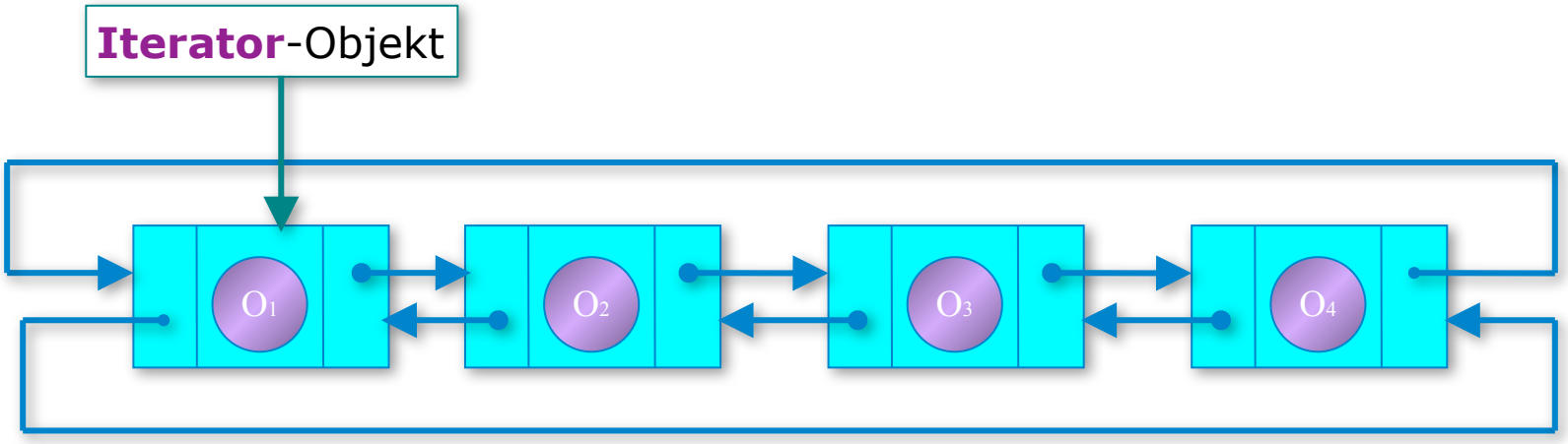
ALP II
Dynamische Datenmengen
Teil III Iteratoren



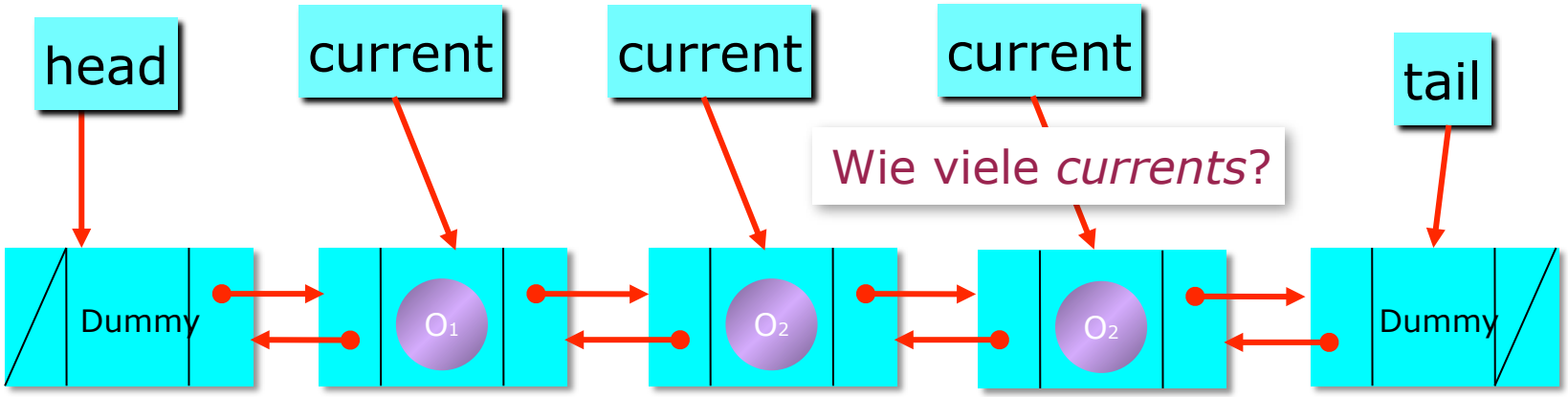
SS 2012

Prof. Dr. Margarita Esponda

Iteratoren

Motivation:

Wir haben für die Implementierung dynamischer Datenmengen mittels verketteter Listen einen **current**-Zeiger als Teil unserer Listen-Objekte verwendet, der sich durch die Liste bewegen kann und diese Position für weitere Operationen bereitstellt.



