RWTH Aachen
Lehrgebiet Theoretische Informatik
Frohn–Reidl–Rossmanith–Sánchez–Ströder

WS 2013/14
Übungsblatt 6
02.12.2013

Übung zur Vorlesung Programmierung

Aufgabe T12

Implementieren Sie eine Klasse *Set*, um eine Menge zu modellieren. Benutzen Sie dafür unverändert die gegebenen Klassen *AbstractIterableSet* und *SimpleFunctionalSet*. Diese Klassen folgen der Idee aus Aufgabe T7, in welcher Mengen durch eine Liste von Einfüge- und Löschoperationen dargestellt werden. Ihre neue Implementierung soll folgendermaßen aussehen:

a) Implementieren Sie die generische Klasse $\mathit{EmptySet}\langle E\rangle$ als Unterklasse von $\mathit{SimpleFunctionalSet}\langle E\rangle$, um die leere Menge darzustellen.

b) Implementieren Sie die generische Klasse $\mathit{AddSet}\langle E\rangle$ als Unterklasse von $\mathit{SimpleFunctionalSet}\langle E\rangle$, um das Hinzufügen eines Elements zu modellieren.

c) Implementieren Sie die generische Klasse $\mathit{RemoveSet}\langle E\rangle$ als Unterklasse von $\mathit{SimpleFunctionalSet}\langle E\rangle$, um das Löschen eines Elements zu modellieren.

d) Schreiben Sie eine generische Klasse $\mathit{FunctionalSet}\langle E\rangle$ als Unterklasse von $\mathit{AbstractIterableSet}\langle E\rangle$. Da die Oberklasse das Interface *java.util.Set* implementiert, muss Ihre Klasse die noch fehlenden Methoden *add*, *clear*, *contains*, *iterator* und *remove* implementieren. Ihre Implementierung soll das Hinzufügen beliebiger Elemente (inklusive *null*) unterstützen. Entwerfen Sie für die *iterator* Methode eine weitere generische Klasse $\mathit{FunctionalSetIterator}\langle E\rangle$, welche das Interface $\mathit{java.util.Iterator}\langle E\rangle$ implementiert.

Achten Sie darauf, dass Sie passende Konstruktoren explizit angeben müssen, und beachten Sie die Prinzipien der Datenkapselung.
Zum Testen Ihrer Implementierung können Sie die Datei *SetTest.java* von unserer Webseite benutzen. Der Aufruf Ihrer *main*-Methode sollte folgende Ausgabe auf der Konsole erzeugen:

```
{}
{1}
{2,1}
{3,2,1}
{3,2,1}
{3,2}
{3}
{3}
{}
{null}
{1}
```

```
{2,1}
{2,1}
{2,1}
{2}
{2}
{5,2,3,null}
{5,2,3,null}
{5,3,null}
{5,3,null}
{5,3}
```

