SE2, Aufgabenblatt 3

Modul: Softwareentwicklung II – Sommersemester 2017

Subtyp-Polymorphie, Typhierarchien

Moodle-URL ..... moodle.informatik.uni-hamburg.de
Projektraum ..... Softwareentwicklung 2 SoSe2017
Ausgabedatum ..... 20. April 2017

Kernbegriffe

In objektorientierten Programmiersprachen können Typen hierarchisch angeordnet werden, es entstehen *Typhierarchien*. Jede Klasse und jedes Interface in Java definiert einen Typ. Subtypen werden gebildet, indem ein Interface ein anderes erweitert oder indem eine Klasse eine andere Klasse erweitert bzw. ein Interface implementiert. *Subtypen* (speziellere Typen) bieten immer mindestens die Schnittstelle ihrer *Supertypen* (allgemeinere Typen), darüber hinaus können sie diese Schnittstelle um eigene Operationen erweitern.

An eine Referenz-Variable eines bestimmten Typs können Exemplare eines spezielleren Typs gebunden werden; dies bezeichnet man als *Subtyp-Polymorphie*. Der deklarierte Typ einer Variablen ist dabei ihr *statischer Typ*. Ist eine Variable an ein Objekt gebunden (und hält damit zur Laufzeit eine gültige Referenz auf ein Objekt), dann bestimmt die Klassenzugehörigkeit dieses Objekts den *dynamischen Typ* der Variablen. Können sich bei einer Zuweisung der statische und der dynamische Typ unterscheiden (in Java bei Referenzvariablen), nennen wir die Variable *polymorph gebunden*. Ruft man an einer polymorph gebundenen Variablen eine Operation auf, hängt es von ihrem dynamischen Typ ab, welche Implementation tatsächlich aufgerufen wird (aus welcher Klassendefinition die gerufene Methode stammt); dies nennt man *dynamisches Binden*.

Will man ein Exemplar eines spezielleren Typs, das an eine Variable mit einem allgemeineren statischen Typ gebunden ist, wieder unter seinem speziellen Typ ansprechen (um auch Operationen aufrufen zu können, die nur der speziellere Typ besitzt), muss eine *Typzusicherung* (auch: *Downcast*) durchgeführt werden. In Java geschieht dies durch das Schreiben des spezielleren Typs in runden Klammern vor einem Ausdruck. *Vorsicht*: Eine Typzusicherung ohne vorherigen *Typtest* (in Java mit `instanceof`) kann zur Laufzeit zu einer `ClassCastException` führen, wenn das Exemplar nicht dem spezielleren Typ entspricht.

Um in Java eine Klasse zu einem Subtyp eines Interfaces zu machen, muss im Klassenkopf mit dem Schlüsselwort `implements` erklärt werden, dass die Klasse das Interface implementiert. Dabei handelt es sich um reine *Typ-Vererbung*, d.h. es werden lediglich Operationsdeklarationen, aber keine implementierenden Methoden vererbt. Wenn eine Klasse, von der Exemplare erzeugt werden können, ein Interface implementiert, müssen für alle im Interface definierten Operationen in dieser Klasse Methoden implementiert werden.

Eine Klasse kann in Java auch mehrere Interfaces implementieren und auch ein Interface kann von mehreren Interfaces erben (mit dem Schlüsselwort `extends`). Dies nennt man *Mehrfach(typ)vererbung* oder *multiples Subtyping*.

Aufgabe 3.1 Typvererbung

Die Mediathek soll nun auch Videospiele verleihen können. Da die Mediathek bereits das Interface `Medium` enthält, ist es recht einfach, neue Medienarten hinzuzufügen. Die Variablen, die für die Ausleihe verwendet werden, sind statisch auf `Medium` getypt.

3.2.1 Um Videospiele verleihen zu können, müssen diese im Programm repräsentiert werden. Hierfür muss es eine Klasse `Videospiel` mit zugehörigem Test geben. Schreibt im Sinne des Test-First-Ansatzes zuerst den Test mit einer leeren Videospiel-Klasse. Danach implementiert ihr die Methoden des Videospiels. Die Klasse `Videospiel` soll, genau wie die Klassen `CD` und `DVD`, das Interface `Medium` implementieren.

3.2.2 Zusätzlich sollen Videospiele noch das System speichern, auf dem das Spiel lauffähig ist, beispielsweise Xbox One, WII U oder PS4. Entscheidet welchen Typ ihr für das Speichern dieser Information verwenden wollt. Erweitert `VideospielTest` und implementiert danach die benötigten Änderungen im `Videospiel`.

3.2.3 Nun sollen auch Videospiele an der Oberfläche angezeigt werden. In der Klasse `MedienEinleser` gibt es eine Methode `leseMediumEin()`. Hier findet ihr eine auskommentierte Zeile, in der Videospiele erzeugt werden. Schaut euch die Parameterreihenfolge im Kontruktoraufruf an. Stimmt dieser mit eurem Konstruktor überein? Nehmt die Kommentarzeichen weg und probiert aus, ob nun auch Videospiele zur Ausleihe angeboten werden.
Notiert im Quelltext, an welchen Stellen sich der dynamische vom statischen Typ einer Variablen unterscheidet.

