

Analyse – Entwurfsmuster Dekorierer

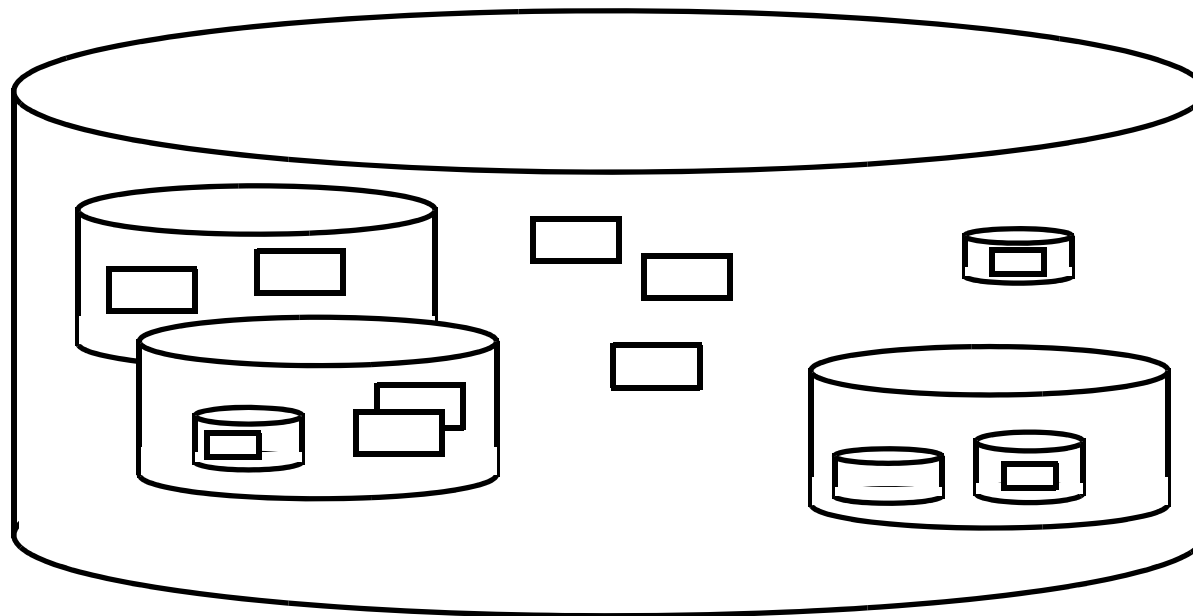
- ❑ Die grundlegende Idee des Dekorierer ist es, einen Stapel/eine Liste aus Objekten unterschiedlicher Klassen aufzubauen, die nach außen die gleiche Schnittstelle anbieten, also gleich behandelt werden können, aber unterschiedliche Implementierungen von Methoden besitzen (können).
- ❑ Diese Idee lässt 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

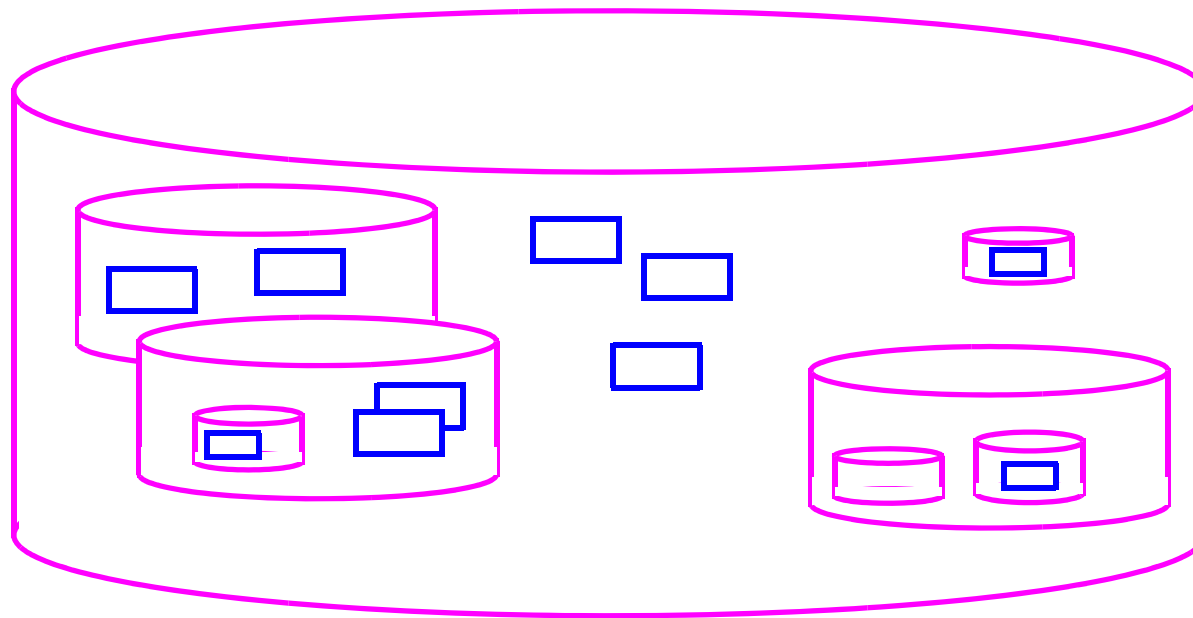
 Directory

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

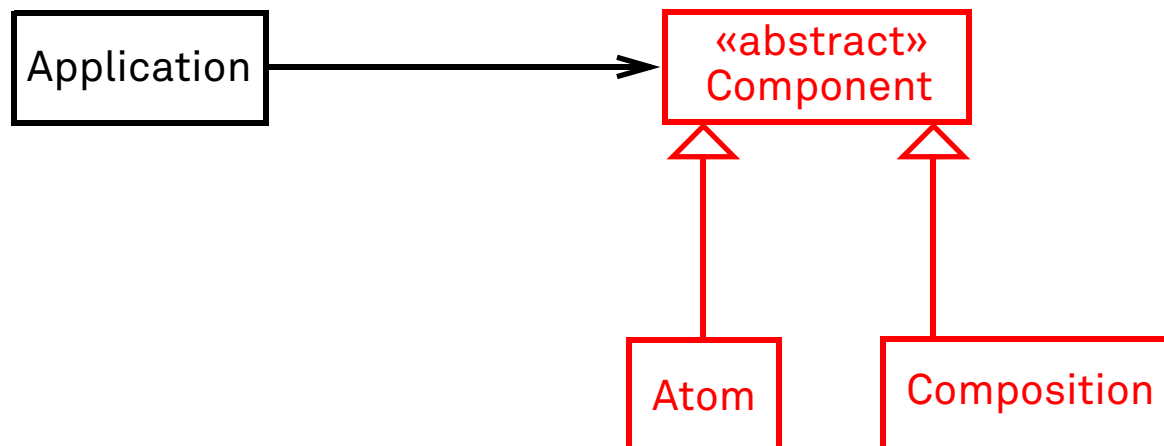
http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_5

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:

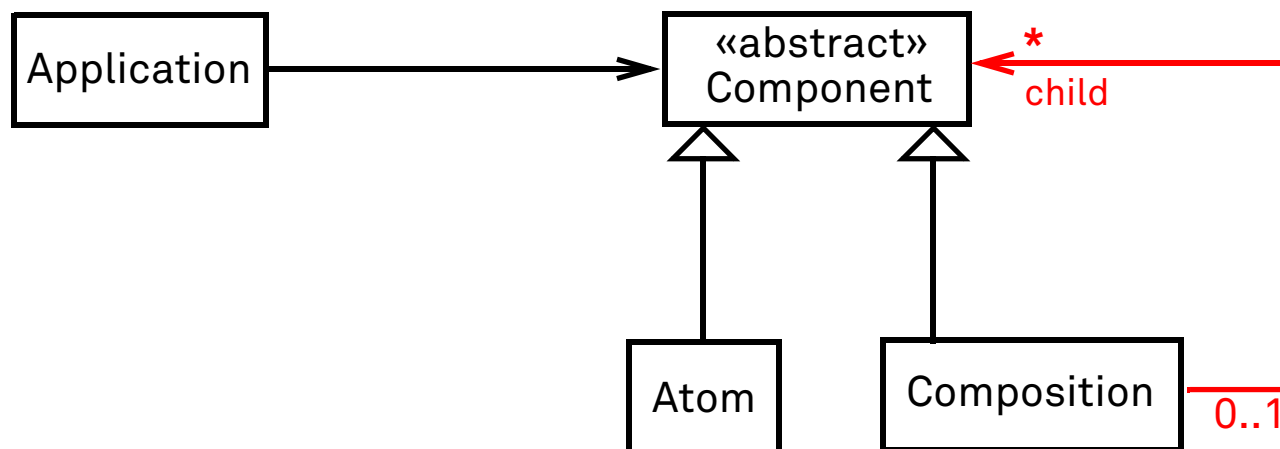


Analyse der Dateistruktur

(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:

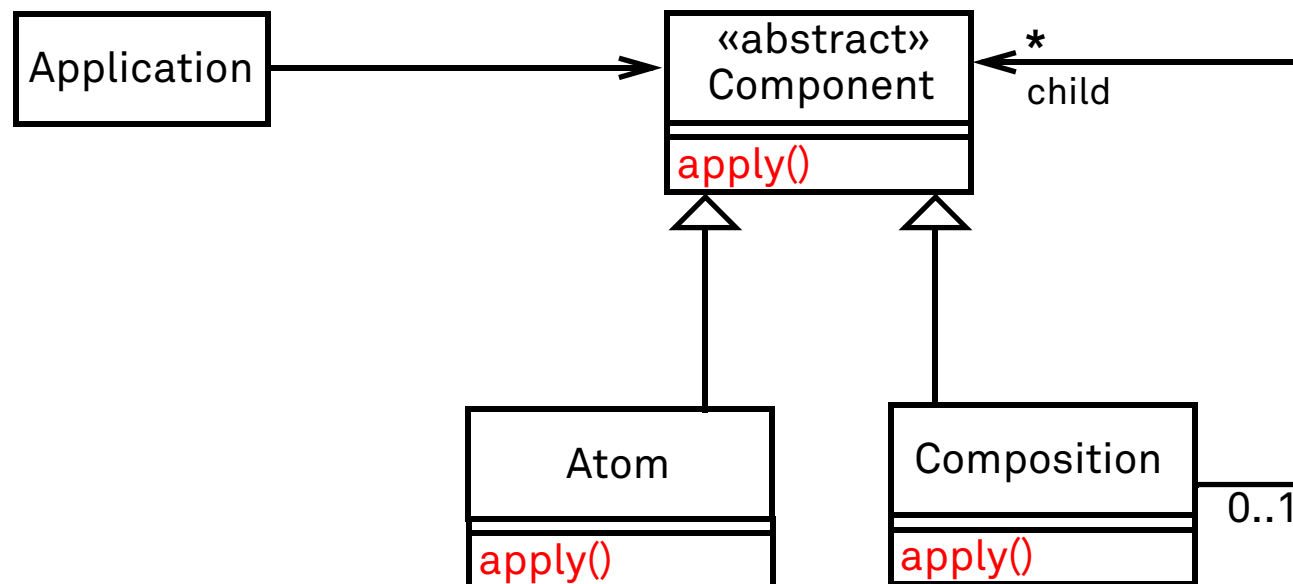


Die Wurzel hat keinen Vorgänger

Analyse der Dateistruktur

(Fortsetzung)

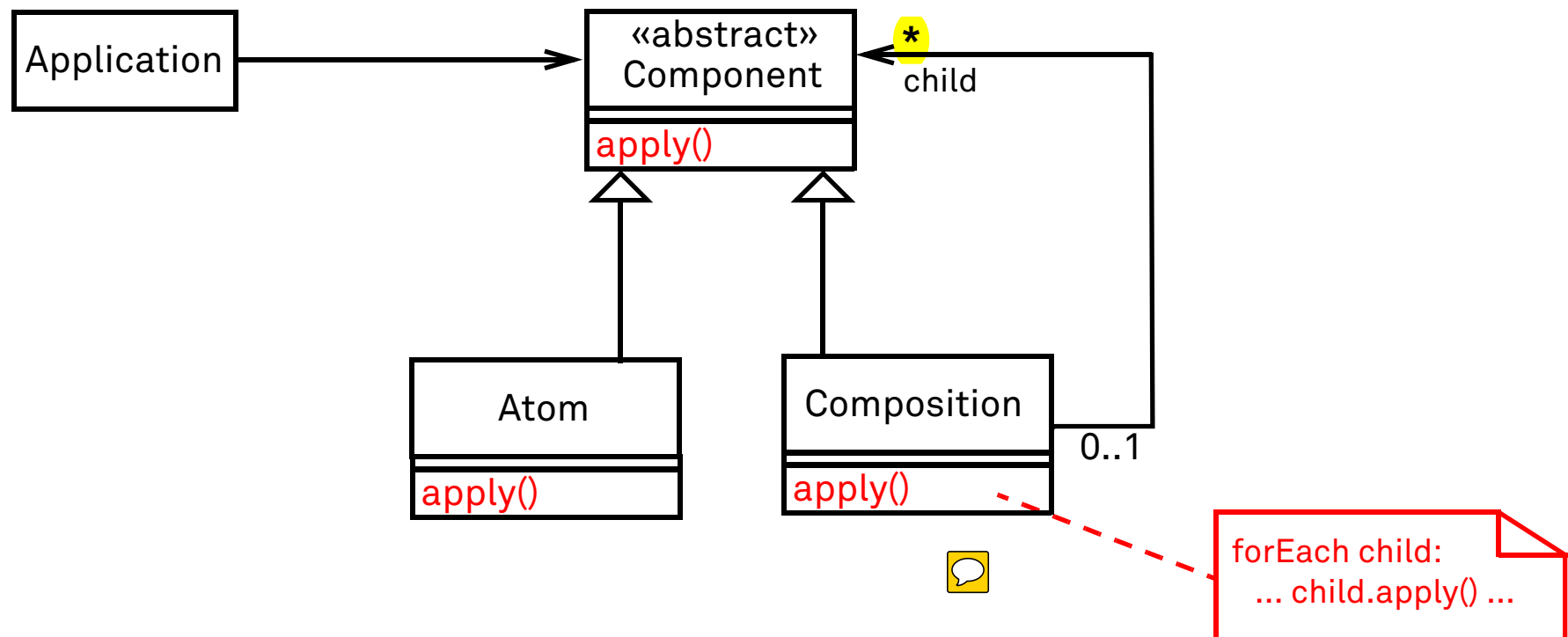
- ❑ Die abstrakte Komponente muss die Methoden anbieten, die für
 - Atome und
 - Kompositionen aufgerufen werden können.
- ❑ Methodenaufrufe auf Kompositionen müssen auf die Kinder übertragen werden.



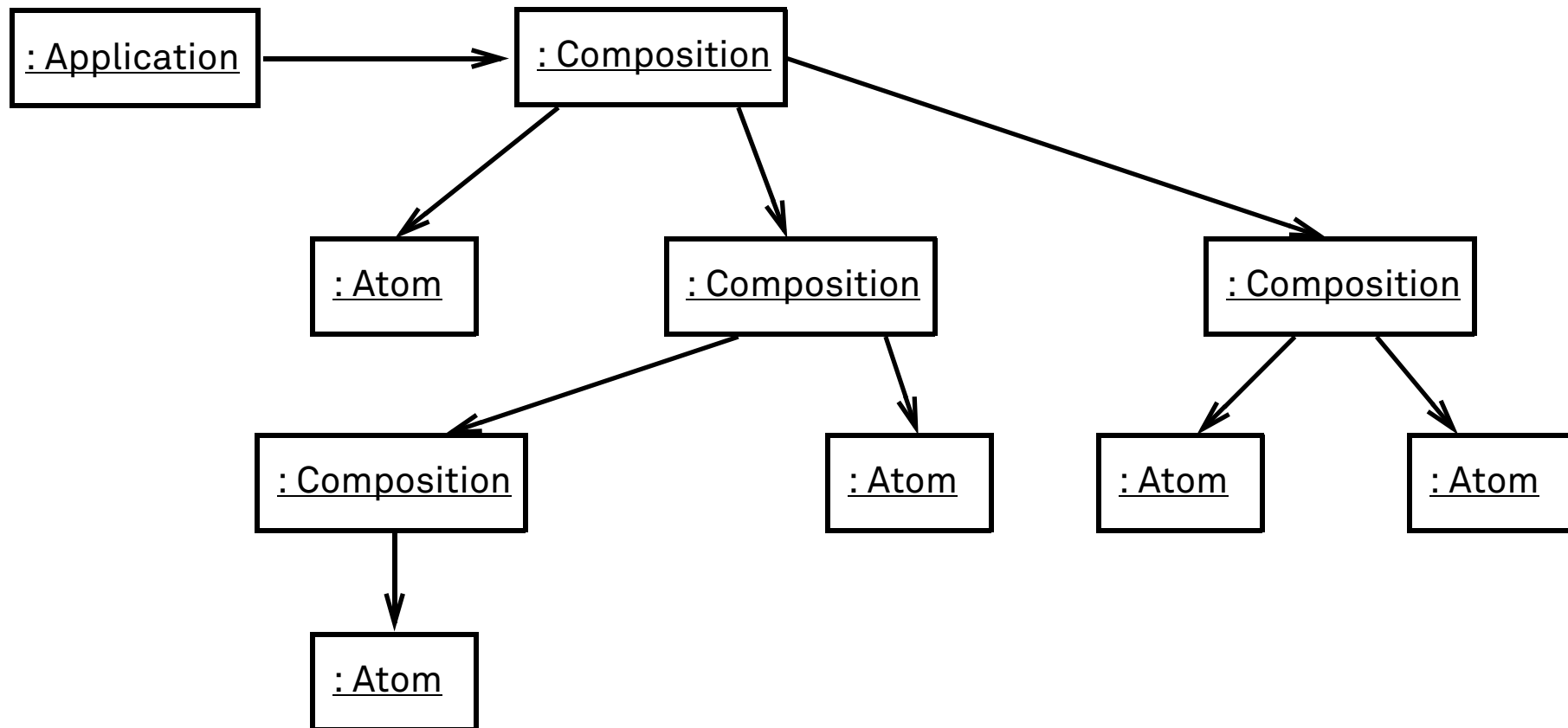
Analyse der Dateistruktur

(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.

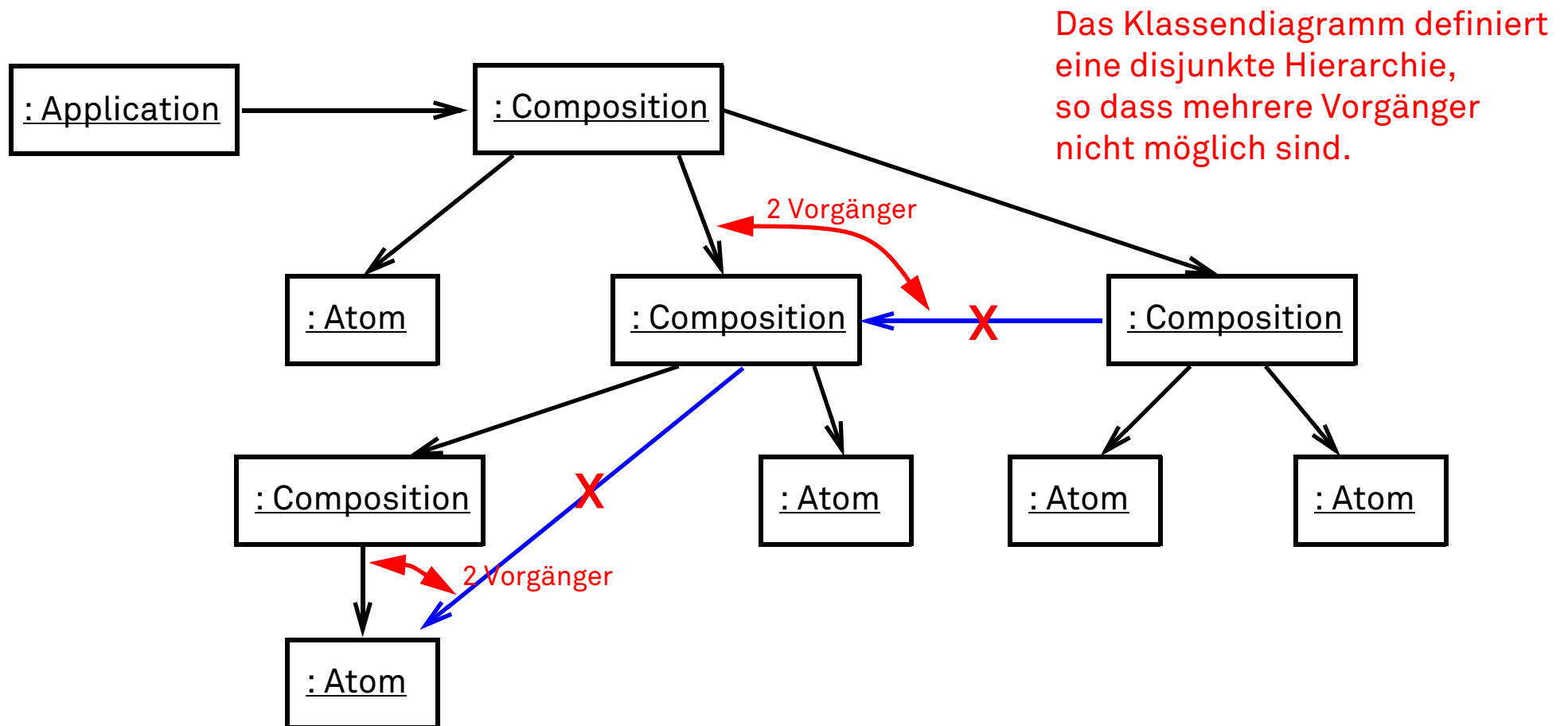


Beispiel für ein zugehöriges Objektdiagramm

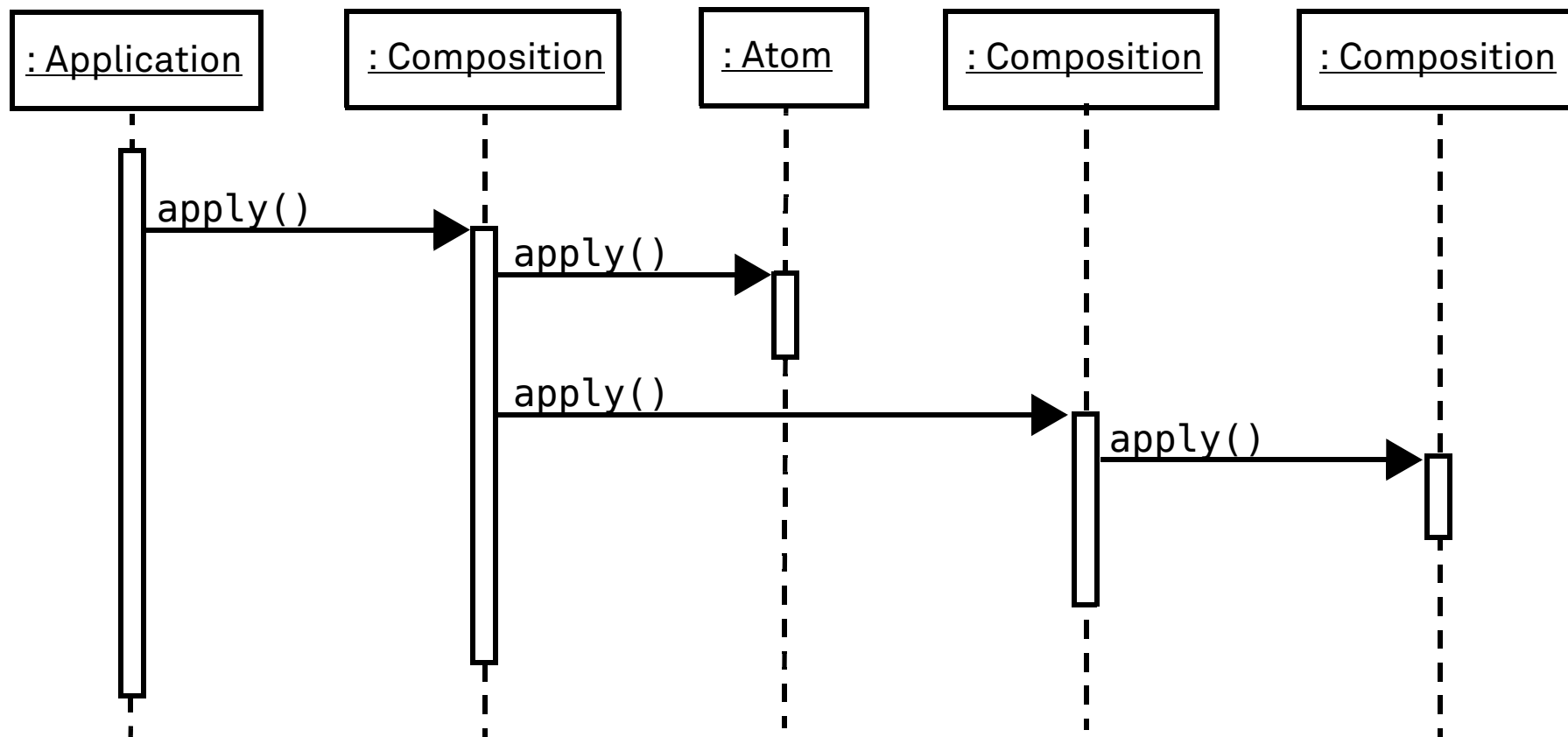


Beispiel für ein Objektdiagramm

(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)

