Objektorientierte Programmiertechniken
LVA 185.A01, VU, 3 Ects, 2013/2014 W

5. Übungsaufgabe

Themen:

Generizität, Sichtbarkeit, Container und Iteratoren

Termine:

Ausgabe: 13.11.2013
Abgabe: 27.11.2013, 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:

- Ein Objekt vom Typ *SList* stellt eine Folge von Elementen dar, wobei Elementtypen durch einen Typparameter bestimmt werden. *SList* implementiert *java.lang.Iterable* sowie folgende Methoden:
  - Zum Einfügen eines Elements gibt es *add* mit zwei Parametern. Ein Parameter bestimmt das einzufügende Element, der andere die Position als ganze Zahl (ganz am Ende bei -1, ganz am Anfang bei 0, direkt nach dem i-ten Eintrag für $i \geq 1$).
  - Der durch *iterator* erzeugte Iterator gibt nacheinander alle Elemente in der gegebenen Reihenfolge zurück. Der Iterator darf keine

UnsupportedOperationException werfen und muss daher *remove* implementieren – siehe *java.util.Iterator*.

- *AList* ist mit entsprechenden Typparameterersetzungen ein Untertyp von *SList*. Jedes Element in einem Objekt vom Typ *AList* ist mit einer möglicherweise leeren Liste assoziierter Elemente verknüpft, und die Typen dieser assoziierten Elemente werden durch einen weiteren Typparameter bestimmt. Folgende Methoden gibt es (zusätzlich):
  - Zum Einfügen eines Elements gibt es *add* auch mit drei Parametern, wobei der dritte Parameter als Array die mit dem einzufügenden Element assoziierten Elemente enthält.
  - Der durch *iterator* erzeugte Iterator ist gegenüber dem von *SList* derart erweitert, dass ein Aufruf von *iterator* im Iterator einen weiteren Iterator liefert, der über die mit dem aktuellen Element assoziierten Elemente in beliebiger Reihenfolge iteriert. Dieser Iterator braucht *remove* nicht zu implementieren.
- *DList* ist mit entsprechenden Typparameterersetzungen ein Untertyp von *AList*. Alle Elemente in einem Objekt vom Typ *DList* müssen die Methode *dependsOn* mit einem Parameter unterstützen, die als Ergebnis genau dann *true* zurückgibt, wenn das Element auf irgendeine Weise vom Parameter abhängt. Mit diesen Elementen assoziierte Elemente brauchen diese Methode nicht zu unterstützen. Ein Objekt vom Typ *DList* hat eine parameterlose Methode *consistent*, die genau dann *true* zurückgibt, wenn kein Element in diesem Objekt von irgendeinem nachfolgenden Element (im selben Objekt) entsprechend *dependsOn* abhängt.
- Das Interface *Dependent* beschreibt nur die Methode *dependsOn* wie in *DList* benötigt. Mehr darf im Interface nicht enthalten sein.
- Die Klasse *Prefixed* implementiert *Dependent*. Über den Konstruktor wird jedes Objekt dieser Klasse mit einem String initialisiert. Ein Aufruf von *x.dependsOn(y)* (wobei *x* und *y* beide vom Typ *Prefixed* sein müssen) liefert genau dann *true*, wenn der String in *y* ein Präfix des Strings in *x* ist (das heißt, wenn der String in *x* den String in *y* ganz am Anfang enthält).
- Auch die abstrakte Klasse *Category* implementiert *Dependent*, wobei *dependsOn* zwei Instanzen von *Category* miteinander vergleicht. Sind die dabei verglichenen Objekte von unterschiedlichen Klassen, gibt *dependsOn* als Ergebnis *false* zurück. Ein Aufruf der parameterlosen Methode *uses* in einem Objekt vom Typ *Category* gibt die Anzahl der Aufrufe von *dependsOn* zurückgibt, in die das Objekt involviert war (entweder weil *dependsOn* im Objekt aufgerufen wurde oder ein Aufruf das Objekt als Argument verwendet hat).
- Die Klasse *IntCategory* ist von *Category* abgeleitet. Über den Konstruktor wird jedes Objekt dieser Klasse mit zwei ganzen Zahlen *a* und *b* initialisiert. Ein Aufruf von *x.dependsOn(y)* liefert genau dann *true*, wenn *x* und *y* vom Typ *IntCategory* sind, *a* in *x* und *a* in *y* gleich sind und *b* in *x* größer oder gleich *b* in *y* ist. Die Methode *toString* soll die beiden Zahlen formatiert in einem String ausgeben.
- Die Klasse *CharCategory* ist ebenso von *Category* abgeleitet. Über den Konstruktor wird jedes Objekt dieser Klasse mit einem Zeichen vom Typ

char initialisiert. Ein Aufruf von *x.dependsOn(y)* liefert genau dann *true*, wenn *x* und *y* vom Typ *CharCategory* sind und dasselbe Zeichen enthalten. Die Methode *toString* soll das Zeichen als Teil eines Strings ausgeben.

