.swtsf.core.player.Player;
 - individueller Zugriff in anderen Paketen: java.util.ArrayList
- ❑ Randbedingung:
Paketname muss der Verzeichnisstruktur entsprechen. (Programmierungsumgebung)
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.core;  
import java.util.ArrayList;  
...  
import edu.udo.cs.swtsf.core.player.Player;  
...  
public class Game { ... }
```

Klasse

- ❑ wesentliches Ziel:
Definition von Eigenschaften einer Gruppe von gleichförmigen Objekten.
- ❑ weitere Ziele:
Festlegen eines i.W. durch die Methoden der Klasse gegebenen Datentyps.
- ❑ Syntax:
 - Deklaration: **public class** HudElement { ... }
 - Benutzung: HudElement
- ❑ Randbedingung:
Klassenname muss dem Dateinamen entsprechen. (Programmierungsumgebung)
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.view;  
import edu.udo.cs.swtsf.core.Game;  
public class HudElement {  
    ...  
}
```

Klasse

(Fortsetzung)

erweitertes Beispiel:

```
package edu.udo.cs.swtsf.view;
import edu.udo.cs.swtsf.core.Game;
public class HudElement {
    private HudElementOrientation orientation = HudElementOrientation.TOP;
    private String text = "";
    private String imagePath;
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;
    public HudElement(HudElementOrientation orientation, String imagePath,
                    int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)
        { ... }
    public void setText(String value) { ... }
    public String getText() { ... }
    public void setOrientation(HudElementOrientation value) { ... }
    public HudElementOrientation getOrientation() { ... }
    ...
}
```

Klasse

(Fortsetzung)

erweitertes Beispiel:

```
package edu.udo.cs.swtsf.view;
import edu.udo.cs.swtsf.core.Game;
public class HudElement {                                     Attribute
    private HudElementOrientation orientation = HudElementOrientation.TOP;
    private String text = "";
    private String imagePath;
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;
    public HudElement(HudElementOrientation orientation, String imagePath,
        int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)
        { ... }
    public void setText(String value) { ... }
    public String getText() { ... }
    public void setOrientation(HudElementOrientation value) { ... }
    public HudElementOrientation getOrientation() { ... }
    ...
}
```


Klasse

(Fortsetzung)

erweitertes Beispiel:

```
package edu.udo.cs.swtsf.view;
import edu.udo.cs.swtsf.core.Game;
public class HudElement {
    private HudElementOrientation orientation = HudElementOrientation.TOP;
    private String text = "";
    private String imagePath;
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;
    public HudElement(HudElementOrientation orientation, String imagePath,
                     int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)
    { ... }
    public void setText(String value) { ... }
    public String getText() { ... }
    public void setOrientation(HudElementOrientation value) { ... }
    public HudElementOrientation getOrientation() { ... }
    ...
}
```

Konstruktor

Klasse

(Fortsetzung)

erweitertes Beispiel:

```
package edu.udo.cs.swtsf.view;
import edu.udo.cs.swtsf.core.Game;
public class HudElement {
    private HudElementOrientation orientation = HudElementOrientation.TOP;
    private String text = "";
    private String imagePath;
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;
    public HudElement(HudElementOrientation orientation, String imagePath,
        int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)
        { ... }
    public void setText(String value) { ... }
    public String getText() { ... }
    public void setOrientation(HudElementOrientation value) { ... }
    public HudElementOrientation getOrientation() { ... }
    ...
}
```

Methoden

Innere Klasse

- ❑ wesentliches Ziel:
Gemeinsames Verwalten konzeptionell zusammengehörender Klassen.
- ❑ weitere Ziele:
Zugriff auf geschützte Eigenschaften der umgebenden Klasse.

- ❑ Anmerkungen:
 - Innere Klassen führen **nicht** zu "geschachtelten" Objekten.
 - Auch innere Klassen dienen nur zur Definition von Eigenschaften einer Gruppe von gleichförmigen Objekten.

Instanzeigenschaften (Instanzattribute/Instanzmethoden)

- ❑ wesentliche Ziel:
Individualisierung von Objekten.
- ❑ weitere Ziele:
 - Jedes durch einen Konstruktor der Klasse erzeugte Objekt (=Instanz) besitzt eigene Speicherbereich für seine Attribute.
 - Aufrufe von Methoden für ein Objekt beziehen sich auf die für dieses Objekt gespeicherten Attributwerte.
- ❑ Beispiel:

```
public class HudElement {  
    private String text = "";  
    public void setText(String value) {  
        if (value == null) { throw new IllegalArgumentException(); }  
        text = value;  
    }  
    public String getText() {  
        return text;  
    }  
    ...  
}
```

Instanzattribut

statische Eigenschaften (statische Attribute/statische Methoden)

- ❑ wesentliches Ziel:
(Globale) Eigenschaften ohne explizites Erzeugen eines Objekts verfügbar machen.
- ❑ weitere Ziele:
Gemeinsame Attribute für alle Objekte einer Klasse schaffen.
- ❑ Syntax:
Deklaration mit dem Schlüsselwort **static**
- ❑ Randbedingung:
Statische Methoden dürfen nicht auf Instanzattribute und Instanzmethoden zurückgreifen, da kein zugehöriges Objekt existiert.
- ❑ Beispiele:

```
static final String TITLE_TEXT = "SWT - Starfighter";
```

```
public static void main(String[] args){ ... }
```

Vererbung

- ❑ wesentliches Ziel:
Schaffen eines gemeinsamen Typs, der eigene Objekte vorgibt und der Objekte von verschiedenen (Unter-)Klassen zusammenfasst.
- ❑ weitere Ziele:
Weitergeben von Eigenschaften an die Deklarationen der Unterklassen
- ❑ Syntax:
 - in der Deklaration der Unterklasse: **extends**
 - eine Klasse kann sich gegen das "Geerbt-Werden" wehren: **final class**
- ❑ Randbedingung:
Jede Klasse erbt von genau einer Oberklasse.
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.example;  
import edu.udo.cs.swtsf.core.Target;  
public class MonsterEasy extends Target {  
    public MonsterEasy() {  
        setMaxHitpoints(2);  
        setSize(32);  
        ...  
    }  
}
```

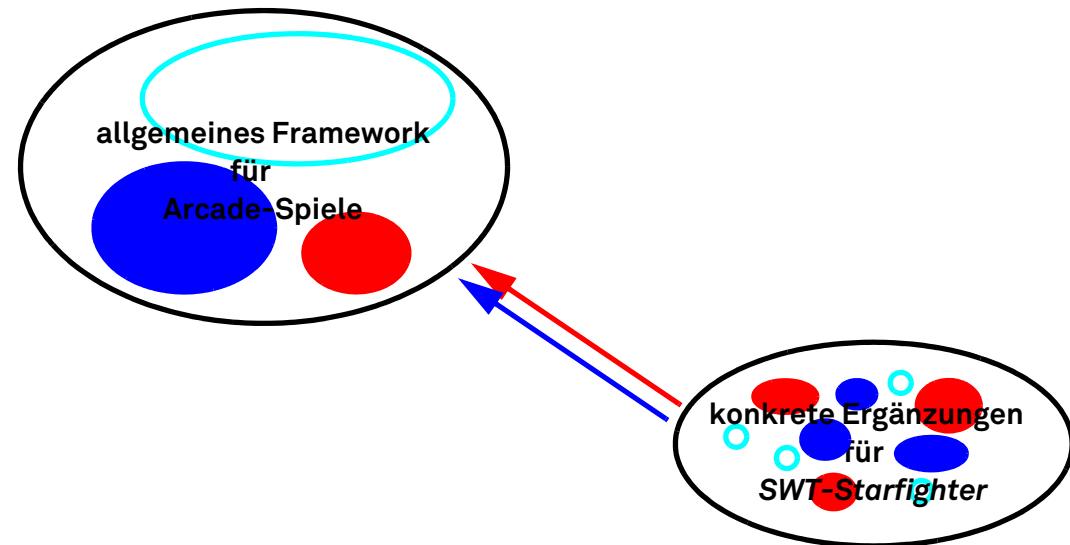
geerbte Methoden



Zugriffsrecht

- ❑ wesentliches Ziel:
Sichtbarkeit von Eigenschaften einschränken, um fehlerhafte Nutzung zu verhindern.
- ❑ Syntax:
 - allgemein sichtbar: **public**
 - im Paket und allen Unterklassen sichtbar: **protected**
 - im Paket sichtbar: *(keine Angabe = package)*
 - nur innerhalb der Klasse sichtbar: **private**
- ❑ Beispiel:

Nutzung von **protected**
im Framework,
um Ergänzungen zu
ermöglichen.



Abstrakte Klasse

- ❑ wesentliches Ziel:
Schaffen eines gemeinsamen Typs, der **keine** eigenen Objekte ermöglicht, der aber Objekte von verschiedenen (Unter-)Klassen zusammenfasst.
- ❑ weitere Ziele:
Weitergeben von Eigenschaften an die Deklarationen der Unterklassen.
- ❑ Syntax:
 - Deklaration der abstrakten Klasse: **abstract**
 - Deklaration abstrakter Methoden: **abstract**
- ❑ Randbedingungen:
 - Eine abstrakte Methode führt zwangsläufig zu einer abstrakten Klasse.
 - Von einer abstrakten Klasse können keine Objekte erzeugt werden.
- ❑ Beispiel:
public abstract class Target extends Entity { ... }

Target soll immer nur als Oberklasse dienen.
Target enthält keine abstrakten Methoden.

Interface

- ❑ wesentliches Ziel:
Schaffen eines gemeinsamen Typs, der **keine** eigenen Objekte ermöglicht,
der aber Objekte von verschiedenen (Unter-)Klassen oder Interfaces zusammenfasst.
- ❑ weitere Ziele:
(Weitergeben von Methoden; keine Deklaration von Attributen)
- ❑ Syntax:
 - Deklaration: **public interface** Group<E> { ... }
 - Nutzung: **public class** BufferedGroup<E> **implements** Group<E> { ... }
- ❑ Randbedingungen:
 - Eine Klasse kann viele Interfaces implementieren.
 - Ein Interface kann von vielen Interfaces erben.
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.util;  
public interface Group<E> {  
    public void add(E element);  
    public void remove(E element);  
    public default int count(E element) { ... }  
    ...  
}
```

Polymorphie

- ❑ wesentliches Ziel:
Ausführen der Implementierung einer Methode, die "nah" an der Deklaration der Klasse liegt – unabhängig vom Typ der auf das Objekt verweisenden Referenz.
- ❑ Syntax:
Deklarieren der gleichen Methode in verschiedenen Klassen einer Vererbungshierarchie durch **Überschreiben** der Methode in einer Unterklasse.
- ❑ Randbedingungen:
 - Signaturen (=Name+Parameterliste) der Methoden müssen exakt übereinstimmen.
 - Eine Methode kann sich gegen Überschreiben wehren: **final**
 - Die Auswahl der Signatur erfolgt durch den Compiler.
 - Die Auswahl der ausgeführten Implementierung erfolgt durch das Laufzeitsystem.
- ❑ Anmerkung:
Die geeignete Nutzung von Vererbung, abstrakten Klassen und Interfaces beruht wesentlich auf dem Konzept der Polymorphie.

Generische Klasse/generisches Interface

- ❑ wesentliches Ziel:
Deklaration eines Datentyps, der mit Objekten anderer Klassen typsicher umgehen kann.
- ❑ Syntax:
Deklaration von Typparametern: < >
- ❑ Anmerkungen:
 - Die erlaubten Typargumente können durch die Angabe von beschränkenden Regeln (**super**, **extends**) präzisiert werden.
 - Die technische Umsetzung in Java ist etwas problematisch, um Kompatibilität mit alten Java-Versionen zu erhalten.
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.util;
public interface Group<E> {
    public void add(E element);
    public void remove(E element);
    public default int count(E element) { ... }
    ...
}
```

Deklaration des Typparameters

Nutzung des Typparameters

anonyme Klasse

- ❑ wesentliche Ziele:
 - Anlegen einer Klasse und Erzeugen von deren einzigem Objekt in einem Schritt innerhalb einer anderen Klasse.
 - Vermeiden der expliziten Deklaration einer nur genau einmal genutzten Klasse.
- ❑ weitere Ziele:
Zugriff auf finale Eigenschaften der umgebenden Klasse.
- ❑ Syntax:
Deklaration und Erzeugung gemeinsam: **new** Target { ... }
- ❑ Randbedingung:
Die anonyme Klasse muss eine deklarierte Klasse erweitern oder ein deklariertes Interface implementieren.

Lamda-Ausdruck

- ❑ wesentliche Ziele:
 - Anlegen einer Klasse und Erzeugen von deren einzigem Objekt in einem Schritt innerhalb einer anderen Klasse.
 - Vermeiden der expliziten Deklaration einer nur genau einmal genutzten Klasse.
 - Verbessern der Lesbarkeit/Reduktion des Schreibaufwands.
- ❑ weitere Ziele:
Zugriff auf finale Eigenschaften der umgebenden Klasse.
- ❑ Syntax:
Parameterliste -> Methodenrumpf
- ❑ Randbedingungen:
Compiler erwartet ein Objekt eines Interfaces mit genau einer abstrakten Methode.
- ❑ Vorgegebene Deklaration:

```
public interface BulletHitStrategy {  
    public void onHit(Bullet host, Target target);  
}
```

- ❑ Beispiel:

```
public static final BulletHitStrategy BULLET_DAMAGE_ON_HIT =  
(bullet, target) -> { target.addHitpoints(-bullet.getDamage());};
```

Enumeration

- ❑ wesentliches Ziel:
Anlegen eines Typs mit endlich vielen vorgegebenen Werten.
- ❑ Syntax:
Deklaration mit: **enum**
- ❑ Beispiel:

```
package edu.udo.cs.swtsf.view;  
public enum HudElementOrientation {  
    TOP, BOTTOM;  
}
```

Zusammenarbeit von Objekten

- ❑ wesentliches Ziel:
Bereitsstellen der Leistung des Softwarersystems.
- ❑ Syntax:
Aufruf der Methoden von erreichbaren Objekten mit der Angabe von Argumenten für die Parameter: . - Notation (Dereferenzierung)
- ❑ Beispiel:

```
target.addHitpoints(-bullet.getDamage());
```



Referenz auf Objekt

Reflection/Introspection

- ❑ wesentliches Ziel:
 - Untersuchung eines Programms durch sich selbst während der Ausführung.
 - Eventuell auch Manipulation der festgestellten Eigenschaften.
- ❑ Syntax:
In Java möglich durch Nutzung der Klasse **Class**.

```
public class Class<T> {  
    public String getName() { ... }  
    public Class<? super T> getSuperclass() { ... }  
    public Class<?>[] getInterfaces() { ... }  
    public Method[] getMethods() { ... }  
    public Method[] getDeclaredMethods() { ... }  
    public Field[] getFields() { ... }  
    public Field[] getDeclaredFields() { ... }  
    public boolean isAnonymousClass() { ... }  
    ...  
}
```


Folien zur Vorlesung **Softwaretechnik**

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

Unified Modeling Language (UML)

ist eine Sammlung von mehreren grafischen Notationen.

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

Anmerkungen:

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

- ❑ Die Spezifikationen der UML kann heruntergeladen werden:
<http://www.omg.org/spec/UML/2.5/>

Unified Modeling Language (UML)

(Fortsetzung)

UML umfasst 14 grafische Notationen, die führen zu

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

Unified Modeling Language (UML)

(Fortsetzung)

Zunächst erfolgt eine Einführung in

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

zur Vorbereitung der nachfolgenden Präsentation von Entwurfsmustern.

Später folgt eine Einführung in

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

Folien zur Vorlesung **Softwaretechnik**

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

Paketdiagramm

Ein Paketdiagramm zeigt

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

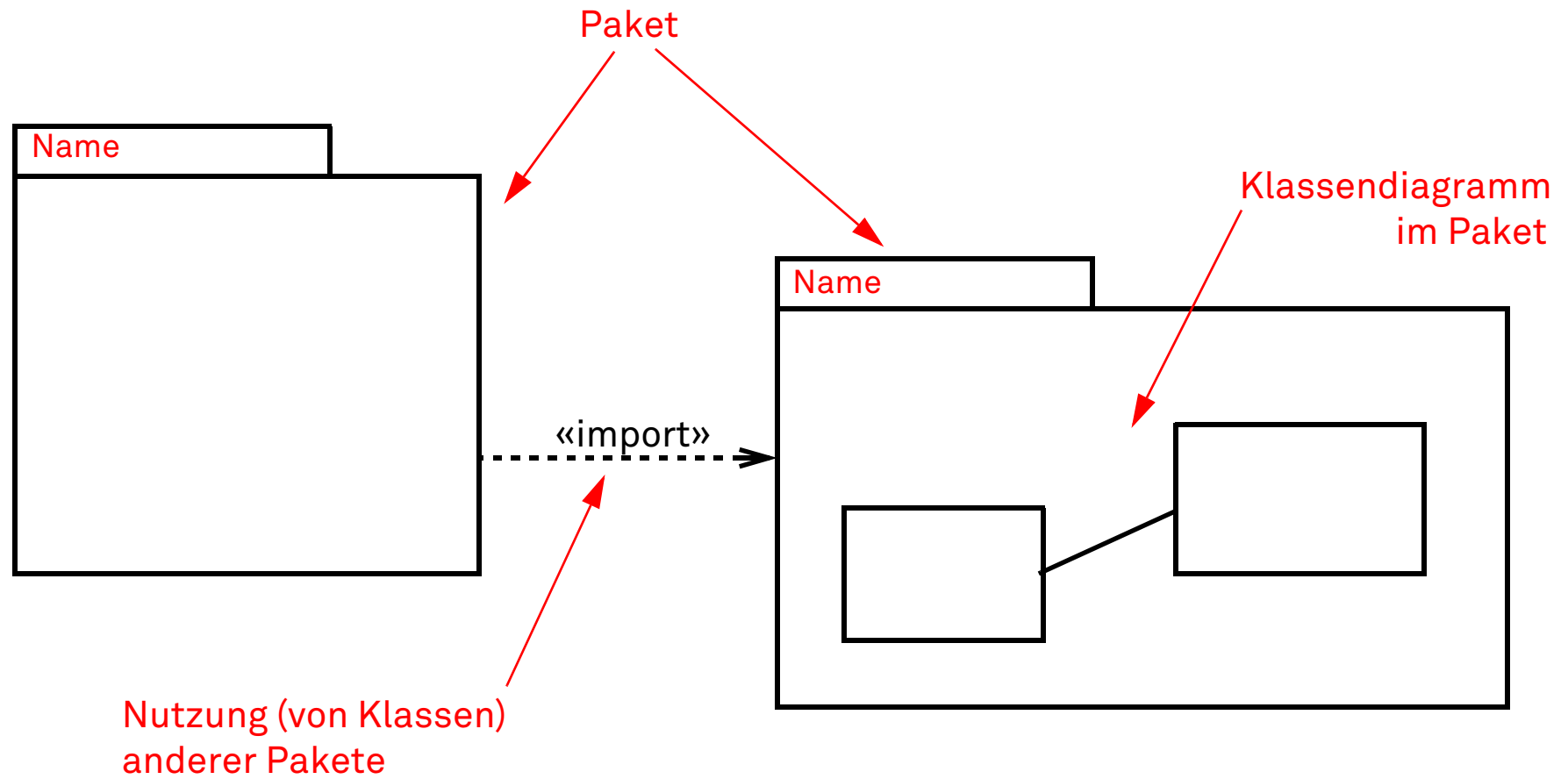
Ein Paketdiagramm schafft eine Darstellung auf einer Abstraktionsebene, die es in Java nicht gibt:

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

Paketdiagramm

(Fortsetzung)

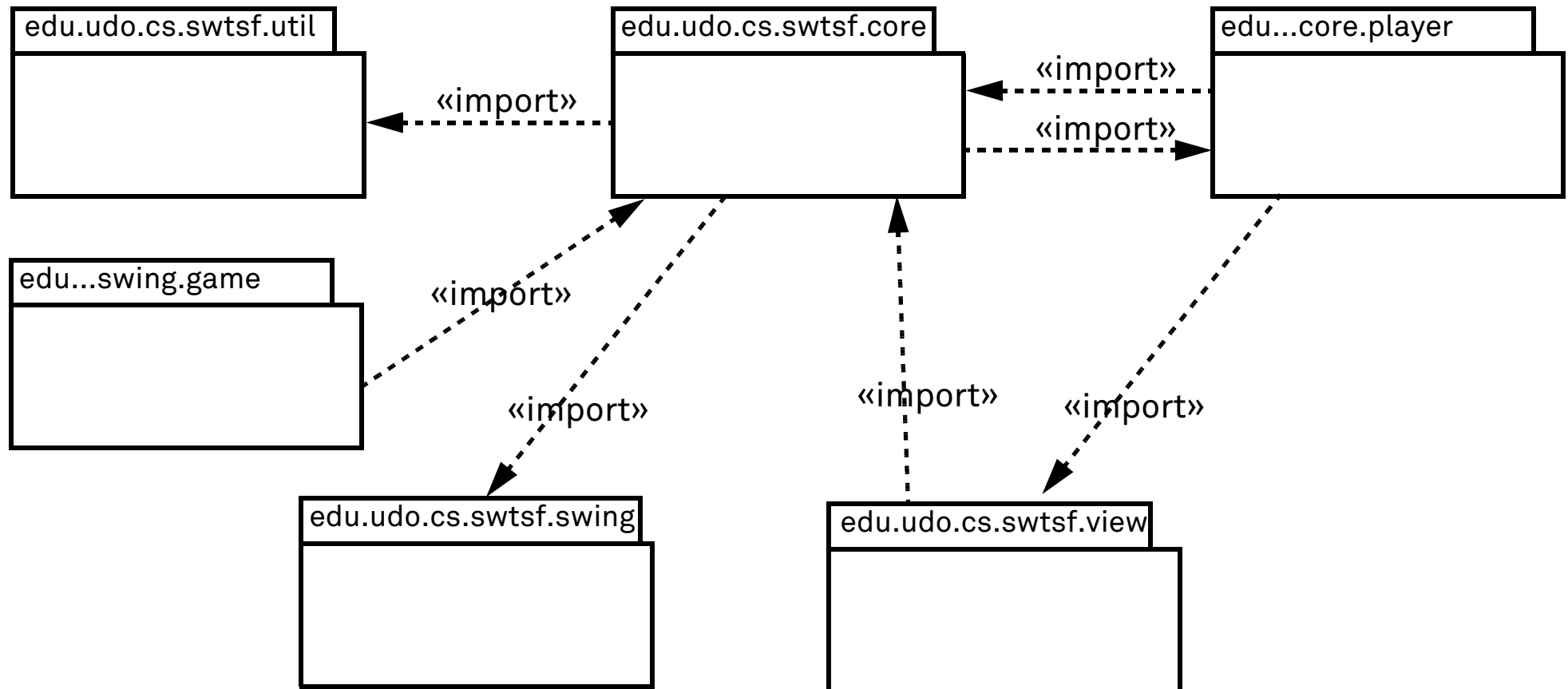
Syntax:



Paketdiagramm

(Fortsetzung)

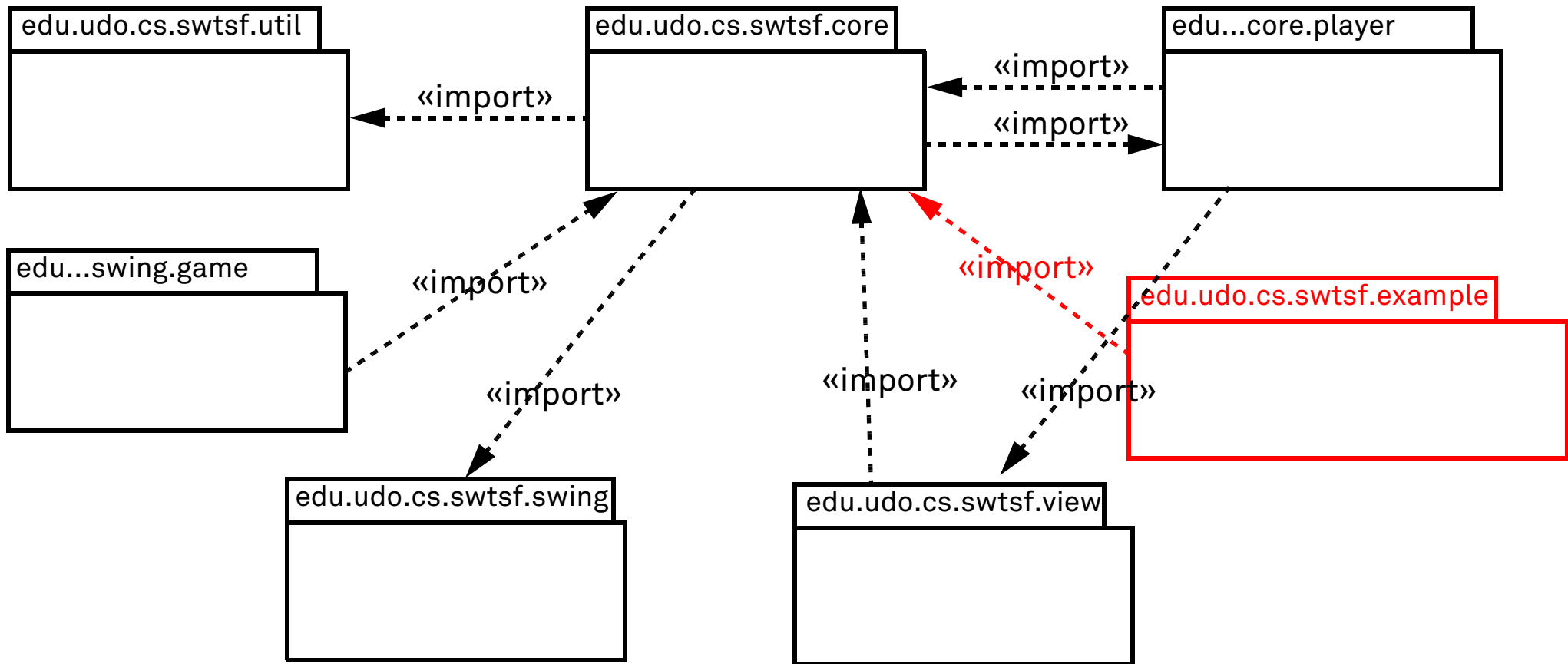
Beispiel *SWT-Starfighter – Framework*:



Paketdiagramm

(Fortsetzung)

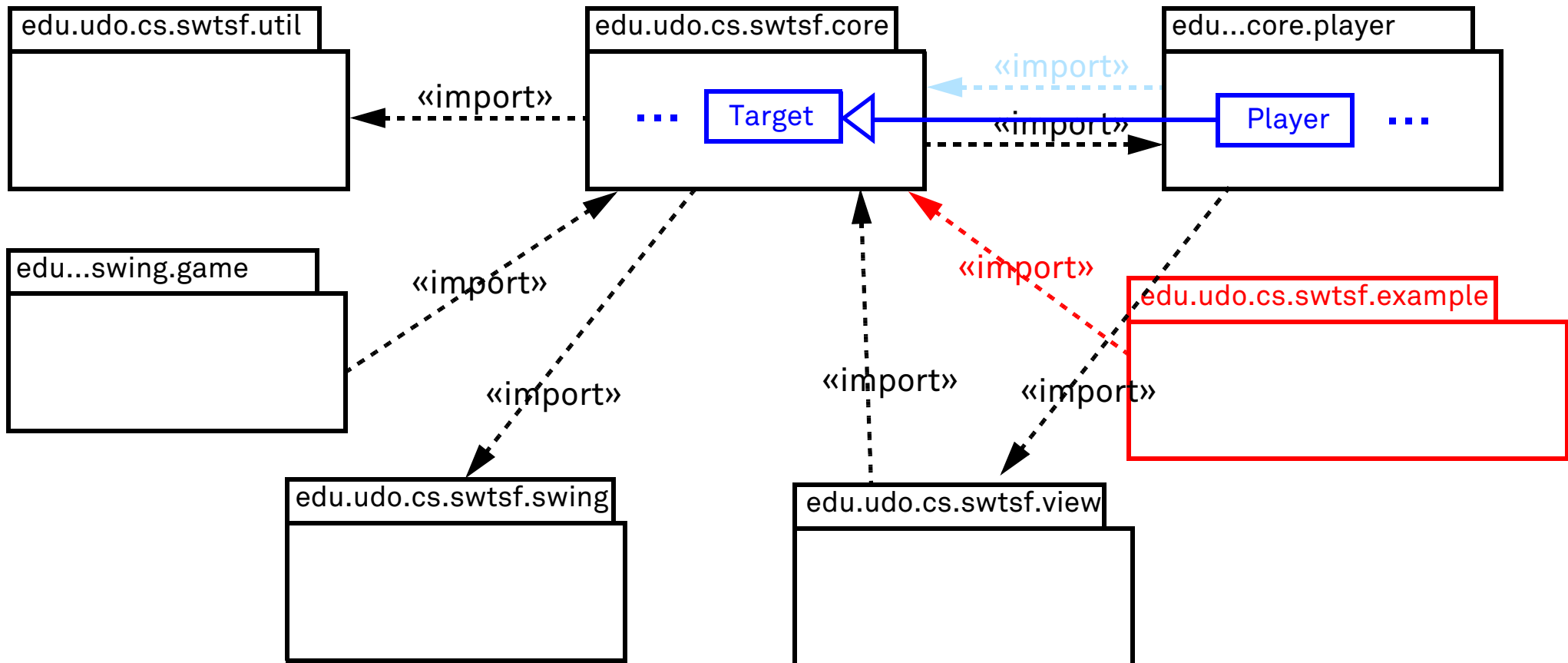
Beispiel *SWT-Starfighter* – Spiel:



Paketdiagramm

(Fortsetzung)

Beispiel *SWT-Starfighter* – Spiel:



Klassendiagramm

Ein Klassendiagramm zeigt

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

Klassendiagramme können genutzt werden, um

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

Klasse

Beispiel einer Java-Implementierung einer Klasse:

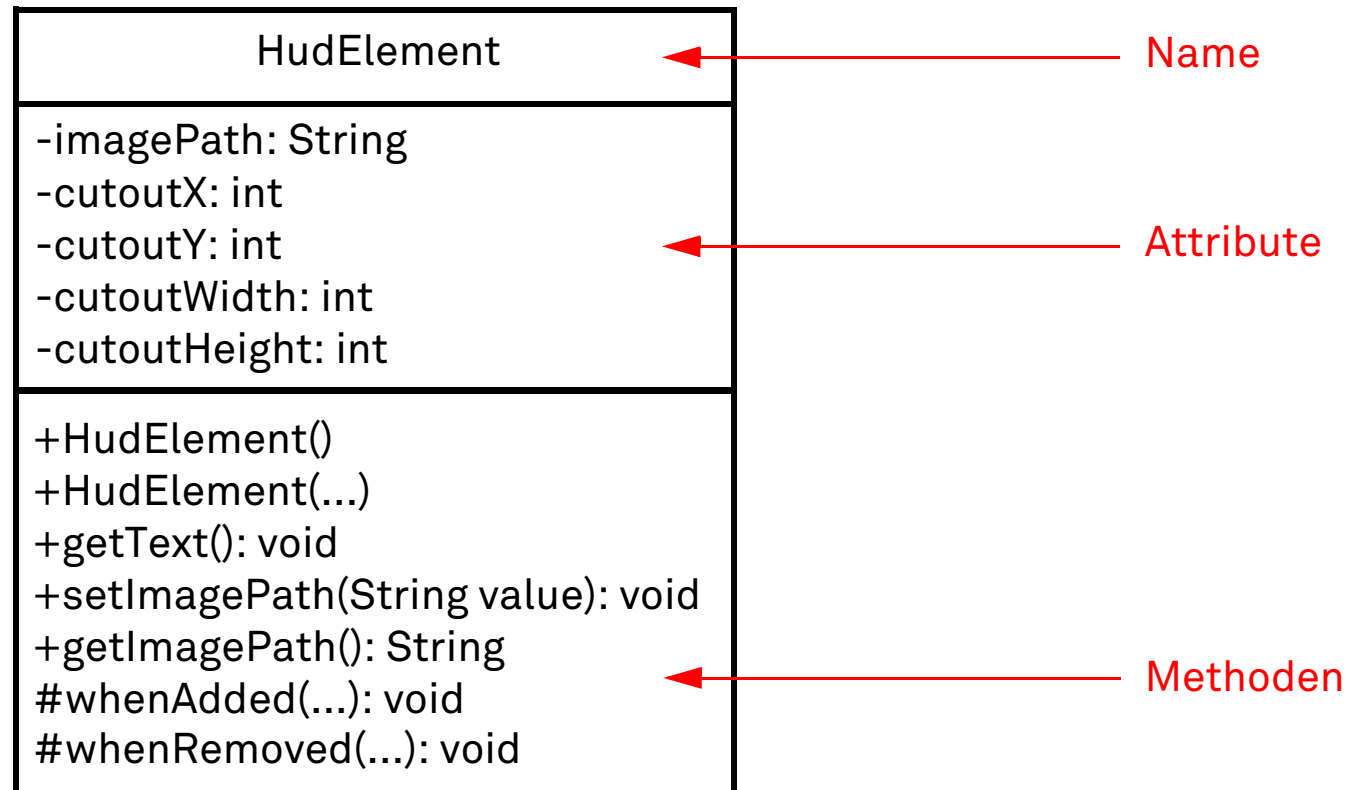
```
public class HudElement {  
    private String imagePath;  
    private int cutoutX, cutoutY, cutoutWidth, cutoutHeight;  
    public HudElement() {}  
    public HudElement(HudElementOrientation orientation, String imagePath,  
        int cutoutX, int cutoutY, int cutoutWidth, int cutoutHeight)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

Attribute

Konstruktoren

Methoden

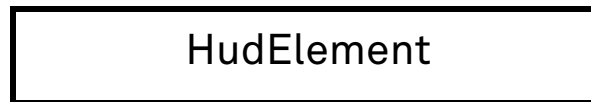
Klassendiagramm – Visualisierung einer Klasse



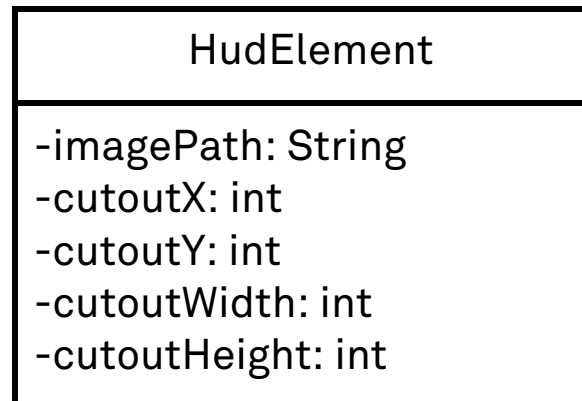
Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

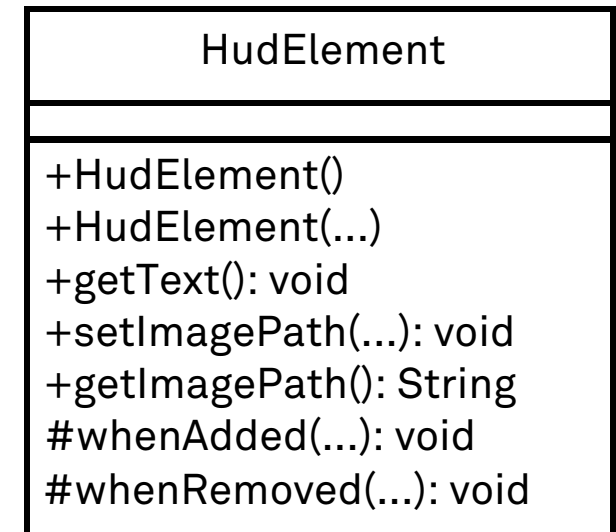
alternative Darstellungen:



nur Klassenname
ohne Details:
um Zusammenhänge
zwischen Klassen zu
visualisieren



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

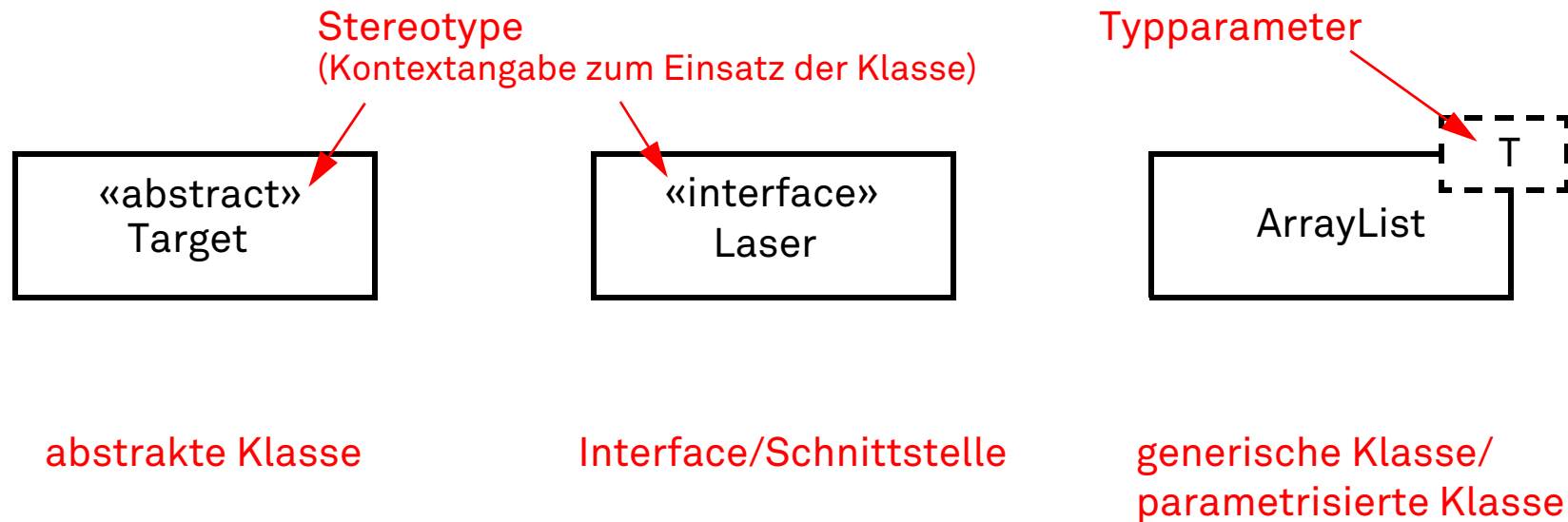


nur mit Methoden:
um die von einem Objekt
angebotenen Operationen
zu visualisieren

Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

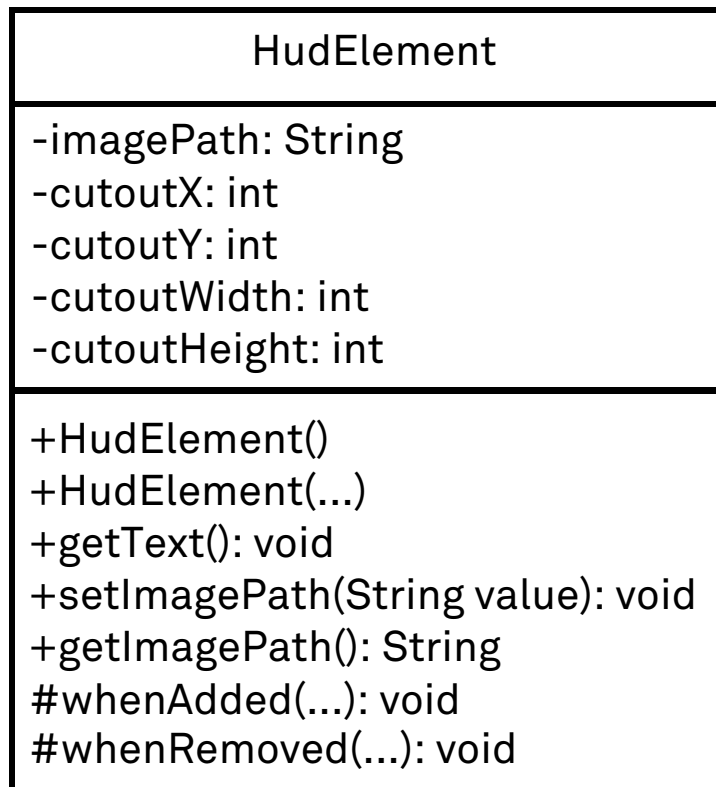
abstrakte Klasse/Interface/generische Klasse



Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

Deklaration von Eigenschaften: Zugriffsrechte

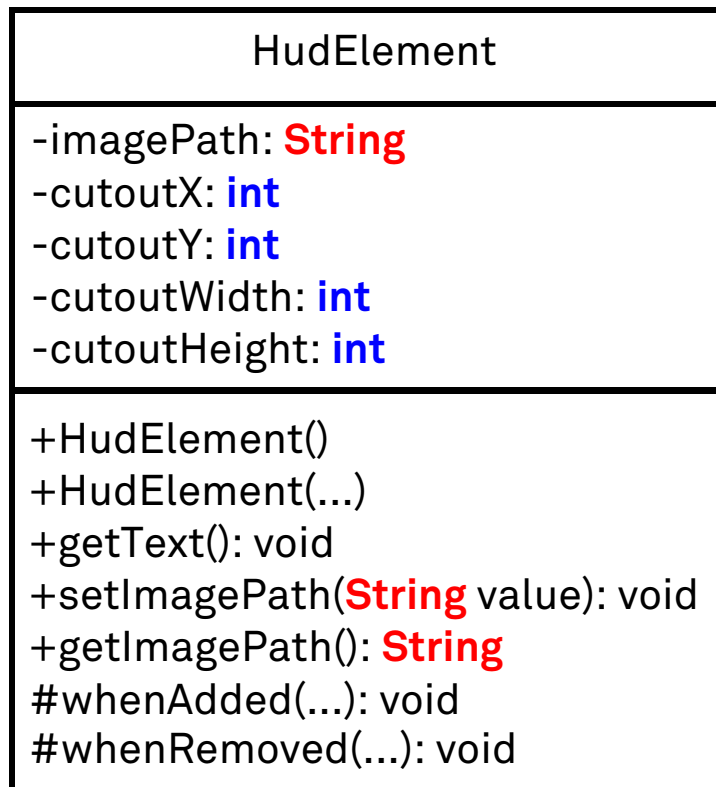


public: +
private: -
protected: #
package: ~

Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

Deklaration von Attributen: Typangaben

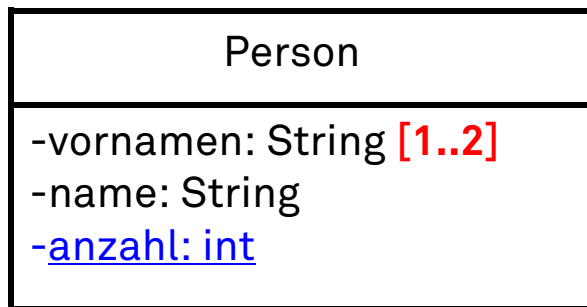


primitive Typen
oder **Klassen**

Klassendiagramm – Visualisierung einer Klasse

(Fortsetzung)

Ergänzende Angaben zu Eigenschaften:



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

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

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

statische Eigenschaft: ___

Klassendiagramm – Generalisierung/Spezialisierung/Vererbung

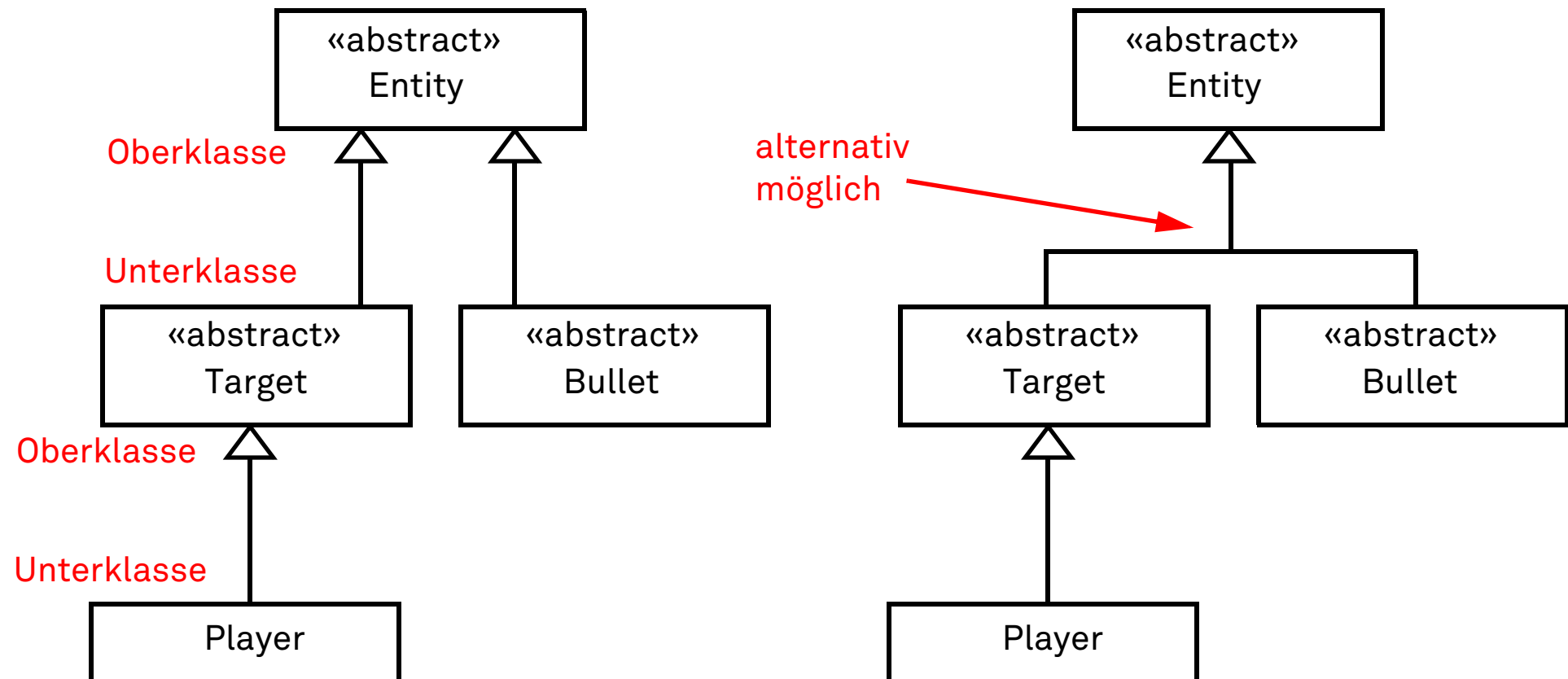
- ❑ Generalisierung/Spezialisierung beschreibt eine Beziehung zwischen Klassen.
- ❑ Eine Klasse kann (mehrere) spezialisierende Klassen besitzen.
- ❑ Jede Instanz einer spezialisierende Klasse wird auch als Instanz der allgemeineren Klasse aufgefasst.
- ❑ Generalisierung/Spezialisierung beschreibt eine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

Anmerkung:

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

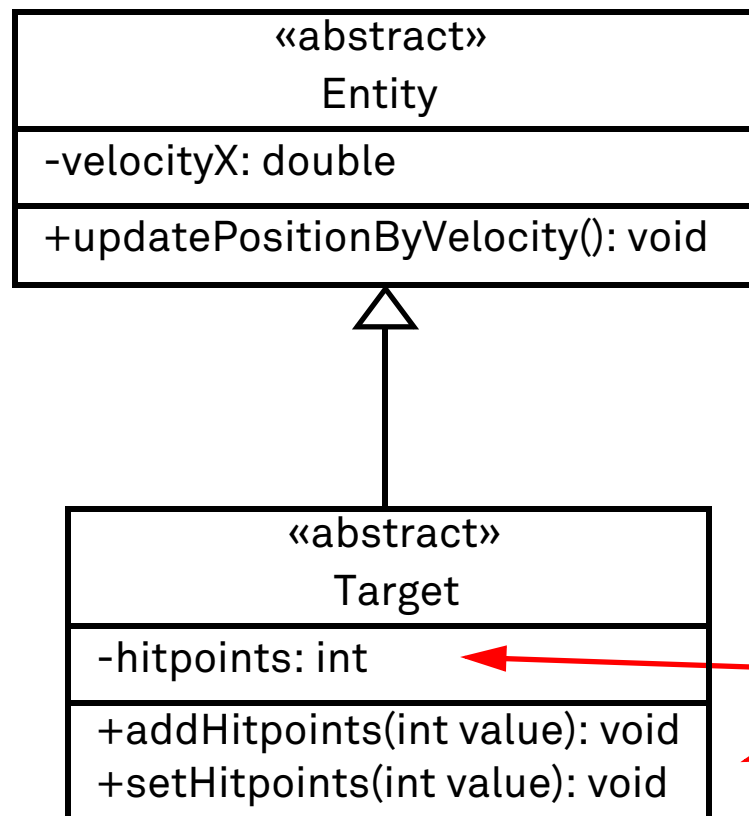
Klassendiagramm – Generalisierung/Spezialisierung

(Fortsetzung)



Klassendiagramm – Generalisierung/Spezialisierung

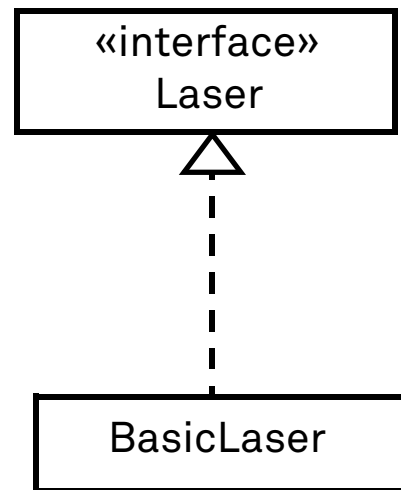
(Fortsetzung)



In Unterklassen werden nur
zusätzliche Attribute und Methoden aufgeführt.

Klassendiagramm – Realisierung/implementierung

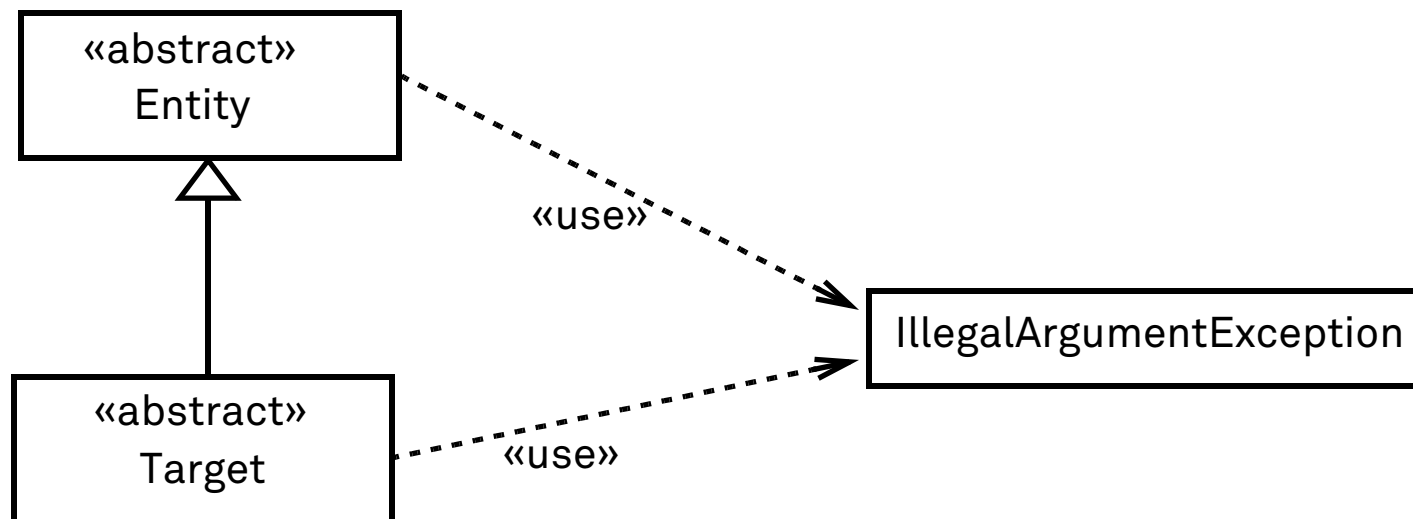
- ❑ Eine Realisierung beschreibt eine Beziehung zwischen einem Interface und einer Klasse.
- ❑ Ein Interface kann durch (mehrere) Klassen realisiert werden.
- ❑ Eine Klasse kann mehrere Schnittstellen realisieren.
- ❑ Realisierung beschreibt eine **Typbeziehung** zwischen Klassen.



Klassendiagramm – Abhängigkeit

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

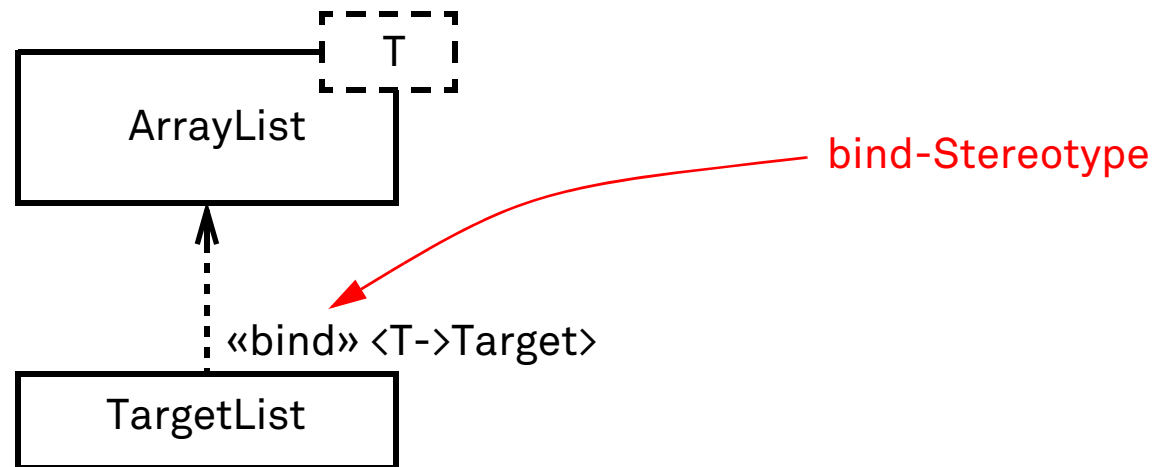
- ❑ in einer Methode von *K* über einen Parameter Zugriff auf ein Objekt von *U* zu bekommen .
- ❑ in einer Methode von *K* ein Objekt von *U* zu erzeugen.



Klassendiagramm – Abhängigkeit

(Fortsetzung)

Binden von Typparametern



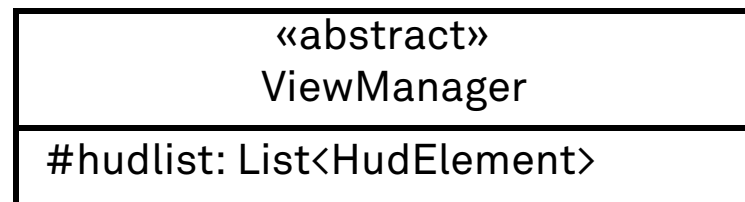
Legt eine neue Klasse
mit eigenem Namen an.

Klassendiagramm – Attribute von Klassen

(Fortsetzung)

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

- textuell in der Visualisierung der Klasse.



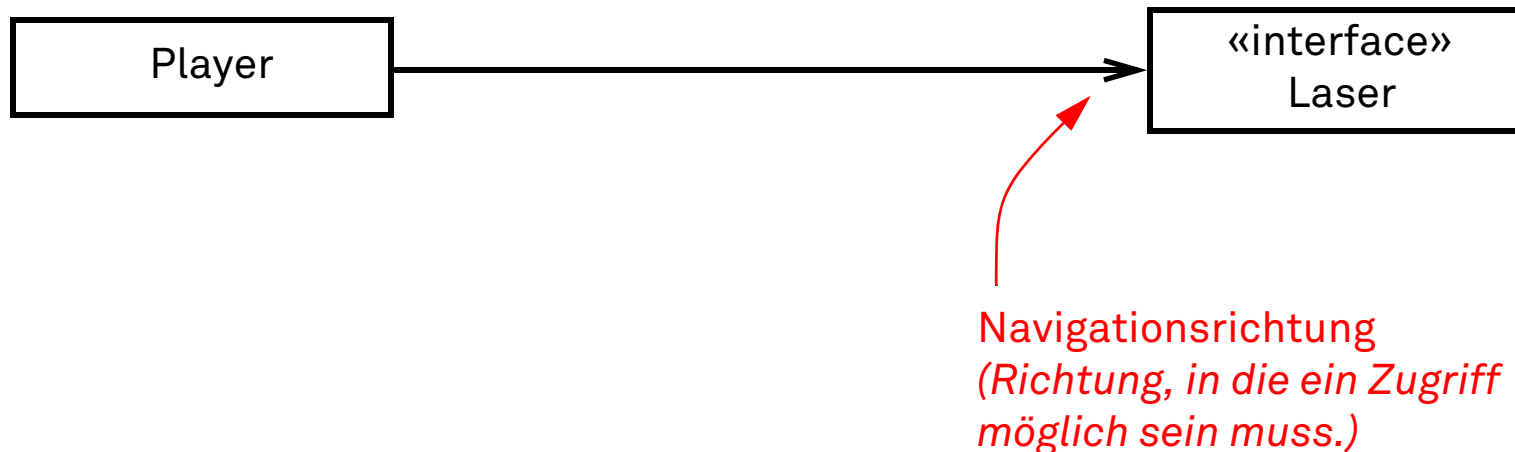
Dann können aber die Beziehungen zwischen den Klassen nicht unmittelbar in der Darstellung erkannt werden.

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

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

Klassendiagramm – Assoziation

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



Klassendiagramm – Assoziation

(Fortsetzung)

Präzisierungsmöglichkeiten

(Beispiel nicht aus SWT-Starfighter)



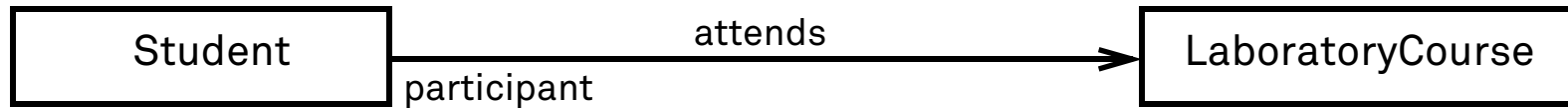
Rolle
(aus Sicht der Klasse am
anderen Ende der Assoziation)

Charakterisierung der Assoziation
(Assoziationsname, vorzugsweise ein Verb)

Klassendiagramm – Assoziation

(Fortsetzung)

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



```
public class Student {
    LaboratoryCourse myCourse;
}

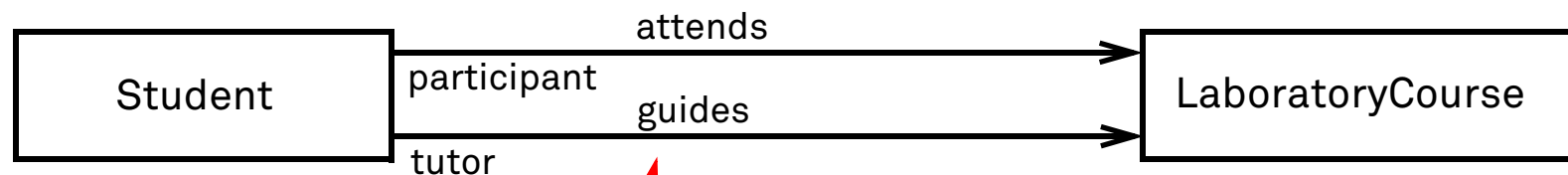
public class LaboratoryCourse { }
```

Klassendiagramm – Assoziation

(Fortsetzung)

Präzisierungsmöglichkeiten

(Beispiel nicht aus SWT-Starfighter)



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

Klassendiagramm – Assoziation

(Fortsetzung)

Präzisierungsmöglichkeiten

(Beispiel nicht aus SWT-Starfighter)



Multiplizität

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

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

Aussage des Diagramms:

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

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

Klassendiagramm – Assoziation

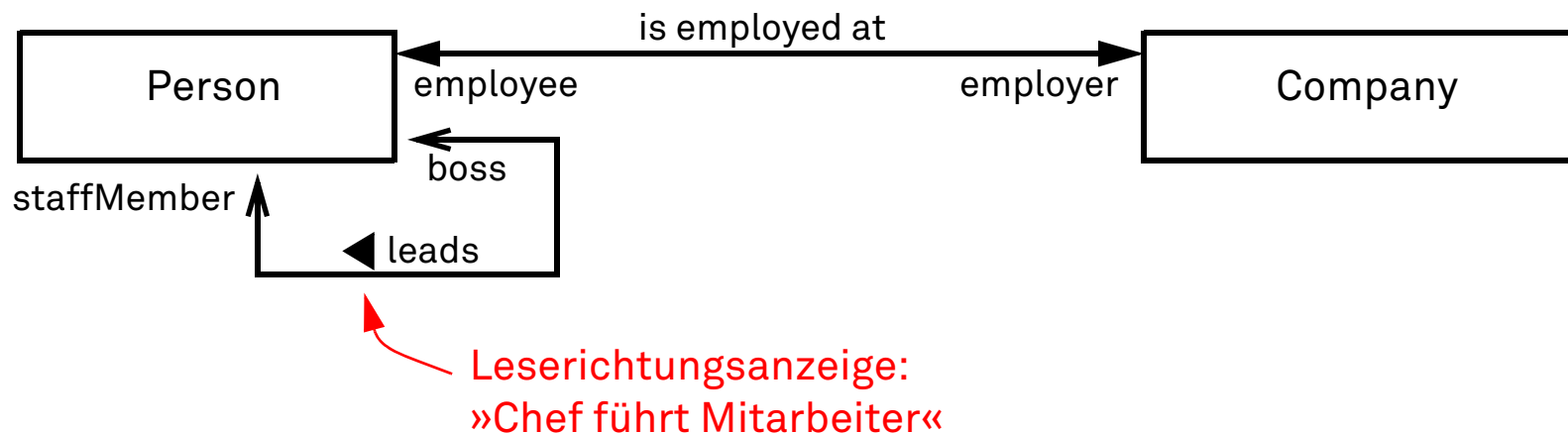
(Fortsetzung)

Präzisierungsmöglichkeiten

(Beispiel nicht aus SWT-Starfighter)

Navigationsrichtung

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

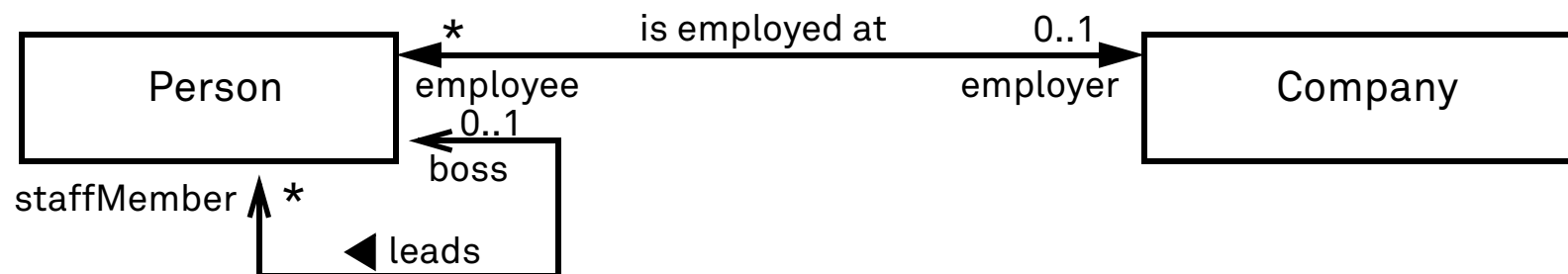


Klassendiagramm – Assoziation

(Fortsetzung)

Präzisierungsmöglichkeiten: mit Angabe von Multiplizitäten

(Beispiel nicht aus SWT-Starfighter)



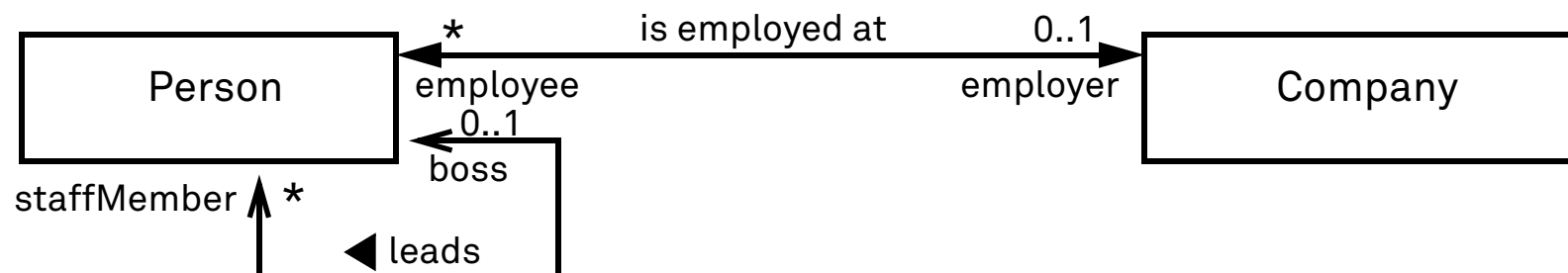
Klassendiagramm – Assoziation

(Fortsetzung)

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

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

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



Klassendiagramm – Assoziation

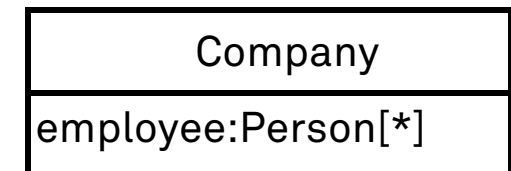
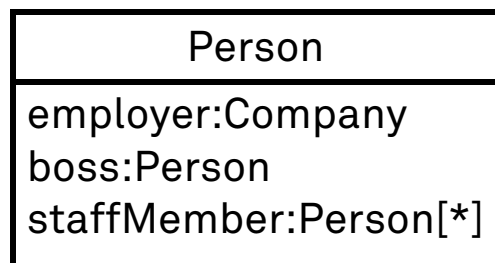
(Fortsetzung)

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

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

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

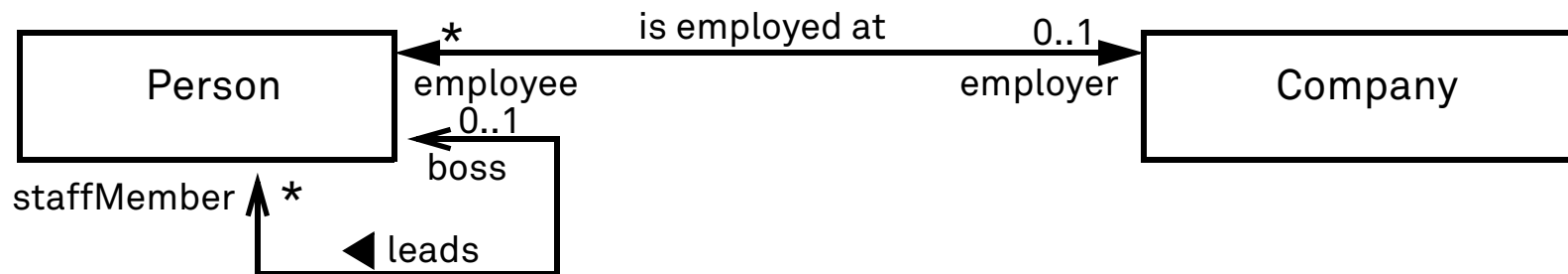
entspricht aber auch



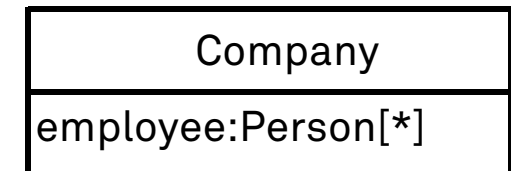
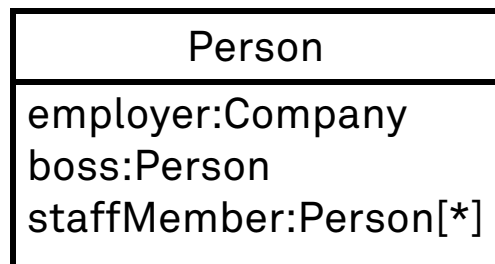
Klassendiagramm – Assoziation

(Fortsetzung)

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



entspricht also



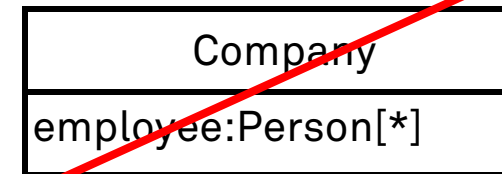
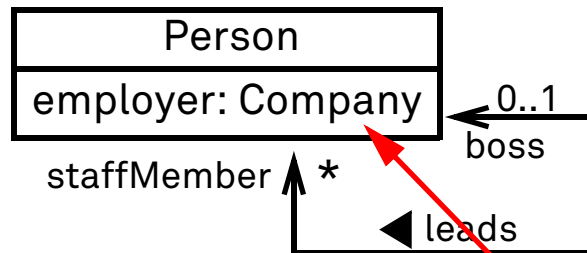
Aber: Die untere Darstellung enthält keine Visualisierung der Beziehungen zwischen Person- und Company-Objekten.

Klassendiagramm – Assoziation

(Fortsetzung)

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

(Beispiel nicht aus SWT-Starfighter)



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

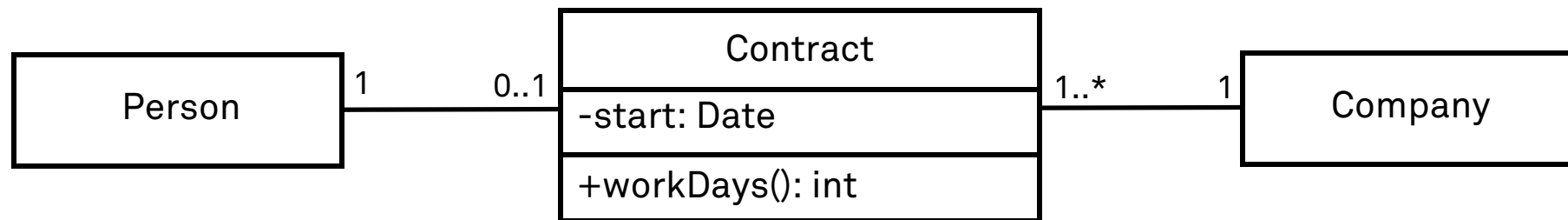
Meist ist das bei Klassen wie z.B. String der Fall,
die zum Standardumfang der eingesetzten
Programmiersprache gehören.

Klassendiagramm – Assoziation

(Fortsetzung)

Assoziation – Abstraktionsmöglichkeit: **Assoziationsklasse**

(Beispiel nicht aus SWT-Starfighter)



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

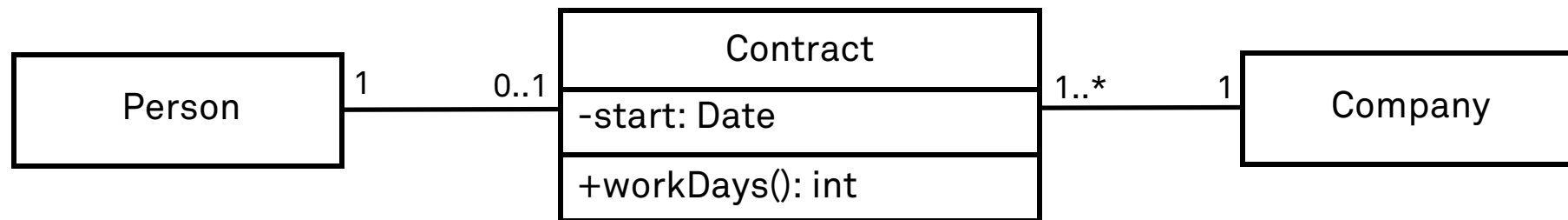
Dieser Kontext kann aber nur durch Analyse aller drei Klassen und der dazwischen stehenden Beziehungen erkannt werden.

Klassendiagramm – Assoziation

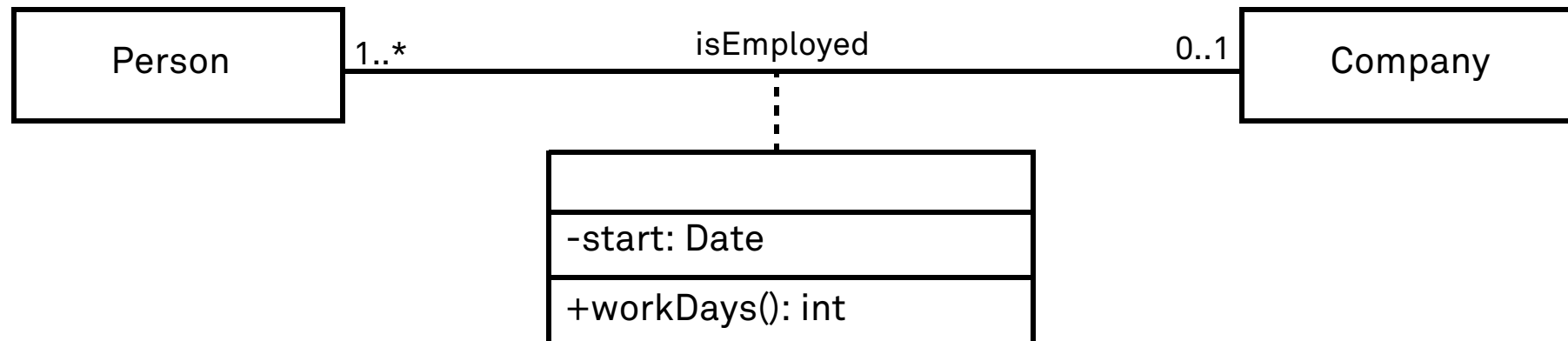
(Fortsetzung)

Assoziation – Abstraktionsmöglichkeit: **Assoziationsklasse**

(Beispiel nicht aus SWT-Starfighter)



Der Kontext kann durch folgende Modellierung deutlich gemacht werden:



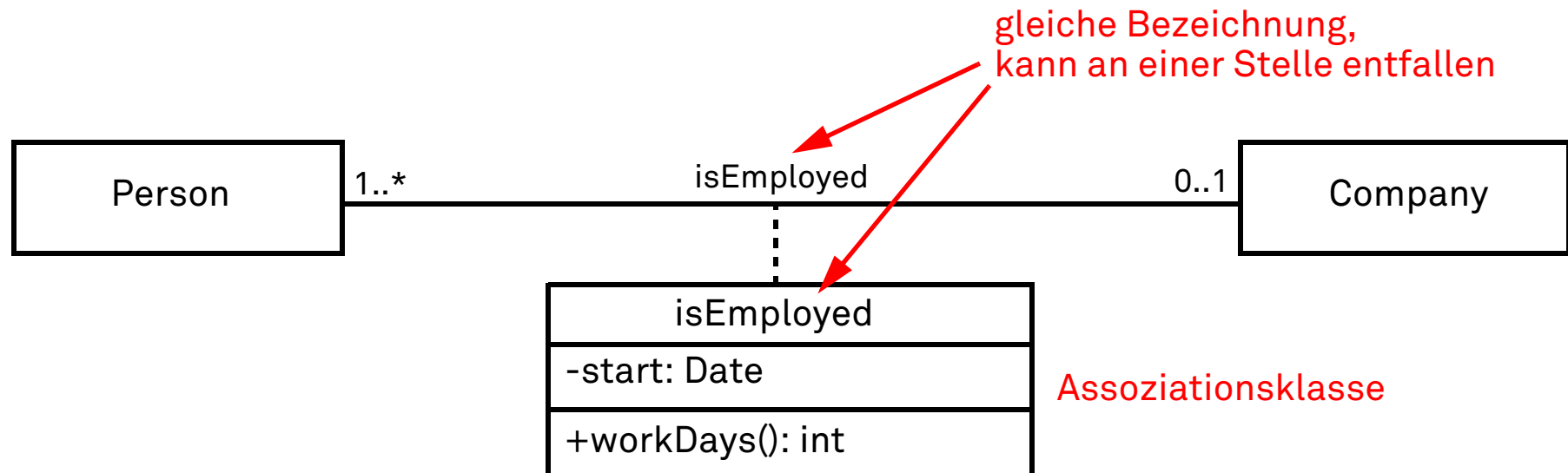
Klassendiagramm – Assoziation

(Fortsetzung)

Assoziation – Abstraktionsmöglichkeit: **Assoziationsklasse**

(Beispiel nicht aus SWT-Starfighter)

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

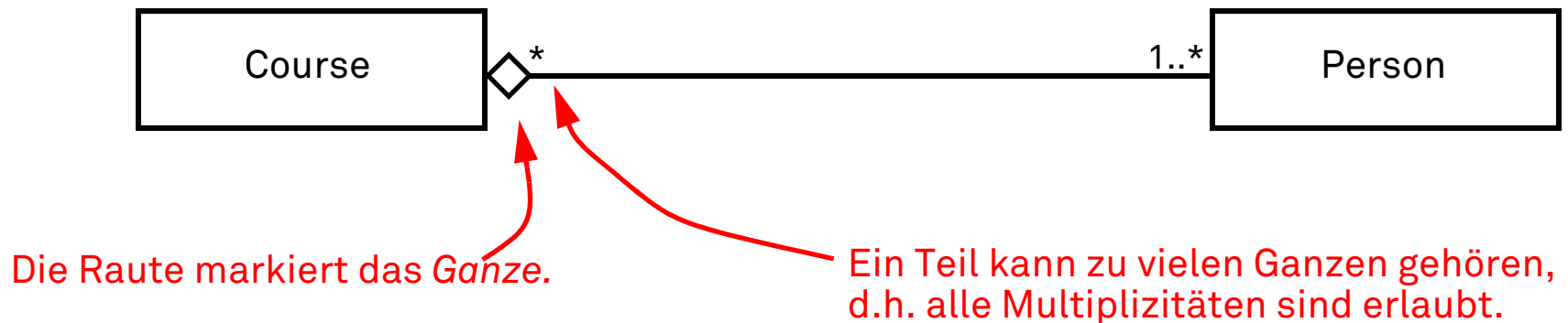


Klassendiagramm – Aggregation

Aggregation ist eine spezielle Form der Assoziation:

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

(Beispiel nicht aus SWT-Starfighter)



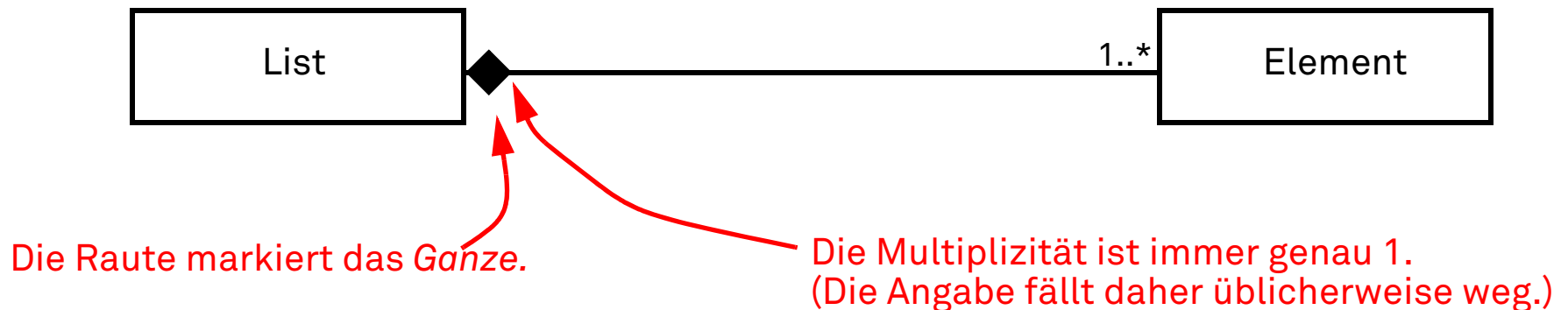
Klassendiagramm – Komposition

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

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

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

(Beispiel nicht aus SWT-Starfighter)



Klassendiagramm – Anmerkungen zu Assoziationen

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

Klassendiagramm – Anmerkungen zur Syntax

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

Klassendiagramm (Wiederholung)

Klassendiagramme können genutzt werden, um

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

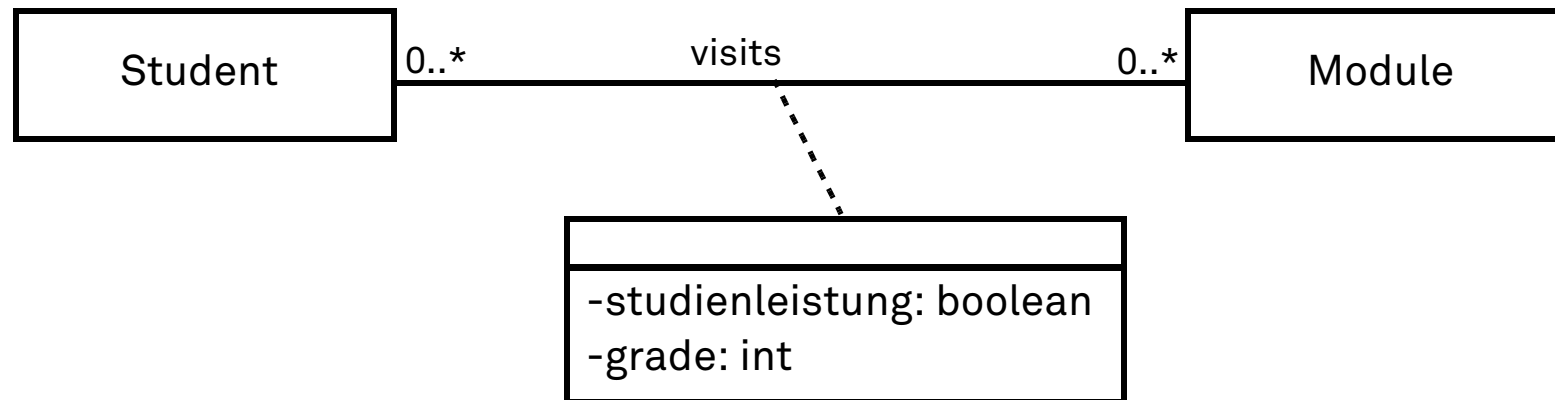
Die Modellierung kann auf unterschiedlichen Abstraktionsniveaus erfolgen und ist abhängig vom

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

Klassendiagramm – Beispiel

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

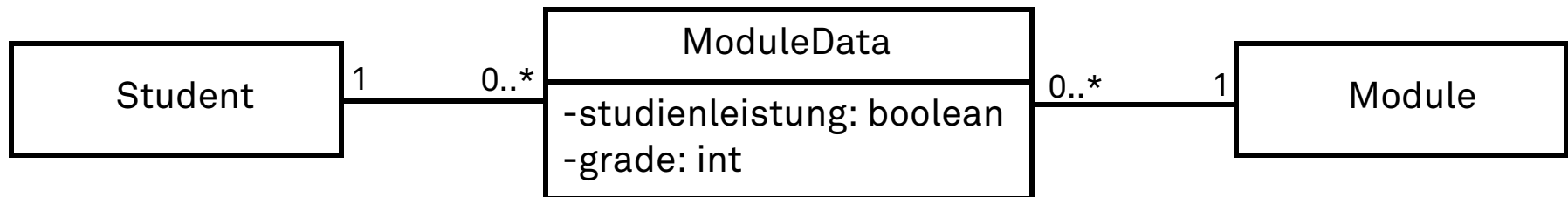
(Beispiel nicht aus SWT-Starfighter)



Klassendiagramm – Beispiel

(Fortsetzung)

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



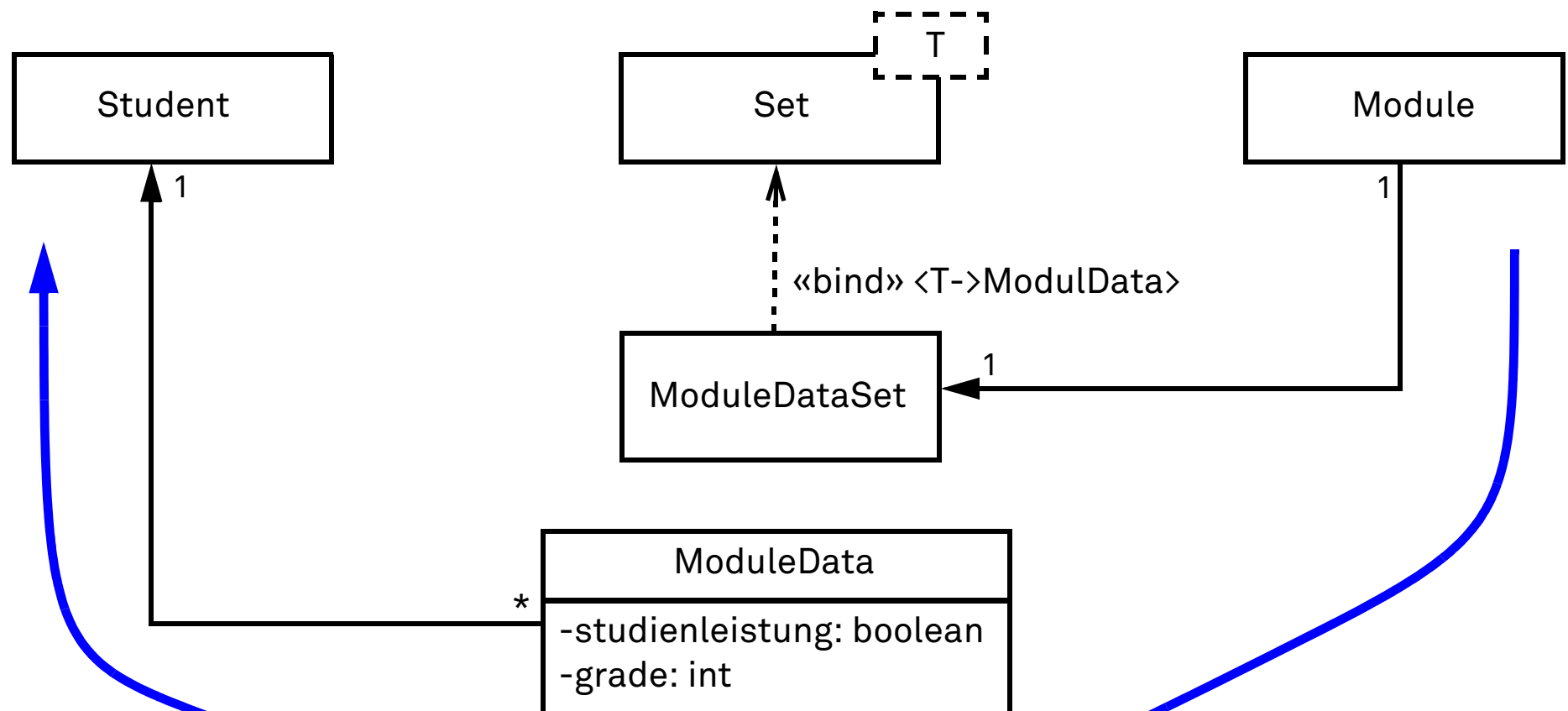
Klassendiagramm – Beispiel

(Fortsetzung)

Vorgabe für die (direkte) Umsetzung in eine Implementierung:

Nutzung der generischen Klasse Set<T>

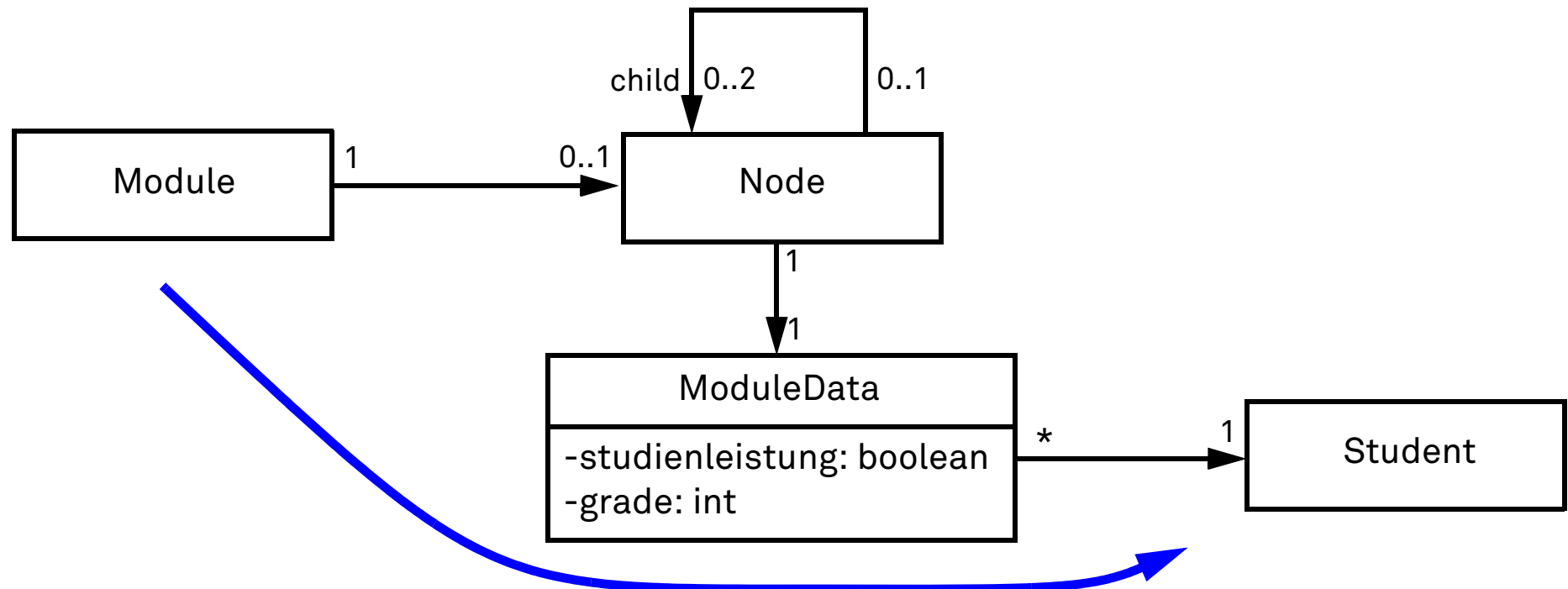
– erlaubt nur die Navigation von einem Modul zu den teilnehmenden Studierenden



Klassendiagramm – Beispiel

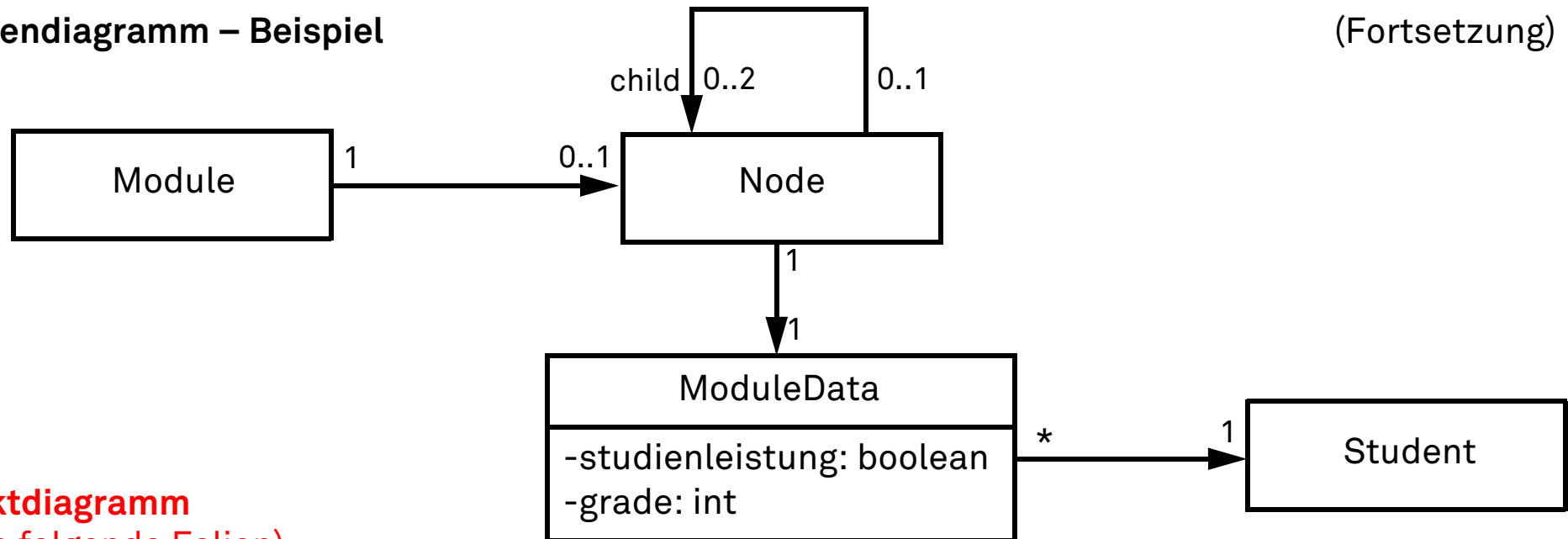
(Fortsetzung)

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

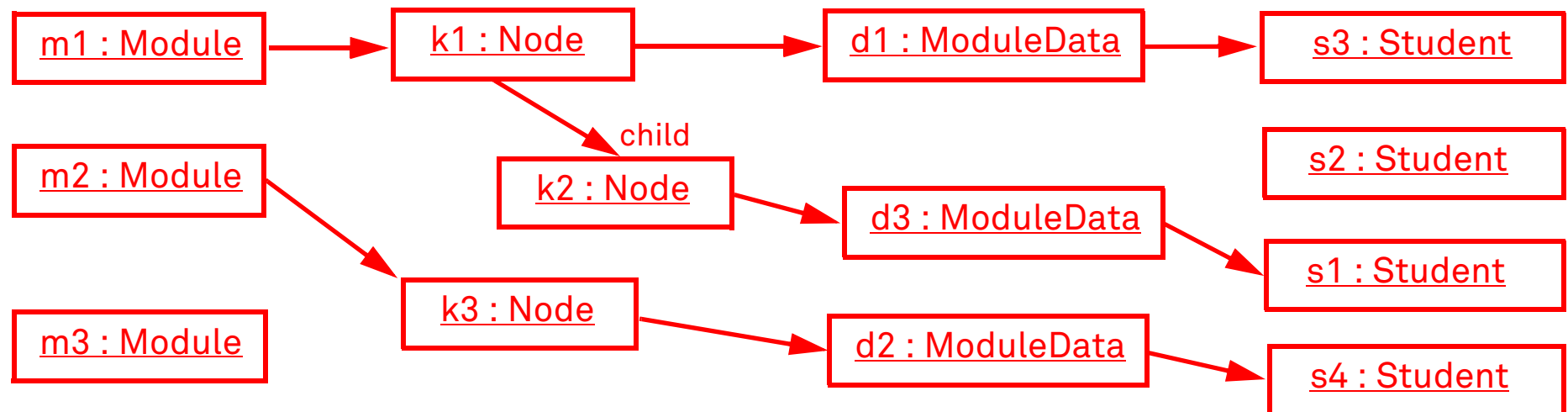


Klassendiagramm – Beispiel

(Fortsetzung)



Objektdiagramm
(siehe folgende Folien)



Klassendiagramm (Wiederholung)

Klassendiagramme können genutzt werden, um

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

Anmerkungen:

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

Die Modellierung muss der gewünschten Abstraktionsstufe entsprechen.

Klassendiagramm – Objektdiagramm

Klassendiagramm

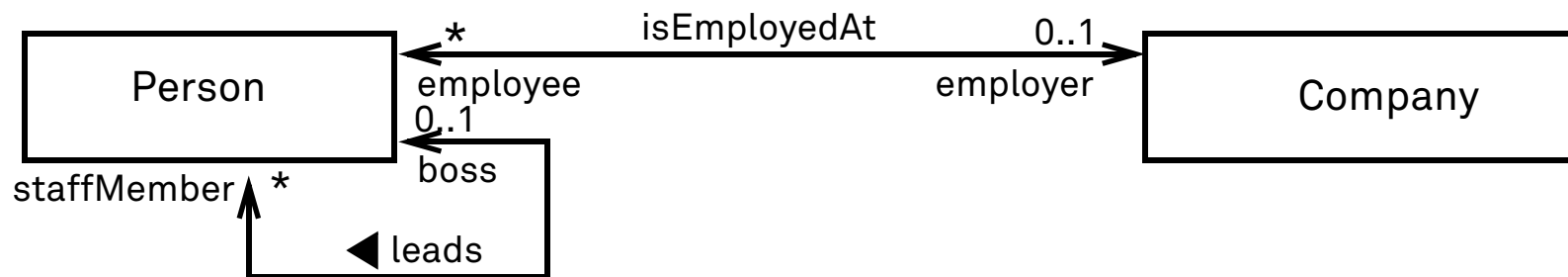
- ❑ beschreibt Beziehungen zwischen **Klassen** in Form von
 - Spezialisierung,
 - Realisierung,
 - Abhängigkeiten.
- ❑ beschreibt die **möglichen** Beziehungen zwischen **Objekten/Instanzen** der Klassen durch Assoziationen.

Objektdiagramm

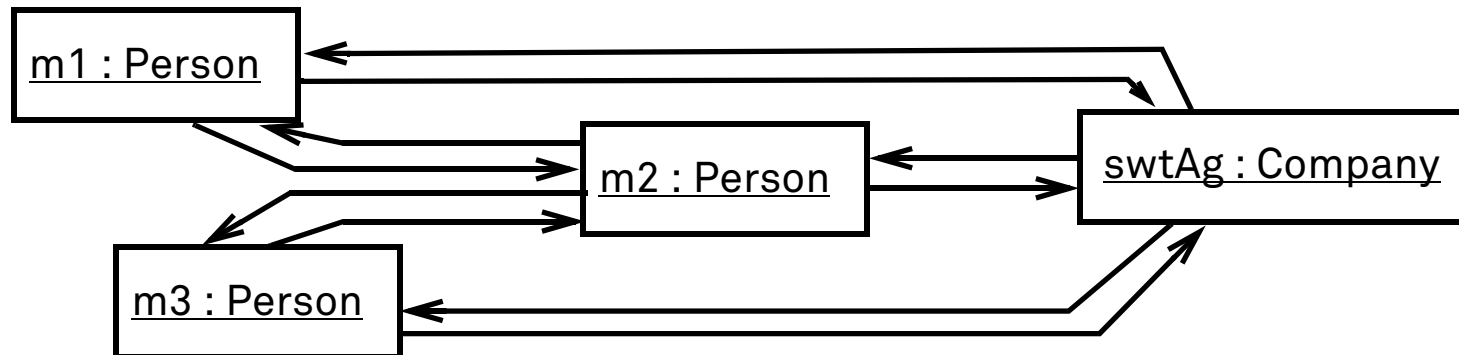
- ❑ zeigt **eine konkrete** Konfiguration von Objekten, die die Vorgaben des zugehörigen Klassendiagramms einhält.
- ❑ enthält ausschließlich Objekte.
- ❑ enthält nur die zwischen diesen Objekten bestehenden Verbindungen.
- ❑ ist i.d.R. nur ein Beispiel aus vielen möglichen Konfigurationen.
- ❑ dient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

Objektdiagramm

Klassendiagramm: Beziehungen zwischen Klassen/Typen



Objektdiagramm: Beziehungen zwischen Objekten der Klassen

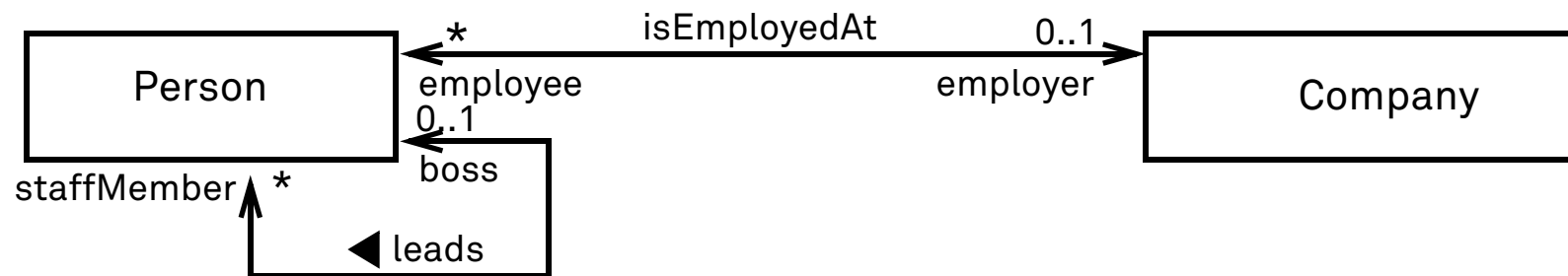


Literatur: Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 49-51
http://link.springer.com/chapter/10.1007/3-540-30950-0_4

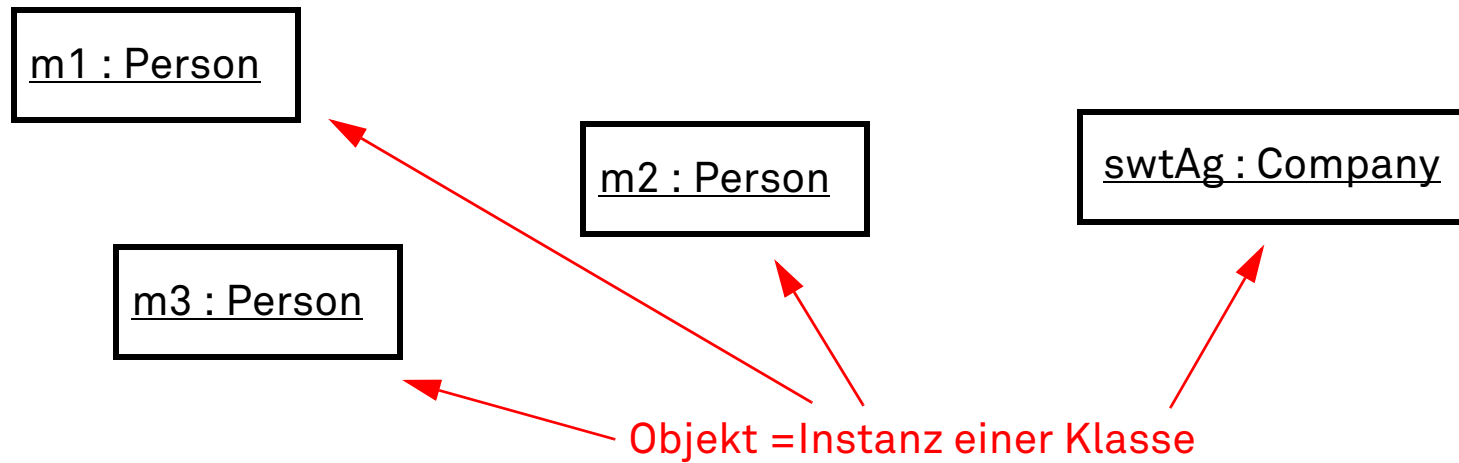
Objektdiagramm

(Fortsetzung)

Klassendiagramm:



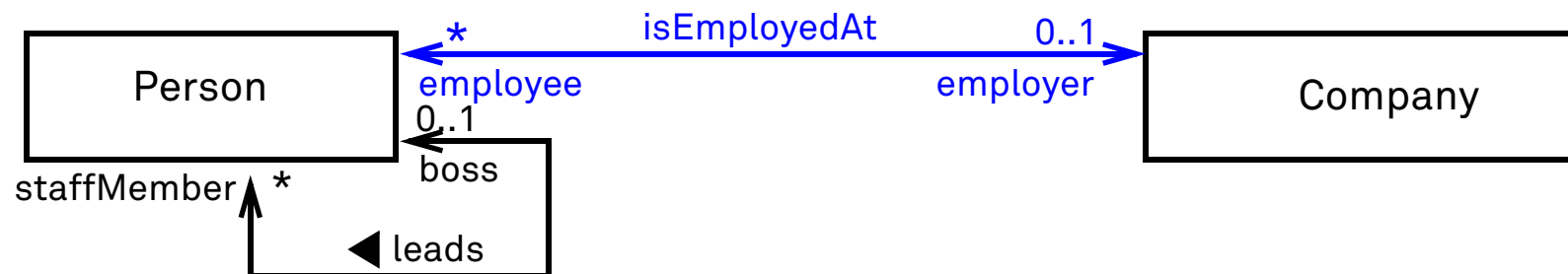
Objektdiagramm:



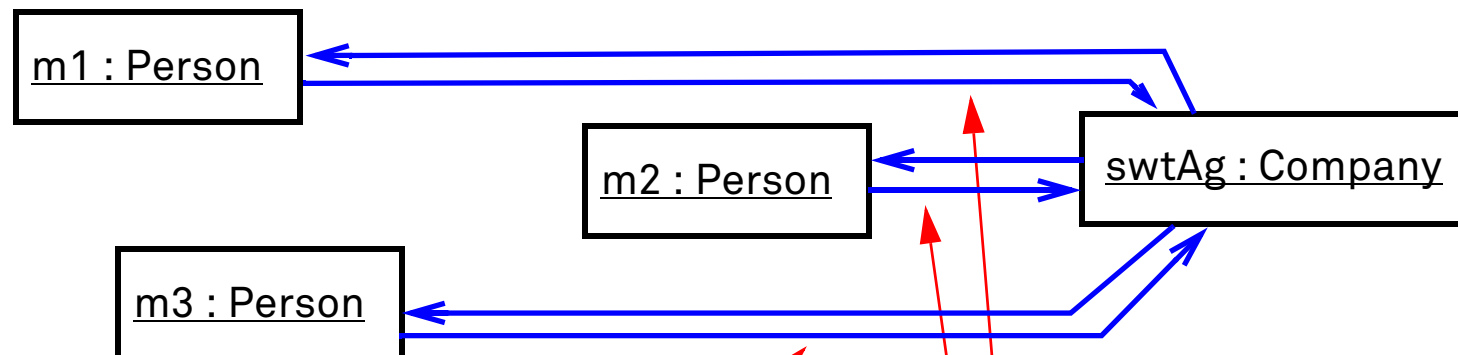
Objektdiagramm

(Fortsetzung)

Klassendiagramm:



Objektdiagramm:

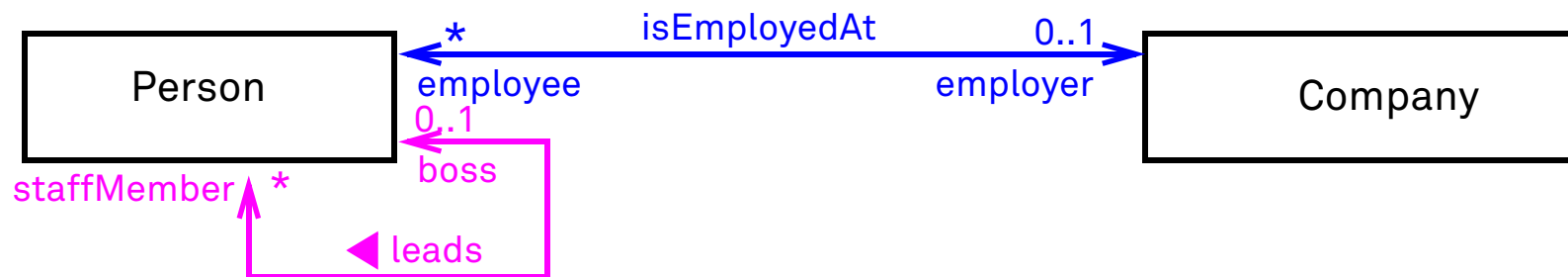


Link = konkrete Verbindung gemäß einer Assoziation

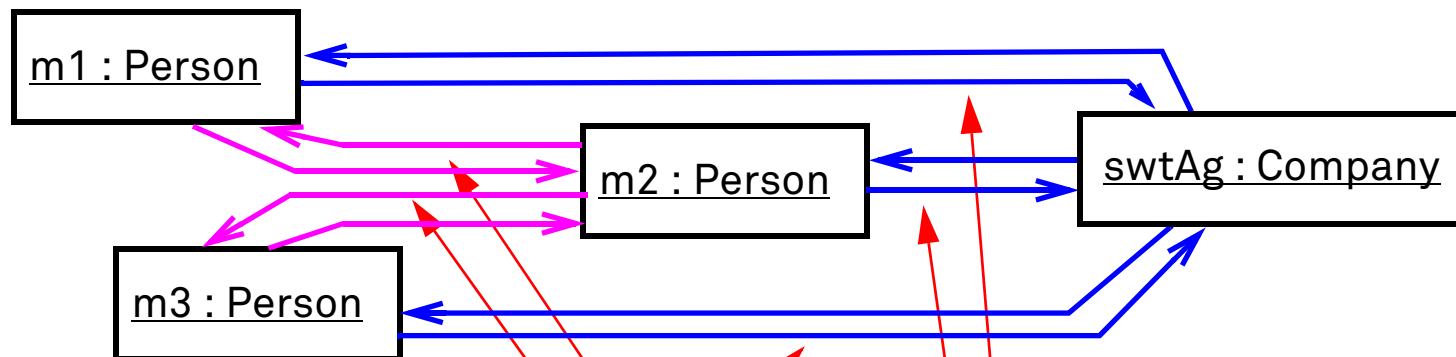
Objektdiagramm

(Fortsetzung)

Klassendiagramm:



Objektdiagramm:



Link = konkrete Verbindung gemäß einer Assoziation

Objektdiagramm

(Fortsetzung)

Ergänzung: Benennung von Objekten

p : Person

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

: Person

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

x

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

.

Objekt mit unbekanntem Namen und unbekannter Klasse

.

ein weiteres Objekt mit unbekanntem Namen und unbekannter Klasse

Objektdiagramm

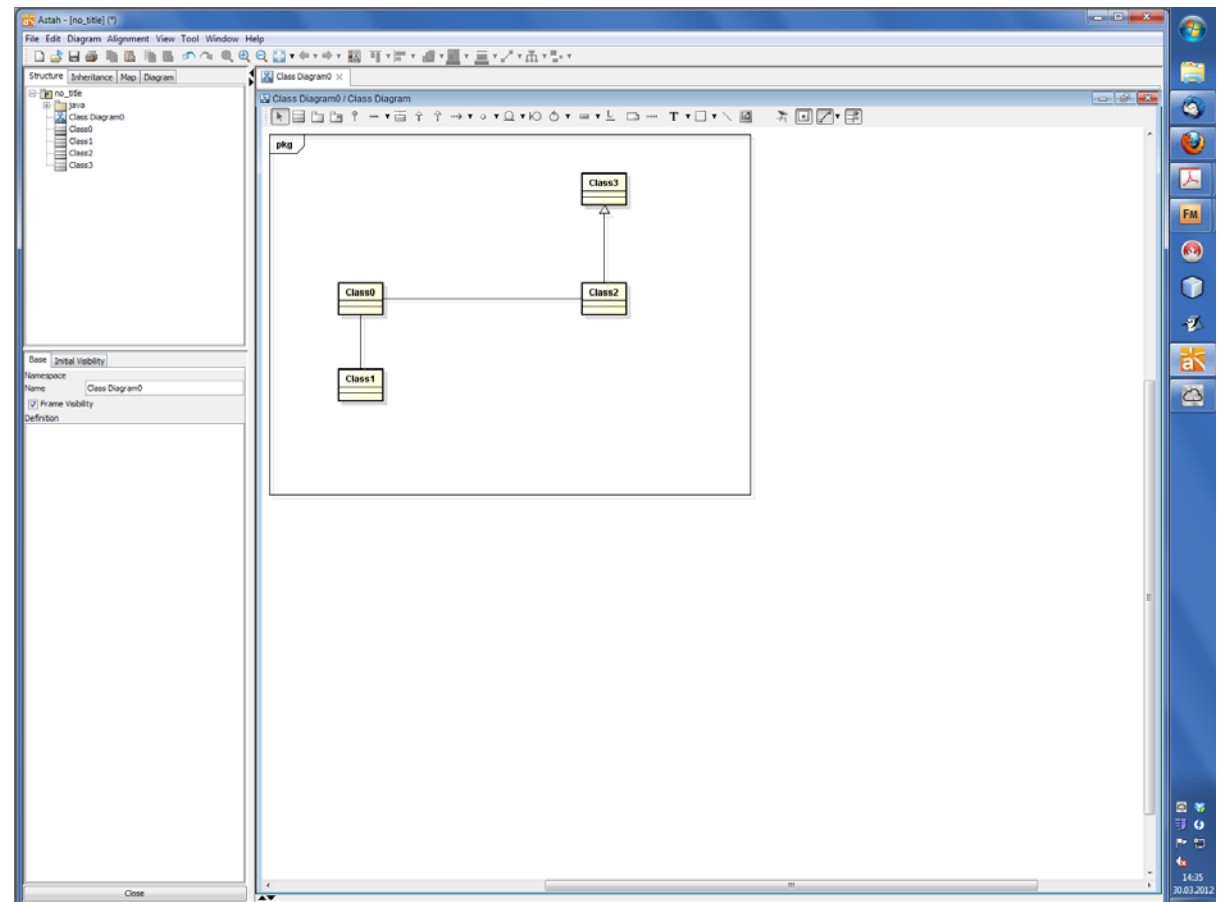
(Fortsetzung)

Zusammenfassung

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

Erstellung von UML-Diagrammen

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



Folien zur Vorlesung **Softwaretechnik**

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

Spezifikation von Klassen

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

Spezifikation durch Klassendiagramm

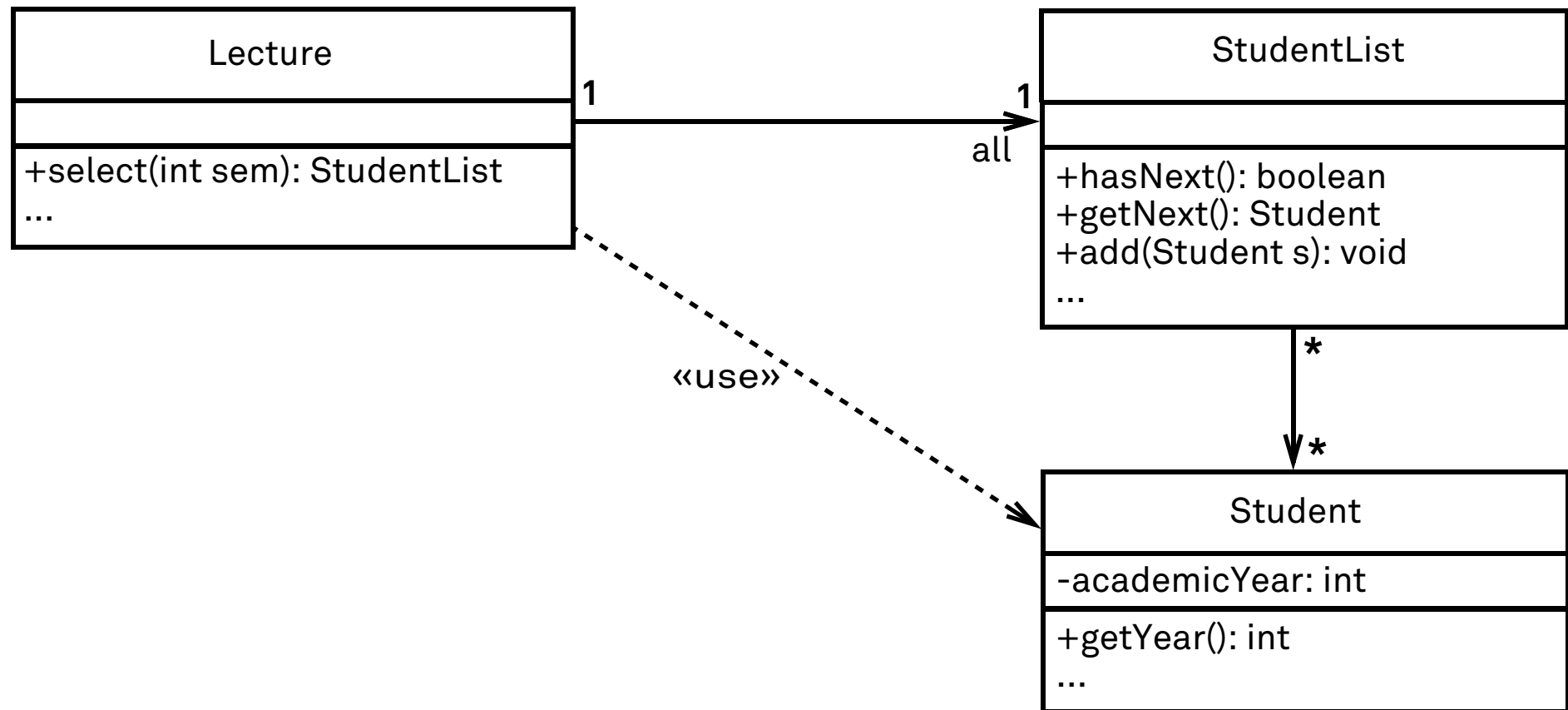
- ❑ mehrere Objekte (verschiedener) Klassen arbeiten zusammen durch Aufrufe ihrer Methoden.
- ❑ **Beispiel (*nicht aus SWT-Starfighter*):**
Erstellen einer Liste der SWT-Teilnehmer, die im zweiten Semester sind:
 - Studierenden-Objekt aus der Liste aller Teilnehmer auswählen
 - Studierenden-Objekt überprüfen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

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

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

Beispiel

Klassendiagramm



Beispiel

(Fortsetzung)

Methode der Klasse Lecture:

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

Methoden der Klasse StudentList:

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

Methode der Klasse Student:

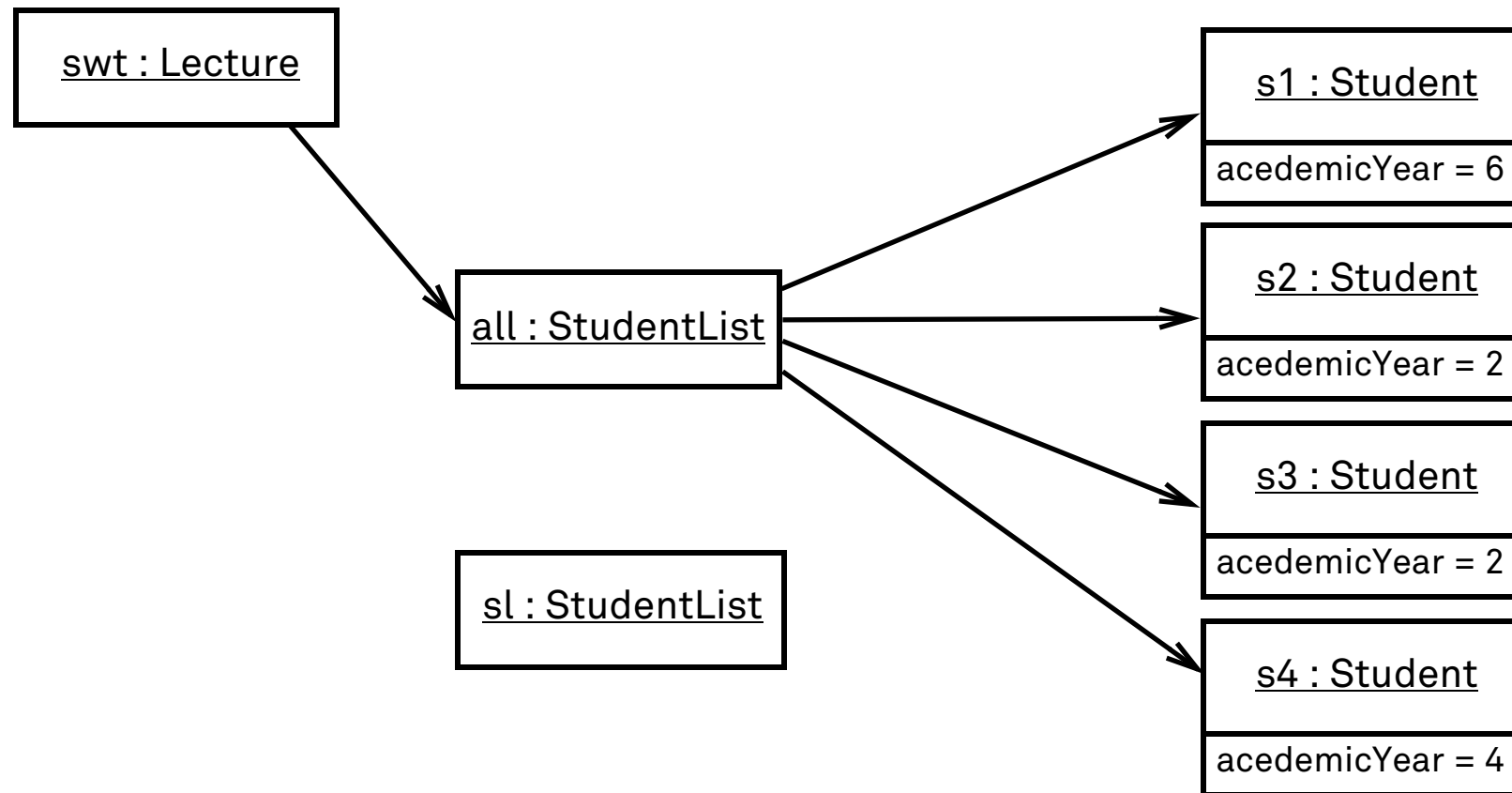
- ❑ `getYear(): int`
liefert die Semesterzahl aus dem Attribut des Student-Objekts

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

Beispiel

(Fortsetzung)

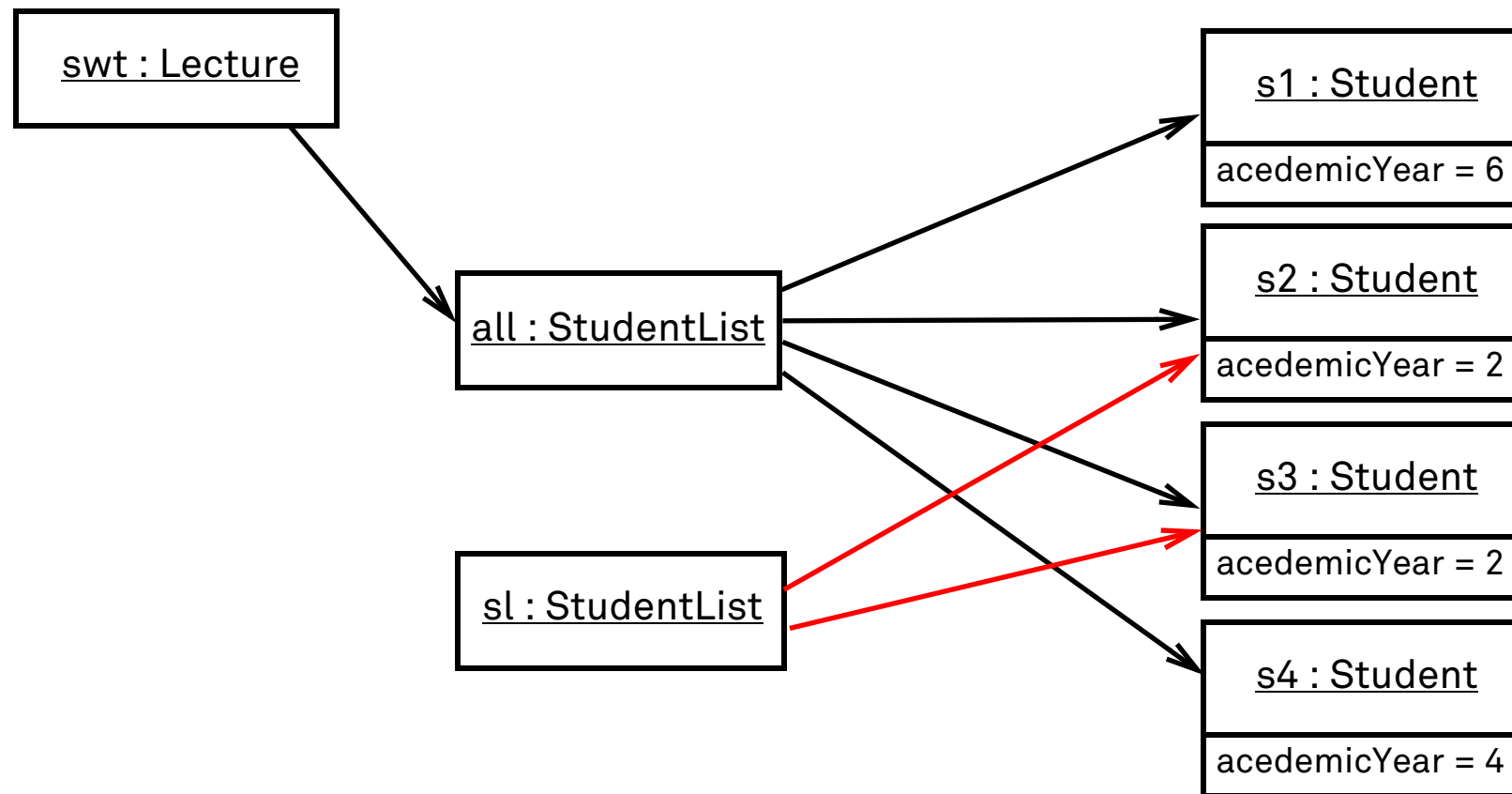
Objektdiagramm (vor dem Aufruf der Methode `select`)



Beispiel

(Fortsetzung)

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



Überblick

Hilfsmittel zur Beschreibung der Reihenfolge von Methodenaufrufen:

Sequenzdiagramm

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

Man spricht dabei auch allgemein von Nachrichtenaustausch:

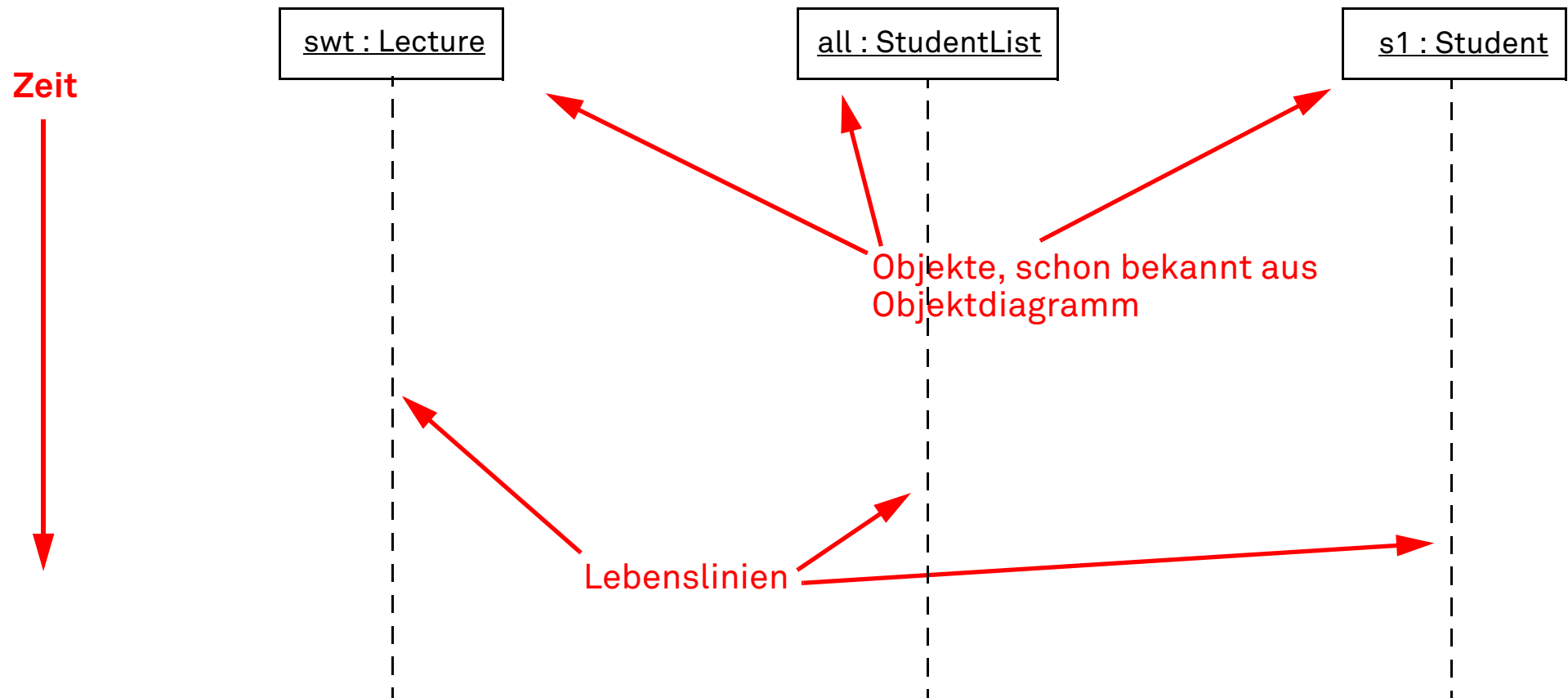
- Der *Aufruf einer Methode* m für ein Objekt o wird als
das *Senden einer Nachricht* an o aufgefasst, da die Methode bestimmt,
was o tun soll.
- Das *Beenden* von m (und eventuell das Zurückgeben eines Wertes)
werden als *Antwort* von o auf die Nachricht verstanden.

