

Musterlösung 5

Aufgabe 1 (Namenskollision und Polymorphie)

Welche der folgenden Behauptungen sind korrekt? Erklären Sie jeweils in einem Satz!

- 1) RICHTIG: Attribute und statische Methoden lassen sich in Java nur verdecken und nicht ersetzen.

Ein einmal eingeführtes Attribut steht in allen Untertypen zur Verfügung.

- 2) FALSCH: Will man beim Ersetzen von nicht-statischen Methoden den Rückgabetyt ändern, so geht dies nur, wenn der neue Rückgabewert Obertyp des Typs des bisherigen Rückgabewertes ist.

Umgekehrt. Er muss Untertyp sein, da sichergestellt werden muss, dass überall dort wo die Methode dynamisch gebunden wird, ein Wert zurückgeliefert wird, der verträglich mit dem erwarteten Typ ist.

- 3) FALSCH: Beim Ersetzen müssen mindestens immer alle Ausnahmen der überschriebenen Methode (bzw. Untertypen von diesen) deklariert werden.

Mindestens ist falsch. Es muss heißen: Beim Ersetzen dürfen maximal alle Ausnahmen der überschriebenen Methode (bzw. Untertypen von diesen) deklariert werden. Allgemeine Begründung: "Die Ersetzung darf nicht mehr Ausnahmen hervorrufen".

- 4) FALSCH: Da Java keine Mehrfachvererbung bei Klassen erlaubt, kann es keine Kollisionen zwischen den Implementierungen von Methoden, sondern immer nur zwischen Signaturen geben.

Es kann sehr wohl zu horizontalen Kollisionen zwischen Methoden kommen, wenn ein Untertyp eine Methode ersetzen möchte. Die genauen Regeln finden Sie auf Folie 2.1 S. 35.

- 5) RICHTIG: Durch Implementierung von Interfaces kann es zu horizontalen Namenskollisionen im Bezug auf Attribute kommen, welche man aber durch explizite Bezugnahme auflösen kann.

Namenskollisionen im Bezug auf Attribute sind immer durch Angabe eines entsprechenden statischen Typbezugs aufzulösen.

- 6) FALSCH: Eine innere Klasse innerhalb einer umschließenden Klasse entspricht einer Vererbungsbeziehung zwischen den beiden.

Vererbung stellt eine "Ist-Ein"-Beziehung zwischen Exemplaren her. Innere Klassen hingegen führen zu einer "Kann-Haben"-Beziehung. Vgl: Jede Bank KANN Konten HABEN, jedes Sparkonto IST EIN Konto.

- 7) RICHTIG: `this` und `super` dürfen nur in nicht-statischen Methoden verwendet werden, da an allen anderen Orten kein Bezug auf ein Exemplar besteht.

`this` und `super` dienen immer dem Bezug auf ein bestimmtes Exemplar mit der Besonderheit, dass `super` einen Zugriff auf die Methoden, Konstruktoren und Attribute der Oberklasse ermöglicht.

- 8) FALSCH: Ein mit `final` markiertes Attribut darf bei der Vererbung nicht verdeckt werden.

Leider verwendet Java den Modifizierer `final` in zwei verschiedenen Bedeutungen. Für Klassen und Methoden wird verboten weitere Änderungen durch Vererbung bzw. Ersetzen vorzunehmen. Attribute werden jedoch durch `final` nur als konstant deklariert, d.h. mit einem unveränderlichen Wert versehen.

- 9) FALSCH: Hat eine Oberklasse keinen Konstruktor, dann kann man in Java keine Unterklasse von ihr ableiten.

Die Lage ist komplizierter: Bei der Erstellung eines Exemplars durch den `new`-Operator muss in der zugehörigen Klasse und allen Oberklassen ein Konstruktor aufgerufen werden. Damit dies nicht zu kompliziert ist, gibt es zwei Hilfestellungen:

- Ist in einer Klasse kein Konstruktor vorhanden, wird ein Standardkonstruktor mit leerer Argumentenliste bereitgestellt.
- Ist in einem Konstruktor kein expliziter Aufruf an einen Konstruktor der Oberklasse mit `super(...)` angegeben, wird ein Aufruf an den parameterlosen Konstruktor `super()` angenommen.

Dies kann zu folgendem Nebeneffekt führen: Hat eine Oberklasse keinen Konstruktor deren Parameterliste leer ist aber einen anderen, dann muss man in einer Unterklasse explizit einen Konstruktor angeben.

- 10) FALSCH: Es kann keine Variablen, Parameter oder Attribute vom Typ einer abstrakten Klasse geben.

