

Klassen und ihre Beziehungen III: Mehrfache Vererbung, Rollen, Schnittstellen und Pakete

Martin Wirsing

in Zusammenarbeit mit
Matthias Hölzl, Piotr Kosciuzenko, Dirk Pattinson

05/03

Informatik II, SS 03

2

Ziele

- Den Begriff der einfachen und mehrfachen Vererbung verstehen
- Verstehen, wann Vererbung eingesetzt wird
- Schnittstellendeklarationen kennen lernen
- Pakete zur Strukturierung von Programmsystemen kennen lernen

M. Wirsing: Klassen und Ihre Beziehungen: Assoziation, Aggregation, Vererbung und Abhängigkeit

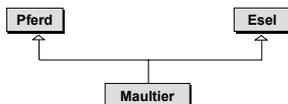
Informatik II, SS 03

3

Einfache und mehrfache Vererbung

- Man spricht von **einfacher Vererbung**, wenn jede Klasse höchstens eine direkte Superklasse besitzt.
- Kann eine Klasse mehrere direkte Superklassen haben, spricht man von **mehrfacher Vererbung**.

Beispiel: mehrfache Vererbung



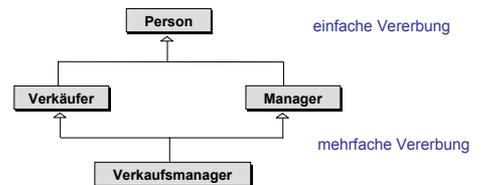
M. Wirsing: Klassen und Ihre Beziehungen: Assoziation, Aggregation, Vererbung und Abhängigkeit

Informatik II, SS 03

4

Einfache und mehrfache Vererbung

Beispiel: einfache und mehrfache Vererbung

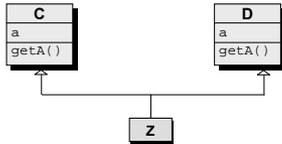


Bemerkung: Java unterstützt nur **einfache Vererbung**.

M. Wirsing: Klassen und Ihre Beziehungen: Assoziation, Aggregation, Vererbung und Abhängigkeit

Einfache und mehrfache Vererbung

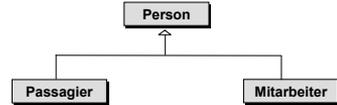
Problem bei mehrfacher Vererbung: mögliche Mehrdeutigkeiten beim Verwenden von Attributen und Methoden:



Welches Attribut `a` und welche Methode `getA()` sollte `z` erben und, wenn jeweils beide geerbt werden, welche der Methoden wird beim Aufruf von `getA()` ausgewählt?

Einsatz von Vererbung: Vererbung vs Rollen

Die Vererbungsbeziehung muß mit Vorsicht eingesetzt werden. Betrachten Sie folgendes Beispiel



Was passiert, wenn ein Mitarbeiter als Passagier auftritt?

Einsatz von Vererbung: Vererbung vs Rollen

In Java kann ein Objekt nicht seine Klasse wechseln. Wird also das Mitarbeiter-Objekt gelöscht und ein neues Passagier-Objekt erzeugt?

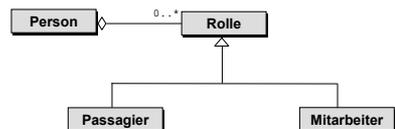
Oder ist Passagier-Mitarbeiter ein Objekt einer gemeinsamen Subklasse von Passagier und Mitarbeiter?

Dies ist in Java **nicht** möglich, da es **keine mehrfache Vererbung** gibt. Besser ist es, das Rollenmuster zu benutzen.

Ein Muster ist eine schematische Darstellung von Entwurfslösungen

Das Rollenmuster

Das **Rollenmuster** erweitert die Verantwortlichkeit eines Objekts durch Delegation von Arbeit an ein anderes Objekt.



Passagier und Mitarbeiter sind die Arten von Rollen, die eine Person besitzt.

