Decorator (Wrapper)

Zweck: gibt Objekten dynamisch neue Verantwortlichkeiten

Alternative zur Vererbung

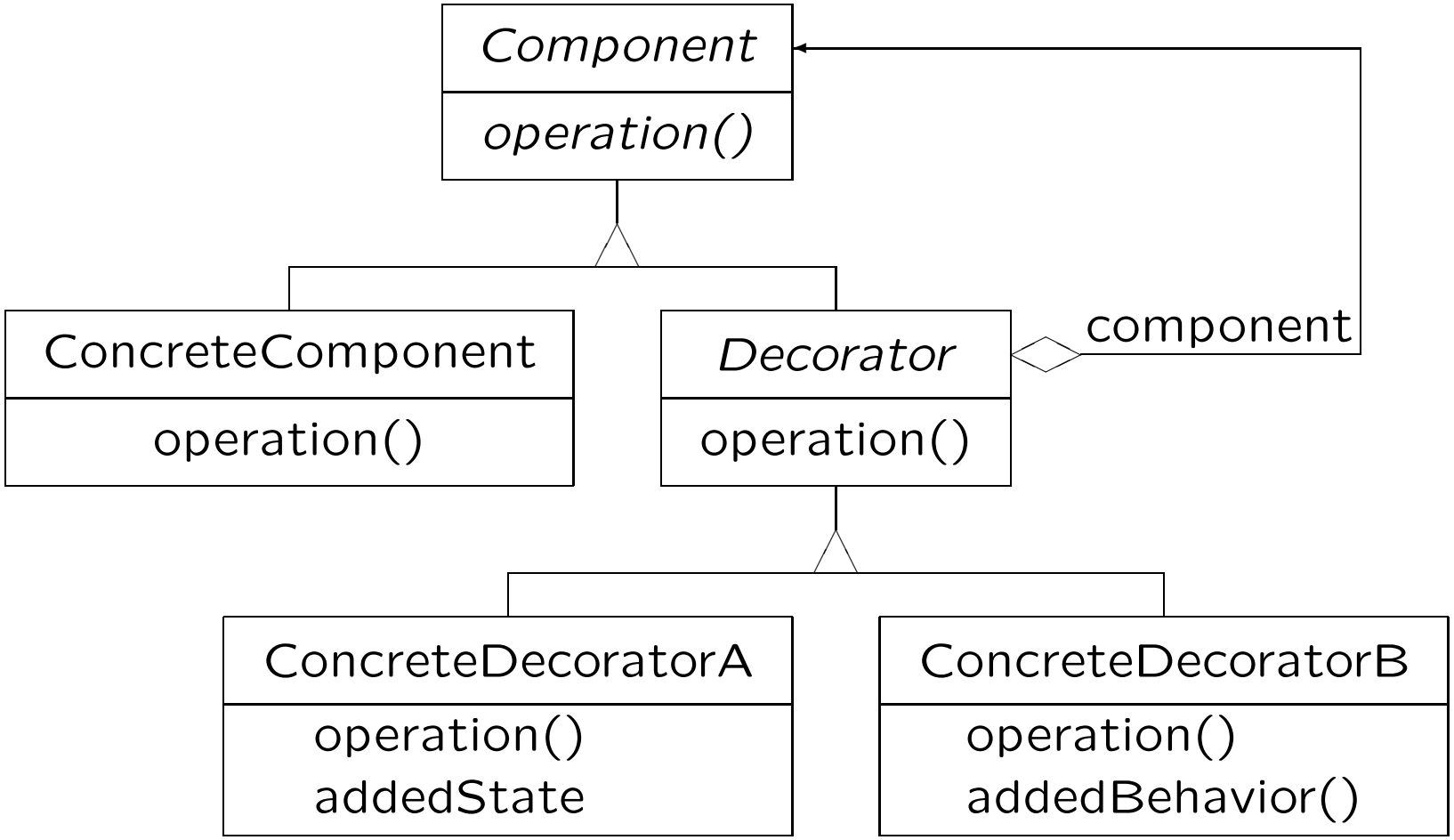
Anwendungsgebiete:

