4. Programmieraufgabe

Programmierparadigmen
LVA-Nr. 194.023
2024/2025 W
TU Wien

Themen:
Untertypbeziehungen, Zusicherungen

Ausgabe:
11. 11. 2024

Abgabe (Deadline):
25. 11. 2024, 14:00 Uhr

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

Programmaufruf:
`java Test`

Grundlage:
Skriptum, Kapitel 3

Kontext und Spezifikation

Die Aufgabe bezieht sich auf ein geplantes System zur digitalen Darstellung der Gebäude einer Stadt. Unter Anderem soll Information verfügbar sein, die von Rettungskräften wie der Feuerwehr benötigt wird. 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:

`Building:` Ein *Gebäude* ist ein Bauwerk, das dafür vorgesehen ist, als Hülle betretbare Räume zu umschließen, die den Aufenthalt von Menschen ermöglichen (und weitere Aufgaben erfüllen können). Es ist mit dem Untergrund verbunden; ein aufgesetztes Geschoss ist kein eigenes Gebäude. Zwei Baukörper sind getrennte Gebäude, wenn jedes von ihnen entfernt werden kann und das andere danach noch immer ein vollständiges Gebäude ist. Andernfalls handelt es sich um ein einziges (aus mehreren Baukörpern zusammengesetztes) Gebäude. Die Methode `spaces()` gibt die Menge aller Innenräume und Außenbereiche zurück, die in das Gebäude eingebunden sind.

`Circulation:` Das sind *Erschließungsflächen,* also alle zur Fortbewegung vorgesehenen Bereiche innerhalb und außerhalb eines Gebäudes, etwa Flure, Treppenhäuser und Zugangswege. Jeder für den kurz- oder langfristigen Aufenthalt von Menschen vorgesehene Bereich muss *erschlossen* sein, das heißt, es muss zumindest einen (nicht zu langen) Weg geben, über den dieser Bereich betreten werden kann. Unter einem Weg verstehen wir hier eine begehbare Abfolge von Innenräumen und Außenbereichen, die bis zu einer öffentlichen Straße führt. Manchmal erfolgt die Erschließung eines Raums über einen anderen Raum, der nicht primär der Erschließung dient, etwa wenn ein Bad über ein Schlafzimmer zugänglich ist. In solchen Fällen betrachten wir das Schlafzimmer neben seiner Hauptfunktion auch als Erschließungsfläche. Das Bad ist im Beispiel keine Erschließungsfläche, wenn kein weiterer Raum über das Bad erreichbar ist. Viele Erschließungsflächen erfüllen nebenbei weitere Aufgaben, beispielsweise kann ein Flur eine Garderobe beherbergen.

`Complex:` Ein *Gebäudekomplex* ist ein Zusammenschluss mehrerer baulich miteinander verbundener Gebäude, die gemeinsam erschlossen sind (gemeinsame Eingänge). Oft kann man über Innenräume von einem Gebäude in ein anderes gelangen. Auch Außenräume, die in kein Gebäude eingebunden sind, etwa Höfe, können in einen Gebäudekomplex eingebunden sein; sie sind der Allgemeinheit meist nicht frei zugänglich. Beispiel: Hofburg. Die Methode `buildings()` gibt die Menge der Gebäude zurück, die zum Komplex gehören. Die Methode `spaces()` gibt die Menge der Außenbereiche zurück, die in den Komplex, aber nicht in Gebäude eingebunden sind.

