IFM 2.1 Programmiermethoden, S20

B03 'Bundesliga': Generics, JUnit, Merge-Request

Carsten Gips, FH Bielefeld

Praktikum: 12.06.2020

ILIAS: 11.06.2020

1 Aufgabe Praktikum

1.1 Generische Klassen und Interfaces

In dieser Aufgabe modellieren Sie in Java verschiedene Spielertypen sowie generische Mannschaften und Ligen, die jeweils bestimmte Spieler bzw. Mannschaften aufnehmen können.

a) Implementieren Sie die Klasse `bundesliga.generic2.Spieler`, die das Interface `bundesliga.generic2.ISpieler` aus den Vorgaben erfüllt.

b) Implementieren Sie die beiden Klassen `bundesliga.generic2.{FussballSpieler,BasketballSpieler}` und sorgen Sie dafür, dass beide Klassen vom Compiler als `Spieler` betrachtet werden (Vererbungshierarchie).

   Der `FussballSpieler` kann mit der Methode `public void schiessTor()` ein Tor schießen und erzielt damit einen Punkt.

   Der `BasketballSpieler` kann mit der Methode `public void wirfKorb()` einen Korb werfen und erzielt damit zwei Punkte.

c) Betrachten Sie das nicht-generische Interface `bundesliga.polymorph.IMannschaft` in den Vorgaben. Erstellen Sie daraus das generische Interface `bundesliga.generic2.IMannschaft` mit einer Typvariablen. Stellen Sie durch geeignete Beschränkung der Typvariablen sicher, dass nur Mannschaften mit von `bundesliga.generic2.ISpieler` abgeleiteten Spielern gebildet werden können.

d) Betrachten Sie das nicht-generische Interface `bundesliga.polymorph.ILiga` in den Vorgaben. Erstellen Sie daraus das generische Interface `bundesliga.generic2.ILiga` mit einer Typvariablen. Stellen Sie durch geeignete Beschränkung der Typvariablen sicher, dass nur Ligen mit von `bundesliga.generic2.IMannschaft` abgeleiteten Mannschaften angelegt werden können.

e) Leiten Sie von `bundesliga.generic2.ILiga` das generische Interface `bundesliga.generic2.IBundesLiga` ab. Stellen Sie durch geeignete Formulierung der Typvariablen sicher, dass nur Ligen mit Mannschaften angelegt werden können, deren Spieler vom Typ `bundesliga.generic2.FussballSpieler` (oder abgeleitet) sind.

   Realisieren Sie die Funktionalität von `bundesliga.generic2.IBundesLiga` als **nicht-generisches** Interface `bundesliga.generic2.IBundesLiga2`.

Hinweis: Betrachten Sie die JUnit4-Tests in den Vorgaben wieder als (zusätzliche) ausführbare Spezifikation.

Thema: Sicherer Umgang mit generischen Klassen und Methoden

1.2 JUnit

Der RSV Flotte Speiche hat in seiner Mitgliederverwaltung (`rsvflottespeiche.MitgliederVerwaltung`) die Methode `beitritt` implementiert. Diese dient dazu, neue Mitglieder im Radsportverein aufzunehmen.

Details zum Verhalten der Methode entnehmen Sie bitte dem Javadoc-Kommentar an der vorgegebenen Methode.

1. Führen Sie eine Äquivalenzklassenanalyse durch und geben Sie die gefundenen Äquivalenzklassen ($\ddot{A}K$) an: laufende Nummer, Definition (Wertebereiche o.ä.), kurze Beschreibung (gültige/ungültige ÄK, Bedeutung).
2. Führen Sie zusätzlich eine Grenzwertanalyse durch und geben Sie die jeweiligen Grenzwerte (GW) an.
3. Erstellen Sie aus den ÄK und GW wie in der Vorlesung diskutiert Testfälle. Geben Sie pro Testfall (TF) an, welche ÄK und/oder GW abgedeckt sind, welche Eingaben Sie vorsehen und welche Ausgabe Sie erwarten.

   Hinweis: Erstellen Sie separate (zusätzliche) TF für die GW, d.h. integrieren Sie diese nicht in die ÄK-TF.

4. Implementieren Sie die Testfälle in JUnit (JUnit 4 oder 5). Fassen Sie die Testfälle der gültigen ÄK in einem parametrisierten Test zusammen. Für die ungültigen ÄKs erstellen Sie jeweils eine eigene JUnit-Testmethode. Beachten Sie, dass Sie auch die Exceptions testen müssen.

