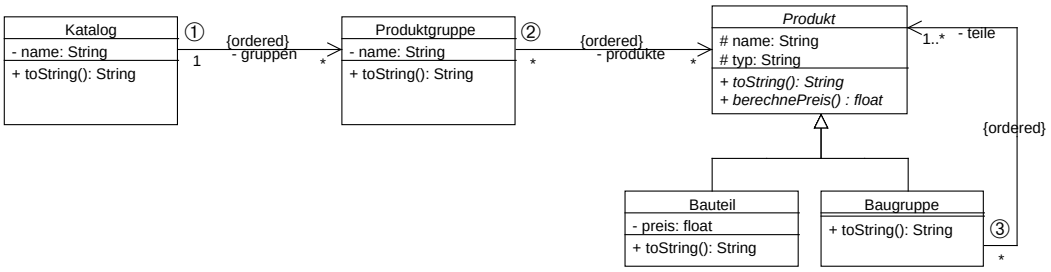
Ludwig-Maximilians-Universität München SS 2006
Institut für Informatik Übungsblatt 9
Prof. Dr. M. Wirsing, M. Hammer, A. Rauschmayer

Informatik II Musterlösung

Zu jeder Aufgabe sind Dateien abzugeben, deren Namen rechts in der Aufgabenüberschrift stehen. Stellen Sie die Dateien in ein extra Verzeichnis (mit beliebigem Namen) und packen Sie dieses zu einem ZIP-Archiv. Geben Sie dieses, wie üblich, per UniWorx ab.

Aufgabe 9-1 Aggregation und Assoziation (10 Punkte, `*.jpg, *.pdf, *.ps, *.java`)

Ein (vereinfachter) Produktkatalog soll abgebildet werden. Ein Katalog besteht aus Produktgruppen, die wiederum Produkte beinhalten. Ein Produkt kann dabei entweder ein einzelnes Bauteil sein, oder eine Baugruppe, die wiederum aus anderen Baugruppen und Bauteilen bestehen kann. Eine UML-Modellierung dieser Beziehungen ist die folgende:



a) Diskutieren Sie, unter welchen Umständen welche Form von Assoziation/Aggregation an den Stellen ① – ③ angebracht ist.

Lösung:

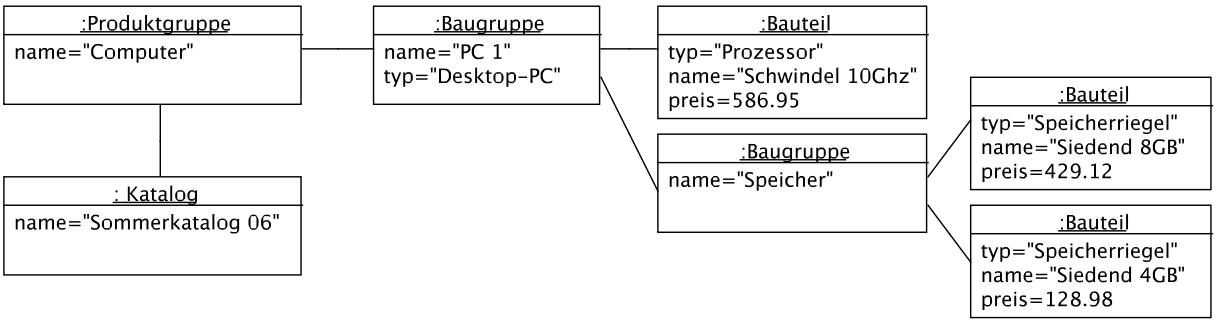
① Hier kann man starke Aggregation verwenden. Ein Katalog besteht aus Produktgruppen, die (durch die Kardinalität 1 ausgedrückt) in keinem anderen Katalog vorkommen sollen. Man kann natürlich auch eine schwächere Variante wählen, aber die Teil-Ganze-Beziehung ist sehr deutlich.

② Hier ist die Lösung vom Zweck des Gesamtsystems abhängig. Werden Produkte anderswo referenziert – beispielsweise in der Lagerverwaltung? Wenn ja, sollte eine einfache Assoziation verwendet werden. Werden die Produkte nur für den Katalog angelegt, ist eine schwache Aggregation sinnvoll. Eine starke Aggregation scheidet aus, weil Produkte offensichtlich mehreren Produktgruppen zugeordnet werden können, wie durch die Kardinalität 1..* ausgedrückt wird.