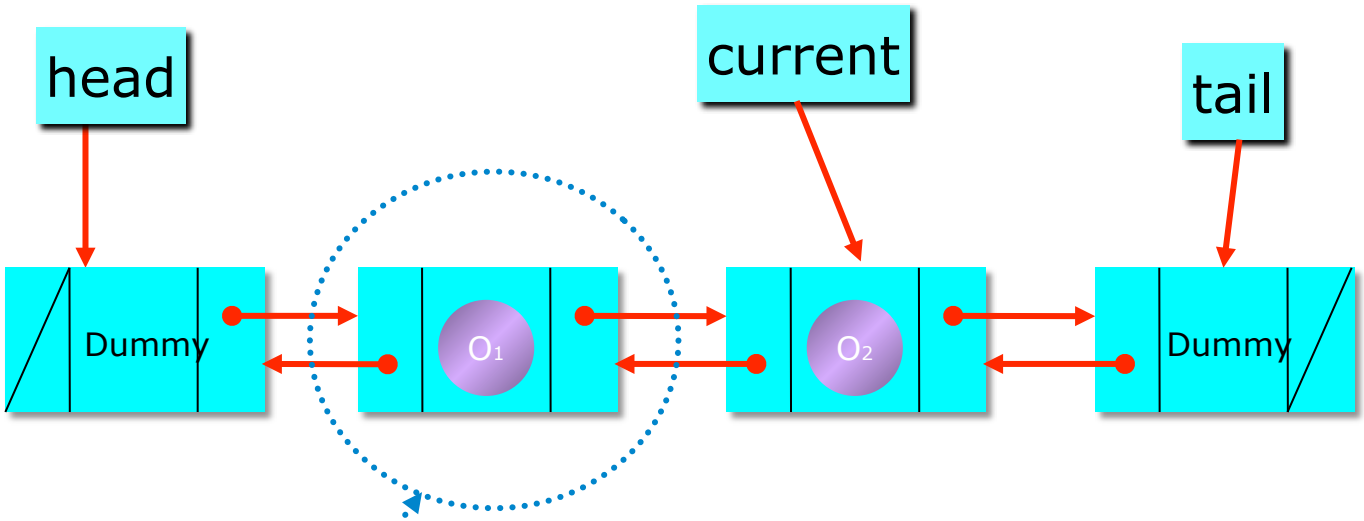
Oft müssen wir für bestimmte Algorithmen mehr als eine Position in einer dynamischen Datenmenge gleichzeitig festhalten.

Iteratoren

Motivation:

Wenn wir den Benutzern unserer Klassen erlauben, beliebige solcher Zeiger selber zu definieren und dadurch direkten Zugriff auf die interne Struktur einer dynamischen Datenmenge (z.B. auf die Listenknoten einer Liste) zu haben, würden wir nicht nur das **Konzept der Kapselung verletzen**, sondern die Gefahr, dass die Benutzer **Fehler** programmieren, wäre zu groß.

Doppelt verkettete Listen




```
class ListNode <T> {
    T element;
    ListNode <T>  next;
    ListNode <T>  prev;
  ...
}
```

Um an einer beliebigen Stelle der Liste Elemente einfügen und löschen zu können, brauchen wir ein Referenz-Objekt (**current**), das sich durch die Liste bewegt.

