OOP

Software-Entwurfsmuster

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)

Iterator (Cursor)

Zweck: sequentieller Zugriff auf Elemente eines Aggregats

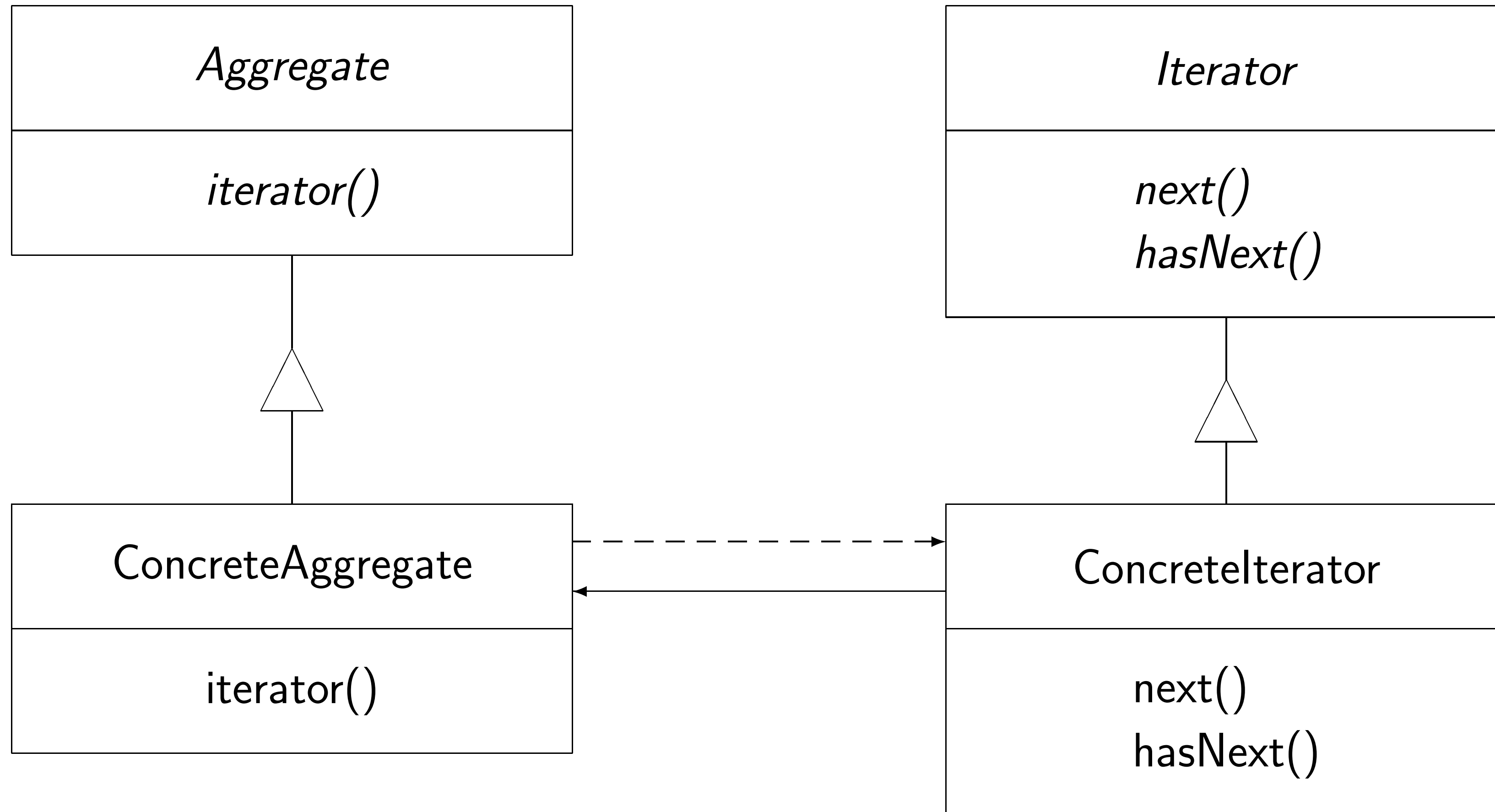
Anwendungsgebiete:

Zugriff auf Aggregatinhalt wobei innere Darstellung gekapselt bleibt

mehrere Abarbeitungen des Aggregatinhalts

einheitliche Schnittstelle für Abarbeitung verschiedener Aggregatstrukturen (polymorphe Iterationen)

Iterator: Struktur



Iterator: 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

Aufgabe: Generische Methoden

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

1. Was unterscheidet interne von externen Iteratoren?
2. Welche Beispiele dafür kennen Sie?
3. Wofür sind interne Iteratoren besser geeignet, wofür externe?