Einsatz von Vererbung: Vererbung vs Rollenmuster

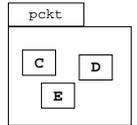
- Das **Rollenmuster** ist gut, wenn ein Objekt seine **Rolle wechseln** kann oder **mehrere Rollen** besitzt.
- Vererbung** ist gut, wenn gilt:
 - „**ist-Spezialisierung-von**“, nicht „Rolle-gespielt-bei“
Beispiel: Passagier und Mitarbeiter sind spezielle Arten **Personenrollen**
 - Das Erbenobjekt muß **niemals seine Klasse wechseln**.
Beispiel: ein Passagierobjekt bleibt immer ein Passagierobjekt
 - Die Oberklasse wird **erweitert** (und nicht redefiniert)
 - Die Unterklasse bezeichnet **spezielle Arten** von Rollen, Transaktionen, Geräten

Pakete

- Pakete dienen zur Strukturierung großer Programmsysteme. Ein Paket (Schlüsselwort `package`) fasst verwandte Klassen zusammen.
- Ein Paket hat einen Namen, der mit einem kleinen Buchstaben beginnt, und steht in einem Verzeichnis mit dem gleichen Namen.
- Alle Klassen eines Pakets `pkg1` stehen in demselben Verzeichnis, aber in unterschiedlichen Dateien. **Jede** dieser Dateien enthält die Anweisung

```
package pkg1;
```

als erste Anweisung.



Darstellung in UML:
 Paket `pkg1` mit 3
 Klassen

Beispiel

Klasse
 im Paket
 point

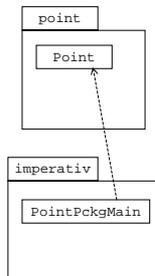
Datei: point/Point.java

```
package point;
public class Point
{
    private int x, y;
    public Point(int a, int b) { ... }
}
```

Benutzung
 einer
 Klasse von
 point

Datei: imperativ/PointPckgMain.java

```
package imperativ;
import point.Point;
public class PointPckgMain
{
    public static void main (String[] args)
    {
        Point p = new Point(10,20); ...
    }
}
```



Benutzung von Paketen

- Eine einzelne** Klasse `C` eines Pakets `pkg1` kann in einem anderen Paket benutzt werden durch die Anweisung

```
import pkg1.C;
```
- Alle Klassen** eines Pakets `pkg1` werden benutzt durch

```
import pkg1.*;
```
- Klassen **unterschiedlicher Pakete** `pkg1` werden benutzt durch

```
import pkg11.C;
import pkg2.*;
import pkg3.D; ...
```

Sichtbarkeitsregeln für Pakete

- **public** es kann von **überall** darauf zugegriffen werden.
- **protected** es kann von jeder Klasse **innerhalb dieses Pakets und von Unterklassen außerhalb des Pakets** zugegriffen werden.
- **private** es kann nur von **innerhalb der Klasse** zugegriffen werden.
- **(keine Beschreibung)** es kann von jeder Klasse **innerhalb dieses Pakets** zugegriffen werden.

Wiederverwendbare Lösungen?

- Wie kann man eine Klasse, die Dienste für einen speziellen Datentyp anbietet, für andere Datentypen wiederverwenden?
- **Beispiel:** Die Klasse `NumberSet` berechne das Maximum und den Durchschnitt einer Menge von Zahlen.
 - Wie kann man `NumberSet` verallgemeinern, um für eine Menge von Bankkonten den maximalen und den durchschnittlichen Kontostand berechnen?
 - Oder wie kann man für eine Menge von Punkten den größten Punkt und den Durchschnitt berechnen?