`Ensemble:` Ein *Gebäudeensemble* ist ein loser Zusammenschluss mehrerer Gebäude oder Gebäudekomplexe, die als zusammengehörig wahrgenommen werden, obwohl sie getrennt voneinander erschlossen sind (getrennte Eingänge). Als Beispiel bilden die Gebäude entlang einer Straße oder rings um einen Platz ein Ensemble, wenn die Straße oder der Platz durch die Gebäude als eigener Bereich erkennbar wird; die Gebäude schließen den Bereich (unvollständig) ab, ohne Teil davon zu sein. Beispiel: Stephansplatz und Graben. Auch verstreut liegende, einander benachbarte Gebäude mit gemeinsamen Merkmalen bilden ein Ensemble, wenn sie als Einheit erkennbar sind. Beispiel: Wohnsiedlung am Stadtrand. In der Regel kann nicht die ganze Stadt als ein einziges Ensemble betrachtet werden. Die Methode `entities()` gibt die Menge der Gebäude(komplexe) zurück, die zum Ensemble gehören. Ein Gebäude(komplex) kann zu mehreren Ensembles gehören (etwa Vorder- und Rückansicht zu unterschiedlichen). Die Methode `space()` gibt den Bereich zurück, der durch das Ensemble umschlossen wird, oder `null` bei Nichtexistenz.

`Entity:` Eine *gebaute Einheit,* also ein Gebäude, Gebäudekomplex, Gebäudeensemble und die ganze gebaute Stadt. Objekte von `Entity` können nicht vom Typ `Space` sein, weil sich im Material des Gebauten (z. B. Mauerwerk) kein Mensch aufhalten kann. Natürlich kann etwas Gebautes Bereiche enthalten, in denen sich Menschen aufhalten können. Es werden Methoden bzw. Konstruktoren benötigt (Details nicht vorgegeben), um gebaute Einheiten einzufügen und zu entfernen. Dabei können sich Arten gebauter Einheiten ändern (z. B. vom Gebäudekomplex bleibt nur ein einzelnes Gebäude). Die Stadt selbst kann nicht entfernt werden. Beim Entfernen gebauter Einheiten sind auch alle betroffenen eingebundenen Bereiche zu entfernen, nach dem Einfügen können neue Bereiche über dafür bereitzustellende Methoden eingebunden werden.[1]

`Escape:` Jeder Bereich, der von Menschen betreten werden kann, muss im Notfall rasch verlassen werden können. Ein Objekt von `Escape` stellt kürzeste Wege dar, die zeigen, wie ein Bereich bis zu einer öffentlichen Straße verlassen oder von einer öffentlichen Straße aus betreten werden kann. Wege sind Listen von Erschließungsflächen, die in der gegebenen Reihenfolge zu durchqueren sind. Der letzte (beim Verlassen) bzw. erste (beim Betreten) Listeneintrag ist eine öffentliche Straße. Der Bereich, der verlassen oder beteten werden soll, ist nicht Teil des Wegs. Anders als normale Wege dürfen *Fluchtwege* nicht über Aufzüge führen, auch zum Betreten in Gefahrensituationen sind Aufzüge nicht verwendbar. Eine öffentliche Straße bietet immer ausreichend Fluchtmöglichkeiten, die nicht dargestellt werden. Die Methode `space()` gibt den Bereich zurück, der verlassen oder betreten werden soll. Die Methode `iterator(boolean lift, boolean enter)` gibt einen Iterator zurück, der alle Erschließungsflächen des kürzesten Wegs in der zu durchquerenden Reihenfolge

[1] Beim Einbinden oder Entfernen von Bereichen in/aus gebaute(n) Einheiten geht es nicht ums Ausgraben oder Zuschütten von Leerräumen, sondern etwa ums Ausstellen oder Zurückziehen von Genehmigungen, ohne die keine Nutzung erlaubt ist.

liefert; dabei besagt `lift`, ob der Weg Aufzüge einbeziehen kann und `enter`, ob der Weg in den Bereich hineinführt (sonst hinaus). Dieser Iterator darf, egal wie aufgerufen, maximal 10 Einträge liefern. Andernfalls gilt der Bereich nicht als ausreichend gut erschlossen (kein ausreichend rasches Betreten oder Verlassen möglich) und darf nicht in dieser Form in eine gebaute Einheit eingebunden sein.

`Exterior:` Ein Bereich außerhalb eines Gebäudes (*Außenbereich*). Auch Terrassen, Balkone oder Loggien sind Außenbereiche, da Öffnungen in der Gebäudehülle nicht schließbar sind, auch wenn sie als Räume wahrgenommen werden und in Gebäude eingebunden sind.

