

Vererbung

Martin Wirsing

in Zusammenarbeit mit
Michael Barth, Philipp Meier und Gefei Zhang

12/04

Ziele

- Den Begriff der einfachen Vererbung verstehen
- Vererbung und Redefinition von Oberklassenmethoden verstehen
- Vererbungspolymorphie verstehen
- Die Klasse `Object` kennenlernen

M. Wirsing: Vererbung

Vererbung

Klasse D ist **Erbe** einer Klasse C, falls D alle Attribute und Methoden von C erbt,

▪ in UML:



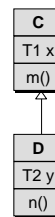
▪ in Java: `class D extends C`

d.h D besitzt alle Methoden und Attribute von C und von allen Oberklassen von C.

Man nennt C auch allgemeiner als D bzw. D spezieller als C .

M. Wirsing: Vererbung

Vererbung



- Attribute von D = $\{y\} \cup$ Attribute von C
- Methoden von D = $\{n()\} \cup$ Methoden von C

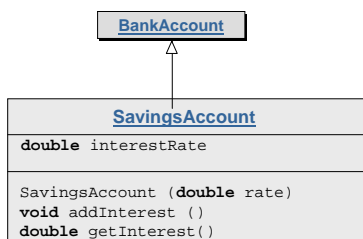
▪ Folgerung: Die Vererbungsbeziehung ist transitiv: Wenn C von B erbt, dann besitzt D auch alle Attribute und Methoden von B

M. Wirsing: Vererbung

Vererbung

Beispiel: Sparkonto

Ein Sparkonto ist ein Bankkonto, bei dem Zinsen gezahlt werden:



M. Wirsing: Vererbung

BankAccount in Java

```

public class BankAccount
{
    private double balance;

    public BankAccount()
    {
        balance = 0.0;
    }

    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }

    public void deposit(double amount)
    {
        balance = balance + amount;
    }

    public void withdraw(double amount)
    {
        balance = balance - amount;
    }

    public double getBalance()
    {
        return balance;
    }

    public void transferTo(BankAccount other, double amount)
    {
        withdraw(amount);
        other.deposit(amount);
    }
}
  
```

M. Wirsing: Vererbung

Implementierung in Java

```
public class SavingsAccount extends BankAccount
{
    private double interestRate;

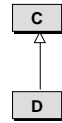
    public SavingsAccount(double rate)
    {
        super(0.0);
        interestRate = rate;
    }

    public void addInterest()
    {
        double interest =
            getBalance() * interestRate/100;
            // auf das Attribut balance kann hier
            // nicht zugegriffen werden
        deposit(interest); // geerbte Methode
    }
}
```

Zugriff auf
Konstruktor der
Oberklasse
siehe später

Vererbung

Ist D ist Erbe von C, so gilt:



- man kann von D aus **nicht direkt auf die privaten Attribute** von C zugreifen, sondern nur mittels nichtprivater (geerbter) Zugriffsmethoden von C.
- Eine Variable der Klasse D kann jede nicht-private Methode von C aufrufen;
- Einer Variablen der Klasse C kann man ein Objekt eines Nachfahren zuweisen.

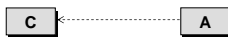
Beispiel: D d = ...; C c = d;

- Umgekehrt kann man einer Variablen vom Typ D **KEIN** Objekt einer Vorfahrenklasse zuweisen. Beispiel: D ~~d~~ = c; //falsch!

Abhängigkeitsrelation (Verwendungsrelation, engl. dependency)

Die Klasse **A** ist **abhängig von der Klasse C**, wenn A Elemente der Klasse C (i.a. Methoden) benutzt,

UML



Beispiel: In unserem Beispiel erhalten wir



SavingsAccountTest in Java

Beispiel:

```
public class SavingsAccountTest
{
    public static void main(String[] args)
    {
        SavingsAccount sparKonto = new SavingsAccount(5);
        BankAccount kontrol = sparKonto; //ok Sparkonto vom
        //spezielleren Typ
        kontrol.deposit(1000); // ok
        // kontrol.addInterest(); // nicht ok, da kontrol
        // nicht den Typ einer
        // Subklasse hat
        (SavingsAccount)kontrol.addInterest(); // ok, wegen Typcast
        sparKonto.getBalance(); // ok, geerbte Methode
        kontrol.getBalance(); // ok
    }
}
```

