4. Programmieraufgabe

Programmierparadigmen
LVA-Nr. 194.023
2023/2024 W
TU Wien

Themen:
Untertypbeziehungen, Zusicherungen

Ausgabe:
06. 11. 2023

Abgabe (Deadline):
20. 11. 2023, 14:00 Uhr

Abgabeverzeichnis:
`Aufgabe4`
nicht mehr `Aufgabe1-3`

Programmaufruf:
`java Test`

Grundlage:
Skriptum, Kapitel 3

Kontext

Ameisen lassen sich vergleichsweise einfach in *Formicarien* halten, das sind speziell für Ameisen vorgesehene Terrarien. Ein Formicarium besteht meist (nicht immer) aus einer *Arena*, das ist ein Behälter, in dem Ameisen sich z. B. zur Futtersuche aufhalten können, sowie einem *Nest*. Das Nest kann sich in der Arena befinden oder außerhalb liegen und etwa über Röhren mit der Arena verbunden sein. Für kurze Zeit können Ameisen auch in einem Nest ohne Arena leben, aber nicht auf Dauer. Daneben braucht es noch einen Ausbruchsschutz, eine Futterquelle, Einrichtungen zur Erzeugung und Überwachung geeigneter klimatischer Bedingungen (Feuchtigkeit, Wärme), Hilfsmittel zur Beobachtung der Ameisen und so weiter. Je nach Ameisenart und gewünschten Beobachtungsmöglichkeiten ergibt sich eine Vielzahl an Varianten.

Die Aufgabe bezieht sich auf ein geplantes System zum Design von Formicarien. Vorerst soll ein Programm entstehen, das zentrale Klassen bzw. Interfaces unabhängig vom Gesamtsystem auf Konsistenz prüft und testet. Bilden Sie zu diesem Zweck jeden der folgenden Begriffe, beschrieben in alphabetischer Reihenfolge, durch je einen Typ im Programm ab:

`AntFarm:` Eine Form eines Ameisennests bestehend aus einem Substrat (z. B. Sand, Kies oder Erde) zwischen zwei Glas- oder Kunststoffplatten, die an den Rändern miteinander verbunden sind. Durch die Platten sind die von den Ameisen ins Substrat gegrabenen Gänge beobachtbar. Je nach Substrat sind solche Ameisenfarmen für Ameisen unterschiedlicher Größen und mit unterschiedlichen natürlichen Lebensräumen geeignet. Auch der Plattenabstand spielt für die Ameisengröße eine Rolle, da Ameisen ohne ausreichend Platz keine Gänge anlegen können und sich bei zu viel Platz in der Mitte verstecken (von außen nicht beobachtbar sind).

`Arena:` Ein mit einem Substrat gefüllter Glas- oder Kunststoffbehälter, in dem sich Ameisen frei bewegen können. Eine Arena alleine (ohne Nest) bietet Ameisenkolonien keine ausreichende Grundlage für ein längeres Überleben. Es wird streng zwischen Nestern und Arenen unterschieden, das heißt, eine Arena ist kein Nest, ein Nest keine Arena und eine Arena alleine kein Formicarium. Je nach Substrat sind Arenen für unterschiedliche Ameisenarten geeignet. Natürlich brauchen größere Ameisen und größere Ameisenkolonien mehr Platz als kleine. Lange Verbindungsgänge zwischen der Arena und dem Nest lassen vor allem größeren Ameisenarten die Umwelt größer erscheinen, als sie tatsächlich ist.

`Compatibility:` Objekte dieses Typs dienen zur Beschreibung des Lebensraums von Ameisen, für die der damit verbundene Gegenstand (Formicarium, ein Bestandteil davon, Messgerät, Werkzeug, . . .) geeignet ist. Die Beschreibung erfolgt über Wertebereiche verschiede-

ner Umweltparameter, abfragbar über Methodenpaare (jeweils für den minimalen und maximalen Wert):

- `minSize`, `maxSize`: geeignete Größe der Ameisen in mm;
- `minTemperature`, `maxTemperature`: Temperatur in °C;
- `minHumidity`, `maxHumidity`: relative Luftfeuchtigkeit in %.

