5. Programmieraufgabe

Objektorientierte Programmiertechniken

LVA-Nr. 185.A01
2017/2018 W
TU Wien

Themen:

Generizität, Container, Iteratoren

Ausgabe:

22. 11. 2017

Abgabe (Deadline):

29. 11. 2017, 12:00 Uhr

Abgabeverzeichnis:

`Aufgabe5`

Programmaufruf:

`java Test`

Grundlage:

Skriptum, Schwerpunkt auf 3.1 und 3.2

Kontext

Eine zentrale Rolle in Versorgungseinrichtungen spielen Verbindungen zwischen Anlagenteilen. Zur Simulation werden folgende nach Bedarf generische (abstrakte oder konkrete) Klassen oder Interfaces benötigt:

- Ein Objekt von `Connection` stellt eine Verbindung dar. Folgende parameterlose Methoden werden unterstützt:
  - `source` gibt das Objekt am Anfangspunkt der Verbindung zurück (oder `null` wenn der Anfang nicht verbunden ist).
  - `sink` gibt das Objekt am Endpunkt der Verbindung zurück (oder `null` wenn das Ende nicht verbunden ist).
- Ein Objekt von `Supply` stellt einen Anlagenteil einer Versorgungseinrichtung für Wasser, Strom und so weiter dar.
- Ein Objekt von `WaterSupply` (Untertyp von `Supply`) stellt einen Teil einer Anlage zur Wasserversorgung dar.
- Ein Objekt von `PowerSupply` (Untertyp von `Supply`) stellt einen Teil einer Anlage zur Stromversorgung dar.
- Ein Objekt von `Hose` (mit geeigneten Typparameterersetzungen Untertyp von `Connection`) stellt einen Wasserschlauch als Verbindung von zwei Objekten des Typs `WaterSupply` dar. Die Methode `diameter` gibt den Durchmesser in Zoll als Fließkommazahl zurück.
- Ein Objekt von `Cable` (mit geeigneten Typparameterersetzungen Untertyp von `Connection`) stellt ein Stromkabel als Verbindung zwischen zwei Objekten des Typs `PowerSupply` dar. Die Methode `strands` gibt eine ganze Zahl mit der Anzahl der Litzen zurück.
- Ein Objekt von `Negator` (dient nur zum Testen, ist mit geeigneten Typparameterersetzungen Untertyp von `Connection`) verbindet zwei Zahlen vom Typ `Integer`, wobei der von `sink` zurückgegebene Wert die Negation des von `source` zurückgegebenen Werts ist.
- Ein Objekt von `Ensemble` ist eine Sammlung von Objekten, deren gemeinsamer Typ über Typparameter festgelegt ist. `Ensemble` implementiert `java.lang.Iterable<...>` und folgende Methoden:
  - `add` fügt den Parameter in die Sammlung ein sofern dieses Objekt (mit derselben Identität) noch nicht vorhanden ist.
  - `iterator` gibt einen neuen Iterator über alle Objekte in der Sammlung zurück, wobei die Objekte in der Reihenfolge des Einfügens zurückgegeben werden. Der Iterator muss `remove` implementieren ohne eine `UnsupportedOperationException` zu werfen (siehe `java.lang.Iterator`).

- Ein Objekt von `Composition` (mit geeigneten Typparametererersetzungen Untertyp von `Connection`, `Supply` und `Ensemble`) ist eine Sammlung von Objekten, deren gemeinsamer Typ über Typparameter festgelegt ist und die durch Verbindungsobjekte (von einem Untertyp von `Connection`) miteinander verbunden sein können. Neben den Methoden aus Obertypen werden folgende Methoden benötigt:
  - `connect` fügt den Parameter `x` als Verbindung zwischen Objekten hinzu, wobei auch die Objekte `x.source()` und `x.sink()` hinzugefügt werden, falls sie ungleich `null` und noch nicht vorhanden sind. Fügt `connect` dieselbe Verbindung mehrfach hinzu, ist sie auch mehrfach vorhanden.
  - `connectionIter` gibt einen Iterator über alle mittels `connect` eingeführte Verbindungen zwischen Objekten zurück. Dieser Iterator implementiert `remove`. Ist eine Verbindung mehrfach vorhanden, gibt der Iterator sie auch entsprechend oft zurück.