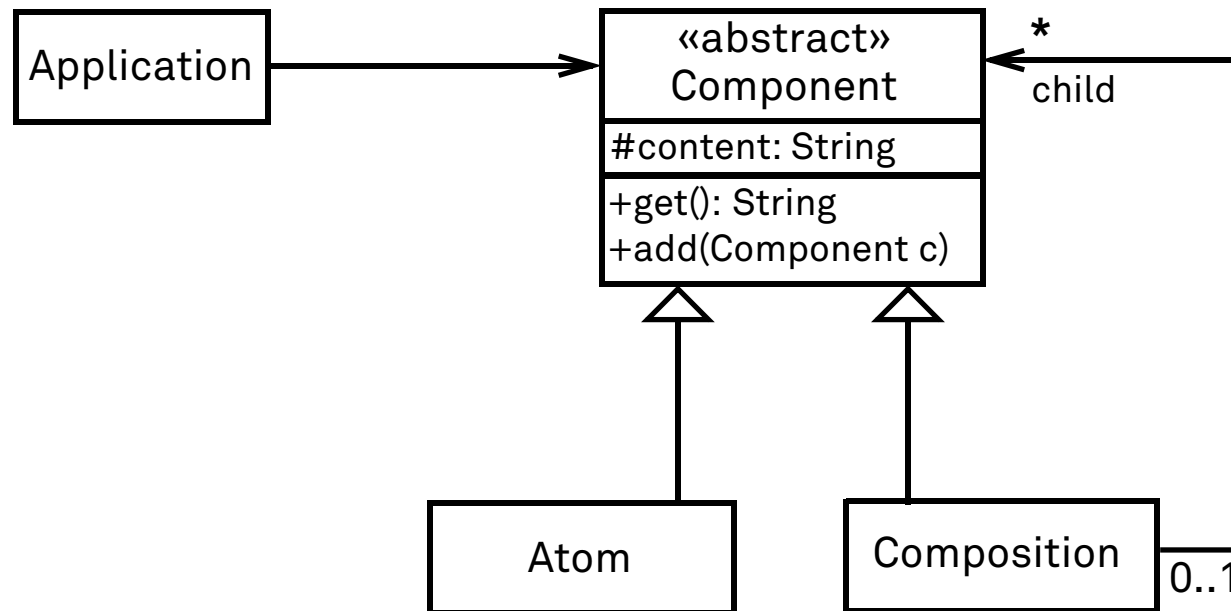
```
public abstract class Component {
    protected String content;
    public abstract String get();
    public abstract void add(Component c);
}

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

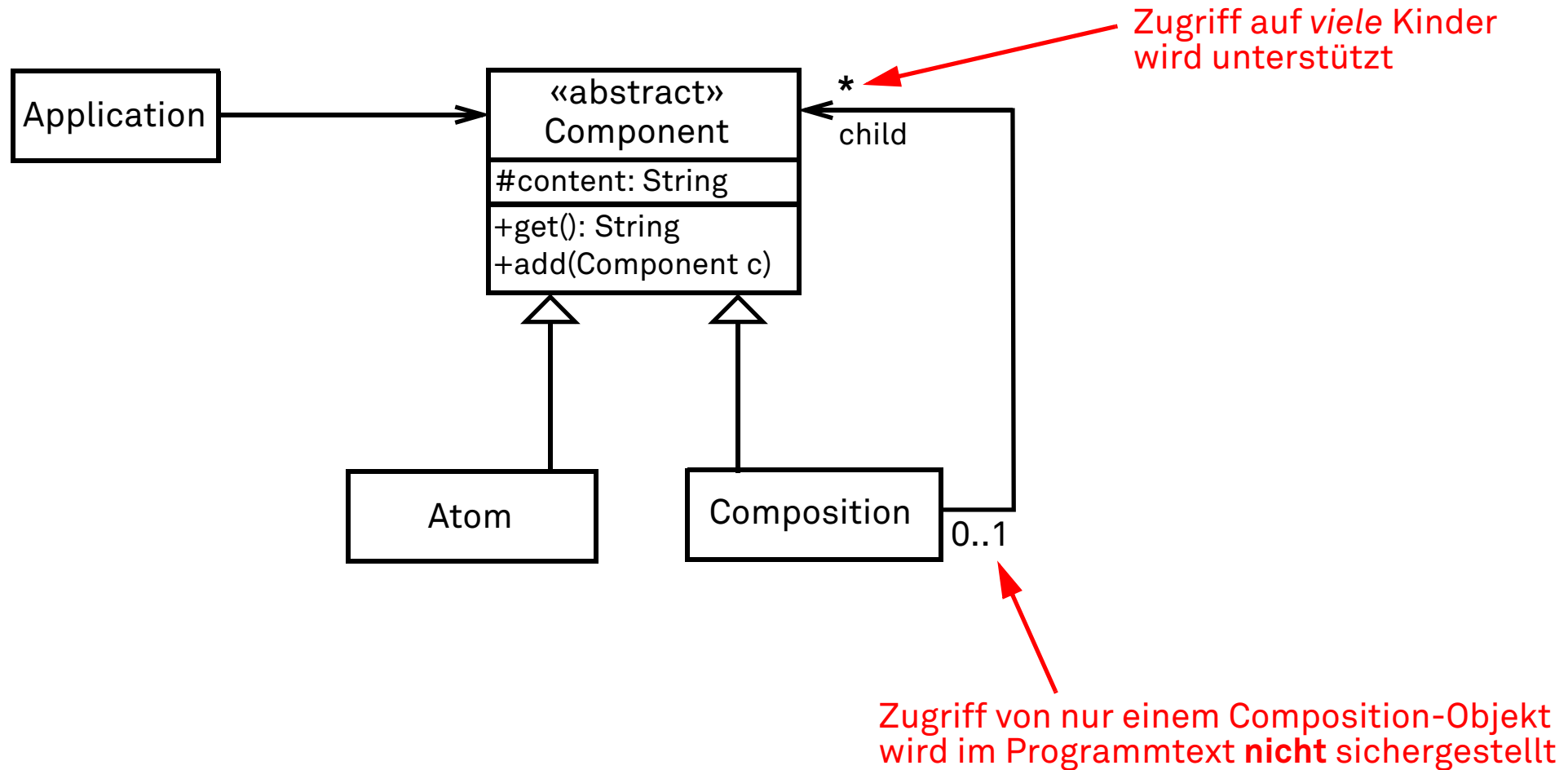
public class Composition extends Component {
    private ArrayList<Component> children = new ArrayList<Component>();
    public Composition(String s) { content = s; }
    public String get() {
        String all = content;
        for (Component c: children) { all += c.get(); }
        return all;
 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

Methode ist nur für eine Unterklasse sinnvoll

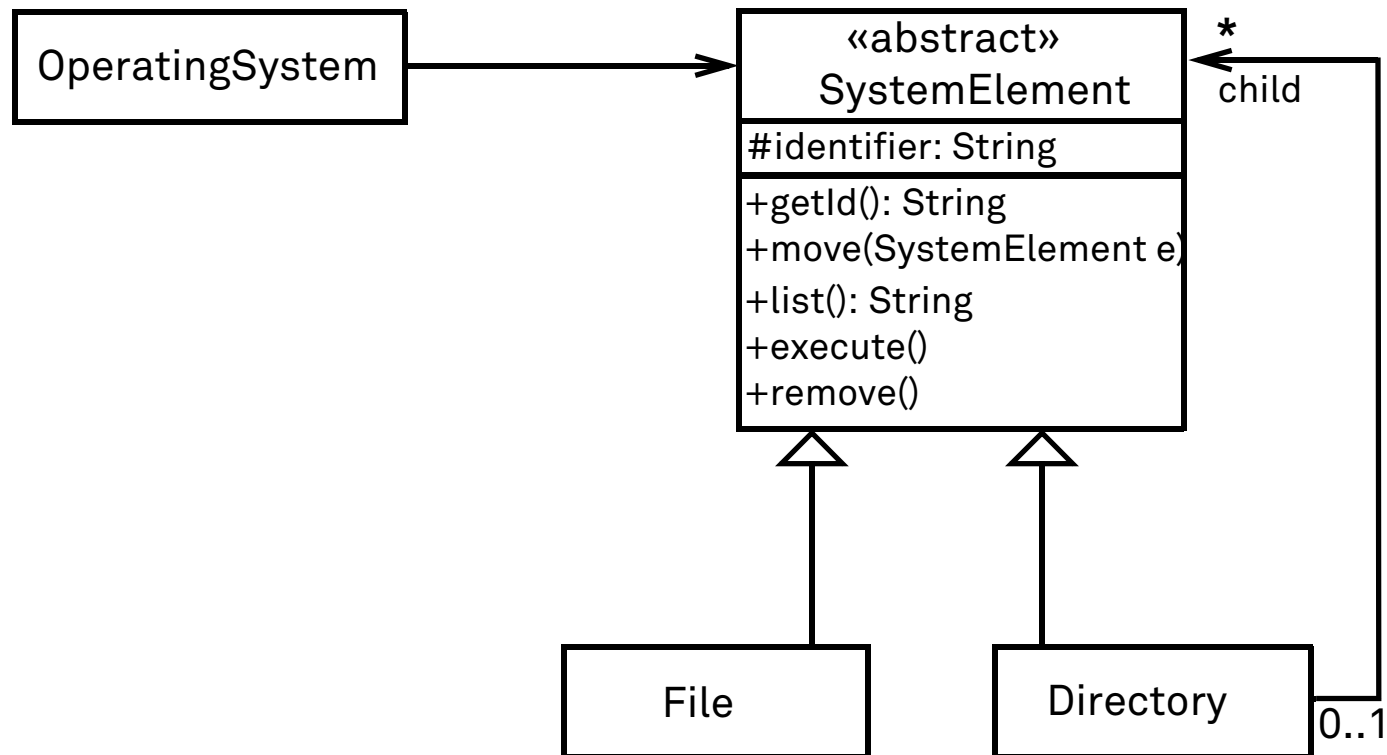
Klassendiagramm für die Beispiel-Implementierung



Anmerkungen zur Implementierung

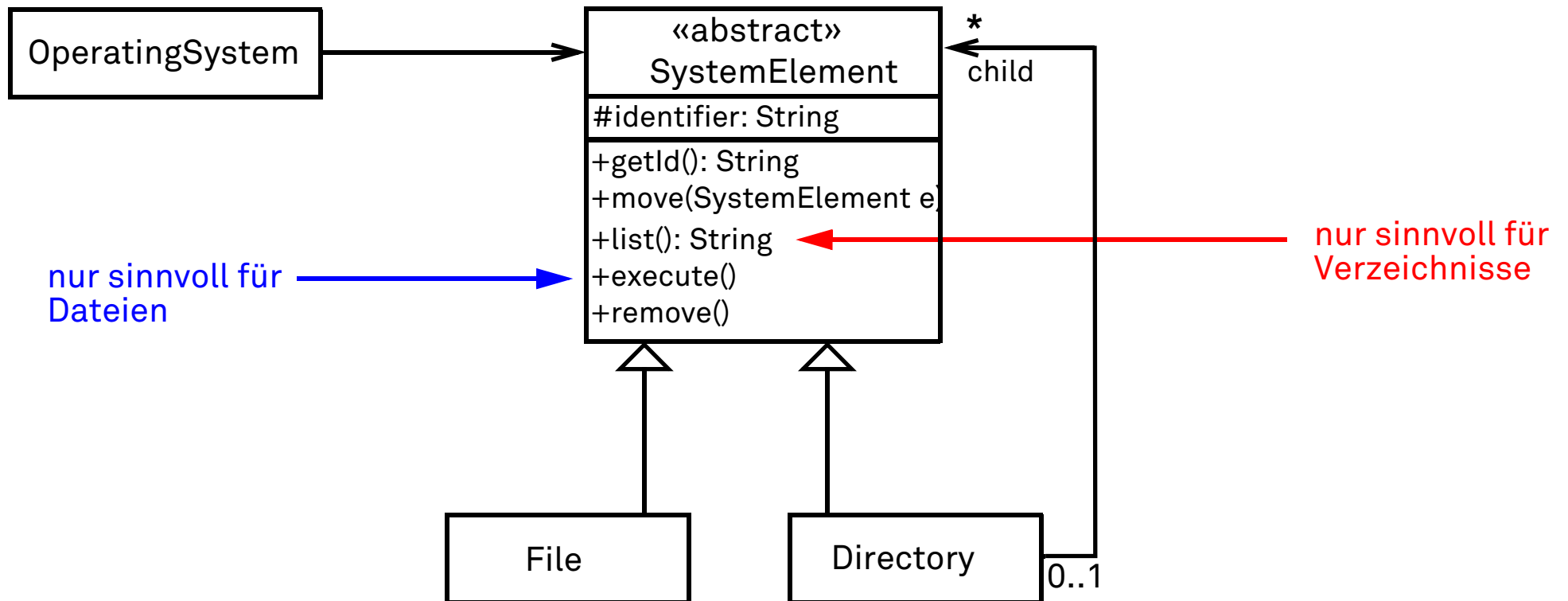


Klassendiagramm für ein Dateisystem (Beispiel)



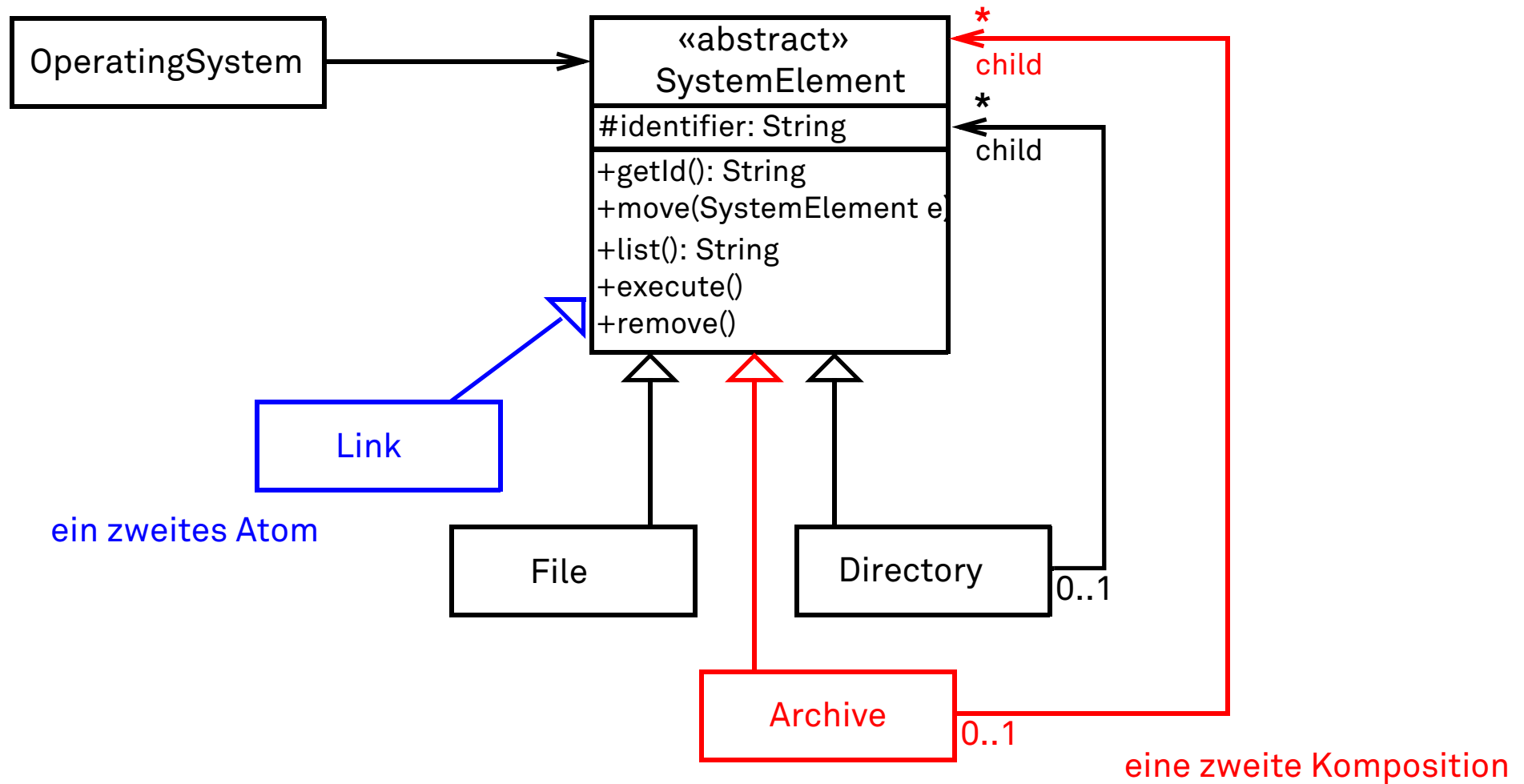
Klassendiagramm für ein Dateisystem (Beispiel)

(Fortsetzung)



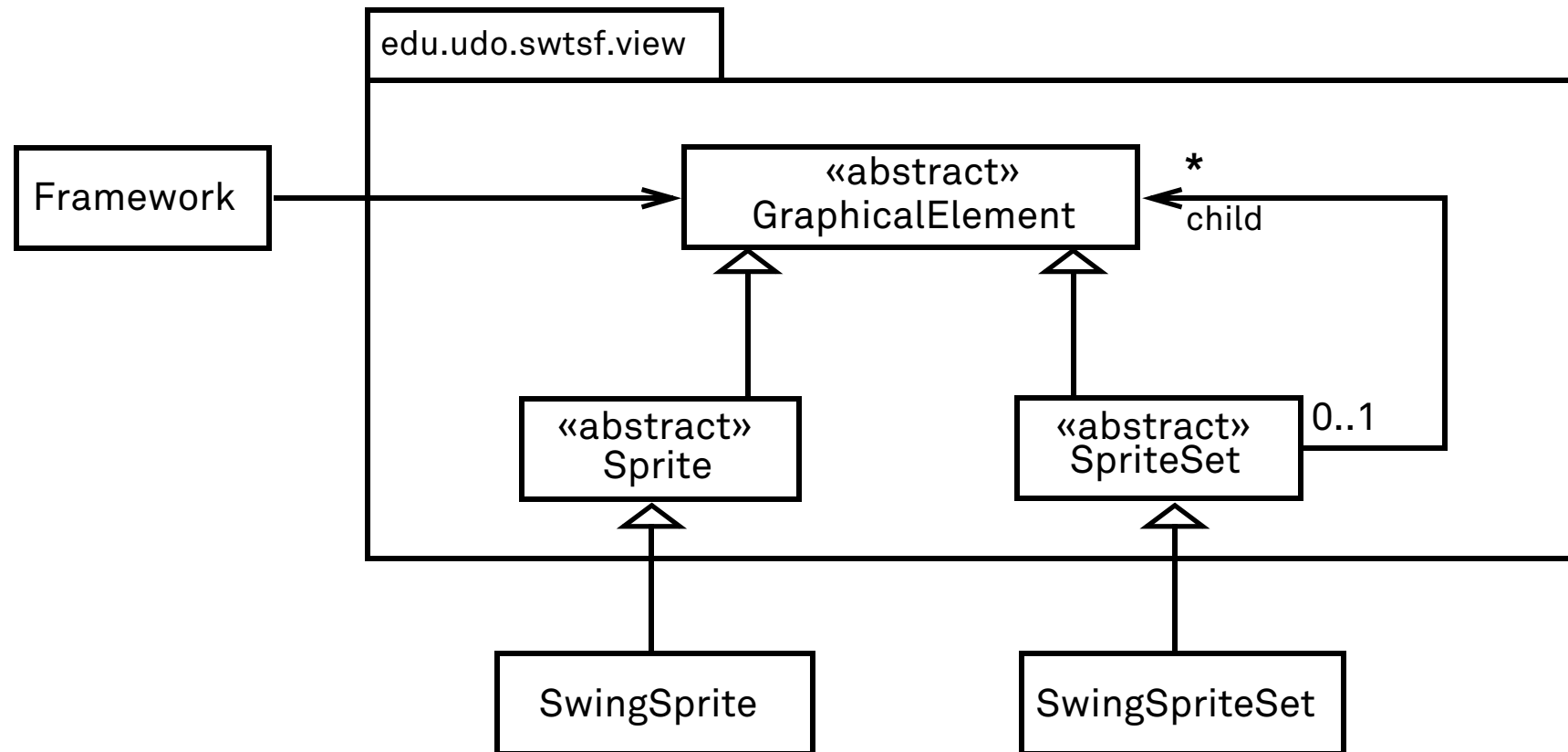
Klassendiagramm für ein Dateisystem (Beispiel)

(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.

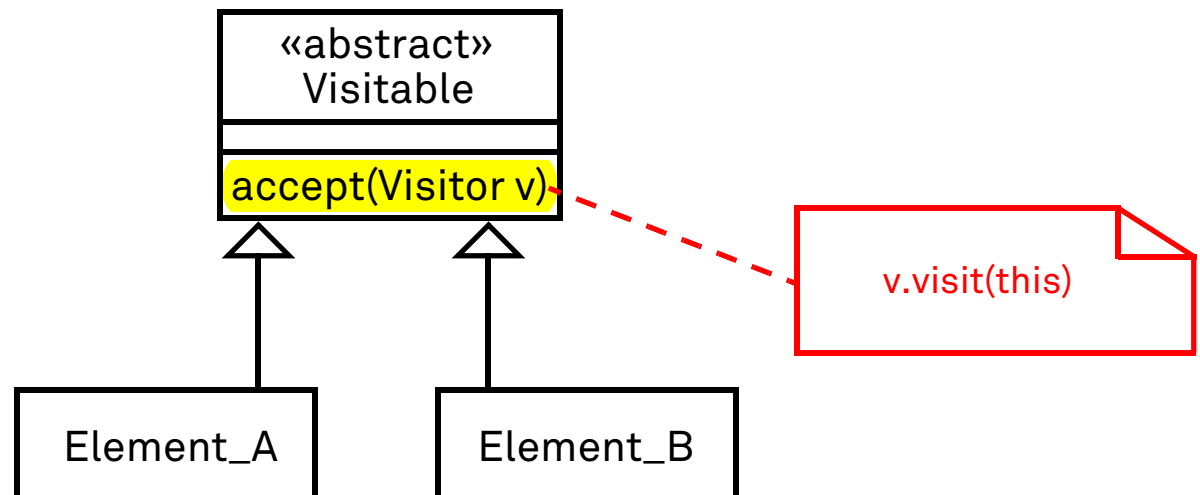
Literatur: Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 52-57
http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_4

