4. Programmieraufgabe

Objektorientierte Programmiertechniken

LVA-Nr. 185.A01
2020/2021 W
TU Wien

Themen:

Untertypbeziehungen, Zusicherungen

Ausgabe:

04. 11. 2020

Abgabe (Deadline):

11. 11. 2020, 12:00 Uhr

Abgabeverzeichnis:

`Aufgabe4`
nicht mehr `Aufgabe1-3`

Programmaufruf:

`java Test`

Grundlage:

Skriptum, Schwerpunkt auf Kapitel 2

Kontext

Für jeden der folgenden Begriffe (in alphabetischer Reihenfolge) soll es einen Typ im zu erstellenden Programm geben:

`Cereal:` Eine Getreidepflanze, also eine Pflanze aus der Familie der Gräser mit unscheinbaren Blüten, die zur Produktion von Kornfrucht gezüchtet wurde und meist auf Ackerflächen kultiviert wird. Sie verträgt kaum Rückschnitte und gedeiht daher nur in Ausnahmefällen auf Grünland, konkret in stark überdüngten, selten gemähten Wiesen. Getreide entzieht dem Boden Stickstoff; ein Aufruf von `nitrogen` (siehe `Plant`) gibt eine negative Zahl zurück. Die Bestäubung erfolgt durch Wind; ein Aufruf der parameterlosen Methode `pollinate` simuliert eine erfolgreiche Windbestäubung, wodurch sich die von `seed` (siehe `Plant`) zurückgegebene Zahl erhöht.

`Flower:` Eine Pflanze mit auffälligen Blüten. Ansprüche und Standorte sind sehr vielfältig und hängen stark von der Art ab; einige Arten gedeihen auf Wiesen, andere nicht. Die Bestäubung erfolgt durch tierische Bestäuber, nicht durch Wind; ein Aufruf der Methode `pollinate` mit dem jeweiligen Bestäuber als Argument simuliert eine erfolgreiche Bestäubung, wodurch sich die von `seed` zurückgegebene Zahl erhöht.

`Grass:` Eine einkeimblättrige Pflanze mit unscheinbaren Blüten und langen, schmalen Blättern aus der Familie der Gräser, die auch häufige Rückschnitte gut verträgt und typischerweise auf Grünland (Wiese und Rasen) wächst. Wie bei Getreide erfolgt die Bestäubung durch Wind; ein Aufruf der parameterlosen Methode `pollinate` simuliert eine erfolgreiche Windbestäubung, wodurch sich die von `seed` zurückgegebene Zahl erhöht.

`HoneyBee:` Eine Honigbiene. Honigbienen sind wichtige tierische Bestäuber vieler Blütenpflanzen.

`MeadowFlower:` Eine typischerweise auf Wiesen wachsende Pflanze mit auffälligen Blüten, die auch gelegentliche Rückschnitte verträgt.

`MeadowPlant:` Eine typischerweise auf Wiesen wachsende Pflanze.

`Plant:` Eine Pflanze. Die Methode `nitrogen` gibt eine Zahl zurück, die besagt, wie viel Stickstoff die Pflanze durch ihr Wachsen und später als Kompost dem Boden in der Summe entzieht (negative Zahl) oder zuführt (positive Zahl). Die Methode `seed` gibt eine ganze Zahl zurück, die besagt, wie viele durch Bestäubung bereits befruchtete Samen in der Pflanze gerade heranreifen.

`Pollinator:` Ein tierischer Bestäuber, häufig ein Insekt. Aber auch Vögel oder Fledermäuse können zu Bestäubern werden.

`RedClover:` Rotklee aus der Familie der Hülsenfrüchtler wächst auch auf Wiesen, vertägt Rückschnitte und zieht mit seinen auffällig roten Blüten mehrmals im Jahr vorwiegend Honigbienen als Bestäuber an. Da Klee ein Stickstoffsammler ist, gibt `nitrogen` eine positive Zahl zurück. Ein Aufruf der Methode `pollinate` mit dem Bestäuber als Argument simuliert eine erfolgreiche Bestäubung und erhöht die von `seed` zurückgegebene Zahl, wobei sich diese Zahl stärker erhöht wenn das Argument eine Honigbiene ist.

Es ist davon auszugehen, dass eine Pflanze oder ein Tier nicht gleichzeitig mehreren Familien angehören kann und gegensätzliche Eigenschaften (wie „verträgt Rückschnitte gut“ und „verträgt Rückschnitte kaum“ sowie „Blüten auffällig“ und „Blüten unscheinbar“) einander ausschließen.

