

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.

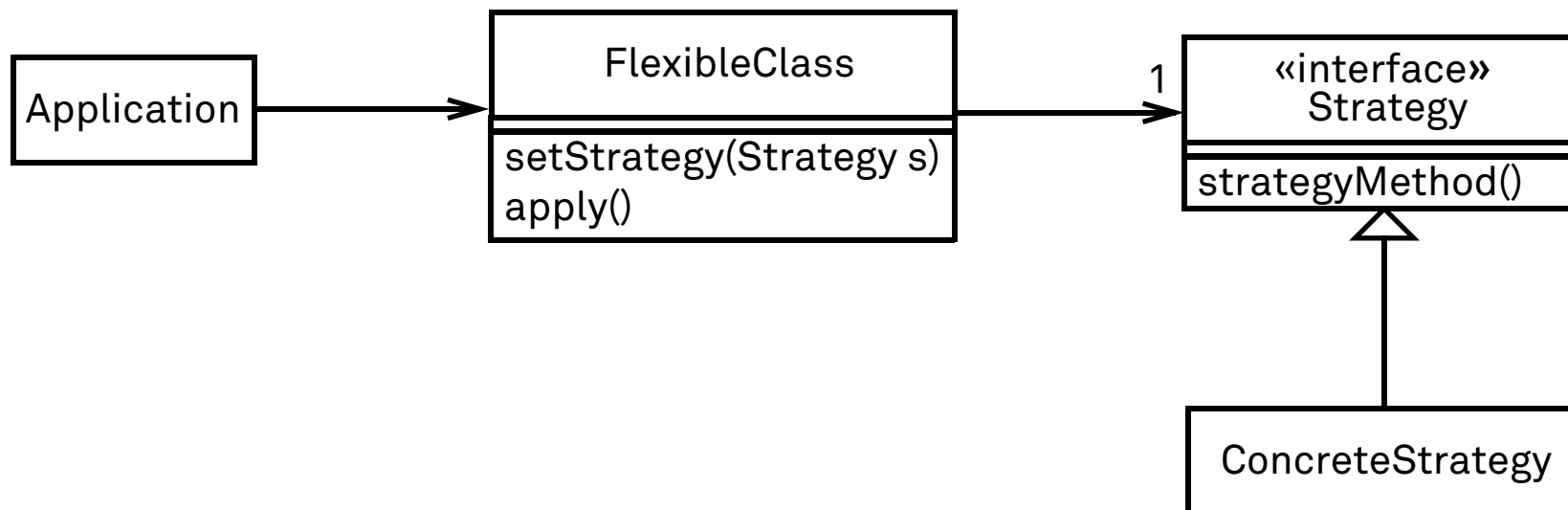
Beispiele:

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

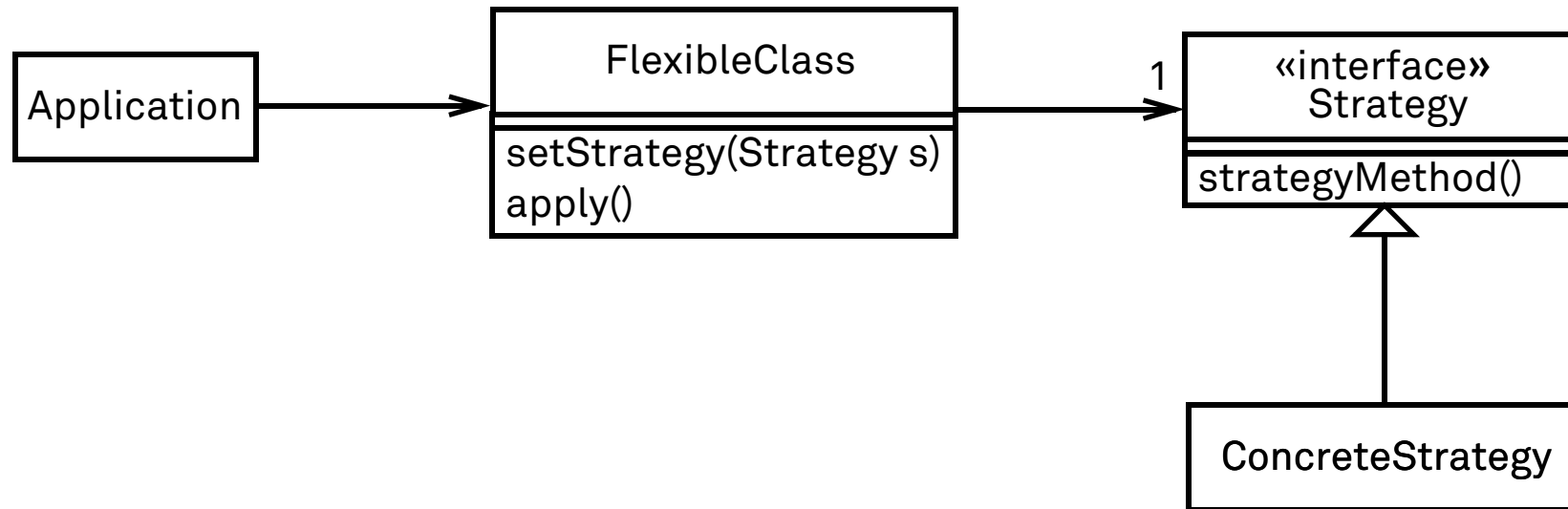
Literatur: Eilebrecht, Karl; Starke, Gernot: Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung, S. 48-52
<http://www.springerlink.com/content/t38726/#section=660020&page=6&locus=71>

