Programmentwicklung II

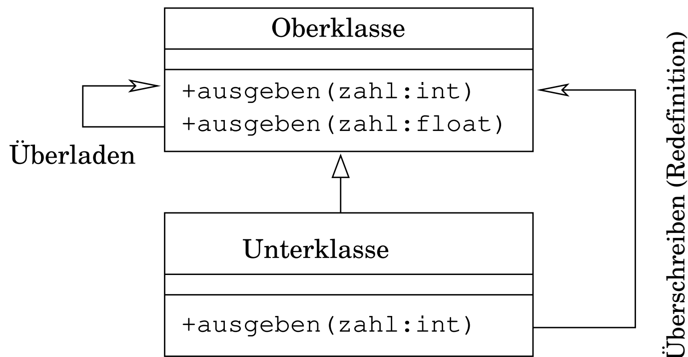
Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Überschreiben vs. Überladen



- Beim Überladen wird derselbe Funktionsname innerhalb einer Klasse mit verschiedenen Parameterschnittstellen verwendet.
- Beim Überschreiben wird eine Operation der Oberklasse mit dergleichen Signatur in der Unterklasse neu implementiert. Die *Signatur* besteht aus dem Namen der Funktion sowie der Anzahl, Reihenfolge und Typen ihrer Parameter.

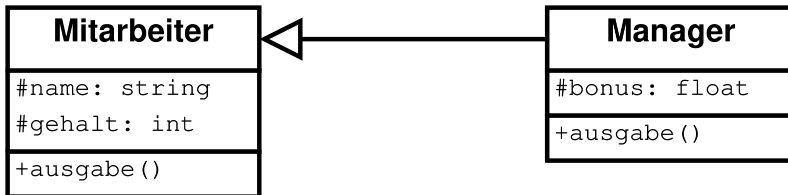
Lernkontrolle

Welche Ausgabe erzeugt das folgende Programm?

```
#include <iostream>
using namespace std;

void ausgabe(double &a) {
    cout << "1\n";
}

void ausgabe(const double &a) {
    cout << "2\n";
}

int main(void) {
    const double pi = 3.141592653589793;
    ausgabe(pi);
}
```

Im folgenden Beispiel haben wir eine Basisklasse `Mitarbeiter` und eine davon abgeleitete Klasse `Manager`.

Beide Klassen haben eine Methode `ausgabe`, die den Namen und das monatliche Gehalt des Beschäftigten ausgibt. Bei Managern ist ein Bonus zu berücksichtigen.



Wir wollen untersuchen, welche Methode `ausgabe` wann aufgerufen wird. In C++ ist dies davon abhängig,

- wie die Methoden deklariert sind und
- ob wir mit Variablen, Referenzen oder Zeigern arbeiten.

Wir definieren zunächst die Basisklasse `Mitarbeiter`:

```
class Mitarbeiter {
protected:
    string _name;
    int _gehalt;

public:
    Mitarbeiter(string name, int gehalt = 2000){
        _name = name;
        _gehalt = gehalt;
    }

    void ausgabe() {
        cout << "Mitarbeiter " << _name;
        cout << " : " << _gehalt << endl;
    }
};
```

Wir leiten dann die Klasse `Manager` von der Basisklasse ab:

```
class Manager : public Mitarbeiter {
protected:
    float _bonus;

public:
    Manager(string name, int gehalt = 2000, float bonus = 1.2)
                : Mitarbeiter(name, gehalt) {
        _bonus = bonus;
    }

    void ausgabe() {
        cout << "Manager " << _name << " : ";
        cout << (_gehalt * _bonus) << endl;
    }
};
```

statische Bindung

```
int main() {
    Mitarbeiter mi("Max Meier", 3000);
    Manager ma("Hans Wurst", 4000, 1.3);

    mi.ausgabe();
    ma.ausgabe();
}
```

Ausgabe:

```
Mitarbeiter Max Meier : 3000
Manager Hans Wurst : 5200
```

Standard in C++ ist die statische Bindung:

- Eine Methode wird zur Compile-Zeit an ein Objekt gebunden,
- eine Veränderung kann zur Laufzeit nicht vorgenommen werden.

statische Bindung

```
int main() {
    Mitarbeiter *m[2];

    m[0] = new Mitarbeiter("Max Meier", 3000);
    m[1] = new Manager("Hans Wurst", 4000, 1.3);

    for (int i = 0; i < 2; i++)
        m[i]->ausgabe();
}
```

Ausgabe:

```
    Mitarbeiter Max Meier : 3000
    Mitarbeiter Hans Wurst : 4000
```

Obwohl das zweite Objekt vom Typ `Manager` ist, wird wegen der voreingestellten statischen Bindung hier die Methode `Mitarbeiter::ausgabe` an beide Objekte gebunden.