Doppelt verkettete Listen

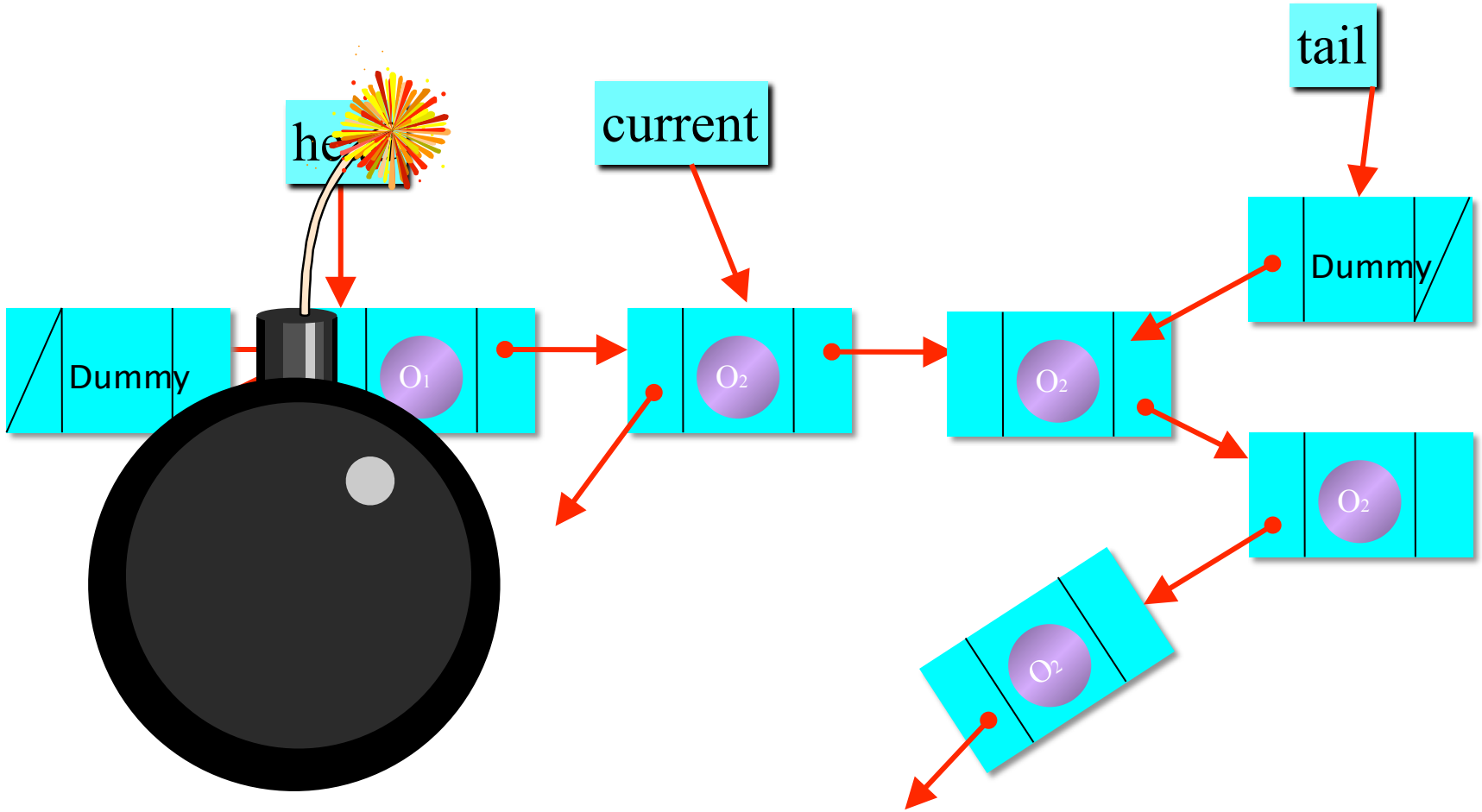
```
class ListNode <T> {
     T element;
     ListNode <T>  next;
     ListNode <T>  prev;
  ...
}
```

```
public class DoubleChainList<T> {

    private ListNode<T>  head;
    private ListNode<T>  tail;
    private ListNode<T>  current;

    public DoubleChainList () {
        head = tail = null;
    }
     /* weitere Methoden */ . . .
}
```

Dynamische Datenstrukturen sind sehr empfindlich.