Aufbau eines Sequenzdiagramms

- ❑ Im Sequenzdiagramm werden dargestellt:
 - die handelnden **Objekte**,
 - deren Existenz als **senkrechte** Lebenslinie,
 - Methodenaufrufe als **waagerechte** Verbindungen zwischen Lebenslinien,
 - Phasen der Kontrolle als Blöcke auf Lebenslinien,
 - die Zeit durch die implizit nach unten verlaufende Zeitachse.

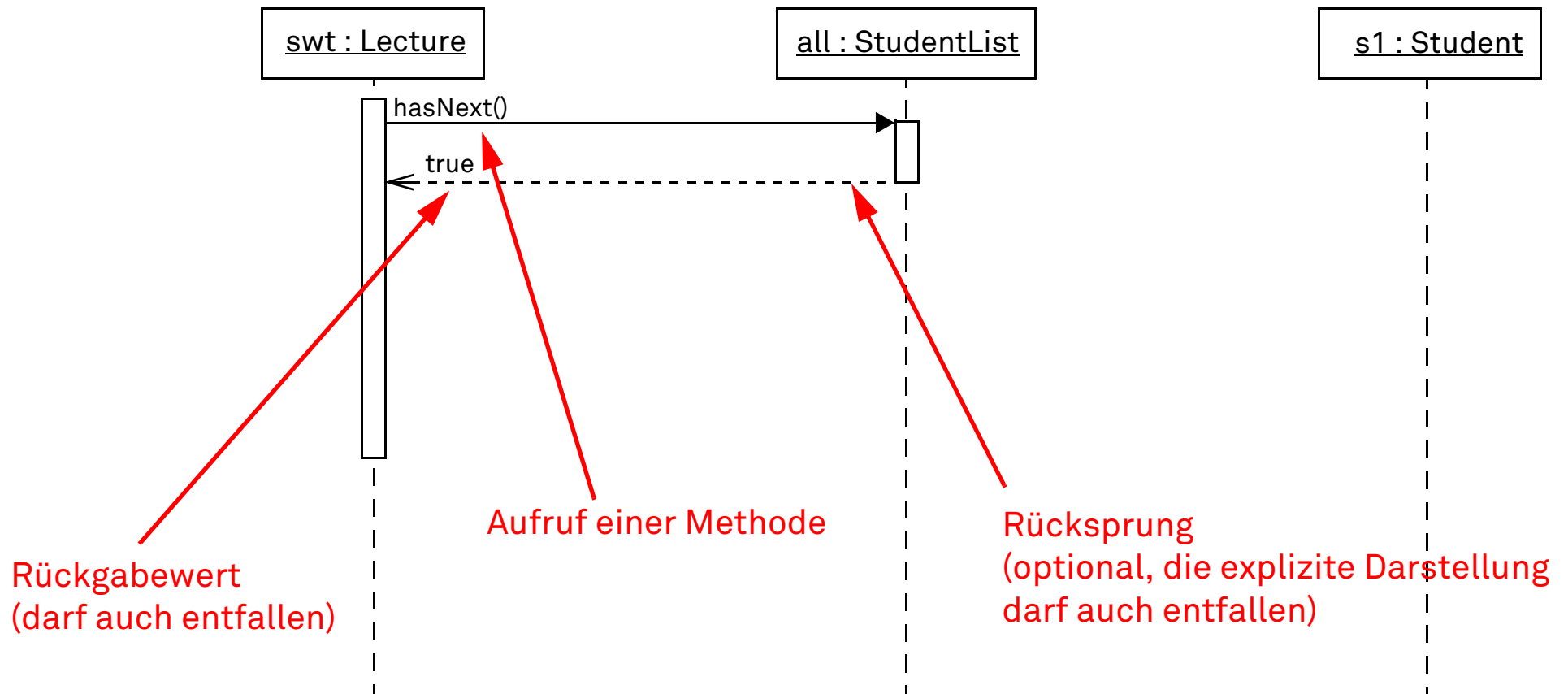
- ❑ Sichtbar wird dadurch
 - welche Objekte an Methodenaufrufen beteiligt sind,
 - von welcher Methode ein Aufruf ausgeht und welche Methode aufgerufen wird,
 - welche Methode den Ablauf über mehrere Aufrufe hinweg kontrolliert,
 - wie der Kommunikationsablauf insgesamt aufgebaut ist.

Beispiel – Sequenzdiagramm



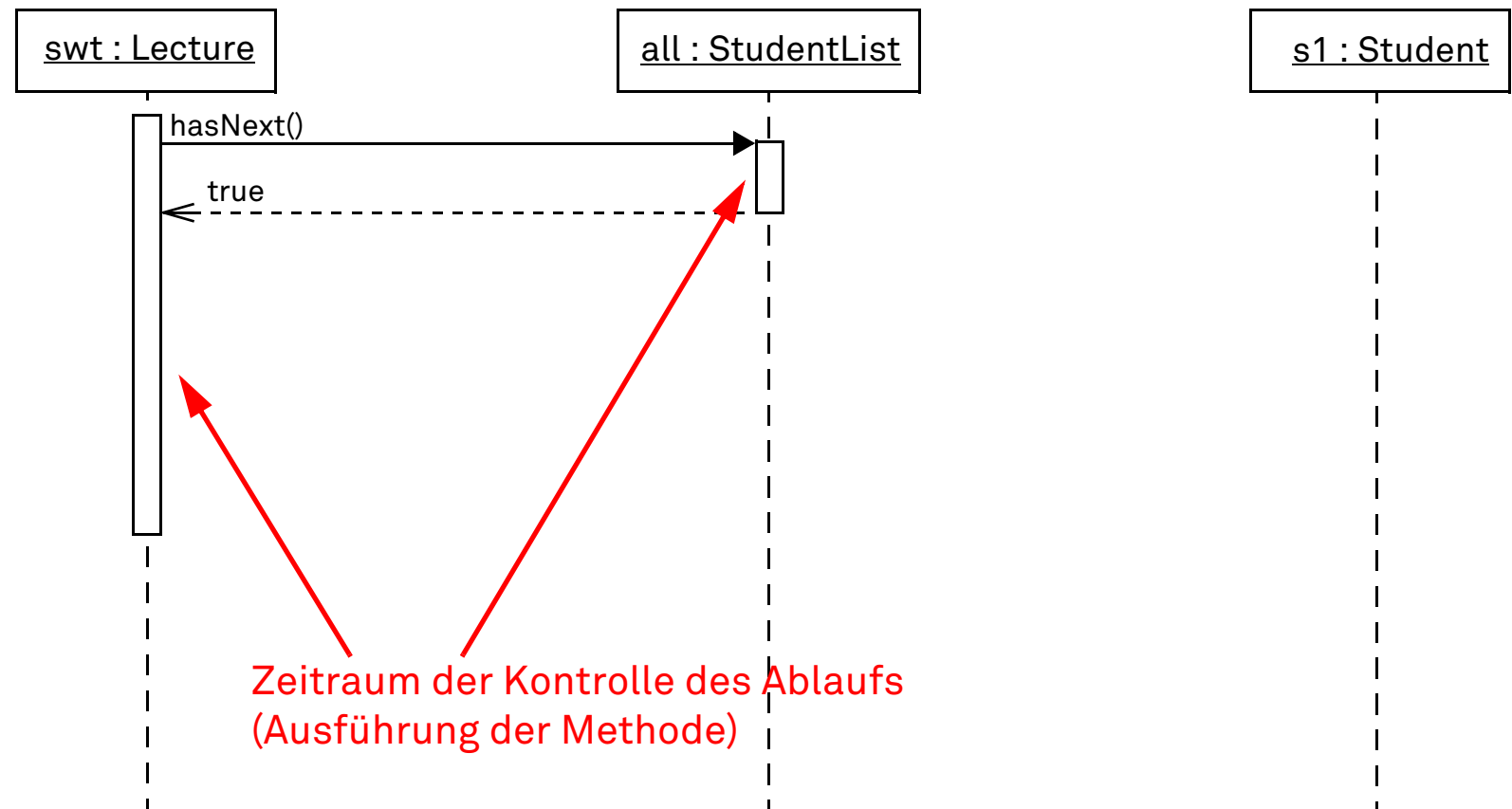
Beispiel – Sequenzdiagramm

(Fortsetzung)



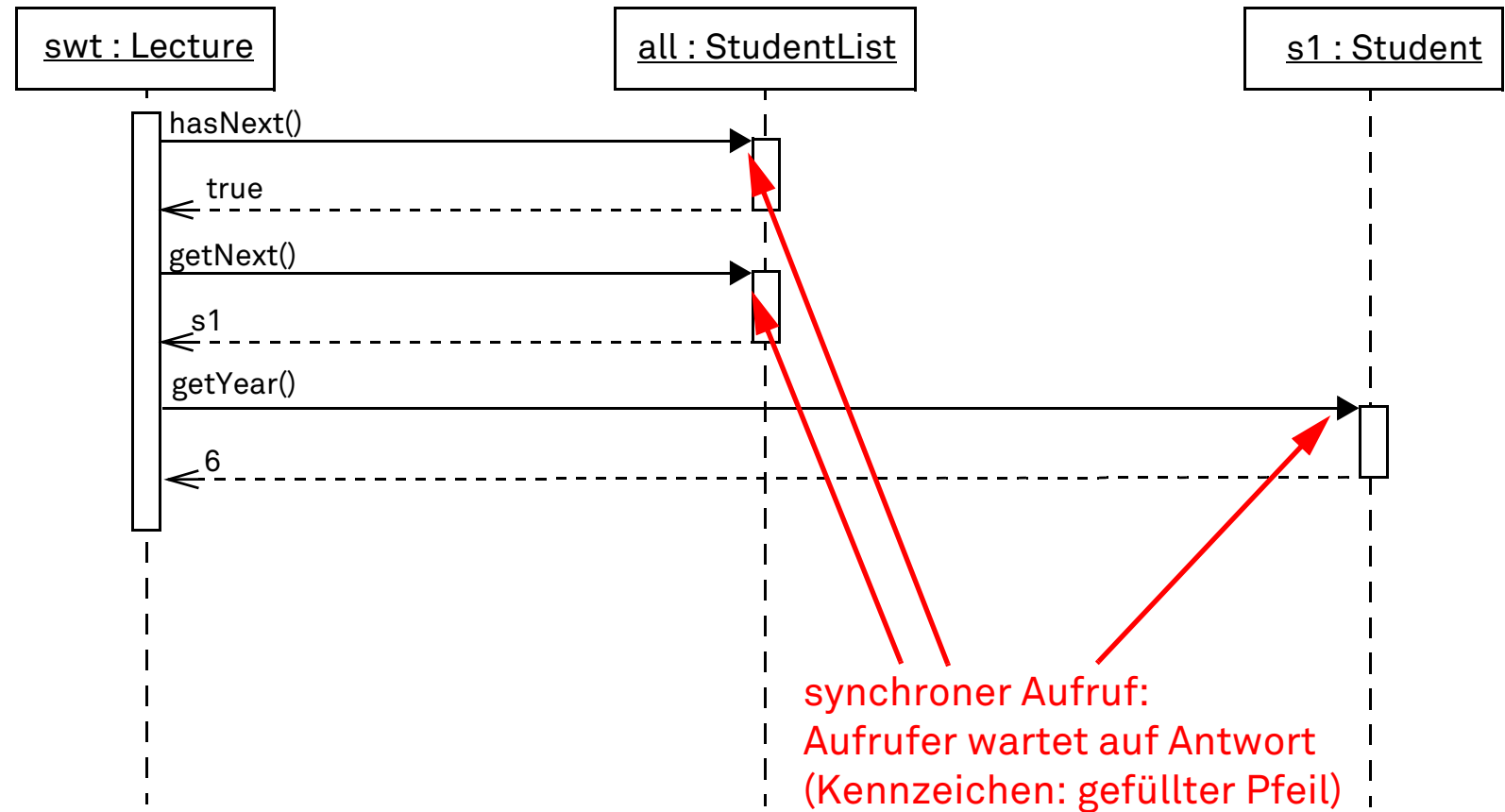
Beispiel – Sequenzdiagramm

(Fortsetzung)



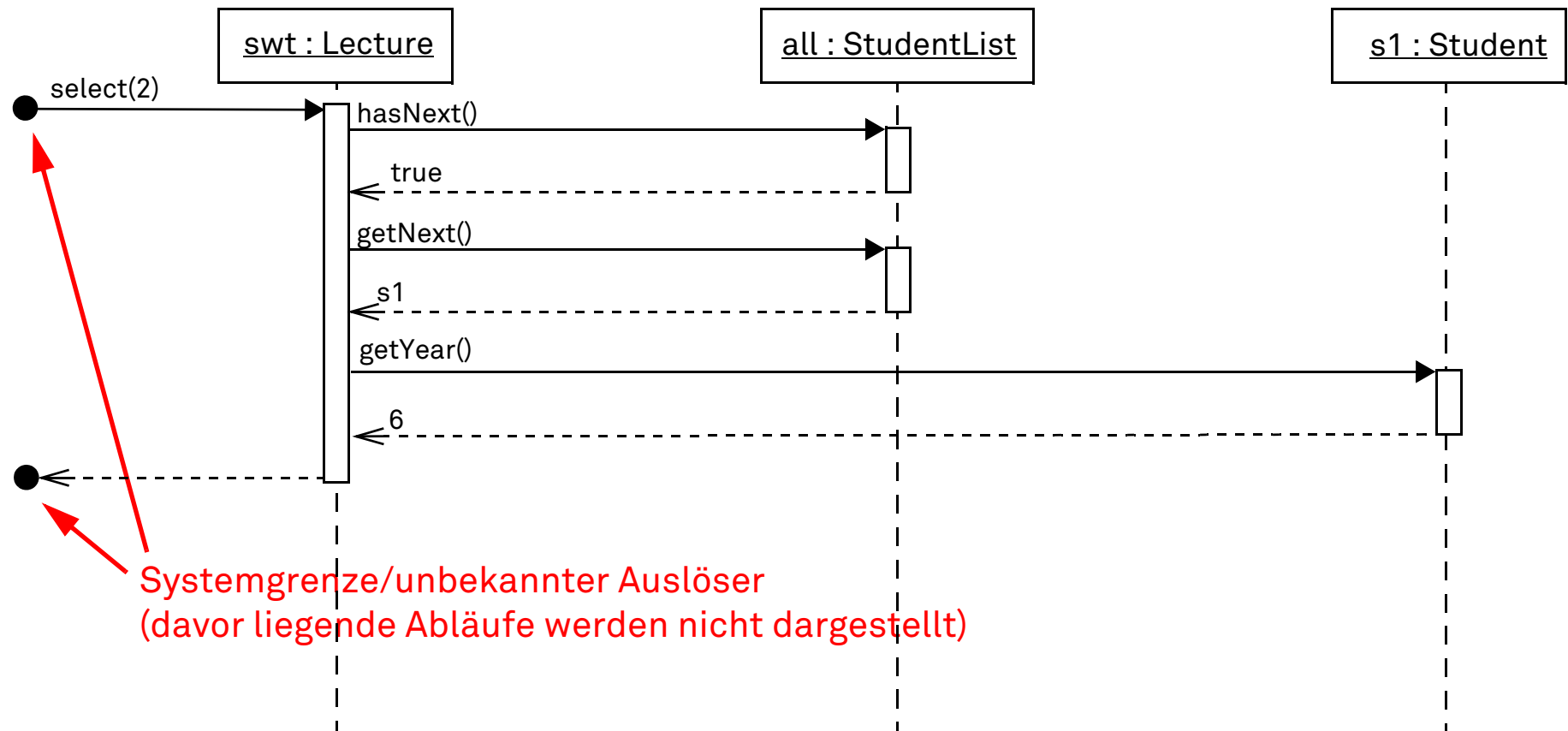
Beispiel – Sequenzdiagramm

(Fortsetzung)



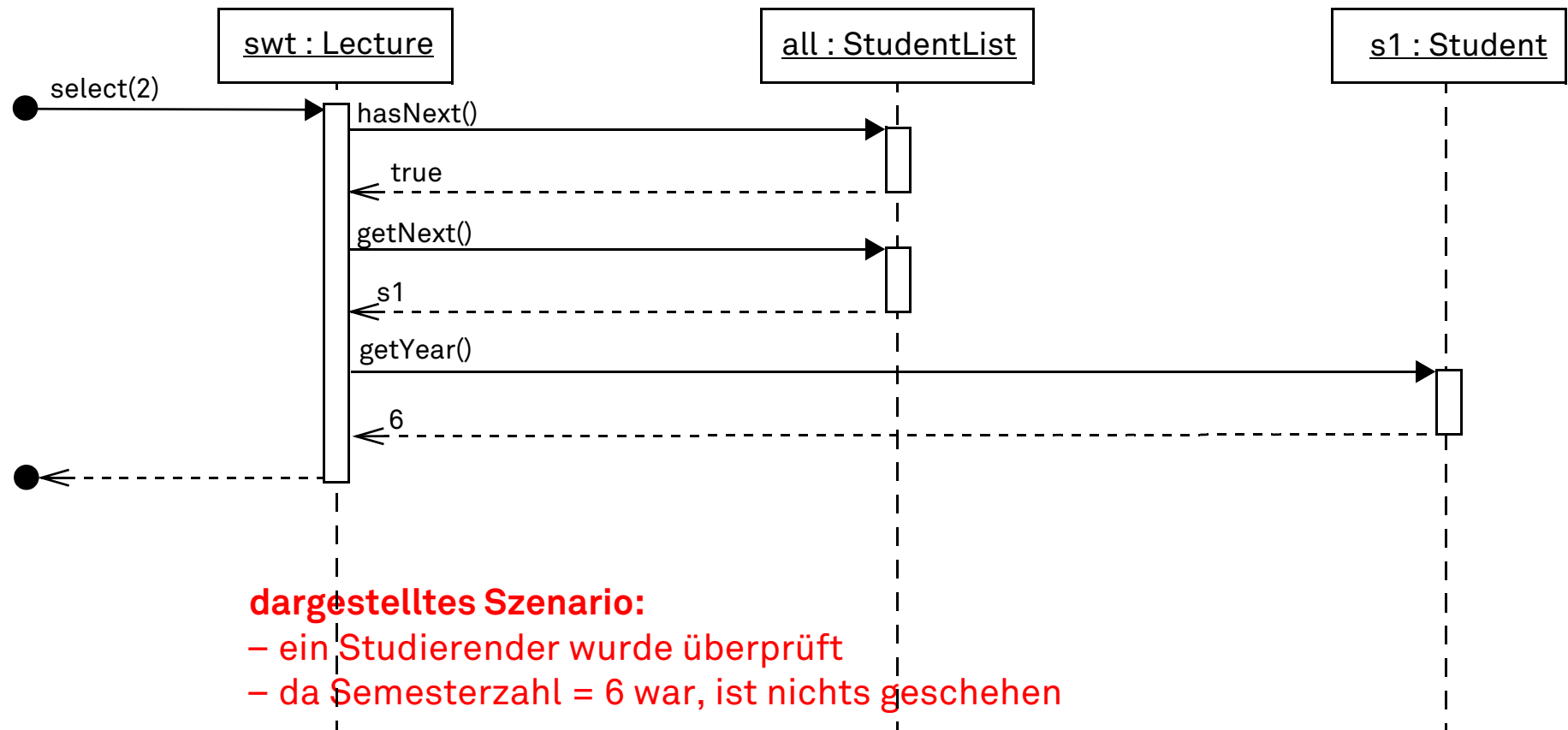
Beispiel – Sequenzdiagramm

(Fortsetzung)



Beispiel – Sequenzdiagramm

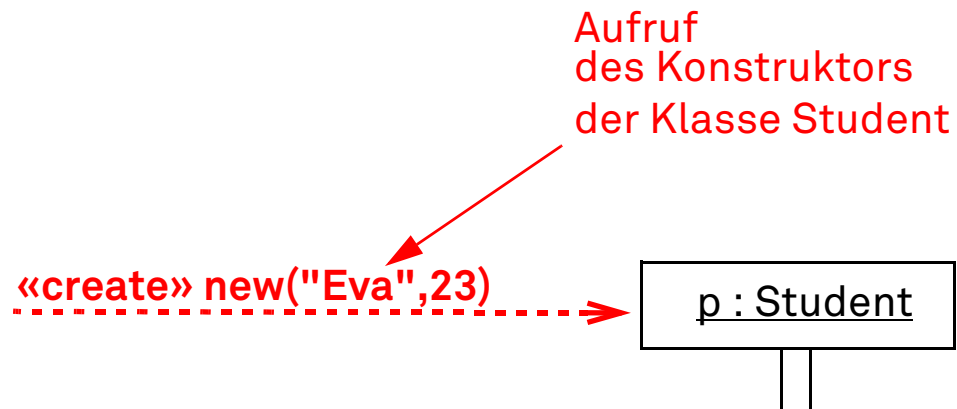
(Fortsetzung)



Sequenzdiagramm – Syntax

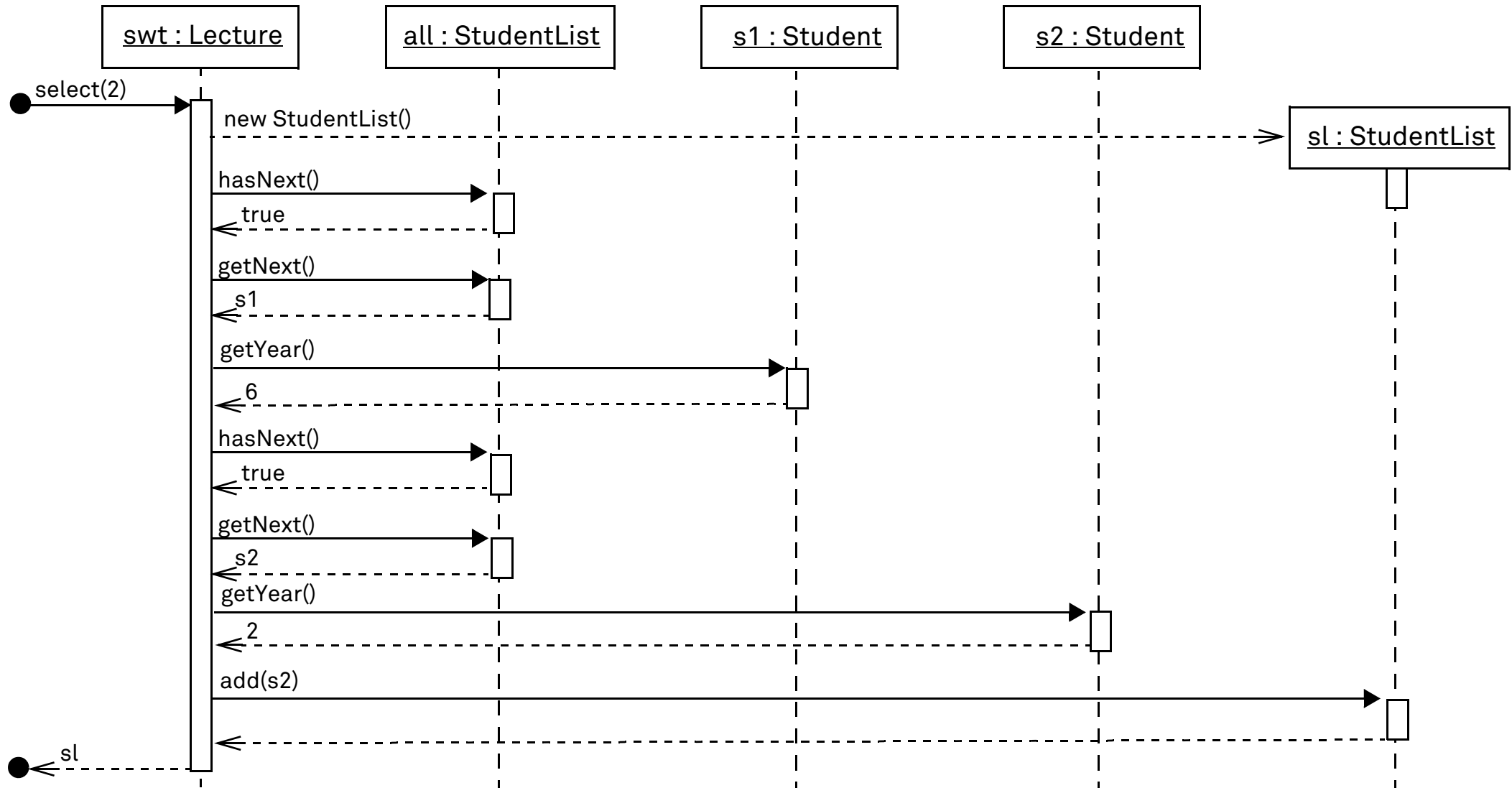
(Fortsetzung)

Erzeugen von Objekten



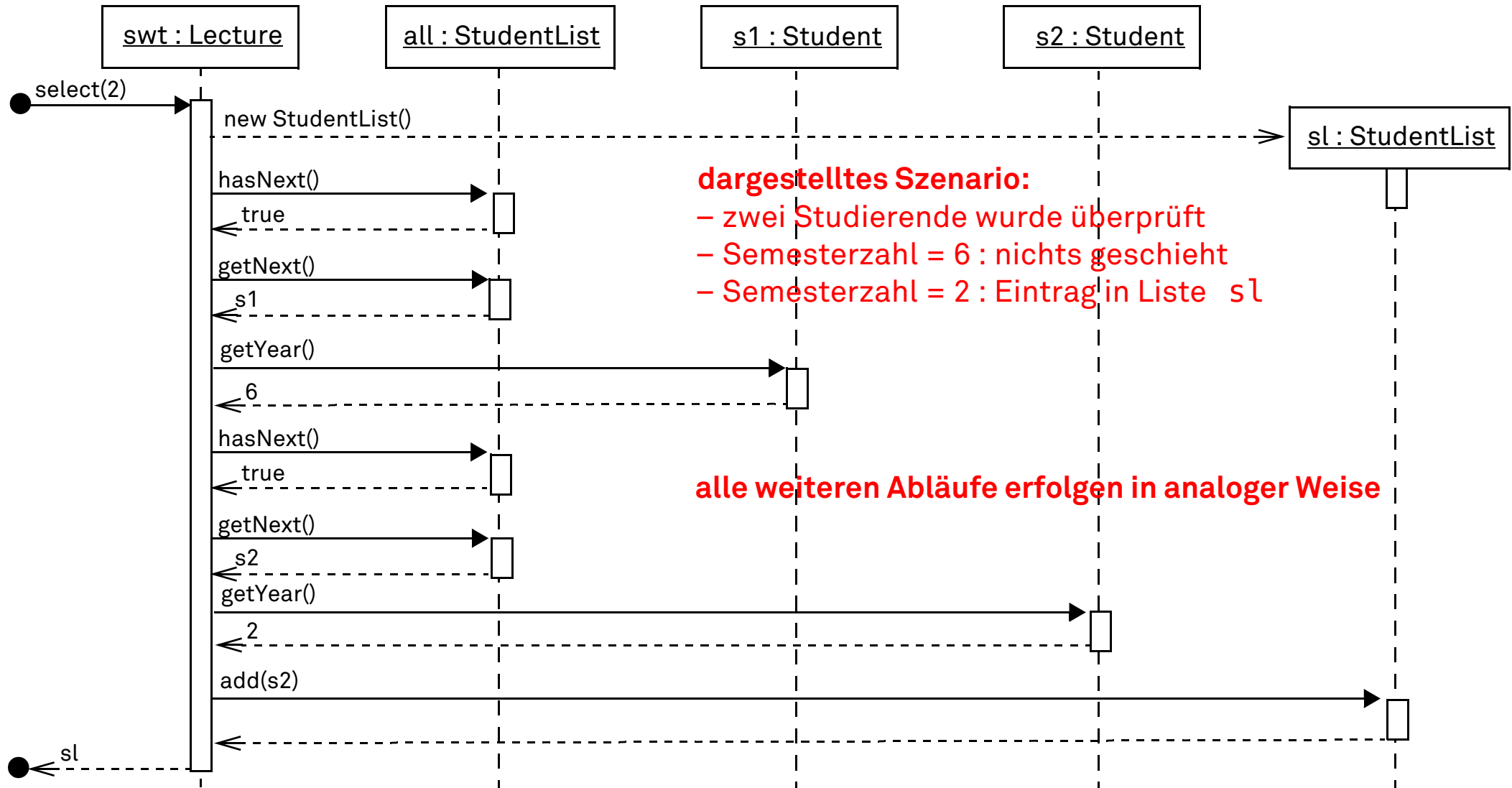
Beispiel – Sequenzdiagramm

(Fortsetzung)



Beispiel – Sequenzdiagramm

(Fortsetzung)



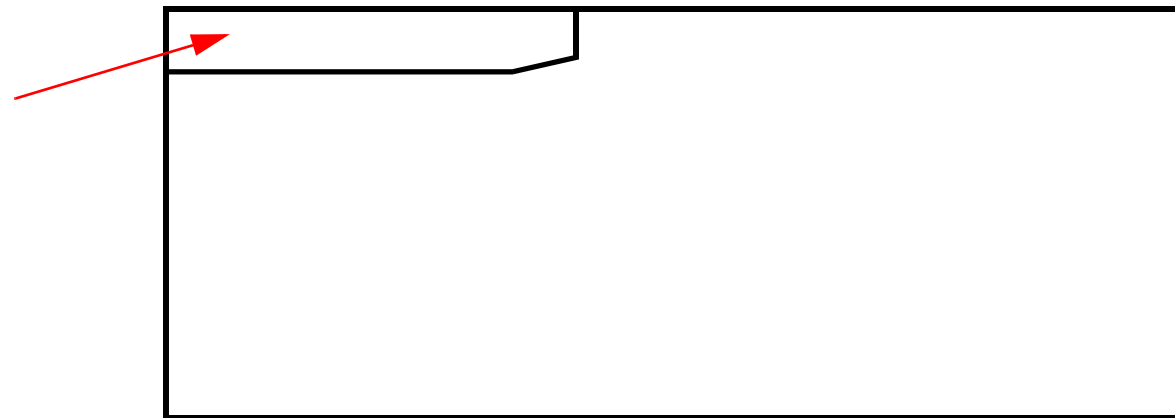
Interaktionsfragment

Das Sequenzdiagramm für das Beispiel
zeigt Ablauf mit zwei Studierenden

Möglich sind auch verallgemeinerte Darstellungen mit

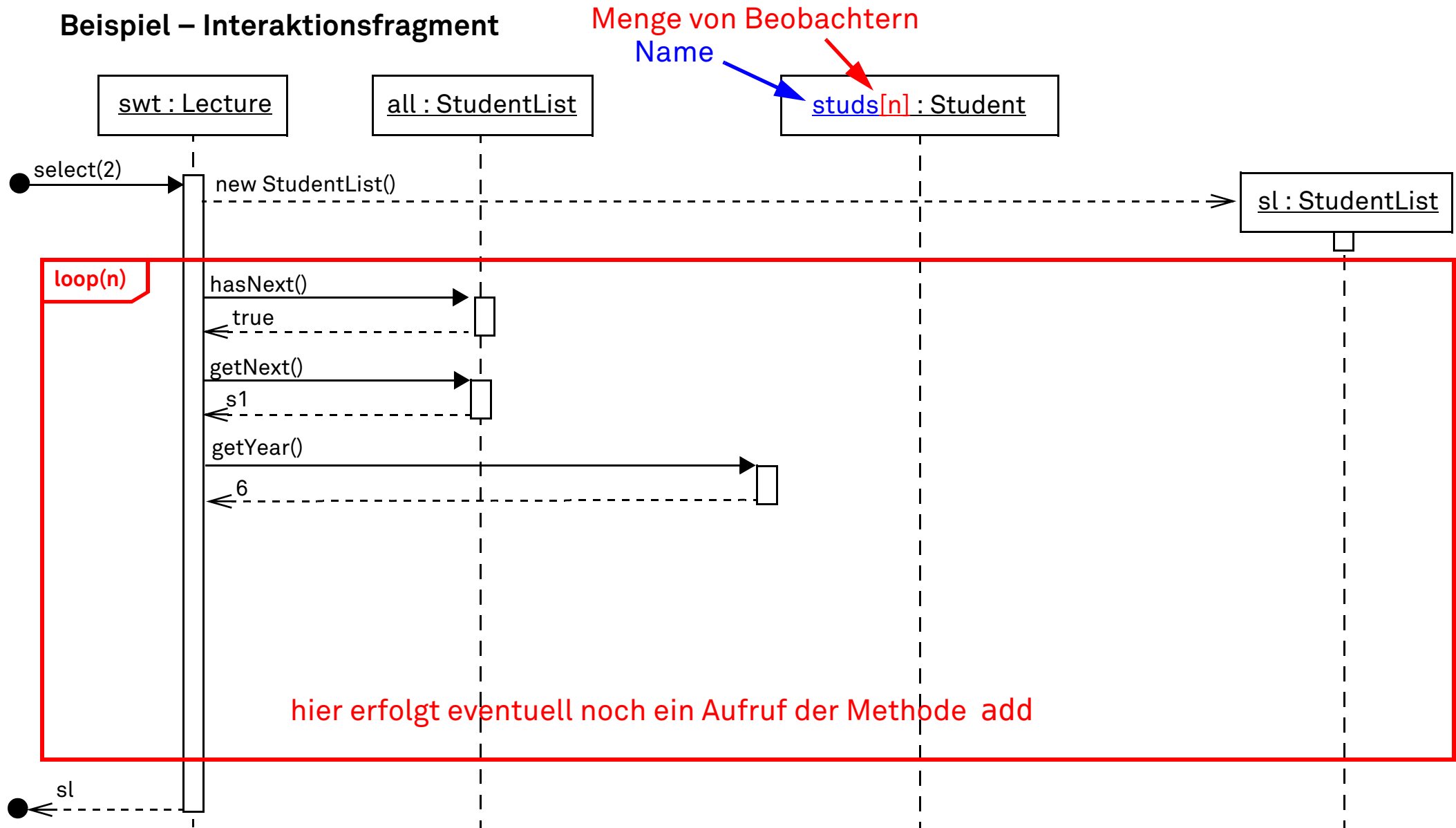
- ❑ Mengen von Objekten und
- ❑ Interaktionsfragmenten: Ablaufsequenzen, die abhängig von Bedingungen (wiederholt) ausgeführt werden können.

Eintrag für einen
Interaktionsoperator

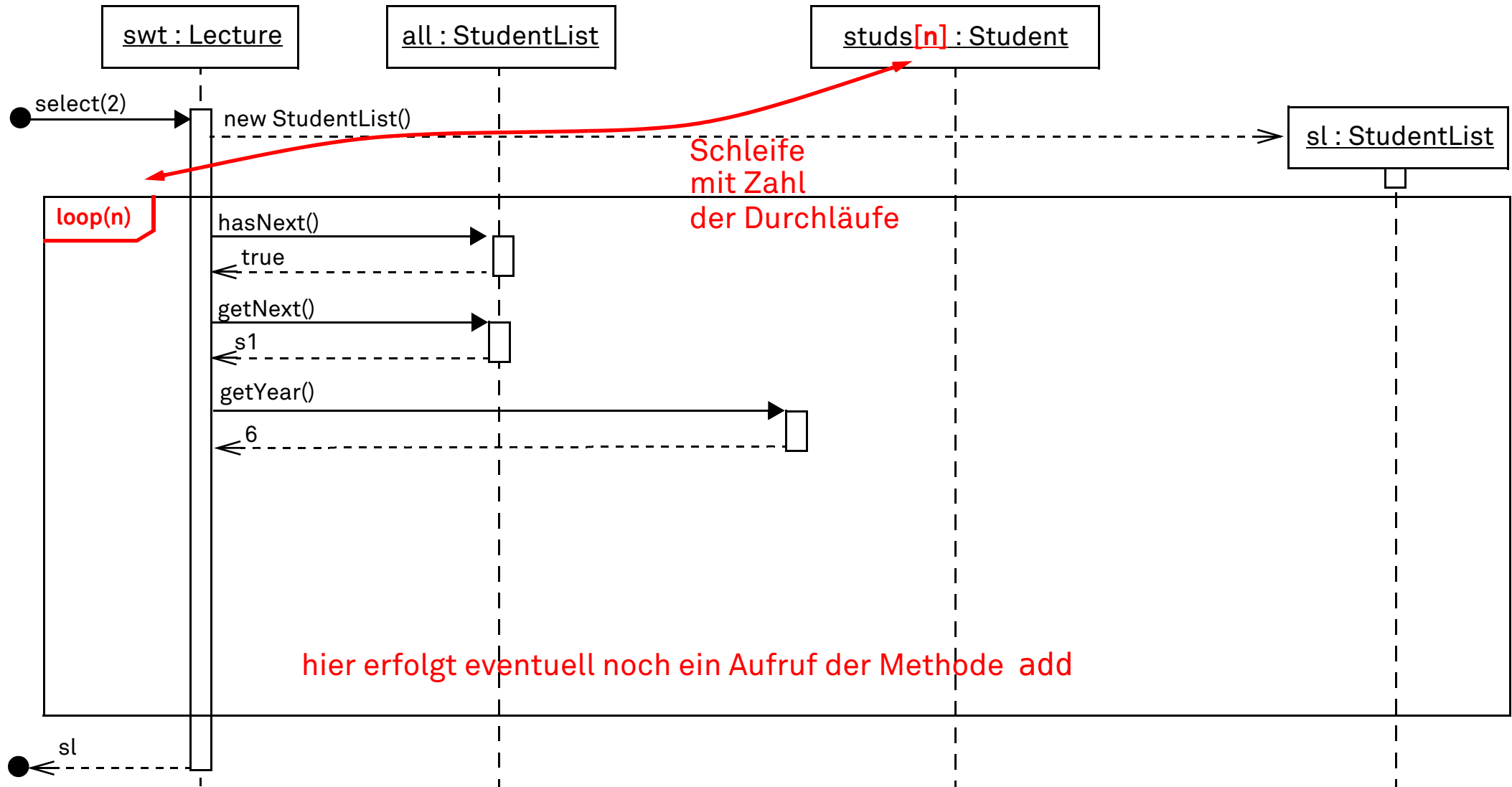


Interaktions-Fragment

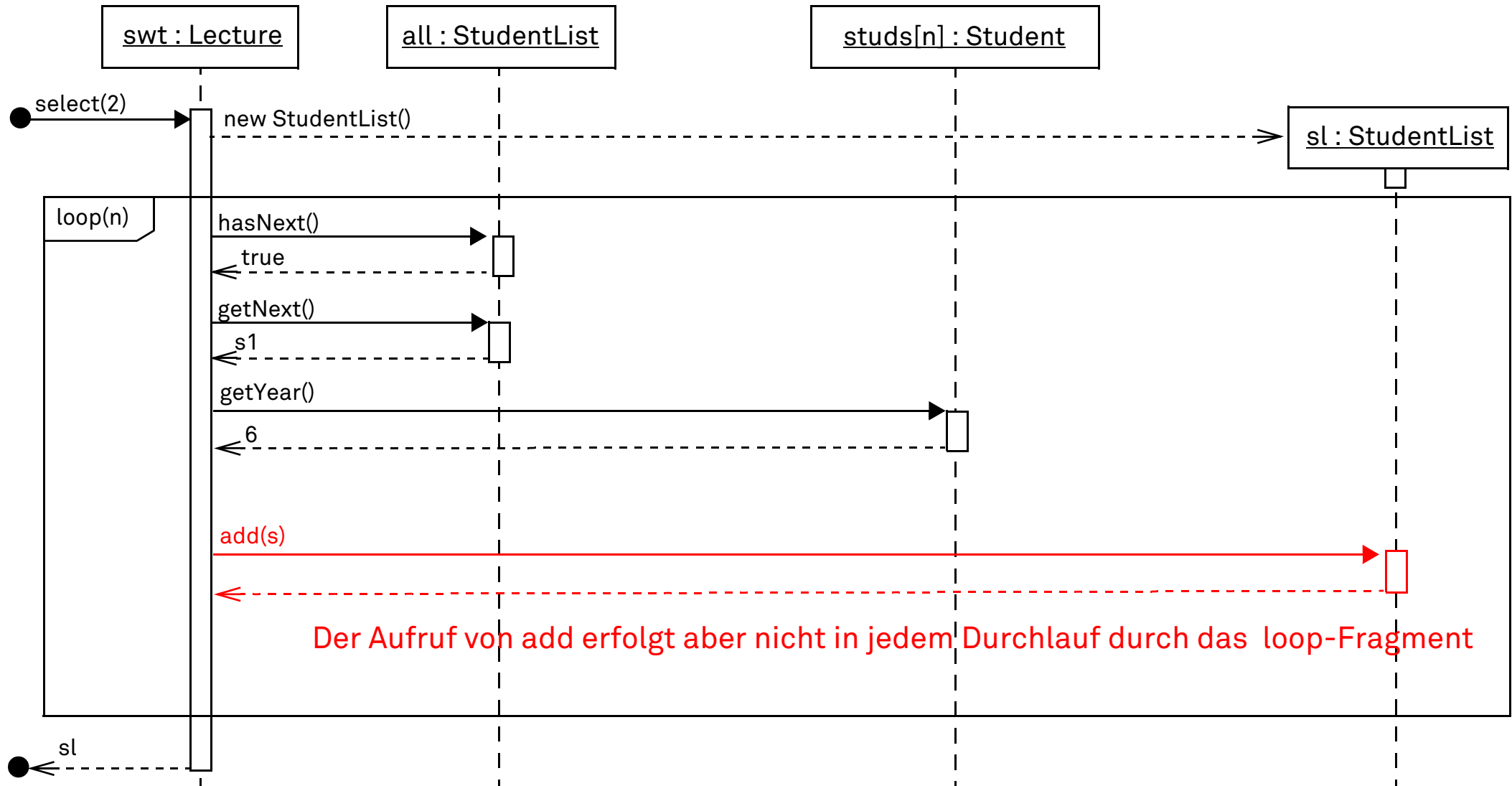
Beispiel – Interaktionsfragment



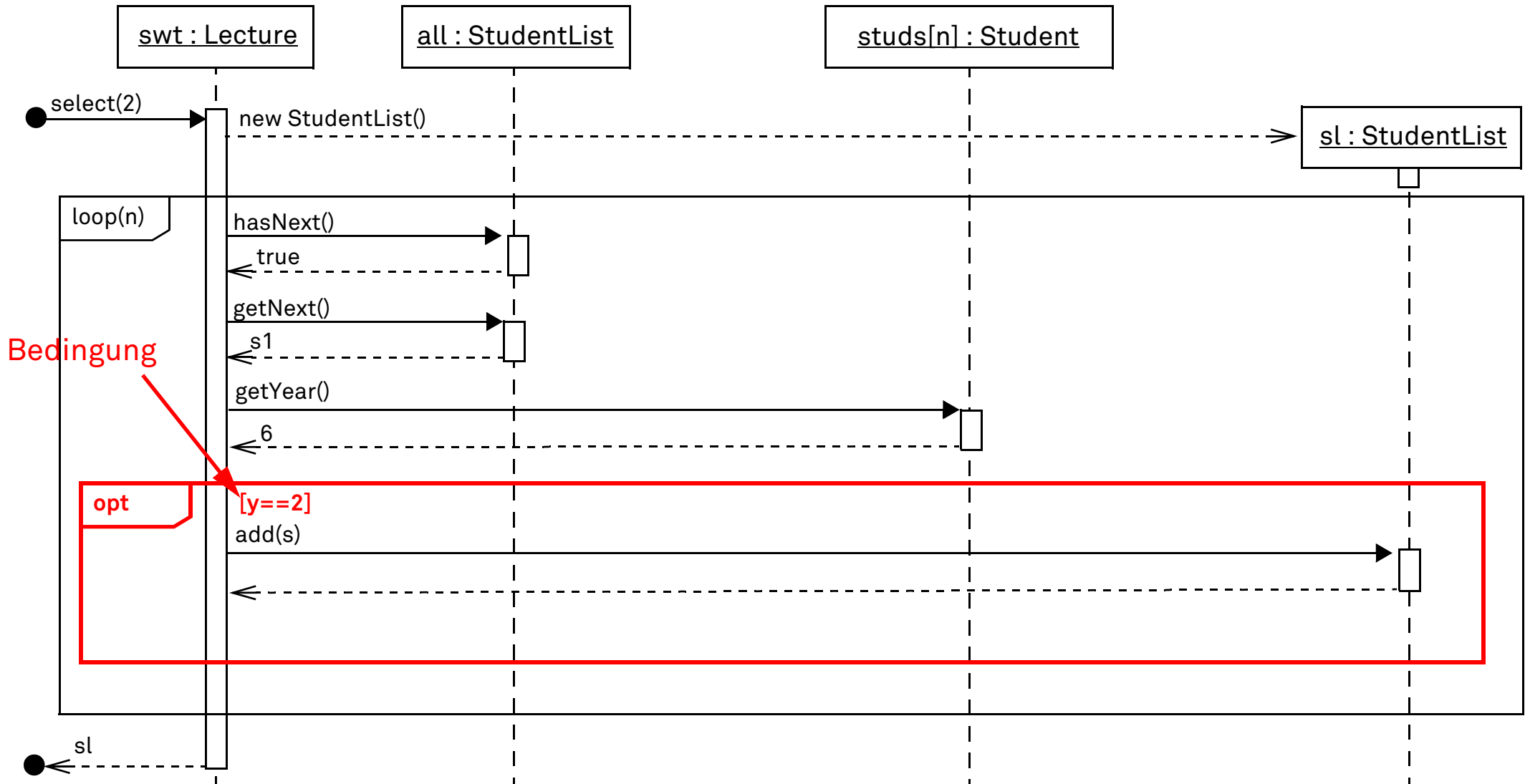
Beispiel – Interaktionsfragment



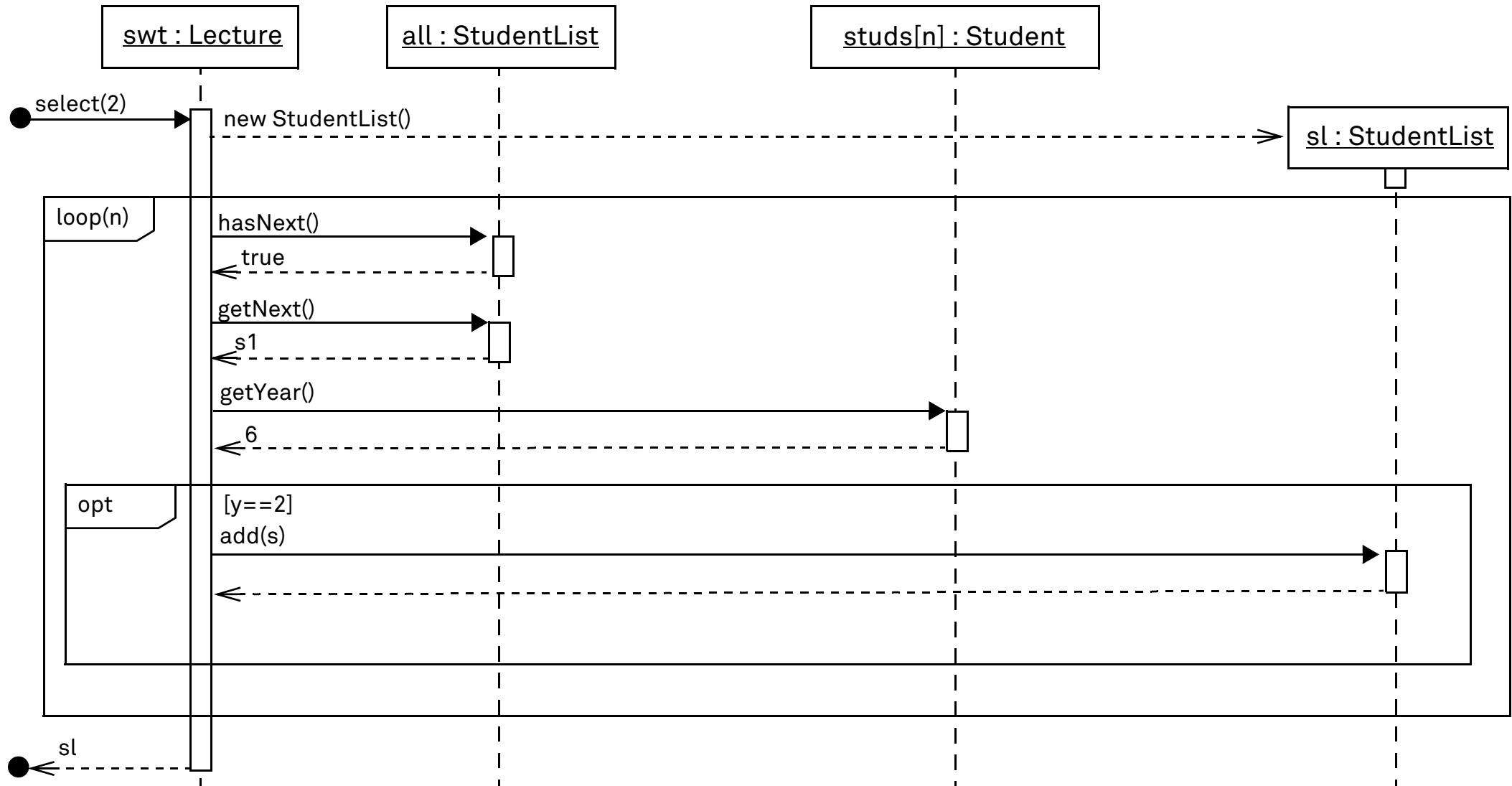
Beispiel – Interaktionsfragment



Beispiel – Interaktionsfragment



Beispiel – Interaktionsfragment

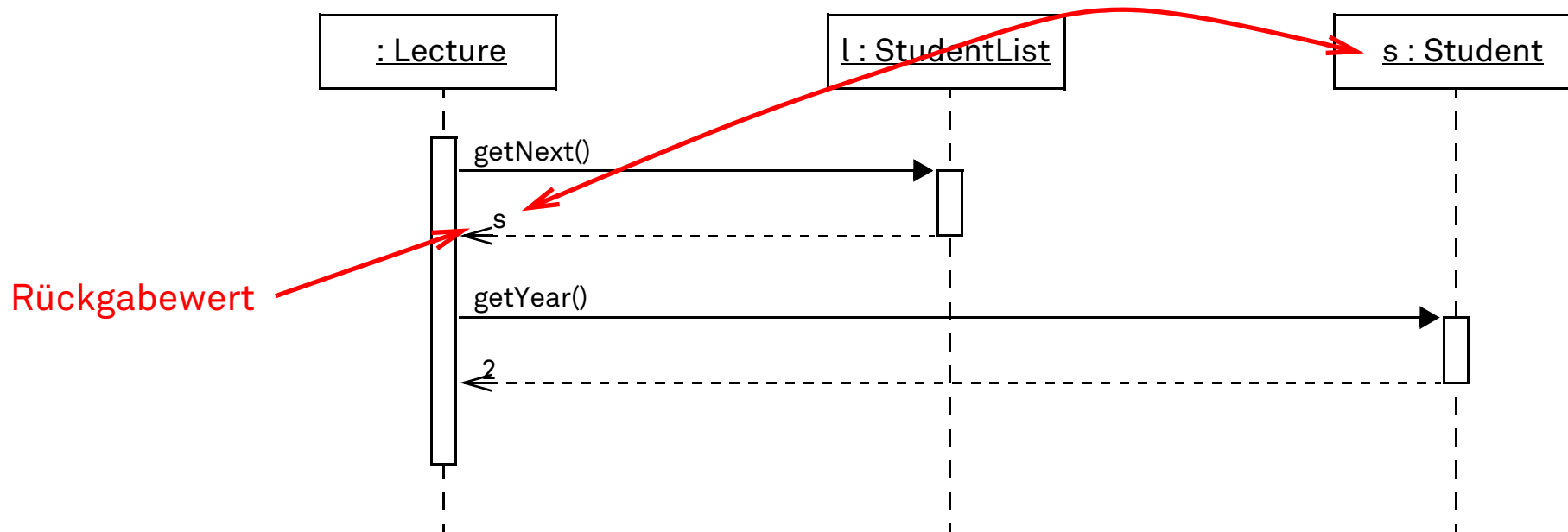


Sequenzdiagramm – Syntax

(Fortsetzung)

Aufruffolgen – von links nach rechts

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

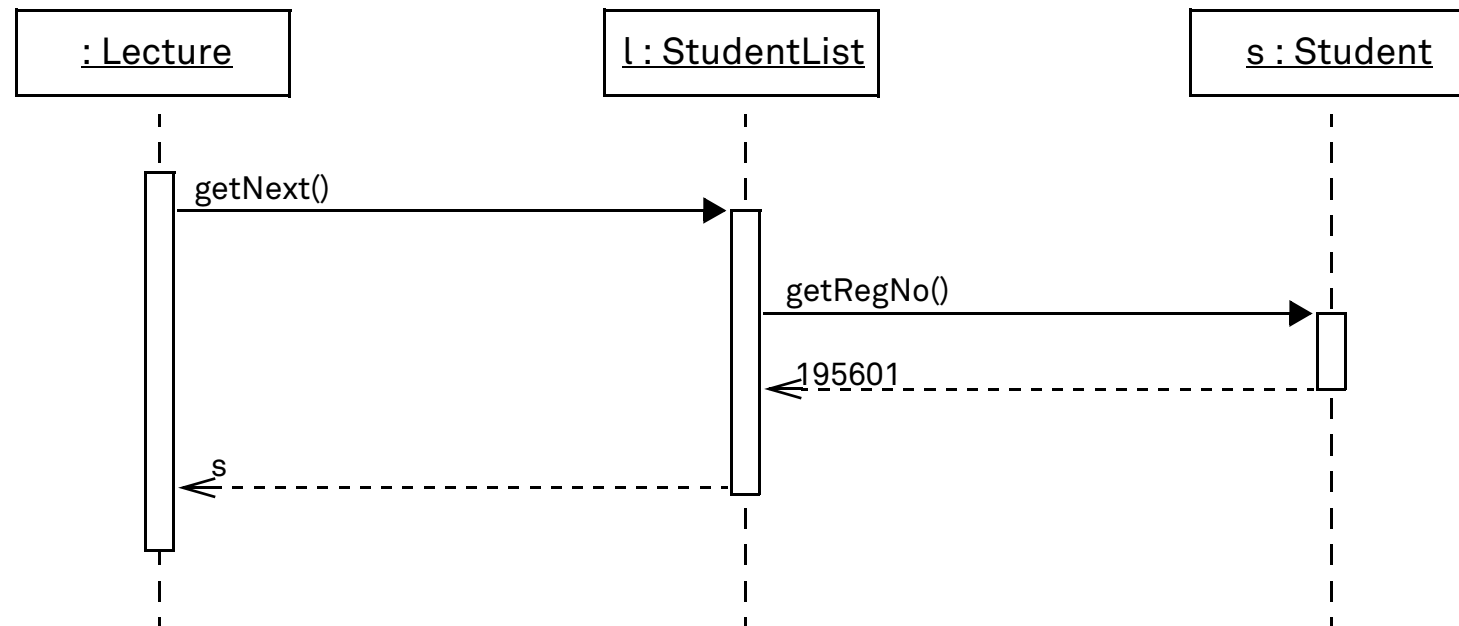


Sequenzdiagramm – Syntax

(Fortsetzung)

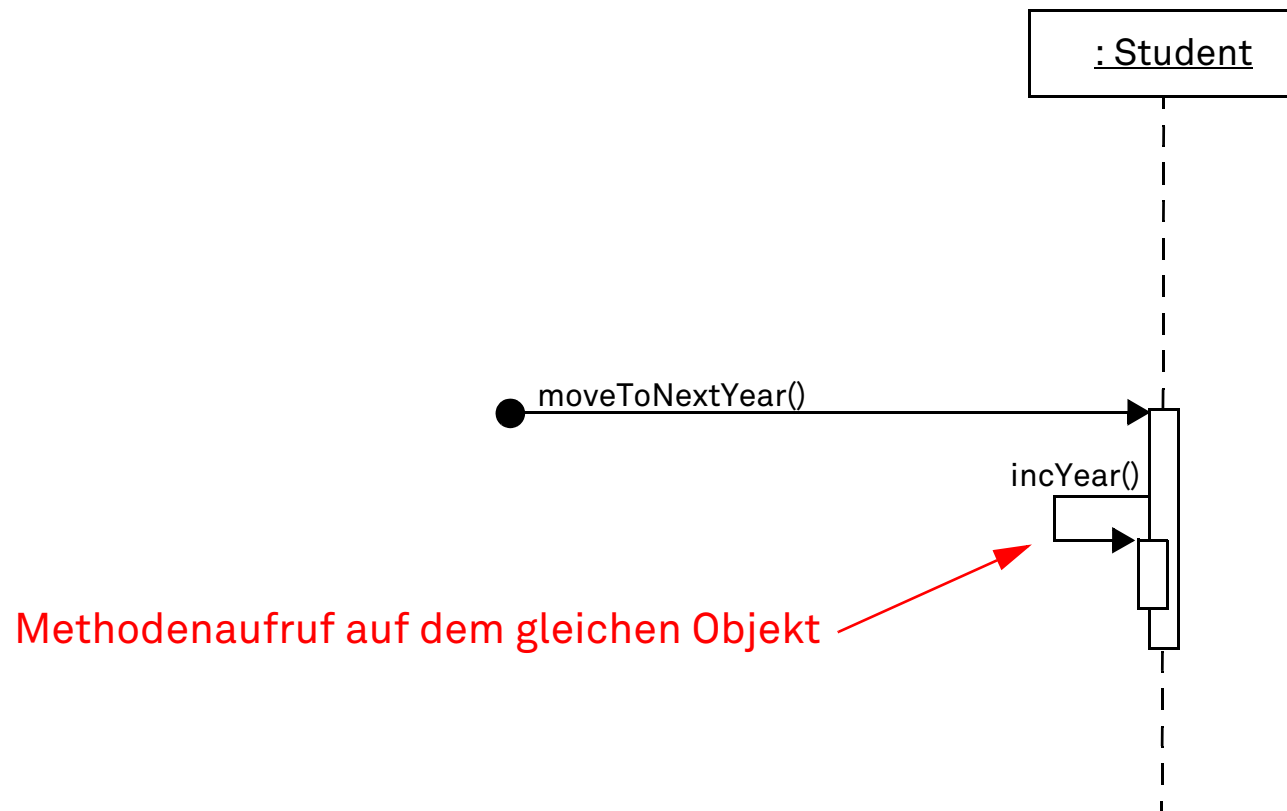
Aufrufschachtelungen

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




Beispiel – Sequenzdiagramm

(Fortsetzung)

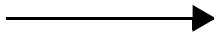



synchrone/asynchrone Nachrichten

bisher: synchrone Aufrufe

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

synchrone/asynchrone Nachrichten

(Fortsetzung)

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

synchrone/asynchrone Nachrichten

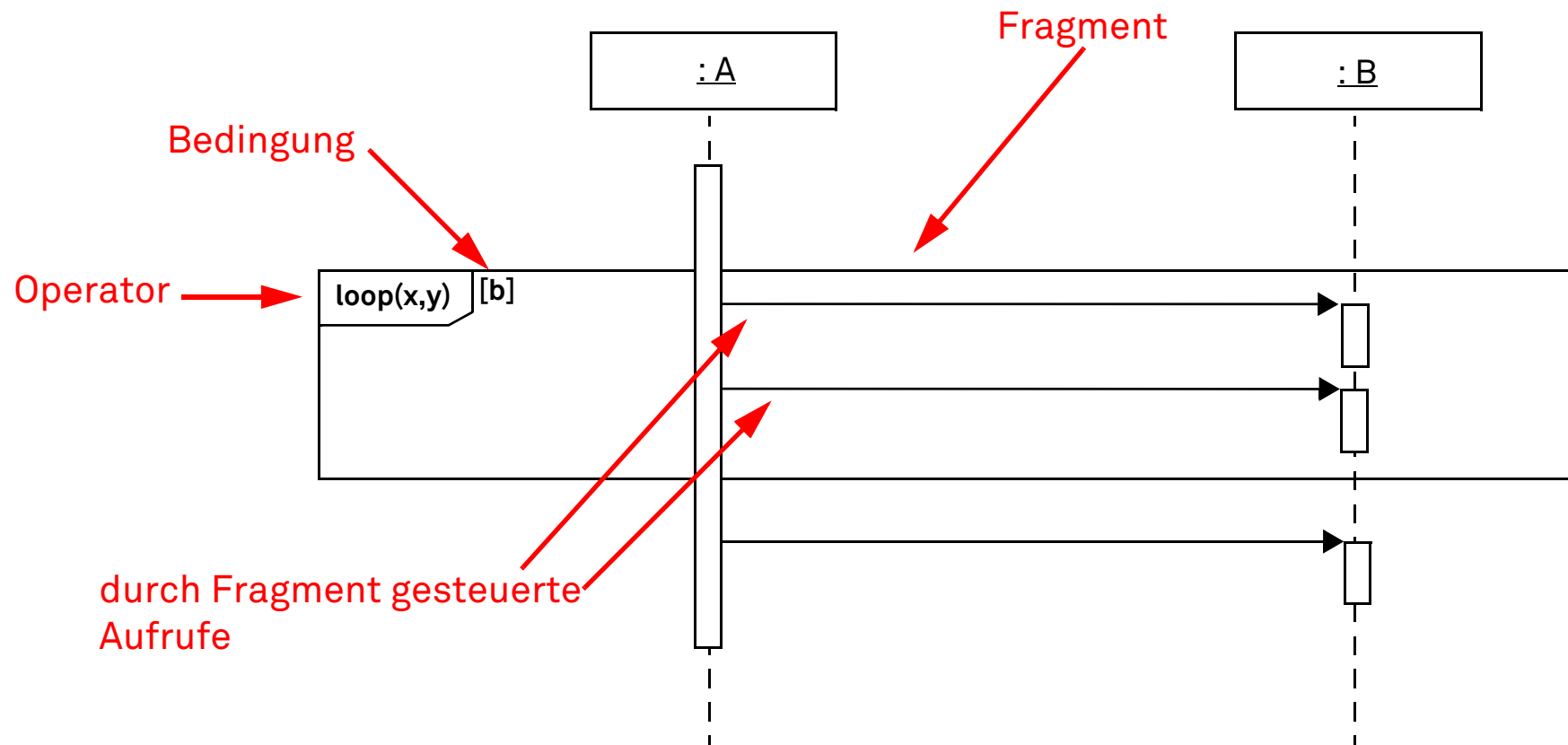
(Fortsetzung)

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

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

Interaktionsfragemente

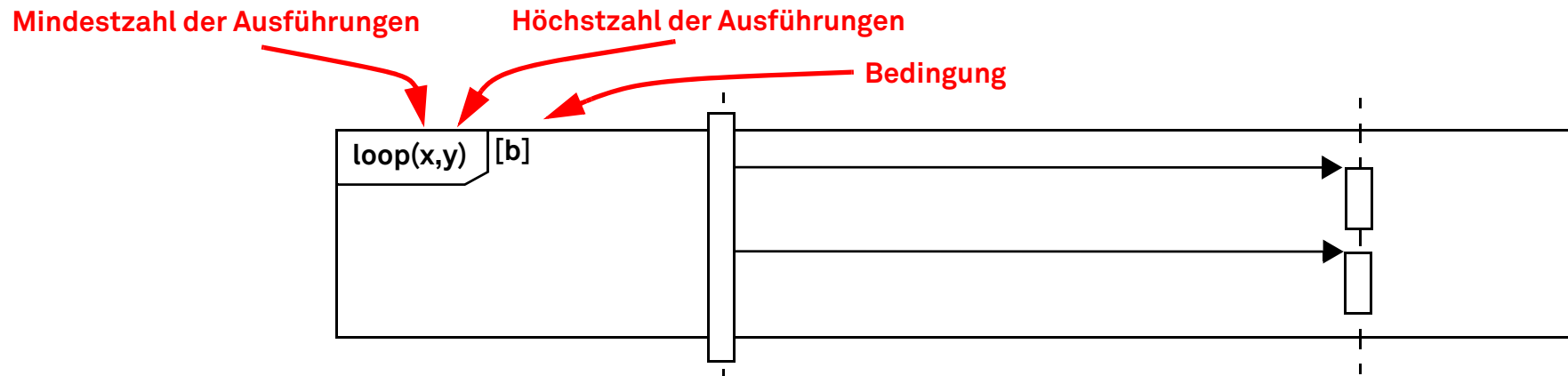
Ergänzungen zum loop-Operator



Interaktionsfragemente

(Fortsetzung)

Ergänzungen zum loop-Operator



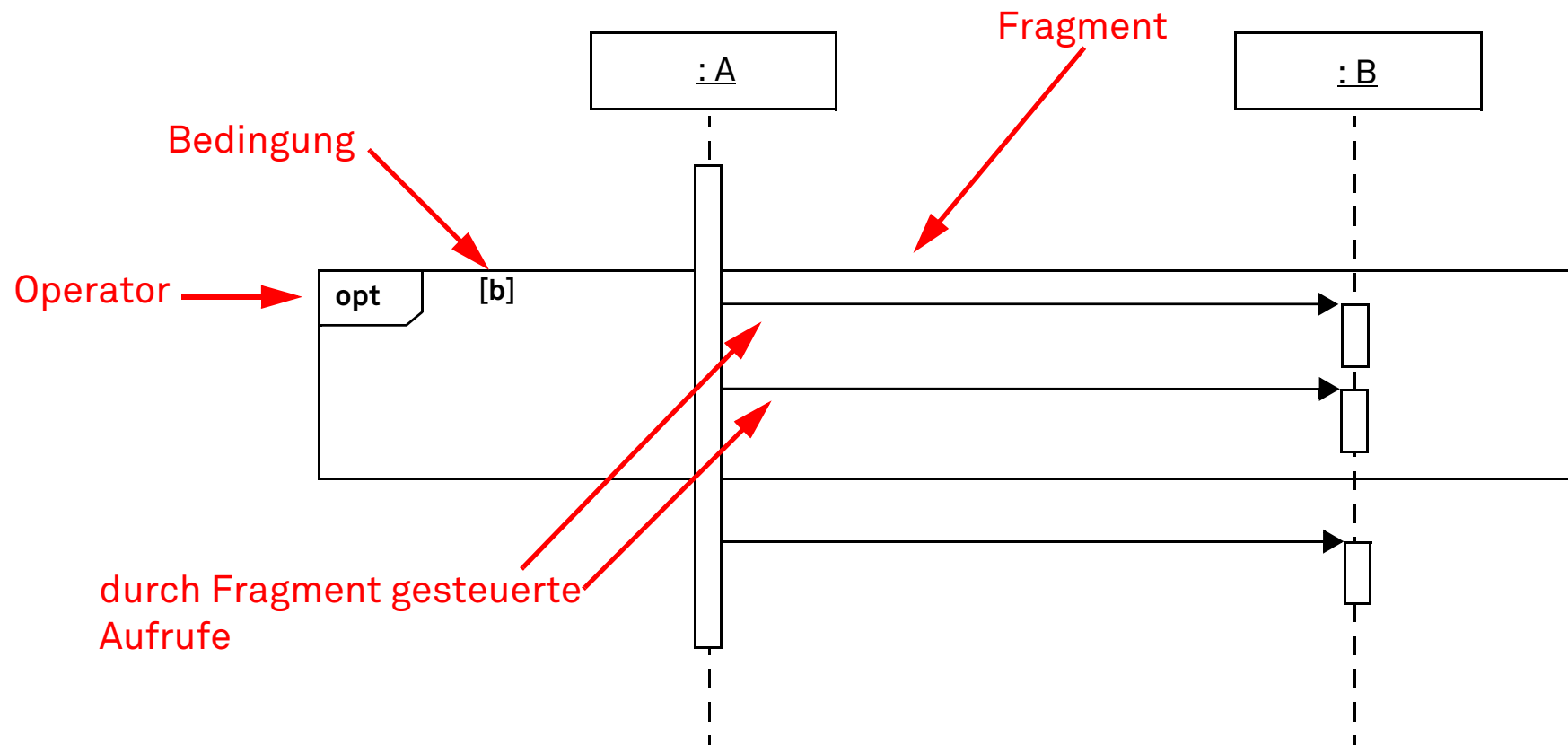
- ❑ `x` muss kleiner gleich `y` sein.
- ❑ Das **loop**-Fragment wird auf jeden Fall `x`-mal ausgeführt (– unabhängig von `b`).
- ❑ Das **loop**-Fragment wird höchstens `y`-mal ausgeführt – aber nur solange `b` gilt.
- ❑ **loop(z) entspricht loop(z,z)**, das **loop**-Fragment wird **genau z**-mal ausgeführt.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

Interaktionsfragemente

(Fortsetzung)

opt-Operator

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



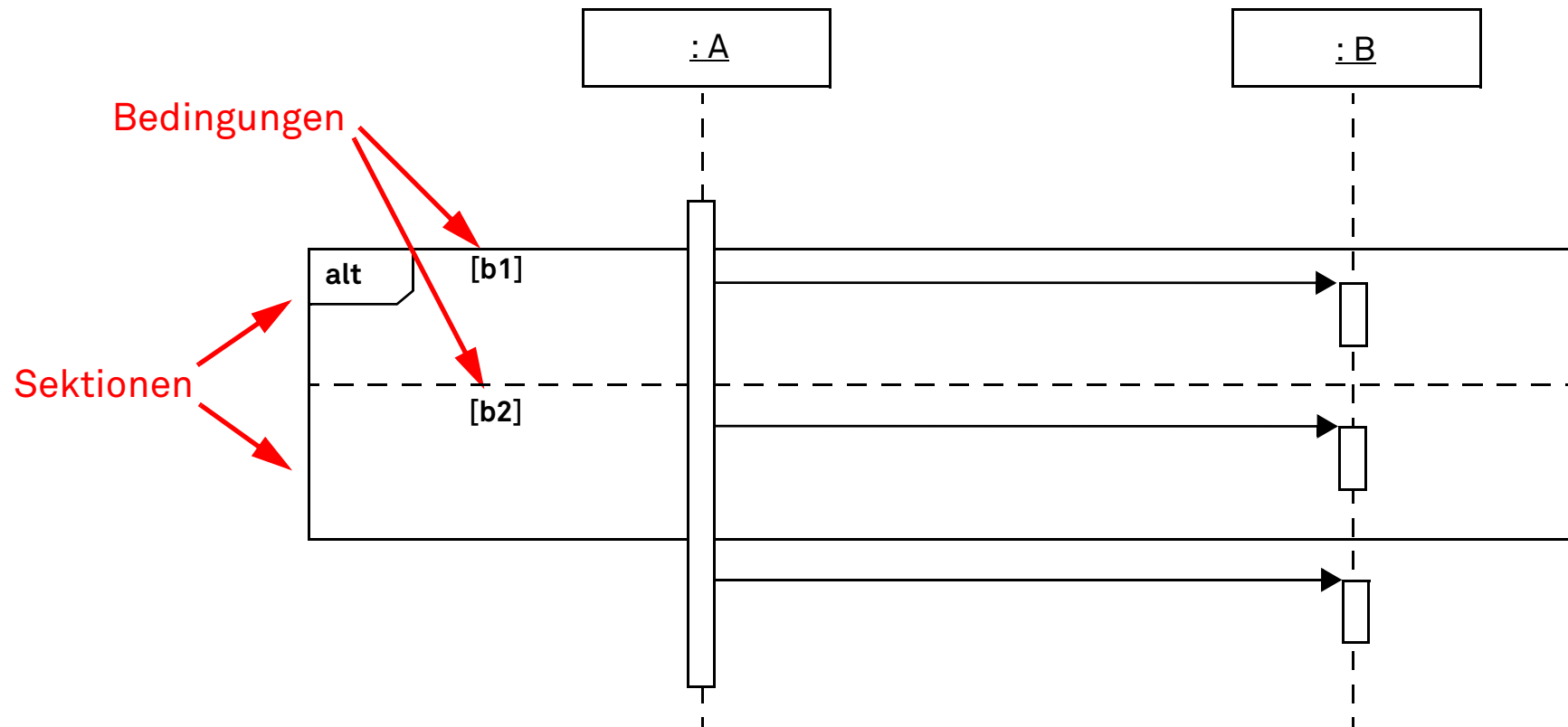
Interaktionsfragemente

(Fortsetzung)

alt-Operator

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

(alternative Abläufe)



Zusammenfassung

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

Folien zur Vorlesung **Softwaretechnik**

Teil 3: Entwurfsmuster

Entwurfsmuster

bilden den Einstieg in die Techniken zum Entwerfen von Softwaresystemen

Entwurfsmuster in der Softwareentwicklung
wurden erstmals vorgestellt in:

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

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

Anmerkung:

Das Buch ist aufgrund der vielen Vorwärtsreferenzen etwas schwer verständlich.

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

Idee der Entwurfsmuster

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

Vorteile beim Einsatz von Entwurfsmustern

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

Arten von Entwurfsmustern

Klassifizierung anhand der betrachteten Strukturierungseinheit:

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

klassenbezogene Muster

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

objektbezogene Muster

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

Klassifizierung anhand des durch das Entwurfsmuster gelösten Problems:

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

Strukturmuster

Verhaltensmuster

Erzeugungsmuster

Arten von Entwurfsmustern

(Fortsetzung)

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

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

Beschreibung von Entwurfsmustern

Angaben zur Zielsetzung:

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

Angaben zum Anwendungsbereich:

- ☐ Für welche Situationen passt das Muster?

Beschreibung von Struktur und Verhalten:

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

Beispiele für die programmtechnische Umsetzung:

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

Entwurfsmuster *Iterator*

Ein **Iterator**

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

Beispiele:

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

Dabei sollen folgende Anforderungen erfüllt werden:

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

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

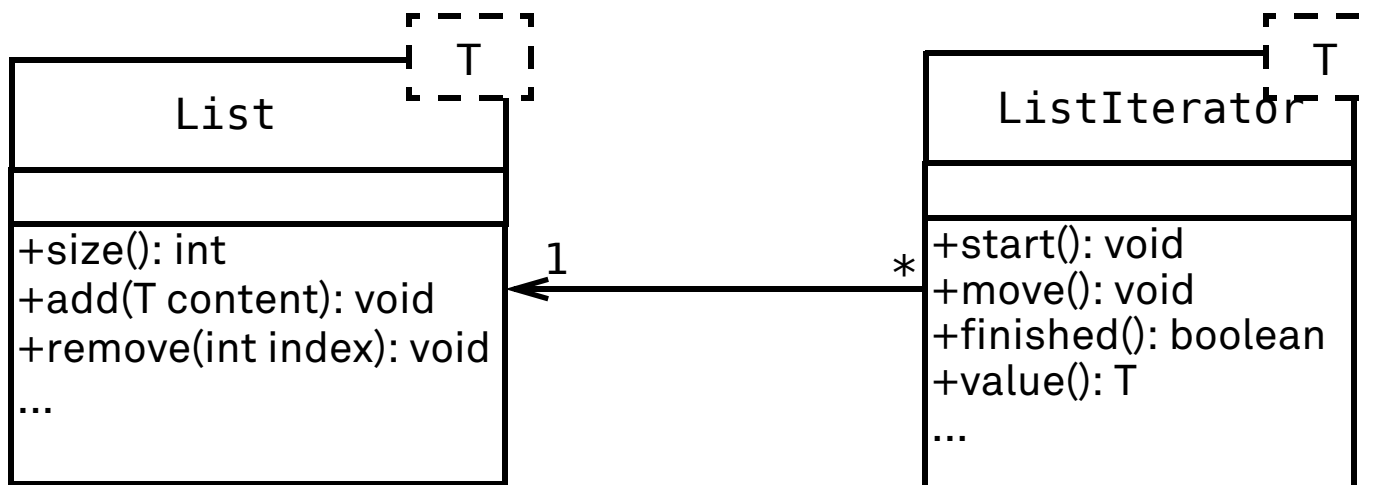
Entwurfsmuster *Iterator*

(Fortsetzung)

Idee:

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

Beispiel Liste: allgemeine Struktur



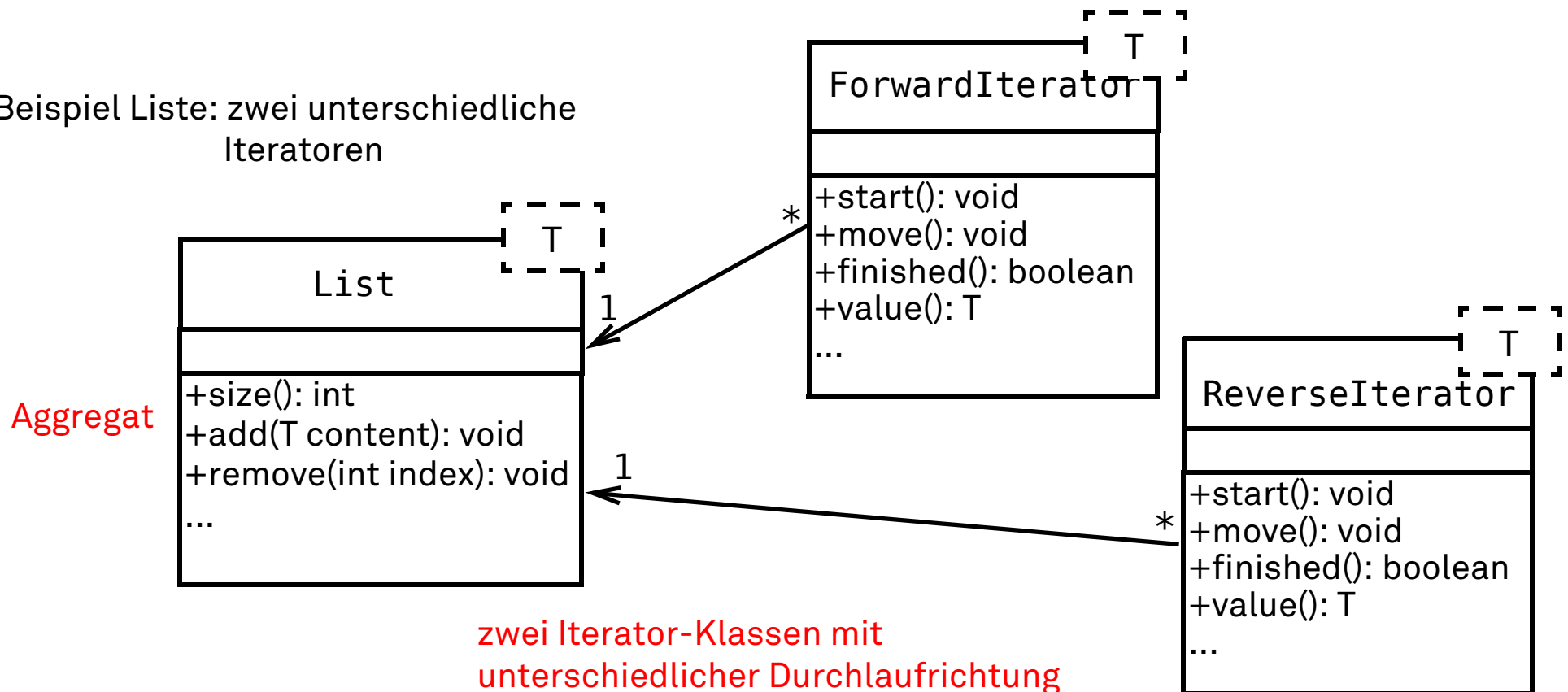
Entwurfsmuster *Iterator*

(Fortsetzung)

Idee:

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

Beispiel Liste: zwei unterschiedliche Iteratoren



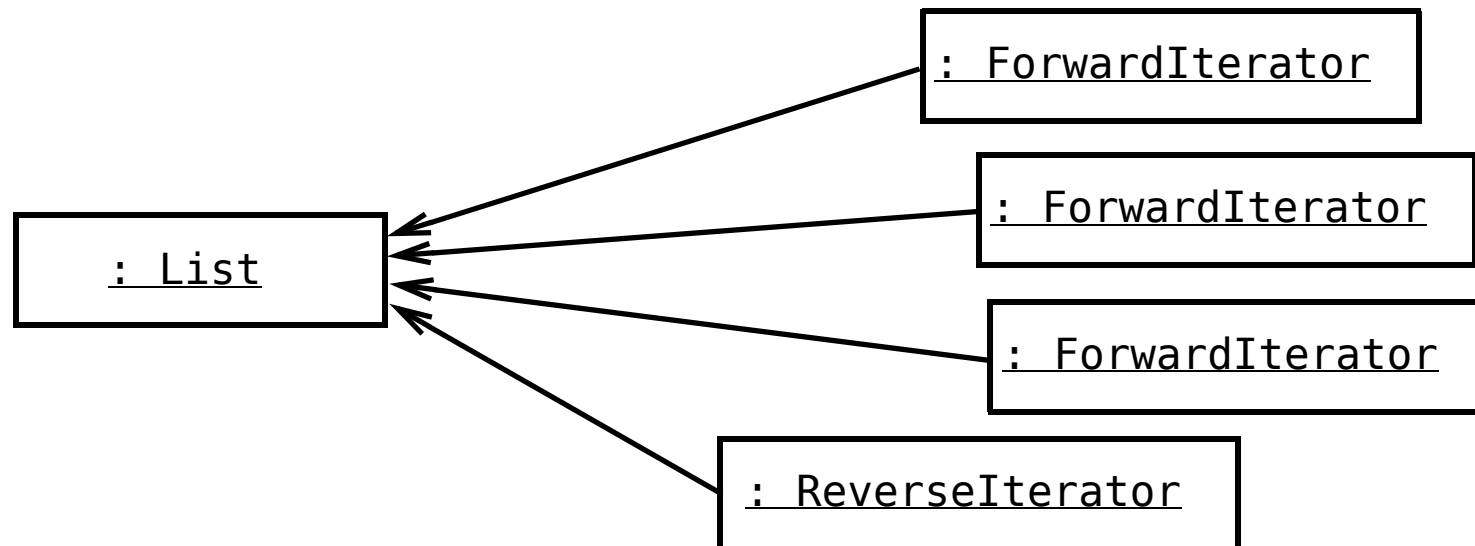
Entwurfsmuster *Iterator*

(Fortsetzung)

Idee:

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

Beispiel Liste: Objektdiagramm



mehrere Iterator-Objekte laufen über die gleiche Liste

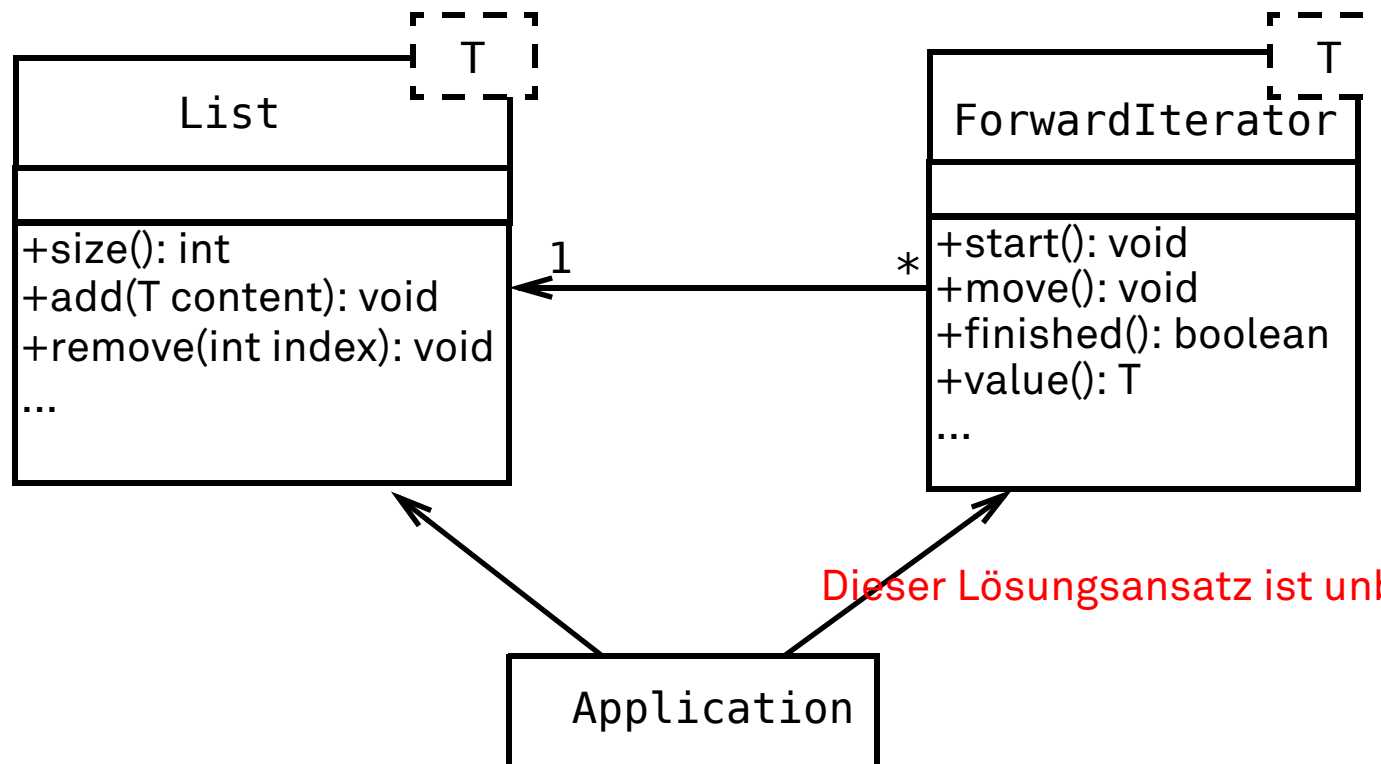
Entwurfsmuster *Iterator*

(Fortsetzung)

Die Anwendung muss Aggregat und Iterator miteinander verbinden.

aber:

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



Dieser Lösungsansatz ist unbrauchbar!

Entwurfsmuster *Iterator*

(Fortsetzung)

Die Anwendung muss Aggregat und Iterator miteinander verbinden.

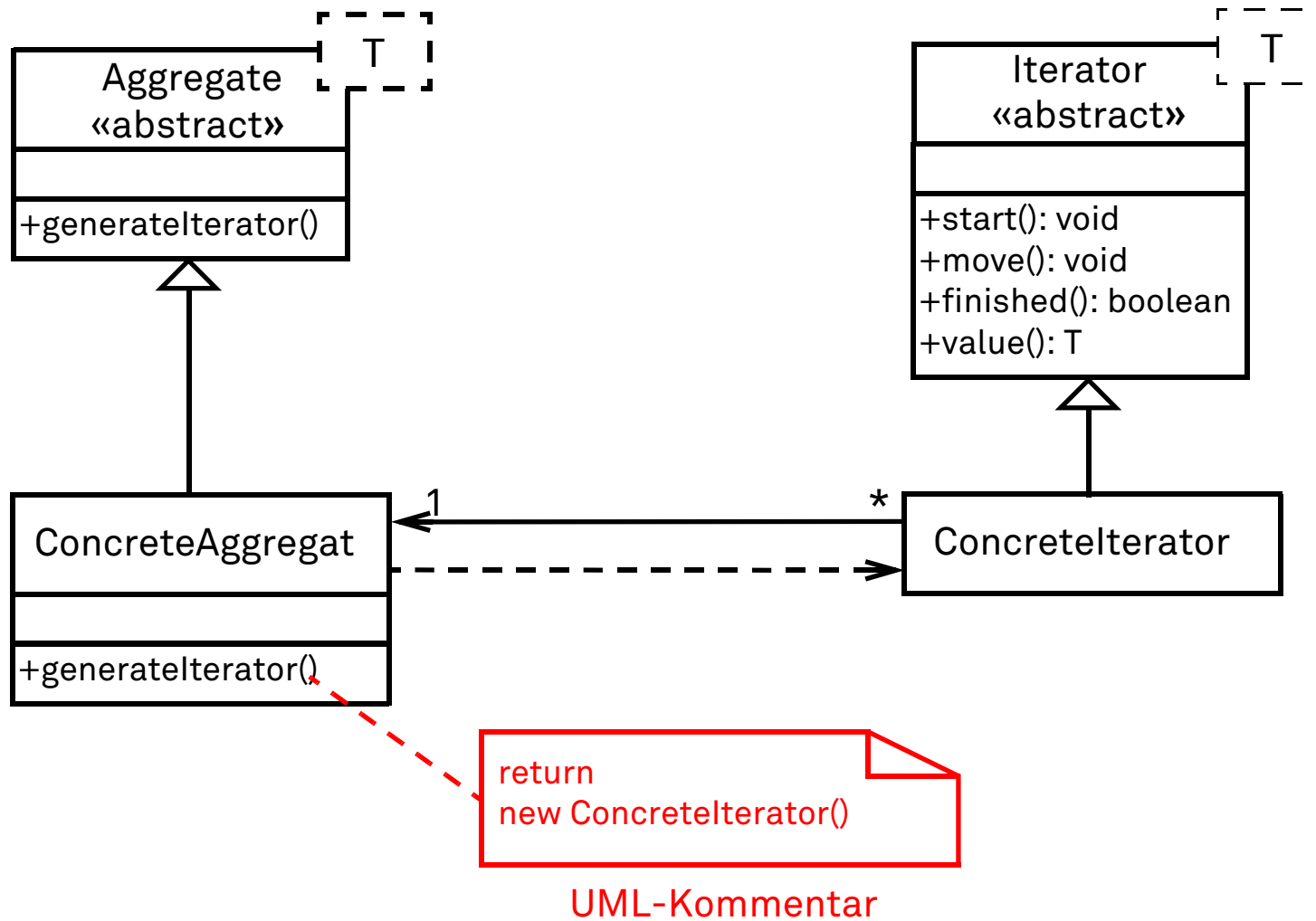
aber:

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

Daher folgende Verbesserungen:

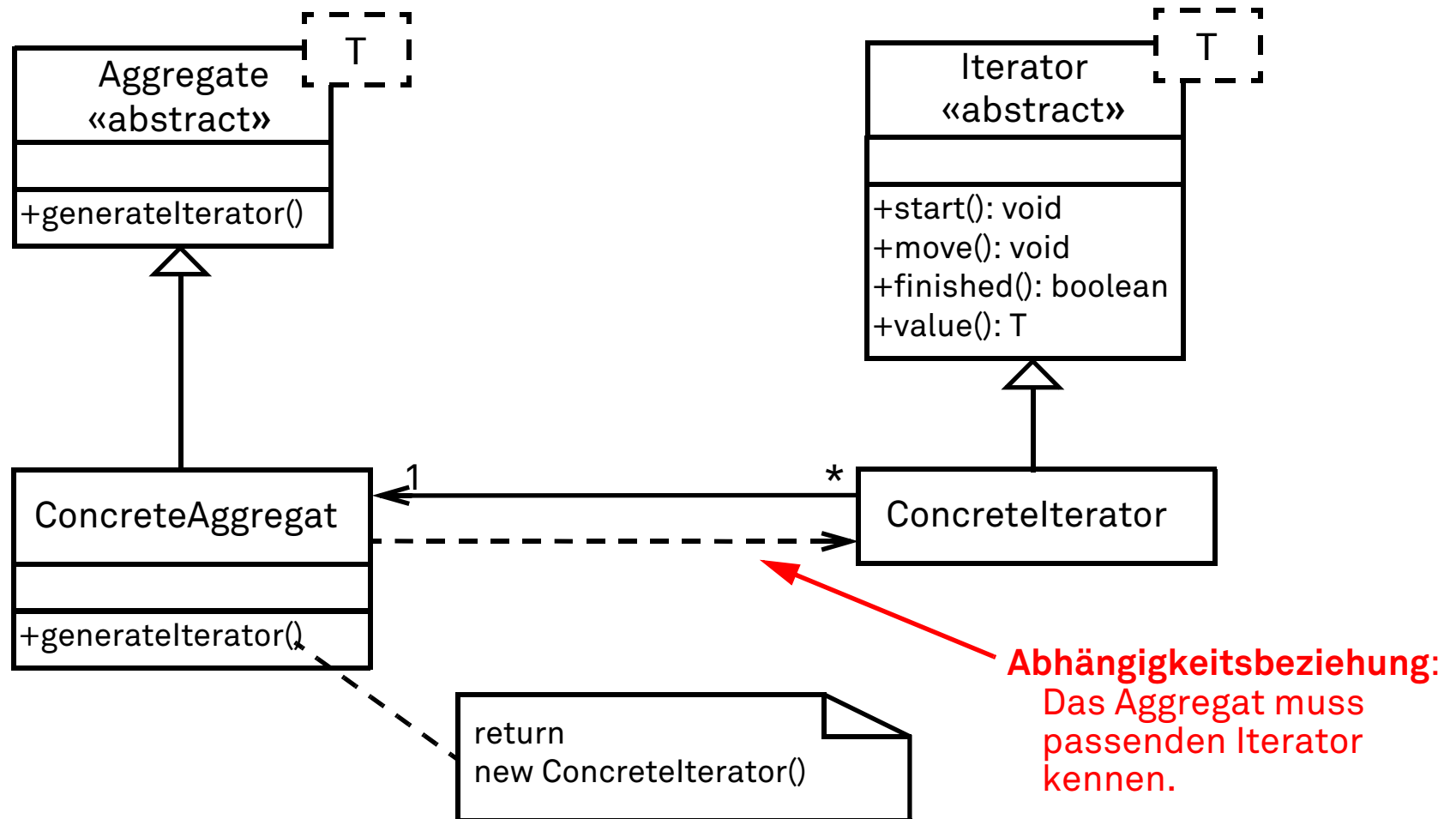
- ❑ Das Aggregat implementiert eine allgemeine Schnittstelle, die das Aggregat als *iterierbar* kennzeichnet.
- ❑ Der Iterator implementiert eine einheitliche Schnittstelle für Iteratoren, so dass alle Iteratoren gleich genutzt werden können.
- ❑ Jedes iterierbare Aggregat besitzt einen *eigenen* Iterator,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

Klassendiagramm *Iterator* (allgemeine Darstellung)



Klassendiagramm *Iterator* (allgemeine Darstellung)

(Fortsetzung)



Eintwurfsmuster *Iterator* – Bewertung

Vorteile:

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

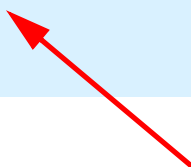
Nachteile:

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

Entwurfsmuster *Iterator*: Umsetzung in Java

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

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



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

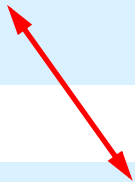
Entwurfsmuster *Iterator*: Umsetzung in Java

(Fortsetzung)

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

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

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



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

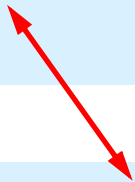
Das Weitersetzen und das Zurückgeben eines Inhalts fallen zusammen in der Methode `next()`.

Entwurfsmuster *Iterator*: Umsetzung in Java

(Fortsetzung)

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

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



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

optionale Methode

Entwurfsmuster *Iterator*: Umsetzung in Java

(Fortsetzung)

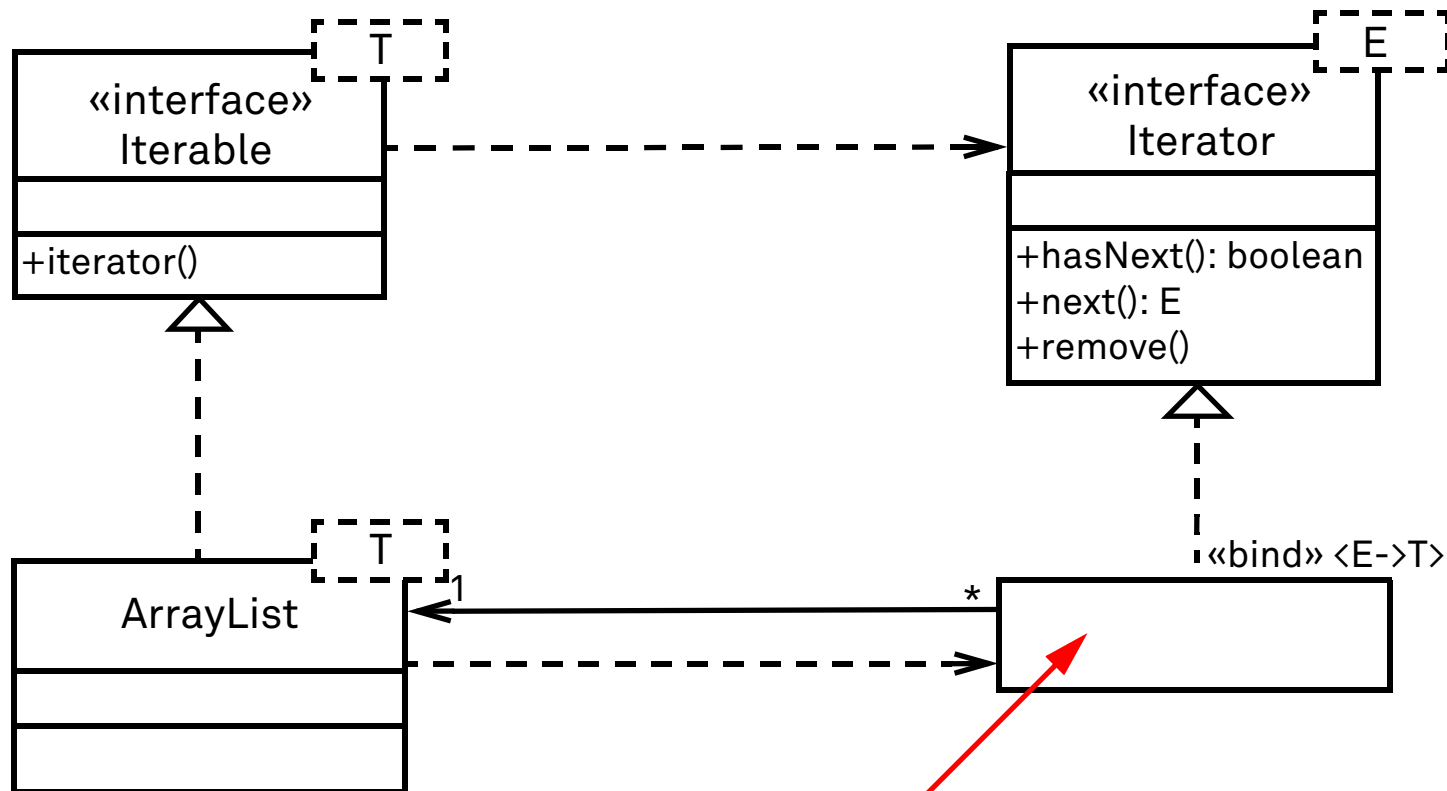
- ❑ Das **interface** `Iterable<T>`
wird u.a. von folgenden Klassen implementiert:
`ArrayList<E>`, `LinkedList<E>`, `PriorityQueue<E>`,
`Stack<E>`, `HashSet<E>`, `TreeSet<E>`
- ❑ Jede dieser Klassen stellt über seine `iterator()`-Methode
ein **passendes** `Iterator<E>`-Objekt bereit.
- ❑ Einige Klassen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

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

```
public ListIterator<E> listIterator(int index)  
    (index ist die Startposition für den Durchlauf des Iterators)
```


Entwurfsmuster *Iterator*: Umsetzung in Java

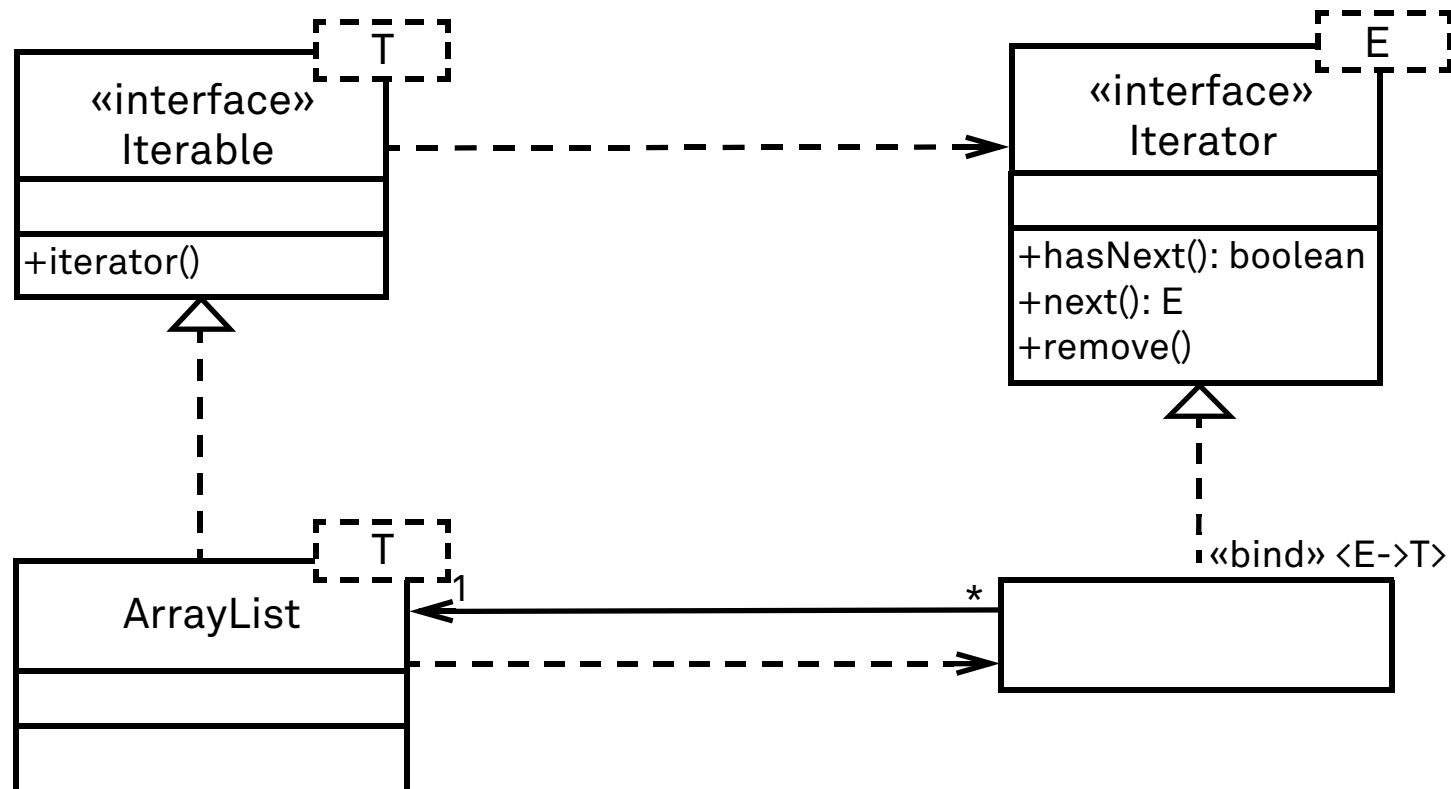
(Fortsetzung)



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

Entwurfsmuster *Iterator*: Umsetzung in Java

(Fortsetzung)



Iterator in Java: Beispiel für die Anwendung

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

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

← liefert Iterator-Objekt

← prüft, ob der Iterator
das letzte Element von
aggregate erreicht hat

← liefert das nächste Element
aus aggregate

Iterator in Java: Beispiel für die Anwendung

(Fortsetzung)

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

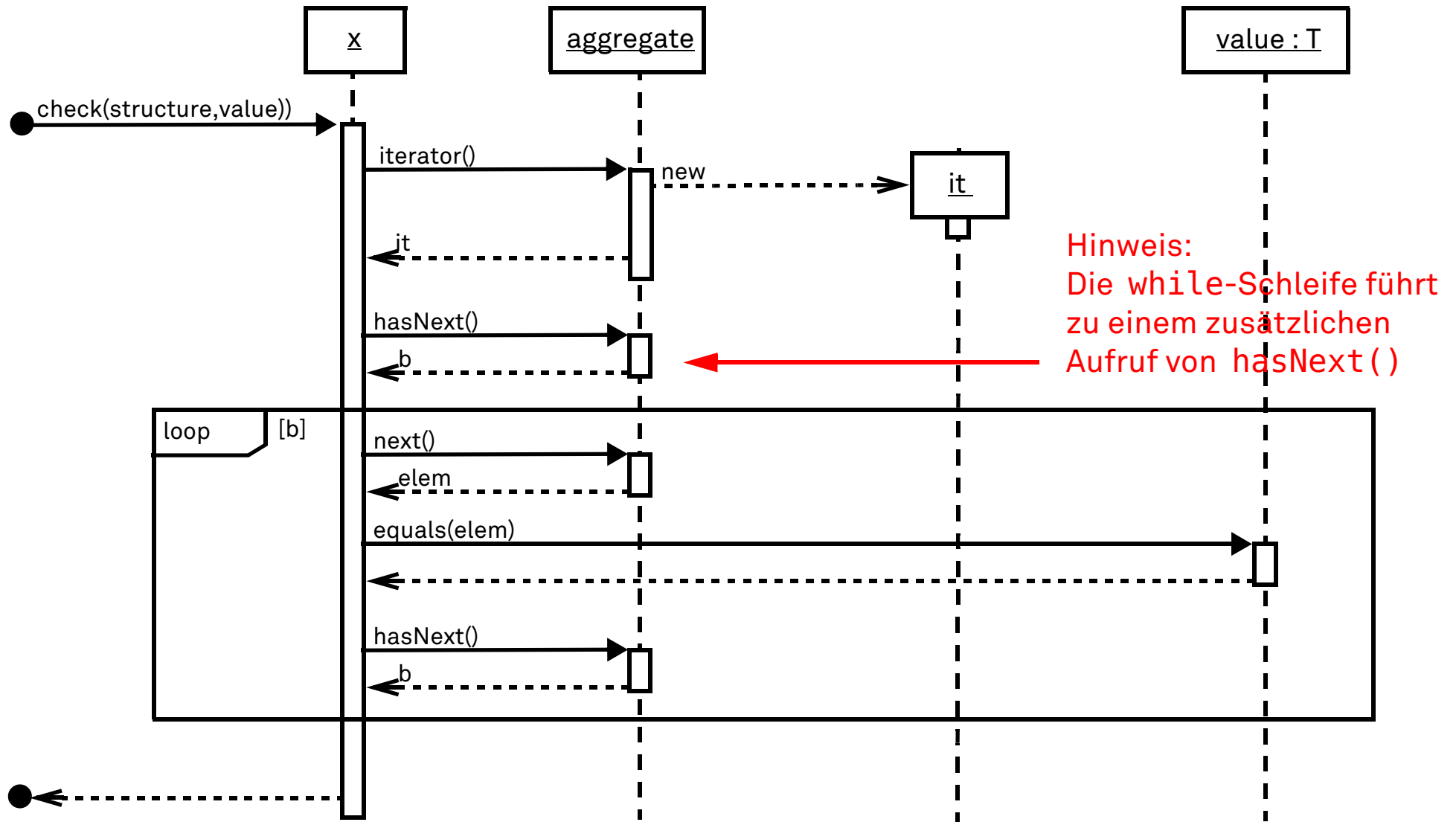
```
<T> boolean check(Iterable<T> aggregate, T value) {  
    Iterator<T> it = aggregate.iterator();  
    while (it.hasNext()) {  
        if (value.equals(it.next())) { return true; }  
    }  
    return false;  
}
```

oder noch einfacher:

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

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

Sequenzdiagramm zum Beispiel (Folie162 – while-Schleife)



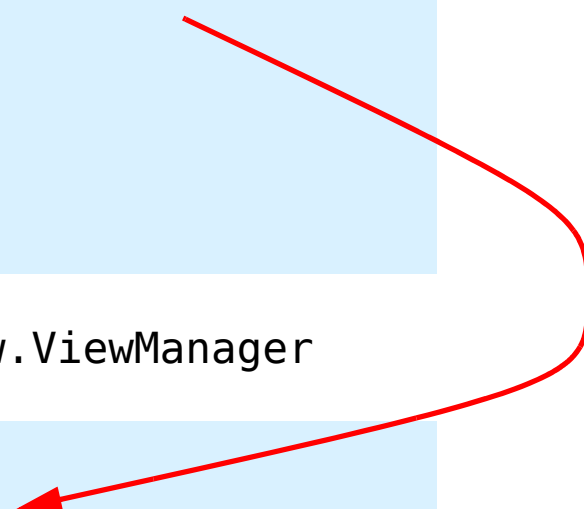
Beispiel für die Nutzung eines Iterators

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

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

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

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



Beispielimplementierung für einen Iterator

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

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

Beispielimplementierung für einen Iterator

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

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

Anmerkungen:

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

Iterator in Java: Beispiel für die Anwendung

(Fortsetzung)

Die Beispiele zeigen die Vorteile des Iterator-Musters:

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

Entwurfsmuster *Iterator*: Umsetzung in Java – Anmerkungen

Neben den bekannten drei Methoden `hasNext()`, `next()` und `remove()` wird im Interface `Iterator<E>` noch eine weitere Methode deklariert:

```
interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    default void remove() { ... }  
    default void forEachRemaining(Consumer<? super E> action) {  
        while (hasNext()) {  
            action.accept(next());  
        }  
    }  
}
```

Die Methode `forEachRemaining` wendet auf alle (verbliebenen) Inhalte des vom Iterator durchlaufenen Aggregats die gleiche `accept`-Methode an. Das soll die Nutzung des Iterators noch weiter vereinfachen.

Entwurfsmuster *Iterator*: Umsetzung in Java – Anmerkungen

(Fortsetzung)

```
interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    default void remove() { ... }  
    default void forEachRemaining(Consumer<? super E> action) {  
        while (hasNext()) {  
            action.accept(next());  
        }  
    }  
}
```

```
interface Consumer<T> {  
    void accept(T t);  
}
```

funktionales Interface, das eine verkürzte
Deklaration der Methode `accept` als
Lambda-Ausdruck ermöglicht.

Beispiel (mit `ArrayList list`):

```
list.iterator().forEachRemaining( t->System.out.println(t) );
```

Entwurfsmuster *Iterator*: Umsetzung in Java – Anmerkungen

(Fortsetzung)

Vergleich

Durchlaufen eines Aggregats mit `hasNext()`–`next()`-Folgen:

- ❑ Die Kontrolle über den Fortschritt erfolgt über den Aufruf von `next()`, also außerhalb des Aggregats und des Iterator.
- ❑ Der Inhalt des Aggregats wird zur weiteren Bearbeitung immer nach außen gegeben durch den Aufruf von `next()`.

