5. Programmieraufgabe

Programmierparadigmen

LVA-Nr. 194.023
2024/2025 W
TU Wien

Themen:

Generizität, Container, Iteratoren

Ausgabe:

18. 11. 2024

Abgabe (Deadline):

02. 12. 2024, 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 eine Software zur digitalen Darstellung von Gebäuden benötigen wir folgende, meist generische Interfaces oder Klassen:

`Admin` ist ein generisches Interface mit den Typparametern `X` und `T` sowie folgenden Methoden, deren Ergebnisse (wie auch `this`) vom Typ `T` sind, jeweils nur von `this` und einem Parameter `x` vom Typ `X` abhängen und die `this` und `x` nicht ändern:

- `add` gibt etwas zurück, das `this` um `x` erweitert. Es ist nicht festgelegt, worin die Erweiterung besteht. Das Ergebnis ist identisch zu `this`, wenn `this` nicht um `x` erweiterbar ist.
- `remove` gibt etwas entsprechend `this` zurück, aus dem `x` entfernt ist. Das Ergebnis ist identisch zu `this`, wenn `x` nicht aus `this` entfernt werden kann.

`Approvable` ist ein generisches Interface mit zwei Typparametern `P` und `T` sowie folgenden Methoden:

- `approved` hat einen Parameter `p` vom Typ `P` und gibt ein Ergebnis vom Typ `T` zurück. Ein Aufruf gibt ein im Zusammenhang mit `this` und dem Kriterium `p` genehmigtes Objekt zurück, oder `null` wenn kein solches Objekt existiert. Ist `this` beispielsweise ein Innenraum und steht `p` für „Fluchtweg“, dann wird eine Beschreibung des genehmigten Fluchtwegs aus dem Raum zurückgegeben, oder `null` wenn es keine Benutzungsbewilligung für den Raum mit Fluchtweg gibt. `this`, `p` und das Ergebnis können aber auch etwas ganz anderes sein. Das Ergebnis hängt nur von Werten in `this` und `p` ab.
- `approve` mit Ergebnistyp `void` hat einen Parameter vom Typ `P` (ungleich `null`) und einen vom Typ `T` (kann auch `null` sein). Direkt nach einem Aufruf von `x.approve(p,t)` geben Aufrufe von `x.approved(p)` den Wert `t` zurück, während Ergebnisse der Aufrufe von `approved` mit anderen Argumenten als `p` durch `x.approve(p,t)` nicht beeiflusst werden.

`ApprovableSet` ist ein Interface mit drei Typparametern `X`, `P` und `T` und dem Obertyp `java.lang.Iterable<X>`. Ein Objekt davon ist ein Container mit Einträgen der Typen `X` und `P`, wobei `X` Untertyp von `Approvable` (mit passenden Typparameterersetzungen) ist, sodass für jeden Eintrag `x` vom Typ `X` und jeden Eintrag `p` vom Typ `P` die Methode `x.approved(p)` aufrufbar ist und ein Ergebnis des Typs `T` (oder `null`) 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.
- `iteratorAll` ohne Parameter gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge im Container läuft, die mittels `add` eingefügt wurden.
- `iterator` mit einem Parameter `p` vom Typ `P` gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge `x` im Container läuft, die mittels `add` eingefügt wurden und für die `x.approved(p)` ein Ergebnis ungleich `null` liefert.
- `iteratorNot` mit einem Parameter `p` vom Typ `P` gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge `x` im Container läuft, die mittels `add` eingefügt wurden und für die `x.approved(p)` als Ergebnis `null` liefert.
- `iterator` (definiert in `Iterable`) gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge `x` im Container läuft, die mittels `add` eingefügt wurden und für die `x.approved(p)` für jeden mittels `addCriterion` eingefügten Eintrag `p` ein Ergebnis ungleich `null` liefert.
- `criterions` ohne Parameter gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge im Container läuft, die mittels `addCriterion` eingefügt wurden.

Die Methode `remove` in jedem der Iteratoren muss so implementiert sein, dass der zuletzt von `next` zurückgegebene Eintrag aus dem Container entfernt wird.

`ApSet` implementiert `ApprovableSet`. Die Typparameter von `ApSet` und `ApprovableSet` entsprechen einander.