Lösungsvorschlag

```
/**
 *@author Thomas Stroeder
 *@param⟨E⟩ Element type.
 *Simple functional set adding an element to another simple functional set.
 */
public class AddSet⟨E⟩ extends SimpleFunctionalSet⟨E⟩ {
    /**
     *The element to add.
     */
    private final E element;
    /**
     *@param elem The element to add.
     *@param s The remaining set.
     */
    public AddSet(final E elem, final SimpleFunctionalSet⟨E⟩ s) {
        super(s);
        this.element = elem;
    }
    @Override
    public boolean contains(final Object o) {
        if (this.element == null) {
            if (o == null) {
                return true;
            } else {
                return this.getRemainingSet().contains(o);
            }
        } else if (this.element.equals(o)) {
            return true;
        } else {
            return this.getRemainingSet().contains(o);
        }
    }
    /**
     *@return The element to add.
     */
    public E getElement() {
        return this.element;
    }
}
/**
 *@author Thomas Stroeder
 *@param⟨E⟩ Element type.
 *Simple functional set containing no element.
 */
public class EmptySet⟨E⟩ extends SimpleFunctionalSet⟨E⟩ {
    /**
     *Creates an empty simple functional set.
```

```
     */
    public EmptySet() {
        super(null);
    }
    @Override
    public boolean contains(final Object o) {
        return false;
    }
}
import java.util.*;

/**
 *@author Thomas Stroeder
 *@param<E> Element type.
 *
 */
public class FunctionalSet<E> extends AbstractIterableSet<E> {
    /**
     *The head of the list of operations representing the set.
     */
    private SimpleFunctionalSet<E> head;
    /**
     *Creates an empty functional set.
     */
    public FunctionalSet() {
        this.head = new EmptySet<E>();
    }
    @Override
    public boolean add(final E e) {
        if (this.contains(e)) {
            return false;
        } else {
            this.head = new AddSet<E>(e, this.head);
            return true;
        }
    }
    @Override
    public void clear() {
        this.head = new EmptySet<E>();
    }
    @Override
    public boolean contains(final Object o) {
        return this.head.contains(o);
    }
    @Override
    public Iterator<E> iterator() {
        return new FunctionalSetIterator<E>(this, this.head);
    }
    @Override
```

```
    public boolean remove(final Object o) {
        if (this.contains(o)) {
            this.head = new RemoveSet<E>(o, this.head);
            return true;
        } else {
            return false;
        }
    }
}
import java.util.*;

/**
 *@author Thomas Stroeder
 *@param<E> Element type.
 *Iterator through a functional set.
 */
public class FunctionalSetIterator<E> implements Iterator<E> {
    /**
     *The current simple functional set. It is always an add set with an unused (i.e., not contained ∈
     *the set named "used" below) element or an empty set.
     */
    private SimpleFunctionalSet<E> current;
    /**
     *The most recent element this iterator has returned.
     */
    private E recentElem;
    /**
     *Flag indicating whether the remove operation is applicable (needed since elements may be null, so
     *we cannot just check whether recentElem is null).
     */
    private boolean removable;
    /**
     *The functional set to which the current simple functional set belongs.
     */
    private final FunctionalSet<E> set;
    /**
     *A set of already seen elements (∈ add or remove sets).
     */
    private final Set<Object> used;
    /**
     *@param functionalSet The functional set containing the simple functional set to iterate over.
     *@param head The head of the simple functional set to iterate over.
     */
    public FunctionalSetIterator(final FunctionalSet<E> functionalSet, final SimpleFunctionalSet<E> head) {
        this.current = head;
        this.recentElem = null;
        this.removable = false;
        this.set = functionalSet;
        this.used = new FunctionalSet<Object>();
```

```
        this.forwardToNextUnusedSet();
    }
    @Override
    public boolean hasNext() {
        return!(this.current instanceof EmptySet);
    }
    @Override
    public E next() {
        if (this.hasNext()) {
            final E elem = ((AddSet<E>) this.current).getElement();
            this.used.add(elem);
            this.recentElem = elem;
            this.removable = true;
            this.current = this.current.getRemainingSet();
            this.forwardToNextUnusedSet();
            return elem;
        } else {
            throw new NoSuchElementException();
        }
    }
    @Override
    public void remove() {
        if (this.removable) {
            this.set.remove(this.recentElem);
            this.removable = false;
        } else {
            throw new IllegalStateException(
                "The next method has not been called before this remove operation!");
        }
    }
    /**
     *Forwards the current set to the next remaining set which is no remove set and which is no add set
     *with an already used element. During forwarding, used objects are added to the corresponding set.
     */
    private void forwardToNextUnusedSet() {
        boolean loop = true;
        while (loop) {
            loop = false;
            while (this.current instanceof RemoveSet) {
                this.used.add(((RemoveSet<E>) this.current).getObject());
                this.current = this.current.getRemainingSet();
            }
            if (this.current instanceof AddSet
                && this.used.contains(((AddSet<E>) this.current).getElement()))
            {
                loop = true;
                this.current = this.current.getRemainingSet();
            }
```

```
        }
    }
}
/**
 *@author Thomas Stroeder
 *@param⟨E⟩ Element type.
 *Simple set removing an object from another simple set.
 */
public class RemoveSet⟨E⟩ extends SimpleFunctionalSet⟨E⟩ {
    /**
     *The object to remove.
     */
    private final Object obj;
    /**
     *@param o The object to remove.
     *@param s The remaining set.
     */
    public RemoveSet(final Object o, final SimpleFunctionalSet⟨E⟩ s) {
        super(s);
        this.obj = o;
    }
    @Override
    public boolean contains(final Object o) {
        if (this.obj == null) {
            if (o == null) {
                return false;
            } else {
                return this.getRemainingSet().contains(o);
            }
        } else if (this.obj.equals(o)) {
            return false;
        } else {
            return this.getRemainingSet().contains(o);
        }
    }
    /**
     *@return The object to remove.
     */
    public Object getObject() {
        return this.obj;
    }
}
```