Redefinition von Methoden

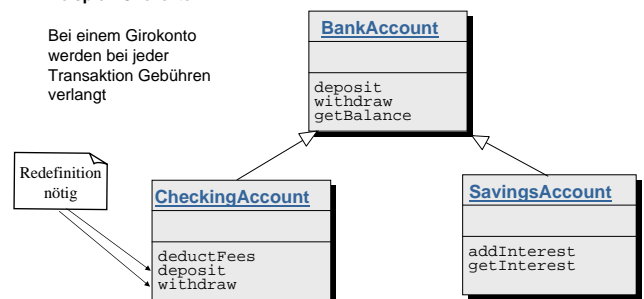
- In vielen Fällen kann man die Implementierung einer Methode *m* nicht direkt von der Superklasse übernehmen, da z.B. die neuen Attribute in der Superklasse nicht berücksichtigt werden (können). Dann ist es nötig, für die Erbenklasse eine neue Implementierung von *m* anzugeben.
- Redefinition von *m*
 - in UML: Methodenkopf von *m* wird in der Erbenklasse noch einmal angegeben;
 - Java: neue Implementierung für *m* im Erben

Bemerkung: Bei der Redefinition wird die alte Methode nicht überschrieben; man kann auf sie mit der speziellen Variable „*super*“ zugreifen. Genauer gesagt, greift man mit `super.m()` auf die nächste Methodenimplementierung in der Vererbungshierarchie zu.

Redefinition von Methoden und Konstruktoren

Beispiel: Girokonto

Bei einem Girokonto werden bei jeder Transaktion Gebühren verlangt



Redefinition von Methoden

```
public class CheckingAccount extends BankAccount
{
    private int transactionCount;
    public static final int FREE_TRANSACTIONS = 3;
    public static final double TRANSACTIONS_FEE = 0.3;

    public void deposit(double d)
    {
        super.deposit(d); // Aufruf von BankAccount::deposit
        transactionCount++;
    }
    public void withdraw(double d)
    {
        super.withdraw(d); // Aufruf von BankAccount::withdraw
        transactionCount++;
    }
}
```

Statische Konstanten,
die für jede Instanz
von CheckingAccount
gelten.

Redefinition von Methoden

Fortsetzung

```
public void deductFees()
{
    if (transactionCount > FREE_TRANSACTIONS)
    {
        double fees = TRANSACTIONS_FEE *
            (transactionCount - FREE_TRANSACTIONS);
        super.withdraw(fees);
    }
    transactionCount = 0;
}
```

Redefinition von Konstruktoren

Zugriff auf einen Konstruktor einer Superklasse:

```
super(); // parameterloser Konstruktor
bzw.
super(p1, ..., pn); // Konstruktor mit n Parametern
```

Bemerkung: Dieser Aufruf **muss** die erste Anweisung des Subklassenkonstruktors sein.

Redefinition von Konstruktoren

Beispiel: [CheckingAccount](#)

```
public CheckingAccount(double initialBalance)
{
    super(initialBalance); // muss 1. Anweisung sein
    transactionCount = 0;
}
```

Äquivalent dazu könnte man die Methode `deposit` verwenden:

```
public CheckingAccount(double initialBalance)
{
    // super(); Standardkonstruktor wird automatisch
    // aufgerufen, wenn 1. Anweisung kein Konstruktor ist.
    transactionCount = 0;
    super.deposit(initialBalance); // super.m() kann
    // überall im Rumpf
    // vorkommen
}
```

Falscher Zugriff auf super

Man kann mit `super(...)` nur auf den Konstruktor der direkten Oberklasse zugreifen, aber nicht transitiv auf Konstruktoren weiter oben liegender Klasse.