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:

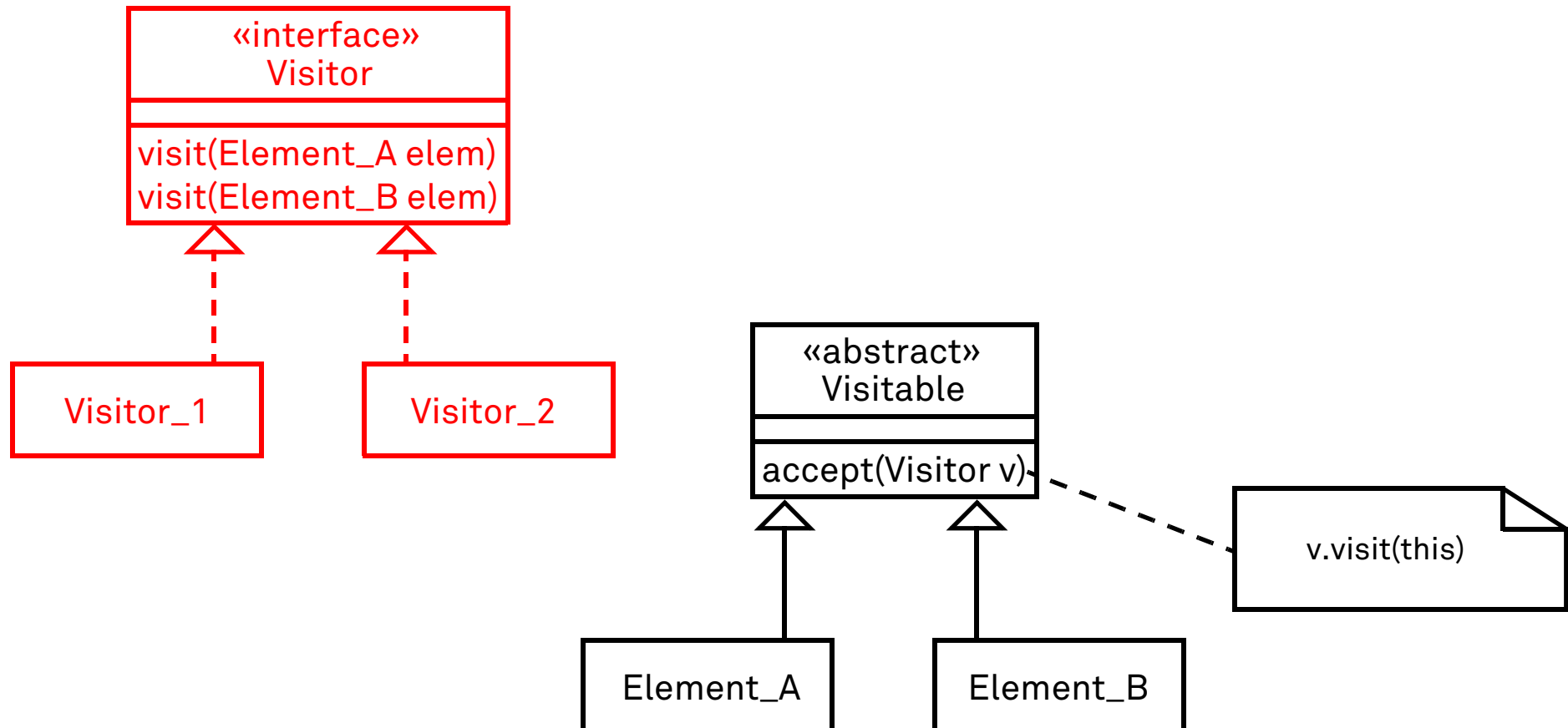
```
void accept(Visitor v) { v.visit(this); }
```



Entwurfsmuster Besucher

(Fortsetzung)

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

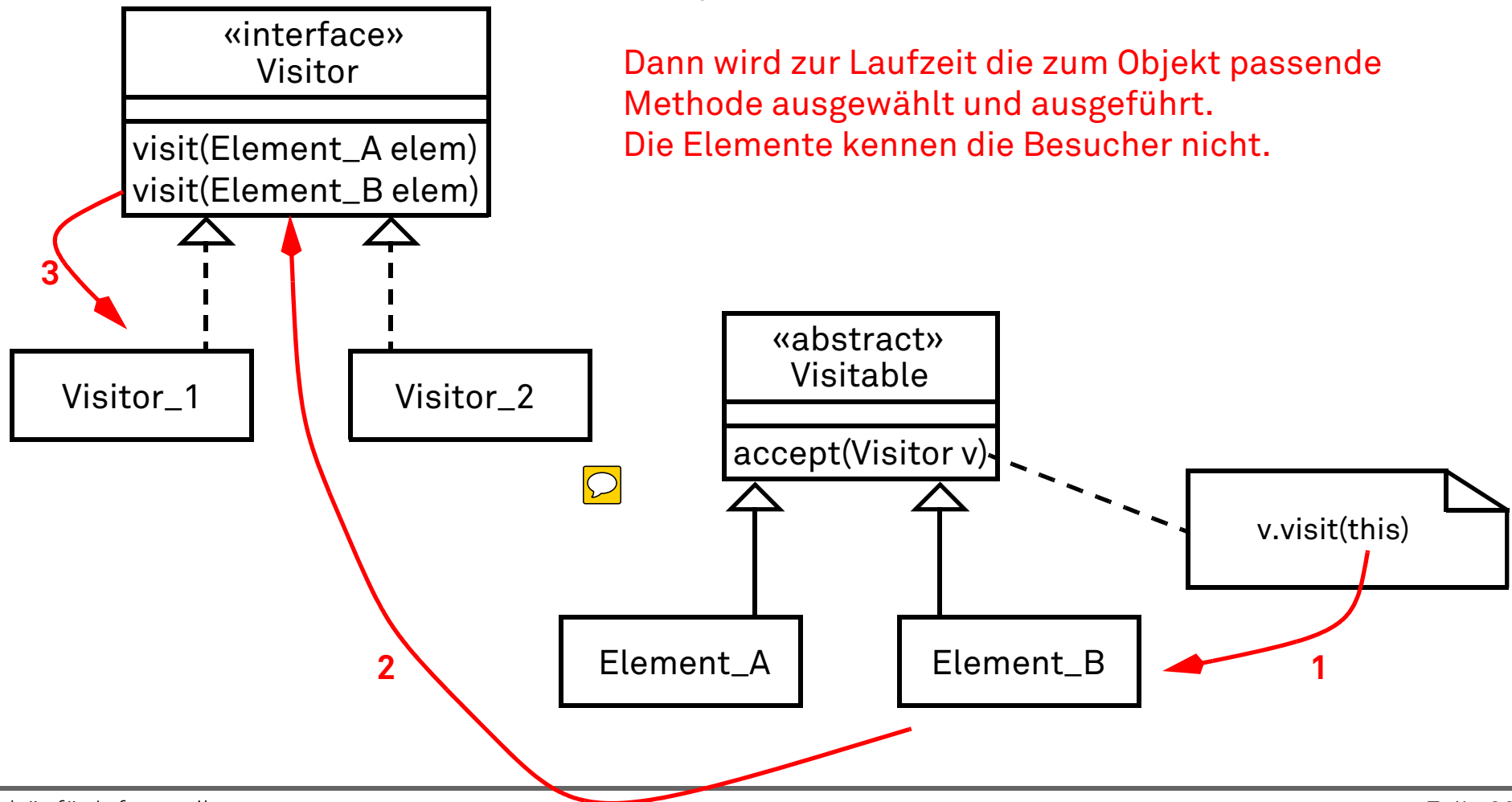


Entwurfsmuster Besucher

(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.

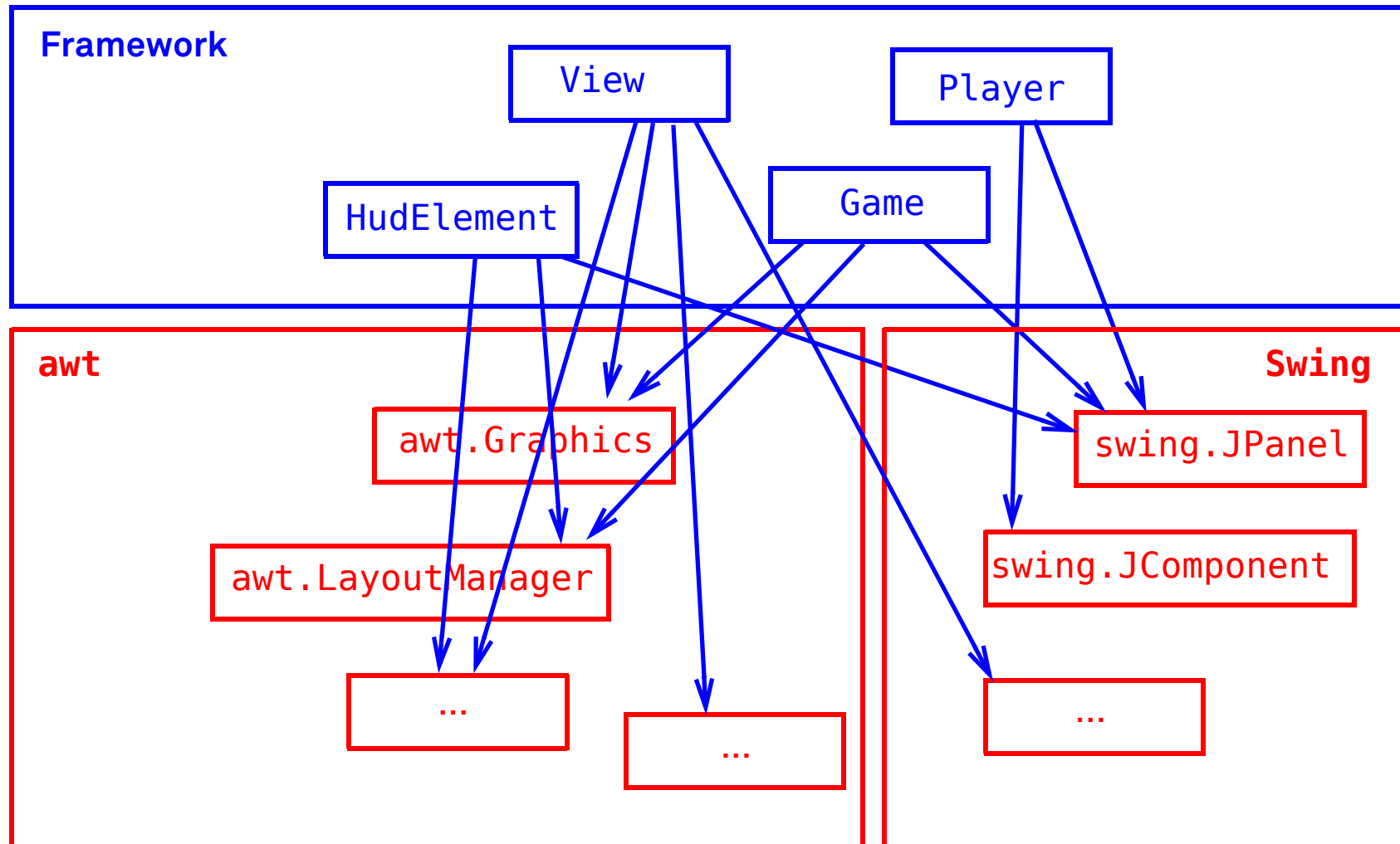
Literatur: Rau, Karl-Heinz: Objektorientierte Systementwicklung – Vom Geschäftsprozess zum Java-Programm, S.219-221

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:

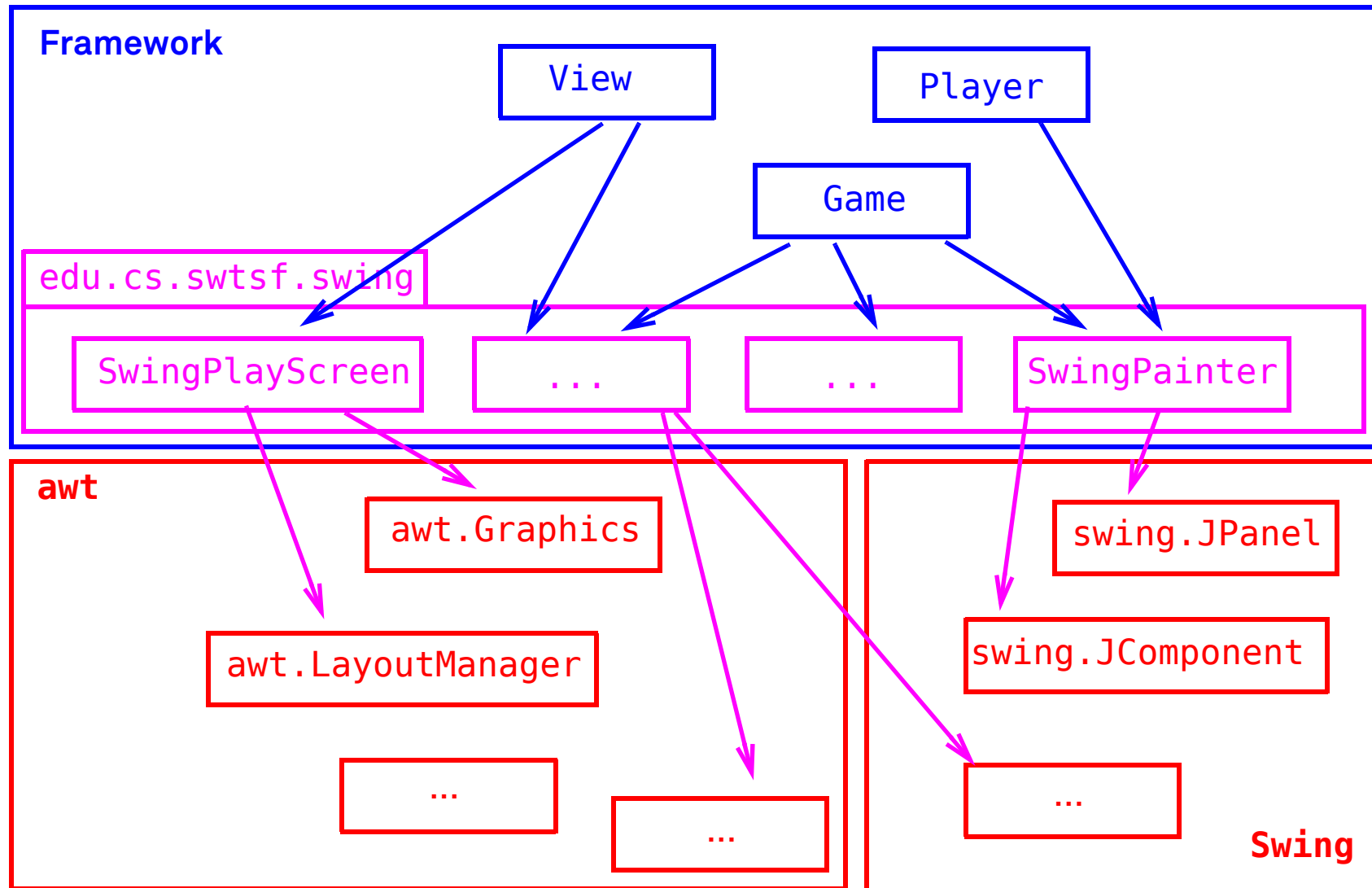
- ❑ Eine Klassen aus dem Framework nutzt eventuell mehrere Klassen der Bibliotheken.
- ❑ Alle Entwickler des Frameworks benötigen die Kompetenz, die Bibliothek nutzen zu können.
- ❑ Da alle Teile der graphischen Benutzungsschnittstelle einer Anwendung ein ähnliches Aussehen besitzen sollen, sind Absprachen 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:

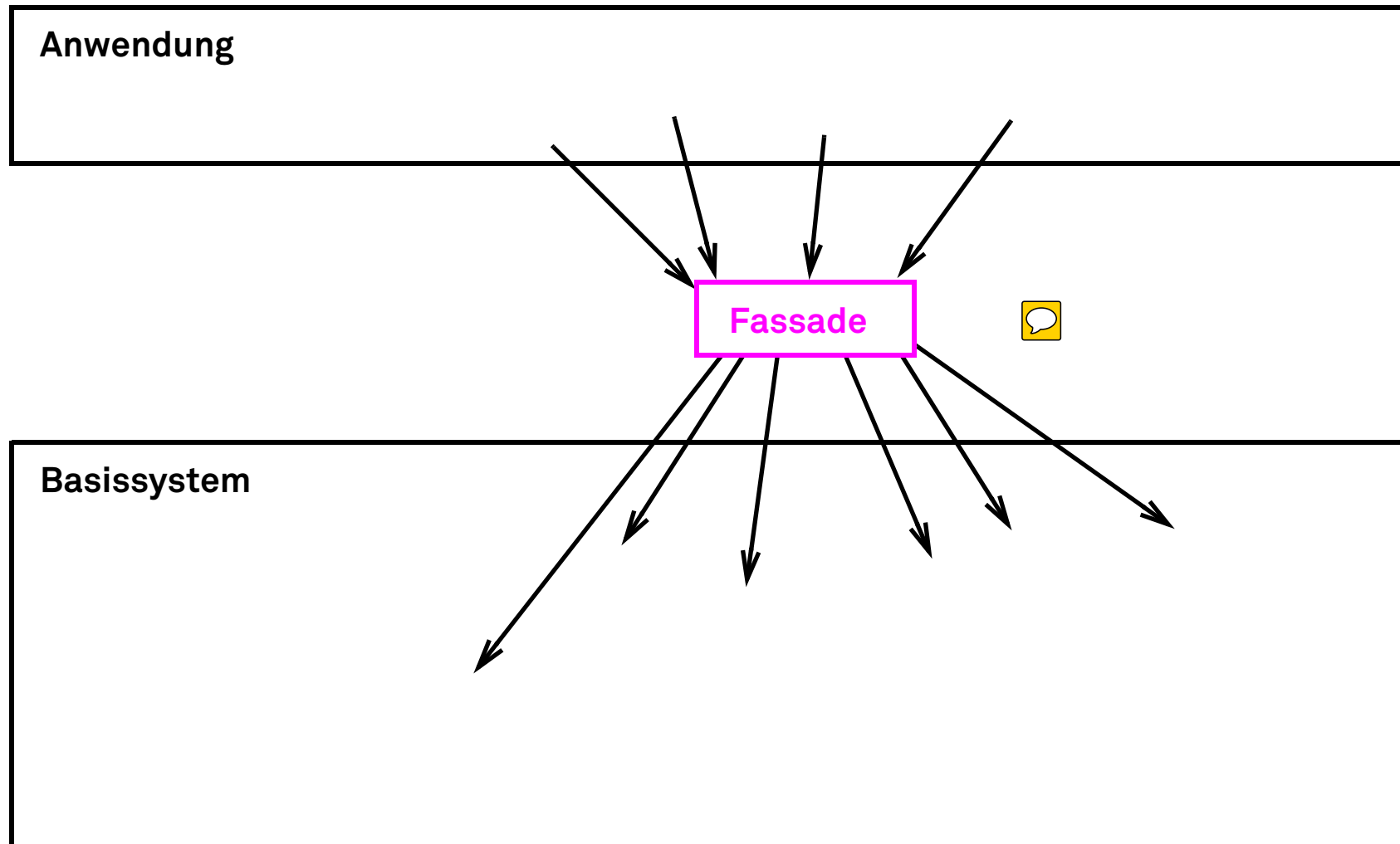
Einführung einer (kompakten) Zwischenschicht, die

- ❑ 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



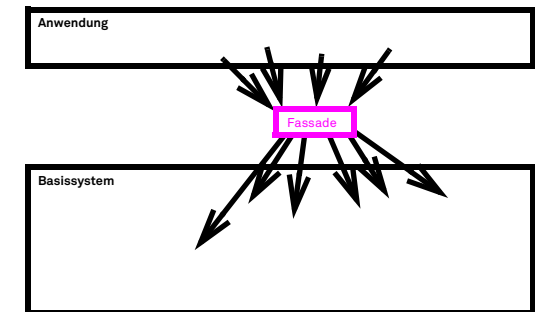
Entwurfsmuster Fassade: allgemeine Darstellung



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.

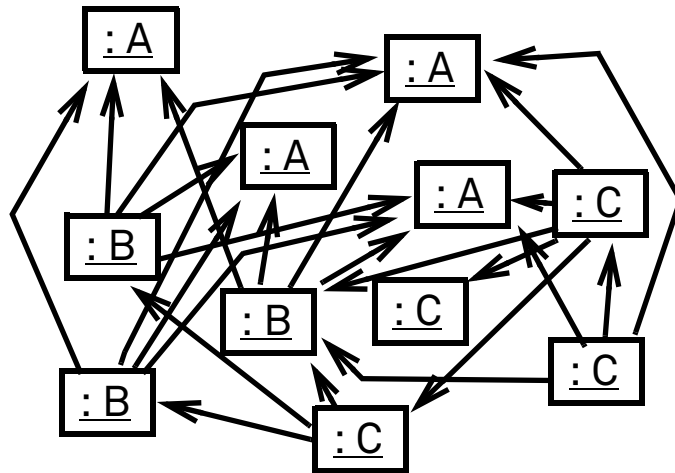
Anmerkungen:

- ❑ 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

