LANDESBERUFSSCHULE 4 SALZBURG

Informatik

Vererbung

LBS 4

Dieses Skript dient als zusätzliche Lernunterlage für Informatik

Inhalt

Lernziele

Die Schülerinnen und Schüler können/kennen

- das Konzept der Vererbung erläutern
- den Ablauf der Erzeugung eines Objektes erklären
- Methoden neu definieren (Re-Definition, überschreiben)

Vererbung

Die Vererbung ist, neben der Kapselung, ein weiteres, sehr wichtiges Grundkonzept für die Wiederverwendbarkeit von Code in der OOP. Über die Vererbung werden neue Klassen (Datentypen) erzeugt, welche alle Eigenschaften und Methoden von einer Basisklasse (Elternklasse) erbt. So kann sehr einfach die Funktionalität einer bereits vorhandenen Klasse erweitern und/oder neu definieren.

In einer Basisklasse sind alle gemeinsamen Attribute und Methoden für die abgeleiteten Klassen vorhanden.

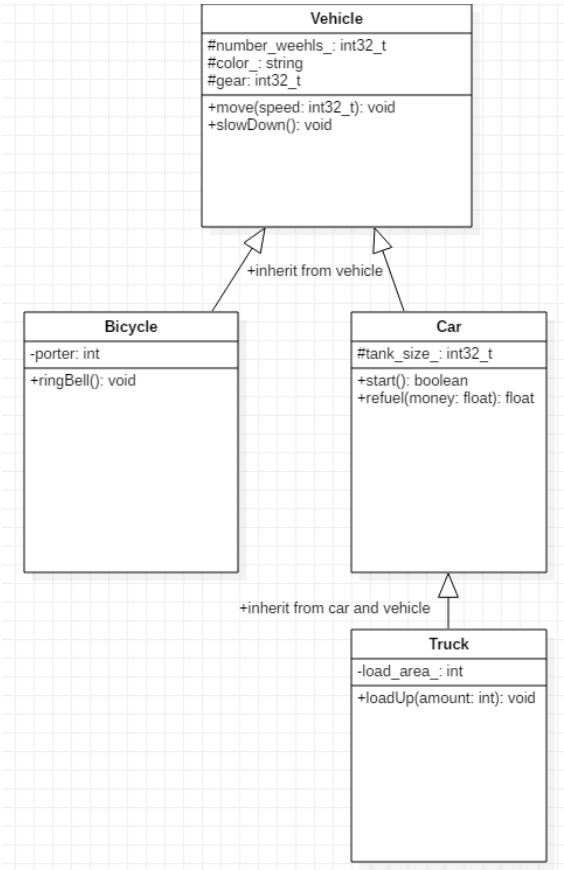


Abbildung 1: Vererbung

Das Erweitern einer Klasse ist nicht schwierig: Sie implementieren dazu einfach weitere Eigenschaften und Methoden in der neuen, abgeleiteten Klasse. Vererbung wird häufig für Programme genutzt, bei denen eine Basisfunktionalität in mehreren, unterschiedlichen Klassen vorhanden sein soll.

Aber auch das Überschreiben von geerbten Methoden mit einer neuen Funktionalität ist möglich. So können Sie neue Klassen erzeugen, die im Prinzip genauso verwendet wird wie die Basisklasse, die aber ein ganz anderes Verhalten zeigt.

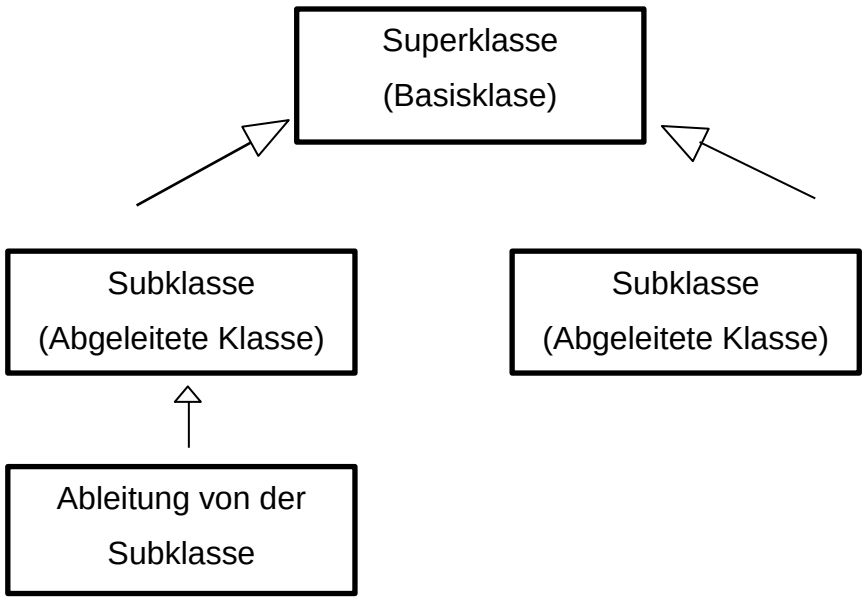
Warum Vererbung?