- dynamisches Hinzufügen von Verantwortlichkeiten ohne Beeinflussung anderer Objekte
- Verantwortlichkeiten, die wieder entzogen werden können
- wenn Erweiterung durch Vererbung unpraktisch
  (Vermeidung vieler Unterklassen; Vererbung unmöglich)

Decorator: 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
```

Decorator: Struktur



Decorator: Eigenschaften

- mehr Flexibilität als statische Vererbung

  Verantwortlichkeiten dynamisch dazu oder weg

- vermeidet Klassen, die weit oben in der Klassenhierarchie mit Eigenschaften überladen sind

- Instanzen von „Decorator“ haben andere Identität als Instanzen von „ConcreteComponent“

  $\Rightarrow$ nicht auf Objektidentität verlassen

- oft viele kleine Objekte

  $\Rightarrow$ einfach konfigurierbar, aber schwer wartbar

Decorator: 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

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

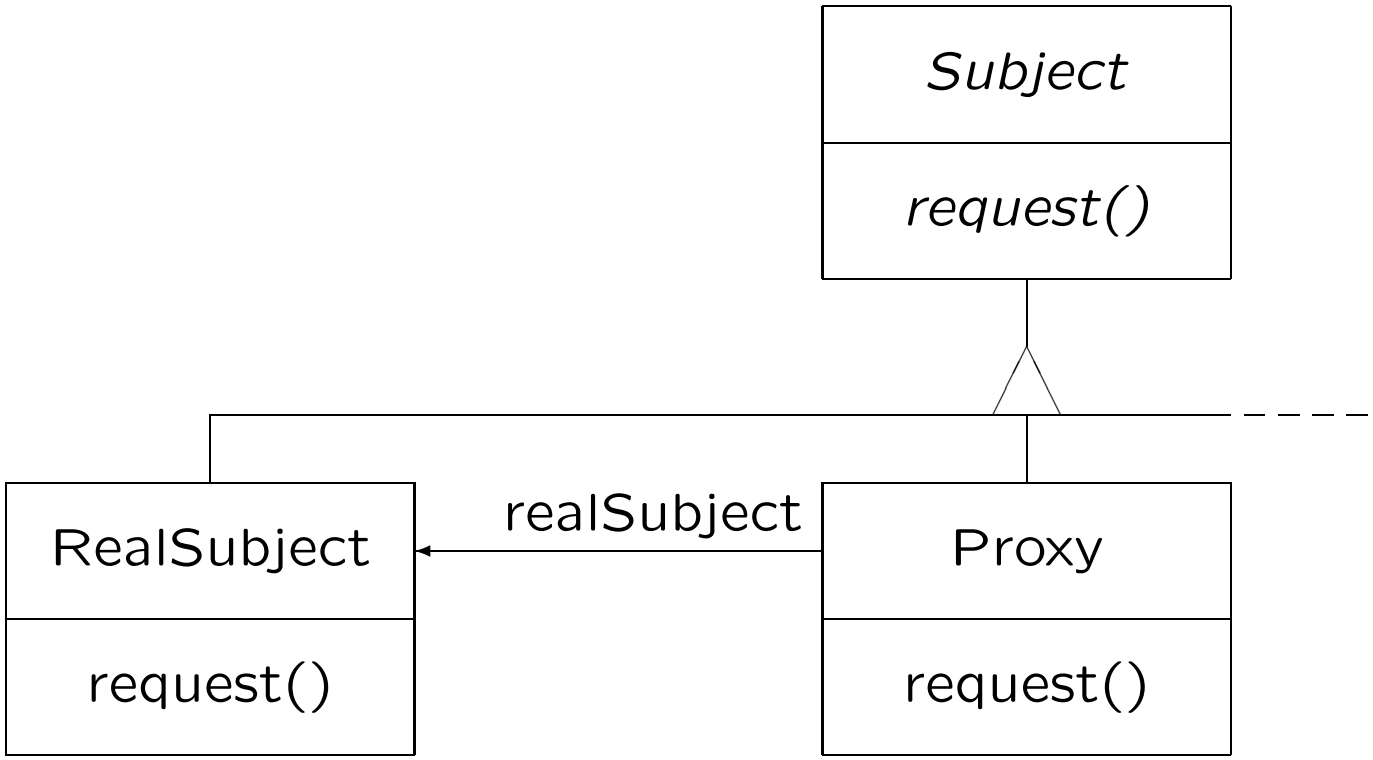
Smart Reference ersetzt einfache Zeiger, z. B. für

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

Proxy: 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();
    }
}
```

