9. Programmieraufgabe

Objektorientierte Programmiertechniken

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

Themen:

Annotationen, Reflexion, Subtyping, Zusicherungen

Ausgabe:

09. 12. 2020

Abgabe (Deadline):

16. 12. 2020, 12:00 Uhr

Abgabeverzeichnis:

`Aufgabe9`

Programmaufruf:

```
java Test
```

Grundlage:

Skriptum, Abschnitt 4.3

Kontext A: Winterruhe

Auch wenn im Spätherbst Ruhe einkehrt, ist die Wiese noch voll Leben. Als Teil einer Simulation benötigen wir folgende Typen zur Darstellung unterschiedlicher Überwinterungsstrategien von Tieren (alphabetisch):

`BrimstoneButterfly:` Der Zitronenfalter ist die einzige heimische Art von Schmetterlingen, die den Winter als Falter im Ruhezustand ungeschützt im Freien verbringt, etwa auf einem Grashalm.

`Earthworm:` Der Regenwurm überwintert meist tief unter der Erdoberfläche. Dort fällt er in der Regel in einen Winterschlaf.

`Hare:` Der Feldhase bleibt den ganzen Winter hindurch aktiv und ernährt sich von Grünpflanzen. Er sucht zwar Schutz in Bodenvertiefungen, vergräbt sich aber nicht unter die Erdoberfläche.

`Hibernating:` Viele Tiere überwintern, indem Sie Aktivitäten weitestgehend vermeiden. Dieser Typ umfasst alle Tiere, die das in der Regel tun, egal ob in Winterstarre, Winterschlaf oder Winterruhe.

`Mole:` Der Maulwurf verlegt seine Grabtätigkeit im Winter tiefer, um sich vor Frost zu schützen, bleibt aber den ganzen Winter über aktiv.

`Subterrestrial:` Unterirdisch sind Tiere gut geschützt. Dieser Typ umfasst alle Tiere, die ganzjährig unter der Erdoberfläche leben oder sich nur über den Winter unter die Erdoberfläche verkriechen.

Es werden Methoden benötigt, um wichtige Parameter abzufragen oder festzulegen, etwa Beginn und Dauer eines Winterschlafs, Energieverbrauch über den Winter, Menge der über den Winter aufgenommenen Nahrung, Überlebenswahrscheinlichkeit und Abhängigkeit der Überlebenswahrscheinlichkeit von der Witterung.

Kontext B: Datenextraktion

Hinter der Softwareentwicklung stehen Managementmaßnahmen, die Projekte vorantreiben und in gewünschte Richtungen lenken. Alle Maßnahmen beruhen auf Daten. Daten sind auch im Quellcode enthalten, müssen aber daraus extrahiert und aufbereitet werden, um verwendbar zu sein. Management benötigen wir auf allen Ebenen, vom Entwerfen einzelner Modularisierungseinheiten bis zu Optimierungen von Kapital, Personal und Finanzen. Entsprechend unterschiedlich sind die benötigten Daten. Konkret sind in dieser Aufgabe folgende Daten mittels Reflexion aus dem laufenden Programm zu extrahieren und übersichtlich darzustellen:

1. Namen aller zur Lösung dieser Aufgabe selbst definierten Klassen, Interfaces und Annotationen zusammen mit allen Untertypbeziehungen, die zwischen diesen Einheiten bestehen.

2. Eine Zuordnung zwischen den Namen aller zur Lösung dieser Aufgabe selbst definierten Klassen, Interfaces und Annotationen zum Namen jeweils eines Gruppenmitglieds, das für die Entwicklung der Einheit hauptverantwortlich ist.

3. Für jede Klasse und jedes Interface die Signaturen aller darin enthaltenen nicht-privaten Methoden und Konstruktoren sowie alle dafür geltenden Zusicherungen (getrennt nach Vor- und Nachbedingungen auf Methoden und Konstruktoren, sowie Invarianten und History-Constraints auf Klassen und Interfaces), einschließlich der Zusicherungen, die aus Obertypen übernommen („geerbt“) wurden.

4. Für jedes Gruppenmitglied die Anzahl der Klassen, Interfaces und Annotationen, für die dieses Gruppenmitglied hauptverantwortlich ist (als eine einzige ganze Zahl).

5. Für jedes Gruppenmitglied die Anzahl der Methoden und Konstruktoren in den Klassen (nur Klassen), für die dieses Gruppenmitglied hauptverantwortlich ist (als eine einzige ganze Zahl).

6. Für jedes Gruppenmitglied die Anzahl der Zusicherungen in den Klassen und Interfaces (samt Inhalten), für die dieses Gruppenmitglied hauptverantwortlich ist (als eine einzige ganze Zahl).

Um die Datenextraktion zur Laufzeit zu ermöglichen, sind hauptverantwortliche Gruppenmitglieder in Form von Annotationen bei Definitionen von Klassen, Interfaces und Annotationen (Definitionen, nicht Verwendungen) anzugeben. Ebenso sind Zusicherungen (getrennt nach Arten) als Annotationen an passenden Programmstellen anzugeben, nicht in Form von Kommentaren. Zusicherungen sollen (trotz Annotationen) für Menschen lesbare Texte darstellen, nicht ausführbaren Code.