Zeit: 2 Minuten

Factory-Method (Virtual-Constructor)

Zweck: Definition einer Schnittstelle für Objekterzeugung

Anwendungsgebiete:

Klasse neuer Objekte bei Objekterzeugung unbekannt

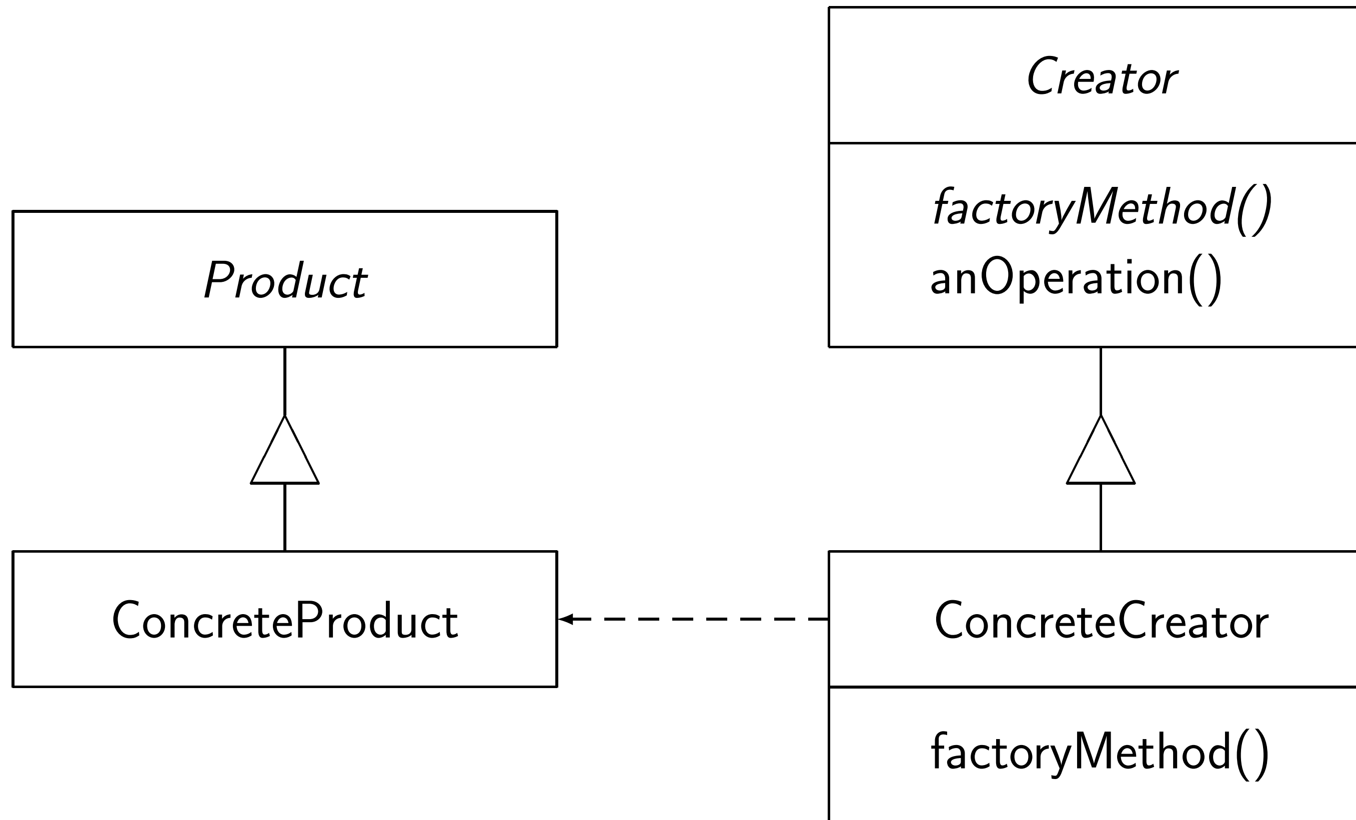
Unterklassen sollen Klasse neuer Objekte bestimmen