Damit immer die zum Objekt passende Methode aufgerufen wird, also

- bei einem `Mitarbeiter`-Objekt die Mitarbeiter-Methode
- und bei einem `Manager`-Objekt die Manager-Methode,

müssen wir die Methode ausgabe als `virtual` kennzeichnen:

```
class Mitarbeiter {
    .....
    virtual void ausgabe() {
        cout << "Mitarbeiter " << _name;
        cout << " : " << _gehalt << endl;
    }
};
```

In der abgeleiteten Klasse `Manager` ist die Methode `ausgabe` dann automatisch auch `virtual`.

Wenn wir jetzt nochmal unser Hauptprogramm ausführen,

```
int main() {
    Mitarbeiter *m[2];

    m[0] = new Mitarbeiter("Max Meier", 3000);
    m[1] = new Manager("Hans Wurst", 4000, 1.3);

    for (int i = 0; i < 2; i++)
        m[i]->ausgabe();
}
```

erhalten wir als Ausgabe:

```
Mitarbeiter Max Meier : 3000
Manager Hans Wurst : 5200
```

Die Methode wird an das tatsächliche Objekt gebunden!

dynamische Bindung

Im UML-Klassendiagramm ist keine besondere Kennzeichnung für virtuelle Methoden vorgesehen. Virtuelle Methoden sind sowohl in UML als auch in Java und anderen Sprachen Standard. Ein Überschreiben muss in Java durch Angabe von `finally` explizit verhindert werden.

Polymorphie tritt immer im Zusammenhang mit Vererbung auf.

- Es existieren verschiedene Methoden mit gleicher Signatur in unterschiedlichen Ebenen einer Vererbungshierarchie.
- Erst zur Laufzeit wird bestimmt, welche der Methoden für ein gegebenes Objekt verwendet wird.

Gleichnamige virtuelle Methoden mit unterschiedlichen Argumenten sind echt verschieden und überschreiben sich nicht.

```
struct B {
    virtual int m(int p1) {                    // M1
        cout << "B::m(int)\n";
    }
    int m(int p1, int p2) {                    // M2
        cout << "B::m(int, int)\n";
    }
};
struct A : public B {
    int m(int p1) {                            // M3
        cout << "A::m(int)\n";
    }
    int m(char *p1) {                          // M4
        cout << "A::m(char*)\n";
    }
};
```

```cpp
int main(void) {
    A *a = new A();
    B *b = new B();

    b->m(1);
    b->m(1, 2);

    a->m(1);
    a->m(1, 2);
    a->m((char *)"abc");
}
```

Compiler liefert:

```
error: no matching function for
call to ‘A::m(int, int)’
      a->m(1, 2);
```

```cpp
int main(void) {
    B *a = new A();
    B *b = new B();

    b->m(1);
    b->m(1, 2);

    a->m(1);
    a->m(1, 2);
    a->m((char *)"abc");
}
```

Compiler liefert:

```
error: invalid conversion from
‘char*’ to ‘int’
      a->m((char *)"abc");
```

```
int main(void) {
    A *a = new A();
    B *b = new B();

    b->m(1);
    b->m(1, 2);

    a->m(1);
    // a->m(1, 2);
    a->m((char *)"abc");
}
```

Ausgabe, wenn Zeile 9 entfernt wird:

```
B::m(int)
B::m(int, int)
A::m(int)
A::m(char*)
```

```
struct A : public B {
    using B::m;

    int m(int p1) {          // M3
        cout << "A::m(int)\n";
    }
    int m(char *p1) {        // M4
        cout << "A::m(char*)\n";
    }
};
```

Ausgabe mit using-Direktive:

```
B::m(int)
B::m(int, int)
A::m(int)
B::m(int, int)
A::m(char*)
```

