# ITERATOR

ein objektorientiertes Verhaltensmuster

von D. Bärthel & C. Haupt

# 1. Zweck

- sequentieller Zugriff auf die Komponenten eines zusammengesetzten Objekts (z.B. eine Liste), ohne die innerer Struktur des Objekts offenzulegen

# 2. Motivation (1/3)

- Zugriff auf Datenstruktur ist
  - unsichtbar, da durch Iterator erfolgt
  - unabhängig von der tatsächlichen Datenstruktur, da Interface der Datenstruktur verallgemeinerte Eigenschaften hat

⇨ Es ist also problemlos möglich die Datenstruktur zu ändern ohne auf den Klient Rücksicht zu nehmen.

# 2. Motivation (2/3)

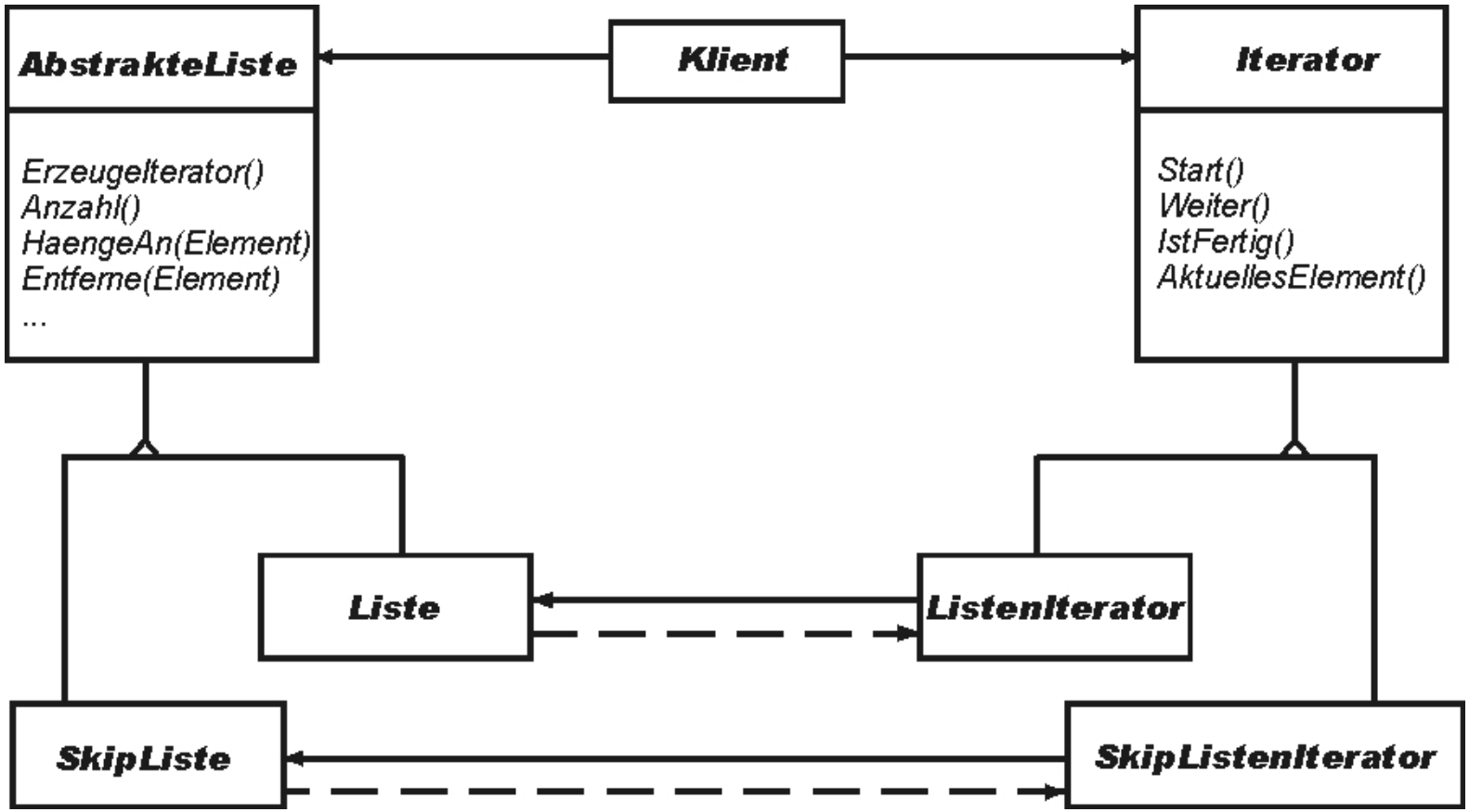
- durch das allgemeine Interface des Iterators kann man verschiedene sequentielle Zugriffsarten implementieren
- Liste-Interface wird dabei nicht überladen
- Voraussetzung:

  Liste-Interface muss so mächtig sein, dass die Iterator-Klassen alle gewünschten Zugriffsarten implementieren können. Für jede neue Zugriffsart benötigt man eine neue Iterator-Klasse.

# 2. Motivation (3/3)

- Information darüber an welcher Stelle in der Liste sich der Iterator gerade befindet wird im Iterator gespeichert

⇨ mehrere (verschiedene) Iteratoren können gleichzeitig auf derselben Liste arbeiten


AbstrakteListe
Erzeugelterator()
Anzahl()
HaengeAn(Element)
Entferne(Element)
...
Klient
Iterator
Start()
Weiter()
IstFertig()
AktuellesElement()
Liste
ListenIterator
SkipListe
SkipListenIterator

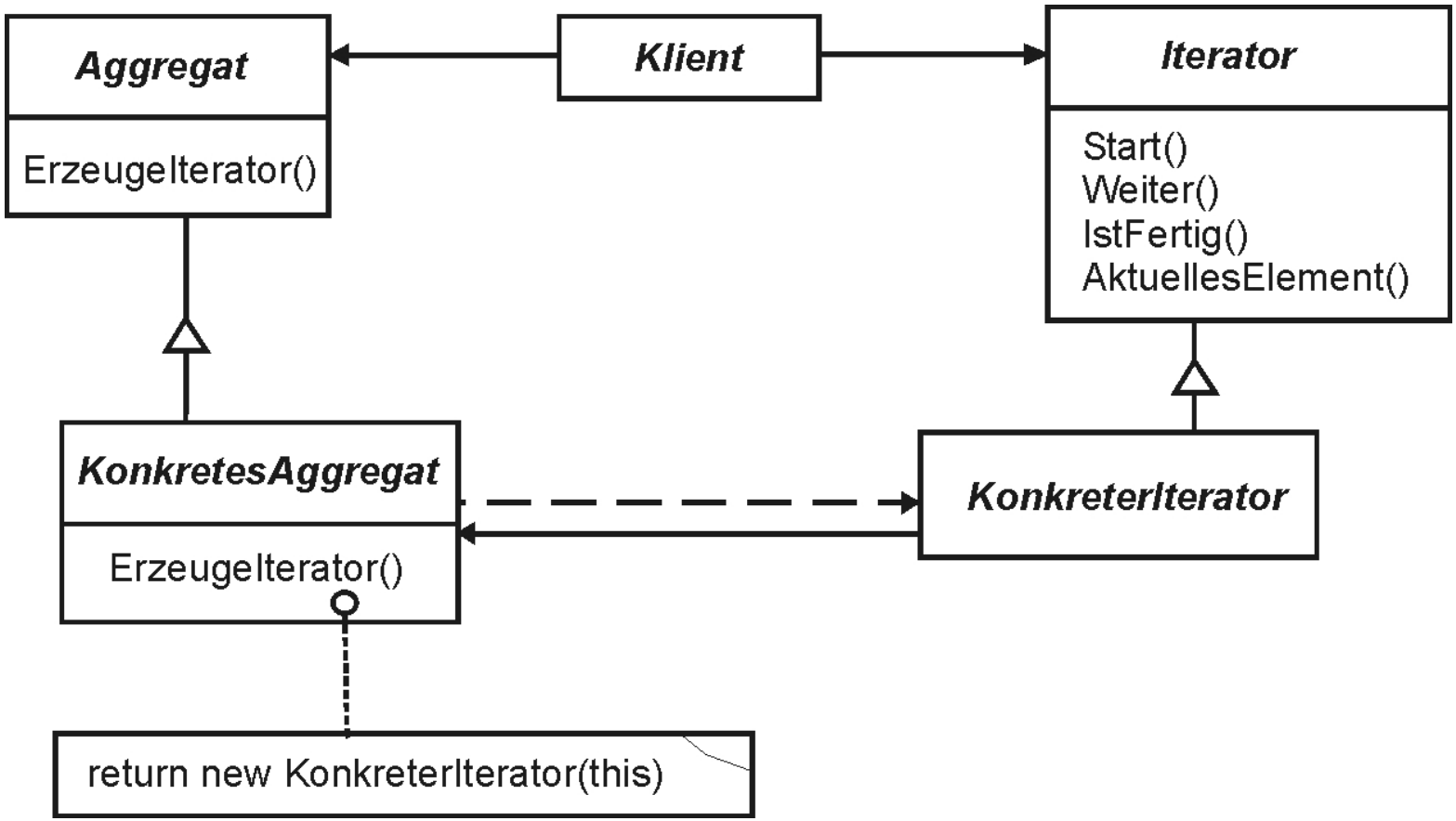
# 3. Anwendbarkeit

Das Iteratormuster wird verwendet

- um Zugriff auf den Inhalt eines zusammengesetzten Objekts zu ermöglichen (ohne interne Struktur offenzulegen)
- um mehrfache gleichzeitige Traversierungen auf zusammengesetzte Objekte zu ermöglichen
- um einheitliche Schnittstelle zur Traversierung unterschiedlicher zusammengesetzter Strukturen anzubieten

  ⇨ Ermöglichung von polymorpher Iteration

# 4. Struktur


Aggregat
ErzeugeIterator()
Klient
Iterator
Start()
Weiter()
IstFertig()
AktuellesElement()
KonkretesAggregat
ErzeugeIterator()
KonkreterIterator
return new KonkreterIterator(this)

# 5. Teilnehmer

- Iterator — definiert Schnittstelle zum Zugriff und zum Durchlaufen von Elementen
- KonkreterIterator — implementiert Iterator-Schnittstelle; verwaltet aktuelle Position während Durchlaufen des Aggregats
- Aggregat — definiert Schnittstelle zum Erzeugen eines Objekts der Klasse Iterator
- KonkretesAggregat — implementiert Operation zum Erzeugen eines konkreten Iterators, in dem es ein Objekt der passenden Konkreter-Iterator-Klasse zurückgibt