Der Gegenstand ist zur Haltung von Ameisen geeignet, die eine passende Größe haben und unter den gegebenen Umweltbedingungen leben können. Zusätzlich geben die Methoden `time` und `maxTime` je einen zeitlichen Horizont für die längstmögliche Dauer der Ameisenhaltung mit diesem Gegenstand zurück – eine Stunde, einen Tag, eine Woche, ein Monat, ein Jahr oder unbeschränkt. Gründe für zeitliche Einschränkungen sind vielfältig, etwa die Abgabe von Giftstoffen oder das Fehlen von Ressourcen. Manche Gründe sind behebbar, andere nicht. Der Wert von `maxTime` bezieht sich auf nicht behebbare Gründe, der von `time` auf den aktuellen Zustand. Durch `setTime` kann der Wert von `time` angepasst werden, wobei jedoch kein Wert gesetzt werden kann, der über `maxTime` hinausgeht. Die Methode `compatible` mit einem Argument von `Compatibility` gibt ein Objekt von `Compatibility` zurück, das für `time` und `maxTime` die jeweils kleineren Werte aus `this` und dem Argument übernimmt und für Umweltparameter die überschneidenden Wertebereiche annimmt; überschneiden sich Wertebereiche für einen Umweltparameter nicht, wird eine Ausnahme ausgelöst. Ein Objekt von `Compatibility` stellt eine Ansammlung von Eigenschaften eines physischen Objekts der realen Welt dar, aber niemals das physische Objekt der realen Welt selbst. Damit ist ein Objekt von `Compatibility` nie gleich einem Objekt, das ein physisches Objekt der realen Welt darstellt.

`CompositeFormicarium:` Ein Formicarium, das durch Hinzufügen und Wegnehmen von Bestandteilen änderbar ist. Über einen Iterator sind alle Bestandteile zugreifbar; `remove` im Iterator entfernt den zuletzt von `next` zurückgegebenen Bestandteil wenn mehr als ein Bestandteil vorhanden ist, sonst wird eine Ausnahme ausgelöst. Die Methode `add` in einem Objekt von `CompositeFormicarium` fügt einen Bestandteil hinzu, sofern er mit dem Formicarium kompatibel ist und nicht schon mit derselben Identität vorkommt (nicht-idente gleiche Bestandteile können mehrfach vorkommen). Kompatibilität ist gegeben, wenn `compatible` aus `Compatibility` angewandt auf die Umweltbeschreibungen des Formicariums und neuen Bestandteils keine Ausnahme auslöst. Von `compatibility` zurückgegebene Werte (siehe `Formicarium`) sind anzupassen. Änderungen durch `add` und `remove` lassen die Identität des Formicariums unverändert.

`Forceps:` Die Pinzette ist ein wichtiges Instrument, das im Zusammenhang von Formicarien häufig verwendet wird (etwa zur genaueren Untersuchung einzelner Ameisen), aber kein Formicarium oder Bestandteil eines Formicariums ist. Die Methode `compatibility` gibt

ein Objekt vom Typ `Compatibility` zurück, das (neben irrelevanten anderen Bedingungen) den Größenbereich von Ameisen angibt, für den die Pinzette gut einsetzbar ist.

`Formicarium:` Ein zur Haltung von Ameisen vorgesehenes Terrarium mit allen dazu gehörenden Bestandteilen. Formicarien können änderbar (vor allem erweiterbar) sein, aber nicht jedes Formicarium ist änderbar. Die Methode `iterator` gibt einen neuen Iterator zurück, der über alle Bestandteile des Formicariums iteriert (falls das Formicarium aus Bestandteilen zusammengesetzt ist; mehrfach vorkommende gleiche, aber nicht identische Bestandteile werden entsprechend oft, jeweils mit unterschiedlicher Identität retourniert) oder nur einmalig `this` zurückgibt (falls das Formicarium nicht weiter in Bestandteile zerlegbar ist). Die Methode `compatibility` gibt ein Objekt vom Typ `Compatibility` zurück, das die vom Formicarium gebotenen Umweltbedingungen beschreibt.

