Factory Method (Virtual Constructor)

Zweck: Definition einer Schnittstelle für Objekterzeugung

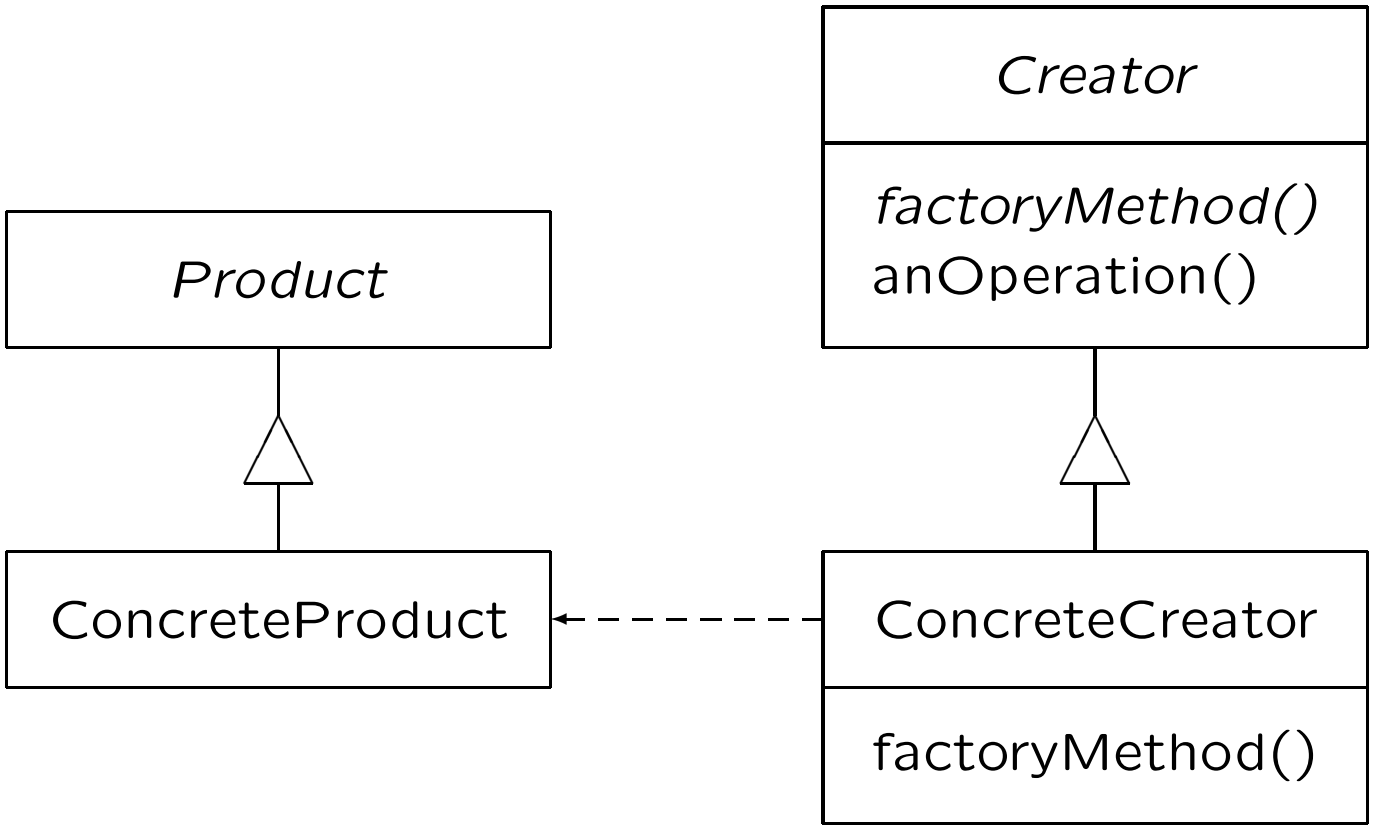
Anwendungsgebiete:

- Klasse neuer Objekte bei Objekterzeugung unbekannt
- Unterklassen sollen Klasse neuer Objekte bestimmen
- Klassen delegieren Verantwortlichkeiten an Unterklassen
  (Wissen um Unterklasse soll lokal bleiben)

Factory Method: Beispiel 1

```
abstract class Document { ... }
class Text extends Document { ... }
...   // classes Picture, Video, ...
abstract class DocCreator {
    abstract Document create();
}
class TextCreator extends DocCreator {
    Document create() { return new Text(); }
}
...   // classes PictureCreator, VideoCreator, ...
class NewDocManager {
    private DocCreator c = ...;
    public void set (DocCreator c) { this.c = c; }
    public Document newDoc() { return c.create(); }
}
```