# 6. Konsequenzen

- ermöglicht Variationen in der Art des Durchlaufens eines Aggregats
- Iteratoren vereinfachen die Aggregat-schnittstelle
- ein Aggregat kann mehr als einmal gleichzeitig durchlaufen werden

# 7. Implementierung

## 7.1 Steuerung der Iteration

*externer* Iterator: Klient steuert Iteration

Klient treibt Durchlaufen des Aggregats an und verlangt explizit nächstes Element vom Iterator

*interner* Iterator: Iterator steuert Iteration

Klient muss auszuführende Operation übergeben

⇨ Iterator wendet Operation auf jedes Element im Aggregat an

⇨ externe Iteratoren sind flexibler als interne

# 7.2 Definition des Travisierungsalgorithmus

- durch Aggregat oder Iterator möglich

*Aggregat*

- Iterator speichert den Zustand der Iteration

  ⇨ *Cursor*, da Iterator nur aktuelles Element im Aggregat zeigt
- Klient ruft Weiter-Operation auf Aggregat auf
- Übergabe des Cursors als Argument
- Weiter-Operation ändert Cursorzustand

*Iterator*

- Verwendung unterschiedlicher Iterationsalgorithmen für dasselbe Aggregat

# 7.3 Robustheit des Iterators

- Ein robuster Iterator stellt sicher, dass das Einfügen und Entfernen von Objekten nicht mit dem Durchlaufen der Elemente in Konflikt gerät.
- häufig: Iterator beim Aggregat registrieren
- beim Einfügen oder Entfernen wird interner Zustand des Iterators durch Aggregat angepasst oder
- Aggregat verwaltet weitere interne Informationen, um Travesierung zu ermöglichen

# 7.4 Zusätzliche Iterator-Operationen

- besondere Iterator-Funktionen im Einzelfall denkbar und wünschenswert, um auf die Iteration Einfluss zu nehmen
- z.B. Ordnung der Datenstruktur (sortiert) oder

  wahlfreien Zugriff auf Datenstruktur (indiziert)
- Die Struktur muss diese Information zur Verfügung stellen, nicht das einzelne Objekt!