`FormicariumItem:` Ein Formicarium oder ein Bestandteil eines Formicariums oder ein Messgerät bzw. Werkzeug, das zusammen mit Formicarien verwendbar ist, unabhängig davon, ob es als Teil eines Formicariums angesehen werden kann oder nicht. Die Methode `compatibility` gibt ein Objekt vom Typ `Compatibility` zurück. Wenn ein von `Compatibility` beschriebenes Umweltkriterium nicht relevant ist, wird dafür der größtmögliche Wertebereich angenommen.

`FormicariumPart:` Ein Formicarium oder ein möglicher Bestandteil eines Formicariums.

`FormicariumSet:` Eine (möglicherweise leere) Menge von Formicarien oder Bestandteilen von Formicarien oder Messgeräten bzw. Werkzeugen, die zusammen mit Formicarien verwendbar sind, unabhängig davon, ob sie als Teil eines Formicariums angesehen werden oder nicht. Es ist nicht notwendig (aber erlaubt), dass alle Elemente der Menge zusammen ein Formicarium ergeben. Z. B. könnte es sich bei der Menge auch um einen Lagerbestand oder eine Bestellliste handeln. Über einen Iterator sind die jeweils voneinander verschiedenen Elemente der Menge auflistbar (gleiche Elemente nicht mehrfach). Eine Methode `count` im Iterator retourniert die Anzahl der vorhandenen Elemente, die gleich dem zuletzt von `next` zurückgegebenen Element sind. Die Methode `remove` des Iterators (ohne Argument) verringert die Anzahl vorhandener Elemente (gleich dem, das zuletzt von `next` zurückgegeben wurde) um 1, eine Methode `remove` des Iterators mit einer Zahl gößer 0 als Argument um die gegebene Anzahl gleicher Elemente, sofern eine ausreichende Zahl gleicher Elemente vorhanden ist. Eine Methode `add` mit einem Parameter in einem Objekt von `FormicariumSet` fügt ein neues Element hinzu, sofern das identische Element nicht schon vorhanden ist (gleiche Elemente können jedoch mehrfach vorhanden sein).

`Instrument:` Ein Messgerät oder ein Werkzeug, unabhängig davon, ob es Bestandteil eines Formicariums sein kann oder nicht. Nicht jedes

Objekt von `Instrument` ist zusammen mit Formicarien verwendbar. Das Ergebnis eines Aufrufs der Methode `quality` besagt, ob das Objekt qualitativ für die professionelle Verwendung ausgelegt ist, semiprofessionellen Ansprüchen genügt oder eher nur für die gelegentliche Verwendung gedacht ist.

`Nest:` Ein Bestandteil eines Formicariums, das dafür vorgesehen ist eine Ameisenkönigin zu beherbergen. Für kurze Zeit (wenige Tage, etwa für den Transport) kann das Nest alleine als Formicarium genutzt werden, auf Dauer sind Ameisen darin ohne zusätzliche Arena nicht überlebensfähig. Bei kleineren Formicarien wird das Nest häufig in einer Arena untergebracht, bei größeren ist das Nest nicht selten ein eigenständiger, mit mehr oder weniger langen Röhren mit der Arena verbundener Behälter.

`Thermometer:` Ein Gerät zur Messung der Temperatur ist ein unverzichtbarer Bestandteil eines Formicariums für Ameisen, die bestimmte Temperaturniveaus benötigen. Als Objekte von `Thermometer` werden hier nur solche Temperaturmessgeräte angesehen, die Bestandteil eines Formicariums sein können.