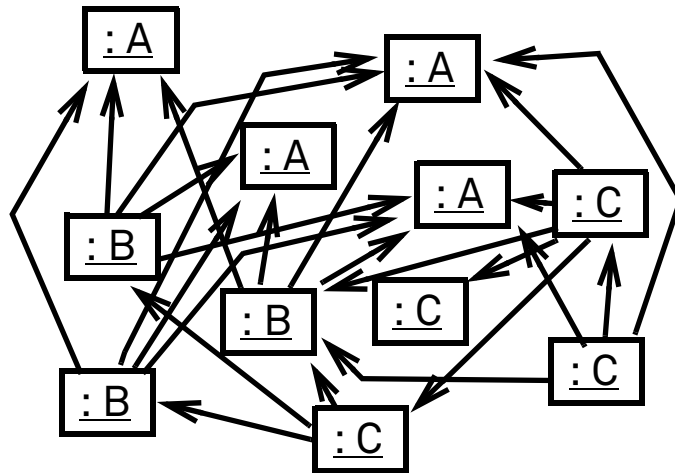


Objektdiagramm

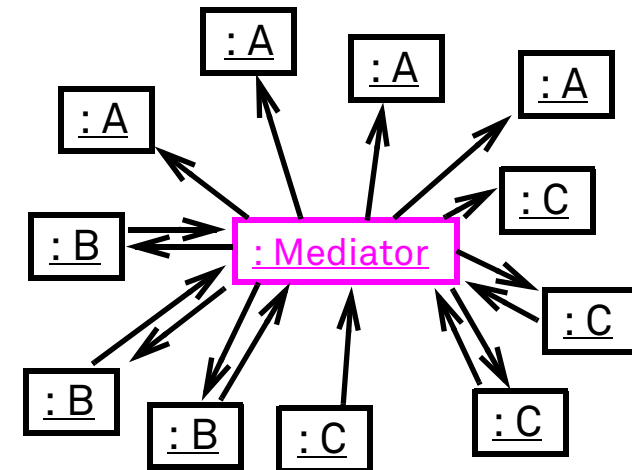
Situation für den Einsatz des Entwurfsmusters Mediator

(Fortsetzung)

Ausgangssituation

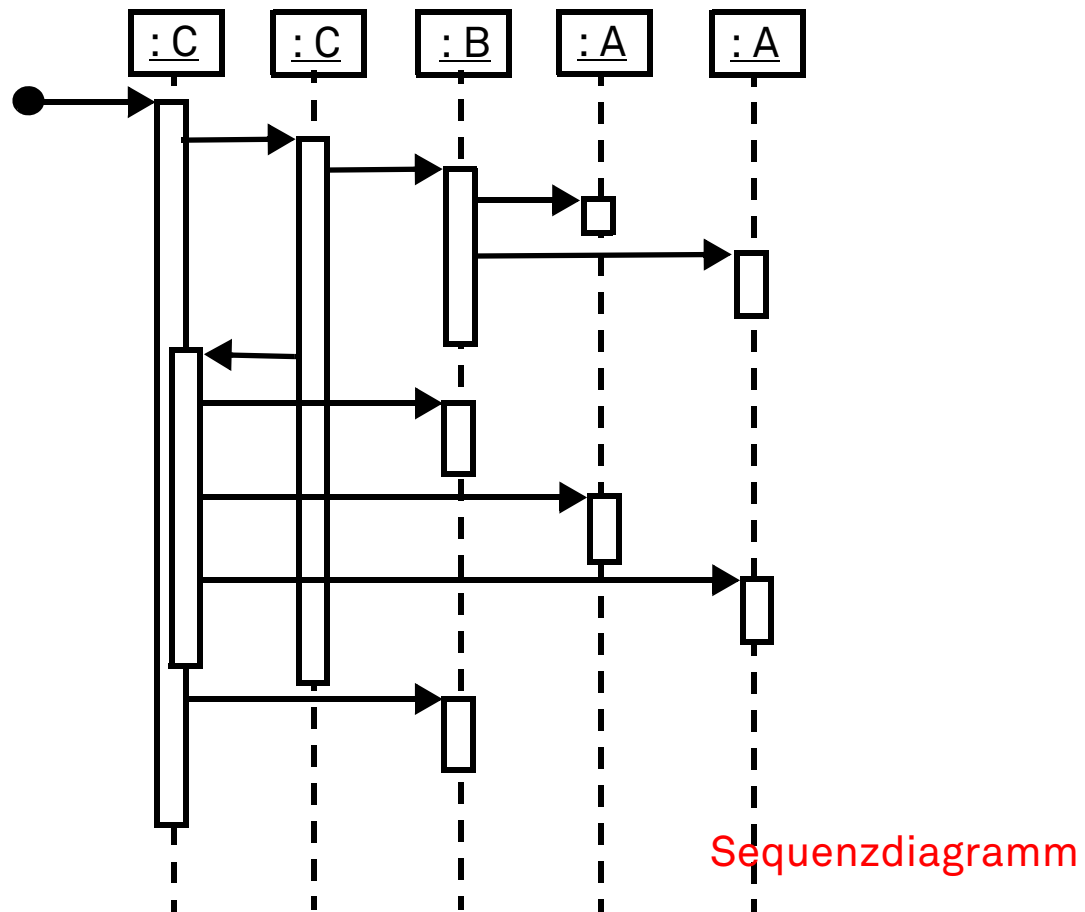


geänderte Struktur durch Einsatz des Mediators



Objektdiagramm

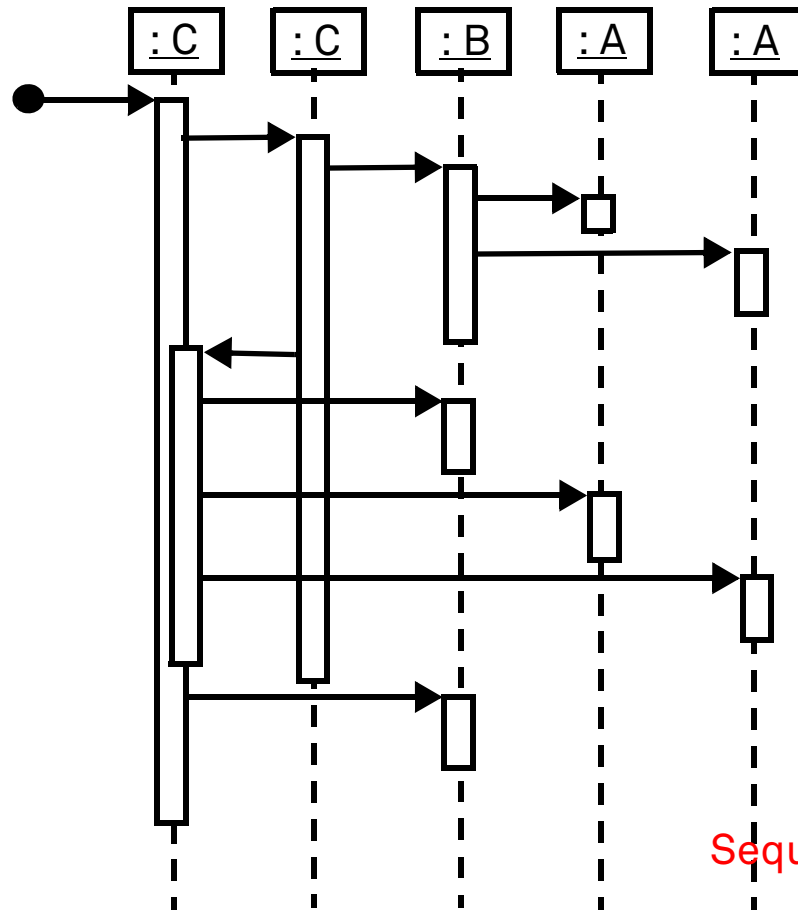
Abläufe des Entwurfsmusters Mediator Ausgangssituation



Abläufe des Entwurfsmusters Mediator

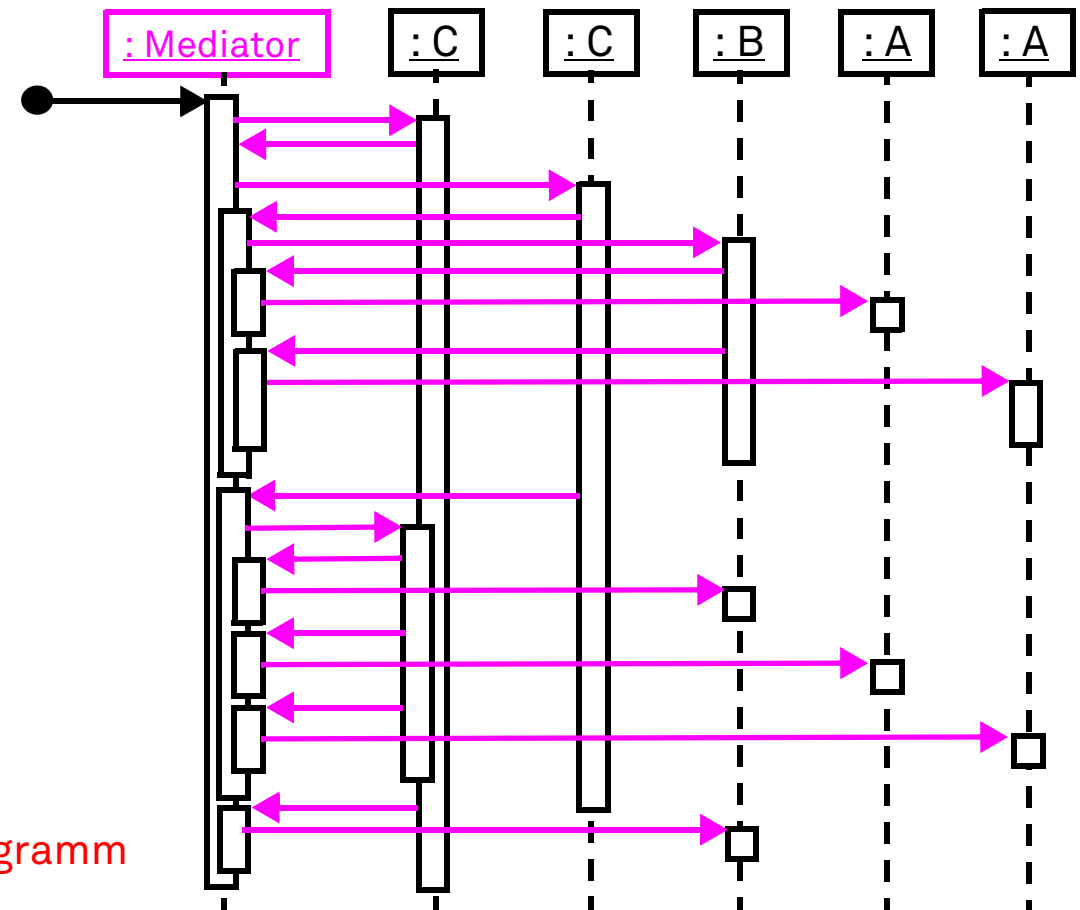
(Fortsetzung)

Ausgangssituation

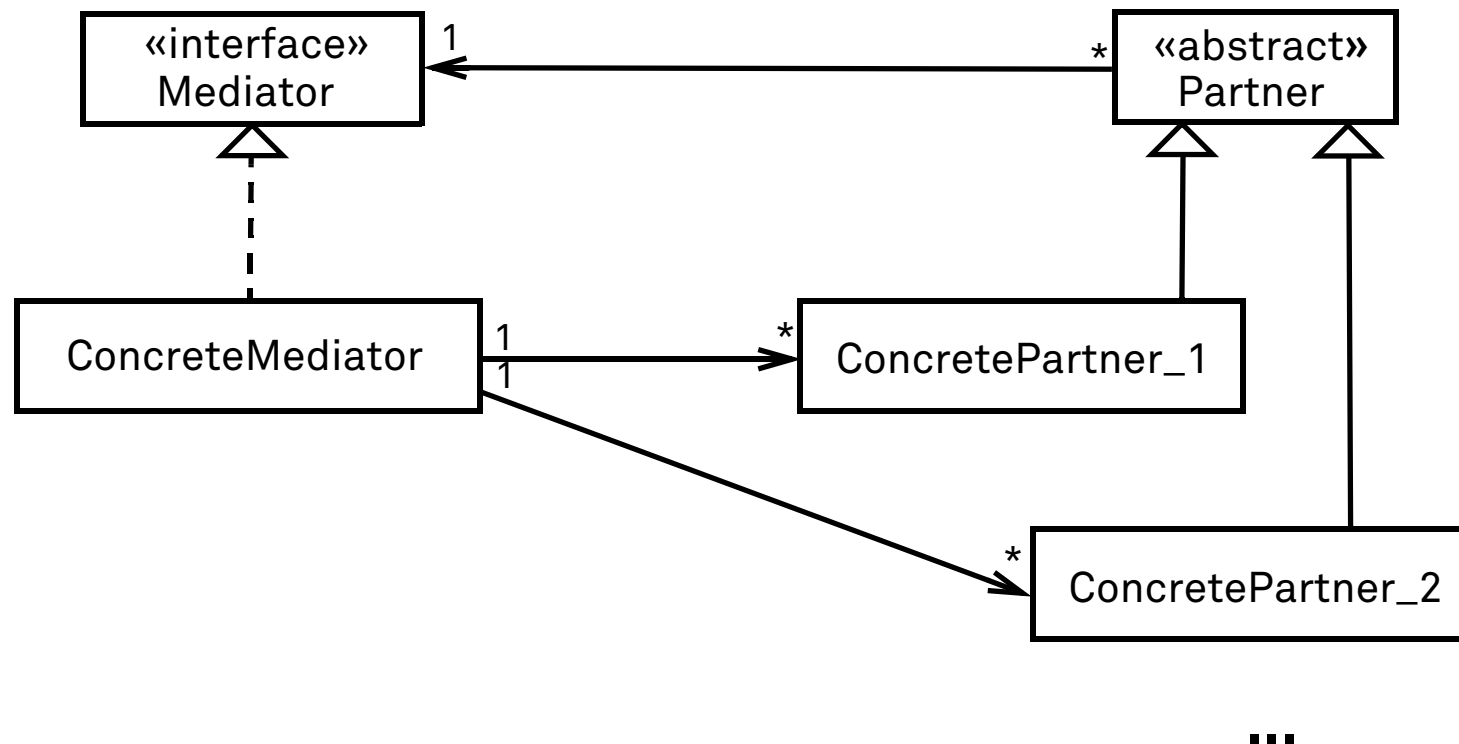


Sequenzdiagramm

geänderte Struktur durch Einsatz des Mediators



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:

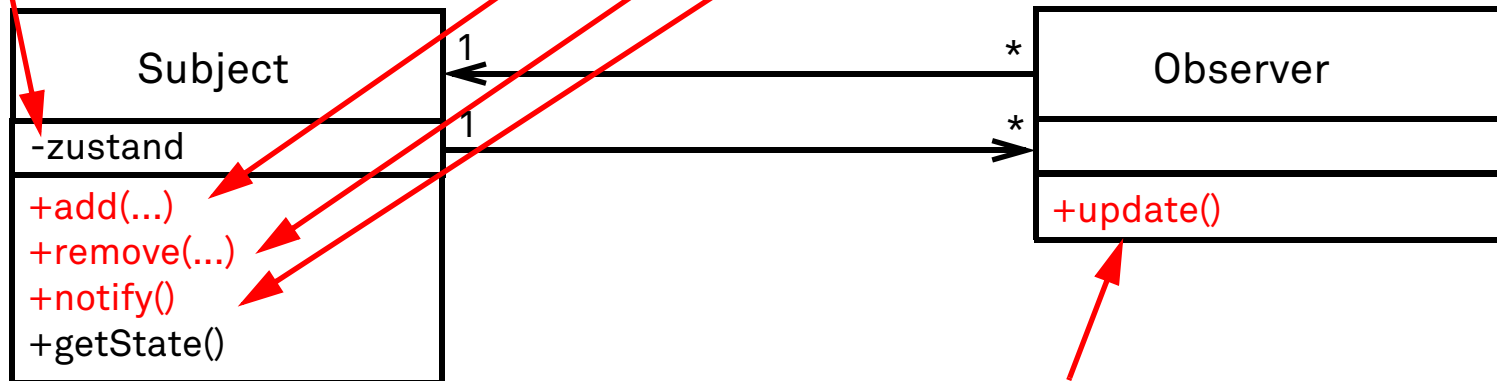
- ❑ Objekte in der Programmwelt kooperieren zuverlässig.
- ❑ Im Beobachter-Muster kooperieren das (beobachtete) Subjekt und der Beobachter. Entsprechungen in der realen Welt wären z.B.:
 - Ein Autohersteller versendet Prospekte zum neuen Modell.
 - Der Kaufhausdieb informiert den Detektiv über seinen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

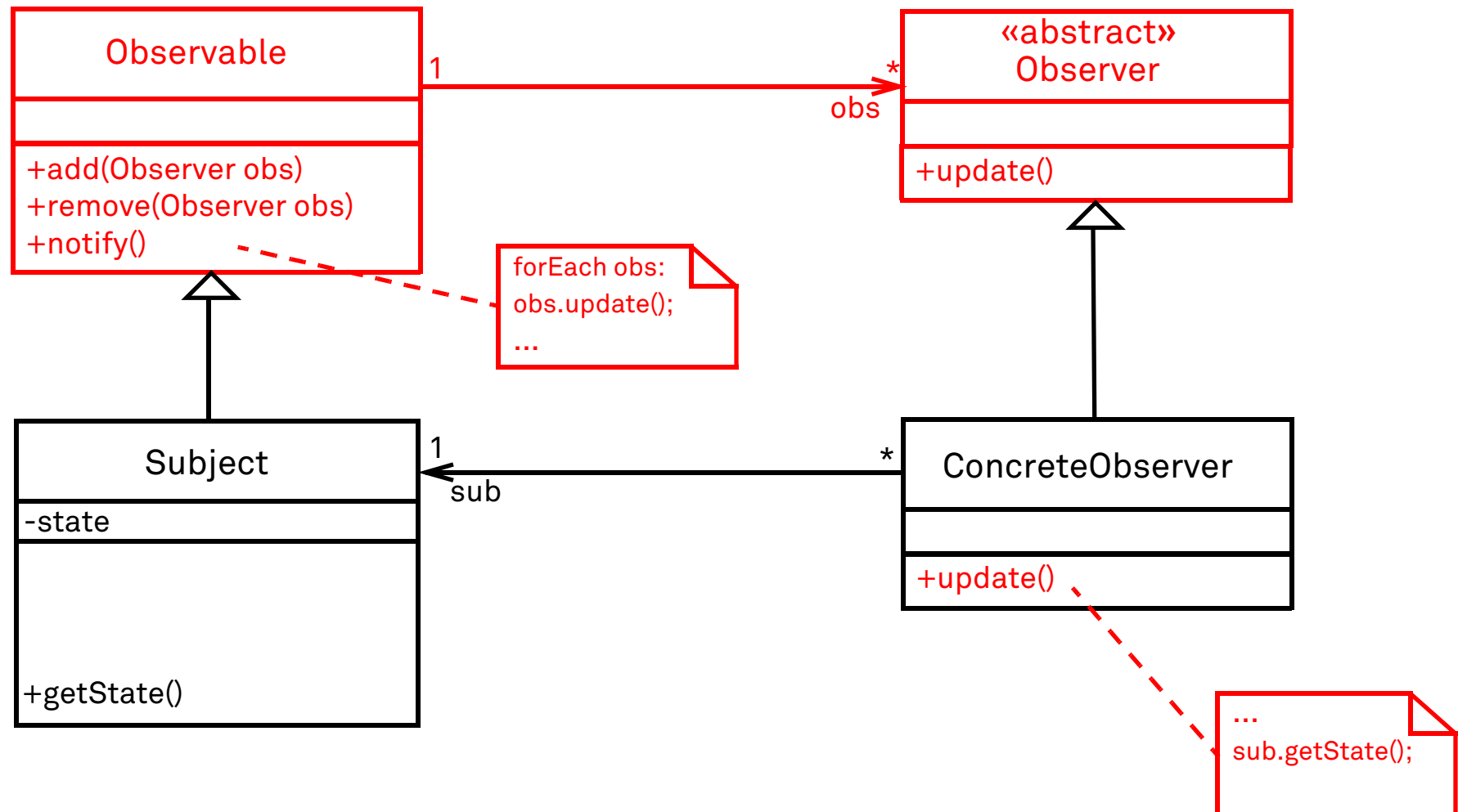
Objekt hat einen
beobachtbaren Zustand



Struktur des Entwurfsmusters Beobachter

allgemeiner Aufbau

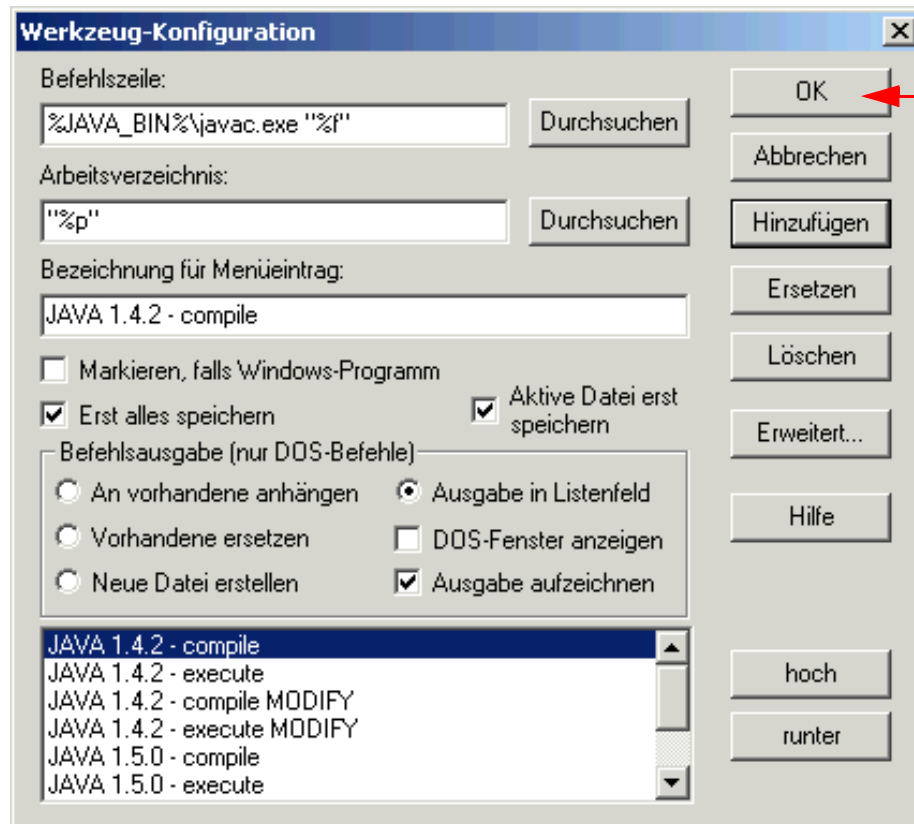
(Fortsetzung)



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