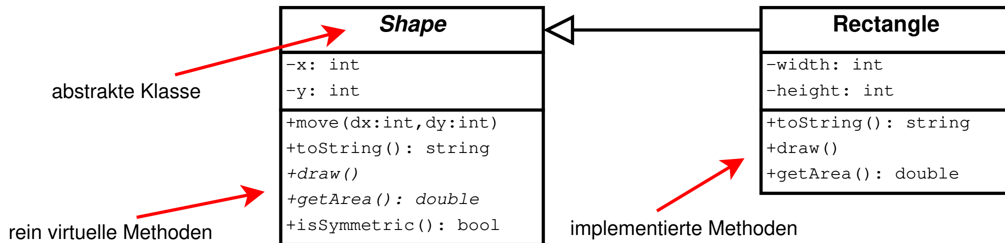
Anmerkungen:

- Der Konstruktor kann nicht als `virtual` definiert werden.
- Manchmal können Methoden in der Basisklasse noch nicht implementiert werden. Trotzdem möchte man sicherstellen, dass alle abgeleiteten Klassen diese Methode bereitstellen. Dann wird die Methode in der Basisklasse zwar definiert, aber nicht implementiert. Solche Methoden heißen rein virtuelle Methoden.

  Syntax: `virtual <type> methodname(<params>) = 0;`
- Rein virtuelle Methoden werden im UML-Diagramm kursiv dargestellt.
- Klassen, die rein virtuelle Methoden enthalten, heißen abstrakte Klassen.
- Man kann keine Objekte von abstrakten Klassen erzeugen.

In der Basisklasse `Shape`, die ein allgemeines geometrisches Objekt darstellt, kann der Flächeninhalt des Objektes nicht berechnet werden.

In der konkreten, abgeleiteten Klasse `Rectangle` kann der Flächeninhalt einfach bestimmt werden.



Die Basisklasse `Shape` erzwingt, dass alle konkreten geometrischen Objekte die Methode `getArea` zur Verfügung stellen.

```
#include <iostream>
using namespace std;

class Shape {
protected:
    int _x, _y;

public:
    void move(int dx, int dy) {
        _x += dx;
        _y += dy;
    }

    virtual string toString() { ..... } // ???

    // rein virtuelle Methode:
    virtual double getArea() = 0;
};
```

