5. Programmieraufgabe

Objektorientierte Programmiertechniken

LVA-Nr. 185.A01
2021/2022 W
TU Wien

Themen:

Generizität, Container, Iteratoren

Ausgabe:

23. 11. 2021

Abgabe (Deadline):

30. 11. 2021, 10:00 Uhr

Abgabeverzeichnis:

`Aufgabe5`

Programmaufruf:

`java Test`

Grundlage:

Skriptum, Schwerpunkt auf 3.1 und 3.2

Kontext

Baumgesellschaften in Wäldern sind als Container darstellbar. Zur Reduktion von Programmtexten bietet sich Generizität an. Für erste Tests benötigen wir folgende, bei Bedarf generische Interfaces oder Klassen:

`Relation` ist ein generisches Interface mit zwei Methoden:

- `related` hat zwei Parameter (eventuell unterschiedlicher Typen bestimmt durch Typparameter) und gibt einen Wahrheitswert zurück. Das Ergebnis von `r.related(x,y)` besagt, ob eine nicht näher bestimmte Beziehung zwischen `x` und `y` gilt, wobei das Ergebnis nur von `x` und `y` sowie unveränderlichen Werten in `r` abhängt. Eine Methodenausführung lässt `x` und `y` unverändert, eine Änderung von `r` ist aber erlaubt.
- `invoked()` gibt (zwecks Effizienz-Analyse) die Anzahl der bisherigen Aufrufe von `related` in diesem Objekt zurück.

`Group` ist ein Interface mit zwei Typparametern `X` und `Y` und den Obertypen `java.lang.Iterable<X>` und `Relation<...>`. Objekte von `Group<X,Y>` sind Container mit Einträgen vom Typ `X`, die mit Einträgen vom Typ `Y` eines (anderen) Containers vergleichbar sind. Folgende Methoden werden benötigt:

- `add` mit einem Argument vom Typ `X` stellt sicher, dass das Argument ein Eintrag im Container ist. Das heißt, falls der Container noch kein identisches Objekt als Eintrag enthält, wird es eingefügt, aber mehrere identische Objekte dürfen im Container nicht vorkommen.
- `related` (geerbt von `Relation`) nimmt ein Argument `x` vom Typ `X` und ein Argument `y` vom Typ `Y`. Das Ergebnis hängt von der Implementierung von `Group` ab.
- `iterator` (geerbt von `Iterable`) gibt einen Iterator zurück, der in beliebiger Reihenfolge alle Einträge `x` im Container zurückgibt, für die `related(x,y)` für mindestens ein `y` gilt, wobei die dafür in Betracht kommenden `y` von der Implementierung von `Group` bestimmt werden. Die Methode `remove` im Iterator muss so implementiert sein, dass der zuletzt von `next` zurückgegebene Eintrag aus dem Container entfernt wird.

`MultiGroup<X,Y>` ist eine Implementierung von `Group<X,Y>`, deren Konstruktor ein Objekt `a` vom Typ `Group<...>` (möglichst wenig einschränkend) und ein Objekt `r` vom Typ `Relation<...>` unveränderlich festlegt. Anfangs ist der Container leer. Ein Methodenaufruf `related(x,y)` in `this` gibt das Ergebnis von `r.related(x,y)` zurück (Aufruf an `r` delegiert). In von `iterator()` erzeugten Iteratoren laufen die `y` in Vergleichen `related(x,y)` über von einem Iterator auf `a` zurückgegebene Objekte, die `x` über Einträge in `this`.

`SingleGroup<X>` ist eine Implementierung von `Group<X,X>`. Anfangs ist der Container leer. Ein Methodenaufruf `related(x,y)` in `this` gibt das Ergebnis von `x==y` zurück. Die `y` in Vergleichen `related(x,y)` in von `iterator()` erzeugten Iteratoren sind stets identisch mit `x`, wodurch alle Einträge im Container zurückgegeben werden; ein Iterator kann auf Aufrufe von `related` verzichten.

`Numeric` implementiert `Relation<...>`. Ein Aufruf von `related(x,y)` vergleicht zwei `int`-Werte und gibt `true` zurück, wenn sich `x` und `y` um maximal `c` voneinander unterscheiden, wobei `c` ein im Konstruktor gesetzter, unveränderlicher `int`-Wert größer 0 ist.

`Tree` ist ein einfaches Interface, dessen Instanzen Bäume sind. Die Methode `height()` gibt die Baumhöhe zurück.