Durchlaufen einer iterierbaren Datenstruktur mit `forEachRemaining`:

- ❑ Alle Inhalte deAggregats werden unmittelbar nacheinander betrachtet.
- ❑ Der Durchlauf ist nach der Ausführung von `forEachRemaining` abgeschlossen, alle Inhalte sind betrachtet worden.
- ❑ Die Bearbeitungsvorschrift (`accept`) wird an den Iterator übergeben und in der Methode `forEachRemaining` ausgeführt, ohne dass Inhalte nach außen gegeben werden.

Die Methode `accept` des übergebenen Consumer-Objekts ist die

Verarbeitungsstrategie.

Entwurfsmuster *Strategie*

Eine **Strategie**
erlaubt

- ❑ das Festlegen des Verhaltens eines Objekts ohne Zugriff auf die Implementierung der Klasse des Objekts oder
- ❑ das Verändern des Verhaltens eines Objekts während der Ausführung.

Das bedeutet:

- ❑ Verschiedene Arten von Verhalten müssen bereitgestellt werden können.
- ❑ Das Verhalten muss nach der Deklaration der Klasse des ausführenden Objekts festgelegt werden können.

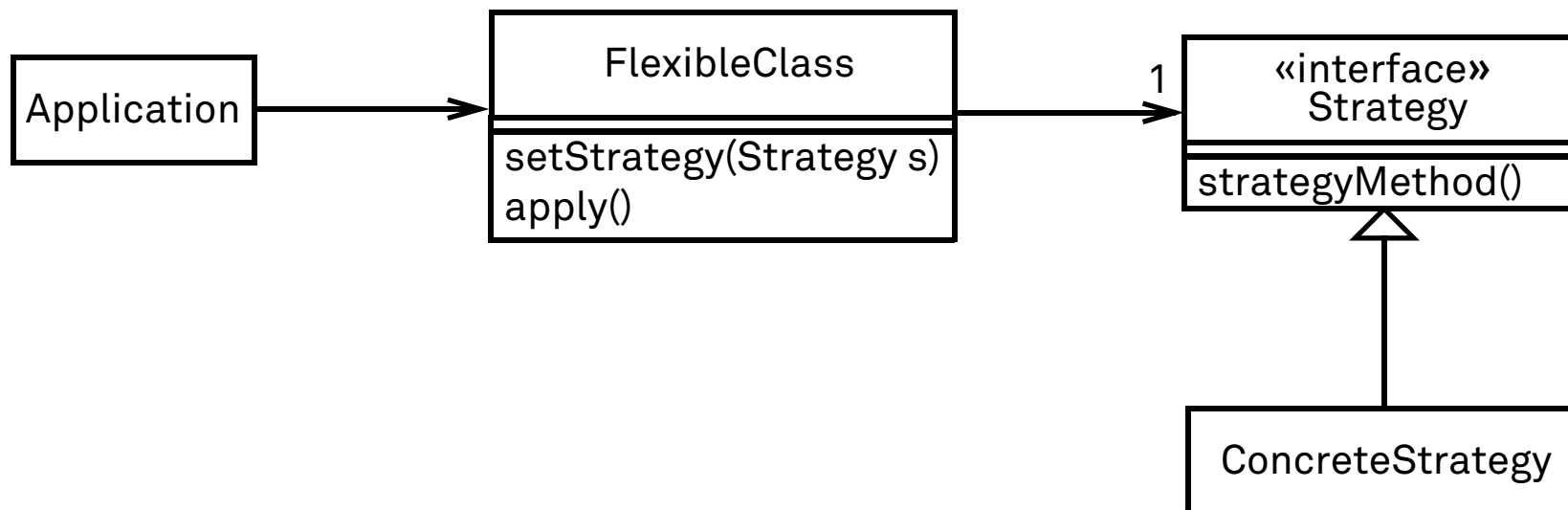
Beispiele:

- ❑ Anordnen von grafischen Elementen in einem Fenster
- ❑ Festlegen des Verhaltens von Monstern im *SWT-Starfighter*-Spiel

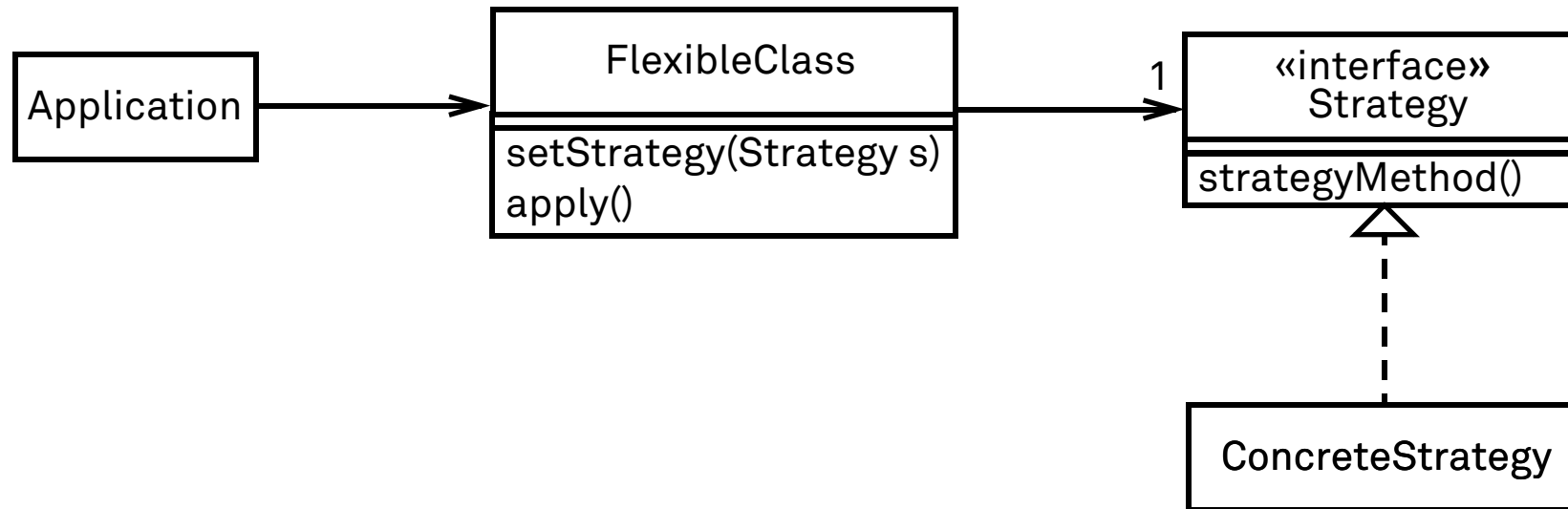
Literatur: Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 48-52
<http://www.springerlink.com/content/t38726/#section=660020&page=6&locus=71>

Entwurfsmuster *Strategie* – Konstruktionsidee

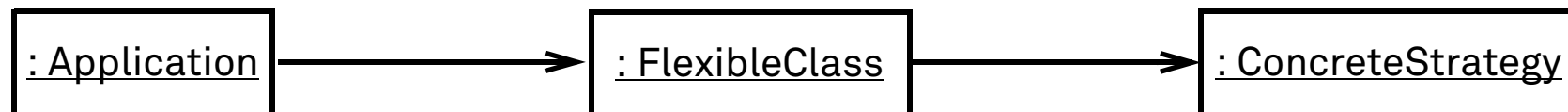
- ❑ Das zu ändernde Verhalten wird in einer eigenen **Strategie**-Klasse gekapselt, die eine vorgegebene Schnittstelle umsetzt.
- ❑ Bei der Ausführung wird auf das Verhalten eines Strategie-Objekts zugegriffen.
- ❑ Bei Änderungen wird das Strategie-Objekt ausgetauscht.



Entwurfsmuster *Strategie* – Objektstruktur

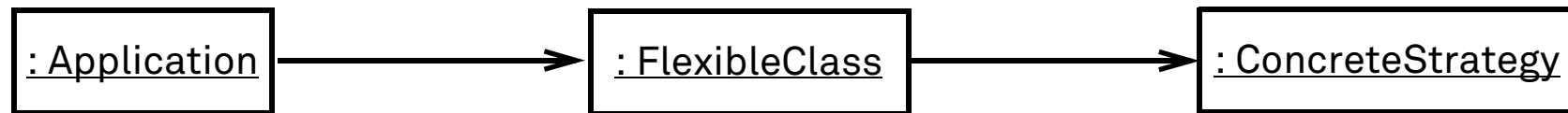


Objektdiagramm

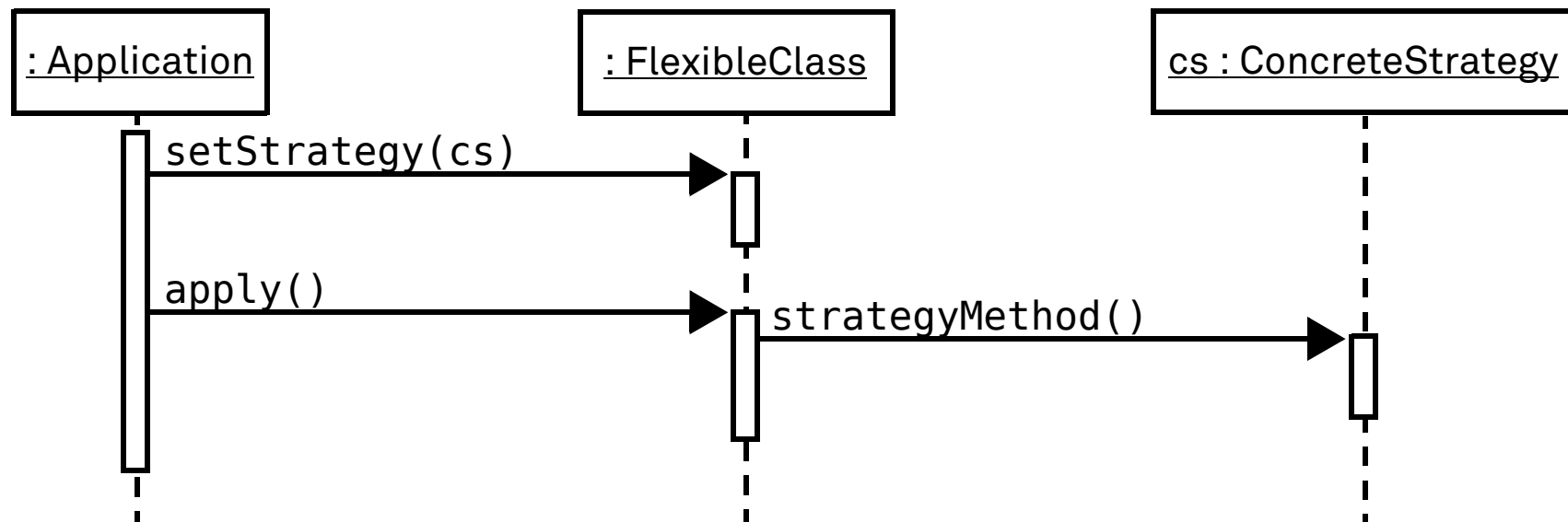


Entwurfsmuster *Strategie* – Objektstruktur

Objektdiagramm

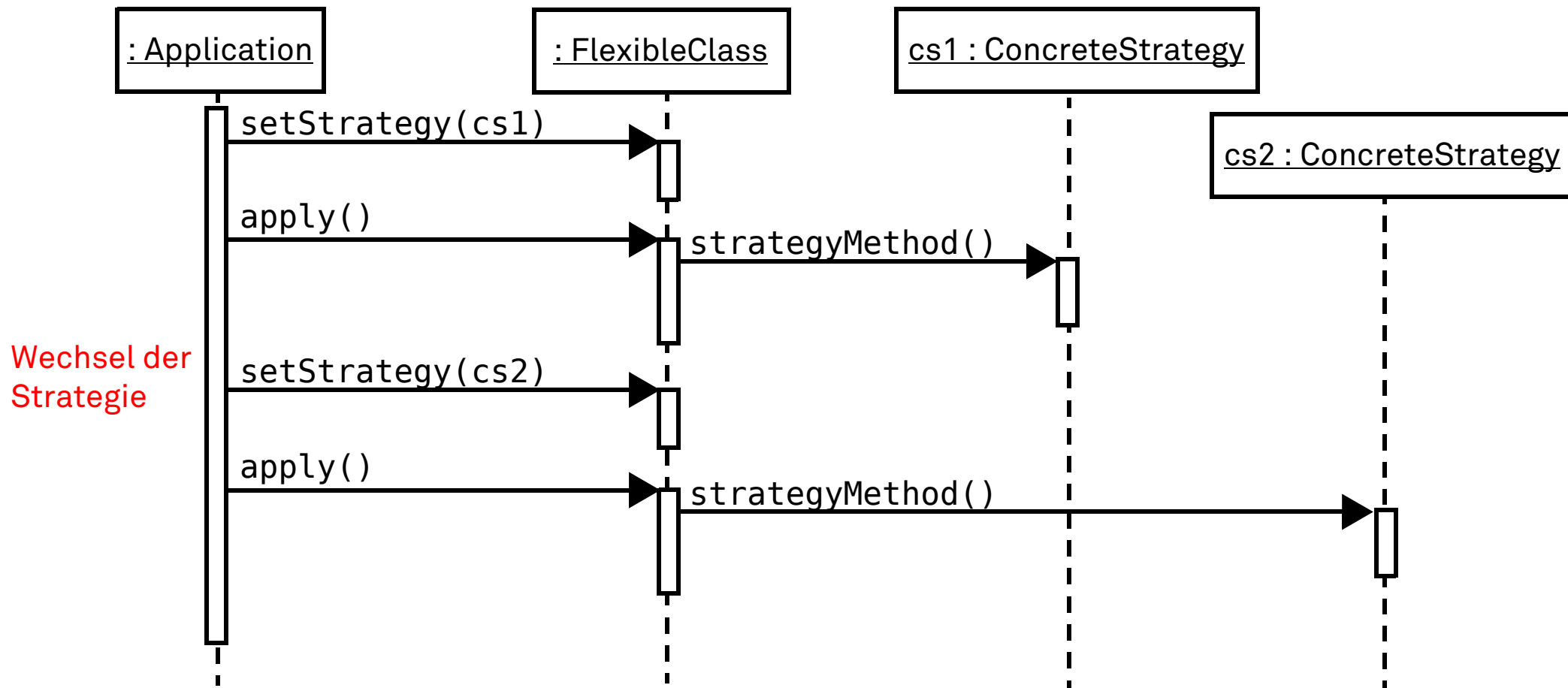


Sequenzdiagramm



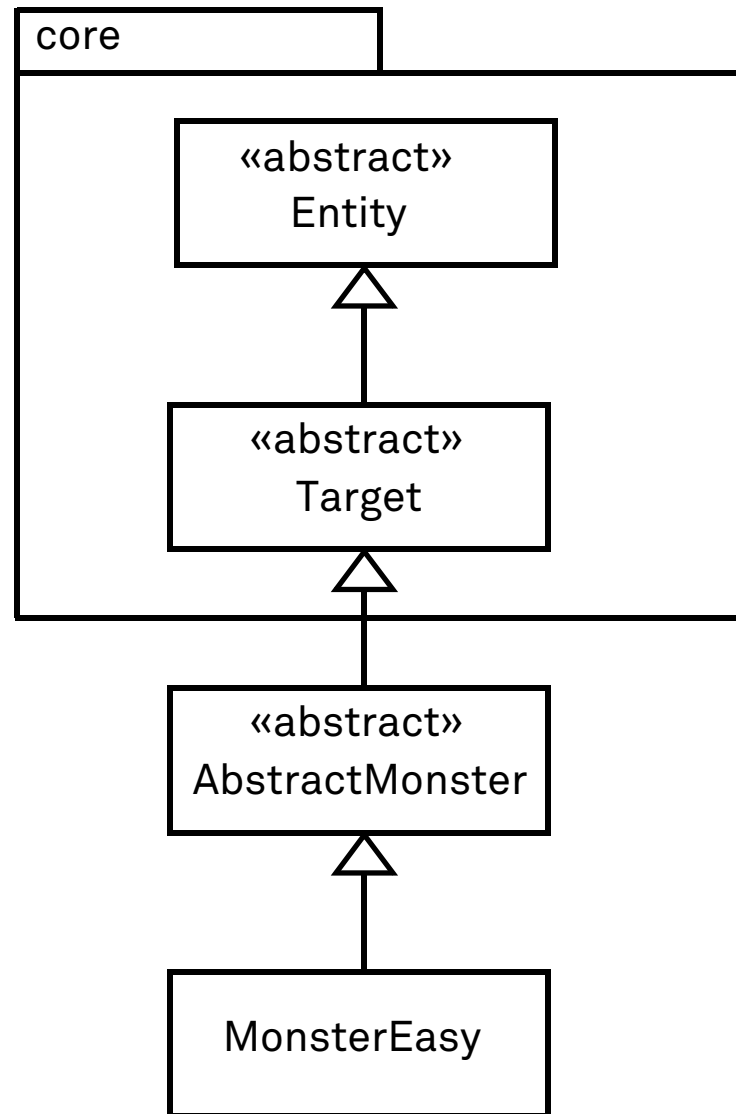
Entwurfsmuster *Strategie* – Objektstruktur

Änderung der Strategie während der Ausführung



Beispiel aus SWT-Starfighter

Nutzung des
Strategie-Musters
in der Klasse Entity



Paket des Frameworks,
keine Änderung
erwünscht

Klasse stellt die allen
Monstern gemeinsamen
Eigenschaften bereit

das einfache Monster,
das zu Beginn des Spiels
erscheint

Beispiel aus SWT-Starfighter

(Fortsetzung)

```
package edu.udo.cs.swtsf.core;  
public interface EntityBehaviorStrategy {  
    public void act(Entity host);  
}
```

funktionales Interface

```
package edu.udo.cs.swtsf.core;  
import edu.udo.cs.swtsf.util.BufferedGroup;  
import edu.udo.cs.swtsf.util.Group;  
public abstract class Entity {  
    private final Group<EntityBehaviorStrategy> behaviorStrategies  
        = new BufferedGroup<>();  
    ...  
}
```

Das Objekt der Datenstruktur Group erlaubt es,
einem Entity-kompatiblen Objekt mehrere verschiedene Strategien zuzuordnen,
die bei der Ausführung gemeinsam das Verhalten des Entity-kompatiblen Objekts bestimmen.

Beispiel aus *SWT-Starfighter*

(Fortsetzung)

```
public abstract class Entity {  
    private final Group<EntityBehaviorStrategy> behaviorStrategies  
                                   = new BufferedGroup<>();  
  
    ...  
    public void addBehaviorStrategy(EntityBehaviorStrategy strategy) {  
        behaviorStrategies.add(strategy);  
    }  
    public void removeBehaviorStrategy(EntityBehaviorStrategy strategy) {  
        behaviorStrategies.remove(strategy);  
    }  
    ...  
}
```

Strategien können als Strategie-Objekte hinzugefügt und entfernt werden.

Beispiel aus SWT-Starfighter

(Fortsetzung)

```
public abstract class Entity {  
    private final Group<EntityBehaviorStrategy> behaviorStrategies  
                                   = new BufferedGroup<>();  
  
    ...  
    public void addBehaviorStrategy(EntityBehaviorStrategy strategy) {  
        behaviorStrategies.add(strategy);  
    }  
    public void removeBehaviorStrategy(EntityBehaviorStrategy strategy) {  
        behaviorStrategies.remove(strategy);  
    }  
    final void updateBehaviors() {  
        behaviorStrategies.forEach( strategy ->  
            { if ( !Entity.this.isDisposed()  
                && Entity.this.getCurrentGame() != null)  
                { strategy.act(this); }  
            });  
    }  
    ...  
}
```

Anwenden aller Strategien

Entität ist noch aktiv
und gehört zum Spiel

Ausführen einer Strategie

Beispiel aus SWT-Starfighter

(Fortsetzung)

Weitere Methoden, um Umgang mit Strategien zu steuern und kontrollieren, da sich das Verhalten einer Entität möglicherweise im Spielverlauf ändern soll.

```
public abstract class Entity {
    private final Group<EntityBehaviorStrategy> behaviorStrategies
                                   = new BufferedGroup<>();

    ...
    public boolean hasBehaviorStrategies() {
        return !behaviorStrategies.isEmpty();
    }
    public boolean hasBehaviorStrategy(EntityBehaviorStrategy strategy) {
        return behaviorStrategies.contains(strategy);
    }
    public <T extends EntityBehaviorStrategy>
    boolean hasBehaviorStrategy(Class<T> strategyClass){
        return getBehaviorStrategy(strategyClass) != null;
    }
    ...
}
```

Verhalten ist festgelegt

ein bestimmtes Verhalten
(Objekt) wird genutzt

eine bestimmte Verhaltensform
(Klasse) wird genutzt

Zusammenfassung – Entwurfsmuster *Strategie*

- ❑ In Java ist eine Methode immer einer Klasse zugeordnet und kann daher nur dann während der Ausführung ausgetauscht werden, wenn ein anderes Objekt – das diese Methode anbietet – verwendet wird.
- ❑ Die Nutzung des Strategie-Musters ist daher in Java die einzige Möglichkeit, Verhalten während der Ausführung auszutauschen.
- ❑ In Java unterstützen Lambda-Ausdrücke die Nutzung des Strategie-Musters.
- ❑ Das Beispiel der Klasse `Entity` zeigt:
 - Den Spielobjekten der Klasse `Entity` kann Verhalten zugeordnet werden, obwohl die Klasse `Entity` selbst im Framework (Paket `core`) liegt und nicht geändert werden kann/soll.
 - Das Verwenden einer Datenstruktur, die mehrere Objekte der Klasse `EntityBehaviorStrategy` ermöglicht einen dynamischen Spielverlauf: bereits im Spiel aktive Spielobjekte wie `Monster` können in bestimmten Spielsituationen, in bestimmten Zeiträumen oder auf verschiedenen Spielebenen zusätzliches Verhalten zugeordnet oder entzogen bekommen. Dadurch kann das Gesamtverhalten aus vielen kleinteiligen Verhaltensdefinitionen gebildet werden.
 - Eine analoger Einsatz des Strategiemusters erfolgt für die Kollision von Spielobjekten durch die Klasse `EntityCollisionStrategy` und die entsprechende Verwaltung.
 - **Das Entwurfsmuster Strategie kann in Implementierungen komplex umgesetzt werden.**

Vergleich Durchlauf mit Iterator/Strategie

Iteratormuster

Vorteile:

- ❑ Der Ablauf kann außerhalb des Aggregats gesteuert werden.
- ❑ Es können gleichzeitig viele Durchläufe durch das Aggregat durchgeführt werden.
- ❑ Die im Aggregat abgelegten Elemente werden außerhalb des Aggregats bereitgestellt.

Nachteile:

- ❑ Die Reaktion des Iterators beim Einfügen/Löschen von Elementen im Aggregat während eines Durchlaufs ist unklar.
- ❑ Der Iterator kann den Fortschritt des Durchlaufs nicht beeinflussen.
- ❑ Das Aggregat kann das Ende eines Durchlaufs nicht erkennen.
- ❑ Konsequenz in Java:
In der Java-Bibliothek sind die Iteratoren der einfachen Aggregate **fail-fast** implementiert:

Sobald das Aggregat geändert wurde, wirft ein bereits existierender Iterator bei seiner nächsten Nutzung **immer** eine Ausnahme: `ConcurrentModificationException`.

Vergleich Durchlauf mit Iterator/Strategie

(Fortsetzung)

Strategiemuster

Vorteile:

- ❑ Der Durchlauf durch das Aggregat erfolgt nur an einer Stelle im Programm.
- ❑ Das Aggregat kontrolliert, wie der Durchlauf erfolgt und wann er beendet ist.
- ❑ Die im Aggregat abgelegten Elemente bleiben verborgen.

Nachteile:

- ❑ Es müssen immer alle Elemente behandelt werden.
Das Unterbrechen eines Durchlaufs ist nicht möglich.
- ❑ Das eventuelle Ergebnis eines Durchlaufs muss zusätzlich bereitgestellt werden.
- ❑ Die Reaktion der Methode zum Durchlaufen ist beim Einfügen/Löschen von Elementen im Aggregat und ineinander geschachtelten Durchläufen eventuell unklar.

Beispiel aus *SWT-Starfighter* – Interface Group

Interface Group aus dem Paket `edu.udo.cs.swtsf.util`

Zielsetzungen bei der Gestaltung:

- ❑ Es soll eine Datenstruktur zur Aufbewahrung von Elementen angeboten werden.
- ❑ Die aufbewahrten Elemente sollen während des Durchlaufs gelöscht oder neue Elemente eingefügt werden können, ohne dass eine undefinierte Reaktion der Datenstruktur eintritt.
- ❑ Mehrere Durchläufe sollen ineinander geschachtelt werden können.
- ❑ Beispielszenario:
Im Rahmen des Spielvorgangs werden zyklisch alle Spielobjekte betrachtet, um deren Folgezustand zu bestimmen. Dabei können zum Beispiel folgende Situationen entstehen:
 - Eine Rakete wird abgeschossen und damit ein neues Spielobjekt erzeugt.
Die Rakete bestimmt selbstständig ihr Ziel und muss dazu die anderen Spielobjekte nach einem geeigneten Ziel durchsuchen.
 - Eine Bombe explodiert und wird dabei vernichtet.
Die Bombe muss bei der Explosion alle Spielobjekte durchsuchen und diejenigen bestimmen, die von der Explosion betroffen sind.
- Eine Implementierung ist mit den von Java bereitgestellten Listen nicht möglich.

Beispiel aus SWT-Starfighter – Interface Group

```
public interface Group<E> {
    public void add(E element);
    public void remove(E element);
    public void forEach(Consumer<? super E> handle);
    ...
    public default int count(E element) {
        class ElementCounter implements Consumer<E> {
            int count;
            public void accept(E t) { if (t.equals(element)) { count++; } }
        };
        ElementCounter c = new ElementCounter();
        forEach(c);
        return c.count;
    }
    ...
}
```

lokale Klasse

nutzt
forEach-Methode

```
public interface Consumer<T> {
    void accept(T t);
}
```

Beispiel aus SWT-Starfighter – Klasse BufferedGroup

Implementierungskonzept:

- ❑ Objekte der Klasse `BufferedGroup` zählen die geschachtelten Aufrufe ihrer `forEach`-Methode im Zähler `iterationCount`.
- ❑ Die Liste `list` enthält die verwalteten Elemente des Typs `E`.
- ❑ Während der Ausführung der `forEach`-Methode wird die Liste nicht geändert. Die Änderungsanforderungen werden in einer zweiten Liste `buffer` abgelegt und erst nach Abschluss aller Ausführungen von `forEach` nachgeholt.

```
public class BufferedGroup<E> implements Group<E> {  
    private final List<E> list = new ArrayList<>(2);  
    private final List<Consumer<List<E>>> buffer = new ArrayList<>(2);  
    private int iterationCount = 0;  
    ...  
}
```

nutzt Consumer-Interface,
um Methodenaufrufe abzulegen

```
public interface Consumer<T> {  
    void accept(T t);  
}
```

Beispiel aus SWT-Starfighter – Interface BufferedGroup

```
public class BufferedGroup<E> implements Group<E> {
    private final List<E> list = new ArrayList<>(2);
    private final List<Consumer<List<E>>> buffer = new ArrayList<>(2);
    private int iterationCount = 0;
    public void forEach(Consumer<? super E> handle) {
        iterationCount++;
        List<E> tempList = list;
        for (E element : tempList) {
            handle.accept(element);
        }
        iterationCount--;
        performBufferedWritesIfPossible();
    }
    public void add(E element) {
        if (iterationCount > 0) {
            buffer.add( list -> list.add(element) );
        } else {
            list.add(element);
        }
    }
    public void remove(E element) { ... }
}
```

iterationCount wird vor dem Durchlauf erhöht und danach vermindert.

gespeicherte Methodenaufrufe werden ausgeführt, falls iterationCount == 0 gilt.

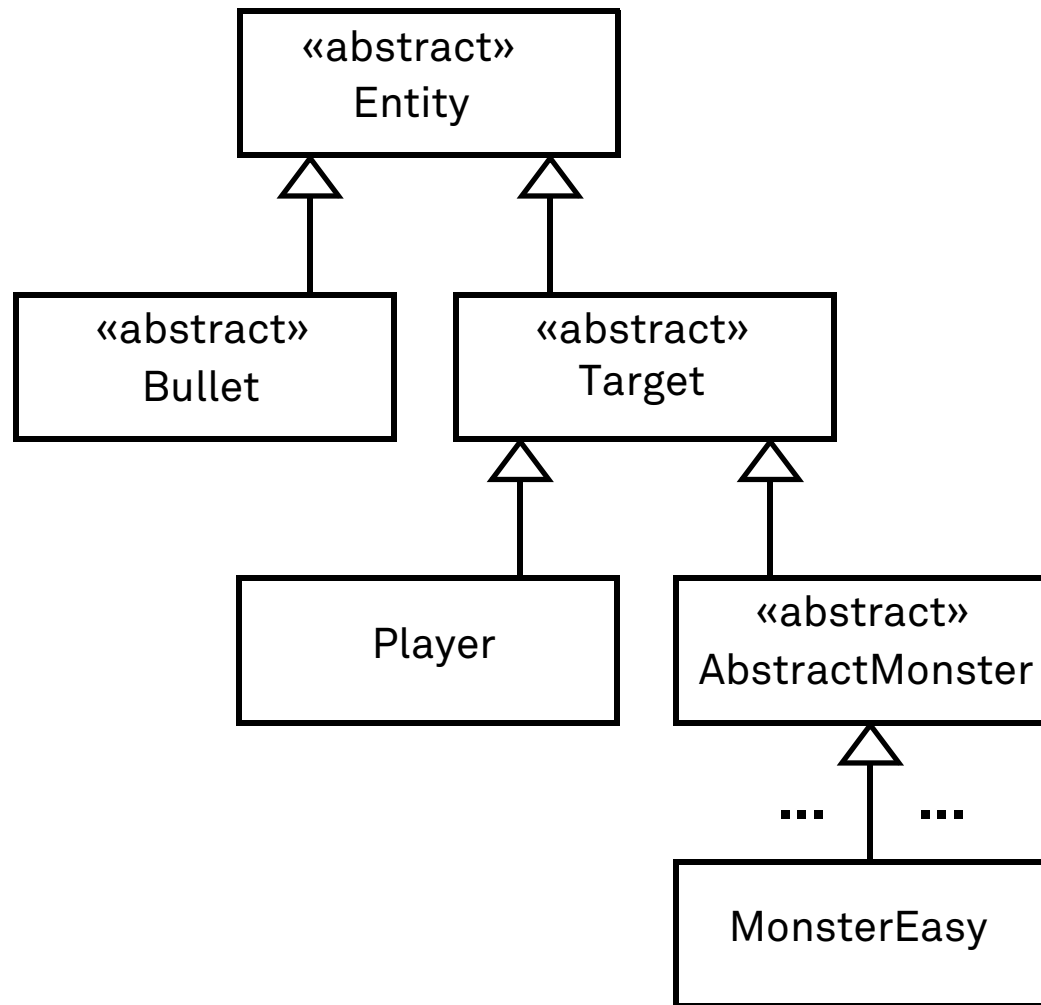
Beispiel aus SWT-Starfighter – Interface BufferedGroup

```
public class BufferedGroup<E> implements Group<E> {
    private final List<E> list = new ArrayList<>(2);
    private final List<Consumer<List<E>>> buffer = new ArrayList<>(2);
    private int iterationCount = 0;
    public void forEach(Consumer<? super E> handle) {
        iterationCount++;
        List<E> tempList = list;
        for (E element : tempList) {
            handle.accept(element);
        }
        iterationCount--;
        performBufferedWritesIfPossible();
    }
    public void add(E element) {
        if (iterationCount > 0) {
            buffer.add( list -> list.add(element) );
        } else {
            list.add(element);
        }
    }
    public void remove(E element) { ... }
}
```

falls aktuell ein Durchlauf erfolgt,
wird der Aufruf der add-Methode
in einem Objekt abgespeichert,
sonst wird list direkt verändert.

analog zu add

Analyse von Klassenstrukturen

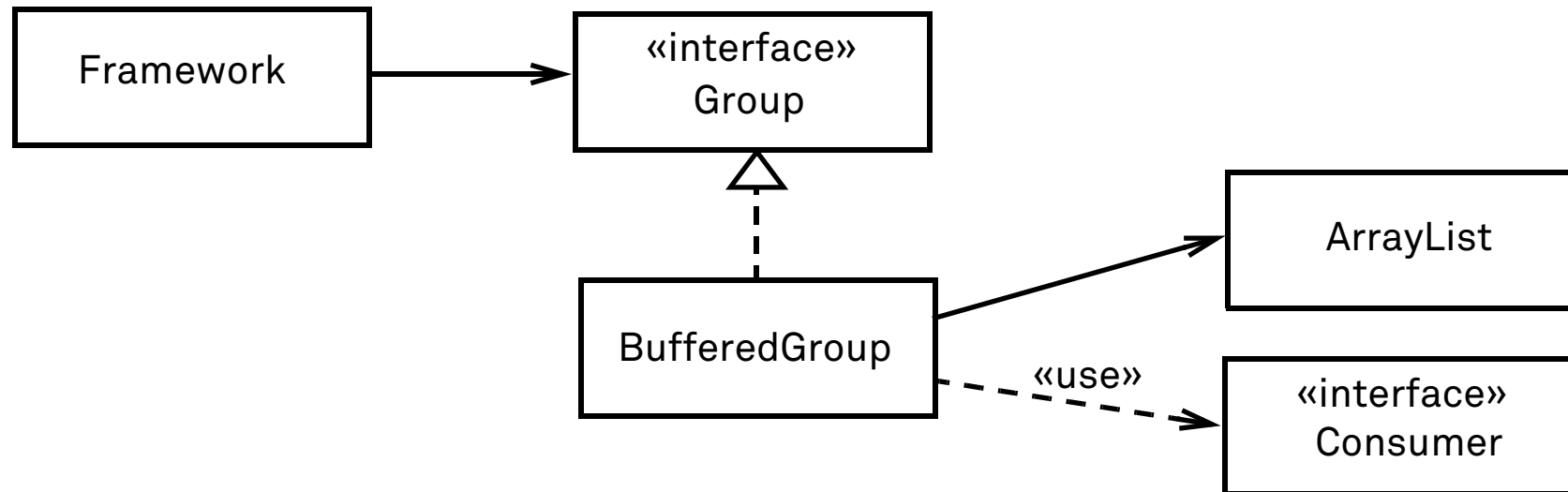


Klassenhierarchie fasst
Gemeinsamkeiten mehrerer
Unterklassen zusammen:

- gemeinsamer Typ durch Oberklasse
- gemeinsame Methoden in Oberklassen

Analyse von Klassenstrukturen

(Fortsetzung)



- ❑ Es wird nur eine Implementierung/Unterklasse von Group benötigt.
- ❑ Das Interface Group dient in dieser Struktur dazu, einen einfachen Austausch der Implementierung dieser Unterklasse zu ermöglichen.
- ❑ Die anderen Klassen des *SWT-Starfighter*-Frameworks nutzen ausschließlich Referenzen auf das Interface Group.
- ❑ Das Interface Group legt die Methoden fest, die vom Framework erwartet werden.
- ❑ BufferGroup nutzt bestehende Klassen, um die Anforderungen des Frameworks zu erfüllen.

Entwurfsmuster *Adapter*

Ein **Adapter**

erlaubt die Verbindung von Dingen mit unterschiedlichen Schnittstellen



Literatur: Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.221-224

<http://www.springerlink.com/content/gh615h/#section=297251&page=13&locus=62>

Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 215-218

<http://www.springerlink.com/content/jm3124/#section=390807&page=13&locus=47>

(nur **Objektadapter**) Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 66-67

<http://www.springerlink.com/content/t38726/#section=660021&page=1&locus=0>

Entwurfsmuster *Adapter*

(Fortsetzung)

allgemeine Beobachtungen:

- ❑ Adapter (in der physikalischen Welt) werden insbesondere dann benötigt, wenn zwei **fertige** Komponenten miteinander verbunden werden sollen.
- ❑ Ein Adapter ist ein vergleichsweise einfaches Verbindungsstück und insbesondere meist billiger als speziell aneinander angepasste Komponenten.

Beobachtungen für Software:

- ❑ Viele Klassen werden unabhängig von speziellen Problemlösungen erstellt.
- ❑ Eine solche Klasse bietet durch die von ihr bereitgestellten Methoden implizit eine Schnittstelle an.
- ❑ Methoden erwarten von den von ihnen benutzten Objekten bestimmte Schnittstellen.
- ❑ Fertige Klassen sollen gemeinsam die Lösung eines neuen Problems ergeben.
- ❑ Problem: Die angebotene und die erwartete Schnittstelle passen nicht zusammen.

Entwurfsmuster *Adapter*

(Fortsetzung)

allgemeine Beobachtungen:

- ❑ Adapter (in der physikalischen Welt) werden insbesondere dann benötigt, wenn zwei **fertige** Komponenten miteinander verbunden werden sollen.
- ❑ Ein Adapter ist ein vergleichsweise billiges Verbindungsstück und insbesondere billiger als speziell aneinander angepasste Komponenten.

Beobachtungen für Software:

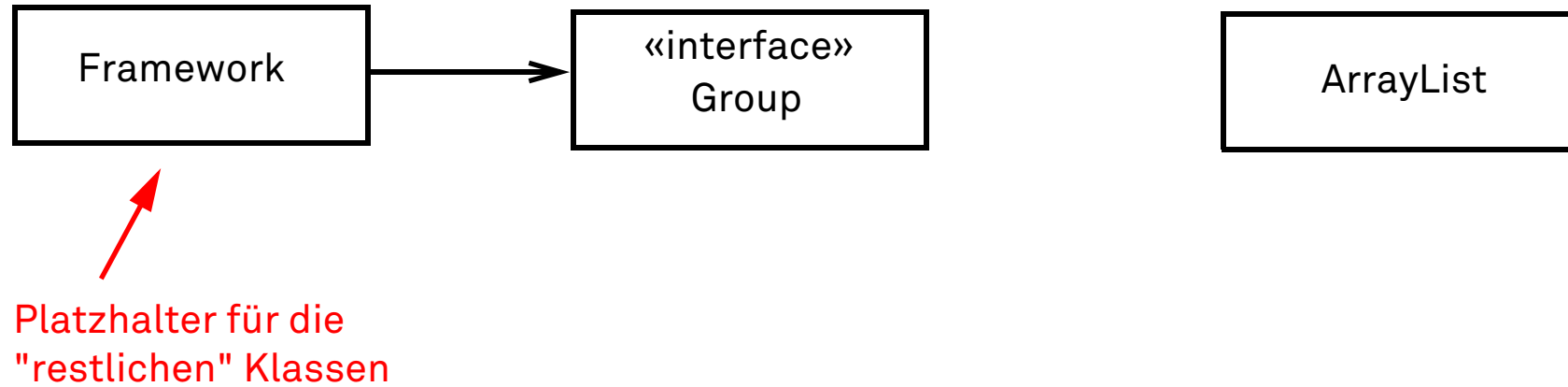
- ❑ Viele Klassen werden unabhängig von speziellen Problemlösungen erstellt.
- ❑ Eine solche Klasse bietet durch die von ihr bereitgestellten Methoden implizit eine Schnittstelle an.
- ❑ Methoden erwarten von den von ihnen benutzten Objekten bestimmte Schnittstellen.
- ❑ Fertige Klassen sollen gemeinsam die Lösung eines neuen Problems ergeben.
- ❑ Problem: Die angebotene und die erwartete Schnittstelle passen nicht zusammen.
Lösung: Implementierung einer (einfachen) Verbindung zwischen zwei Klassen
⇒ **Adapter** ist eine Klasse,
 - die die Methoden fertiger Klassen nutzt und
 - darauf aufbauend die Methoden bereitstellt,
 - die von den nutzenden Klassen gefordert werden.

Beispiel Adapter

- ❑ Problem beim *SWT-Starfighter*-Projekt:
 - Das Framework benötigt eine Datenstruktur zur Verwaltung von Spielobjekten, die beim Durchlaufen keine `ConcurrentModificationException` werfen soll.
 - Das Framework wird **fertig** entwickelt und fordert eine **vorgegebene** Schnittstelle:

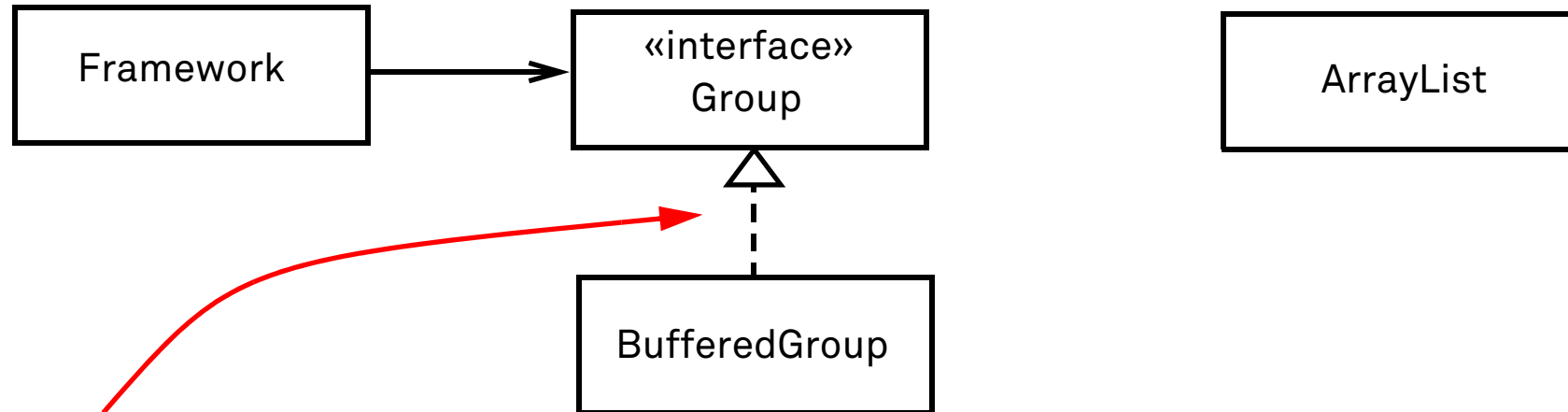
```
public interface Group<E> {  
    public void add(E element);  
    public void remove(E element);  
    public void forEach(Consumer<? super E> handle);  
    ...  
}
```
- ❑ Unterstützung für die Implementierung:
 - Es liegt eine implementierte, **getestete und praktisch bewährte** Klasse `ArrayList` vor, die Elemente verwalten kann und Iteratoren zum Durchlaufen anbietet.
- ❑ Lösung:
 - Da die bereits implementierten Klassen nicht geändert werden sollen, wird die Klasse `BufferedGroup` als Adapter implementiert, der die Verbindung zwischen `Group` und `ArrayList` herstellt.

Visualisierung Adapter



Visualisierung Adapter

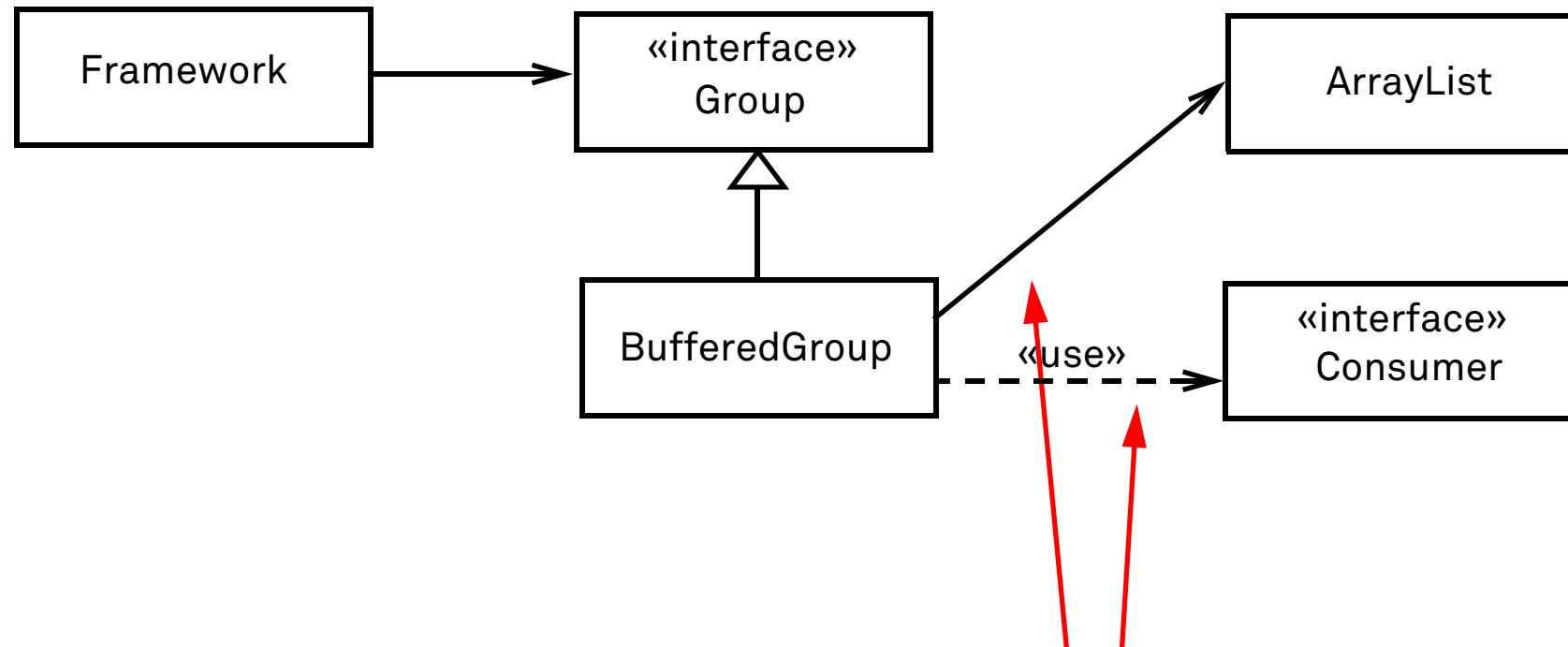
(Fortsetzung)



Realisierung
(notwendig, da Anwendung
ein Objekt des Adapters
benutzen soll)

Visualisierung Adapter

(Fortsetzung)

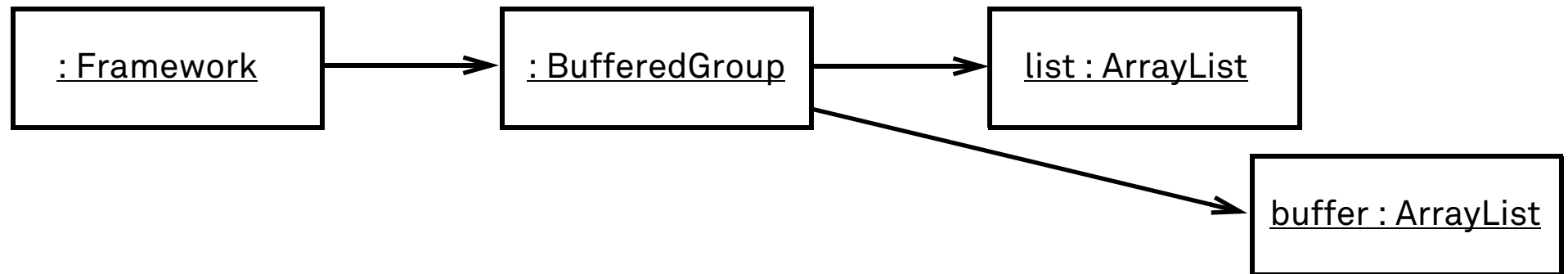


Nutzung bestehender Klassen,
um BufferedGroup mit geringem
Aufwand zu implementieren

Visualisierung Adapter

(Fortsetzung)

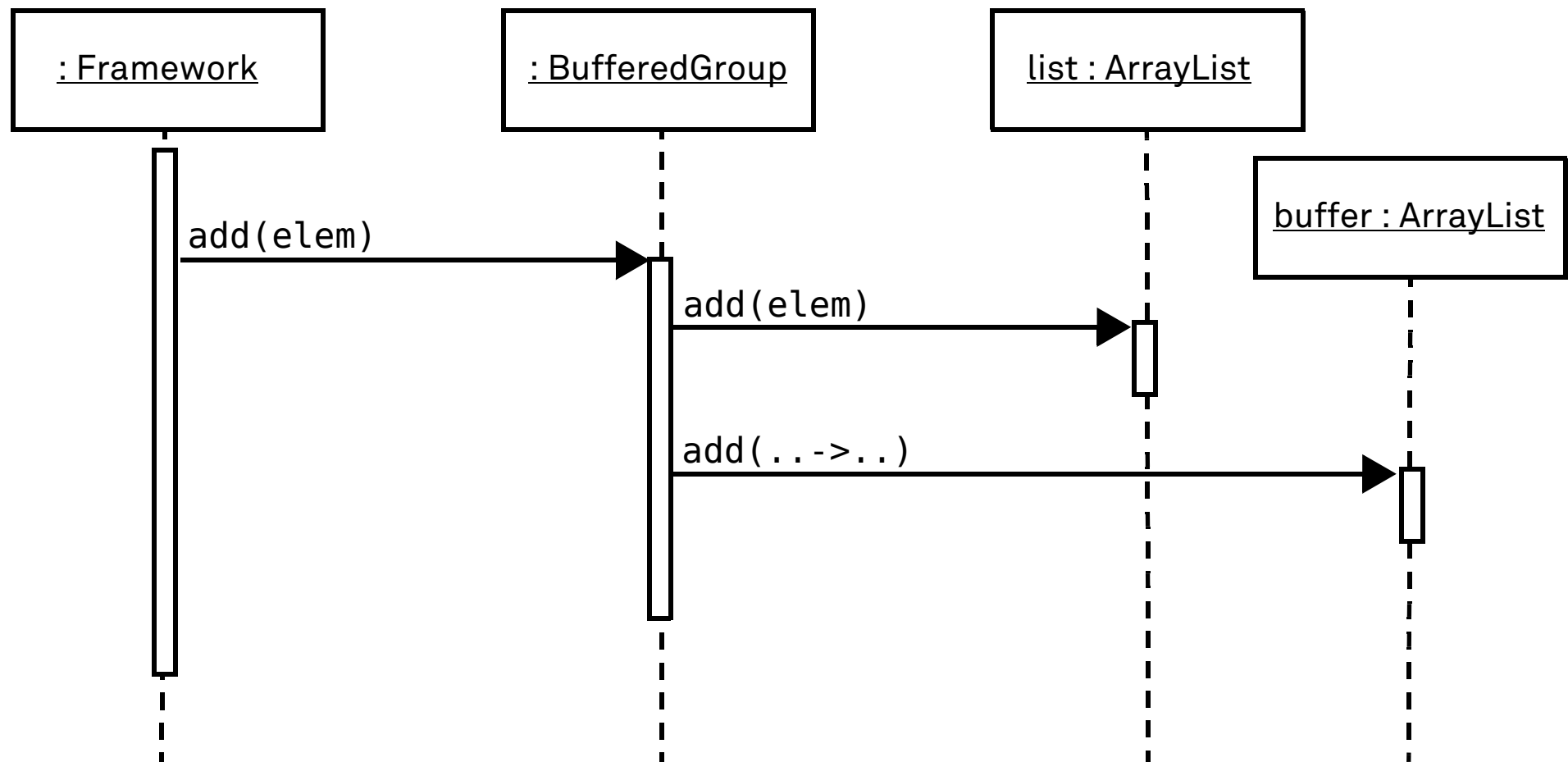
Objektstruktur bei der Ausführung



Visualisierung Adapter

(Fortsetzung)

Aufruffolge bei der Ausführung



Zusammenfassung

- ❑ vorgestelltes Entwurfsmuster:
Objektadapter
- ❑ Die Adapter-Klasse realisiert die gewünschte Schnittstelle (Group).
- ❑ Die Adapter-Klasse nutzt **Objekte** einer vorhandenen Klasse (ArrayList).
- ❑ Die Methoden der gewünschten Schnittstelle werden i.d.R. durch Benutzung von Methoden der vorhandenen Klasse umgesetzt.
- ❑ Die Anwendung (Framework) benutzt ein Objekt der Adapter-Klasse, auf das über eine Referenz (mit dem Typ des realisierten Interfaces) zugegriffen wird. Die Anwendung muss dazu die Realisierung und die Methodenabläufe im Adapter nicht kennen.

Literatur: Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.221-224
http://link.springer.com/chapter/10.1007/978-3-8348-9174-7_8

Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 215-218
http://link.springer.com/chapter/10.1007/3-540-30950-0_12

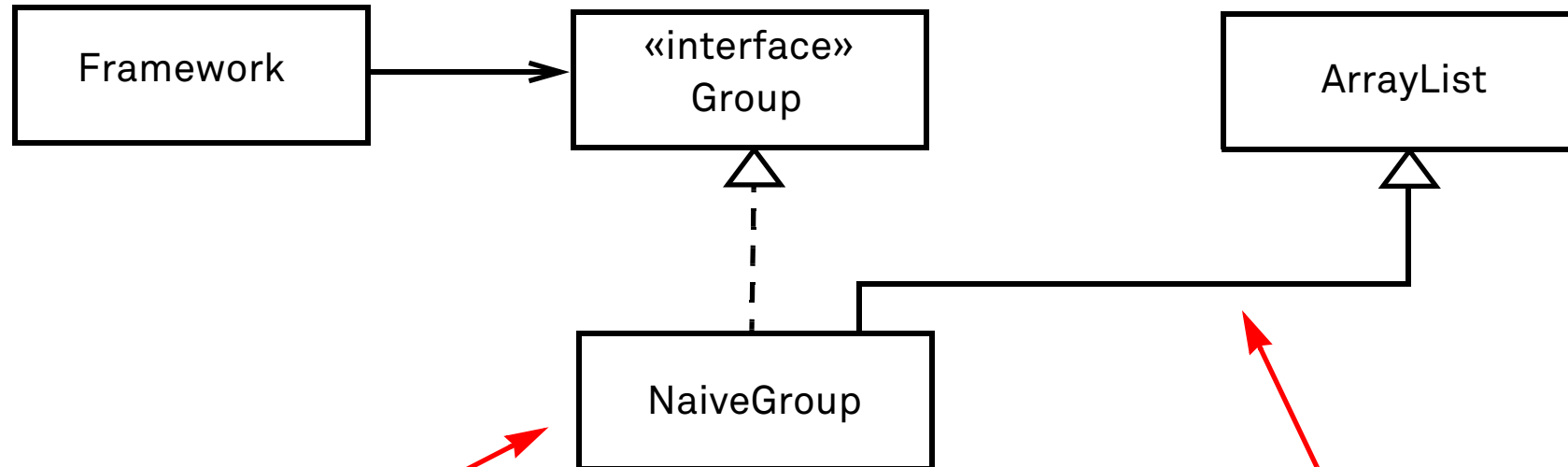
Zusammenfassung

(Fortsetzung)

- ❑ auch möglichst eine Implementierung als **Klassenadapter**
- ❑ Die Adapter-Klasse erbt die vorhandene Klasse, die zu seiner Realisierung eingesetzt wird:
 - Die Beziehung zwischen den Klassen wird auf der strukturellen Ebene hergestellt.
 - Der Adapter besitzt (durch das Erben) alle Eigenschaften der Klasse, die verwendet werden soll und daher angepasst werden muss.
(Es kann dadurch insbesondere auch mehr öffentliche Methoden geben, als der Adapter tatsächlich benötigt.)
- ❑ Die Adapter-Klasse realisiert die gewünschte Schnittstelle (Group):
 - Die Methoden der gewünschten Schnittstelle werden i.d.R. durch Benutzung von Methoden der geerbten, vorhandenen Klasse umgesetzt.
 - Bei Methoden mit gleicher Signatur ist in Java keine Umsetzung notwendig, falls die geerbte Methode bereits die gewünschte Semantik besitzt.
- ❑ Die Anwendung (Framework) benutzt ein Objekt der Adapter-Klasse über eine Referenz auf die vorgegebene Schnittstelle.

Visualisierung Klassenadapter

(Fortsetzung)



Anforderung:

NaiveGroup soll die Problematik der *fail-fast*-Klasse **ArrayList** ignorieren, beispielsweise

- um schnell eine erste Testversion der Software zu erstellen oder
- da sicher ist, dass im Spielverlauf keine Änderungen an der Liste auftreten werden

NaiveGroup erbt und besitzt damit alle Eigenschaften von **ArrayList**

Visualisierung Klassenadapter

(Fortsetzung)

Implementierung:

```
public class NaiveGroup extends ArrayList<E> implements Group<E> {  
    ...  
}
```

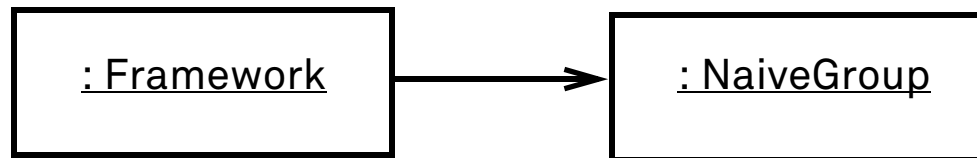
Anmerkungen:

- ❑ Die Methoden `add`, `remove` und `forEach` müssen nicht implementiert werden, da die Klasse `ArrayList` diese bereitstellt und vererbt.
- ❑ `NaiveGroup` besitzt durch die Spezialisierung jedoch auch noch alle anderen von `ArrayList` bereitgestellten Methoden.
- ❑ Innerhalb des Frameworks können diese jedoch nicht genutzt werden, da ausschließlich über Referenzen auf `Group` zugegriffen wird und diese Methoden dort nicht deklariert sind.

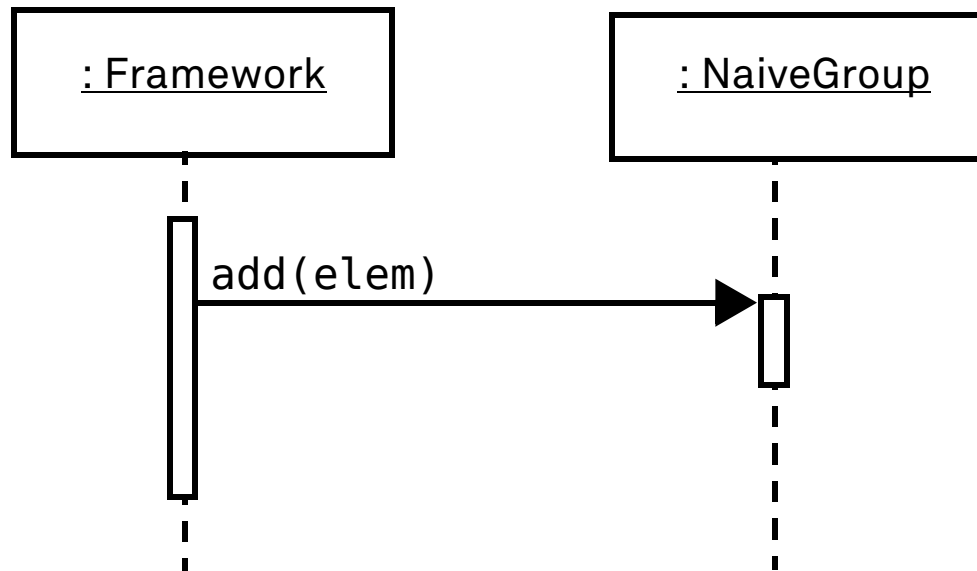
Visualisierung Klassenadapter

(Fortsetzung)

Objektstruktur bei der Ausführung



Aufruffolge bei der Ausführung



Vergleich Klassenadapter – Objektadapter

Vergleich der Objektdiagramme zeigt

- ❑ Klassenadapter
 - besteht aus nur einem Objekt,
 - das die geforderten Aufgaben selbst übernehmen kann.

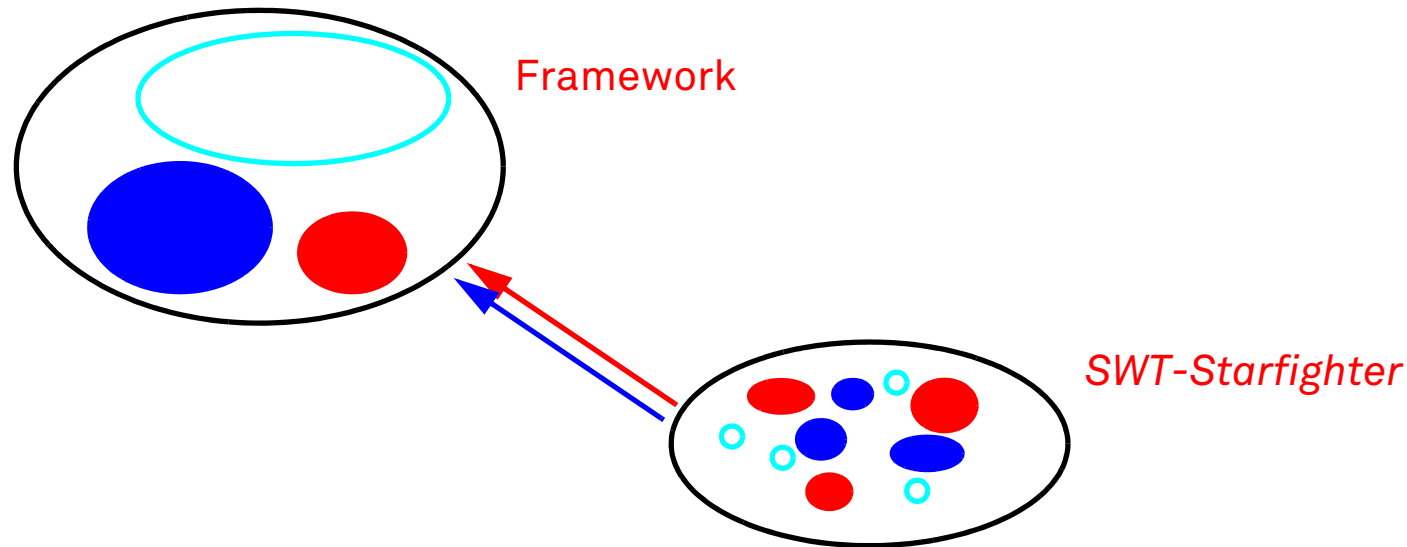
- ❑ Objektadapter
 - umfasst mehrere Objekte,
 - von denen der Adapter die passende Schnittstelle herstellt
 - und die "Arbeit" (teilweise) an andere Objekte weitergibt (**Delegation**).

Das Adapter-Objekt im Objektadapter erbringt also nur einen (eventuell kleinen) Teil der Anforderungen selbst und kennt zusätzlich weitere Objekt(e), mit deren Hilfe es die geforderten Aufgaben erfüllt!

Zusammenfassung: Entwurfsmuster Adapter

- ❑ Ein Adapter wird dann eingesetzt, wenn fertige Lösungen verwendet werden sollen, die nicht genau zu den Anforderungen passen.
- ❑ Adapter kommen daher insbesondere dann zum Einsatz, wenn Klassen aus Bibliotheken verwendet werden sollen.
- ❑ Ein Adapter ist immer eine einfach umzusetzende Komponente.
- ❑ Der Einsatz von Adaptern kann auch bei der Entwicklung neuer Software eingeplant werden, wenn dadurch der Entwicklungsaufwand verringert werden kann. Das ist genau in der Implementierung des *SWT-Starfighter* der Fall.
- ❑ Ohne Kenntnis der Entwicklungsgeschichte kann ein Adapter im fertigen Programmcode möglicherweise nur schwer erkannt werden.
(Der Adapter ist letztlich nur eine einzelne Klasse innerhalb einer größeren Struktur.)

Analyse SWT-Starfighter – Spielgestaltung und Weiterentwicklung



Erinnerung:

- ❑ Framework soll bei der Spielgestaltung unverändert bleiben.
- ❑ Spiel soll interessant/komplex/unerwartet weiterentwickelt werden können

Konsequenz:

- ❑ Es werden Datenstrukturen benötigt, die im Framework ausgewertet werden können und zugleich aber auch flexibel im Spiel zur Gestaltung benutzt werden können.

Entwurfsmuster *Dekorierer*

(Fortsetzung)

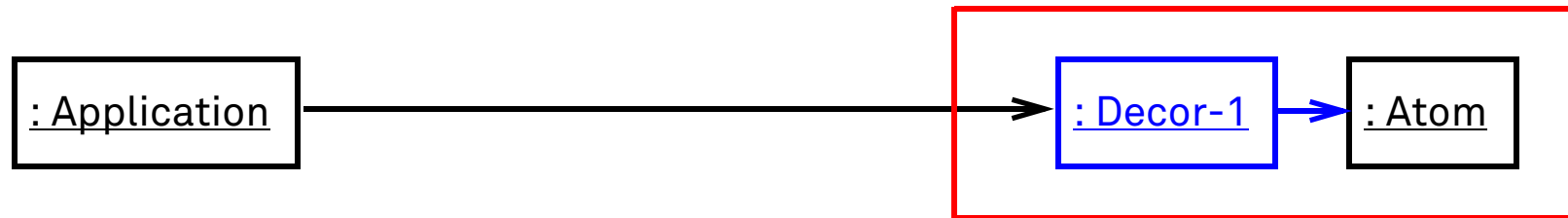
Ein Dekorierer

erlaubt das Anlegen von heterogenen Stapel-Strukturen.

Ansatz:

- ❑ Ein Stapel besteht aus strukturell unterschiedlichen Elementen,
 - dem letzten Element ohne einen Nachfolger und
 - vorangehenden Elementen, die genau einen direkten Nachfolger besitzen.
- ❑ Die unterschiedlichen Elemente bieten unterschiedliche Attribute und unterschiedliches Verhalten.
- ❑ Attribute und Verhalten beziehen sich aber inhaltlich immer auf alle folgenden Elemente.
- ❑ Jedes hinzukommende Element ergänzt (= "dekoriert") die vorhandenen Elemente.
- ❑ Der gesamte Stapel besitzt die gleiche Semantik wie sein erstes Element,
- ❑ Voraussetzung:
Alle Elemente müssen nach außen die gleiche Schnittstelle anbieten.

Visualisierung Entwurfsmuster *Dekorierer* (Objektdiagramm)



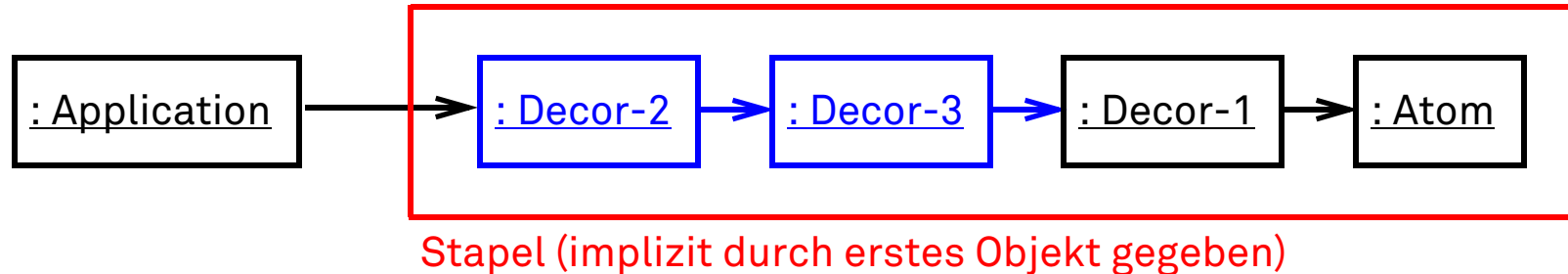
Stapel (implizit durch erstes Objekt gegeben)

Einsatzszenarien für solche Stapel:

- ❑ Einem Ausgangsobjekt (= letztes Element) sollen dynamisch weitere Eigenschaften (= vorangehende Elemente) hinzugefügt werden.
- ❑ Mit dem Hinzufügen einer Eigenschaft soll sich auch das Gesamtverhalten des Stapels ändern.

Visualisierung Entwurfsmuster *Dekorierer* (Objektdiagramm)

(Fortsetzung)

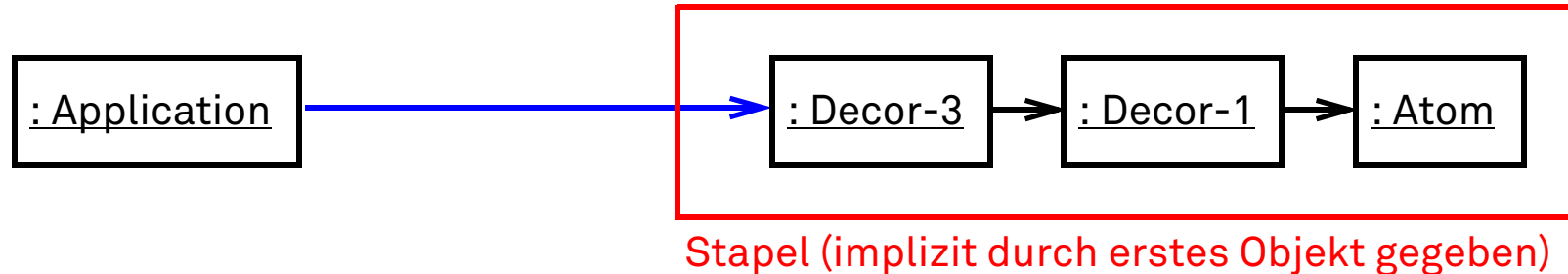


Einsatzszenarien für solche Stapel:

- ❑ Einem Ausgangsobjekt (= letztes Element) sollen dynamisch weitere Eigenschaften (= vorangehende Elemente) hinzugefügt werden.
- ❑ Mit dem Hinzufügen einer Eigenschaft soll sich auch das Gesamtverhalten des Stapels ändern.
- ❑ Die Reihenfolge, in der Eigenschaften hinzugefügt werden können, soll beliebig sein.

Visualisierung Entwurfsmuster *Dekorierer* (Objektdiagramm)

(Fortsetzung)

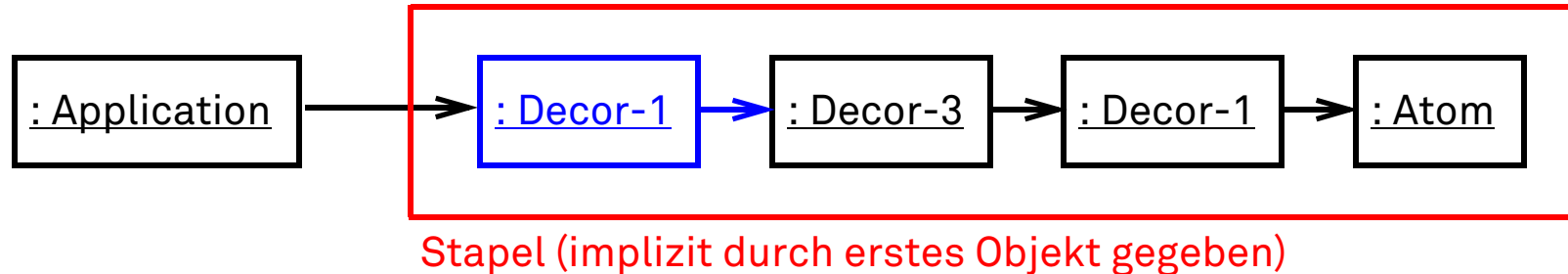


Einsatzszenarien für solche Stapel:

- ❑ Einem Ausgangsobjekt (= letztes Element) sollen dynamisch weitere Eigenschaften (= vorangehende Elemente) hinzugefügt werden.
- ❑ Mit dem Hinzufügen einer Eigenschaft soll sich auch das Gesamtverhalten des Stapels ändern.
- ❑ Die Reihenfolge, in der Eigenschaften hinzugefügt werden können, soll beliebig sein.
- ❑ **Eigenschaften sollen auch wieder entfernt werden können.**

Visualisierung Entwurfsmuster *Dekorierer* (Objektdiagramm)

(Fortsetzung)

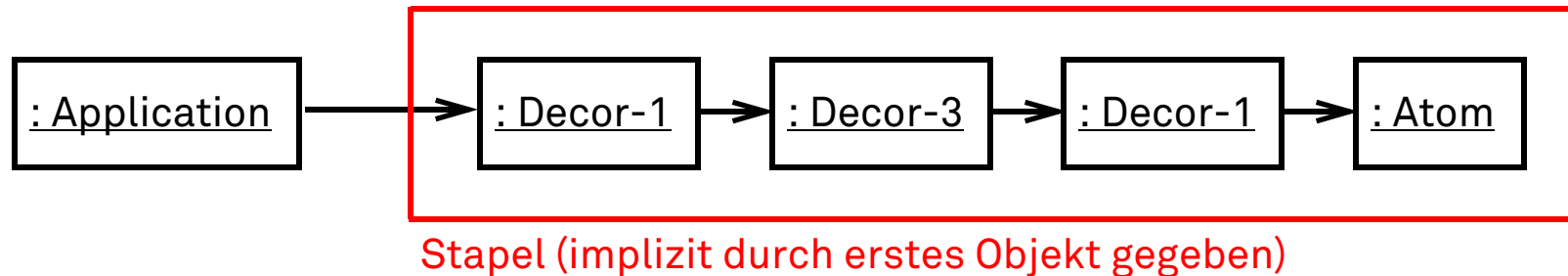


Einsatzszenarien für solche Stapel:

- ❑ Einem Ausgangsobjekt (= letztes Element) sollen dynamisch weitere Eigenschaften (= vorangehende Elemente) hinzugefügt werden.
- ❑ Mit dem Hinzufügen einer Eigenschaft soll sich auch das Gesamtverhalten des Stapels ändern.
- ❑ Die Reihenfolge, in der Eigenschaften hinzugefügt werden können, soll beliebig sein.
- ❑ Eigenschaften sollen auch wieder entfernt werden können.
- ❑ Eine Eigenschaft soll eventuell auch mehrfach hinzugefügt werden können.

Visualisierung Entwurfsmuster *Dekorierer* (Objektdiagramm)

(Fortsetzung)

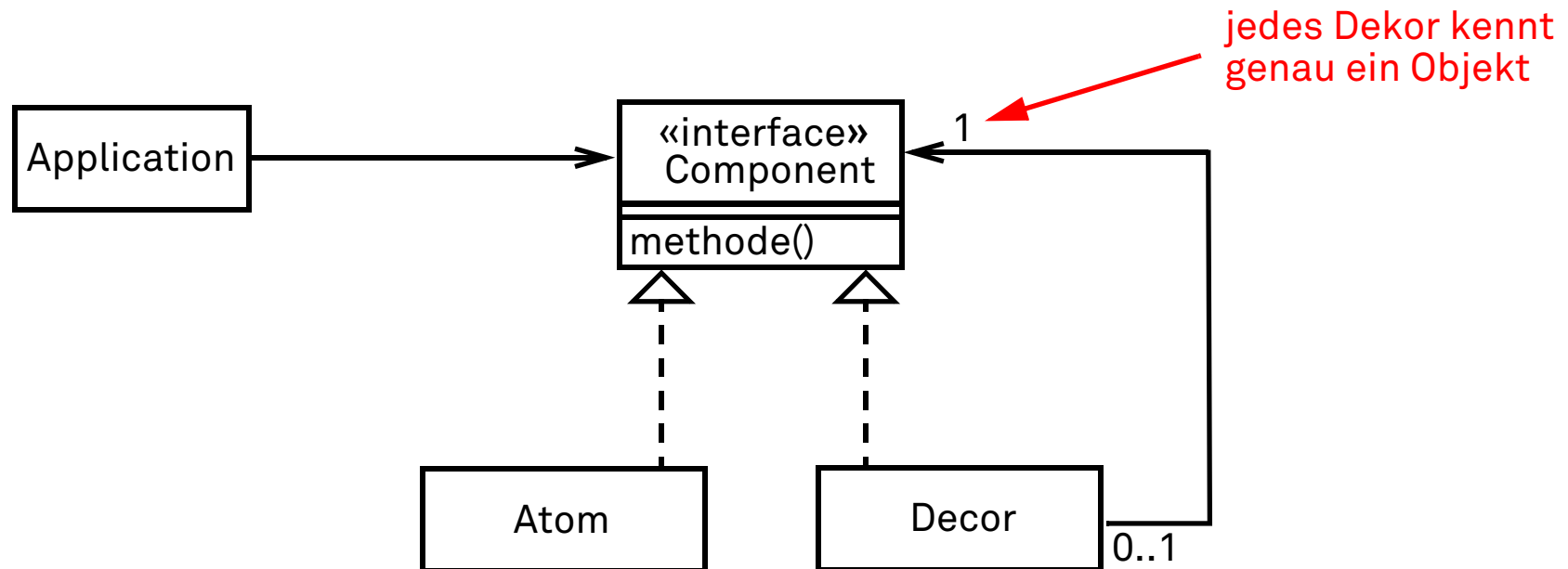


Einsatzszenarien für solche Stapel:

- ❑ Einem Ausgangsobjekt (= letztes Element) sollen dynamisch weitere Eigenschaften (= vorangehende Elemente) hinzugefügt werden.
- ❑ Die Reihenfolge, in der Eigenschaften hinzugefügt werden können, soll beliebig sein.
- ❑ Eigenschaften sollen auch wieder entfernt werden können.
- ❑ Mit dem Hinzufügen einer Eigenschaft soll sich auch das Gesamtverhalten des Stapels ändern.
- ❑ Eine Eigenschaft soll eventuell auch mehrfach hinzugefügt werden können.
- ❑ Weitere Eigenschaften sollen im Rahmen der Entwicklung leicht ergänzt werden können: Alle Objekte müssen die gleiche Schnittstelle erfüllen.

Einsatzszenarien für solche Stapel

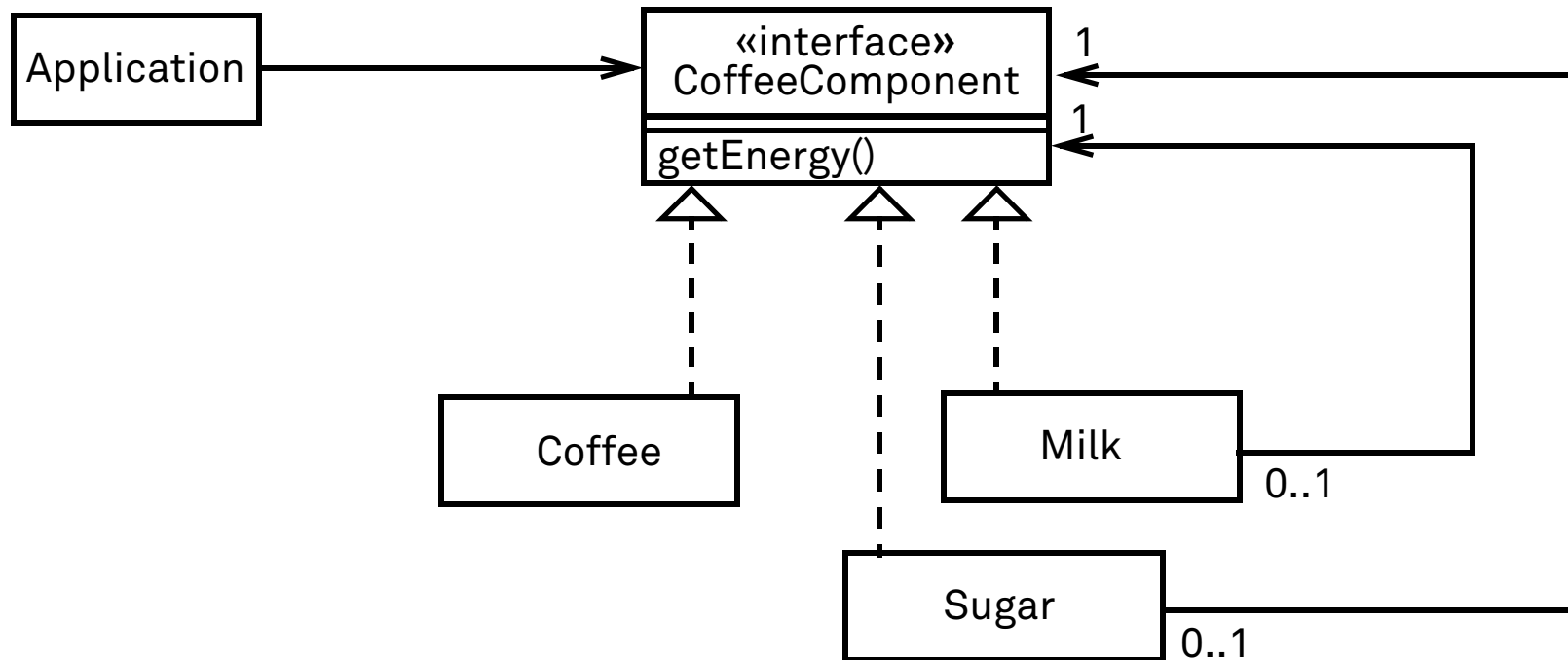
- Weitere Eigenschaften sollen im Rahmen der Entwicklung leicht ergänzt werden:
Alle Objekte müssen die gleiche Schnittstelle erfüllen.



Beispiele für den Einsatz des Entwurfsmusters *Dekorier*

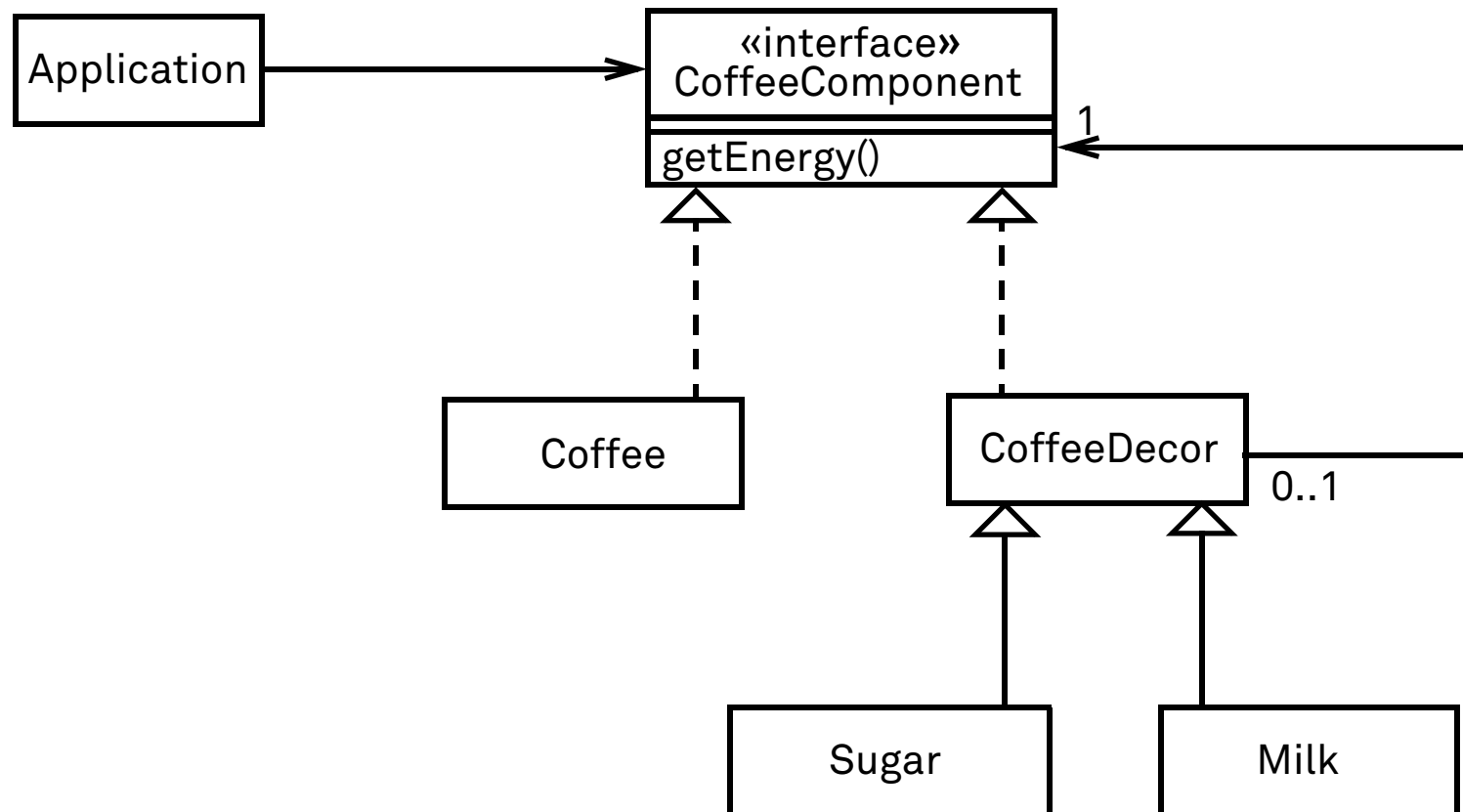
- ❑ GUI-Komponenten werden "dekoriert":
 - Umrahmungen werden zu Textfeld oder Graphikbereich hinzugefügt.
 - Funktionsleisten werden zu einem Fenster hinzugefügt.
 - Horizontaler und vertikaler Scrollbar wird zu Textfeld hinzugefügt.
- ❑ Dateien (Stream-Klassen der Java-Bibliothek)
 - Datei ist eine Folge von Byte (Hardware-nahe Betrachtung).
 - Durch Dekoration kann Verhalten hinzugefügt werden, das mehrere Bytes gemeinsam interpretiert.
 - Durch Dekoration kann Verhalten hinzugefügt werden, das die Dateiverarbeitung verändert (z.B. Puffern beim Einlesen und Schreiben).
- ❑ nicht-technisches Beispiel:
Kaffee wird (eventuell mehrfach) dekoriert durch
 - Zucker, Sahne, Milch, Schokolade, ...
 - wobei jedes Dekor zu einer spezifischen Erhöhung des Energiegehalts führt.

Visualisierung (Beispiel: Energieberechnung für Kaffee)



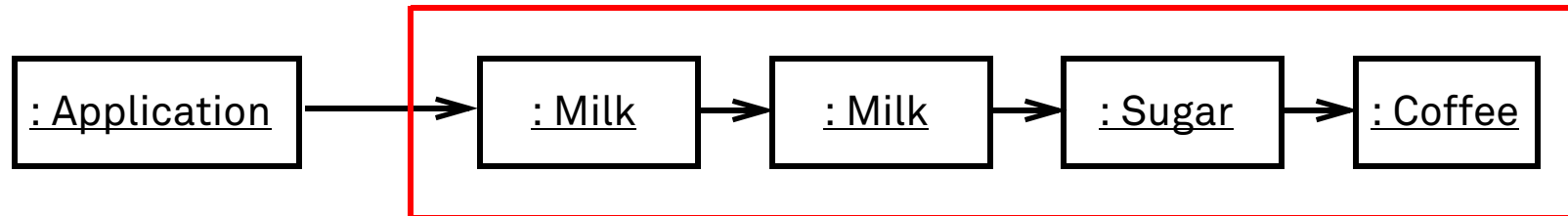
Visualisierung (Beispiel: Energieberechnung für Kaffee) (alternative Implementierung)

(Fortsetzung)



Visualisierung (Beispiel: Energieberechnung für Kaffee)

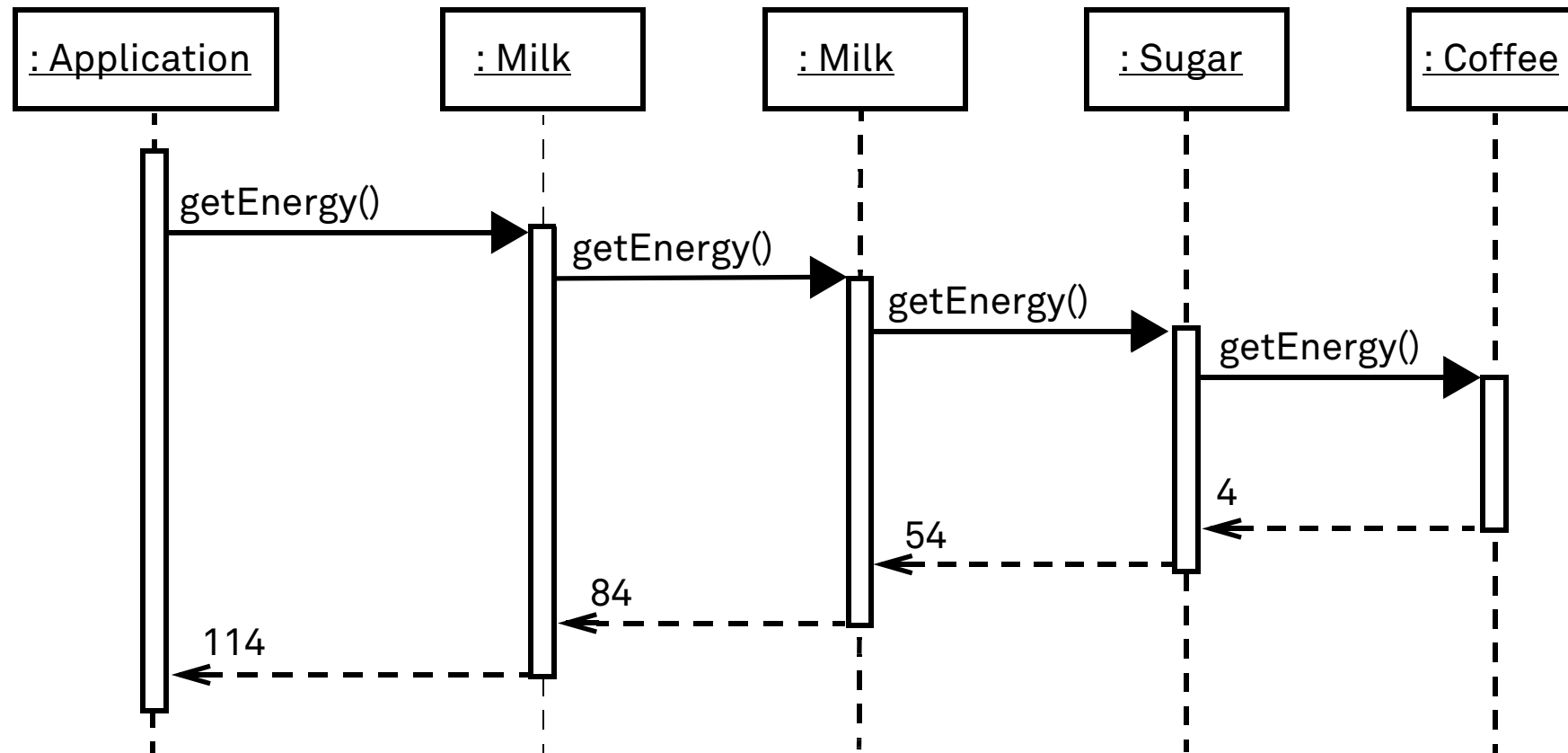
(Fortsetzung)



```
public class Milk implements CoffeeComponent {  
    ...  
    public int getEnergy() { return 30 + next.getEnergy(); }  
}  
public class Sugar implements CoffeeComponent {  
    ...  
    public int getEnergy() { return 50 + next.getEnergy(); }  
}  
public class Coffee implements CoffeeComponent {  
    ...  
    public double getEnergy() { return 4; }  
}
```

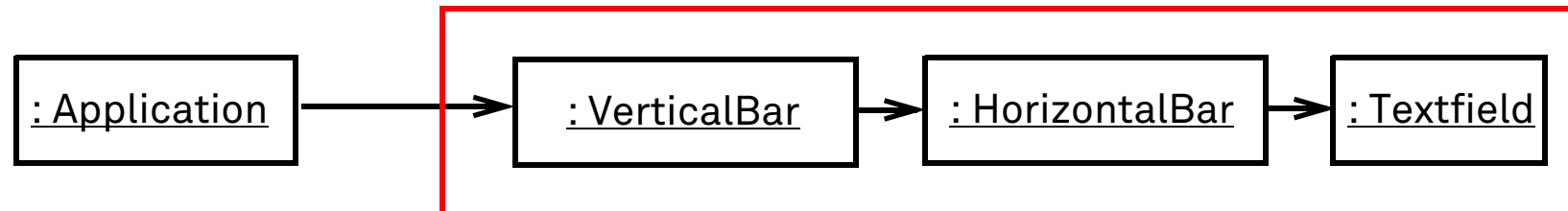
Visualisierung (Beispiel: Energieberechnung für Kaffee) (Methodenaufrufe)

(Fortsetzung)

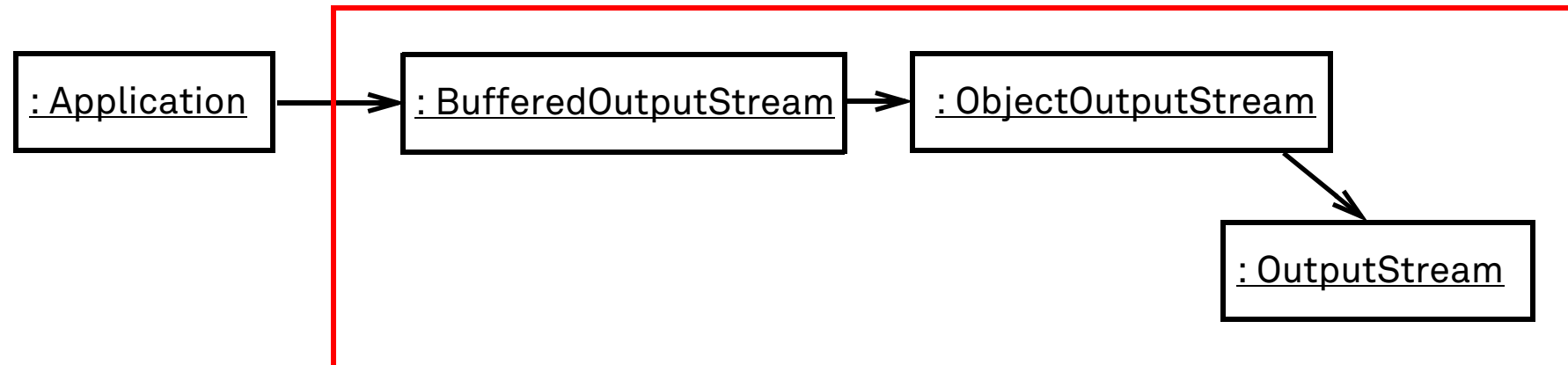


Visualisierungen (weitere Beispiele)

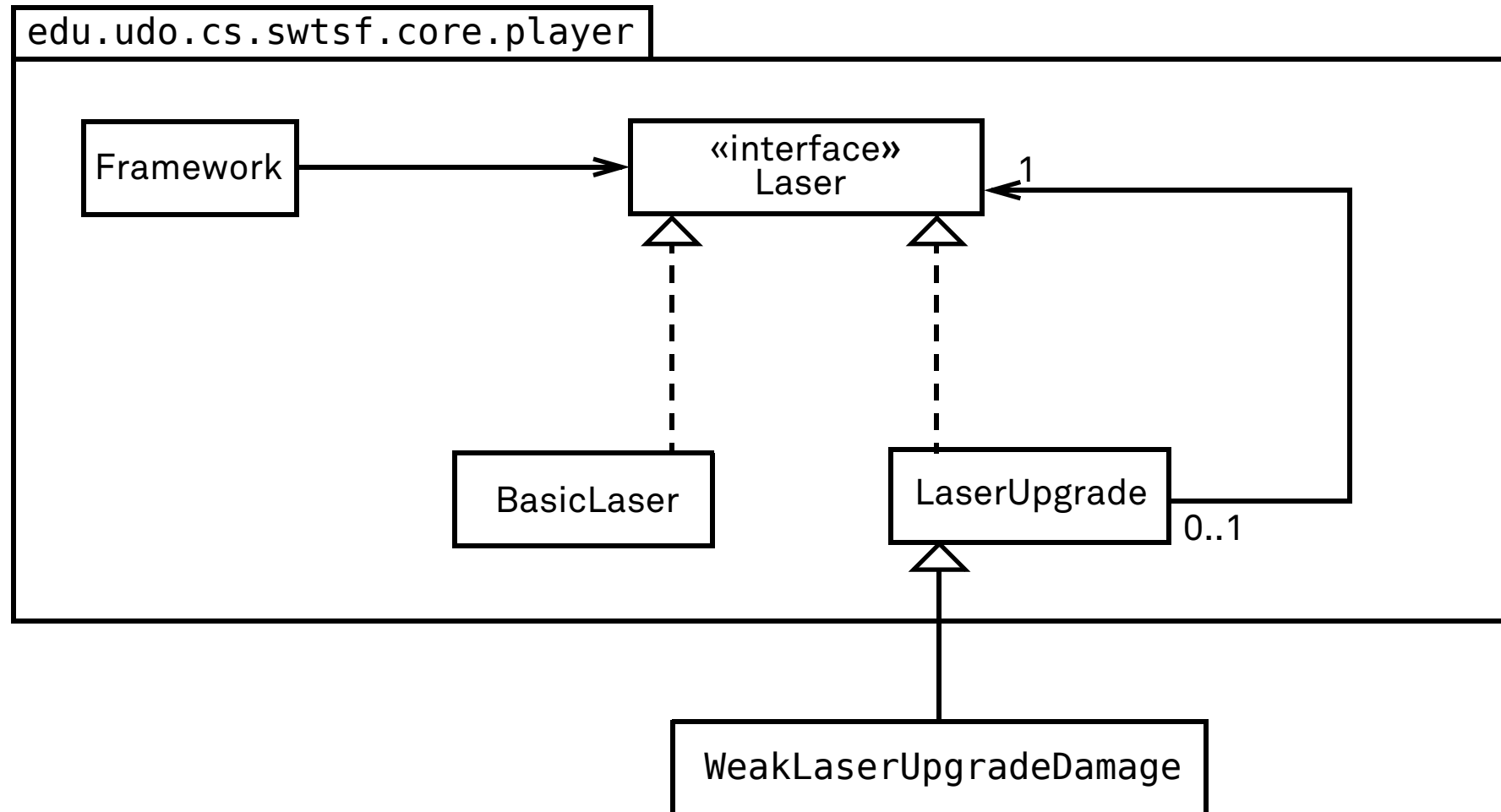
dekorierte Fenster



dekorierte Streams




Beispiel aus SWT-Starfighter – Dekorierer zum Interface Laser



Beispiel aus SWT-Starfighter – Dekorierer zum Interface Laser

(Fortsetzung)

```
public interface Laser extends Iterable<Laser> {  
    public Laser getDecorated();  
    public int getDamage();  
    public int getBulletSize();  
    public double getBulletSpeed();  
    public int getBulletLifeTime();  
    public int getCooldownTime();  
    public default Iterator<Laser> iterator() {  
        return new LaserIterator(this);  
    }  
    public default boolean contains(LaserUpgrade upgrade) {  
        return contains(upgrade.getClass());  
    }  
    public default boolean contains(Class<? extends LaserUpgrade> lu) {  
        return lu.isInstance(this) || (getDecorated() != null  
            && getDecorated().contains(lu));  
    }  
    ...  
}
```



siehe Folie 166

Beispiel aus SWT-Starfighter – Dekorierer zum Interface Laser

(Fortsetzung)

```
public interface Laser extends Iterable<Laser> {  
    public Laser getDecorated();           gibt das nächste Dekor zurück  
    public int getDamage();  
    public int getBulletSize();  
    public double getBulletSpeed();  
    public int getBulletLifeTime();  
    public int getCooldownTime();  
    public default Iterator<Laser> iterator() {  
        return new LaserIterator(this);  
    }  
    public default boolean contains(LaserUpgrade upgrade) {  
        return contains(upgrade.getClass());  
    }  
    public default boolean contains(Class<? extends LaserUpgrade> lu) {  
        return lu.isInstance(this) || (getDecorated() != null  
            && getDecorated().contains(lu));  
    }  
    ...  
}
```

Beispiel aus SWT-Starfighter – Dekorierer zum Interface Laser

(Fortsetzung)

```
public interface Laser extends Iterable<Laser> {  
    public Laser getDecorated();  
    public int getDamage();  
    public int getBulletSize();  
    public double getBulletSpeed();  
    public int getBulletLifeTime();  
    public int getCooldownTime();  
    public default Iterator<Laser> iterator() {  
        return new LaserIterator(this);  
    }  
    public default boolean contains(LaserUpgrade upgrade) {  
        return contains(upgrade.getClass());  
    }  
    public default boolean contains(Class<? extends LaserUpgrade> lu) {  
        return lu.isInstance(this) || (getDecorated() != null  
            && getDecorated().contains(lu));  
    }  
    ...  
}
```

} geben Informationen über den
dekorierten Laser zurück

Beispiel aus SWT-Starfighter – Dekorierer zum Interface Laser

(Fortsetzung)

```
public interface Laser extends Iterable<Laser> {  
    public Laser getDecorated();  
    public int getDamage();  
    public int getBulletSize();  
    public double getBulletSpeed();  
    public int getBulletLifeTime();  
    public int getCooldownTime();  
    public default Iterator<Laser> iterator() {  
        return new LaserIterator(this);  
    }  
    public default boolean contains(LaserUpgrade upgrade) {  
        return contains(upgrade.getClass());  
    }  
    public default boolean contains(Class<? extends LaserUpgrade> lu) {  
        return lu.isInstance(this) || (getDecorated() != null  
            && getDecorated().contains(lu));  
    }  
    ...  
}
```

prüft auf bereits vorhandene Dekore

prüft auf ein bereits vorhandenes Dekor einer Klasse

Beispiel aus SWT-Starfighter – Dekorierer zum Interface Laser

(Fortsetzung)

```
public class BasicLaser implements Laser {  
    private final int damage;  
    private final int lifeTime;  
    private final int coolDownTime;  
    private final int bulletSize;  
    private final double bulletSpeed;  
    public BasicLaser(...) {  
        ...  
    }  
    public Laser getDecorated() {  
        return null;  
    }  
    public int getDamage() {  
        return damage;  
    }  
    public int getBulletSize() {  
        return bulletSize;  
    }  
    ...  
}
```

ist **Atom** dieses Dekorierers

Attribute für die Eigenschaften
eines Lasers

Attribute werden im Konstruktor
mit Werten belegt

BasicLaser ist Atom, daher
kein Nachfolger möglich

get-Methoden werden durch das
Interface Laser vorgegeben

Beispiel aus SWT-Starfighter – Dekorierer zum Interface Laser

(Fortsetzung)

```
public class LaserUpgrade implements Laser {  
    private Laser decorated = null;  
    public void setDecorated(Laser laser) {  
        decorated = laser;  
    }  
    public Laser getDecorated() {  
        return decorated;  
    }  
    public int getDamage() {  
        if (getDecorated() == null) {  
            throw new IllegalStateException("getDecorated() == null");  
        }  
        return getDecorated().getDamage();  
    }  
    ...  
}
```

ist Oberklasse für die
Dekore dieses Dekorierers

Die Klasse LaserUpdate bildet die Schnittstelle zwischen dem Framework und einer konkreten Implementierung des Spiels und bietet get-Methoden ohne Wirkung an.

Beispiel aus *SWT-Starfighter* – Dekorierer zum Interface Laser

(Fortsetzung)

Das Dekor WeakLaserUpgradeDamage sorgt dafür, das bisher schwache Laser verbessert werden.

Hat der Laser allerdings schon ein Dekor der Klasse WeakLaserUpgradeDamage, so führt ein weiteres nicht zu einer weiteren Verbesserung.

```
public class WeakLaserUpgradeDamage implements LaserUpgrade {  
    public int getDamage() {  
        if (!contains(this.getClass())) {  
            return getDecorated().getDamage()+2;  
        }  
        return getDecorated().getDamage();  
    }  
}
```

weiteres Beispiel aus *SWT-Starfighter* – Dekorierer zum Interface `EntityStream`

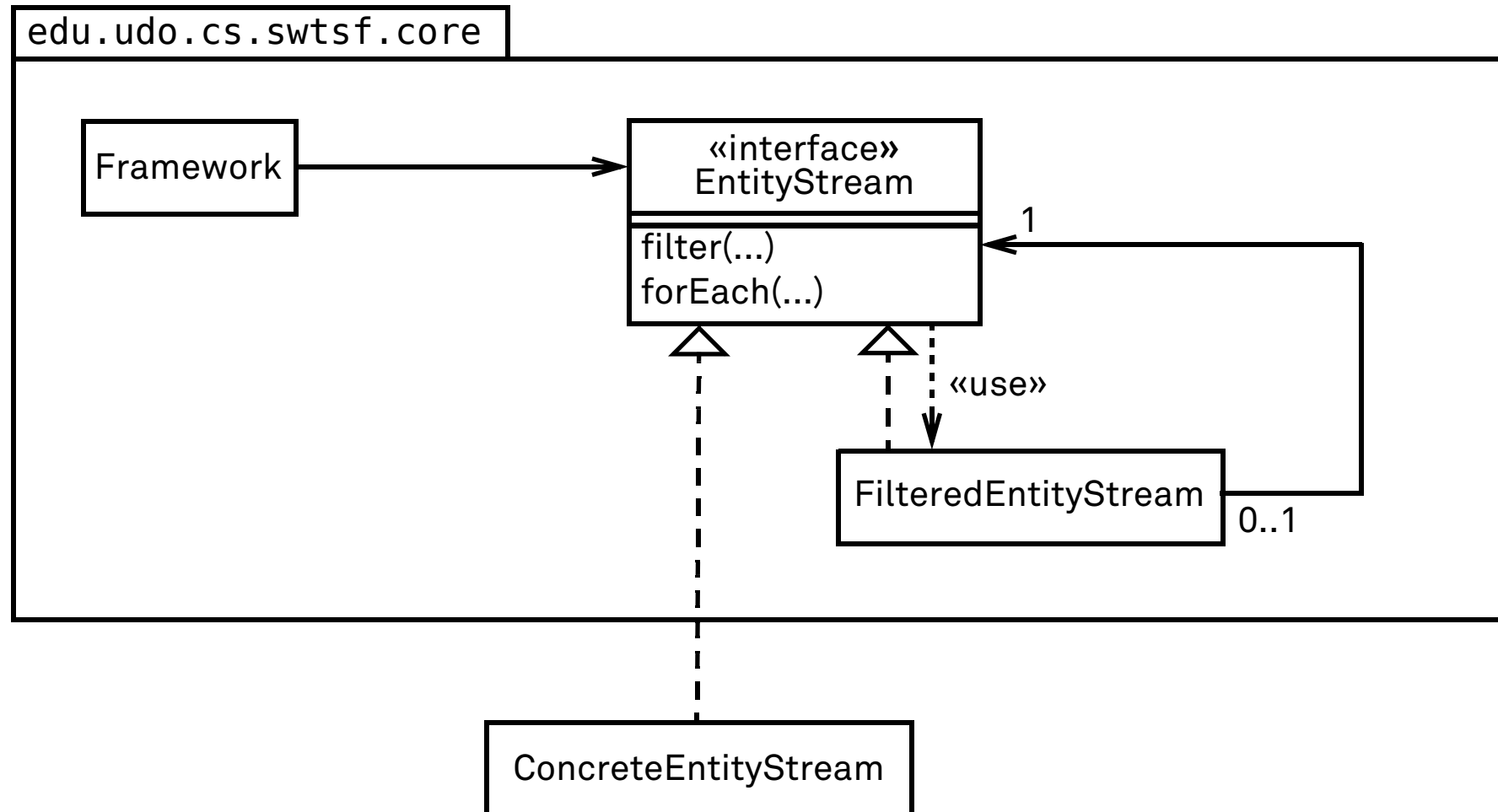
Dieses Beispiel zeigt eine etwas andere Umsetzung des Dekorierermusters, bei dem auf ein explizites Verbinden der Dekore verzichtet wird. Stattdessen werden die Dekore während der Ausführung über Methodenaufrufe erzeugt und über Referenzen in den Methodenaufrufen aneinander gehängt.

Idee zur dieser Nutzung des Dekorierermusters:

- ❑ Der Dekorierer arbeitet auf einem Objekt einer das Interface `EntityStream` implementierenden Klasse. Ein solches Objekt verwaltet eine Menge von zu `Entity` kompatibler Objekte.
- ❑ Der Dekorierer bietet im Prinzip nur zwei Methoden `filter` und `forEach` an. Der Aufruf von `filter` erzeugt ein zu `EntityStream`-kompatibles Objekt, dessen `forEach`-Methode nur auf der durch `filter` bestimmten Teilmenge arbeitet.
- ❑ Die Regeln zur Bestimmung der Teilmenge werden der Methode `filter` als Argument übergeben, so dass jeder Aufruf von `filter` zu einem individuellen Ergebnis führen kann.

weiteres Beispiel aus SWT-Starfighter – Dekorierer zum Interface EntityStream

(Fortsetzung)



weiteres Beispiel aus SWT-Starfighter – Dekorierer zum Interface EntityStream

(Fortsetzung)

```
public interface EntityStream<T extends Entity> {
    public void forEach(Consumer<T> action);
    public default EntityStream<T> filter(Predicate<T> filter) {
        return new FilteredEntityStream<>(this, filter);
    }
    public default List<T> createList() {
        List<T> result = new ArrayList<>();
        forEach(e -> result.add(e));
        return result;
    }
    ...
    public static class FilteredEntityStream<...> { ... }
}
```

ermöglicht das
Extrahieren der
betrachteten Entity-Objekte

```
public interface Predicate<T> {
    boolean test(T t);
}
```

weiteres Beispiel aus SWT-Starfighter – Dekorierer zum Interface EntityStream

(Fortsetzung)

```
public interface EntityStream<T extends Entity> {
    ...
    public static class FilteredEntityStream<T extends Entity>
                                implements EntityStream<T> {
        protected final EntityStream<T> baseStream;
        protected Predicate<T> filter;
        public FilteredEntityStream(EntityStream<T> baseStream,
                                    Predicate<T> filter) {
            this.baseStream = baseStream;
            this.filter = filter;
        }
        public void forEach(Consumer<T> action) {
            baseStream.forEach(
                e -> { if (filter.test(e)) { action.accept(e); }
                });
        }
    }
}
```

ermöglicht die default-Implementierung von filter

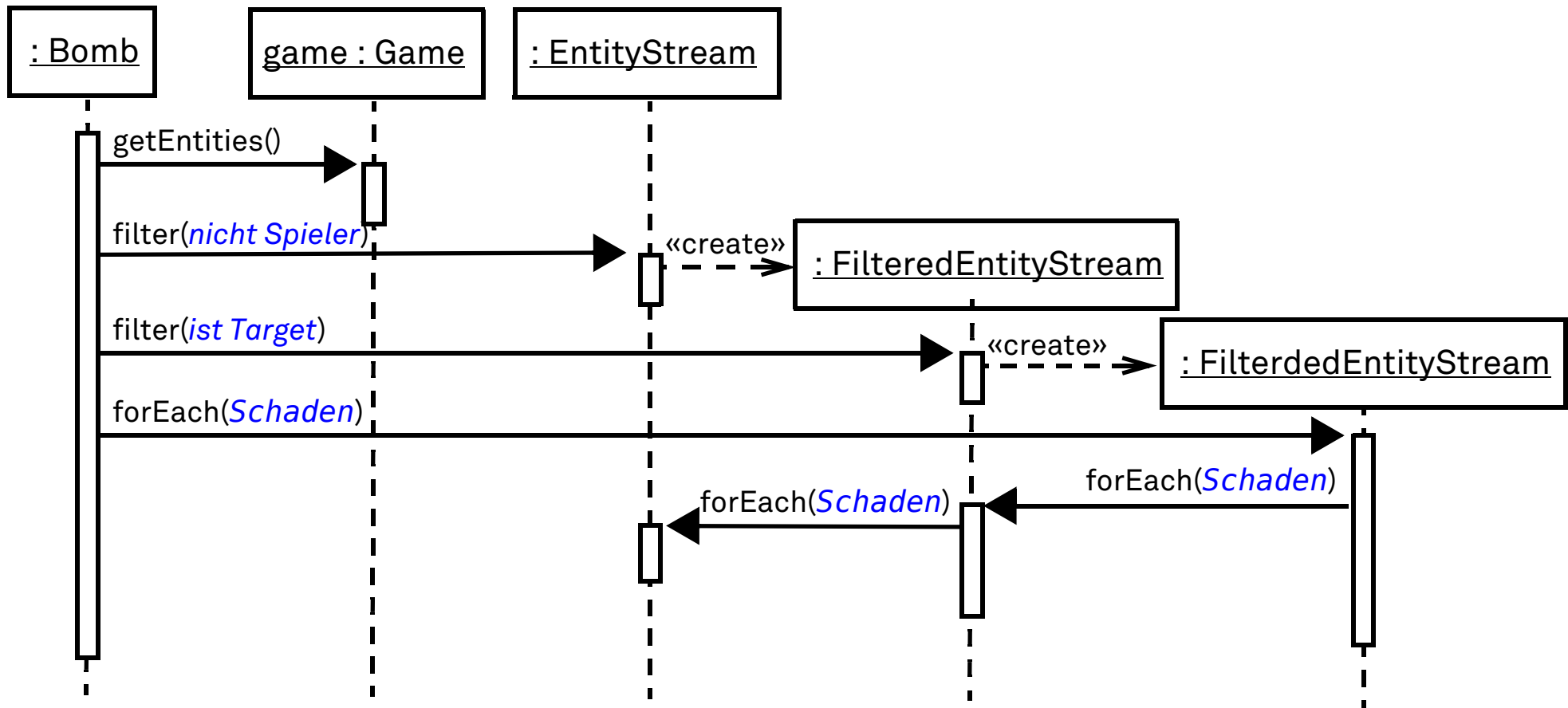
reicht das accept des Arguments weiter an den forEach-Aufruf des baseStream-Objekts, bewacht durch die test-Methode

weiteres Beispiel aus SWT-Starfighter – Dekorierer zum Interface EntityStream

(Fortsetzung)

Anwendung (vereinfachte Darstellung des Konzepts):

```
game.getEntities().filter(nicht Spieler).filter(ist Target).forEach(Schaden)
```



weiteres Beispiel aus SWT-Starfighter – Dekorierer zum Interface EntityStream (Fortsetzung)

- Die tatsächliche Implementierung von EntityStream enthält weitere Methoden, um einfachere Dekorierungen zu ermöglichen:

filter(...), forEach(...), without(...), ofType(...),
withinRadiusOfEntity(...), withinRadiusOfPoint(...)

Die Implementierungen der weiteren Methoden basieren aber alle auf der bekannten Methode filter(...).

- Dann sind einfache Dekorierungen möglich wie zum Beispiel in **public class Bomb implements EntityBehaviorStrategy**

game.getAllEntities()	alle Entity aus Spiel
.without(host)	ohne Spieler (host)
.ofType(Target.class)	nur Ziele (Target)
.filter(e -> e.isAlive())	noch im Spiel
.withinRadiusOfEntity(host, RADIUS)	im Umkreis RADIUS
.forEach(e -> e.addHitpoints(-DAMAGE));	verschlechtern um DAMAGE

Zusammenfassung – Entwurfsmuster Dekorier

Vorteile:

- ❑ Atome und Dekore werden einheitlich behandelt.
- ❑ Es sind mehrere Arten von Atomen oder Dekoren möglich.
- ❑ Neue Atom- oder Dekor-Klassen können leicht ergänzt werden.
- ❑ Es wird immer ein Atom mit weiteren Dekoren verknüpft.
Ein Umwandeln oder Ändern schon vorhandener Objekte ist nicht notwendig.
- ❑ Es ist keine Überprüfung des Typs einer Komponente notwendig.
- ❑ Die aufgebaute Objektstruktur ist unbegrenzt.
- ❑ Es entsteht ein Stapel, der aus spezialisierten, heterogenen Knoten aufgebaut ist.

Analyse – Entwurfsmuster Dekorierer

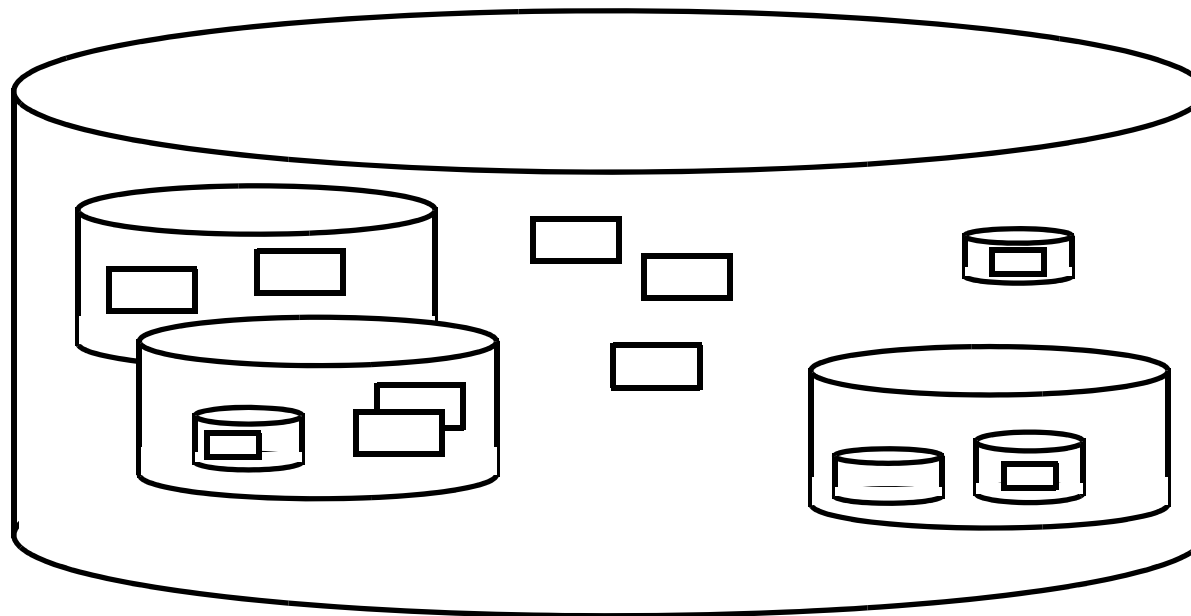
- ❑ Die grundlegende Idee des Dekorierer ist es, einen Stapel/eine Liste aus Objekten unterschiedlicher Klassen aufzubauen, die nach außen die gleiche Schnittstelle anbieten, also gleich behandelt werden können, aber unterschiedliche Implementierungen von Methoden besitzen (können).
- ❑ Diese Idee lässt sich auch auf baumartige Datenstrukturen übertragen: Die Knoten eines Baums werden durch Objekte unterschiedlicher Klassen gebildet, die nach außen die gleiche Schnittstelle anbieten.

Das zugehörige Entwurfsmuster wird als **Kompositum** bezeichnet.

Entwurfsmuster Kompositum

Ein **Kompositum**
erlaubt das Anlegen von baumartigen Strukturen aus heterogenen Komponenten.