Es ist davon auszugehen, dass gegensätzliche Eigenschaften einander ausschließen (beispielsweise darf ein Gegenstand, der nicht Teil eines Formicariums sein kann, niemals als Teil eines Formicariums vorkommen). Außerdem können Objekte zur Darstellung von Gegenständen, die in der realen Welt eindeutig voneinander verschieden sind und gänzlich unterschiedliche Aufgaben erfüllen (etwa Thermometer, Pinzetten, Glasgefäße) nicht als gleich angesehen werden.

Selbstverständliches

Obige Beschreibungen der Typen sind dahingehend als vollständig anzusehen, dass alle Themen, die bei der Typbildung eine Rolle spielen sollen, angesprochen wurden. Weitere Eigenschaften sollen unberücksichtigt bleiben. Wenn keine konkreten Daten (etwa erlaubte Ameisengrößen) genannt sind, dürfen auch keine konkreten Daten im Typ fix festgelegt werden; aber entsprechende Werte können z. B. über einen Konstruktor gesetzt und in Testfällen mit fixen Werten verwendet werden.

keine zusätzlichen Eigenschaften annehmen

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.

Methoden aus Obertypen übernehmen

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

Begründung wenn keine Untertypbeziehung

Sie eine textuelle Begründung, das Auskommentieren von Programmteilen oder Ähnliches reicht nicht. Das Fehlen einer Methode in einer Typbeschreibung ist als Begründung ungeeignet, weil zusätzliche Methoden hinzugefügt werden dürfen. Auch widersprüchliche Iteratoren eignen sich nicht als Begründung, weil gleichzeitig mehrere Arten von Iteratoren vorkommen dürfen. Sie brauchen keine Begründung dafür angeben, dass A kein Untertyp von B ist, wenn B ein Untertyp von A ist. Um übermäßig lange Texte zu vermeiden, sollten Sie Typen, für die jeweils die gleichen Begründungen gelten, zu Gruppen zusammenfassen.

gegebene Typen mit gegebenen Namen

Alle oben in blauer Schrift genannten Typen müssen mit den vorgegebenen Namen vorkommen, auch solche, die Sie für unnötig erachten. Vermeiden Sie wenn möglich zusätzliche abstrakte Klassen und Interfaces. Vordefinierte Interfaces wie `Iterable` und `Iterator` können natürlich implementiert werden. Vermutlich brauchen Sie zusätzliche Klassen für Iteratoren. Zum Testen können Sie zusätzliche Klassen einführen, aber kennzeichnen Sie diese bitte als nicht zum eigentlichen System gehörend. Viele vorgegebene Methoden sind semantisch sehr einfach und sollen auch so einfach implementiert sein.

Testklasse

Schreiben Sie eine Klasse `Test` zum Testen Ihrer Lösung. Ihr Programm muss (nach erneuter Überstzung aller `.java`-Dateien) vom Abgabeverzeichnis `Aufgabe4` aus durch `java Test` ausführbar sein. Überprüfen Sie mittels Testfällen, ob dort, wo Sie eine Untertypbeziehung annehmen, Ersetzbarkeit wirklich gegeben ist.

Aufgabenaufteilung beschreiben

Außerdem 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.

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

- Untertypbeziehungen richtig erkannt und eingesetzt — 40 Punkte
- nicht bestehende Untertypbeziehungen gut begründet — 15 Punkte
- Zusicherungen passend und zweckentsprechend — 20 Punkte
- Lösung so getestet, dass mögliche Verletzungen der Ersetzbarkeit auffindbar wären — 15 Punkte
- Lösung vollständig (entsprechend Aufgabenstellung) — 10 Punkte

inhaltlich testen

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, weil dies nur die auf Signaturen beruhenden Typeigenschaften berücksichtigt, die ohnehin vom Compiler garantiert werden. Testfälle müssen deutlich über das hinausgehen, was aus Signaturen ablesbar ist.

