Iterator (Cursor)

Zweck: sequentieller Zugriff auf Elemente eines Aggregats

Anwendungsgebiete:

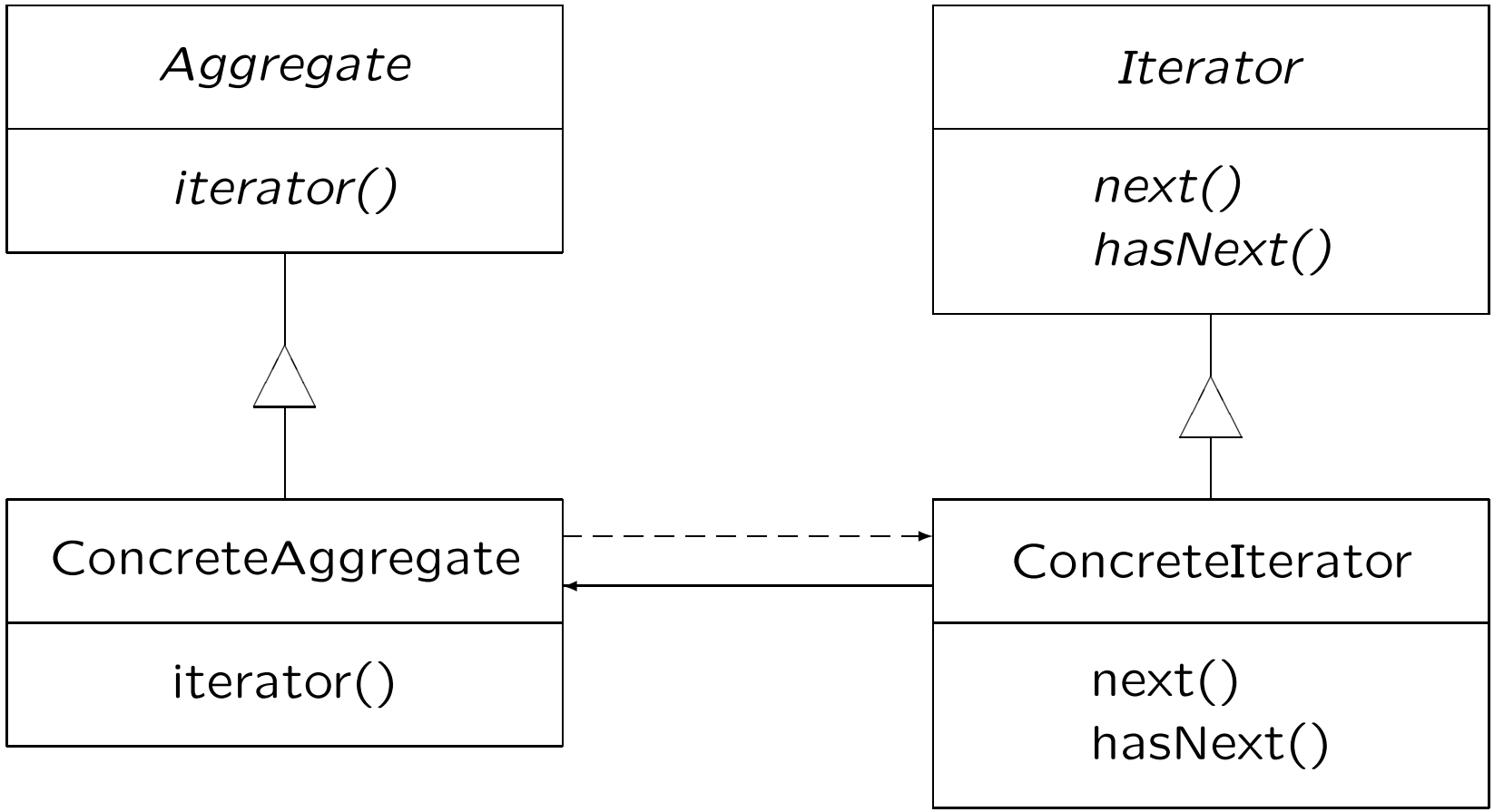
- Zugriff auf Aggregatinhalt

  innere Darstellung bleibt gekapselt

- mehrere Abarbeitungen des Aggregatinhalts

- einheitliche Schnittstelle für Abarbeitung verschiedener Aggregatstrukturen (polymorphe Iterationen)