`Interior:` Ein Raum innerhalb eines Gebäudes (*Innenraum*), der abgesehen von schließbaren Öffnungen (Fenster und Türen) rundum durch eine Hülle (nicht nur scheinbar) abgeschlossen ist. Manchmal sind Raumgrenzen so gestaltet, dass sie kaum in Erscheinung treten (etwa Glaswände). Dennoch müssen sie vorhanden sein.

`Lift:` Ein *Aufzug* ist ein (meist kleiner) Innenraum ohne öffenbare Fenster, der ausschließlich der maschinell unterstützten vertikalen Erschließung dient. Im Gegensatz zu anderen Erschließungsflächen dürfen Aufzüge in vielen Gefahrensituationen nicht benutzt werden, sodass Fluchtwege ohne Aufzüge auskommen müssen.

`PublicRoad:` Eine *öffentliche Straße* ist ein Bereich, der uneingeschränkt zugänglich ist und ausschließlich der Fortbewegung (Personen- und Fahrzeugverkehr) dient. Einrichtungen wie Parkplätze, Baumreihen, Grünstreifen oder Sitzbänke werden nicht als andere Nutzung gesehen, sondern als Unterstützung für bestimmte Formen der Fortbewegung. Die Methode `escape()` gibt immer `null` zurück.

`PureCirculation:` Das ist eine Erschließungsfläche, die ausschließlich der Erschließung (also der Fortbewegung) dient und keine andere Nutzung erlaubt. Reine Erschließungsflächen sind dort vorhanden, wo viele Personen unterwegs sind und nötigenfalls flüchten müssen, ohne durch andere Nutzungen der Fläche behindert zu werden.

`Room:` Ein *Raum* ist ein rundum abgeschlossener Bereich, abgesehen von (möglicherweise schließbaren) Öffnungen, die immer nötig sind, um den Raum zu betreten. Die Größe der Öffnungen ist nicht entscheidend dafür, ob ein Bereich als Raum gesehen wird. Vielmehr geht es um die Erkennbarkeit von Raumgrenzen. Beispielsweise kann ein Raum nach oben bzw. auf ein oder mehreren Seiten offen sein, wenn die Ränder der Wände oder hinzugefügte Rahmen klar erkennbare Begrenzungslinien bilden. Die Hülle, die den Raum (physisch oder scheinbar) umschließt, ist selbst nicht Teil des Raums. Räumen wird eine Schutzfunktion zugesprochen. Die Hülle um den Raum kann vor Umwelteinflüssen und Gefahren schützen. Aber man fühlt sich im Raum auch dann geschützt, wenn die (scheinbare) Hülle tatsächlich kaum Schutz bietet. Meist bemüht man sich, Aufenthaltsbereiche von Menschen als Räume zu gestalten. Manchmal versucht man jedoch Räume zu vermeiden, also Übergänge so diffus zu gestalten, dass keine Raumgrenzen erkennbar sind.

`ServantSpace`: Eine Trennung der einfachen Räume der Diener von repräsentativen Räumen der Herrschaften gibt es heute nicht mehr. Aber die von Louis Kahn geprägte Unterscheidung zwischen *dienenden* (servant space) und *bedienten* Innenräumen (served space) ist auch heute noch präsent: Dienend sind Nebenräume, die nur auf die Erfüllung bestimmter Aufgaben und den kurzzeitigen Aufenthalt von Menschen ausgerichtet sind (WC, Technikraum, Treppenraum, ...), während bediente Räume für den längeren Aufenthalt von Menschen konzipiert sind (Wohnzimmer, Schlafzimmer, Büro, ...). Etwa eine Küche oder ein Bad kann als dienender Raum mit rein funktionaler Ausstattung klein und versteckt gehalten werden, oder als bedienter Raum großzügig gestaltet eine hohe Wohnqualität bieten. Häufig werden mehrere dienende Räume zu Blöcken zusammengefasst und so positioniert, dass sie von den bedienten Räumen aus leicht erreichbar sind, diese aber nicht stören. Wir gehen davon aus, dass alle Objekte von `ServantSpace` und `ServedSpace` immer Innenräume sind. Bediente Räume müssen bestimmte Qualitätsmerkmale erfüllen – ausreichende natürliche und künstliche Belichtung und Belüftung, Heizung, Mindesthöhen, Bewegungsflächen, zweite Fluchtwege über öffenbare Fenster ins Freie –, die für dienende Räume nicht nötig sind (wobei es jedoch für Sanitärräume, Küchen, Treppen, ... unabhängig davon eigene Vorschriften gibt). Alle Innenräume, die hinsichtlich Positionierung und Ausstattung als bedient anzusehen sind (bei denen also erwartet werden kann, dass sich Menschen dort länger aufhalten), müssen die in Bauordnungen vorgegebenen Mindeststandards erfüllen.