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
$\longrightarrow$ 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)

Prototype

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

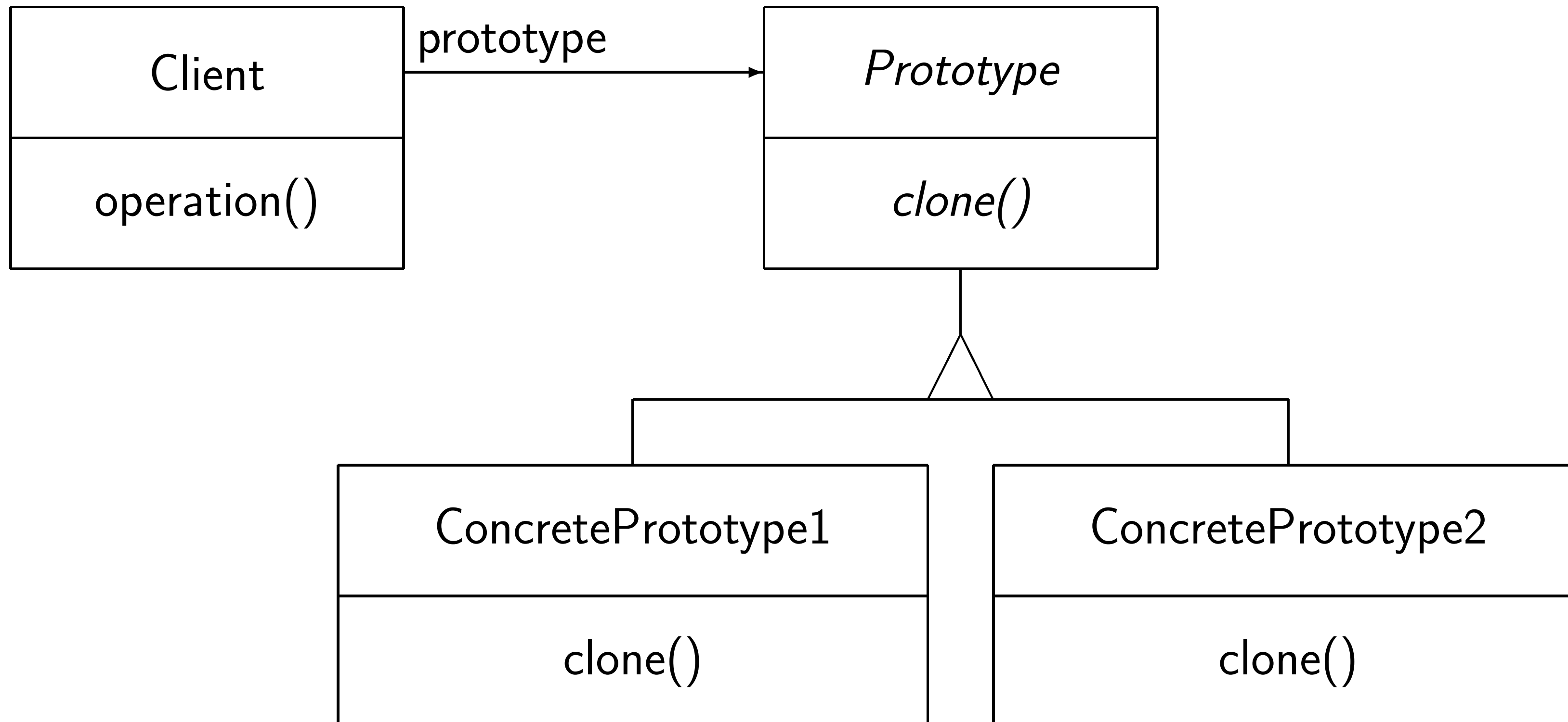
Anwendungsgebiete:

Klasse des neuen Objekts erst zur Laufzeit bekannt

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

jede Instanz hat einen von wenigen Zuständen
$\longrightarrow$ 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)

$\longrightarrow$ 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