Die wichtigen Punkte bei der Vererbung sind:
- die Wiederverwendung von Programmcode
- die Wartbarkeit von Programmen
- der über die Vererbung erreichte Polymorphismus.

In der Programmierung ist es wichtig eine Basisfunktionalität in einer erweiterten Form zu verwenden ohne dass die Basisfunktionalität verändert wird.

Das nachstehende Beispiel zeigt das Prinzip der Vererbung. In der Praxis wird dann von Ableitung gesprochen.

Hierarchie der Vererbung:



Eine Subklasse erbt alle Methoden und Attribute die mit public oder protected in der Basisklasse deklariert sind.

Syntax der Vererbung:

```
class <Name der neuen Klasse> : Zugriffsspezifizierer <Name der Basisklasse>
class Arectangle : public Rectangle
```

Als Zugriffsspezifizierer können wieder drei verschiedene Schlüsselwörter verwendet werden

.

Arten:

Ableitungsart	Sichtbarkeit in Basisklasse	Sichtbarkeit in abgeleiteter
Public	Private	kein Zugriff
	Protected	protected
	Public	public
Protected	Private	kein Zugriff
	Protected	protected
	Public	protected
Private	Private	kein Zugriff
	Protected	private
	Public	private

Beispiel: Rectangle

Basisklasse Header

```
//Rectangle.h
class Rectangle
{
protected:                              //für die Abgeleiteten Klassen
   int32_t length_, width_;                    // die Länge
public:                                 //von außen aufrufbar
   void setLength(int32_ length_)   ;          // Methode zum Setzen der Länge
   int32_t getLength();                        // Methode zum Lesen der Länge
   double getCircum();                  // Methode zur Berechnung des Umfangs
};
```

Basisklasse Implementierung:

```
 //Rectangle.cpp
#include “Rectangle.h”

 void Rectangle::setLength(int32_t length)        // Methode zum Setzen der Länge
  {
    if (length > 0)
      this->length = length;
    else
      this->length = 0;
  }
  int32_t Rectangle::getLength()           // Methode zum Lesen der Länge
  {
    return this->length;
  }
  double Rectangle::getCircum()            // Methode zur Berechnung des Umfangs
  {
    return 2*(this->length + this->width);
  }
```

Abgeleitete Klasse:

Die neue Klasse wird um das Attribut *area* und der Methode *calcArea()* erweitert.

```
//Abgeleitete Klasse Arectangle
#include "Rectangle.h"

class Arectangle : public Rectangle          //Vererbte Klasse Public setzen
{
  private:
     double area;                      //zusätzliches Attribut
  public:
   double calcArea();                          //zusätzliche Methode
}
```

In der *.cpp muss jetzt nur die neue Methode ergänzt werden. Alle anderen können einfach verwendet werden.

```
//Implementation
#include "Arectangle.h"

double Arectangle::calcArea(){
return this->length * this->width;
}
```

Aufruf in der main() Funktion:

```
#include <iostream>
#include "Arectangle.h"
using namespace std;

int main()
{
   Arectangle r1;
   r1.setLength(5);
   r1.setWidth(10);
   cout << "Der Umfang betraegt: " << r1.getCircum() << endl;
   cout << "Die Fläche betraegt: " << r1.calcArea() << endl;
   return 0;
}
```

Konstruktoren:

Konstruktoren werden in der Vererbung nicht weitergegeben. Es muss für jede abgeleitete Klasse ein eigener Konstruktor erstellt werden.

Als erstes wird der Konstruktor der Basisklasse aufgerufen und danach der Konstruktor der abgeleiteten Klasse.

Wenn B eine Subklasse von A ist dann wird als erstes der Konstruktor der Klasse A und dann der Konstruktor der Klasse B aufgerufen.

Wenn in der Basisklasse A kein Standard-Konstruktor vorhanden ist, muss mit einer Initialisierungsliste der Konstruktor der Klasse A aus der abgeleiteten Klasse aufgerufen werden.