Aufgabe T13

Ein *Binärbaum* ist eine Datenstruktur, in welcher Zahlen (oder andere geordnete Elemente) gespeichert werden. Jeder Baumknoten enthält dabei ein Element x sowie zwei

Referenzen zu Unterbäumen; im linken Teilbaum sind dabei Elemente gespeichert, die strikt kleiner als x sind, im rechten Teilbaum diejenigen, die strikt größer als x sind. Jedes Element kann höchstens einmal im Baum enthalten sein. Implementieren Sie folgende Methoden der Klassen *BinaryTree* und *BinaryTreeNode*:

- **boolean** *contains*(**int** v): gibt *true* genau dann zurück, wenn der Binärbaum das Element v enthält.
- **void** *insert*(**int** v): fügt das Element v in den Baum ein, wenn es nicht schon darin enthalten ist.

```
public class BinaryTree {
  private BinaryTreeNode root;
  public BinaryTree() {
    root = null;
  }
  public boolean contains(int v) {// TODO }
  public void insert(int v) {// TODO }
}
public class BinaryTreeNode {
  private int value;
  private BinaryTreeNode left, right;
  public BinaryTreeNode(int v) {
    value = v;
    left = right = null;
  }
  public boolean contains(int v) {// TODO }
  public void insert(int v) {// TODO }
}
```

Lösungsvorschlag

```
public class BinaryTree {
  private BinaryTreeNode root;
  public BinaryTree() {
    root = null;
  }
  public boolean contains(int v) {
    return root != null && root.contains(v);
  }
  public void insert(int v) {
    if (root == null) {
      root = new BinaryTreeNode(v);
    } else {
      root.insert(v);
    }
  }
}
public class BinaryTreeNode {
```

```
  private int value;
  private BinaryTreeNode left, right;
  public BinaryTreeNode(int v) {
    value = v;
    left = right = null;
  }
  public boolean contains(int v) {
    if (v < value) {
      return left != null && left.contains(v);
    }
    if (v > value) {
      return right != null && right.contains(v);
    }
    return true;
  }
  public void insert(int v) {
    if (v < value) {
      if (left == null) {
        left = new BinaryTreeNode(v);
      } else {
        left.insert(v);
      }
    } else if (v > value) {
      if (right == null) {
        right = new BinaryTreeNode(v);
      } else {
        right.insert(v);
      }
    }
  }
}
```

Aufgabe H12 (5 Punkte)[1]

Sie finden eine leichte Abwandlung der Listen-Klasse aus der Vorlesung auf unserer Webseite unter dem Namen *IntList*. Entwerfen Sie ein Interface *IntListVisitor* und erweitern Sie die Klasse *IntList* um eine Methode **void** *accept*(*IntListVisitor visitor*), um im Sinne des Visitor-Patterns ein solches Objekt entgegenzunehmen.

Implementieren Sie anschließend eine Klasse *SummingIntListVisitor*, welche dieses Interface implementiert, um die Summe einer Liste mit Hilfe der *accept*-Methode zu berechnen.

Lösungsvorschlag

```
public class IntList {
  private final boolean empty; // ist diese Liste leer?
  private final int value; // das erste Element dieser Liste
  private final IntList rest; // die restliche Liste ohne das erste Element
  public IntList() {// erzeuge eine neue leere Liste
    empty = true;
```

```
    value = 0;
    rest = null;
  }
  private IntList(int elem, IntList rest) {
    this.empty = false;
    this.value = elem;
    this.rest = rest;
  }
  public boolean isEmpty() {
    return empty;
  }
  public int head() {
    return value;
  }
  public IntList tail() {
    return rest;
  }
  public IntList add(int elem) {
    return new IntList(elem, this);
  }
  @Override
  public String toString() {
    if (isEmpty()) {
      return "";
    } else {
      return "" + head() + "  " + tail().toString();
    }
  }
  public void accept(IntListVisitor visitor) {
    if(empty) {
      return;
    }
    visitor.visit(value);
    rest.accept(visitor);
  }
}
public interface IntListVisitor {
  public void visit(int e);
}
public class SummingIntListVisitor implements IntListVisitor {
  private int sum = 0;
  @Override
  public void visit(int e) {
    sum += e;
  }

  public int getSum() {
    return sum;
```

```
  }
}
```

Aufgabe H13 (2 + 15 Punkte)[1]

In dieser Aufgabe geht es um die Implementierung einer Datenstruktur für Mengen, welche in das bestehende Collections Framework eingebettet werden soll. Sie benötigen dafür die auf unserer Webseite verfügbare abstrakte Klasse *AbstractIterableSet*. Alle in dieser Aufgabe zu implementierenden Klassen sollen nicht abstrakt sein und die vorgegebene abstrakte Klasse soll nicht verändert werden.

