Objektorientierte Programmiertechniken
LVA 185.A01, VU, 3 Ects, 2012/2013 W

5. Übungsaufgabe

Themen:

Generizität, Container und Iteratoren

Termine:

Ausgabe:	14.11.2012
reguläre Abgabe:	21.11.2012, 12:00 Uhr
nachträgliche Abgabe:	28.11.2012, 12:00 Uhr

Abgabeverzeichnis:

Gruppe/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:

- Eine Instanz von *Set* stellt eine Menge dar, deren Elementtypen durch einen Typparameter bestimmt werden. *Set* implementiert das Interface *java.lang.Iterable* sowie folgende Methoden:
  - *insert* nimmt ein Argument, das in die Menge eingefügt wird, wenn nicht bereits ein identisches Element vorhanden war. Mehrere gleiche Elemente dürfen in der Menge sein (aber nicht mehrere identische).
  - *iterator* liefert als Ergebnis einen Iterator, über den nacheinander auf alle Elemente der Menge in nicht weiter bestimmter Reihenfolge

zugegriffen werden kann. Der Iterator muss auch *remove* implementieren und darf keine *UnsupportedOperationException* werfen (siehe *java.lang.Iterator*).

- Eine Instanz von *OrderedSet* stellt eine Menge dar, die wie *Set* das Interface *java.lang.Iterable* implementiert. Ein Typparameter bestimmt den Typ der Elemente. Die Elemente müssen eine Methode *shorter* mit einem Parameter unterstützen, die genau dann *true* zurückgibt, wenn *this* (auf nicht näher bestimmte Weise) kürzer als das übergebene Argument ist. *OrderedSet* implementiert folgende Methoden:
  - *insert* entspricht der gleichnamigen Methode in *Set*.
  - *iterator* liefert als Ergebnis einen Iterator, über den nacheinander auf alle Elemente der Menge zugegriffen werden kann, wobei nachfolgende Elemente entsprechend *shorter* niemals kürzer sein dürfen als vorangegangene. Der Iterator muss *remove* implementieren und darf keine *UnsupportedOperationException* werfen.

  Wenn möglich soll *OrderedSet* ein Untertyp von *Set* sein.
- *OrderedMap* unterscheidet sich von *OrderedSet* nur dadurch, dass jedes Element der Menge auf weitere Objekte verweisen kann. Der Typ dieser Objekte wird durch einen weiteren Typparameter bestimmt. In *OrderedMap* werden folgende Methoden benötigt:
  - *insert* wie in *OrderedSet* und *Set*.
  - *iterator* entspricht im Wesentlichen der gleichnamigen Methode in *OrderedSet*. Der zurückgegebene Iterator hat jedoch (neben *hasNext*, *next* und *remove*) ebenfalls eine Methode *iterator*, die einen Iterator über den Objekten zurückgibt, auf welche das aktuelle Element verweist. Der zuletzt genannte Iterator gibt die Objekte in der beim Einfügen bestimmten Reihenfolge zurück. Neben *hasNext* und *next* unterstützt er auch die Methoden *remove* zum Löschen des zuletzt zurückgegebenen Elements sowie *add* zum Einfügen eines neuen Elements an der aktuellen Position. Diese Methoden entsprechen jenen in der API-Spezifikation von *java.util.ListIterator* (jedoch ohne weitere Methoden wie *previous*, *set*, etc.).

  Wenn möglich soll *OrderedMap* ein Untertyp von *OrderedSet* oder zumindest *Set* sein.
- *Shorter* beschreibt nur die Methode *shorter* (wie in *OrderedSet* und *OrderedMap* gefordert).
- *ElapsedTime* stellt die gemessene Zeit eines Vorgangs (z.B. einer Programmausführung) dar und ist ein Untertyp von *Shorter*, wobei *shorter* zwei Instanzen von *ElapsedTime* hinsichtlich der gemessenen Zeit vergleicht. *ElapsedTime* beschreibt auch die Methode *count*, welche die Anzahl durchgeführter Messungen ermittelt – genaue Bedeutung von Untertypen abhängig.
- *MeanElapsedTime* ist ein Untertyp von *ElapsedTime* und stellt die durchschnittliche Zeit einer Messreihe dar. Die einzelnen Zeiten in der Messreihe werden in einem (nach außen nicht sichtbaren) Objekt des Typs *Set<Double>* abgelegt. Für den Vergleich mit *shorter* wird der

