Interfaces Konkrete Klassen

5. Programmieraufgabe

Programmierparadigmen
LVA-Nr. 194.023
2023/2024 W
TU Wien

Themen:
Generizität, Container, Iteratoren

Ausgabe:
13. 11. 2023

Abgabe (Deadline):
27. 11. 2023, 14:00 Uhr

Abgabeverzeichnis:
`Aufgabe5`

Programmaufruf:
`java Test`

Grundlage:
Skriptum, Schwerpunkt auf 4.1 und 4.2

Kontext

Sammlungen von Objekten aller Art sind auf natürliche Weise als Container darstellbar. Zu deren Implementierung bietet sich Generizität an. Für unsere Formicariums-Software benötigen wir folgende, bei Bedarf generische Interfaces oder Klassen:

`Calc` ist ein generisches Interface mit einem Typparameter `R` und folgenden Methoden, deren Ergebnisse nur von `this` und einem Parameter abhängen und die `this` und den Parameter nicht verändern:

- `sum` mit einem Parameter und Ergebnis vom Typ `R` gibt die Summe von `this` und dem Parameter zurück, wobei die Summe nicht unbedingt eine Zahlensumme sein muss, sondern auch anders definiert sein kann.
- `ratio` mit einem Parameter vom Typ `int` und einem Ergebnis vom Typ `R` gibt das Ergebnis der Division von `this` durch den Parameter zurück, wobei eine Division auch anders definiert sein kann als es auf Zahlen üblich ist.
- `atLeast` mit einem Parameter vom Typ `R` und einem Ergebnis vom Typ `boolean` gibt genau dann `true` zurück, wenn `this` größer oder gleich dem Parameter ist. Auch Größenvergleiche basieren nicht notwendigerweise auf Vergleichen von Zahlen.

`Rated` ist ein generisches Interface mit zwei Typparametern `P` und `R` sowie folgenden Methoden:

- `rated` hat einen Parameter `p` vom Typ `P` und gibt ein Ergebnis vom Typ `R` zurück. Ein Aufruf gibt eine Beurteilung von `this` hinsichtlich des Kriteriums `p` zurück. Das Ergebnis hängt nur von unveränderlichen Werten in `this` und `p` ab. Eine Methodenausführung lässt `this` und `p` unverändert.
- `setCriterion` hat einen Parameter `p` vom Typ `P` und gibt kein Ergebnis zurück. Ein Aufruf legt den Parameter `p` in `this` ab und sorgt dafür, dass künftige Aufrufe der unten beschriebenen Methode vom Wert von `p` abhängen.
- `rated` existiert auch in einer Variante ohne Parameter. Aufrufe von `rated()` bewirken das Gleiche wie von `rated(p)`, wobei `p` der zuletzt von `setCriterion(p)` gesetzte Wert ist. Das Ergebnis ist implementierungsabhängig, wenn in diesem Objekt zuvor nie `setCriterion(p)` aufgerufen wurde.

overload

`RatedSet` ist ein Interface mit drei Typparametern `X`, `P` und `R` und dem Obertyp `java.lang.Iterable<X>`. Ein Objekt davon ist ein Container mit Einträgen der Typen `X` und `P`, wobei `X` Untertyp von `Rated` (mit passenden Typparameterersetzungen) sein muss, sodass für jeden Eintrag `x` vom Typ `X` und jeden Eintrag `p` vom Typ `P` die

extends Iterable

Methode `x.rated(p)` aufrufbar ist und ein Ergebnis des Typs `R` zurückgibt. 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 wenn ein identisches Objekt zuvor schon mittels `add` eingefügt wurde, wird es nicht noch einmal eingefügt.
- `addCriterion` mit einem Argument vom Typ `P` ist wie `add` definiert, abgesehen davon, dass es um Einträge vom Typ `P` geht und keine identischen Objekte, die mittles `addCriterion` eingefügt werden, mehrfach vorkommen dürfen.
- `iterator` (definiert in `Iterable`) gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge im Container läuft, die mittels `add` eingefügt wurden. Die Methode `remove` im Iterator muss so implementiert sein, dass der zuletzt von `next` zurückgegebene Eintrag aus dem Container entfernt wird.