Die 10 Punkte für „Lösung vollständig“ gelten für jene Bereiche, für die keine speziellen Punkte 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 (geeignet wären z. B. Gegenbeispiele) für nicht bestehende Untertypbeziehungen.

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. Es ist Ihre Aufgabe, relevante von irrelevanten Textteilen zu unterscheiden. 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 wird und damit eine falsche Typstruktur mit höherer Wahrscheinlichkeit erkannt werden kann. Wenn aus *Kontext* nicht klar hervorgeht, wie in einem bestimmten Fall (vor allem im Fehlerfall) genau vorzugehen ist, sollen Sie selbst eine geeignete Vorgehensweise wählen, die aber über Zusicherungen klar beschrieben sein muss. Häufig haben Sie es selbst in der Hand, durch geeignete Wahl der Details die verwendete Art von Zusicherung zu bestimmen (etwa Vorbedingung oder Nachbedingung). Zur Vermeidung von Missverständnissen wird empfohlen, die Arten der Zusicherungen anzugeben (Vor- oder Nachbedingung, Invariante oder History-Constraint). Nicht die Quantität der Kommentare ist entscheidend, sondern die Qualität (Verständlichkeit, Vollständigkeit, Aussagekraft, ...). Auf die Form oder Syntax der Kommentare (umgangssprachlich, formal, javadoc-Stil, ...) kommt es nicht an, aber auf den Inhalt. Zusicherungen in `Test.java` werden bei der Beurteilung aus praktischen Gründen nicht berücksichtigt.

Zusicherungen

Auch beim Testen kommt es auf Qualität, nicht nur Quantität an. Testfälle sollen grundsätzlich in der Lage sein, Verletzungen der Ersetzbarkeit aufzudecken, die nicht ohnehin vom Compiler erkannt werden. Es muss auch der notwendige Umfang gegeben sein, der erwarten lässt, dass Fehler in allen relevanten Bereichen des Programms erkennbar sind.

Qualität vor Quantität

Zur Lösung dieser Aufgabe müssen Sie Untertypbeziehungen und den Einfluss von Zusicherungen genau verstehen. Lesen Sie Kapitel 3 des Skriptums. Folgende Hinweise könnten hilfreich sein:

nicht nur intuitiv vorgehen

- Konstruktoren werden in einer konkreten Klasse aufgerufen und sind daher vom Ersetzbarkeitsprinzip nicht betroffen.
- Ein Objekt eines Untertyps darf nur Ausnahmen auslösen, 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 Ähnlichkeiten im Text oder 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 gleichzeitig B Untertyp von A sein, außer wenn A und B gleich sind. Bei echten Untertypbeziehungen ist immer eindeutig klar, welcher Typ Untertyp des anderen ist.

Form $\neq$ Inhalt

Achten Sie auf die Sichtbarkeit. Alle beschriebenen Typen und Methoden sollen überall verwendbar sein, sonst nichts. Sichtbare Implementierungsdetails beeinflussen die Ersetzbarkeit. Wenn Sie Implementierungsdetails unnötig weit sichtbar machen, sind möglicherweise bestimmte Untertypbeziehungen nicht mehr gegeben, wodurch Sie das Ziel verfehlen, alle realisierbaren Untertypbeziehungen zu ermöglichen.

Sichtbarkeit

Was generell beachtet werden 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, beispielsweise auch nicht `junit`. Viele Übersetzungsfehler kommen von falschen `package`- oder `import`-Anweisungen und unabsichtlich abgegebenen, nicht zur Lösung gehörenden `.java`-Dateien (Reste früher Lösungsversuche oder Backups). Vermeiden Sie auch Umlaute in den Namen von Klassen und Paketen (wobei Pakete für die aktuelle Aufgabe ohnehin wenig sinnvoll sind).

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

Selbstkontrolle