③ Baugruppen bestehen aus Produkten, hier ist offensichtlich eine Teil-Ganzes-Beziehung gegeben. Da Produkte jedoch – anders als im klassischen Composite-Pattern – mehreren Baugruppen zugeordnet werden können (z.B. könnte ein Prozessortyp in verschiedenen Computertypen verbaut werden), scheidet hier starke Aggregation aus.

b) Implementieren Sie das UML-Diagramm als Java-Klassen. `berechnePreis` soll dabei für ein Produkt die Summe der Preise seiner Teile angeben. `toString` soll die jeweils assoziierten Objekte mit ausgeben. Verwenden Sie Exceptions (z.B. die `IllegalArgumentException`, oder eine eigene Subklasse), um die korrekten Kardinalitäten sicherzustellen. Es sollte nicht möglich sein, einen inkonsistenten Zustand herzustellen.

c) Ein möglicher Zustand des Objektmodells ist im folgenden Objektdiagramm gegeben:



Fügen Sie eine `main`-Methode in die Klasse `Katalog` ein, die den angegebenen Zustand herstellt und auf `System.out` ausgibt.

Lösung: Der kniffelige Teil bei dieser Aufgabe ist, dass eine Baugruppe aus wenigstens einem Produkt bestehen muss. Dies kann dadurch erzwungen werden, dass man im Konstruktor bereits die Liste der zugeordneten Produkte setzt. Es muss allerdings noch geprüft werden, ob die übergebene Liste nicht etwa leer ist.

Zusätzlich ist dafür Sorge zu tragen, daß eine Produktgruppe nur einem Katalog zugeordnet werden kann (lt. UML-Diagramm). Dies kann dadurch erreicht werden, daß eine Builder-Methode `buildProduktgruppe` verwendet wird, so daß nur neue Produktgruppen zu einem Katalog hinzugefügt werden können.

```
package uml;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class Katalog {
    private List<Produktgruppe> gruppen;

    private String name;

    public String toString() {
        StringBuffer ret = new StringBuffer("Katalog " + name + "\n");
        for (Produktgruppe pg : gruppen) {
            ret.append(pg);
        }
        return ret.toString();
    }

    public Produktgruppe buildProduktgruppe(String name) {
        Produktgruppe g = new Produktgruppe(name);
        gruppen.add(g);
        return g;
    }

    public Katalog(String name) {
        this.name = name;
        this.gruppen = new LinkedList<Produktgruppe>();
    }

    /** Teilaufgabe c.) */
    public static void main(String[] args) {
        Katalog k = new Katalog("Sommerkatalog 06");
```

```
        Produktgruppe pg = k.buildProduktgruppe("Computer");
        /* Die Produkte bauen wir "bottom-up" */
        List<Produkt> splist = new ArrayList<Produkt>(2);
        splist.add(new Bauteil("Siedend 8GB", "Speicherriegel", 429.12f));
        splist.add(new Bauteil("Siedend 4GB", "Speicherriegel", 128.98f));
        Baugruppe speicher = new Baugruppe("Speicher", "Ausstattung", splist);
        List<Produkt> clist = new ArrayList<Produkt>(2);
        clist.add(new Bauteil("Schwindel 10Ghz", "Prozessor", 586.95f));
        clist.add(speicher);
        Produkt computer = new Baugruppe("PC 1", "Desktop-PC", clist);
        pg.addProdukte(computer);

        System.out.println(pg.toString());
    }
}

class Produktgruppe {
    private List<Produkt> produkte;

    private String name;

    public String toString() {
        StringBuffer ret = new StringBuffer("  Gruppe " + name + "\n");
        for (Produkt p : produkte) {
            ret.append(p);
        }
        return ret.toString();
    }

    public void addProdukte(Produkt p) {
        this.produkte.add(p);
    }

    public Produktgruppe(String name) {
        this.name = name;
        this.produkte = new LinkedList<Produkt>();
    }
}

abstract class Produkt {
    protected String name;

    protected String typ;

    public abstract float berechnePreis();

    public abstract String toString();

    public Produkt(String name, String typ) {
        this.name = name;
        this.typ = typ;
    }
}

class Bauteil extends Produkt {
    private float preis;
```