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

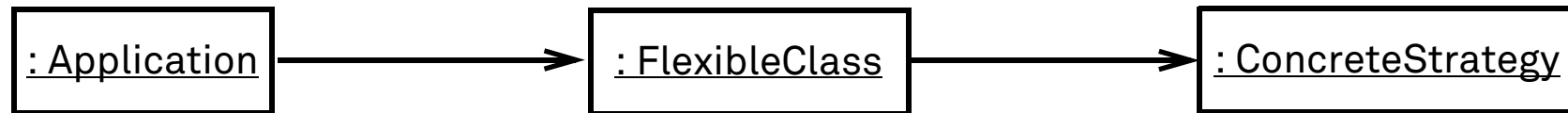


Objektdiagramm

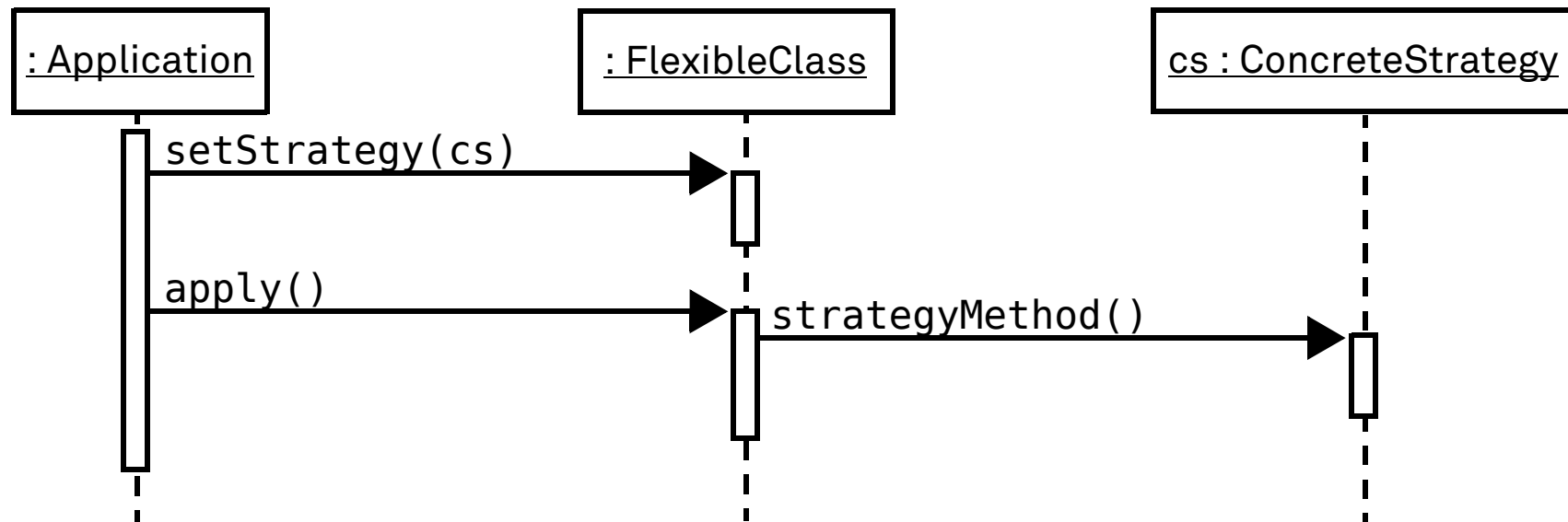


Entwurfsmuster *Strategie* – Objektstruktur

Objektdiagramm

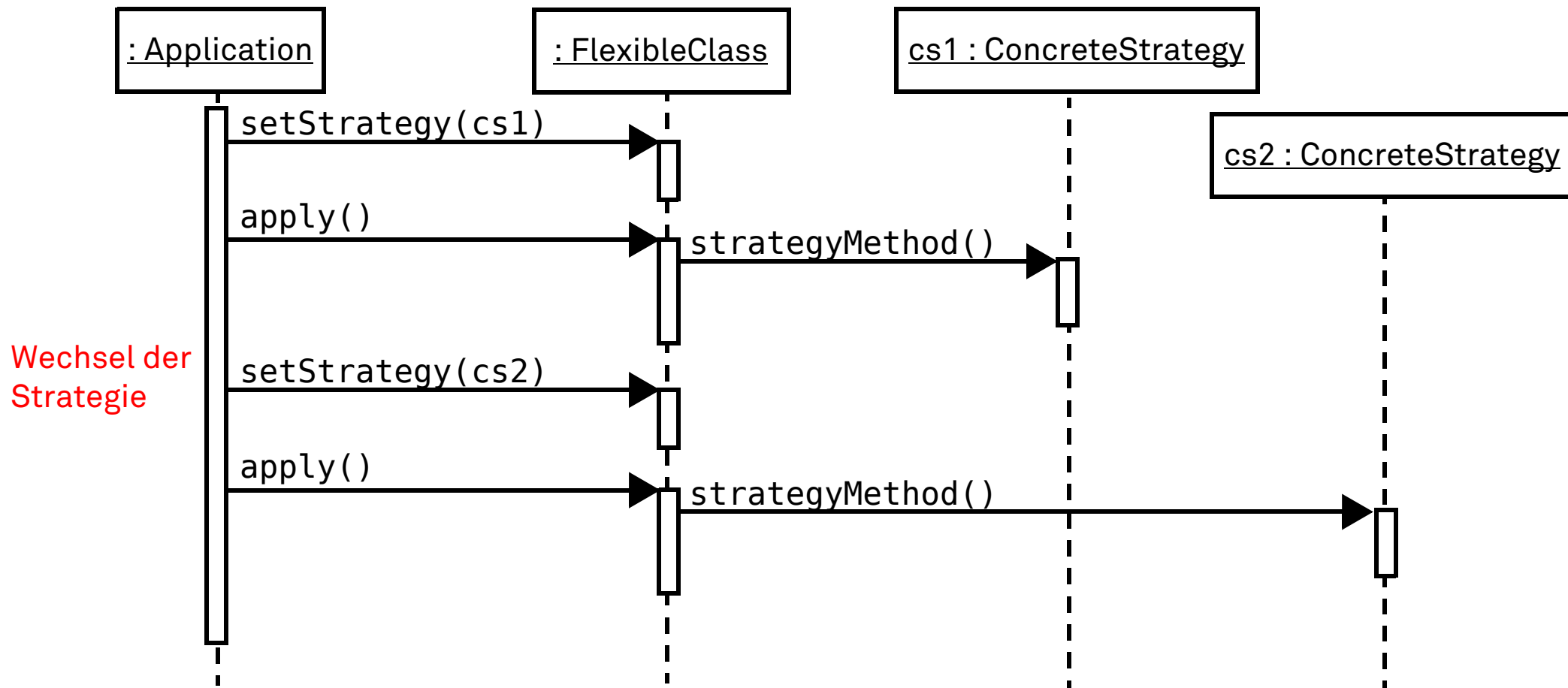


Sequenzdiagramm



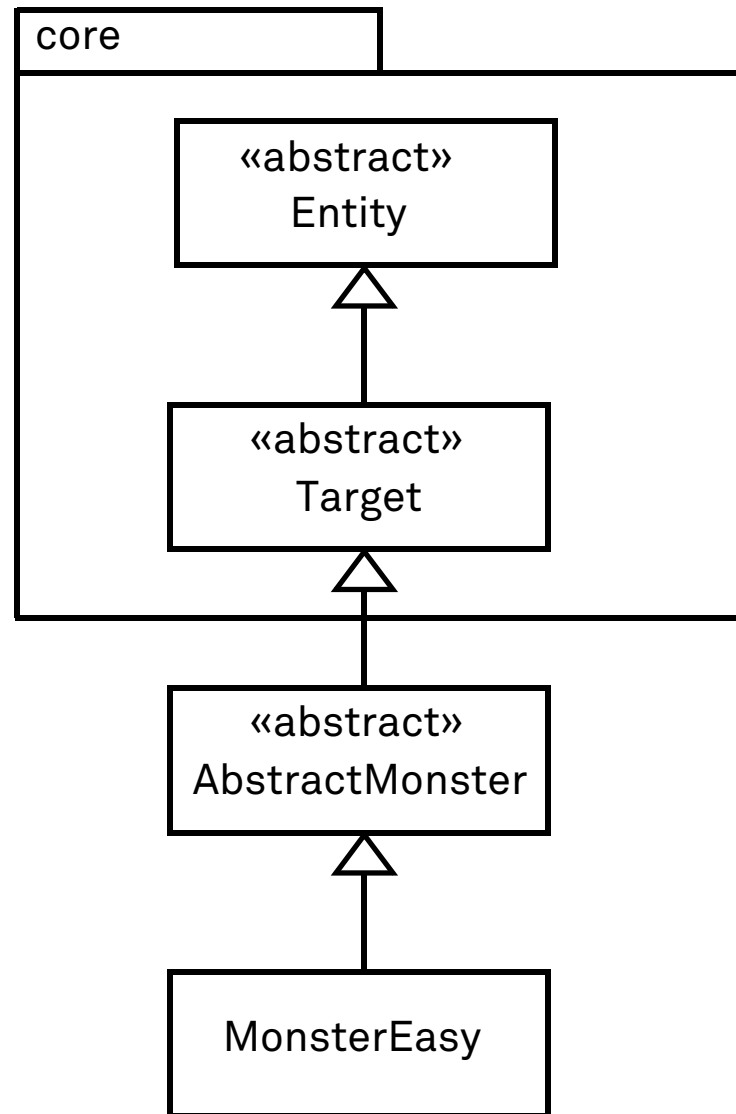
Entwurfsmuster *Strategie* – Objektstruktur

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.