Beispiel: Dateistruktur eines Betriebssystems



Komponenten:

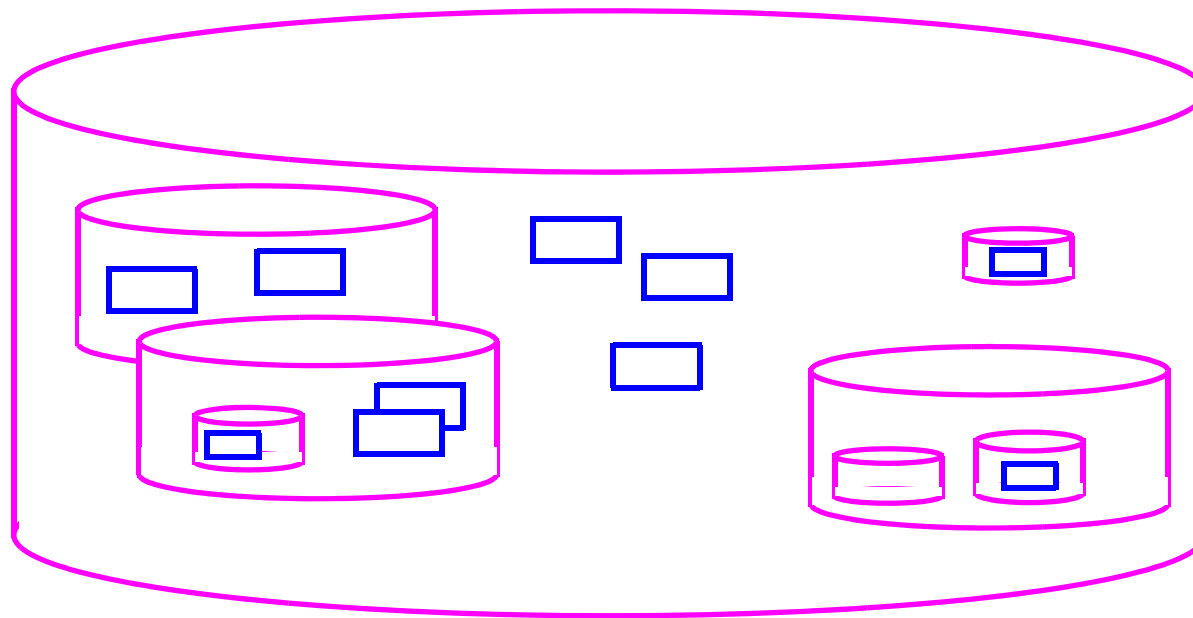
 File

 Directory

Entwurfsmuster Kompositum

Ein **Kompositum**
erlaubt das Anlegen von baumartigen Strukturen aus heterogenen Komponenten.

Beispiel: Dateistruktur eines Betriebssystems



Komponenten:

File
Atom

Directory
Komposition

Analyse der Dateistruktur

- ❑ Es gibt zwei Arten von Komponenten:
 - Atome, die keine weiteren Komponenten enthalten können.
 - Kompositionen, die wiederum Komponenten enthalten können.
- ❑ Komponente ist der Oberbegriff für Atom oder Komposition.
- ❑ Die Struktur baut sich baumartig (= rekursiv) aus (beiden Arten von) Komponenten auf.
- ❑ Die Atome bilden die Blätter der Baumstruktur.
- ❑ Die Kompositionen bilden innere Knoten oder Blätter der Baumstruktur.

Literatur: Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.224-230

http://link.springer.com/chapter/10.1007/978-3-8348-9174-7_8

Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 205-210

http://link.springer.com/chapter/10.1007/3-540-30950-0_12

Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 46-48

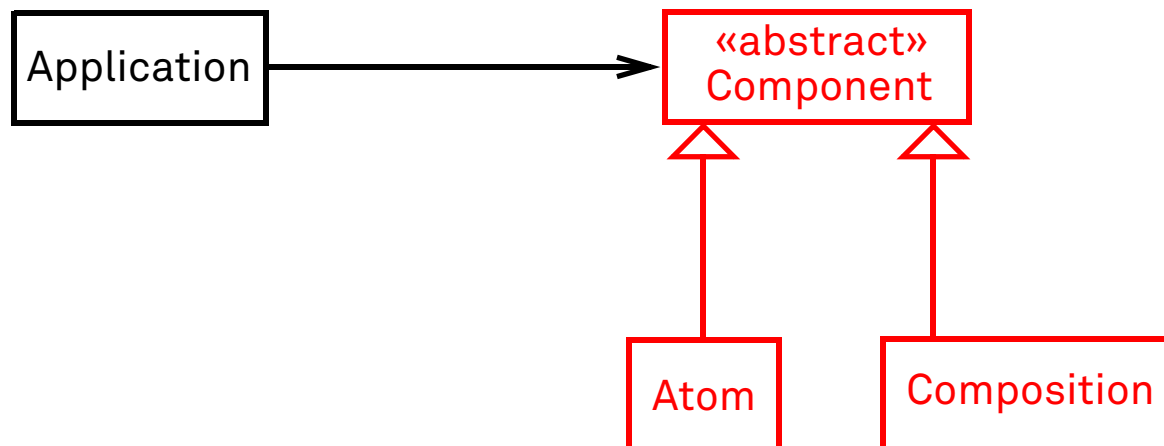
http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_5

Analyse der Dateistruktur

(Fortsetzung)

- ❑ Es gibt zwei Arten von Komponenten:
 - Atome, die keine weiteren Komponenten enthalten können.
 - Kompositionen, die wiederum Komponenten enthalten können.
- ❑ Komponente ist der Oberbegriff für Atom oder Komposition.
- ❑ Die Struktur baut sich baumartig (= rekursiv) aus (beiden Arten von) Komponenten auf.

Modellierung als Klassendiagramm:

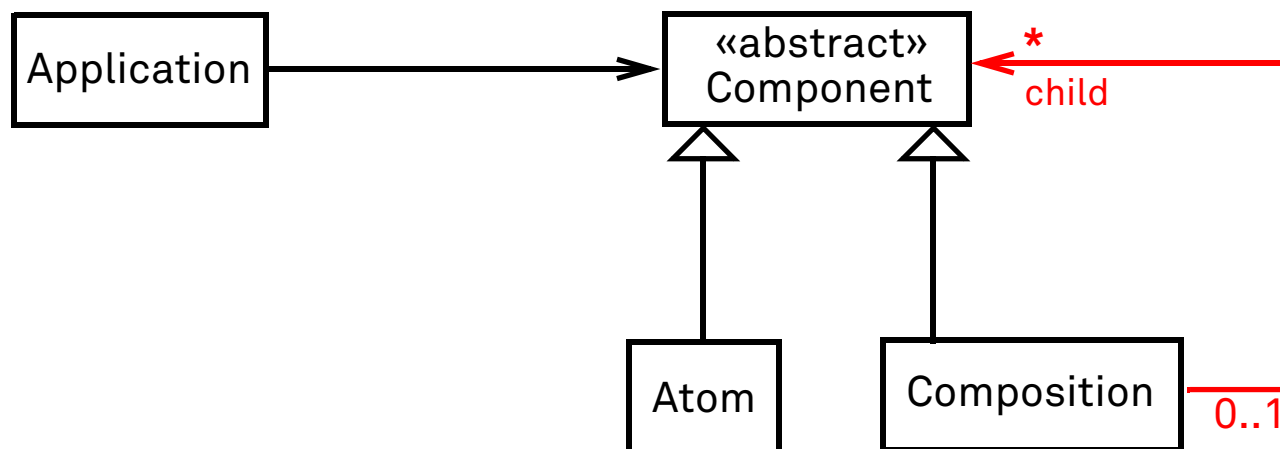


Analyse der Dateistruktur

(Fortsetzung)

- ❑ Es gibt zwei Arten von Komponenten:
 - Atome, die keine weiteren Komponenten enthalten können.
 - Kompositionen, die wiederum Komponenten enthalten können.
- ❑ Komponente ist der Oberbegriff für Atom oder Komposition.
- ❑ Die Struktur baut sich baumartig (= rekursiv) aus (beiden Arten von) Komponenten auf.

Modellierung als Klassendiagramm:

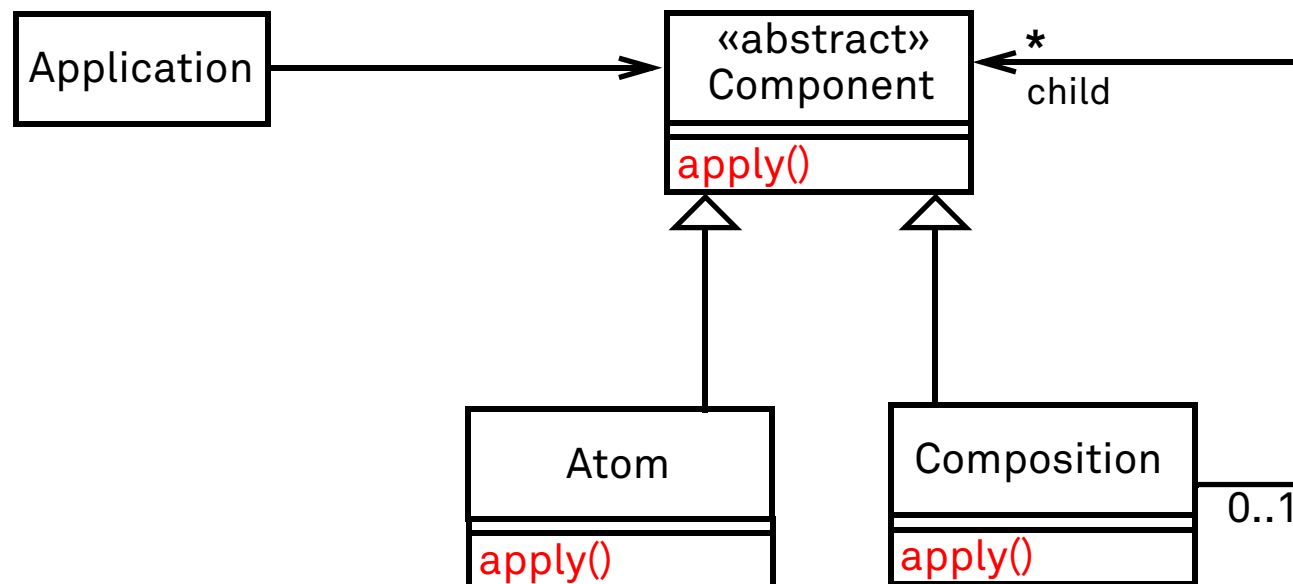


Die Wurzel hat keinen
Vorgänger

Analyse der Dateistruktur

(Fortsetzung)

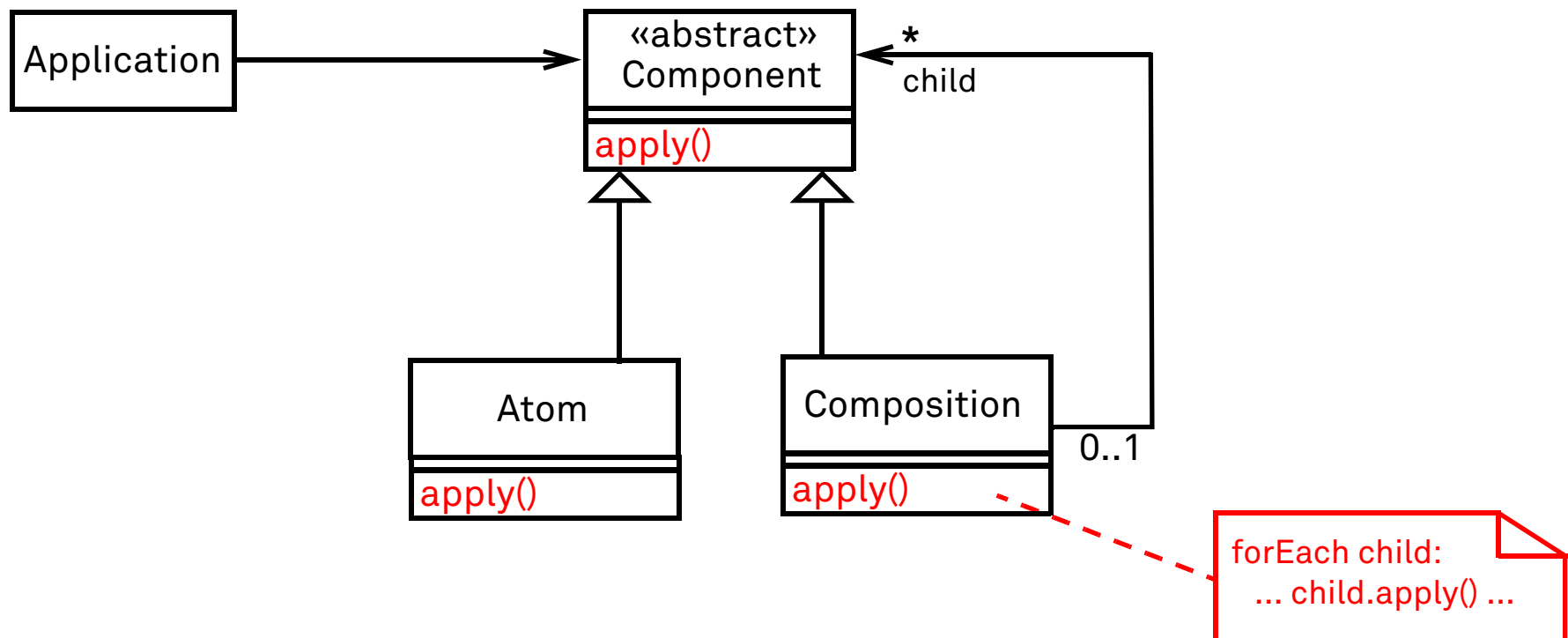
- ❑ Die abstrakte Komponente muss die Methoden anbieten, die für
 - Atome und
 - Kompositionen aufgerufen werden können.
- ❑ Methodenaufrufe auf Kompositionen müssen auf die Kinder übertragen werden.



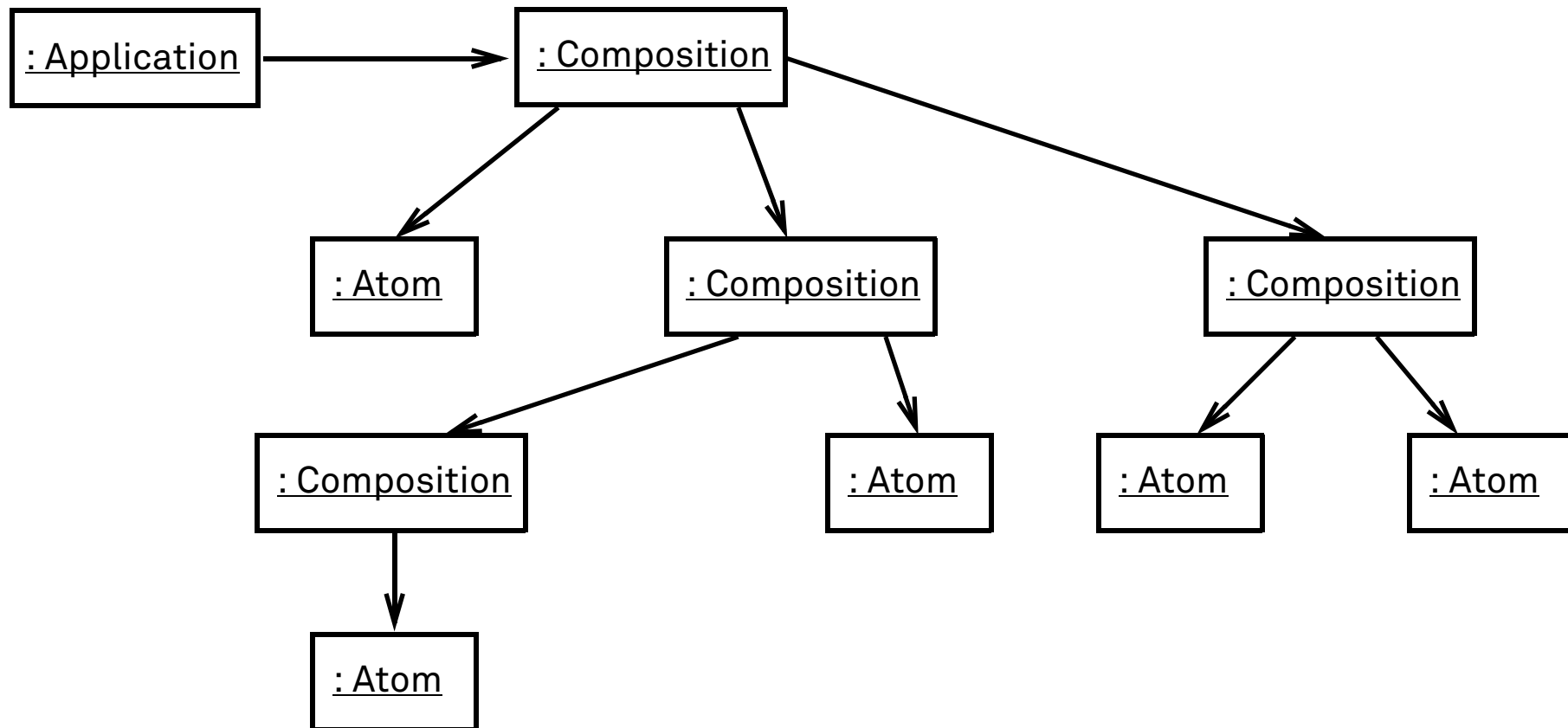
Analyse der Dateistruktur

(Fortsetzung)

- ❑ Die abstrakte Komponente muss die Methoden anbieten, die für
 - Atome und
 - Kompositionen aufgerufen werden können.
- ❑ Methodenaufrufe auf Kompositionen müssen auf die Kinder übertragen werden.

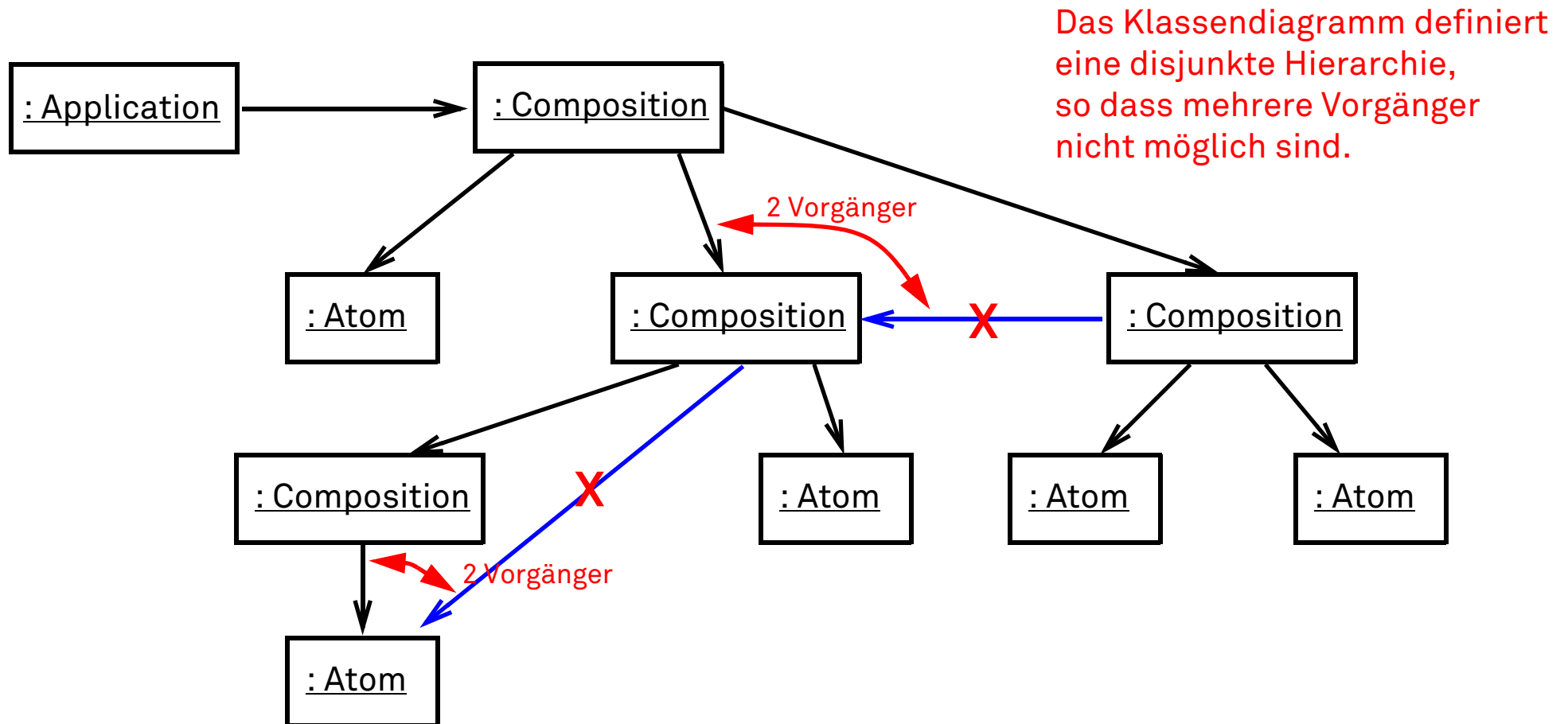


Beispiel für ein zugehöriges Objektdiagramm

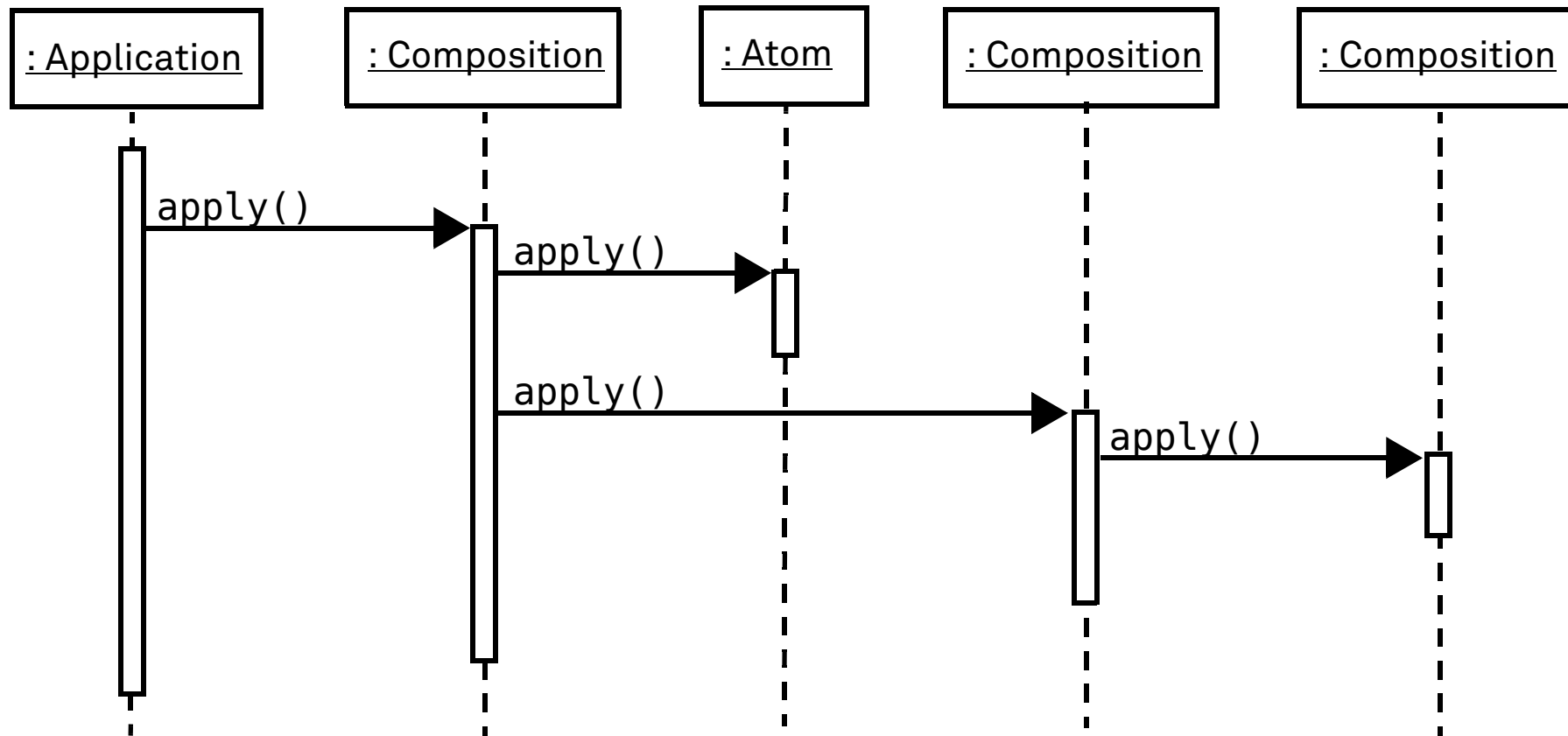


Beispiel für ein Objektdiagramm

(Fortsetzung)



Beispiel für Verhalten



Beispiel für eine Implementierung des Musters Kompositum

```
public abstract class Component {  
    protected String content;  
    public abstract String get();  
    public abstract void add(Component c);  
}
```

Beispiel für ein Attribut

Beispiele für Methoden

Beispiel für eine Implementierung des Musters Kompositum

(Fortsetzung)

```
public abstract class Component {  
    protected String content;  
    public abstract String get();  
    public abstract void add(Component c);  
}  
  
public class Atom extends Component {  
    public Atom (String s) { content = s; }  
    public String get() { return content; }  
    public void add(Component c) {};  
}
```

Beispiel für eine Implementierung des Musters Kompositum

(Fortsetzung)

```
public abstract class Component {
    protected String content;
    public abstract String get();
    public abstract void add(Component c);
}

public class Atom extends Component {
    public Atom (String s) { content = s; }
    public String get() { return content; }
    public void add(Component c) {};
}

public class Composition extends Component {
    private ArrayList<Component> children = new ArrayList<Component>();
    public Composition(String s) { content = s; }
    public String get() {
        String all = content;
        for (Component c: children) { all += c.get(); }
        return all;
    }
    public void add(Component c){ children.add(c); }
}
```

Attribut, um Kinder zu verwalten
(realisiert * aus Diagramm)

Beispiel für eine Implementierung des Musters Kompositum

(Fortsetzung)

```

public abstract class Component {
    protected String content;
    public abstract String get();
    public abstract void add(Component c);
}

public class Atom extends Component {
    public Atom (String s) { content = s; }
    public String get() { return content; }
    public void add(Component c) {};
}

public class Composition extends Component {
    private ArrayList<Component> children = new ArrayList<Component>();
    public Composition(String s) { content = s; }
    public String get() {
        String all = content;
        for (Component c: children) { all += c.get(); }
        return all;
    }
    public void add(Component c){ children.add(c); }
}

```

Methode ist für beide Unterklassen sinnvoll

delegiert Aufruf an Kinder

Beispiel für eine Implementierung des Musters Kompositum

(Fortsetzung)

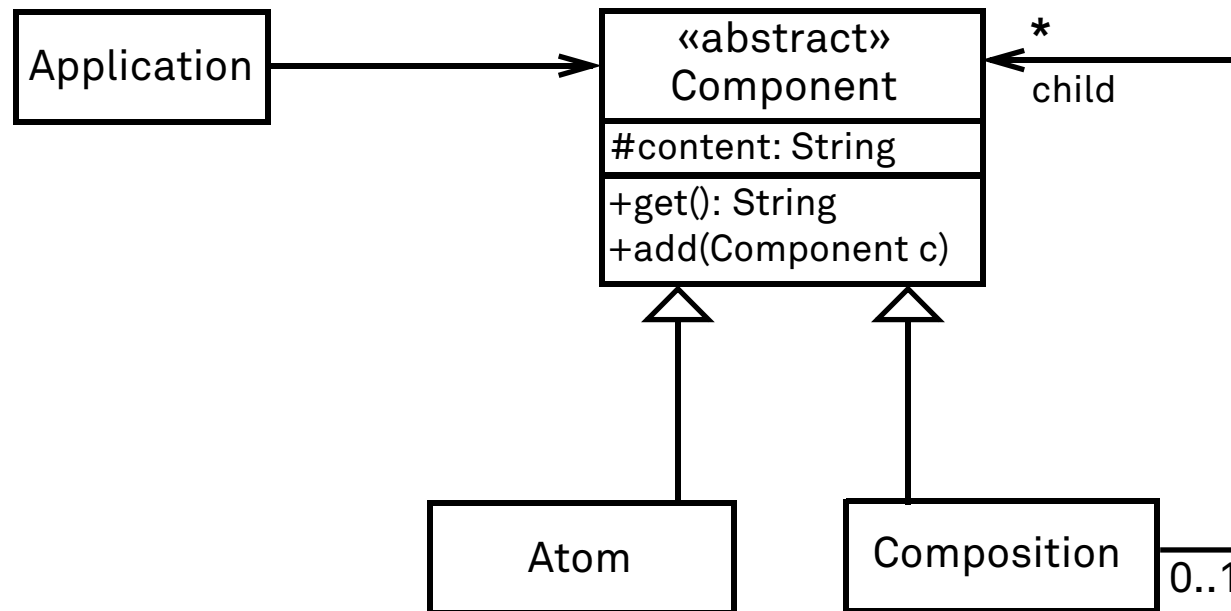
```
public abstract class Component {
    protected String content;
    public abstract String get();
    public abstract void add(Component c);
}

public class Atom extends Component {
    public Atom (String s) { content = s; }
    public String get() { return content; }
    public void add(Component c) {};
}

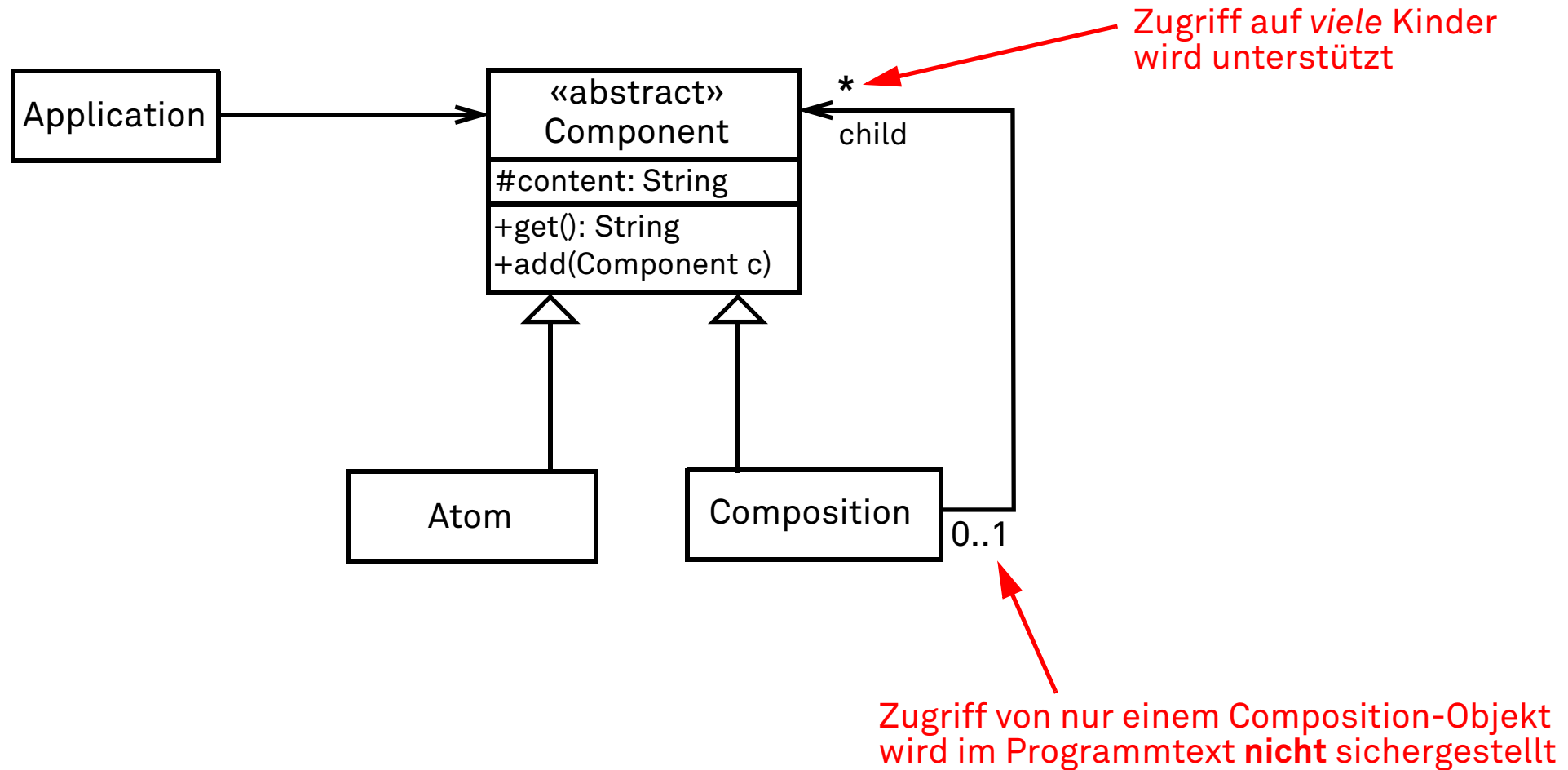
public class Composition extends Component {
    private ArrayList<Component> children = new ArrayList<Component>();
    public Composition(String s) { content = s; }
    public String get() {
        String all = content;
        for (Component c: children) { all += c.get(); }
        return all;
    }
    public void add(Component c){ children.add(c); }
}
```

Methode ist nur für eine Unterklasse sinnvoll

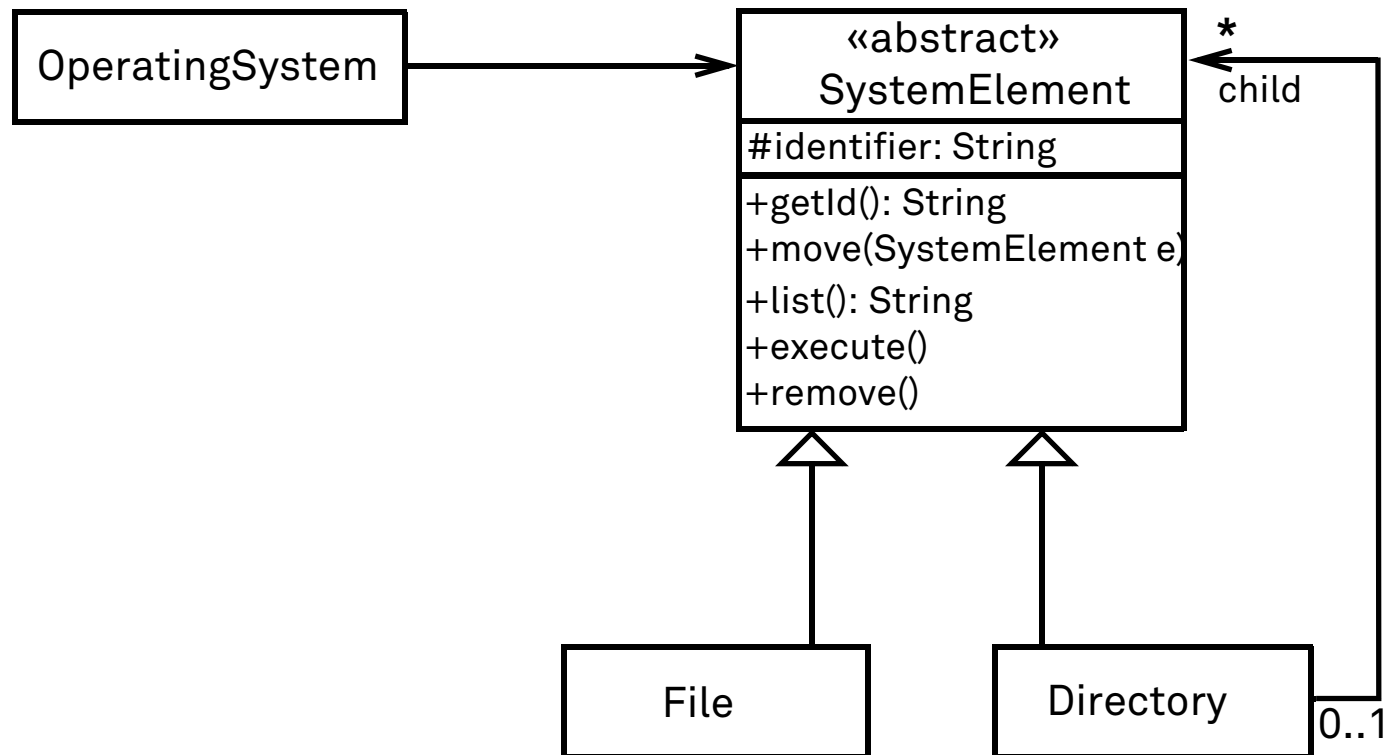
Klassendiagramm für die Beispiel-Implementierung



Anmerkungen zur Implementierung

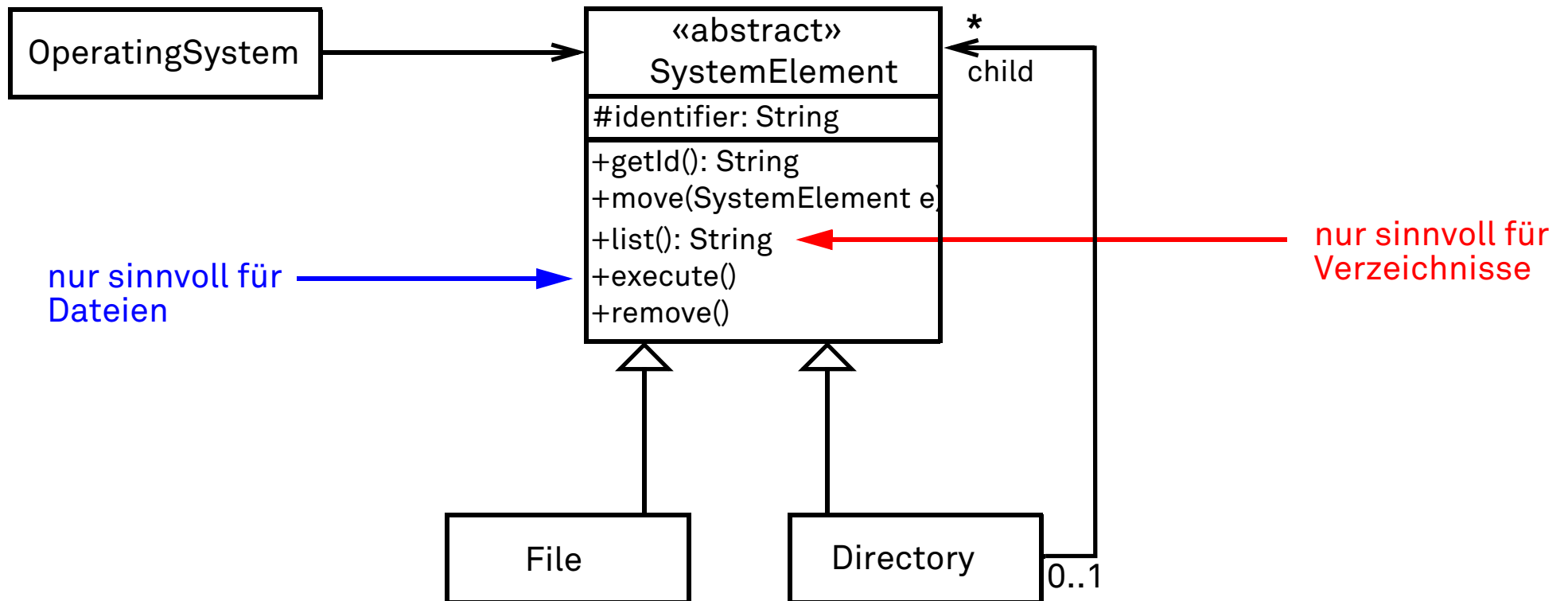


Klassendiagramm für ein Dateisystem (Beispiel)



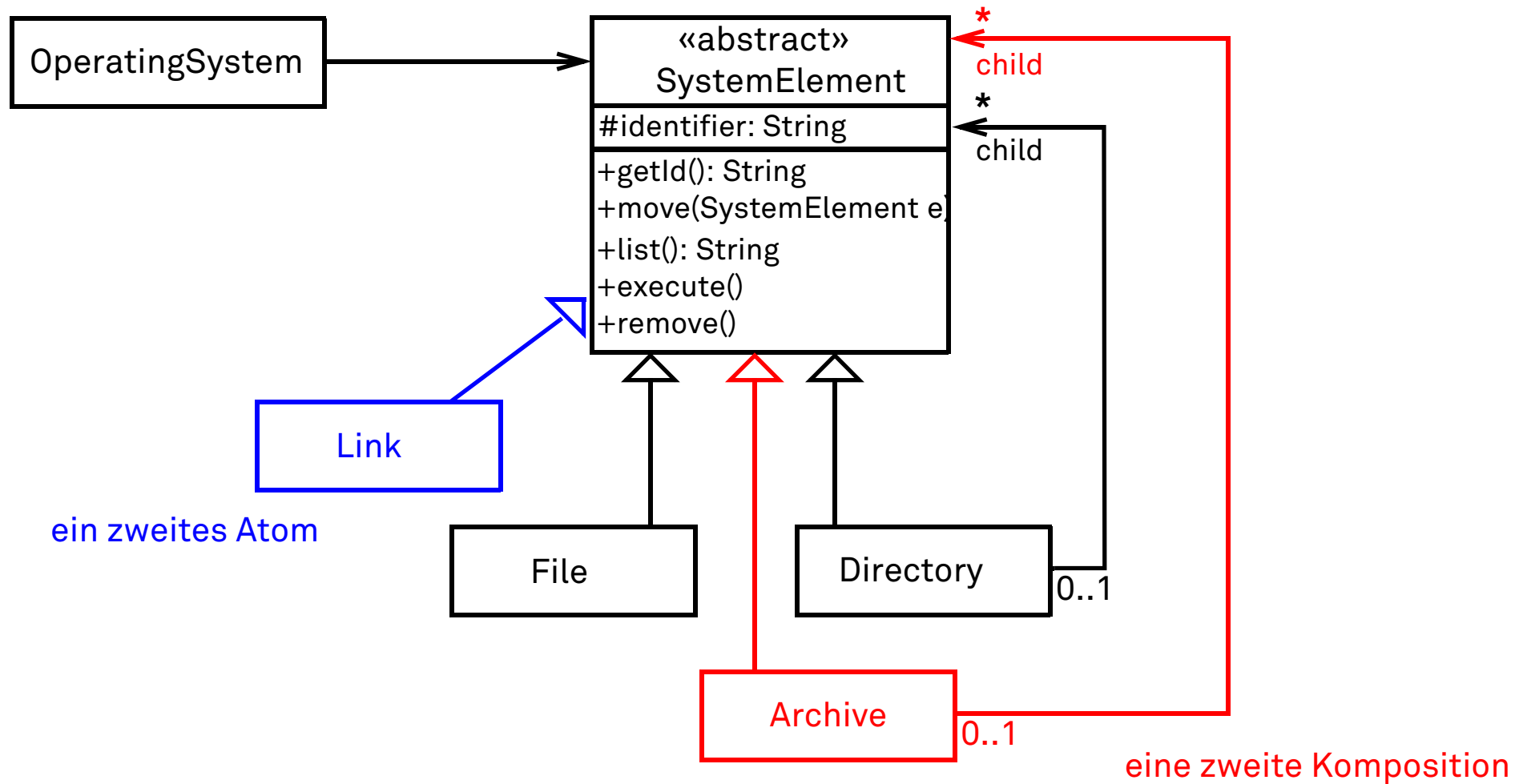
Klassendiagramm für ein Dateisystem (Beispiel)

(Fortsetzung)



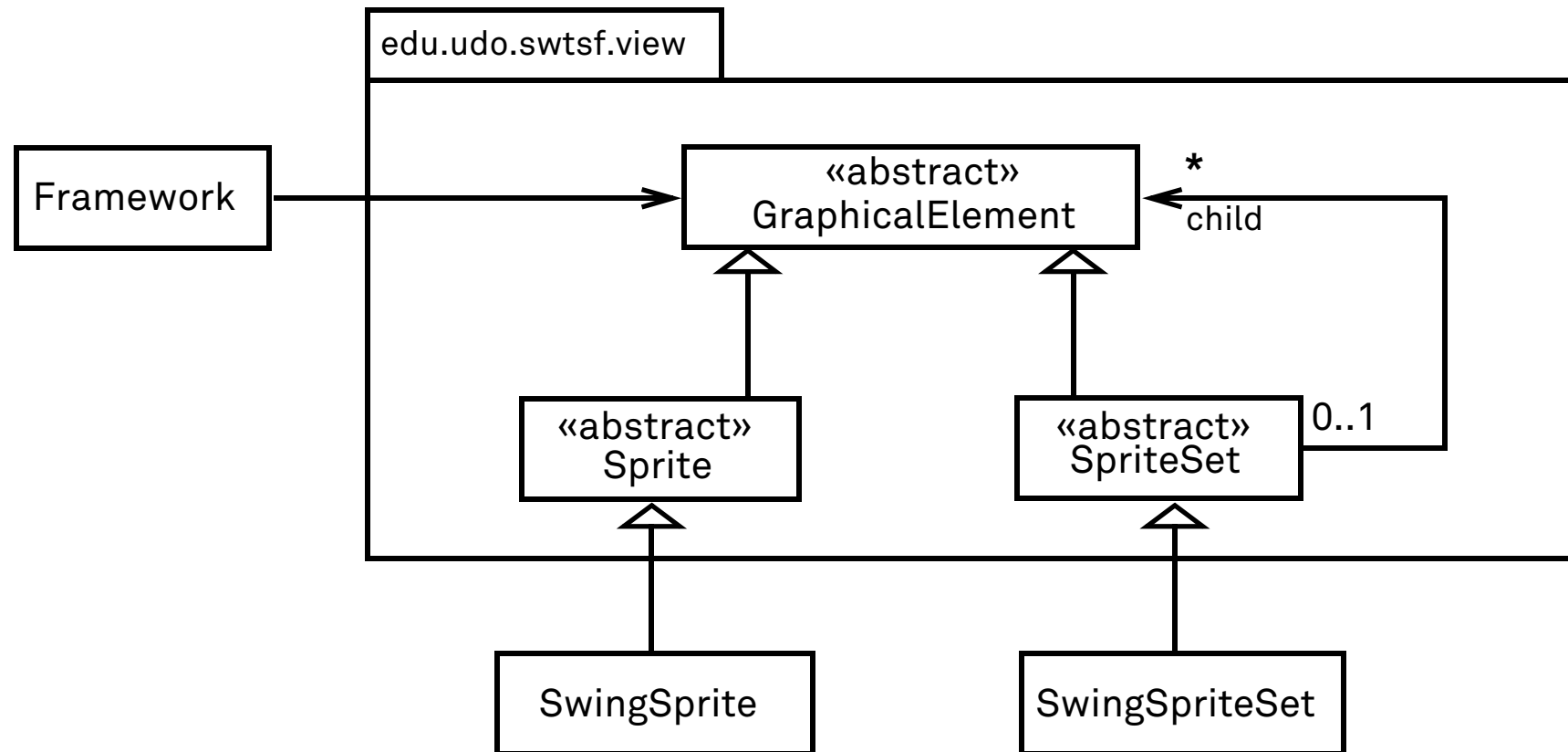
Klassendiagramm für ein Dateisystem (Beispiel)

(Fortsetzung)



Beispiel aus SWT-Starfighter – Kompositum zur Klasse GraphicalElement

Das Kompositum wird verwendet, um einen Szenegraph umzusetzen.



Beispiel aus *SWT-Starfighter* – Kompositum zur Klasse `GraphicalElement`

(Fortsetzung)

Das Kompositum wird verwendet, um einen Szenegraph umzusetzen.

- ❑ Im Szenegraph werden die im Spiel benutzten grafischen Elemente zu lokalen Einheiten zusammengefasst, die auf der visuellen Ebene gemeinsam behandelt werden, also beispielsweise gemeinsam erzeugt, vernichtet oder bewegt werden.
- ❑ Der Szenegraph wird als Baum modelliert, in dem ein Teilbaum eine lokale Einheit, also den Szenegraph eines Teils der Darstellung, repräsentiert. Ein Methodenaufruf für die Wurzel eines Teilbaums führt zu entsprechenden Änderungen auf allen Knoten des Teilbaums.
- ❑ Die Klasse `GraphicalElement` enthält daher Methoden, die eine Änderung der Orientierung innerhalb ihres lokalen Szenegraphs bewirken:
 `setTranslation`, `getTranslateX`, `getTranslateY`, `setScale`, `getScale`,
 `setRotation`, `getRotation`
- ❑ Die Klasse `Sprite` enthält zusätzlich Methoden, die die Visualisierung betreffen:
 `setImagePath`, `getImagePath`, `setImageCutout`, `setImageCutoutX`,
 `getImageCutoutX`, `setImageCutoutY`
- ❑ Die Klasse `SpriteSet` enthält zusätzlich Methoden, die den Aufbau des Szenegraph betreffen: `add`, `remove`, `getChildren`

Zusammenfassung – Entwurfsmuster *Kompositum*

Vorteile:

- ❑ Atome und Kompositionen werden einheitlich behandelt.
- ❑ Es sind mehrere Arten von Atomen oder mehrere Arten von Kompositionen möglich.
- ❑ Weitere Atome oder Kompositionen können leicht ergänzt werden.
- ❑ Es ist keine Überprüfung des Typs einer Komponente notwendig.
- ❑ Die aufgebaute Objektstruktur ist unbegrenzt.
- ❑ Es entsteht ein Baum, der aus **spezialisierten, heterogenen** Knoten aufgebaut ist.

Nachteile:

- ❑ Die gemeinsame Schnittstelle für Atome und Kompositionen führt zu Methoden, die nicht auf allen Objekte sinnvolle Aktionen auslösen.
- ❑ Die Struktur kann **zu** allgemein werden, da Kompositionen nur schwer beschränkt werden können:
 - Zahl der Kinder
 - Art der Kinder
 - disjunkte Struktur

Entwurfsmuster Besucher

(Kurzpräsentation)

Idee von Kompositum und Dekorierer:

Alle an der Struktur beteiligten Klassen implementieren die gleiche Schnittstelle mit klassenspezifisch deklarierter Funktionalität.

Bei der Ausführung können alle Objekte gleich "behandelt" werden.

Folge: Bei vielen beteiligten Klassen entsteht eine komplexe gemeinsame Schnittstelle, die von allen Klassen umgesetzt werden muss.

Idee des Entwurfsmusters **Besucher**:

- ❑ Komplexe Operationen werden strukturell von den Klassen getrennt, die die Daten enthalten, mit denen die Operationen arbeiten.
- ❑ Dann müssen die Operationen nicht für alle Klassen einzeln implementiert werden. Stattdessen können die Implementierungen einer Operation für verschiedene Klassen zusammengefasst werden.
- ❑ Das Entwurfsmuster Besucher ermöglicht es auch, neue Operationen auf den Elementen einer Struktur zu definieren, ohne die Elemente der Struktur anzupassen.

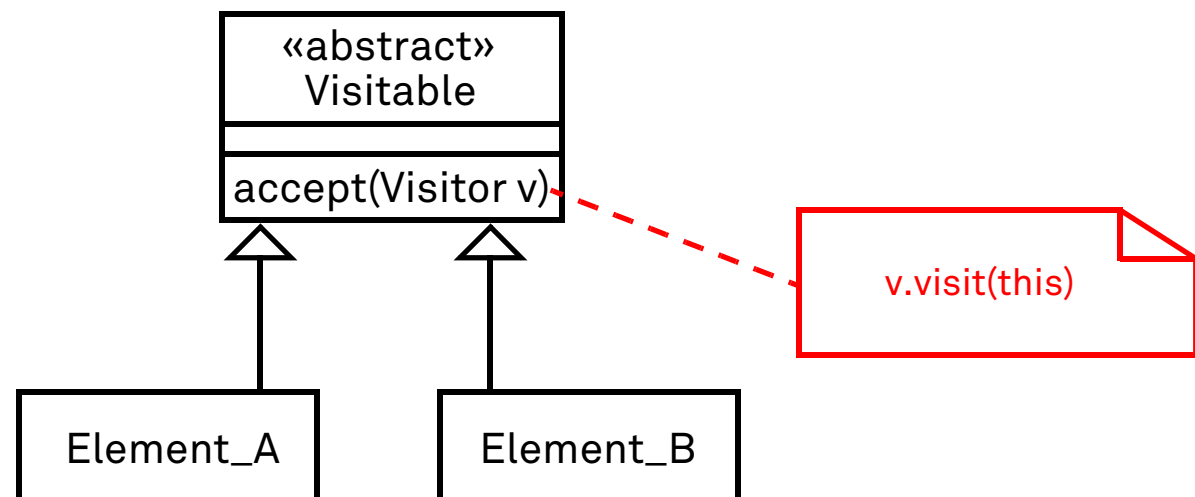
Literatur: Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 52-57
http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_4

Entwurfsmuster Besucher

(Fortsetzung)

- ❑ Soll eine Datenstruktur, die aus Objekten verschiedener Klassen besteht, mit dem Besucher-Muster bearbeitet werden, so müssen alle Elemente der Datenstruktur eine gemeinsame Schnittstelle zum Besuchen bieten. Hier wird als Beispiel die abstrakte Klasse `Visitable` verwendet.
- ❑ Die Klasse `Visitable` besitzt eine Methode, die den Aufruf einer Methode eines Besuchers ermöglicht, im Beispiel:

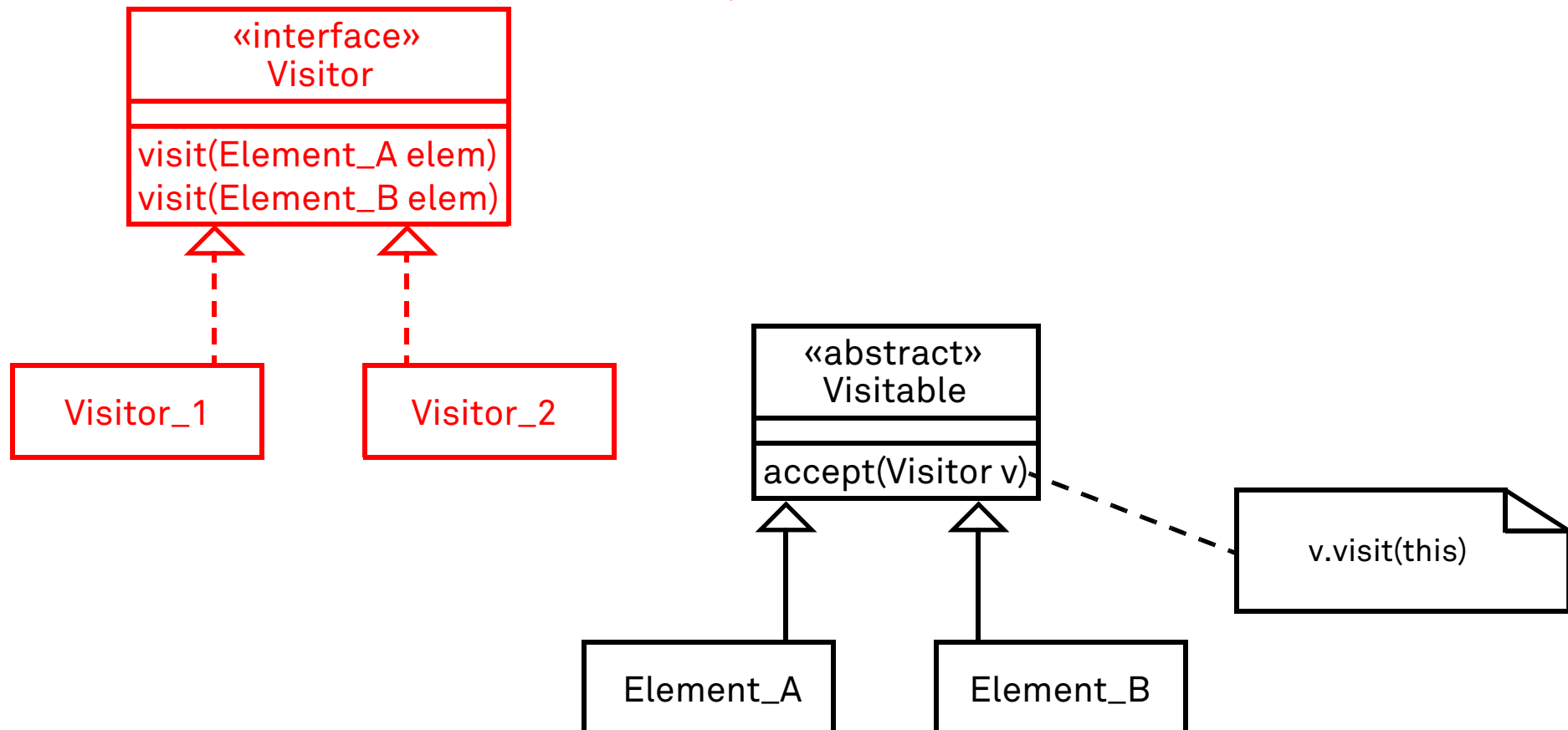
```
void accept(Visitor v) { v.visit(this); }
```



Entwurfsmuster Besucher

(Fortsetzung)

Besucher müssen für jedes Element der Datenstruktur eine passende visit-Methode anbieten.

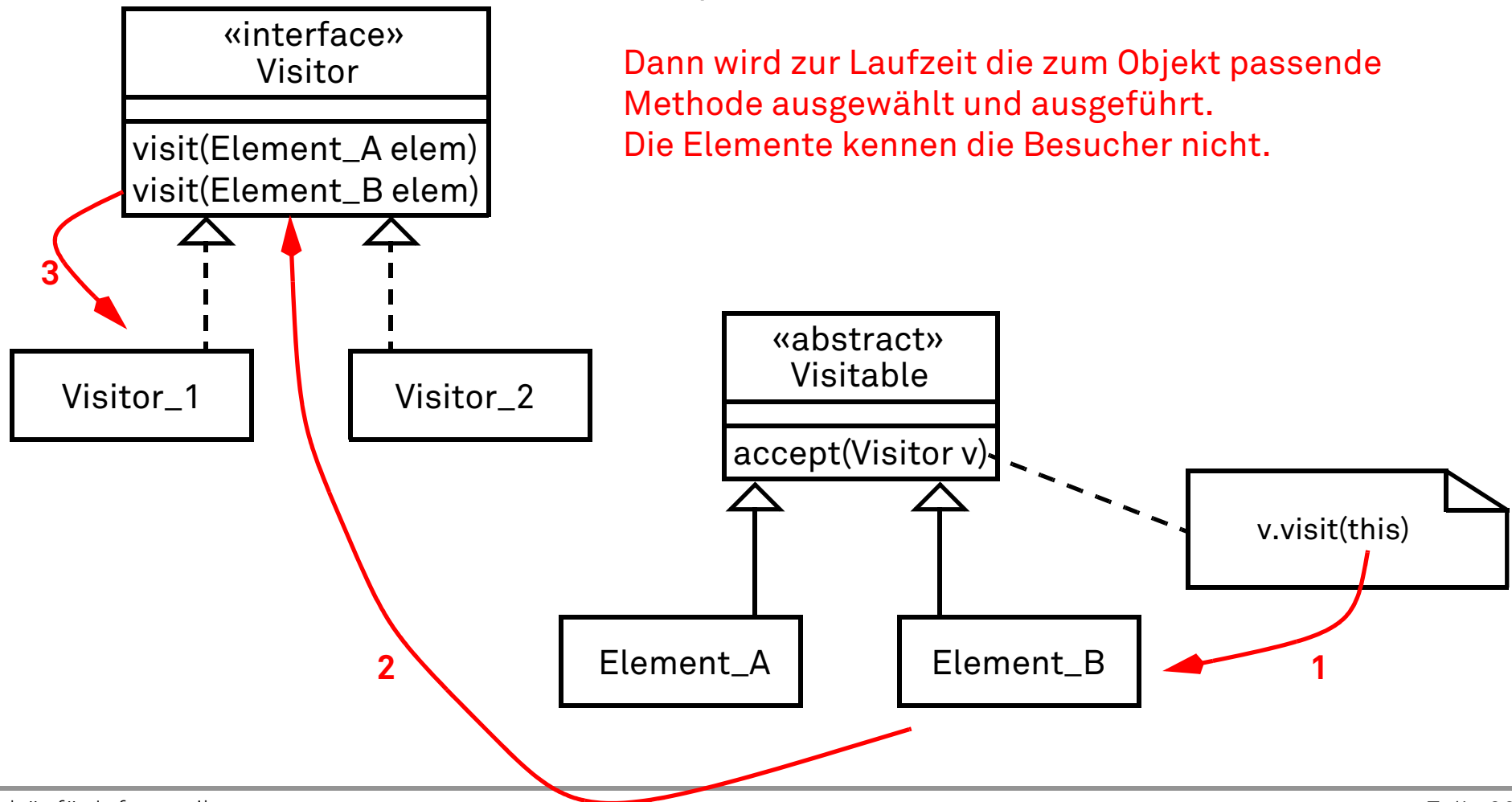


Entwurfsmuster Besucher

(Fortsetzung)

Besucher müssen für jedes Element der Datenstruktur eine passende visit-Methode anbieten.

Dann wird zur Laufzeit die zum Objekt passende Methode ausgewählt und ausgeführt.
Die Elemente kennen die Besucher nicht.



Visitor_1-Objekt *besucht* Element_B-Objekt

Entwurfsmuster *Fassade*

Eine **Fassade**

erlaubt das Verstecken von komplexen Schnittstellen.

Problemstellung aus dem *SWT-Starfighter*-Projekt:

Im Framework muss an verschiedenen Stellen, also in Methoden von verschiedenen Klassen, die Ein- und Ausgabe von Informationen auf dem Spielfeld implementiert werden.

Im SWT-Starfighter ist die graphische Oberfläche mit Hilfe der Klassen der Bibliotheken *awt* und *Swing* implementiert worden.

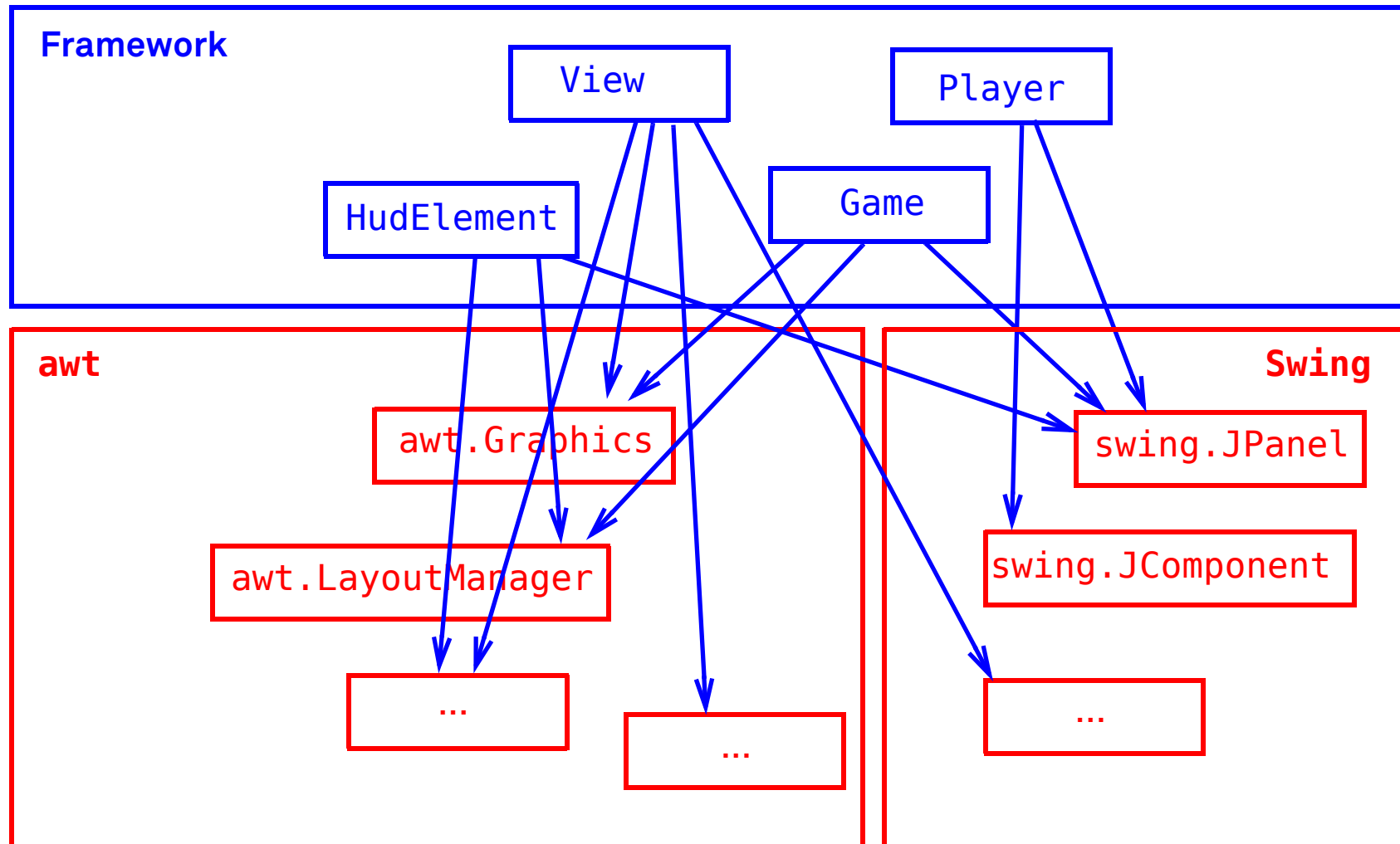
Literatur: Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.219-221

http://link.springer.com/chapter/10.1007/978-3-8348-9174-7_8

Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 73-75

http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_6

Klassenstruktur des Beispiels SWT-Starfighter



Klassenstruktur des Beispiels *SWT-Starfighter*

(Fortsetzung)

Konsequenzen:

- ❑ Eine Klassen aus dem Framework nutzt eventuell mehrere Klassen der Bibliotheken.
- ❑ Alle Entwickler des Frameworks benötigen die Kompetenz, die Bibliothek nutzen zu können.
- ❑ Da alle Teile der graphischen Benutzungsschnittstelle einer Anwendung ein ähnliches Aussehen besitzen sollen, sind Absprachen zwischen den Entwicklern notwendig.
- ❑ Soll das *ähnliche* Aussehen geändert werden, so müssen Änderungen an vielen Stellen der Anwendung vorgenommen werden.
- ❑ Sollen die Grafikbibliotheken durch eine andere Grafikimplementierung ersetzt werden, so müssen Änderungen in vielen Klassen des Frameworks vorgenommen werden.

Klassenstruktur des Beispiels SWT-Starfighter

(Fortsetzung)

Konsequenzen:

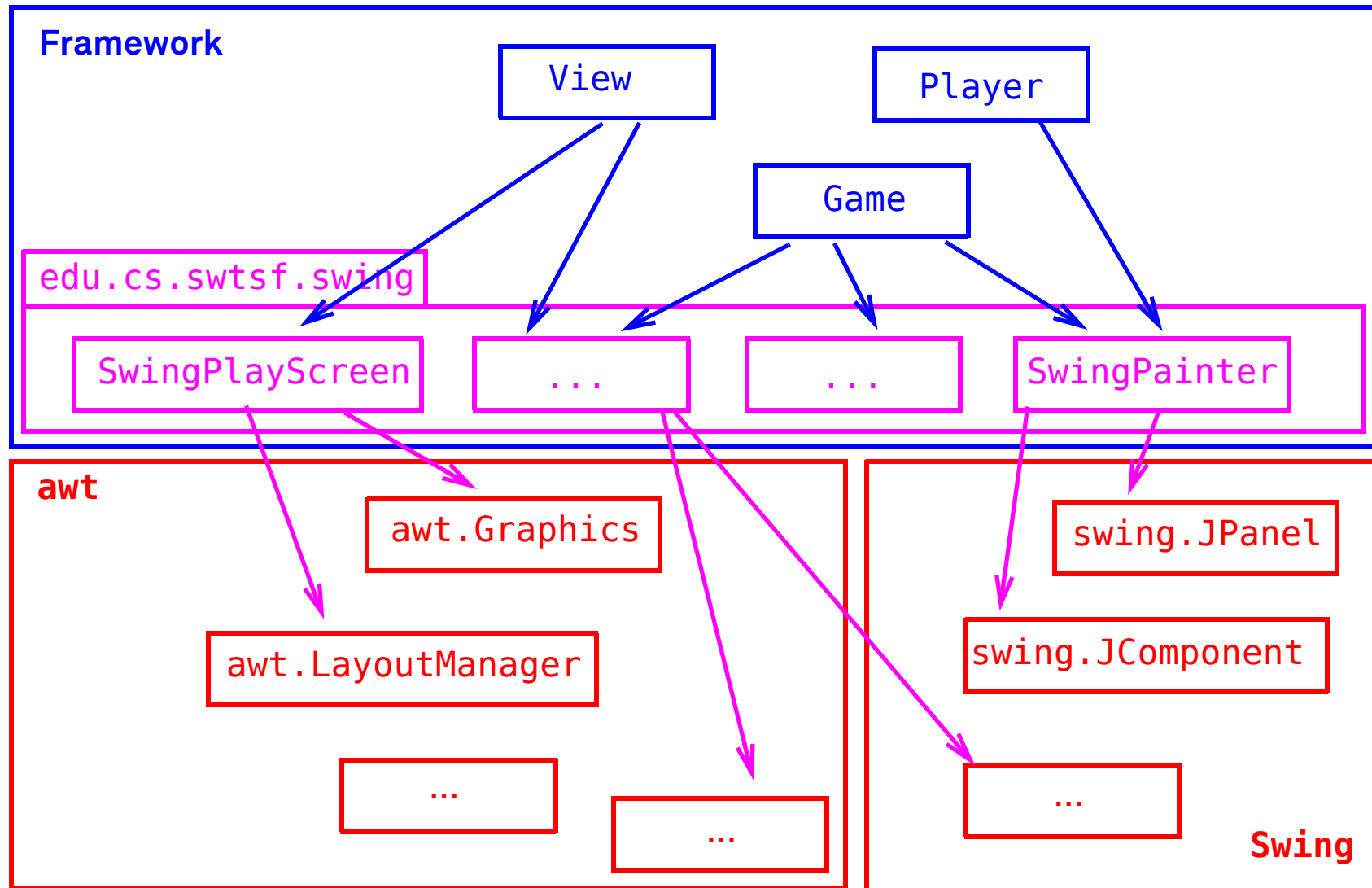
- ❑ Eine Klassen aus dem Framework nutzt eventuell mehrere Klassen der Bibliotheken.
- ❑ Alle Entwickler des Frameworks benötigen die Kompetenz, die Bibliothek nutzen zu können.
- ❑ Da alle Teile der graphischen Benutzungsschnittstelle einer Anwendung ein ähnliches Aussehen besitzen sollen, sind Absprachen zwischen den Entwicklern notwendig.
- ❑ Soll das *ähnliche* Aussehen geändert werden, so müssen Änderungen an vielen Stellen der Anwendung vorgenommen werden.
- ❑ Sollen die Grafikbibliotheken durch eine andere Grafikimplementierung ersetzt werden, so müssen Änderungen in vielen Klassen des Frameworks vorgenommen werden.

Verbesserung:

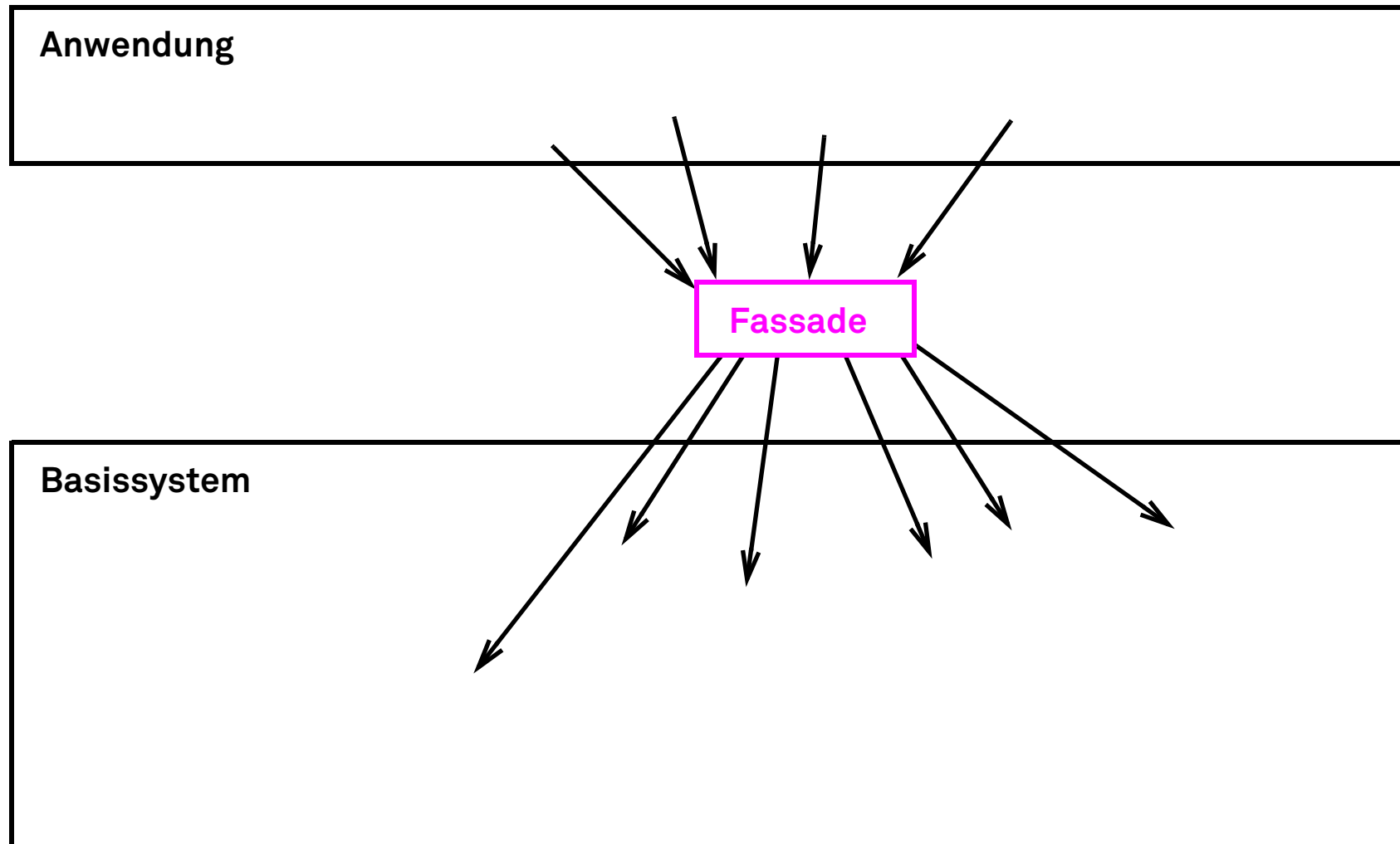
Einführung einer (kompakten) Zwischenschicht, die

- ❑ spezialisierte Methoden bereitstellt, die gezielt für die Klassen des Frameworks bei der Arbeit mit den Grafikbibliotheken unterstützen, und so
- ❑ den Zugriff auf die Bibliotheken vereinfacht und dadurch
- ❑ die Bibliotheken verdeckt (– also eine **Fassade** vor den Bibliotheken aufbaut).

Entwurfsmuster Fassade



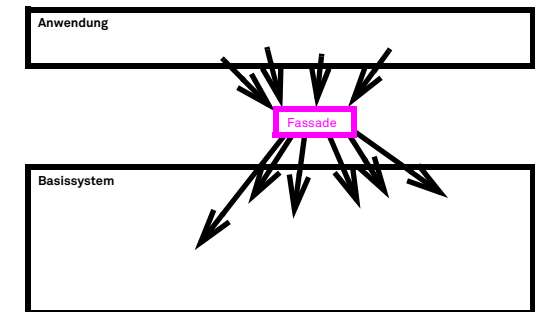
Entwurfsmuster Fassade: allgemeine Darstellung



Bewertung des Entwurfsmusters *Fassade*

Vorteile:

- ❑ Der Zugriff auf das Basissystem wird vereinfacht.
- ❑ Die Anwendung wird vom Basissystem entkoppelt.
- ❑ Für ein Basissystem kann es mehrere Fassaden geben.
- ❑ Die Klassen des Basissystems kennen die Fassade nicht.
- ❑ Die Nutzung des Basissystems ist auch ohne Fassade möglich.
- ❑ Die Bündelung der Aufrufe in der Fassade kann die Performanz verbessern.



Nachteile:

- ❑ Die Struktur wird durch eine zusätzliche Ebene komplexer.
- ❑ Die Performanz kann durch die zusätzliche Aufrufebene schlechter werden.

Vergleich Fassade – Adapter:

- ❑ Eine Fassade schafft in einer komplexen Situation eine zusätzliche, einfacher zu nutzende Einheit aus eventuell mehreren Klassen.
- ❑ Ein Adapter wird durch eine verbindende, einfache Klasse geschaffen.

Entwurfsmuster *Mediator*

(Kurzpräsentation)

Ein **Mediator**

fördert die *lose Kopplung* von Objekten,
indem eine explizite Beziehung zwischen den beteiligten Objekten vermieden wird.

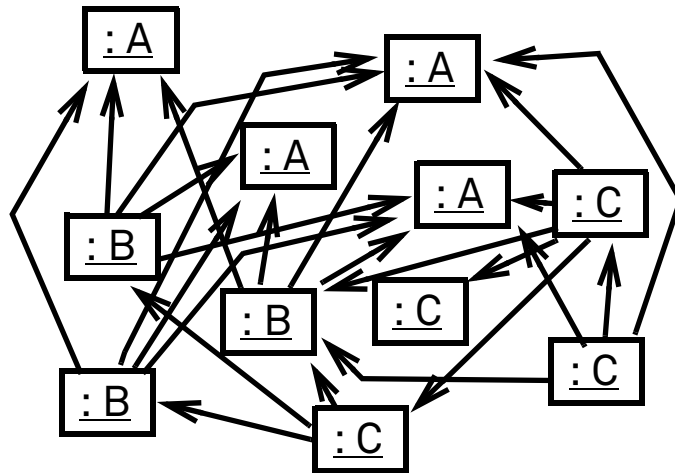
Anmerkungen:

- ❑ Die Verteilung des Verhaltens auf viele Objekte ist normalerweise die Basis für die gute Änderbarkeit und Wiederverwendbarkeit von objektorientierten Systemen.
- ❑ **aber:** zu viele Beziehungen zwischen zu vielen Objekten reduzieren Änderbarkeit und Wiederverwendbarkeit, da das Aufbauen solcher Objektstrukturen komplex ist.

Idee:

- ❑ Ein Objekt operiert als Vermittler zwischen den anderen Objekten.
- ❑ Die Kommunikation zwischen allen beteiligten Objekten läuft immer über den Vermittler.
- ❑ Weitere Objekte können so sehr einfach angebunden werden.

Situation für den Einsatz des Entwurfsmusters Mediator

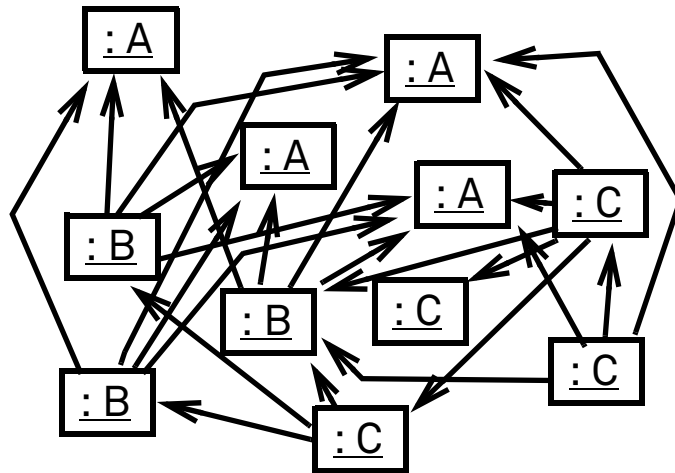


Objektdiagramm

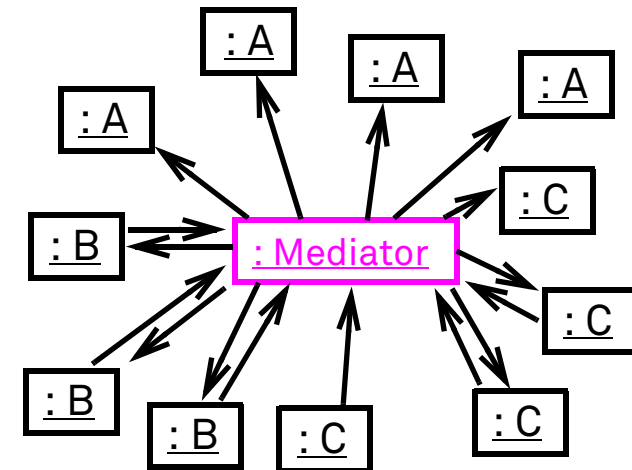
Situation für den Einsatz des Entwurfsmusters Mediator

(Fortsetzung)

Ausgangssituation

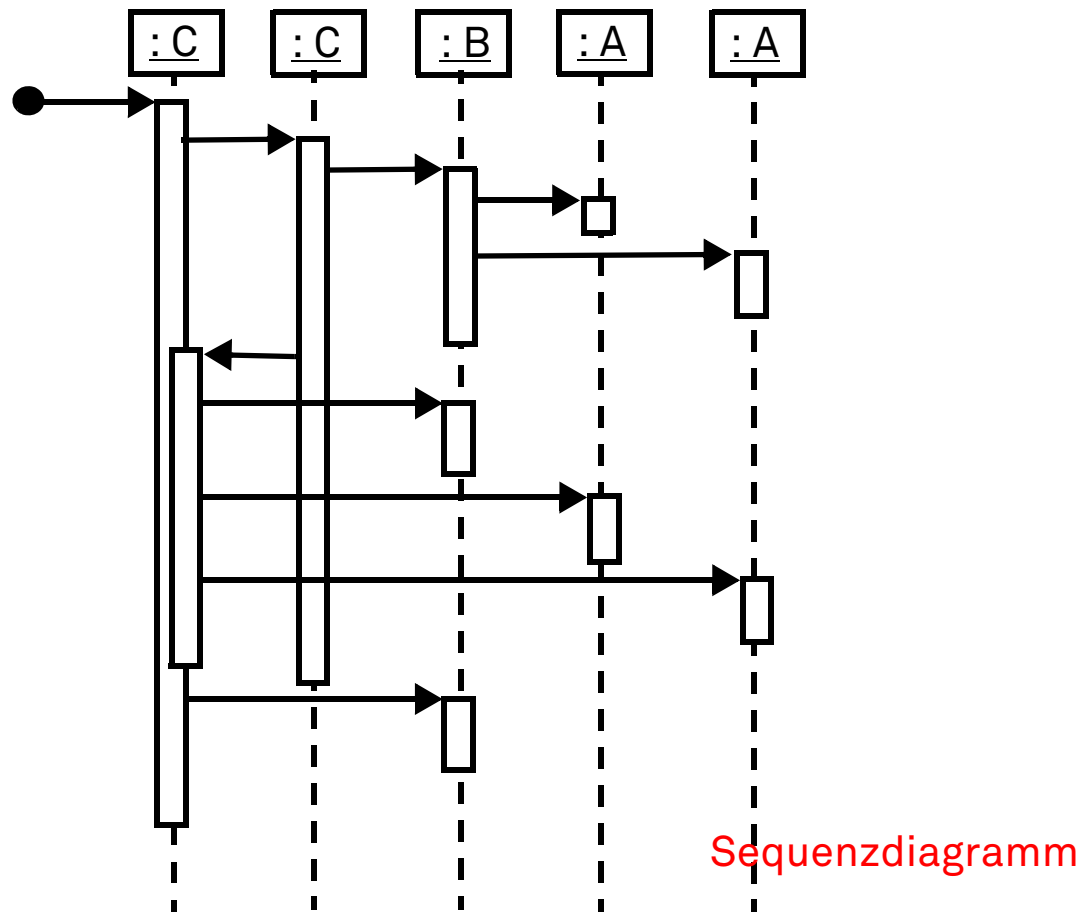


geänderte Struktur durch Einsatz des Mediators

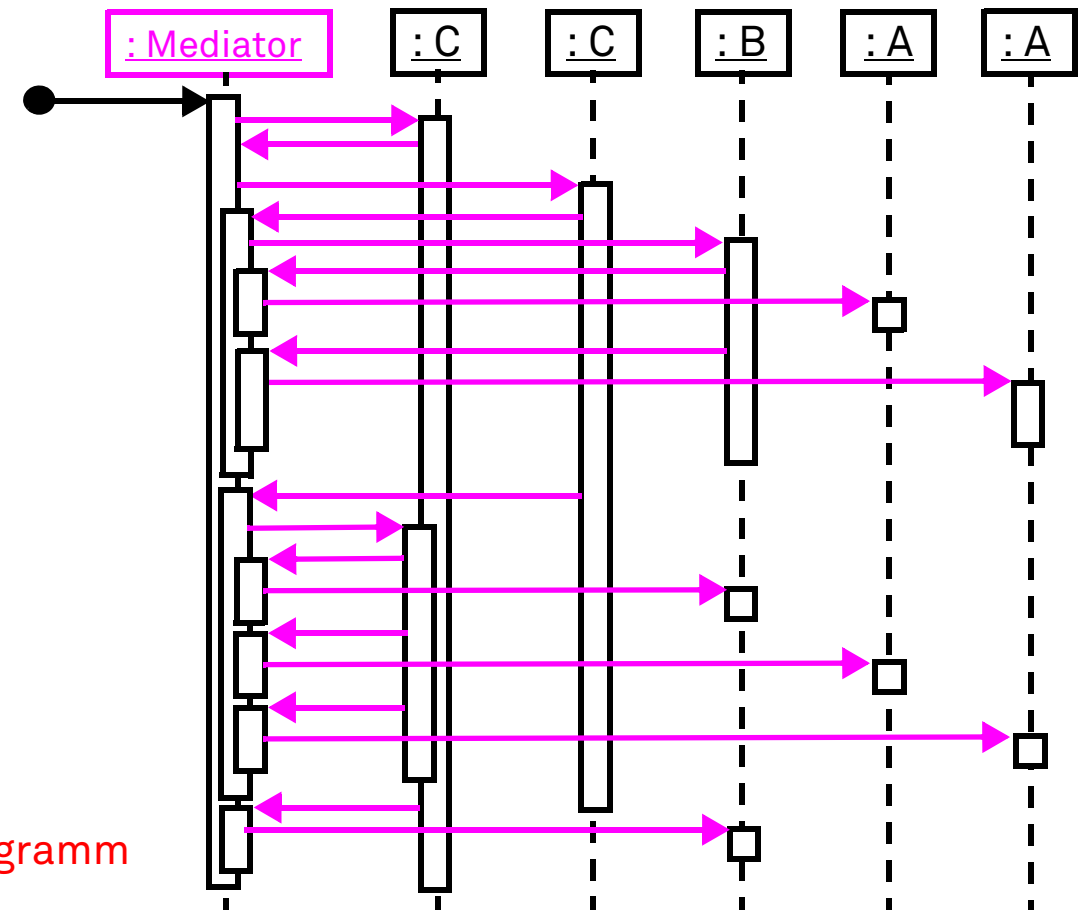


Objektdiagramm

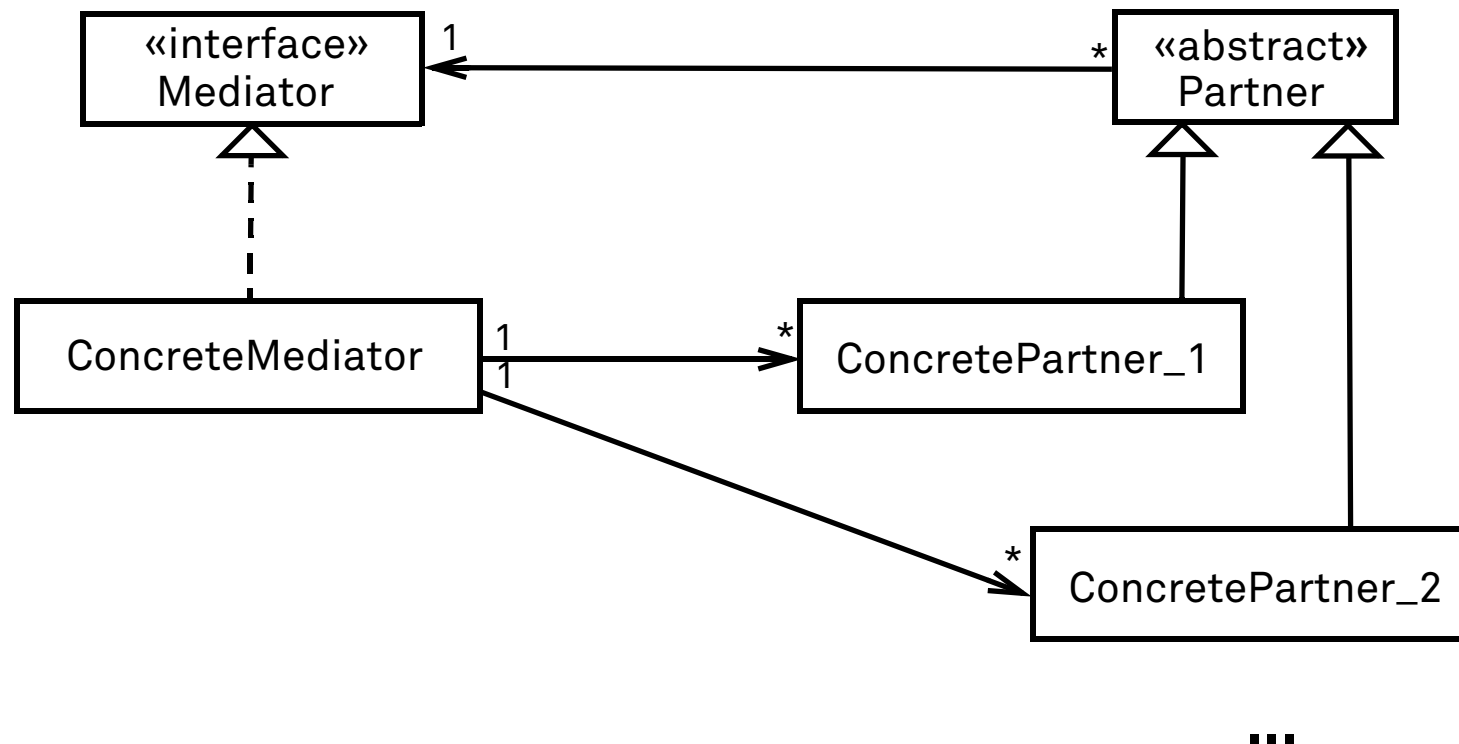
Abläufe des Entwurfsmusters Mediator Ausgangssituation



(Fortsetzung)



allgemeine Struktur des Entwurfsmusters Mediator: Klassendiagramm



Zusammenfassung Entwurfsmuster Mediator

Vorteile:

- ❑ Das Mediator-Muster vereinfacht das Protokoll zwischen den Objekten:
Alle Partner-Objekte rufen ausschließlich Methoden des Vermittlers auf.
- ❑ Das Mediator-Muster abstrahiert von der Zusammenarbeit zwischen den Objekten:
Alle Partner-Objekte kennen nur den Vermittler.
- ❑ Das Mediator-Muster entkoppelt so die Objekte des Systems:
Weitere Objekte und auch weitere Partner-Klassen lassen sich leicht integrieren.

Nachteile:

- ❑ Der Vermittler erfüllt eine zentrale Aufgabe:
Die Komplexität der Interaktion wird ersetzt durch die Komplexität des Vermittlers.

Anmerkung:

- ❑ Vergleich mit dem Fassade-Muster:
 - Eine Fassade bietet eine passende Schnittstelle zur Vereinfachung der Benutzung.
 - Ein Mediator unterstützt ein Protokoll zur Vereinfachung der Zusammenarbeit von Objekten.

Entwurfsmuster *Beobachter*

Ein **Beobachter**

erlaubt das Erkennen (Beobachten) von Änderungen an Objekten.

Beispiele:

- ❑ Eintreffen eines neuen Auftrags
- ❑ Anmelden eines neuen Benutzers
- ❑ Auftreten eines neuen Monsters (*SWT-Starfighter*)

- ❑ In der realen Welt ist Beobachten eine aktive Tätigkeit durch die Beobachter.
- ❑ Viele Beobachter können die gleiche Änderung unmittelbar gleichzeitig bemerken.
- ❑ Dann sind aber **alle** Beobachter dauerhaft mit dem Vorgang *Beobachten* beschäftigt.

Literatur: Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.231-233
http://link.springer.com/chapter/10.1007/978-3-8348-9174-7_8

Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 61-65
http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_5

Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 210-214
http://link.springer.com/chapter/10.1007/3-540-30950-0_12

Zielsetzung des Entwurfsmusters Beobachter

- ❑ Interessierten Objekten – den Beobachtern – sollen Änderungen an einem anderen Objekt – dem Subjekt – schnell und zugleich
- ❑ mit wenig Aufwand bekannt gemacht werden.

Dazu wird eine *Eigenschaft* von programmierten Lösungen genutzt:

- ❑ Objekte in der Programmwelt kooperieren zuverlässig.
- ❑ Im Beobachter-Muster kooperieren das (beobachtete) Subjekt und der Beobachter. Entsprechungen in der realen Welt wären z.B.:
 - Ein Autohersteller versendet Prospekte zum neuen Modell.
 - Der Kaufhausdieb informiert den Detektiv über seinen Diebstahl.

Idee:

- ❑ Das (beobachtete) Subjekt erlaubt das An-und Abmelden von Beobachtern.
- ❑ Beobachter warten passiv auf eine Benachrichtigung durch das Subjekt.
- ❑ Subjekt besitzt einen Benachrichtigungsmechanismus und informiert alle angemeldeten Beobachter, dass ein Ereignis aufgetreten ist.
- ❑ Der Beobachter kann sich nach der Benachrichtigung über ein aufgetretenes Ereignis Informationen über das Subjekt beschaffen und so das Beobachten abschließen.

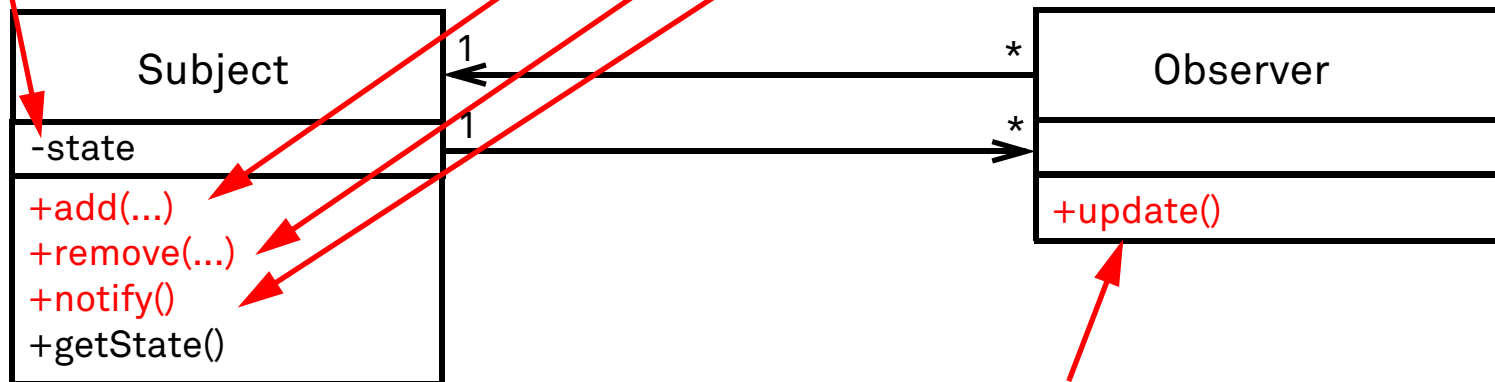
Struktur des Entwurfsmusters Beobachter

Objekt hat einen
beobachtbaren Zustand

Anmelden eines Beobachters

Abmelden eines Beobachters

Benachrichtigen aller Beobachter

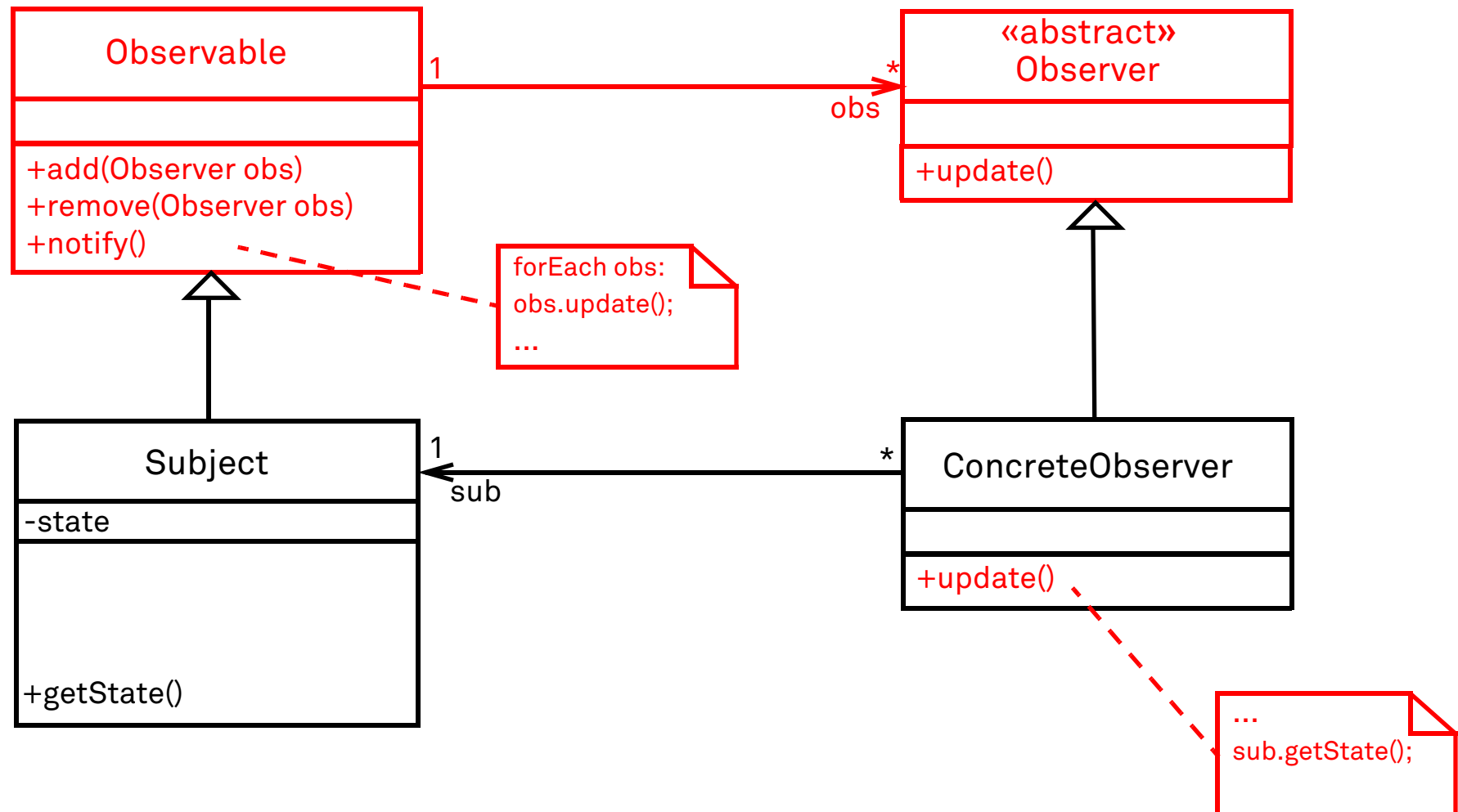


Aktion des Beobachters, die bei
Änderungen des Subjekts ausgeführt wird

Struktur des Entwurfsmusters Beobachter

allgemeiner Aufbau

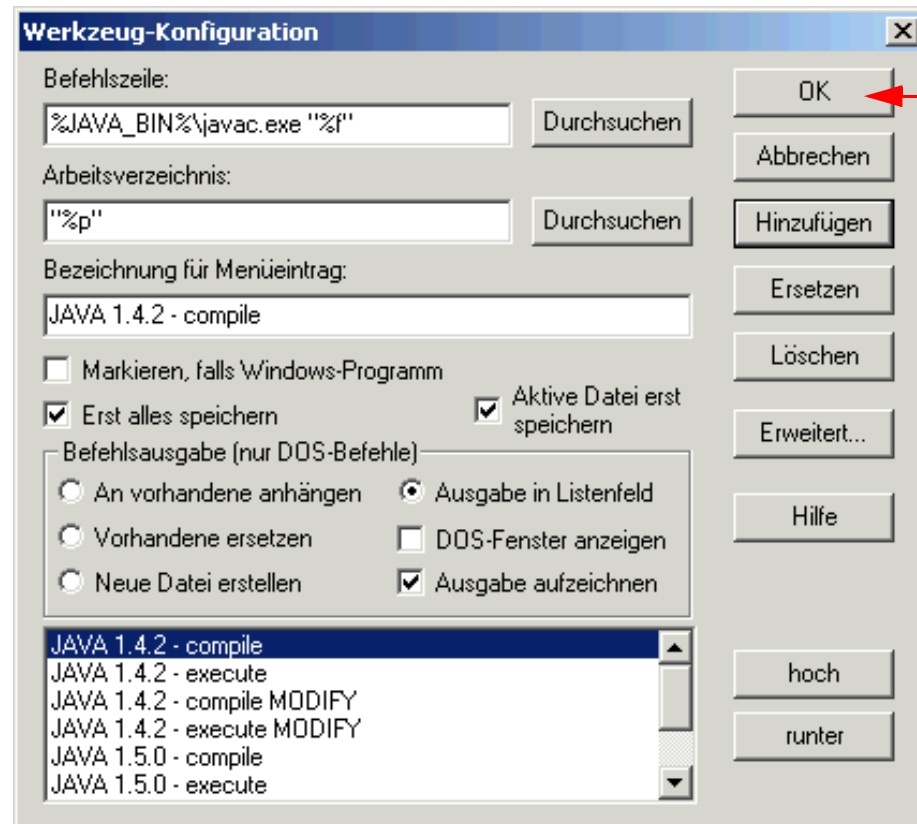
(Fortsetzung)



Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing

- ❑ Während der Ausführung können beliebig viele Beobachter mit Informationen versorgt werden.
- ❑ Das Muster kann daher gut bei der Gestaltung graphischer Oberflächen eingesetzt werden:
 - Beobachtet werden dann die graphischen Elemente der Oberfläche, die der Benutzer manipulieren kann: Menüs, Schaltknöpfe, Textfelder, ...
 - Beobachter sind Programmabschnitte, die bei einer Manipulation durch den Benutzer reagieren sollen.
- ❑ In Java ist dieses Konzept fest in die graphische Bibliothek Swing eingebaut. Die Beobachter heißen dort *Listener*, die entsprechenden Klassen also z.B. *ActionListener*, ...

Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing

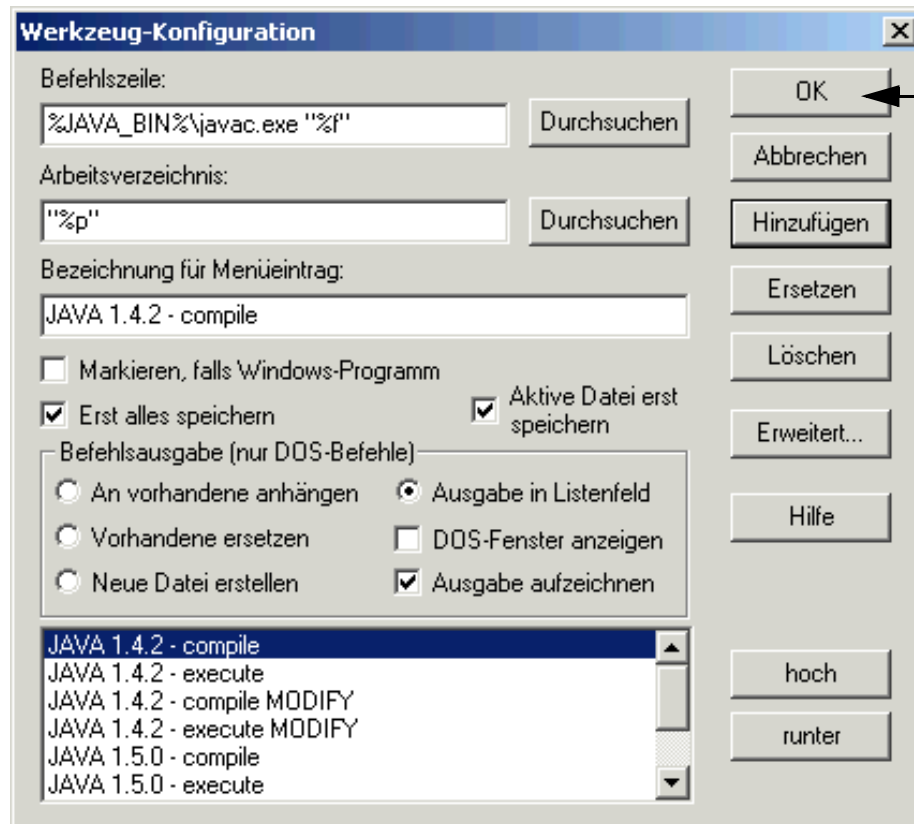


`JButton ok = new JButton("OK");`

Die Klasse `JButton` stellt ein beobachtbares Objekt bereit.

Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing

(Fortsetzung)



`JButton ok = new JButton("OK");`

Die Klasse JButton stellt ein beobachtbares Objekt bereit.

Zur Bearbeitung des Drückens des OK-Buttons wird eine Klasse implementiert:

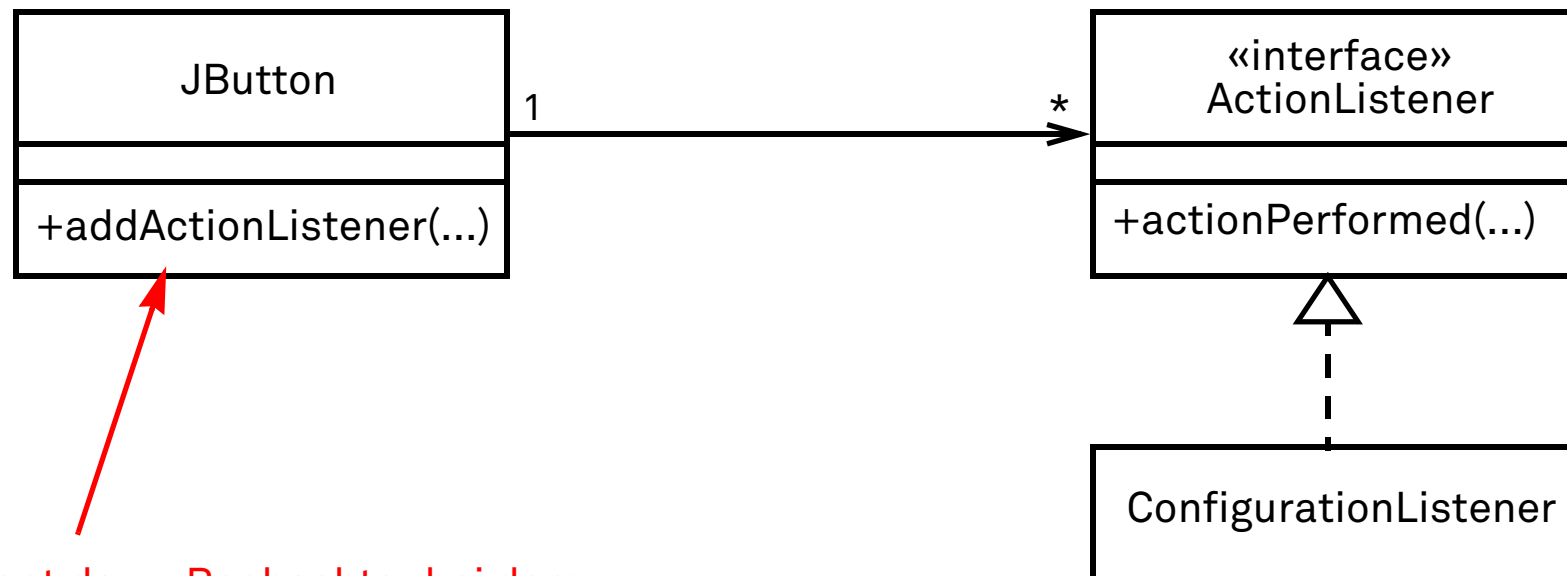
```
class ConfigurationListener
    implements ActionListener {
    ...
}
```

ActionListener ist ein Beobachter für ein JButton-Objekt!

Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing

(Fortsetzung)

(Darstellung als Klassendiagramm)



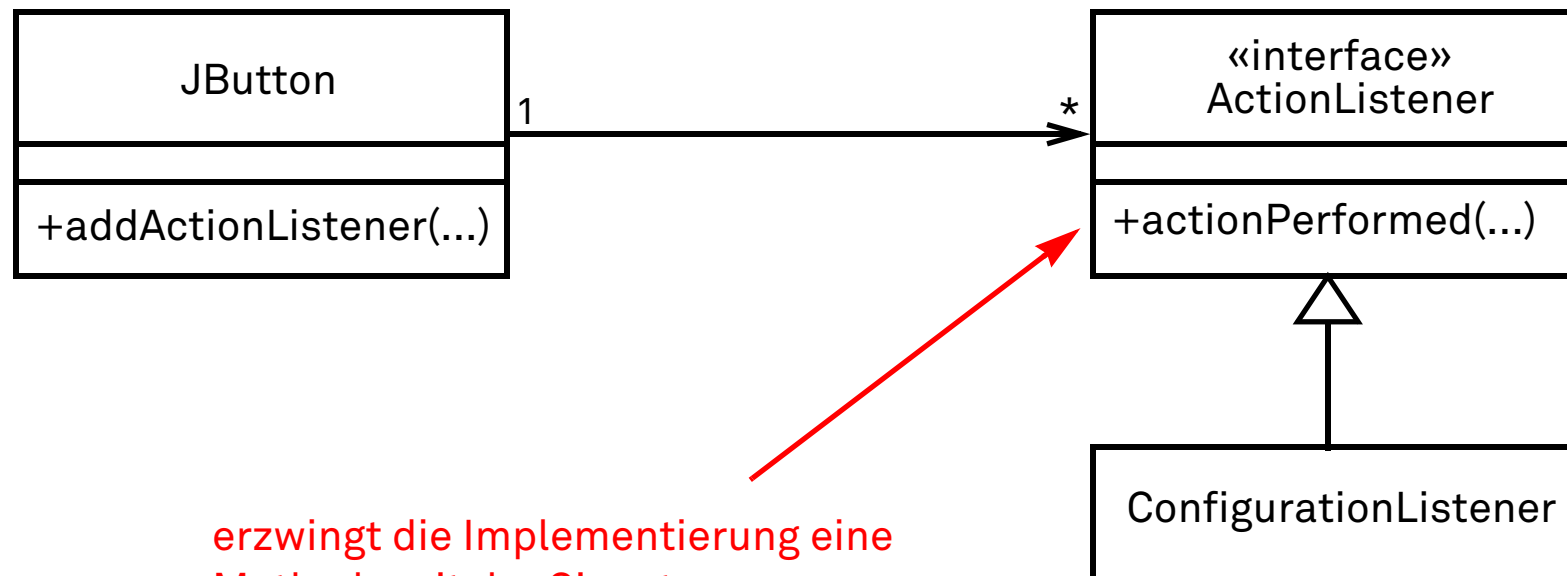
dient dazu, Beobachter bei dem
Button ok anzumelden, im Beispiel:

```
ok.addActionListener(new ConfigurationListener());
```

Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing

(Fortsetzung)

(Darstellung als Klassendiagramm)

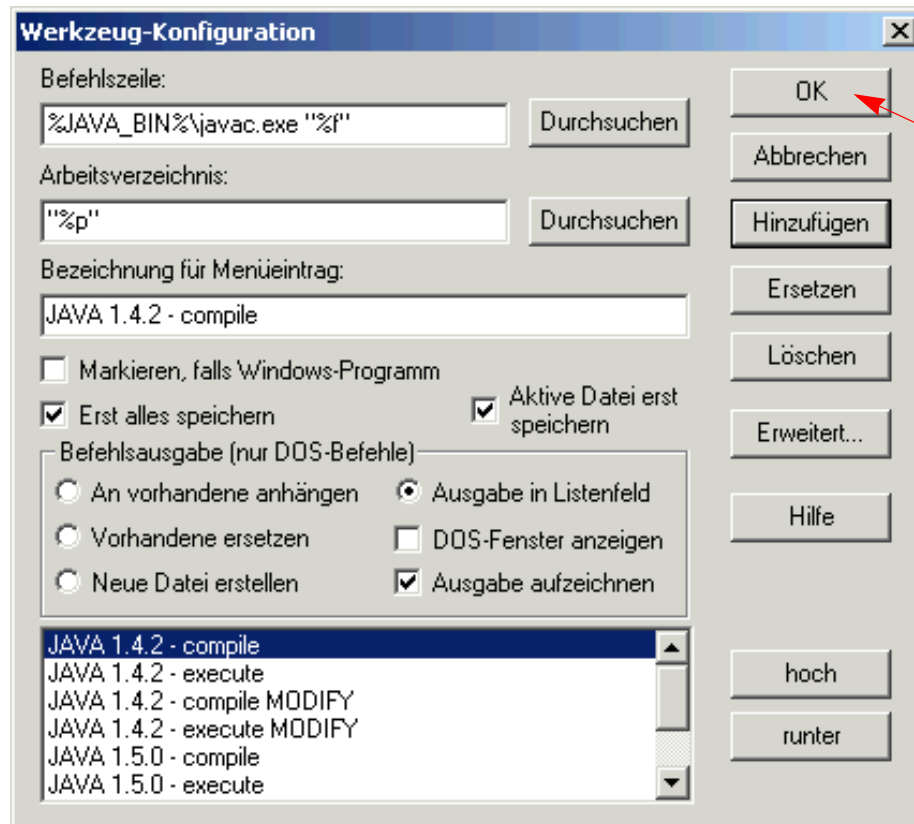


erzwingt die Implementierung eine
Methode mit der Signatur

`actionPerformed(...);`

Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing

(Fortsetzung)



Ablauf:

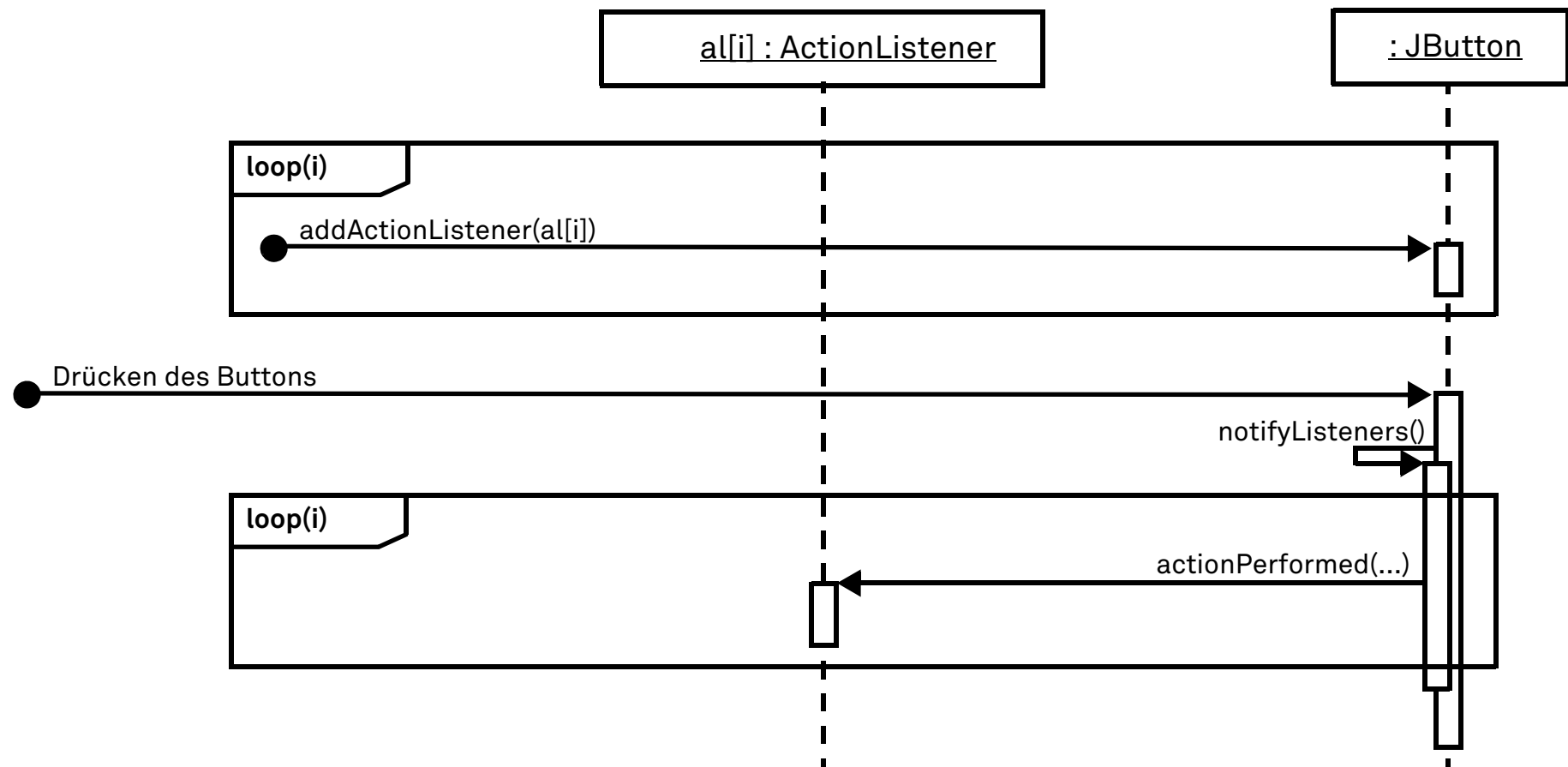
Wird der OK-Button gedrückt, so wird bei allen Beobachtern die Methode `actionPerformed` ausgeführt.

Vorteile:

- Viele Objekte können auf ein einziges Ereignis reagieren.
- Diese Objekte müssen sich nicht kennen.
- Diese Objekte müssen nicht aktiv warten.
- Diese Objekte können während der Ausführung geändert werden.

Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing (Darstellung als Sequenzdiagramm)

(Fortsetzung)



Erkenntnisse aus Klassen- und Sequenzdiagramm

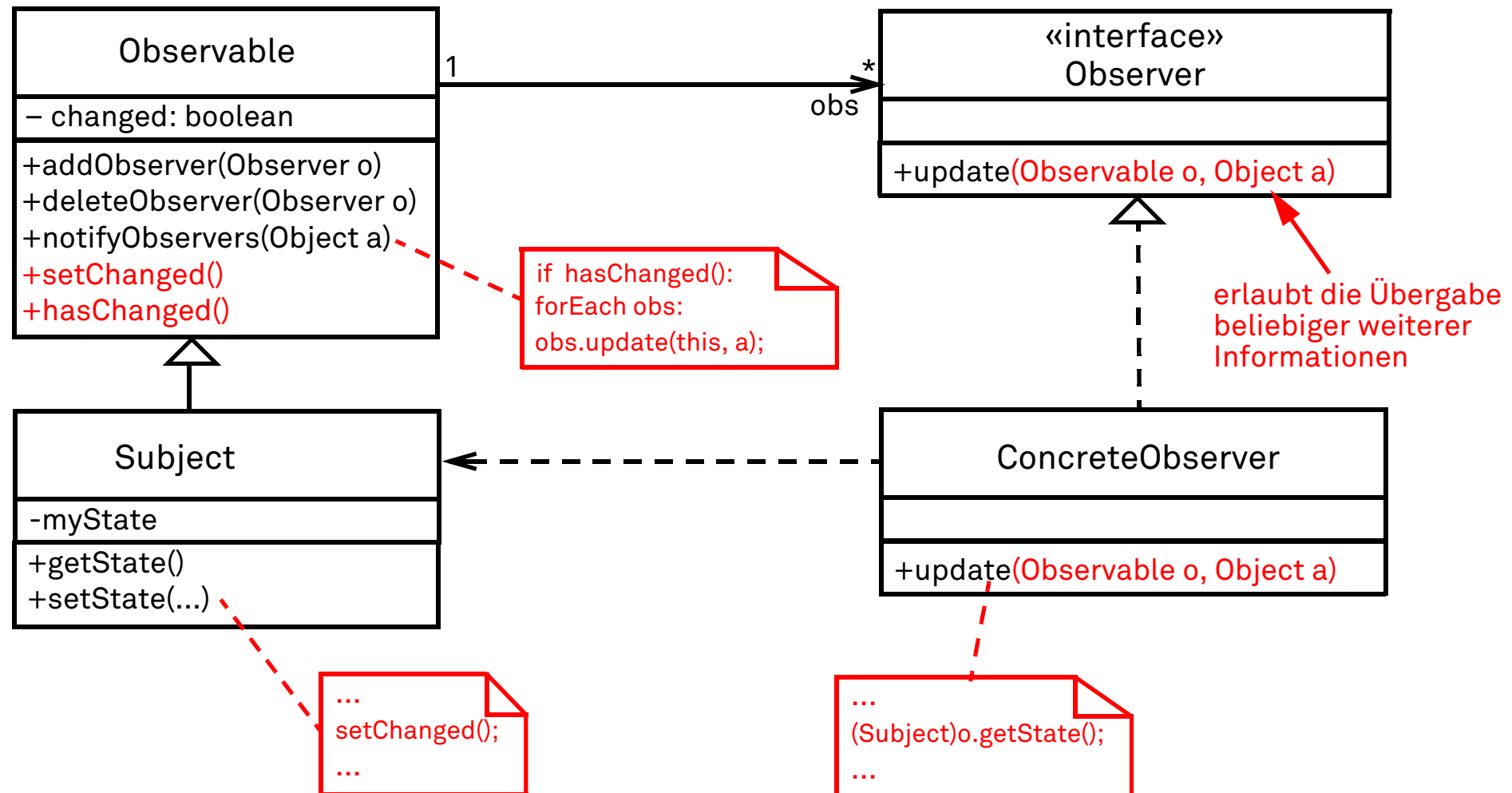
- ❑ Die Implementierung des Beobachter-Musters erfordert nur wenig Aufwand, da wesentliche Abläufe in vordefinierten Klassen vorgegeben werden können: Anmeldung, Abmeldung, Benachrichtigung
- ❑ Während der Ausführung können dann beliebig viele Beobachter mit Informationen versorgt werden.

- ❑ Das Muster wird auch bei der Gestaltung graphischer Oberflächen eingesetzt:
 - Beobachtet werden dann die graphischen Elemente der Oberfläche, die der Benutzer manipulieren kann: Menüs, Schaltknöpfe, Textfelder, ...
 - Beobachter sind Programmabschnitte, die bei einer Manipulation durch den Benutzer reagieren sollen.

Struktur des Entwurfsmusters Beobachter

(Fortsetzung)

Realisierung in Java: Die Java-Bibliothek stellt die Klassen `Observable` und `Observer` bereit.



Entwurfsmuster Beobachter: Realisierung in Java

Methoden der Klasse Observable:

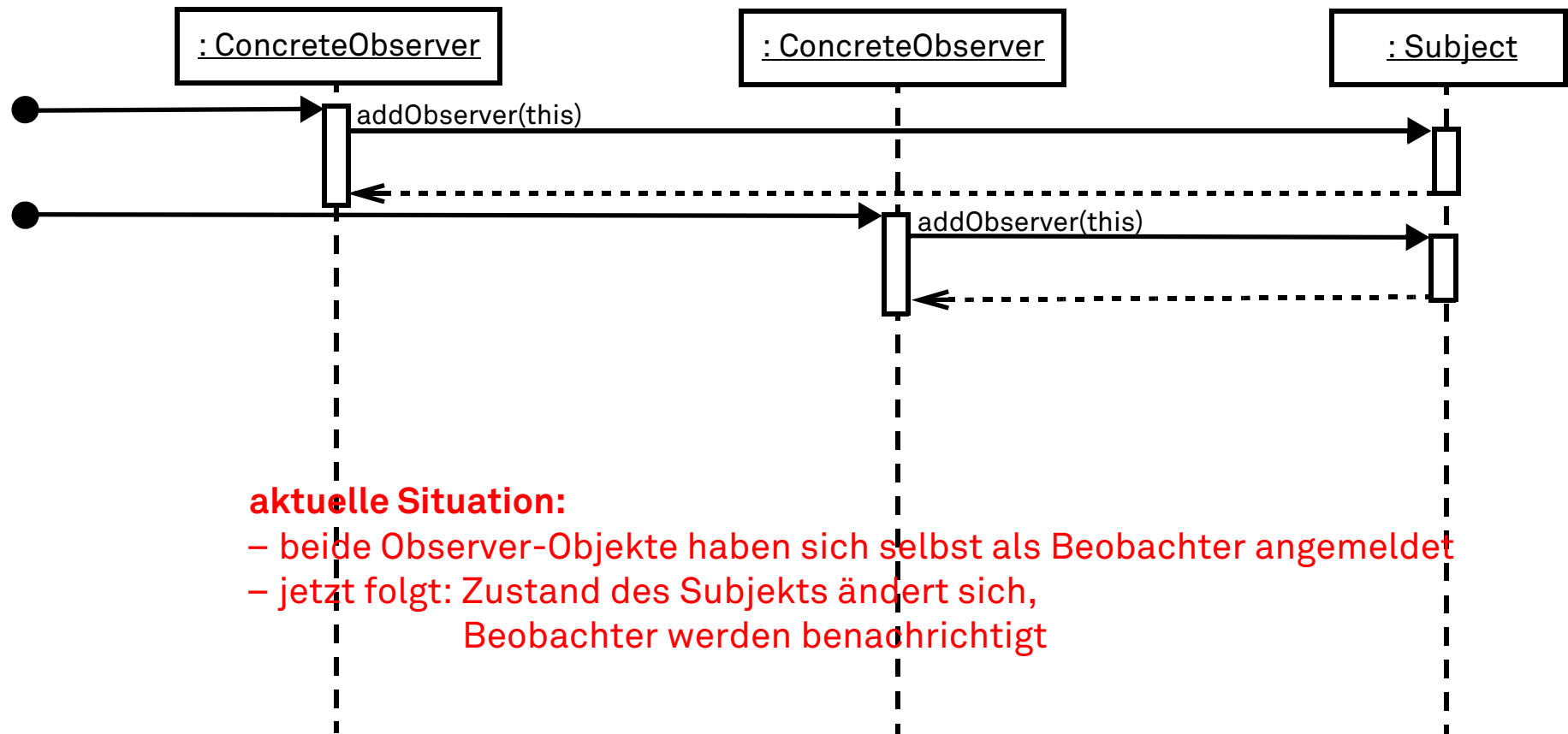
- ❑ `addObserver` meldet einen Beobachter an, der das Interface `Observer` realisiert
- ❑ `deleteObserver` meldet einen Beobachter wieder ab
- ❑ `setChanged` setzt `changed`-Attribut, das anzeigt, ob das Subjekt geändert wurde
- ❑ `hasChanged` liefert den Wert des `changed`-Attributs
- ❑ `notifyObservers` benachrichtigt alle angemeldeten Beobachter; aber nur dann, wenn das Subjekt tatsächlich geändert wurde:
das `changed`-Attribut ermöglicht so eine Entkopplung von Änderung und Benachrichtigung

Methode des Interface Observer

- ❑ `update`
 - wird von der Methode `notifyObservers` aufgerufen und muss im konkreten Beobachter die Aktionen implementieren, die bei einer Änderung auszuführen sind.
 - Da das beobachtete Subjekt sich selbst als Parameter übergibt, benötigt der Beobachter kein Attribut, um sich dieses zu merken!

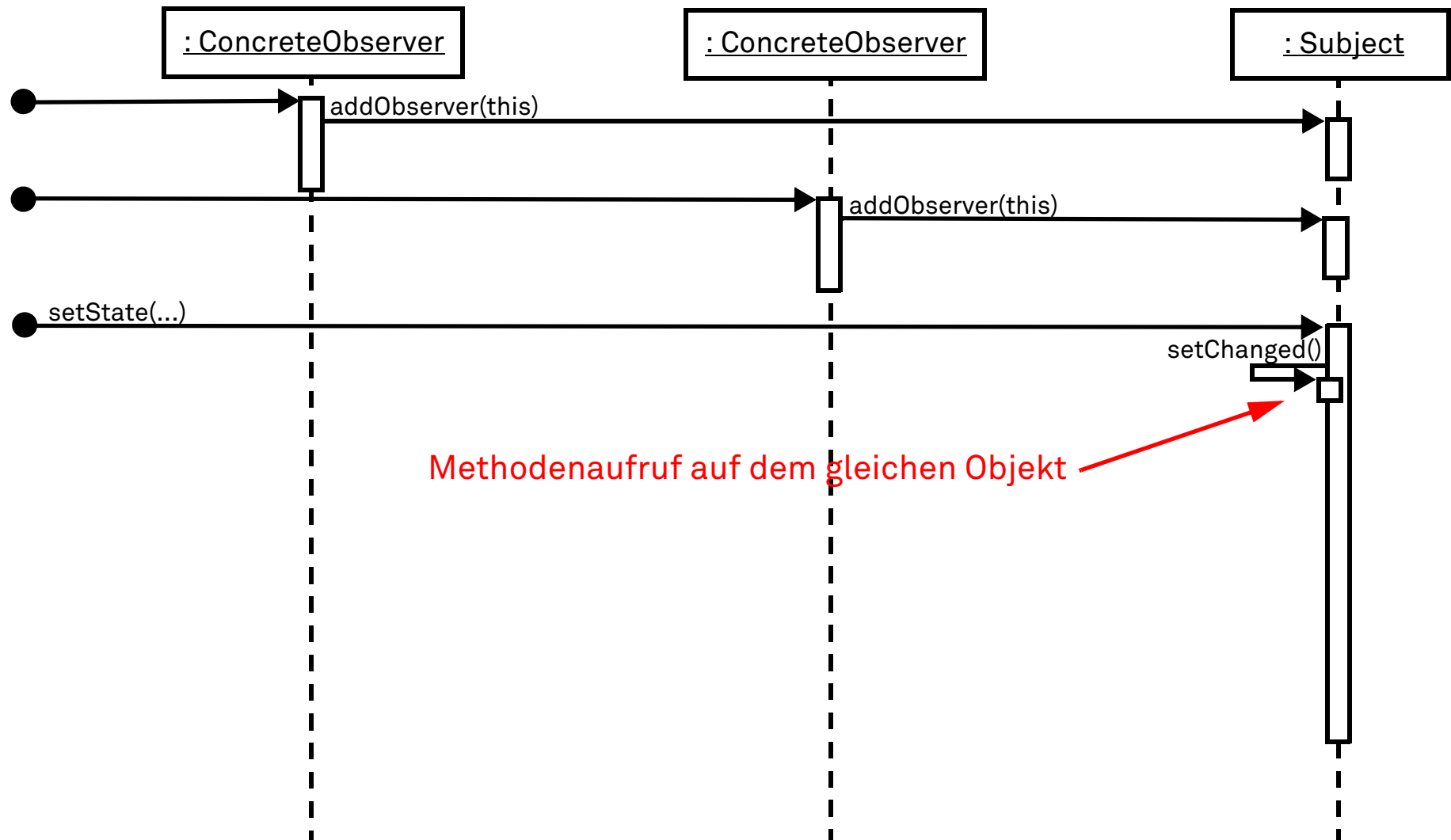
⇒ **der Ablauf wird jetzt etwas komplexer, seine Beschreibung auch!**
(Das Beobachter-Muster ist ein Verhaltensmuster!)

Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter



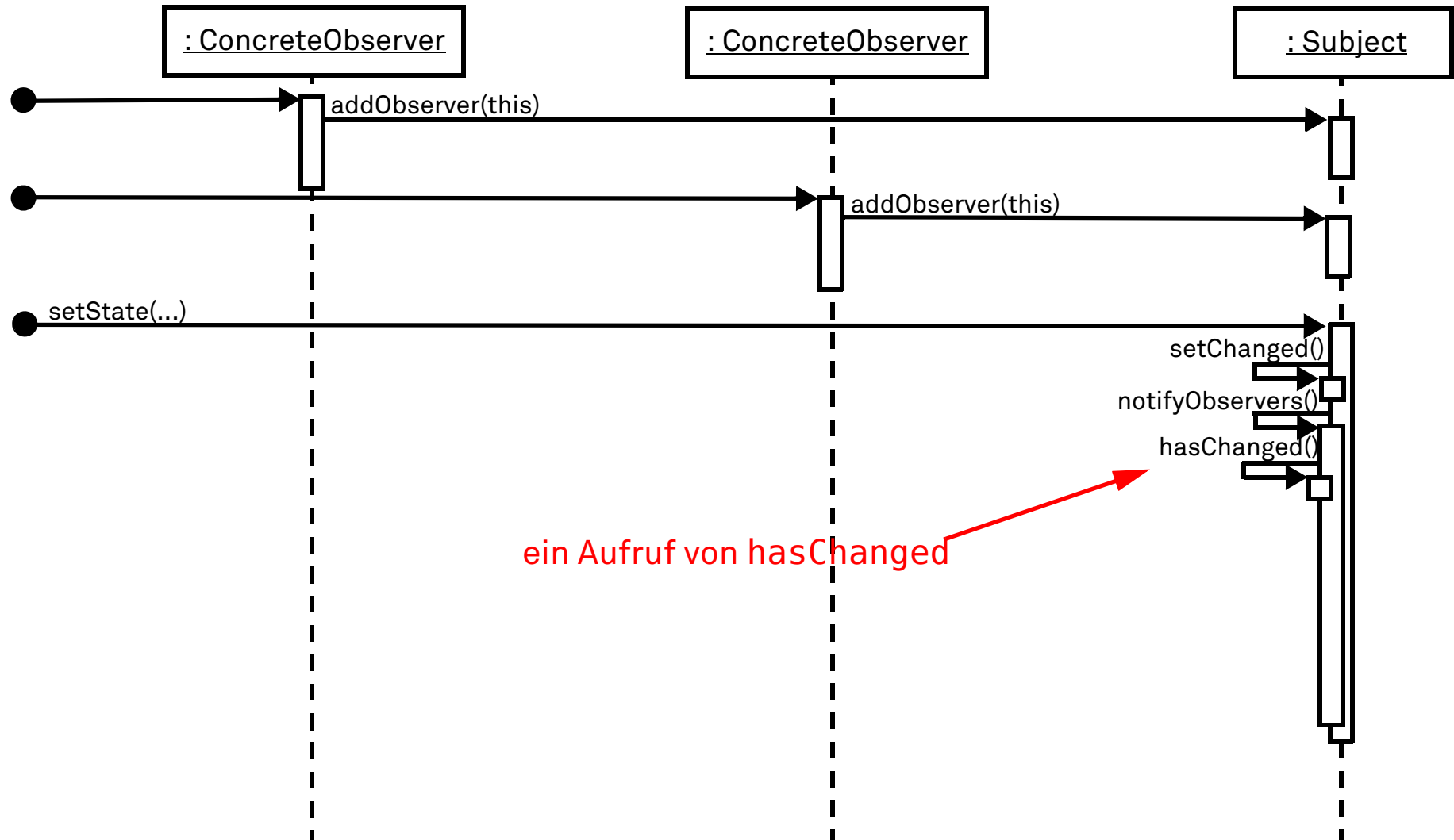
Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



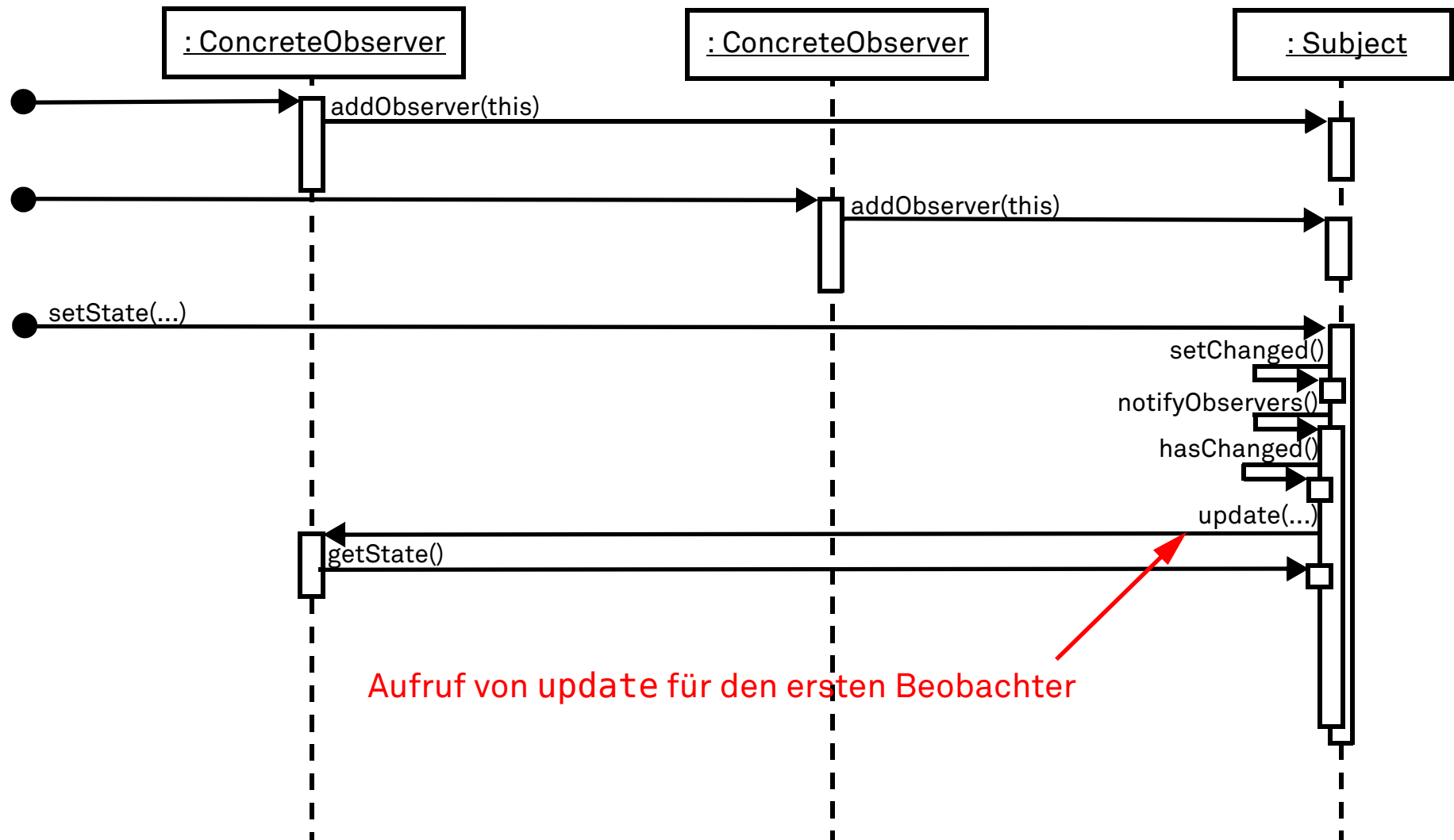
Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



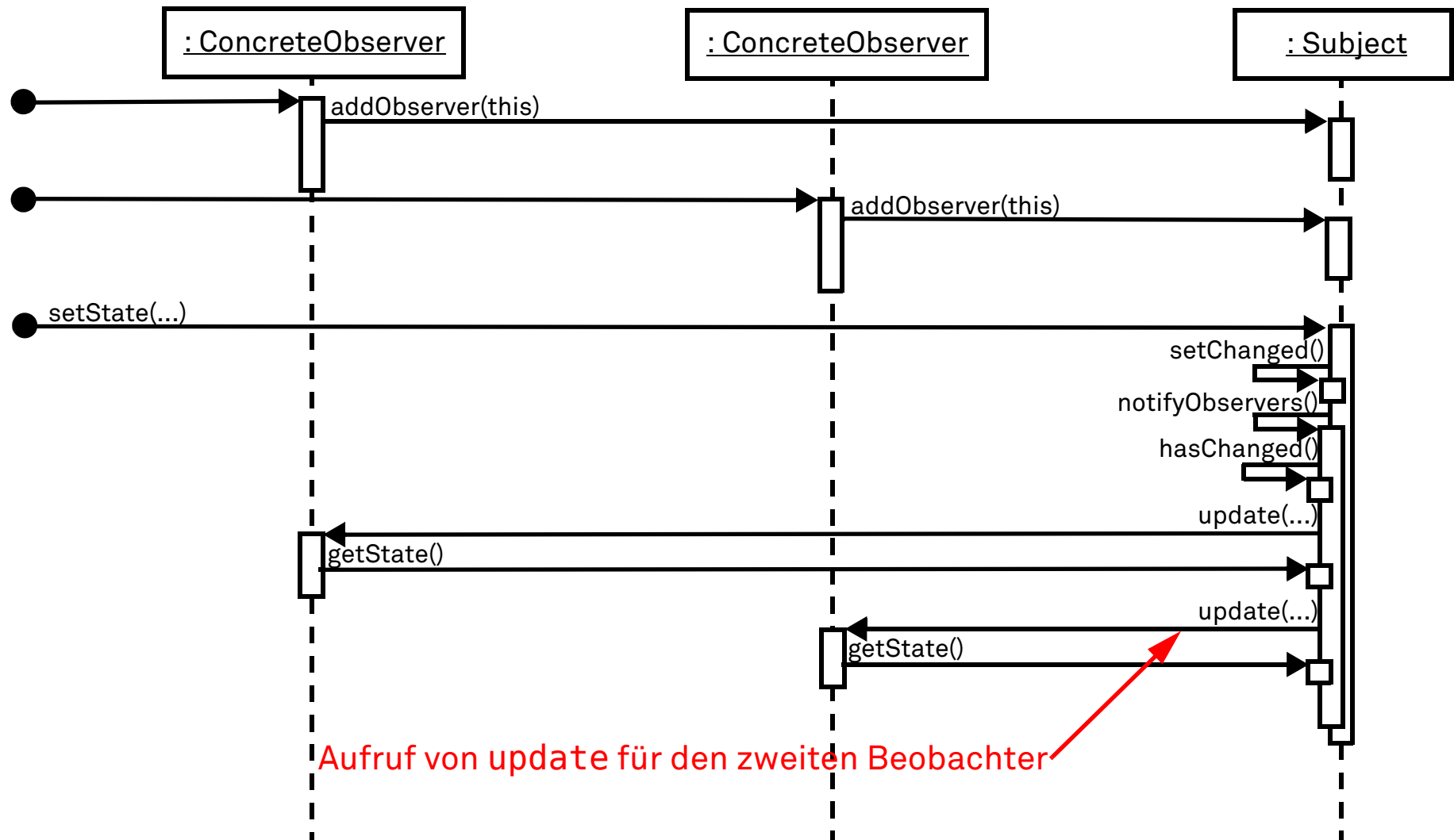
Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



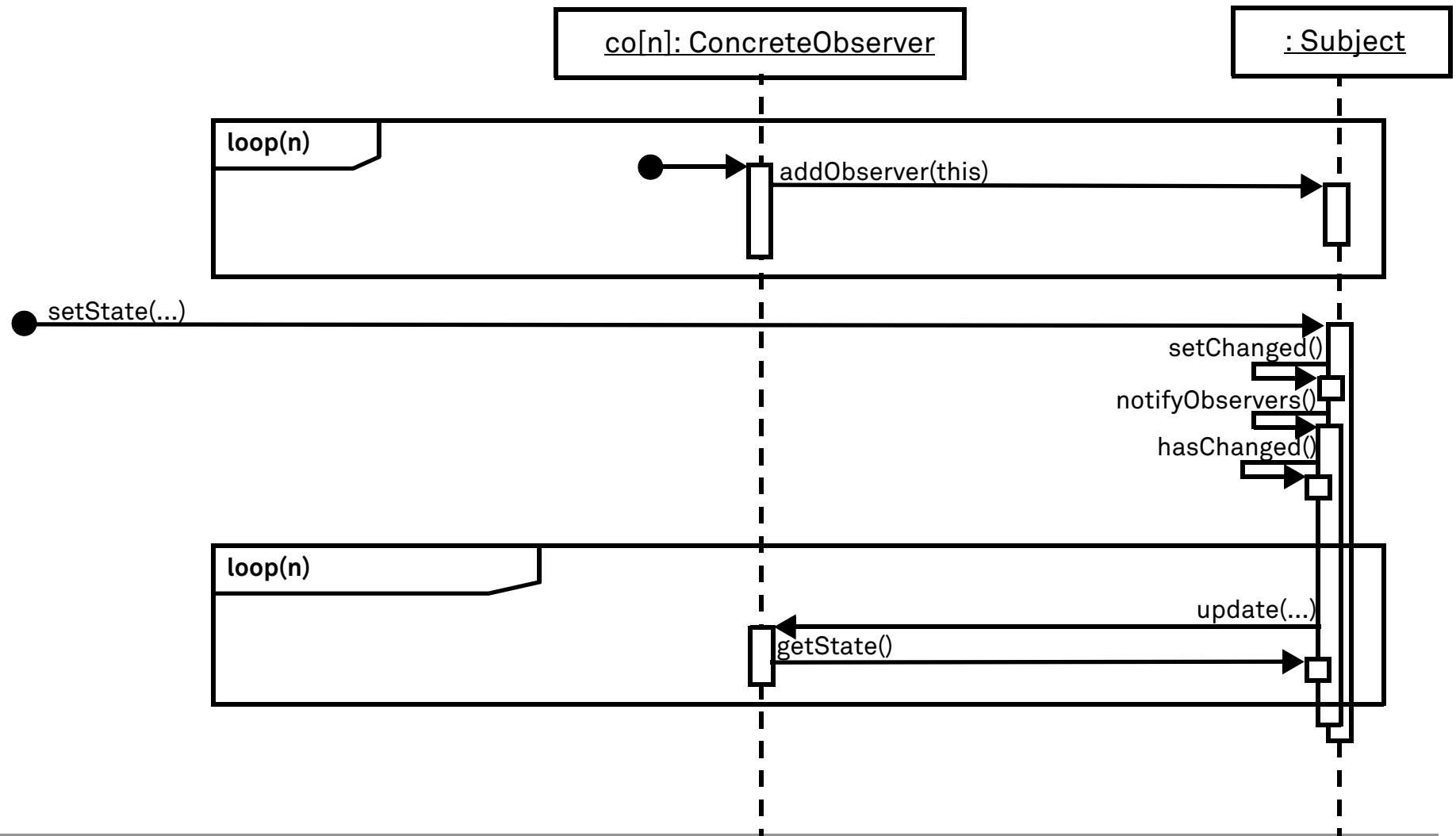
Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



allgemeine Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



Entwurfsmuster Beobachter: Bewertung der Realisierung in Java

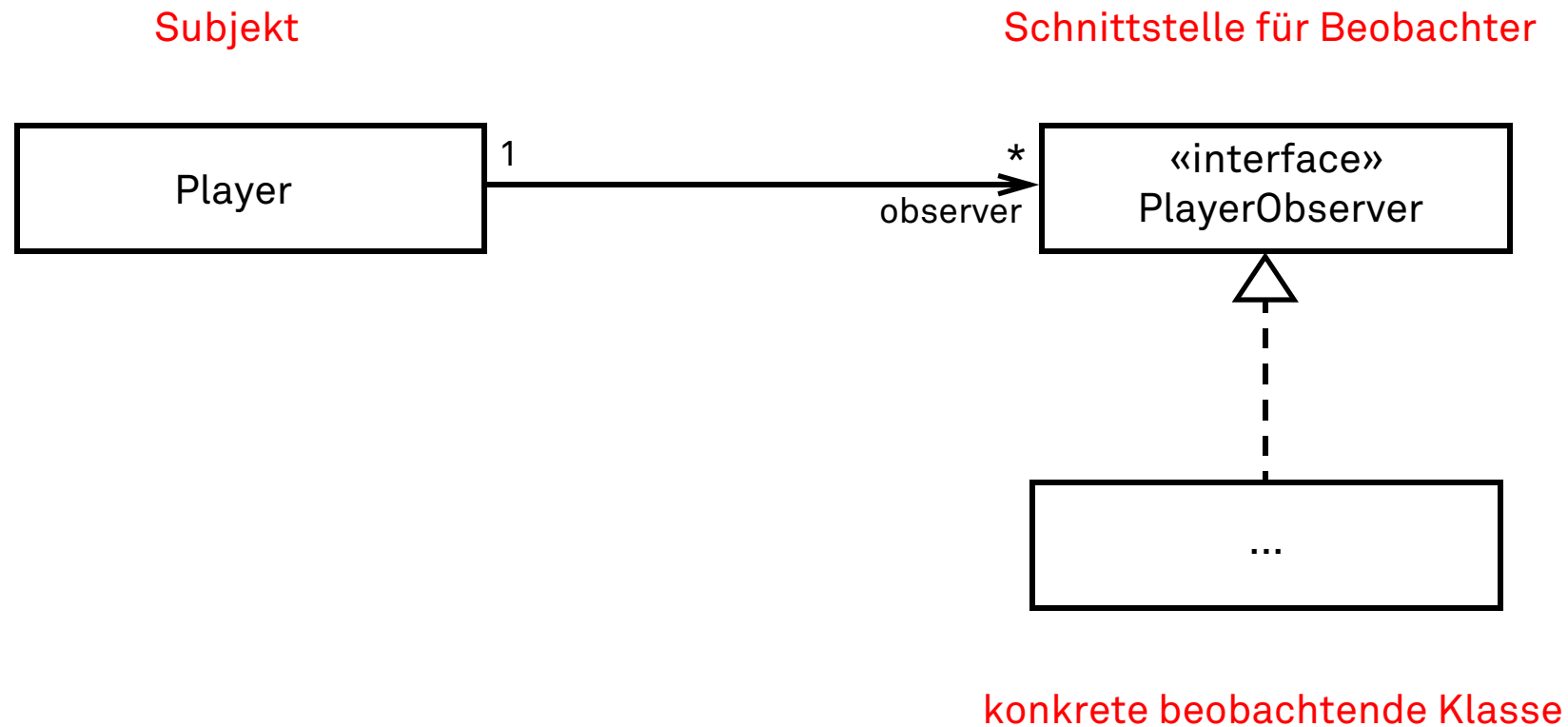
- ❑ Die Realisierung in Java ermöglicht es der konkreten Implementierung, die Abläufe in der Methode `notify` über das Attribut `changed` zu steuern.
- ❑ Eine Klasse, die `Observer` konkretisiert, kann nur eine einzige `update`-Methode implementieren.
- ❑ Soll ein Beobachter an mehreren Umsetzungen des Entwurfsmusters Beobachter beteiligt sein, so müssen alle `update`-Abläufe in einer Methode zusammengefasst werden und dort dann anhand des ersten Parameters von `update` ausgewählt und ausgeführt werden.
- ❑ Letztlich ist die Java-Implementierung recht unhandlich.
Sie wird daher nur selten eingesetzt.

Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

- ❑ Handlungen des Spielers (Starfighter) verändern die Spielsituation. Daher beobachten viele andere Objekte des Spiels den Spieler.
- ❑ Die Klasse `Player` im Paket `edu.udo.cs.swtsf.core.player` ist daher als beobachtbares Subjekt implementiert, dessen Methoden an die Situationen des Spiels angepasst sind.
- ❑ Die Klasse `Player` enthält viele Methoden, die der Benachrichtigung der Beobachter dienen.
Beobachter müssen daher im Regelfall bei einer Benachrichtigung nicht mehr den Zustand des Subjekts abfragen, sondern werden unmittelbar durch die Benachrichtigung informiert.
- ❑ Das Interface `PlayerObserver` kennzeichnet Klassen, die sich bei einem `Player`-Objekt als Beobachter anmelden können.
Für alle Methoden des Interfaces sind `default`-Implementierungen vorgenommen worden, so dass eine Beobachterklasse nur die Methoden für die Benachrichtigungen überschreiben muss, bei denen eine Aktion erfolgen soll.

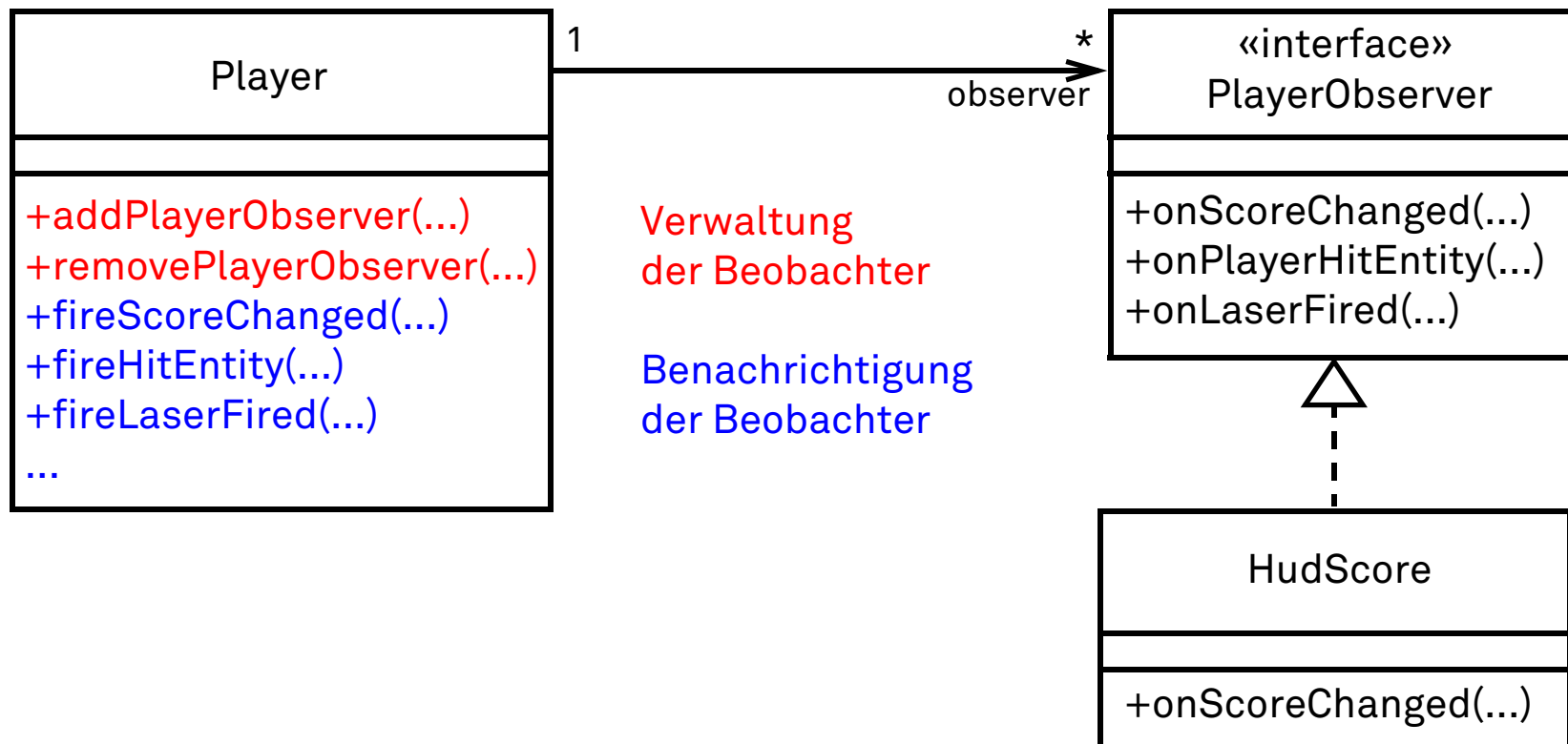
Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

(Fortsetzung)



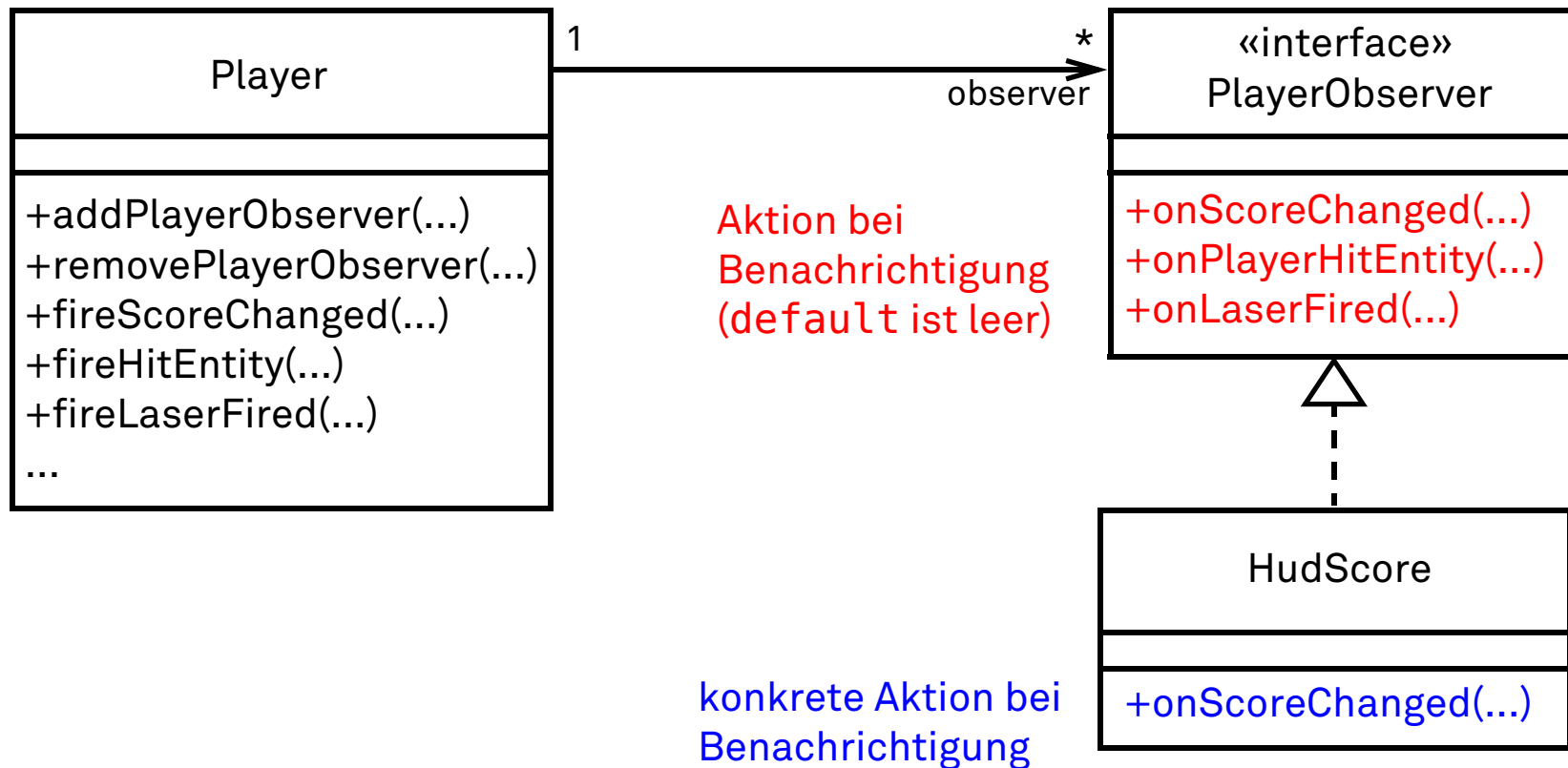
Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

(Fortsetzung)



Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

(Fortsetzung)



Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

(Fortsetzung)

Implementierung der Klasse Player

```
public class Player extends Target {  
    private Group<PlayerObserver> observers = new BufferedGroup<>();  
    ...  
    public void addPlayerObserver(PlayerObserver observer) {  
        observers.add(observer);  
    }  
    public void removePlayerObserver(PlayerObserver observer) {  
        observers.remove(observer);  
    }  
    protected void fireScoreChanged(int value) {  
        observers.forEach((observer) -> observer.onScoreChanged(this));  
    }  
    protected void fireHitEntity(Target t) {  
        observers.forEach((observer) -> observer.onPlayerHitEntity(this, t));  
    }  
    protected void fireLaserFired(Collection<Bullet> b) {  
        observers.forEach((observer) -> observer.onLaserFired(this, b));  
    }  
    ...  
}
```

Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

(Fortsetzung)

Implementierung des Interfaces PlayerObserver

```
public interface PlayerObserver {  
  
    public default void onScoreChanged(Player player)  
    {}  
    public default void onPlayerHitEntity(Player player, Target target)  
    {}  
    public default void onLaserFired(Player player,  
                                     Collection<Bullet> bullets)  
    {}  
    ...  
}
```

Die leeren default-Implementierungen dienen dem Komfort bei der Implementierung von Klassen, die dieses Interface implementieren. Solche Klassen sollen meist nur auf wenige Benachrichtigungen reagieren und müssen für Benachrichtigungen, an denen sie kein Interesse haben, auch keine Methode implementieren.

Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

(Fortsetzung)

Implementierung der Klasse HudScore

```
public class HudScore extends HudElement implements PlayerObserver {  
  
    public HudScore() {  
        super(HudElementOrientation.TOP, "HUD/Score", 0, 0, 32, 32);  
    }  
  
    public void onScoreChanged(Player player) {  
        setText(Integer.toString(player.getScore()));  
    }  
  
    protected void afterAdded(ViewManager view, Game game) {  
        onScoreChanged(game.getPlayer());  
    }  
}
```

Die Methode `afterAdded` wird aufgerufen, sobald das `HudScore`-Objekt als Anzeige der Punkte zum Spiel hinzugefügt wurde. Der Aufruf `onScoreChanged` bewirkt, dass direkt eine Anzeige erfolgt.

Zusammenfassung Entwurfsmuster *Beobachter*

Vorteile:

- ❑ Das Beobachter-Muster gibt ein Protokoll vor, an dem sich der Informationsaustausch zwischen Objekten orientiert.
- ❑ Das Beobachter-Muster entkoppelt das beobachtete Subjekt von seinen Beobachtern. Dadurch lassen sich in der Entwicklung leicht weitere Beobachter-Klassen anlegen. Während der Ausführung ist die Zahl der Beobachter dynamisch und nicht begrenzt.
- ❑ Der Mechanismus zur Benachrichtigung kann unabhängig von der konkreten Problemstellung implementiert werden.

Nachteil:

- ❑ Beobachtet ein Beobachter-Objekt verschiedene Subjekte, so kann die Identifizierung des benachrichtigenden Objekts problematisch sein. Die Objekte sollten dann verschiedene Methoden zur Benachrichtigung verwenden.

Entwurfsmuster Fabrikmethode

Eine **Fabrikmethode**

ermöglicht es, auf einfache Weise die Auswahl einer zu benutzenden Klasse für das gesamte System an nur einer Stelle des Systems festzulegen.

Motivation:

- ❑ Es gibt Klassen, die an verschiedenen Stellen im System genutzt werden und für die es mehrere Implementierungen gibt.
- ❑ Beispiel: Implementierungen für das Interface Group (siehe Folie 195)

Idee:

- ❑ Das Erzeugen von Objekten wird nicht direkt durch den Konstruktor vorgenommen, sondern in eine eigene Klasse ausgelagert, in der eine Fabrikmethode den Konstruktoraufruf einkapselt.
- ❑ Bei einer Änderung der zu verwendenden Klasse muss dann nur an einer Stelle im System die Fabrikmethode geändert werden.

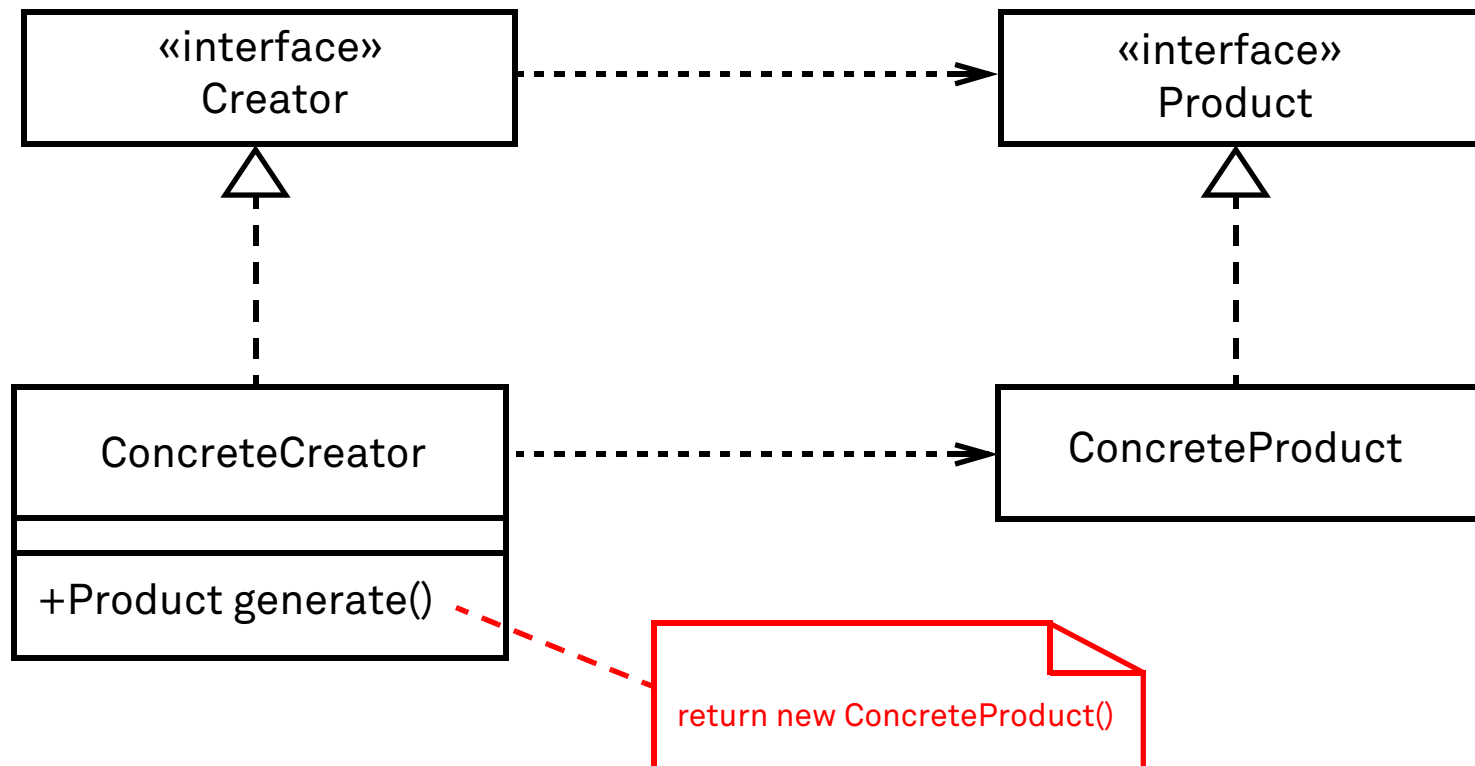
Literatur: Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 31-35
http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_4

Entwurfsmuster Fabrikmethode

(Fortsetzung)

allgemeine Struktur des Entwurfsmusters

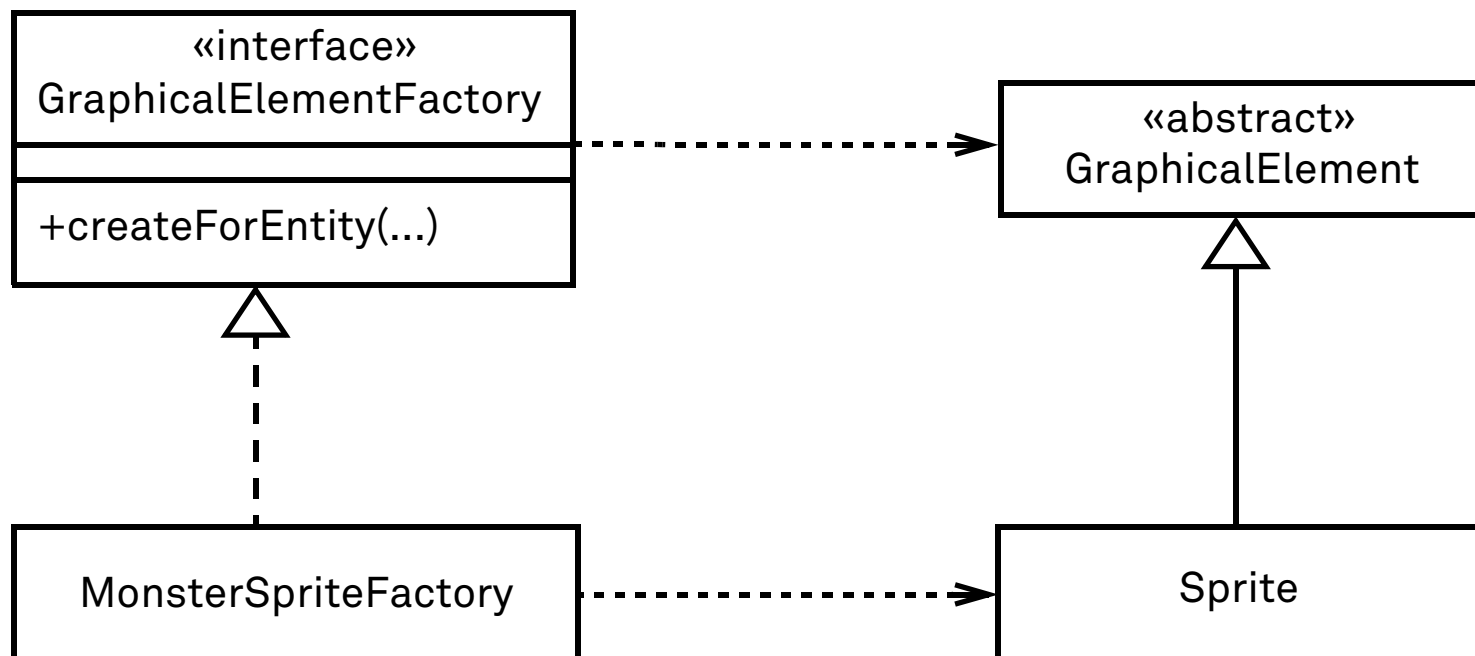
Produkte werden im System
nur über Product referenziert



Entwurfsmuster Fabrikmethode im Beispiel SWT-Starfighter

(Fortsetzung)

Beispiel GraphicalElementFactory



Entwurfsmuster Fabrikmethode im Beispiel SWT-Starfighter

(Fortsetzung)

Beispiel GraphicalElementFactory

Implementierung

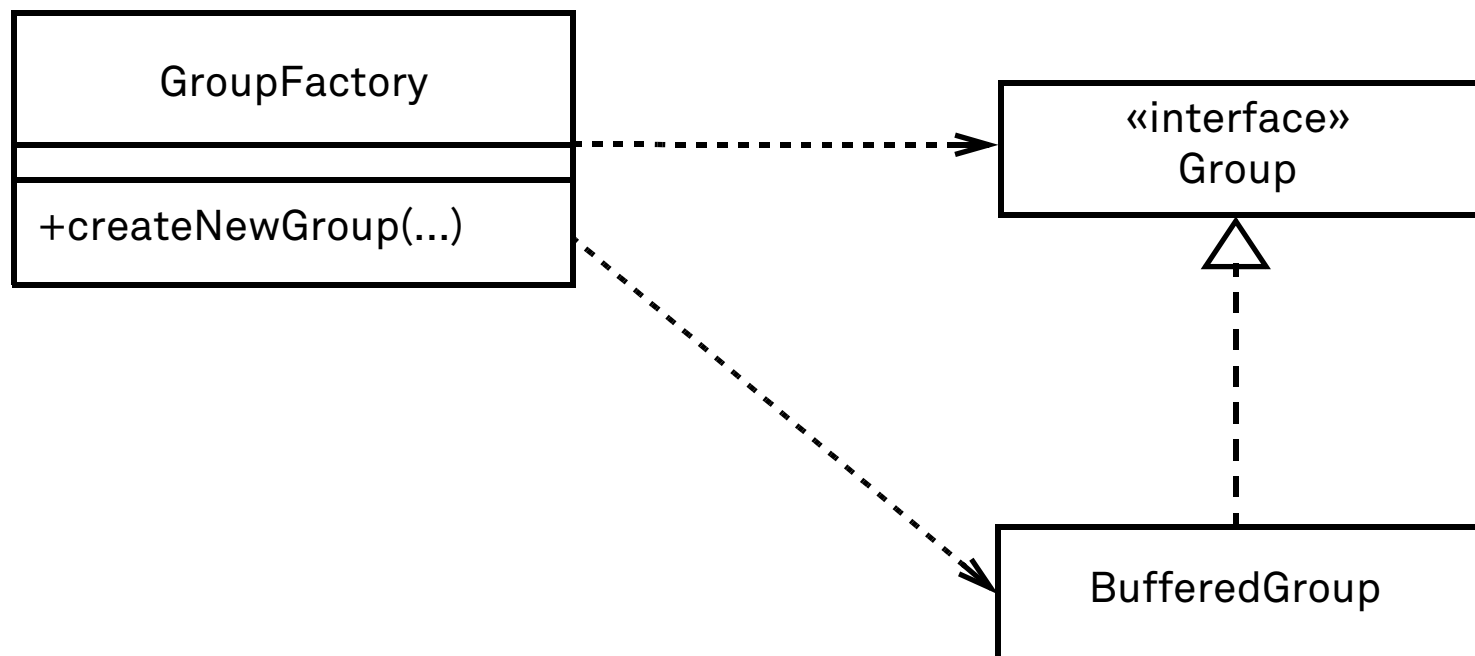
```
public interface GraphicalElementFactory {  
    public GraphicalElement createForEntity(ViewManager v, Entity e);  
}
```

```
public class MonsterSpriteFactory implements GraphicalElementFactory {  
    ...  
    public GraphicalElement createForEntity(ViewManager v, Entity e) {  
        Sprite sprite = v.newEntitySprite(e);  
        sprite.setImagePath(imageName);  
        sprite.setAnimator(new MonsterAnimator());  
        return sprite;  
    }  
}
```


Entwurfsmuster Fabrikmethode im Beispiel SWT-Starfighter

(Fortsetzung)

Beispiel GroupFactory



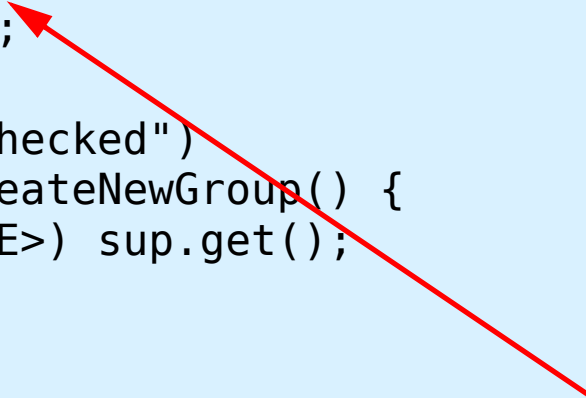
Entwurfsmuster Fabrikmethode im Beispiel SWT-Starfighter

(Fortsetzung)

Beispiel GroupFactory

Implementierung

```
import java.util.function.Supplier;
public class GroupFactory {
    private Supplier<Group<?>> sup = () -> new BufferedGroup<>();
    public void setSupplier(Supplier<Group<?>> supplier) {
        sup = supplier;
    }
    @SuppressWarnings("unchecked")
    public <E> Group<E> createNewGroup() {
        return (Group<E>) sup.get();
    }
}
```



```
public interface Supplier<T> {
    T get();
}
```

komfortable Lösung:
ermöglicht den Austausch der zu
erzeugenden Group-Klasse
während der Ausführung

Entwurfsmuster Singleton

Ein **Singleton**

stellt sicher, dass es von einer Klasse nur ein einziges Objekt gibt.

Motivation:

- ❑ Es gibt Klassen, von denen zur Laufzeit nur genau ein Objekt existieren darf.
- ❑ Beispiel: Vergabe von eindeutigen Schlüsseln wie Bestellnummern, Kundennummern, ...

Idee:

- ❑ Die zugehörige Klasse sorgt selbst dafür, dass es nur ein Objekt gibt:
 - Die Umsetzung ist abhängig von den Möglichkeiten der Programmiersprache.
 - Der Zugang zu Konstruktoren muss eingeschränkt werden,
z.B. durch Verhindern des Zugriffs: Die Konstruktoren werden privat vereinbart!
 - Statt des Konstruktors kontrollieren dann spezielle statische Methoden
das Erzeugen von nur einem Objekt.

Literatur: Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.217-219

http://link.springer.com/chapter/10.1007/978-3-8348-9174-7_8

Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 35-39

http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_4

Implementierung in Java

Standardvariante:

```
public class Singleton {  
    private static Singleton theInstance;  
    private Singleton() {}  
    public static Singleton getInstance() {  
        if (theInstance == null) {  
            theInstance = new Singleton();  
        }  
        return theInstance;  
    }  
    ...  
}
```

Diagram illustrating the implementation of the Singleton pattern in Java, with annotations explaining key components:

- private static Singleton theInstance;**: einzige Instanz der eigenen Klasse
- private Singleton() {}**: privater Konstruktor
- public static Singleton getInstance() {**: Klassenmethode für den Zugriff
- if (theInstance == null) {**: Prüfen der Existenz
- theInstance = new Singleton();**: Aufruf des privaten Konstruktors

Problem:

Nebenläufige Prozesse (Threads) können sich bei der Erzeugung überlappen!

Implementierung im Beispiel SWT-Starfighter

(Fortsetzung)

Von der Klasse GroupFactory sollte nur genau einmal ein Objekt erzeugt werden, damit die gleiche Implementierung der Klasse Group an allen Stellen im System verwendet wird.

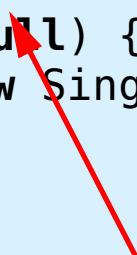
Die Klasse GroupFactory wird daher ergänzt um:

```
public class GroupFactory {  
    private static GroupFactory INSTANCE;  
    public static GroupFactory get() {  
        if (INSTANCE == null) {  
            INSTANCE = new GroupFactory();  
        }  
        return INSTANCE;  
    }  
    ...  
}
```

alternative Implementierung in Java

Lösung mit Synchronisation (Ausschluss konkurrierender Zugriffe):

```
public class Singleton {  
    private static Singleton theInstance;  
    private Singleton() {}  
    public static synchronized Singleton getInstance() {  
        if (theInstance == null) {  
            theInstance = new Singleton();  
        }  
        return theInstance;  
    }  
    ...  
}
```



sequentialisiert Methodenaufrufe
aus verschiedenen Threads

Probleme:


- ❑ Die Synchronisation verlangsamt die Ausführung!
- ❑ Die Synchronisation wird aber nur genau beim ersten Methodendurchlauf benötigt!

alternative Implementierung in Java

(Fortsetzung)

Lösung mit vorzeitig erzeugter Instanz:

```
public class Singleton {  
    private static final Singleton theInstance = new Singleton();  
    private Singleton() {}  
    public static Singleton getInstance() {  
        return theInstance;  
    }  
    ...  
}
```



erzeugt Instanz beim Laden
der Klasse in die
Virtuelle Maschine (VM)

Problem:

- ❑ Diese Lösung ist nur möglich, wenn – wie im Beispiel – keine zuvor zu berechnenden Werte in die Erzeugung eingehen!

alternative Implementierung in Java

(Fortsetzung)

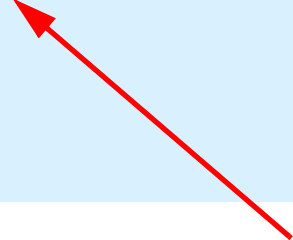
flexible und performante Lösung:

```
public class Singleton {  
    private Singleton() { }  
    private static class SingletonHolder {  
        public static final Singleton instance = new Singleton();  
    }  
    public static Singleton getInstance() {  
        return SingletonHolder.instance;  
    }  
    ...  
}
```

innere Klasse erzeugt
Instanz von Singleton



innere Klasse wird erst dann angelegt,
wenn getInstance aufgerufen wird



alternative Implementierung in Java

(Fortsetzung)

weitere flexible und performante Lösung:

```
public enum Singleton {  
    INSTANCE;  
    ...  
}
```

mehr Informationen:

http://en.wikipedia.org/wiki/Singleton_pattern#The_Enum-way

<http://electrotek.wordpress.com/2008/08/06/singleton-in-java-the-proper-way/>

Es gibt also für das recht einfache Muster Singleton
in nur einer Programmiersprache (Java)
mehrere verschiedene Lösungen

**aber: Singleton-Muster sollte nur dann eingesetzt werden, wenn die Gefahr besteht,
dass mehrere Instanzen erzeugt werden – z.B. bei nebenläufigen Prozessen.**

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

Entwurfsmuster *Abstrakte Fabrik*

(Fortsetzung)

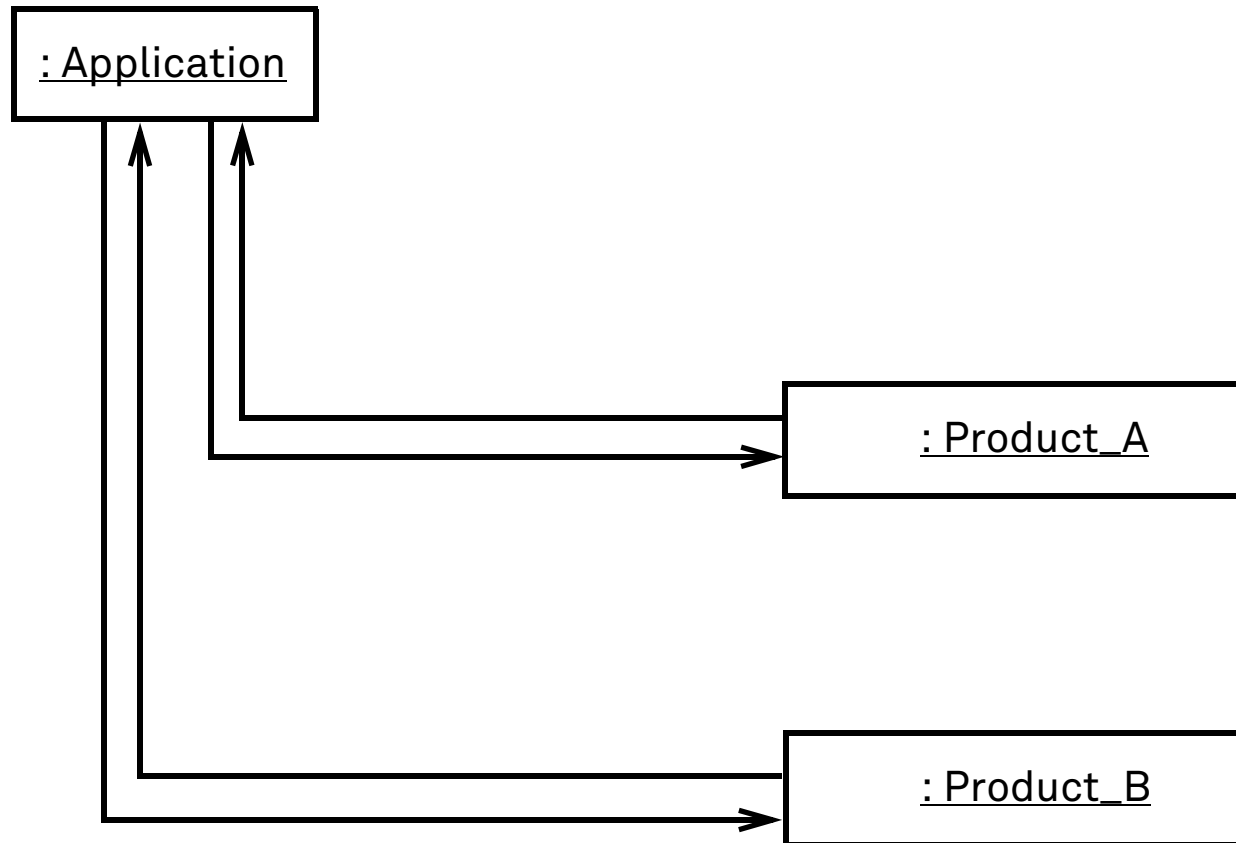
Beispiele:

- ❑ 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

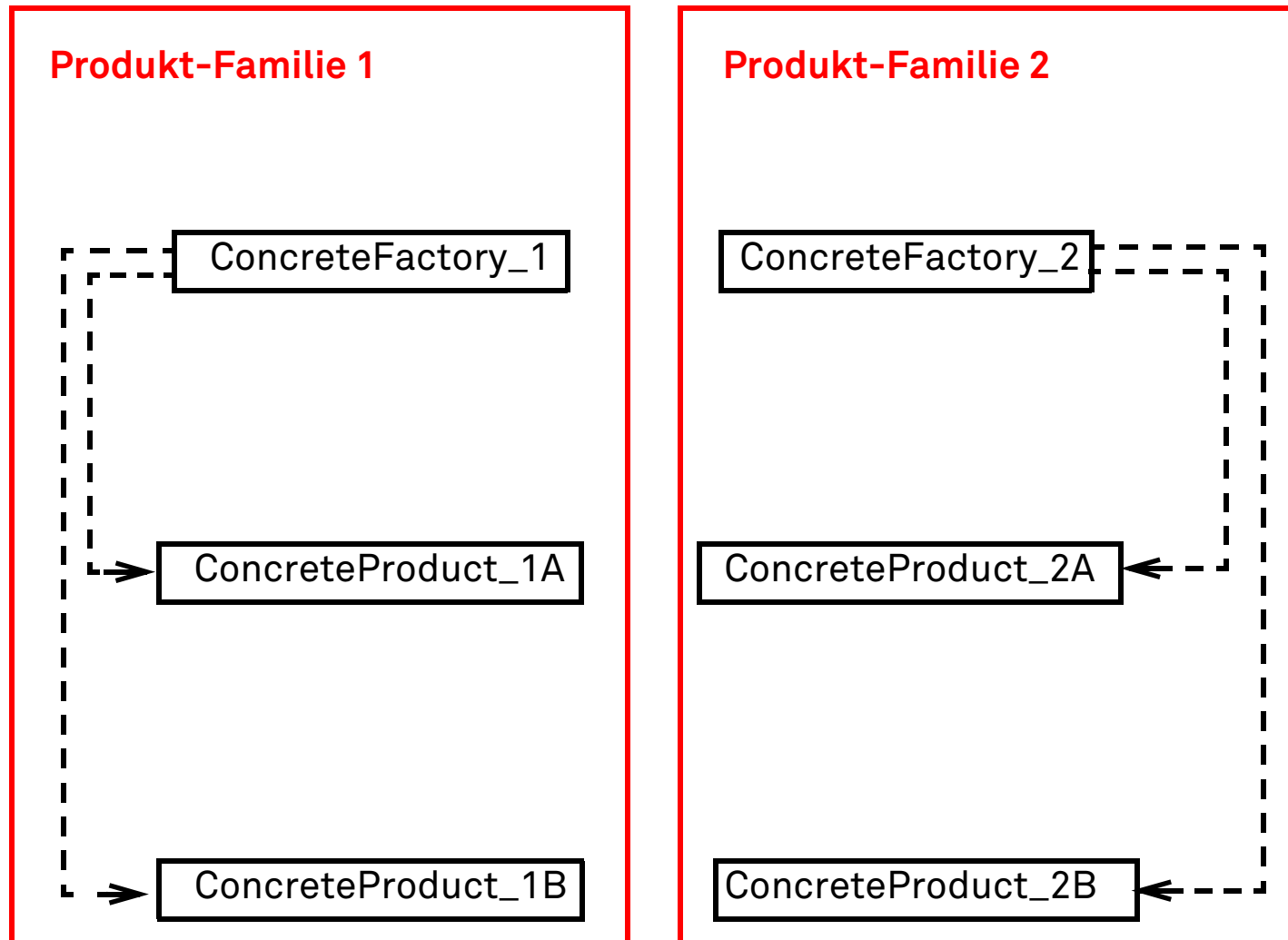
- ❑ Ziel: Einsatz eines Softwareprodukts in unterschiedlichen Anwendungsbereichen
Lösung: mehrere Familien mit Klassen für die Benutzeroberfläche mit unterschiedlicher Präsentation, eine dieser Familien wird ausgewählt und die zugehörigen Objekte werden erzeugt

- ❑ 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

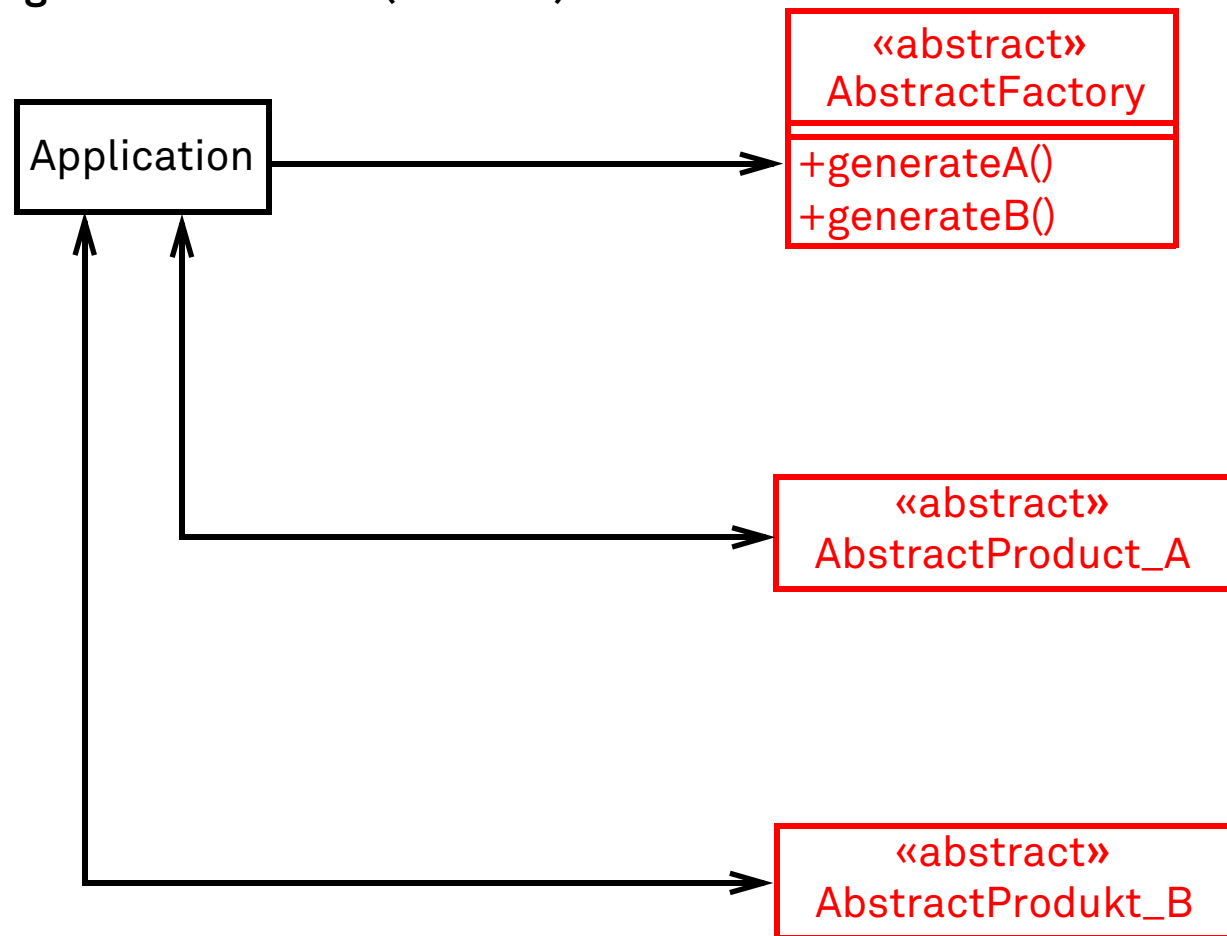
allgemeine Struktur (Objekte der Anwendung)



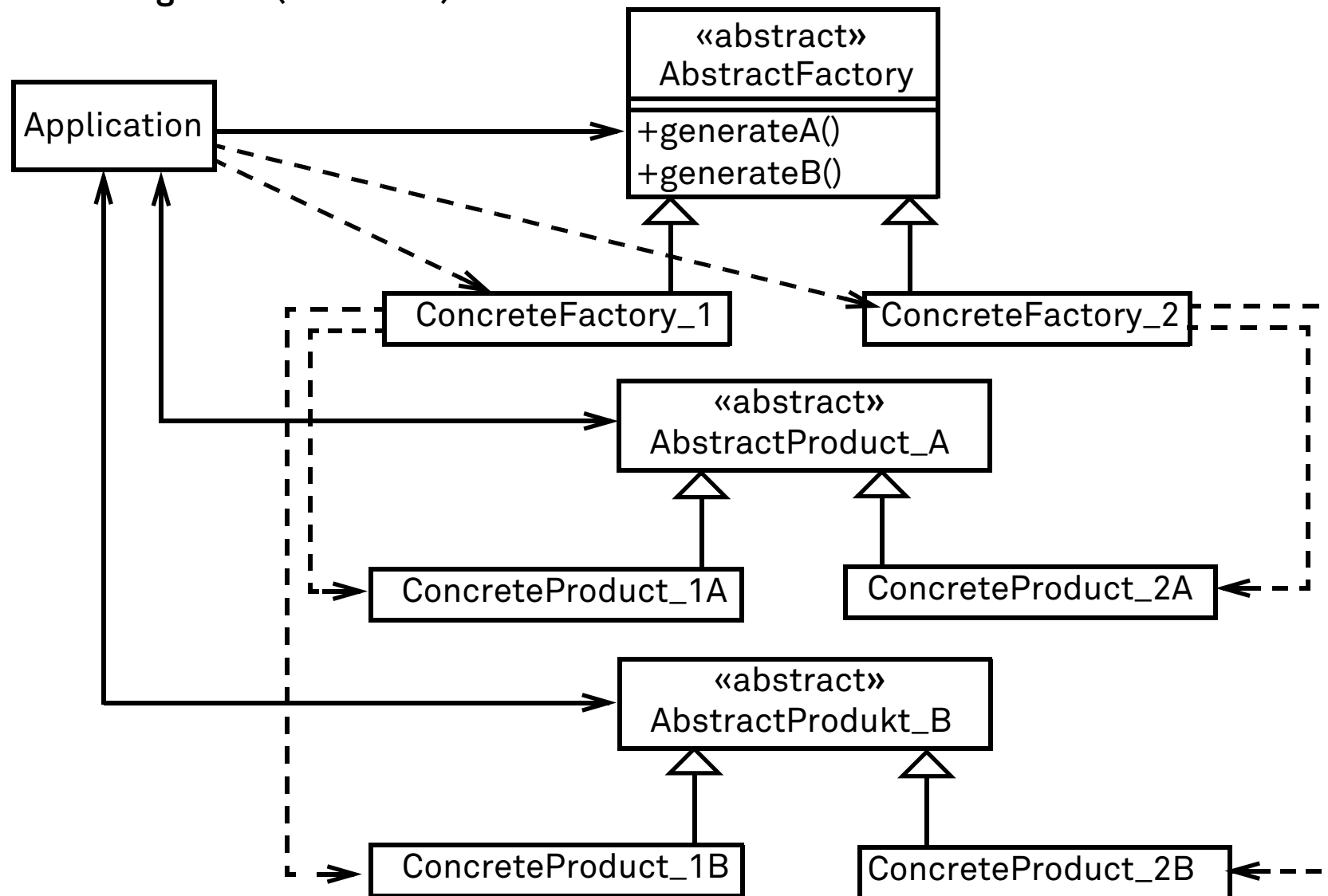
allgemeine Struktur (Klassendiagramm Produkt-Familien)



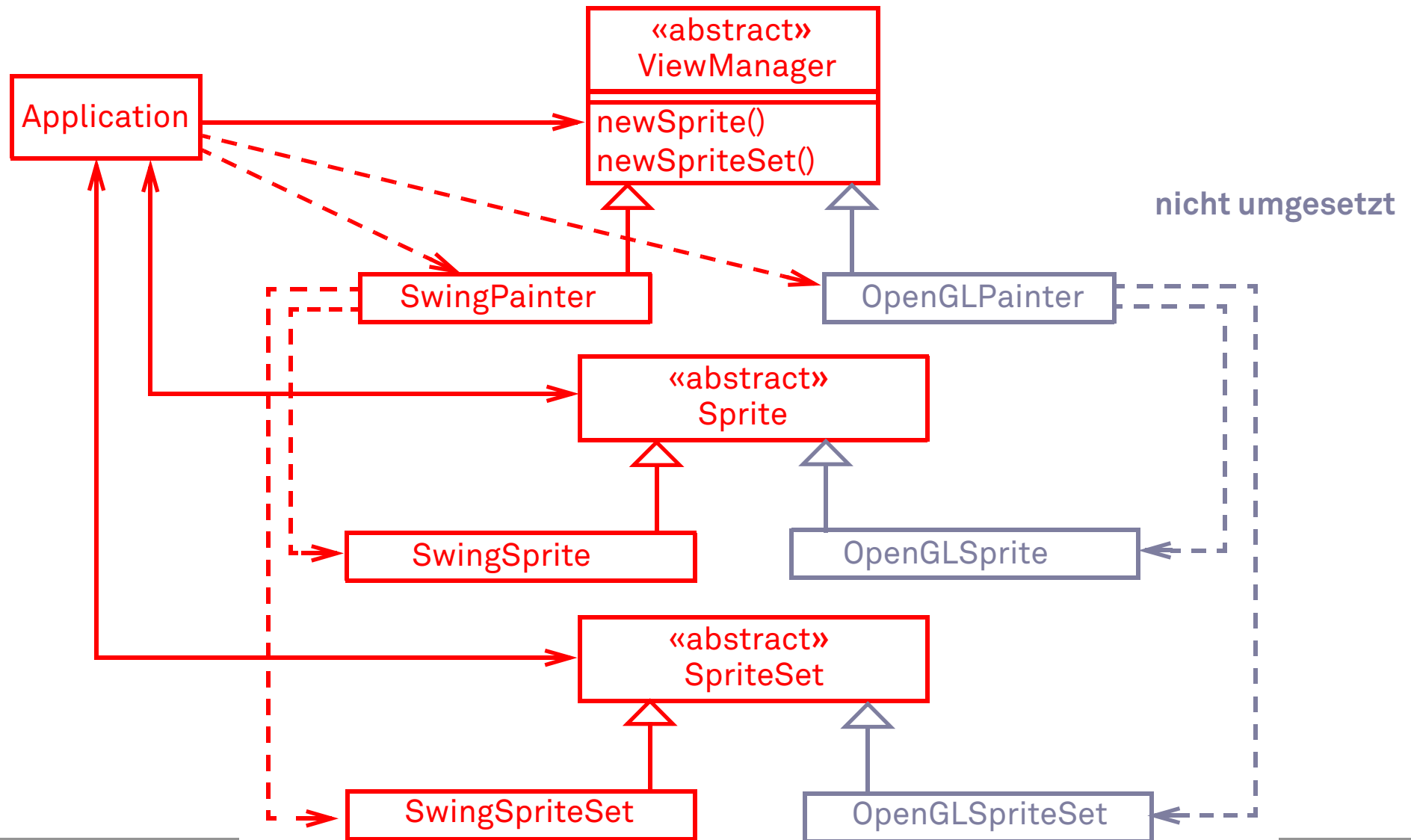
allgemeine Struktur (Klassen)



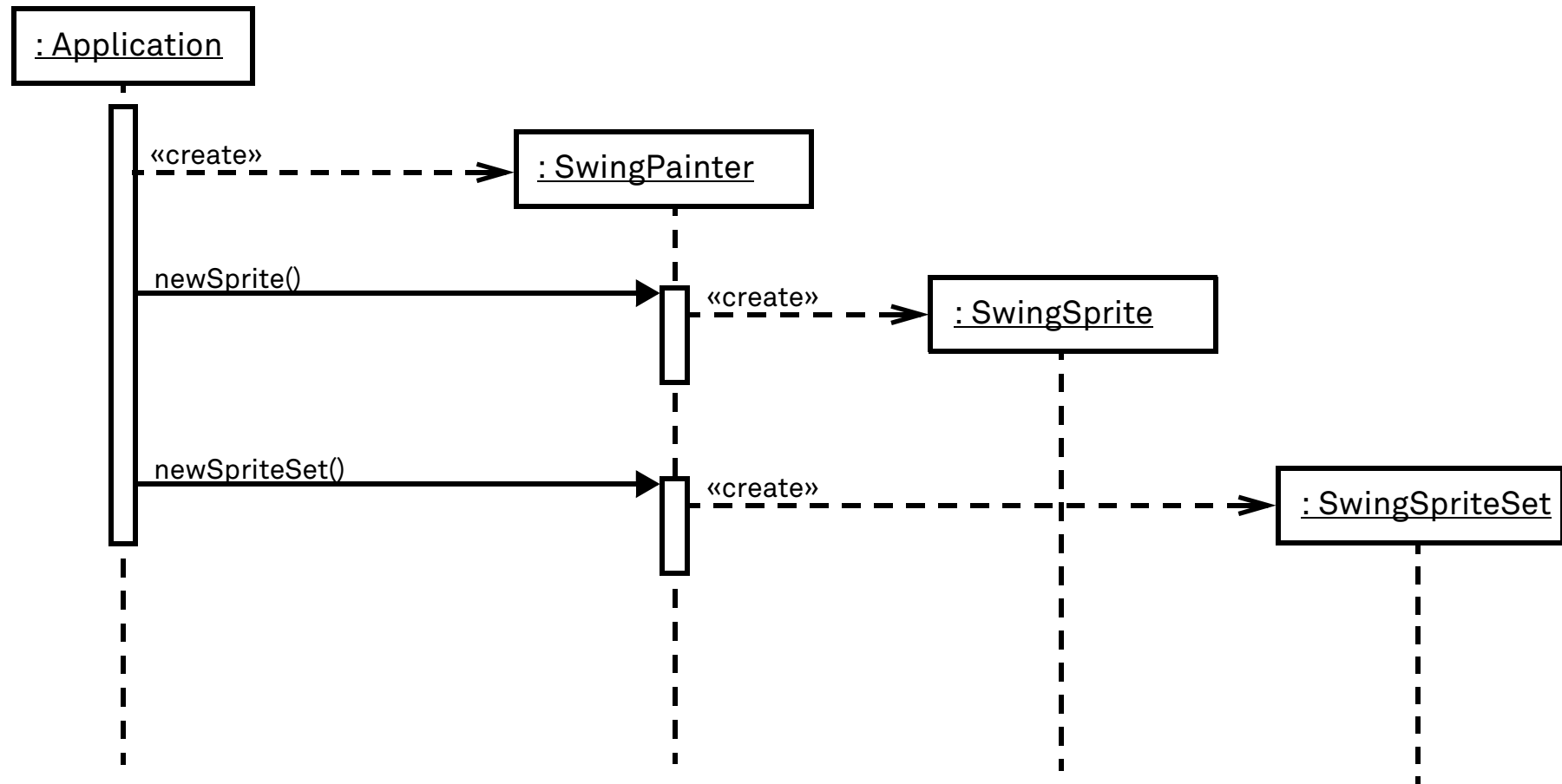
Klassendiagramm (Übersicht)



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
klassenbezogene Muster	Klassenadapter		Fabrikmethode
objektbezogene Muster	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

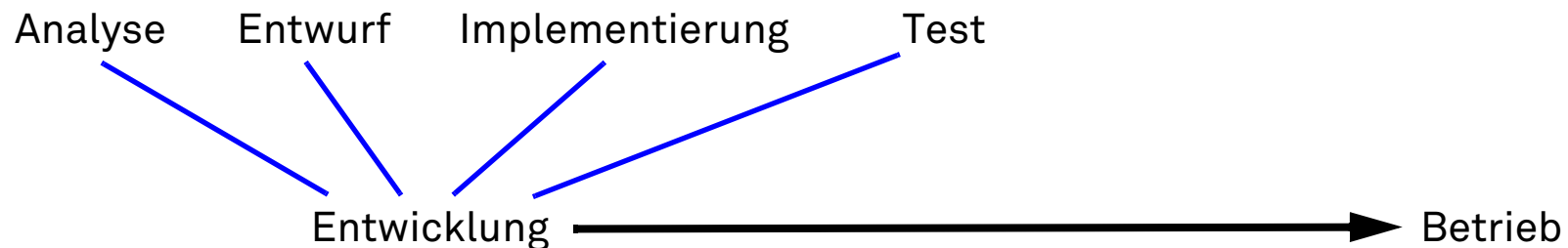
(Fortsetzung)

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)

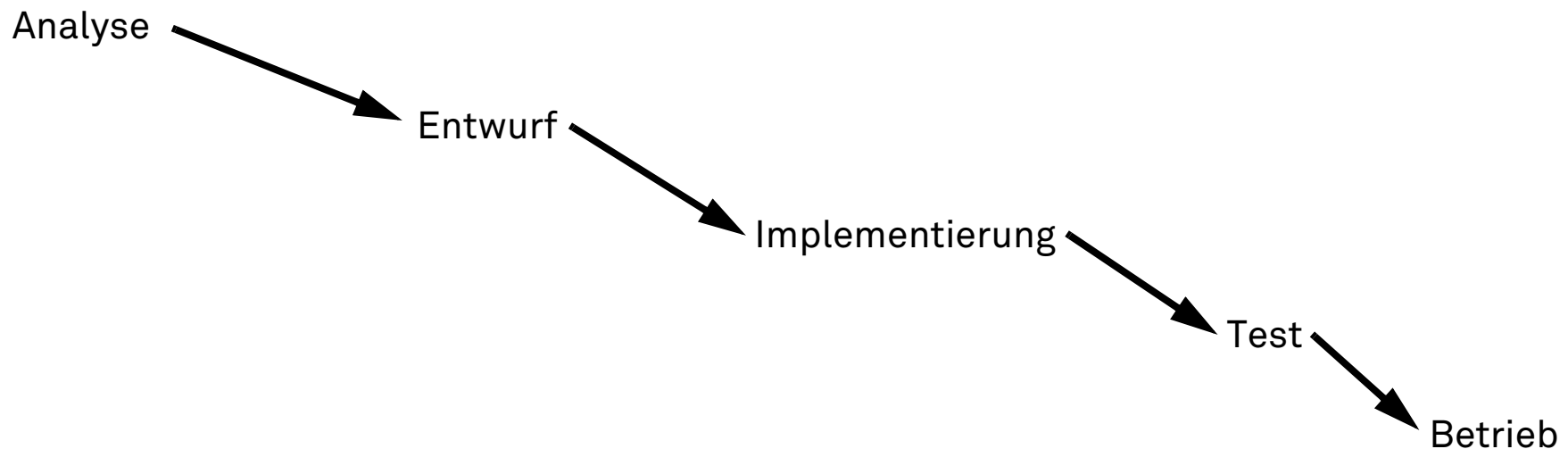


Literatur: Brandt-Pook, Hans; Kollmeier, Rainer: Softwareentwicklung kompakt und verständlich – Wie Softwaresysteme entstehen, S. 1-42
<http://www.springerlink.com/content/r66585/#section=77668&page=1&locus=0>

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)

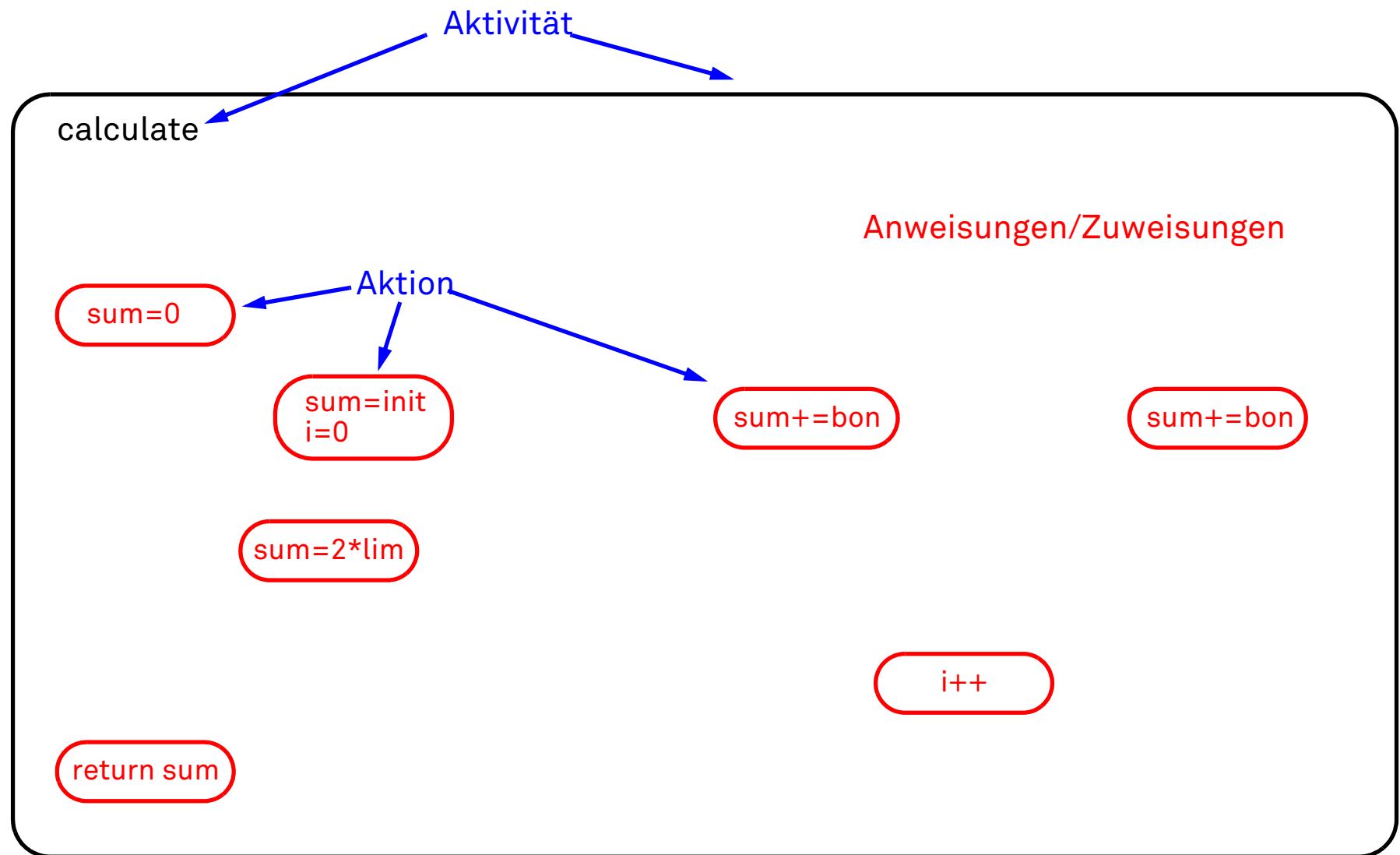
```
int calculate ( int end, int init, int lim, int bon )
{
    int sum = 0;
    if (end > 0)
    {
        sum = init;
        for (int i=0; i < end; i++)
        {
            sum += bon;
            if (sum > lim)
            {
                sum += bon;
            }
        }
        if (sum >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



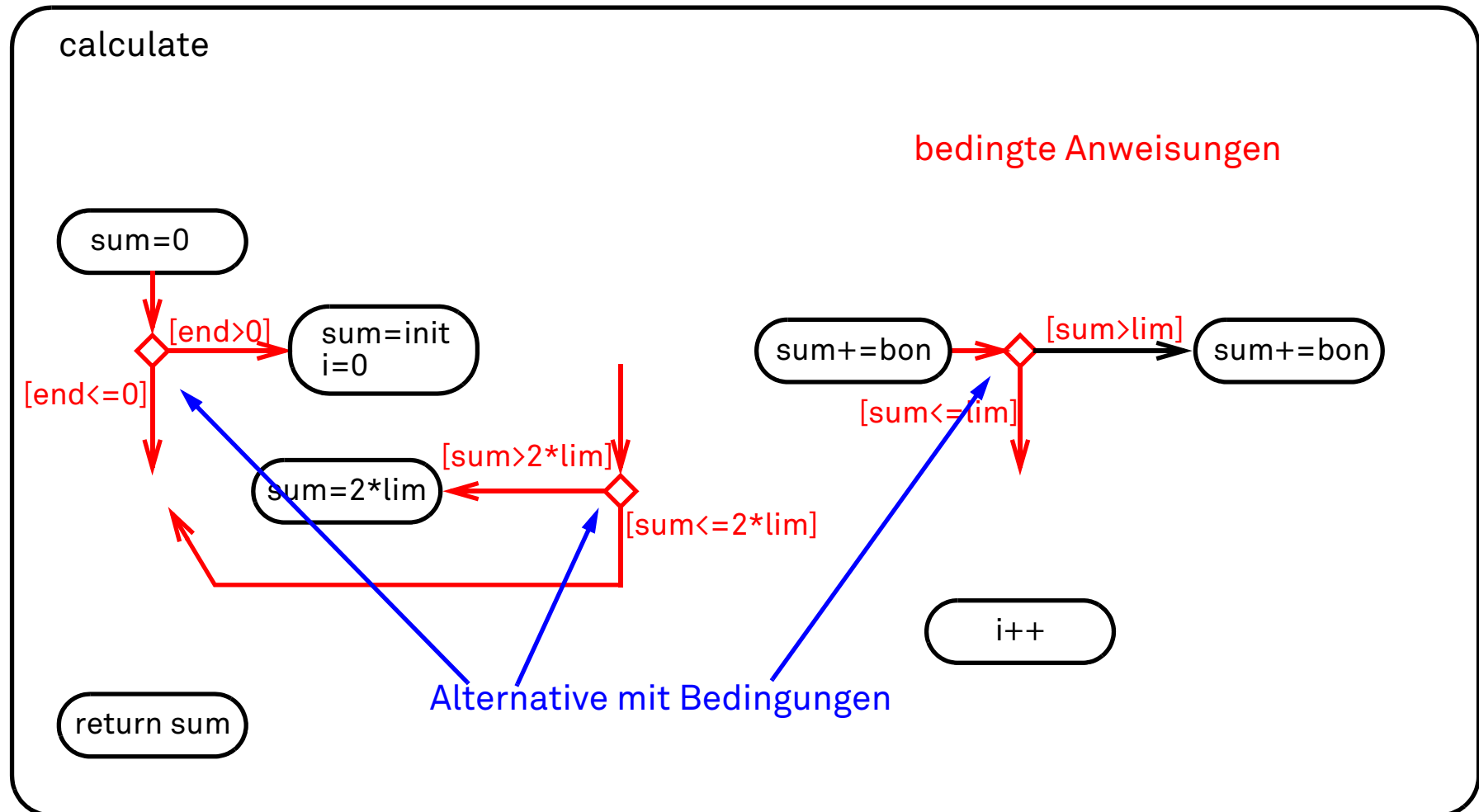
Beispiel (*irgendeine* Java-Methode)

```
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)



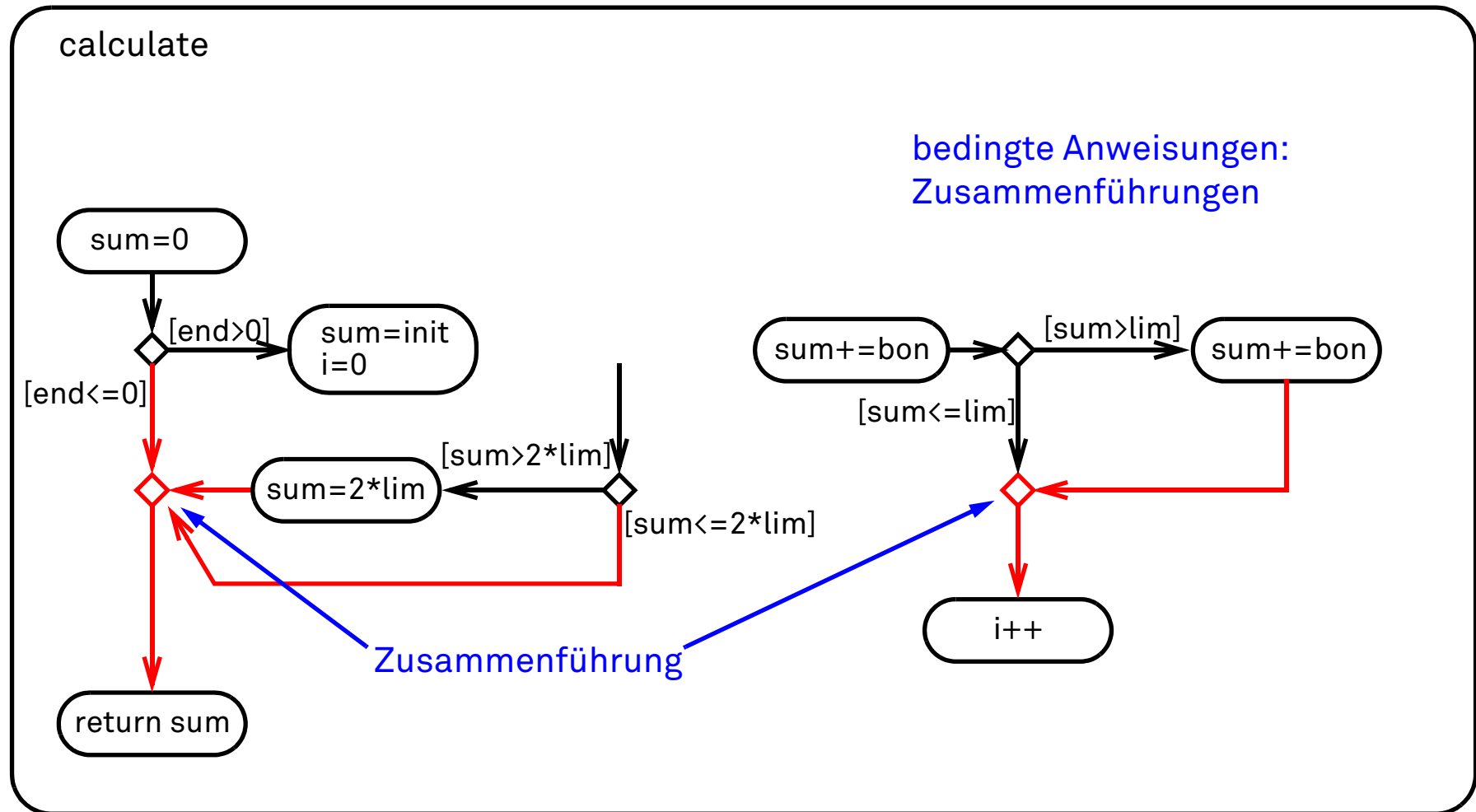
Beispiel (*irgendeine* Java-Methode)

```
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)



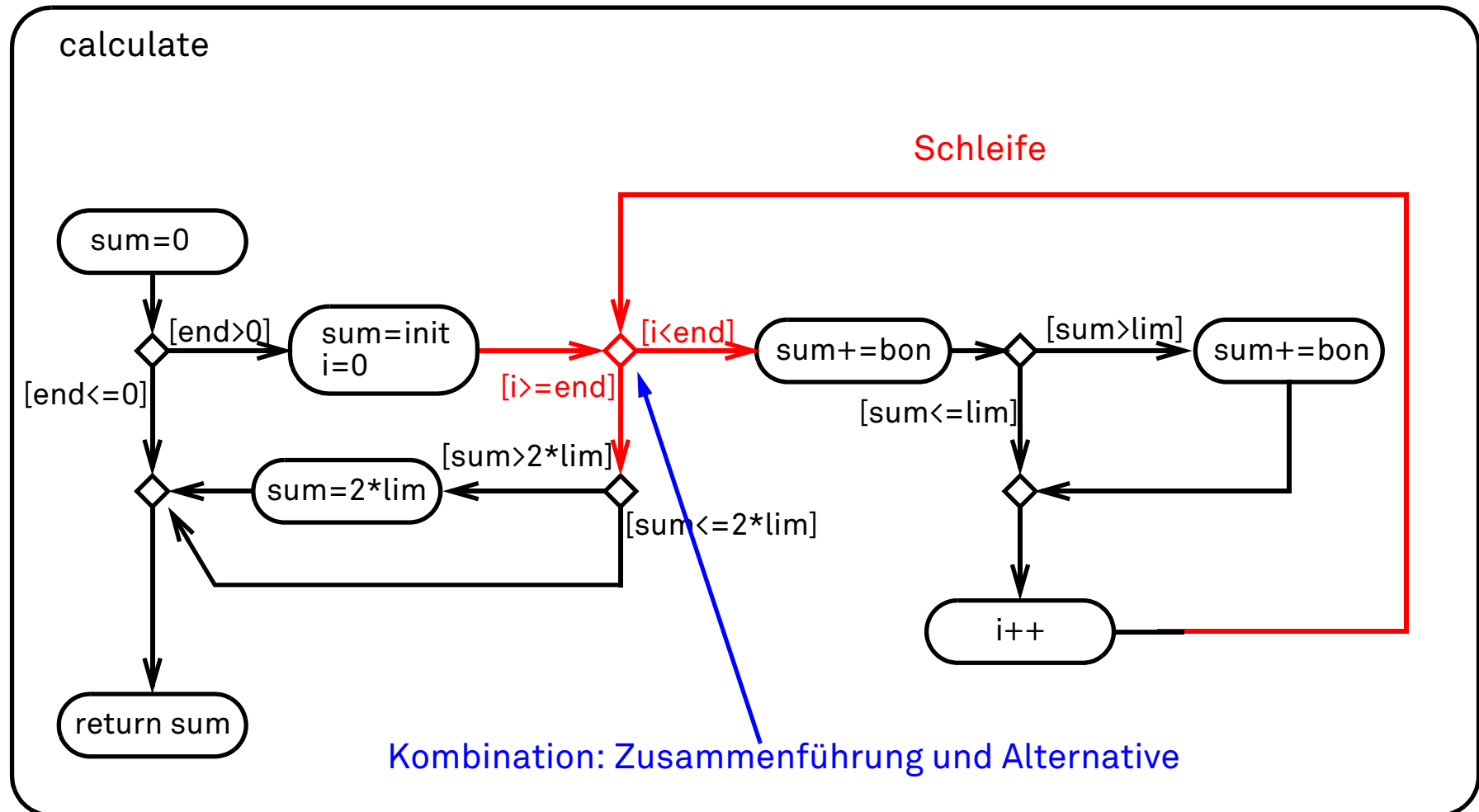
Beispiel (*irgendeine* Java-Methode)

```
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

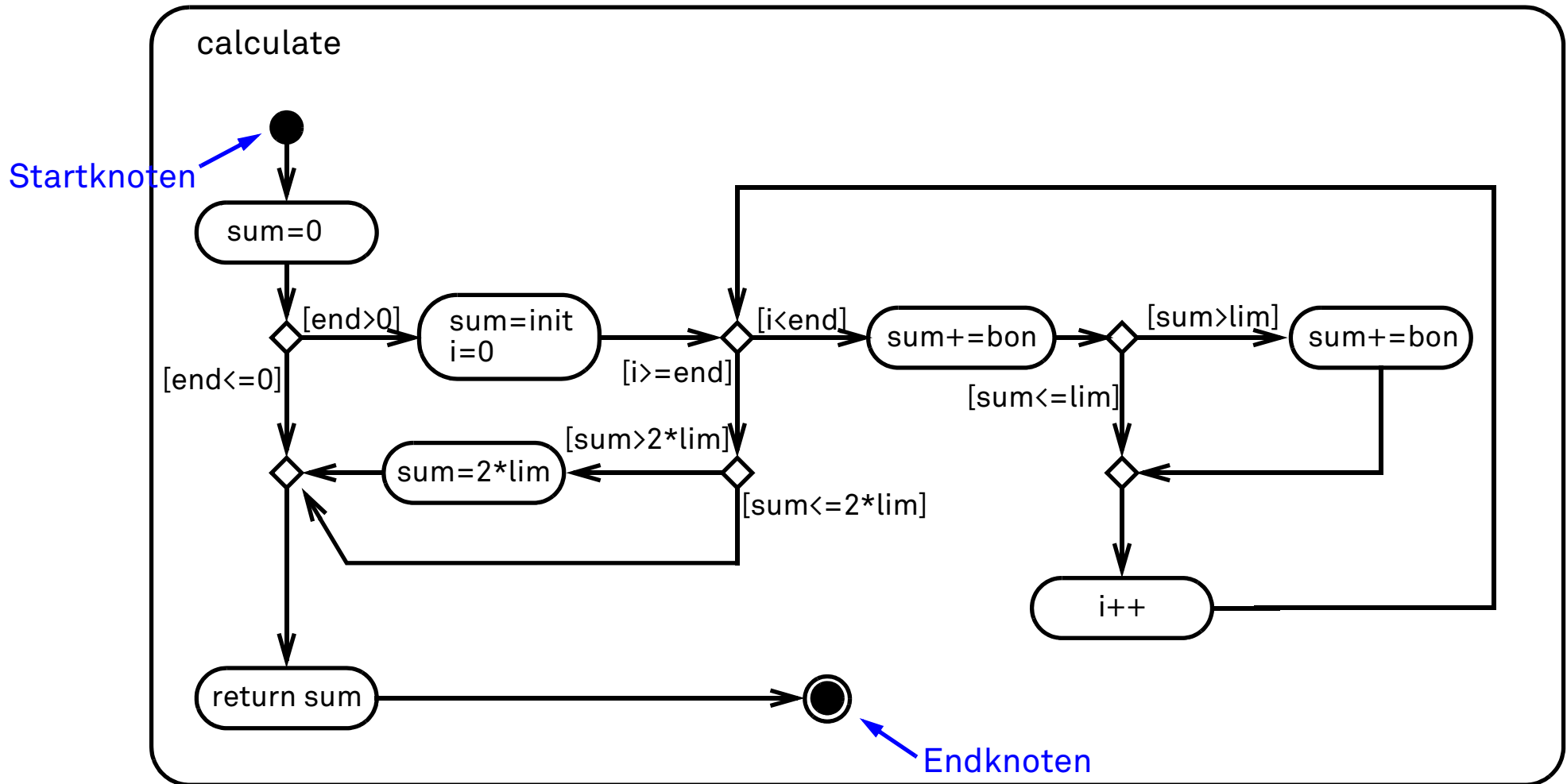
Beispiel (irgendeine Java-Methode)

(Fortsetzung)



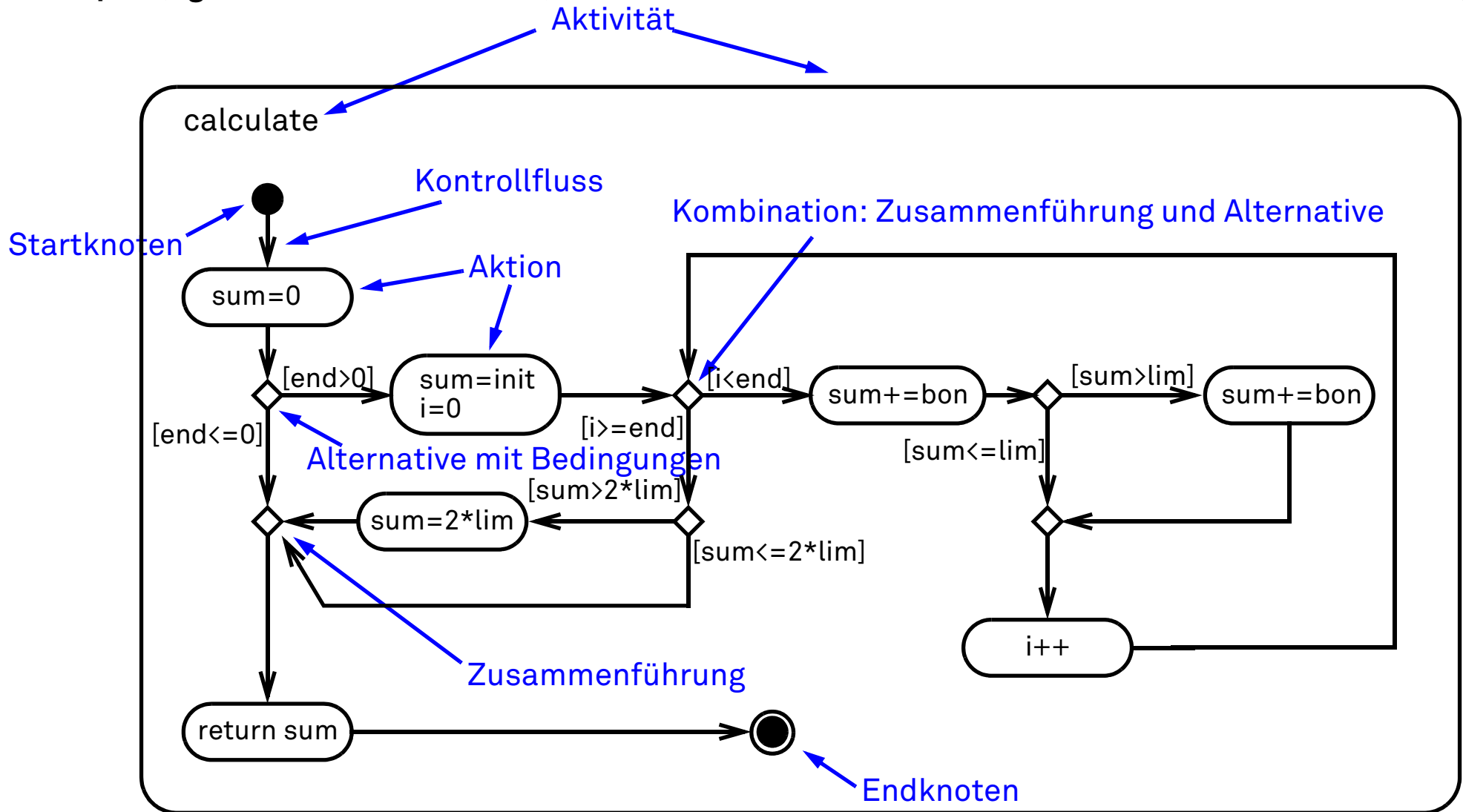
Beispiel (irgendeine Java-Methode)

(Fortsetzung)



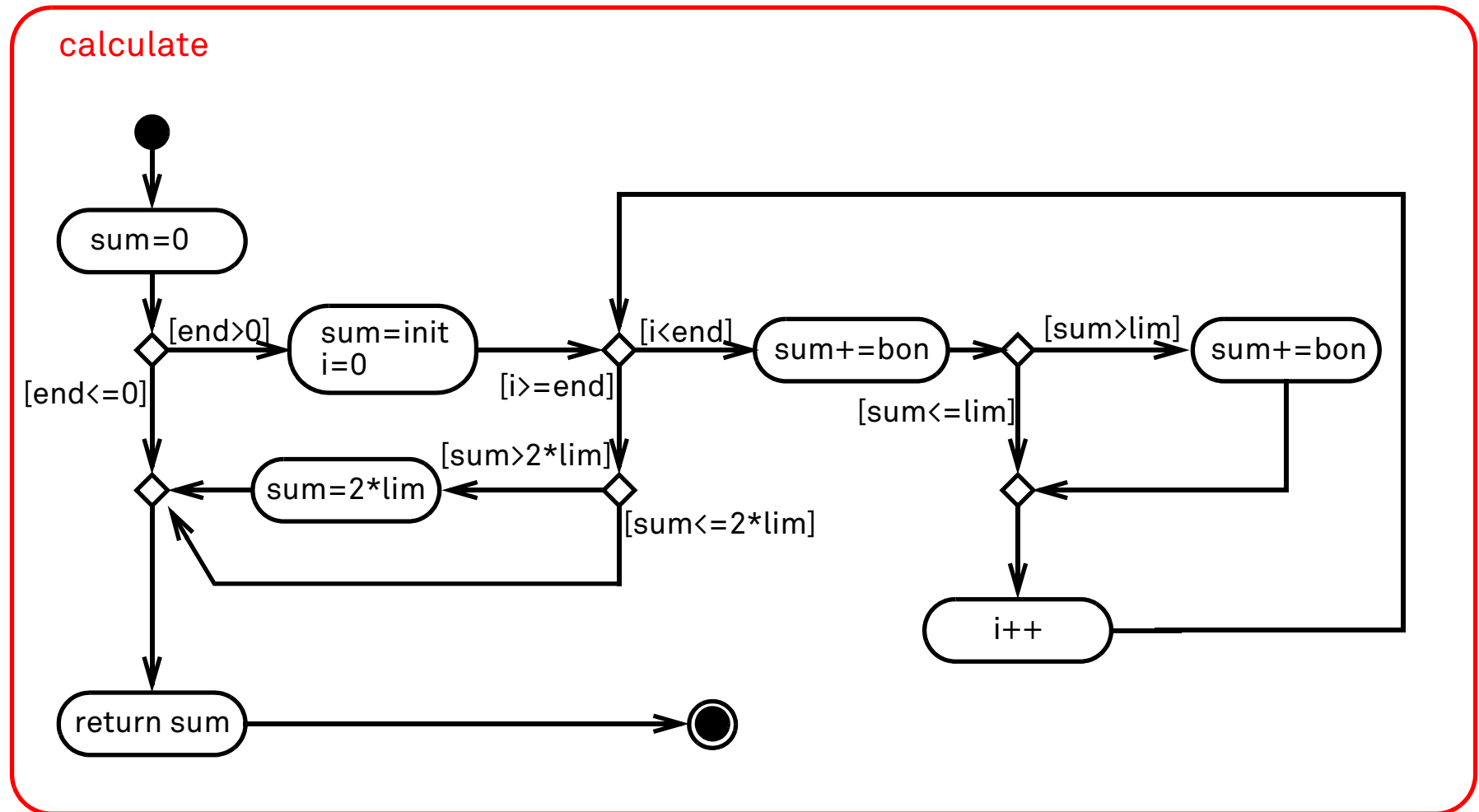
Beispiel (irgendeine Java-Methode)

(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.