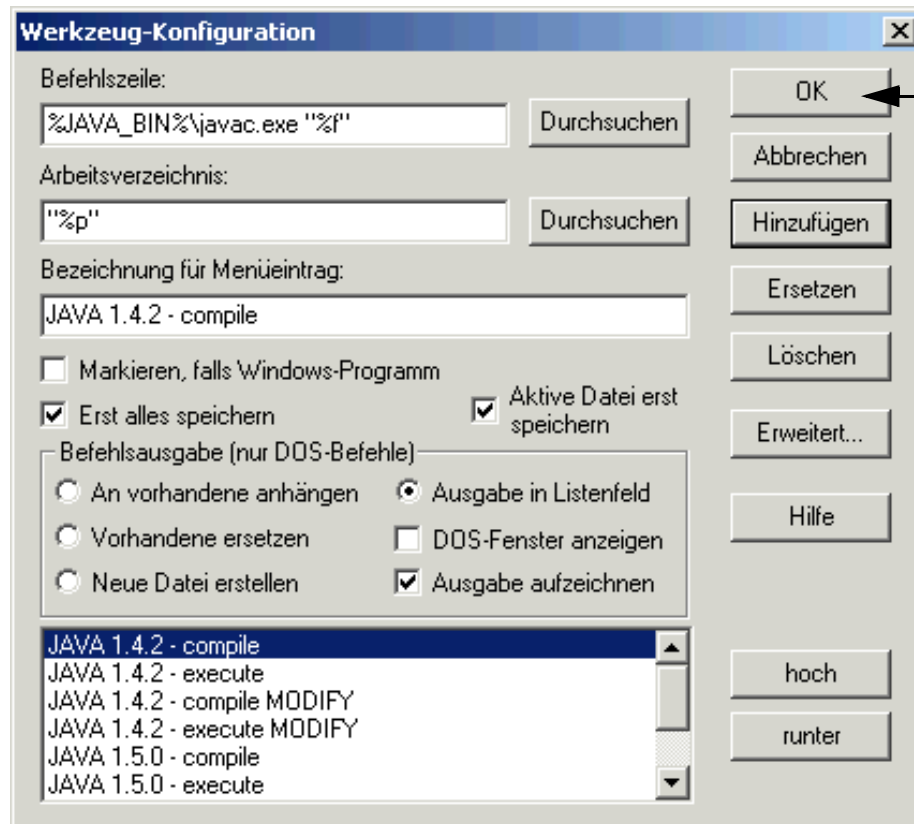


`JButton ok = new JButton("OK");`

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:

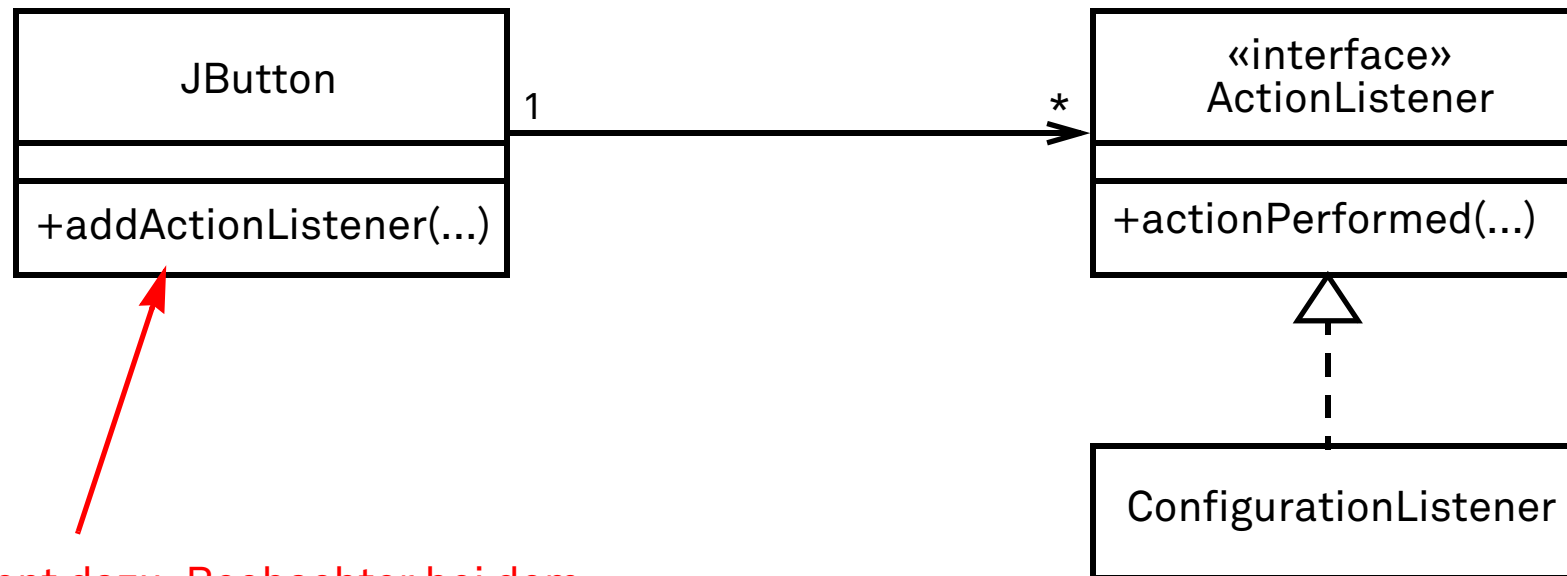
```
class ConfigurationListener
    implements ActionListener {
    ...
}
```

ActionListener ist ein Beobachter für ein JButton-Objekt!

Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing

(Fortsetzung)

(Darstellung als Klassendiagramm)

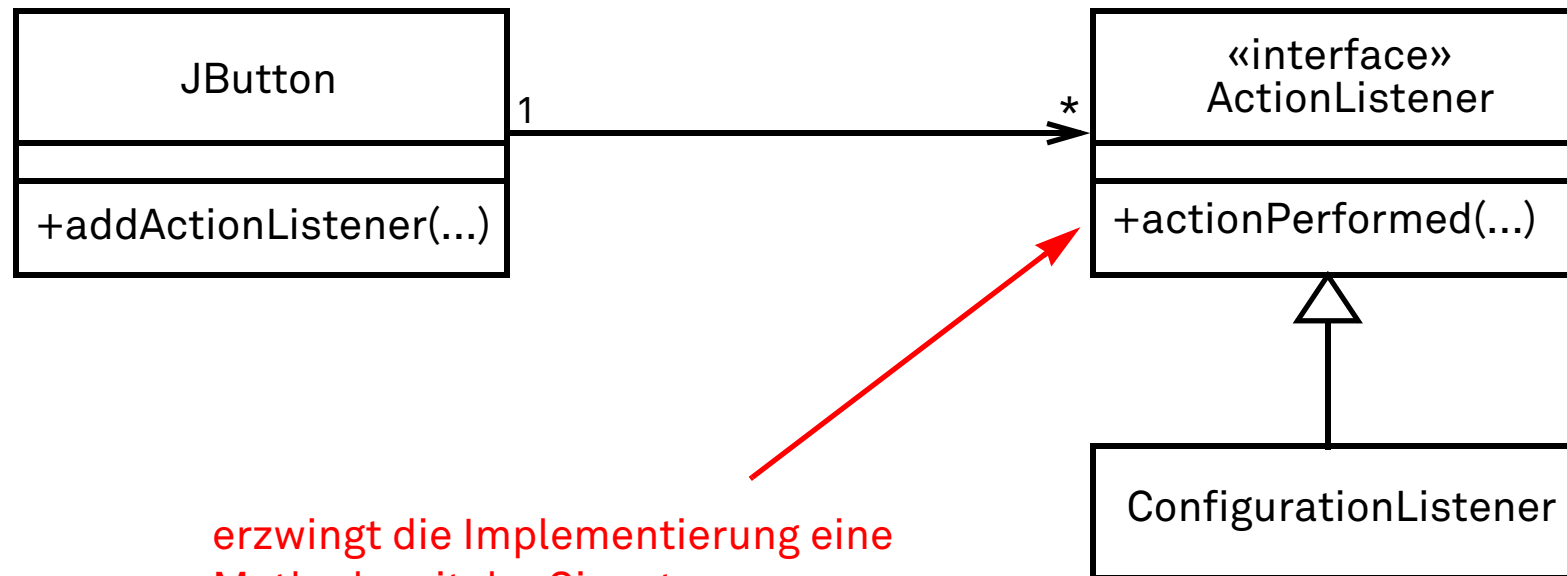


dient dazu, Beobachter bei dem
Button ok anzumelden, im Beispiel:

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

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

(Fortsetzung)

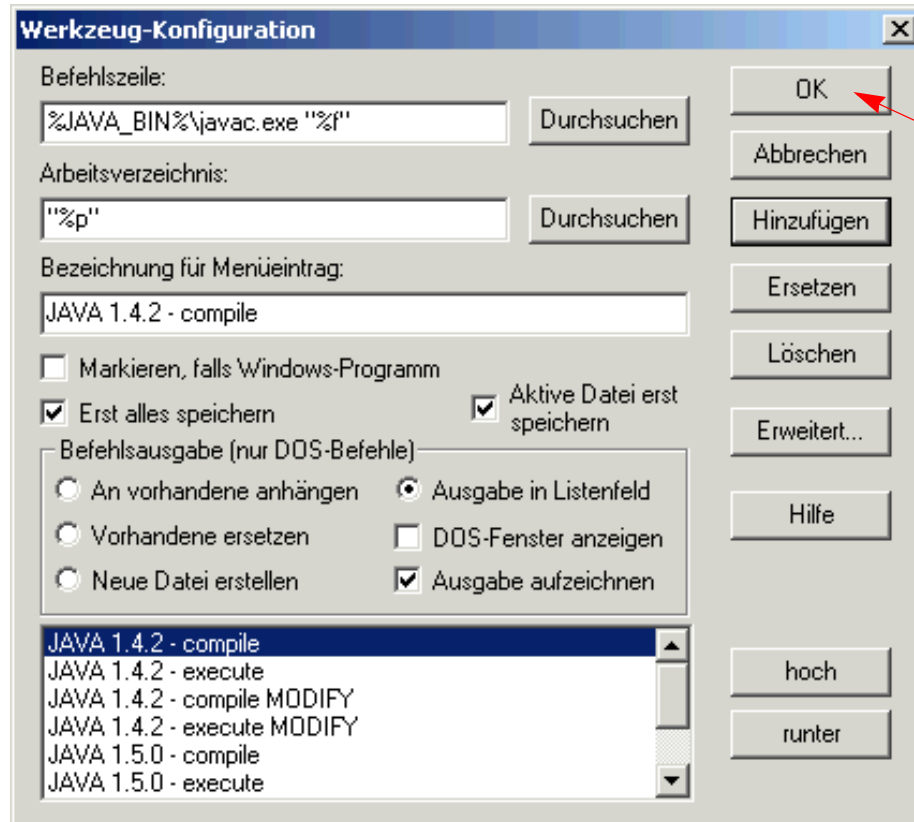


erzwingt die Implementierung eine
Methode mit der Signatur

`actionPerformed(...);`

Einsatz des Entwurfsmusters Beobachter in Java-Bibliothek Swing

(Fortsetzung)



Ablauf:

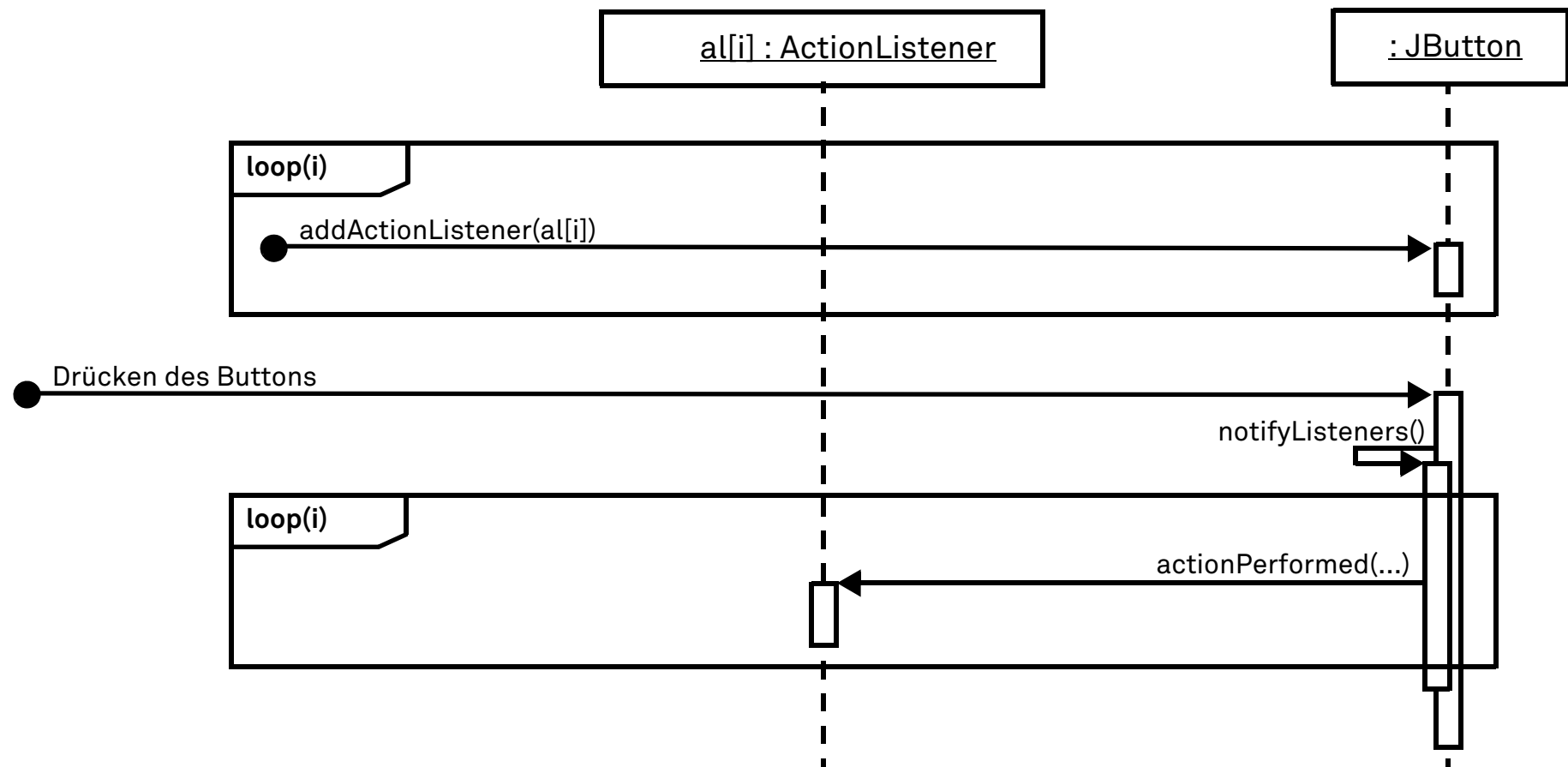
Wird der OK-Button gedrückt,
so wird bei allen Beobachtern die Methode
`actionPerformed` ausgeführt.

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

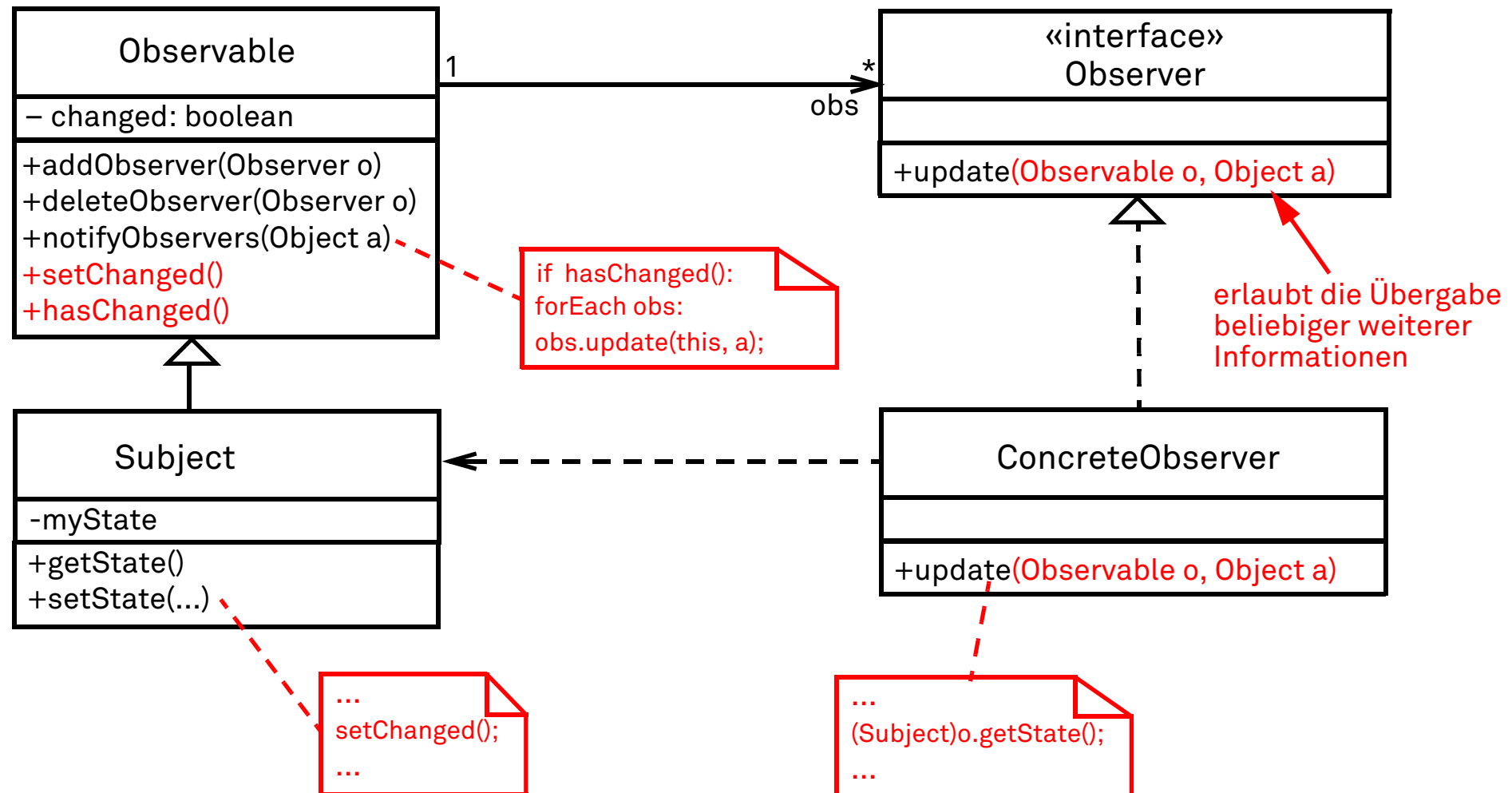
- ❑ Die Implementierung des Beobachter-Musters erfordert nur wenig Aufwand, da wesentliche Abläufe in vordefinierten Klassen vorgegeben werden können: Anmeldung, Abmeldung, Benachrichtigung