Die Anzahl der obigen Beschreibungen von Methoden ist klein gehalten. Zur Realisierung von Untertypbeziehungen können zusätzliche Methoden in den einzelnen Typen nötig sein, weil Methoden von Obertypen auf Untertypen übertragen werden.

Welche Aufgabe zu lösen ist

Schreiben Sie ein Java-Programm, das für jeden unter *Kontext* angeführten Typ eine (abstrakte) Klasse oder ein Interface bereitstellt. Versehen Sie alle Typen mit den notwendigen Zusicherungen und stellen Sie sicher, dass Sie nur dort eine Vererbungsbeziehung (`extends` oder `implements`) verwenden, wo eine Untertypbeziehung besteht. Ermöglichen Sie Untertypbeziehungen zwischen *allen* diesen Typen, außer wenn sie den Beschreibungen der Typen (siehe *Kontext*) widersprechen würden.

alle Zusicherungen und Untertypbeziehungen

Besteht zwischen zwei Typen keine Untertypbeziehung, geben Sie in einem Kommentar in `Test.java` eine Begründung dafür an. Bitte geben Sie eine textuelle Begründung, auskommentierte Programmteile reichen nicht. Das Fehlen einer Methode in einer Typbeschreibung ist als Begründung ungeeignet, weil zusätzliche Methoden hinzugefügt werden dürfen. Sie brauchen keine Begründung dafür angeben, dass A kein Untertyp von B ist, wenn B ein Untertyp von A ist.

Begründung wenn keine Untertypbeziehung

Alle oben genannten Typen müssen mit vorgegebenen Namen vorkommen, auch solche, die Sie für unnötig erachten. Vermeiden Sie wenn möglich zusätzliche abstrakte Klassen und Interfaces. Vorgegebene Methoden sind semantisch sehr einfach und sollen auch so einfach implementiert sein (z. B. einen im Konstruktor gesetzten, konstant vorgegebenen oder auf einfache Weise berechneten Wert zurückgeben, oder einen kurzen Text für Testzwecke ausgeben).

gegebene Typen mit gegebenen Namen

Schreiben Sie eine Klasse `Test` zum Testen Ihrer Lösung. Das Programm muss vom Abgabeverzeichnis (`Aufgabe4`) aus durch `java Test` ausführbar sein. Überprüfen Sie mittels Testfällen, ob dort, wo Sie eine Untertypbeziehung annehmen, Ersetzbarkeit gegeben ist.

Testklasse

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

Was im Hinblick auf die Beurteilung wichtig ist

Die insgesamt 100 für diese Aufgabe erreichbaren Punkte sind folgendermaßen auf die zu erreichenden Ziele aufgeteilt:

- Untertypbeziehungen richtig erkannt und eingesetzt — 40 Punkte
- nicht bestehende Untertypbeziehungen gut begründet — 15 Punkte
- Zusicherungen passend und zweckentsprechend — 20 Punkte
- Lösung entsprechend Aufgabenstellung getestet — 15 Punkte
- Lösung vollständig (entsprechend Aufgabenstellung) — 10 Punkte

Schwerpunkte berücksichtigen

Die 15 Punkte für das Testen sind nur erreichbar, wenn die Testfälle mögliche Widersprüche in den Untertypbeziehungen aufdecken könnten. Abfragen mittels `instanceof` sowie Casts sind dafür ungeeignet.

Die 10 Punkte für eine „vollständige Lösung“ gelten für jene Bereiche, für die keine speziellen Abzüge vorgesehen sind. Das können z. B. kleine Logikfehler, fehlende kleine Programmteile, unzureichende Beschreibungen der Aufgabenaufteilung, oder Probleme mit Data-Hiding bzw. beim Compilieren sein. Fehlen wichtige Programmteile oder treten schwere Fehler auf, werden auch Untertypbeziehungen, Zusicherungen und das Testen betroffen sein, was zu deutlich größerem Punkteverlust führt.

