TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Computer Vision Group
Einführung in die Informatik I für Ingenieure (MSE)
Prof. Dr. Daniel Cremers, Mohamed Souiai

WS 2012
Übungsblatt 12
25.01.2013

Aufgabe 12.1 (Ü) Polymorphie verstehen

Diskutieren Sie, was in dem folgenden Programm passiert:

```
1  class A {
2      public void m(){ System.out.println("m() in A"); }
3  }
4  class B extends A {
5      public void m(){ System.out.println("m() in B");  }
6  }
7  class C {
8      public void m(){ System.out.println("m() in C");  }
9  }
10 public class Polymorphie{
11     public static void main(String[] args){
12         A a = new A();
13         a.m();
14         call(a);
15
16         B b = new B();
17         b.m();
18         call(b);
19
20         C c = new C();
21         c.m();
22         call(c);
23
24         a = c;
25
26         a = b;
27         a.m();
28         call(a);
29
30         specCall(b);
31         specCall(a);
32     }
33     public static void call(A a){
34         a.m();
35     }
36     public static void specCall(B b){
37         b.m();
38     }
39 }
```

Lösungsvorschlag 12.1

Erklärungsmuster

Versuchen Sie, die Studenten an die übliche Methode heranzuführen, Typen aufzulösen bzw. zu erkennen, welche Methoden tatsächlich aufgerufen werden:

Man unterscheidet zwischen zwei Typen in Java:

a) **Statischer Typ** eines Ausdrucks („Übersetzungszeit-Typ“)
   Der statische Typ eines Ausdrucks bestimmt, ob die Typüberprüfung erfolgreich ist, bestimmt die Signatur eines Methodenaufrufs, ordnet statischen Methodenaufrufen Code zu und ist dafür verantwortlich, ob ein Compile-Fehler auftritt oder nicht.
b) **Dynamischer Typ** eines Ausdrucks („Laufzeit-Typ“)
   Der dynamische Typ eines Ausdrucks bestimmt die Semantik des Codes und entscheidet für polymorphe Aufrufe, welcher Code tatsächlich ausgeführt wird.

Während der statische Typ eines Bezeichners einmal bei seiner Deklaration festgelegt wird und sich während der Lebenszeit des Bezeichnes nicht ändert, kann der dynamische Typ einer Variablen durch Zuweisung oder Parameterübergabe weitergegeben werden und spezieller sein als ihr statischer Typ.

Das Identifizieren des dynamischen Typs von Methodenaufrufen ist wichtig, um entscheiden zu können, welcher Methodenrumpf bei einem Aufruf zur Laufzeit ausgeführt wird. Um ihn zu ermitteln, muß

a) die Signatur S des konkreten Aufrufs bestimmt werden, d.h. der Name der Methode sowie der statische Typ ihrer aktuellen Parameter.
b) der statische Typ T des Objekts o bestimmt werden, auf dass sich dieser Aufruf bezieht, d.h. entweder
   - der Typ des Ausdrucks, von dem dereferenziert wird oder
   - implizit der Typ des konkreten Objekts, in welchem der Aufruf stattfindet bei Verwendung von `this` bzw. auch bei gar keinem dereferenzierten Objekt.
c) in allen Klassen der Typhierarchie zwischen `Object` und T nach der Deklaration einer Methode m gesucht werden, deren Signatur *kompatibel* zu S ist und *am meisten spezialisiert* ist. Die Signatur $S^\star$ dieser Methode m ist die Signatur der zur Laufzeit ausgeführten Methode.
d) nun noch in den Klassen der Typhierarchie zwischen `Object` und *dem Laufzeittyp* von o nach der jüngsten Ausprägung einer Methode mit der Signatur $S^\star$ gesucht werden. Diese Methode ist diejenige, die zur Laufzeit ausgeführt wird.

Lösung

In diesem Programm kommt es zu Fehlern zur Compile-Zeit in folgenden Zeilen:

22 Fehler: `method (..) cannot be applied to given types`. Die Methode `call` in der Klasse `Polymorphie` muss als Parameter ein Objekt der Klasse `A` bekommen. Auch wenn die Klassen `A` und `C` ziemlich ähnlich aussehen, haben sie nichts mit einander zu tun.

24 Fehler: `incompatible types`. `a` ist ein Objekt der Klasse `A` und `c` ein Objekt der Klasse `C`. Da also `a` und `c` unterschiedliche Typen haben, kann `c` nicht `a` zugewiesen werden.

