Objektorientierte Programmiertechniken
LVA 185.A01, VU 2.0, 2011 W

5. Übungsaufgabe

Themen:

Generizität, Container und Iteratoren; nebenbei auch Sichtbarkeit, Zusicherungen und Ersetzbarkeit

Termine:

Ausgabe:	23.11.2011
reguläre Abgabe:	30.11.2011, 13:45 Uhr
nachträgliche Abgabe:	07.12.2011, 13:45 Uhr

Abgabeverzeichnis:

Gruppe/Aufgabe5

Programmaufruf:

java Test

Grundlage:

Skriptum bis Seite 106, Schwerpunkt auf Abschnitt 3.1

Aufgabe

Welche Aufgabe zu lösen ist:

Passen Sie die Lösung von Aufgabe 4 an folgende Punkte an:

- Das generische Interface *Iter* spezifiziert die Iterator-Methoden *next*, *previous*, *hasNext* und *hasPrevious*, wobei *previous* und *hasPrevious* analog zu *next* und *hasNext* funktionieren, aber in die entgegengesetzte Richtung laufen sollen. Wenn beispielsweise drei aufeinanderfolgende Aufrufe von *next* in *Iter<Integer>* die Zahlen 1, 2 und 3 liefern, dann gibt ein darauffolgender Aufruf von *previous* 2 zurück, und ein weiterer Aufruf von *next* wiederum 3.
- Für die Untersuchung von Bäumen wird ein spezieller Iterator benötigt:

Das generische Interface *TreeIter* erweitert *Iter<...>* um die Methode *down*, die einen neuen Iterator vom Typ *TreeIter<...>* zurückgibt, der nur über das aktuelle Element und die Elemente des entsprechenden Teilbaums iteriert. Das aktuelle Element ist dabei jenes, das beim letzten Aufruf von *next* oder *previous* zurückgegeben wurde. Wurden *next* und *previous* noch nie zuvor aufgerufen, so ist das bei einem gedachten Aufruf von *next* als nächstes zurückzugebende Element das aktuelle Element. Ist das aktuelle Element gleich *null*, wird ein leerer Iterator (in dem *next* und *previous* stets *null* liefern) zurückgegeben.

- Der generische Typ *Tree* stellt einen binären Baum dar. Knoten des Baumes sind nach außen nicht direkt sichtbar, sondern werden nur durch Iteratoren vom Typ *TreeIter<...>* über den Labels der Knoten (= Elemente) dargestellt. Es soll möglich sein, mehrere Iteratoren zu haben, die (möglicherweise auch gleichzeitig) über denselben Baum oder Teilbaum iterieren. Außerdem werden auf *Tree* folgende Methoden benötigt:
  - *contains* sucht ein Element im Baum und gibt einen Iterator vom Typ *TreeIter<...>* zurück, der über dem gefundenen Element und den Elementen des enstprechenden Teilbaums iteriert. Enthält der Baum das gesuchte Element nicht, soll *null* zurückgegeben werden.
  - *iterator* gibt einen neuen Iterator vom Typ *TreeIter<...>* über allen Elementen des Baums zurück. Der Iterator soll jenem entsprechen, der von *contains* bei der Suche nach dem Wurzelknoten zurückgegeben wird.
  - *search* sucht ein Element im Baum und gibt einen Iterator vom Typ *Iter<Boolean>* aus, der den Pfad von der Wurzel des Baums bis zum gefundenen Knoten beschreibt. Der Wert *false* steht dabei fÃ¼r den linken und *true* fÃ¼r den rechten Zweig des Baums. Enthält der Baum keinen entsprechenden Knoten, soll *null* zurückgegeben werden. Entspricht der gefundene Knoten der Wurzel, soll dagegen ein leerer Iterator zurückgegeben werden. Enthält der Baum das gesuchte Element mehrfach, so sollen *contains* und *search* denselben Baumknoten finden.
  - *add* fügt ein Element in den Baum ein. Die genaue Stelle, an welcher der Knoten angelegt wird, wird in Untertypen festgelegt.

  Jede Suche im Baum beruht auf der Gleichheit der Elemente, nicht auf Identität.
- Der generische Typ *ReplaceableTree* ist ein Untertyp von *Tree<...>*, bei dem ein Teilbaum durch einen anderen Teilbaum ersetzt werden kann:
  - *replace* soll gegenüber Aufgabe 4 so abgeändert werden, dass es möglich ist, alle Instanzen von *Tree<...>* (mit geeigneten Typparameterersetzungen) als Teilbäume einzufügen. Beispielsweise soll in einen Baum vom Typ *ReplaceableTree<Person>* als Teilbaum auch eine Instanz von *Tree<Student>* eingefügt werden können (siehe unten). Der Typ, der den Typparameter des Teilbaums ersetzt, darf also ein Untertyp jenes Typs sein, der den Typparameter des Baums ersetzt, in den eingefügt wird. Dazu ist es erforderlich, dass