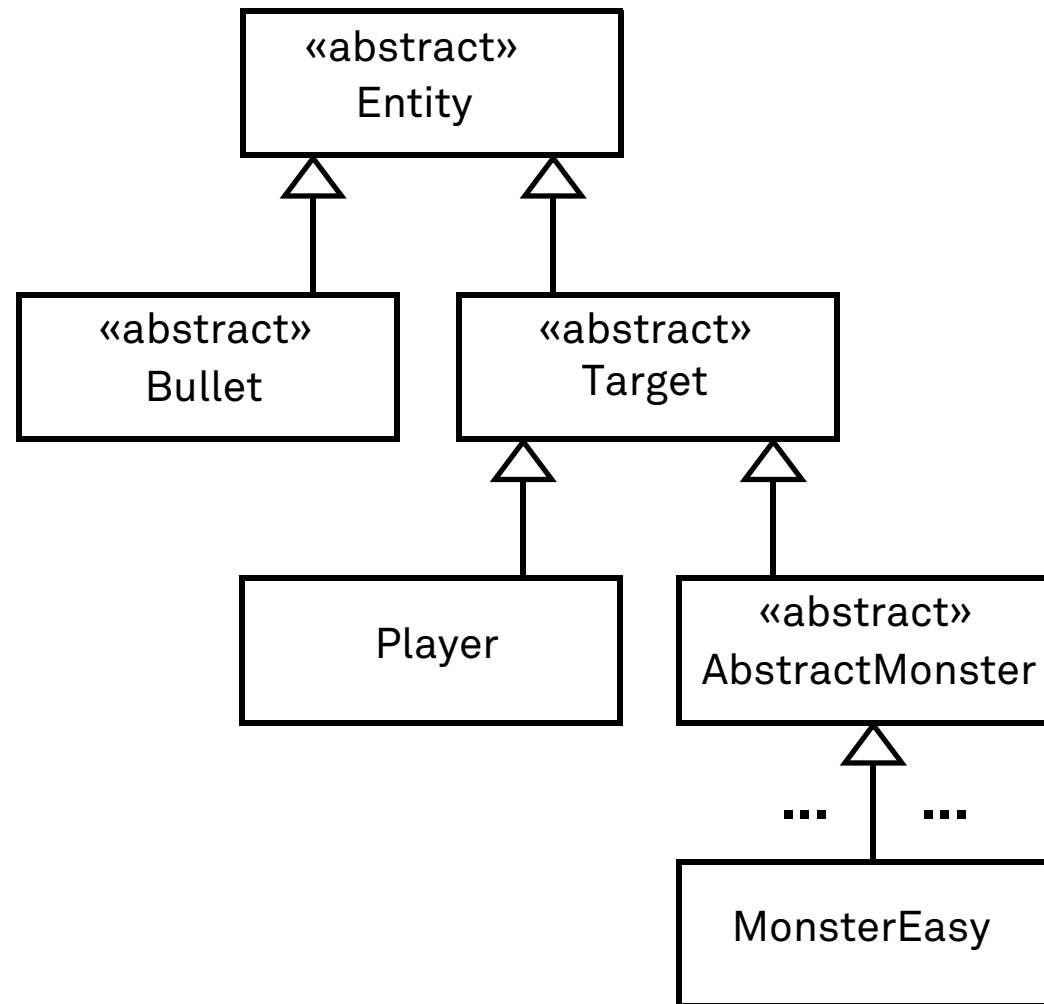
Beispiel aus SWT-Starfighter – Interface BufferedGroup

```
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

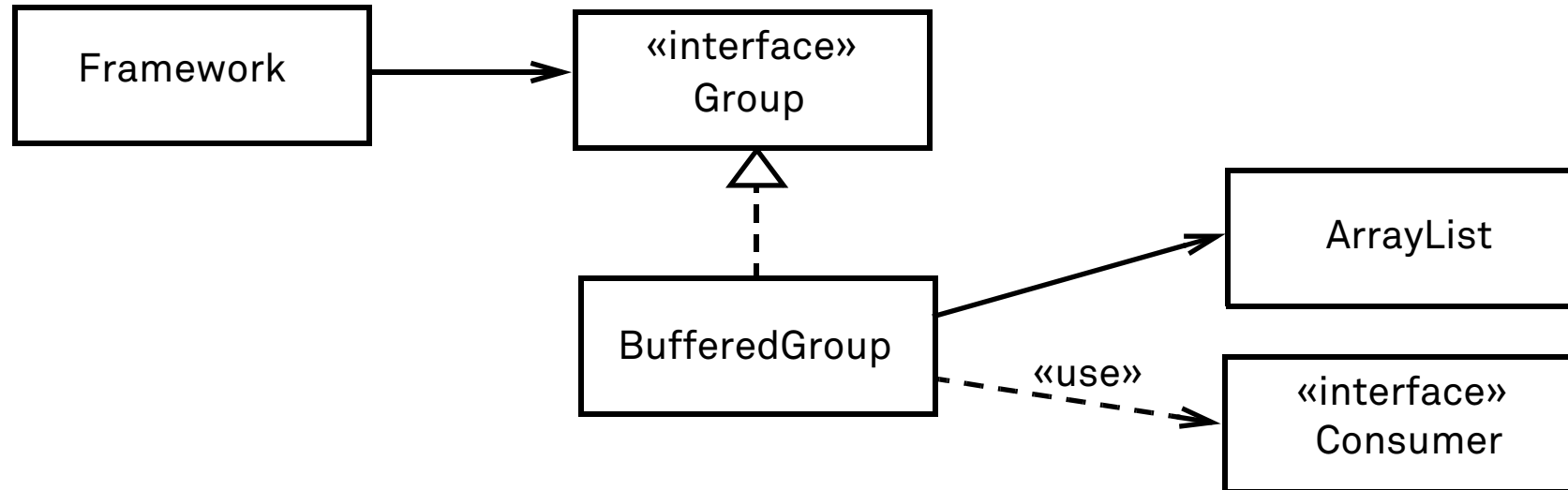


Klassenhierarchie fasst
Gemeinsamkeiten mehrerer
Unterklassen zusammen:

- 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:

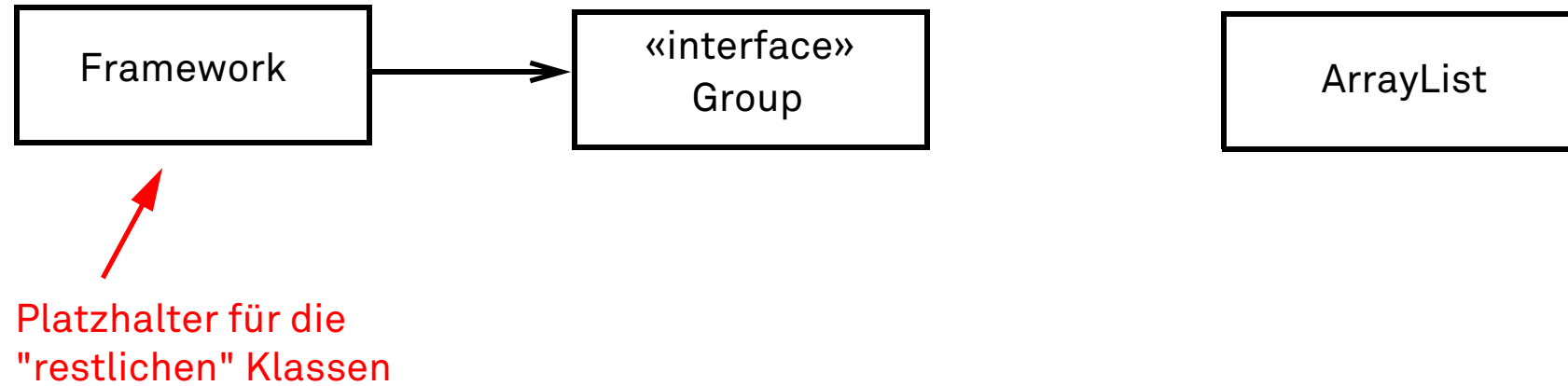
- ❑ Viele Klassen werden unabhängig von speziellen Problemlösungen erstellt.
- ❑ Eine solche Klasse bietet durch die von ihr bereitgestellten Methoden implizit eine Schnittstelle an.
- ❑ Methoden erwarten von den von ihnen benutzten Objekten bestimmte Schnittstellen.
- ❑ Fertige Klassen sollen gemeinsam 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:

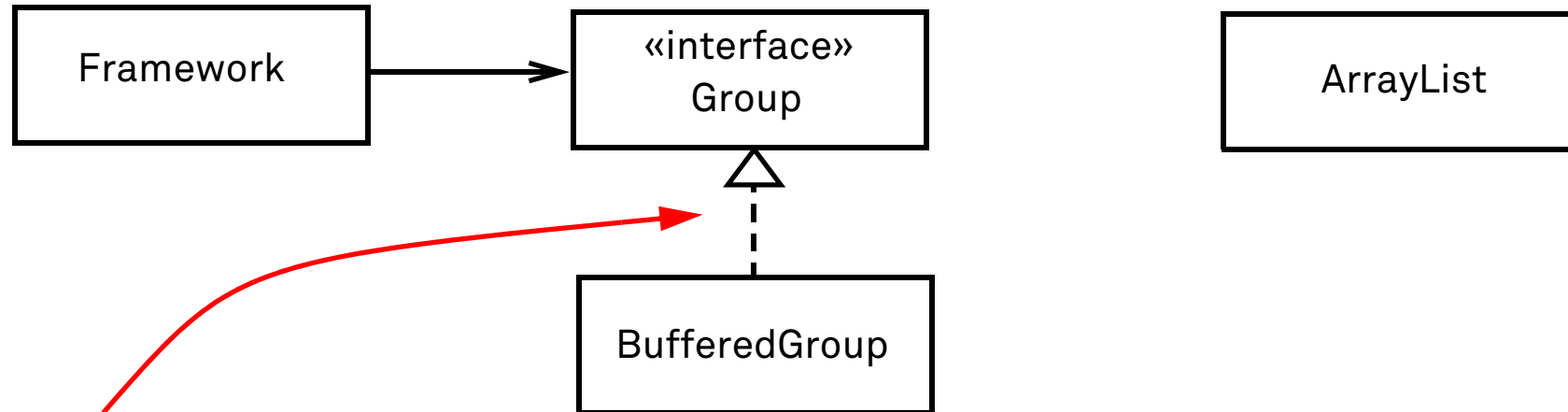
```
public interface Group<E> {  
    public void add(E element);  
    public void remove(E element);  
    public void forEach(Consumer<? super E> handle);  
    ...  
}
```
- ❑ Unterstützung für die Implementierung:
 - Es liegt eine implementierte, **getestete und praktisch bewährte** Klasse `ArrayList` 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

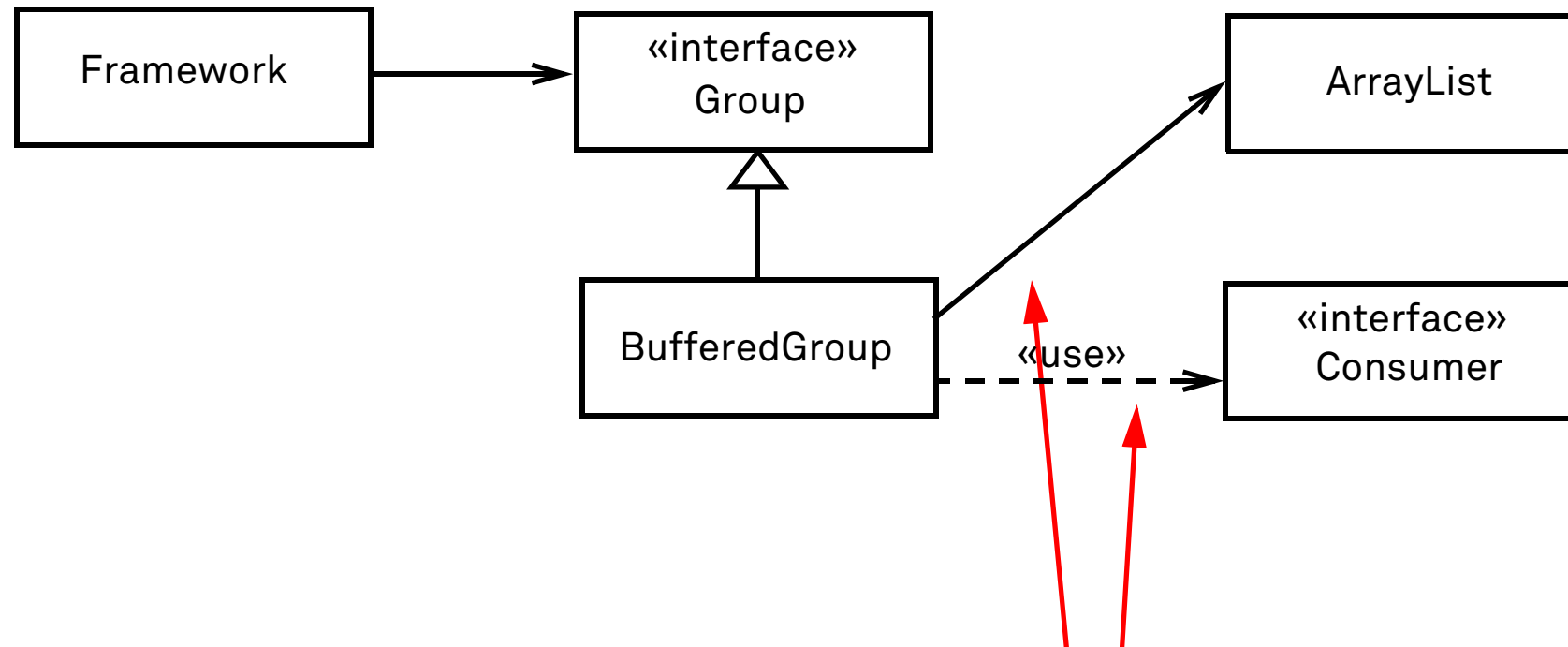
(Fortsetzung)



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

Visualisierung Adapter

(Fortsetzung)

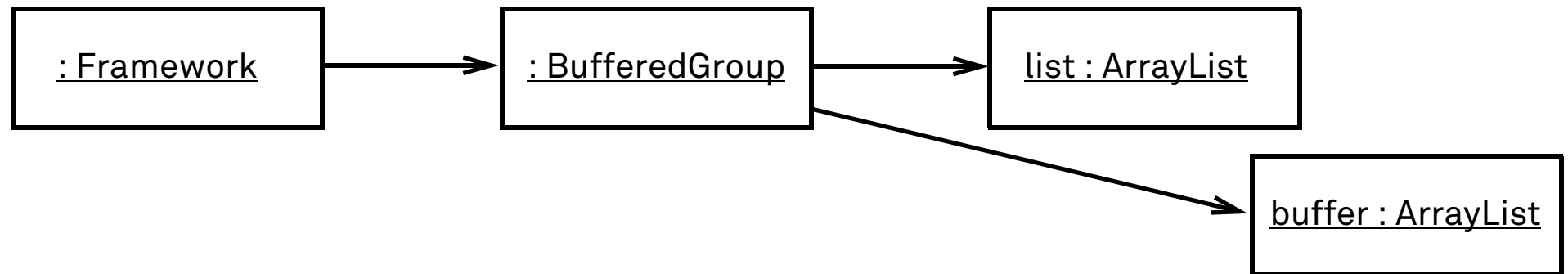


Nutzung bestehender Klassen,
um BufferedGroup mit geringem
Aufwand zu implementieren

Visualisierung Adapter

(Fortsetzung)

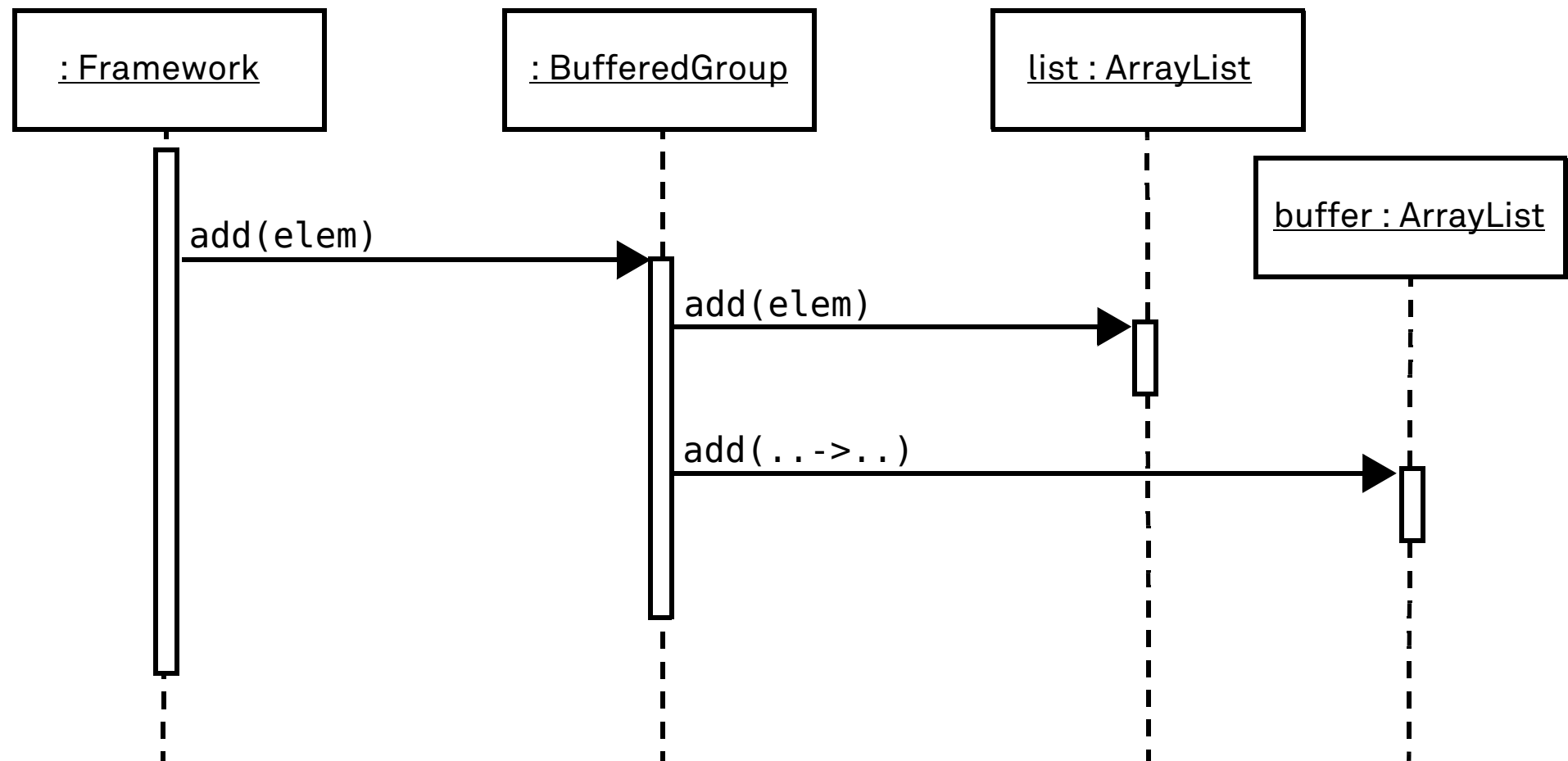
Objektstruktur bei der Ausführung



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

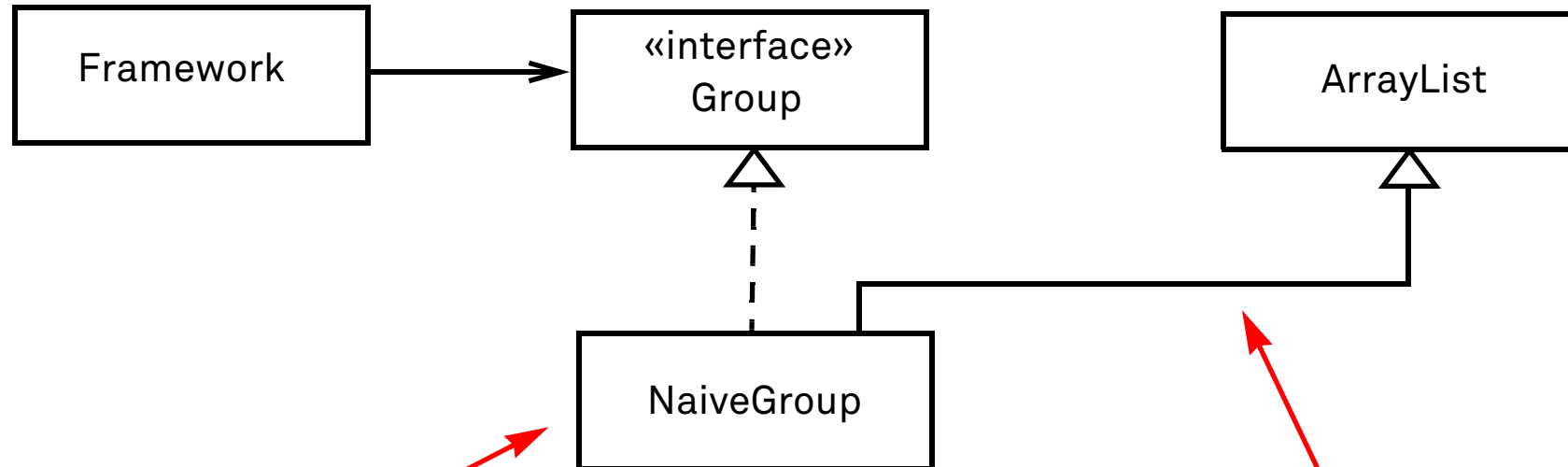
Zusammenfassung

(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> {  
    ...  
}
```