Welche Aufgabe zu lösen ist

Implementieren Sie die oben beschriebenen Klassen und Interfaces, wobei Sie selbst nach Bedarf Typparameter wählen müssen.

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 eine Datenstruktur der fünf Typen

   ```
   Ensemble<String>
   Composition<Integer, Negator>
   Composition<WaterSupply, Hose>
   Composition<PowerSupply, Cable>
   Composition<Composition<WaterSupply, Hose>,
               Composition<PowerSupply, Cable>>
   ```

   Typparameter können im Detail anders sein

   oder sinngemäß entsprechende (bei anderer Verwendung von Typparametern). Überprüfen Sie die Funktionalität durch Einfügen einiger Objekte (wo möglich auch solche, die Verbindungen darstellen), Ausgeben der von Iteratoren zurückgegebenen Objekte, Entfernen einiger Objekte, Einfügen weiterer Objekte und erneutes Ausgeben, sodass wichtige Details getestet werden. Fügen Sie auch Objekte ein, die nicht verbunden sind. Bilden Sie Summen der Durchmesser aller vom Iterator zurückgegebenen Schläuche sowie unabhängig davon Summen der Anzahl der Litzen aller vom Iterator zurückgegebenen Kabel; dazu müssen Sie die Typen kennen.

   einfügen
   auslesen
   entfernen
   erneut einfügen

   Typen sind bekannt

2. Erzeugen Sie eine Datenstruktur vom Typ

   ```
   Composition<Supply, Connection<Supply>>
   ```

   oder einem entsprechenden. Lesen Sie über Iteratoren alle Objekte (auch Verbindungen) aus den drei in Punkt 1 erzeugten Datenstrukturen mit Objekten des Typs `Supply` aus und fügen Sie diese in die neue Datenstruktur ein. Überprüfen Sie die Funktionalität wie in Punkt 1, jedoch ohne Summenberechnungen.

   ausgelesene Obj. einfügen

3. Betrachten Sie die in Punkt 2 erzeugte Datenstruktur so, als ob sie vom Typ `Ensemble<Supply>` (oder entsprechend) wäre, z. B., indem Sie die Datenstruktur an eine Variable vom deklarierten Typ `Ensemble<Supply>` zuweisen und die Tests auf dieser Variablen ausführen. Testen Sie die Funktionalität wie in Punkt 2, allerdings ohne die zusätzliche Funktionalität von `Composition` zu verwenden.

   Ersetzbarkeitstest

4. Machen Sie optional weitere Überprüfungen, die Ihnen notwendig oder sinnvoll erscheinen.

   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

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 sind ohne Verwendung von Arrays, ohne vorgefertigte Container-Klassen (wie `LinkedList`, `HashSet`, etc.) und ohne vorgefertigte Iterator-Implementierungen zu lösen. Benötigte Container und Iteratoren sind selbst zu schreiben.

Verbote beachten

Typsicherheit soll so weit wie möglich vom Compiler garantiert werden. Auf die Verwendung von 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 vom Compiler nur eine harmlos aussehende Meldung. Überprüfungen durch den Compiler dürfen nicht ausgeschaltet werden.

Compiler-Feedback einschalten

Es ist nicht beschrieben, woher von einigen Methoden zurückgegebene Daten stammen. Setzen Sie diese Daten über Konstruktoren. Es wird empfohlen, in allen Klassen `toString` passend zu überschreiben.

Konstruktoren
`toString`

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
- Zusicherungen konsistent und zweckentsprechend — 15 Punkte
- Sichtbarkeit auf kleinstmögliche Bereiche beschränkt — 15 Punkte
- Lösung wie vorgeschrieben und sinnvoll getestet — 20 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.

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 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 daher 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 zum Einen, dass Schwierigkeiten nicht selbst gelöst sondern nur an Bibliotheken weitergeleitet werden, gibt zum Anderen aber auch Gelegenheit, das Erstellen typischerweise generischer Programmstrukturen zu üben.

Schwierigkeiten erkennen
Skriptum anschauen

Daneben wird 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 auf die Sichtbarkeit von Implementierungsdetails nach außen hat.

innere Klassen verwenden