Singleton

Zweck: Klasse hat nur eine Instanz und erlaubt globalen Zugriff

Anwendungsgebiete:

- es soll genau eine global zugreifbare Instanz geben
- Klasse soll erweiterbar sein und Ersetzbarkeit garantieren

Struktur: Klasse `Singleton` mit statischer Methode `instance`
(gibt einzige Instanz zurück)

Singleton: Eigenschaften

- erlaubt kontrollierten Zugriff auf einzige Instanz
- vermeidet unnötige Namen im System
  (keine globale Variable)
- unterstützt Vererbung
- leicht änderbar, wenn mehrere Instanzen benötigt
  Klasse hat dennoch Kontrolle über Anzahl der Instanzen
- flexibler als statische Methoden
- Pattern oder Anti-Pattern?

Singleton: Beispiel (1)

einfache Lösung mit Lazy-Initialization:

```
class Singleton {
    private static Singleton singleton = null;
    protected Singleton() {}
    public static Singleton instance() {
        if (singleton == null)
            singleton = new Singleton();
        return singleton;
    }
}
```

Lösung für mehrere alternative Implementierungen ungeeignet und Lazy-Initialization für Nebenläufigkeit ungeeignet

Singleton: Beispiel (2)

```
class Singleton {
    private static Singleton singleton = null;
    public static int kind = 0;
    protected Singleton() { ... }
    public static Singleton instance() {
        if (singleton == null)
            switch (kind) {
            case 1:  singleton = new SingletonA(); break
            case 2:  singleton = new SingletonB(); break
            default: singleton = new Singleton();
            }
        return singleton;
    }
}
class SingletonA extends Singleton { SingletonA() { ... } }
class SingletonB extends Singleton { SingletonB() { ... } }
```

Singleton: Beispiel (3)

```
class Singleton {
    protected static Singleton singleton = null;
    protected Singleton() { ... }
    public static Singleton instance() {
        if (singleton == null) singleton = new Singleton();
        return singleton;
    }
}
class SingletonA extends Singleton {
    protected SingletonA() { ... }
    public static Singleton instance() {
        if (singleton == null) singleton = new SingletonA();
        return singleton;
    }
}
```

Decorator (Wrapper)

Zweck: gibt Objekten dynamisch neue Verantwortlichkeiten

Alternative zur Vererbung

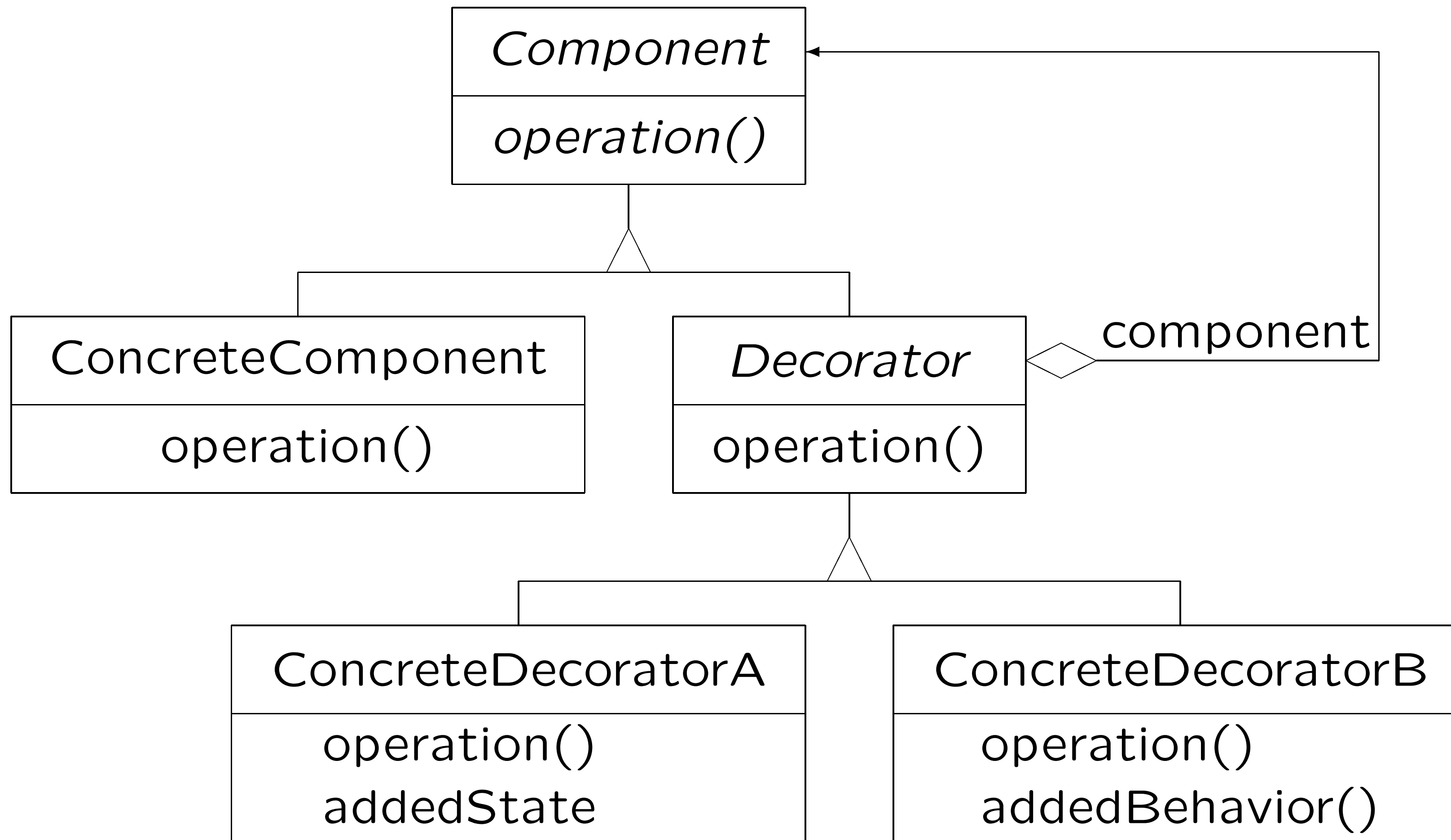
Anwendungsgebiete:

- 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

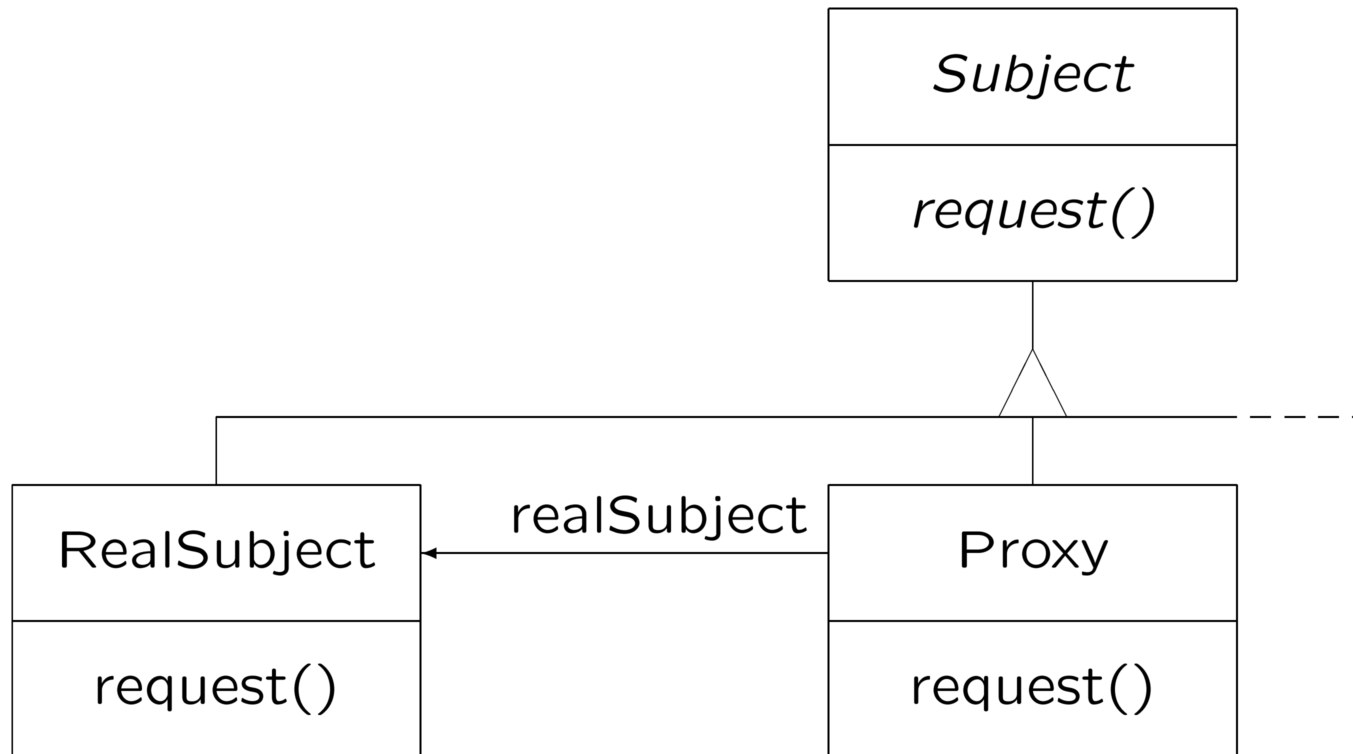
Smart Reference ersetzt einfache Zeiger, z. B. für

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