`AdminSet` implementiert `ApprovableSet`. Typparameter von `AdminSet` und `ApprovableSet` ähneln einander, jedoch muss `T` in `AdminSet` (anders als in `ApprovableSet`) ein Untertyp von `Admin` (mit geeigneten Typparameterersetzungen) sein. Folgende zusätzliche Methoden werden benötigt:

- `extend` (ohne Parameter und Ergebnis) führt für jedes von einem Iterator `criterions` zurückgegebene `p` und jedes von einem Iterator `iterator(p)` zurückgegebene `x` folgende Operation aus: `x.approve(p, x.approved(p).add(x));`
- `shorten` (ohne Parameter und Ergebnis) führt für jedes von einem Iterator `criterions` zurückgegebene `p` und jedes von einem Iterator `iterator(p)` zurückgegebene `x` folgende Operation aus: `x.approve(p, x.approved(p).remove(x));`

`Path` ist eine Klasse mit einem Typparameter `X`, die `Admin` (mit passenden Typparameterersetzungen, wobei `X` einfach nur von `Path` an `Admin` weitergereicht wird) sowie `java.lang.Iterable<X>` implementiert. Objekte von `Path` stellen Pfade (beispielsweise Fluchtwege) als Listen mit Elementen des Typs `X` dar. Die Methode `add` fügt den Parameter als ersten Listeneintrag hinzu, wenn der Parameter noch nicht identisch vorkommt. Die Methode `remove` entfernt den Parameter aus der Liste, falls er identisch vorkommt. Der Iterator iteriert von vorne nach hinten über die Liste.

`Space` ist eine Klasse, deren Objekte Bereiche innerhalb und außerhalb von Gebäuden darstellen. `Space` implementiert `Approvable` und hat einen Typparameter `P`, der im Wesentlichen dem gleichnamigen Typparameter von `Approvable` entspricht. Der Typparameter `T` von `Approvable` ist durch `Path<Space<P>>` ersetzt, Objekte davon entsprechen Wegen innerhalb und außerhalb von Gebäuden. Bedeutungen dieser Wege hängen von gegebenen Werten des Typs ab, für den `P` steht. Die Implementierungen von `approved` und `approve` folgen nur den Beschreibungen in `Approvable`, das heißt, eine Methode setzt Werte, die andere gibt gesetzte Werte zurück, wobei die Werte (außer beim Setzen) unverändert bleiben. Die Methode `toString` gibt eine textuelle Beschreibung des Bereichs zurück, z. B. „zweites Vorzimmer“ oder „Gemüsegarten“.

`Interior` ist ein Untertyp von `Space` (mit entsprechenden Typparametern), der einen Innenraum darstellt. Zusätzlich zu den Methoden aus den Obertypen gibt die parameterlose Methode `area` die Größe des Raums in m^2 zurück.

`Exterior` ist ein Untertyp von `Space` (mit entsprechenden Typparametern), der einen Außenbereich darstellt. Zusätzlich zu den Methoden aus den Obertypen gibt die parameterlose Methode `isPublic` einen Booleschen Wert zurück, der besagt, ob der Außenbereich öffentlich zugänglich ist.

`Counter` ist eine Klasse, deren Objekte zu Testzwecken Methodenaufrufe zählen. `Counter` implementiert `Approvable`, wobei ein Typparameter `T` von `Counter` an `Approvable` weitergereicht wird und `P` durch `Counter` (mit passendem Typ für den Typparameter) ersetzt ist. Objekte von `Counter` enthalten eine Zählvariable. Ein Aufruf von `x.approved(y)` erhöht die Zählvariable von `x` um 1000 und die von `y` um 1; zurückgegeben wird der über `approve` bzw. den Konstruktor gesetzte Wert. Aufrufe von `approve` verhalten sich nur wie in `Approvable` beschrieben. Aufrufe von `toString` geben den Inhalt der Zählvariable als Text zurück (z. B. `"10"`, nicht `"zehn"`).

`RCounter` ist eine Klasse, die `Counter` ähnelt. Jedoch hat `RCounter` keinen Typparameter. `RCounter` ist Untertyp von `Approvable`, wobei `P` durch `RCounter` und `T` durch `Path<RCounter>` ersetzt ist. Wie `Counter` wird bei jedem Aufruf von `x.approved(y)` die Zählvariable von `x` um 1000 und die von `y` um 1 erhöht; Werte von Zählvariablen werden über `toString` als Texte sichtbar.

Alle in obigen Beschreibungen genannten Typparameter müssen wenn nötig mit Schranken, die auch Wildcards enthalten können, versehen sein. Alle genannten Klassen sollen die in `Object` vordefinierte Methode `toString` so überschreiben, dass sinnvolle beschreibende Texte zurückgegeben werden. Die Methoden `equals` und `hashCode` sind dagegen nicht zu überschreiben; Gleichheit von Objekten beruht auf Objektidentität. Viele der Klassen und Interfaces sind absichtlich nur auf einem sehr abstrakten Niveau zu verstehen, um einen breiten Einsatzbereich zu ermöglichen.

Welche Aufgabe zu lösen ist

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.

Verbote beachten

Lassen Sie `AdminSet` und `ApSet` sowie `Counter` und `RCounter` (mit geeigneten Typparameterersetzungen) in einer Untertypbeziehung zueinander stehen, oder geben Sie in `Test.java` Gründe an, warum keine solche Untertypbeziehung besteht (in beide Richtungen).