Die größte Schwierigkeit dieser Aufgabe liegt darin, *alle* Untertypbeziehungen zu finden und Ersetzbarkeit sicherzustellen. Vererbungsbeziehungen, die keine Untertypbeziehungen sind, führen zu sehr hohem Punkteverlust. Hohe Punkteabzüge gibt es auch für nicht wahrgenommene Gelegenheiten, Untertypbeziehungen zwischen vorgegebenen Typen herzustellen, sowie für fehlende oder falsche Begründungen für nicht bestehende Untertypbeziehungen. Geeignet wären z. B. Gegenbeispiele.

alle Untertypbeziehungen

Eine Grundlage für das Auffinden der Untertypbeziehungen sind gute Zusicherungen. Wesentliche Zusicherungen kommen in obigen Beschreibungen vor. Allerdings ist nicht jeder Teil einer Beschreibung als Zusicherung von Bedeutung. Untertypbeziehungen ergeben sich aus erlaubten Beziehungen zwischen Zusicherungen in Unter- und Obertypen. Es ist günstig, alle Zusicherungen, die im Obertyp gelten, auch im Untertyp (noch einmal) hinzuschreiben, weil dadurch so mancher Widerspruch deutlich sichtbar und damit eine falsche Typstruktur mit höherer Wahrscheinlichkeit vermieden wird. Nicht die Quantität der Kommentare ist entscheidend, sondern die Qualität (Verständlichkeit, Vollständigkeit, Aussagekraft, ...). Zusicherungen in `Test.java` werden bei der Beurteilung aus praktischen Gründen nicht berücksichtigt.

Auch beim Testen kommt es auf Qualität, nicht Quantität an. Testfälle sollen grundsätzlich in der Lage sein, Verletzungen der Ersetzbarkeit aufzudecken, die nicht ohnehin vom Compiler erkannt werden.

Zur Lösung dieser Aufgabe müssen Sie Untertypbeziehungen und den Einfluss von Zusicherungen genau verstehen. Lesen Sie Kapitel 2 des Skriptums. Folgende zusätzlichen Informationen könnten hilfreich sein:

- Konstruktoren werden in einer konkreten Klasse aufgerufen und sind daher vom Ersetzbarkeitsprinzip nicht betroffen.

- Zur Lösung der Aufgabe sind keine Exceptions nötig. Generell darf ein Objekt eines Untertyps nur Exceptions werfen, die man auch von Objekten des Obertyps in dieser Situation erwarten würde.
- Mehrfachvererbung gibt es nur auf Interfaces. Sollte ein Typ mehrere Obertypen haben, müssen diese (bis auf einen) Interfaces sein.

Lassen Sie sich von der Form der Beschreibung nicht täuschen. Aus einem Verweis auf einen anderen Typ folgt noch keine Ersetzbarkeit. Sie sind auf dem falschen Weg, wenn es den Anschein hat, A könne Untertyp von B und B Untertyp von A sein, außer wenn A und B gleich sind.

Achten Sie auf die Sichtbarkeit. Alle beschriebenen Typen und Methoden sollen überall verwendbar sein, sonst nichts. Sichtbare Implementierungsdetails beeinflussen die Ersetzbarkeit.

Sichtbarkeit

Was man generell beachten sollte (alle Aufgaben)

Es werden keine Ausnahmen bezüglich des Abgabetermins gemacht. Beurteilt wird, was im Abgabeverzeichnis `Aufgabe4` im Repository steht. Alle `.java`-Dateien im Abgabeverzeichnis (einschließlich Unterverzeichnissen) müssen auf der `g0` gemeinsam übersetzbar sein. Auf der `g0` sind nur Standardbibliotheken installiert; es dürfen keine anderen Bibliotheken verwendet werden. Viele Übersetzungsfehler kommen von falschen `package`- oder `import`-Anweisungen und unabsichtlich abgegebenen `.java`-Dateien.

Abgabetermin einhalten

auf g0 testen

Schreiben Sie nur eine Klasse in jede Datei (außer geschachtelte Klassen), halten Sie sich an übliche Namenskonventionen, und verwenden Sie die Namen, die in der Aufgabenstellung vorgegeben sind. Andernfalls könnte es zu Fehlinterpretationen Ihrer Absichten kommen, die sich negativ auf die Beurteilung auswirken.

eine Klasse pro Datei

vorgegebene Namen

Warum die Aufgabe diese Form hat

Die Beschreibungen der Typen bieten wenig Interpretationsspielraum bezüglich Ersetzbarkeit. Die Aufgabe ist so formuliert, dass Untertypbeziehungen eindeutig sind. Über Testfälle und Gegenbeispiele sollten Sie schwere Fehler selbst finden können.

eindeutig