Objektorientierte Programmiertechniken
LVA 185.A01, VU, 3 ECTS, 2014/2015 W

5. Übungsaufgabe

Themen:

Generizität, Container und Iteratoren

Termine:

Ausgabe: 12.11.2014
Abgabe: 19.11.2014, 12:00 Uhr

Abgabeverzeichnis:

Aufgabe5

Programmaufruf:

java Test

Grundlage:

Skriptum, Schwerpunkt auf den Abschnitten 3.1 und 3.2

Aufgabe

Welche Aufgabe zu lösen ist:

Entwickeln Sie in Java Klassen bzw. Interfaces entsprechend folgender Beschreibung:

- Das generische Interface `Iter` erweitert `java.util.Iterator<...>` um die Methoden `previous` und `hasPrevious`, die analog zu `next` und `hasNext` funktionieren, aber in die entgegengesetzte Richtung laufen. Wenn beispielsweise drei aufeinanderfolgende Aufrufe von `next` in `Iter<Integer>` die Zahlen 1, 2 und 3 liefern, dann gibt ein darauffolgender Aufruf von `previous` nocheinmal 3 zurück, und ein weiterer Aufruf von `next` wiederum 3.
- Das generische Interface `LeveledIter` erweitert `Iter<...>` um die Methode `sub`, die einen neuen Iterator vom Typ `LeveledIter<...>` zurückgibt, welcher über die mit dem aktuellen Element verbundenen Elemente der nächsten,

tieferen Ebene iteriert (siehe `Wood`). Das aktuelle Element ist dabei jenes, das bei einem Aufruf von `next` zurückgegeben werden würde. Ist das aktuelle Element gleich `null`, wird `null` zurückgegeben. Auch eine Methode `add` wird benötigt. Sie fügt ein Element unmittelbar vor dem Element ein, das durch `next` zurückgegeben werden würde, und unmittelbar nach dem Element, das durch `previous` zurückgegeben werden würde (bzw. am Anfang oder Ende, wenn keine Vorgänger- oder Nachfolger-Elemente existieren). Außerdem erzeugt `add` auch eine neue, mit dem eingefügten Element verbundene leere Ebene. Die Methode `remove` muss implementiert sein, darf also keine `UnsupportedOperationException` werfen.

- Der generische Typ `Wood` stellt eine Liste von Baumstrukturen dar. Unter jedem Baumknoten können beliebig viele Teilbäume hängen. Die Struktur wird nach außen nur über Iteratoren des Typ `LeveledIter<...>` sichtbar, wobei `sub` einen Iterator über die Wurzeln der unter dem entsprechenden Knoten hängenden Teilbäume liefert. Mehrere Iteratoren sollen gleichzeitig über die gesamte Struktur bzw. über Teilbäume iterieren können. Auf `Wood<...>` werden folgende Methoden benötigt:
  - `contains` sucht alle Elemente in der Datenstruktur, die gleich dem (nicht notwendigerweise identisch zum) übergebenen Argument sind. Das Ergebnis ist ein Iterator vom Typ `LeveledIter<...>`, der über alle gefundenen gleichen Elemente iteriert.
  - `iterator` gibt einen neuen Iterator vom Typ `LeveledIter<...>` über die Wurzeln aller Bäume (auf oberster Ebene) zurück. Der Iterator gibt die Elemente in der durch `add` bestimmten Reihenfolge zurück.
- Der generische Typ `SortedWood` ist ein Untertyp von `Wood<...>`, wobei die Elemente `Prec<...>` (siehe unten) implementieren und daher durch `prec` miteinander vergleichbar sind. Die Methode `sorted` gibt einen Iterator vom Typ `Iter<...>` zurück, der über alle Elemente der gesamten Datenstruktur in (über `next` aufsteigend) sortierter Reihenfolge iteriert. Außerdem implementiert `SortedWood` selbst `Prec<...>`, wobei `a.prec(b)` genau dann `true` zurückgibt, wenn für alle Elemente `x` aus `a` und alle Elemente `y` aus `b` auch `x.prec(y)` gilt.
- Das generische Interface `Prec` beschreibt nur die boolesche Methode `prec`, die zwei Objekte zusammenpassender Typen miteinander vergleicht.
- Die zum Testen benötigte abstrakte Klasse `Person` enthält den Namen einer Person als String, der über den Konstruktor gesetzt wird. Die abstrakte Klasse soll `Prec<...>` implementieren, wobei `x.prec(y)` genau dann gelten soll, wenn der Name von `x` gleich dem von `y` ist oder alphabetisch vor dem von `y` kommt. Beim Vergleich der Namen sollen Unterschiede aufgrund von Abständen zwischen Namensteilen (White-Space) sowie Groß- und Kleinschreibung ignoriert werden.
- `Skier` erweitert `Person` um die Angabe des Körpergewichts (`double`).
- `Snowboarder` erweitert `Person` um die Angabe der Körpergröße (`int`).