Die Compilerausgabe für diesen, im folgenden [Beispiel](#) zu findenden Fehler lautet:

```
>javaC C.java
C.java:17: cannot resolve symbol
symbol : constructor B (int,int)
location: class B
    super(a, b);
    ^
1 error
```

Redefinition von Methoden und Konstruktoren

```
class A
{
    A(int a, int b)
    {
        System.out.println(a);
        System.out.println(b);
    }
    A(){}
}

class B extends A
{
    B(int a, int b, int c)
    {
        super(a, b);
    }
}

class C extends B
{
    C(int a, int b, int c, int d)
    {
        super(a, b);
    }

    public static void main(String args[])
    {
        C c = new C(1, 2, 3, 4);
    }
}
```

Vererbungspolymorphie und dynamisches Binden

Vererbungspolymorphie

Man spricht von **Vererbungspolymorphie**, wenn eine Methode von Objekten von Subklassen aufgerufen werden kann.

Dynamisches Binden

Falls eine Methode $T_m(T_1, x_1, \dots, T_n, x_n)$ mehrere Implementierungen besitzt (die im Vererbungsbaum übereinander liegen), so wird bei einem Aufruf

$o.m(a_1, \dots, a_n)$ die „richtige“ Implementierung dynamisch

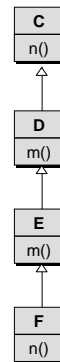
bestimmt und zwar sucht man ausgehend von der Klasse des **dynamischen Typs** von o die speziellste Methodendeklaration, auf die der Methodenaufruf anwendbar ist (genauer siehe übernächste Folie).

Man nennt dies auch **dynamische Bindung**, da der Methodenrumpf erst zur **Laufzeit** ausgewählt wird.

Dagegen wird bei **statischer Bindung** (nicht in Java) der Methodenrumpf zur **Übersetzungszeit** bestimmt.

M. Wirsing: Vererbung

Beispiel für Dynamische Bindung



```

D d = exp1;           //exp1 sei vom Typ D
d.n();               //Aufruf von n in C
F f = exp2;          //exp2 sei vom Typ F
d = f;               //Wert von d ist Instanz von F
d.m();               //Aufruf von m in E
d.n();               //Aufruf von n in F
  
```

M. Wirsing: Vererbung

Vererbungspolymorphie und dynamisches Binden

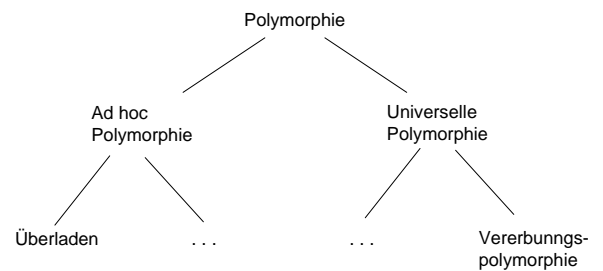
• Dynamisches Binden

Methodenaufruf in Java von $o.m(a_1, \dots, a_n)$ mit o vom Typ D und a_1, \dots, a_n vom Typ T_1, \dots, T_n .

- Ein Methodenkopf $R_m(P_1, \dots, P_n)$ der Klasse C heißt **anwendbar** auf $o.m(a_1, \dots, a_n)$, wenn C gleich D oder allgemeiner als D ist und wenn jedes P_i gleich T_i oder allgemeiner als T_i ist (für $i=1, \dots, n$).
- Für den Aufruf einer Methode wird zunächst zur **Übersetzungszeit** der **speziellste** Methodenkopf $R_m(P_1, \dots, P_n)$ bestimmt, der auf $o.m(a_1, \dots, a_n)$ anwendbar ist.
- Zur **Laufzeit** suche die speziellste Klasse C mit einer Methodendeklaration mit Namen m und Parametertypen P_1, \dots, P_n , so daß C allgemeiner oder gleich D ist. Wähle diese **Methodendeklaration** für den Aufruf.