  Anonyme Klasse
- `iterator` mit einem Parameter `p` vom Typ `P` und einem Parameter `r` vom Typ `R` gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge `x` im Container läuft, die mittels `add` eingefügt wurden und für die `x.rated(p)` ein Ergebnis liefert, das größer oder gleich `r` ist. Größenvergleiche erfolgen durch eine Methode aus `Calc`. Die Methode `remove` im Iterator muss so implementiert sein, dass der zuletzt von `next` zurückgegebene Eintrag aus dem Container entfernt wird.
- `iterator` mit nur einem Parameter `r` vom Typ `R` gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge `x` im Container läuft, die mittels `add` eingefügt wurden und für die der Durchschnitt aller durch `x.rated(p)` ermittelten Werte (für alle mittels `addCriterion` eingefügten Einträge `p` im Container) größer oder gleich `r` ist. Größenvergleiche und Berechnungen von Durchschnittswerten (Summenbildung mittels `sum`, Division durch die Anzahl aufsummierter Werte mittels `ratio`) erfolgen durch Methoden aus `Calc`. Die Methode `remove` im Iterator muss den zuletzt von `next` zurückgegebenen Eintrag aus dem Container entfernen.
- `criterions` ohne Parameter gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge im Container läuft, die mittels `addCriterion` eingefügt wurden. `remove` im Iterator muss so implementiert sein, dass der zuletzt von `next` zurückgegebene Eintrag aus dem Container entfernt wird.

`StatSet` implementiert `RatedSet`. Typparameter sind wie in `RatedSet`. Zusätzlich zu den Methoden von `RatedSet` gibt es die parameterlose Methode `statistics`, die Informationen zur Anzahl aller bisher erfolgten Aufrufe alle Methoden in diesem Objekt als Zeichenkette zurückgibt (jede Methode einzeln aufgelistet), einschließlich der Aufrufe der Methoden in den dazu gehörenden Iteratoren. Zwei Objekte von `StatSet` werden als gleich betrachtet (`equals`), wenn sie identische Einträge jeder Art enthalten, ungeachtet der Reihenfolge.

`CompatibilitySet` ist eine Implementierung von `RatedSet`, bei der für `X` und `P` der gleiche Typ eingesetzt wird. Das Ergebnis von `rated` drückt aus, wie gut Objekte des gleichen Typs zusammenpassen. Wie in `StatSet` gibt es die Methode `statistics`. Die zusätzliche Methode `identical` gibt einen Iterator zurück, der über alle Einträge iteriert, die sowohl mittels `add` als auch `addCriterion` eingefügt wurden (und daher identisch zwei Mal eingefügt wurden). Zwei Objekte von `CompatibilitySet` werden als gleich betrachtet, wenn sie (getrennt betrachtet für Objekte eingefügt mit `add` und solche eingefügt mit `addCriterion`) identische Einträge enthalten, ungeachtet der Reihenfolge.

`Quality` ist eine Implementierung von `Calc`. Es gibt nur vier voneinander verschiedene Objekte von `Quality`, die folgenden vier Qualitätsbeschreibungen entsprechen:

- „für den professionellen Einsatz geeignet“
- „für den semiprofessionellen Einsatz geeignet“
- „für den Einsatz im Hobbybereich geeignet“
- „nicht für den Einsatz geeignet“

Die Methode `atLeast` gibt `true` zurück, wenn der Wert von `this` in dieser Aufzählung nicht weiter unten steht als der Wert des Parameters; das entspricht der intuitiven Bedeutung. Die Methode `sum` gibt den kleineren, also in der Aufzählung weiter unten stehenden Wert von `this` und dem Parameter zurück; das entspricht der Qualitätsbeschreibung bei kombiniertem Einsatz entsprechender Produkte. Die Methode `ratio` gibt einfach nur `this` zurück, ohne den Parameter zu beachten. Die von `toString` zurückgegebene Zeichenkette liefert die entsprechende Qualitätsbeschreibung als Text.