Die hier zu implementierende Mengenstruktur basiert auf einer Liste von Elementen mit einer *active* Markierung vom Typ **boolean**. Nur solche Elemente sind in der Menge enthalten, deren *active* Markierung den Wert *true* hat. Um Elemente aus der Menge zu löschen, werden keine Objekte aus dieser Liste entfernt, sondern lediglich die entsprechende Markierung auf *false* gesetzt. Um Duplikate zu vermeiden, soll beim Einfügen die Liste durchsucht werden und für ein ggf. bereits gelöschtes Element, das wieder eingefügt wird, schlicht die entsprechende Markierung wieder auf *true* gesetzt werden. Beispielsweise kann die Menge $\{1, 2, 3\}$ als die Liste 3 *true*, 2 *true*, 1 *true* dargestellt werden. Löscht man nun das Element 2, so ergibt sich die Liste 3 *true*, 2 *false*, 1 *true*. Ein erneutes Einfügen des Elementes 2 liefert wieder die ursprüngliche Liste. Einfügen von bereits vorhanden Elementen oder Löschen von nicht vorhandenen Elementen soll die Datenstruktur unverändert lassen—lediglich die Methode *clear* soll tatsächlich Objekte aus der Liste entfernen.

a) Schreiben Sie eine generische Klasse *SetNode*$\langle E \rangle$ mit drei Attributen *active* vom Typ **boolean**, *element* vom Typ E und *next* vom Typ *SetNode*$\langle E \rangle$. Implementieren Sie Getter-Methoden für alle Attribute und ermöglichen Sie beliebige Veränderungen am Attribut *active* (beachten Sie dabei die Prinzipien der Datenkapselung).

b) Schreiben Sie eine generische Klasse *FlagSet*$\langle E \rangle$ als Unterklasse der bereitgestellten Klasse *AbstractIterableSet*$\langle E \rangle$ mit genau einem Attribut *head* vom Typ *SetNode*$\langle E \rangle$. Da die Oberklasse das Interface *java.util.Set* implementiert, benötigt Ihre Klasse die noch fehlenden Methoden *add*, *clear*, *contains*, *iterator* und *remove*. Nutzen Sie dazu die in der vorherigen Teilaufgabe implementierte Klasse *SetNode*. Ihre Menge soll das Hinzufügen von beliebigen Elementen (inklusive *null*) unterstützen und nur dann ein neues *SetNode*-Objekt anlegen, wenn ein Element in eine Menge eingefügt wird, das noch nie in dieser Menge enthalten war (bis zurück zur Erstellung der Menge oder zum letzten Aufruf von *clear*). Benutzen Sie die *active* Markierungen in den *SetNode*-Objekten, um, wie am Anfang dieser Aufgabe beschrieben, den Inhalt der Menge zu verwalten. Implementieren Sie für die Methode *iterator* eine weitere generische Klasse *FlagSetIterator*$\langle E \rangle$, welche das Interface *java.util.Iterator*$\langle E \rangle$ und somit die Methoden *hasNext*, *next* und *remove* sinnvoll implementiert.

Die gewünschte Funktionalität der Methoden aus den Interfaces *Set* und *Iterator* schlagen Sie bitte in der Java-API nach.

Zum Testen Ihrer Implementierung können Sie ebenfalls die Datei *SetTest.java* von unserer Webseite benutzen, indem Sie alle Vorkommen von *FunctionalSet* durch *FlagSet*

ersetzen. Der Aufruf Ihrer *main* Methode sollte die gleiche Ausgabe wie in der Tutoraufgabe T12 erzeugen.

Hinweis: Für den Iterator könnte es hilfreich sein, das nächste zurückzuliefernde und das zuletzt zurückgelieferte Element zwischenzuspeichern.