Ein Programmierer kann z.B. sehr leicht die Knoten-Referenzen zweier verschiedener Listen verwechseln und die Listen unwiederbringlich beschädigen, weil durch die rekursive Definition der Knoten einer Liste, der Zeiger auf den Listenanfang, der Zeiger der die Liste durchläuft und der Zeiger, der zwei Knoten der Liste verkettet, den **gleichen Datentyp haben.**

Doppelt verkettete Listen



Iteratoren

Lösung:

Die objektorientierte Lösung zu diesem Problem ist die Verwendung von Iteratoren.

Iteratoren sind Objekte, die eine Datenstruktur erhalten und Operationen zur Verfügung stellen, mit Hilfe derer man diese Datenstruktur durchlaufen kann.

Iteratoren

Iteratoren sind ein Softwarepattern zur Datenkapselung. Sie abstrahieren von der darunter liegenden Repräsentation der Daten.

Beispiel:

```
public interface Iterator <E> {
        boolean  hasNext();
        E  next();
        void remove();
}
```

Das Iteratormuster

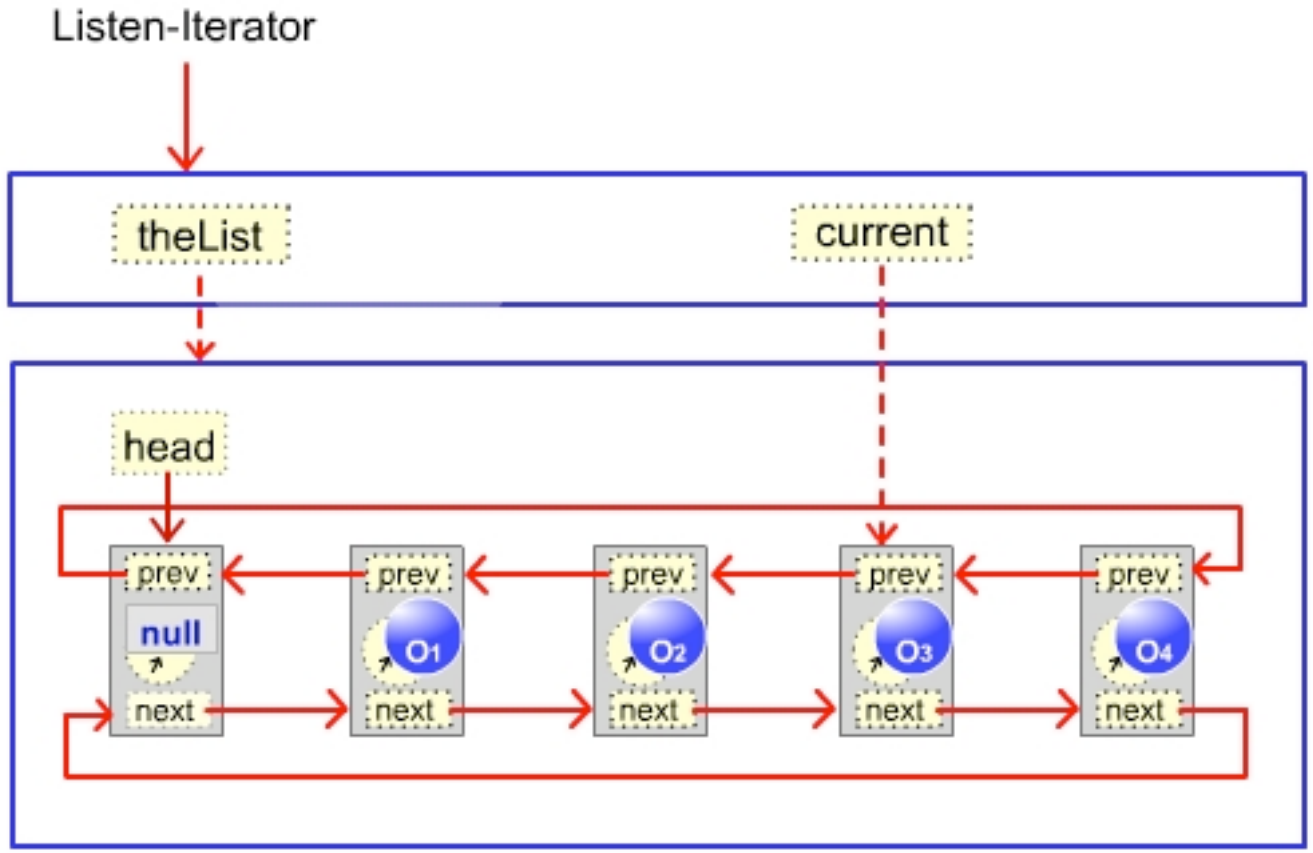
Iterator Pattern

Um eine einheitliche Schnittstelle zur Traversierung unterschiedlicher zusammengesetzter Strukturen (das heißt, um polymorphe Iteration) zu ermöglichen, bieten die Standardbibliotheken vieler Sprachen das Iteratormuster.