- ❑ Während der Ausführung können dann beliebig viele Beobachter mit Informationen versorgt werden.

- ❑ 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:

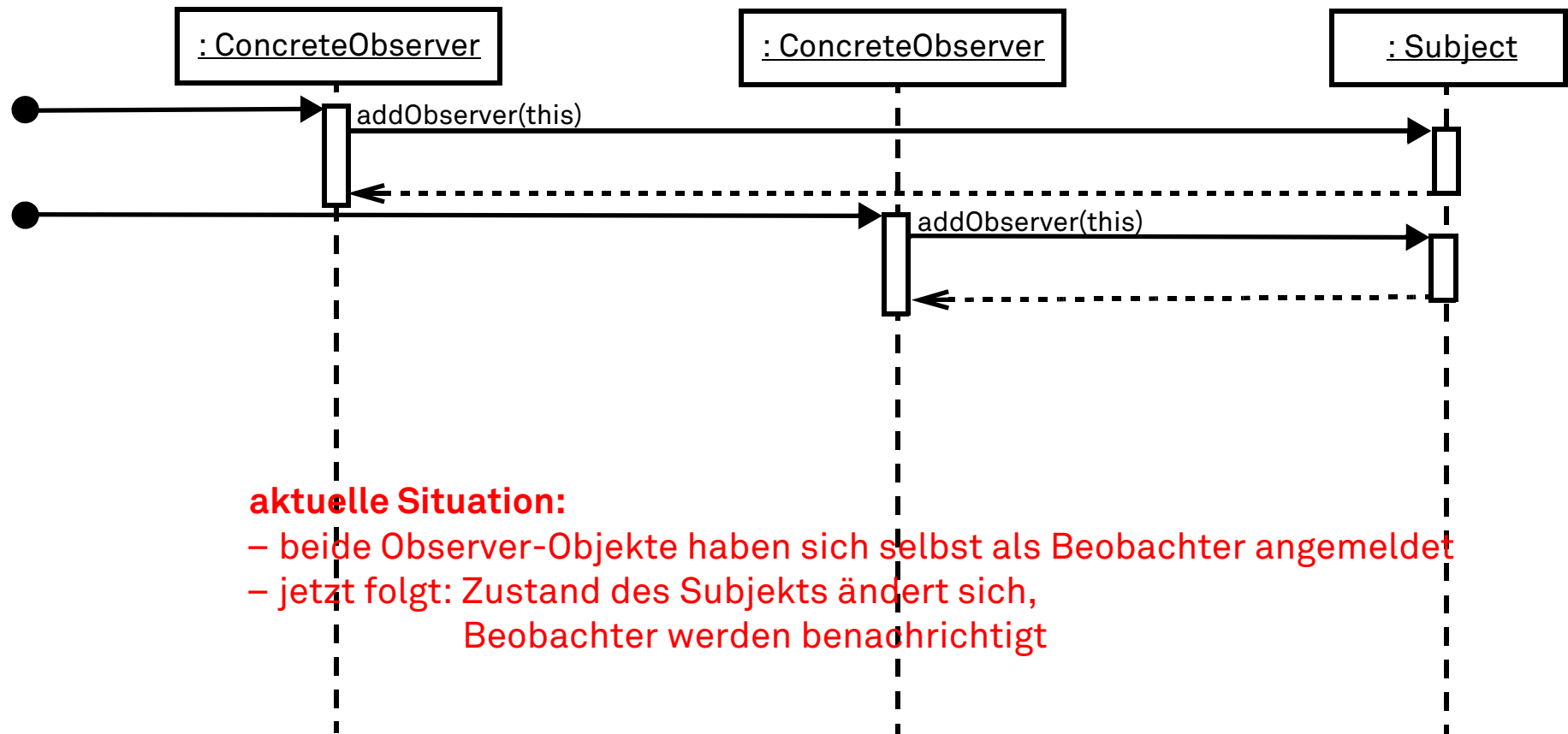
- ❑ `addObserver` meldet einen Beobachter an, der das Interface `Observer` realisiert
- ❑ `deleteObserver` meldet einen Beobachter wieder ab
- ❑ `setChanged` setzt `changed`-Attribut, das anzeigt, ob das Subjekt geändert wurde
- ❑ `hasChanged` liefert den Wert des `changed`-Attributs
- ❑ `notifyObservers` 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!

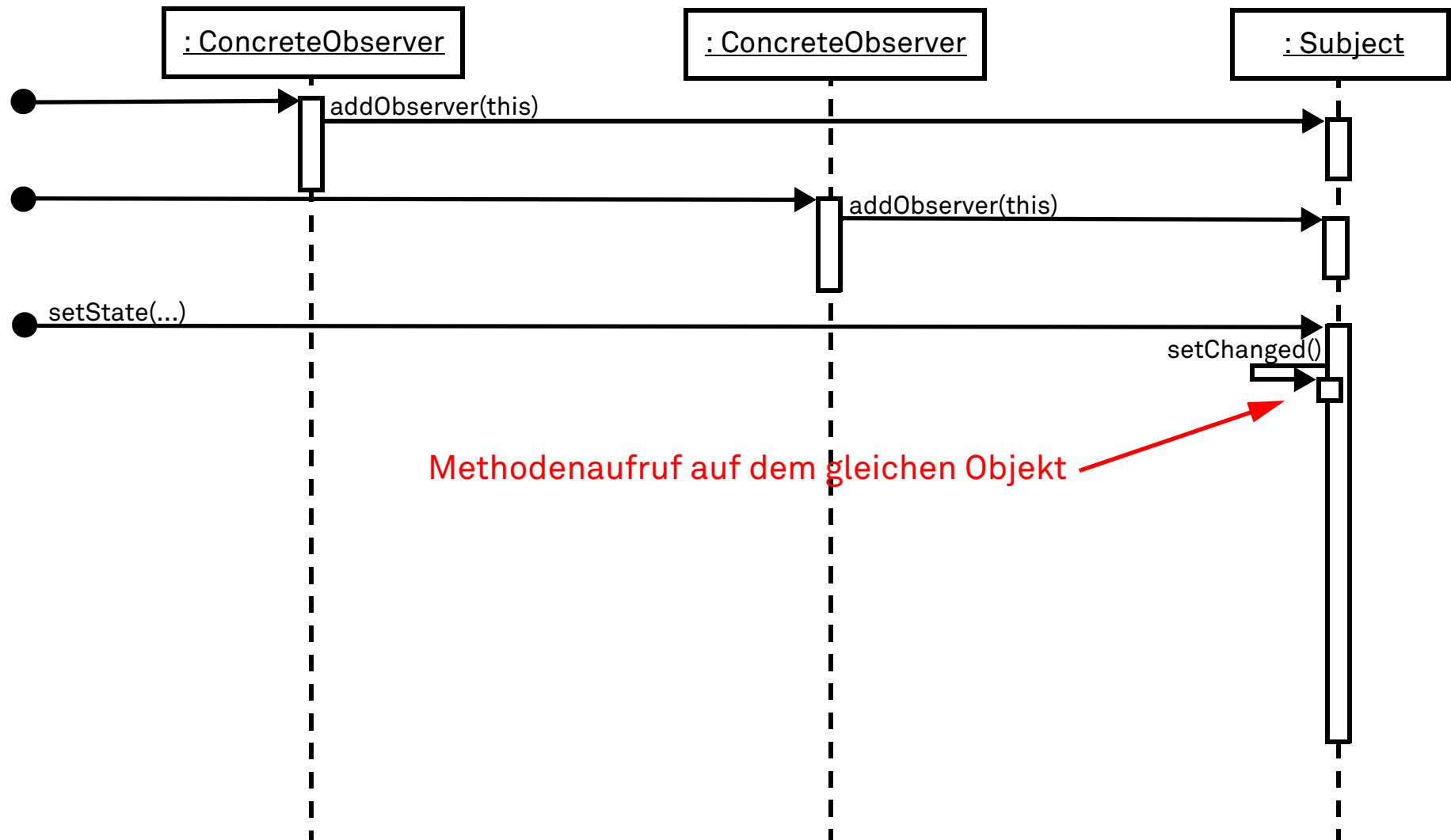
⇒ **der Ablauf wird jetzt etwas komplexer, seine Beschreibung auch!**
(Das Beobachter-Muster ist ein Verhaltensmuster!)

Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter



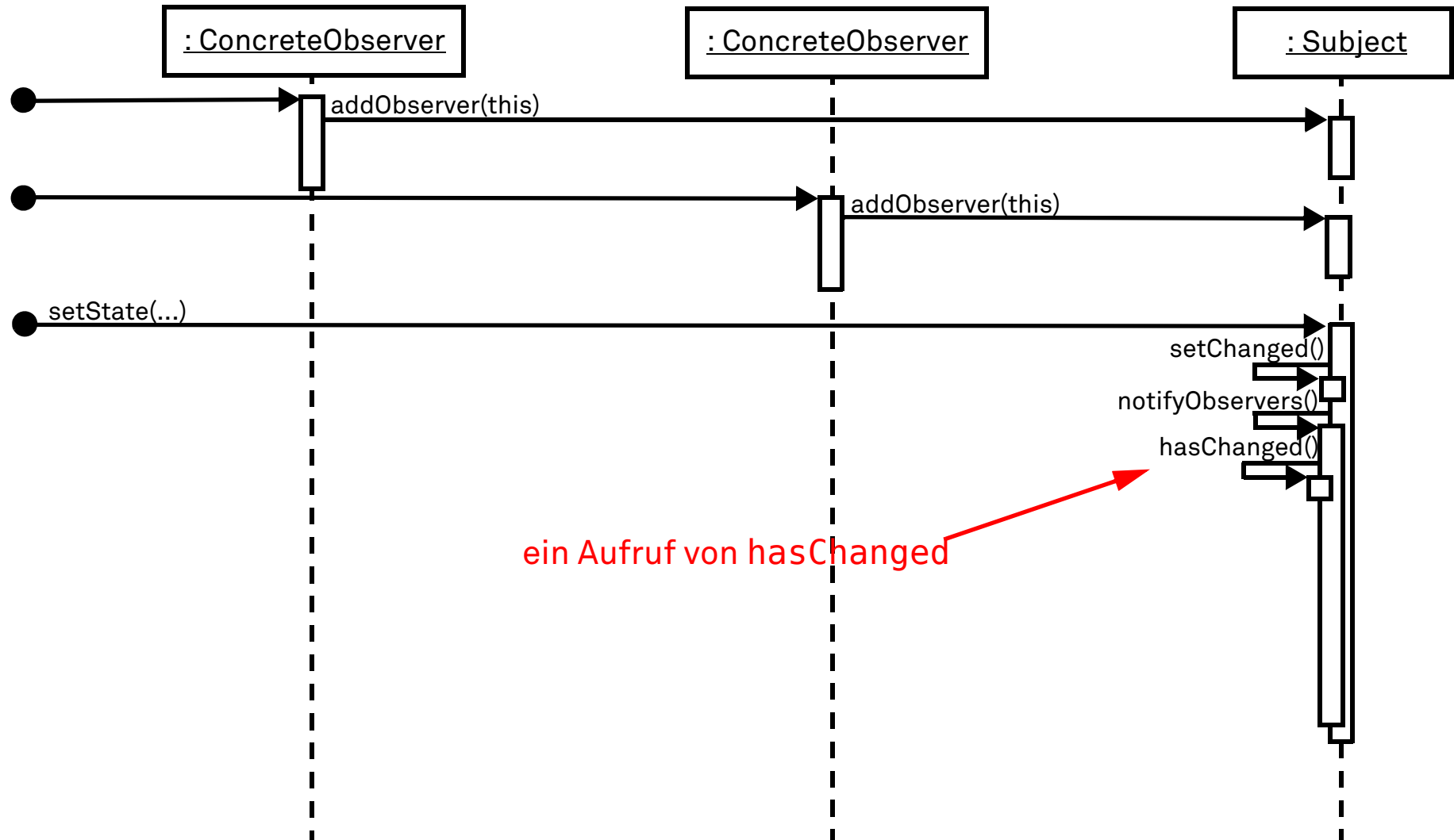
Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



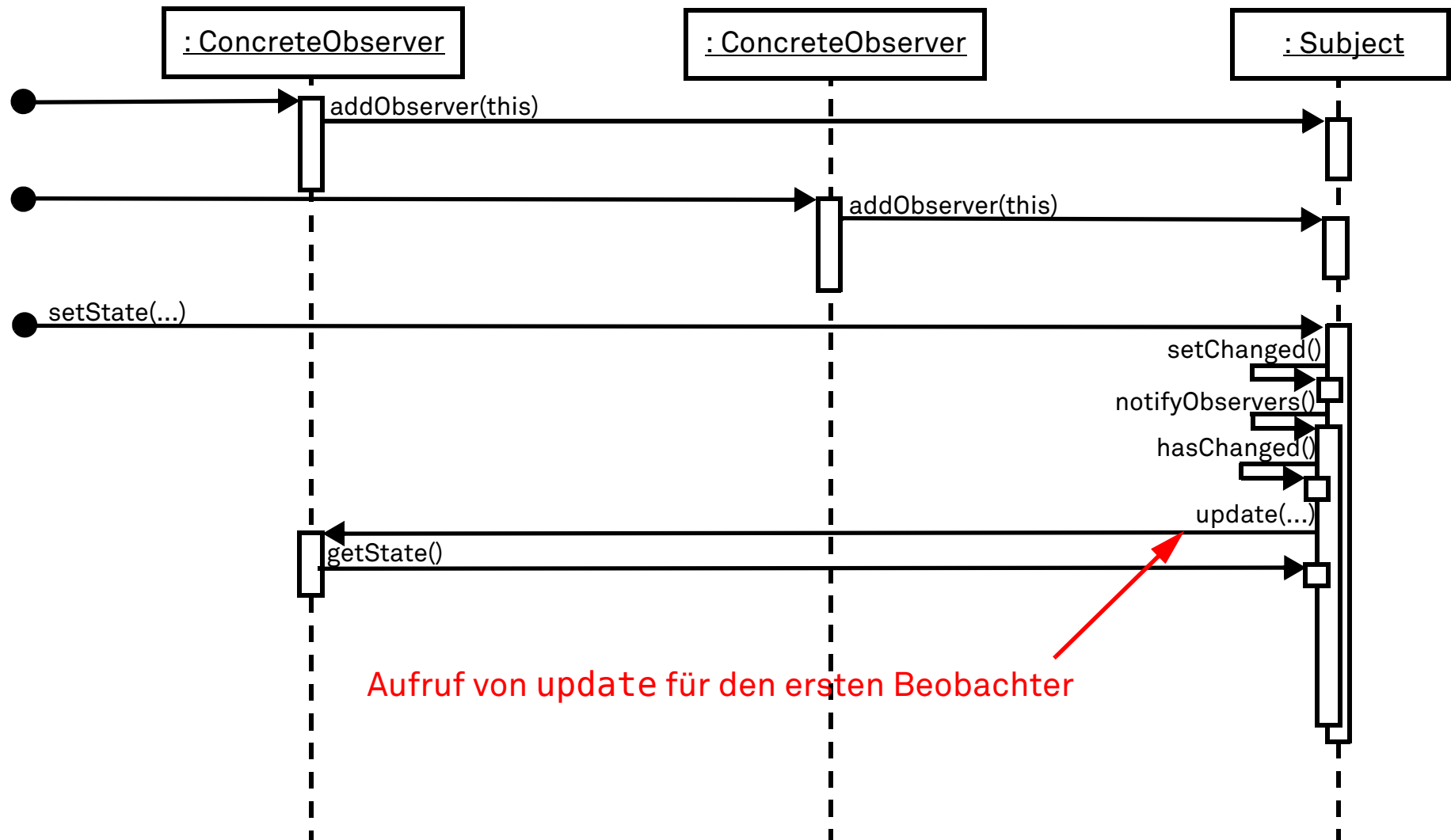
Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



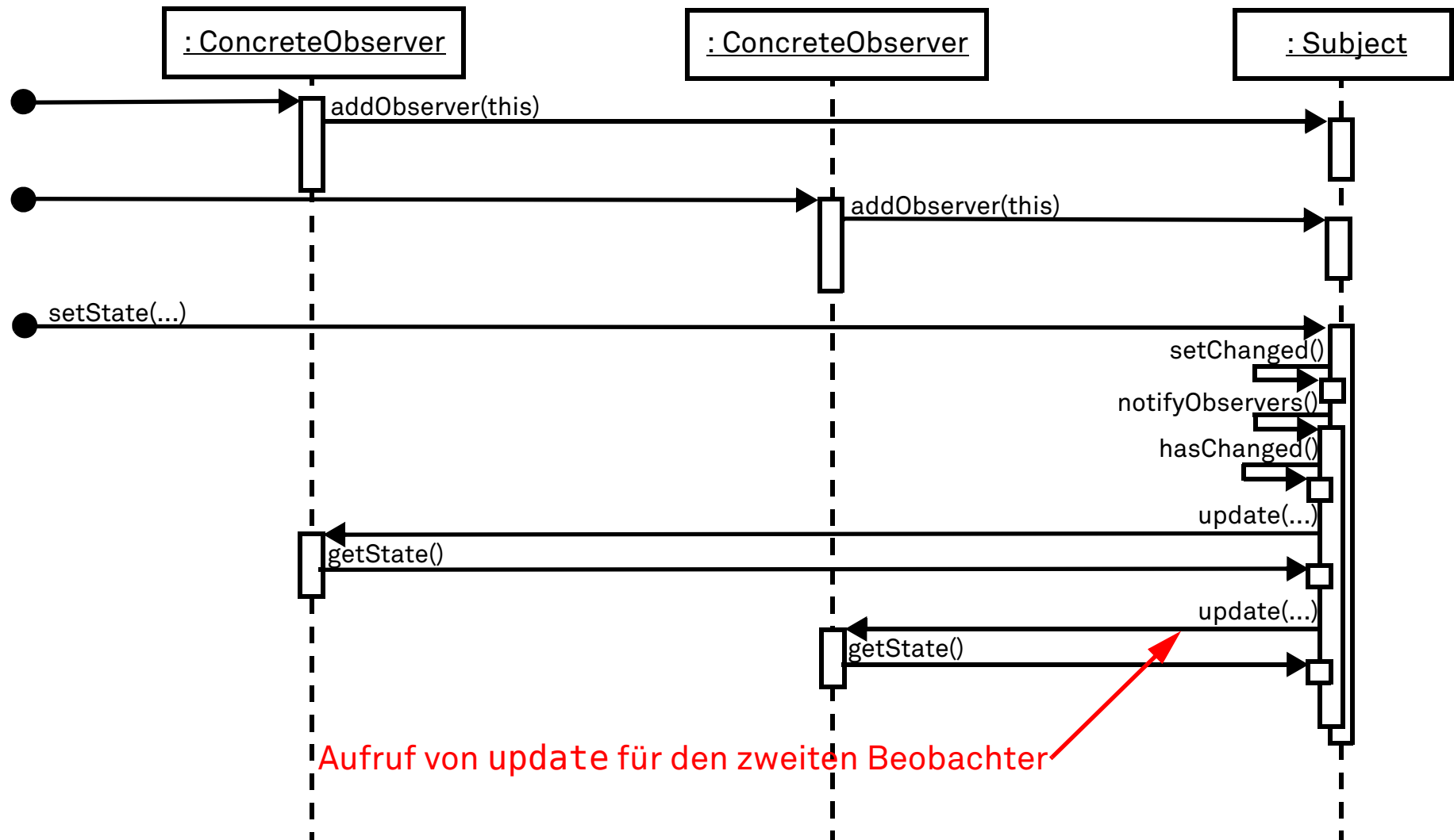
Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



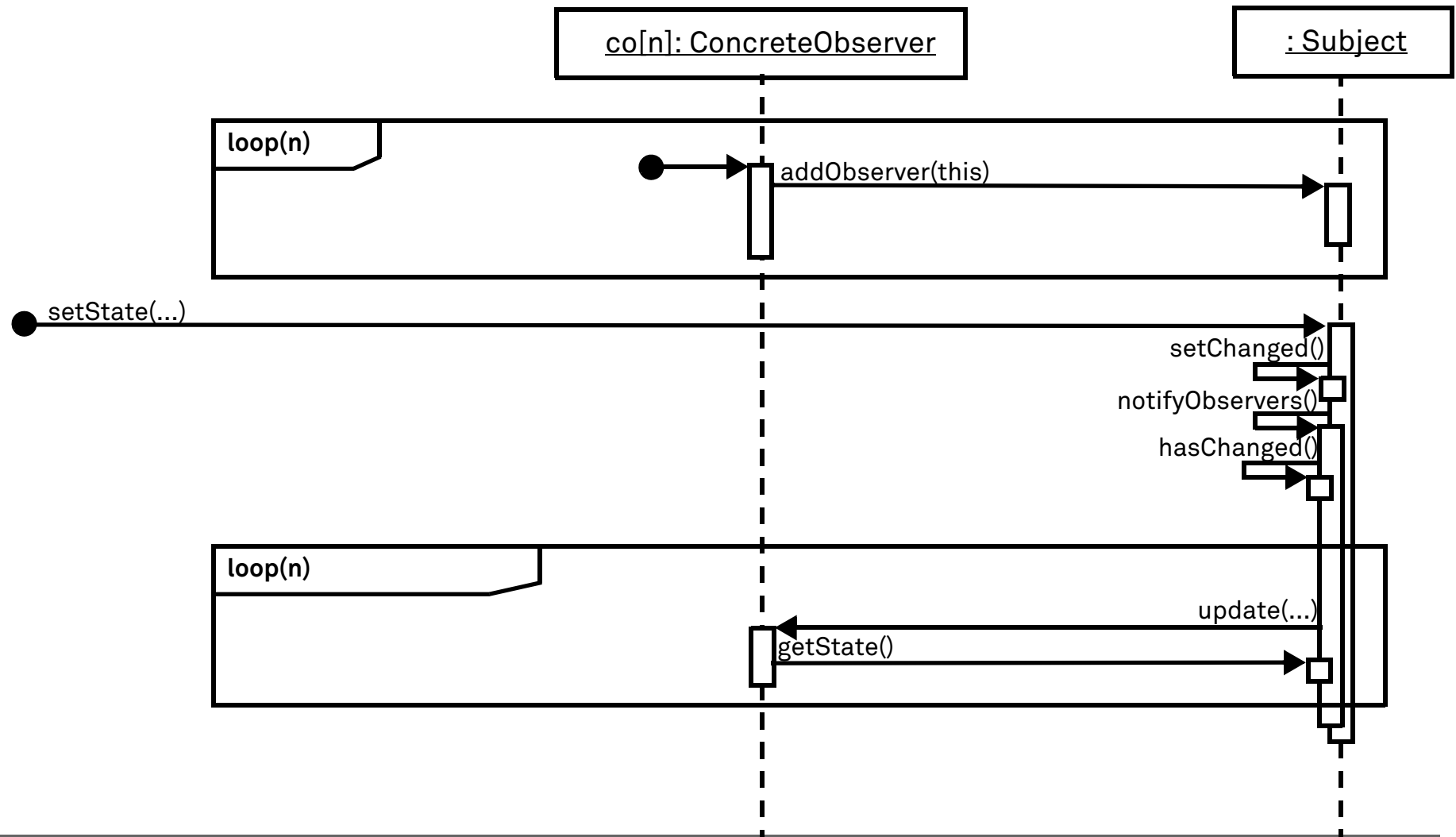
Sequenzdiagramm zur Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



allgemeine Beschreibung des Ablaufs beim Beobachter

(Fortsetzung)



Entwurfsmuster Beobachter: Bewertung der Realisierung in Java

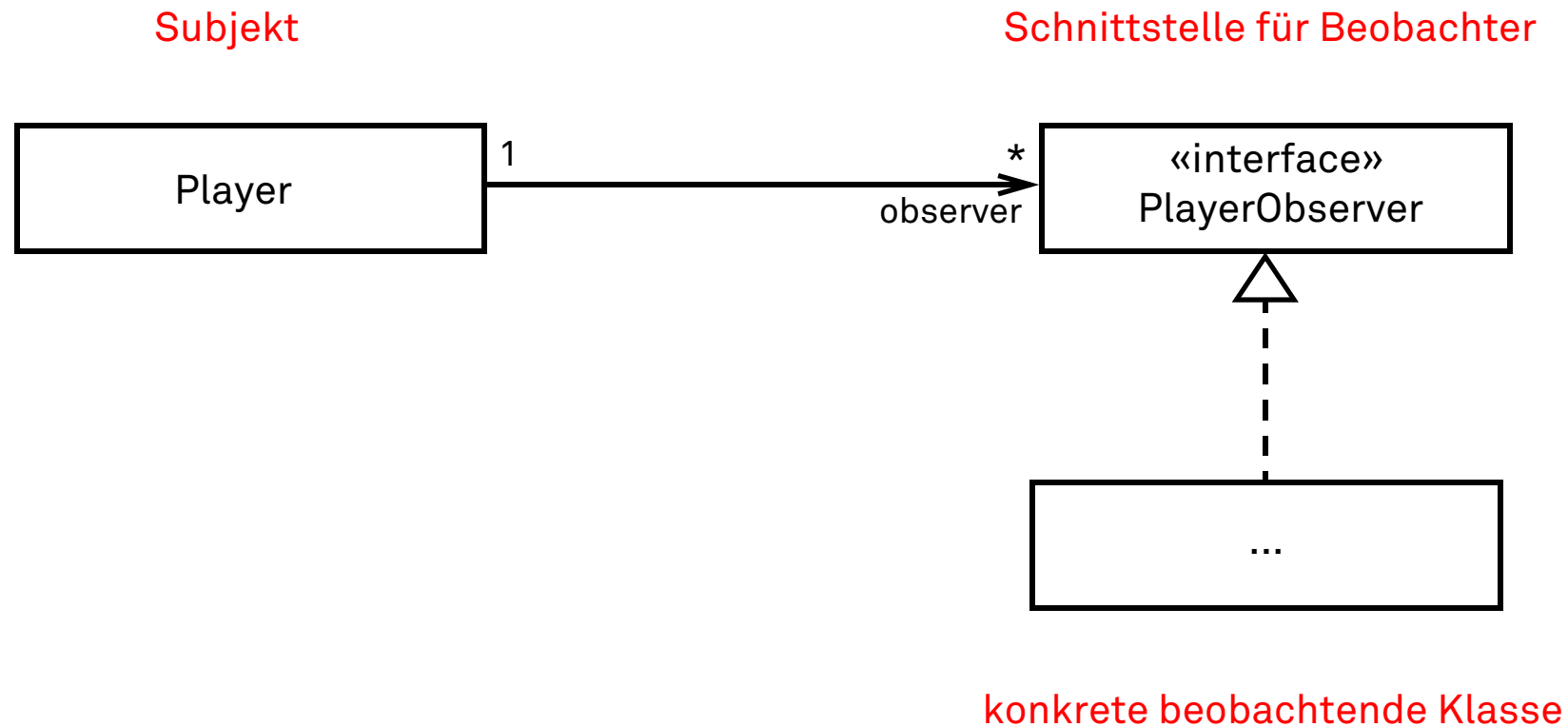