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 eine Datenstruktur der Typen `Wood<String>`, `SortedWood<Skier>` und `SortedWood<Snowboarder>` mit mehreren Ebenen, fügen Sie jeweils einige Elemente ein, und geben Sie alle Elemente (bei Personen inklusive Köpergewicht bzw. Körpergröße) so aus, dass die Struktur sichtbar wird. Geben Sie die Elemente von `SortedWood<Skier>` in entsprechend `prec` aufsteigend sortierter Reihenfolge aus, jene von `SortedWood<Snowboarder>` in absteigend sortierter Reihenfolge. Suchen Sie über `contains` nach jeweils einigen Elementen und geben Sie die Teilbäume unterhalb der gefundenen Elemente aus. Entfernen Sie Elemente, fügen Sie Elemente ein und geben Sie die Inhalte der Datenstrukturen erneut aus.
2. Erzeugen Sie eine Datenstruktur vom Typ `SortedWood<Person>`. Lesen Sie über Iteratoren alle Elemente aus den in Punkt 1 erzeugten Datenstrukturen der Typen `SortedWood<Skier>` und `SortedWood<Snowboarder>` aus und fügen Sie diese in die neue Datenstruktur ein. Überprüfen Sie die Funktionalität der Datenstruktur wie in Punkt 1, allerdings ohne Köpergewichte bzw. Körpergrößen auszugeben.
3. Erzeugen Sie weitere Datenstrukturen vom Typ `SortedWood<Person>` und fügen Sie diese in eine Datenstruktur vom Typ `SortedWood<SortedWood<Person>>` ein. Testen Sie die Funktionalität wie in Punkt 1, wobei Sie jedoch statt der Köpergewichte bzw. Körpergrößen die Inhalte der enthaltenen Elemente vom Typ `SortedWood<Person>` ausgeben.
4. Betrachten Sie die in Punkt 2 erzeugte Datenstruktur so, als ob sie vom Typ `Wood<Person>` wäre (z.B., indem Sie die Datenstruktur an eine Variable vom deklarierten Typ `Wood<Person>` zuweisen und die folgenden Tests auf dieser Variablen ausführen). Testen Sie die Funktionalität wie in Punkt 1, allerdings ohne Köpergewichte bzw. Körpergrößen auszugeben und die zusätzliche Funktionalität von `SortedWood` zu verwenden.
5. Machen Sie optional weitere Überprüfungen, die Ihnen notwendig und sinnvoll erscheinen.

Daneben soll die Klasse *Test.java* als Kommentar eine kurze, aber verständliche Beschreibung der Aufteilung der Arbeiten auf die einzelnen Gruppenmitglieder enthalten – wer hat was gemacht.

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. Benötigte 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 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: ..."). Überprüfungen durch den Compiler dürfen nicht ausgeschaltet werden, auch nicht für Warnungen.

Warum die Aufgabe diese Form hat:

Die Aufgabe ist so konstruiert, dass dabei Schwierigkeiten auftauchen, für die wir Lösungsmöglichkeiten kennengelernt haben.

Durch die Typhierarchie auf `Person`, `Skier` und `Snowboarder` muss Generizität über mehrere Ebenen hinweg betrachtet werden, da vereinfachende Sichtweisen durch die von dieser Hierarchie unabhängigen Typen `SortedWood<SortedWood<Person>>` und `Wood<String>` ausgeschlossen sind.

Vorgegebene Testfälle stellen sicher, dass die Schwierigkeiten erkannt werden. Um Umgehungen zu vermeiden sind Typumwandlungen ebenso verboten wie das Ausschalten von Compilerhinweisen auf unsichere Verwendungen von Generizität. 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 innerer Klassen auf die Sichtbarkeit von Implementierungsdetails nach außen hat.

Was im Hinblick auf die Beurteilung wichtig ist:

Die insgesamt 100 für diese Aufgabe erreichbaren Punkte sind folgendermaßen auf die zu erreichenden Ziele aufgeteilt:

Generizität zusammen mit geforderten Untertypbeziehungen richtig verwendet, sodass die Tests (ohne Tricks beim Testen) durchführbar sind	35 Punkte
Zusicherungen richtig und sinnvoll eingesetzt	15 Punkte
Sichtbarkeit auf so kleine Bereiche wie möglich beschränkt	15 Punkte
Lösung wie vorgeschrieben und sinnvoll getestet	20 Punkte
Geforderte Funktionalität vorhanden (so wie in Aufgabenstellung beschrieben)	15 Punkte

Am wichtigsten ist die sinnvolle und korrekte Verwendung von Generizität. Es gibt bedeutende Punkteabzüge, wenn der Compiler mögliche Probleme im Zusammenhang mit Generizität meldet oder wichtige Teilaufgaben nicht gelöst

oder umgangen werden.

Ein zusätzlicher Schwerpunkt liegt auf dem gezielten Einsatz von Sichtbarkeit. Es gibt Punkteabzüge, wenn Programmteile, die überall sichtbar sein sollen, nicht `public` sind, oder Teile, die nicht für die allgemeine Verwendung bestimmt sind, unnötig weit sichtbar sind. Durch die Verwendung innerer 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.