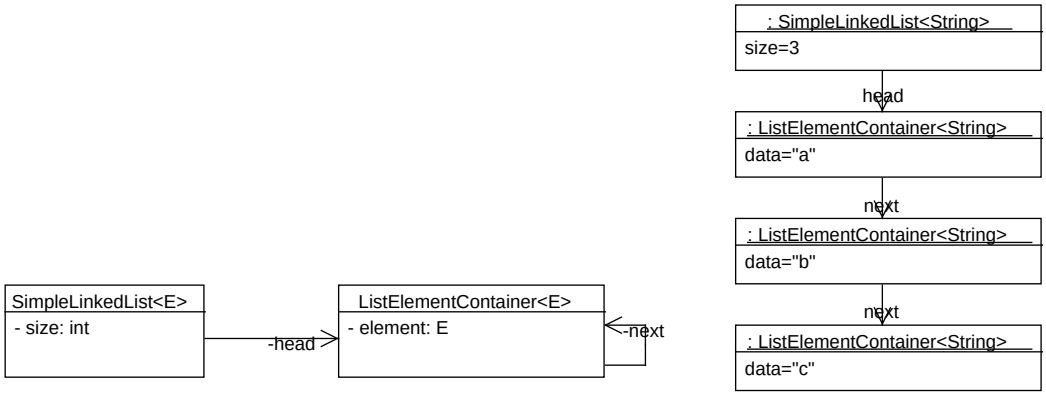
```
    public Bauteil(String name, String typ, float preis) {
        super(name, typ);
        this.preis = preis;
    }

    public float berechnePreis() {
        return preis;
    }

    public String toString() {
        return "    " + name + " (" + typ + "), Preis: " + preis + "\n";
    }
}

class Baugruppe extends Produkt {
    private List<Produkt> teile;

    /*************************************************************************
     * Durch die Uebergabe der Liste im Konstruktor erreichen wir, dass die
     * Kardinalitaet 1.. gewahrt wird. Es darf keine leere Liste uebergeben
     * werden - in diesem Fall werfen wir eine IllegalArgumentException.
     */
    public Baugruppe(String name, String typ, List<Produkt> teile) {
        super(name, typ);
        this.teile = teile;
        if (teile == null || teile.size() < 1) {
            throw new IllegalArgumentException(
                    "Kann Baugruppe nicht fuer leere Produkt-Liste instanziieren!");
        }
    }

    public float berechnePreis() {
        float ret = 0;
        for (Produkt t : teile) {
            ret += t.berechnePreis();
        }
        return ret;
    }

    public String toString() {
        StringBuffer ret = new StringBuffer("    Baugruppe " + name + " ("
                + typ + "), Preis " + berechnePreis() + ", bestehend aus: \n");
        for (Produkt t : teile) {
            ret.append(t);
        }
        return ret.toString();
    }
}
```

Aufgabe 9-2 Verkettete Liste (10 Punkte, `SimpleLinkedList.java`, `LinkedList.java`, `LinkedListTest.java`)

Gegebene Dateien: `AbstractLinkedList.java`, `ListIteratorImpl`, `ListElementContainer`.

Eine Möglichkeit, Listen zu implementieren, ist als sogenannte *verkettete Liste* (engl. *linked list*). Jedes der eigentlichen Listenelementen wird dabei in einem speziellen Objekt gespeichert, das wir für diese Aufgabe *Container* nennen. Die Container sind untereinander über Referenzen verlinkt. Jeder Container

zeigt dabei über eine Assoziation auf einen Nachfolger, der die Rolle `next` hat. Die untenstehenden Abbildungen zeigen das Klassendiagram einer verketteten Liste und beispielhaft das Objektdiagramm der Liste `["a", "b", "c"]`.



a) Einfache iterierbare Liste: Erstellen Sie eine Klasse `SimpleLinkedList` als Unterklasse der gegebenen Klasse `AbstractLinkedList`. Der Iterator ist ebenfalls gegeben, als Klasse `ListIteratorImpl`. Folgende Methoden in `AbstractLinkedList` sind abstrakt und müssen folglich in `SimpleLinkedList` implementiert werden:

- Methoden `size, get, set, add, remove`: Diese Methoden stellen eine minimale Schnittstelle für Listen dar und werden von `ListIteratorImpl` verwendet.
- Methode `getContainer`: Diese Methode soll Ihnen bei der Implementierung von `SimpleLinkedList` helfen. Beschreibung siehe JavaDoc.