31 Fehler: `method (..) cannot be applied to given types`. `a` ist als Objekt der Klasse `A` deklariert. Die Methode `specCall` muss aber ein Objekt der Klasse `B` als Parameter übergeben bekommen. Dieses muss bereits als solches deklariert worden sein, ansonsten erkennt der Compiler nicht, dass der Aufruf funktionieren kann.

Kommentiert man diese Zeilen aus, passiert folgendes:

1-9 3 Klassen – jeweils mit einer Methode `m` – werden deklariert.

4 Klasse `B` ist eine Unterklasse von Klasse `A`.

10-39 "Hauptklasse" `Polymorphie`

12 Neues Objekt der Klasse `A`.

13 Methode `m` aus Klasse `A` wird aufgerufen ⟶ Ausgabe `m() in A`

14 Methode `call` aus Klasse `Polymorphie` mit Objekt der Klasse `A` wird aufgerufen (Zeilen 33-35). Dadurch wird wieder die Methode `m` aus Klasse `A` aufgerufen ⟶ Ausgabe `m() in A`

16 Neues Objekt der Klasse `B`.

17 Methode `m` aus Klasse `B` wird aufgerufen ⟶ Ausgabe `m() in B`

18 Methode `call` aus Klasse `Polymorphie` mit Objekt der Klasse `B` wird aufgerufen (Zeilen 33-35). Dies ist möglich, da `b` gleichzeitig auch ein Objekt der Klasse `A` ist (Vererbung), `B` ist eine Unterklasse von `A`. Trotzdem wird wieder die Methode `m` aus Klasse `B` (nicht aus Klasse `A`) aufgerufen ⟶ Ausgabe `m() in B`

20 Neues Objekt der Klasse `C`.

21 Methode `m` aus Klasse `C` wird aufgerufen ⟶ Ausgabe `m() in C`

22 Auskommentiert wegen Typfehlers.

24 Auskommentiert wegen Typfehlers.

26 Da `B` eine Unterklasse von `A` ist, ist `b` auch ein Objekt vom Typ `A`. Damit kann `b` der Variable `a` vom Typ `A` zugewiesen werden. `a` ist dann eine Referenz auf das Objekt `b`.

27 `a` ist jetzt ein Objekt der Klasse `B`, da `b` ein solches ist. Methode `m` aus Klasse `B` wird aufgerufen ⟶ Ausgabe `m() in B`

28 Methode `call` aus Klasse `Polymorphie` mit Objekt der Klasse `B` wird aufgerufen. Auch hier wird wieder die Methode `m` aus Klasse `B` (nicht aus Klasse `A`) aufgerufen ⟶ Ausgabe `m() in B`

30 Methode `specCall` aus Klasse `Polymorphie` mit Objekt `b` der Klasse `B` wird aufgerufen. Auch hier wird wieder die Methode `m` aus Klasse `B` aufgerufen ⟶ Ausgabe `m() in B`

31 Auskommentiert wegen Typfehlers. Der Compiler weiß nicht, dass `a` jetzt ein Objekt der Klasse `B` ist. `a` ist als Objekt der Klasse `A` deklariert, daher passt der Aufruf `specCall(a)` nicht zur Signatur von `specCall`.

Aufgabe 12.2 (Ü) Karriere

Im Personalwesen einer Firma nehmen Personen verschiedene Rollen ein. Dabei wird zwischen *Kunden* und *Angestellten* unterschieden. Jeder Angestellte verfügt über einen eigenen Kundenstamm, bzgl. des Gehaltes existieren jedoch folgende Unterschiede:

- *Tarifliche Mitarbeiter* bekommen ein festes Gehalt, das bei ihrer Einstellung fest vereinbart wird. Sie bekommen keinen Bonus.
- *Außertarifliche Mitarbeiter* bekommen ein festes Grundgehalt zzgl. eines Bonus von 1% des Gehaltes aller Kunden, die sie betreuen.
- *Führungskräfte* bekommen ein festes Grundgehalt zzgl. eines Bonus von 10% des Gehaltes aller Kunden, die von ihnen oder ihren untergebenen Angestellten betreut werden.

Angestellte können befördert werden, wobei sie dann jeweils vom tariflichen Mitarbeiter zum außertariflichen sowie vom außertariflichen zur Führungskraft aufsteigen.

