Tagesprogramm

Decorator

Proxy

Iterator

Decorator (Wrapper)

Zweck: gibt Objekten dynamisch neue Verantwortlichkeiten,
Alternative zur Vererbung

Anwendungsgebiete:

dynamisches Hinzufügen von Verantwortlichkeiten
ohne Beeinflussung anderer Objekte

Verantwortlichkeiten, die wieder entzogen werden können

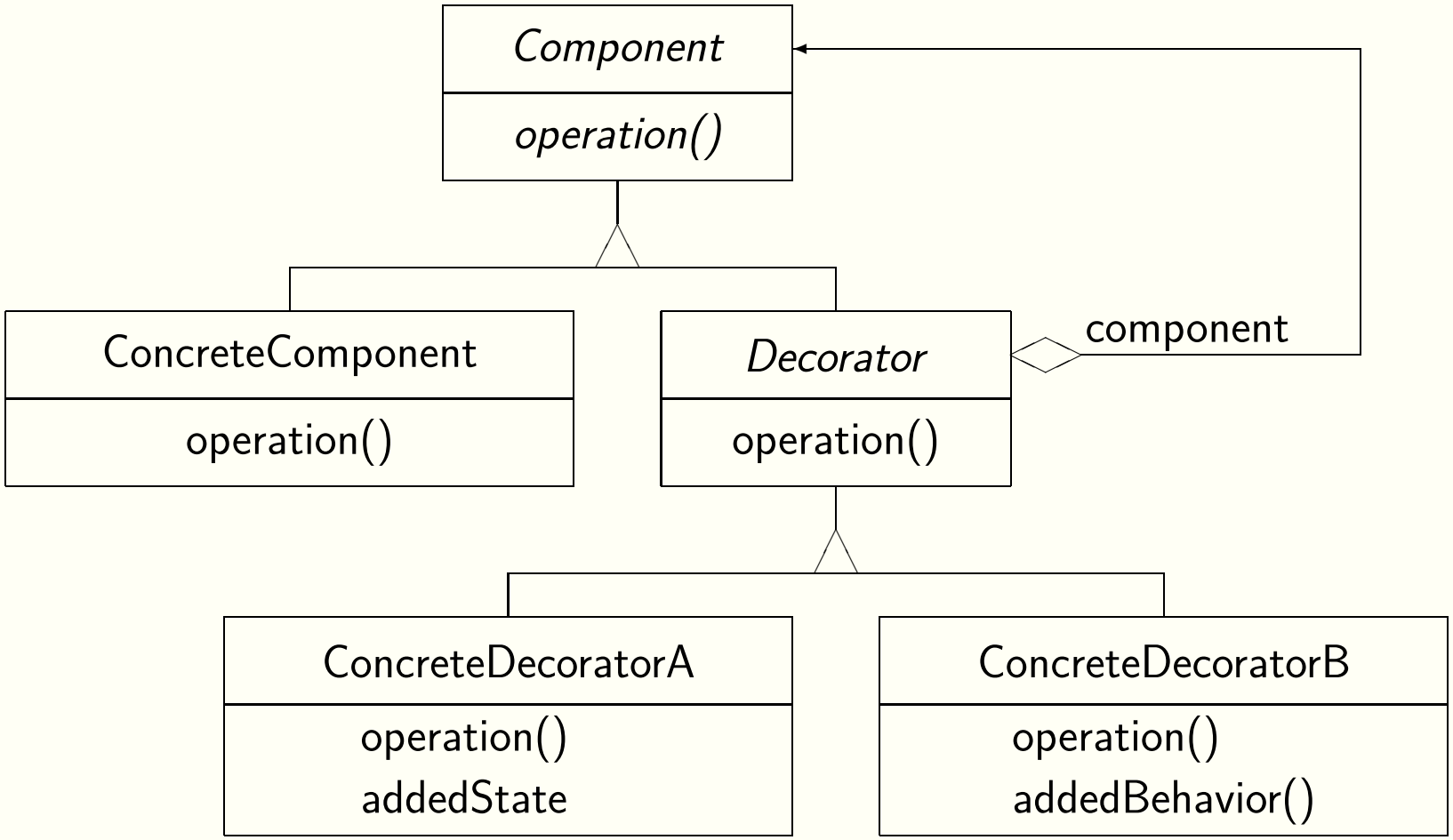
wenn Erweiterung durch Vererbung unpraktisch
(Vermeidung vieler Unterklassen; Vererbung unmöglich)