Beispiel NumberSet

```
public class NumberSet
{
    private double sum;
    private double maximum;
    private int count;

    public NumberSet()
    {
        sum = 0; count = 0; maximum = 0;
    }

    public void add(double x)
    {
        sum = sum + x;
        if (count == 0 || maximum < x) maximum = x;
        count++;
    }

    public double getMaximum()
    {
        return maximum;
    }
}
```

Konstruiert leere Zahlenmenge

Fügt x zu der Zahlenmenge hinzu

Berechnet Maximum

Versuch: Modifikation für BankAccount-Objekte

```
public class AccountSet //Modifikation von NumberSet für BankAccount Objekte
{
    private double sum;
    private BankAccount maximum;
    private int count;

    ...

    public void add(BankAccount x)
    {
        sum = sum + x.getBalance();
        if (count == 0 || maximum.getBalance() < x.getBalance())
            maximum = x;
        count++;
    }

    public BankAccount getMaximum()
    {
        return maximum;
    }
}
```

Fügt x zu den Bankkonten hinzu

Berechnet Maximum

Versuch: Modifikation für Point-Objekte

```
public class PointSet //Modifikation von NumberSet für Point Objekte
{
    private double sum;
    private Point maximum;
    private int count;
    ...
    public void add(Point p)
    {
        sum = sum + p.getDistFromZero() ;
        if (count == 0
            || maximum.getDistFromZero() < p.getDistFromZero())
            maximum = p;
        count++;
    }
    public Point getMaximum()
    {
        return maximum;
    }
}
```

Berechnet Abstand $\sqrt{x^2+y^2}$ vom Ursprung

Berechnet Maximum

Besser: Verwende Schnittstelle

```
public interface Measurable
{
    double getMeasure();
}

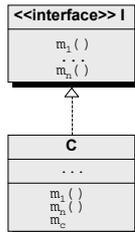
public class DataSet
{
    private double sum;
    private Measurable maximum;
    private int count;
    ...
    public void add(Measurable p)
    {
        sum = sum + p.getMeasure() ;
        ...
    }
}
```

Allgemeine Schnittstelle für messbare Objekte

Allgemeine Klasse, die Measurable verwendet

Schnittstellen

- Eine **Schnittstelle** in Java (Schlüsselwort „interface“) deklariert eine Menge von Methoden (ohne Angabe eines Rumpfs) und Konstanten (**aber** keine Attribute). Man nennt eine Methode ohne Rumpf „abstrakte Methode“. Im Gegensatz zu Klassen ist Mehrfachvererbung erlaubt, d.h. eine Schnittstelle kann Erbe mehrerer Schnittstellen sein.
- Eine Klasse **c** **implementiert** eine Schnittstelle **I**, wenn alle Methoden der Schnittstelle in **c** mit ihrer exakten Funktionalität implementiert werden, und zwar durch „öffentliche“ Methoden.

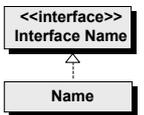


Schnittstellen

Schema:

```
public interface InterfaceName
{
    Konstanten
    MethodenSignaturen // Methodenköpfe fuer „public“ Methoden
}

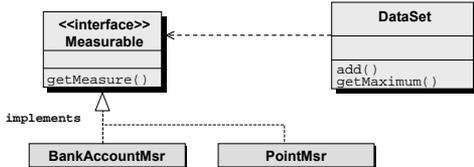
class Name implements InterfaceName
{
    Attribute
    Methoden // mindestens eine konkrete Methode fuer
              // jede abstrakte Methode aus InterfaceName
}
```



Schnittstellen

Beispiel: (1) Wiederverwendbare Lösung für das „Mengenproblem“

DataSet verwendet die Schnittstelle Measure, die von BankAccount und Point implementiert wird.



getMeasure() **muß** in BankAccountMsr und PointMsr implementiert werden

Implementierungen der Schnittstelle Measure

```

public class BankAccountMsr extends BankAccount implements Measurable
{
    public double getMeasure()
    {
        return getBalance();
    }
}
    
```