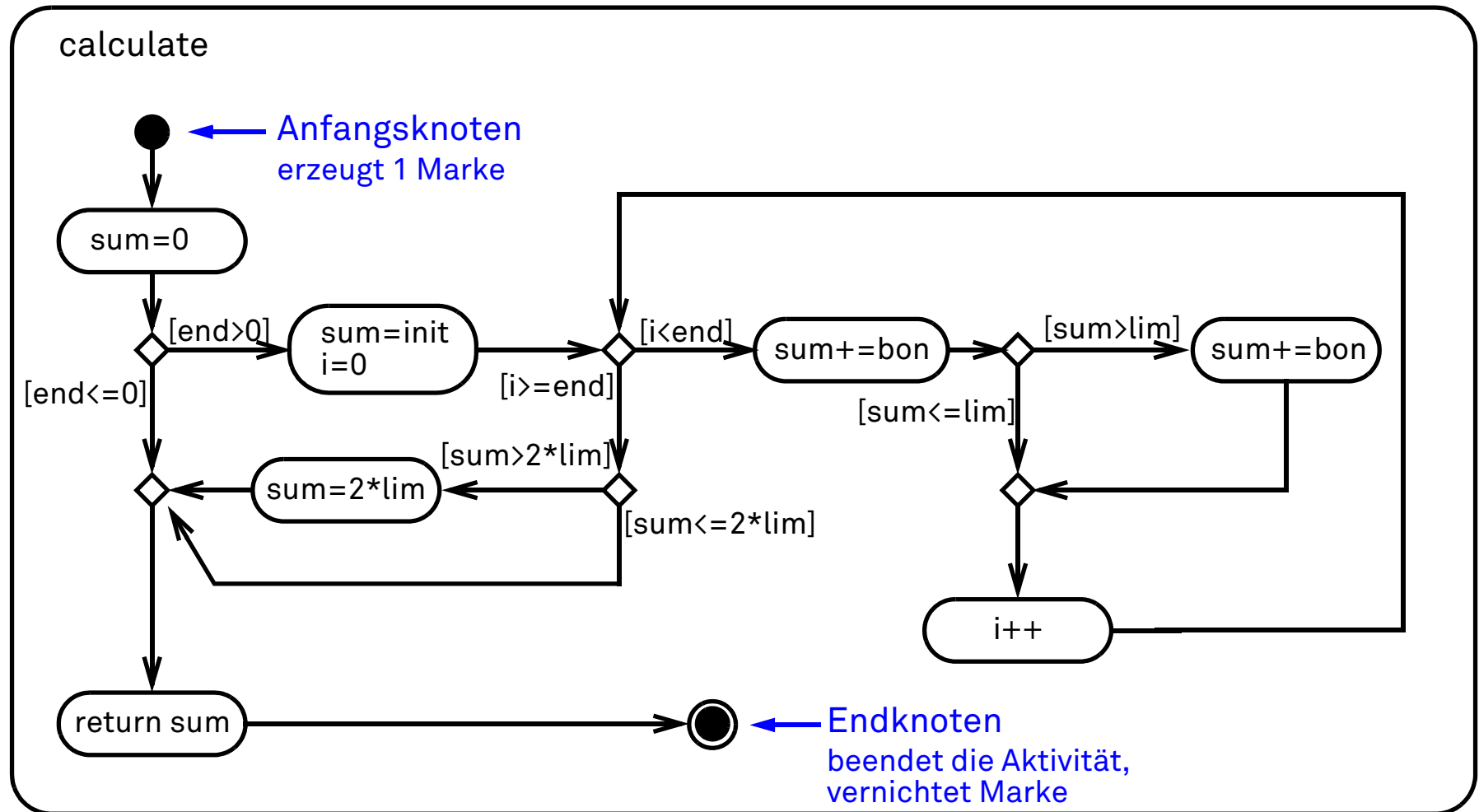


Anmerkungen zu Aktivitätsdiagrammen

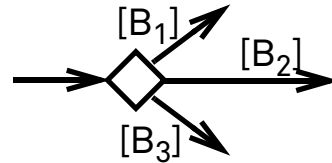
(Fortsetzung)

- ❑ 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

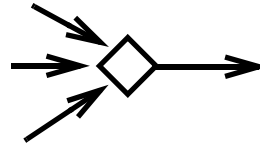


Verzweigungsknoten

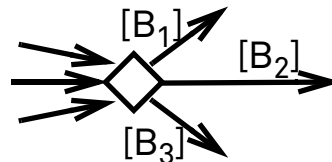


- ❑ Ein Verzweigungsknoten spaltet **eine** Eingangskante in **mehrere alternative** Ausgangskanten auf.
- ❑ Eine Bedingung B_i an einer Ausgangskante formuliert die Anforderungen, die das Passieren dieser Kante erlauben.
- ❑ Die Bedingungen der Ausgangskanten **müssen sich gegenseitig** ausschließen.
- ❑ 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



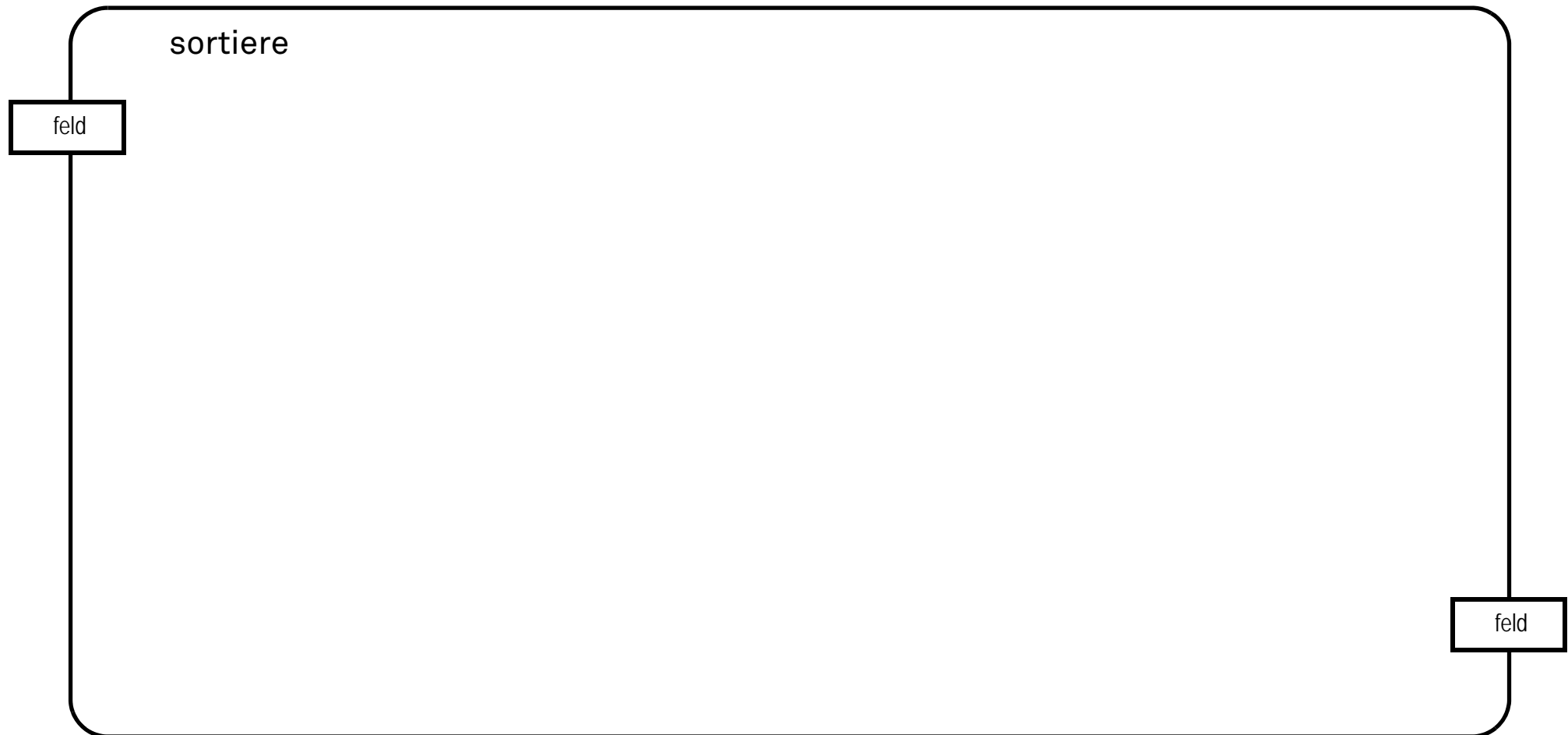
- ❑ Ein Verbindungsknoten führt mehrere Eingangskanten zu einer Ausgangskante zusammen.
- ❑ Die auf einer der Eingangskanten ankommende Marke wird auf die **einzigste** Ausgangskante weitergeleitet.
- ❑ Durch einen Verbindungsknoten werden keine Marken erzeugt oder vernichtet.
- ❑ Die Kombination aus 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)

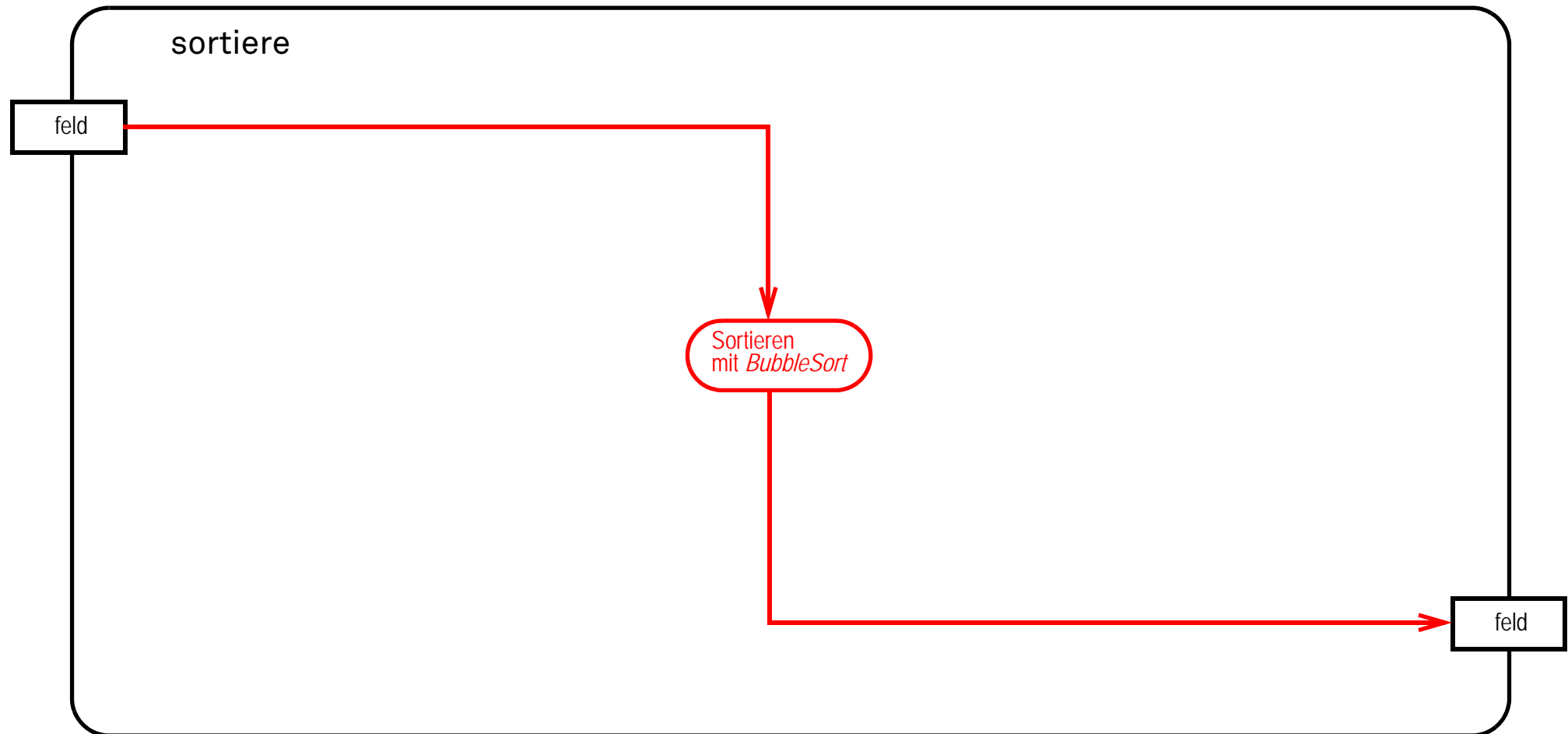


Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(gültiges Aktivitätsdiagramm, sehr abstrakt)

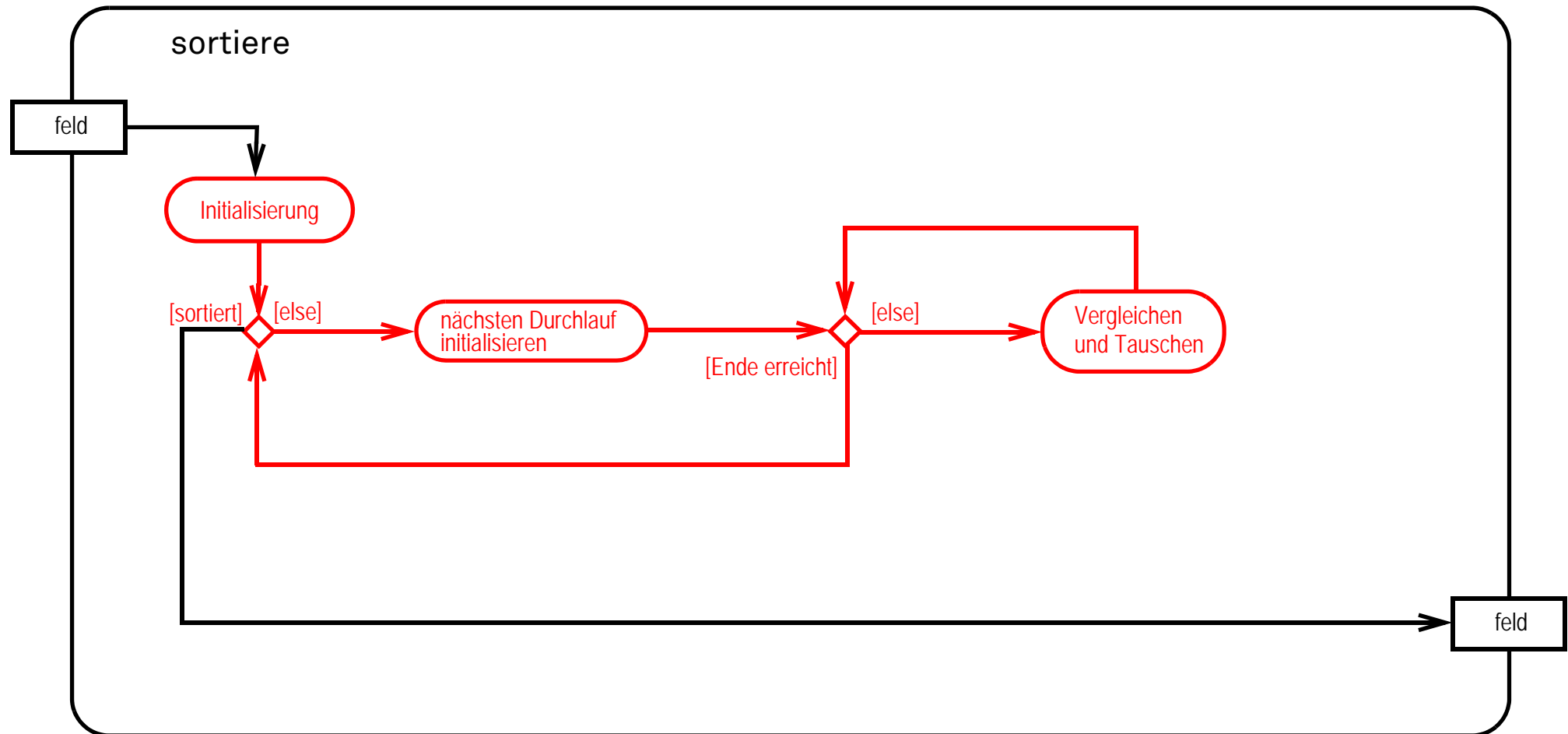


Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(Aktivitätsdiagramm, abstrakt)

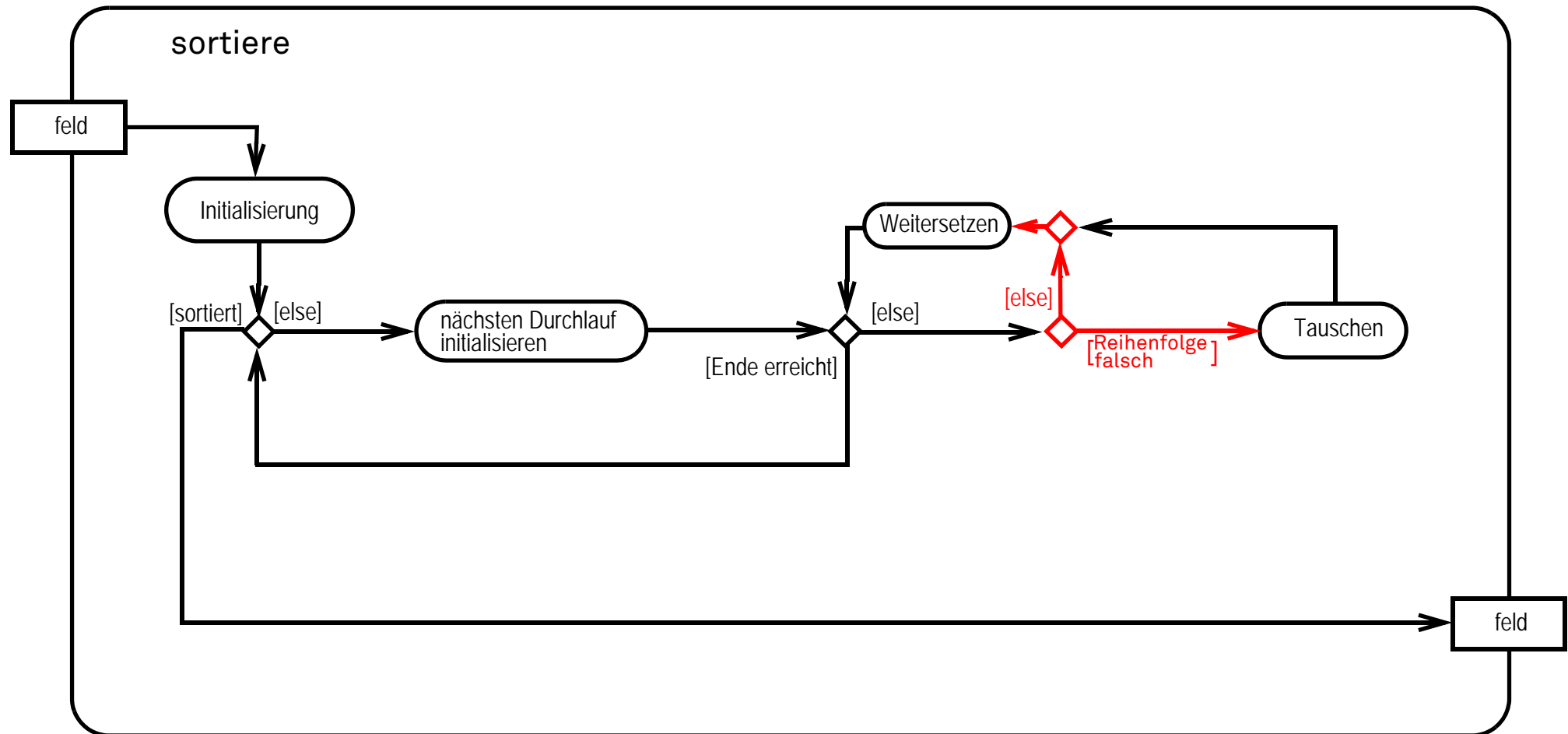


Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

(gültiges Aktivitätsdiagramm, Ablauf genau beschrieben)

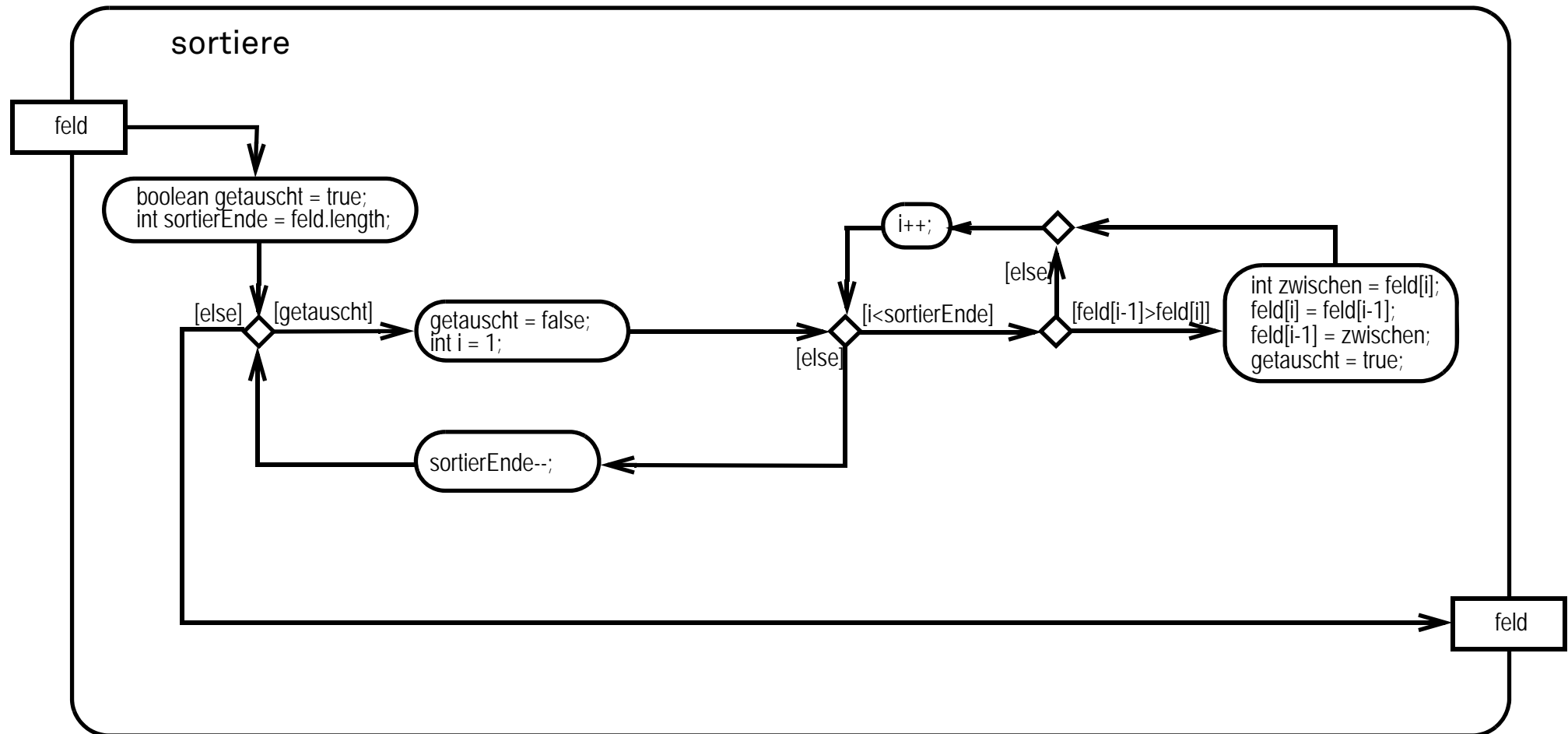


Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

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

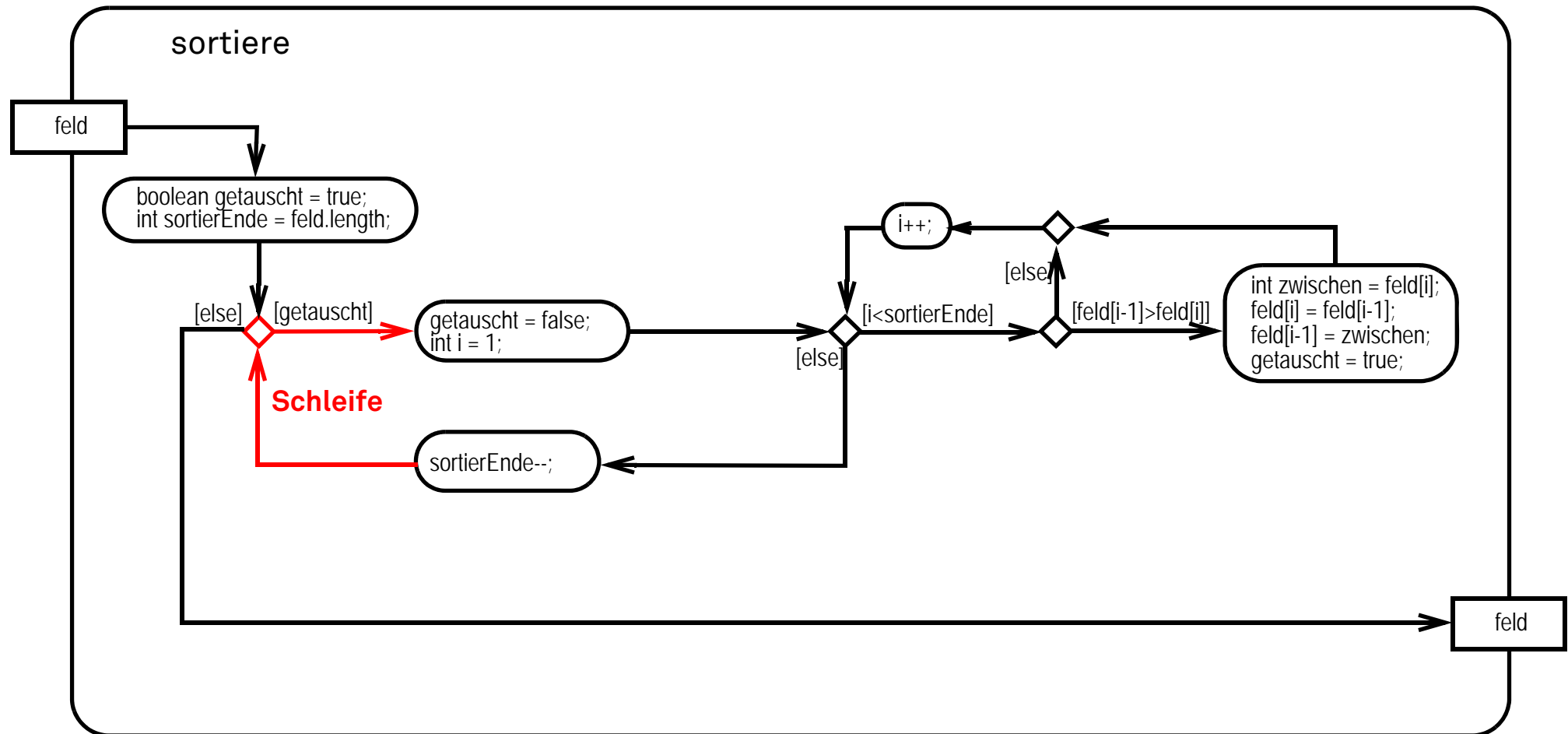


Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

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

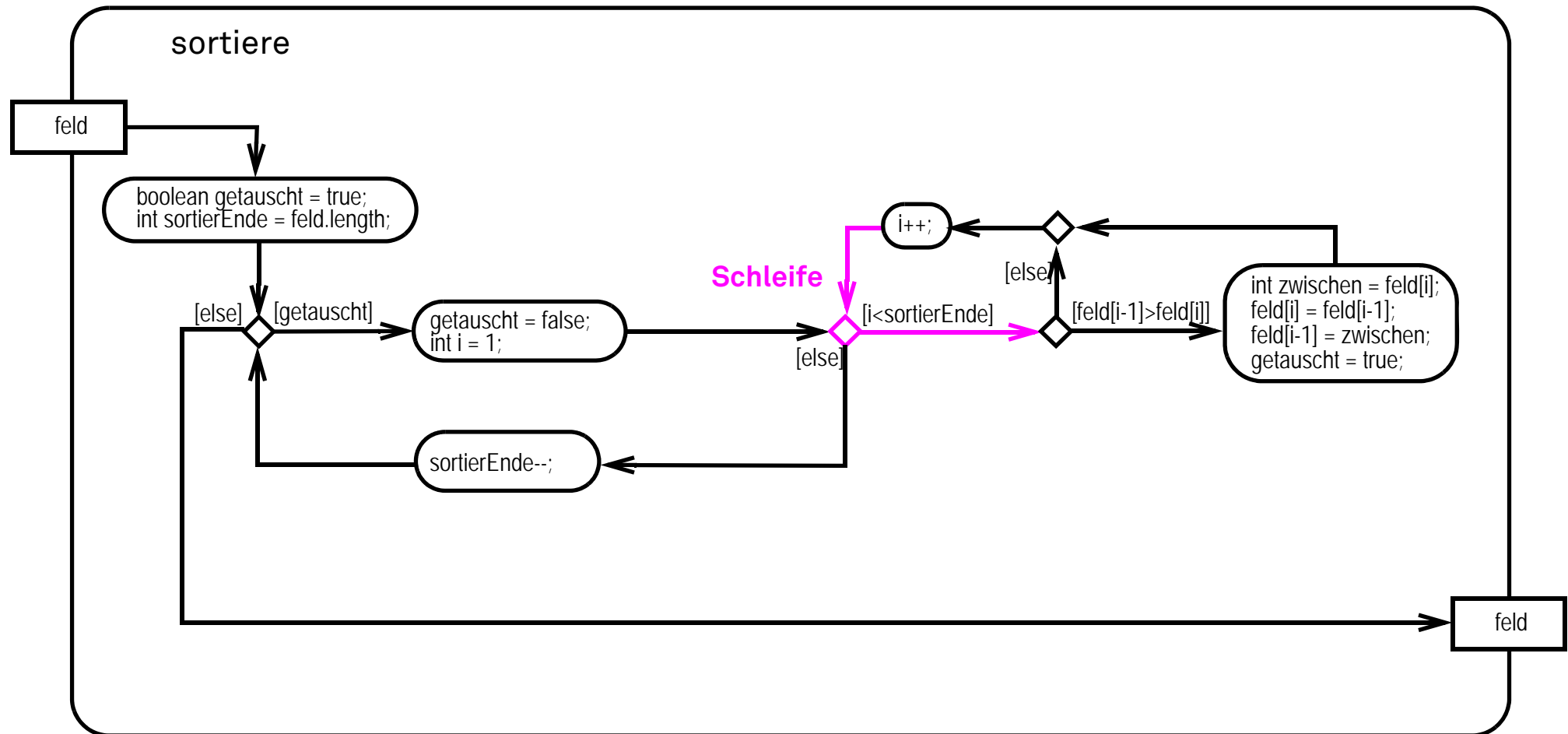


Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(Fortsetzung)

Aufgabe: Sortieren eines Feldes

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

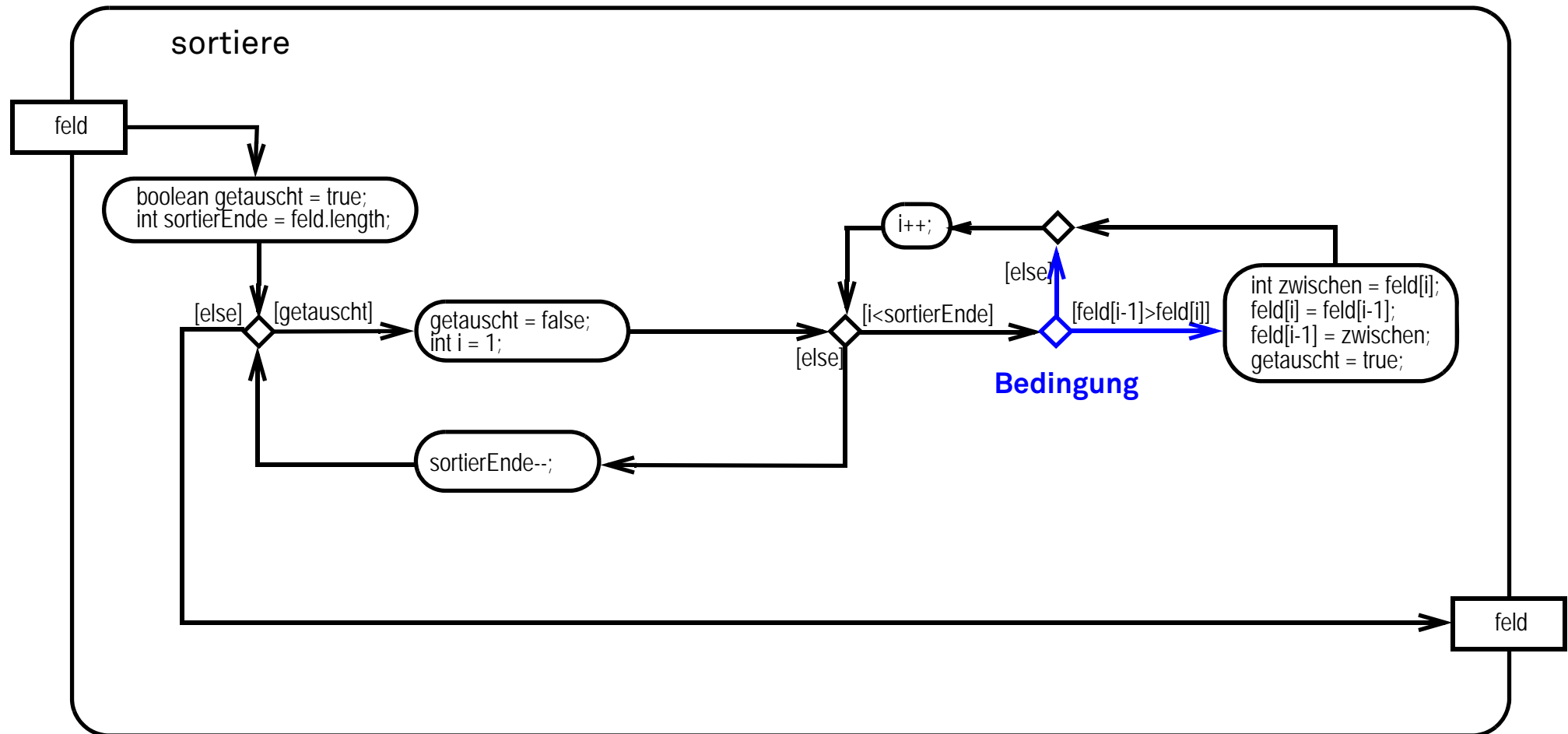


Schrittweise Entwicklung von Algorithmen mit Aktivitätsdiagramm – Beispiel

(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?

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 4.4: Funktionsorientierter Test (Black-Box-Test)

Bildung von Äquivalenzklassen für Testfälle

- ❑ Ein **Grenzwert** für die Eingabe
ist ein Wert, dessen minimale Änderung
zu einem **veränderten algorithmischen Verhalten** des getesteten Systems führt.
- ❑ Ein **Extremwert** für die Eingabe
ist ein Wert, dessen minimale Änderung
zu einer ungültigen Eingabe führt.
- ❑ Eine **Äquivalenzklasse**
wird durch die Menge von Eingabewerten gebildet, die zwischen
 - zwei Grenzwerten,
 - einem Grenz- und einem Extremwert,
 - zwei Extremwerten oder
 - jenseits eines Extremwertesliegen
und zu einem **gleichen algorithmischen Verhalten** des getesteten Systems führen.
- ❑ Der Begriff *Äquivalenzklasse* wird hier nicht im Sinne der gleichnamigen mathematische Definition verwendet.

Äquivalenzklassenbasierter Test

Ablauf:

- ❑ Bestimme alle Äquivalenzklassen für die möglichen Eingaben.
- ❑ Wähle **einen** Repräsentanten aus jeder Äquivalenzklasse als Testfall aus, der nicht Grenz- oder Extremwert ist.
- ❑ Teste alle so bestimmten Testfälle.
- ❑ Teste mit **allen** Grenz- und Extremwerten.

- ❑ Probleme bei der Bestimmung der Äquivalenzklassen:
 - Bestimmung setzt geordnete Mengen von Eingabewerten voraus .
 - Über Intervallen dieser Werte wird ein gleiches Verhalten erwartet.
 - Eine geeignet interpretierbare Beschreibung des algorithmischen Verhaltens des Systems wird für jede mögliche Eingabe benötigt.

Beispiel *Rabattsystem einer Boutique*

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert}$	200,00	150,00

Beispiel *Rabattsystem einer Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert} \leq \text{maximaler Wert}$	200,00	150,00
4 (ungültig)	$\text{minimaler Wert} \leq \text{Warenwert} < 0$	-15,00	Fehler

- Die Äquivalenzklasse 3 gibt möglicherweise einen Hinweis auf einen Fehler in der Spezifikation:
Die funktionale Beschreibung enthält keine Angabe zu einer oberen Grenze des Warenwerts, bis zu der Rabatt gewährt werden soll.

Beispiel *Rabattsystem einer Boutique*

(Fortsetzung)

Funktionale Spezifikation:

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert} \leq \text{maximaler Wert}$	200,00	150,00
4 (ungültig)	$\text{minimaler Wert} \leq \text{Warenwert} < 0$	-15,00	Fehler
5 (ungültig)	$\text{Warenwert} < \text{minimaler Wert}$		
6 (ungültig)	$\text{Warenwert} > \text{maximaler Wert}$		

– Eventuell können für die Äquivalenzklassen 5 und 6 keine Werte eingegeben werden.

Beispiel *Rabattsystem einer Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert} \leq \text{maximaler Wert}$	200,00	150,00
4 (ungültig)	$\text{minimaler Wert} \leq \text{Warenwert} < 0$	-15,00	Fehler

Testfälle für Grenzwerte

Grenzwert	Repräsentanten für Testfälle
0	-0,01; 0,00
25	25,00; 25,01
150	149,99; 150,00

Beispiel *Rabattsystem einer Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert} \leq \text{maximaler Wert}$	200,00	150,00
4 (ungültig)	$\text{minimaler Wert} \leq \text{Warenwert} < 0$	-15,00	Fehler

Testfälle für Extremwerte

Extremwert	Repräsentanten für Testfälle (Eingabe eventuell nicht möglich)
minimale Zahl	minimale Zahl; minimale Zahl-0,01
maximale Zahl	maximale Zahl; maximale Zahl+0,01

Analyse des vorangehenden Beispiels

- ❑ Die Zahl der Äquivalenzklassen hängt im Beispiel nur von **einem** Parameter ab:
Bei mehr als einem Parameter wird die Bildung der Äquivalenzklassen aufwändig.
- ❑ Hängt das Ergebnis der Ausführung einer Methode nicht nur von Parametern sondern auch vom Zustand (also den Werten der Attribute) des ausführenden Objekts ab, so muss der Zustand bei der Bildung von Äquivalenzklassen ebenfalls berücksichtigt werden:

zustandsbasierter Test

Bei einem äquivalenzklassenbasierter Test
mit mehreren Parametern oder Zuständen,
die voneinander abhängen,
müssen die Testfälle aus **Entscheidungstabellen** abgeleitet werden.

Beispiel *Happy Hour für Kunden der Boutique*

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Analyse:

Es müssen zuerst die möglichen Kombinationen von Bedingungen und die zugehörigen Aktionen ermittelt und aufgelistet werden.

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Entscheidungstabelle:

Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Entscheidungstabelle:

Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein
Aktionen				
7% Rabatt				
5% Rabatt				
12% (=5%+7%) Rabatt				
regulärer Preis				

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Entscheidungstabelle:

Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein
Aktionen				
7% Rabatt		X		
5% Rabatt				
12% (=5%+7%) Rabatt				
regulärer Preis				

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Entscheidungstabelle:

Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein
Aktionen				
7% Rabatt		X		
5% Rabatt			X	
12% (=5%+7%) Rabatt	X			
regulärer Preis				X

Beispiel *Happy Hour* für Kunden der Boutique

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Jede Spalte beschreibt eine Äquivalenzklasse!

Entscheidungstabelle:



Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein
Aktionen				
7% Rabatt		X		
5% Rabatt			X	
12% (=5%+7%) Rabatt	X			
regulärer Preis				X

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Anmerkung:

Entscheidungstabellen mit vielen Bedingungen werden schnell unübersichtlich,
häufig können aber **irrelevante** Kombinationen gestrichen werden!

Das nächste Beispiel zeigt, wie vier Äquivalenzklassen zusammenfallen.

Beispiel *Happy Hour* für Kunden der Boutique (Modifikation)

Erweiterung der funktionalen Beschreibung:

Zusätzlich gibt es Kundenkarten mit 15% Rabatt ohne die Möglichkeit von Zusatzrabatten.

Entscheidungstabelle:

Bedingungen					
Kundenkarte	ja	nein	nein	nein	nein
Gutschein	egal	ja	ja	nein	nein
zwischen 8 und 10 Uhr	egal	ja	nein	ja	nein
Aktionen					
7% Rabatt			X		
5% Rabatt				X	
12% (=5%+7%) Rabatt		X			
regulärer Preis					X
15% Rabatt	X				

weiteres Beispiel – Berechnung der Binominalkoeffizienten

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

Dabei gilt:

$0! = 1$, $1! = 1$, $n! = (n-1)! \cdot n$, für negative Werte n ist $n!$ nicht definiert.

Ist die Berechnung nicht möglich, soll von `bin` die Meldung "error" ausgegeben und der Wert 0 zurückgegeben werden.

- ❑ Wie würde ein äquivalenzklassenbasierter Test von `bin` aussehen?
- ❑ Bestimmen Sie Äquivalenzklassen und daraus Testfälle mit Kombinationen von Werten für `n` und `k` an.

weiteres Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

Dabei gilt:

$0! = 1$, $1! = 1$, $n! = (n-1)! \cdot n$, für negative Werte n ist $n!$ nicht definiert.

Ist die Berechnung nicht möglich, soll von `bin` die Meldung "error" ausgegeben und der Wert 0 zurückgegeben werden.

- ❑ Wie würde ein äquivalenzklassenbasierter Test von `bin` aussehen?
- ❑ Bestimmen Sie Äquivalenzklassen und daraus Testfälle mit Kombinationen von Werten für n und k an.

Äquivalenzklassen:

- ❑ für gültige Werte muss gelten: $n \geq 0$, $k \geq 0$, $n \geq k$
- ❑ $n == 0$, $k == 0$ und $n \geq k$ begrenzen die gültigen Wertekombinationen
- ❑ Vorschläge für Testfälle:
(0,0), (7,0), (7,3), (7,7) für Ergebnisse ohne "error"-Meldung
(3,7), (-1,7), (7,-1), (-1,-1) für Ergebnisse mit "error"-Meldung
(MIN_Value, MIN_Value), (MIN_Value, MAX_Value), (MAX_Value, MIN_Value), (MAX_Value, MAX_Value)

Einordnung von Tests

Geht die Ermittlung von Testfällen

– wie in den vorangehenden Beispielen –

von der Beschreibung der erwarteten Funktionalität aus:

funktionsorientierter Test

(auch **black-box-Test**, da die konkrete Implementierung nicht betrachtet wird)

Eigenschaften des funktionsorientierten Tests:

- Die Übereinstimmung von Spezifikation und Implementierung wird überprüft.

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 4.5: Strukturorientierter Test (White-Box-Test)

Einordnung von Tests

Geht die Ermittlung von Testfällen

– wie in den vorangehenden Beispielen –

von der Beschreibung der erwarteten Funktionalität aus:

funktionsorientierter Test

(auch **black-box-Test**, da die konkrete Implementierung nicht betrachtet wird)

Eigenschaften des funktionsorientierten Tests:

- ❑ Die Übereinstimmung von Spezifikation und Implementierung wird überprüft.

aber:

- ❑ Bei unzureichender Beschreibung ist ein funktionsorientierter Test nicht möglich.
Beispiel: falsche Uhrzeit für morgendlichen Rabatt am Tag der Sommerzeitumstellung
- ❑ Selten auftretende Fehler werden nur zufällig aufgedeckt.
Beispiel: 20% Rabatt für eine bestimmte Kundenkarte
- ❑ Zusätzlich in die Implementierung eingefügte Funktionen werden nur zufällig aufgedeckt.
Beispiel: 20% Rabatt für eine bestimmte Kundenkarte

Ein funktionsorientierter Test reicht also nicht aus, um Vertrauen in die Software zu schaffen!

Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

```
public static long bin (int n, int k) {
    if (n < 0 || k < 0 || n > 50) { System.out.print(" error "); return 0; }
    // Optimierungen
    if (k == 0 || k == n) return 1;
    if (k == 1 || k == n - 1) return k;
    if (2 * k > n) k = n - k;
    // Berechnung
    if (n < k) { System.out.print(" error "); return 0; }
    long c=1, d=1;
    for (int i=0; i < k; ++i) {
        c *= n - i;
        d *= i + 1;
    }
    return c / d;
}
```

Werden alle Fehler dieser Implementierung
durch die geplanten Testfälle erkannt?
(siehe Folie 406)

Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

```
public static long bin (int n, int k) {
    if (n < 0 || k < 0 || n > 50) { System.out.print(" error "); return 0; }
    // Optimierungen
    if (k == 0 || k == n) return 1;
    if (k == 1 || k == n - 1) return k;
    if (2 * k > n) k = n - k;
    // Berechnung
    if (n < k) { System.out.print(" error "); return 0; }
    long c=1, d=1;
    for (int i=0; i < k; ++i) {
        c *= n - i;
        d *= i + 1;
    }
    return c / d;
}
```

Werden alle Fehler dieser Implementierung
durch die geplanten Testfälle erkannt?
(siehe Folie 406)

nein

Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode **long** bin(**int** n, **int** k) soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

```
public static long bin (int n, int k) {
    if (n < 0 || k < 0 || n > 50) { System.out.print(" error "); return 0; }
    // Optimierungen
    if (k == 0 || k == n) return 1;
    if (k == 1 || k == n - 1) return k;
    if (2 * k > n) k = n - k;
    // Berechnung
    if (n < k) { System.out.print(" error "); return 0; }
    long c=1, d=1;
    for (int i=0; i < k; ++i) {
        c *= n - i;
        d *= i + 1;
    }
    return c / d;
}
```

Wertebereich von long wird bei der Berechnung überschritten

in diesem Sonderfall muss der Wert von n zurückgegeben werden

Abfrage kommt zu spät:
so können in Optimierung negative
Werte für k entstehen

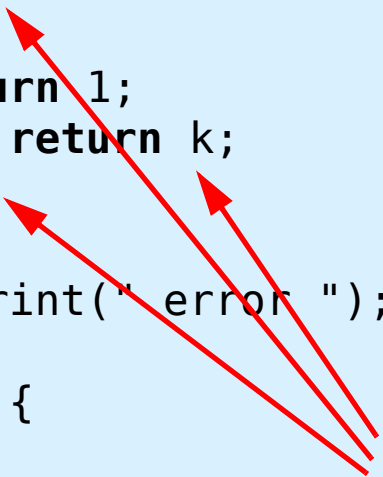
3 Fehler

Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

```
public static long bin (int n, int k) {
    if (n < 0 || k < 0 || n > 50) { System.out.print(" error "); return 0; }
    // Optimierungen
    if (k == 0 || k == n) return 1;
    if (k == 1 || k == n - 1) return k;
    if (2 * k > n) k = n - k;
    // Berechnung
    if (n < k) { System.out.print(" error "); return 0; }
    long c=1, d=1;
    for (int i=0; i < k; ++i) {
        c *= n - i;
        d *= i + 1;
    }
    return c / d;
}
```



Probleme entstehen durch
spezifische Implementierung

3 Fehler

Einordnung von Tests

(Fortsetzung)

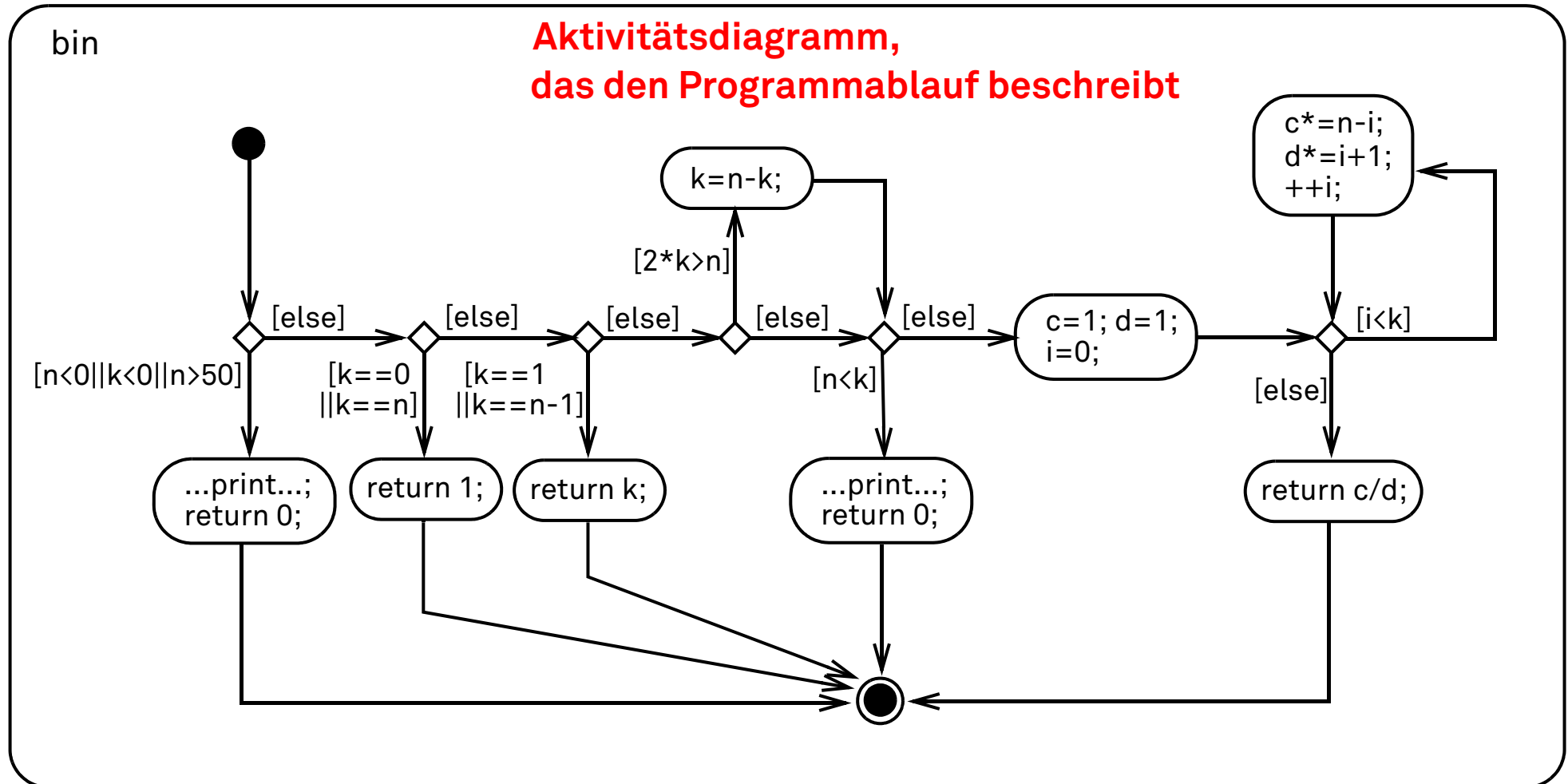
Geht die Ermittlung von Testfällen
vom Quelltext der Implementierung aus:
strukturorientierter Test (auch **white-box-Test**)

Die Testfälle werden so angelegt, dass bestimmte Code-Sequenzen ausgeführt werden.

Eigenschaften des strukturorientierten Tests

- ❑ Stellt Überprüfung des (gesamten) implementierten Codes sicher.
- aber:**
- ❑ Die Beschreibung wird nicht zur Auswahl von Testfällen genutzt, sondern nur zur Bestimmung der Soll-Ergebnisse,
 - es wird daher nicht versucht, die funktionale Richtigkeit gemäß einer vorher angegebenen Zielvorstellung nachzuweisen.
 - ❑ Strukturorientierte Tests sind häufig aufwändig,
 - da zunächst der Quelltext analysiert werden muss,
 - es eventuell sehr viele alternative Programmabläufe gibt,
 - zur Bestimmung der Soll-Ergebnisse der Quelltext den entsprechenden Teilen der Spezifikation zugeordnet werden muss.

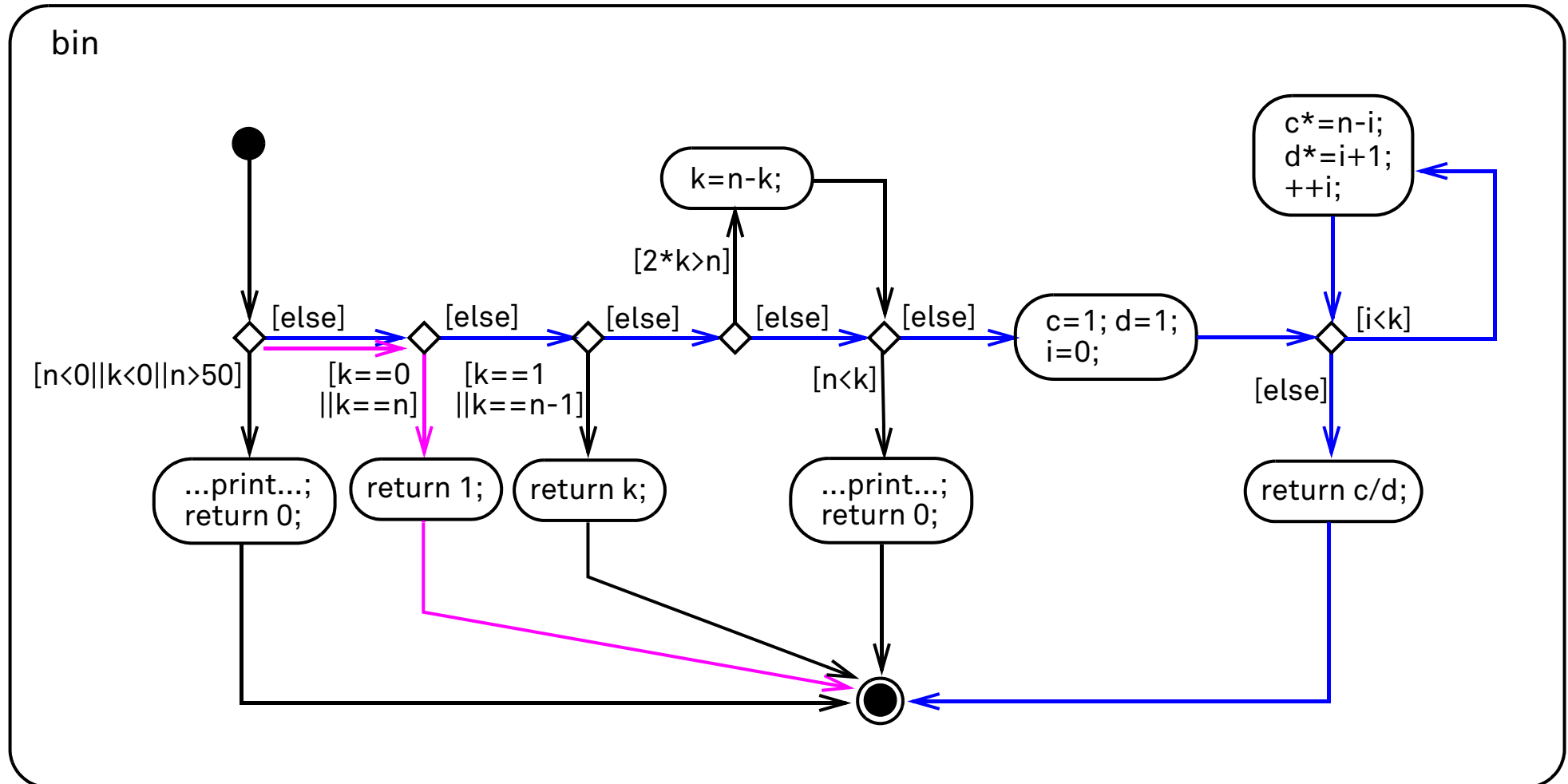
Darstellung der Struktur von bin(int n, int k)



Darstellung der Struktur von bin(int n, int k)

(Fortsetzung)

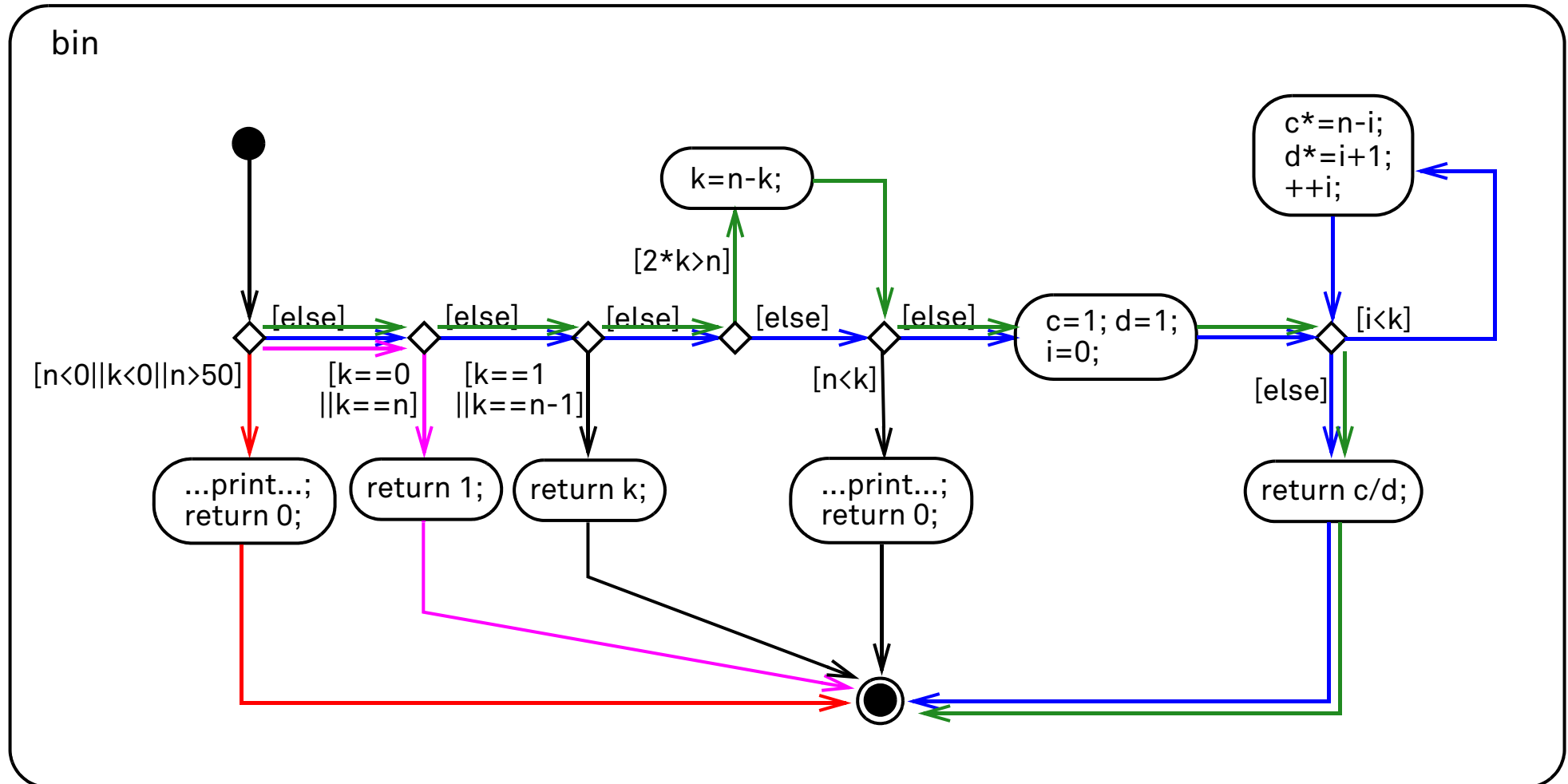
Testfälle aus black-box-Test (Folie 406): (0,0), (7,0), (7,3), (7,7)



Darstellung der Struktur von bin(int n, int k)

(Fortsetzung)

Testfälle aus black-box-Test (Folie 406): (3,7), (-1,7), (7,-1), (-1,-1)



Darstellung der Struktur von `bin(int n, int k)`

(Fortsetzung)

Analyse der von den Testfällen benutzten Pfade durch die Methode `bin`:

- ❑ Der (verborgene und fehlerhafte) Teil der Bedingung `... || n < 50` wird nicht getestet.
Der Fehler wird nicht gefunden.
- ❑ Die (verborgene und fehlerhafte) Optimierung `k == 1 || k == n - 1` wird nicht getestet.
Der Fehler wird nicht gefunden.
- ❑ Ein ungültiger Testfall (3,7) führt nicht zum erwünschten Ergebnis.
Dieser Fehler wird erkannt.
- ❑ Es gibt keinen ungültigen Testfall mit `n < k`.
Diese Situation wird nicht getestet.
- ❑ Es gibt keinen gültigen Testfall mit `2 * k > n`.
Diese Situation wird nicht getestet.

- ❑ Die Methode `bin` wird nicht ausreichend getestet.
- ❑ Aus der Betrachtung der Struktur der Methode lässt sich auf die Qualität des Tests schließen.

Strukturorientiertes Testen

Geht die Ermittlung von Testfällen
vom Quelltext der Implementierung aus:
strukturorientierter Test (auch **white-box-Test**)

Die Testfälle werden so angelegt, dass bestimmte Code-Sequenzen ausgeführt werden.

Eigenschaften des strukturorientierten Tests

- ❑ Stellt Überprüfung des (gesamten) implementierten Codes sicher.
- aber:**
- ❑ Die Spezifikation wird nicht zur Auswahl von Testfällen genutzt, sondern nur zur Bestimmung der Soll-Ergebnisse,
 - es wird daher nicht versucht, die funktionale Richtigkeit gemäß einer vorher angegebenen Zielvorstellung nachzuweisen.
 - ❑ Strukturorientierte Tests sind häufig aufwändig,
 - da zunächst der Quelltext analysiert werden muss,
 - es eventuell sehr viele alternative Programmabläufe gibt,
 - zur Bestimmung der Soll-Ergebnisse der Quelltext den entsprechenden Teilen der Spezifikation zugeordnet werden muss.

Strukturorientiertes Testen

(Fortsetzung)

Strukturorientiertes Testen wird nun mit Hilfe von UML-Aktivitätsdiagrammen präzisiert

Literatur: Hoffmann, Dirk W.: Software-Qualität, S. 157-216
http://link.springer.com/chapter/10.1007/978-3-540-76323-9_4

Liggesmeyer, Peter: Software-Qualität – Testen, Analysieren und Verifizieren von Software, S. 49-117, S136-138
http://link.springer.com/chapter/10.1007/978-3-8274-2203-3_2

Beispiel für eine Strukturanalyse (siehe Folie 349)

```
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 für eine Strukturanalyse (siehe Folie 349)

```
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++) {
```

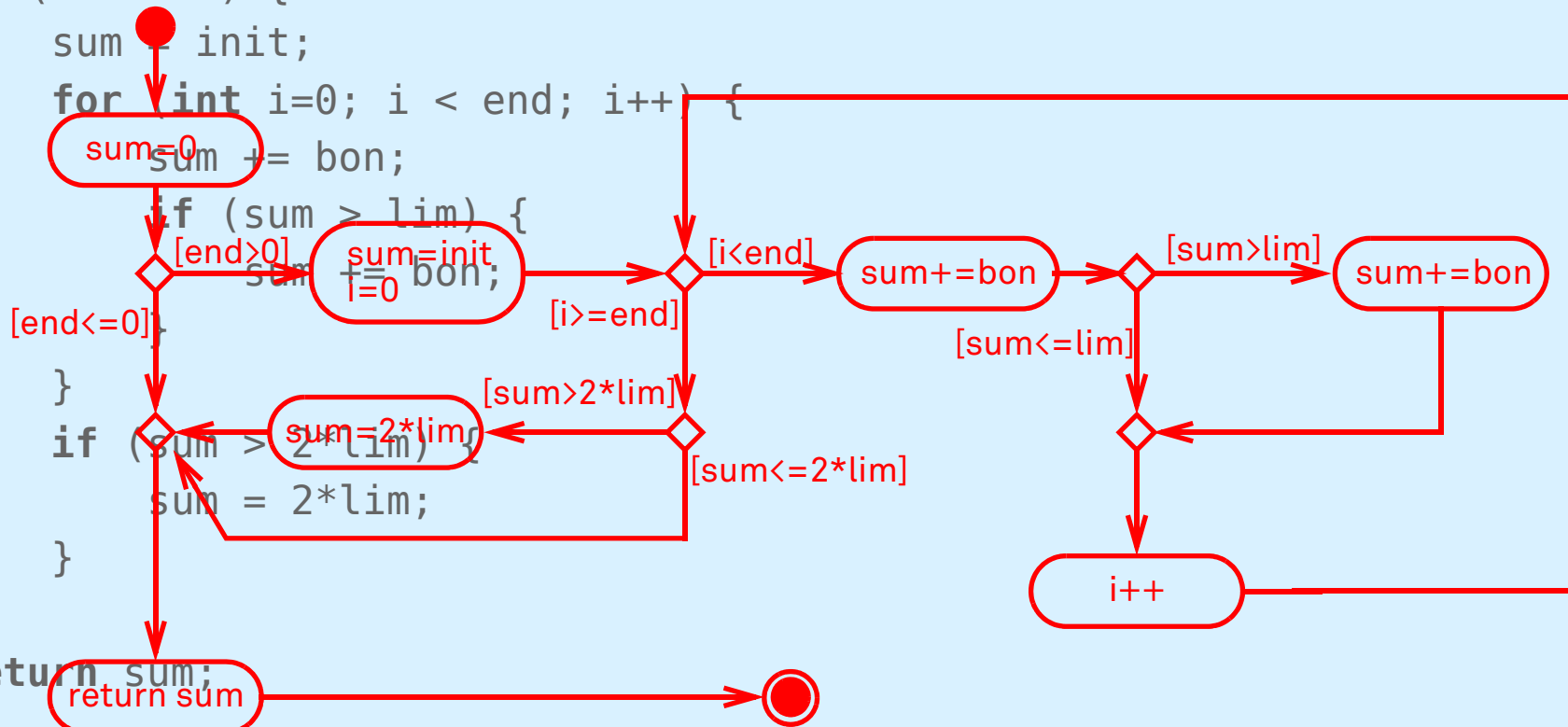
```
            sum += bon;
```

```
            if (sum > lim) {
```

```
                sum = init;
```

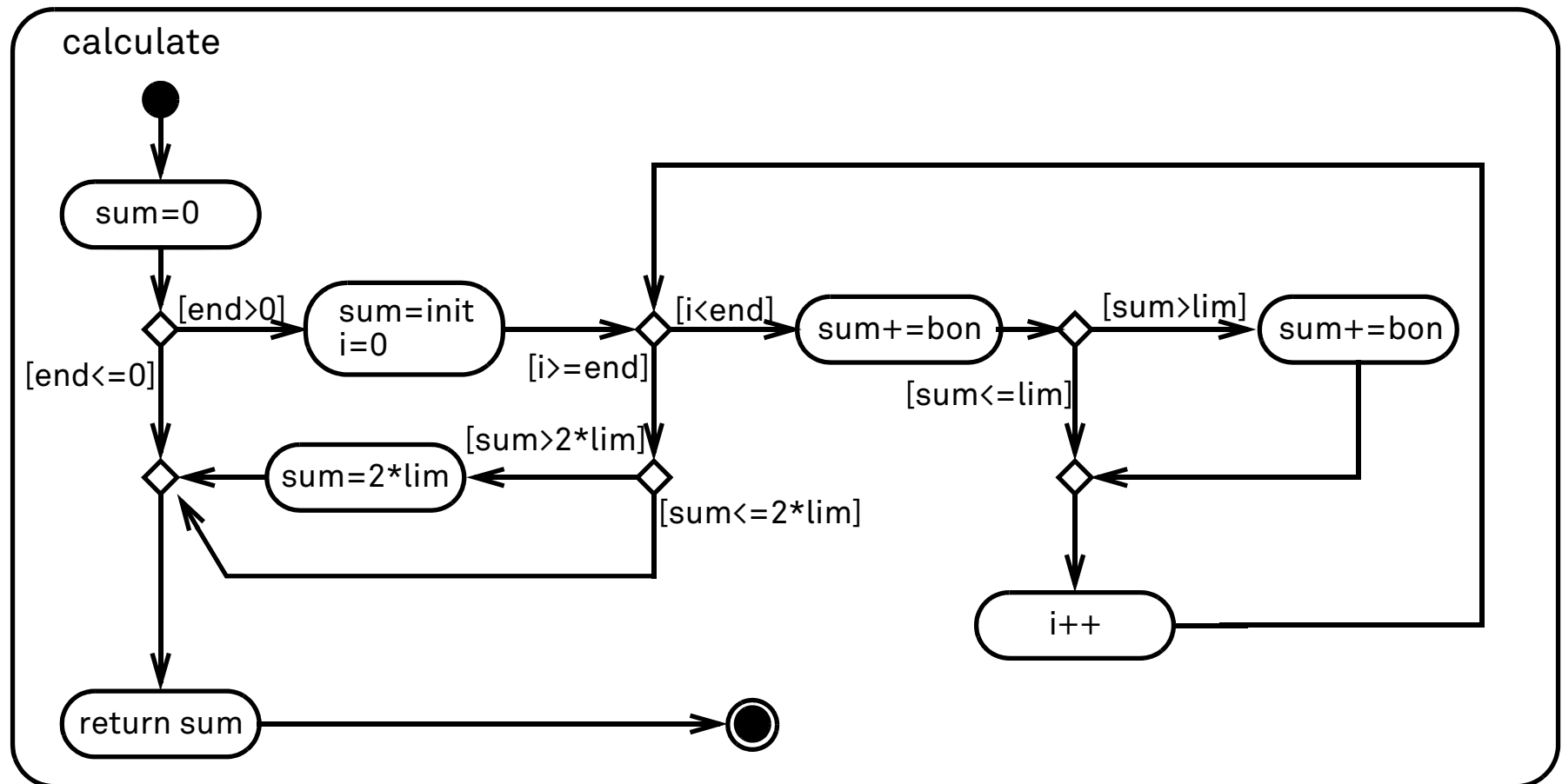
```
                i = 0;
```

```
            }
        }
    }
    if (sum > 2*lim) {
        sum = 2*lim;
    }
    return sum;
}
```



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

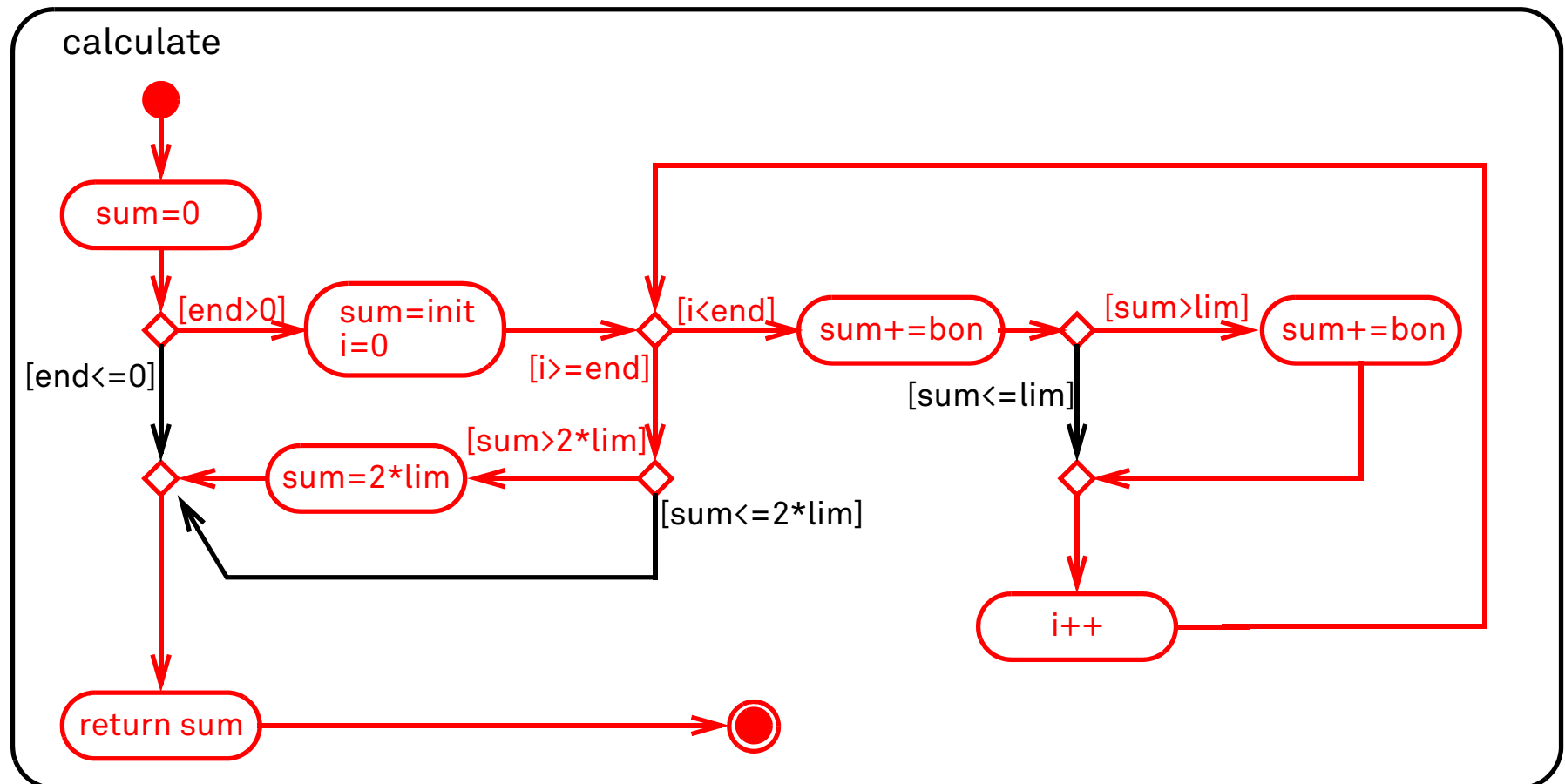
Vorgabe für **C₀-Test**: Jede Anweisung im Quelltext **muss mindestens einmal** ausgeführt werden.
= Jede Aktion des Diagramms muss von mindestens einem Testfall erreicht werden.



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

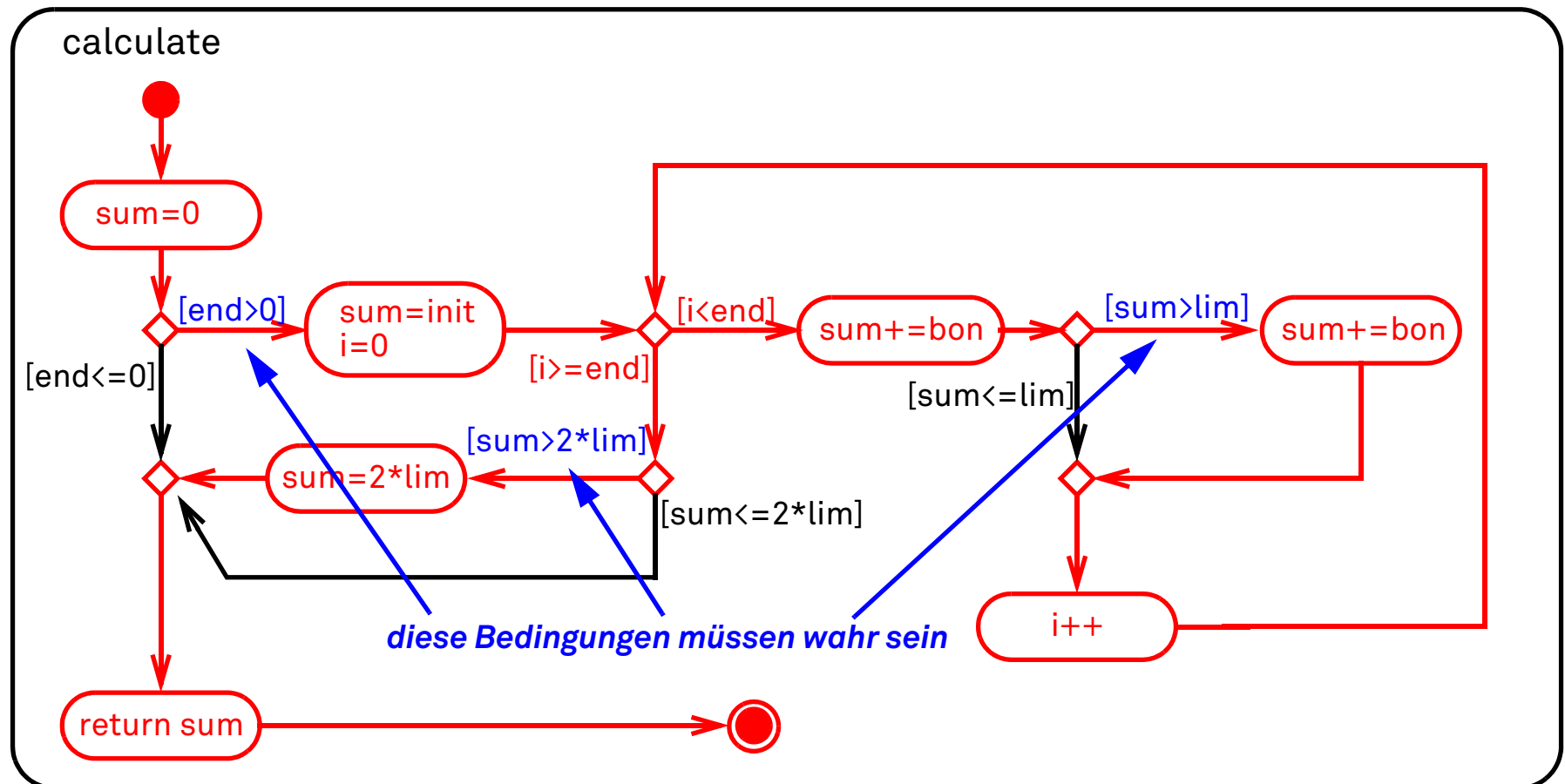
Für das Beispiel reicht ein Testfall aus, um alle Aktionen zu erreichen:
Die Parameter können geeignet gesetzt werden,
so dass alle Aktionen auf einem Pfad liegen.



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

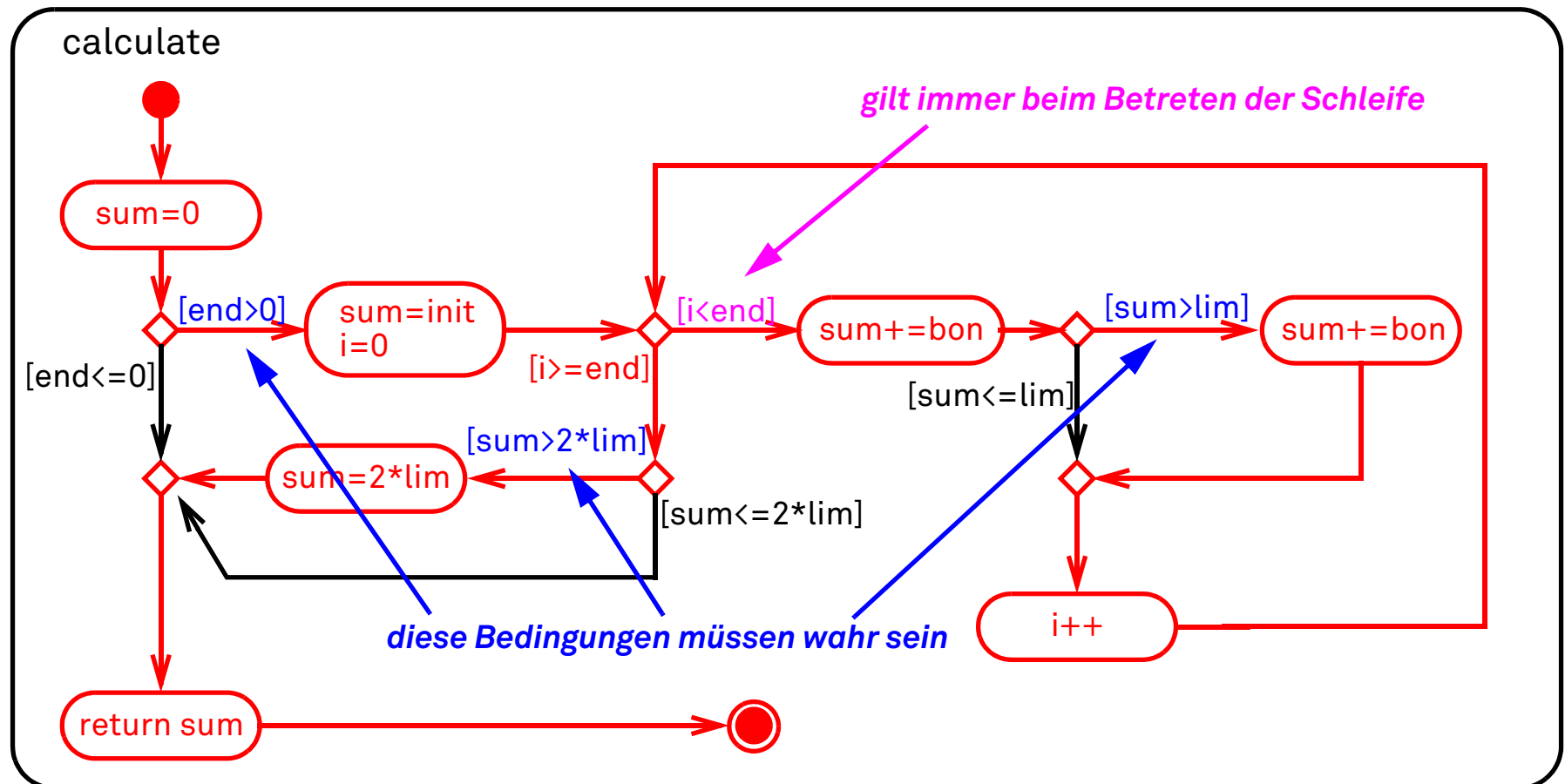
Für das Beispiel reicht ein Testfall aus, um alle Aktionen zu erreichen:
Die Parameter können geeignet gesetzt werden,
so dass alle Aktionen auf einem Pfad liegen.



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

Für das Beispiel reicht ein Testfall aus, um alle Aktionen zu erreichen:
Die Parameter können geeignet gesetzt werden,
so dass alle Aktionen auf einem Pfad liegen.



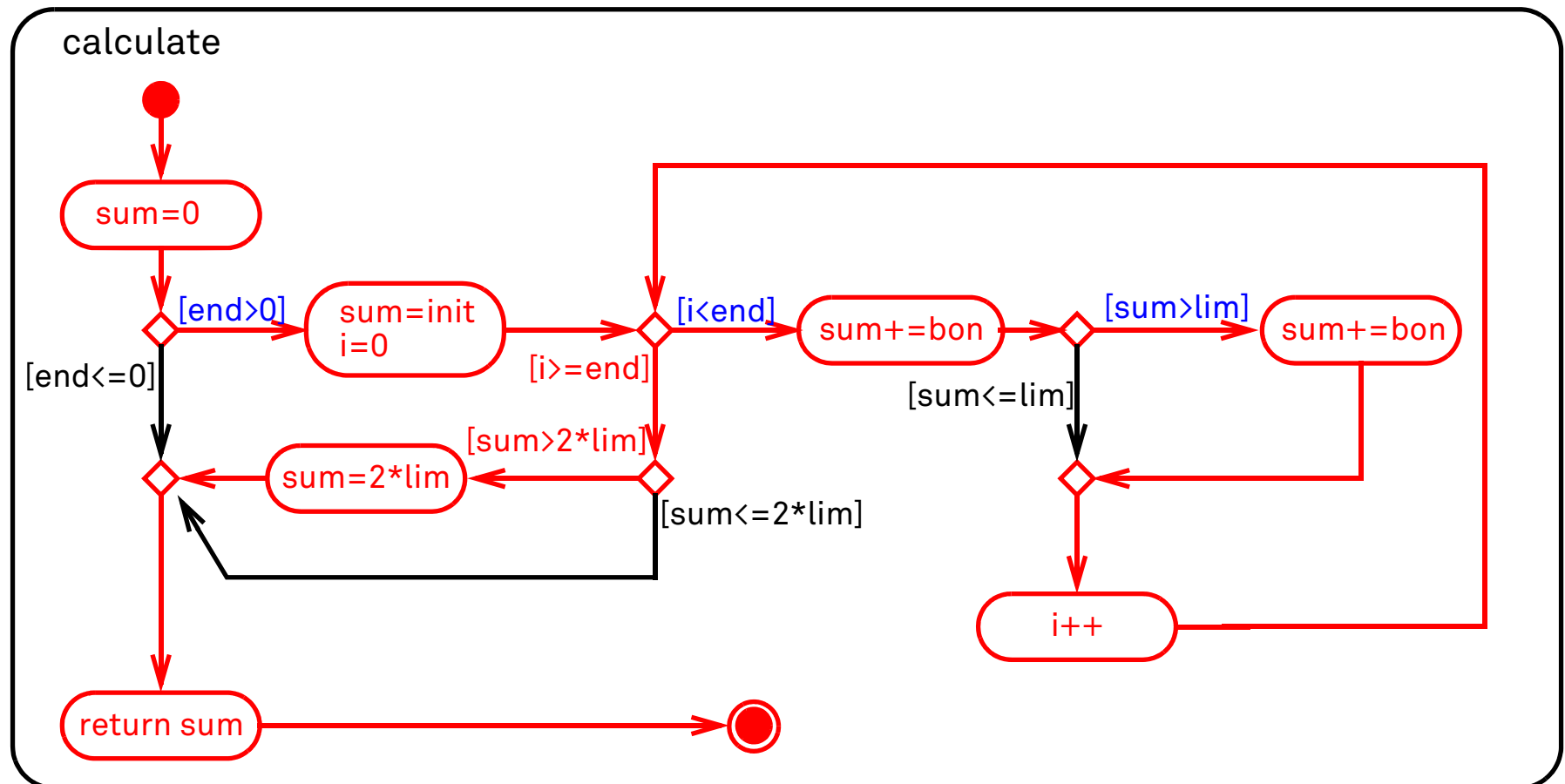
Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

wird für `calculate(int end, int init, int lim, int bon)`

z.B. erreicht mit: `end=1, init=0, lim=1, bon=2`

Aufruf also z.B.: `calculate(1, 0, 1, 2)`



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

Analyse der Leistungsfähigkeit der Anweisungsüberdeckung:

- ❑ Die Anweisungsüberdeckung erkennt nicht erreichbare Anweisungen.
- ❑ Das Einhalten bestimmter Reihenfolgen bei der Ausführung von Aktionen in Abhängigkeit von Bedingungen wird aber nicht berücksichtigt.
- ❑ Es werden nur wenige Ausführungsfolgen getestet und es wird so nur ein Teil der algorithmisch möglichen Abläufe geprüft.

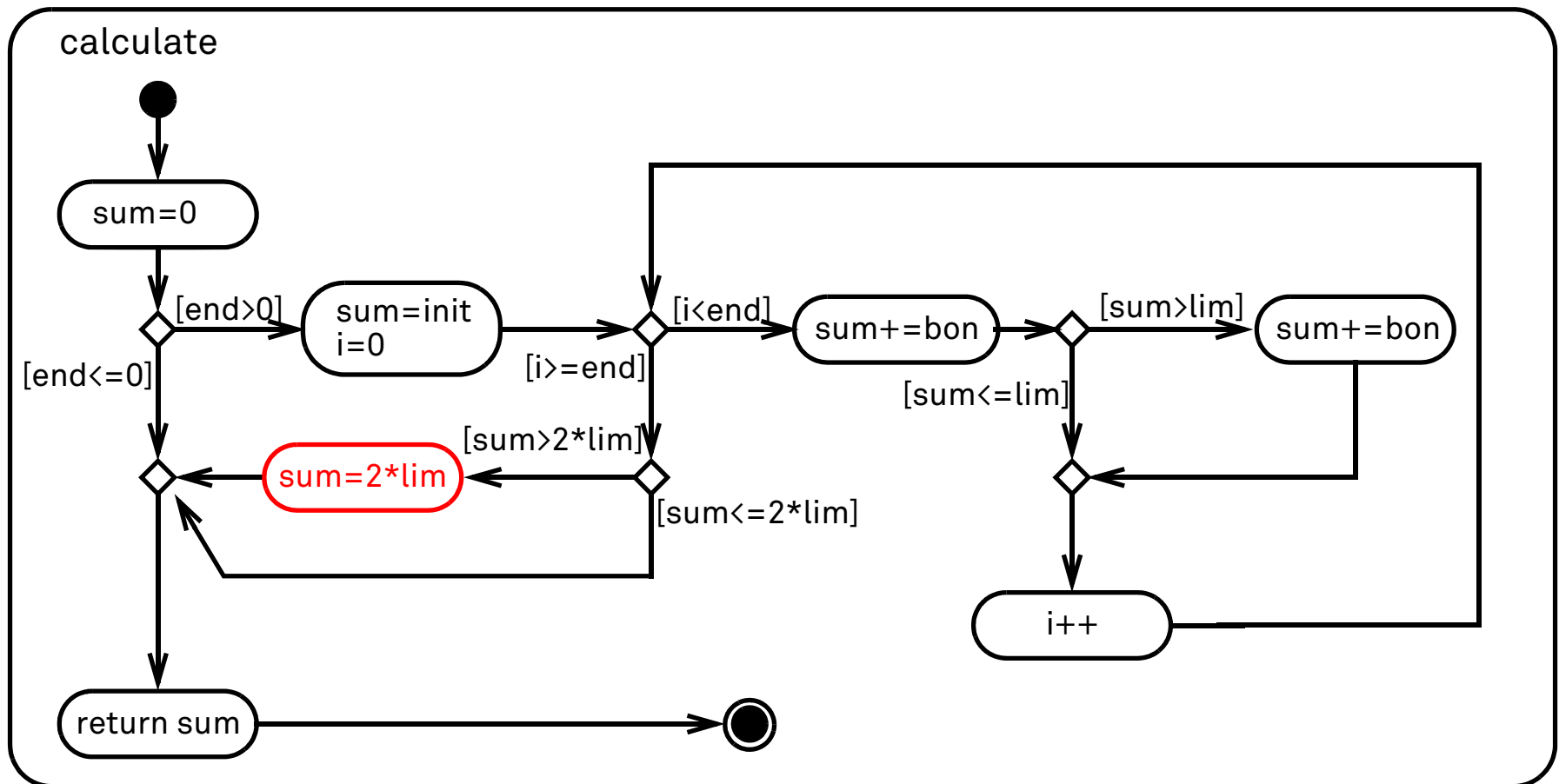
Fehler werden daher nur zufällig erkannt.

Güte eines Überdeckungstests T

Überdeckungsgrad $G = (\text{unter } T \text{ überdeckte Anweisungen}) / (\text{alle Anweisungen})$

für das Beispiel: `calculate(1, 0, 1, 2)` führt zu $G = 1.0$

`calculate(1, 3, 3, 1)` führt zu $G = 0.875$



Ermittlung von Testfällen: Zweigüberdeckung (C_1 -Test)

Da die Anweisungsüberdeckung nur zu unzureichenden Aussagen führt:

Zweigüberdeckung (C_1 -Test)

Vorgabe für **C_1 -Test**:

Jeder alternative Zweig im Programmtext wird mindestens einmal durchlaufen.

= Jede Kante des Diagramms wird mindestens einmal durchlaufen.

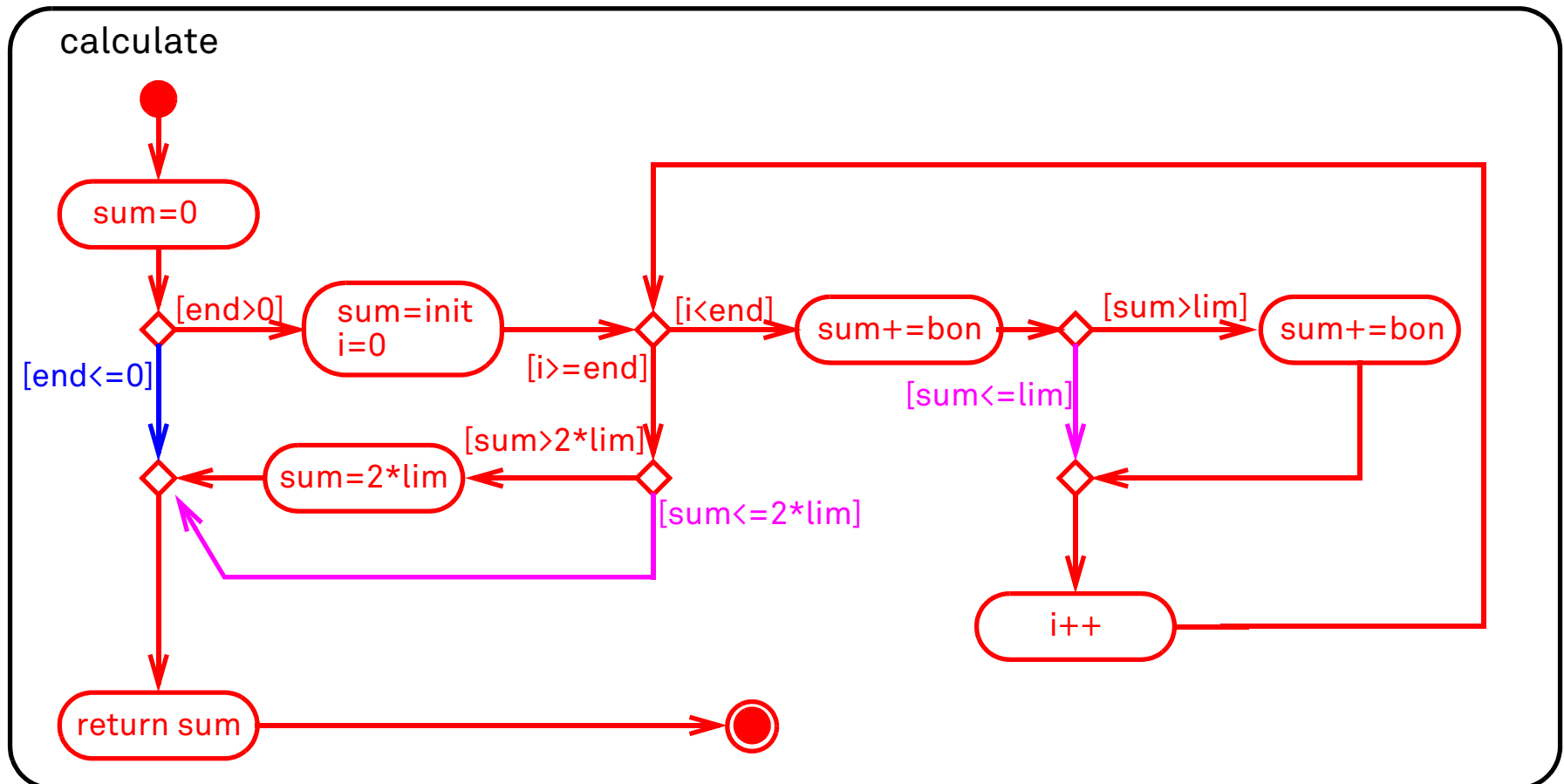
Anmerkungen:

- ❑ Die Zweigüberdeckung umfasst immer die Anweisungsüberdeckung.
- ❑ Es gibt möglicherweise schwer erreichbare Zweige, wenn die notwendigen Bedingungen nur schwer gezielt hergestellt werden können.
- ❑ Bei einfachen bedingten Anweisungen gibt es *leere Zweige*, die bei der Zweigüberdeckung durchlaufen werden müssen.
- ❑ Bei Schleifen genügt ein einziger Durchlauf. ==> **unzureichende Prüfung**
- ❑ Für das Beispiel gilt:
Zusätzlich zum Testfall aus der Anweisungsüberdeckung müssen Testfälle für die *drei leeren Zweige* ergänzt werden.

Ermittlung von Testfällen: Zweigüberdeckung (C₁-Test)

(Fortsetzung)

wird z.B. erreicht mit: `end=1, init=0, lim=1, bon=2` (aus C₀-Überdeckung bekannt),
 und `end=0` (*leerer Zweig*)
 und `end=1, init=0, lim=1, bon=0` (*2 leere Zweige*)



Ermittlung von Testfällen: Zweigüberdeckung (C₁-Test)

(Fortsetzung)

Analyse der Leistungsfähigkeit der Zweigüberdeckung:

- ❑ Die Zweigüberdeckung erkennt alle nicht erreichbaren Zweige und alle nicht erreichbaren Anweisungen.
- ❑ Die Zweigüberdeckung berücksichtigt keine Abhängigkeiten zwischen den Zweigen.
- ❑ Komplexe Bedingungen werden nur rudimentär getestet:
Bei Disjunktionen reicht ein *wahrer*,
bei Konjunktionen ein *falscher* Teilausdruck,
um den Wert der gesamten Bedingung zu setzen.

Beispiele, die mit nur zwei Belegungen für *a, b, c, d* beide Zweige abdecken:

```
if ( a | b | c | d ) A1 else A2;  
if ( a & b & c & d ) A1 else A2;
```

- ❑ Eine vollständige Zweigüberdeckung (Überdeckungsgrad $G = 1.0$) ist für Software in bestimmten Anwendungsbereichen der Mindeststandard:
z.B. für kritische Software im Bereich der Luftfahrt (Standard RTCA DO-178B).

Ermittlung von Testfällen: Mehrfachbedingungsüberdeckung

Da auch die Zweigüberdeckung nur zu unzureichenden Aussagen führt:

Mehrfachbedingungsüberdeckung

Vorgabe für die Mehrfachbedingungsüberdeckung:

Jede mögliche Kombination von Wahrheitswerten der atomaren Prädikate wird ausgeführt.

Anmerkungen:

- ❑ Die Mehrfachbedingungsüberdeckung umfasst die Anweisungsüberdeckung.
- ❑ Die Mehrfachbedingungsüberdeckung umfasst die Zweigüberdeckung.
- ❑ Es kann nicht-realisierte Kombinationen von Wahrheitswerten geben.
Diese müssen keine Fehler sein, z.B. bei Short-Cut-Operatoren wie && oder ||.
- ❑ Beispiel:
Der Ausdruck $(nr > 1 \ \& \ fehler < 0) \mid abbruch$ hat drei atomare Prädikate,
also müssen 8 Testfälle erstellt werden.
- ❑ **Problem:** bei n Teilausdrücken entstehen 2^n Testfälle
==> Die Mehrfachbedingungsüberdeckung ist aufgrund der kombinatorischen
Komplexität in der Praxis häufig unbrauchbar.

Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung

Pragmatische Variante der Mehrfachbedingungsüberdeckung:
modifizierte Bedingungs-/Entscheidungsüberdeckung

Vorgabe für die modifizierte Bedingungs-/Entscheidungsüberdeckung:

Für jedes atomare Prädikat muss im Test nachgewiesen werden,
dass es das Ergebnis der Auswertung der Bedingung unabhängig
von den anderen atomaren Prädikaten beeinflussen kann.

Präzisierung:

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen,
die sich **nur genau** im Wert dieses Prädikats unterscheiden
und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

Anmerkungen:

- ❑ Die modifizierte Bedingungs-/Entscheidungsüberdeckung umfasst die Anweisungs- und die Zweigüberdeckung.
- ❑ Auch die modifizierte Bedingungs-/Entscheidungsüberdeckung ist sehr aufwändig.

Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen,
die sich nur genau im Wert dieses Prädikats unterscheiden
und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

- Ein erstes, einfaches Beispiel.

Der Ausdruck $a \mid b \mid c$ benötigt vier Testfälle:

- a bestimmt nur dann das Ergebnis, wenn $b \mid c == \text{false}$ gilt,
also $b == \text{false}$ und $c == \text{false}$,
die Testfälle sind also $\text{true}, \text{false}, \text{false}$ und $\text{false}, \text{false}, \text{false}$
- b bestimmt nur dann das Ergebnis, wenn $a \mid c == \text{false}$ gilt,
die Testfälle sind also $\text{false}, \text{true}, \text{false}$ und $\text{false}, \text{false}, \text{false}$
- c bestimmt nur dann das Ergebnis, wenn $a \mid b == \text{false}$ gilt,
die Testfälle sind also $\text{false}, \text{false}, \text{true}$ und $\text{false}, \text{false}, \text{false}$

- Anmerkung:

Das Beispiel zeigt, dass die Testfälle für verschiedene Prädikate identisch sein können.
Bei n Prädikaten werden also höchstens – aber nicht immer – $2n$ Testfälle benötigt.

Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen, die sich nur genau im Wert dieses Prädikats unterscheiden und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

- ❑ weiteres Beispiel: $(nr > 1 \ \& \ fehler > 0) \mid abbruch$
- ❑ Das Prädikat $nr > 1$ hat nur dann eine Wirkung, wenn
 - $fehler > 0$ den Wert `true` hat (denn sonst liefert die Konjunktion `false`) und
 - $abbruch$ den Wert `false` hat (denn sonst liefert die Disjunktion immer `true`).
- ❑ $nr > 1$ muss selbst die beiden Werte `true` und `false` annehmen.

$nr > 1$	$fehler > 0$	$abbruch$	Ergebnis
true	true	false	true
false	true	false	false



Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen, die sich nur genau im Wert dieses Prädikats unterscheiden und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

- ❑ weiteres Beispiel: $(nr > 1 \ \& \ fehler > 0) \mid abbruch$
- ❑ Das Prädikat $fehler > 0$ hat nur dann eine Wirkung, wenn
 - $nr > 1$ den Wert `true` hat (denn sonst liefert die Konjunktion `false`) und
 - $abbruch$ den Wert `false` hat (denn sonst liefert die Disjunktion immer `true`).
- ❑ $fehler > 0$ muss selbst die beiden Werte `true` und `false` annehmen.

$nr > 1$	$fehler > 0$	$abbruch$	Ergebnis
true	true	false	true
false	true	false	false
true	false	false	false

bekannter Testfall

bekannter Testfall

neuer Testfall



Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen, die sich nur genau im Wert dieses Prädikats unterscheiden und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

- ❑ weiteres Beispiel: weiteres Beispiel: $(nr > 1 \ \& \ fehler > 0) \mid abbruch$
- ❑ Das Prädikat `abbruch` hat nur dann eine Wirkung, wenn
 - die Konjunktion den Wert `false` liefert,
also `nr > 1` **oder** `fehler > 0` den Wert `false` besitzt
- ❑ `abbruch` muss selbst die beiden Werte `true` und `false` annehmen.

nr>1	fehler>0	abbruch	Ergebnis
true	true	false	true
false	true	false	false
true	false	false	false
false	true	true	true

bekannter Testfall

bekannter Testfall

bekannter Testfall

neuer Testfall

- ❑ Für den Test des Beispiels werden also 4 Testfälle benötigt.

Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Analyse der modifizierten Bedingungs-/Entscheidungsüberdeckung:

- ❑ Für n atomare Prädikate werden mindestens $n+1$ Testfälle benötigt.
- ❑ Für n atomare Prädikate werden höchstens $2n$ Testfälle benötigt.
- ❑ Es gibt in der Regel verschiedene Kombinationen von Testfällen, die eine modifizierte Bedingungs-/Entscheidungsüberdeckung mit $G=1.0$ herstellen.
- ❑ Sind atomare Prädikate voneinander abhängig, so kann eine modifizierte Bedingungs-/Entscheidungsüberdeckung mit $G=1.0$ nicht sichergestellt werden.
- ❑ Eine modifizierten Bedingungs-/Entscheidungsüberdeckung mit $G=1.0$ ist ebenfalls ein für Software in bestimmten Anwendungsbereichen (z.B. Luftfahrt) vorgeschriebener Standard.

Ermittlung von Testfällen: Pfadüberdeckung

Vorgabe für die **Pfadüberdeckung**:

Jeder mögliche Pfad wird mindestens einmal durchlaufen.

- ❑ In dieser allgemeinen Form nur von theoretischer Bedeutung:
Schleifen können zu sehr vielen Pfaden führen, die nicht alle getestet werden können.
- ❑ pragmatische Alternative:
 k -begrenzte strukturierte Pfadüberdeckung
Jede Anweisung wird **höchstens** k -mal durchlaufen.
Insbesondere werden also die Rümpfe von Schleifen nur bis maximal k -mal durchlaufen.

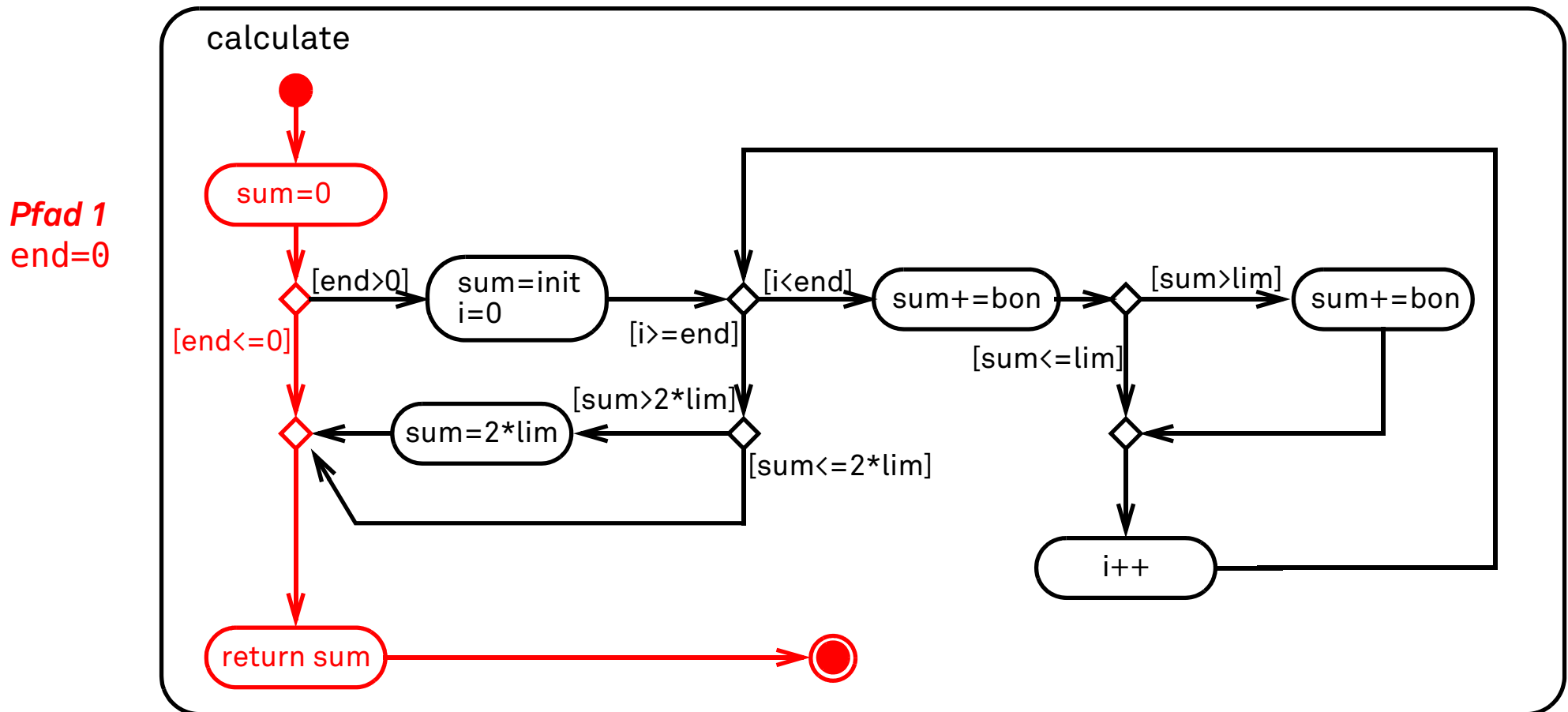
Anmerkungen:

- ❑ Für k wird typischerweise ein kleiner Wert, z.B. $k=2$, gewählt.
- ❑ Eine **2-begrenzte** strukturierte Pfadüberdeckung schließt also immer die **1-begrenzte** strukturierte Pfadüberdeckung ein.

Ermittlung von Testfällen: strukturierte Pfadüberdeckung

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):

Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.

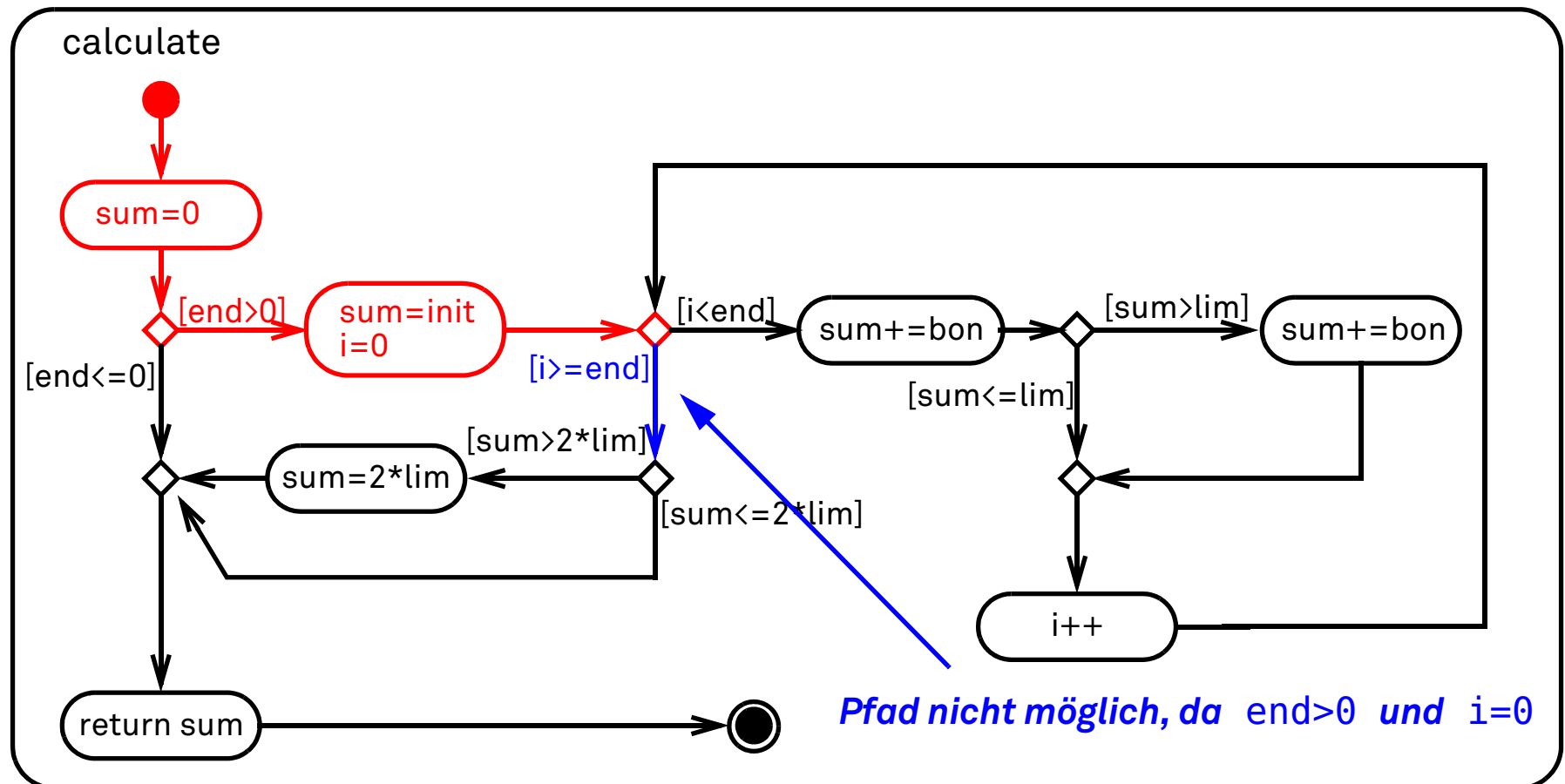


Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):

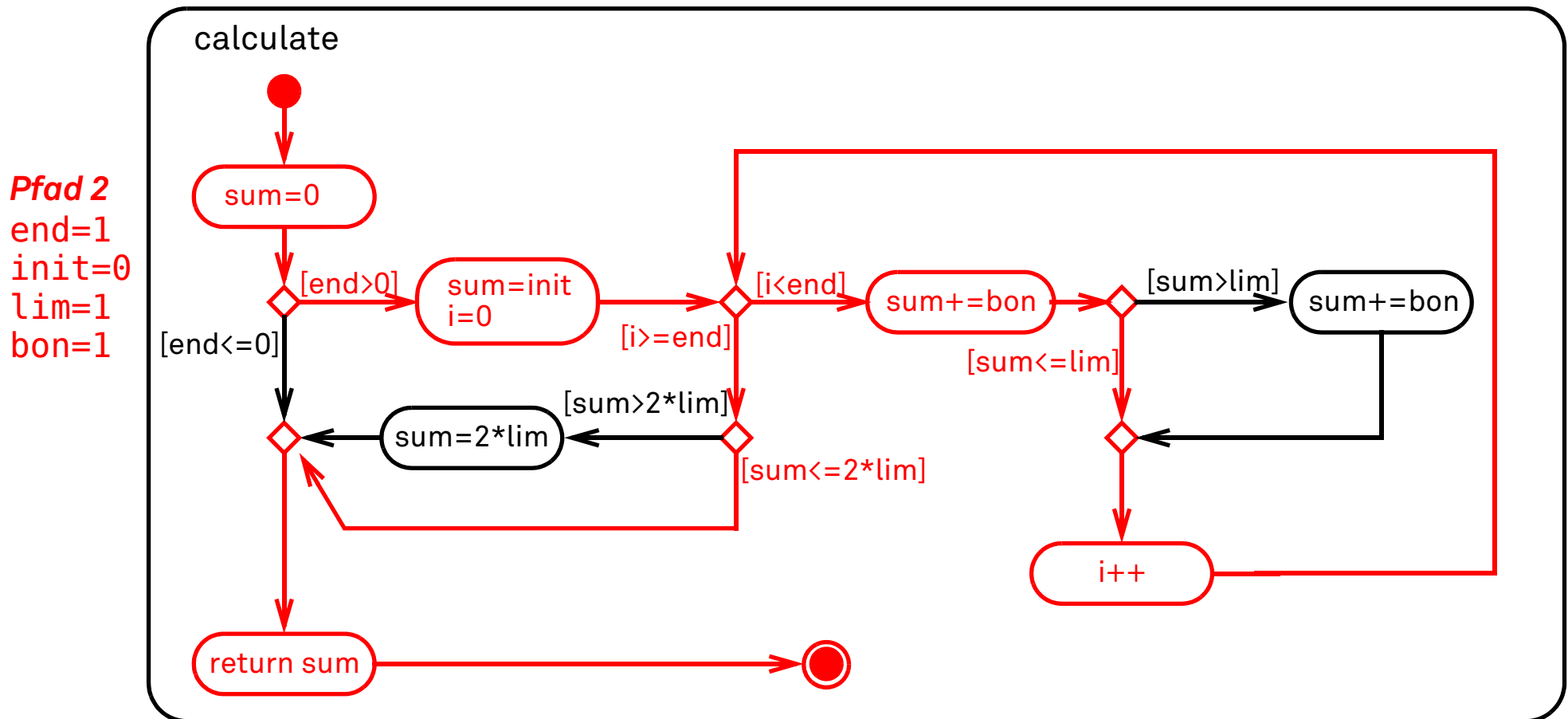
Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.



Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):
Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.



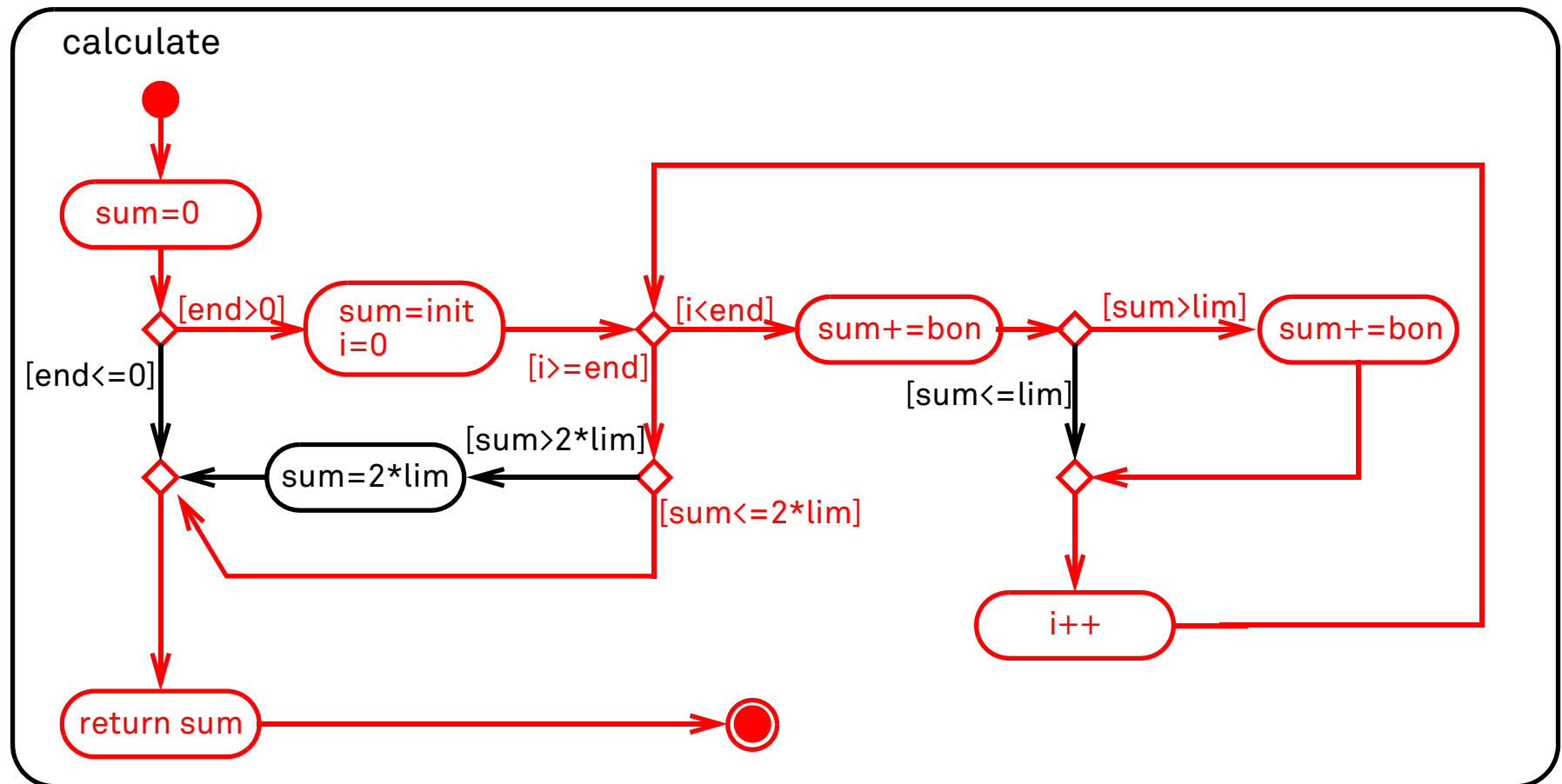
Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung (k=1):

Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.

Pfad 3
end=1
init=2
lim=2
bon=1

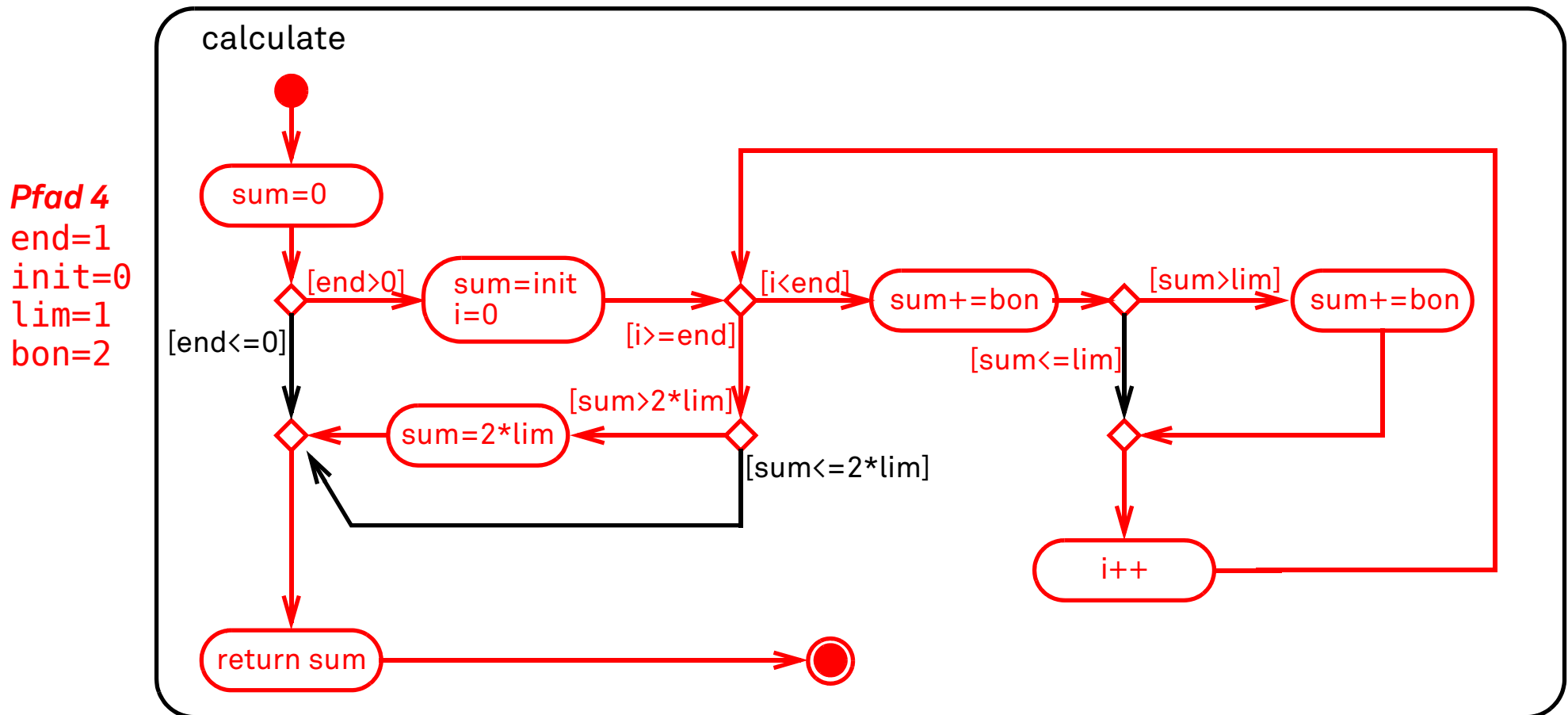


Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):

Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.

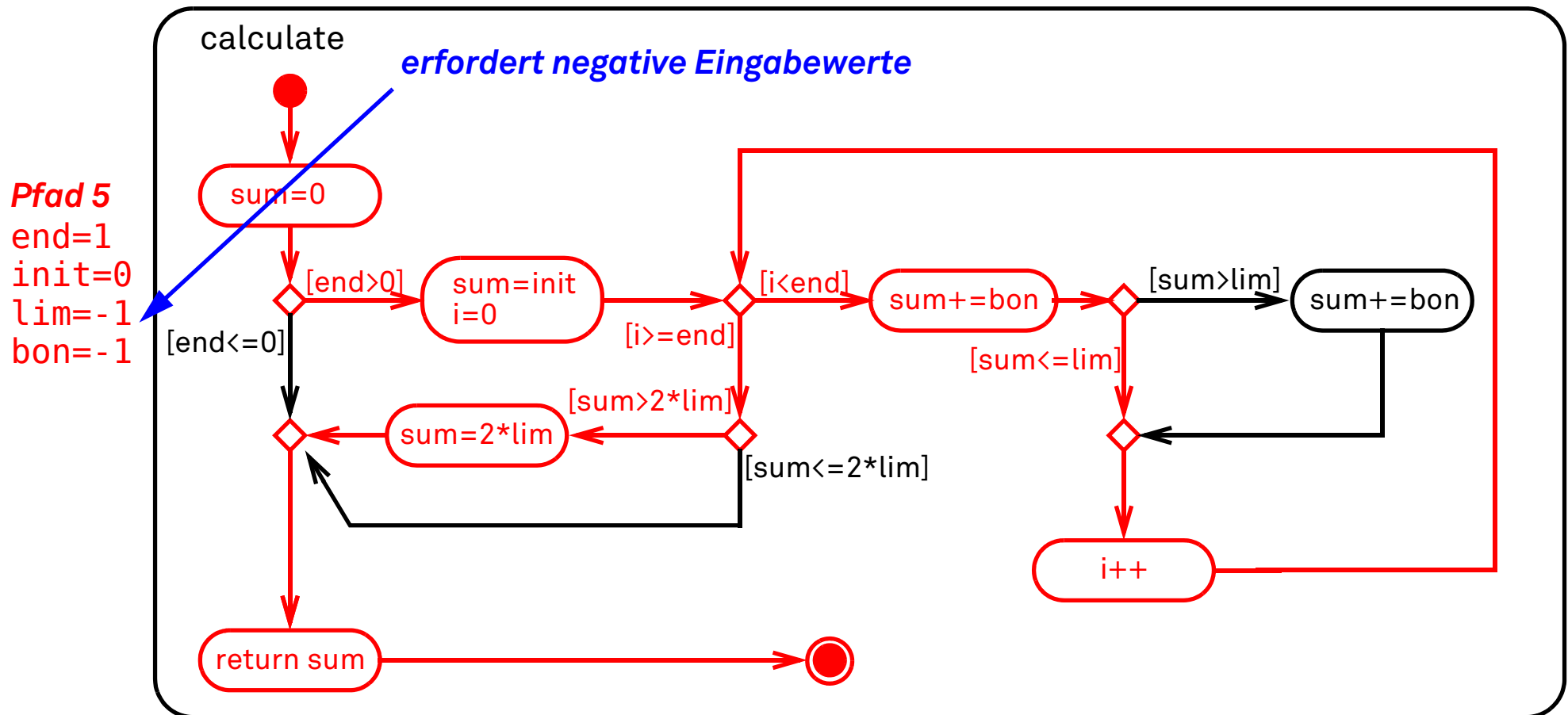


Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung (k=1):

Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.



Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

Analyse der Leistungsfähigkeit der Pfadüberdeckung

- ❑ Die Pfadüberdeckung ist wegen der kombinatorischen Komplexität in der Praxis kaum anwendbar.
- ❑ Eine – ebenfalls aufwändige – Alternative ist die k -begrenzte strukturierte Pfadüberdeckung

Güte eines Tests T :

- ❑ Überdeckungsgrad $G = (\text{unter } T \text{ überdeckte Pfade}) / (\text{alle Pfade})$
- ❑ **aber:** in der Regel können nicht alle theoretisch möglichen Pfade während der Ausführung tatsächlich abgedeckt werden.
Daher kann die Güte nur schwer festgestellt werden.

Vereinfachte Schleifenüberdeckung

Eine Überprüfung von Schleifen kann zusätzlich nach folgendem Schema erfolgen:

Vorgaben für die **vereinfachte Schleifenüberdeckung**:

Die folgenden fünf Testfälle müssen berücksichtigt werden, sofern sie auftreten können:

- ❑ Schleife wird nicht betreten – prüft den Misserfolg der Eintrittsbedingung
 - ❑ Es erfolgt genau 1 Durchlauf durch die Schleife – prüft Initialisierungen
 - ❑ Es erfolgen genau 2 Durchläufe durch die Schleife – prüft Initialisierungen
 - ❑ Es erfolgt eine typische Anzahl von Durchläufen – prüft Abbruchkriterien
 - ❑ Die maximale Anzahl von Durchläufen wird erreicht – prüft Abbruchkriterien
-
- ❑ Bei zwei geschachtelten Schleifen kombinieren sich diese Tests bereits zu 21 Fällen:
 - Die äußere Schleife wird nicht betreten.
 - Jeder der anderen 4 Testfälle für die äußere Schleife erfordert 5 Testfälle für die innere Schleife.

weitere strukturorientierte Tests

bisher vorgestellt:

Die Testfälle für strukturorientierte Tests werden so bestimmt, dass eine bestimmte Form der Überdeckung des Quelltextes von Methoden erreicht wird.

Für *größere* Programmstrukturen

können die Testfälle für strukturorientierte Tests zusätzlich anhand der Überdeckung der Bestandteile dieser Strukturen ermittelt werden:

- ❑ Überdeckung der von einer Klasse angebotenen Methoden
- ❑ Überdeckung der in einer Klasse aufgerufenen Methoden
- ❑ Überdeckung der ausgelösten/behandelten Ausnahmen
- ❑ ...

Strukturorientierte Tests (Zusammenfassung)

- ❑ Anweisungsüberdeckung (C_0)
- ❑ Zweigüberdeckung (C_1)
- ❑ Mehrfachbedingungsüberdeckung
- ❑ modifizierte Bedingungs-/Entscheidungsüberdeckung
- ❑ Pfadüberdeckung
- ❑ k-begrenzte strukturierte Pfadüberdeckung
- ❑ vereinfachte Schleifenüberdeckung

Beispiel: Apple iOS Implementierung des Secure Sockets Layer (SSL)

Das Beispiel zeigt, dass an einem kritischen Teil des Apple iOS-Betriebssystems auf sorgfältige strukturierte Tests verzichtet wurde.

Die entsprechende Implementierung wurde erstellt: September 2012

Der Fehler wurde erkannt: Februar 2014

- ❑ Ziel der Software u.a.:
Prüfung, dass der Schlüssel für eine gesicherte Verbindung auch vom gewünschten Partner bereitgestellt wird
- ❑ fehlerhafte Implementierung:
 - Prüfung entfällt völlig
 - jeder Schlüssel wird akzeptiert

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

Aufbau der Software:

Viele aufeinander folgende Methodenaufrufe, die bei korrekter Ausführung jeweils den Wert 0 zurückgeben. Bei nicht-korrekter Ausführung wird der Wert gespeichert und mit einer Fehlerbehandlung an der Marke `fail` fortgefahren.

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

Fortsetzung bei fail:

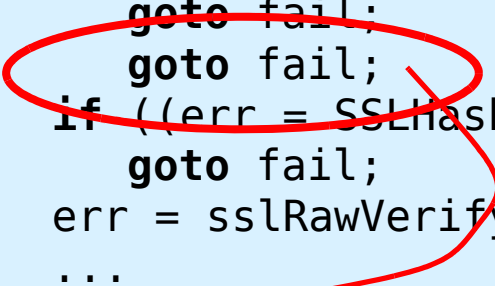
An der Marke `fail` wird *nicht* abgebrochen, da vom Scheitern einer der vorangehend aufgerufenen Methoden ausgegangen wird.

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```



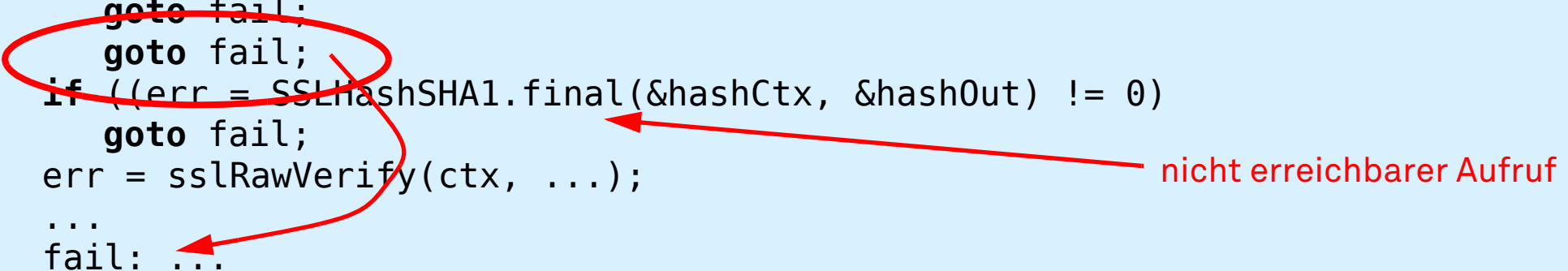
Fehler: unkontrollierter Sprung, die nachfolgenden Anweisungen sind nicht erreichbar.

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```



nicht erreichbarer Aufruf

Es kann unmittelbar festgestellt werden:

- zusammenhängendes Aktivitätsdiagramm kann nicht gezeichnet werden
- C_0 -Überdeckung mit $G=1,0$ kann nicht erreicht werden
- ein ausreichender strukturorientierter Test kann nicht erfolgt sein

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

- ❑ unkontrollierter Sprung, die nachfolgenden Anweisungen sind nicht erreichbar
- ❑ ein Scheitern der Funktionalität von `sslRawVerify` ist nicht betrachtet worden:
 - Äquivalenzklassen für funktionales Verhalten sind nicht angelegt und getestet worden
 - ein ausreichender funktionsorientierter Test ist ebenfalls nicht erfolgt

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

- ❑ unkontrollierter Sprung, die nachfolgenden Anweisungen sind nicht erreichbar
- ❑ ein Scheitern der Funktionalität von `sslRawVerify` ist nicht betrachtet worden
- ❑ eine andere Programmstruktur hätte den Fehler möglicherweise konstruktiv vermieden: Blöcke, `if-else`, `switch`, eine gemeinsame Bedingung mit short-cut-Disjunktionen

Testen objektorientierter Systeme

Objektorientierung führt zu komplexen Beziehungen zwischen den zu testenden Methoden:

- ❑ Die Ausführung einer Methode hinterläßt einen Zustand, der die nachfolgenden Ausführungen von Methoden beeinflusst.
- ❑ Vererbung schafft (verborgene) Abhängigkeiten.
- ❑ Polymorphie/dynamische Bindung führt zu einer Vervielfachung der möglichen Pfade.
- ❑ Kapselung durch Zugriffsrechte erschwert die Sicht auf die Zustände von Objekten:
 - Die Herstellung eines bestimmten Zustands ist möglicherweise schwer machbar.
 - Die Abfrage des Zustands ist möglicherweise schwer machbar.
- ❑ Klassen sind häufig funktional umfangreicher als notwendig:
=> Testen nicht benötigter Methoden verursacht zusätzlichen Aufwand.

Testen objektorientierter Systeme

(Fortsetzung)

Die Ausführung einer Methode hinterläßt einen Zustand,
der die nachfolgenden Ausführungen von Methoden beeinflusst.

Beispiel

Klasse `ArrayList<E>` mit den Methoden `add(E elem)` und `remove(E elem)`:

- ❑ Jeder Aufruf von `add` erfolgt für eine Liste, die eventuell Elemente enthält, jeder Aufruf von `add` hinterläßt immer eine veränderte Liste.
- ❑ Jeder Aufruf von `remove` erfolgt für eine Liste, die eventuell Elemente enthält, ein Aufruf von `remove` hinterläßt manchmal eine veränderte Liste.
- ❑ Neben dem Parameter `elem` bildet das `ArrayList`-Objekt selbst mit allen seinen Elementen für beide Methode eine weitere Eingabe für die Ausführung.
- ❑ Zugleich ist die geänderte Liste auch das Ergebnis dieser Ausführung.
- ❑ Der Zustand der Liste geht in die Gestaltung von Äquivalenzklassen ein und hat für die Bestimmung der Testfälle eine wesentliche Bedeutung:

zustandsbasiertes Testen

Testen objektorientierter Systeme

(Fortsetzung)

Folgen des Einsatzes von Spezialisierung/Vererbung:

- ❑ Fehler in Oberklassen werden an die Unterklassen weitergegeben.
- ❑ Das Testen der Unterklassen ist nicht ohne Einbeziehung der Oberklassen möglich.
- ❑ Änderungen an Oberklassen erfordern immer erneutes Testen aller Unterklassen.
- ❑ Objekte von Unterklassen können Objekte der Oberklassen ersetzen:
 - Wird Software mit Referenzen/Zeigern auf Oberklassen getestet, müssen eventuell Objekte aller Unterklassen in den Test eingebracht werden.
 - Wird eine Unterklasse geändert, muss eventuell die Nutzung von Referenzen auf die Oberklasse überprüft werden.

Anmerkungen: Testen objektorientierter Systeme

(Fortsetzung)

Folgen des Einsatzes von Spezialisierung/Vererbung

Beispiel

```
class ArrayList<E> {  
    public void add(E elem) { ... }  
    public void remove(E elem) { ... }  
}
```

```
class SpecialList<E> extends ArrayList<E> {  
    public void add(E elem) { ... super.add(x); ... }  
}
```

- ❑ Ein Änderung der Methode `add` in der Klasse `ArrayList` ändert unmittelbar das Verhalten der Klasse `SpecialList`.
- ❑ Ein Änderung der Methode `remove` in der Klasse `ArrayList` ändert ebenfalls unmittelbar das Verhalten der Klasse `SpecialList`.

Testen objektorientierter Systeme

(Fortsetzung)

Folgen des Einsatzes von dynamischer Bindung:

- ❑ Der Programmablauf kann nicht unmittelbar aus Programmcode abgeleitet werden.
- ❑ Alle durch dynamisches Binden mögliche Abläufe müssen getestet werden.
- ❑ Methoden aus verschiedenen Stufen der Vererbungshierarchie arbeiten in verschiedenen Kombinationen zusammen und müssen in (allen) möglichen Kombinationen getestet werden.

Testen objektorientierter Systeme

(Fortsetzung)

Folgen des Einsatzes von dynamischer Bindung

Beispiel

```
public void sort(Comparable[] elems) { ... }
```

- ❑ Die Methode `sort` soll das übergebene Feld sortieren.
- ❑ Die Methode `sort` kann mit allen Feldern aufgerufen werden, deren Elemente das Interface `Comparable` implementieren, also eine `compareTo`-Methode bereitstellen.
- ❑ Die `sort`-Methode kann also auf Felder mit sehr unterschiedlichen Inhalten angewandt werden, soll aber für alle Aufrufe korrekt arbeiten.
- ❑ In dem als Parameter übergebenen Feld könnten auch Objekte verschiedener typkompatibler Klassen abgelegt sein. Dann ist bei der Ausführung möglicherweise entscheidend, welches Objekt die `compareTo`-Methode bereitstellt.
- ❑ Beim Implementieren und Testen der Methode `sort` kann immer nur von einer korrekten Implementierung der aufgerufenen `compareTo`-Methoden ausgegangen werden.
- ❑ Bei einer Nutzung der Methode `sort` muss deren *Eignung* aber möglicherweise in jedem Einzelfall durch Tests nachgewiesen werden.

Operationalisierung der Testdurchführung:

- ❑ Um Vertrauen in die Korrektheit einer Implementierung zu gewinnen müssen funktionsorientierte Tests
und
strukturorientierte Tests
immer gemeinsam durchgeführt werden.
- ❑ Schon kleine Beispiele zeigen,
dass häufig eine (sehr) große Zahl von Testfällen benötigt wird.
- ❑ Die Durchführung von Tests muss daher systematisch erfolgen.
- ❑ Tests müssen nachvollziehbar dokumentiert werden.
- ❑ Tests müssen nach jeder Änderungen an der Software wiederholt werden (können).

daraus folgt:

- ❑ Tests müssen als Testprogramme entwickelt werden,
die jederzeit erneut ausgeführt werden können.
- ❑ Hierfür gibt es Hilfsmittel (= Software-Werkzeuge), z.B. das Framework **JUnit**.

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 4.6: Testunterstützung durch JUnit

JUnit

- ❑ Java-Framework
 - Menge von Klassen, die durch Erben und Benutzen verwendet werden können
 - feste Ausführungskomponenten für die benutzten Klassen
 - aktuelle Version: 4.12
 - Download: <http://junit.org/>
- ❑ grundlegende Ideen:
 - Implementierung und Test werden in getrennten Klassen verwaltet
 - eigenes Testprogramm
 - standardisierte Implementierung der Tests
 - standardisierte Überprüfung der Testergebnisse

- ❑ Beispiel:

```
public class Simple {  
    private double value;  
    public Simple(double v) { value = v; }  
    public double getValue() { return value; }  
}
```

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
}
```

Test:

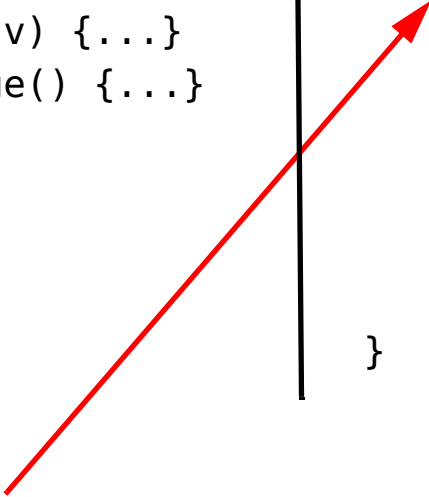
```
import org.junit.Test;  
import static org.junit.Assert.*;  
import org.junit.runner.RunWith;  
  
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(2.0,two.getValue(),0.01);  
    }  
    public static void main(String[] args) {  
        JUnitCore.run(SimpleTest.class);  
    }  
}
```

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
}
```

Test:

```
import org.junit.Test;  
import static org.junit.Assert.*;  
import org.junit.runner.RunWith;  
  
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(2.0,two.getValue(),0.01);  
    }  
    public static void main(String[] args) {  
        JUnitCore.run(SimpleTest.class);  
    }  
}
```



durch Annotation gekennzeichnete Methode, die einen Testfall implementiert:
`@Test public void test...()`
(So annotierte Methoden werden im Rahmen der Testausführung erkannt und ausgeführt.)

Exkurs: Annotationen in Java

- ❑ Annotationen sind eine Möglichkeit, Programmtexte mit zusätzlichen Informationen (Metadaten) anzureichern.
- ❑ Annotationen haben einen fest vorgegebenen syntaktischen Aufbau.
- ❑ Annotationen können
 - während der Übersetzung oder
 - während der Ausführung ausgewertet werden.
- ❑ Beispiele für vordefinierte Annotationen:
 - `@Override` markierte Methode überschreibt Methode der Oberklasse
 - `@Deprecated` «veraltete» Methode, sollte nicht genutzt werden
 - `@SuppressWarnings` unterdrückt Warnungen des Compilers

Implementierung:

```
public class Simple {
    private double value;
    public Simple(double v) {...}
    public double getValue() {...}
}
```

Test:

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;
```

```
public class SimpleTest {
    @Test public void testValue() {
        Simple two = new Simple(2.0);
        assertEquals(2.0, two.getValue(), 0.01);
    }
    public static void main(String[] args) {
        JUnitCore.run(SimpleTest.class);
    }
}
```

importiert statische Methoden so,
dass sie ohne Klassenprefix
verwendet werden können

Methode zur Überprüfung einer Bedingung:
assertEquals erwartet, dass Werte von erstem und zweitem
Parameter um höchstens den Wert des dritten Parameters
voneinander abweichen.