Durchschnittswert herangezogen, und *count* liefert die Anzahl der Messungen. Methoden zum Hinzufügen weiterer Messwerte sowie zur Ermittlung des größten Messwerts sind nötig. Andere Messwerte sollen nicht direkt abfragbar sein.
- *CompositeTime* ist ein Untertyp von *ElapsedTime* und stellt die Gesamtzeit eines aus mehreren Einzelzeiten zusammengesetzten Vorgangs dar. Die Einzelzeiten werden in einer Instanz von *Double[]* abgelegt, wobei das Array im Konstruktor gesetzt wird. Für den Vergleich mit *shorter* wird die Summe der Einzelzeiten herangezogen, und *count* liefert die Länge des Arrays. Eine Methode zur Ermittlung der kürzesten Einzelzeit ist nötig, aber andere Einzelzeiten sollen nicht abfragbar sein.
- *Description* ist ein Untertyp von *Shorter*, der einen beschreibenden Text darstellt. Der Text wird im Konstruktor gesetzt und durch *toString* ausgelesen. Ein Vergleich mit *shorter* vergleicht die Texte anhand ihrer Längen in Zeichen. Es gibt eine Methode um die Anzahl der Zeilen zu ermitteln.

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 *OrderedSet* deren Elemente vom Typ *Description* sind. Fügen Sie einige Elemente in unsortierter Reihenfolge ein, lesen Sie alle Elemente der Menge über den Iterator aus, und schreiben Sie die Anzahlen der Zeilen 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 *OrderedMap*, deren Elemente vom Typ *MeanElapsedTime* sind und die auf Objekte vom Typ *CompositeTime* verweisen – nicht sehr sinnvoll, aber gut zum Testen geeignet. Fügen Sie einige Elemente und damit verbundene Objekte ein, lesen Sie alles über die Iteratoren aus, und schreiben Sie jeweils den größten Messwert (für Elemente) bzw. die kürzeste Einzelzeit (für Objekte, auf die Elemente verweisen) in die Standard-Ausgabe. Testen Sie Änderungen ähnlich wie bei Punkt 1.
3. Falls *OrderedMap* mit entsprechenden Typparametererersetzungen ein Untertyp von *OrderedSet* ist, betrachten Sie die in Punkt 2 erzeugte Menge als Instanz von *OrderedSet*, fügen Sie noch einige Elemente ein, lesen Sie alle Elemente über den Iterator aus, und schreiben Sie die größten Messwerte in die Standard-Ausgabe. Falls *OrderedMap* kein Untertyp von *OrderedSet* ist, geben Sie anstelle der Testergebnisse eine Begründung dafür aus, warum zwischen diesen Typen keine Untertypbeziehung besteht.
4. Erzeugen Sie eine Instanz von *OrderedSet*, deren Elemente vom Typ *ElapsedTime* sind. Lesen Sie alle Elemente der in Punkt 2 erzeugten (und

möglicherweise in Punkt 3 erweiterten) Menge und alle Objekte, auf welche die Elemente verweisen, aus und fügen Sie diese (Instanzen von *MeanElapsedTime* ebenso wie von *CompositeTime*) in die neue Menge ein. Lesen Sie alle Elemente der neuen Menge aus, und schreiben Sie die durch *count* ermittelten Werte in die Standard-Ausgabe.
5. Bei Bedarf weitere Testsfälle, die Ihnen nötig erscheinen, die Sie aber nicht in den anderen Punkten untergebracht haben.

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 *MeanElapsedTime* und *CompositeTime*). Der Bereich, in dem weitere Methoden, Variablen, Klassen und Interfaces sichtbar sind, soll so klein wie möglich gehalten werden.

Alle Klassen in dieser Aufgabe sind ohne Verwendung von Arrays (außer in *CompositeTime* und falls gewünscht auschließlich für Testzwecke in *Test*), 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.

Durch die Untertypen von *ElapsedTime* muss Generizität über mehrere Ebenen hinweg betrachtet werden. Vereinfachende Sichtweisen sind durch den von diesen Typen unabhängigen Typ *Description* ausgeschlossen. Dieser Teil der Aufgabe ist z.B. durch Verwendung von Typ-Wildcards einfach lösbar, bei falschem Ansatz aber praktisch unlösbar.

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:

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 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.

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.