beim Einfügen eine Kopie des eingefügten Baumes erstellt und nicht einfach der Originalbaum eingehängt wird. Dabei ändern sich auch die Eigenschaften des Teilbaums (z.B. Reihenfolge der Elemente bei der Iteration), sodass sie denen von *ReplaceableTree* entsprechen. Die Stelle, an welcher der Teilbaum zu ersetzen ist, wird durch eine Instanz von *Iter<Boolean>* bestimmt. Zumindest der Elternknoten dieser Stelle muss existieren, damit das Ersetzen möglich ist; wenn kein Elternknoten existiert, bleibt der Baum unverändert.
    - *add* fügt ein neues Element wie in Aufgabe 4 so nahe wie möglich an der Wurzel des Baumes ein, wobei die Pfade aller schon bestehenden Knoten unverändert bleiben. Gibt es mehrere freie Stellen gleich weit von der Wurzel entfernt (aber keinen näheren freien Platz), so wird der neue Knoten an der am weitesten links stehenden freien Stelle eingefügt.
    - Die Reihenfolge, in der die durch *contains* und *iterator* erzeugten Iteratoren Elemente zurÃ¼ckgeben, kann beliebig gewählt werden.
- Der generische Typ *SortedTree* ist ein Untertyp von *Tree<...>*, wobei die Elemente sortierbar sind; das heißt, sie unterstützen eine Methode *compareTo* (wie auf Seite 103 des Skriptums). Die Label der Knoten in jedem linken Teilbaum müssen kleiner als die der Knoten im rechten Teilbaum sein, und die im rechten Teilbaum müssen größer oder gleich denen im linken Teilbaum sein.
    - *search* soll sich diese Sortierung zu Nutze machen, um das Durchsuchen des Baums in durchschnittlich logarithmischer Zeit (in Abhängigkeit von der Anzahl der Knoten) zu ermöglichen.
    - *add* soll einen neuen Knoten so in den Baum einfügen, dass die Sortierung der Knoten gewahrt bleibt.
    - Die Methode *traverse* aus Aufgabe 4 wird nicht mehr benötigt. Stattdessen erzeugt *iterator* einen Iterator, der die Elemente in der Reihenfolge einer Traversierung zurückgibt.
- Der generische Typ *PreorderTree* ist ein Untertyp von *SortedTree<...>*, in dem der durch *iterator* erzeugte Iterator den Baum in Preorder traversiert.
- Der generische Typ *InorderTree* ist ein Untertyp von *SortedTree<...>*, in dem der durch *iterator* erzeugte Iterator den Baum in der Ordnung entsprechend *compareTo* traversiert.
- Der generische Typ *PostorderTree* ist ein Untertyp von *SortedTree<...>*, in dem der durch *iterator* erzeugte Iterator den Baum in Postorder traversiert.

Weiters wird zum Testen obiger Klassen und Interfaces Folgendes benötigt:

- *Person* ist eine abstrakte Klasse, die einen String für den Namen enthält, welcher über einen Konstruktor gesetzt wird. Weiters unterstützt sie die Methode *compareTo*, die die Namen in zwei Instanzen von Person alphabetisch miteinander vergleicht, aber Unterschiede in der Groß- und Kleinschreibung ignoriert. Natürlich muss auch *equals* so definiert sein, dass *equals* in genau den Fällen *true* liefert, in denen *compareTo* 0 liefert.

- *Student* ist eine Unterklasse von *Person* und enthält zusätzlich eine Matrikelnummer vom Typ *int*.
- *Professor* ist ebenfalls eine Unterklasse von *Person* und enthält einen zusätzlichen String für das Institut.

Ein Aufruf von *java Test* soll wie gewohnt Testfälle ausführen und die Ergebnisse in allgemein verständlicher Form darstellen. Anders als in bisherigen Aufgaben sind die Überprüfungen jedoch vorgegeben und in dieser Reihenfolge auszuführen:

1. Erzeugen Sie je einen Baum der Typen *ReplaceableTree<String>*, *InorderTree<Integer>*, *PreorderTree<Student>* und *PostorderTree<Professor>*, fügen Sie einige Instanzen der entsprechenden Typen in beliebiger Reihenfolge ein, und geben Sie alle Elemente (bei Personen inklusive Matrikelnummern bzw. Instituten) in der Reihenfolge aus, in der sie vom durch *iterator* erzeugten Iterator zurückgegeben werden. Suchen Sie auch über *contains* und *search* nach jeweils einigen Elementen in den Bäumen und geben Sie alle Elemente der Teilbäume unter den gefundenen Knoten sowie die Pfade zu den gefundenen Knoten aus.
2. Erzeugen Sie einen Baum vom Typ *ReplaceableTree<Person>* und fügen Sie einige Instanzen von *Student* und *Professor* ein. Ersetzen Sie dann zwei unterschiedliche Teilbäume des Baums durch die in Punkt 1 erzeugten Bäume der Typen *PreorderTree<Student>* und *PostorderTree<Professor>*. Geben Sie über den Iterator den so erzeugten Baum aus (nur die Namen der Personen), wobei Sie durch Verwendung von *down* die Struktur des Baumes sichtbar werden lassen.
3. Machen Sie weitere Überprüfungen, die Ihnen notwendig und sinnvoll erscheinen.