`Part` ist ein Interface, das `Rated` erweitert. Der Typparameter `R` ist durch `Quality` ersetzt, der Typparameter `P` durch einen Untertyp von `Part`. Objekte von `Part` stellen Bestandteile von Formicarien dar. Die Art des Bestandteils wird durch die von `toString` zurückgegebene Zeichenkette beschrieben. Jeder Bestandteil hat einen bestimmten Einsatzbereich: professionell, semiprofessionell oder hobbymäßig (ähnlich wie in `Quality`, aber ungeeignet kann kein einzelner Bestandteil sein). Die Methode `rated` mit einem Parameter `p` gibt ein `Quality`-Objekt zurück, das eine Qualitätsbeschreibung für den gemeinsamen Einsatz von `this` und `p` darstellt. Häufig wird das der geringere Qualitätsanspruch von `this` und `p` sein, aber manchmal können Bestandteile inkompatibel sein, sodass das Ergebnis „nicht für den Einsatz geeignet“ darstellt.

`Arena` ist eine Implementierung von `Part`. Objekte davon stellen Arenen dar. Ein Formicarium kann beliebig viele Arenen enthalten, wodurch jede Arena mit anderen Teilen eines Formicariums kompatibel ist. Die Methode `volume` gibt das Fassungsvermögen der Arena (das ist im Wesentlichen ein Gefäß) in Liter zurück. Es gibt *keine* Methode namens `antSize`.

`Nest` ist eine Implementierung von `Part`. Objekte davon stellen Nester dar. Es wird davon ausgegangen, dass jedes Formicarium nur ein Nest enthalten kann. Daher sind zwei Objekte von `Nest` nicht miteinander kompatibel, was im Ergebnis von `rated` sichtbar werden soll. Die Methode `antSize` gibt die empfohlene Durchschnittsgröße von Ameisen in Millimetern zurück, für die das Nest geeignet ist. Es gibt *keine* Methode namens `volume`.

`Numeric` ist eine Implementierung von `Calc` und `Rated` auf Basis von `double`. Der Typparameter `R` ist in beiden Interfaces durch `Numeric` ersetzt. Die Methoden `sum`, `ratio` und `atLeast` bilden numerische Summen, Divisionen und Größenvergleiche von `double`-Werten, die in Objekten von `Numeric` abgelegt sind. Für den Typparameter `P` in `Rated` wird `java.util.function.DoubleUnaryOperator` verwendet, das ist ein funktionales Interface. Argumente von `rated` und `setCriterion` sind (in der Regel) Lambdas, die einen `double`-Wert als Parameter haben und einen `double`-Wert zurückgeben. Die Methode `rated` ruft die abstrakte Methode im Parameter `p` (das ist das Lambda) mit dem in `this` (vom Typ `Numeric`) abgelegten `double`-Wert auf, packt den daraus resultierenden `double`-Wert in ein neues Objekt von `Numeric` und gibt dieses zurück.

Alle in obigen Beschreibungen genannten Typparameter können bei Bedarf mit Schranken bzw. Wildcards versehen sein. Bitte beachten Sie, dass die oben genannten Typen nicht immer vollständig ausgeschrieben sind. Beispielsweise könnte es nötig sein, statt `Rated` eine generische Variante `Rated<...>` zu verwenden.

Ein Objekt vom Typ `StatSet<Part,Part,Quality>` oder etwas kürzer `CompatibilitySet<Part,Quality>` steht für ein Produktverzeichnis, das es ermöglicht, über Iteratoren zu prüfen, welche Produkte mit welchen bestimmten anderen Produkten (bezogen auf geforderte Qualitäten) kompatibel sind. Ein Objekt vom Typ `StatSet<Arena,Nest,Quality>` ermöglicht z. B. die gezielte Suche nach Arenen, die zu vorgegebenen Nestern passen. Objekte von `StatSet<Numeric,Numeric,Numeric>` und `CompatibilitySet<Numeric,Numeric>` haben im Gegensatz dazu den recht abstrakten Zweck, eine größere Zahl von Zahlen und einfachen Funktionen darauf in Beziehung zueinander zu setzen. Damit erlauben diese Objekte auch, die benötigten Klassen auf einfache Weise zu testen.

Welche Aufgabe zu lösen ist

Linked List in Set impl. schreiben

Implementieren Sie die oben beschriebenen Klassen und Interfaces mit Hilfe von Generizität. Vorgefertigte Container, Arrays, Raw-Types und ähnliche Sprachkonzepte dürfen dabei nicht verwendet werden. Casts und explizite dynamische Typabfragen sind ebenso verboten, außer in der Implementierung von `rated` in `Nest`.

