Tagesprogramm

Software-Entwurfsmuster

Factory-Method

Prototype

(Singleton)

Zweck von Entwurfsmustern

Benennen wiederkehrender Probleme und Lösungen

Austasch von Erfahrungen

Wiederverwendung von Erfahrung wo Wiederverwendung von Code versagt (sehr abstrakt, daher häufig wiederverwendbar)

Bestandteile von Entwurfsmustern

Name

Problemstellung

Lösung

Konsequenzen

(Implementierungshinweise)

Factory-Method (Virtual-Constructor)

Zweck: Definition einer Schnittstelle für Objekterzeugung

Anwendungsgebiete:

Klasse neuer Objekte bei Objekterzeugung unbekannt

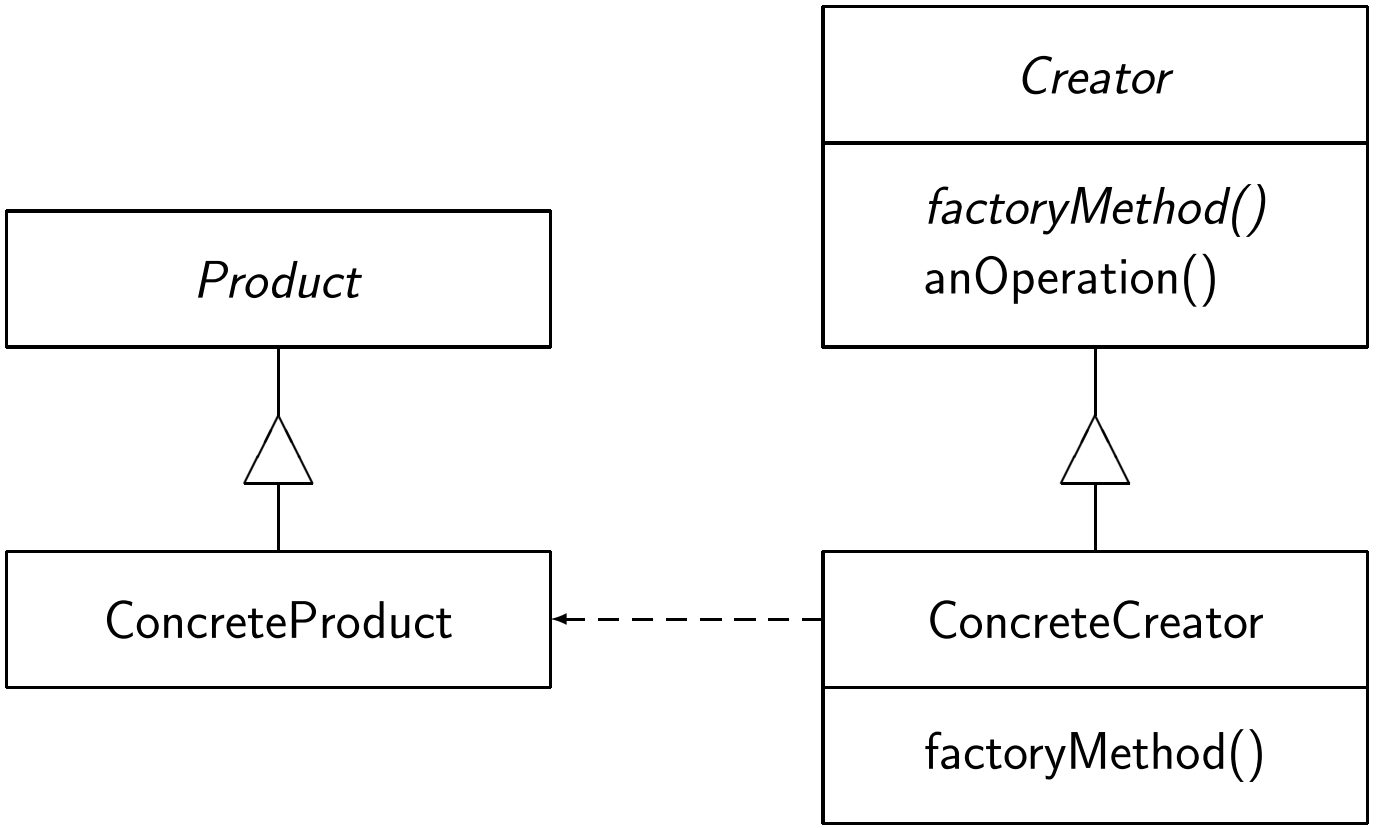
Unterklassen sollen Klasse neuer Objekte bestimmen

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

→ flexibel und Entwicklung von Unterklassen vereinfacht

verknüpfen parallele Klassenhierarchien (Creator- und Product-Hierarchie)

Beispiel:
`generiereFutter` vom Typ `Futter` in `Tier` (abstrakt)
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 ist)

Aufgabe: Generische Methoden

Such Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Fragen:

1. Wozu verwendet man Factory-Method statt Objekterzeugung mit `new`?
2. In welchen Fällen ist die Verwendung von Factory-Method schwierig?
3. Wie würden Sie mit diesen Schwierigkeiten umgehen?