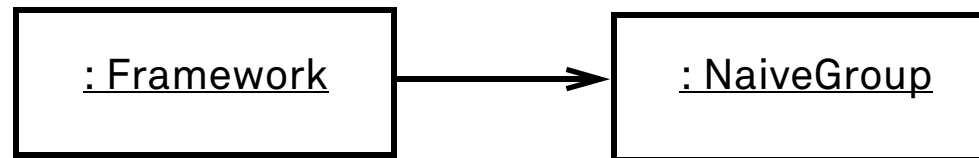
Anmerkungen:

- ❑ 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.

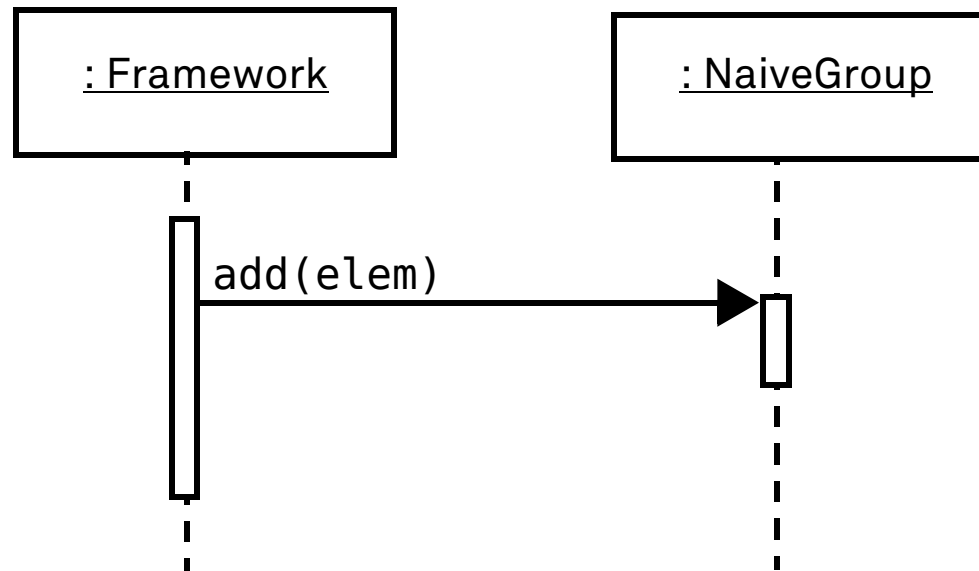
Visualisierung Klassenadapter

(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.

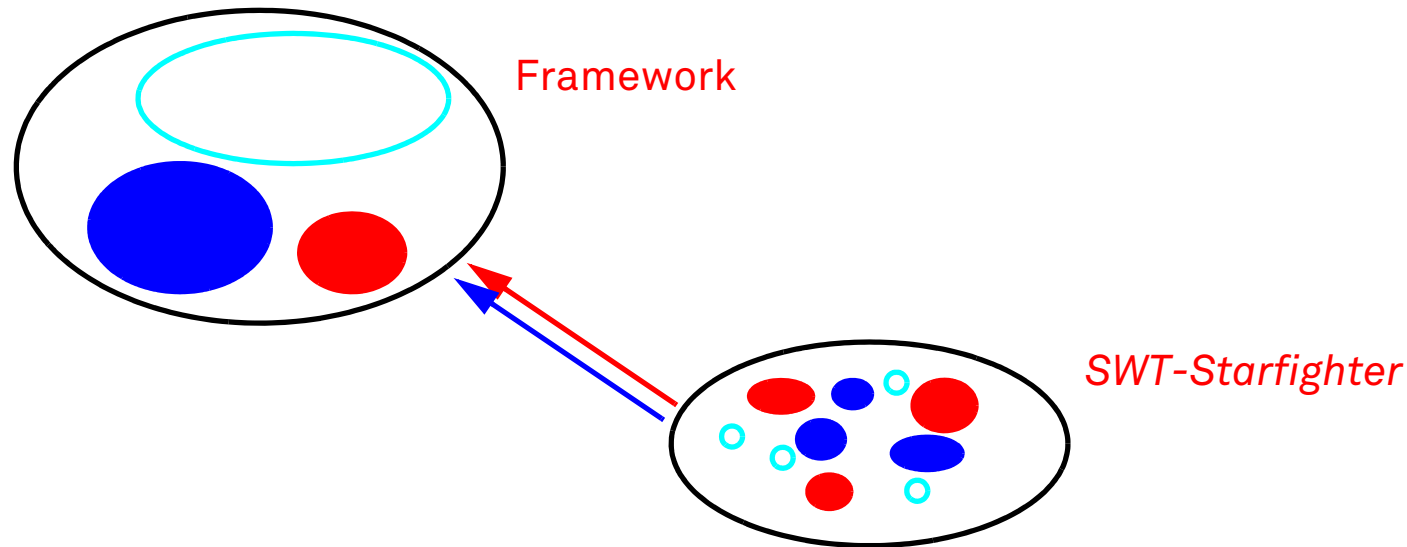
- ❑ Objektadapter
 - umfasst mehrere Objekte,
 - von denen der Adapter die passende Schnittstelle herstellt
 - und die "Arbeit" (teilweise) an andere Objekte weitergibt (**Delegation**).

