# Abend 7
# Vererbung und Polymorphie, Abstrakte Klassen

## Ziel, Inhalt

- Wir sehen heute weitere Beispiele für Polymorphie und virtuelle Methoden. Wir lernen auch Klassen kennen, von denen man keine Objekte erzeugen kann, die aber ds Verhalten von Objekten definieren.

# Beispiele zur Polymorphie

## Objektfabrik

Bei diesem Beispiel definieren wir zuerst eine Basisklasse und zwei abgeleitete Klassen davon. Um Objekte zu erzeugen verwenden wir aber nicht den Konstruktor, sondern wir definieren eine Methode, die nach dem Zufallsprinzip Objekte einer der drei Klassen erzeugt. Solche Methoden nennt man auch „Factory-Method“, eine Fabrik-Methode.

### Die Basisklasse

Als Basisklasse können wir nachdem wir das letzte mal Autos die Umwelt „vergiftet“ haben, etwas aus der Natur nehmen. Bio-C++ sozusagen! Nehmen wir doch eine Klasse *Vogel* als Basisklasse. Ein Vogel hat folgende Eigenschaften, die als Datenelemente definiert werden: Gewicht in Gramm und das Geschlecht. Als Methoden kommt in Frage was ein Vogel kann. Wir gehen von Vögeln aus, die Fliegen können. Ein Vogel kann auch Brüten. Hier also die Definition der Klasse Vogel:

```
typedef long Gramm;

enum Geschlecht
{
    maennlich,
    weiblich
};

class Vogel
{
    public:
        // Konstruktor
        Vogel(Gramm gewicht, Geschlecht geschlecht);
        // Destruktor
        ~Vogel();

        // Methoden
        void flieg();
        void bruete();

    private:
        Gramm      m_gewicht;
        Geschlecht m_geschlecht;
};
```

In der Implementation dieser Klasse, in der .cpp-Datei also, geben wir bei den verschiedenen Methoden einfach etwas auf der Konsole aus.

```
//z.B.
void Vogel::flieg()
{
    cout << „Ein Vogel fliegt“ << endl;
}
```

Erzeuge nun ein Projekt und Implementiere diese Klasse *Vogel*.

**Die abgeleiteten Klassen**

Nun erzeugen wir zwei neue Klassen. Es ist dir überlassen was für „Spezialisierungen“ von *Vogel* du wählst. Mein Vorschlag hier ist zum Beispiel eine Klasse *Sperling* und eine Klasse *Falke*.
Erzeuge diese Klassen (möglichst mit eigenen Header- und Quellcode-Dateien) und redefiniere alle Methoden. Um diese ein wenig spannender zu machen, kannst du in der Klasse *Falke* und der Klasse *Sperling* noch einen Zeiger auf ein mögliches „Kind“ definieren. In der Methode *bruete* erzeugst du das „Junge“ mit *new* und weist dem Kind-Zeiger das neu erzeugte Objekt zu. Unlogischerweise sollst du diesen Zeiger löschen (mit *delete*) wenn das Eltern-Objekt zerstört wird. Mit diesem Kniff zeigen wir später, dass es wichtig ist einen virtuellen Destruktor zu definieren.

**Die Objektfabrik**

Bis jetzt können wir diese Objekte einfach erzeugen in dem wir eine Variable vom jeweiligen Datentyp definieren.

```
#include „Vogel.h“
#include „Falke.h“
#include „Sperling.h“

int main()
{
    Vogel einVogel(34, maennlich);
    Falke einFalke(380, weiblich);
    Sperling einSperling(80, weiblich);

    return 0;
}
```

Jetzt ändern wir aber unsere Klassen so ab, dass das nicht mehr möglich ist. Wir verhindern das direkte Erzeugen von Objekten indem wir die Konstruktoren in den *private* Teil der Klassendeklaration verschieben.
Wie erzeugen wir nun aber Objekte?
Hier eine Möglichkeit, die zwar nicht einen Software-Design Preis gewinnen wird, aber für unsere Möglichkeiten genügend einfach ist.