Zeit: 3 Minuten

Prototype

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

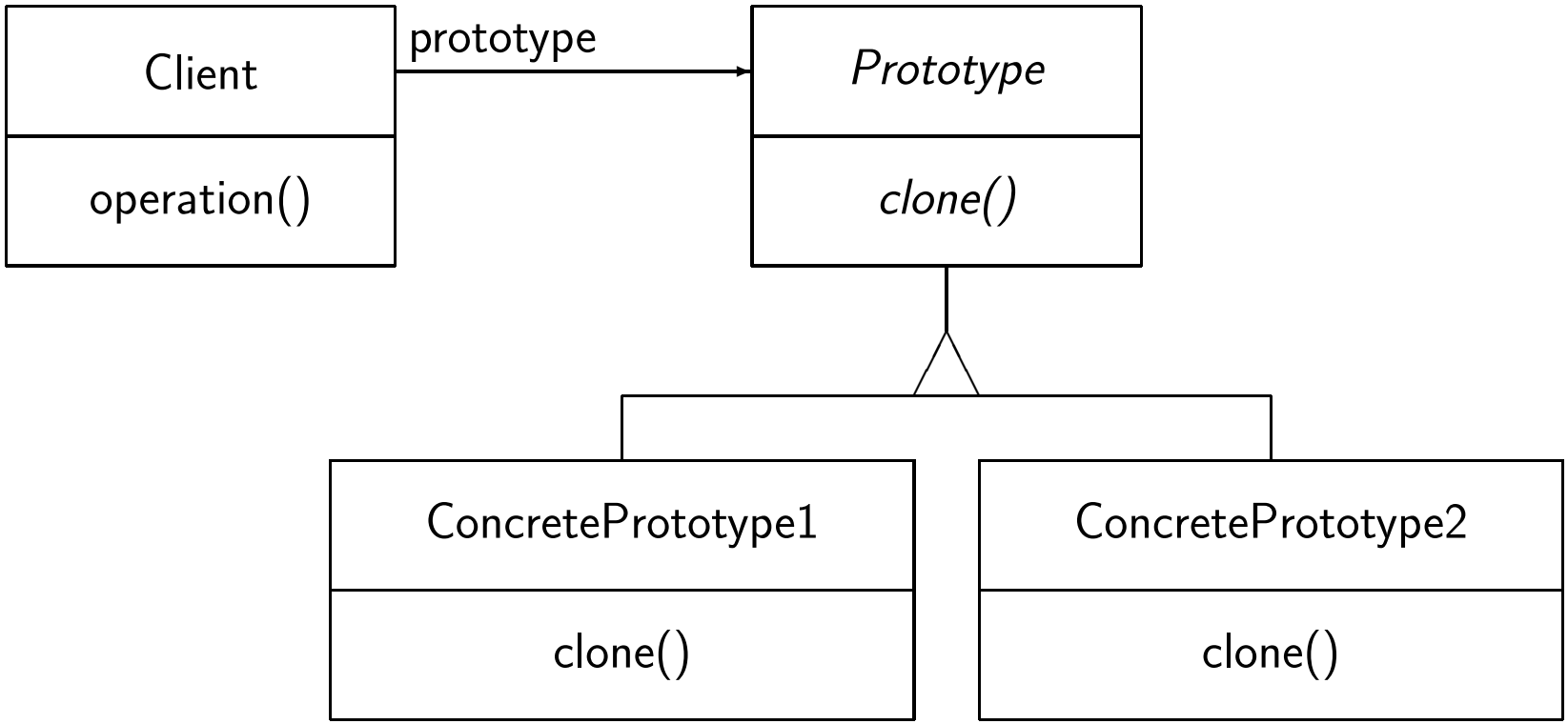
Anwendungsgebiete:

Klasse des neuen Objekts erst zur Laufzeit bekannt

Vermeidung von Creator- parallel zu Product-Hierarchie (Factory-Method)

jede Instanz hat einen von wenigen Zuständen
$\rightarrow$ Kopieren einfacher als Konstruktoraufruf, übernimmt Objektzustand

Prototype: Struktur



Prototype: Eigenschaften

versteckt Product-Klassen (aus Factory-Method) vor Anwendern,
daher beeinflussen geänderte Product-Klassen Anwender nicht

Prototyp-Menge dynamisch änderbar (Klassenstruktur nicht)

Prototypen dynamisch änderbar (Klassen nicht)

$\rightarrow$ in hochdynamischen Systemen:
Verhalten durch Objektkomposition statt Klassendefinition festlegbar

vermeidet große Anzahl an Unterklassen

erlaubt dynamische Konfiguration von Programmen auch in Sprachen wie C++

Prototype: Implementierungshinweise

`clone` in Java für **flache** Kopien in `Object` vordefiniert
(verwendbar wenn `Cloneable` implementiert)

Erzeugen **tiefer** Kopien schwierig — zyklische Strukturen

Prototyp-Manager zur Verwaltung der Prototypen

`clone` hat keine (geeigneten) Parameter, oft Initialisierungsmethoden nötig

von dynamischen Sprachen direkt unterstützt

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

Singleton: Beispiel (1)

einfache Lösung:

```
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

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;
    }
}
```