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)

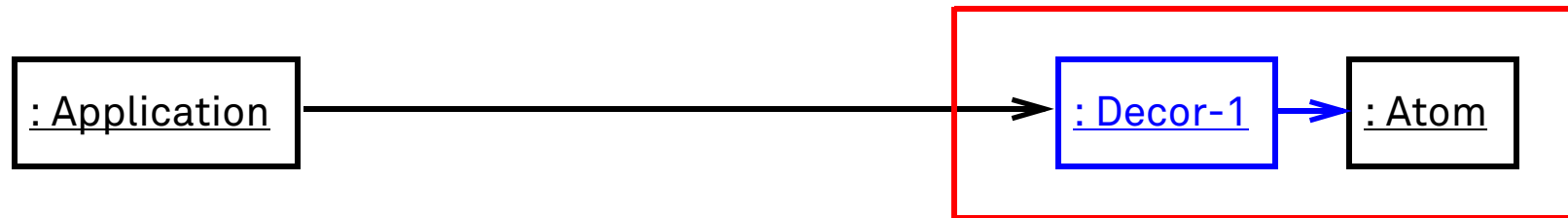
Ein Dekorierer

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)



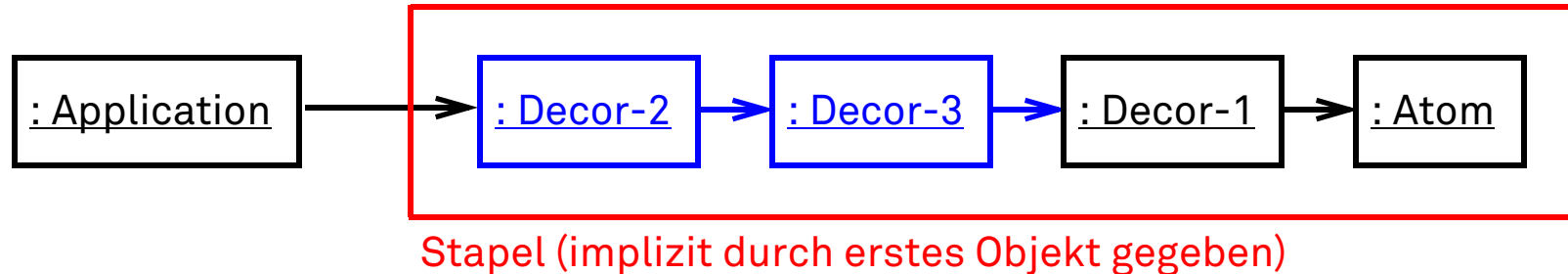
Stapel (implizit durch erstes Objekt gegeben)

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)

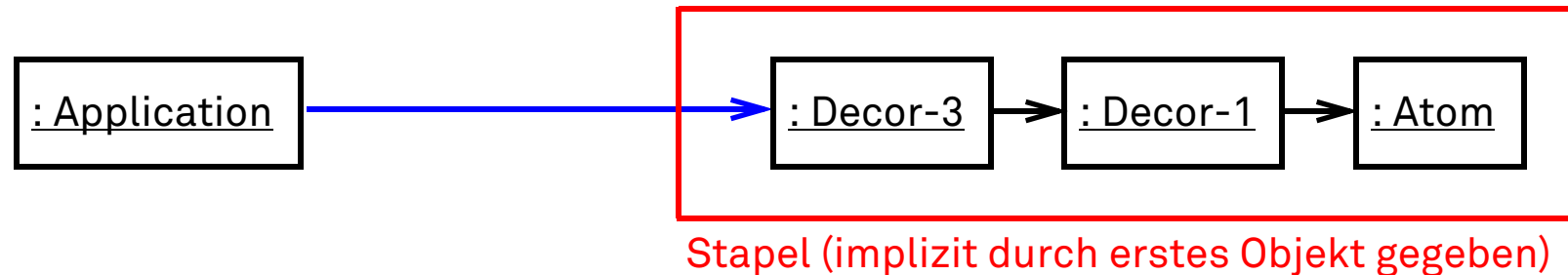


Einsatzszenarien für solche Stapel:

- ❑ 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)

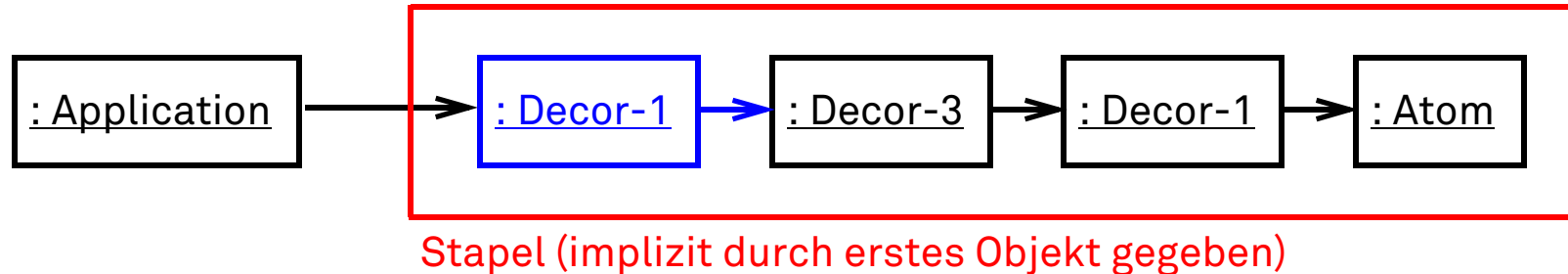


Einsatzszenarien für solche Stapel:

- ❑ 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)

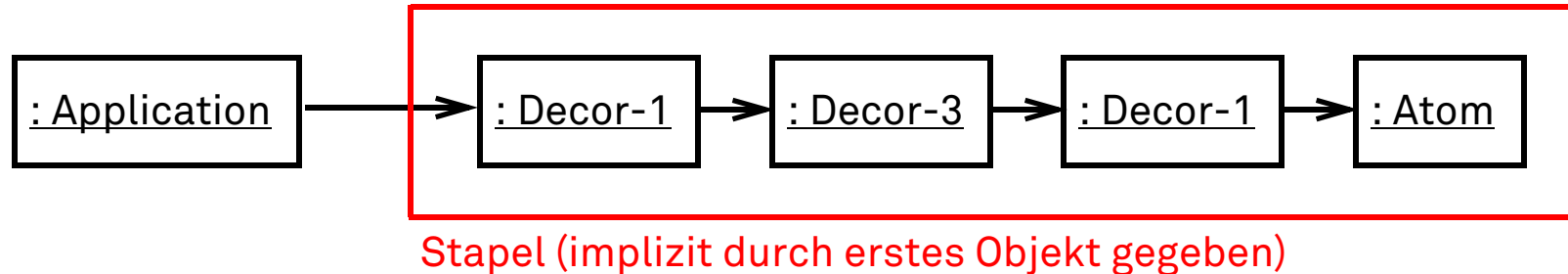


Einsatzszenarien für solche Stapel:

- ❑ 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)

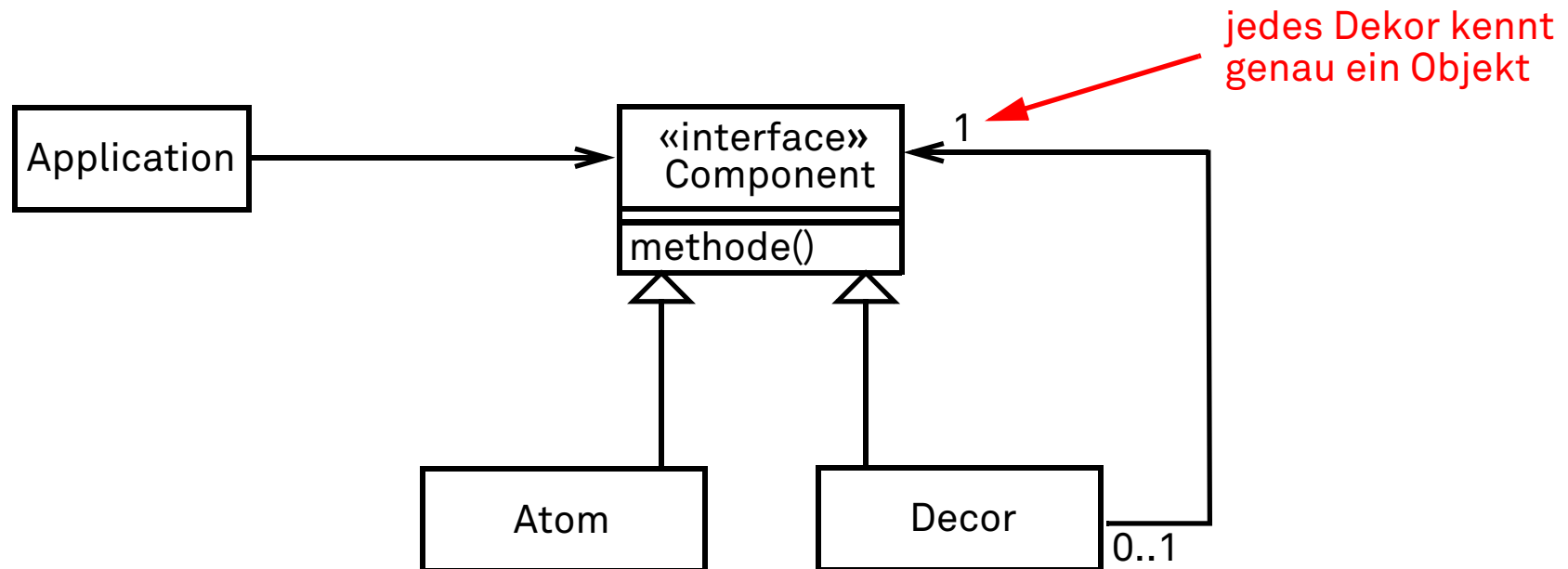


Einsatzszenarien für solche Stapel:

- ❑ 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

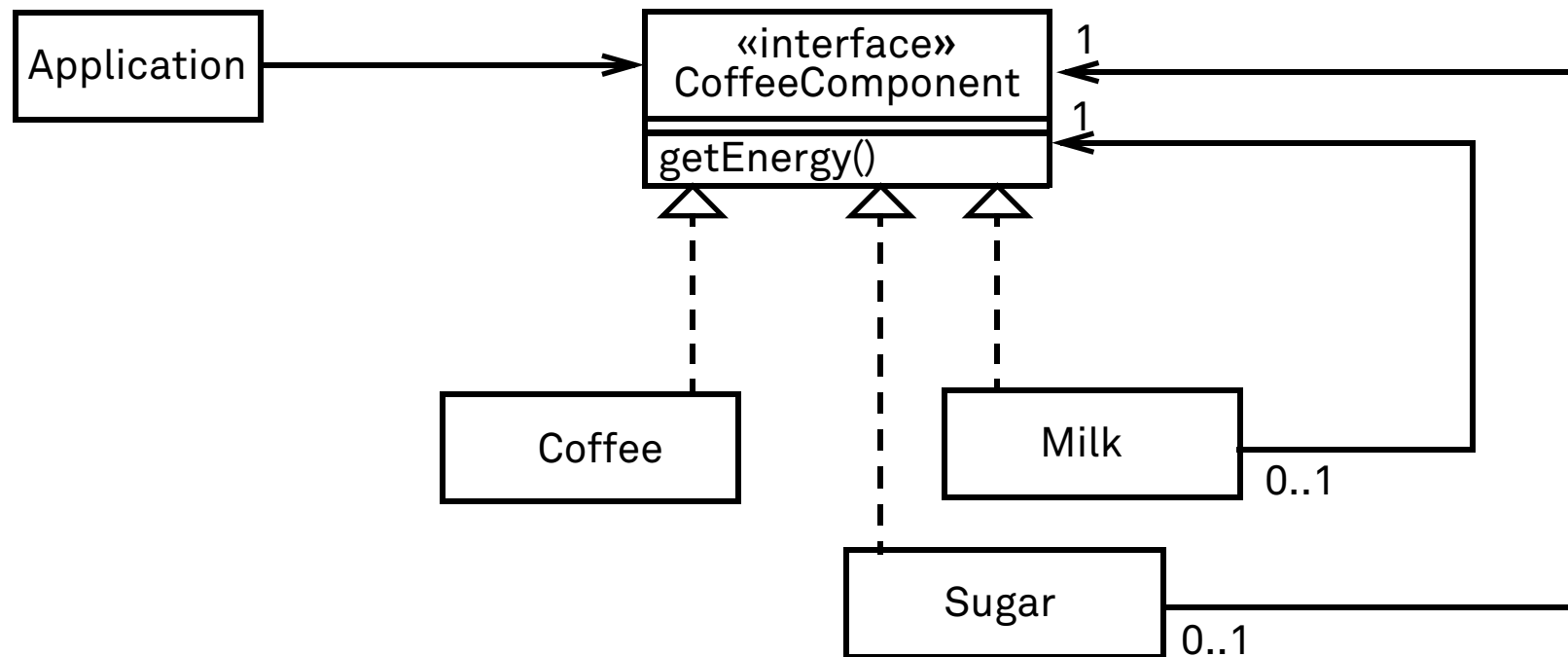
- Weitere Eigenschaften sollen im Rahmen der Entwicklung leicht ergänzt werden:
Alle Objekte müssen die gleiche Schnittstelle erfüllen.



Beispiele für den Einsatz des Entwurfsmusters *Dekorier*

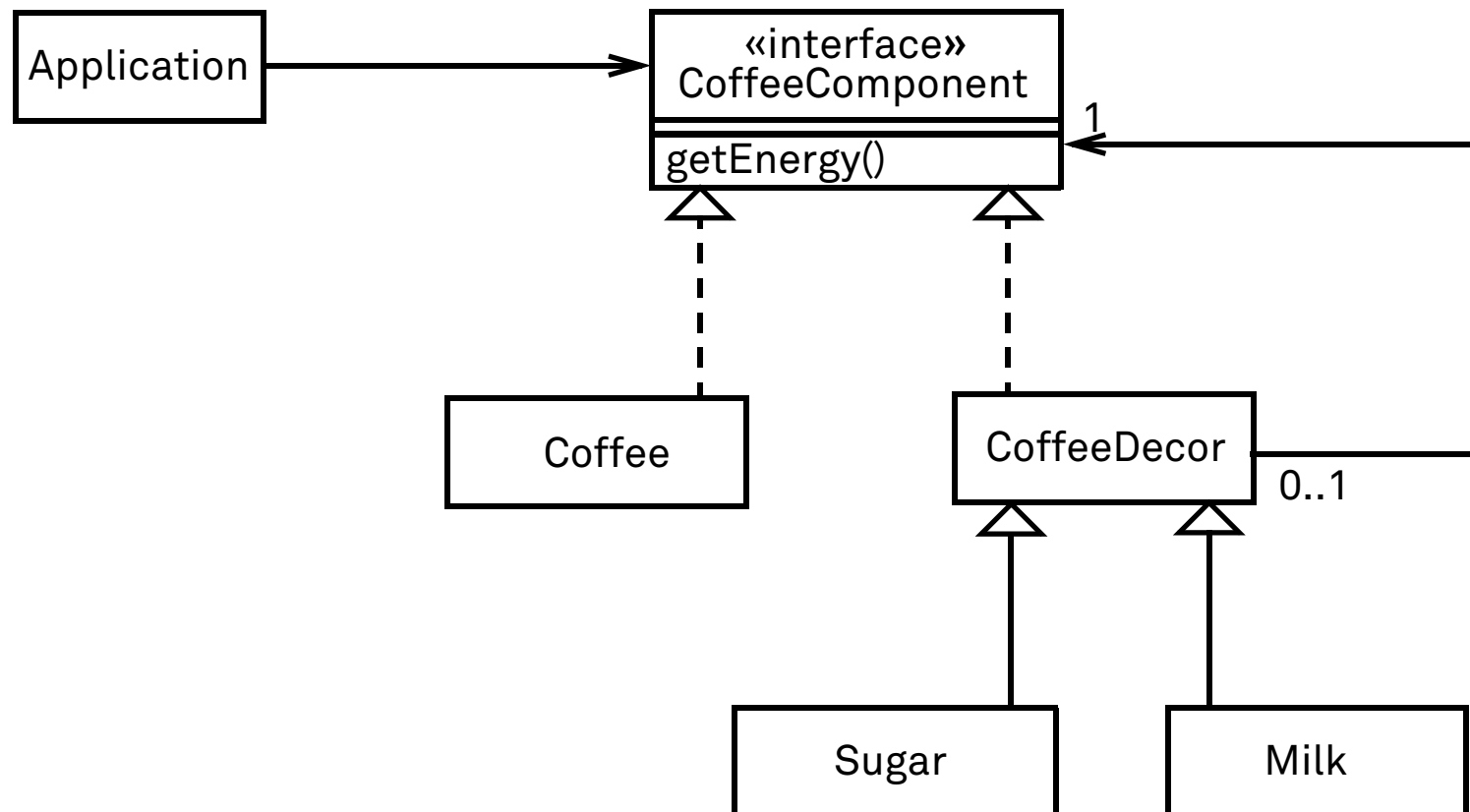
- ❑ GUI-Komponenten werden "dekoriert":
 - Umrahmungen werden zu Textfeld oder Graphikbereich hinzugefügt.
 - Funktionsleisten werden zu einem Fenster hinzugefügt.
 - Horizontaler und vertikaler Scrollbar wird zu Textfeld hinzugefügt.
- ❑ Dateien (Stream-Klassen der Java-Bibliothek)
 - Datei ist eine 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)



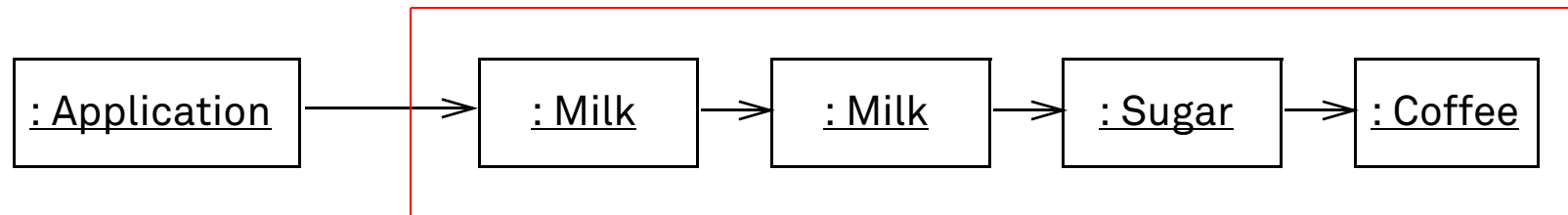
Visualisierung (Beispiel: Energieberechnung für Kaffee) (alternative Implementierung)

(Fortsetzung)



Visualisierung (Beispiel: Energieberechnung für Kaffee)