Es soll nur einen nullstelligen Konstruktor geben. Hinweise:

- `AbstractLinkedList` implementiert die Schnittstelle `Iterable`, so dass Sie mit der vereinfachten `for`-Schleife über die Listenelemente iterieren können. Weiterhin gibt es die Instanzmethode `equals` zum Vergleichen einer Instanz mit einer anderen Liste, die Instanzmethode `toString` zum Darstellen des Inhalts und die statische Methode `safeEquals` zum sicheren Vergleichen zweier Objekte (selbst wenn eines davon `null` ist).
- Eclipse bietet Fehlerbehebungen an, bei denen automatisch die Rümpfe von zu implementierenden abstrakten Methoden der Oberklasse erzeugt werden.

b) Vollständige Implementierung der Schnittstelle `java.util.List`: Schreiben Sie eine Implementierung `LinkedList`. Es soll zwei Konstruktoren geben: `LinkedList()` und `LinkedList(Collection<E>)`. Letzterer initialisiert die Liste mit den Elementen einer bestehenden Collection. Nicht implementieren müssen Sie die Methoden `toArray(T[])` (`toArray()` schon!), `subList(int,int)` und `retainAll(Collection<?>)`. Dort können Sie im Methoden-Rumpf einfach eine `UnsupportedOperationException` werfen; das ist eine übliche Technik, um nicht unterstützte Funktionalität zu kennzeichnen. Hinweis:

- Nicht verwirren lassen: Das Interface `List` ist nicht komplett generisch (Beispiele: `remove(Object), contains(Object)`. Leider müssen wir uns daran halten, um mit der Java-API kompatibel zu bleiben.

c) Implementieren Sie folgende Unit-Tests:

- `testSimpleLinkedList`: Testen Sie die Methoden `get(int), set(int, E), add(int, E), remove(int), size()`. Wird bei einem falschen Zugriffsindex wirklich eine `IndexOutOfBoundsException` geworfen?
- `testLinkedList`: Testen Sie die Methoden `addAll(int, Collection<? extends E>), indexOf(Object), contains(Object), remove(Object), toArray()`.

- `testIterator`: Testen Sie, ob der Iterator korrekt funktioniert (Methoden `hasNext, next, add, remove, previous, hasPrevious`).
- `testNulls`: Kommt Ihre Implementierung auch mit `null`-Elementen klar? Schreiben Sie einen Test dafür, bei dem insbesondere `remove, contains, indexOf` berücksichtigt werden.

Hinweise:

- Wer (a) und (b) nicht geschafft hat, kann bei (c) trotzdem mitmachen, denn hier kann man auch eine beliebige `List`-Implementierung testen (z.B. `java.util.ArrayList`). Natürlich wird dann die Freude über gefundene Fehler weitgehend ausbleiben.
- Die Tests müssen nicht erschöpfend sein (gerade im JavaDoc zu `ListIterator` stehen sehr viele Bedingungen), sondern sollen ein paar grundlegende Dinge überprüfen.
- `testSimpleLinkedList`: Da eine SimpleLinkedList nicht das Interface `List` implementiert, kann man eine normale List-Implementierung nicht mit SimpleLinkedList vergleichen (umgekehrt schon). Nun ist es aber leider gerade so, dass bei einem Test `assertEquals(erwartet, tatsaechlich)` die `equals`-Methode des ersten Arguments eingesetzt wird (wo das zweite Argument in den Tests eine `SimpleLinkedList` sein wird). Abhilfe: verwenden Sie `assertTrue(tatsaechlich.equals(erwartet))`
- Arrays vergleichen: Leider scheitert `assertEquals` von zwei gleichen Arrays, wenn sie nicht identisch sind (also die selbe Instanz). Man kann sich behelfen mit `assertTrue(Arrays.equals(...))`.
- `java.util.Arrays.asList`: Diese Methode kann beliebig viele Argumente erhalten und erzeugt aus diesen eine Liste. Das hilft einem besonders bei Unit-Tests. Beispielaufruf:

```
Arrays.asList("a", "b", "c");
```