# 7.5 Polymorphe Iteratoren

- über virtuelle Member-Funktion in der ListenKlasse geeigneten Iterator per Fabrikmethode generieren lassen

⇨ Liste bekommt stets einen passenden Iterator
(ohne ListenKlasse zu nutzen)

- in unserem Bsp. ist es die Methode *ErzeugeIterator*()
- der Klient verantwortlich für Freigabe des Speichers für den erzeugten Iterator

# 7.6 Iteratoren mit privilegiertem Zugriff (1/2)

- Erweiterung des Aggregats, das ihn erzeugt
- Aggregat und Iterator miteinander gekoppelt
- in C++: Iterator wird *friend* vom Aggregat

⇨ Aggregat muss Iterator effiziente Implementierung des sequentiellen Durchlaufens ermöglichen

# 7.6 Iteratoren mit privilegiertem Zugriff (2/2)

- Problem: beim Arbeiten mit Exceptions oder mehreren Exit-Punkten in einer Routine besteht die Gefahr den Speicher nicht freizugeben
- Lösung: ein Proxymuster, das auf dem Stack angelegt wird, mit einem Iterator initialisiert wird und sich verhält wie der Iterator. Beim Verlassen des Bezugsbereiches wird das Proxy-Objekt gelöscht und dabei auch der Destruktor für den Iterator aufgerufen

# 7.7 Iteratoren für Komposita

- am einfachsten mit interne Iterator
- aktuelle Position wird durch rekursiven Aufruf des internen Iterators und gleichzeitigem Ablegen des Pfad auf dem Stack gespeichert
- auch als externer Iterator möglich
- speichert Pfad durch ein Komposita

# 7.8 Nulliteratoren

- ist degenerierter Iterator
- dient zu Handhabung von Grenzbedingungen
- Per Definition ist ein Nulliterator **immer** am Ende der Iteration angelangt

  ⇨ *IstFertig*-Operation liefert immer *true*
- z.B. bei Travesierung von baumförmigen Aggregaten

  ⇨ Blattobjekte geben Exemplar eines Nulliterator zurück

  ⇨ ermöglicht Travesierung der gesamten Struktur auf einheitliche Weise zu implementieren

# 8. Beispiel

```
final void outTable(){

    // hier wird die generische liste erzeugt
    LinkedList<CdbResult> liste = new LinkedList<CdbResult>();

    JTable meineRsTab = new JTable();
    liste = rsToArray(getRecordsets("esartikel","artikelnr"));

    // Hier wird der Iterator initiert
    Iterator<CdbResult> it = liste.iterator();


    // Zwei Möglichkeiten, die erste mit Iterator schön einfach...
    // man muss sich um die Struktur und Länge nicht kümmern
    while(it.hasNext()){
        System.out.print(it.next().artikelnr + "\n");
    }
    // hier die klassische Methode mit for-schleife und "länge" der liste
    /*
     * for(int i=0;i<=liste.size()-1;i++){
     *  System.out.print(liste.get(i).artikelnr);
     * }
     */
```

# 9. Schlussbemerkung

- Das Iteratormuster ist dazu da Items aus einer Datenstruktur zurückzuliefern.
- Der Iterator führt keine Operationen auf den Items der Datenstruktur oder den zurückgegebenen Items aus!
- Der Iterator kann aber im Auftrag des Clients derartige Operationen durchführen.
- Der Klient ist dann aber für Sinn oder Unsinn der Operation verantwortlich.
- Mehrere (verschiedene) Iteratoren können gleichzeitig auf einer Datenstruktur operieren.