Die Klasse Vogel, dient ab jetzt als Objektfabrik. Wir definieren in der Klasse Vogel eine statische Methode (*static*), die eines der drei möglichen Vogelobjekte erzeugt und einen Zeiger auf Vogel zurückgibt:

```
class Vogel
{
    public:
        // Fabrikmethode
        static Vogel* erzeugeVogel();
        // Destruktor
        ~Vogel();

        // Methoden
        void flieg();
        void bruete();

    private:
        // Konstruktor nicht mehr öffentlich!
        Vogel(Gramm gewicht, Geschlecht geschlecht);

    private:
        Gramm      m_gewicht;
        Geschlecht m_geschlecht;
};
```

Die Implementation sieht dann etwa so aus:

```
#include „Falke.h“
#include „Sperling.h“
#include <time.h> // für time-Aufruf für die Zufallszahlen
#include <stdlib.h> // für rand(), Zufallszahlen

Vogel* Vogel::erzeugeVogel()
{
    // eine Saat für den Zufallszahlengenerator
    // erzeugen
    srand((unsigned int)time(0));
    // Zufallszahl erzeugen
    int vogelArt = rand();
    // wir nehmen den Rest von der Division
    // durch drei (modulo) als Kriterium für
    // Vogelart
    vogelArt = vogelArt % 3;
    // das gleich machen wir für das Geschlecht
    int g = rand();
    g = g % 2;
    Geschlecht geschlecht = g ? maennlich : weiblich;
    // und das Gewicht
    long gewicht = rand();
    gewicht = gewicht % 100;
```

```
    Vogel* pVogel = 0;
    switch(vogelArt)
    {
        case 0:
            pVogel = new Vogel(gewicht, geschlecht);
        break;
        case 1:
            pVogel = new Falke(gewicht, geschlecht);
        break;
        case 2:
            pVogel = new Sperling(gewicht, geschlecht);
        break;
        default:
            // das ist unmöglich!!!
        break;
    }
    return pVogel;
}
```

Damit wir von der Klasse *Vogel* aus den Konstruktor von *Falke* oder *Sperling* aufrufen dürfen, müssen wir die Klasse *Vogel* in den Klasse *Falke* und *Sperling* als *friend* deklarieren!

**Die main-Funktion**

In der main-Funktion können wir nun mit #include „Vogel.h“ die statische Methode aufrufen um *Vogel** zu erhalten.
Probier das aus und merke die diese Zeiger in einem Array.

```
Vogel* vogelZeigerArray[20];
```

Rufe auf die erzeugten Objekte auch die verschiedenen Methoden auf und lösche die erzeugten Objekte am Ende wieder.

**Beobachtung**

Nun bemerkt ihr sicher, wie schlecht das funktioniert, denn wir halten uns im main nur *Vogel**. Alle Aufrufe werden darum so behandelt als wären die Objekte nur *Vogel*-Objekte.

**Polymorphe Vögel**

Wir kennen nun aber eine einfache Möglichkeit jedes Objekt dazu zu bringen, das zu tun was es eigentlich sollte. Es genügt die Methoden, die normalerweise redefiniert werden in der Basisklasse mit dem Schlüsselwort *virtual* zu „schmücken“.
Ändere zunächst die beiden Methoden *bruete* und *fliege* so ab und teste dein Programm wieder. Die Methode *bruete* sollte jetzt auch neue Objekte erzeugen. Da aber der Destruktor noch nicht virtuell ist, erzeugen wir Speicherlecks. Ich zeige euch in der Lektion wie wir solche Speicherlecks erkennen können. Es genügt folgende Ergänzungen im main zu machen:

```
#include <crtdbg.h>

int main()
{
    // einschalten der leck-Funktionen
    _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF |
                     _CRTDBG_LEAK_CHECK_DF );

    int* test = new int[50];

    return 0;
}
```

Beim verlassen wird nun Information über die Speicherlecks unten in das „Debug“-Fenster geschrieben.
Durch „virtualisieren“ des Destruktors ist auch dieses Problem beseitigt.