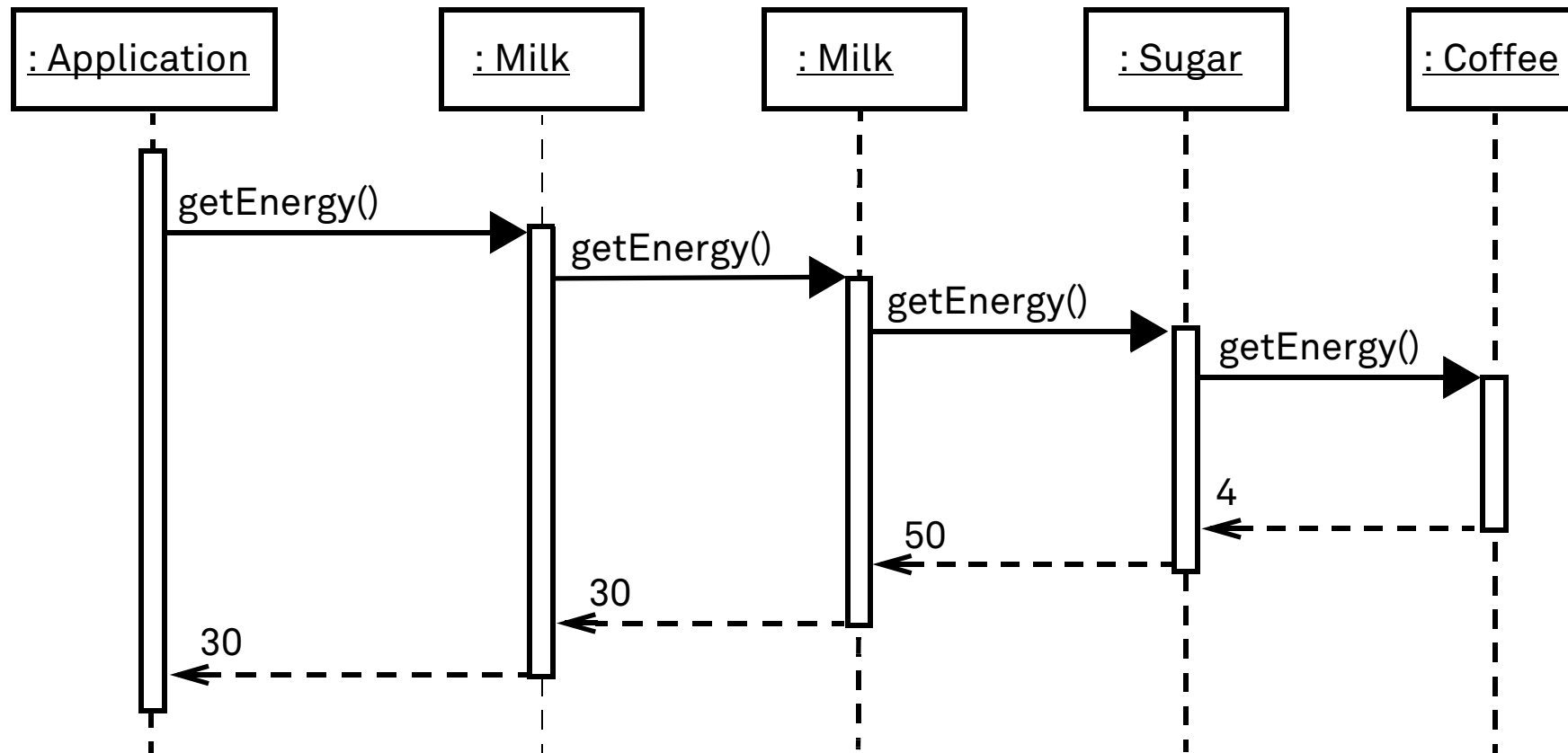
(Fortsetzung)



```
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; }
}
```

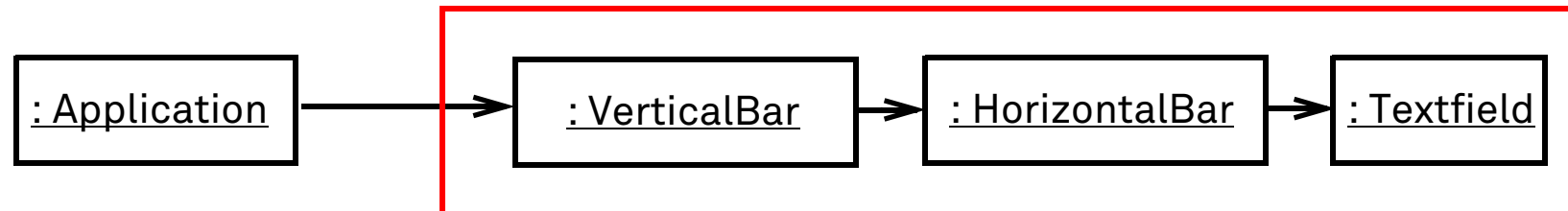
Visualisierung (Beispiel: Energieberechnung für Kaffee) (Methodenaufrufe)

(Fortsetzung)

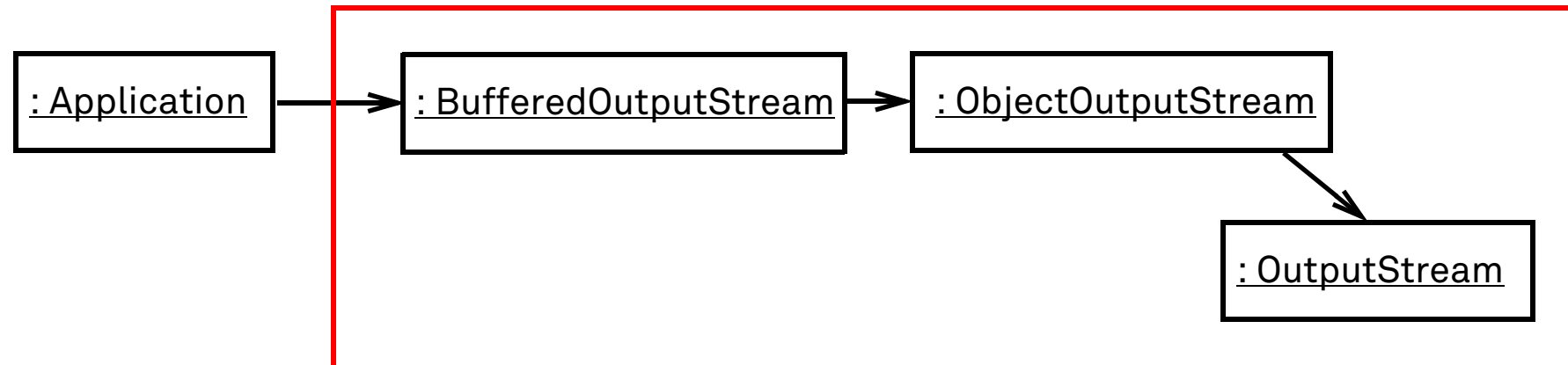


Visualisierungen (weitere Beispiele)

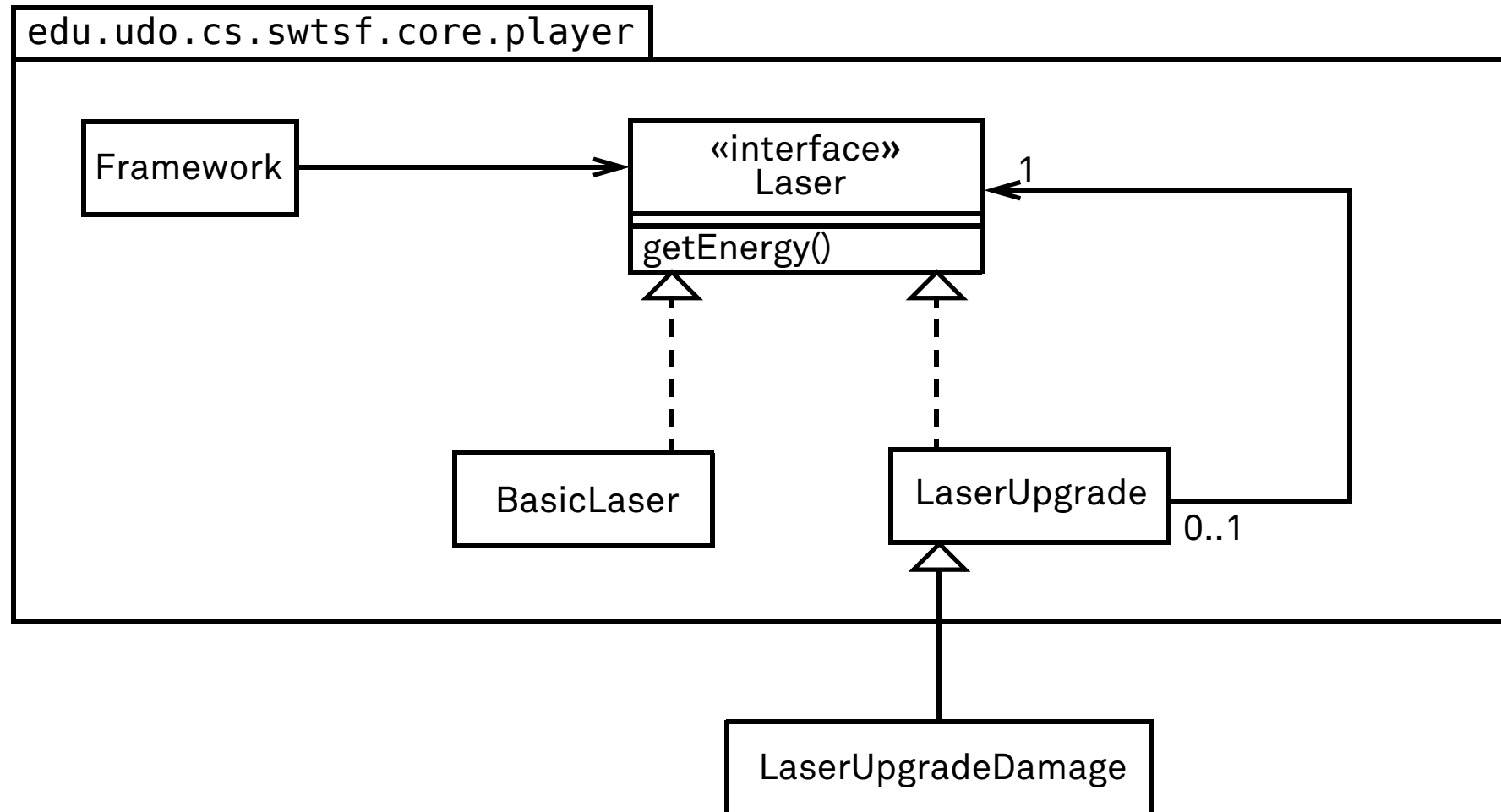
dekorierte Fenster



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 schon ein gleiches Dekor, führt WeakLaserUpgradeDamage nicht zu einer Änderung.