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
}
```

Test:

```
import org.junit.Test;  
import static org.junit.Assert.*;  
import org.junit.runner.JUnitCore;  
  
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(2.0,two.getValue(),0.01);  
    }  
    public static void main(String[] args) {  
        JUnitCore.run(SimpleTest.class);  
    }  
}
```



Ausführung der Testfälle (= mit @Test gekennzeichnete Methoden)
(Der explizite Aufruf entfällt in Programmier- und Entwicklungsumgebungen.)

assert...-Methoden

- ❑ assert-Methoden melden das Ergebnis ihrer Auswertung an das Framework.
- ❑ Das Framework sammelt diese Meldungen und erstellt einen Gesamtbericht zum Testerfolg.

Beispiele für assert-Methoden:

- ❑ `assertTrue(boolean c), assertFalse(boolean c)`
- ❑ `assertEquals(Object expected, Object actual),
assertEquals(String e, String a)`
(basieren auf Vergleich mit der `equals`-Methode der entsprechenden Klasse)
`assertEquals(int e, int a),
assertEquals(double e, double a, double delta), ...`
- ❑ `assertNull(Object o)`
`assertNotNull(Object o)`
- ❑ `assertSame(Object e, Object a)`
`assertNotSame(Object e, Object a)`
(Diese Methoden basieren auf dem Vergleich der Referenzen.)

Erweiterung

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
    public Simple add(Simple s) {...}  
}
```

Test:

```
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(two.getValue(),2.0,0.01);  
    }  
    @Test public void testAdd() {  
        Simple two = new Simple(2.0);  
        Simple sum = two.add(two);  
        assertEquals(4.0,sum.getValue(),0.01);  
        assertEquals(2.0,two.getValue(),0.01);  
    }  
}
```

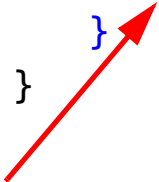
Erweiterung

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
    public Simple add(Simple s) {...}  
}
```

Test:

```
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(two.getValue(), 2.0, 0.01);  
    }  
    @Test public void testAdd() {  
        Simple two = new Simple(2.0);  
        Simple sum = two.add(two);  
        assertEquals(4.0, sum.getValue(), 0.01);  
        assertEquals(2.0, two.getValue(), 0.01);  
    }  
}
```



Prüfung, ob two
unverändert bleibt

Erweiterung

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
    public Simple add(Simple s) {...}  
}
```

gleiche Anweisungen zur
Vorbereitung der Testfälle



Test:

```
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(two.getValue(), 2.0, 0.01);  
    }  
    @Test public void testAdd() {  
        Simple two = new Simple(2.0);  
        Simple sum = two.add(two);  
        assertEquals(4.0, sum.getValue(), 0.01);  
        assertEquals(2.0, two.getValue(), 0.01);  
    }  
}
```

Erweiterung

Implementierung:

```
public class Simple {
    private double value;
    public Simple(double v) {...}
    public double getValue() {...}
    public Simple add(Simple s) {...}
}
```

@Before: Die Methode wird
vor jedem Testfall aufgerufen
und fasst Initialisierungen
zusammen.

analog:
@After: Die Methode wird
nach jedem Testfall aufgerufen.

Test:

```
public class SimpleTest {
    private Simple two;
    @Before public void setUp() {
        two = new Simple(2.0);
    }
    @Test public void testValue() {
        Simple two = new Simple(2.0);
        assertEquals(two.getValue(), 2.0, 0.01);
    }
    @Test public void testAdd() {
        Simple two = new Simple(2.0);
        Simple sum = two.add(two);
        assertEquals(4.0, sum.getValue(), 0.01);
        assertEquals(2.0, two.getValue(), 0.01);
    }
}
```

auch möglich in JUnit:

- ❑ Zusammenfassen von Testklassen zu *Testsuiten*, die dann gemeinsam ausgeführt werden
- ❑ Unterstützung des Testens von Ausnahmen
- ❑ Einschränkung:
Da die Testfälle in einer eigenen Klasse programmiert werden, ist **kein** Zugriff zu **privaten** Eigenschaften des getesteten Objekts möglich.

==> Eventuell müssen in der getesteten Klasse zusätzliche Methoden zur Unterstützung des Testens bereitgestellt werden.

alternativ: Build-In-Tests innerhalb der zu testenden Klasse

- ❑ Testwiederholung:
JUnit ermöglicht nach Änderungen an der Implementierung ein einfaches Wiederholen von (allen) Tests.

Debugger

Ein Debugger ist ein Software-Werkzeug, das bei der Lokalisierung von Fehlern eingesetzt wird.

- ❑ Ein Debugger ermöglicht es, die Anweisungen eines Programms einzeln oder bis zu einer bestimmten Position («Breakpoint») auszuführen.
- ❑ Nach Erreichen des Breakpoints hält die Ausführung an.
- ❑ Falls die Ausführung angehalten hat, ermöglicht der Debugger auch das Betrachten der Werte von Variablen und Attributen.
- ❑ Eventuell ermöglicht der Debugger auch ein Ändern der Werte von Variablen oder Attributen.
- ❑ Ein Debugger arbeitet immer interaktiv, erfordert eine manuelle Steuerung des Vorgangs und eine visuelle Prüfung des dargestellten Programmzustands.

- ❑ Der Debugger wird während der Implementierung oder nach dem Erkennen eines Fehlers eingesetzt.
- ❑ Der Debugger ist **nicht** für die allgemeine Überprüfung auf Korrektheit geeignet.

Zusammenfassung Testen

- ❑ funktionsorientierter Test:
Testfälle werden über Äquivalenzklassen aufgrund der Beschreibung bestimmt.
- ❑ strukturorientierter Test:
Testfälle werden aus dem Quellcode abgeleitet mit dem Ziel, bestimmte Überdeckungen zu erreichen.
- ❑ Visualisierung von Programmstrukturen:
UML-Aktivitätsdiagramme
- ❑ Testunterstützung durch Werkzeuge:
Beispiel **JUnit**

Zusammenfassung Testen

(Fortsetzung)

offensichtliche Beobachtung:

- ❑ Viele Änderungen erfordern häufiges Testen.

Lösung:

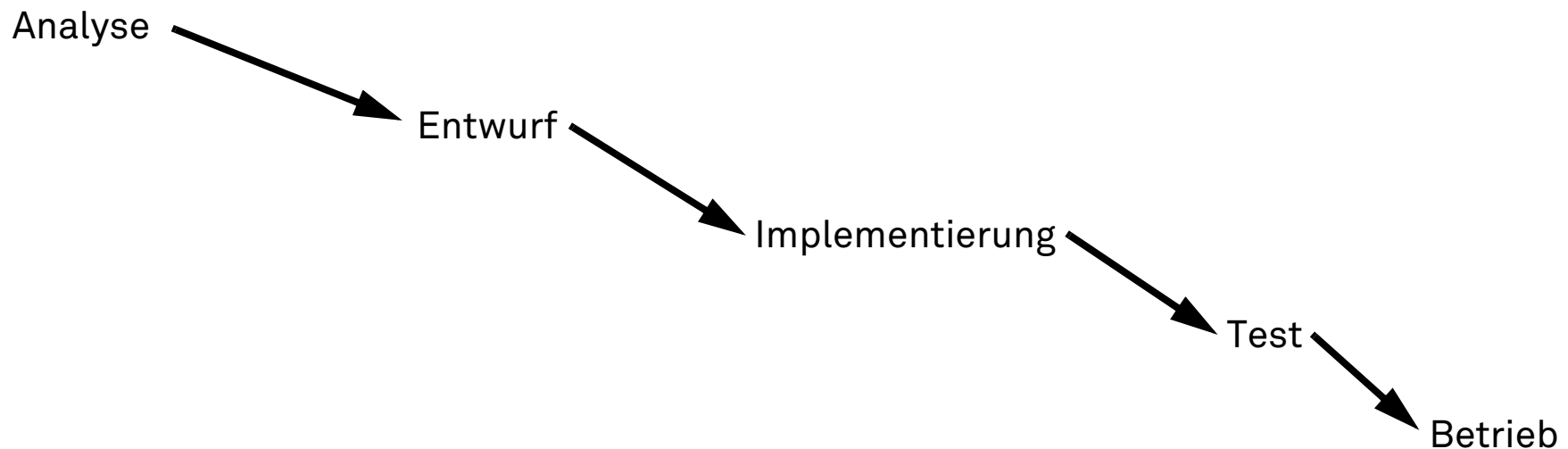
- ❑ **Regressionstest**
besteht aus der Wiederholung
aller für eine Vorversion erfolgreich ausgeführten Testfälle.
- ❑ Die Korrektheit der Testergebnisse kann teilweise durch den Vergleich
der Ergebnisse der aktuellen Version
mit den Ergebnissen der Vorgängerversion bestimmt werden.
- ❑ Regressionstests müssen automatisiert werden, da
 - viele Testfälle ständig wiederholt werden müssen und zugleich
 - in jedem Testdurchlauf nur wenige Fehler gefunden werden,
da geänderter Programmcode vor der Änderung immer bereits getestet war.

Folien zur Vorlesung **Softwaretechnik**

Teil 5: Vorgehensmodelle **Abschnitt 5.1: Motivation**

Organisation: Prozessmodelle/Vorgehensmodelle (Wiederholung/Folie 342)

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 (Wiederholung/Folie 343)

(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 (Wiederholung/Folie 345)

(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.**

Verbesserte Vorgehensmodelle

Inkrementelles Vorgehen:

- ❑ Das Softwareprodukt wird in mehreren, **vorab** geplanten Zyklen erstellt und erweitert.
- ❑ Nach einer ersten Anforderungsanalyse werden die Zyklen festgelegt.
- ❑ Jeder Zyklus umfasst Analyse, Entwurf, Implementierung und den Test für einen **vorab** festgelegten Teil des Produktes.

Vorteile:

- ❑ Spätere Zyklen können Ergebnisse und Erfahrungen früherer Zyklen berücksichtigen.
- ❑ Große Produkte können in überschaubare Teile zerlegt werden.

Nachteile:

- ❑ Das Vorgehen besitzt nur eingeschränkte Flexibilität, da die bereits entwickelten Inkremente erhalten bleiben sollen.
- ❑ Die frühe Planung geeigneter Zyklen ist schwierig und fehleranfällig.

Verbesserte Vorgehensmodelle

(Fortsetzung)

Spiralmodell (Boehm, 1988):

- ❑ Ein inkrementelles Vorgehen, bei dem die Zyklen **nacheinander** geplant werden.
- ❑ Softwareprodukt durchläuft in jedem Zyklus vier Segmente:
 - Planung der aktuellen Erweiterung,
 - Festlegen der Ziele dieser Erweiterung (Reifegrad des Produkts),
 - Risikoanalyse (Simulationen, Voruntersuchungen),
 - Entwicklung und Test.

Vorteile:

- ❑ Mehr Flexibilität durch inhaltlich weitgehend abgeschlossene Zyklen.
- ❑ Die Risikoanalyse zeigt frühzeitig Probleme auf.

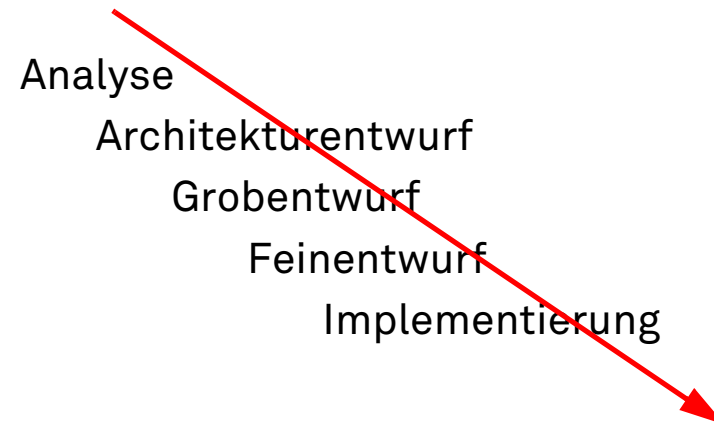
Nachteil:

- ❑ Der Aufwand vergrößert sich.

Organisation: Prozessmodelle/Vorgehensmodelle

(Fortsetzung)

V-Modell (ursprünglich auch von Boehm, 1981)

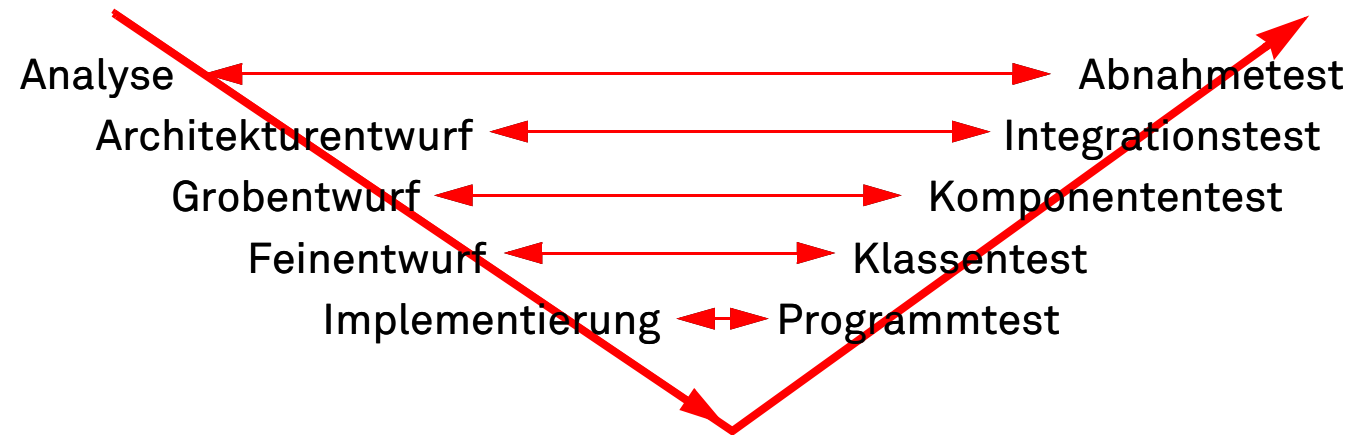


aus Wasserfall-Modell abgeleitet

Organisation: Prozessmodelle/Vorgehensmodelle

(Fortsetzung)

V-Modell (ursprünglich auch von Boehm, 1981)

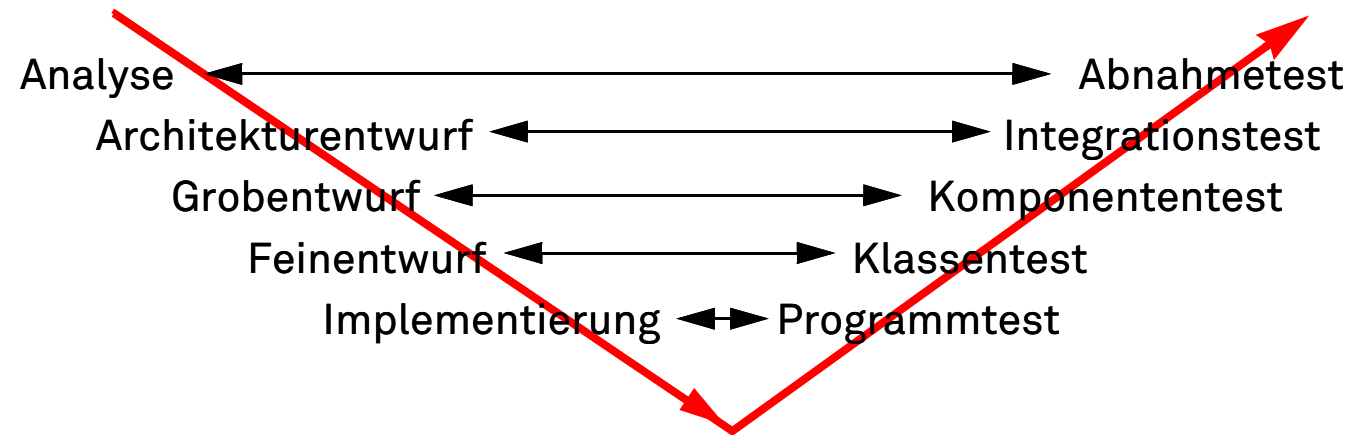


**stärkere Betonung der Qualitätssicherung
auf allen Ebenen der Entwicklung**

Organisation: Prozessmodelle/Vorgehensmodelle

(Fortsetzung)

V-Modell (ursprünglich auch von Boehm, 1981)



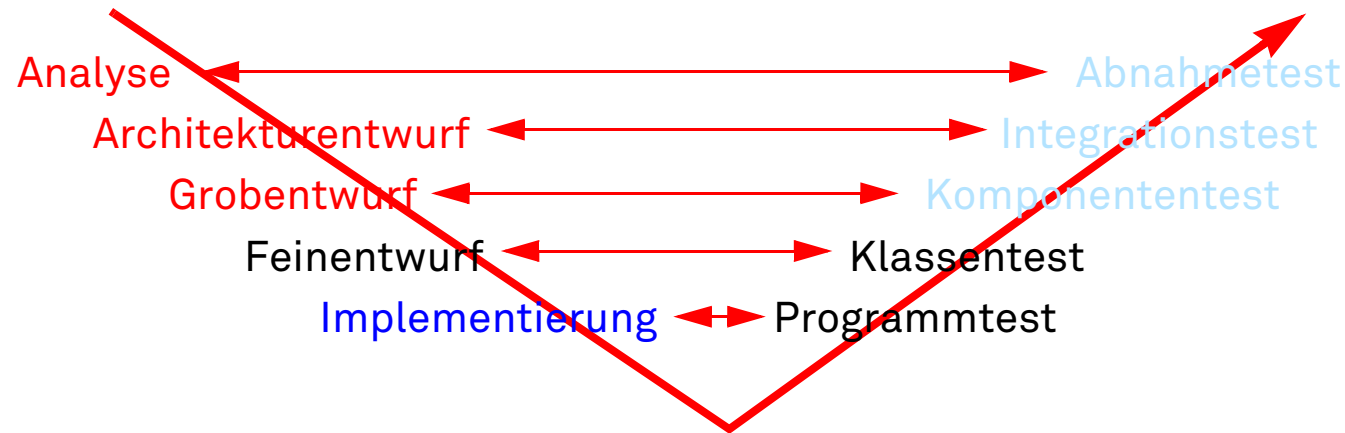
**Das Vorgehen ist heute als V-Modell XT
das fest vorgeschriebene Vorgehen für
Entwicklungsprojekte für den öffentlichen Bereich
in Deutschland.**

V-Modell XT (*eXtreme Tailoring*)

- ❑ V-Modell ist eine geschützte Marke der Bundesrepublik Deutschland.
- ❑ Das V-Modell wurde im Auftrag des Bundesministerium für Verteidigung seit 1986 entwickelt und die Verwendung ist dort vorgeschrieben seit 1991.
- ❑ Seit 1997 wird es auch im Bereich der zivilen Projekte der öffentlichen Verwaltung empfohlen: V-Modell '97
- ❑ Erweiterung zum seit 2005 veröffentlichten V-Modell XT (Version 2.0 seit 2015)
offizielle Web-Seite: <http://www.v-modell-xt.de>
- ❑ Das V-Modell XT fordert Vorgaben zu
Projektmanagement, Qualitätssicherung, Konfigurationsmanagement,
Problem- und Änderungsmanagement.
- ❑ Das V-Modell XT beschreibt die Beiträge von
Auftraggeber und Auftragnehmer.
- ❑ Das V-Modell XT umfasst das Referenzmodell (verpflichtender Teil) und
optionale, organisations- oder projektspezifische Anteile.
- ❑ Das V-Modell XT ist ein produktorientiertes Vorgehensmodell:
Das Ergebnis eines Entwicklungsschritts (Produkt) wird detailliert vorgegeben,
nicht der Vorgang der Erstellung des Produkts

Inhalte der Vorlesung Softwaretechnik

V-Modell



In dieser Vorlesung werden behandelt:

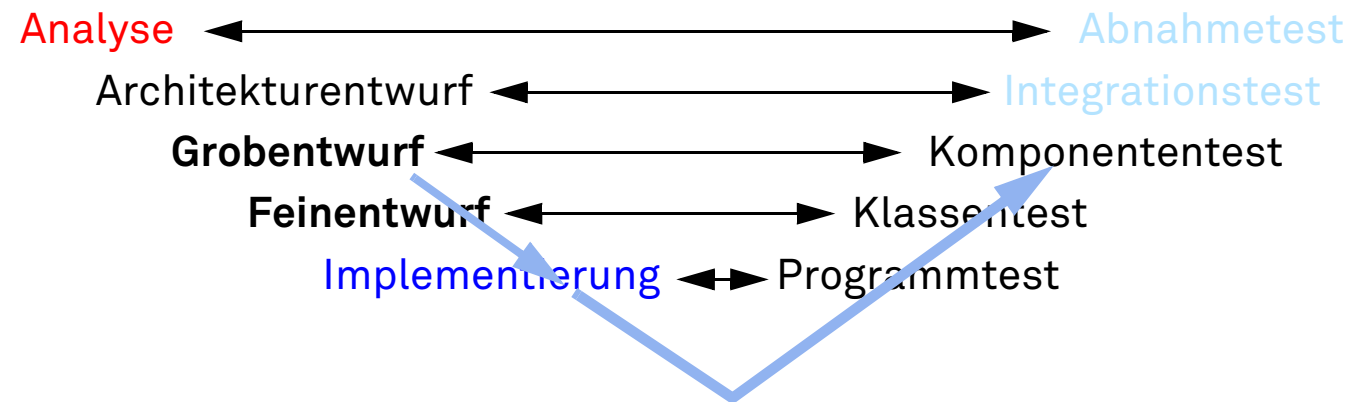
- ❑ Analyse
- ❑ Architekturentwurf
- ❑ Grobentwurf
- ❑ Feinentwurf: Entwurfsmuster, Beschreibung durch UML
- ❑ Implementierung: Java, Visualisierung durch UML
- ❑ Test: JUnit-Bibliothek, Unterstützung durch UML

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 5.2: Analyse

Rückblick

V-Modell



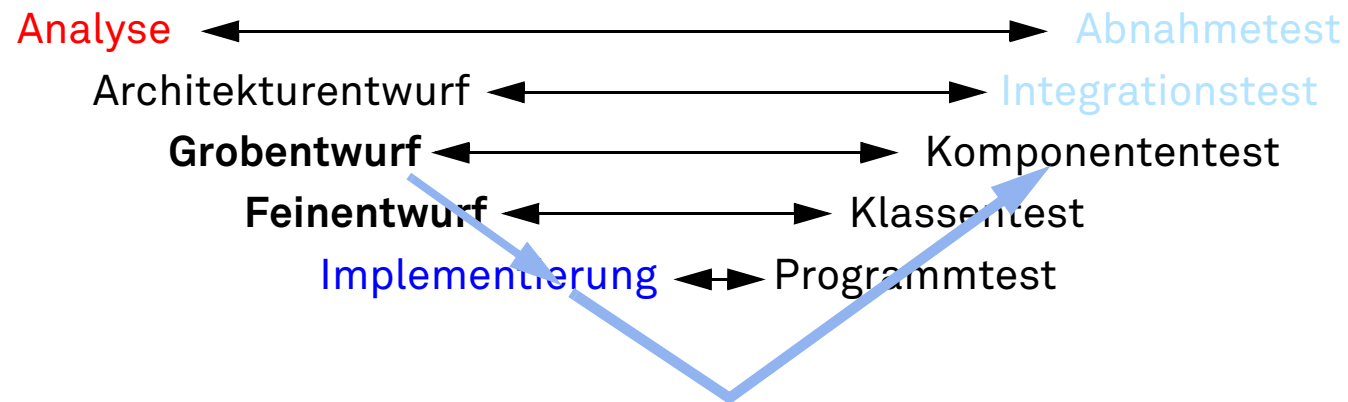
bisher im Rahmen des Entwurfs kennengelernt:

- ❑ Klassendiagramme zur Spezifikation/Dokumentation der Datenorganisation,
- ❑ Objektdiagramme zur Veranschaulichung von Objektstrukturen,
- ❑ Aktivitätsdiagramme zur Beschreibung von Algorithmen:
 - einzelne Methoden
 - grobes Zusammenwirken von Methoden
- ❑ Sequenzdiagramme zur Veranschaulichung von Abläufen mit mehreren Objekten

Rückblick

(Fortsetzung)

V-Modell



Mit den bisher kennengelernten Methoden, Konzepten und Notationen lässt sich Software

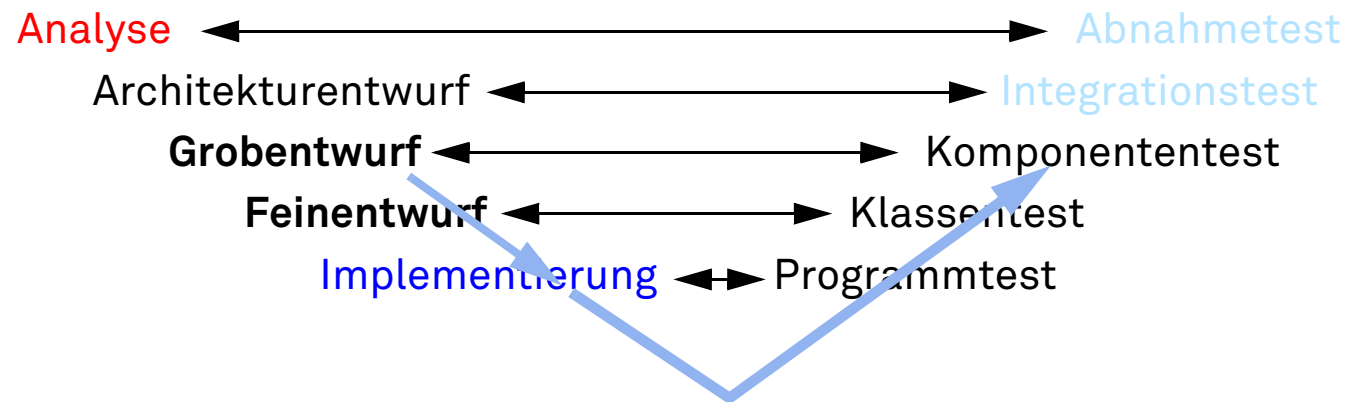
technisch

gestalten, veranschaulichen und prüfen.

Rückblick

(Fortsetzung)

V-Modell



Mit den bisher kennengelernten Methoden, Konzepten und Notationen lässt sich Software

technisch

gestalten, veranschaulichen und prüfen.

offene Fragestellung: **Was** soll die Software tun, die technisch realisiert wird?

Aufgaben der (Problem-)Analyse

Anforderungen an das zu entwickelnde System

- ☐ ermitteln,
- ☐ verstehen,
- ☐ klären,
- ☐ spezifizieren,
- ☐ dokumentieren.

Anforderung

Eine Anforderung ist eine Eigenschaft,
die ein System oder eine Person benötigt,
um ein Problem zu lösen oder ein Ziel zu erreichen

Literatur: Partsch, Helmuth A.: Requirements-Engineering systematisch – Modellbildung für softwaregestützte Systeme, S. 19-68
http://link.springer.com/chapter/10.1007/978-3-642-05358-0_2

Arten von Anforderungen

- ❑ Eine **funktionale Anforderung**
definiert eine vom System/einer Systemkomponente bereitzustellende Funktion.
Beispiele:
"Zugriff zu den Daten soll erst nach Eingabe eines Schlüsselworts möglich sein."
"Die Note soll als gewichteter Durchschnitt berechnet werden."

- ❑ Eine **Qualitätsanforderung**
definiert eine qualitative Eigenschaft eines Systems/einer Systemkomponente oder einer funktionalen Anforderung.
Beispiele:
Vorgaben für Zeitverhalten, Genauigkeit, Speicherbedarf

- ❑ Eine **Rahmenbedingung**
ist eine von außen kommende funktionale Anforderung oder Qualitätsanforderung, die im Rahmen der Entwicklung nicht geändert oder beeinflusst werden kann.
Beispiele:
gesetzliche Regelungen, ethische Aspekte, physikalische Gesetze

Arten von Anforderungen

(Fortsetzung)

Eine Anforderung beschreibt in der Regel das **Ziel**
und nicht seine technische **Umsetzung**.

Arten von Anforderungen

(Fortsetzung)

Eine Anforderung beschreibt in der Regel das **Ziel**
und nicht seine technische **Umsetzung**.

Allerdings sind **Ziel** und **Umsetzung** abhängig vom Betrachter!

- in der Analyse:
Das System soll gut abgesichert sein. ist **Zielvorgabe** des Auftraggebers.

Arten von Anforderungen

(Fortsetzung)

Eine Anforderung beschreibt in der Regel das **Ziel**
und nicht seine technische **Umsetzung**.

Allerdings sind **Ziel** und **Umsetzung** abhängig vom Betrachter!

- ❑ in der Analyse:
 - Das System soll gut abgesichert sein.* ist **Zielvorgabe** des Auftraggebers.
 - Es soll ein 256-bit-Schlüssel verwendet werden.* ist **Umsetzung** des Analysten.
- ❑ im Entwurf:
 - Es soll ein 256-bit-Schlüssel verwendet werden.* ist **Zielvorgabe** des Analysten.
 - Es soll der Rijndael-Algorithmus verwendet werden.* ist **Umsetzung** des Designers.

Arten von Anforderungen

(Fortsetzung)

Eine Anforderung beschreibt in der Regel das **Ziel**
und nicht seine technische **Umsetzung**.

Allerdings sind **Ziel** und **Umsetzung** abhängig vom Betrachter!

- ❑ in der Analyse:
 - Das System soll gut abgesichert sein.* ist **Zielvorgabe** des Auftraggebers.
 - Es soll ein 256-bit-Schlüssel verwendet werden.* ist **Umsetzung** des Analysten.
- ❑ im Entwurf:
 - Es soll ein 256-bit-Schlüssel verwendet werden.* ist **Zielvorgabe** des Analysten.
 - Es soll der Rijndael-Algorithmus verwendet werden.* ist **Umsetzung** des Designers.
- ❑ in der Implementierung:
 - Es soll der Rijndael-Algorithmus verwendet werden.* ist **Zielvorgabe** des Designers.
 - passender Programmcode in Java* ist **Umsetzung** des Programmierers.

Arten von Anforderungen

(Fortsetzung)

Eine Anforderung beschreibt in der Regel das **Ziel**
und nicht seine technische **Umsetzung**.

Allerdings sind **Ziel** und **Umsetzung** abhängig vom Betrachter!

- ❑ in der Analyse:
 - Das System soll gut abgesichert sein.* ist **Zielvorgabe** des Auftraggebers.
 - Es soll ein 256-bit-Schlüssel verwendet werden.* ist **Umsetzung** des Analysten.
- ❑ im Entwurf:
 - Es soll ein 256-bit-Schlüssel verwendet werden.* ist **Zielvorgabe** des Analysten.
 - Es soll der Rijndael-Algorithmus verwendet werden.* ist **Umsetzung** des Designers.
- ❑ in der Implementierung:
 - Es soll der Rijndael-Algorithmus verwendet werden.* ist **Zielvorgabe** des Designers.
 - passender Programmcode in Java* ist **Umsetzung** des Programmierers.

In der Analyse sollen die Anforderungen des Auftraggebers ermittelt werden,
die Zielvorgaben für den folgenden Entwurf sind.

Probleme in der Anforderungsanalyse

- ❑ Es sind viele Personen beteiligt, die direkt oder indirekt Einfluss auf Anforderungen nehmen:
 - spätere Benutzer
 - spätere Betreiber (technische Administration)
 - Experten des Anwendungsbereichs (Domänenexperten)
 - Management des Auftraggebers
 - Mitarbeitervertreter/Betriebsrat
 - eventuell Vertriebspersonal
 - Entwickler
 - ...
- ❑ Es wird angestrebt, **alle** Anforderungen zu erheben.
- ❑ Es wird angestrebt, **alle** Anforderungen **konsistent** zu beschreiben.
- ❑ Neben den eigentlichen Anforderungen müssen Kriterien für das Prüfen der Umsetzung von funktionalen und von qualitativen Anforderungen gefunden werden:
 - Es muss beschrieben werden, in welcher Weise eine Anforderung wie *einfach verständliche Bedienung* am fertigen Produkt geprüft werden kann.

Konsequenzen

Die Analyse ist ein **iterativer** Prozess aus:

- ❑ **Ermitteln** von Anforderungen bei einer Gruppe von Projektbeteiligten.
- ❑ **Verstehen** und **Klären** der ermittelten Anforderungen.
- ❑ **Spezifizieren** und **Dokumentieren** der ermittelten Anforderungen.
- ❑ **Validieren** und **Konsolidieren** der beschriebenen Anforderungen
 - mit der betroffenen Gruppe,
 - durch Vergleich mit anderen, bereits dokumentierten Anforderungen,
 - mit anderen beteiligten Gruppen.

Ermitteln von Anforderungen

- ❑ Verstehen der Anwendungsdomäne:
 - *Beispiel:* Das Ermitteln von Anforderungen für Software zur Kreditabwicklung erfordert Kenntnisse im Kreditgeschäft.
 - *Problem:* Wissenserwerb ist ein langwieriger Vorgang.
- ❑ Verstehen des konkreten Problems:
 - *Beispiel:* Die Abwicklung der Baufinanzierung bei der Bank X muss erlernt werden.
 - *Problem:* Für die Abwicklung bei X müssen Informationsquellen bestimmt werden.
- ❑ Verstehen des Geschäftsumfelds:
 - *Beispiel:* Der Beitrag der Software zum gesamtunternehmerischen Erfolg muss verstanden werden.
 - *Problem:* Hier hat z.B. das Management eine andere Sicht als die späteren Benutzer.
- ❑ Verstehen der beteiligten Geschäftsprozesse:
 - *Beispiel:* Software betrifft unterschiedliche Bereiche (Kreditwesen, Innenrevision).
 - *Problem:* Die veränderlichen Parameter dieser Bereiche, z.B. durch Gesetzesänderungen, müssen verstanden werden.

Ermitteln von Anforderungen

(Fortsetzung)

Informationsquellen zur Ermittlung von Anforderungen sind:

- ❑ Die Befragung von Personen,
die ein potentiell Interesse am zukünftigen System haben.
- ❑ Die Analyse von existierenden Dokumenten.
- ❑ Die Beobachtung von existierenden Abläufen.
- ❑ Die Analyse existierender Systeme:
 - Alt- oder Vorgängersysteme,
 - Konkurrenzprodukte,
 - ähnliche Systeme für andere Domänen.

Ermitteln von Anforderungen

(Fortsetzung)

Die durchzuführenden Tätigkeiten sind:

- ❑ Anforderungen erfassen, beschreiben und verfeinern.
- ❑ Szenarien erfassen und beschreiben.
Ein *Szenario* beschreibt ein konkretes Beispiel für die Erfüllung oder auch Nicht-Erfüllung von einer oder mehreren Anforderungen und konkretisiert diese dadurch.
- ❑ Lösungsorientierte Anforderungsaspekte erfassen und beschreiben:
 - Datenperspektive (strukturelle Beziehungen von Daten)
 - Funktionsperspektive (Manipulation von Daten durch Funktionen)
 - Verhaltensperspektive (Reaktion auf externe Signale, Zustandsänderungen)

Verstehen und Klären der ermittelten Anforderungen

- ❑ Prüfen der *Notwendigkeit*:
 - Leistet die Anforderung einen Beitrag zur Problemlösung?
- ❑ Prüfen der *Konsistenz*:
 - Stehen Anforderungen zueinander im Konflikt oder behindern sich?
 - Unterstützen sich Anforderungen, sind sie möglicherweise äquivalent?
- ❑ Prüfen der *logischen Vollständigkeit*:
 - Werden alle erfassten Daten verarbeitet?
 - Werden die zur Ausführung von Funktionen benötigten Daten bereitgestellt?
- ❑ Prüfen der *Machbarkeit*:
 - Sind die Anforderungen unter Zeit-, Personal-, Budgetrestriktionen umsetzbar?
 - Kann die Anforderungserfüllung geprüft werden?
- ❑ *Priorisierung*:
 - Die Bedeutung der Anforderungserfüllung für das Gesamtvorhaben wird festgelegt.

Verstehen und Klären der ermittelten Anforderungen

(Fortsetzung)

Die Analyse von Anforderungen ist zeitaufwändig und teuer:

- ❑ Experten müssen sich das durch die Anforderungen beschriebene System *vorstellen* und seine Ausführung – zumindest mental – *simulieren*.
- ❑ Die Konsequenzen des Zusammenwirkens aller Anforderungen müssen *durchdacht* werden.

Ein allgemeines systematisches Vorgehen für die Analyse lässt sich nicht angeben.

Validieren und Konsolidieren der ermittelten beschriebenen Anforderungen

- ❑ Überprüfen der gesammelten Anforderungen
 - mit Auftraggeber,
 - mit Betroffenen,
 - mit externen Experten.

- ❑ Prüfen von
 - Richtigkeit,
 - Verständlichkeit,
 - Nachvollziehbarkeit,
 - Verifizierbarkeit.

- ❑ Techniken:
 - Lesen, Vorstellen, Nachvollziehen
 - Prototyp-Erstellung,
 - Definition von Testfällen.

Spezifizieren und Dokumentieren von Anforderungen

Die Dokumentation kann erfolgen durch:

- ❑ natürlich-sprachliche Texte,
- ❑ standardisierte Formulare,
- ❑ graphische Notationen oder
- ❑ formale Spezifikationen.

Spezifizieren und Dokumentieren von Anforderungen

(Fortsetzung)

Inhalte einer solchen Dokumentation sind:

- ❑ eindeutige Identifikation einer Anforderung,
- ❑ Festlegen der Bedeutung der Anforderung,
- ❑ Angabe des Kontextes, in dem die Anforderung aufgetreten oder wichtig ist,
- ❑ ausführliche Beschreibung der Anforderung,
- ❑ Bezüge zu anderen Anforderungen.

Beispiel für Notation:
tabellarische Übersicht

Spezifizieren und Dokumentieren von Anforderungen

(Fortsetzung)

Tabellarische Dokumentation für **eine** Anforderung:

Abschnitt	Inhalt
Identifikation	Bezeichnung, Autoren, Version, Erstellungsdatum, Änderungshistorie
Kritikalität	Wichtigkeit der Anforderung, Bedeutung für den Erfolg
Kontext	Bezeichnung der Quellen, die die Anforderung genannt haben, Profiteure der Realisierung der Anforderung; Angabe der Detaillierungsebene der Anforderung
Beschreibung	ausführliche Beschreibung des Anforderung, Szenarien benennen, die die Anforderung erläutern
Bezüge zu anderen Anforderungen	übergeordnete Ziele: Angabe allgemeinerer Anforderungen, untergeordnete Ziele: detailliertere Anforderungen Konflikte: Konkurrenzbeziehungen zu anderen Anforderungen

Hinweis: Es sollen Ziele und nicht Realisierungen beschrieben werden.

Spezifizieren und Dokumentieren von Anforderungen

(Fortsetzung)

Dokumentation von Szenarien

- ❑ UML-Anwendungsfalldiagramm (*Use-Case-Diagramm*)
- ❑ tabellarische Beschreibungen
- ❑ UML-Sequenzdiagramm (*bereits bekannt*)
- ❑ UML-Aktivitätsdiagramm (*bereits bekannt*)

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 5.3: Anwendungsfalldiagramme

Spezifizieren und Dokumentieren von Anforderungen

(Fortsetzung)

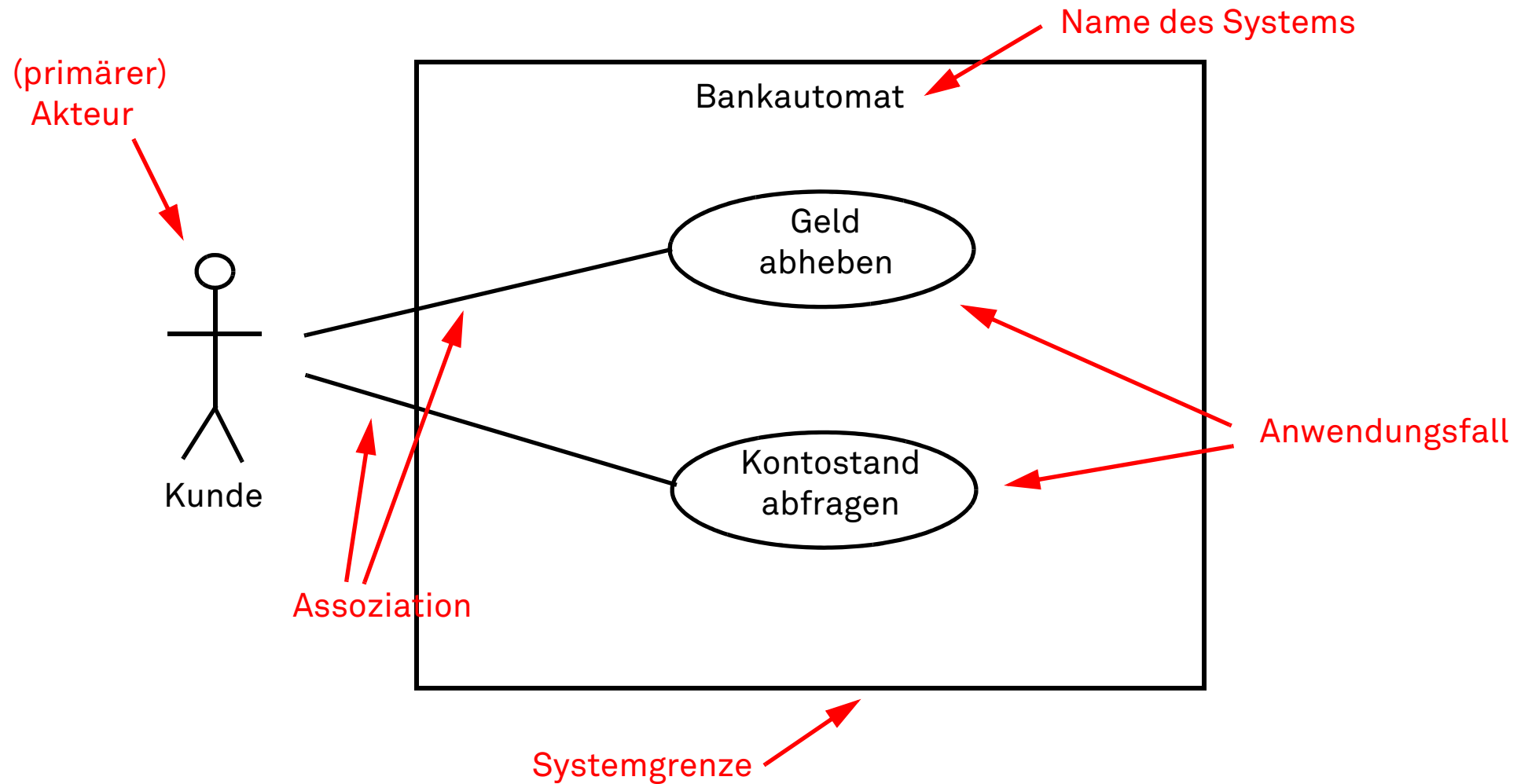
Anwendungsfalldiagramm

Die wesentlichen Modellierungselemente sind:

- ❑ **Akteur**
ist ein externes System,
das mit dem zu entwickelnden System interagiert:
Person, andere Hardware oder andere Software
- ❑ **Anwendungsfall** (engl. Use Case)
ist ein aus Sicht eines Akteurs zusammenhängendes Verhalten,
welches das zu entwickelnde System nach außen sichtbar anbieten soll.

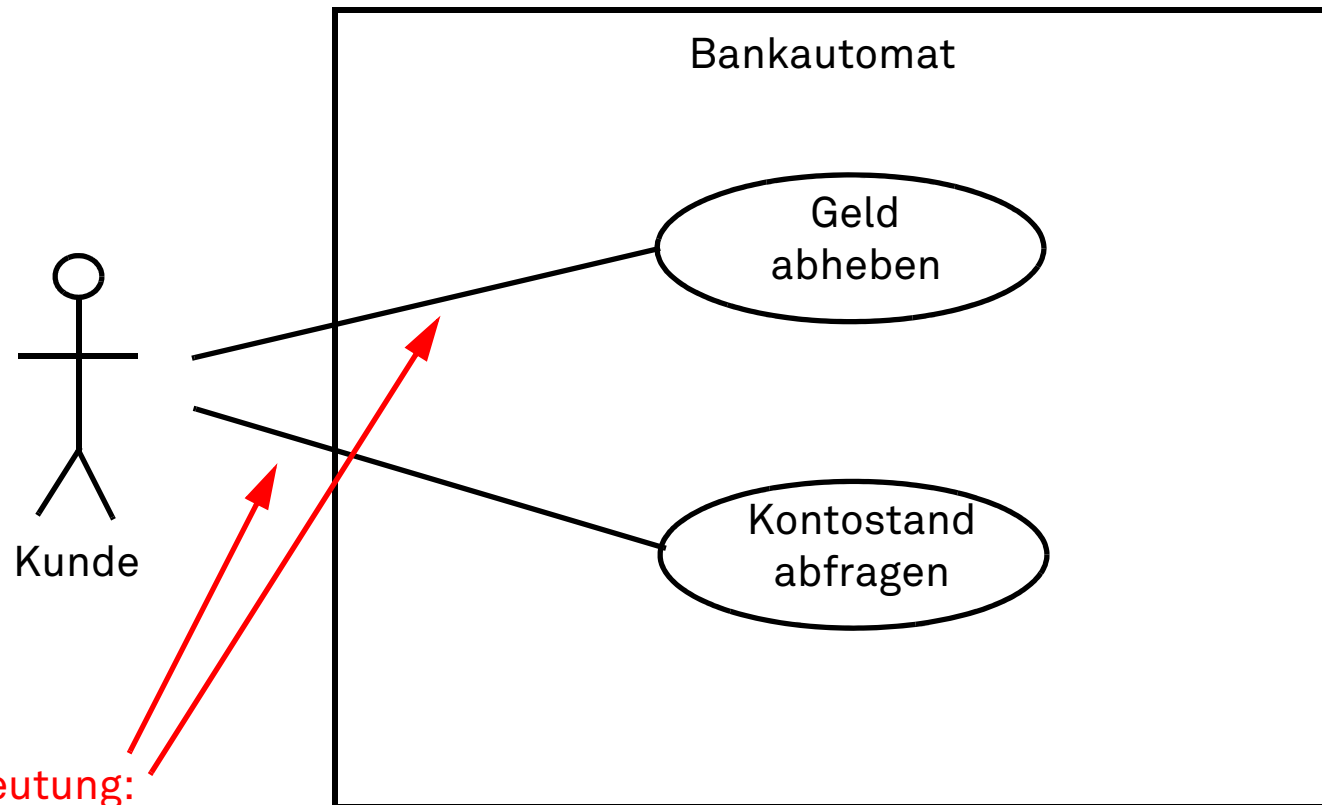
Literatur: Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 15-25
<http://www.springerlink.com/content/jm3124/#section=390797&page=1&locus=0>

Anwendungsfalldiagramm (Beispiel)



Anwendungsfalldiagramm

(Fortsetzung)



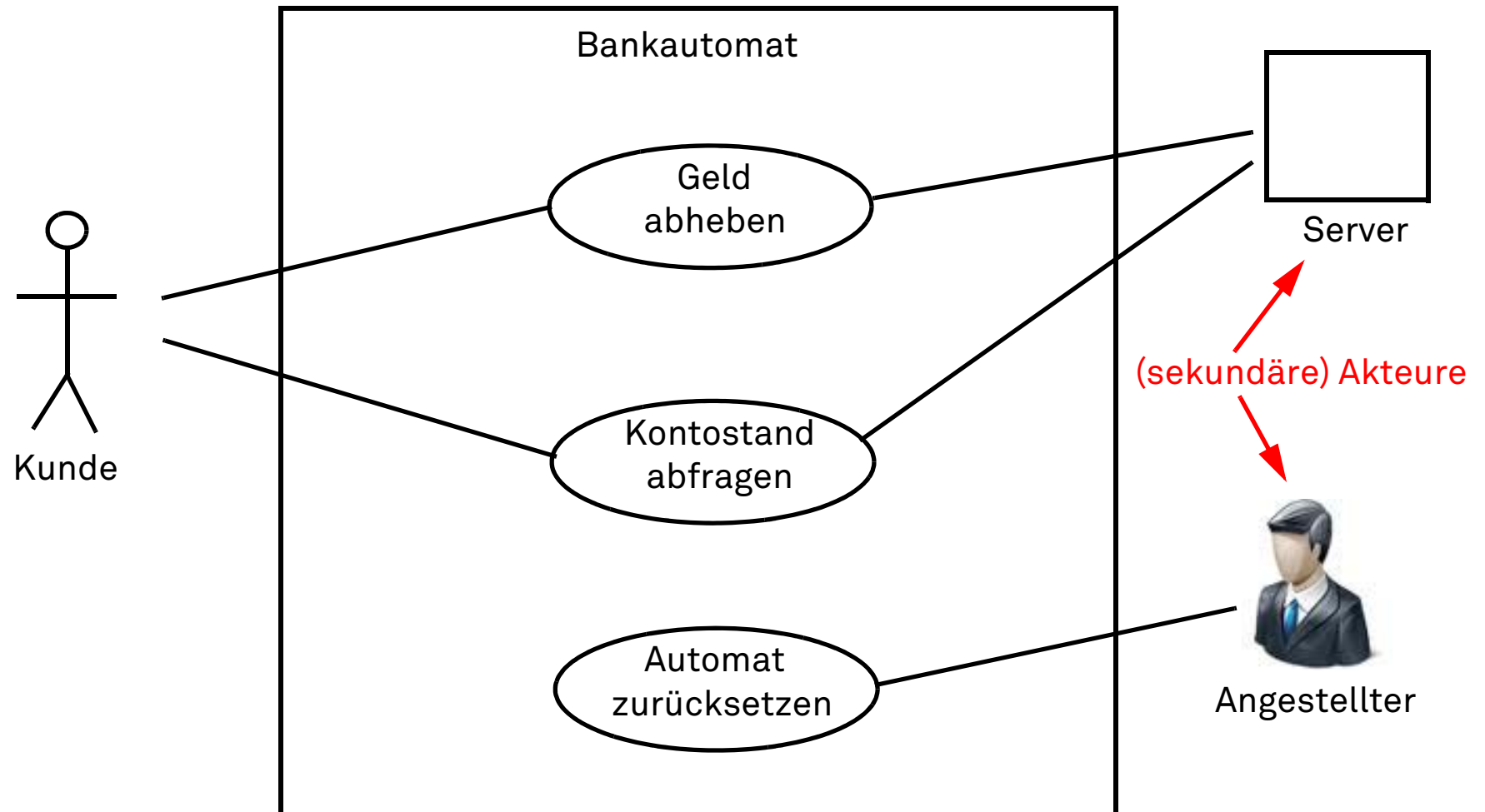
Bedeutung:

Kunde ist an den Anwendungsfällen *Geld abheben*
und *Kontostand abfragen* beteiligt.

Beide Anwendungsfälle können unabhängig voneinander ausgeführt werden.

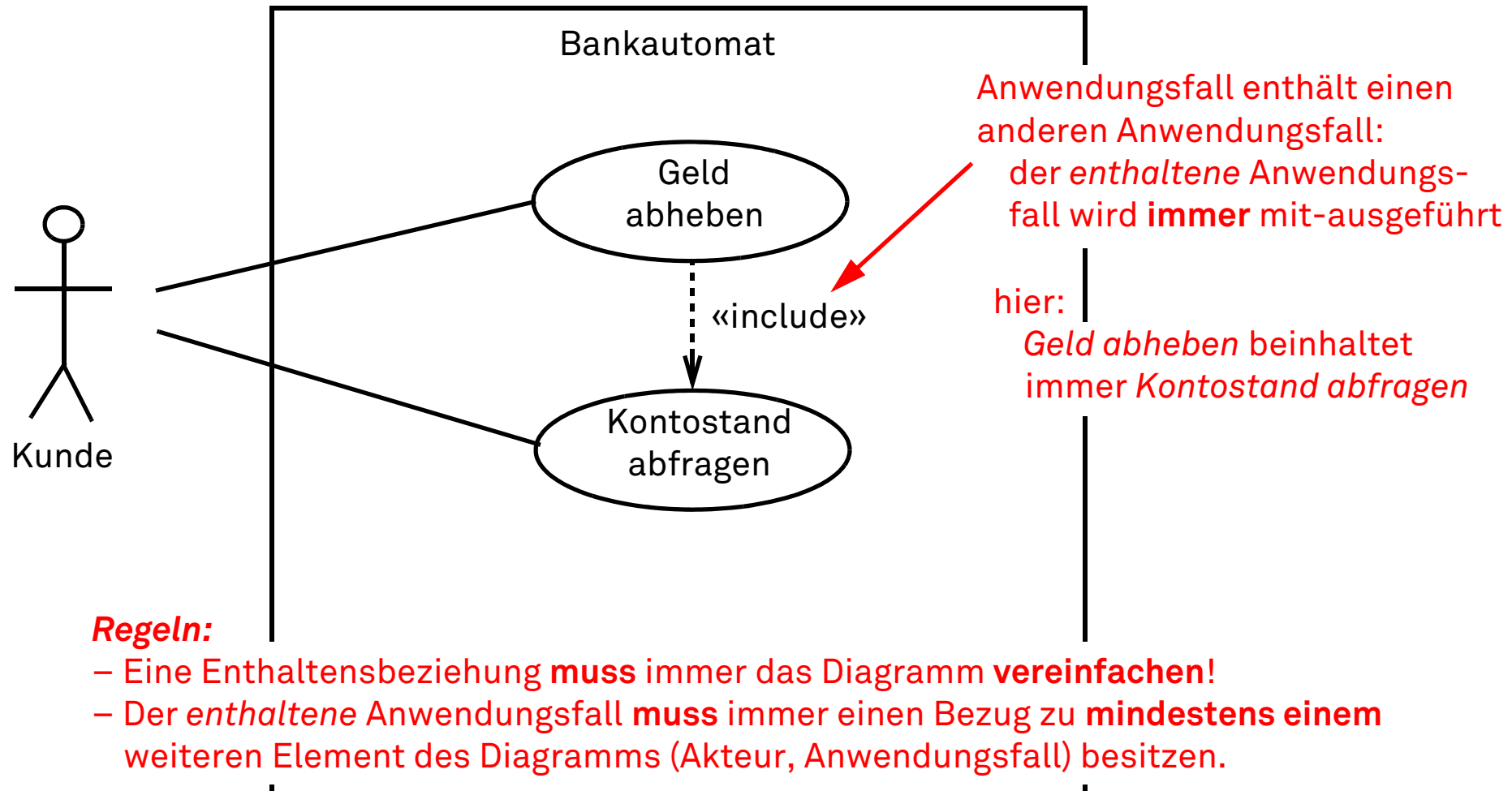
Anwendungsfalldiagramm

(Fortsetzung)



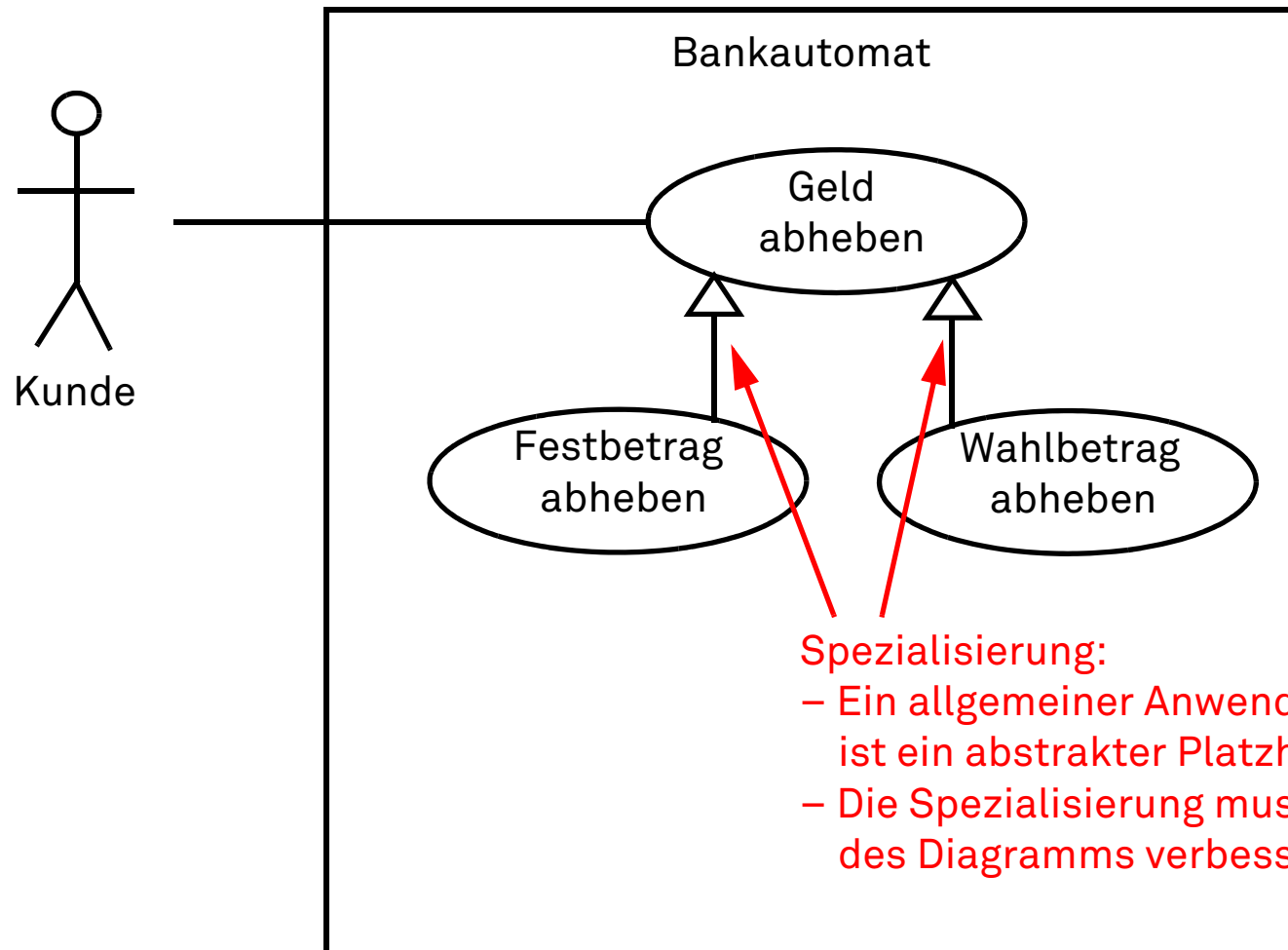
Anwendungsfalldiagramm

(Fortsetzung)



Anwendungsfalldiagramm

(Fortsetzung)

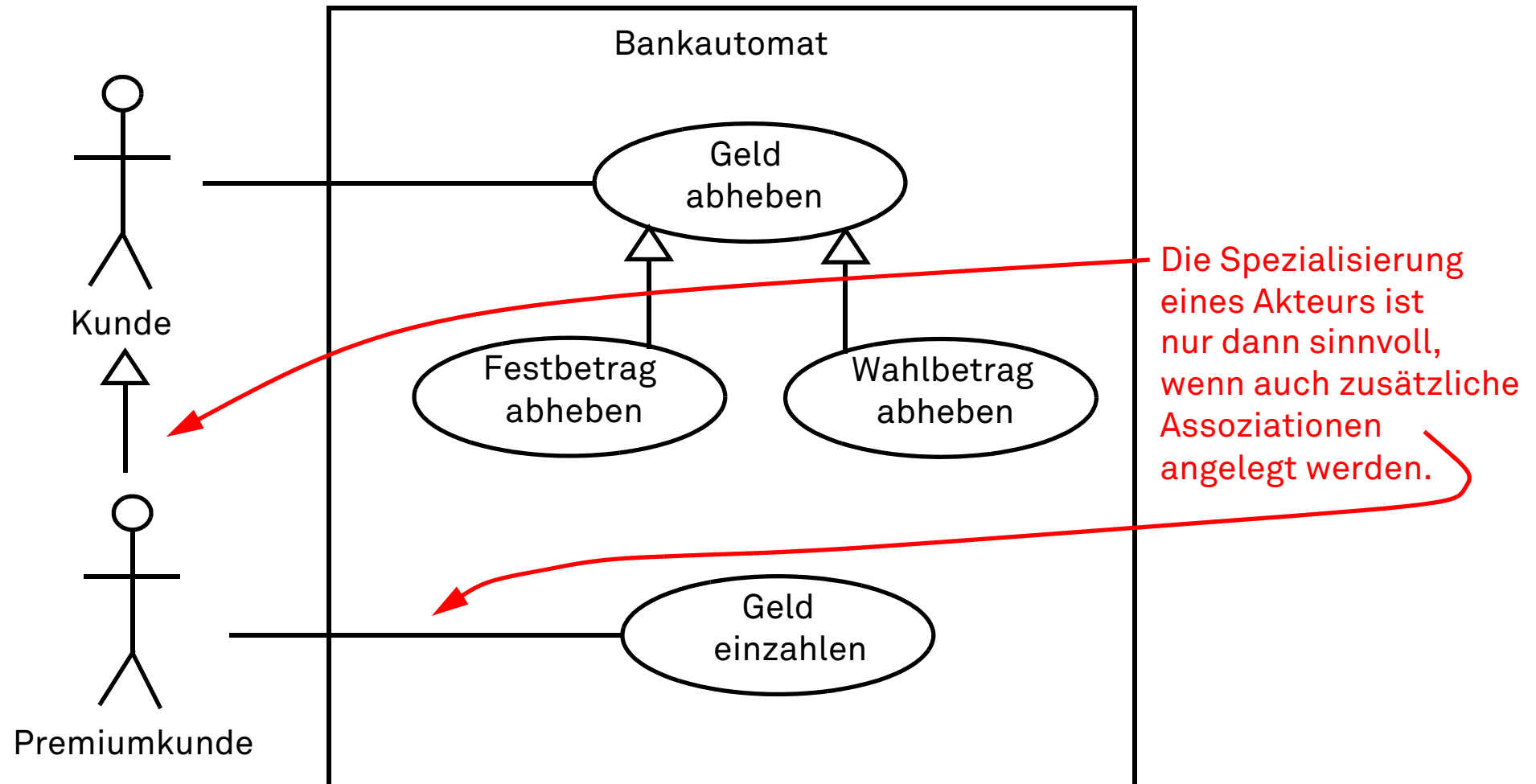


Spezialisierung:

- Ein allgemeiner Anwendungsfall ist ein abstrakter Platzhalter.
- Die Spezialisierung muss die Verständlichkeit des Diagramms verbessern.

Anwendungsfalldiagramm

(Fortsetzung)



Anwendungsfalldiagramm

(Fortsetzung)

zusätzlich **immer** notwendig: tabellarische Beschreibung für **jeden** Anwendungsfall

Abschnitt	Inhalt
Identifikation	Bezeichner des Anwendungsfalls, Autoren, Version, Erstellungsdatum, Änderungshistorie
Kritikalität	Wichtigkeit, Bedeutung für den Erfolg des Systems beschreiben
Ursprung	Bezeichnung der Quellen, auf die der Anwendungsfall zurück geht
Beschreibung	kurze Beschreibung des Anwendungsfalls, Ziele, die durch den Anwendungsfall erfüllt werden sollen
Vorbedingung	Voraussetzungen für die Ausführung des Anwendungsfalls
Nachbedingung	Zustand nach Ausführung des Anwendungsfalls
Hauptszenario	Beispiel für normalen Ablauf (tabellarisch; auch möglich als Sequenz- oder Aktivitätsdiagramm)
Alternativszenarien	Darstellung von Variationen des normalen Ablaufs
Ausnahmeszenarien	Abläufe, bei denen die des Anwendungsfalls Ziele nicht erreicht werden
Qualität	Bezüge zu Qualitätsanforderungen

Anwendungsfalldiagramm

(Fortsetzung)

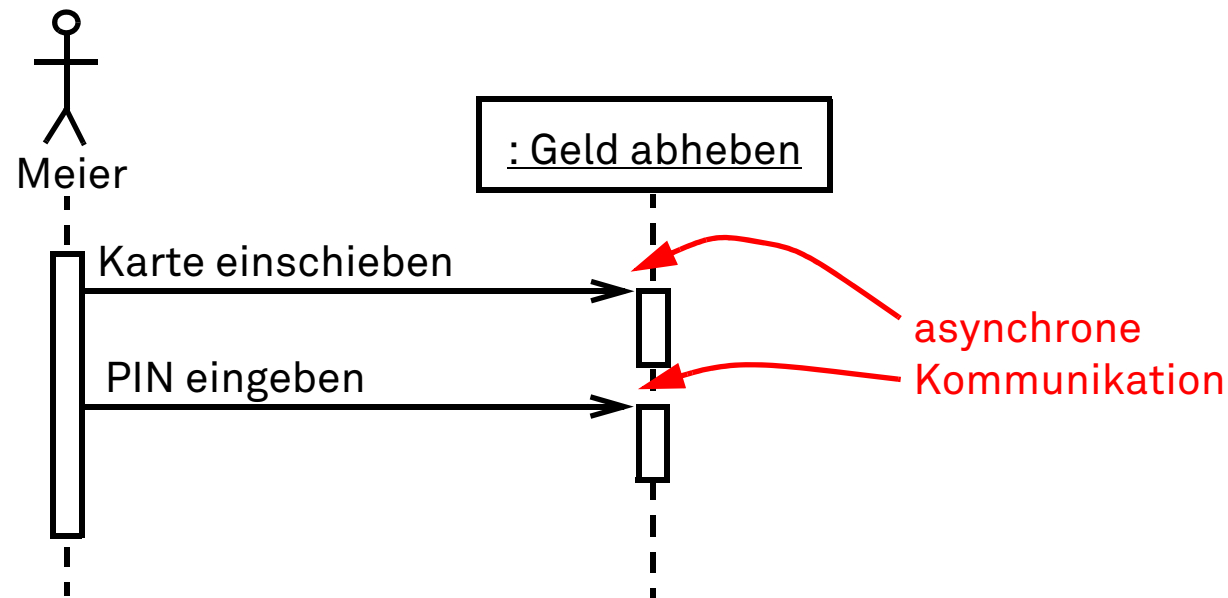
- ❑ Ein Anwendungsfalldiagramm zeigt
 - funktionale Anforderungen (= Anwendungsfälle),
 - Benutzer/externe Schnittstellen (= Akteure),
 - Interaktionsmöglichkeiten der Benutzer mit dem System (Assoziationen).
- ❑ Ein Anwendungsfalldiagramm zeigt **keine** Abläufe, sondern nur **strukturelle** Zusammenhänge.
- ❑ Ein Anwendungsfalldiagramm beschreibt die Konzeption aus Sicht der Akteure und **keine** technische (De-)Komposition

- ❑ Hinweis:
Daraus folgt, dass **nie** Assoziationen zwischen Anwendungsfällen auftreten.

Spezifizieren und Dokumentieren von Anforderungen: Sequenzdiagramm

Sequenzdiagramm
in der Analyse:

- ❑ Sequenzdiagramme ergänzen die Aussage von Anwendungsfalldiagrammen.
- ❑ Das Diagramm zeigt die Interaktion zwischen Akteuren und Anwendungsfällen.
- ❑ Die Kanten zeigen den *Nachrichtenfluss* (= die Weitergabe von Informationen).
- ❑ Die Nachrichten sind asynchron.
- ❑ Die Nachrichten werden informell beschrieben.



Folien zur Vorlesung **Softwaretechnik**

Abschnitt 5.4: Aktivitätsdiagramme (Ergänzung zu Teil 4.2)

Spezifizieren und Dokumentieren von Anforderungen: Aktivitätsdiagramm

Aktivitätsdiagramm
in der Analyse:

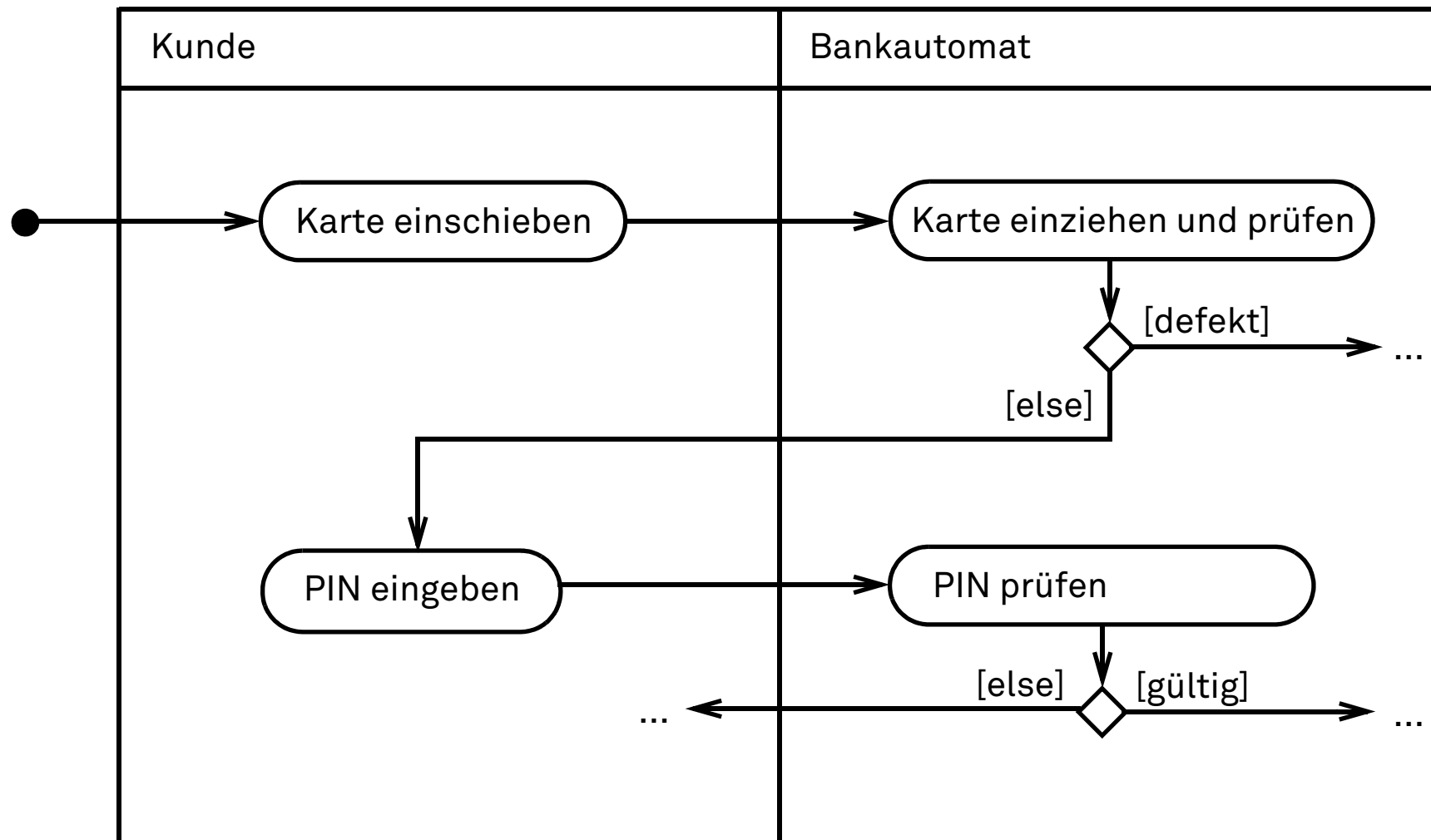
- ❑ Das Diagramm zeigt eventuell die Folge der Interaktionen zwischen Akteuren und Anwendungsfällen.
- ❑ Die Interaktionen werden informell beschrieben.
- ❑ Hilfsmittel zur Strukturierung von Aktivitätsdiagrammen:

Verantwortungsregionen

Region 1	Region 2	Region 3

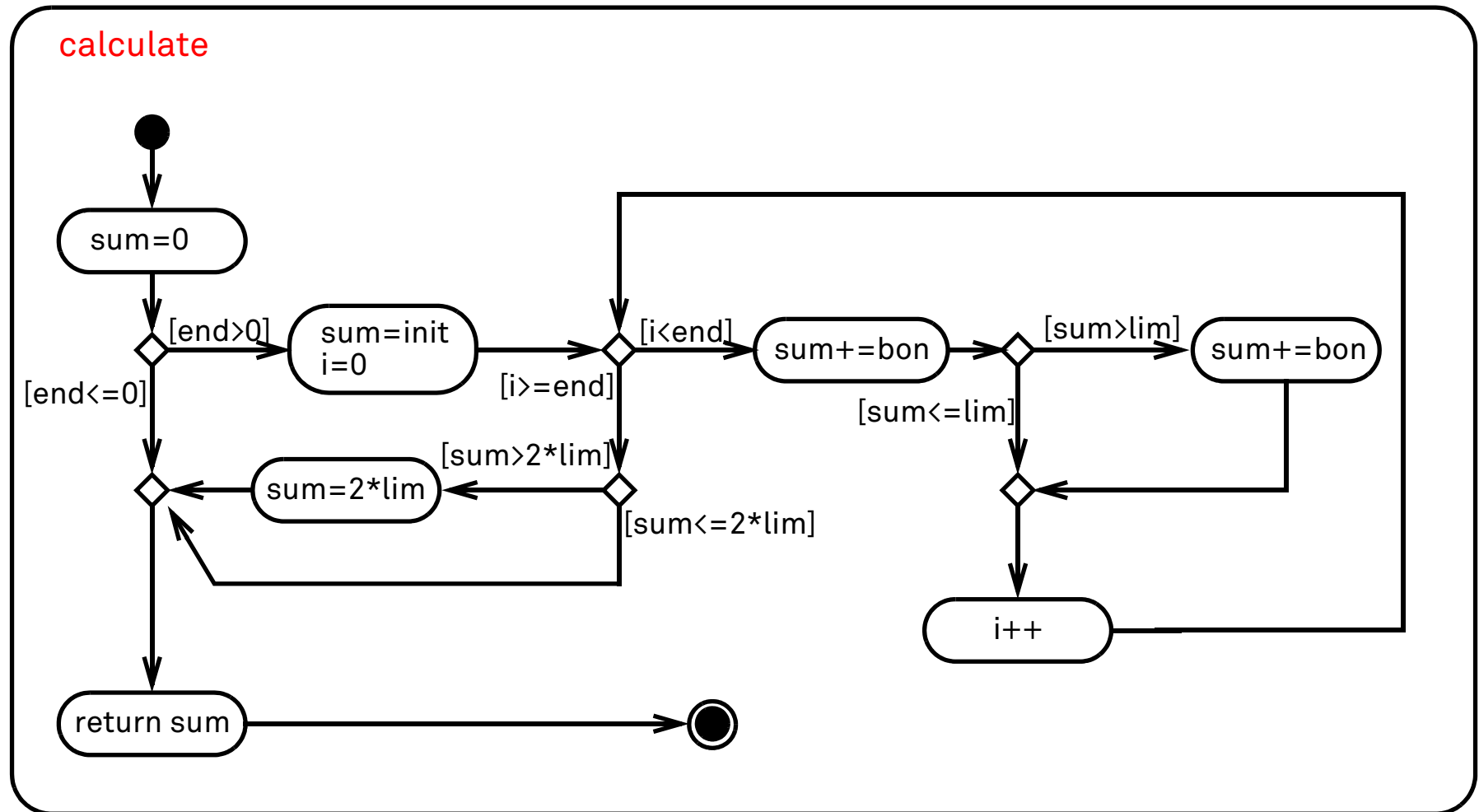
Dokumentieren von Anforderungen: Aktivitätsdiagramm

(Fortsetzung)



Aktivitätsdiagramm (Wiederholung Folie 360)

= Spezifikation eines Verhaltens als koordinierte Folge der Ausführung von Aktionen



Aktivitätsdiagramm – weitere Modellierungselemente

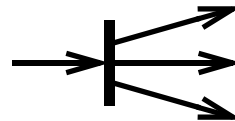
bisher:

- ❑ Aktivitätsdiagramme zur Visualisierung/Planung von Java-Programmen eingesetzt
- ❑ Folge: nur sequentielle Abläufe modelliert
- ❑ Im Aktivitätsdiagramm wird nur eine Marke erzeugt.

Ergänzen:

- ❑ Elemente zur Modellierung von nebenläufigen Prozessen durch Aktivitätsdiagramme:
 - Verteilungsknoten
 - Synchronisationsknoten
- ❑ Folge: Modellierung von Abläufen mit autonom handelnden Akteuren ist möglich.

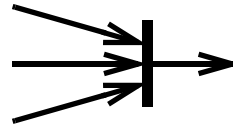
Verteilungsknoten



- ❑ Jede eingehende Marke wird so oft **vervielfacht**,
dass auf jeder Ausgangskante genau eine Marke weitergeleitet wird.
- ❑ Da mehrere Marken erzeugt werden,
entstehen mehrere nebenläufige Ausführungsstränge:

⇒ Mehrere Aktionen einer Aktivität können gleichzeitig aktiv sein.

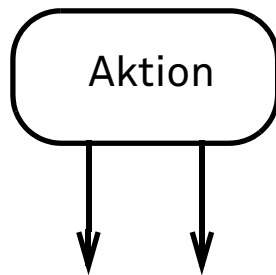
Synchronisationsknoten



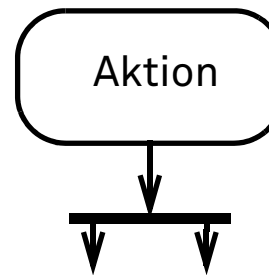
- ❑ Der Synchronisationsknoten schaltet nur genau dann, wenn auf **allen** eingehenden Kanten eine Marke eintrifft.
 - ❑ Auf **jeder** Eingangskante wird dabei genau eine Marke vernichtet, auf der Ausgangskante wird zeitgleich **genau eine einzige** Marke erzeugt.
 - ❑ Es werden so mehrere nebenläufige Ausführungen zusammengeführt und zeitlich synchronisiert.
-
- ❑ Die Kombination von Verteilungs- und Synchronisationsknoten muss Eigenschaften beider Typen besitzen.

implizite Verteilung und Synchronisation

- Aktionen und Verteilung:



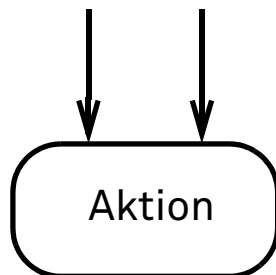
entspricht



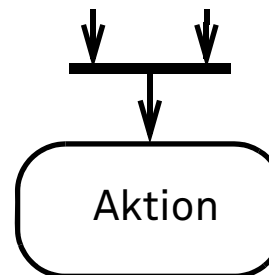
übersichtlichere Darstellungsform

- Aktionen und Synchronisation:

Vorsicht!



entspricht



Aktionen sollten also immer nur genau eine Eingangs- und genau eine Ausgangskante besitzen!

Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

Problemstellung:

Bei der Bestellung in einem Online-Shop soll der Kunde zunächst drei Tätigkeiten in **beliebiger** Reihenfolge durchführen können:

- ☐ das Angeben der Lieferanschrift,
- ☐ das Angeben der Kontodaten für die aktuelle Bestellung,
- ☐ das sequentielle Auswählen von Produkten.

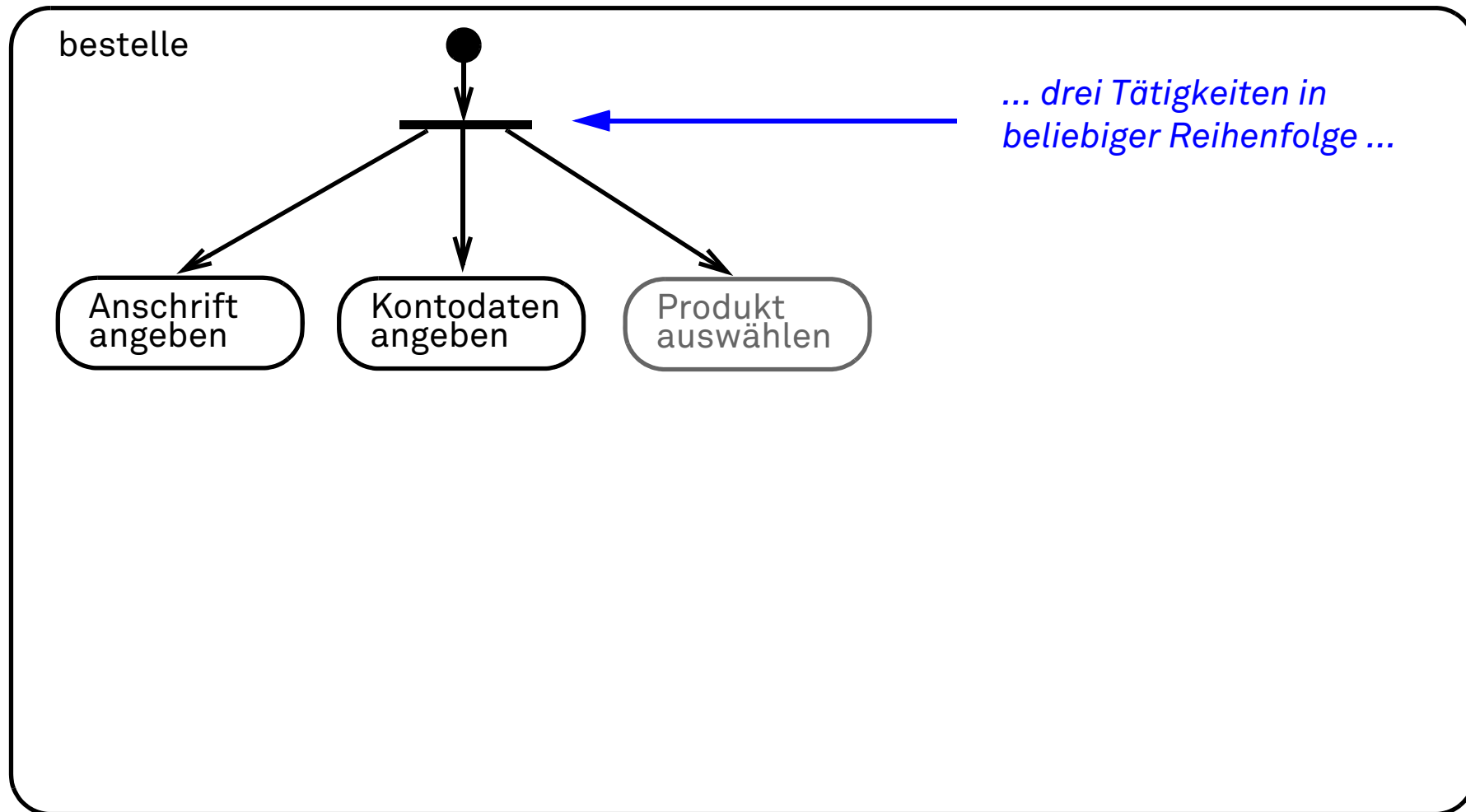
Nach Abschluss der Produktauswahl und der Angabe der Kontodaten soll dem Kunden eine Bestellübersicht angezeigt werden.

Aufgabe:

Dieser Vorgang soll als Aktivitätsdiagramm modelliert werden.

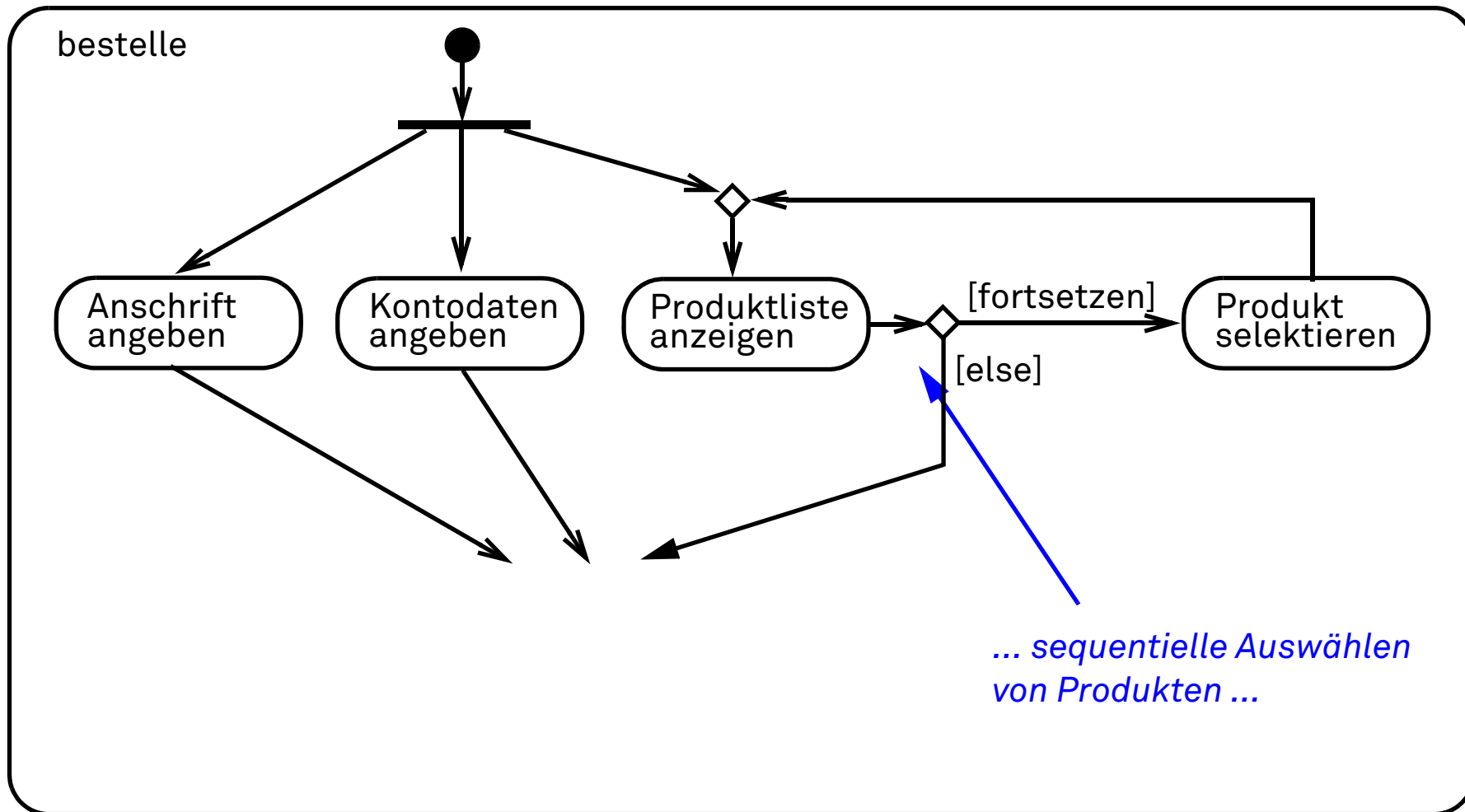
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



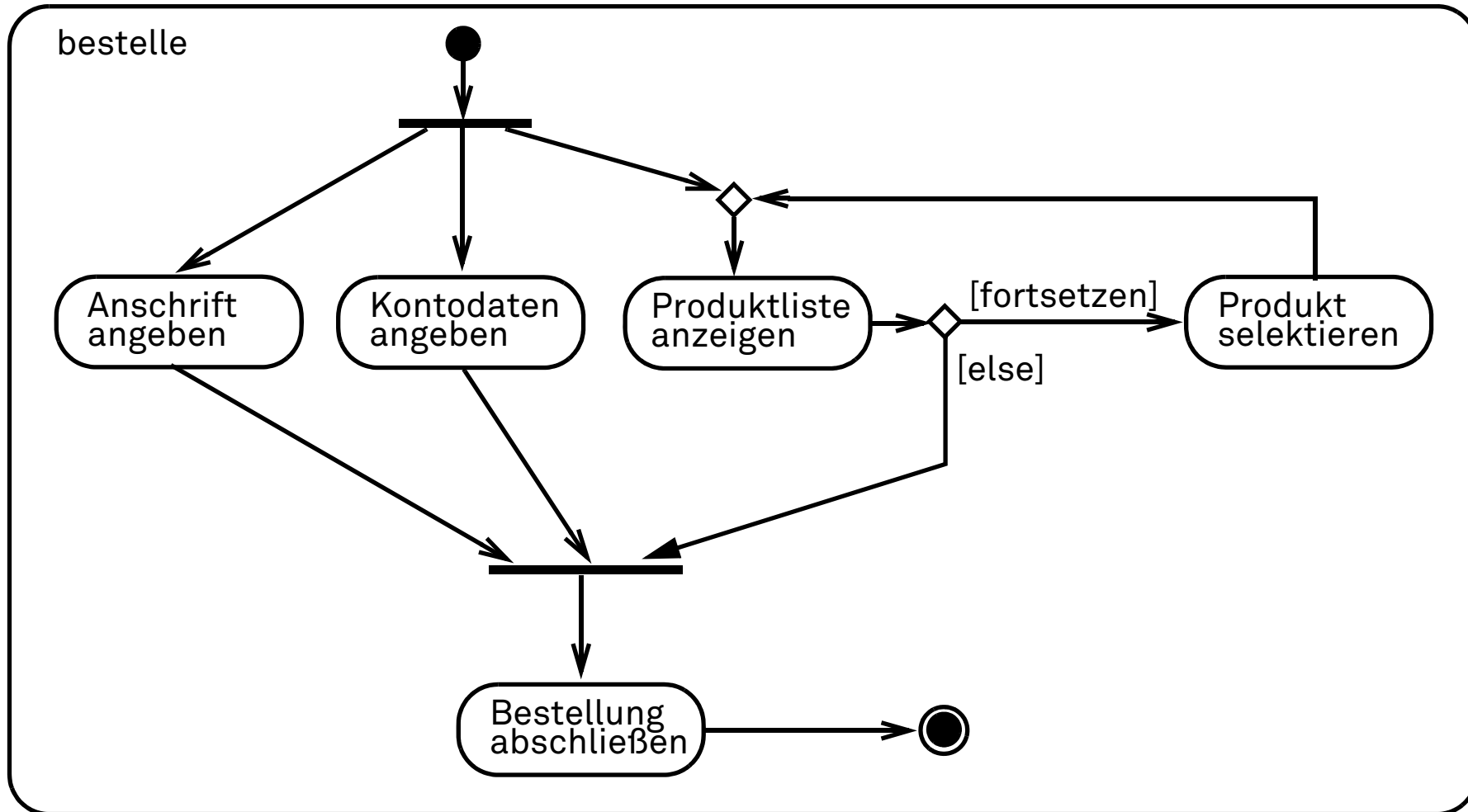
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



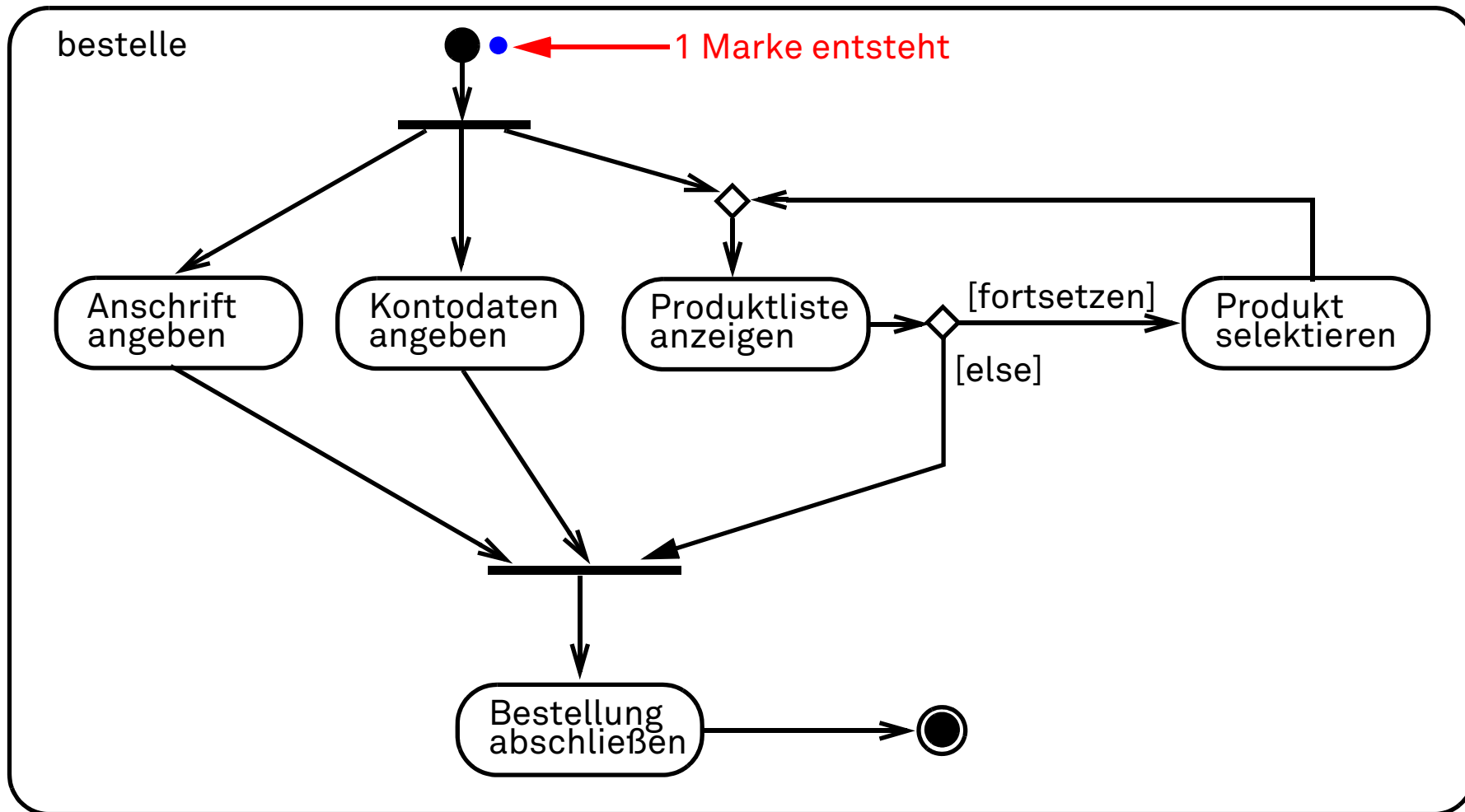
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



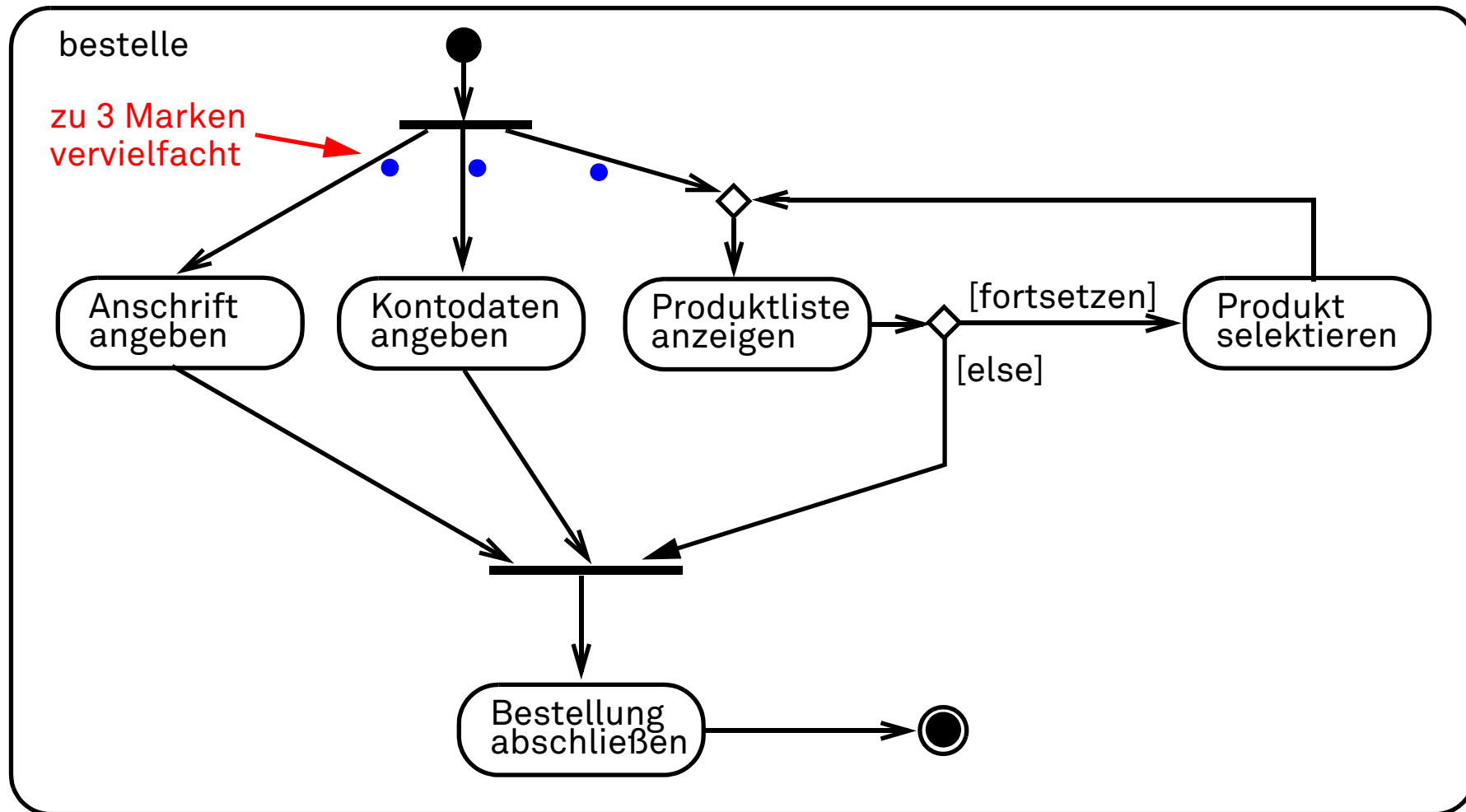
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



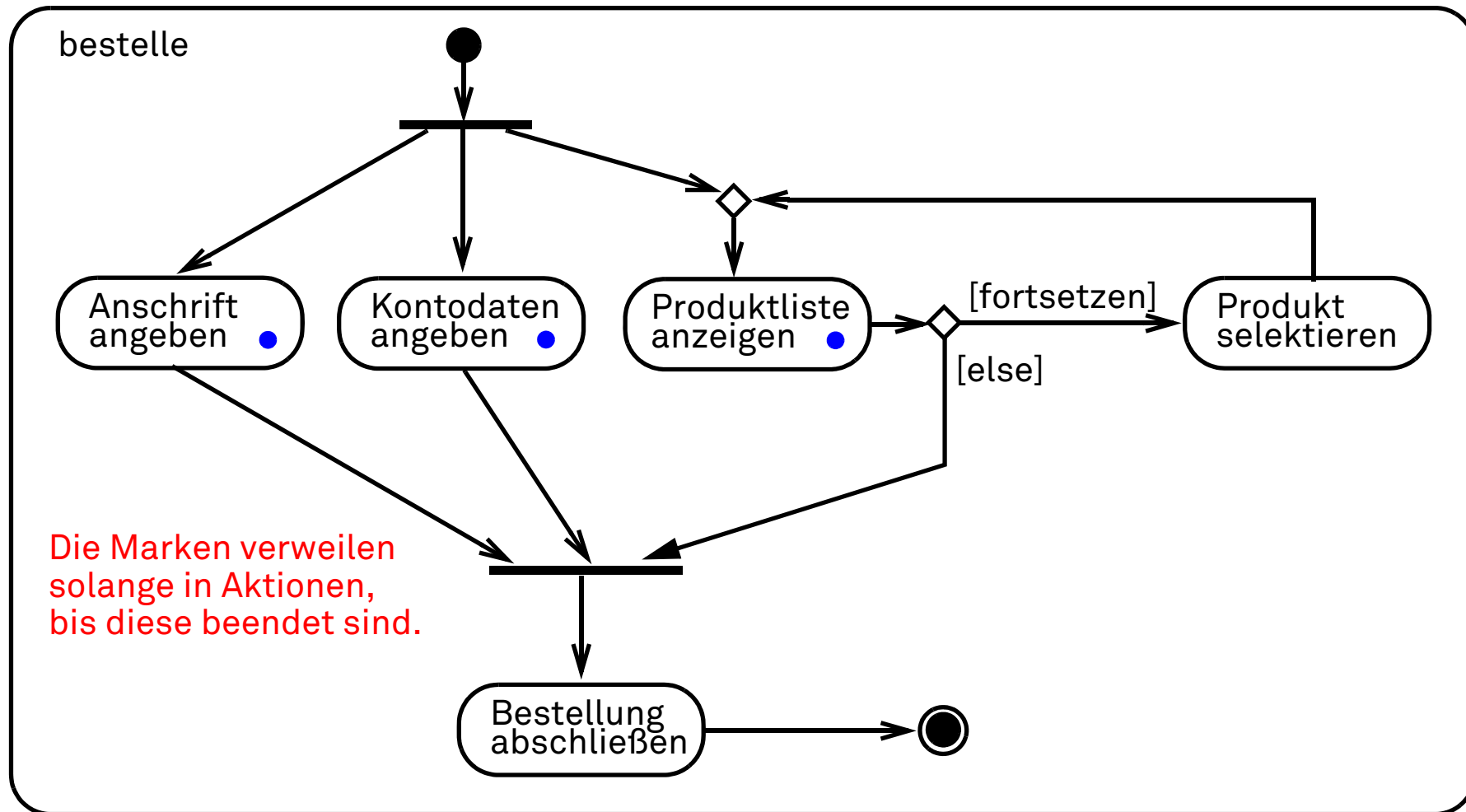
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