Wie die Aufgabe zu lösen ist:

Von allen oben beschriebenen Interfaces, Klassen und Methoden wird erwartet, dass sie überall verwendbar sind. Der Bereich, in dem weitere eventuell benötigte Klassen, Methoden, Variablen, etc. sichtbar sind, soll jedoch so klein wie möglich gehalten werden.

Alle Teile dieser Aufgabe sind ohne Verwendung von Arrays, ohne vorgefertigte Container-Klassen (wie LinkedList, HashSet, etc.) und ohne vorgefertigte Iterator-Implementierungen zu lösen. Die benötigten Container und Iteratoren sind selbst zu schreiben.

Typsicherheit soll so weit wie möglich vom Compiler garantiert werden. Auf die Verwendung von Typumwandlungen (casts) und ähnliche Techniken ist daher zu verzichten, und der Compiler darf keine Hinweise auf mögliche Probleme im Zusammenhang mit Generizität geben. Achtung: Übersetzen Sie die Klassen mittels *javac -Xlint:unchecked *.java*; dieses Compiler-Flag schaltet genaue

Compiler-Meldungen im Zusammenhang mit Generizität ein. Andernfalls bekommen Sie auch bei schweren Fehlern vom Compiler nur eine harmlos aussehende Meldung (“Note: …”). Entsprechende Überprüfungen durch den Compiler dürfen nicht ausgeschaltet werden.

Warum die Aufgabe diese Form hat:

Die Aufgabe ist so konstruiert, dass dabei einige Schwierigkeiten auftauchen, für die wir Lösungsmöglichkeiten kennengelernt haben. Beispielsweise wird gebundene Generizität benötigt, und vermutlich ist in diesem Zusammenhang ein (bereits im Java-System vordefinierter) Typ erforderlich, der in der Aufgabenstellung nicht vorkommt.

Durch die Typhierarchie auf Person, Student und Professor muss Generizität über mehrere Ebenen hinweg betrachtet werden, da vereinfachende Sichtweisen durch die von dieser Hierarchie unabhängigen Typen *String* und *Integer* ausgeschlossen sind. Dieser Teil der Aufgabe ist z.B. durch Verwendung von Typ-Wildcards einfach lösbar (ähnlich wie im Beispiel auf einer Vorlesungsfolie).

Vorgegebene Testfälle stellen sicher, dass die Schwierigkeiten erkannt werden. Um Umgehungen zu vermeiden sind Typumwandlungen verboten, und Hinweise des Compilers auf unsichere Verwendungen von Generizität dürfen nicht ausgeschaltet werden. Neben Techniken zur Lösung der speziellen Schwierigkeiten wird in dieser Aufgabe auch der Umgang mit Sichtbarkeit und Untertypbeziehungen auf generischen Typen geübt. Am Beispiel von Iteratoren soll intuitiv klar werden, welchen Einfluss die Verwendung oder Nichtverwendung von inneren Klassen (speziell für Iteratoren) auf die Sichtbarkeit von Implementierungsdetails nach außen hat.

Was im Hinblick auf die Beurteilung wichtig ist:

Der wichtigste Schwerpunkt bei der Beurteilung liegt auf der sinnvollen und korrekten Verwendung von Generizität. Dementsprechend gibt es bedeutende Punkteabzüge, wenn der Compiler mögliche Probleme im Zusammenhang mit Generizität meldet oder wichtige Teilaufgaben nicht gelöst oder (durch Nichtbefolgung von Teilen der Aufgabenstellung) umgangen werden.

Ein zusätzlicher Schwerpunkt liegt auf dem gezielten Einsatz von Sichtbarkeit. Es gibt Punkteabzüge, wenn Programmteile, die überall sichtbar sein sollten, nicht public sind, oder Teile, die nicht für die allgemeine Verwendung bestimmt sind, unnötig weit sichtbar sind. Durch die Verwendung von inneren Klassen kann das Sichtbarmachen mancher Programmteile nach außen verhindert werden.

Nach wie vor spielen auch Untertypbeziehungen und Zusicherungen eine große Rolle bei der Beurteilung.

Generell führen Abänderungen der Aufgabenstellung (beispielsweise die Verwendung von Typumwandlungen, Arrays oder vorgefertigten Containern und Iteratoren oder das Ausschalten von Überprüfungen durch @SuppressWarning) zu bedeutenden Punkteabzügen.

Vermeiden Sie die Verwendung von packages auch schon in Ihrem ersten Entwurf, da diese immer wieder zu Problemen bei der Übertragung und Beurteilung führen. Legen Sie im Abgabeverzeichnis keine Unterverzeichnisse an.