`ServedSpace`: Ein bedienter Innenraum (siehe `ServantSpace`). Die Methode `alternativeEscape()` gibt die Summe der lichten Flächen (in m^2) aller ausreichend dimensionierten öffenbaren Fenster vom Raum ins Freie zurück, die im Notfall als zweite Fluchtwege nutzbar wären. Diese muss mindestens 1,1 m^2 betragen.

`Space`: Ein Bereich, der (in eine gebaute Einheit eingebunden) den Aufenthalt von Menschen ermöglicht. Er kann ein Raum sein, muss aber nicht. Obwohl die meisten Aufenthaltsbereiche von Menschen Räume sind, wird dennoch oft von `Space` gesprochen, um die Fälle, in denen Übergänge zu diffus gestaltet sind, um von Raumgrenzen und Räumen zu sprechen, nicht auszuschließen. Zur Vereinfachung gehen wir davon aus, dass verschiedene Objekte von `Space` (ebenso wie Objekte von `Room`) sich nicht überlappen; sie können sich aber an definierten Rändern (die es zwar bei Räumen immer geben muss, aber nicht bei allen Bereichen) berühren, wenn sie neben- oder übereinander liegen. Die Methode `entity()` gibt die kleinstmögliche gebaute Einheit zurück, in die der Bereich eingebunden ist, oder `null`, wenn er nirgends eingebunden und daher nicht benutzbar ist. Die Methode `escape()` gibt ein Objekt von `Escape` zurück, das kürzeste Wege beschreibt, oder `null`, wenn der Bereich nirgends eingebunden oder eine öffentliche Straße ist. Die Methode `remove()` entfernt den Bereich aus seiner Umgebung (macht ihn unbenutzbar) und ebenso alle anderen Bereiche, die danach nicht

mehr ausreichend gut erschlossen sind (siehe `Escape`); zurückgegeben wird die Menge der entfernten Bereiche. Durch `remove()` können sich auch kürzeste Wege anderer Bereiche ändern. Methoden zum Einbinden von Bereichen in gebaute Einheiten gibt es in `Entity`. Auch diese Methoden können Wege ändern.

Selbstverständliches

Gegensätzliche Eigenschaften schließen einander aus. Beispielsweise kann eine rundum geschlossene Hülle nicht gleichzeitig offen und ein in keine gebaute Einheit eingebundener Bereich nicht gleichzeitig Teil eines Fluchtwegs sein. Objekte zur Darstellung von Dingen, die in der realen Welt eindeutig voneinander verschieden sind und gänzlich unterschiedliche Aufgaben erfüllen, können nicht gleich sein. Allerdings dürfen Feinheiten in der Verwendung natürlicher Sprache nicht überinterpretiert werden. Semantisch unterschiedliche Begriffe können im Zusammenhang doch das Gleiche bedeuten. Beispielsweise kann ein und dieselbe Sache manchmal als „Fläche“ (wenn für einen Teil der Beschreibung nur zwei Dimensionen relevant sind) und manchmal als „Raum“ (wenn auch die dritte Dimension von Bedeutung ist) oder neutral als „Bereich“ bezeichnet werden.

keine zusätzlichen Eigenschaften annehmen

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 Positionen von und Verbindungen zwischen Räumen) 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.