Der Initialisierer wird nach dem Konstruktor mit einem Doppelpunkt getrennt angegeben.

```
#include <iostream>

class Pet {
protected:
int32_t age_;
public:

Pet(int32_t age);
};
//cpp zur Klasse Pet:

Pet::Pet(int32_t age) {
std::cout << "Konstruktor in Pet (--- age ----)" << std::endl;
}
//abgeleitete Klasse Dog
class Dog : public Pet {
public:
Dog(int32_t v);
```

```
};

//cpp zur Klasse Dog:
/*Dog::Dog():Pet(INTEGER) {
std::cout << "Standard-Konstruktor in Dog()" << std::endl;
}*/
Dog::Dog(int32_t age) : Pet(age) {
std::cout << "Konstruktor in Dog Initialisierung mit Pet:( << age << )" << std::endl;
};

//Initialisierung der Klassen

int main() {

std::cout << "Aufruf der Klasse Pet" << std::endl;
Pet my_pet(34); // Ausgabe: Konstruktor Pet ….

std::cout << "\nAufruf der Klasse Dog mit Parameter" <<std::endl;
Dog my_dog(5); // Ausgabe: Konstruktor Pet …Konstruktor Dog…

return 0;
}
```

Wichtig:

Wenn abgeleitete Klassen auf private Elemente zugreifen wollen, muss die Sichtbarkeit in der Basisklasse auf *protected* geändert werden.

Der Kernpunkt der Vererbung ist die einfache Wiederverwendung von Programmcode. Durch die Vererbung kann Code an zentraler Stelle sehr einfach gewartet werden.

Konstruktoren vererben

Ein Konstruktor der Basisklasse kann wiederverwendet werden, wenn die Abgeleitete Klasse keine neuen Attribute besitzt. Wenn der Konstruktor der Basisklasse vererbt werden soll, muss dieser mit der using Deklaration geschehen.

```
class Basisklasse {
private:
std::string name_;

public:
Basisklasse(const std::string& bezeichner)
: name_ {bezeichner} {
}
const std::string& getAttribut() const {
return name_;
}
};
```

```
class Abgeleiteteklasse : public Basisklasse {
private:
int wert (99);

public:
using Basisklasse::Basisklasse; // Basisklassen-Konstruktoren erben

Abgeleiteteklasse(const std::string& bez, int32_t nr): Basisklasse {bez}, wert {nr} {
}
int32_t getWert() const {
return wert;
}
```

Verwenden der Objekte

```
int main() {
Abgeleiteteklasse first_test("Hallo", 42); // allgemeiner Konstruktor
cout << first_test.getAttribut() << " " // Hallo
<< first_test.getWert() << endl; // 42

Abgeleiteteklasse second_test("Hallo Welt!"); // geerbter Konstruktor:
cout << second_test.getAttribut() << " " // Hallo Welt
<< second_test.getWert() << endl; // 99 !
}
```

Re-Definition (Überschreiben) von Methoden:

In der abgeleiteten Klasse können Methoden mit dem gleichen Namen und selber Signatur definiert werden. Dabei entsteht kein Überladen der Methoden. Es existieren somit mehrere Versionen der Methode mit der gleichen Signatur. Das überladen der Methoden ist ebenfalls möglich.

Es besteht die Möglichkeit die Methode der Basisklasse ebenfalls zu verwenden, wenn diese als public oder protected definiert ist.

Allgemein sollen überschriebene Methoden grundsätzlich virtuell sein!

Beispiel: C++-Code

```
class Rectangle{
    private:
    protected:
        int32_t length_;
        int32_t width_;
    public:
        Rectangle();
        Rectangle(int32_t length, int32_t width);
        virtual ~Rectangle();
        void show();

};
```

```
//Abgeleitete Klasse
class Arectangle : public Rectangle
{
   private:
   protected:
   public:
      Arectangle(int32_t a,int32_t b);
      virtual ~Arectangle();
      void show();
};
```

Die Methode show() in der Basisklasse:

```
#include "Rectangle.h"
#include <iostream>
Rectangle::Rectangle(int32_t length, int32_t width)
{
   this->length_ = length;
   this->width_ = width;
}

void Rectangle::show(){
std::cout << "Die Länge beträgt: " << this->length_ << "m und die Breite beträgt: " <<
this->width_ << "m" << std::endl;
}

Rectangle::~Rectangle()
{
}
```

Die Methode in der vererbten Klasse

```
#include "Arectangle.h"
#include <iostream>
Arectangle::Arectangle(int width, int length):Rectangle(width, length)
{

}
void Arectangle::show(){
Rectangle::show();          //Aufruf von show() von der Basisklasse
std::cout << "Die Fläche beträgt: " << this->length_ * this->width_ << "m^2" <<
std::endl;
}
Arectangle::~Arectangle()
{
}
```

Aufruf der Methoden:

```
#include <iostream>
#include "Arectangle.h"

int main()
{
   Rectangle r(1,2);
   Arectangle *my_rectangle = new Arectangle(12,23);
   std::cout << "-----------------------------------------------"<< std::endl;
   std::cout << "Die Show-Methode in Rectangle\n";
   r.show();
   std::cout << "-----------------------------------------------"<< std::endl;
   std::cout << "Die Show-Methode in der abgeleiteten Klasse: Arectangle" <<
std::endl;
   my_rectangle->show();
   std::cout << "-----------------------------------------------"<< std::endl;
   return 0;
}
```