5. Programmieraufgabe

Objektorientierte Programmiertechniken

LVA-Nr. 185.A01
2022/2023 W
TU Wien

Themen:

Generizität, Container, Iteratoren

Ausgabe:

14. 11. 2022

Abgabe (Deadline):

21. 11. 2022, 14:00 Uhr

Abgabeverzeichnis:

`Aufgabe5`

Programmaufruf:

`java Test`

Grundlage:

Skriptum, Schwerpunkt auf 3.1 und 3.2

Kontext

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

`Compatible` ist ein (eventuell generisches) Interface mit zwei Methoden:

- `compatible` hat einen Parameter und gibt einen Wahrheitswert zurück. Das Ergebnis von `x.compatible(y)` besagt, ob `x` auf eine nicht näher bestimmte Weise kompatibel mit `y` ist, wobei das Ergebnis nur von unveränderlichen Werten in `x` und `y` abhängt. Eine Methodenausführung lässt `y` unverändert, eine Änderung von `x` ist erlaubt (sofern sich die Änderung nicht auf Ergebnisse künftiger Aufrufe von `compatible` auswirkt).
- `rate()` gibt den Anteil der bisherigen Aufrufe von `compatible` in diesem Objekt zurück, für den das Ergebnis `true` war.

`CompatibilityCollection` ist ein Interface mit zwei Typparametern `X` und `Y` (wenn nötig mit Schranken bzw. Wildcards) und dem Obertyp `java.lang.Iterable<X>`. Ein Objekt davon ist ein Container mit Einträgen der Typen `X` und `Y`, wobei `X` Untertyp von `Compatible` sein muss, sodass für jeden Eintrag `x` vom Typ `X` und jeden Eintrag `y` vom Typ `Y` die Methode `x.compatible(y)` aufrufbar ist. Folgende Methoden werden benötigt:

- `addX` 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 `addX` eingefügt wurde, wird es nicht noch einmal eingefügt.
- `addY` mit einem Argument vom Typ `Y` ist genau gleich wie `addX` definiert, abgesehen davon, dass es um Einträge vom Typ `Y` geht und keine identischen Objekte, die mittles `addY` 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 `addX` eingefügt wurden. Die Methode `remove` im Iterator muss so implementiert sein, dass der zuletzt von `next` zurückgegebene Eintrag aus dem Container entfernt wird.
- `iterator` mit einem Parameter `x` vom Typ `X` gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge `y` im Container läuft, die mittels `addY` eingefügt wurden und für die `x.compatible(y)` gilt. Die Methode `remove` im Iterator muss so implementiert sein, dass der zuletzt von `next` zurückgegebene Eintrag aus dem Container entfernt wird.

`GeneralCatalog` mit einem Typparameter `X` (wenn nötig mit Schranke bzw. Wildcards) implementiert `CompatibilityCollection<X,X>`. Die Methode `generalIterator` gibt einen Iterator zurück, der in beliebiger Reihenfolge über alle Einträge iteriert, egal ob die Einträge über `addX` oder `addY` eingefügt wurden. Über mehrere Vorkommen identischer Objekte wird nur ein ein Mal iteriert.

`SpecialCatalog` mit zwei Typparametern `X` und `Y` (ev. mit Schranken bzw. Wildcards) implementiert `CompatibilityCollection<X,Y>`.

`PVComponent` ist ein Interface, das `Compatible` erweitert. Ein Objekt dieses Typs stellt eine Komponente einer PV-Anlage dar. Die Art der Komponente wird durch die von `toString` zurückgegebene Zeichenkette beschrieben. Wie alle elektrischen Anlagenteile ist die Komponente nur für bestimmte Stromnetze ausgelegt, die über Zeichenketten beschrieben sind. Z. B. steht `"DC decoupled"` für ein Gleichspannungsnetz, das von anderen Netzen elektrisch entkoppelt ist, `"DC european"` für ein Gleichspannungsnetz, das auf dem Spannungsniveau des europäischen Wechselstromnetzes liegen darf und `"AC european"` für das europäische Wechselstromnetz. Die Methode `certified` gibt eine Liste von Zeichenketten zurück, in der jeder Eintrag ein kompatibles Stromnetz beschreibt. Zwei Komponenten sind genau dann kompatibel zueinander, wenn die Ergebnisse von `certified` zumindest eine Zeichenkette enthalten, die für beide Komponenten gleich ist.