Beispiel

```
interface Window { void show (String text); }
class WindowImpl implements Window {
    public void show(String text) { ... }
}
abstract class WinDecorator implements Window {
    protected Window win;
    public void show(String text) { win.show(text); }
}
class ScrollBar extends WinDecorator {
    public ScrollBar(Window w) { win = w; }
}
...

    Window myWindow = new WindowImpl(); // no scroll bar
    myWindow = new ScrollBar(myWindow); // add scroll bar
```

Struktur


Component
operation()
ConcreteComponent
operation()
Decorator
operation()
component
ConcreteDecoratorA
operation()
addedState
ConcreteDecoratorB
operation()
addedBehavior()

Eigenschaften

mehr Flexibilität als statische Vererbung,
Verantwortlichkeiten dynamisch dazu oder weg

vermeidet Klassen, die weit oben in der Klassenhierarchie mit vielen Methoden überladen sind

Instanzen von „Decorator“ haben andere Identitäten als Instanzen von „ConcreteComponent“
→ nicht auf Objektidentität verlassen

oft viele kleine Objekte
→ einfach konfigurierbar, aber schwer wartbar

Implementierungshinweise

abstrakte Klasse „Decorator“ nicht nötig, aber sinnvoll

„Component“ so klein wie möglich halten

gut geeignet zur Erweiterung der Oberfläche,
schlecht geeignet für inhaltliche Erweiterungen,
auch schlecht geeignet für umfangreiche Objekte

Aufgabe: Alternative zur Vererbung

Warum gilt das Decorator-Pattern als Alternative zur Vererbung?

A: Weil Methodenaufrufe an andere Klassen weitergeleitet werden.

B: Weil man Decorator-Objekte wie Component-Objekte verwenden kann.

C: Weil man dieselbe Flexibilität wie durch Vererbung erhält.

D: Weil die Vererbung durch Aufrufweiterleitung nachgebildet wird.

Vererbung versus Delegation

```
class A {
    public void x() { z(); }
    protected void z() { /* A-Code */ ... }
}

class B extends A {
    protected void z() { /* B-Code */ ... }
    public void y() { delegate.x(); }
    private A delegate = new A();
}
```

Vererbung: `new B().x()` $\rightarrow$ B-Code

Delegation: `new B().y()` $\rightarrow$ A-Code

Proxy (Surrogate)

Zweck: Platzhalter für Objekt, kontrolliert Zugriffe

zahlreiche **Anwendungsgebiete:**

Remote-Proxy kontaktiert Objekt in anderem Namensraum

Virtual-Proxy erzeugt Objekt bei Bedarf

Protection-Proxy kontrolliert Objektzugriffe

Smart-Reference ersetzt einfache Zeiger, z. B. für

- Mitzählen von Referenzen (reference counting)
- Laden persistenter Objekte beim ersten Zugriff
- Verhindern mehrerer gleichzeitiger Zugriffe