Methoden aus Obertypen übernehmen

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

alle Zusicherungen und Untertypbeziehungen

Schreiben Sie ein Java-Programm, das für jeden unter *Kontext und Spezifikation* angeführten Typ eine (abstrakte) Klasse oder ein Interface bereitstellt. Versehen Sie alle Typen mit 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 vorgegebenen Beschreibungen der Typen widersprechen würden.

Begründung wenn keine Untertypbeziehung

Besteht zwischen zwei Typen keine Untertypbeziehung, geben Sie in einem Kommentar in `Test.java` eine Begründung dafür an. Bitte geben 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. 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. Ver-

meiden Sie wenn möglich zusätzliche abstrakte Klassen und Interfaces. Vordefinierte Interfaces wie `Iterator` können natürlich implementiert werden. Vermutlich brauchen Sie zusätzliche Klassen für Iteratoren. Zum Testen können Sie zusätzliche Klassen einführen (etwa konkrete Klassen als Untertypen vorgegebener abstrakter Klassen oder Interfaces), aber kennzeichnen Sie diese bitte als nicht zum eigentlichen System gehörend.

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

Die Datei `Test.java` soll als Kommentar auch 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 Fehler, 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.

alle Untertypbeziehungen

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. Bedenken Sie, dass das Herstellen aller Untertypbeziehungen etwas anderes ist, als eine „vernünftige“ Programmstruktur zu finden. Besteht zwischen

zwei Typen keine Untertypbeziehung, können Sie eine Begründung aus den gegebenen Typbeschreibungen herauslesen. Diese Begründung sollen Sie nennen. Als Begründung ungeeignet sind Argumente, die sich auf Annahmen beziehen, die nicht in den Typbeschreibungen stehen, oder die den Folgenden ähneln: „Untertypbeziehung nicht benötigt“, „ist offensichtlich“ oder „Methoden passen nicht zusammen“.

Zusicherungen

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 und Spezifikation* nicht hervorgeht, wie in einem bestimmten Fall genau vorzugehen ist, sollen Sie selbst eine geeignete Vorgehensweise wählen, die über Zusicherungen zu beschreiben ist. 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.

Halten Sie die Anzahl der eingeführten Klassen und Methoden möglichst klein. Jeder zusätzliche Typ und jede Methode erhöht die Fehlerwahrscheinlichkeit, durch den paarweisen Vergleich der Typen eher in quadratischem als linearem Ausmaß.

Qualität vor Quantität

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. Vermeiden Sie die Verwendung vorgefertigter Testumgebungen, da diese überwiegend andere Programmeigenschaften prüfen als jene, auf die es in dieser Aufgabe ankommt. Verzichten Sie auch auf die Entwicklung einer eigenen Testumgebung. Achten Sie lieber darauf, welche Art von Testfällen zur Lösung der aktuellen Aufgabe am besten geeignet ist und schreiben Sie dafür möglichst einfachen Code zur Überprüfung. Einfacher Code bedeutet nicht eine kleine Zahl an Testfällen, sondern im Gegenteil, viele Testfälle, die alle etwas anders aufgebaut (und daher keinem fixen Schema unterworfen) sind, aber auf Code zum Einbetten in eine Testumgebung verzichten. Die Ausgabe der Testergebnisse soll übersichtlich, muss aber nicht „schön“ sein. Es wäre eine Verschwendung von Ressourcen, die Überprüfung eines einfachen Methodenaufrufs in mehrzeiligen Code einzubetten, der im Erfolgsfall anderen Text ausgibt als im Fehlerfall; eine

informativ, nicht „schön“

einfache Ausgabe des Methodenergebnisses wäre meist informativer.

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. Interfaces können Default-Implementierungen enthalten.

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 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. Vermeiden Sie 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 so eindeutig wie möglich sind. Über Testfälle und Gegenbeispiele sollten Sie schwere Fehler selbst finden können.

eindeutig

Selbstkontrolle