Mehrere Iteratoren können mit der gleichen Datenstruktur gestartet werden, und der Benutzer hat somit keinen direkten Zugriff auf die interne Struktur der dynamischen Datenmenge wie z.B. auf den internen **current**-Zeiger einer Liste.

Das Iteratormuster

Grundprinzip



Die Java-Implementierung des Iteratormusters

Die Java-Bibliothek bietet eine Interface für die Implementierung von Iteratoren. Man muss allerdings folgende Probleme beachten:

1. Der Iterator muss immer seinen Zustand innerhalb der Iteration speichern.

2. Die Iteratoren sollen invalidiert werden, wenn sich die Sammlung oder die Datenstruktur durch Einfüge- oder Löschoperationen verändert hat.

   (CurrentModificationException).

Das Iteratormuster von Java

Einfaches Beispiel:

```
public class LinkedList<T> implements Iterable<T> {
        ListNode<T> head;
        ListNode<T> tail;

        public ListIterator<T> iterator() {
            return new ListIterator<T>(head);
        }

         public void insert( T elem ){
            ListNode<T> temp;
            if (head==null) {
                temp = new ListNode<T>(elem,null,null);
            }else{
                temp = new ListNode<T>(elem,head,null);
                head.prev = temp;
            }
            head = temp;
         }
         ...
}
```

Das Iteratormuster von Java

Innere-Klasse

```
...
private class ListNode<E> {
    E elem;
    ListNode<E> next;
    ListNode<E> prev;

    public ListNode(E elem, ListNode<E> next, ListNode<E> prev) {
            this.elem = elem;
            this.next = next;
            this.prev = prev;
    }
    ...
}
...
```

Das Iteratormuster von Java

Einfaches Beispiel:

Innere-Klasse

```
...
 class ListIterator<T> implements Iterator<T> {

      ListNode<T> current;

      ListIterator(ListNode<T> head) {
            current = head;
       }
       public boolean hasNext() {
            return (current.next!= null);
       }
       public T next() {
           if (current == null){
                return null;
           }
          T ret = current.elem;
          current = current.next;
          return ret;
       }
       public void remove(){ // not implemented }
  }
...
```

Das Iteratormuster von Java

Einfaches Beispiel:

```
public class TestListIterator {

    public static void main( String[] args ){
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.insert(1);
        ll.insert(2);
        ll.insert(3);
        ll.insert(4);
        ll.insert(1);
        ll.insert(5);
        ListIterator li = ll.iterator();
        for ( ; li.hasNext() ; ) {
                System.out.println( li.next() );
        }
    }
}
```

Das Iteratormuster von Java

Beispiel:

```
public class PrimesIterator implements Iterator<Integer>{

   int current = 1;

   public boolean hasNext() {
         return true;
   }

   public Integer next() {
          while ( true ) {
                   current++;
                   for ( int i = 2; current % i != 0; i++) {
                            if (i*i >= current) return current;
                   }
           }
   }

  public void remove(){}
}
```

Das Iteratormuster von Java

Beispiel:

```
public class TestPrimesIterator {
  . . .
  public static void main(String[] args) {
      PrimesIterator pi = new PrimesIterator();
      int n = 2;
      while(n<100){
            n = pi.next();
            System.out.println(n);
      }
   }
}// end of class TestPrimesIterator
```

Iteratoren

```
public interface Iterator <E>
```

```
public interface ListIterator <E>
```

```
public interface Iterator <E> {
        void     add( E o )
     boolean     hasNext();
     boolean     hasPrevious()
           E     next();
           E     previous();
        void     remove();
        void     set( E o ) ;
         int     nextIndex();
         int     previousIndex()
}
```