M. Wirsing: Vererbung

Formen der Polymorphie



M. Wirsing: Vererbung

Formen der Polymorphie

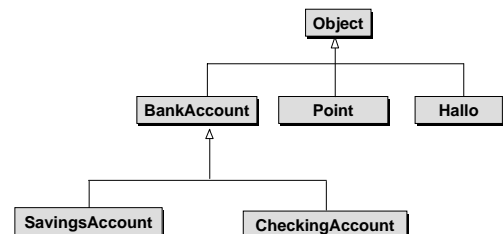
- Polymorphie:** aus dem Griechischen: „vielgestaltig“
- Überladen**
2 oder mehrere Operationen mit demselben Namen, aber verschiedener Implementierung und Semantik
Beispiel: Addition auf ganzen Zahlen und Gleitpunktzahlen
- Vererbungspolymorphie:**
Eine Methode der Klasse C kann auch von Objekten eines Subtyps von C aufgerufen werden.
Beispiel: deposit von BankAccount kann auch von Instanzen von SavingsAccount aufgerufen werden.

M. Wirsing: Vererbung

Die Klasse Object

Object ist die allgemeinste Klasse in Java. Alle Klassen sind Erben von Object.

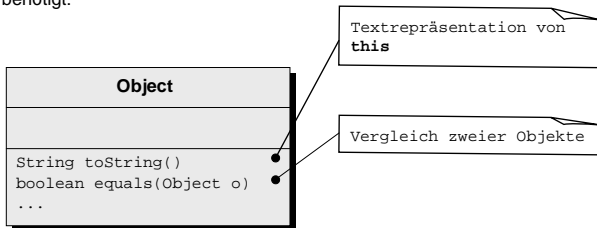
Beispiel



M. Wirsing: Vererbung

Die Klasse Object

Die Klasse `Object` besitzt u.a. die folgenden Methoden, die man häufig benötigt:



M. Wirsing: Vererbung

Die Klasse Object

- `String toString()`: Die `toString`-Methode erzeugt eine Textrepräsentation einer Klasse. Im Allgemeinen ist es nötig, für selbstdefinierte Klassen eine `toString`-Methode zu definieren.

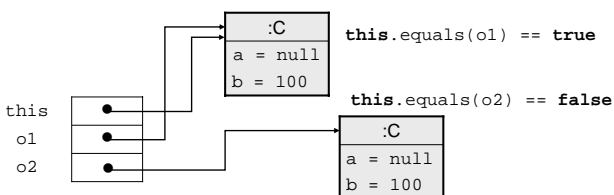
Beispiel: BankAccount

```
String toString()
{
    return „BankAccount[balance is „ + balance + „“;
}
```

M. Wirsing: Vererbung

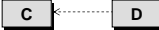

Die Klasse Object

- `boolean equals(Object o)`: `equals` vergleicht die Objektreferenzen von `this` und `o`.



M. Wirsing: Vererbung

Zusammenfassung (I)

- Die Abhängigkeitsbeziehung  gibt an, dass `D` Symbole der Klasse `C` verwendet.
- Die Vererbungsbeziehung  hat folgende Eigenschaften:

Für Variablen gilt:

- Jedes Attribut von `C` ist automatisch Attribut von `D`. Möglicherweise kann man aber auch von `D` nicht direkt darauf zugreifen!
- Ein neu definiertes Attribut von `D` ist **nicht** Attribut von `C`.
- Einer lokalen Variablen oder einem Parameter der Klasse `C` kann ein Objekt der Klasse `D` zugewiesen werden (aber nicht umgekehrt, dazu ist ein gültiger Cast nötig!)

M. Wirsing: Vererbung

Zusammenfassung (II)

Für Methoden gilt:

- Jede Methode von `C` ist automatisch eine Methode von `D` und kann daher mit Objekten von `D` aufgerufen werden (Vererbungspolymorphie). Eine Methode von `D` kann aber nicht von einer lokalen Variablen vom Typ `C` aufgerufen werden.
- Soll in einem Methodenrumpf auf die Methode der Superklasse zugegriffen werden, verwendet man spezielle Variable `super`.
- In der Subklasse `D` können Methoden redefiniert werden. Solche Methoden müssen im UML-Diagramm der Klasse `D` explizit genannt werden.

M. Wirsing: Vererbung