Wie auch schon bei Schnittstellen kann es von abstrakten Klassen nur Variablen, Parameter und Attribute aber keine Exemplare geben.

Exemplare gibt es nur von Klassen die vollständig implementiert sind.

Aufgabe 2 (Polymorphie in der Praxis)

Durchsuchen Sie JHotDraw mit Ihrem neuen Verständnis zum Thema Vererbung nach folgenden Punkten:

- *Finden Sie drei Methoden die überladen (overloaded) sind. Worin unterscheiden sich die verschiedenen Versionen?*
 1. In der Klasse `AbstractFigure` gibt es zwei überladene Versionen der Methode `displayBox`. Eine wird mit zwei Punkten und eine mit einem Rechteck aufgerufen. Die Funktion der zweiten ist nur, das

Rechteck in zwei Punkte zu zerlegen und diese an die erste Methode zu übergeben.

2. Die Klasse CompositeFigure hat drei überladene Versionen der Methode findFigure. Bei der ersten und zweiten Version kann man eine Figur suchen, welche einen Punkt enthält bzw. ein Rechteck schneidet. Die dritte Version ist dann eine Erweiterung der zweiten und bietet an, dass eine bestimmte Figur aus der Suche ausgeschlossen wird. Interessanterweise gibt es dann sogar eine vierte Version, welche an einem Punkt sucht (wie Version eins), aber wie Version drei eine Figur ausschließen kann. Diese Version heißt dann allerdings findFigureWithout, obwohl auch eine Überladung mit findFigure möglich gewesen wäre.
3. Die Klasse DecoratorFigure hat zwei Methoden mit Namen getAttribute. Die erste erwartet eine Zeichenkette während bei der zweiten eine FigureAttributeConstant erwartet wird. Interessant an diesem Beispiel ist, dass die erste Version "deprecated" (gemissbilligt) ist. Damit wird einem Programmierer signalisiert, dass die Verwendung der zweiten Variante angeraten wird.

Zusammenfassung:

Man verwendet Überladen, um Methoden mit sehr ähnlicher oder gleicher Funktionalität zu gruppieren und dem Programmierer lästige Konvertierungen zu ersparen (Rechteck in zwei Punkte).

- *Finden Sie drei Methoden, die ersetzt (overridden) wurden. Wieso war es nötig die alte Version zu ersetzen?*
 1. Die Methode includes(Figure) in der Klasse CompositeFigure wurde erstmals in der Klasse AbstractFigure implementiert und anschließend in CompositeFigure ersetzt. Während bei abstrakten Figuren nur überprüft wird, ob die übergebene Figur gleich dem aktuellen Exemplar ist, muss bei der zusammengesetzten Figur zusätzlich geprüft werden, ob eine der Teilfiguren die übergebene Figur enthält.
 2. Die Methode mouseDown der Schnittstelle Tool wird von der Klasse AbstractTool erstmals implementiert und dann nahezu in jedem anderen Werkzeug ersetzt, um das Verhalten der auszuführenden Aktion anzupassen. Als Beispiel: Die Klasse AbstractTool speichert die Position des Klicks und merkt sich, in welcher Ansicht der Klick erfolgte. Die Methode wird dann in der Klasse ZoomTool ersetzt, damit je nach geklicktem Mausknopf die Ansicht vergrößert oder verkleinert wird. In einem anderen Werkzeug (z.B. dem SelectionTool) erfolgt eine ganz andere Ersetzung der Methode.

3. Die Methode draw der Schnittstelle Figure verhält sich ähnlich: In nahezu allen implementierenden Klassen wird die Methode ersetzt, um die repräsentierte Figur darzustellen.

Zusammenfassung:

Beim Ersetzen wird Funktionalität den spezifischen Gegebenheiten der Unterklassen angepasst. Unter Umständen wird hierbei auf die Funktionalität der Oberklasse zurückgegriffen (mittels super). Durch den "Dynamic Dispatch" der Polymorphie kann ein Benutzer durch Aufruf der allgemeinen Methode der Oberklasse einen Aufruf der speziellen Funktionalität erreichen. Fallunterscheidungen, welche der Methoden, in welcher Unterklassen aufgerufen werden muss, werden zur Laufzeit aufgelöst.

- *Kontrastieren Sie mit diesem Wissen kurz die Begriffe überladen und ersetzen.*

Ersetzen ist ein Herzstück der Vererbung und ein wichtiges Konzept dieser Vorlesung. Überladen hingegen ist eher ein Bonbon für den Programmierer, der damit ähnliche Funktionen unter einem Namen ansprechen kann. Für den Übersetzer sind die überladenen Funktionen dank unterschiedlicher Signatur leicht auseinanderzuhalten.