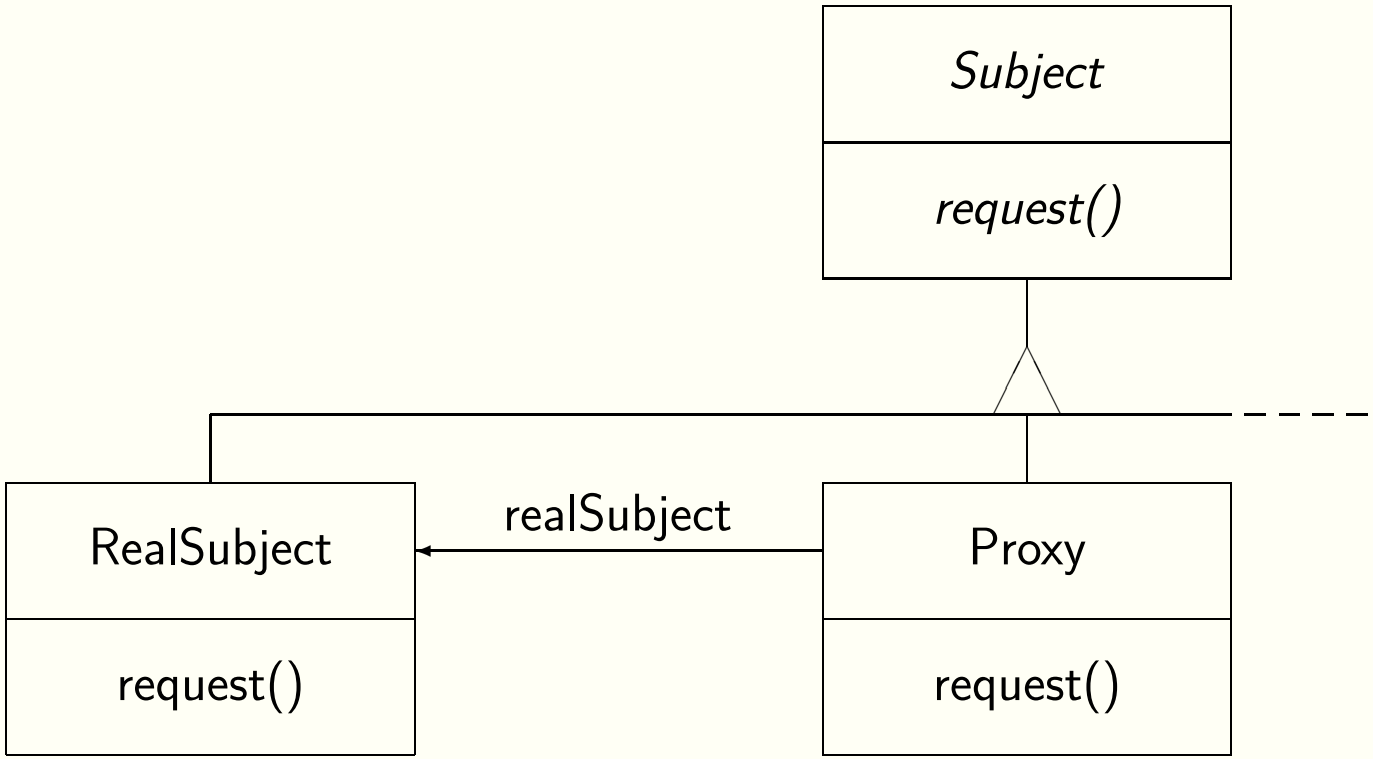
Beispiel

```
interface Something { void doSomething(); }

class ExpensiveSomething implements Something {
    public void doSomething() { ... }
}

class VirtualSomething implements Something {
    private ExpensiveSomething real = null;
    public void doSomething() {
        if (real == null)
            real = new ExpensiveSomething();
        real.doSomething();
    }
}
```

Struktur


Subject
request()
RealSubject
request()
realSubject
Proxy
request()

Implementierungshinweise

Proxy	verwaltet Referenz auf „RealSubject“, ersetzt eigentliches Objekt (Ersetzbarkeit), kontrolliert Zugriffe auf eigentliches Objekt, weitere Verantwortlichkeiten von Art abhängig

mehrere Proxies können verkettet sein

Darstellung nicht existierender Objekte

manchmal kennt „Proxy“ nur „Subject“

selbe Struktur wie Decorator möglich, aber anderer Zweck

Iterator (Cursor)