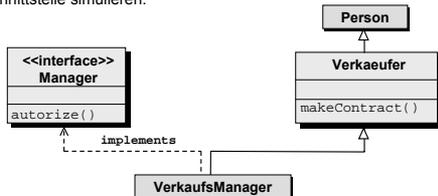
```

public class PointMsr extends Point implements Measurable
{
    public double getMeasure()
    {
        return Math.sqrt(getX()*getX() + getY()*getY());
    }
}
    
```

Schnittstellen

Beispiel: (2) Simulation mehrfacher Vererbung

Die mehrfache Vererbung des Verkaufsmanagers läßt sich in Java durch eine Schnittstelle simulieren:



Hier muß authorize() in VerkaufsManager implementiert werden

Schnittstellen

Beispiel: (3) Noch einmal wiederverwendbarer Entwurf

In java.lang gibt es eine Schnittstelle Comparable zum Vergleich von Objekten:

```

public interface Comparable
{
    int compareTo(Object other); // keine Implementierung
                                // Resultat 0, wenn beide Objekte
                                // „gleich“,
                                // Resultat >0, wenn this „größer“,
                                // Resultat <0, wenn other „größer“
}
    
```

Schnittstellen

Man kann SavingsAccount um eine compareTo-Operation erweitern:

```
public class SavingsAccountCmp extends SavingsAccount implements
Comparable
{
    ...
    public int compareTo(Object other)
    { SavingsAccount otherAccount = (SavingsAccount)other;
      if (interestRate < otherAccount.interestRate)
          return -1;
      else if (interestRate > otherAccount.interestRate)
          return 1;
      else
          return 0;
    }
    ...
}
```

M. Wirsing: Klassen und Ihre Beziehungen: Assoziation, Aggregation, Vererbung und Abhängigkeit

Verwendung von Schnittstellen

- Wenn eine Klasse ein Interface implementiert, kann man ein Objekt dieser Klasse in ein Objekt der Schnittstelle konvertieren:


```
SavingsAccount sparkonto = new SavingsAccount(5);
Comparable erster = sparkonto;
```
- Andere Klassen können mit dem Comparable-Objekt `erster` arbeiten, ohne die Implementierung zu kennen. (Natürlich können diese Klassen nur die Methoden von Comparable verwenden).

- Man kann aber nie eine Schnittstelle konvertieren:

```
Comparable second; // Ok
second = new Comparable(); // Fehler!
```

M. Wirsing: Klassen und Ihre Beziehungen: Assoziation, Aggregation, Vererbung und Abhängigkeit

Verwendung von Schnittstellen

- Konstanten in Schnittstellen sind automatisch „public final“.

Beispiel:

```
public interface ComparableConstants
{
    boolean IST_GROESSER = true;
    boolean IST_KLEINER = false;
}
```

- Die in Schnittstellen deklarierten Methoden sind **public!**

```
public interface I
{
    void m();
    ...
}
```

Jede Implementierung von I muss m als public deklarieren:

```
public class C implements I
{
    public void m();
    ...
}
```

M. Wirsing: Klassen und Ihre Beziehungen: Assoziation, Aggregation, Vererbung und Abhängigkeit

Zusammenfassung

- **Mehrfache Vererbung** kann in Java mit Hilfe von Schnittstellen („interface“) simuliert werden.
- Die **Vererbungsbeziehung muß mit Vorsicht eingesetzt** werden. Die Vererbung ist angebracht bei einer Spezialisierung. Falls Objekte verschiedene Rollen spielen können, ist es besser, Komposition oder Assoziation zu verwenden.
- Eine **Schnittstelle** deklariert abstrakte Methoden (Methodenköpfe) und Konstanten. Eine Klasse **implementiert eine Schnittstelle**, wenn sie alle Methoden der Schnittstelle implementiert. Schnittstellen können mehrfach von anderen Schnittstellen erben.
- Pakete dienen zur Strukturierung großer Programmsysteme; sie fassen verwandte Klassen zusammen.

M. Wirsing: Klassen und Ihre Beziehungen: Assoziation, Aggregation, Vererbung und Abhängigkeit