Schreiben Sie ein Programm für die Personalverwaltung, um die Verknüpfungen zwischen Kunden, gewöhnlichen Angestellten und Führungskräften zu verwalten. Implementieren Sie dafür die folgenden, ggf. abstrakten Klassen:

a) `Person`, wobei jede Person durch ihren Namen und ihr Gehalt bestimmt ist.
b) `Angestellter`, die von `Person` erbt und Methoden zur Berechnung des Gehalts sowie zur Beförderung des Mitarbeiters (abstrakt) bereitstellt. Sie soll außerdem eine Liste der Kunden des Mitarbeiters verwalten und eine Methode besitzen, um beliebige Personen als Kunden aufzunehmen.
c) `Tariflicher`, `AusserTariflicher` sowie `Fuehrungskraft`. Führungskräfte verfügen über eine Liste untergebener Angestellter und einer entsprechenden Methode, um neue Angestellte zu dieser Liste hinzuzufügen.
Zusätzlich zur korrekten Gehaltsberechnung sollen diese drei Klassen geeignete Implementierung zur Beförderung besitzen.
d) `Personalverwaltung`, in deren `main`-Methode Sie ihre Klassen möglichst umfangreich testen.

Hinweis: Nutzen Sie bei Ihrer Implementierung Vererbung möglichst gut aus. Zur Umsetzung des Listen bedienen Sie sich der entsprechenden Vorlage.

Lösungsvorschlag 12.2

```
public class Person {

    protected String name;
    protected double gehalt;

    protected Person(String name, double gehalt) {
        this.name = name;
        this.gehalt = gehalt;
    }

    @Override
    public String toString() {
        return name;
    }

}
```

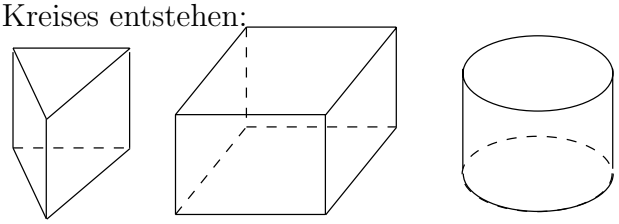
```
public abstract class Angestellter extends Person {

    protected PersonenListe kunden;
    protected double bonus;

    public Angestellter(String name, double gehalt, double bonus) {
        super(name, gehalt);
        this.bonus = bonus;
    }

    public void kundenHinzufuegen(Person kunde) {
        if (kunden == null)
            kunden = new PersonenListe(kunde);
        else
            kunden = kunden.add(kunde);
    }

    protected double summeKundenGehaelter() {
        double result = 0;
        for (PersonenListe kl = kunden; kl != null; kl = kl.getNext())
            result += kl.getPerson().gehalt;
        return result;
    }

    public double berechneGehalt() {
        return gehalt + summeKundenGehaelter() * bonus;
    }

    public abstract Angestellter befoerdern();

}

public class Tariflicher extends Angestellter {

    public Tariflicher(String name, double gehalt) {
        super(name, gehalt, 0.0);
    }

    @Override
    public AusserTariflicher befoerdern() {
        AusserTariflicher at = new AusserTariflicher(name, gehalt);
        at.kunden = kunden;
        return at;
    }

}

public class AusserTariflicher extends Angestellter {

    public AusserTariflicher(String name, double gehalt) {
        super(name, gehalt, 0.01);
    }

    @Override
    public Fuehrungskraft befoerdern() {
        Fuehrungskraft fk = new Fuehrungskraft(name, gehalt);
        fk.kunden = kunden;
```

```
        return fk;
    }

}

public class Fuehrungskraft extends Angestellter {

    protected PersonenListe untergebene;

    public Fuehrungskraft(String name, double gehalt) {
        super(name, gehalt, 0.1);
    }

    public void untergebenenHinzufuegen(Angestellter angestellter) {
        if (untergebene == null)
            untergebene = new PersonenListe(angestellter);
        else
            untergebene = untergebene.add(angestellter);
    }

    @Override
    public Fuehrungskraft befoerdern() {
        Fuehrungskraft fk = new Fuehrungskraft(name, gehalt * 1.05);
        fk.untergebene = untergebene;
        fk.kunden = kunden;
        return fk;
    }

    @Override
    public double berechneGehalt() {
        double result = super.berechneGehalt();
        for (PersonenListe ul = untergebene; ul != null; ul = ul.
           getNext()) {
            Angestellter a = (Angestellter) ul.getPerson();
            result += a.summeKundenGehaelter() * bonus;
        }
        return result;
    }

}

public class Personalverwaltung {

    public static void main(String[] args) {
        Tariflicher t = new Tariflicher("T",100);
        t.kundenHinzufuegen(new Person("Kunde1",343.0));

        AusserTariflicher at = new AusserTariflicher("AT",100.0);
        at.kundenHinzufuegen(new Person("Kunde2",200.0));
        at.kundenHinzufuegen(new Person("Kunde3",200.0));
        at.kundenHinzufuegen(new Person("Kunde4",200.0));
        at.kundenHinzufuegen(new Person("Kunde5",200.0));

        Fuehrungskraft chef = new Fuehrungskraft("El␣Capitan",0);
        chef.untergebenenHinzufuegen(t);
        chef.untergebenenHinzufuegen(at);
```

```
        System.out.println(chef.berechneGehalt());
        at=t.befoerdern();
        System.out.println(chef.berechneGehalt());
                chef=at.befoerdern();
                System.out.println(chef.berechneGehalt());
    }

}
```