Aufgabe 3.2 Typzusicherung und dynamisch gebundener Operationsaufruf

Euch ist sicherlich aufgefallen, dass in der Programmoberfläche in dem rechten freien Bereich mit der Überschrift „Ausgewählte Medien“ stets nur eine weiße Fläche zu sehen ist. Dies wollen wir nun ändern und alle Details zu den selektierten CDs, DVDs und Videospielen in der Medientabelle anzeigen lassen.

3.3.1 Öffnet die Klasse `AusleihWerkzeug`. Sucht die Methode `zeigeAusgewaehlteMedien`. Sie wird immer aufgerufen, wenn sich die Selektion in der Medien-Tabelle ändert. Folgt dem Methodenaufruf `setMedien`. In dieser Methode ist bereits ein Quelltextgerüst enthalten. An der mit ToDo markierten Stelle wollen wir nun die spezifischen Medieninformationen an der Benutzungsschnittstelle anzeigen lassen. Macht zuerst zu jedem Medium einen Typtest auf `CD, DVD` und `Videospiel`. Führt danach eine Typzusicherung durch. Fragt dann die spezifischen Informationen zur CD, DVD bzw. zum Videospiel an den Exemplaren ab, und fügt diese Informationen der `TextArea` hinzu. Eure Änderungen sollten in der Programmoberfläche zu einem sichtbaren Ergebnis führen.

3.3.2 Jetzt vereinfachen wir den Quelltext, indem wir einen dynamisch gebundenen Operationsaufruf einbauen. Ergänzt hierzu die Schnittstelle des Interfaces `Medium` um die Operation `String getFormatiertenString()`. Denkt an das Vertragsmodell. Schreibt einen Schnittstellenkommentar, der wiedergibt, dass eine implementierende Methode eine Text-Repräsentation mit allen Eigenschaften des speziellen Mediums zurückgibt.

3.3.3 *Erstellt eine auskommentierte Kopie der Methode `setMedien` für die spätere Abnahme der Aufgabe*. Baut nun euren in 3.3.2 geschriebenen Quelltext um, so dass die neue Operation `getFormatiertenString()` verwendet wird, um ein Medium als Text darzustellen. **Schreibt die Antworten zu den folgenden Fragen auf**: Warum braucht ihr nun kein **`instanceof`** mehr? Was ist dynamisches Binden?

Wir brauchen kein instanceof mehr, da jede Implementation des Medium-Interfaces die Formatierung der Informationen übernimmt.

Dynamisches Binden beschreibt ein Konzept, wobei die Methodenimplementation der aktuellen Klasse genutzt wird.

Aufgabe 3.3 Statischer und dynamischer Typ, Typhierarchien

3.1.1 Beantwortet die gegebenen Fragen zu folgendem Programmausschnitt:

```
 1   Girokonto giro = new Girokonto();
 2   giro.setzeDispo(12000);
 3   giro.zahleEin(1320);
 4   Sparkonto spar = new Sparkonto(2000);
 5   Konto y = new Girokonto();
 6   Konto x = giro;
 7   Object o = giro;
 8   x.zahleEin(100);
 9   x = spar;
10   System.out.println(o.toString());
11   o = x;
12   System.out.println(o.toString());
```

Hinweis: Die Klassen `Girokonto` und `Sparkonto` implementieren das Interface Konto.

Zeile 4: Unterscheiden sich statischer und dynamischer Typ der Variablen? nein

Zeile 5: Unterscheiden sich statischer und dynamischer Typ der Variablen? ja

Zeile 6: Unterscheiden sich statischer und dynamischer Typ der Variablen? ja

Zeile 6: Was ist der statische Typ von x? Konto

Zeile 6: Was ist der dynamische Typ von x? Girokonto

Zeile 10: Was ist der statische Typ von o? Object

Zeile 10: Was ist der dynamische Typ von o? Girokonto

Zeile 12: Was ist der dynamische Typ von o? Sparkonto

3.1.2 Formuliert die Antworten auf die folgenden Fragen **schriftlich**.

Ein Beispiel für eine Typhierarchie bilden die Interfaces im Java Collections Framework. Die euch bekannten Interfaces `Set` und `List` haben einen gemeinsamen Supertyp. **Welcher ist das?**

Schaut euch an, welche Operationen in diesen drei Interfaces definiert werden. Fügen die Subtypen weitere Operationen hinzu, **wenn ja, welche und warum?**

Werden Operationen doppelt deklariert? Was bedeutet es, wenn Operationen in Subtypen erneut deklariert werden; handelt es sich dann um eine neue Operation?

Diskutiert mit den Betreuern und Betreuerinnen über euren Text.

Collection<E>, Iterable<E>

List.add(int index, E element)
List.get(int index)
List.indexOf(Object o)
List.lastIndexOf(Object o)
List.listIterator()
List.set(int index, E element)
List.sublist(int fromIndex, int toIndex)

Nein, es handelt sich um eine neue Implementation