Hinweis: Geben Sie die Lösung zu den Punkten (1) bis (3) als Javadoc-Kommentar Ihrer in Punkt (4) implementierten JUnit-Klasse(n) ab.

Thema: Test mit JUnit

1.3 Merge-Request

Erstellen Sie für Ihre Implementierung aus der vorigen Aufgabe ("JUnit") einen Merge-Request gegen den `master`-Branch aus dem Vorgabe-Repo.

2 Tutorium

Lösen Sie die folgenden Aufgaben selbstständig **vor** dem Tutorium und diskutieren Sie Ihre Lösung und ggf. aufgetauchte Probleme und Fragen im Tutorium.

Hinweis: Beachten Sie dabei auch den Zeitplan der Themen in der Vorlesung: Sie werden entsprechend nicht alle Fragen gleich zum ersten Tutorium lösen können!

2.1 Fehlersuche mit Git Bisect

Erstellen Sie sich ein Git-Repo mit einigen Commits. Nutzen Sie dabei auch Branches und mergen Sie diese. Bauen Sie irgendwo einen Fehler ein und demonstrieren Sie, wie Sie mit Git Bisect den fehlerhaften Commit finden können. Schreiben Sie sich dazu einen passenden Testfall, den Sie in jedem Schritt zur Prüfung laufen lassen.

Hinweis: Wenn Ihr Team-Repo zur Bearbeitung der obigen Aufgabe (oder vom letzten Blatt) bereits die gestellten Bedingungen erfüllt, können Sie natürlich auch dieses zur Demo nutzen.

Thema: Selbstständige Literaturrecherche, Praktischer Umgang mit Git Bisect

3 Generische verkettete Listen

a. Sie finden in den Vorgaben ein Interface für eine (stark vereinfachte) Listen-API (`linkedlist.ISimpleList`).

   Implementieren Sie eine generische Klasse `linkedlist.SimpleLinkedList`, die dieses Interface als **einfach verkettete Liste** implementiert.

   Hinweis: Sie sollen den Listen-Datentyp und ggf. damit zusammenhängende Hilfsklassen etc. selbst realisieren! Verwenden Sie also **nicht** intern einfach Datenstrukturen aus dem JDK o.ä.!

b. Implementieren Sie das Interface `java.lang.Iterable` für Ihre Klasse `linkedlist.SimpleLinkedList`, um mit einer `foreach`-Schleife elementweise über alle Elemente einer `SimpleLinkedList` iterieren zu können:

```
SimpleLinkedList<Integer> speicher = ...;
for (Integer i : speicher) {
    summe += i;
}
```

c. Machen Sie Ihre Liste sortierbar: Implementieren Sie das Interface `linkedlist.ISortable` für Ihre Klasse `linkedlist.SimpleLinkedList`.[1]

   Um die Liste sortieren zu können, muss der Typparameter vergleichbar sein, d.h. das Interface `java.lang.Comparable` implementieren. Bilden Sie den Vergleich zwischen zwei Listenelementen auf den Vergleich zwischen den gespeicherten Daten ab!

[1] Normalerweise würde man das Interface `java.util.List` implementieren und könnte dann die selbst implementierten verketteten Listen auch durch `java.util.Collections#sort()` sortieren lassen. Dann müssten Sie aber noch etliche weitere Listenmethoden implementieren. Diese Mehrarbeit wollten wir Ihnen ersparen :-)

Definieren Sie zwei verschiedene Comperatoren, so dass Sie eine Liste auf- bzw. absteigend sortieren können.

Hinweis: Vergessen Sie nicht, neben `compareTo` auch `equals` und `hashCode` passend zu implementieren!

d. Ein Stack kann intern durch eine Liste realisiert werden. Implementieren Sie dazu das Interface `linkedlist.IStack` für Ihre Klasse `linkedlist.SimpleLinkedList`.

Hinweis: Betrachten Sie die JUnit4-Tests in den Vorgaben wieder als (zusätzliche) ausführbare Spezifikation.

Hinweis: Sie können die Vorgabe-Tests auch einzeln ausführen, um die Implementierung schrittweise zu erweitern. Beachten Sie bitte dabei, dass jeweils die Funktionalitäten der verketteten Liste (`linkedlist.ISimpleList`) benötigt werden. D.h. um beispielsweise die Tests zur Sortierbarkeit (`linkedlist.SortTest`) ausführen zu können, muss die Klasse `linkedlist.SimpleLinkedList` bereits das Interface `linkedlist.ISimpleList` implementieren ...

Thema: Komplexere generische Datenstrukturen selbst implementieren