Lösung:

```
package list;

public class SimpleLinkedList<E> extends AbstractLinkedList<E> {

    protected ListElementContainer<E> head = null;
    protected int size = 0;

    public SimpleLinkedList() {
    }

    @Override
    public E get(int index) {
        return getContainer(index).getData();
    }

    @Override
    public E set(int index, E element) {
        ListElementContainer<E> container = getContainer(index);
        E oldData = container.getData();
        container.setData(element);
        return oldData;
    }

    @Override
    public void add(int index, E element) {
        ensureValidAddIndex(index);
        ListElementContainer<E> newContainer = new ListElementContainer<E>(element);
        if (index == 0) {
            // insert before first element
```

```
            newContainer.setNext(head);
            head = newContainer;
        } else {
            ListElementContainer<E> prevContainer = getContainer(index-1);
            newContainer.setNext(prevContainer.getNext());
            prevContainer.setNext(newContainer);
        }
        this.size++;
    }

    @Override
    public E remove(int index) {
        ensureValidIndex(index);
        ListElementContainer<E> currentContainer = getContainer(index);
        if (index == 0) {
            this.head = currentContainer.getNext();
        } else {
            ListElementContainer<E> prevContainer = getContainer(index-1);
            prevContainer.setNext(currentContainer.getNext());
        }
        this.size--;
        return currentContainer.getData();
    }

    @Override
    public int size() {
        return this.size;
    }

    //---------------------- Helper methods

    protected void ensureValidAddIndex(int index) {
        if (index < 0 || index > size()) {
            throw new IndexOutOfBoundsException();
        }
    }

    protected void ensureValidIndex(int index) {
        if (index < 0 || index >= size()) {
            throw new IndexOutOfBoundsException();
        }
    }

    @Override
    protected ListElementContainer<E> getContainer(int index) {
        ensureValidIndex(index);
        ListElementContainer<E> currentContainer = head;
        for(int i=0; i < index; i++) {
            currentContainer = currentContainer.getNext();
        }
        return currentContainer;
    }
}

package list;

import java.util.Collection;
```

```
import java.util.Iterator;
import java.util.List;

public class LinkedList<E> extends SimpleLinkedList<E> implements List<E> {

    public LinkedList() {
    }

    public LinkedList(Collection<E> coll) {
        this.addAll(coll);
    }

    //---------------------

    public boolean add(E element) {
        this.add(size(), element);
        return true;
    }

    public boolean addAll(Collection<? extends E> collection) {
        return this.addAll(size(), collection);
    }

    public boolean addAll(int index, Collection<? extends E> collection) {
        for(E elem : collection) {
            this.add(index, elem);
            index++;
        }
        return collection.size() > 0;
    }

    public void clear() {
        this.head = null;
        this.size = 0;
    }

    public boolean contains(Object search) {
        for(E element : this) {
            // equals handles null arguments correctly!
            if ((search == null && search == element)
                    || (search != null && search.equals(element))) {
                return true;
            }
        }
        return false;
    }

    public boolean containsAll(Collection<?> collection) {
        for(Object search : collection) {
            if (! this.contains(search)) {
                return false;
            }
        }
        return true;
    }
```

```
public int indexOf(Object search) {
    for(int i=0; i<this.size; i++) {
        E current = this.get(i);

        // equals handles null arguments correctly!
        if ((search == null && current == null) ||
                (search != null && search.equals(current))) {
            return i;
        }
    }
    return -1; // Not found
}

public int lastIndexOf(Object search) {
    for(int i=this.size-1; i>=0; i--) {
        E current = this.get(i);
        // equals handles null arguments correctly!
        if ((search == null && current == null) ||
                (search != null && search.equals(current))) {
            return i;
        }
    }
    return -1; // Not found
}

public boolean isEmpty() {
    return this.size == 0;
}

public boolean remove(Object search) {
    for(Iterator<E> iter=this.iterator(); iter.hasNext(); ) {
        if (safeEquals(search, iter.next())) {
            iter.remove();
            return true;
        }
    }
    return false;
}

public boolean removeAll(Collection<?> collection) {
    boolean hadChange = false;
    for(Object search : collection) {
        hadChange = remove(search) || hadChange;
    }
    return hadChange;
}

public Object[] toArray() {
    Object[] result = new Object[this.size];
    for(int i=0; i < this.size; i++) {