`Fagus` ist ein Untertyp von `Tree`. Der Konstruktor setzt die Baumhöhe sowie den Anteil an Schattenblättern (Fließkommazahl zwischen 0 und 1, die besagt, welcher Anteil der Blätter wegen Beschattung kaum Zugang zu direktem Sonnenlicht hat). Die statische Methode `relation()` gibt ein neues Objekt von `Relation<...>` zurück, dessen Methode `related(x,y)` zwei Instanzen von `Fagus` vergleicht und `true` zurückgibt, wenn `x` höher ist und einen geringeren Anteil an Schattenblättern aufweist als `y` (wodurch `y` unterhalb von `x` wachsen und in `x` hinein wachsen könnte).

`Quercus` ist ein konkreter (also nicht abstrakter) Untertyp von `Tree`, der neben der Baumhöhe auch die Stammhöhe (Höhe, an der die Krone beginnt, daher kleiner als die Baumhöhe) über den Konstruktor definiert. Die statische Methode `relation()` gibt ein neues Objekt von `Relation<...>` zurück, dessen Methode `related(x,y)` ein `x` vom Typ `Quercus` mit einem `y` vom Typ `Tree` vergleicht und `true` zurückgibt, wenn die Stammhöhe von `x` mindestens so groß ist wie die Baumhöhe von `y` (wodurch `y` unterhalb von `x` wachsen könnte).

`QuercusRobur` ist ein Untertyp von `Quercus`, der neben den Methoden von `Quercus` auch eine Methode `resistance()` definiert, die eine über den Konstruktor gesetzte textuelle Beschreibung (String) der Wiederstandsfähigkeit gegenüber Klimaschwankungen zurückgibt.

Ein Objekt von `MultiGroup<Fagus,Fagus>` steht für eine Gruppe an Bäumen, unter denen eine weitere Baumgruppe wächst, darunter vielleicht weitere (Gruppen sind listenförmig verbunden), abgeschlossen durch ein Objekt von `SingleGroup<Fagus>`. Bäume anderer Arten können unter Buchen nicht wachsen. Ein Objekt von `Group<Quercus,Quercus>` könnte analog dazu aufgebaut sein. Unter Eichen wachsen aber auch andere Baumarten, sodass `Group<Quercus,Fagus>` und `Group<Quercus,Tree>` sinnvoll ist. Statt `Quercus` kann überall auch `QuercusRobur` verwendet werden, wodurch die Methode `resistance()` aufrufbar wird, aber dann kann eine Baumgruppe nur Stileichen umfassen, keine Traubeneichen. Daher muss sowohl mit `Quercus` als auch `QuercusRobur` getestet werden. Für erste Tests sollten diese Baumarten reichen. Objekte von `Group<Integer,Integer>` sind möglicherweise für diverse Verwaltungszwecke einsetzbar.

Welche Aufgabe zu lösen ist

Implementieren Sie die oben beschriebenen Klassen und Interfaces mit Hilfe von Generizität. Vorgefertigte Container, Arrays, Raw-Types, explizite Casts, explizite dynamische Typabfragen und ähnliche Sprachkonzepte dürfen dabei nicht verwendet werden.

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

vorgegebene Tests

1. Erzeugen Sie mindestens je ein Objekt sinngemäß folgender Typen (wozu Objekte weiterer Typen nötig sind):

   ```
   MultiGroup<Fagus, Fagus>
   MultiGroup<QuercusRobur, Fagus>
   MultiGroup<Quercus, Fagus>
   MultiGroup<QuercusRobur, QuercusRobur>
   MultiGroup<QuercusRobur, Quercus>
   MultiGroup<Quercus, QuercusRobur>
   MultiGroup<Quercus, Quercus>
   MultiGroup<QuercusRobur, Tree>
   MultiGroup<Quercus, Tree>
   MultiGroup<Integer, Integer>
   ```

   Möglichst viele dieser Objekte sollen über den Parameter `a` des Konstruktors listenförmig verbunden werden. Bei der Erzeugung mindestens eines Objekts vom Typ `MultiGroup<...,A>` muss als Parameter `a` ein Objekt vom Typ `MultiGroup<B,...>` verwendet werden, wobei `A` und `B` verschieden sind, aber in einer Untertypbeziehung stehen. Befüllen Sie die Container mit einigen Einträgen.

2. Überprüfen Sie die Funktionalität mittels Iterationen durch Container, Löschen und (erneutes) Einfügen von Objekten und die Ausgabe abfragbarer Daten jeder Art in die Standardausgabe.