Iterator: Struktur



Iterator: Eigenschaften

- unterstützen unterschiedliche Arten der Abarbeitung von Aggregaten (mehrere Iteratorklassen pro Aggregatklasse)
- vereinfachen Schnittstelle von *Aggregate*
- mehrere gleichzeitige Abarbeitungen möglich

Iterator: Implementierungshinweise

- externe Iteratoren flexibler, aber weniger einfach

  extern: Anwender holt nächstes Element (siehe Beispiele)

  intern: Iterator wendet Operation auf alle Elemente an

- interne Iteratoren besser wenn Beziehungen zwischen Elementen bei Abarbeitung zu berücksichtigen sind

- Algorithmus zum Durchwandern des Aggregats in Aggregat oder Iterator definiert (innere Klassen für beides)

- Aggregatänderungen während der Abarbeitung berücksichtigen (robuster Iterator)

- auch auf leeren Aggregaten brauchbar

Factory Method (Virtual Constructor)

Zweck: Definition einer Schnittstelle für Objekterzeugung

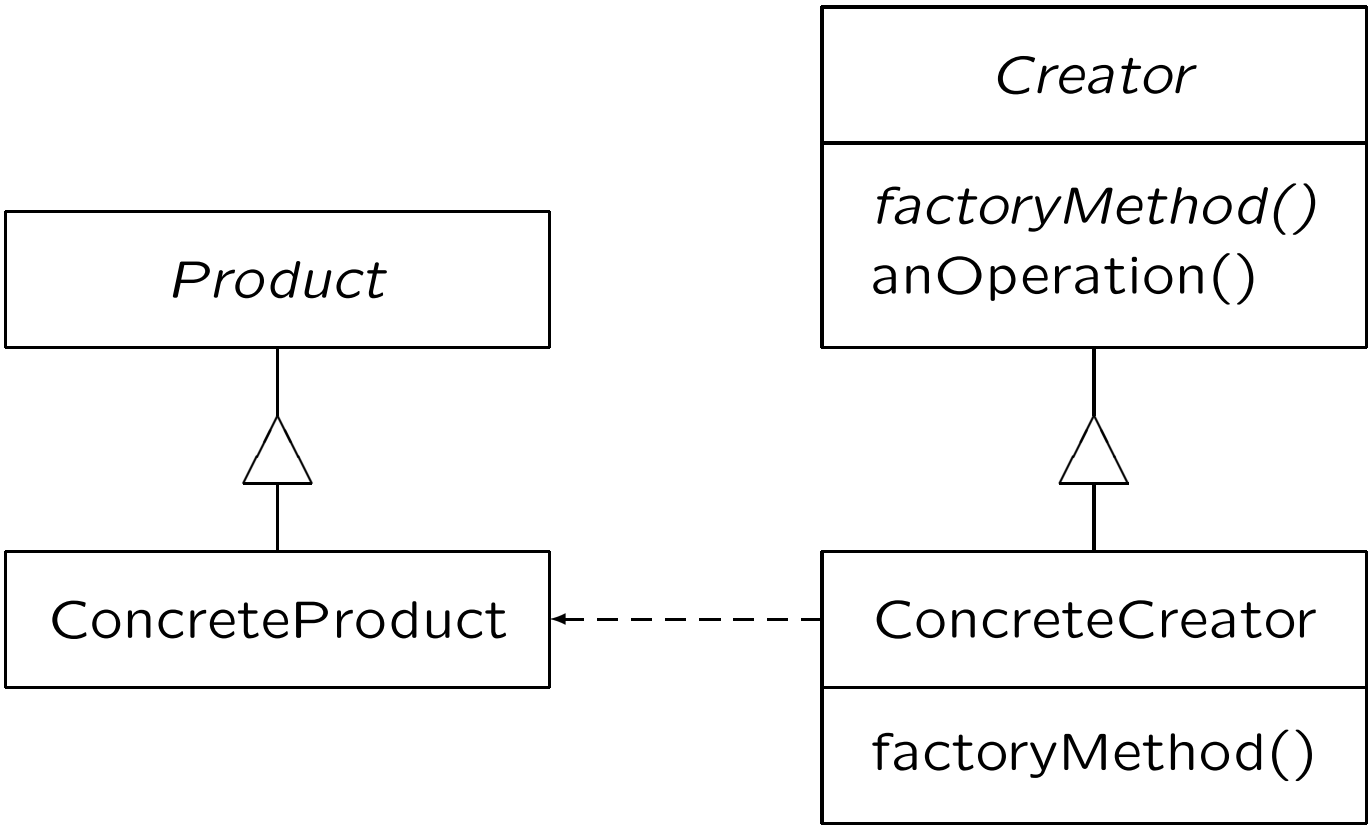
Anwendungsgebiete:

- 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

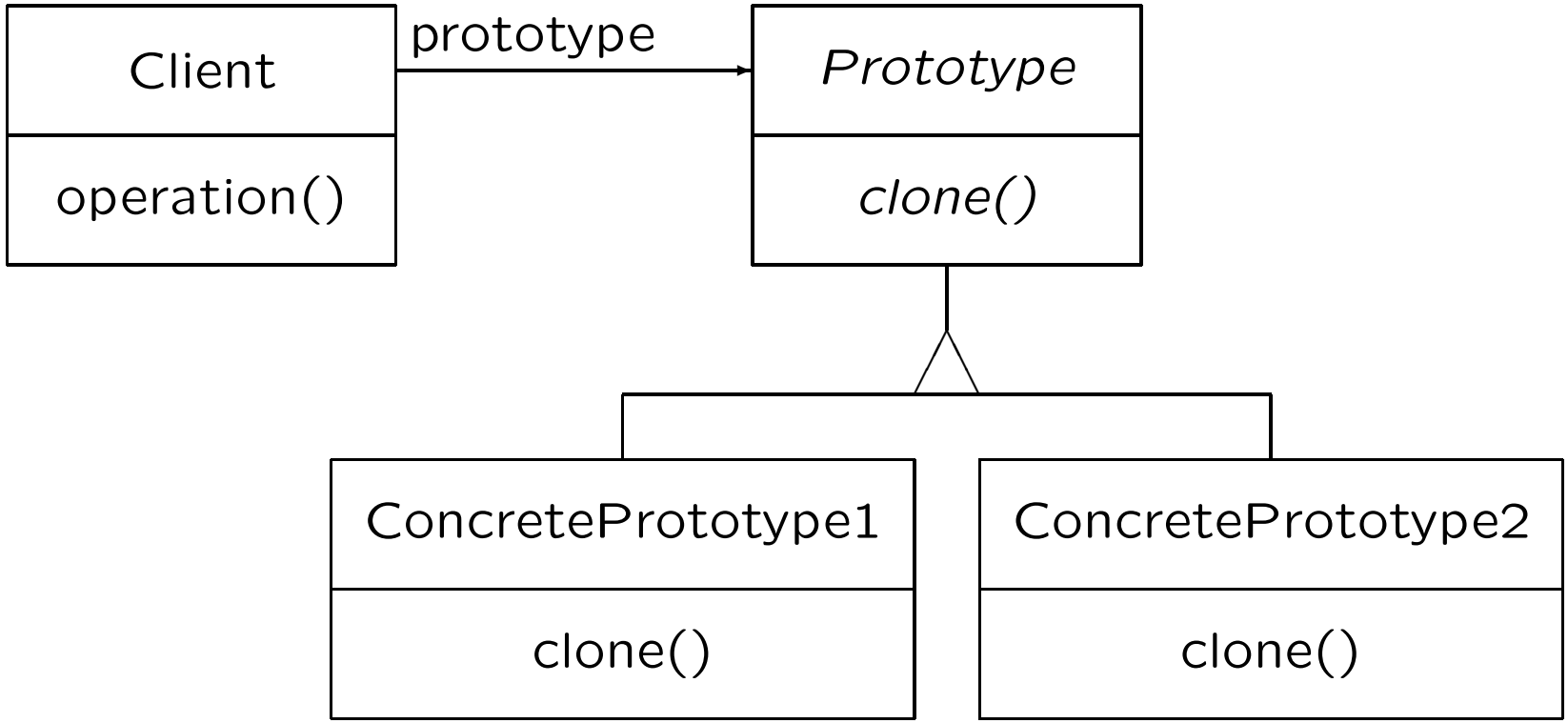
Objekterzeugung durch Kopieren des Prototyps

Anwendungsgebiete:

- Klasse des neuen Objekts erst zur Laufzeit bekannt
- Creator-Hierarchie parallel zu Product-Hierarchie (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
  (verwendbar 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

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;
    }
}
```