## Vogel als abstrakte Basisklasse

Im Beispiel konnten wir problemlos Objekte der Basisklasse *Vogel* erstellen. Je nach Problemstellung oder Design müssen wir aber verhindern, das Objekte einer gewissen Basisklasse erzeugt werden. Trotzdem möchten wir die Basisklasse definieren, um ein gemeinsames Verhalten von Objekten zu erzwingen. Das heisst wir wollen zwar mit einer Basisklasse eine gemeinsame Schnittstelle von verschiedenen Klassen definieren, wir wollen aber keine Objekte der Basisklasse. Man spricht hier von abstrakten Klassen im Gegensatz zu den konkreten Klassen. Ich verwenden dafür auch den Begriff von Interface-Klassen, denn es sind Klassen die dazu dienen eine gemeinsame Schnittstelle zu definieren.

### Klasse Vogel

Diese Klasse soll also neu ein Verhalten definieren, ohne dass man aber *Vogel*-Objekte erzeugen kann. Die Klasse Vogel sieht neu so aus:

```
class Vogel
{
    public:
        // Fabrikmethode
        static Vogel* erzeugeVogel();
        // Destruktor
        virtual ~Vogel();

        // Methoden
        virtual void flieg() = 0;
        virtual void bruete() = 0;

    private:
        Gramm      m_gewicht;
        Geschlecht m_geschlecht;
};
```

Beachte die beiden Methoden *flieg* und *bruete*. Wir setzen die Methoden einfach „gleich 0“. Diese Methoden sind jetzt *rein virtuell*. Wir weisen den Compiler an, diese beiden Methoden als nicht existent zu betrachten. Wenn wir versuchen ein Objekt der Klasse *Vogel* zu erstellen sucht der Compiler nicht nach dem Code der Methoden, sondern er gibt eine Fehlermeldung aus.
Der Konstruktor macht nun auch keinen Sinn mehr, wir können kein *Vogel*-Objekt mehr erzeugen. Hingegen lassen wir den Destruktor so wie er ist. Es ist nicht möglich den Destruktor *rein virtuell* zu machen.

### Ändern der Fabrikmethode

Es sollte nicht schwierig sein die Fabrikmethode *erzeugeVogel* so abzuändern, dass sie nicht mehr versucht reine *Vogel*-Objekte zu erzeugen, sondern nur nach Falken oder Spatzen (Sperlinge) erzeugt.

## Anwendung von abstrakten Klassen

Abstrakte Klassen werden in objektorientierten Systemen häufig angewendet. Sie dienen jeweils dazu das grundsätzlich Verhalten, also die Schnittstelle von Objekten zu definieren.
Wenn wir bei unserem Beispiel einfach nur Vögel fliegen oder brueten lassen wollen, brauchen wir nicht mehr als das *Vogel*-Interface. Es ist nicht nötig die „Falke.h“ und die „Sperling.h“-Dateien zu includieren. Dadurch sind wir auch überhaupt nicht abhängig von diesen Klassen. Wenn sie sich ändern, stört uns das überhaupt nicht. Wir haben lose Kopplung. Je grösser ein System ist, desto wichtiger ist lose Kopplung, denn Änderungen an einem Ort bewirken nicht unbedingt Änderungen an einem anderen Ort. Das kann

sich auch auf die Kompilierzeit auswirken. Grosse Projekte benötigen manchmal lange Kompilierzeiten. Durch lose Kopplung lassen sich ganze „Rebuilds“ also das komplette Neukompilieren vermeiden. Das gilt solange die Interfaces sich nicht ändern. Beim Systemdesign ist es darum von Vorteil schon bald die Schnittstellen festzulegen. Werden diese als abstrakte Klassen bereitgestellt, können die Entwickler einigermassen unabhängig voneinander das System implementieren.