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

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

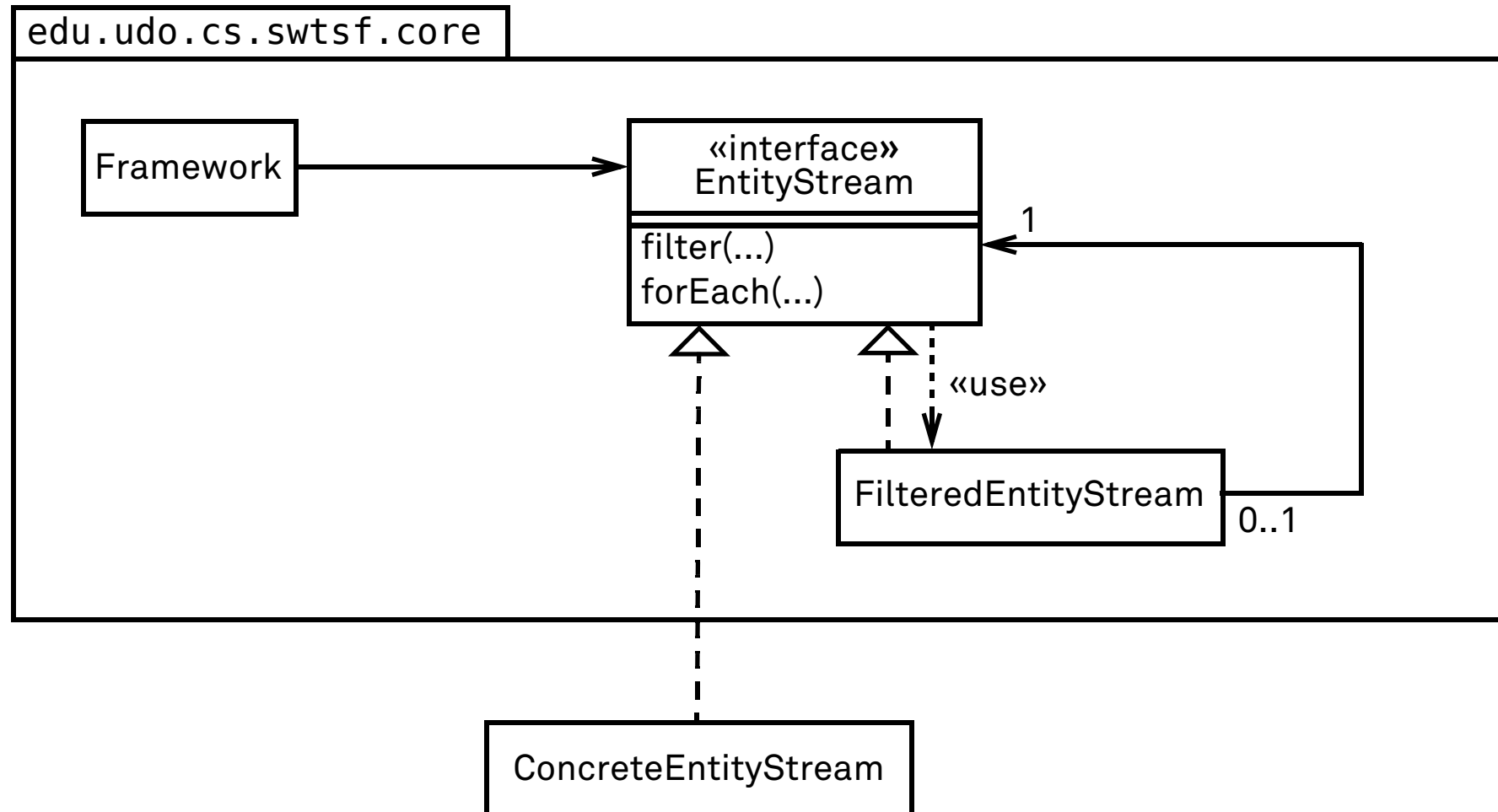
Dieses Beispiel zeigt eine etwas andere Umsetzung des Dekorierermusters, bei dem auf ein explizites Verbinden der Dekore verzichtet wird. Stattdessen werden die Dekore während der Ausführung über Methodenaufrufe erzeugt und über Referenzen in den Methodenaufrufen aneinander gehängt.

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)

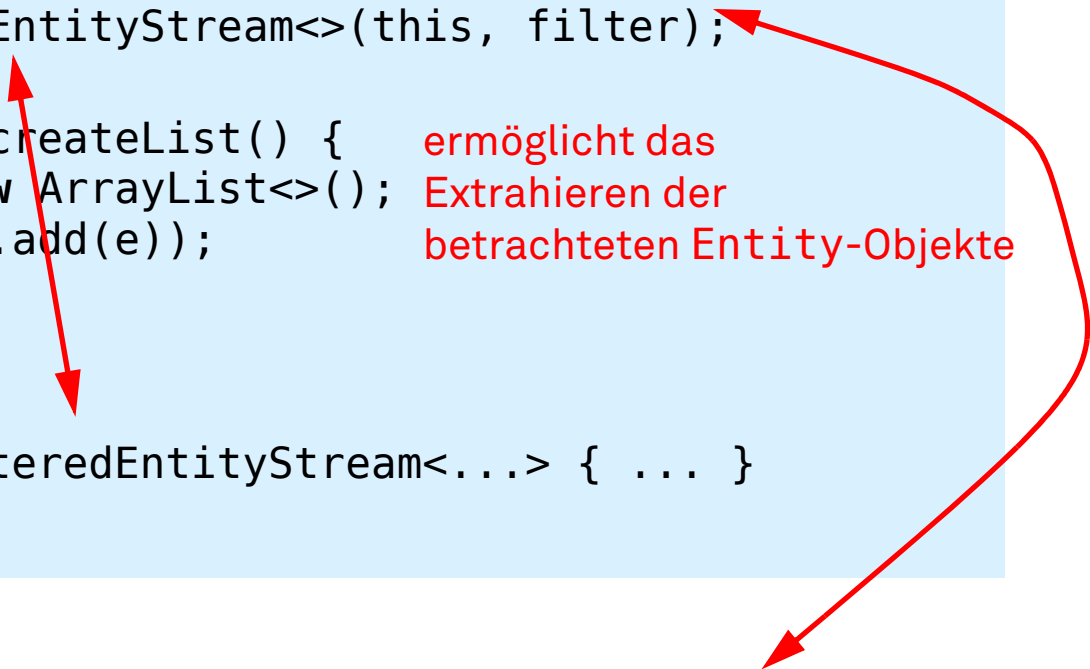


weiteres Beispiel aus SWT-Starfighter – Dekorierer zum Interface EntityStream

(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

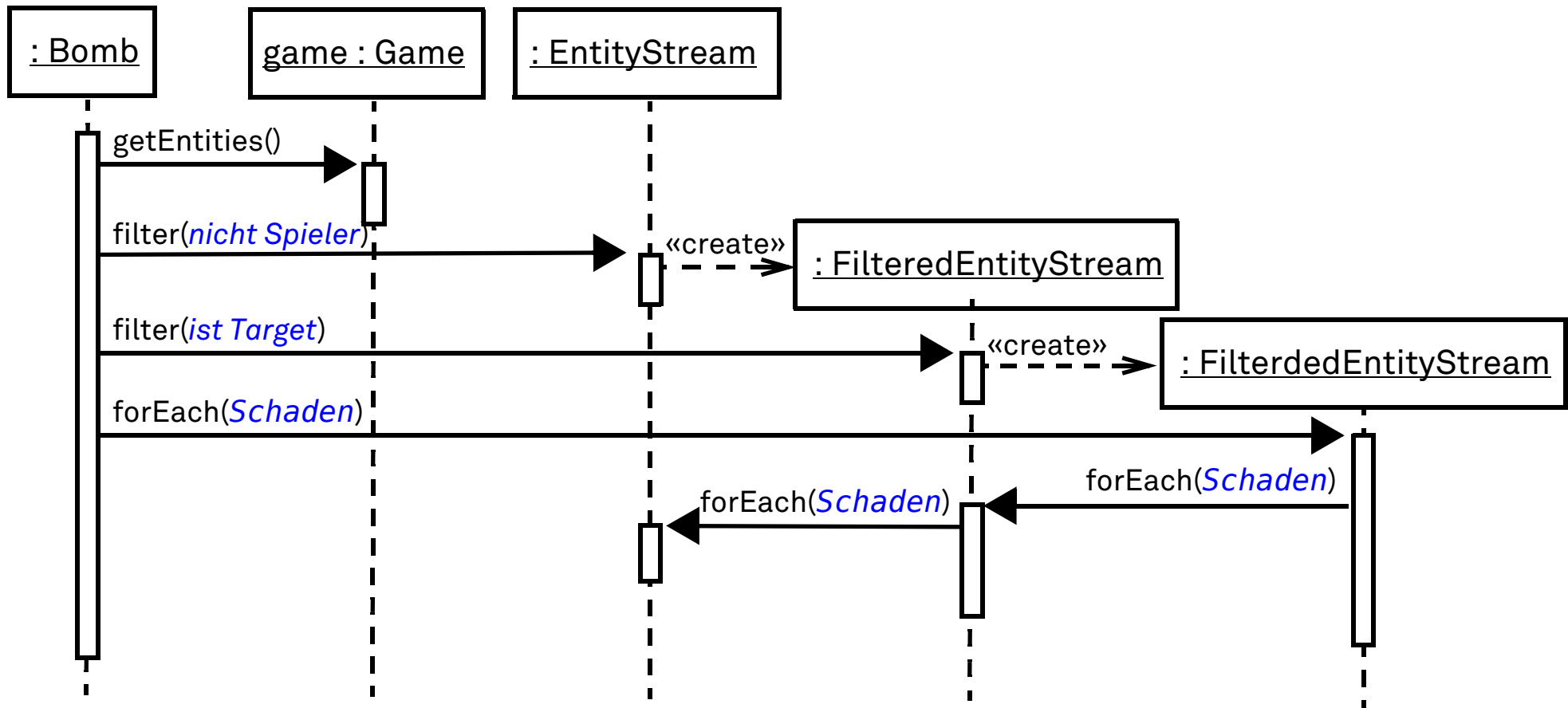
reicht das accept des Arguments weiter an den forEach-Aufruf des baseStream-Objekts, bewacht durch die test-Methode

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.