Zweck: sequentieller Zugriff auf Elemente eines Aggregats

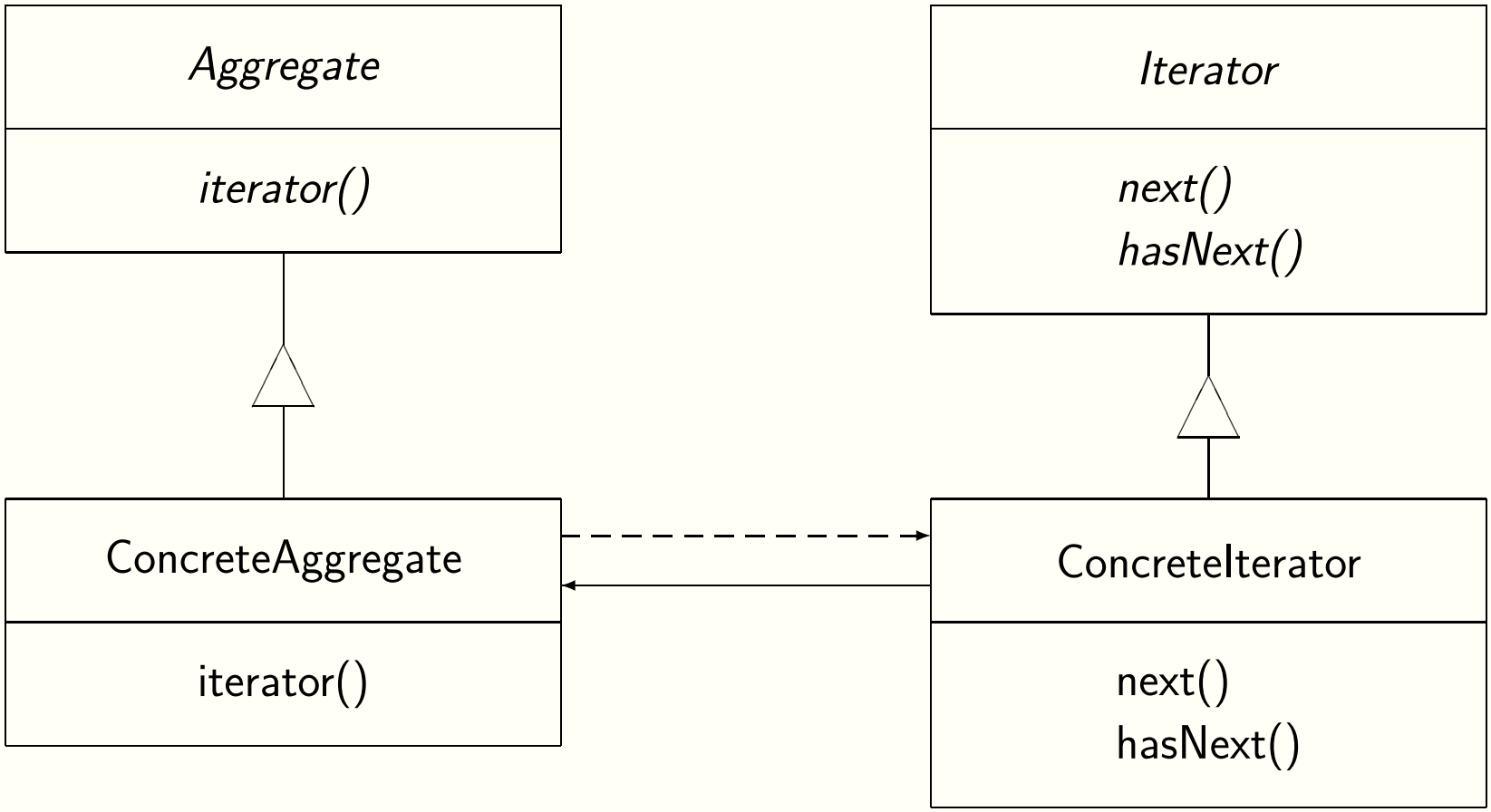
Anwendungsgebiete:

Zugriff auf Aggregatinhalt wobei innere Darstellung gekapselt bleibt

mehrere Abarbeitungen des Aggregatinhalts

einheitliche Schnittstelle für Abarbeitung verschiedener Aggregatstrukturen (polymorphe Iterationen)

Iterator: Struktur



Eigenschaften

unterstützt unterschiedliche Arten der Abarbeitung von Aggregaten
(mehrere Iteratorklassen pro Aggregatklasse)

vereinfacht Schnittstelle von *Aggregate*

mehrere gleichzeitige Abarbeitungen möglich

Iterator: Implementierungshinweise

externe Iteratoren flexibler, interne Iteratoren einfacher

extern: Anwender holt nächstes Element (siehe Beispiele)
intern: Iterator wendet Operation auf alle Elemente an (map, fold, ...)

interne Iteratoren besser wenn komplexe Beziehungen zwischen Elementen bei Abarbeitung zu berücksichtigen sind

Algorithmus zum Durchwandern des Aggregats in Aggregat oder Iterator definiert (beides gleichzeitig wenn Iterator innere Klasse des Aggregats)

Aggregatänderungen während der Abarbeitung beachten (robuster Iterator)

auch auf leeren Aggregaten brauchbar

Danke für Ihre Mitarbeit

Viel Erfolg

bei Abgabegespräch, Prüfung und
Anwendung des Erlernten in der Praxis