Factory Method: Struktur



Factory Method: Eigenschaften

- Anknüpfungspunkte (hooks) für Unterklassen $\Rightarrow$ flexibel

  Entwicklung von Unterklassen vereinfacht

- verknüpfen parallele Klassenhierarchien
  (Creator-Hierachie mit Product-Hierarchie)

  Beispiel: `generiereFutter` vom Typ `Futter` in `Tier` (abstr.)
  erzeugt in `Rind` neue Instanz von `Gras`
  und in `Tiger` neue Instanz von `Fleisch`

- oft große Anzahl an Unterklassen nötig

Factory Method: Beispiel 2

Anwendung einer Factory Method für lazy initialization

```
abstract class Creator {
    private Product product = null;
    protected abstract Product createProduct();
    public Product getProduct() {
        if (product == null)
            product = createProduct();
        return product;
    }
}
```

ConcreteProduct kann in Java nicht als Typparameter angegeben werden (da nach `new` kein Typparameter erlaubt)

Prototype

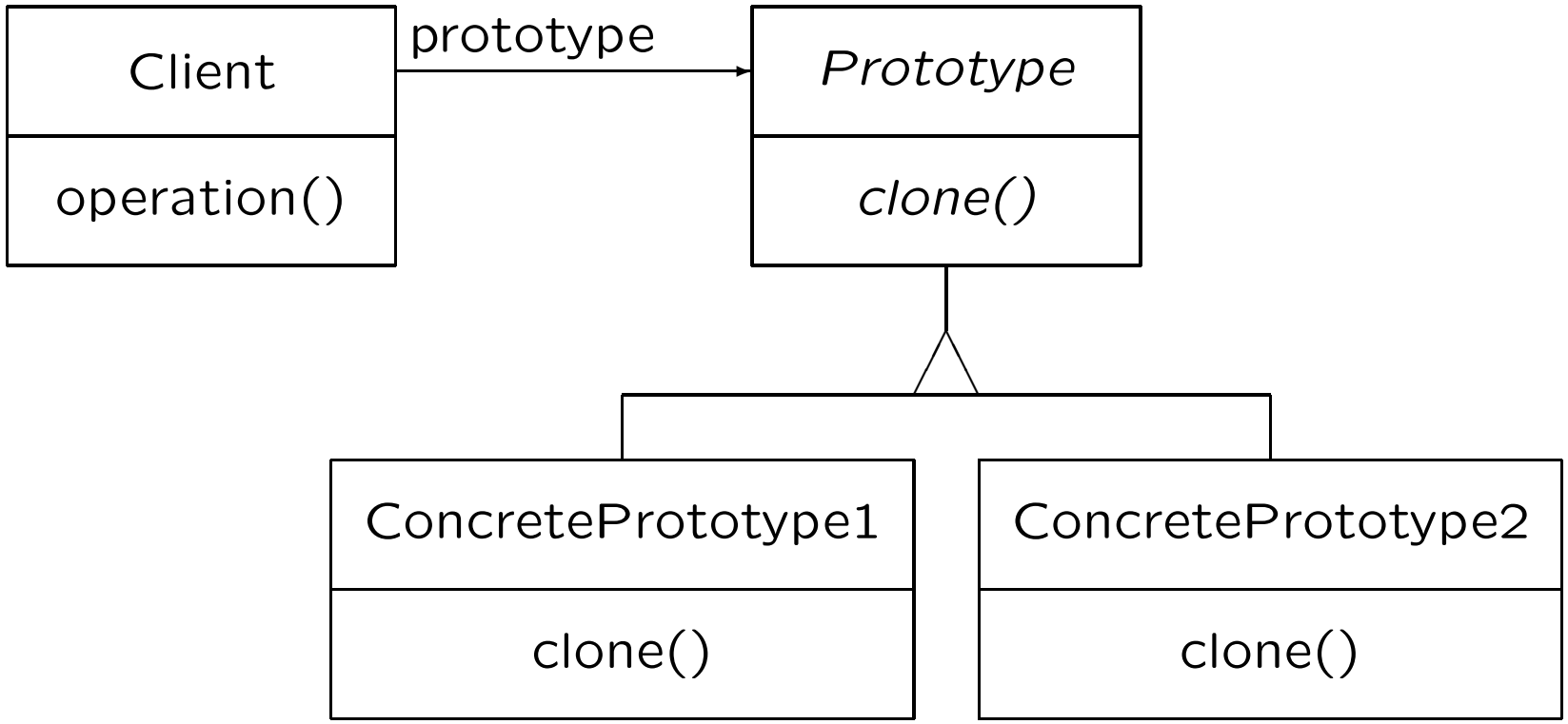
Zweck: Prototyp-Objekt spezifiziert Art eines neuen Objekts
Objekterzeugung durch Kopieren des Prototyps