- ❑ Die Realisierung in Java ermöglicht es der konkreten Implementierung, die Abläufe in der Methode `notify` über das Attribut `changed` zu steuern.
- ❑ Eine Klasse, die `Observer` konkretisiert, kann nur eine einzige `update`-Methode implementieren.
- ❑ Soll ein Beobachter an mehreren Umsetzungen des 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.

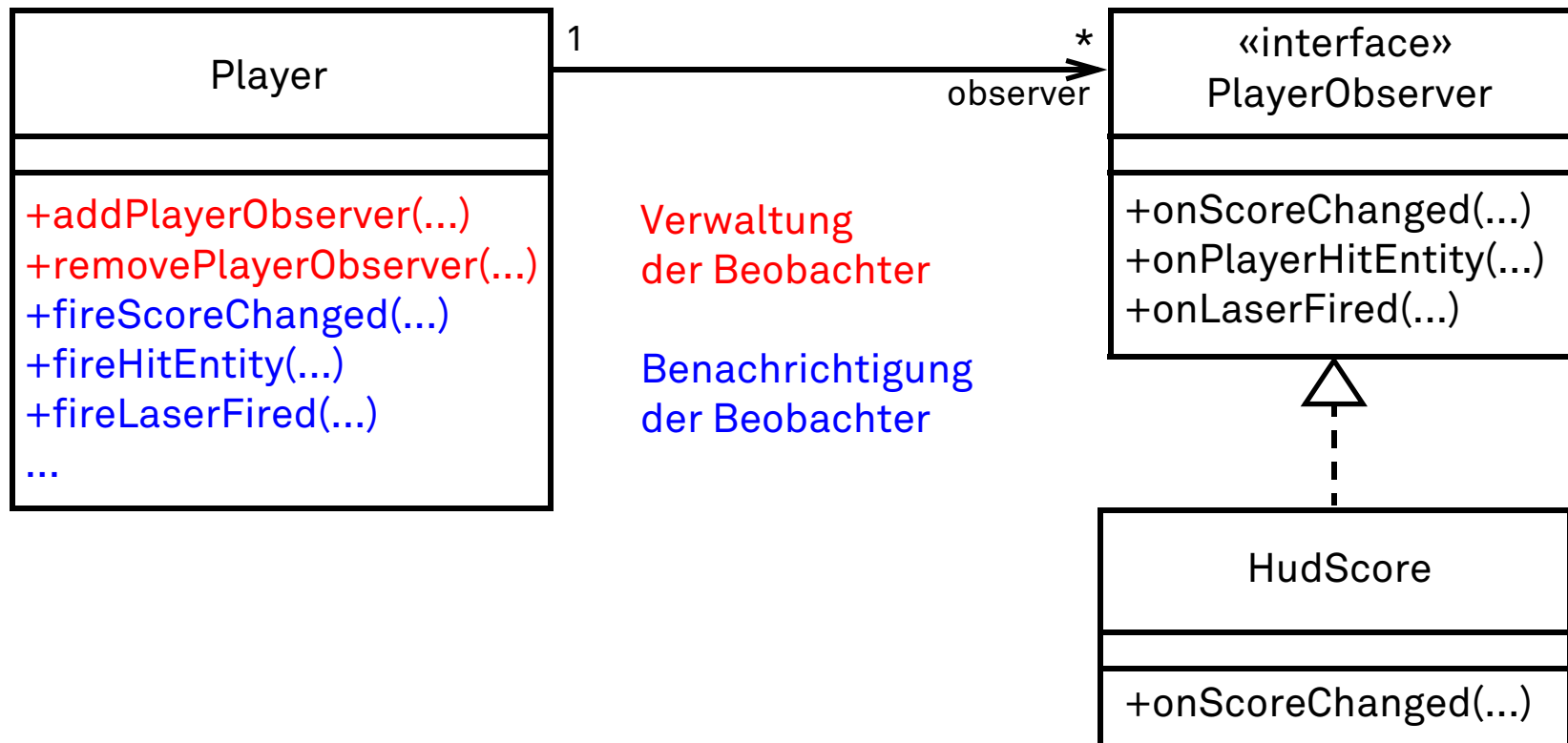
Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

(Fortsetzung)



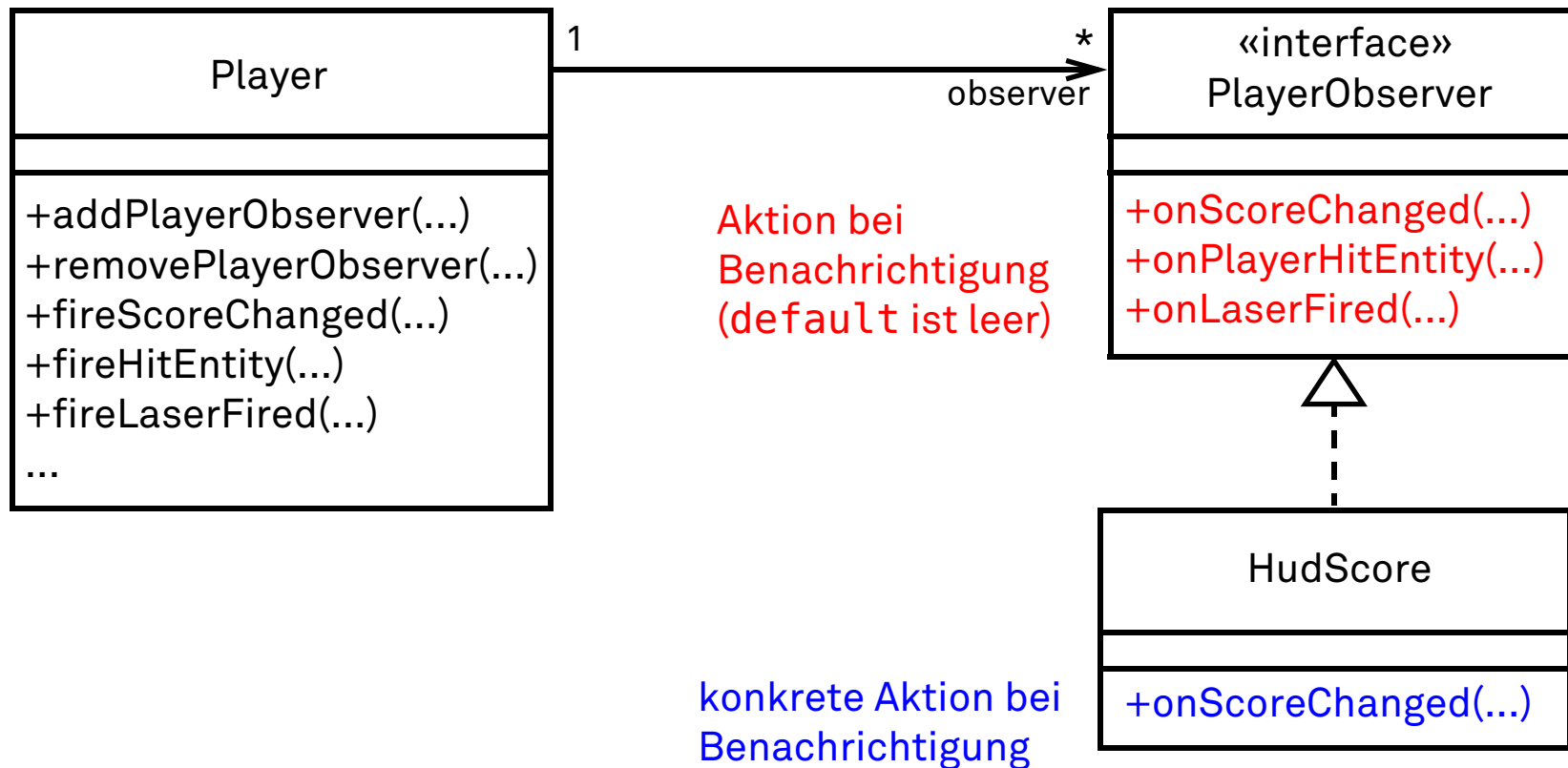
Entwurfsmuster Beobachter im Beispiel SWT-Starfighter

(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.

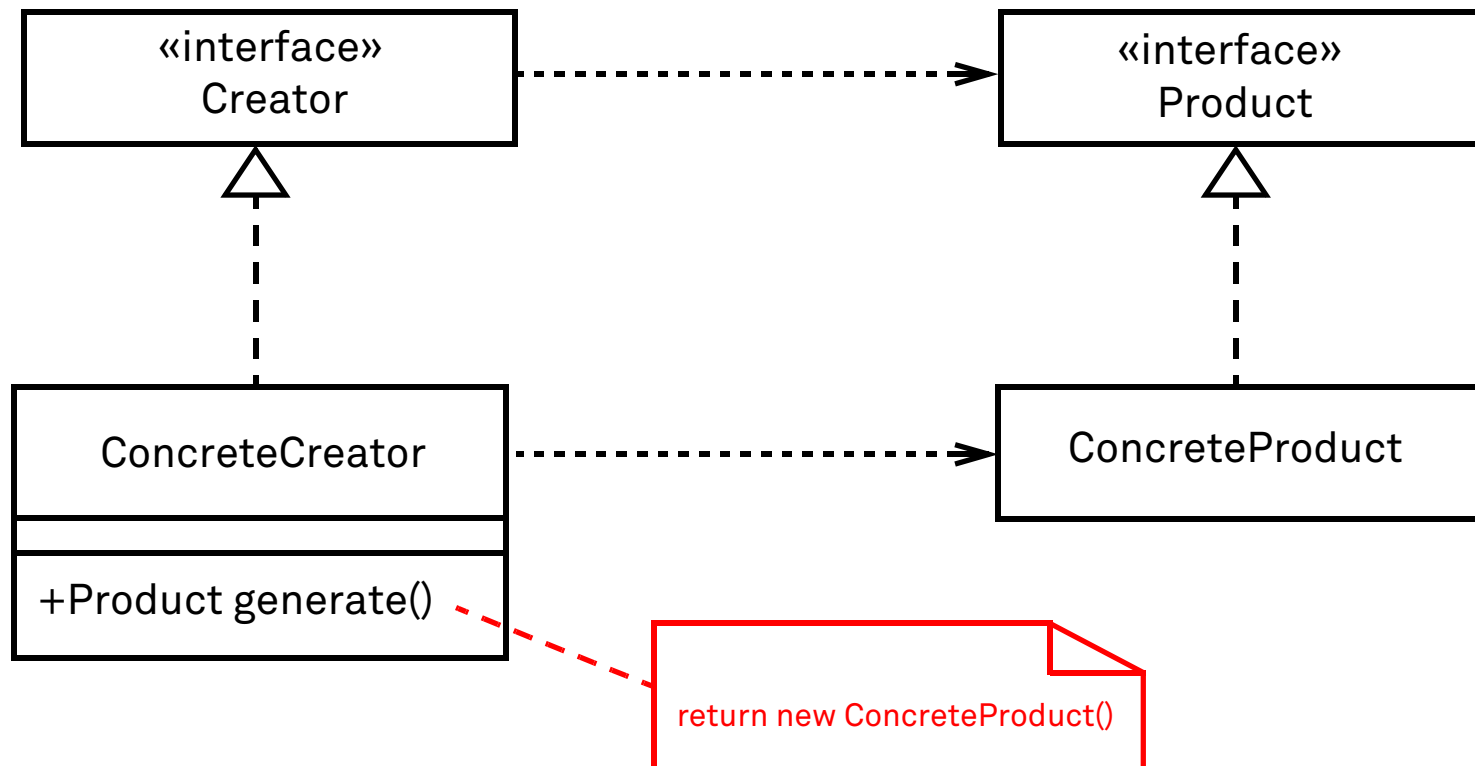
Literatur: Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 31-35
http://link.springer.com/chapter/10.1007/978-3-8274-2526-3_4

Entwurfsmuster Fabrikmethode

(Fortsetzung)

allgemeine Struktur des Entwurfsmusters

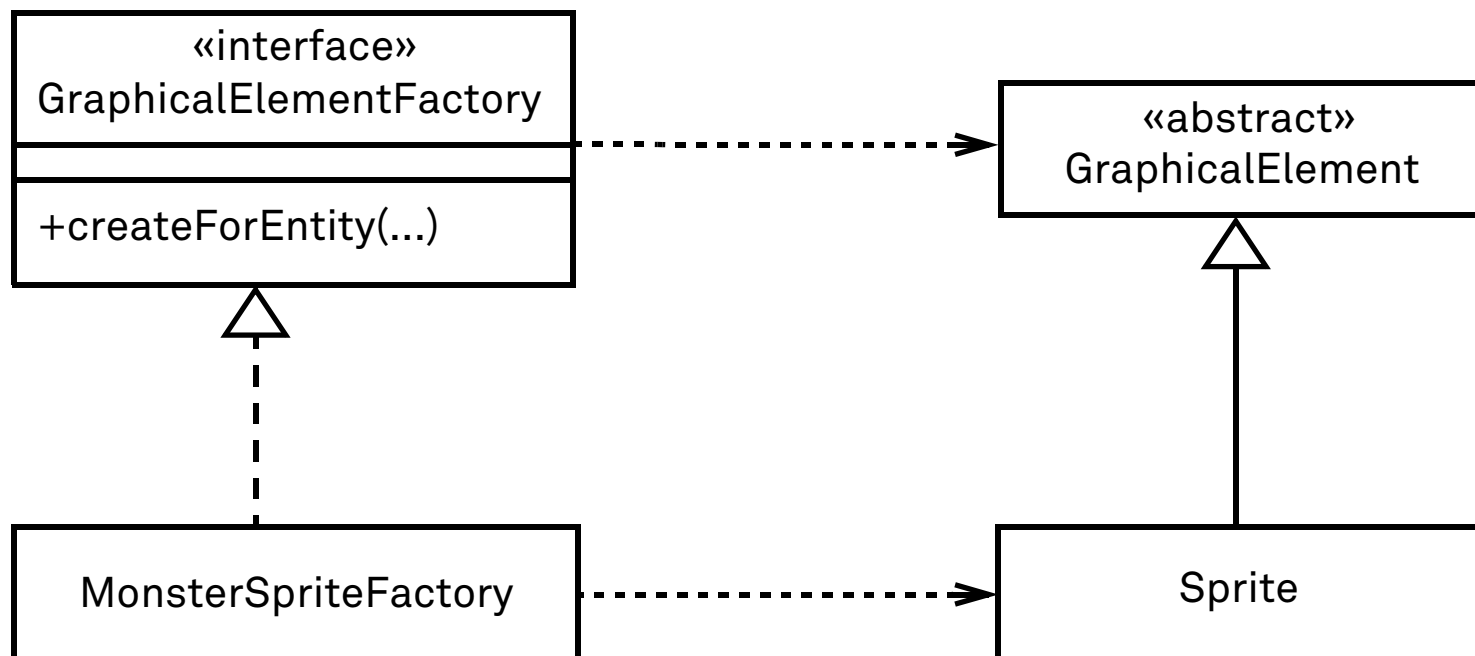
Produkte werden im System
nur über Product referenziert



Entwurfsmuster Fabrikmethode im Beispiel SWT-Starfighter

(Fortsetzung)

Beispiel GraphicalElementFactory



Entwurfsmuster Fabrikmethode im Beispiel SWT-Starfighter

(Fortsetzung)

Beispiel GraphicalElementFactory

Implementierung

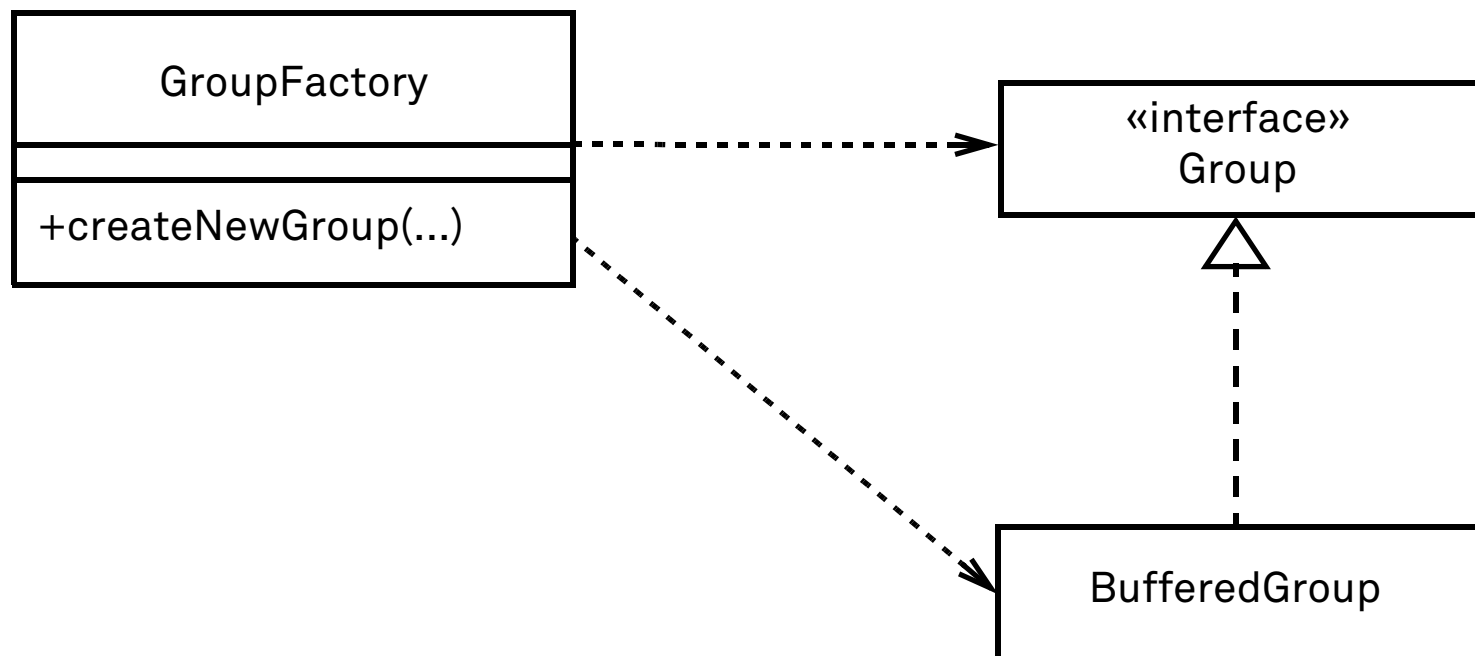
```
public interface GraphicalElementFactory {  
    public GraphicalElement createForEntity(ViewManager v, Entity e);  
}
```