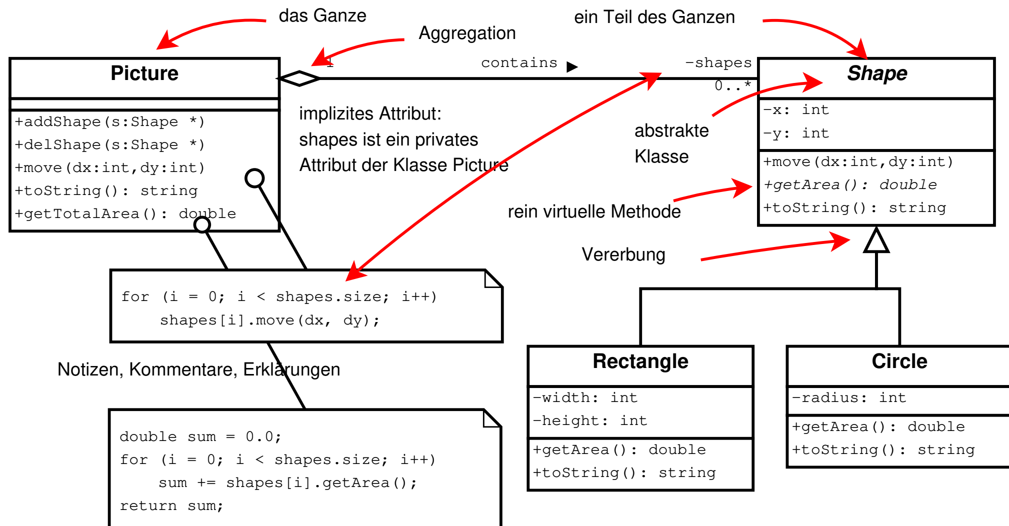
```
class Rectangle: public Shape {
private:
    int _w, _h;

public:
    Rectangle(int x, int y, int w, int h) {
        _x = x;              // vererbt von Shape
        _y = y;              // vererbt von Shape
        _w = w;
        _h = h;
    }

    string toString() { ..... }        // ?????

    double getArea() {     // definiert getArea()
        return _w * _h;
    }
};
```

```
class Circle: public Shape {
private:
    int _r;

public:
    Circle(int x, int y, int r) {
        _x = x;           // vererbt von Shape
        _y = y;           // vererbt von Shape
        _r = r;
    }

    string toString() { ..... }        // ?????

    double getArea() {    // definiert getArea()
        return _r * _r * 3.14159265358979 3;
    }
};
```

```
class Picture {
private:
    Liste<Shape *> _shapes;

public:
    void addShape(Shape *s) {
        _shapes.append(s);
    }

    void delShape(Shape *s) {
        _shapes.remove(s);
    }

    void move(int dx, int dy) {
        for (int i = 0; i < _shapes.size(); i++)
            _shapes[i]->move(dx, dy);
    }
    ....
```

```
    string toString() {
        string res;

        for (int i = 0; i < _shapes.size(); i++)
            res += _shapes[i]->toString();
        return res;
    }

    double getTotalArea() {
        double sum = 0.0;

        for (int i = 0; i < _shapes.size(); i++)
            sum += _shapes[i]->getArea();
        return sum;
    }
};
```

```
int main(void) {
    Picture pic;

    pic.addShape(new Rectangle(2, 0, 1, 2));
    pic.addShape(new Rectangle(0, 2, 3, 7));
    pic.addShape(new Circle(0, 1, 3));
    pic.addShape(new Circle(1, 0, 4));

    cout << "Summe Flaecheninhalt: "
         << pic.getTotalArea() << endl;

    cout << " vorher:\n" << pic.toString();

    pic.move(3, 4);
    cout << "nachher:\n" << pic.toString();
}
```



Delegation: Mechanismus, bei dem ein Objekt eine Nachricht nicht (vollständig) selbst interpretiert, sondern an ein anderes Objekt weiterleitet.

Typprüfung zur Laufzeit

Wie funktioniert die Typprüfung zur Laufzeit?

- Der Compiler erweitert den Code um eine Tabelle.
- Die Tabelle enthält für jede Klasse alle virtuellen Methoden.
- Anhand des Typs eines Objekts kann in dieser Tabelle die passende Methode gefunden werden.
- Dieses Nachschlagen in der Tabelle führt zu einer geringfügigen Verlangsamung des Programms.

Redefinition ist kein Polymorphismus!

Typprüfung zur Laufzeit

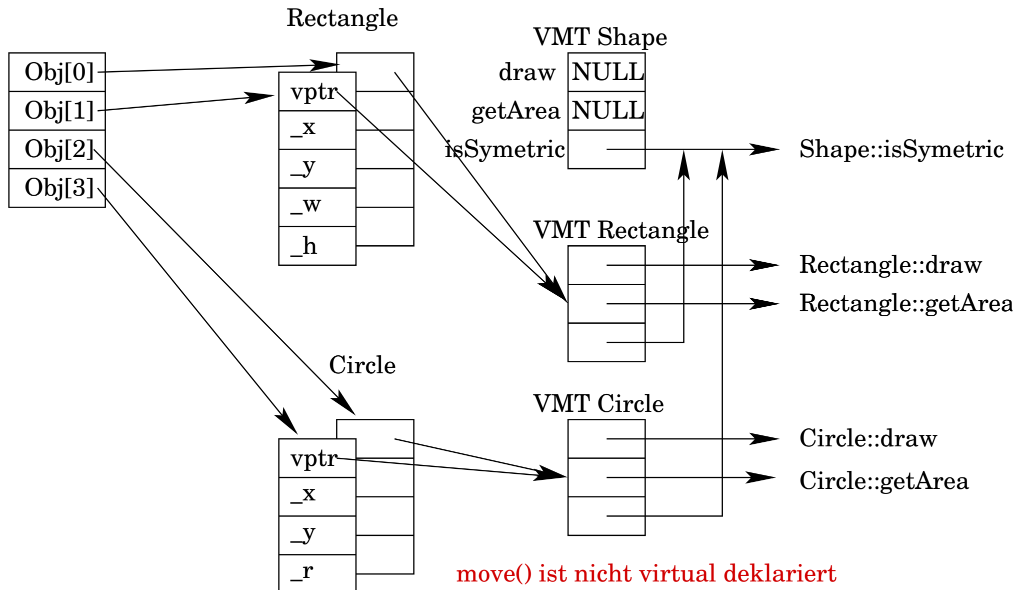
im Detail:

- Jeder Methodenaufruf wird zunächst anhand einer (laufenden) Nummer beschrieben.
- Für jede Klasse wird eine Virtual Method Table (VMT) aufgestellt, die für jede Methode den Verweis auf die Adresse der Methode enthält.
- Alle Objekte erhalten automatisch einen Verweis auf die VMT ihrer Klasse.
  - Lassen Sie sich zur Kontrolle `sizeof(Rectangle)` ausgeben!
  - Die Differenz zwischen angezeigtem Wert und der eigentlichen Größe sollte genau der Größe eines Zeigers entsprechen.
- Zur Laufzeit wird mit der Methodennummer aus dem Methodenaufruf die aktuelle VMT untersucht und daraus die Adresse der Methode ermittelt.

Betrachten wir eine Variante unseres Shape-Beispiels:

```
class Shape {
protected:
    int _x, _y;

public:
    virtual void draw() = 0;
    virtual float getArea() = 0;

    void move(int dx, int dy) {
        _x += dx;
        _y += dy;
    }
    virtual bool isSymetric() {
        return true;
    }
};
```

```
class Rectangle: public Shape {
protected:
    int _w, _h;

public:
    void draw() {           // hier implementieren
        .....
    }
    float getArea() {       // hier implementieren
        return _w * _h;
    }
    // move()          wird von Shape geerbt
    // isSymetric()    wird von Shape geerbt
};

class Circle: public Shape {
    .....   // analog zu Rectangle
};
```


Rectangle
VMT Shape
Obj[0]
Obj[1]
Obj[2]
Obj[3]
vptr
_x
_y
_w
_h
draw
NULL
getArea
NULL
isSymetric
Shape::isSymetric
VMT Rectangle
Rectangle::draw
Rectangle::getArea
Circle
VMT Circle
vptr
_x
_y
_r
Circle::draw
Circle::getArea
move() ist nicht virtual deklariert

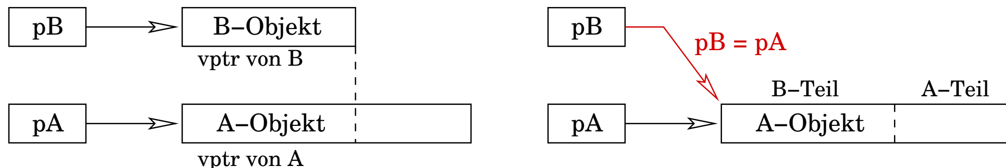
Wenn Sie unter Linux den C++-Compiler `g++` mit der Option `-fdump-class-hierarchy` aufrufen, werden Ihnen die angelegten virtual method tables angezeigt:

```
Vtable for Shape
Shape::_ZTV5Shape: 5u entries
0      (int (*)(...))0
4      (int (*)(...))(& _ZTI5Shape)
8      __cxa_pure_virtual
12     __cxa_pure_virtual
16     Shape::isSymetric
```

```
Vtable for Rectangle
Rectangle::_ZTV9Rectangle: 5u entries
0     (int (*)(...))0
4     (int (*)(...))(& _ZTI9Rectangle)
8     Rectangle::draw
12    Rectangle::getArea
16    Shape::isSymetric

Vtable for Circle
Circle::_ZTV6Circle: 5u entries
0     (int (*)(...))0
4     (int (*)(...))(& _ZTI6Circle)
8     Circle::draw
12    Circle::getArea
16    Shape::isSymetric
```

Typprüfung zur Laufzeit

Im Folgenden sei `pB` vom Typ „Zeiger auf Basisklasse“ und `pA` vom Typ „Zeiger auf abgeleitete Klasse“.