Welche Aufgabe zu lösen ist

Entwickeln Sie (ähnlich wie für Aufgabe 4) in Java eine Typhierarchie, die zumindest die unter „Kontext A“ genannten Typen mit entsprechenden Methoden umfasst und überall eine Untertypbeziehung herstellt, wo dies ohne Verletzung des Ersetzbarkeitsprinzips möglich ist. Berücksichtigen Sie dabei auch „Kontext B“, das heißt, schreiben Sie keine Kommentare, sondern verwenden Sie selbst eingeführte Annotationen für Zusicherungen und zur Spezifikation der Hauptverantwortlichen, und sorgen Sie dafür, dass die genannten Daten über Reflexion aus dem laufenden Programm extrahierbar sind.

Schreiben Sie eine direkt im Abgabeverzeichnis ausführbare Klasse `Test`, welche alle unter „Kontext B“ genannten Daten mittels Reflexion extrahiert und in der gegebenen Reihenfolge übersichtlich dargestellt ausgibt. Führen Sie danach in `Test` Testfälle aus, um die Gegebenheiten aus „Kontext A“ zu überprüfen, wobei die Testfälle in der Lage sein sollen, mögliche Verletzungen der Ersetzbarkeit aufzudecken. Unter diesen Testfällen sollen auch solche sein, die über Reflexion testen; das heißt, sie ermitteln z.B. über `getClass` die interne Darstellung (vom Typ `Class`) des Typs eines gegebenen Objekts, überprüfen über Methoden von `Class`

ob bestimmte Methoden entsprechend „Kontext A“ auf dem Objekt ausführbar sind und führen diese gegebenenfalls (wieder über Methoden von `Class`) aus.

Wie die Aufgabe zu lösen ist

Alle Interfaces, Klassen und Definitionen von Annotationen sind mit selbst-definierten Annotationen zu versehen, die angeben, welches Gruppenmitglied jeweils dafür hauptverantwortlich ist. Dabei wird angenommen, dass diese Person auch für alle Inhalte dieser Einheiten verantwortlich ist. Ebenso sind auch Zusicherungen durch selbst-definierte Annotationen darzustellen, wobei für jede Art von Zusicherung eine eigene Art von Annotation vorzusehen ist. Es ist darauf zu achten, an welche Programmteile die Annotationen angeheftet werden können, da nicht jede Art von Zusicherung überall sinnvoll ist. Annotationen müssen auch zur Laufzeit verfügbar sein, da über Reflexion darauf zugegriffen wird.

Daten sind über Reflexion zu extrahieren. Das impliziert die Unabhängigkeit des Extraktionsprozesses von den Einheiten, aus denen die Daten extrahiert werden. Die Änderung einer Klasse, deren eigentliche Aufgabe nicht die Datenextraktion ist, darf nicht dazu führen, dass der Code für die Datenextraktion (oder eine für die Datenextraktion verwendete Datenstruktur) geändert werden muss. Das heißt, man muss (ausgehend von einem festgelegten Anfangspunkt) zur Laufzeit ermitteln, wie die Typstruktur ausschaut und welche Methoden, Annotationen und so weiter vorhanden sind. Es reicht nicht, diese Daten statisch festzulegen.

Methoden entsprechend „Kontext A“ sind im Wesentlichen frei wählbar. Die Methoden sind aber so zu wählen, dass sie nicht im Widerspruch zur Forderung stehen, Untertypbeziehungen überall dort herzustellen, wo es ohne Verletzung der Ersetzbarkeit möglich ist. Beim Überprüfen der Ersetzbarkeit muss man davon ausgehen, dass Tiere nicht gleichzeitig oberirdisch und unterirdisch überwintern. Sie können auch nicht gleichzeitig aktiv und im Winterschlaf sein. Regenwürmer, Hasen, Maulwürfe und Schmetterlinge sind ganz unterschiedliche Tierarten, die nicht in Untertypbeziehung zueinander stehen können.

Kommentare sind nicht nötig. Alles, was bisher über Kommentare gemacht wurde, soll über Annotationen erledigt werden. Entsprechende Arten von Annotationen sind selbst zu definieren.

Beachten Sie die feinen Unterschiede in den Details von „Kontext B“: Beispielsweise sind entsprechend Punkt 3 auch von Obertypen übernommene Zusicherungen auszugeben, während ererbte Zusicherungen entsprechend Punkt 6 nicht zählen.

Was im Hinblick auf die Beurteilung wichtig ist

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

- Geforderte Funktionalität vollständig vorhanden — 25 Punkte
- Annotationen und Reflexion richtig verwendet — 30 Punkte
- Zusicherungen und Untertypbeziehungen (samt Methoden) richtig und sinnvoll eingesetzt — 25 Punkte

- Test-Output sinnvoll und nachvollziehbar gestaltet 10 Punkte
- Sichtbarkeit auf kleinstmögliche Bereiche beschränkt 10 Punkte

Kräftige Punkteabzüge gibt es, wenn das Programm nicht wie verlangt funktioniert oder die geforderten Konzepte nicht oder nicht zweckgemäß eingesetzt werden.

Warum die Aufgabe diese Form hat

Die entsprechend „Kontext A“ nötigen Typen sind bewusst einfach und relativ frei gestaltet. Aufgrund der notwendigen formalen Zuordnung von Zusicherungen zu Zusicherungsarten ist es jedoch nicht möglich, diesbezügliche Unsicherheiten zu überspielen.

Neben dem Sammeln von Erfahrungen im Umgang mit Annotationen und Reflexion geht es auch darum, das Denken auf gleichzeitig mehreren Ebenen zu üben. Am Rande soll darauf aufmerksam gemacht werden, dass im Programmtext scheinbar viel Meta-Information enthalten ist, die etwa als Basis für Personalentscheidungen herangezogen werden könnte, aber kaum tatsächliche Gegebenheiten widerspiegelt.