Aufgabe 12.3 (H) Prismen

Prismen oder *Zylinder* sind geometrische Körper, die durch Parallelverschiebung ihrer Grundfläche im Raum entstehen. Die folgende Abbildung zeigt als Beispiele für Prismen ein Dreiecksprisma, einen Quader und einen Zylinder, die durch Verschiebung eines Dreiecks, eines Rechtecks bzw. eines Kreises entstehen:



Prismen sind durch ihre Höhe und ihre jeweilige Grundfläche bestimmt. Als Grundflächen in Frage kommen (zum Beispiel):

- *regelmässige n-Ecke*; bestimmt durch die Anzahl der Ecken und die Seitenlänge
- *Kreise*; bestimmt durch ihren Radius
- *Rechtecke*; bestimmt durch Breite und Länge

An Operationen muss eine Grundfläche, die Berechnung von Umfang und Flächeninhalt zur Verfügung stellen; ein Prisma die Berechnung von Oberfläche und Volumen.

In dieser Aufgabe soll eine Klassenhierarchie für diese geometrischen Körper in Java implementiert werden.

a) Erstellen Sie die Klasse `Grundflaeche.java`, als Oberklasse für alle Grundflächen. Legen sie Basisversionen der Methoden `umfang()` und `flaeche()`, sowie eine geeignete `toString()`-Methode an.
b) Erstellen Sie Klassen `Kreis.java`,`Rechteck.java` und `NEck.java` zur Repräsentation von Kreisen, Rechtecken und n-Ecken. Diese Klassen erweitern `Grundflaeche` um die benötigten Werte und überschreiben die Methoden zur Flächen- und Umfangberechnung. Ergänzen sie die `toString()`-Methode sinnvoll.
c) Erstellen sie die Klasse `Prisma.java` mit den benötigten Feldern und den Methoden `volumen()` und `oberflaeche()`, sowie einer geeigneten `toString()`-Methode.
d) Ergänzen Sie alle Grundflächen um die Methode `istQuadrat()`, die zurückgibt, ob es sich bei der aktuellen Instanz um ein Quadrat handelt. Z.B. falls bei einem Rechteck Länge und Breite gleich sind.
e) Erstellen Sie die Klasse `Quadrat.java`, welche durch ihre Seitenlänge eindeutig bestimmt ist, als weitere Unterklasse von `Grundflaeche` und ergänzen sie alle Grundflächen um einem Methode `zuQuadrat()`, die, falls es sich um ein Quadrat (auch passende Rechtecke und n-Ecke) handelt, ein `Quadrat`-Object mit der entsprechenden Seitenlänge zurückgibt. Handelt es sich nicht um ein Quadrat, dann wird `null` zurückgegeben.
f) Ergänzen sie die Klasse `Prisma` um eine Methode `istWuerfel()`, die zurückgibt, ob es sich bei dem aktuellen Prisma um einen Würfel handelt.

Hinweise:

- Fassen Sie gleichartige Attribute und Methoden in einer geeigneten Oberklasse zusammen.
- Verwenden sie Getter- und ggf. Setter-Methoden.
- Greifen sie falls möglcih auf bereits implementierte Methoden zurück.
- Die Fläche eines regelmäßigen n-Ecks mit Seitenlänge a ist $\frac{n*a^2}{4*\tan(\frac{\pi}{n})}$.
- Die Java-Klasse `Math` stellt in der Klassenvariablen `Math.PI` einen Wert für π sowie in der Klassenmethode `Math.tan()` die Berechnung des Tangens zur Verfügung.
- Testen sie ihre Implementierun mit einer geeigneten Test-Klasse.