Lassen Sie `StatSet<X,X,R>` und `CompatibilitySet<X,R>` in einer Untertypbeziehung zueinander stehen (für geeignete Typen `X` und `R`), oder geben Sie Gründe an, warum keine solche Untertypbeziehung (in beide Richtungen) bestehen kann.

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:

   ```
   StatSet<Numeric,Numeric,Numeric>
   StatSet<Part,Part,Quality>
   StatSet<Arena,Part,Quality>
   StatSet<Nest,Part,Quality>
   StatSet<Part,Arena,Quality>
   StatSet<Arena,Arena,Quality>
   StatSet<Nest,Arena,Quality>
   StatSet<Part,Nest,Quality>
   StatSet<Arena,Nest,Quality>
   StatSet<Nest,Nest,Quality>
   CompatibilitySet<Numeric,Numeric>
   CompatibilitySet<Part,Quality>
   CompatibilitySet<Arena,Quality>
   CompatibilitySet<Nest,Quality>
   ```

   Befüllen Sie die Container mit einigen Einträgen. Um den Schreibaufwand zu reduzieren, verwenden Sie dafür am besten (generische) Methoden, die Inhalte einer Collection in eine andere Collection kopieren (über Iteratoren).

2. Wählen Sie je ein in Punkt 1 eingeführtes Objekt:

   `a` vom Typ `StatSet<Part,...,Quality>`
   `b` vom Typ `StatSet<...,Part,Quality>`
   `c` vom Typ `StatSet<Arena,Nest,Quality>`

   Lesen Sie über Iteratoren alle Einträge aus `c` aus (sowohl die über `add` als auch die über `addCriterion` eingefügten), rufen Sie auf ihnen (je nach Typ) `volume()` oder `antSize()` auf und fügen Sie sie mittels `add` in `a` und mittels `addCriterion` in `b` ein.

   Generizität so planen, dass das geht

   aber ohne Typprüfung?

3. Falls `CompatibilitySet<X,R>` und `StatSet<X,X,R>` in einer Untertypbeziehung zeinander stehen, führen Sie Testfälle aus, die das überprüfen.

4. Überprüfen Sie die Funktionalität mittels Löschen und (erneutes) Einfügen von Objekten und die Ausgabe abfragbarer Daten jeder Art in die Standardausgabe. Bringen Sie alle Methoden der in *Kontext* beschriebenen Typen zur Ausführung. Zum Testen von `Numeric` sind mindestens fünf verschiedene Lambdas zu verwenden.

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

Zur einfacheren Testdurchführung ist die Verwendung von Arrays und vorgefertigten Containern in der Klasse `Test` (aber nur dort) erlaubt. Andere verbotene Sprachkonzepte (etwa Casts, dynamische Typprüfungen, Raw-Types) sind auch in `Test` verboten. Casts und dynamische Typprüfungen sind ausschließlich in `rated` in `Nest` erlaubt.

Außerdem soll die Datei `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

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.

Sichtbarkeit beachten

Alle Teile dieser Aufgabe (abgesehen von `Test`) sind ohne Arrays und ohne vorgefertigte Container (etwa Klassen des Collection-Frameworks) zu lösen. Benötigte Container und Iteratoren sind selbst zu schreiben.

Verbote beachten!!

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. Nur in `rated` in `Nest` dürfen dynamische Typabfragen und Casts verwendet werden, um dort gegebenenfalls nötige Typunterscheidungen zu vereinfachen. Raw-Types dürfen nicht verwendet werden.

Generizität statt dynamischer Prüfungen

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.

Compiler-Feedback einschalten

Beginnen Sie frühzeitig mit dem Testen. Die Aufgabe enthält Schwierigkeiten, auf die Sie vielleicht 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 getestet — 20 Punkte
- Zusicherungen konsistent und zweckentsprechend — 15 Punkte
- Sichtbarkeit richtig gewählt — 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

Iterator

Verwendung (anonymer) innerer Klassen und Lambdas 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. Außerdem wird das Erstellen typischer generischer Programmstrukturen geübt. In kleinen Teilbereichen wird auch der Umgang mit Lambdas geübt.

Schwierigkeiten erkennen

Skriptum anschauen

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

innere Klassen verwenden