Lösungsvorschlag

```
import java.util.*;
/**
 *@author Thomas Stroeder
 *@param<E> Element type.
 */
public class FlagSet<E> extends AbstractIterableSet<E> {
    /**
     *The first set node of this set.
     */
    private SetNode<E> head;
    /**
     *Creates an empty FlagSet.
     */
    public FlagSet() {
        this.head = null;
    }
    @Override
    public boolean add(final E e) {
        SetNode<E> current = this.head;
        while (current != null) {
            final E elem = current.getElement();
            if (elem == null) {
                if (e == null) {
                    if (current.isActive()) {
                        return false;
                    } else {
                        current.activate();
                        return true;
                    }
                }
            } else if (elem.equals(e)) {
                if (current.isActive()) {
                    return false;
                } else {
                    current.activate();
                    return true;
                }
            }
            current = current.getNext();
        }
        this.head = new SetNode<E>(e, this.head);
```

```
        return true;
    }
    @Override
    public void clear() {
        this.head = null;
    }
    @Override
    public boolean contains(final Object o) {
        for (final E elem : this) {
            if (elem == null) {
                if (o == null) {
                    return true;
                }
            } else if (elem.equals(o)) {
                return true;
            }
        }
        return false;
    }
    @Override
    public Iterator<E> iterator() {
        return new FlagSetIterator<E>(this.head);
    }
    @Override
    public boolean remove(final Object o) {
        SetNode<E> current = this.head;
        while (current != null) {
            final E elem = current.getElement();
            if (elem == null) {
                if (o == null) {
                    if (current.isActive()) {
                        current.deactivate();
                        return true;
                    } else {
                        return false;
                    }
                }
            } else if (elem.equals(o)) {
                if (current.isActive()) {
                    current.deactivate();
                    return true;
                } else {
                    return false;
                }
            }
            current = current.getNext();
        }
        return false;
```

```
    }
}
import java.util.*;

/**
 *@author Thomas Stroeder
 *@param⟨E⟩ Element type.
 *Iterator through a FlagSet.
 */
public class FlagSetIterator⟨E⟩ implements Iterator⟨E⟩ {
    /**
     *The current active node.
     */
    private SetNode⟨E⟩ current;
    /**
     *The most recent active node. This is null if the next() method has not been called since the last
     *remove operation.
     */
    private SetNode⟨E⟩ recent;
    /**
     *@param head The start node.
     */
    public FlagSetIterator(final SetNode⟨E⟩ head) {
        this.current = head;
        this.recent = null;
        this.forwardToNextActiveNode();
    }
    @Override
    public boolean hasNext() {
        return this.current != null;
    }
    @Override
    public E next() {
        if (this.hasNext()) {
            final E res = this.current.getElement();
            this.recent = this.current;
            this.current = this.current.getNext();
            this.forwardToNextActiveNode();
            return res;
        } else {
            throw new NoSuchElementException();
        }
    }
    @Override
    public void remove() {
        if (this.recent == null) {
            throw new IllegalStateException(
                "The next method has not been called before this remove operation!");
        } else {
```

```
            this.recent.deactivate();
            this.recent = null;
        }
    }
    /**
     *Forwards the current node to the next active node.
     */
    private void forwardToNextActiveNode() {
        while (this.current != null &&!this.current.isActive()) {
            this.current = this.current.getNext();
        }
    }
}
/**
 *@author Thomas Stroeder
 *@param⟨E⟩ Element type.
 */
public class SetNode⟨E⟩ {
    /**
     *Flag indicating activeness of the element.
     */
    private boolean active;
    /**
     *The element.
     */
    private final E element;
    /**
     *The next node.
     */
    private final SetNode⟨E⟩ next;
    /**
     *Creates an active set node.
     *@param elem The element.
     *@param n The next node.
     */
    public SetNode(final E elem, final SetNode⟨E⟩ n) {
        this.element = elem;
        this.next = n;
        this.active = true;
    }
    /**
     *Sets the element active.
     */
    public void activate() {
        this.active = true;
    }
    /**
     *Sets the element inactive.
```

```
     */
    public void deactivate() {
        this.active = false;
    }
    /**
     *@return The element.
     */
    public E getElement() {
        return this.element;
    }
    /**
     *@return The next node.
     */
    public SetNode<E> getNext() {
        return this.next;
    }
    /**
     *@return True iff the element is active.
     */
    public boolean isActive() {
        return this.active;
    }
}
```

Abgabe zum 10.12.2013

[1] Bitte Quelltext für die Abgabe ausdrucken und zusätzlich per E-mail an den jeweiligen Tutor senden.