Anwendungsgebiete:

- Klasse des neuen Objekts erst zur Laufzeit bekannt
- Creator-Klassenhierarchie parallel zu Product-Klassenhierarchie (also Factory Method) soll vermieden werden
- jede Instanz hat einen von wenigen Zuständen

  $\Rightarrow$ Kopieren des Prototyps im gewünschten Zustand einfacher als Konstruktoraufruf

Prototype: Struktur



Prototype: Eigenschaften

- verstecken Product-Klassen vor Anwendern

  geänderte Product-Klassen beeinflussen Anwender nicht

- Prototypmenge dynamisch änderbar (Klassenstrukt. nicht)

- Prototypen dynamisch änderbar (Klassen nicht)

  in hochdynamischen Systemen: Verhalten durch Objektkomposition statt Klassendefinition festlegbar

- vermeiden große Anzahl an Unterklassen

- erlauben dynamische Konfiguration von Programmen auch in Sprachen wie C++

Prototype: Implementierungshinweise

- `clone` in Java für flache Kopien in `Object` vordefiniert
  (wenn `Cloneable` implementiert)
- Erzeugen tiefer Kopien schwierig — zyklische Strukturen
- Prototyp-Manager zur Verwaltung der Prototypen
- `clone` hat keine (geeigneten) Parameter
  daher oft Initialisierungsmethoden nötig
- von dynamischen Sprachen direkt unterstützt

Decorator (Wrapper)

Zweck: gibt Objekten dynamisch neue Verantwortlichkeiten

Alternative zur Vererbung

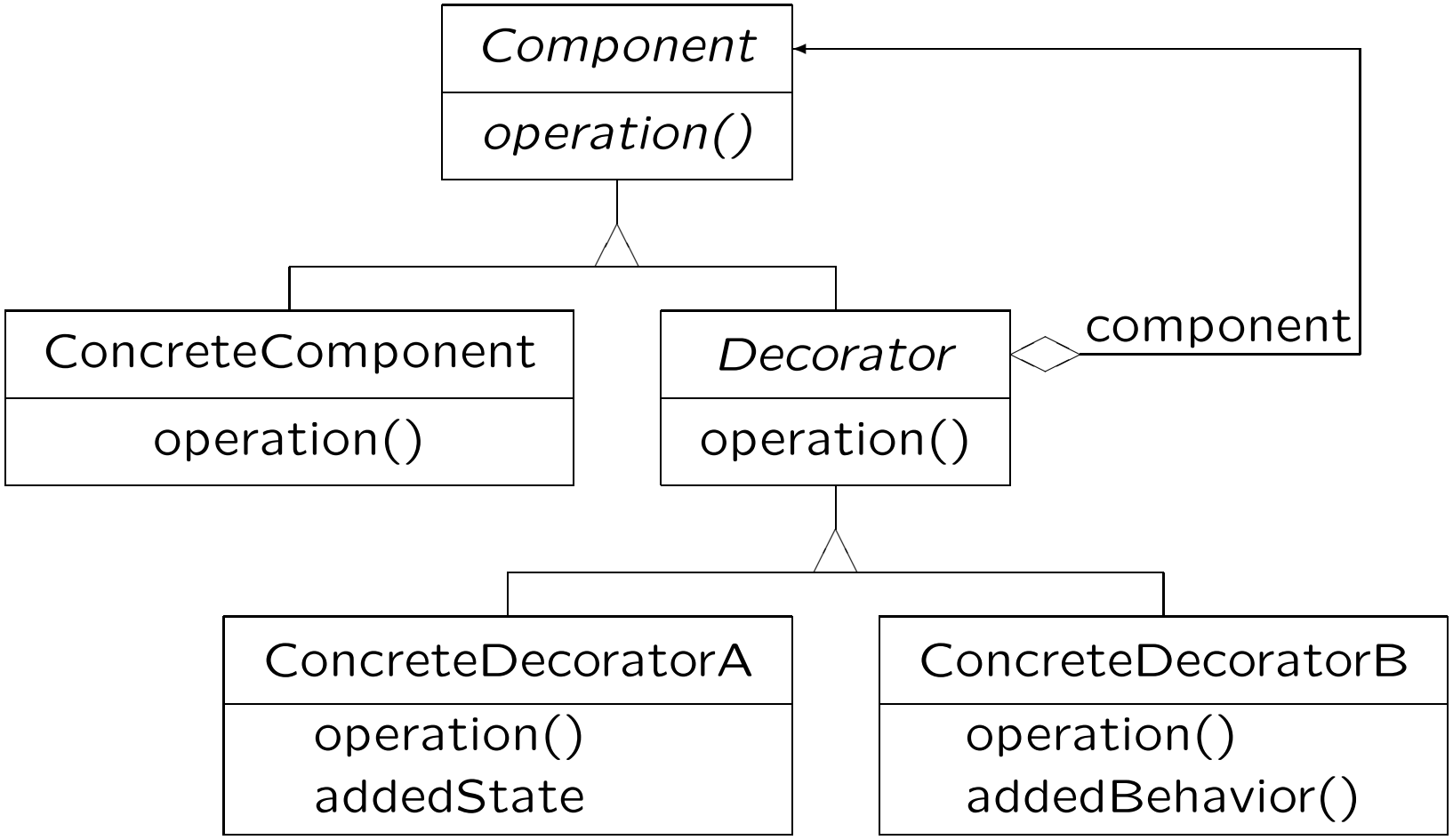
Anwendungsgebiete:

- 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

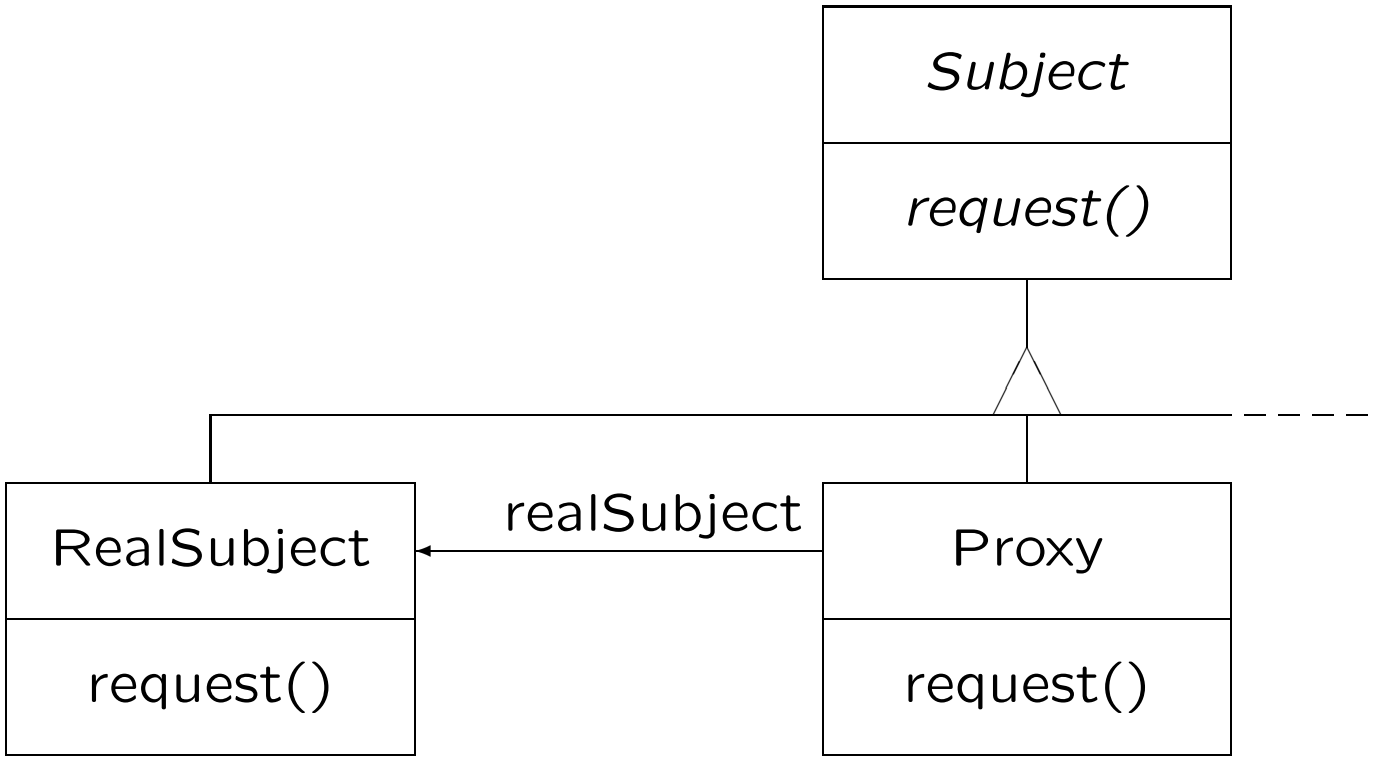
Smart Reference ersetzt einfache Zeiger, z. B. für

- 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