- Der „Basisteil“ der abgeleiteten Klasse ist ansprechbar.
- Jedes Objekt besitzt einen virtual table pointer (vptr), der auf die VMT seiner Klasse zeigt.
- Nach dem „umbiegen“ von `pB` auf das A-Objekt `pA` arbeitet `pB` mit dem `vptr` vom Objekt A, es wird also immer die passende Methode aufgerufen.
- Mit `pB` kann nicht auf Methoden oder Attribute der abgeleiteten Klasse zugegriffen werden!

Destruktoren bei „virtuellen Klassen“

Der Einsatz virtueller Methoden in einer Klasse verlangt oft, dass auch der Destruktor dieser Klasse `virtual` ist, damit der richtige Destruktor entsprechend des tatsächlichen Typs des Objekts aufgerufen wird!

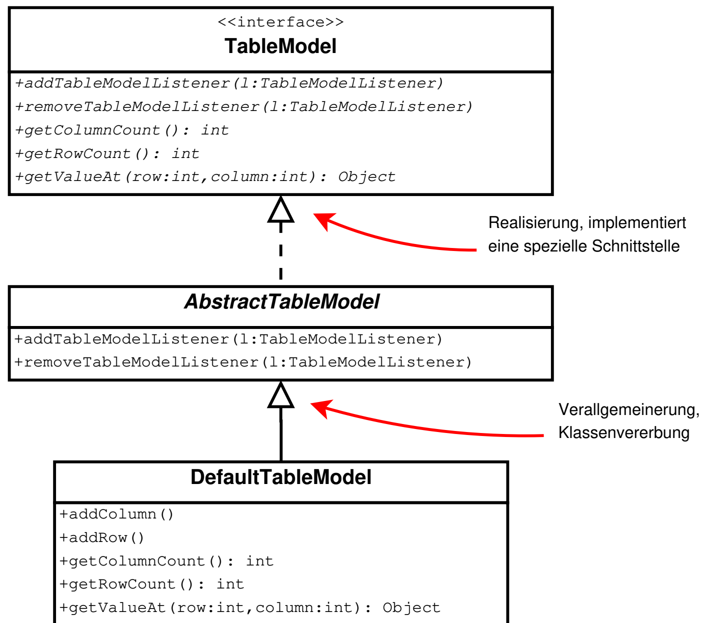
Ein Objekt der abgeleiteten Klasse kann den Platz eines Objekts der Basisklasse einnehmen!

```
class B {
public:
    virtual ~B();
    virtual int m();
};

class A: public B {
public:
    ~A();
    int m();
};

B *pB;
pB = new A();    // !!!!!
delete pB;       // !!!!!
```

Abstrakte Klassen

- Eine Klasse, die mindestens eine rein virtuelle Methode enthält, nennen wir abstrakte Klasse. In einer abstrakten Klasse sind also nicht alle Methoden implementiert.
- Als Folge der nicht-implementierten Methoden können keine Objekte abstrakter Klassen erzeugt werden.
- Rein virtuelle Methoden nennen wir auch abstrakte Methoden.
- Abstrakte Klassen definieren Schnittstellen, sie legen fest, was eine Klasse können soll. Die Realisierung der Funktionalität muss dann in einer konkreten Unterklasse erfolgen.
- Eine von einer abstrakten Klasse abgeleiteten Klasse muss alle vererbten abstrakten Methoden implementieren, damit die erbende Klasse selbst nicht abstrakt ist.
- In Java: Ein `Interface` ist eine besondere Form einer Klasse, die nur abstrakte Methoden sowie Konstanten enthält.

Abstrakte Klassen



Abstrakte Klassen

- Das Interface `TableModel` legt die Schnittstelle fest, beschreibt also, welche Methoden implementiert werden müssen.
- Die abstrakte Klasse `AbstractTableModel` stellt eine partielle Implementierung bereit: Nicht alle, aber einige Methoden werden implementiert.

  Es werden solche Methoden implementiert, die wahrscheinlich für die meisten Realisierungen vom `TableModel` nutzbar sind, bspw. das An- und Abmelden von Beobachtern (siehe Entwurfsmuster Beobachter).
- Am Ende der Hierarchie steht eine Implementierung, die in vielen Projekten nutzbar ist, das `DefaultTableModel`.

  Diese Klasse kann bei speziellen Bedürfnissen weiter spezialisiert werden.