```
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



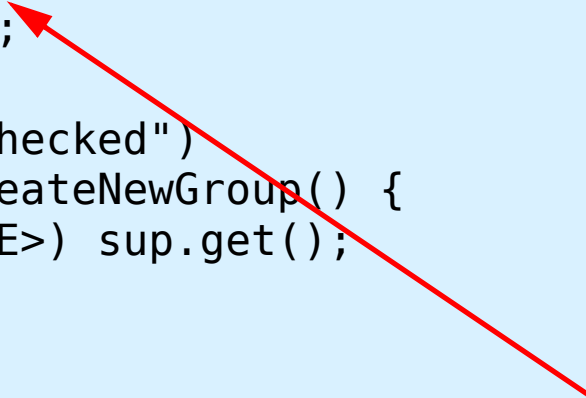
Entwurfsmuster Fabrikmethode im Beispiel SWT-Starfighter

(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.

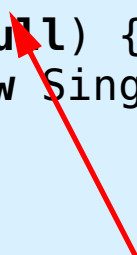
Die Klasse GroupFactory wird daher ergänzt um:

```
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)

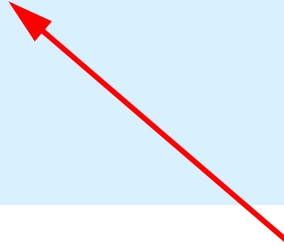
flexible und performante Lösung:

```
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)

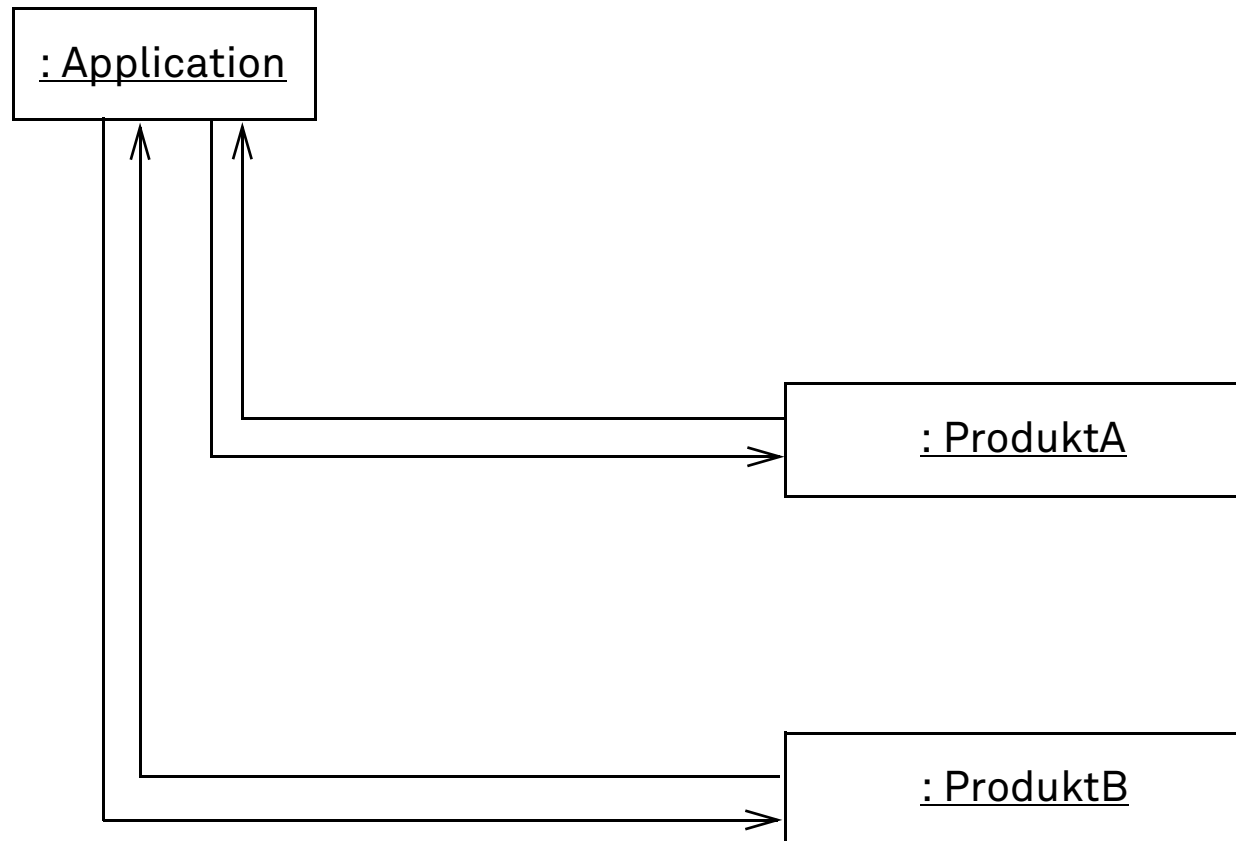
Beispiele:

- ❑ 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

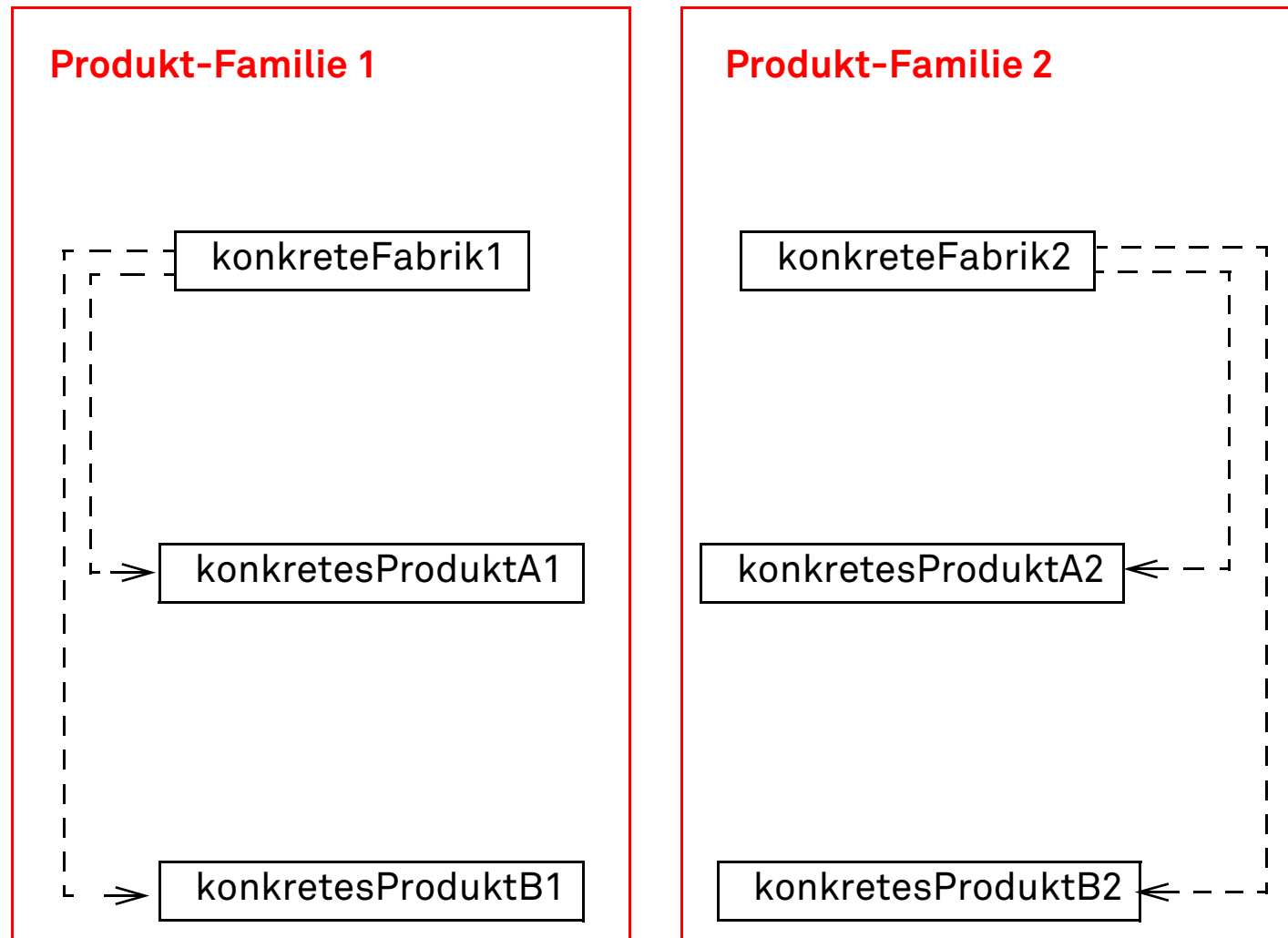
- ❑ Ziel: Einsatz eines Softwareprodukts in unterschiedlichen Anwendungsbereichen
Lösung: mehrere Familien mit Klassen für die Benutzeroberfläche mit unterschiedlicher Präsentation, eine dieser Familien wird ausgewählt und die zugehörigen Objekte werden erzeugt

- ❑ Ziel: Einsatz eines Softwareprodukts auf der Basis unterschiedlicher Techniken zur Datenspeicherung
Lösung: mehrere Familien mit Klassen für die Speicherung von Werten, eine dieser Familien wird ausgewählt und die zugehörigen Objekte werden erzeugt

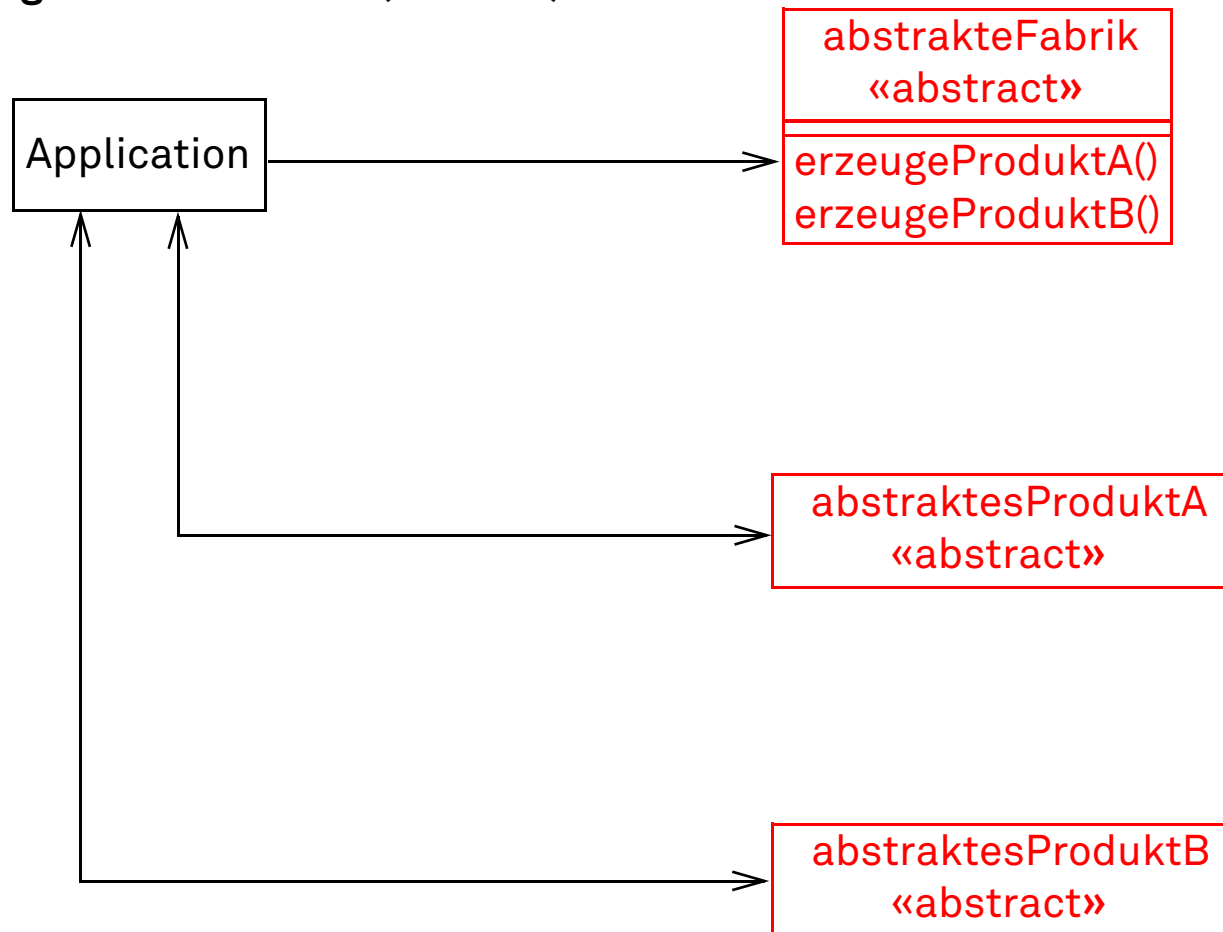
allgemeine Struktur (Objekte der Anwendung)



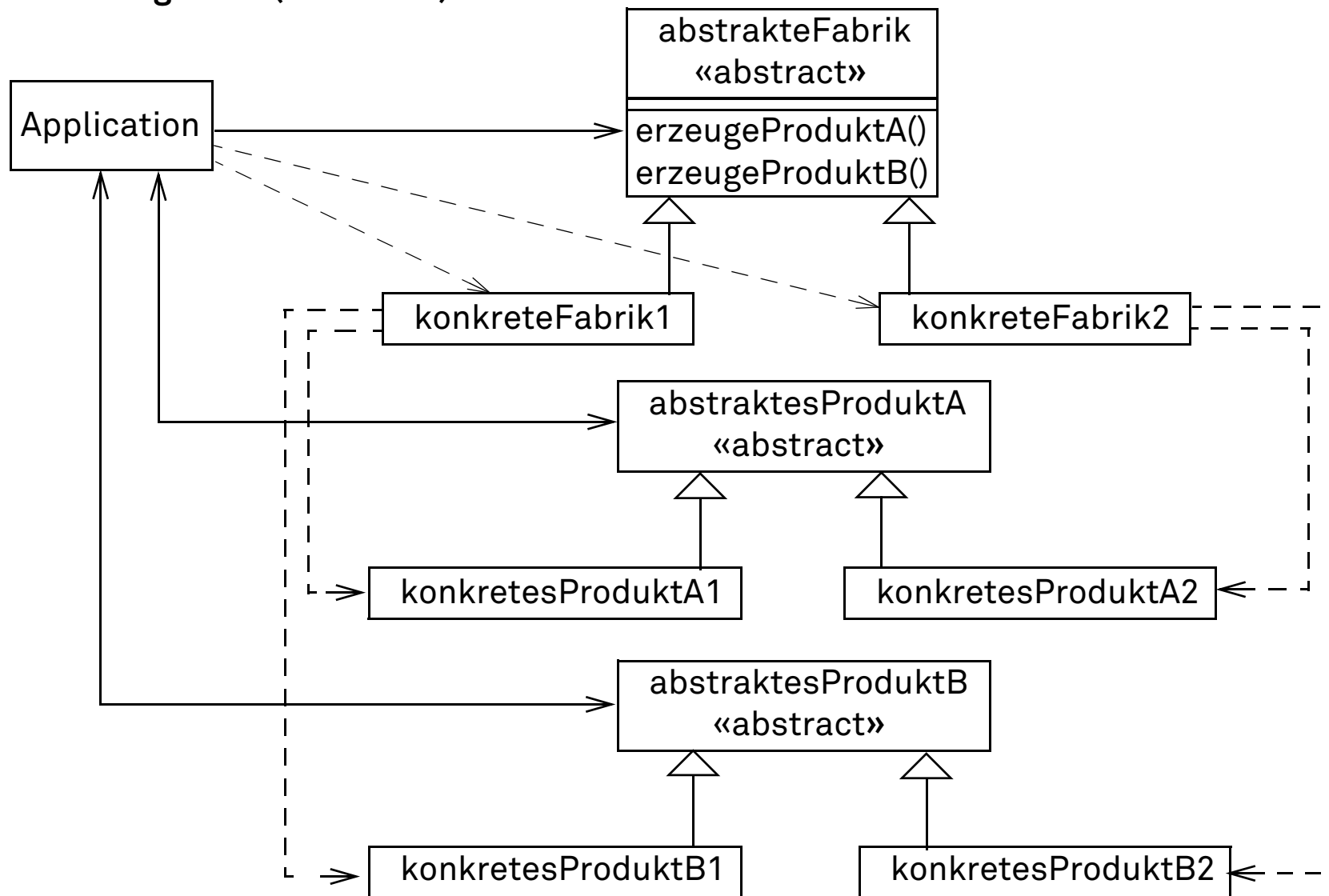
allgemeine Struktur (Klassendiagramm Produkt-Familien)



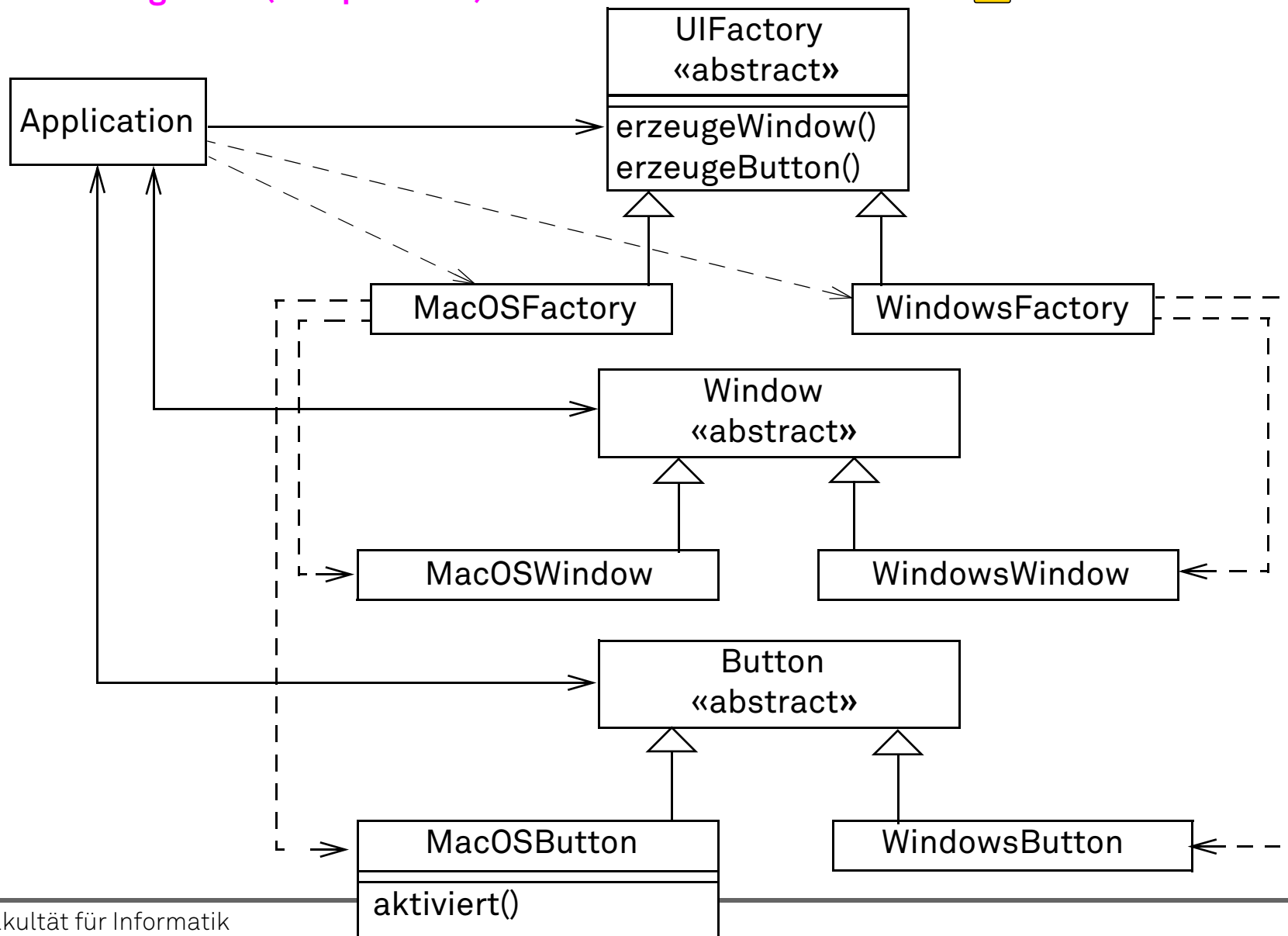
allgemeine Struktur (Klassen)



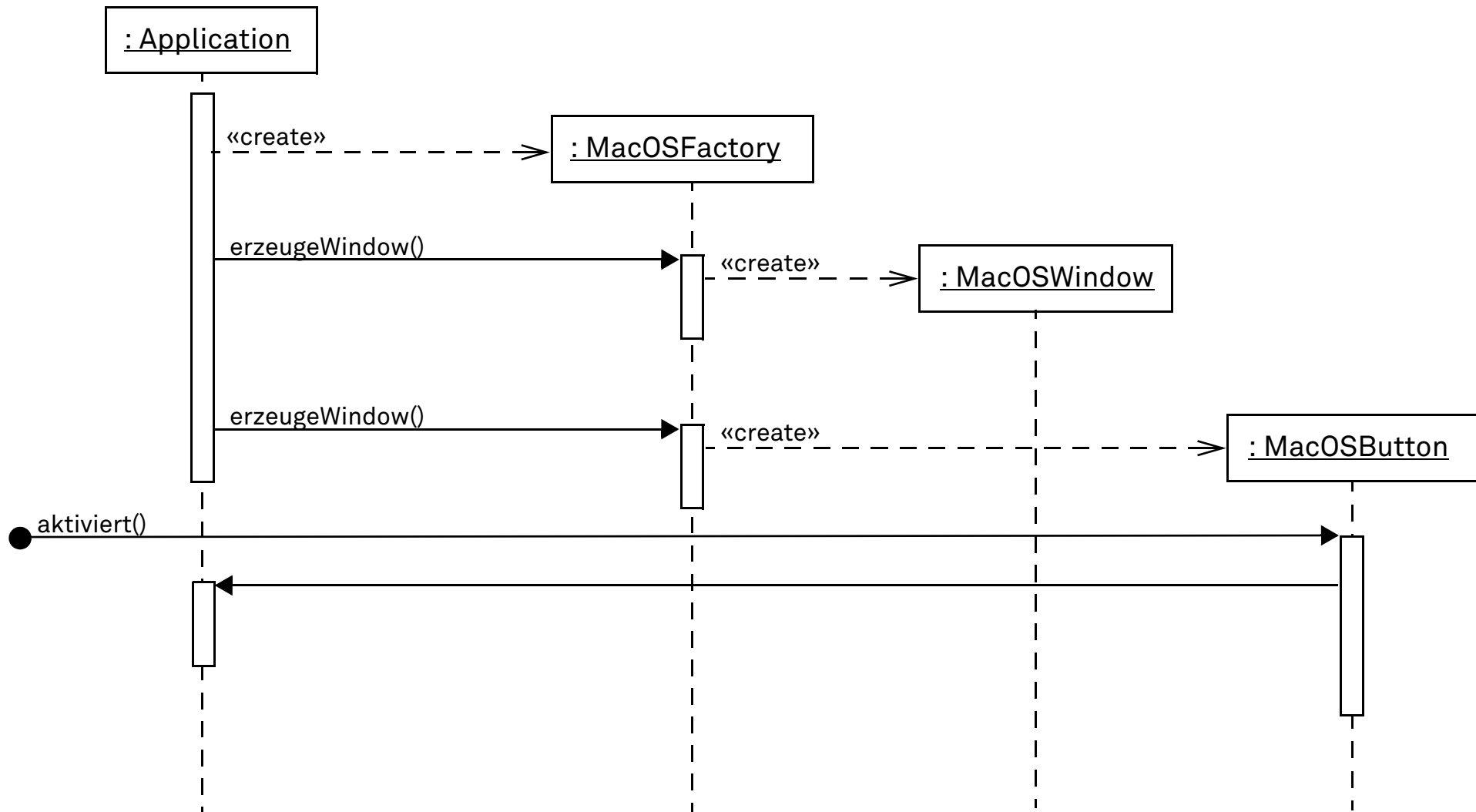
Klassendiagramm (Übersicht)



Klassendiagramm (Beispiel SWT)



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.