Ein Aufruf von `java Test` soll wie gewohnt die wichtigsten Normal- und Grenzfälle überprüfen und die Ergebnisse in allgemein verständlicher Form darstellen. Anders als in bisherigen Aufgaben sind die Überprüfungen jedoch teilweise vorgegeben und in dieser Reihenfolge auszuführen:

1. Erzeugen Sie eine Instanz von *SList*, deren Elemente vom Typ *Prefixed* sind. Fügen Sie einige Elemente ein, lesen Sie alle Elemente der Menge über den Iterator aus, und schreiben Sie die Strings in den Elementen in die Standard-Ausgabe. Führen Sie Änderungen durch und geben Sie die Elemente erneut aus. Diesen Vorgang können Sie mit unterschiedlichen Änderungen so oft wiederholen, wie es Ihnen als nötig erscheint.
2. Erzeugen Sie eine Instanz von *AList*, deren Elemente vom Typ *String* und deren assoziierte Elemente vom Typ *Integer* sind. Fügen Sie einige Elemente ein, führen Sie Änderungen durch und geben Sie die Elemente und assoziierten Elemente aus, so oft es Ihnen für den Test als notwendig erscheint.
3. Erzeugen Sie eine Instanz von *DList*, deren Elemente vom Typ *IntCategory* und deren assoziierte Elemente vom Typ *CharCategory* sind. Fügen Sie einige Elemente und damit assoziierte Elemente ein, lesen Sie alles über die Iteratoren aus, und schreiben Sie die über *toString* erhaltenen Zeichenketten in die Standard-Ausgabe. Testen Sie Änderungen ähnlich wie bei Punkt 1 und 2. Testen Sie auch die Methode *consistent* so, dass sie nach Änderungen zu anderen Ergebnissen führt.
4. Betrachten Sie die in Punkt 3 erzeugte Liste als Instanz von *AList*, fügen Sie noch einige Elemente und damit assoziierte Elemente hinzu, lesen Sie alles über Iteratoren aus und schreiben Sie das Gelesene in die Standard-Ausgabe.
5. Betrachten Sie die in Punkt 3 erzeugte und in Punkt 4 veränderte Liste als Instanz von *SList*, fügen Sie noch einige Elemente hinzu, lesen Sie die zugreifbaren Elemente über Iteratoren aus und schreiben Sie diese in die Standard-Ausgabe.
6. Erzeugen Sie ein Objekt vom Typ *DList*, dessen Elemente vom Typ *Category* und dessen assozierte Elemente vom Typ *Object* sind. Lesen Sie alle Elemente und assoziierten Elemente der in Punkt 3 erzeugten und in Punkt 4 und 5 veränderten Liste aus und fügen Sie diese (sowohl Instanzen von *IntCategory* wie auch solche von *CharCategory* sollen vorkommen) in die neue Liste ein. Außerdem soll jedes Element in der neuen Menge zwei assoziierte Elemente haben: den durch *toString* aus dem Element erzeugten String und (als Instanz von *Integer*) die zum Zeitpunkt des Einfügens von einem Aufruf von *uses* zurückgegebene Zahl. Lesen Sie alle Elemente und assoziirten Elemente der neuen Menge aus und schreiben Sie diese in die Standard-Ausgabe. Überprüfen Sie die Methode *consistent*.
7. Machen Sie bei Bedarf weitere Tests, die Ihnen als nötig erscheinen, die

Sie aber nicht in den anderen Punkten untergebracht haben.

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

Wie die Aufgabe zu lösen ist:

Von den oben beschriebenen Interfaces, Klassen und Methoden wird erwartet, dass sie überall verwendbar sind (außer den Daten in den Untertypen von *Dependent*). Der Bereich, in dem weitere Methoden, Variablen, Klassen und Interfaces sichtbar sind, soll so klein wie möglich gehalten werden.

SList, *AList* und *DList* sind ohne Verwendung von Arrays (außer als Parametertypen wo gefordert), ohne vorgefertigte Container-Klassen (wie *LinkedList*, *HashSet*, etc.) und ohne vorgefertigte Implementierungen von Iteratoren zu lösen, aber die vordefinierten Interfaces *Iterator* und *Iterable* sollen verwendet werden. Die benötigten Container und Iteratoren sind selbst zu implementieren.

Typsicherheit soll so weit wie möglich vom Compiler garantiert werden. Auf die Verwendung von Typumwandlungen (Casts) und ähnliche Techniken ist daher zu verzichten, auch zu Testzwecken, und der Compiler darf keine Hinweise auf mögliche Probleme im Zusammenhang mit Generizität geben. Achtung: Übersetzen Sie die Klassen mit dem Compiler-Flag *-Xlint:unchecked*, das genaue Compiler-Meldungen im Zusammenhang mit Generizität einschaltet. 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:

Diese Aufgabe ist künstlich konstruiert und hat keinen erkennbaren praktischen Hintergrund. Sie ist so konstruiert, dass dabei zahlreiche Schwierigkeiten in großer Dichte auftreten, für die wir Lösungen kennengelernt haben. Das Erkennen entsprechender Situationen und die Beherrschung der Lösungsansätze soll gezeigt werden.

Durch die Untertypen von *Category* muss Generizität über mehrere Ebenen hinweg betrachtet werden. Vereinfachende Sichtweisen sind durch den von diesen Typen unabhängigen Typ *Prefixed* ausgeschlossen. Dieser Teil der Aufgabe ist mit einem falschen Lösungsansatz kaum lösbar.

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 innerer Klassen (speziell für Iteratoren) 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 (dort, wo sie verboten sind) oder vorgefertigten Containern und Iteratoren oder das Ausschalten von Überprüfungen durch @SuppressWarning – zu bedeutenden Punkteabzügen.