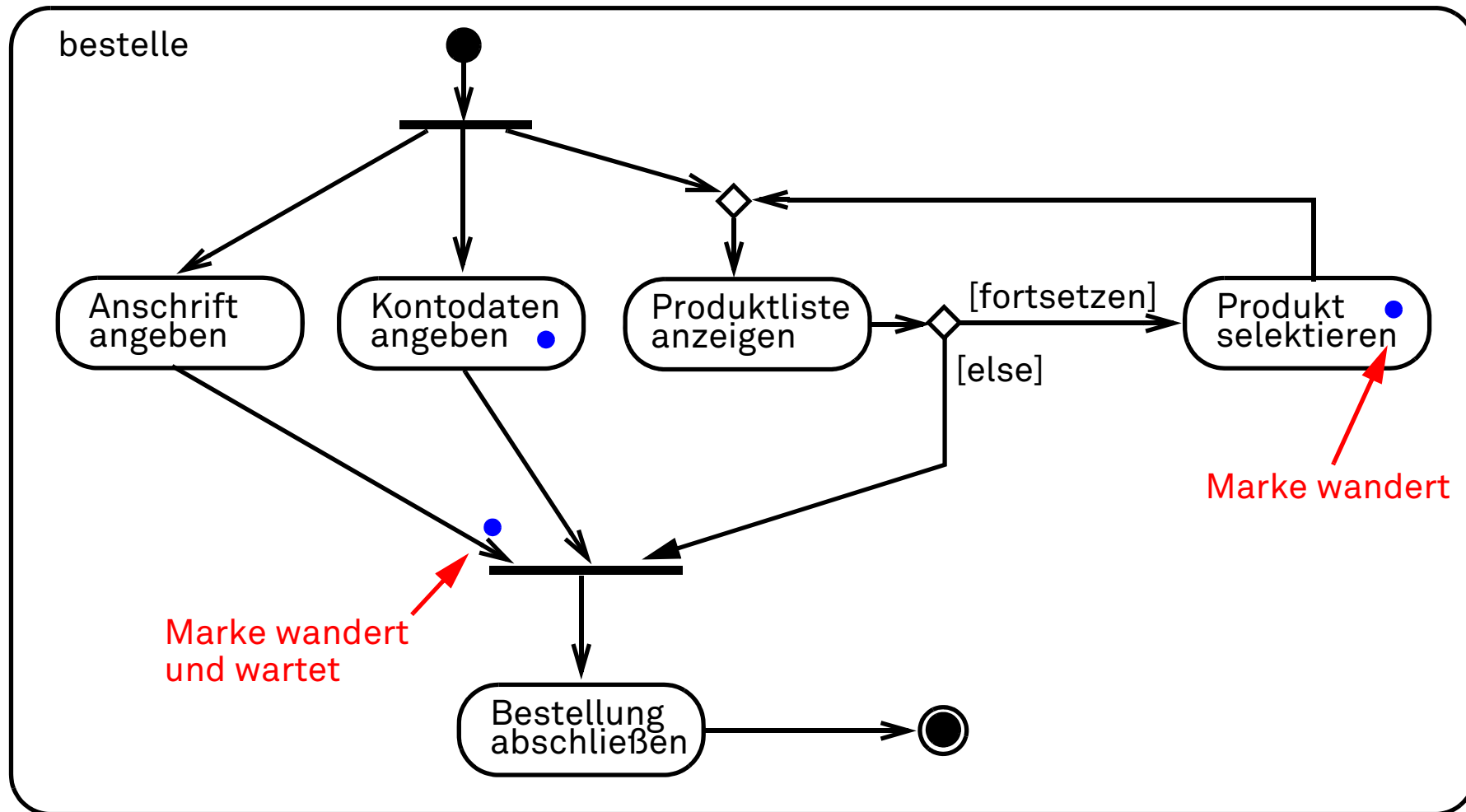
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



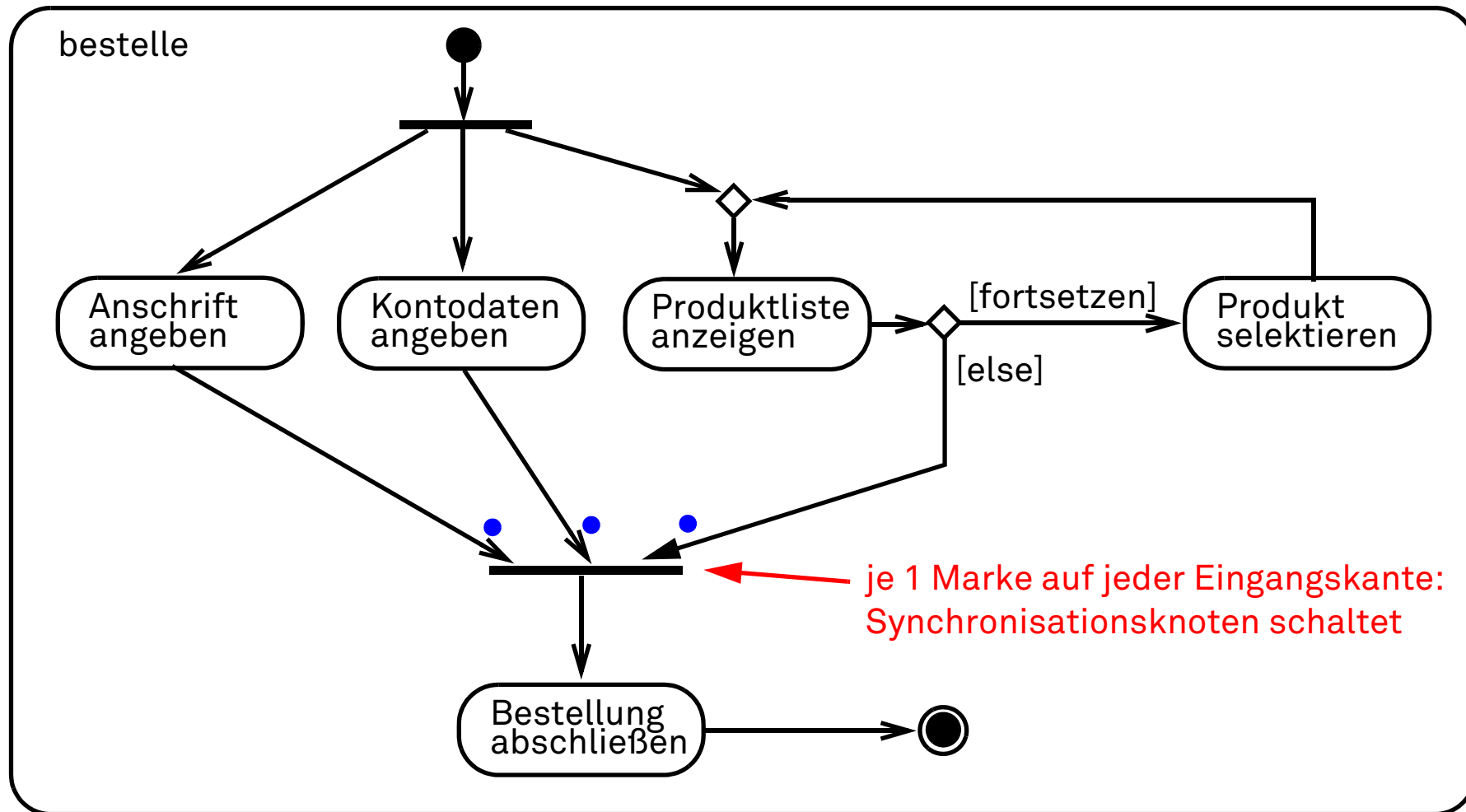
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



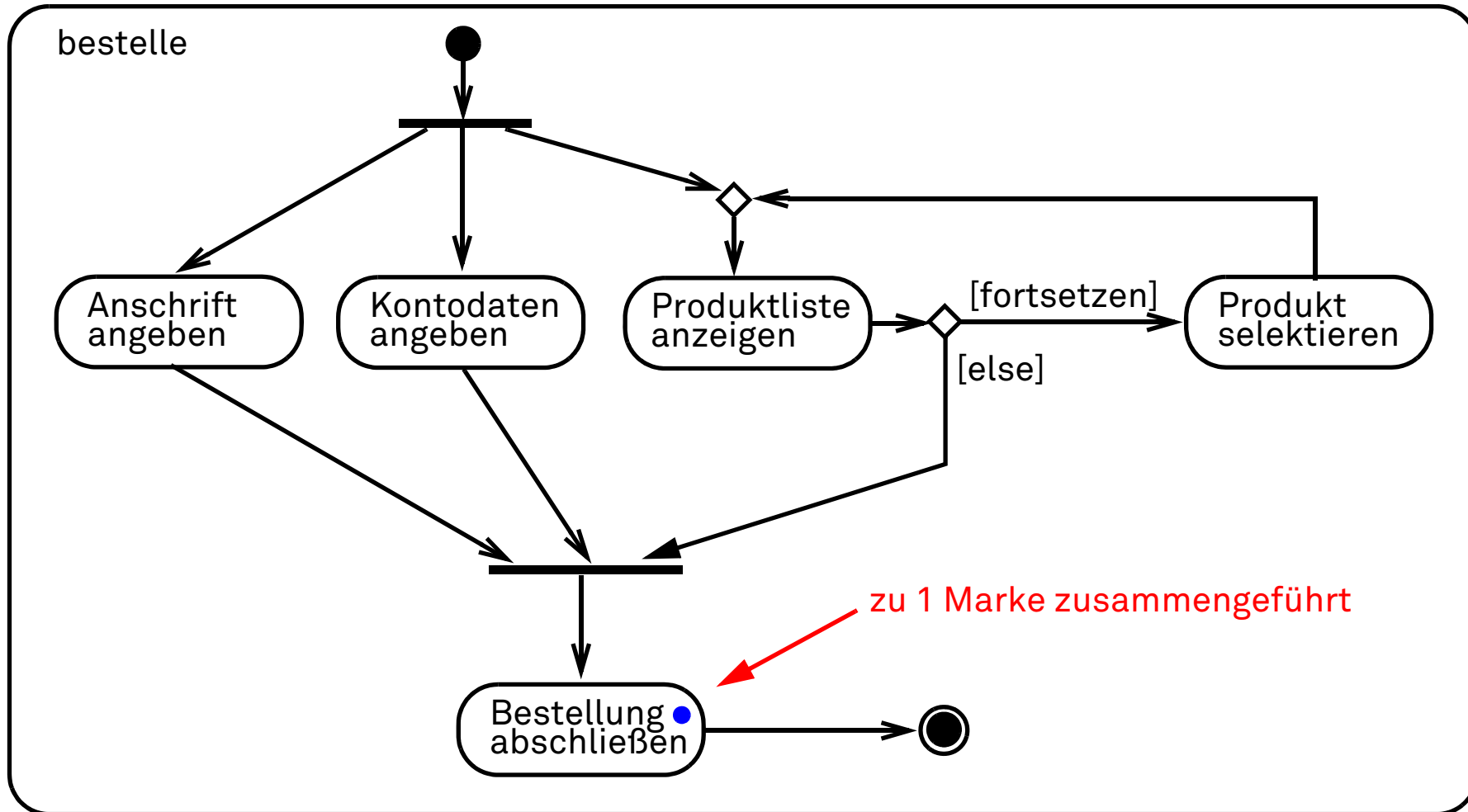
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



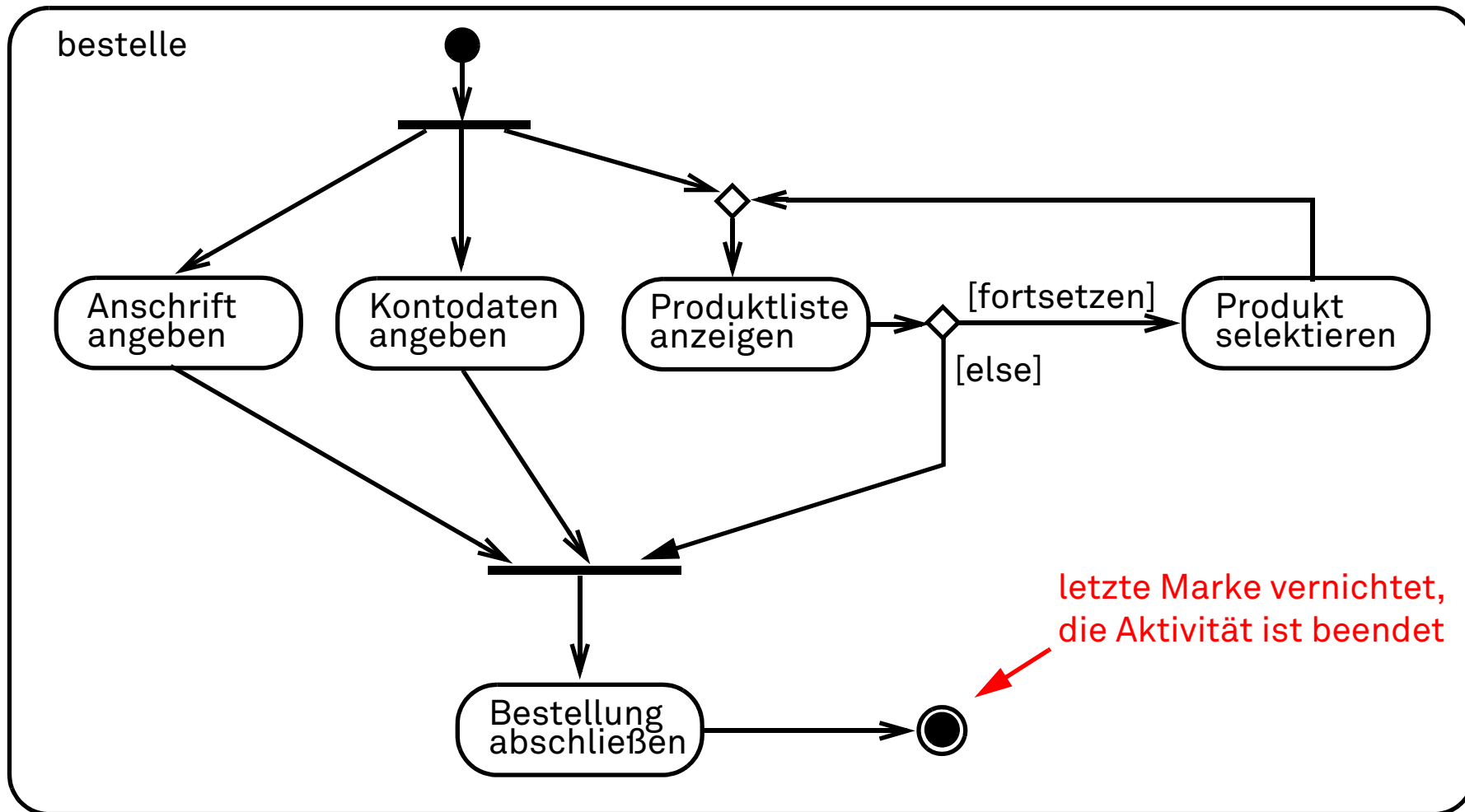
Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)

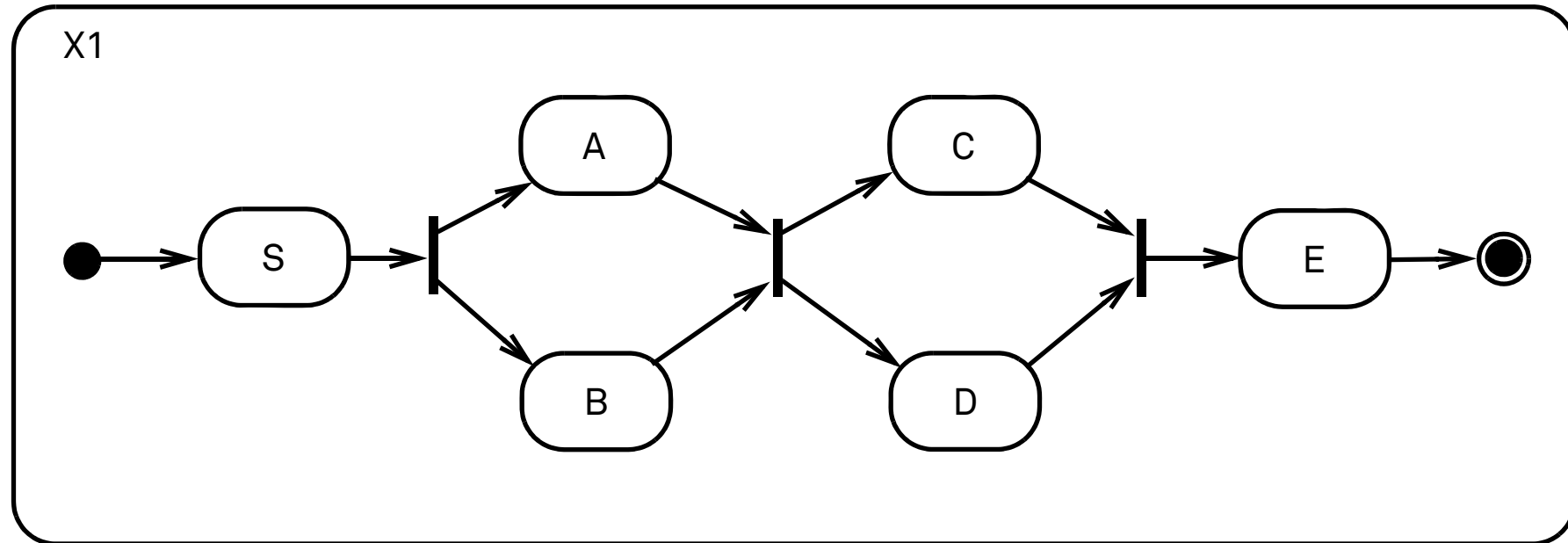


Beispiel – Nebenläufigkeit in Aktivitätsdiagrammen

(Fortsetzung)



Analyse von Aktivitätsdiagrammen – Beispiele

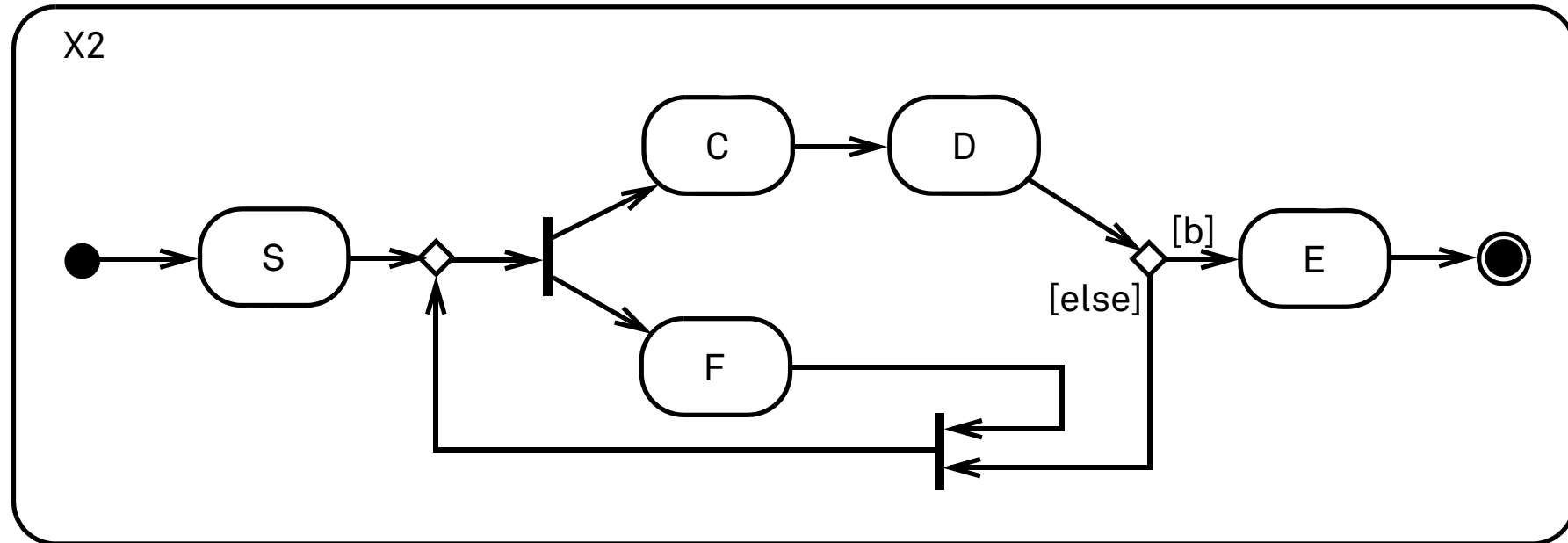


Aussagen, die sich aus diesem Diagramm ableiten lassen:

- ❑ Die Aktion S wird immer **vor** allen anderen Aktionen ausgeführt.
- ❑ A und B sowie C und D starten immer gleichzeitig.
- ❑ Die Aktionen A und B werden **immer vor** C und D ausgeführt und abgeschlossen.
- ❑ Die Aktion E wird erst nach Abschluss von S, A, B, C **und** D ausgeführt.
- ❑ Bei den Paaren A, B und C, D bestimmt jeweils die länger dauernde Aktion den Fortschritt des Kontrollflusses am folgenden Synchronisationsknoten.

Analyse von Aktivitätsdiagrammen – Beispiele

(Fortsetzung)

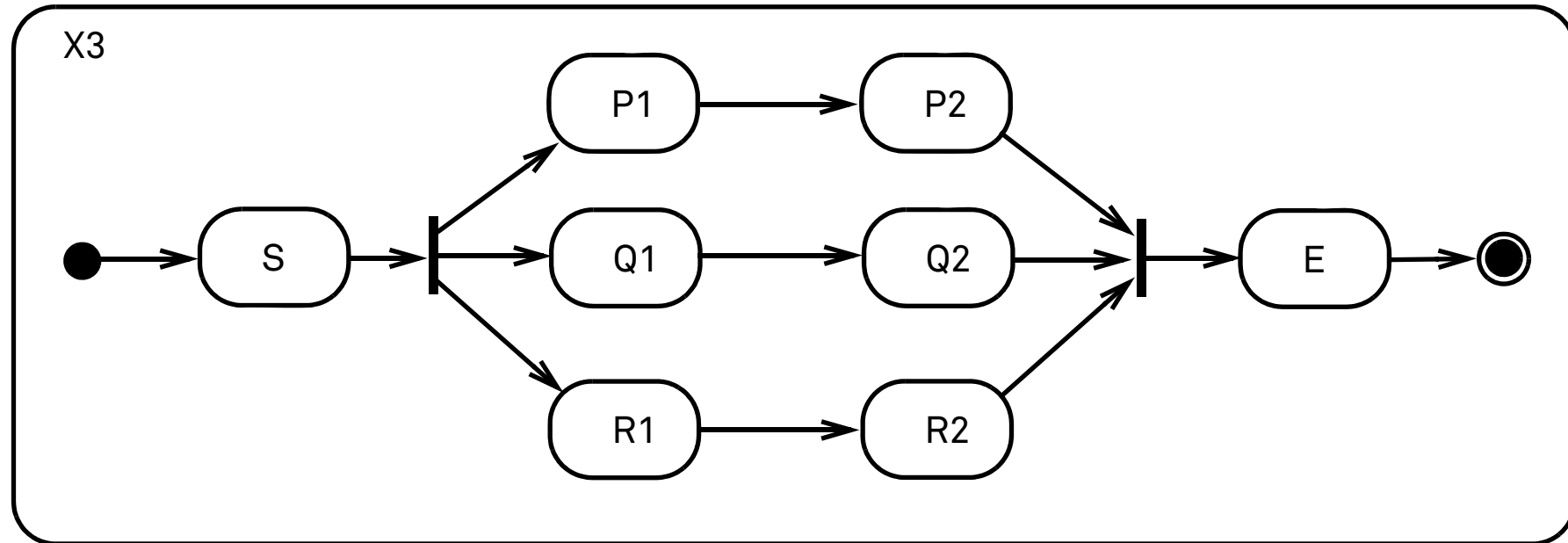


Aussagen:

- ❑ C und F starten gleichzeitig.
- ❑ Die Ausführung von Aktion C wird immer **vor** D abgeschlossen.
- ❑ Wenn nach Abschluss von D die Bedingung [b] **falsch** ist **und** F ausgeführt worden ist, wird der Durchlauf hinter S fortgesetzt.
- ❑ F kann vor oder gleichzeitig mit C ausgeführt werden und eventuell erst nach D enden.
- ❑ Ist [b] **wahr**, so wird die Aktivität über E beendet:
die in F oder hinter F verbliebene zweite Marke wird dann gelöscht.

Analyse von Aktivitätsdiagrammen – Beispiele

(Fortsetzung)



Aussagen:

- ❑ Die Aktion S wird **vor** allen anderen Aktionen ausgeführt.
- ❑ Die Aktionen P1, Q1, R1 starten zur **gleichen** Zeit.
- ❑ P1 wird **vor** dem Starten von P2 beendet, Q1 **vor** Q2, R1 **vor** R2.
- ❑ Die Aktion E wird **nach** Abschluss aller anderen Aktionen ausgeführt.
- ❑ Die weiteren Abläufe sind unbestimmt:
P2 kann z.B. parallel zu Q1 aber auch erst nach Q2 ausgeführt werden.

Zusammenfassung

Aktivitätsdiagramme

- ❑ dienen zur Beschreibung von Abläufen als Folgen von Aktionen,
- ❑ ermöglichen die graphische Beschreibung von nebenläufigem Verhalten.

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.

Grenzen der Visualisierung:

- ❑ Es werden i.d.R. nur Abläufe dargestellt.
- ❑ Der Umgang mit umfangreichen Diagrammen ist schwierig.
- ❑ Die Zwischenzustände der Ausführung ergeben sich aus der Belegung der Aktionen mit Marken. Die Bestimmung der Zwischenzustände benötigt eine Simulation.

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 5.5: Zusammenfassung Analyse

Ermitteln von Anforderungen (Wiederholung Folie 507)





Die durchzuführenden Tätigkeiten sind:

- ❑ Anforderungen erfassen, beschreiben und verfeinern.
- ❑ Szenarien erfassen und beschreiben.
Ein *Szenario* beschreibt ein konkretes Beispiel für die Erfüllung oder auch Nicht-Erfüllung von einer oder mehreren Anforderungen und konkretisiert diese dadurch.
- ❑ Lösungsorientierte Anforderungsaspekte erfassen und beschreiben:
 - Datenperspektive (strukturelle Beziehungen von Daten)
 - Funktionsperspektive (Manipulation von Daten durch Funktionen)
 - Verhaltensperspektive (Reaktion auf externe Signale, Zustandsänderungen)

Ermitteln von Anforderungen (Wiederholung Folie 507)

(Fortsetzung)

Die durchzuführenden Tätigkeiten sind:







- ❑ Anforderungen erfassen, beschreiben und verfeinern. 
- ❑ Szenarien erfassen und beschreiben.
Ein *Szenario* beschreibt ein konkretes Beispiel für die Erfüllung oder auch Nicht-Erfüllung von einer oder mehreren Anforderungen und konkretisiert diese dadurch. 
- ❑ Lösungsorientierte Anforderungsaspekte erfassen und beschreiben:
 - Datenperspektive (strukturelle Beziehungen von Daten)
 - Funktionsperspektive (Manipulation von Daten durch Funktionen) 
 - Verhaltensperspektive (Reaktion auf externe Signale, Zustandsänderungen) 

Anwendungsfalldiagramm
Sequenzdiagramm
Aktivitätsdiagramm

Ermitteln von Anforderungen (Wiederholung Folie 507)

(Fortsetzung)

Die durchzuführenden Tätigkeiten sind:

- ❑ Anforderungen erfassen, beschreiben und verfeinern. 
 - ❑ Szenarien erfassen und beschreiben.
Ein *Szenario* beschreibt ein konkretes Beispiel für die Erfüllung oder auch Nicht-Erfüllung von einer oder mehreren Anforderungen und konkretisiert diese dadurch. 
 - ❑ Lösungsorientierte Anforderungsaspekte erfassen und beschreiben:
 - Datenperspektive (strukturelle Beziehungen von Daten)
 - Funktionsperspektive (Manipulation von Daten durch Funktionen)
 - Verhaltensperspektive (Reaktion auf externe Signale, Zustandsänderungen)
- Anwendungsfalldiagramm**
Sequenzdiagramm
Aktivitätsdiagramm
Klassendiagramm

Klassendiagramm in der Analyse

Problembereich (auch als Domäne bezeichnet)
ist der Ausschnitt der Realität, in den das zu modellierende System eingebettet ist.

Problembereichsmodell (Domänenmodell)

- ❑ Modell der *Entitäten* (Dinge) des Problembereichs,
- ❑ die (vermutlich) als Informationen im System verwaltet werden müssen,
- ❑ zusammen mit ihren Beziehungen untereinander.

⇒ **Klassendiagramm**

Ein Klassendiagramm für die Analyse enthält

- ❑ keine Angaben zur technischen Realisierung (wie z.B. Zugriffsrechte),
- ❑ keine Angaben zu Navigationsrichtungen bei Assoziationen,
- ❑ möglicherweise keine oder nur wenige (wesentliche) Methoden.

Lastenheft/Pflichtenheft

- ❑ Lastenheft:
enthält die Gesamtheit der Forderungen an die Lieferungen und Leistungen,
die ein Auftraggeber erwartet. [DIN 69905 1997]
(Vorgabe des Auftraggebers)

- ❑ Pflichtenheft:
enthält die vom Auftragnehmer aufgrund des Lastenhefts erarbeiteten Leistungen und
beschreibt das zu erstellende System.
[DIN 69905 1997]
(Planung des Auftragnehmers)

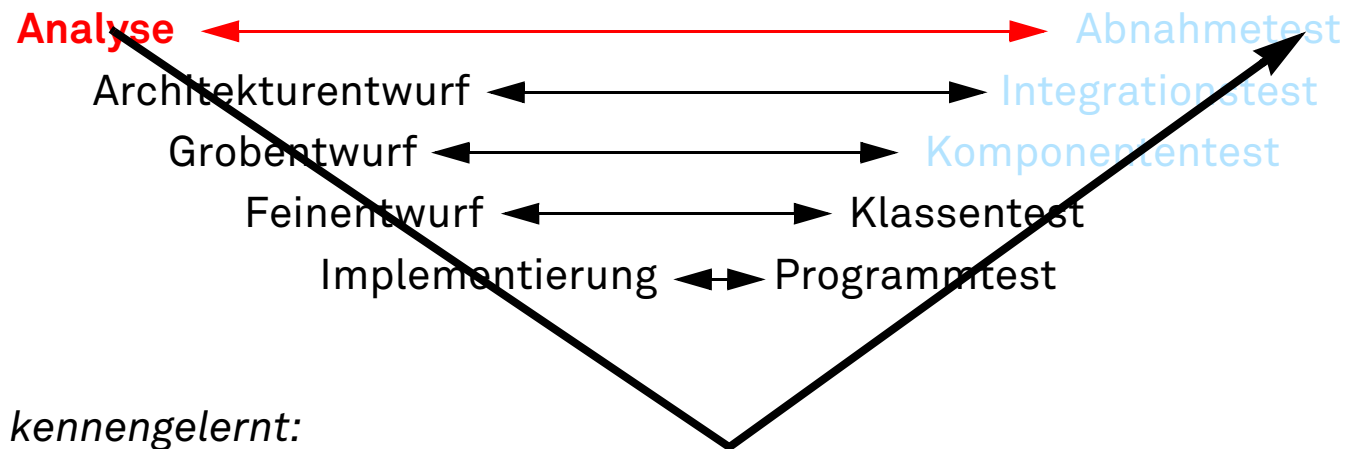
- ❑ Bedeutung des Pflichtenhefts:
 - vertragliche Basis zwischen Auftraggeber und Auftragnehmer,
 - Planungsgrundlage für den Auftragnehmer,
 - inhaltliche Vorgabe für die Arbeit des Auftragnehmers,
 - inhaltliche Vorgabe für die Prüfung des Ergebnisses der Arbeit des Auftragnehmers.

Aufbau eines Pflichtenhefts

Abschnitt	Inhalt
Erstellungsdaten	Autoren, Version, Erstellungsdatum, Änderungshistorie
Geschäftsfeld	Anwendungsbereich, Zielgruppen, Betriebsbedingungen, Modell des Problembereichs
Glossar	Erklärung fachlicher Begriffe
Zielbestimmung	Muss- und Wunschziele auflisten
Geschäftsprozesse	Beschreibung der Geschäftsprozesse durch Anwendungsfalldiagramme oder Aktivitätsdiagramme
Produktfunktionen	Beschreibung der Produktfunktionen durch Anwendungsfalldiagramme oder Aktivitätsdiagramme, Erläuterung durch Sequenzdiagramme, eventuell auch Angabe von Skizzen für Benutzungsschnittstellen
Produktdaten	Beschreibung der langfristig zu verwaltenden Daten (Klassendiagramm)
Qualität	Beschreibung der Qualitätsanforderungen
Testfälle	Testfälle zu einzelnen Funktionen und Qualitätsanforderungen
organisatorische Voraussetzungen	vorausgesetzte Hardware, vorausgesetzte Software, vorausgesetzte Abläufe

Zusammenfassung

V-Modell



als Notationen kennengelernt:

- ❑ Anwendungsfalldiagramme
- ❑ Klassendiagramme
- ❑ Objektdiagramme
- ❑ Aktivitätsdiagramme
- ❑ Sequenzdiagramme

Analyse

Analyse, Implementierung

Analyse, Implementierung

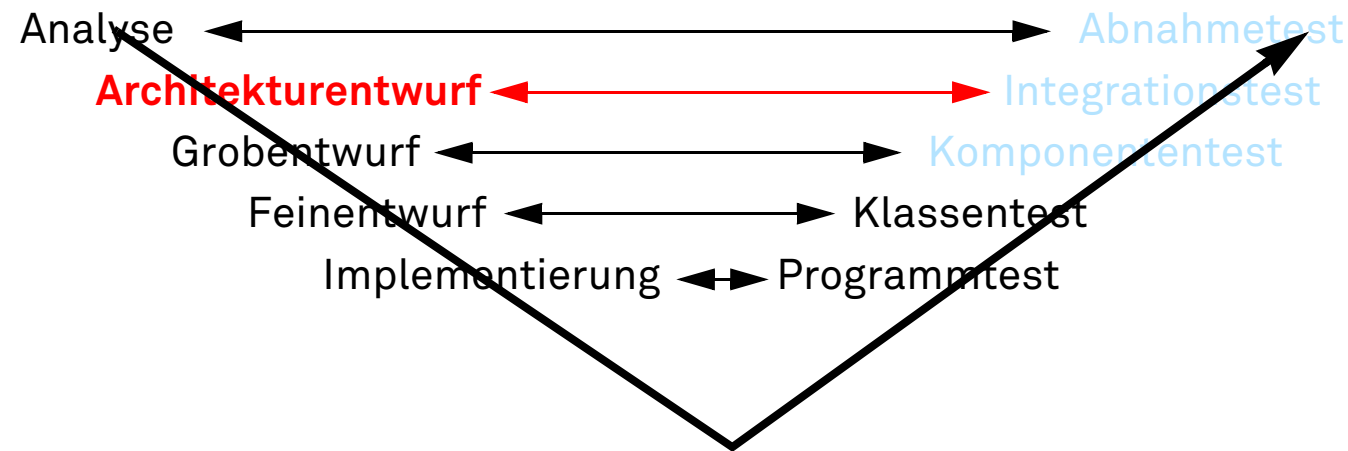
Analyse, Implementierung

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 5.6: Übergang Analyse – Architekturentwurf

Architekturentwurf

V-Modell



in der Realität gilt:

- ❑ Umfangreiche Systeme bestehen aus mehreren ausführbaren Programmen.
- ❑ Die Entwicklung verschiedener Teile der Software findet in verschiedenen (Teil-)Projekten an verschiedenen Orten und zu verschiedenen Zeiten statt.

Lösung:

Es muss eine übergreifende **Softwarearchitektur** festgelegt werden.

Softwarearchitektur

ist die strukturierte Anordnung von Komponenten
mit einer Beschreibung ihrer Beziehungen.
Dabei bilden die Komponenten eine Zerlegung des Gesamtsystems.

Das Festlegen einer Softwarearchitektur erfolgt **als frühe** technische Entscheidung
aufgrund der Anforderungen aus der Analyse:

Architekturentwurf

Anmerkungen:

- ❑ Softwarearchitektur ist ein unscharfer, weit gefasster Begriff.
- ❑ Eine Softwarearchitektur beschreibt sehr abstrakt den groben Aufbau eines Softwareprodukts.
- ❑ Die Softwarearchitektur wird meist für jedes Softwareprodukt individuell festgelegt, es gibt aber einige allgemeine Stilarten.
- ❑ Für einzelne Anwendungsbereiche gibt es vordefinierte Architekturen:
 - Framework
 - Application Server

Beispiel: Softwareunterstützung für eine (Ausleih-)Bibliothek

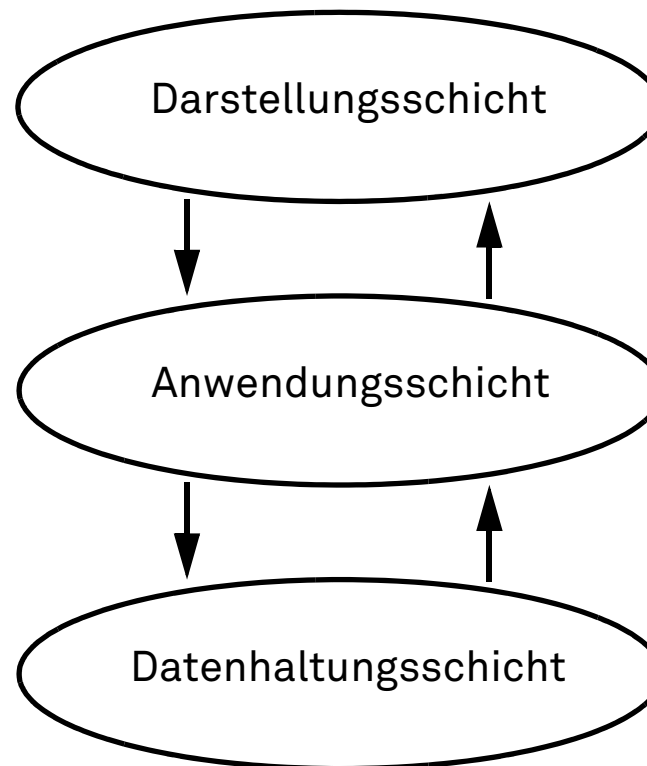
- ❑ Es werden verschiedene Datenmengen verwaltet:
 - Bücher
 - Leser
 - Buchhändler
 - Verlage
 - ❑ Es treten verschiedene Geschäftsvorfälle auf:
 - Recherche, Reservierung, Ausleihe, Rückgabe, Verlängerung, Mahnwesen, Gebührenberechnung, Beschaffung, Inventarisierung, ...
 - ❑ Es gibt verschiedene Akteure:
 - Leser an verschiedenen Orten
 - Personal mit verschiedenen Aufgaben an verschiedenen Orten
- ⇒ Es werden verschiedene, voneinander unabhängige Softwarekomponenten benötigt, die auf unterschiedlicher Hardware betrieben werden müssen.
- ❑ Es gibt einige wichtige Qualitätsanforderungen:
 - Skalierbarkeit, Zuverlässigkeit, Robustheit, Wartbarkeit

Beispiel: Softwareunterstützung für eine (Ausleih-)Bibliothek

(Fortsetzung)

mögliche Lösung:

3-Schichten-Architektur
(nimmt eine Aufteilung anhand **technischer** Aufgaben vor)

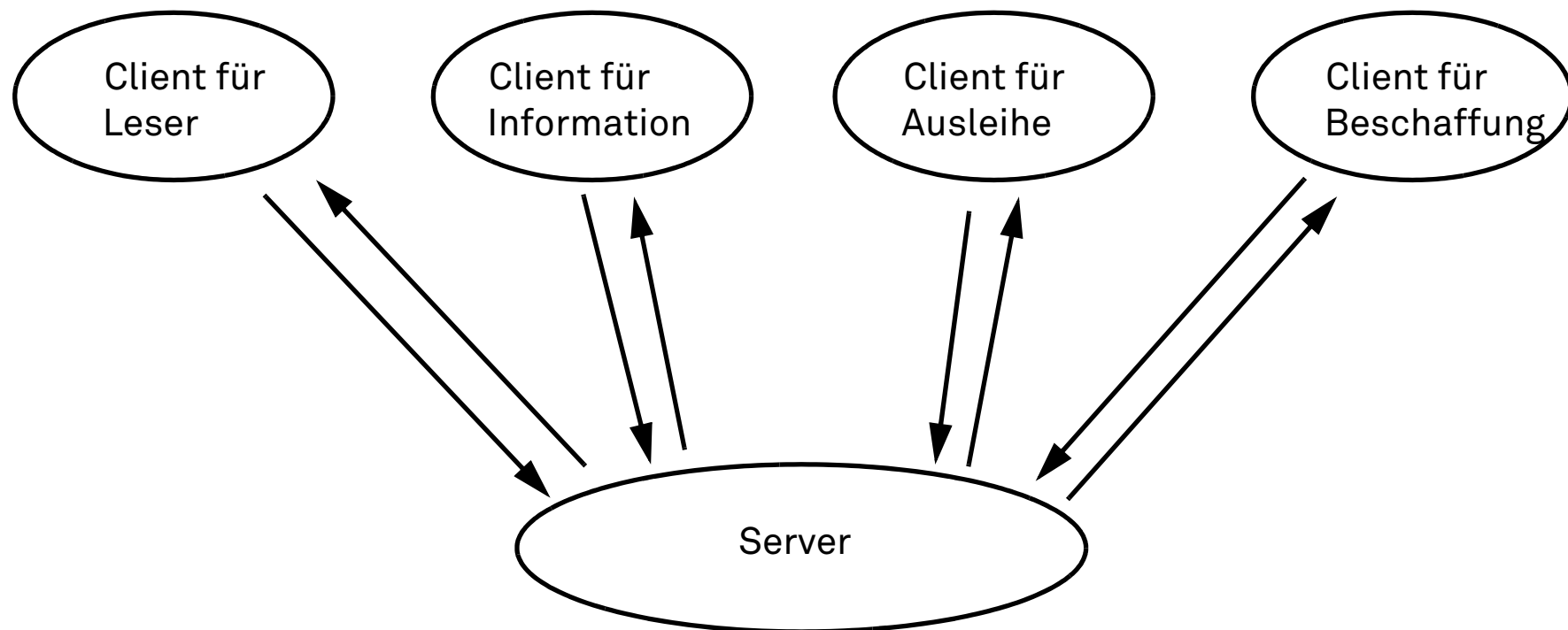


Beispiel: Softwareunterstützung für eine (Ausleih-)Bibliothek

(Fortsetzung)

andere Lösung:

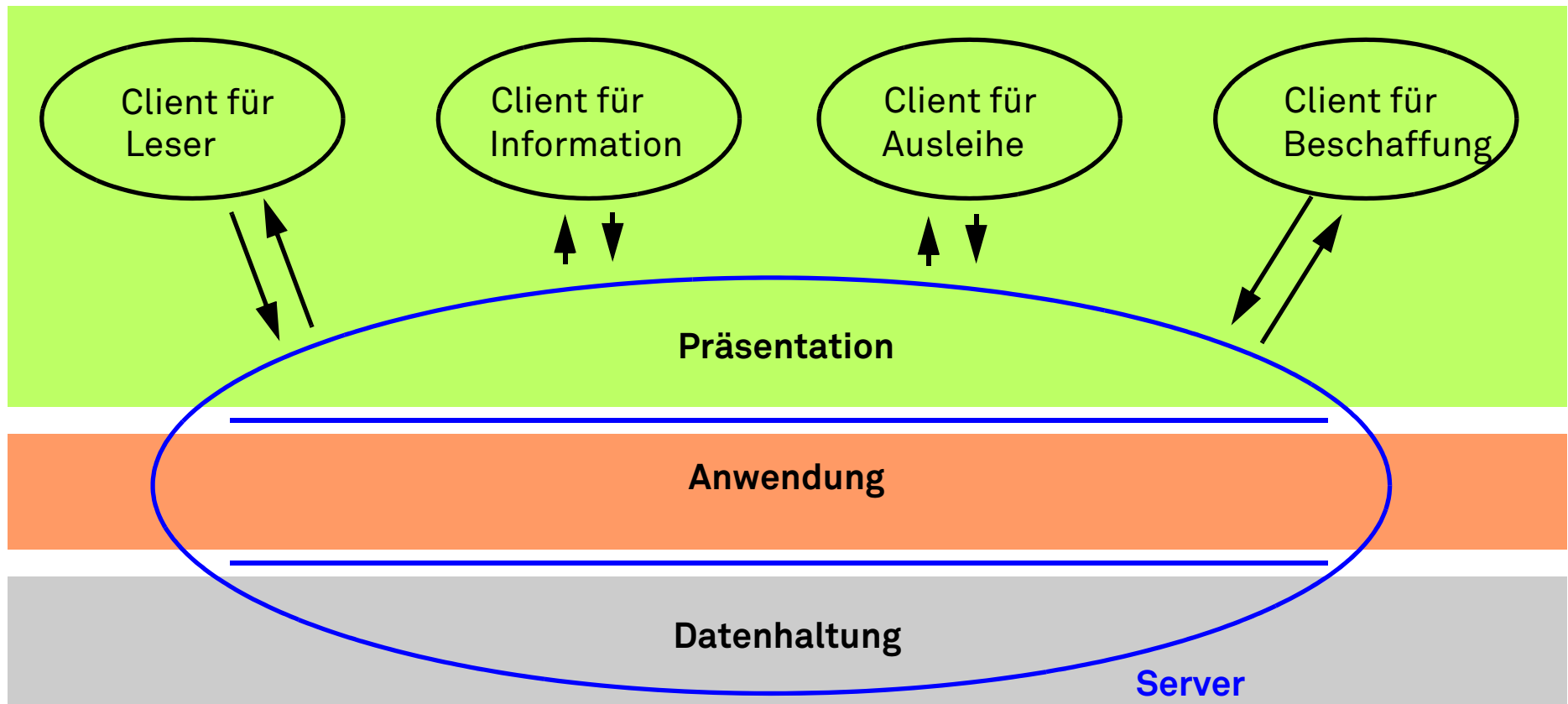
Client-Server-Architektur
(nimmt eine Aufteilung anhand *räumlicher* Aspekte vor)



Beispiel: Softwareunterstützung für eine (Ausleih-)Bibliothek

(Fortsetzung)

Kombination: 3-Schichten-Client-Server-Architektur



Beispiel: Softwareunterstützung für eine (Ausleih-)Bibliothek

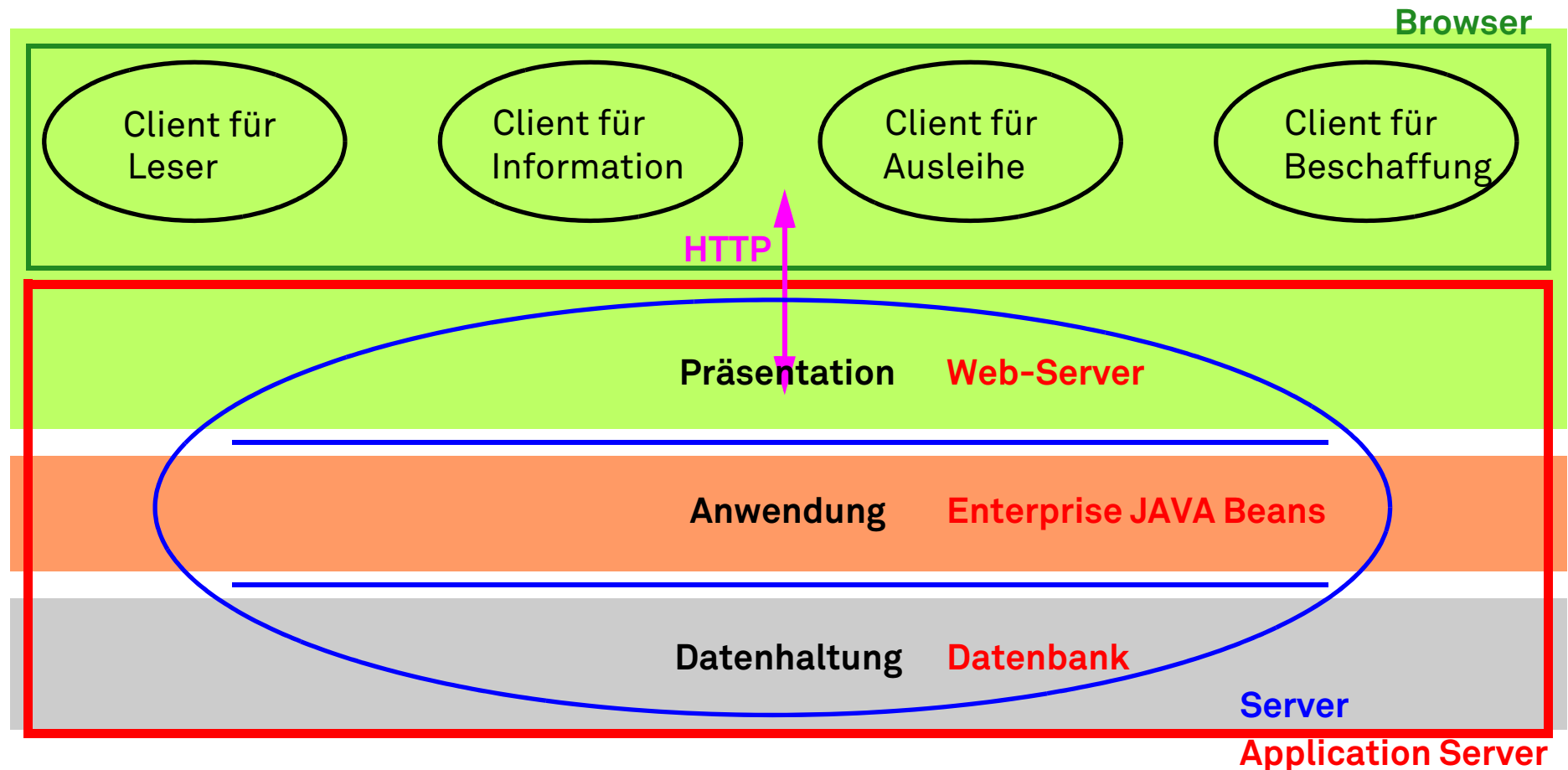
(Fortsetzung)

Kombination:

3-Schichten-Client-Server-Architektur

Realisierung:

durch den Einsatz eines **Application Servers**



Architekturstil

gibt grobe Restriktionen für die Beziehungsstrukturen zwischen den Komponenten der Architektur vor.

- ❑ Stile für die Organisation von Komponenten:
 - Schichtenarchitektur
 - Pipes-and-Filters-Architektur
 - Repository-basierte Architektur
- ❑ Stil für verteilte Systeme:
 - Client-Server-Architektur

Gemeinsamkeiten aller Architekturen sind:

- ❑ Komponenten und
- ❑ Konnektoren/Kommunikationskanäle zwischen Komponenten.

Anmerkung:

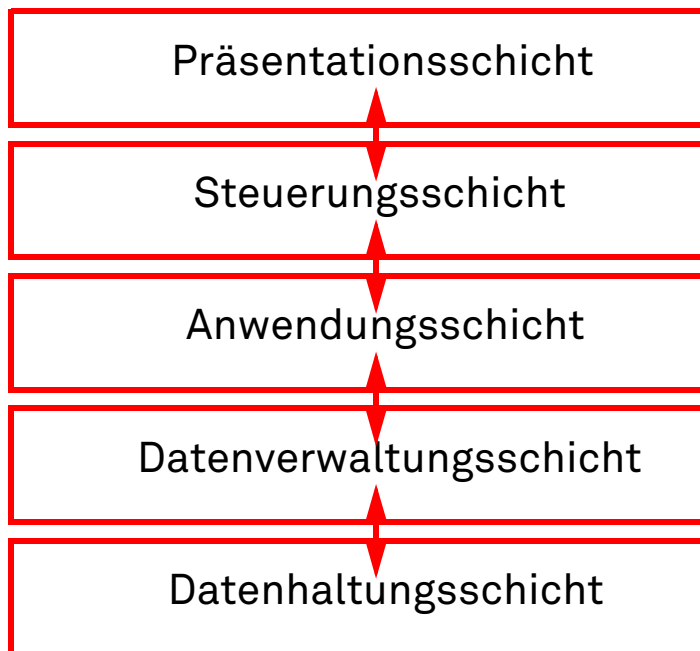
Die konkrete Architektur bestimmt die Qualität des Softwareprodukts:

- ❑ Performanz wird auch vom Kommunikationsaufwand zwischen den Komponenten beeinflusst.
- ❑ Robustheit hängt auch von den zwischen den Komponenten genutzten Protokollen ab.
- ❑ Skalierbarkeit ergibt sich aus der Möglichkeit, die Anzahl ausgewählter Komponenten anzupassen.
- ❑ Zuverlässigkeit kann durch redundante Komponenten verbessert werden.
- ❑ Wartbarkeit verbessert sich, wenn Komponenten austauschbar sind.

⇒ Der Architekturentwurf hat großen Einfluss auf das entstehende Softwaresystem und muss sorgfältig geplant werden.

Architekturstil »Schichtenarchitektur«

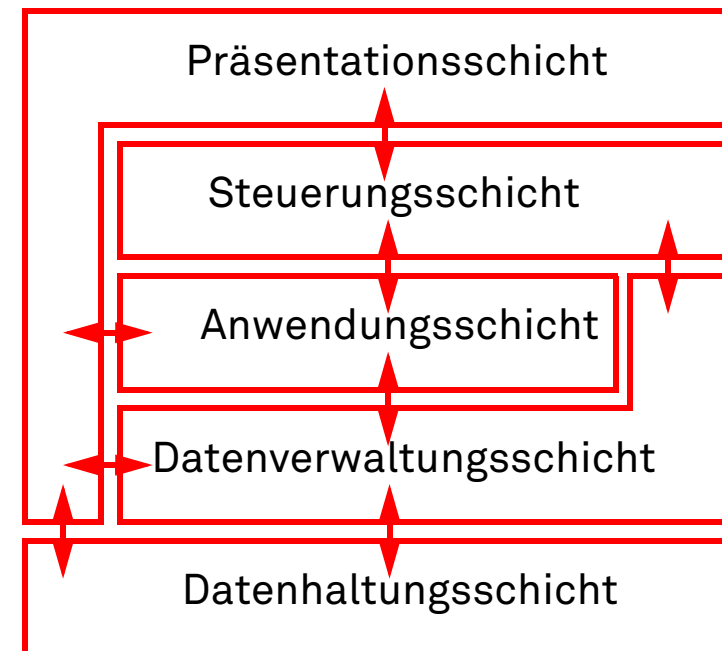
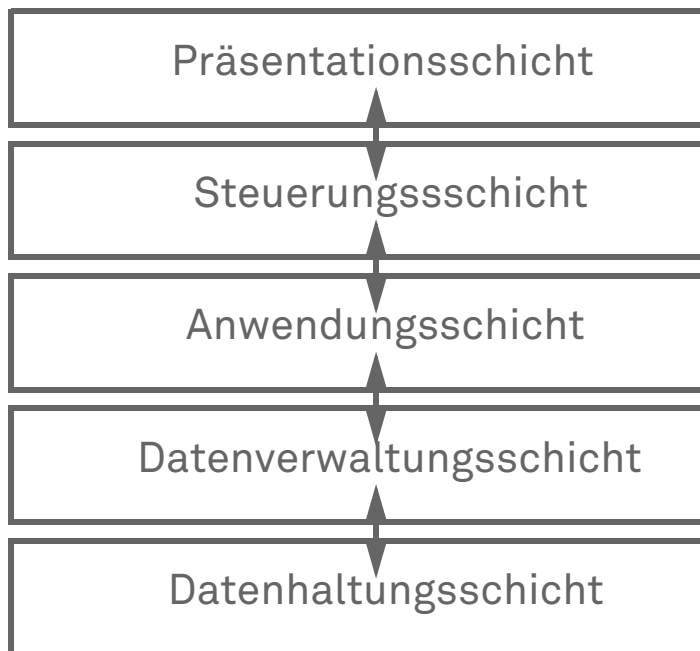
- ❑ Die Komponenten werden auf semantisch zusammenhängende Schichten verteilt. (Statt Schicht wird auch von *Ebene* (engl. *tier* oder *layer*) gesprochen.)
- ❑ Innerhalb einer Schicht können die Komponenten beliebig kommunizieren.
- ❑ Meist ist nur die Kommunikation zwischen benachbarten Schichten erlaubt.



Architekturstil »Schichtenarchitektur«

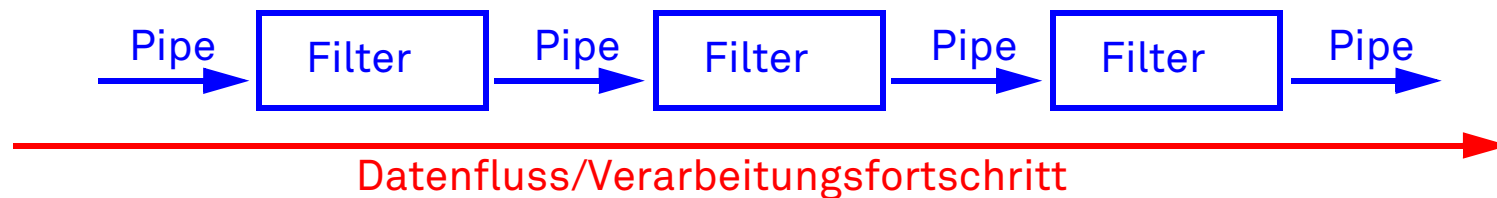
(Fortsetzung)

- ❑ Die Komponenten werden auf semantisch zusammenhängende Schichten verteilt. (Statt Schicht wird auch von *Ebene* (engl. *tier* oder *layer*) gesprochen.)
- ❑ Innerhalb einer Schicht können die Komponenten beliebig kommunizieren.
- ❑ Meist ist nur die Kommunikation zwischen benachbarten Schichten erlaubt.
- ❑ Bei vielschichtigen Architekturen lässt sich diese strikte Begrenzung der Kommunikation nicht immer durchhalten (z.B. beim Auftreten von Performanzproblemen).



Architekturstil »Pipes-and-Filters-Architektur«

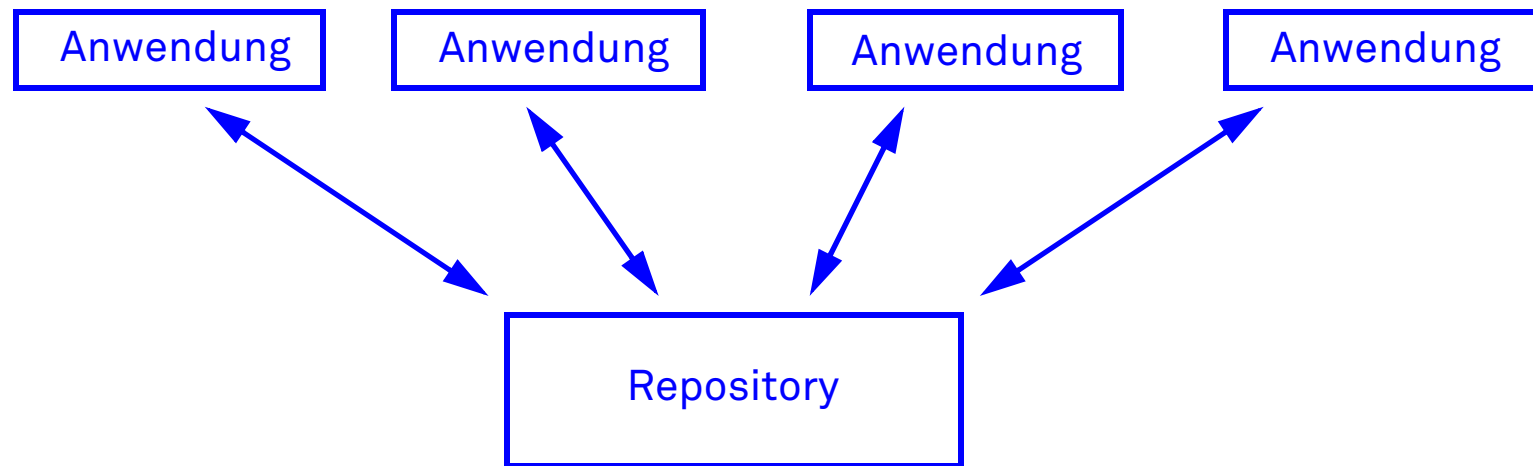
- ❑ Filter (= Komponente)
 - erfüllt einen Verarbeitungsschritt: Eingabe wird in Ausgabe umgewandelt,
 - kennt weder Vorgänger noch Nachfolger,
 - kooperiert dementsprechend nicht mit den benachbarten Komponenten.
- ❑ Pipe (= Konnektor)
 - verbindet zwei Filter, so dass nur ein sequentieller Ablauf möglich ist.



- ❑ Vorteile: Das System lässt sich leicht anpassen, ergänzen oder umkonfigurieren.
- ❑ Variationen:
 - ohne oder mit Verzweigungen,
 - ohne oder mit Zyklen (wiederholtes Durchlaufen des gleichen Filters),
- ❑ Hinweis: Pipes-and-Filters ist der Architekturstil in der *funktionalen Programmierung*.

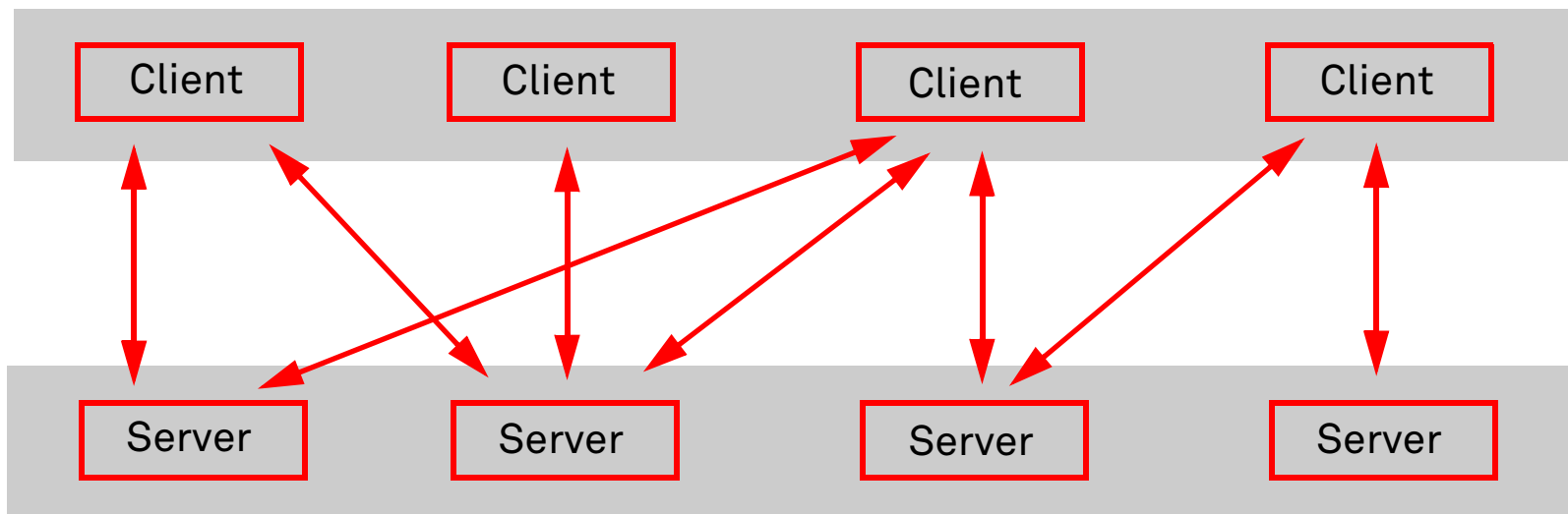
Architekturstil »Repository-basierte Architektur«

- ❑ Es werden 2 Arten von Komponenten unterschieden:
 - zentraler Datenspeicher (Repository) und die
 - auf dem Datenspeicher operierenden Anwendungen.
- ❑ Konnektoren existieren nur zwischen Datenspeicher und jeder Anwendung.
- ❑ Variation: Die Anwendungen werden in einer vorgegebenen Reihenfolge ausgeführt.



Architekturstil »Client-Server-Architektur«

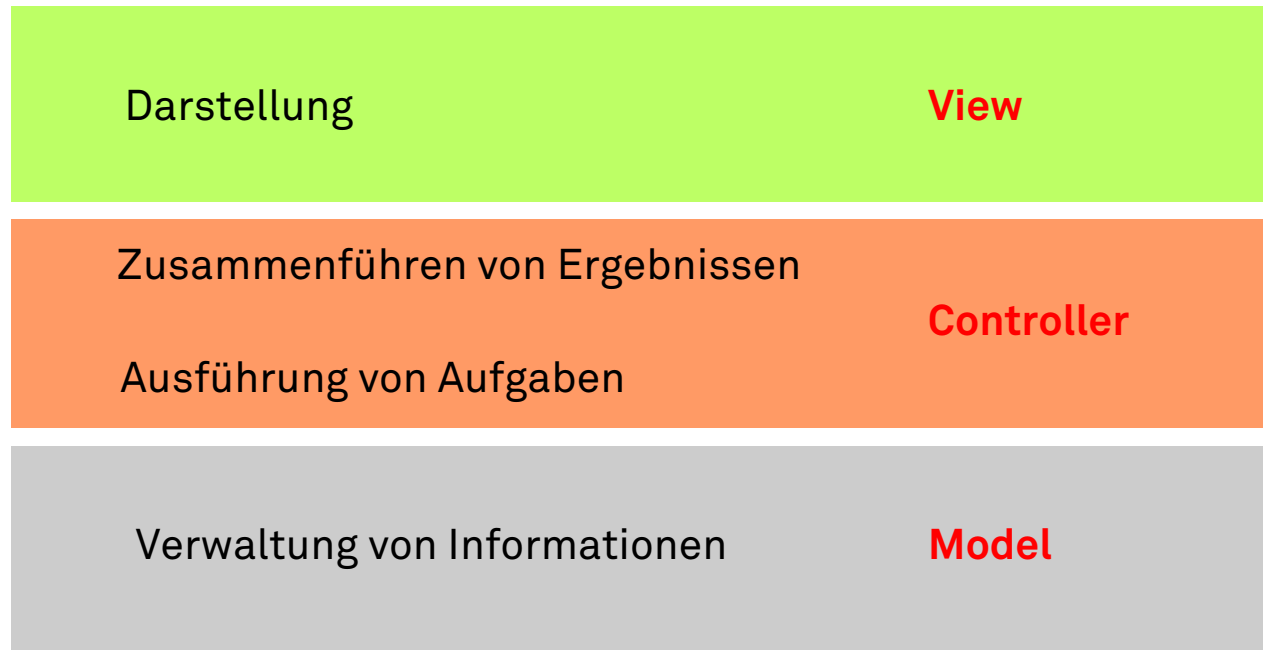
- ❑ spezielle 2-Schichten-Architektur mit
 - Server-Schicht (mit möglicherweise mehreren Servern)
 - Client-Schicht (mit möglicherweise mehreren Clients)
- ❑ Konnektoren existieren nur zwischen Server und Client.
- ❑ Kommunikationsprinzip: Client fordert eine Leistung des Servers über ein Netzwerk an.
- ❑ Variation: Komponenten können gleichzeitig die Rollen *Server* und *Client* übernehmen.



Model – View – Controller (Wiederholung Folie 20)

Beispiel:

Umsetzung einer einfachen Mehrschichtarchitektur: MVC
Model – View – Controller



Komponentendiagramm

Softwarearchitekturen können in UML durch
Komponentendiagramme veranschaulicht werden:

- ❑ Komponente
= konzeptionelle Einheit eines Softwaresystems, deren Bestandteile gekapselt sind und die Funktionalität über Schnittstellen zur Verfügung stellt.
- ❑ Der Begriff der Komponente ist nur unscharf definiert.
- ❑ Meistens wird unter einer Komponente eine Menge von Klassen verstanden.
- ❑ Eine Hierarchisierung von Komponenten ist möglich.

Komponentendiagramme werden hier nicht eingeführt.

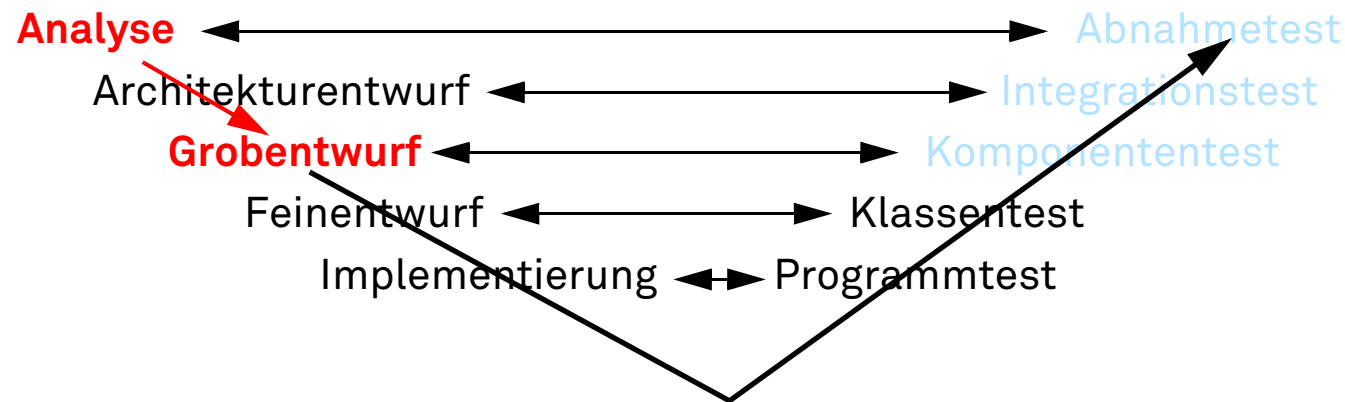
Literatur: Seemann, Jochen; von Gudenberg, Jürgen: Software-Entwurf mit UML 2 – Objektorientierte Modellierung mit Beispielen in Java, S. 121-144
http://link.springer.com/chapter/10.1007/3-540-30950-0_8

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 5.7: Übergang Analyse – Grobentwurf

Vorgehensmodell

V-Modell



- ❑ Noch offen ist eine genaue Beschreibung des Prozesses, der **von** den Ergebnissen der Analyse **zum** GrobEntwurf führt.
- ❑ In diesem Prozess müssen die Anforderungen aus Benutzer/Betreiber-Sicht in ein technisches Lösungskonzept überführt werden.

Ergebnisse der Analyse

Pflichtenheft:

- ❑ Anwendungsfalldiagramme mit
 - textuellen, tabellarischen Beschreibungen
 - Sequenzdiagrammen
 - Aktivitätsdiagrammen
- ❑ Problembereichsmodellierung mit
 - Klassendiagrammen
- ❑ Skizzen von Benutzungsschnittstellen
 - zugehörige Abläufe eventuell als Aktivitätsdiagramme

Alle Diagramme modellieren die Anforderungen an das zu erstellende System aus der **Sicht von Benutzern und Betreibern**.

Ergebnisse des Grobentwurfs

- ❑ Klassendiagramme mit
 - Attributen
 - Methoden
- ❑ Sequenzdiagramme (für Folgen von Aufrufen von Methoden)
- ❑ Aktivitätsdiagramme (für Abläufe in Methoden)

Alle Diagramme modellieren Eigenschaften des zu erstellenden Systems aus einer **technischen Sicht**.

Ergebnisse des Grobentwurfs

(Fortsetzung)

- ❑ Klassendiagramme mit
 - Attributen
 - Methoden
- ❑ Sequenzdiagramme (für Folgen von Aufrufen von Methoden)
- ❑ Aktivitätsdiagramme (für Abläufe in Methoden)

Alle Diagramme modellieren Eigenschaften des zu erstellenden Systems aus einer **technischen Sicht**.

⇒ **Erkenntnis:**

In Analyse und Entwurf
werden die gleichen Diagrammtypen
mit unterschiedlichen Zielsetzungen verwendet!

⇒ **Folgerung**

Die verschiedenen Diagrammtypen lassen sich universell einsetzen!

Problembereichsanalyse

Vorgehensweise bei der Erstellung des Problembereichsmodell in der Analyse:

- ❑ Beim Herausarbeiten von Anwendungsfällen werden Entitäten ermittelt:
 - Objekte, Attribute, Klassen, Beziehungen.
- ❑ Anschließend werden
 - gleiche Objekte zusammengefasst und auf einen Platzhalter reduziert,
 - Objekte zu Klassen verallgemeinert,
 - die Zuordnung von Attributen zu Klassen vorgenommen,
 - Klassen mit gleichen Aufgaben zusammengefasst,
 - ähnliche Klassen in Hierarchien angeordnet und
 - die Beziehungen zwischen den Klassen festlegt.
- ❑ Abschließend
 - wird das Modell überprüft
 - und der Vorgang möglicherweise wiederholt.

Problembereichsanalyse

(Fortsetzung)

Regeln für das Problembereichsmodell:

- ❑ Das Bereichsmodell beschreibt die Realität der Problemwelt.
- ❑ Das Bereichsmodell zeigt **keine** technische Lösungen.
- ❑ Jedes Element des Bereichsmodells bezieht sich auf ein Ding (eine Entität) aus der Problemwelt.
- ❑ Auf jede Klasse muss in mindestens einem Anwendungsfall Bezug genommen werden.
- ❑ Jede Klasse sollte mindestens ein Attribut oder mehr als eine Assoziation besitzen.
- ❑ Im Normalfall sollten bei der späteren Ausführung des Systems von jeder Klasse mehrere Objekte erzeugt werden.

Übergang: Analyse → Grobentwurf

Ausgangspunkt ist das Problembereichsmodell (Klassendiagramm) aus der Analyse.

- ❑ Klassen aus dem Problembereichsmodell sollen möglichst erhalten bleiben, um einen Bezug zwischen Analyse und Entwurf herzustellen und die Verständlichkeit zu erhöhen.
- ❑ Aus technischer Sicht notwendige Klassen werden hinzugefügt.
- ❑ Aus technischer Sicht notwendige Methoden werden hinzugefügt.
- ❑ Für die Ausführung notwendige Beziehungen werden hinzugefügt.
- ❑ Das so entstandene Modell wird anschließend verbessert.

Ziel des Vorgehens ist das Erstellen des technisch orientierten Klassendiagramms für den Entwurf.

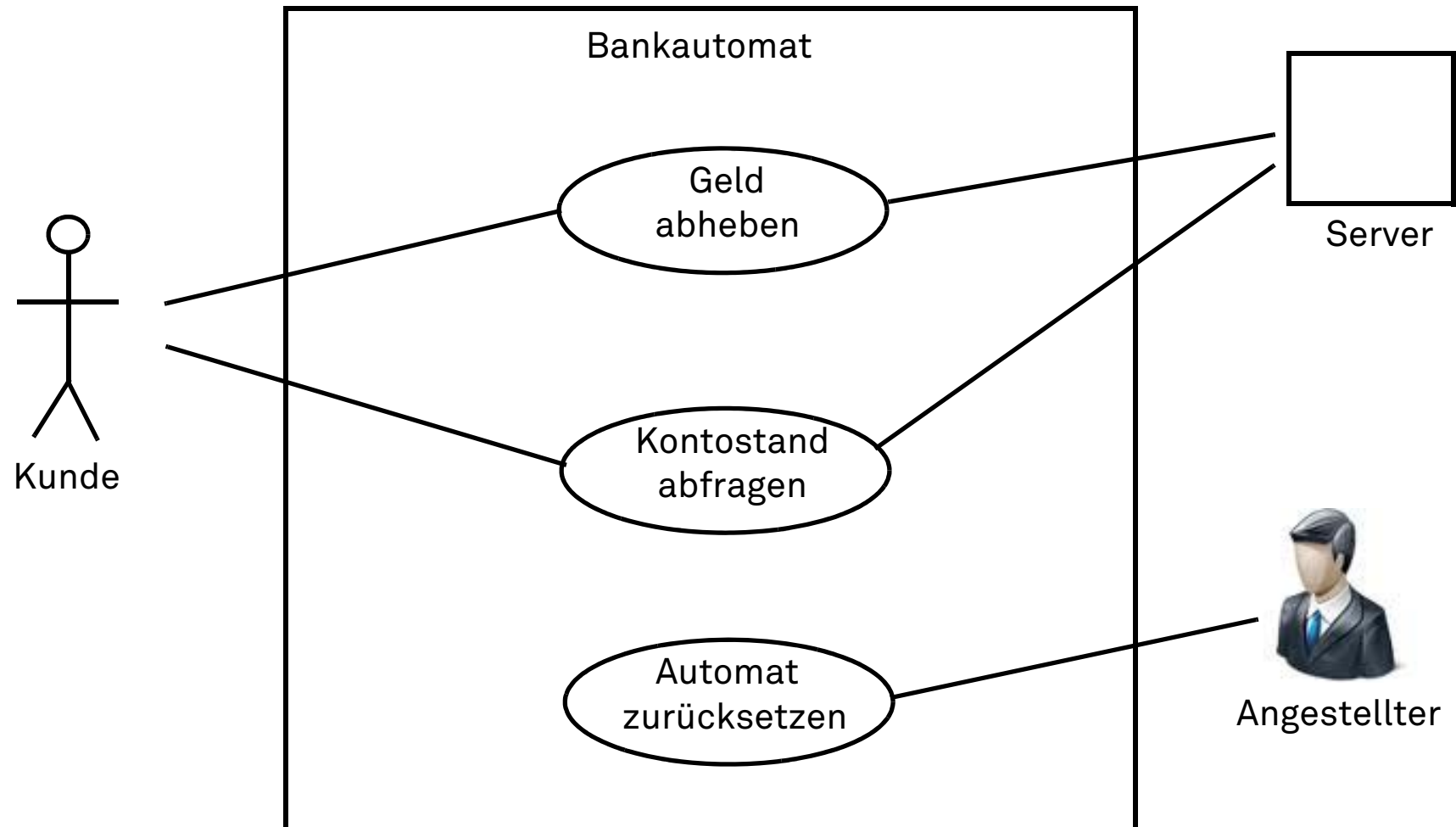
Übergang: Analyse → Grobentwurf

(Fortsetzung)

Konstruktion der Klassenstruktur des Entwurfs:

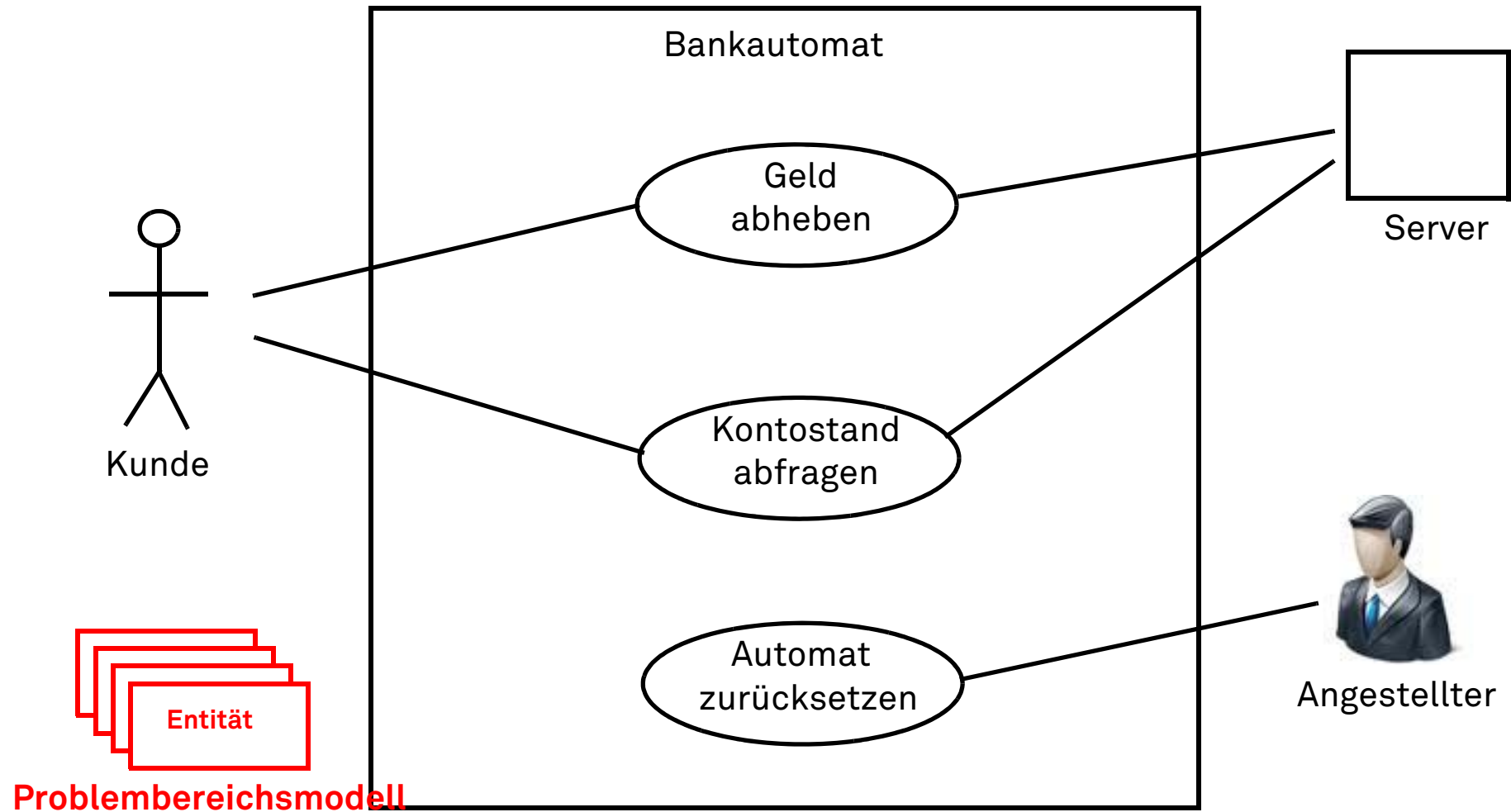
- ❑ Ausgangspunkt Problembereichsmodell:
Die Klassen des Problembereichsmodells werden zu **Entitätsklassen** (und dienen der Datenhaltung).
- ❑ Ausgangspunkt iAnwendungsfallmodell:
 - Zu jedem Anwendungsfall wird eine **Steuerungsklasse** geschaffen, die mindestens eine Methode zur Steuerung des Ablaufs des Anwendungsfalls enthält.
 - Zu jeder Assoziation mit einem Akteur wird eine **Schnittstellenklasse** geschaffen, die die Kommunikation mit dem Akteur abwickelt.

Übergang: Analyse → Grobentwurf (Beispiel)



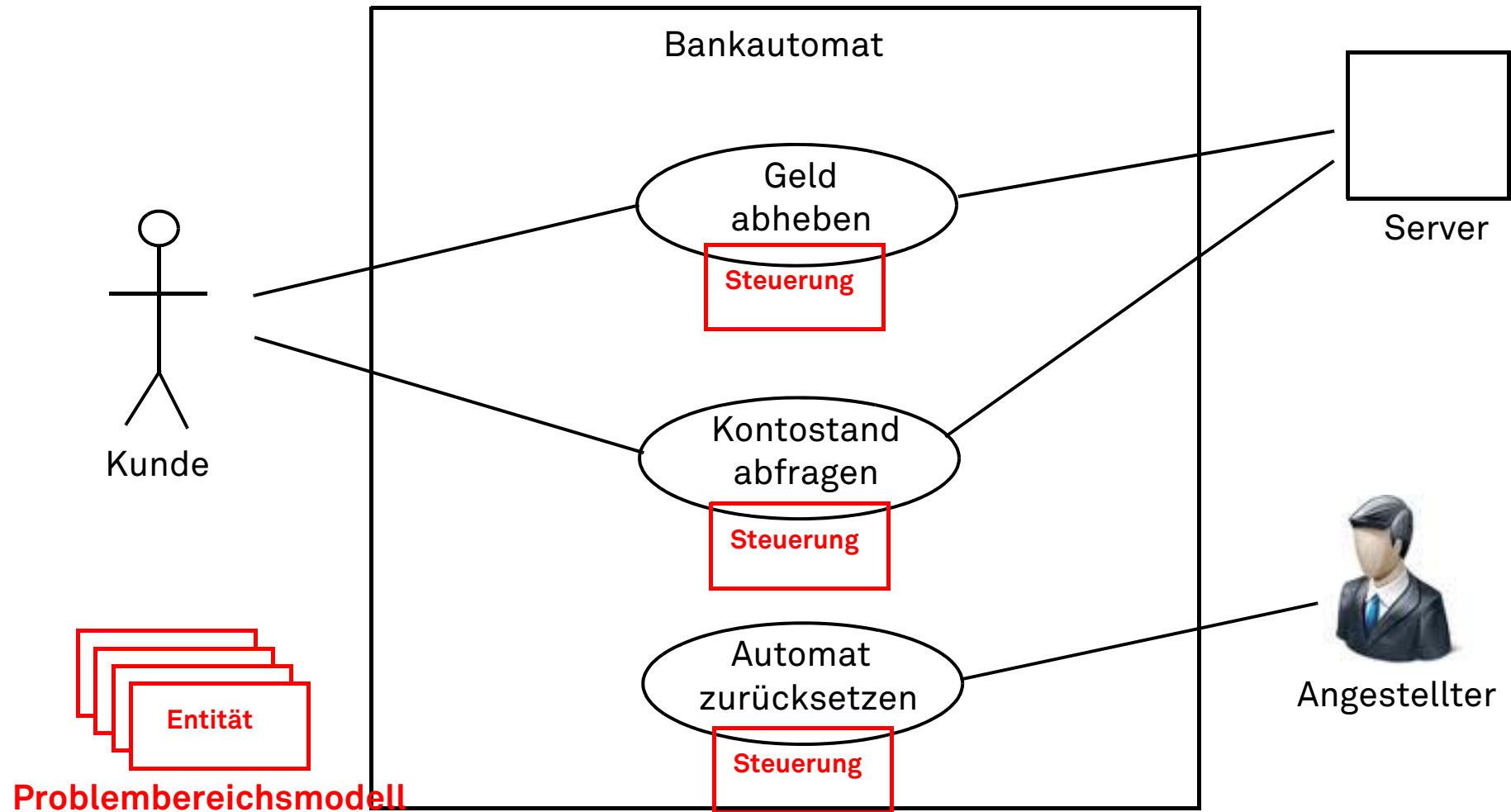
Übergang: Analyse → Grobentwurf (Beispiel)

(Fortsetzung)



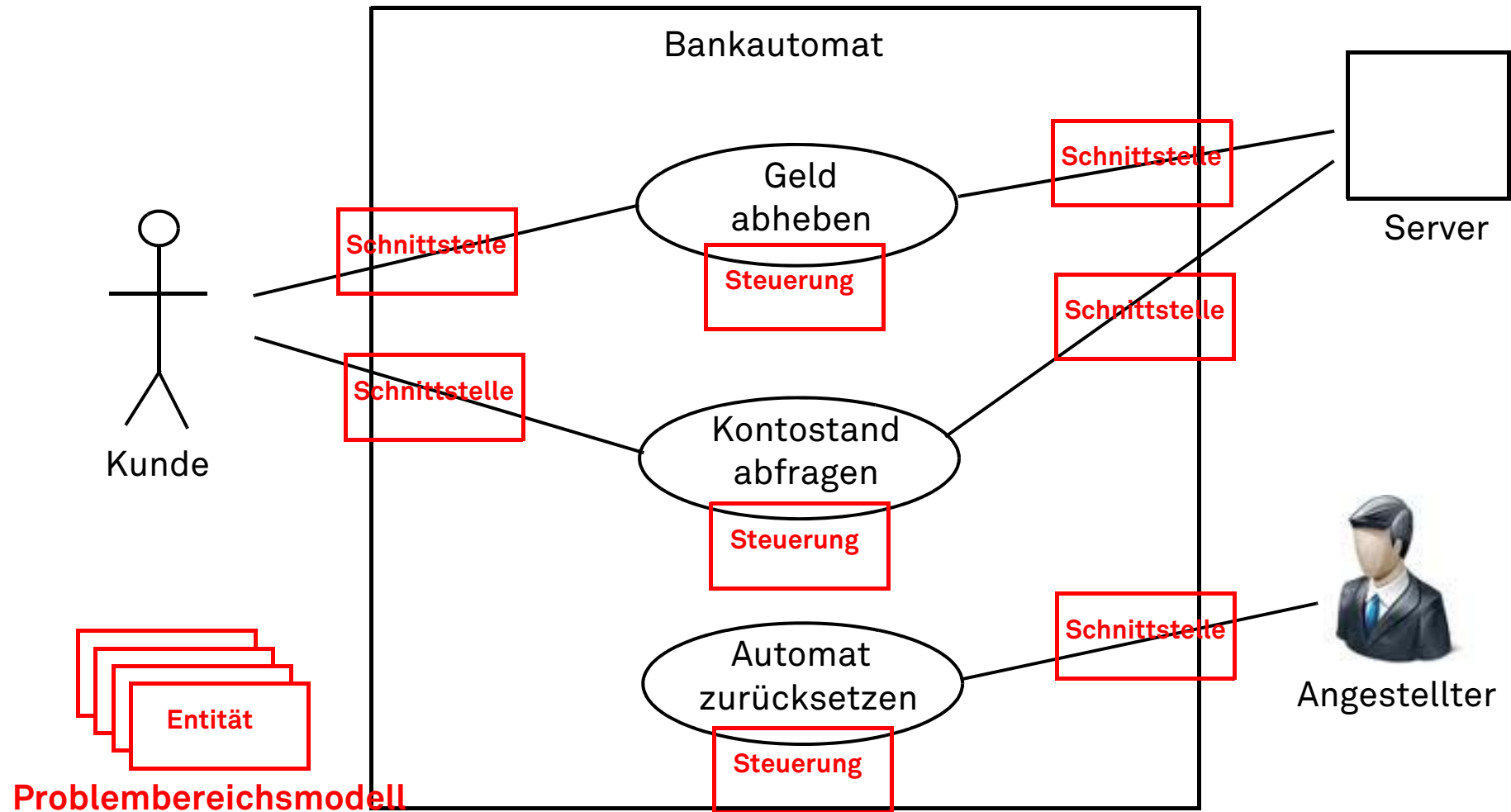
Übergang: Analyse → Grobentwurf (Beispiel)

(Fortsetzung)



Übergang: Analyse → Grobentwurf (Beispiel)

(Fortsetzung)



Übergang: Analyse → Grobentwurf (Beispiel)

(Fortsetzung)

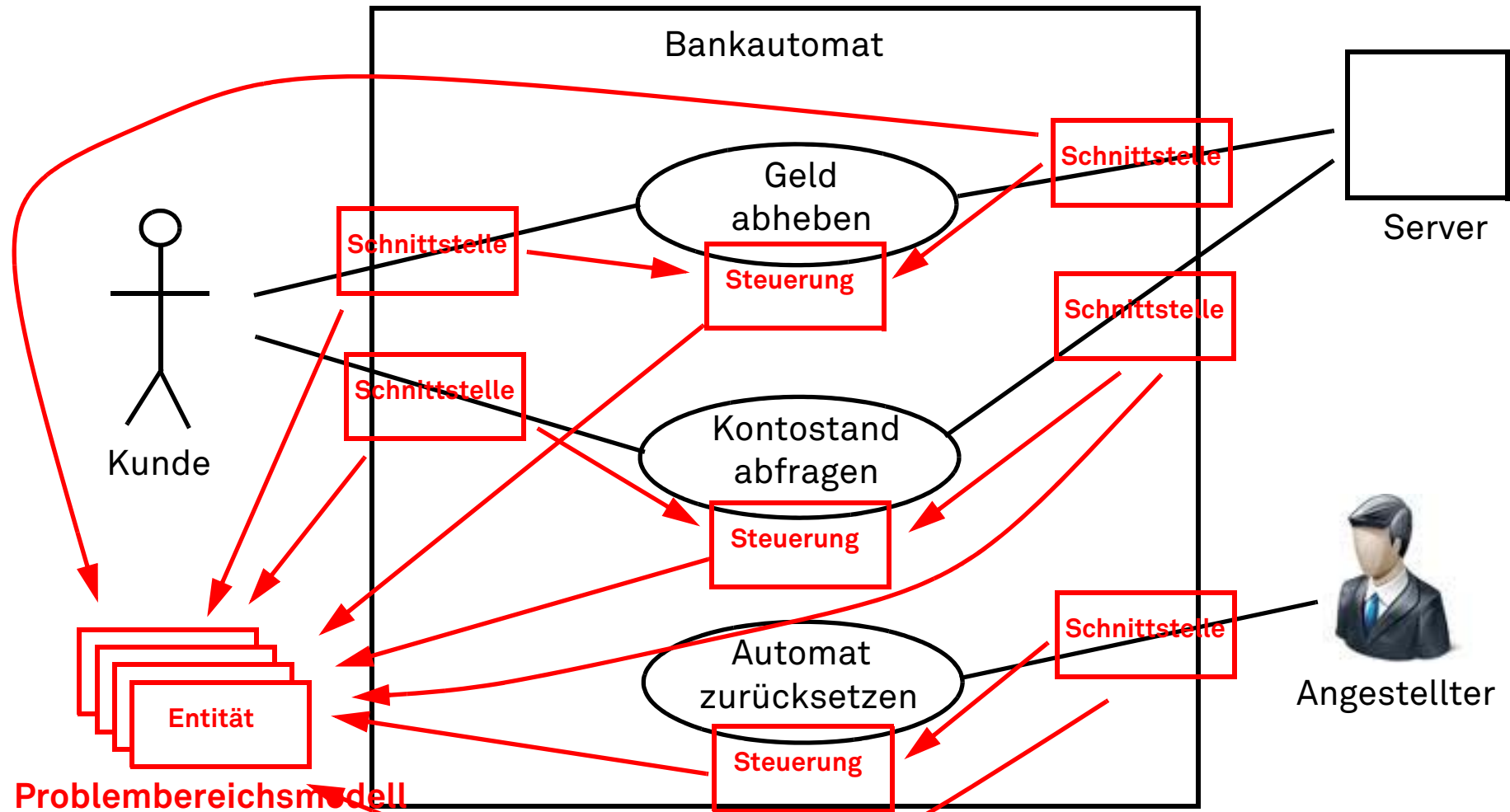
Konstruktion der Klassenstruktur des Entwurfs:

- ❑ **Entitätsklassen**
enthalten in der Regel die Daten, die in dem System benötigt werden.
- ❑ **Steuerungsklassen**
nutzen oder manipulieren die in den Entitätsklassen verwalteten Daten.
- ❑ **Schnittstellenklassen**
 - nutzen die Steuerungsklassen zum Anbieten von Funktionalität für einen Akteur,
 - nutzen Entitätsklassen zum Abrufen von Daten und zur Speicherung von Eingaben.

Dieser Aufgabenzuteilung liegt die Idee der MVC-Architektur zugrunde.

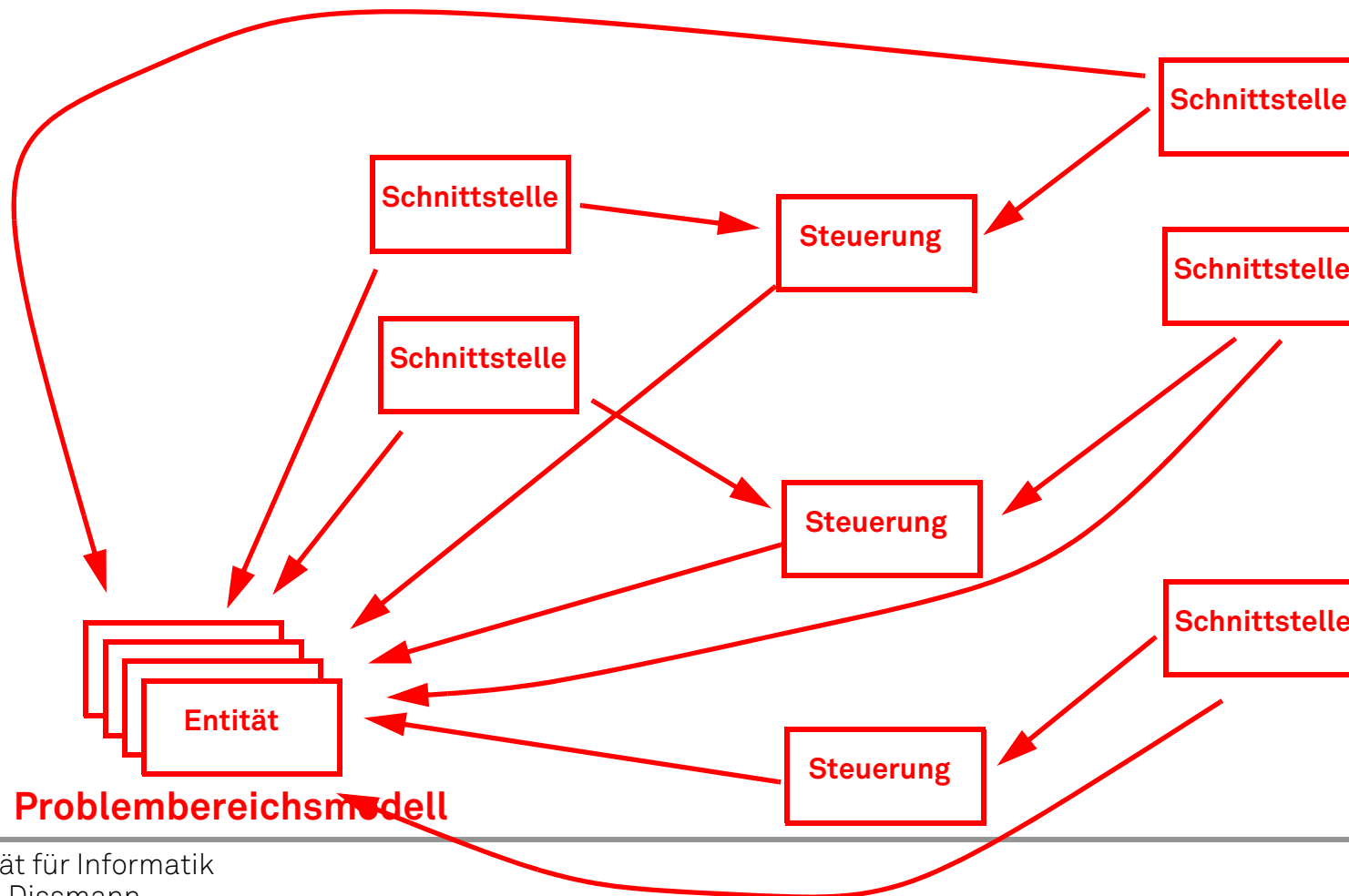
Übergang: Analyse → Grobentwurf (Beispiel) (Beispiel)

(Fortsetzung)



Übergang: Analyse → Grobentwurf (Beispiel) (Beispiel)

(Fortsetzung)



Übergang: Analyse Grobentwurf (Beispiel)

(Fortsetzung)

Ziele des Entwurfs:

- ❑ Es werden die in der Problemanalyse noch fehlende Beziehungen zwischen Klassen ergänzt, die für den Ablauf notwendig sind.
- ❑ Es werden fehlende technische Aspekte ergänzt.
- ❑ Die technische Umsetzbarkeit wird vorbereitet.
- ❑ Die organisatorische Umsetzbarkeit wird vorbereitet:
Klassen werden nicht nur als technische Einheiten betrachtet,
sondern dienen bei der Realisierung auch der Zuordnung von Aufgaben an Entwickler.
- ❑ Die durch bereits realisierte Programmteile gegebenen Schnittstellen müssen berücksichtigt werden.

- ❑ Gegebenenfalls muss auch die programmtechnische Umsetzung
in einer bestimmten Programmiersprache muss vorbereitet werden.

Übergang: Analyse → Grobentwurf (Beispiel)

(Fortsetzung)

Ergänzung technischer Aspekte

- ❑ Komplexe Attribute werden zu einer entsprechenden Klasse erweitert.
Beispiele: Währungen, Adressen
- ❑ Multiplizitäten im Klassendiagramm werden durch geeignete Klassen realisiert.
Beispiele: Listen von Objekten, Mengen von Objekten, Tabellen von Objekten
- ❑ Methoden für Konstruktion und zum Abbau von Strukturen werden ergänzt.
Beispiele: Konstruktoren, Methoden zum Aufbau von Verbindungen zwischen Objekten
- ❑ Methoden werden angeglichen.
Ähnliche Aufgaben werden durch Methoden mit ähnlichen Schnittstellen und ähnlichem Verhalten gelöst
- ❑ Die Kommunikationsabläufe werden angeglichen.
Ähnliche Kommunikationsabläufe zwischen verschiedenen Klassen werden in ähnlicher Weise organisiert. Dabei werden auch Entwurfsmuster berücksichtigt.
- ❑ Algorithmen werden von ProblemDetails abgetrennt.
Algorithmen identifizieren, generische Klassen bilden.

Übergang: Analyse → Grobentwurf (Beispiel)

(Fortsetzung)

Lebenszyklen von Objekten organisieren

- ❑ Das Erzeugen von Objekten wird organisiert.
Rahmenbedingungen für das Erzeugen und Initialisieren festlegen.
- ❑ Das Vernichten von Objekten wird organisiert.
Rahmenbedingungen und Abschlussaktionen festlegen
(– hängt auch von der verwendeten Programmiersprache ab).
- ❑ Die Zuständigkeit für das Erzeugen von Objekten wird festgelegt.
Gegebenenfalls spezielle Klassen ergänzen, die Objekte anderer Klassen erzeugen.
- ❑ Die Zuständigkeiten für die Vernichtung werden festgelegt.
Spezielle Zustände definieren, entsprechende Klassen ergänzen.

Übergang: Analyse → Grobentwurf (Beispiel)

(Fortsetzung)

Konzeption der Benutzungsschnittstelle

- ❑ Benutzungsschnittstellen sind ereignisgesteuert:
 - Die fensterbasierte Oberfläche ist objektorientiert strukturiert.
 - Der Benutzer löst ein Ereignis aus (durch Cursor-Position, Tastendruck, Mausklick ...).
 - Die Semantik des Ereignisses hängt vom betroffenen Oberflächen-Objekt ab.
 - Die aktivierten Oberflächen-Objekte rufen beim Auftreten eines Ereignisses Methoden von Objekten der Schnittstellen- oder Steuerungsklassen auf.
- ❑ Es bestehen aber vielfältige gegenseitige Abhängigkeiten:
 - Steuerungsobjekt öffnet Fenster, Ereignisse bestimmen Ablauf im Steuerungsobjekt*
 - Der Fokus (d.h. die Position des Zeigers) bestimmt den Wechsel der Kontrolle, die Auswahl des aktiven Teils der Anwendung erfolgt durch den Benutzer.*
- ❑ Sicherstellen eines einheitlichen Steuerungskonzepts.
 - Alternativen:
 - Die Steuerungsklasse kontrolliert Aufbau und Änderungen von Fensterinhalten.
 - Die Schnittstellenklasse steuert sich selbst.

Übergang: Analyse Grobentwurf (Beispiel)

(Fortsetzung)

nach der – eventuell zyklisch wiederholten –
Ausführung der vorgestellten Schritte

- ❑ sind die Anforderungen aus dem Pflichtenheft
in den Entwurf überführt worden,
- ❑ liegt ein realisierbares technisches Konzept vor,
das die Anforderungen umsetzt.

Es müssen nun noch solche Verbesserungen am Entwurf vorgenommen werden,
die die einige technischen Aspekte stärker berücksichtigen:

- ❑ Steuerung von Abläufen
- ❑ Datenhaltung
- ❑ Realisierbarkeit verbessern
- ❑ Performanz verbessern
- ❑ Implementierung vorbereiten

Ergänzungen des Grobentwurfs

Konzeption der Ablaufsteuerung

- ❑ Objektorientierte Systeme steuern sich selbst:
 - Methodenaufrufe bestimmen den Wechsel der Kontrolle zwischen den Objekten.
 - Die Steuerung des gesamten Systems ist auf **viele** Objekte verteilt.
- ❑ Eine einheitliche Strategie verbessert Verständlichkeit.
- ❑ Übergreifende Mechanismen (z.B. Fehlerbehandlung) müssen auch übergreifend geregelt werden.
- ❑ Eine klare Trennung von steuernden und von reagierenden Klassen erleichtert die Konstruktion der Abläufe und insbesondere auch das Testen.
- ❑ Nebenläufigkeit:
 - sich ausschließende Aktivitäten identifizieren
 - kritische Bereiche ermitteln
 - Synchronisationsmechanismen festlegen
- ❑ Konzeption der persistenten (= dauerhaften) Datenhaltung:
 - Festlegen von Orten der Datenaufbewahrung (Datei/Datenbank)
 - Festlegen von Zeitpunkten für die Persistierung

Ergänzungen des Grobentwurfs

(Fortsetzung)

Verbesserung der Realisierung von Klassen und deren Beziehungen

- ❑ Hinzunahme zusätzlicher Attribute/Beziehungen, um Abläufe zu vereinfachen oder beschleunigen (z.B. Pufferung).
- ❑ Hinzunahme redundanter Attribute, um die Ausführung zu beschleunigen.
- ❑ Gleiche Abläufe in verschiedenen Klassen werden mit den zugehörigen Attributen zusammengefasst und ausgelagert. Eventuell muss eine Oberklasse oder eine generische Klasse geschaffen werden.
- ❑ Teile einer Klasse werden ausgelagert, um deren Komplexität zu verringern.
- ❑ Hinzunahme zusätzlicher direkter Assoziationen zwischen Klassen, um mehrstufige Delegation zu vermeiden.
- ❑ implementierungsgerechte Umgestaltung von Klassen:
 - Hinzunahme zusätzlicher Klassen zur Umsetzung von $n:m$ -Beziehungen
 - Hinzunahme zusätzlicher Klassen zur Umsetzung von Assoziationsklassen
 - Hinzunahme zusätzlicher Klassen zur Umsetzung von Kompositionen
 - Festlegen der Navigationsrichtung für ungerichtete Assoziationen

Verbesserung des Grobentwurfs

(Fortsetzung)

Der Übergang vom Entwurf zur Implementierung ist unscharf:
UML-Editoren erzeugen z.B. Coderahmen und verschieben so Aspekte aus der Implementierung in den Entwurf.

Anpassung an die vorgesehene Programmiersprache

- ❑ Geeigneten Typen für Attribute festlegen.
- ❑ Nutzung von Bibliotheksklassen vorsehen und vorbereiten.
Gegebenenfalls muss eine erneute Anpassung der Klassenstruktur erfolgen.
- ❑ Insbesondere kann Wiederverwendung auch durch die Konkretisierung von bereits vorhandenen generischen Klassen erfolgen.
- ❑ Umgestaltung von den in der ausgewählten Programmiersprache nicht realisierbaren Strukturen des Entwurfs.

Zusammenfassung

Das Ergebnis der vorgestellten Überarbeitungen/Ergänzungen ist ein

Entwurf,

- ❑ der die Anforderungen umsetzt und
- ❑ die technische Realisierung in einer bestimmten Programmiersprache ermöglicht.

Anmerkungen:

- ❑ Alle beschriebenen Überarbeitungen/Ergänzungen finden auf der konzeptionellen Ebene statt – also **bevor** Programmtext erstellt wird.
- ❑ Die Überarbeitung von bereits existierenden Programmen kann nach ähnlichen Regeln erfolgen und wird als **Refactoring** bezeichnet.

Hinweis

Die hier vorgestellten Entwicklungsschritte können nur schwer im Rahmen überschaubarer Übungsaufgaben durchgeführt werden.

Das Üben dieser Schritte erfolgt daher im Rahmen des

Softwarepraktikums.

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 5.8: Agile Vorgehensmodelle

Entwicklungsprozesse – kritische Betrachtung

Die Folge der wohldokumentierten Phasen

Analyse, Konzeption/Entwurf, Realisierung/Implementierung, Überprüfung/Test

wird in anderen Ingenieurwissenschaften in ähnlicher Form angewandt:
Architektur, Maschinenbau, Elektrotechnik, Anlagenbau, ...

aber:

Software ist immateriell

- ⇒
 - leichte Änderbarkeit
 - fast beliebige Änderbarkeit
 - Modell/Prototyp kann zugleich Endprodukt sein
 - Möglichkeiten einer visuellen Prüfung sind eingeschränkt

- ⇒ andere Entwicklungsprozesse sind möglich

Entwicklungsprozesse – kritische Betrachtung

(Fortsetzung)

Probleme, die bei jeder Softwareentwicklung auftreten können:

- ❑ Verständnisschwierigkeiten mit dem Problembereich fallen erst spät bei der Konkretisierung (= Implementierung) auf.
- ❑ Der Aufwand für die Analyse und deren Dokumentation ist hoch.
- ❑ Spätere Änderungen müssen in allen Dokumenten vorgenommen werden.
- ❑ Die Phasen sind nicht gleichgewichtig:
Implementierung und Testen benötigen viel Zeit.
- ❑ Empirische Untersuchung (CHAOS-Studie, Standish Group) behauptet:
 - nur ca. 16 % der Softwareentwicklungen sind erfolgreich,
 - 53 % sind nur teilweise erfolgreich,
 - 31 % werden ohne Erfolg abgebrochen.
- ❑ Gründe für Misserfolg von Softwareprojekten:
 - Zeitüberschreitung,
 - Budgetüberschreitung,
 - unzureichende Qualität (fehlerhafte Funktionalität),
 - falsch ermittelte Anforderungen,
 - zu spät erkannte zu hohe Komplexität.

Agile Vorgehensmodelle

Unter dem Begriff *Agile Vorgehensweise* werden zusammengefasst:

- ❑ größere Flexibilität
- ❑ weniger "Bürokratie"
- ❑ weniger Dokumentation
- ❑ stärkere Fokussierung auf die Programmiertätigkeit

- ❑ erste Veröffentlichung (Beck 1999):
eXtreme Programming (XP)

- ❑ weiteres Beispiel:
Scrum

Literatur: Hanser, Eckhart: Agile Prozesse: Von XP über Scrum bis MAP, S. 1-45, S. 61-77.
http://link.springer.com/chapter/10.1007/978-3-642-12313-9_3
http://link.springer.com/chapter/10.1007/978-3-642-12313-9_5

Vorgehensmodell eXtreme Programming (XP)

Prinzipien des eXtreme Programming:

- ❑ Alle beteiligten Personen betreiben ständige, persönliche Kommunikation.
- ❑ Die Entwickler handeln schnell, flexibel und selbstständig.
- ❑ Das Programm bildet das zentrale Dokument.
- ❑ Die Programmkonstruktion wird bewusst einfach gehalten.
- ❑ Das Programm wird ständig verändert.
- ❑ Das Programm besitzt jederzeit eine hohe Qualität.

Vorgehensmodell eXtreme Programming (XP)

(Fortsetzung)

Praktiken des eXtreme Programming

- ❑ **on-site customer:**
Der Kunde ist direkt in die Entwicklung eingebunden und Teil des Entwicklungsteams.
 - *Vorteil:* Fragen/Probleme können sofort geklärt werden.
 - *Nachteil:* Aufwand für den Kunden ist hoch, «entbehrliche» Mitarbeiter mit großer Kompetenz geben.
- ❑ **planning game:**
Das Produkt wird durch Planspiele interaktiv vom ganzen Team (inkl. Kunde) ermittelt.
- ❑ **metaphor:**
 - Fachbegriffe werden durch den Kunden in Form von *user stories* erstellt.
 - Es wird ein Glossar der benutzten Fachbegriffe erstellt.
- ❑ **simple design:**
Es wird bewusst auf die Entwicklung allgemeingültiger Strukturen verzichtet.
Es wird bewusst auf technisch anspruchsvolle Lösungen verzichtet.
 - *Vorteile:* Kosten werden reduziert, unmittelbare Qualität wird verbessert
 - *Nachteil:* eventuell späteres Überarbeiten notwendig

Vorgehensmodell eXtreme Programming (XP)

(Fortsetzung)

Praktiken des eXtreme Programming (Fortsetzung)

- ❑ **pair programming:**
Ein Entwickler (Driver) programmiert, ein zweiter (Observer) beobachtet die Arbeit. Die Rollen und die Partner werden häufig gewechselt.
 - *Vorteile:* Steigerung der Qualität durch Kontrolle, schwere Aufgaben werden bei der Lösung diskutiert, Entwickler lernen voneinander
 - *Nachteil:* doppelter Personalbedarf
- ❑ **collective ownership:**
Alle Ergebnisse sind kollektives Eigentum des Teams. Die Aufgaben wechseln (sehr) häufig, jeder Entwickler bearbeitet alles.
 - *Vorteile:* weniger Probleme bei Inkompetenz, Krankheit, Urlaub usw.,
 - *Nachteil:* Spezialisten werden nicht adäquat eingesetzt, ständiges Einarbeiten in geänderten Code
- ❑ **coding standards:**
 - Alle Programme halten sich an strenge Codestandards.
 - Codestandards sind die Grundvoraussetzung, damit Paarprogrammierung und kollektives Eigentum angewandt werden können.

Vorgehensmodell eXtreme Programming (XP)

(Fortsetzung)

Praktiken des eXtreme Programming (Fortsetzung)

- ❑ **short releases:**
Eine Vorausplanung erfolgt nur in sehr kurzen Zeitabschnitten (Stunden/Tage).
 - *Vorteil:* flexible Anpassung an Kundenwünsche möglich.
 - *Nachteil:* kein systematisches Hinarbeiten auf ein Gesamtziel
- ❑ **continous integration:**
Alle Neuentwicklungen werden sofort integriert, Programm ist immer ausführbar.
 - *Vorteil:* Kunde sieht immer das ganze Produkt und kann es prüfen.
 - *Nachteil:* Entwicklung beginnt zwangsläufig mit der Benutzungsoberfläche
- ❑ **test first:**
Testfälle werden vor der Implementierung einer Methode festgelegt und ersetzen eine detaillierte Spezifikation. Ständiges Testen ermöglicht die ständige Integration.
 - *Vorteil:* Qualität der Implementierung wird jederzeit sichergestellt.
- ❑ **refactoring:**
Die Weiterentwicklung wird für eine Überarbeitungsphase unterbrochen, in der das gesamte Programm technisch restrukturiert und verbessert wird.
 - *Vorteil:* Überarbeitung erfolgt bedarfsorientiert und explizit.
 - *Nachteil:* zusätzlichen Aufwand, da die Einzelmaßnahmen aufwändig sein können.

Vorgehensmodell eXtreme Programming (XP)

(Fortsetzung)

Praktiken des eXtreme Programming (Fortsetzung)

- ❑ **40 hour week:**
Annahmen: Überstunden zeigen unzureichender Planung, Entwickler werden mit langer Arbeitsdauer unproduktiver und liefern schlechtere Qualität

Vorgehensmodell eXtreme Programming (XP)

Gesamtbewertung:

- ❑ eXtreme Programming ist **anders** als *klassische* Softwareentwicklung.
- ❑ eXtreme Programming lässt sich schwer planen und überwachen.
- ❑ Eine Idee zur Kostenplanung ist:
 Es wird einfach solange inkrementell geplant und weiterentwickelt,
 bis der Kunde zufrieden oder das Budget aufgebraucht ist.
- ❑ eXtreme Programming ist ein dogmatischer Ansatz (= alle Praktiken sind unverzichtbar).
- ❑ eXtreme Programming erfordert kompetente Kunden.
- ❑ eXtreme Programming erfordert kompetente Entwickler.
- ❑ eXtreme Programming kann nur in Teams mit überschaubarer Größe eingesetzt werden.

aber:

- ❑ Testgetriebene Entwicklung (*test first*) wird inzwischen auch in der *klassischen* Entwicklung als Teilschritt am Übergang Entwurf/Implementierung eingesetzt.
- ❑ *refactoring* (mit entsprechenden Überarbeitungsphasen) wird inzwischen auch in der *klassischen* Entwicklung eingesetzt.

Vorgehensmodell Scrum

- ❑ Begriff:
Scrum = Gedränge im Rugby (organisiertes Chaos)
- ❑ Eigenschaften:
 - Rahmen ist nicht so dogmatisch wie beim eXtreme Programming.
 - Schwerpunkt liegt auch auf Agilität.
 - Einsatz auch nur in kleinen Gruppen mit bis zu 10 Entwicklern.
(Vollzeit im Projekt beschäftigt, Arbeitsplätze in einem Raum)
- ❑ Rollen
 - *pig* = Entwickler
 - *scrum master* = Moderator: überwacht die Einhaltung der Prozessregeln
 - *product owner*: vertritt den Auftraggeber im Team
 - *chicken* = Informationsquelle

Vorgehensmodell Scrum

(Fortsetzung)

- ❑ Das Projekt wird in *sprints* organisiert:
 - *sprint* dauert bis zu 30 Tagen.
 - Umfang/Ziel wird vorher festgelegt und mit dem *product owner* abgestimmt.
 - Ergebnis wird vom *product owner* im *sprint review* bewertet.
 - Prozess wird in *retrospective* vom Team bewertet.

- ❑ *daily scrum*
ist die tägliche Arbeitsplanung in der Gruppe.

Vorgehensmodell Scrum

(Fortsetzung)

Artefakte

- ❑ Anforderungen
sind (natürlichsprachlich formulierte) *stories* :
Wer braucht was wozu/warum?
- ❑ *stories* werden im *backlog* (Arbeitsrückstand) gesammelt.
- ❑ Ausgewählte *stories* bilden den *sprint backlog*.
- ❑ *sprint burndown chart* zeigt den Arbeitsfortschritt.
- ❑ *product increment* ist das Ergebnis eines *sprint*.

Vorgehensmodell Scrum

(Fortsetzung)

Gesamtbewertung:

- ❑ Scrum versucht, große Projekte in überschaubare Zwischenschritte (*sprints*) aufzuteilen.
- ❑ Scrum läßt sich dadurch etwas besser planen und überwachen als XP.
- ❑ Scrum verzichtet (wie XP) auf aufwändige Dokumentation.
- ❑ Scrum erfordert kompetente Kunden.
- ❑ Scrum kann ebenfalls nur in Teams mit überschaubarer Größe eingesetzt werden.
- ❑ Scrum kann etwas besser mit heterogenen Kompetenzen der Mitarbeiter umgehen.

Zusammenfassung agile Methoden

- ❑ Agile Methoden stellen die Programmier-Tätigkeit und den Programmier-Experten mehr in den Vordergrund.
- ❑ Agile Methoden setzen darauf, dass Software einfach änderbar ist.
- ❑ Agile Methoden setzen darauf, dass Software verständlich gestaltet werden kann.
- ❑ Agile Methoden erfordern viel Kommunikation im Team und mit dem Auftraggeber.
- ❑ Agile Methoden erfordern geeignete Werkzeugunterstützung für die Teamarbeit.

Folien zur Vorlesung **Softwaretechnik**

Teil 6: Zusammenfassung

Abschluss Vorlesung Softwaretechnik

Inhalte

- ❑ UML
 - Klassendiagramm/Objektdiagramm
 - Aktivitätsdiagramm
 - Sequenzdiagramm
 - Anwendungsfalldiagramm
- ❑ Testen
 - funktionsorientierter Test
 - strukturorientierter Test
 - JUnit
- ❑ Vorgehensmodelle
 - Software-Architekturen
 - Entwurfsprozess
 - agile Softwareentwicklung
- ❑ Entwurfsmuster

Abschluss Vorlesung Softwaretechnik

(Fortsetzung)

Die Inhalte repräsentieren den «**State-of-the-Art**» der einfachen Softwaretechnik.

fehlende Inhalte

- ❑ vertiefte Darstellung aller Inhalte
- ❑ formale Techniken zur Spezifikation
- ❑ (formale) Techniken zur Spezifikation spezieller Eigenschaften (z.B. Sicherheit)
- ❑ Prozesse zur Entwicklung spezieller Software
- ❑ Projektplanung und -durchführung
- ❑ unterstützende Softwarewerkzeuge