Proxy: Struktur



Proxy: 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
- manchmal kennt „Proxy“ nur „Subject“
- Darstellung nicht existierender Objekte
- selbe Struktur wie Decorator möglich, aber anderer Zweck

Template Method

Zweck: definiert Grundgerüst eines Algorithmus,

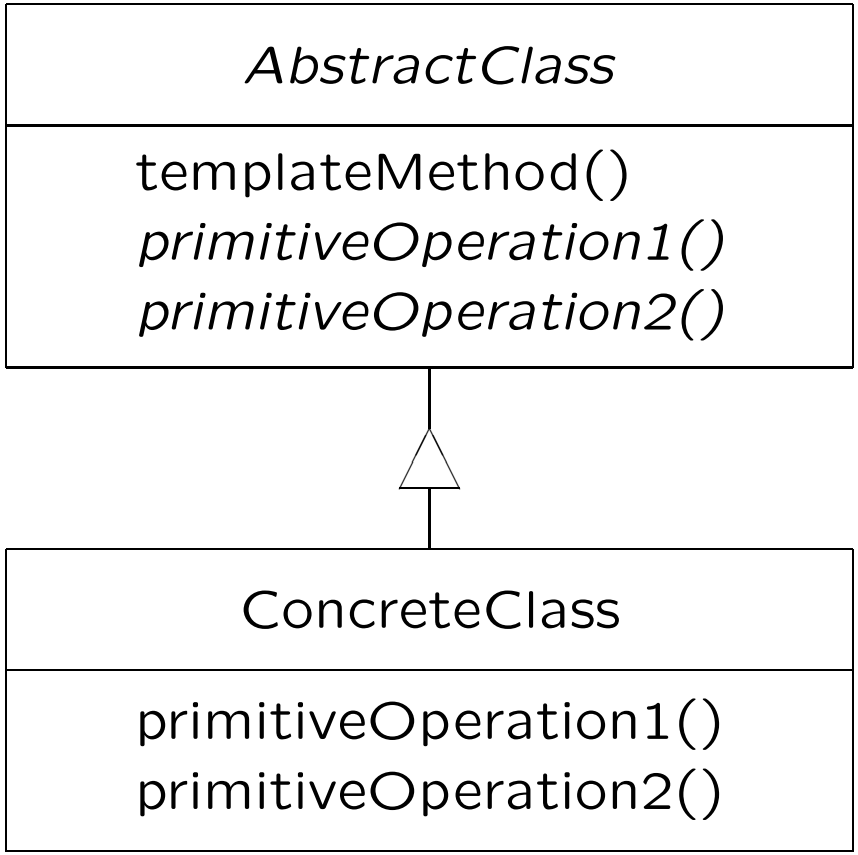
Implementierung einzelner Schritte in Unterklasse

Anwendungsgebiete:

- unveränderlicher Teil eines Algorithmus einmal implementiert, veränderliche Teile in Unterklassen
- gemeinsames Verhalten mehrerer Unterklassen lokal zusammengefasst (Refaktorisierung)
- mögliche Erweiterungen durch *hooks* kontrollieren

  hooks in Unterklassen überschreibbar

Template Method: Struktur



Template Method: Eigenschaften

- fundamentale Technik zur direken Codewiederverwendung
- Oberklasse ruft Methoden der Unterklasse auf (umgekehrte Kontrollstruktur)
- neben konkreten Operationen in „AbstractClass“ meist nur eine von mehreren Arten von Operationen aufgerufen:
  - abstrakte primitive Operationen
  - Factory Methods
  - hooks

Template Method: Implementierung

- möglichst wenige primitive Operationen
- primitive Operationen sind `protected`
- primitive Operationen sind `abstract`, wenn sie überschrieben werden müssen
- Template Method selbst kann `final` sein