3. Wählen Sie ein in Punkt 1 eingeführtes Objekt `u` von einem Typ `MultiGroup<Quercus,...>`, ein `v` von `SingleGroup<Tree>` und ein `w` von `Group<QuercusRobur,...>`. Lesen Sie über den Iterator Einträge aus `w` aus, rufen Sie auf ihnen `resistance()` auf und fügen Sie sie mittels `add` zu `u` und `v` hinzu.

   Generizität so planen, dass das geht

4. Geben Sie für alle Objekte von `Relation<...>` die von `invoked()` zurückgegebenen Zahlen in die Standardausgabe aus.

5. Machen Sie optional weitere Überprüfungen, die jedoch nicht direkt in die Beurteilung einfließen.

   nicht verpflichtend

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.

Aufgabenaufteilung beschreiben

Wie die Aufgabe zu lösen ist

Sichtbarkeit beachten

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.

Verbote beachten!!

Alle Teile dieser Aufgabe sind ohne Arrays und ohne vorgefertigte Container (wie z. B. Klassen des Collection-Frameworks) zu lösen. Benötigte Container und Iteratoren sind selbst zu schreiben.

Generizität statt dynamischer Prüfungen

Typsicherheit soll vom Compiler garantiert werden. Auf Typumwandlungen (Casts) und ähnliche Techniken ist zu verzichten, und der Compiler darf keine Hinweise auf mögliche Probleme im Zusammenhang mit Generizität geben. Raw-Types dürfen nicht verwendet werden.

Compiler-Feedback einschalten

Übersetzen Sie die Klassen mittels `javac -Xlint:unchecked ...` Dieses Compiler-Flag schaltet genaue Compiler-Meldungen im Zusammenhang mit Generizität ein. Andernfalls bekommen Sie auch bei schweren Fehlern möglicherweise nur eine harmlos aussehende Meldung. Überprüfungen durch den Compiler dürfen nicht ausgeschaltet werden.

Beginnen Sie frühzeitig mit dem Testen. Die Aufgabe enthält Schwierigkeiten, auf die Sie vermutlich erst beim Testen aufmerksam werden.

Was im Hinblick auf die Beurteilung wichtig ist

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

Schwerpunkte berücksichtigen

- Generizität und geforderte Untertypbeziehungen richtig eingesetzt, sodass die Tests ohne Tricks durchführbar sind — 40 Punkte
- Lösung wie vorgeschrieben und sinnvoll getestet — 20 Punkte
- Zusicherungen konsistent und zweckentsprechend — 15 Punkte
- Sichtbarkeit auf kleinstmögliche Bereiche beschränkt — 15 Punkte
- Lösung vollständig (entsprechend Aufgabenstellung) — 10 Punkte

Am wichtigsten ist die korrekte Verwendung von Generizität. Es gibt bedeutende Punkteabzüge, wenn der Compiler mögliche Probleme im Zusammenhang mit Generizität meldet oder wichtige Teilaufgaben nicht gelöst bzw. umgangen werden. Beachten Sie, dass Raw-Types nicht verwendet werden dürfen, der Compiler aber auch mit `-Xlint:unchecked` nicht alle Verwendungen von Raw-Types meldet.

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 (anonymer) 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. Geforderte Untertypbeziehungen müssen gegeben sein. Nötige Zusicherungen, die aus „Kontext“ hervorgehen, müssen als Kommentare im Programmtext ersichtlich sein.

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

Aufgabe nicht abändern

Warum die Aufgabe diese Form hat

Die Aufgabe ist so konstruiert, dass dabei Schwierigkeiten auftauchen, für die wir Lösungsmöglichkeiten kennengelernt haben. Wegen der vorgegebenen, in die Typparameter einzusetzenden Typen muss Generizität über mehrere Ebenen hinweg betrachtet werden. Vorgegebene Testfälle stellen sicher, dass einige bedeutende Schwierigkeiten erkannt werden. Um Umgehungen außerhalb der Generizität zu vermeiden, sind Typumwandlungen ebenso verboten wie das Ausschalten von Compilerhinweisen auf unsichere Verwendungen von Generizität. Das Verbot der Verwendung vorgefertigter Container-Klassen verhindert, dass Schwierigkeiten nicht selbst gelöst, sondern an Bibliotheken weitergereicht werden. Zusätzlich wird das Erstellen typischerweise generischer Programmstrukturen geübt.

Schwierigkeiten erkennen
Skriptum anschauen

Daneben wird auch der Umgang mit Sichtbarkeit geübt. Am Beispiel von Iteratoren soll intuitiv klar werden, welchen Einfluss innere Klassen auf die Sichtbarkeit von Implementierungsdetails haben.

innere Klassen verwenden