Untertypbeziehungen oder Begründungen

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 in vorgegebener Reihenfolge

1. Erzeugen Sie mindestens je ein Objekt sinngemäß folgender Typen:

   ```
   ApSet<Counter<String>, Counter<String>, String>
   ApSet<Counter<Integer>, Counter<Integer>, Integer>
   ApSet<Counter<Path<String>>, Counter<Path<String>>,
                                        Path<String>>
   ApSet<RCounter, RCounter, Path<RCounter>>
   ApSet<Space<String>, String, Path<Space<String>>>
   ApSet<Interior<String>, String, Path<Space<String>>>
   ApSet<Exterior<String>, String, Path<Space<String>>>
   AdminSet<RCounter, RCounter, Path<RCounter>>
   AdminSet<Space<String>, String, Path<Space<String>>>
   AdminSet<Interior<String>, String, Path<Space<String>>>
   AdminSet<Exterior<String>, String, Path<Space<String>>>
   ```

   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 `AdminSet<Space<String>, ...>`
   `b` vom Typ `AdminSet<Interior<String>, ...>`
   `c` vom Typ `AdminSet<Exterior<String>, ...>`

   Lesen Sie über geeignete Iteratoren alle mittls `add` eingefügten Einträge aus `b` und `c` aus, rufen Sie auf ihnen je nach Typ `area()` oder `isPrivate()` auf und fügen Sie sie mittels `add` in `a` ein.

Generizität so planen, dass das geht

3. Falls zwischen `AdminSet` und `ApSet` oder `Counter` und `RCounter` eine Untertypbeziehung besteht, führen Sie Testfälle aus, die überprüfen, ob entsprechende Zusicherungen erfüllt sind.

4. Überprüfen Sie die Funktionalität mittels Löschen und (erneutem) Einfügen von Objekten und die Ausgabe abfragbarer Daten jeder Art in die Standardausgabe. Bringen Sie alle Methoden der in *Kontext* beschriebenen Typen zur Ausführung.

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.

Wie gewohnt 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. 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 harmlos aussehende Meldungen. 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. Am besten erstellen Sie in einem ersten Schritt alle Interfaces und Klassen mit entsprechenden Methoden (anfangs ohne auszuführenden Code) und vorgegebenen Typparametern, wobei Sie die nötigen Schranken auf Typparametern (noch ohne Wildcards) so festlegen, dass in `Test` Variablen der vorgegebenen Typen angelegt werden können. In einem zweiten Schritt führen Sie überall Wildcards ein, wo die Verwendungen der Typen bzw. Typparameter dies erlauben. Danach (und nach vollständiger Implementierung der Methoden) sollten die Testfälle problemlos durchführbar sein. Alternativ dazu könnten Sie den zweiten Schritt weglassen und erst dann Wildcards einführen, wenn andernfalls nicht ausführbare Testfälle Sie dazu zwingen. Allerdings könnte die alternative Vorgehensweise dazu führen, dass Sie sehr viele Variationen ausprobieren müssen.

empfohlene Vorgehensweise

Es ist nicht beschrieben, woher einige der von Methoden zurückgegebenen Daten stammen. Das müssen Sie selbst bestimmen. Setzen Sie diese Daten am besten über Konstruktoren.

Konstruktoren

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 ausgiebig 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.

besonders auf Vermeidung von Raw-Types achten

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 und Lambdas kann das Sichtbarmachen mancher Programmteile nach außen verhindert werden.

Programmstruktur wirkt sich auf Sichtbarkeit aus

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.

Untertypbeziehungen über Zusicherungen sicherstellen

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. Durch vereinfachende Annahmen lässt sich die Aufgabe nicht lösen. 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

Schwierigkeiten erkennen
Skriptum anschauen

selbst gelöst, sondern an Bibliotheken weitergereicht werden. Außerdem wird das Erstellen typischer generischer Programmstrukturen geübt.

Auch der Umgang mit Sichtbarkeit wird geübt. Am Beispiel von Iteratoren soll intuitiv klar werden, welchen Einfluss innere Klassen auf die Sichtbarkeit von Implementierungsdetails haben. Ein gezielter Einsatz innerer Klassen ist auch im Zusammenhang mit Sichtbarkeit vorteilhaft, ein falscher Umgang damit kann aber schwerwiegende Probleme verursachen.

innere Klassen verwenden

Auf das Überschreiben von `equals` und `hashCode` soll verzichtet werden, da entsprechender Code einerseits das Verbot von Casts und dynamischen Typabfragen in Frage stellen würde und andererseits die Kenntnis von Programmiertechniken im Zusammenhang mit Generizität voraussetzen würde, die erst in späteren Aufgaben geübt werden.