- *Finden Sie eine abstrakte Klasse, die sowohl abstrakte, als auch bereits implementierte Methoden besitzt.*

Die Klasse AbstractFigure hat z.B. vier abstrakte Methoden und unzählige Methoden, die bereits grundlegende Funktionalitäten zur Verfügung stellen. Ein schönes Beispiel ist:

```
/**
 * Moves the figure by the given offset.
 */
public void moveBy(int dx, int dy) {
    willChange();
    basicMoveBy(dx, dy);
    changed();
}

/**
 * Moves the figure. This is the method that subclasses override. Clients
 * usually call displayBox.
 * @see #moveBy
 */
protected abstract void basicMoveBy(int dx, int dy);
```

Die Methode moveBy wird bereits in der Klasse AbstractFigure implementiert und ruft eine Methode auf, die erst in einer Unterklasse implementiert werden wird (basicMoveBy ist noch abstrakt). Gespart wird hierdurch, dass jede Unterklasse willChange() und changed() in der Implementierung von basicMoveBy aufrufen muss.

Aufgabe 3 (Vererbung)

Wichtigste Punkte der Lösung:

- Eingeführt wurden zwei abstrakte Klassen Figure und Figure2D, welche von verschiedenen Klassen gemeinsam genutzte Funktionalität aufnehmen. Die Klasse Figure übernimmt die Methode translate und einen ersten Punkt. Abstrakt bleibt in ihr die Methode draw, welche von allen konkreten Unterklassen implementiert werden muss. Die Klasse Figure2D führt zwei weitere Methoden ein, welche von allen Klassen, die eine Fläche haben, implementiert werden soll. Von dieser erben alle anderen Klassen, außer der Klasse Circle, welche als Spezialisierung/Untertyp von Ellipse gebildet wird.
- Es mag verwundern, dass ein Kreis, welcher das Feld r2 der Klasse Ellipse eigentlich nicht in Anspruch nimmt, als Spezialisierung dieser formuliert ist. Man greife dann aber wieder zur "IST-EIN"-Hilfe: Ein "Kreis IST EINE Ellipse", umgekehrtes gilt nicht.
- Die Stärke der Polymorphie werden in zwei Punkten sichtbar:
 1. Die Methode translate ist zentral gekapselt und wird in jeder Klasse unverändert wiederverwendet.
 2. Durch die Verwendung der dynamischen Bindung kann in der Klasse Main der Aufruf der Zeichenmethode wesentlich vereinfacht werden. Obwohl man auch in der alten Version ohne mehrere Datenstrukturen ausgekommen wäre, hätte man dann eine Fallunterscheidung in Kauf genommen:

```
for (Object o : vector){  
    if (o instanceof Circle) ((Circle)o).draw(g);  
    if (o instanceof Rectangle) ((Rectangle)o).draw(g);  
    ...  
}
```

Sollten Sie je in Sprachen ohne Objektorientierung arbeiten (z.B. C), dann werden Ihnen diese Arten von Fallunterscheidungen öfter begegnen.

Lösung: <http://www.inf.fu-berlin.de/lehre/WS05/ALP3/loesungen/U5.A3.jar>

Aufgabe 4 (GGT)

Idee bei der Lösung ist, dass man in einer abgeleiteten Klasse B die Methode internalGGT mit der richtigen Implementierung eines GGTs überschreibt. Im Testfall muss nur der Konstruktor für eine gültige Lösung geändert werden. Illustriert werden sollte damit, welche Änderungen man mit Hilfe der Polymorphie an existierendem Code vornehmen kann.

Man beachte jedoch, dass die Lösung nur möglich ist, da der Sichtbarkeitsmodifizierer protected gewählt wurde. Wäre die Methode internalGGT private gewesen, dann hätte man sie nicht ersetzen können.

```
class B extends A {  
  
    protected int internalGgt(int a, int b) {  
        int c = a % b;  
        if (c == 0)  
            return b;  
        return internalGgt(b, c);  
    }  
}  
  
public void testGGT() {  
    A a = new B();  
  
    assertEquals(150, a.ggt(150));  
    assertEquals(150, a.ggt(300));  
    assertEquals(50, a.ggt(350));  
    assertEquals(1, a.ggt(7));  
}
```

Lösung: <http://www.inf.fu-berlin.de/lehre/WS05/ALP3/loesungen/U5.A4.jar>