`Inverter` implementiert `PVComponent`. Da es sich dabei um einen Wechselrichter handelt, enthält das Ergebnis von `certified` mindestens zwei Zeichenketten, eine für ein Gleichspannungsnetz und eine für ein Wechselspannungsnetz. Die Methode `kWp` gibt die Maximalleistung in kW zurück. Es gibt *keine* Methode `size`.

`Module` implementiert `PVComponent`. Jedes Objekt dieses Typs ist ein PV-Paneel. Die Methode `size` gibt die Paneelgröße in m^2 zurück. Es gibt *keine* Methode `kWp`.

`Multiple` implementiert `Compatible`. Ein Objekt davon enthält eine unveränderliche positive ganze Zahl. Ein Aufruf von `x.compatible(y)` gibt genau dann `true` zurück, wenn die Zahl in `x` durch die Zahl in `y` ohne Rest teilbar ist. Es besteht *keine* Untertypbeziehung zu `PVComponent`, `Inverter` oder `Module`.

Ein Objekt vom Typ `SpecialCatalog<PVComponent,PVComponent>` oder `GeneralCatalog<PVComponent>` steht für einen Produktkatalog, der es ermöglicht, über Iteratoren gezielt nach Produkten aus dem Bereich von PV-Komponenten zu suchen, die mit einem bestimmten Produkt kompatibel sind. Ein Objekt von `SpecialCatalog<Module,Inverter>` ermöglicht die gezielte Suche nach Wechselrichtern, die mit einem bestimmten PV-Paneel kompatibel sind. Im Gegensatz dazu haben Objekte von `SpecialCatalog<Multiple,Multiple>` und `GeneralCatalog<Multiple>` den Zweck, die Klassen auf einfache Weise testen zu können.

Bitte beachten Sie, dass die oben genannten Typen nicht immer vollständig ausgeschrieben sind. Z. B. könnte es nötig sein, statt `Compatible` eine generische Variante `Compatible<...>` zu verwenden.

Welche Aufgabe zu lösen ist

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

Lassen Sie `GeneralCatalog<X>` und `SpecialCatalog<X,X>` in einer Untertypbeziehung zueinander stehen (für geeignete Typen `X`), oder geben Sie Gründe an, warum keine solche Untertypbeziehung 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:

   ```
   SpecialCatalog<Multiple, Multiple>
   SpecialCatalog<PVComponent, PVComponent>
   SpecialCatalog<Inverter, PVComponent>
   SpecialCatalog<Module, PVComponent>
   SpecialCatalog<PVComponent, Inverter>
   SpecialCatalog<Inverter, Inverter>
   SpecialCatalog<Module, Inverter>
   SpecialCatalog<PVComponent, Module>
   SpecialCatalog<Inverter, Module>
   SpecialCatalog<Module, Module>
   GeneralCatalog<Multiple>
   GeneralCatalog<PVComponent>
   GeneralCatalog<Inverter>
   GeneralCatalog<Module>
   ```

   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 `SpecialCatalog<PVComponent, ...>`
   `b` vom Typ `SpecialCatalog<..., PVComponent>`
   `c` vom Typ `SpecialCatalog<Module, Inverter>`

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

   Generizität so planen, dass das geht

3. Falls `GeneralCatalog<X>` und `SpecialCatalog<X,X>` 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.

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.

Daneben 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 (wie z. B. 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 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 und sinnvoll 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 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

Außerdem wird 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