        result[i] = get(i);
    }
    return result;
}

//---------------------- Unsupported
```

```
    public <T> T[] toArray(T[] a) {
        throw new UnsupportedOperationException();
    }

    public List<E> subList(int fromIndex, int toIndex) {
        throw new UnsupportedOperationException();
    }

    public boolean retainAll(Collection<?> c) {
        throw new UnsupportedOperationException();
    }
}
```

```
package list;

import java.util.Arrays;
import java.util.Collections;
import java.util.ListIterator;

import junit.framework.TestCase;

public class LinkedListTest extends TestCase {
    public void testSimpleLinkedList() {
        SimpleLinkedList<String> list = new SimpleLinkedList<String>();
        list.add(0, "b");
        list.add(0, "a");
        list.add(2, "c");
        assertEquals("a", list.get(0));
        assertEquals("b", list.get(1));
        assertEquals("c", list.get(2));
        assertEquals(3, list.size());
        assertTrue(list.equals(Arrays.asList("a", "b", "c")));
        list.remove(0);
        assertTrue(list.equals(Arrays.asList("b", "c")));
        list.remove(1);
        assertTrue(list.equals(Arrays.asList("b")));
        list.set(0,"x");
        assertTrue(list.equals(Arrays.asList("x")));
        try {
            list.get(5);
            fail();
        } catch(IndexOutOfBoundsException e) {
            // success
        }
    }

    public void testLinkedList() {
        LinkedList<String> list = new LinkedList<String>(Arrays.asList("a"));
        assertEquals(Arrays.asList("a"), list);
        list.addAll(0, Arrays.asList("x", "y"));
        assertEquals(Arrays.asList("x", "y", "a"), list);
        assertEquals(1, list.indexOf("y"));
        assertEquals(-1, list.indexOf("r"));
        assertTrue(list.contains("a"));
        assertFalse(list.contains("s"));
        list.remove("a");
```

```
    assertEquals(Arrays.asList("x", "y"), list);
    assertTrue(Arrays.equals(new Object[] { "x", "y" }, list.toArray()));
}

// Methoden \texttt{hasNext, next, add, remove, previous, hasPrevious}
public void testIterator() {
    LinkedList<String> list = new LinkedList<String>(Arrays.asList("a", "b"));
    ListIterator<String> iter = list.listIterator();

    // next, add
    assertTrue(iter.hasNext());
    assertEquals("a", iter.next());
    assertEquals("b", iter.next());
    assertFalse(iter.hasNext());
    iter.add("c");
    assertFalse(iter.hasNext()); // next() is unaffected by add()
    assertEquals(Arrays.asList("a", "b", "c"), list);

    // previous, remove
    assertTrue(iter.hasPrevious());
    assertEquals("c", iter.previous()); // returns the element that has been added
    assertEquals("b", iter.previous());
    assertEquals("a", iter.previous());
    assertFalse(iter.hasPrevious());
    iter.remove();
    assertEquals(Arrays.asList("b", "c"), list);
}

public void testPositionAfterRemove() {
    LinkedList<String> l = new LinkedList<String>(Arrays.asList("a", "b", "c"));
    ListIterator<String> it = new ListIteratorImpl<String>(l, 0);
    assertEquals("a", it.next());
    it.remove();
    assertEquals("b", it.next());
    assertEquals("c", it.next());
    assertEquals("c", it.previous());
    it.remove();
    assertEquals(Arrays.asList("b"), l);
    assertEquals("b", it.previous());
}
public void testPositionAfterAdd() {
    LinkedList<String> l = new LinkedList<String>(Arrays.asList("a", "b", "c"));
    ListIterator<String> it = new ListIteratorImpl<String>(l, 0);

    it.add("x");
    assertEquals(Arrays.asList("x", "a", "b", "c"), l);

    assertEquals("a", it.next());
    assertEquals("a", it.previous());
    assertEquals("x", it.previous());
}

public void testRemoveAfterAdd() {
    LinkedList<String> l = new LinkedList<String>(Collections.singletonList("a"));
    ListIterator<String> it = new ListIteratorImpl<String>(l, 0);
    it.next(); // Make sure that remove would work without add()
```

```
        try {
            it.add("x");
            it.remove(); // Cannot remove after add
            fail();
        } catch(IllegalStateException e) {
            // succeed
        }
    }

    public void testSet() {
        LinkedList<String> l = new LinkedList<String>(Arrays.asList("a", "b", "c"));
        ListIterator<String> it = new ListIteratorImpl<String>(l, 0);
        assertEquals("a", it.next());
        it.set("x");
        assertEquals(Arrays.asList("x", "b", "c"), l);
        assertEquals("b", it.next());
        assertEquals("b", it.previous());
        it.set("y");
        assertEquals(Arrays.asList("x", "y", "c"), l);
    }

    public void testNulls() {
        LinkedList<String> list = new LinkedList<String>(Arrays.asList("a", null, "a"));
        assertEquals(3, list.size());
        assertEquals(1, list.indexOf(null));
        assertTrue(list.contains(null));
        list.remove(null);
        assertEquals(Arrays.asList("a", "a"), list);
        assertEquals(2, list.size());
    }
}
```

Abgabe: Per UniWorx, bis spätestens Montag, den 10.7.2006 um 9:00 Uhr.