Lösungsvorschlag 12.3

```
public abstract class Grundflaeche {

    public abstract double umfang();

    public abstract double flaeche();

    public boolean istQuadrat(){
        return false;
    }

    public Quadrat zuQuadrat(){
        return null;
    }

    @Override
    public String toString() {
        return "{Umfang: " + umfang() + "; Flaeche: "+ flaeche() + "; "
           Quadrat?: " + istQuadrat() + '}';
    }
}

public class Kreis extends Grundflaeche{
    private double radius;

    @Override
    public double umfang() {
        return 2*Math.PI*radius;
    }

    @Override
    public double flaeche() {
        return Math.PI*Math.pow(radius,2);
    }

    public Kreis(double radius){
        this.radius = radius;
    }

    public double getRadius(){
        return radius;
    }

    @Override
    public String toString() {
```

```
        return "Kreis{" + "radius=" + radius + super.toString() +'}';
    }


}

public class Rechteck extends Grundflaeche{
    private double breite;
    private double laenge;

    public double getBreite() {
        return breite;
    }

    public double getLaenge() {
        return laenge;
    }

    @Override
    public double umfang() {
        return 2*breite+2*laenge;
    }

    @Override
    public double flaeche() {
        return breite*laenge;
    }

    public Rechteck(double breite, double laenge){
        this.breite = breite;
        this.laenge = laenge;
    }

    @Override
     public boolean istQuadrat(){
        return laenge==breite;
    }
    @Override
     public Quadrat zuQuadrat(){
        if (istQuadrat()){
            return new Quadrat(breite);
        } else {
            return null;
        }
     }

    @Override
    public String toString() {
        return "Rechteck{" + "breite=" + breite + ", laenge=" + laenge
           + super.toString() + '}';
    }



}

public class NEck extends Grundflaeche{
```

```
    private int n;
    private double laenge;

    public double getLaenge() {
        return laenge;
    }

    public int getN() {
        return n;
    }

    public NEck(int n, double laenge) {
        this.n = n;
        this.laenge = laenge;
    }

    @Override
    public double umfang() {
        return n*laenge;
    }

    @Override
    public double flaeche() {
       return n*Math.pow(laenge,2)/4*Math.tan(Math.PI/n);
    }

    @Override
    public boolean istQuadrat(){
        return n==4;
    }

    @Override
    public Quadrat zuQuadrat(){
        if (istQuadrat()){
            return new Quadrat(laenge);
        } else {
            return null;
        }
    }

    @Override
    public String toString() {
        return "NEck{" + "n=" + n + ", laenge=" + laenge+ super.
           toString()  + '}';
    }

}

public class Quadrat extends Grundflaeche{
    private double laenge;

    public double getLaenge() {
        return laenge;
    }

    @Override
    public double umfang() {
```

```
        return 4*laenge;
    }

    @Override
    public double flaeche() {
        return laenge*laenge;
    }

    @Override
    public boolean istQuadrat(){
        return true;
    }

    @Override
    public Quadrat zuQuadrat(){
        return this;
    }

    public Quadrat(double laenge) {
        this.laenge = laenge;
    }

    @Override
    public String toString() {
        return "Quadrat{" + "laenge=" + laenge+ super.toString()  + '}'
            ;
    }

}

public class Prisma {
    private Grundflaeche basis;
    private double hoehe;

    public Grundflaeche getBasis() {
        return basis;
    }

    public double getHoehe() {
        return hoehe;
    }

    public Prisma(Grundflaeche basis, double hoehe) {
        this.basis = basis;
        this.hoehe = hoehe;
    }

    public double oberflaeche(){
        return hoehe*basis.umfang()+2*basis.flaeche();
    }

    public double volumen(){
        return hoehe*basis.flaeche();
    }

    public boolean istWuerfel(){
        if (basis.istQuadrat()){
```

```
            return (Math.abs(basis.zuQuadrat().getLaenge() - hoehe)) <
                Float.MIN_NORMAL;
        } else {
            return false;
        }
    }

    @Override
    public String toString() {
        return "Prisma{" + "basis=␣" + basis + ",␣hoehe=␣" + hoehe + ",
            ␣Volumen=␣" +volumen() +
                ",␣Oberflaeche=␣" +oberflaeche() + ",␣Wuerfel?=␣" +
                    istWuerfel() + '}';
    }

}

public class Test {
   public static void main (String[] args) {
      Grundflaeche g1 = new Kreis(7);
      Grundflaeche g2 = new Rechteck(2,3);
      Grundflaeche g3 = new Rechteck(4, 4);
      Grundflaeche g4 = new NEck(7, 3);
      Grundflaeche g5 = new NEck(4, 5);
      System.out.println(g1);
      System.out.println(g2);
      System.out.println(g3);
      System.out.println(g4);
      System.out.println(g5);
      Prisma p1 = new Prisma(g1, 1);
      Prisma p2 = new Prisma(g2, 2);
      Prisma p3 = new Prisma(g3, 3);
      Prisma p4 = new Prisma(g4, 4);
      Prisma p5 = new Prisma(g5, 5);
      System.out.println(p1);
      System.out.println(p2);
      System.out.println(p3);
      System.out.println(p4);
      System.out.println(p5);
   }
}
```