Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Warum sprechen wir von Objektorientierung und nicht von Klassenorientierung? Wir definieren in C++ doch Klassen.

- Wir Menschen nehmen einzelne Objekte wahr, wir beobachten Objekte, wir interagieren mit Objekten und
- wir finden gemeinsame Verhaltensweisen und Eigenschaften.

→ Wir klassifizieren und abstrahieren!

Beispiel: Bello und Rex haben vier Beine und machen Wau-Wau.

- Wir klassifizieren: Beides sind Hunde. Wir fassen Gegenstände allgemein auf und finden einen Begriff dafür. Und wir grenzen ab: Hunde unterscheiden sich von Katzen.
- Wir abstrahieren: Innere Abläufe wie Atmung, Verdauung oder Durchblutung spielen für uns keine Rolle. Wir vereinfachen die komplexe Welt, um sie leichter verstehen zu können.

Objekte, die ein gemeinsames Verhalten aufweisen, werden in Klassen allgemein beschrieben.

- im Vordergrund: Verhalten der Objekte nach außen
- im Hintergrund: Daten, die intern notwendig sind, um das Verhalten nach außen zu erfüllen

→ *konzentrieren auf das Verhalten (Schnittstelle)*

Klassen werden eingesetzt,

- um Objekte mit gleichem Sachverhalt allgemein gültig zu beschreiben und
- um einen komplexen Sachverhalt zu kapseln.

Eine Klasse ist ein Bauplan für gleichartige Objekte.

Betrachten wir als Beispiel unsere Liste:

- Bei einem Array
  - findet keine Bereichsüberprüfung statt und
  - die Größe wird nicht automatisch angepasst.
- Kapsele Datentyp Array als Klasse `Liste` und schütze den Zugriff auf die Daten.
- Nach den Tests können ohne Bedenken viele Objekte der Klasse angelegt werden.

Vorteile dieses Vorgehens:

- Die Komplexität wird reduziert.
- Programmierer konzentrieren sich auf die Lösung des eigentlichen Problems.
- Eine Wiederverwendung von Code wird möglich.

Beispiele aus anderen Bereichen:

- Ein Fernseher hat eine einfache Benutzer-Schnittstelle.
  → Die komplexe Technik im Innern muss vom Nutzer nicht verstanden werden.
- Im Computer sind einzelne Komponenten durch ein einheitliches Bus-System verbunden.
  → Die einzelnen Komponenten können unabhängig von den anderen entworfen und betrieben werden, solange die Bus-Spezifikation eingehalten wird.

→ So sollte auch Software entworfen sein!

Abstraktionsmittel

- Oberbegriff (*ist-ein*-Beziehung)

  Fasst mehrere Arten oder Varianten von Objekten unter einem Begriff zusammen.

  Man sagt: *Da hinten fahren 3 Autos.*
  nicht: *ein Opel, ein Ford und ein VW*

- Teile/Ganzes (*hat-eine*-Beziehung)

  Fasst mehrere Einzelteile zu einem Objekt zusammen.

  Man sagt: *Da hinten fährt ein Auto.*
  nicht: *ein Motor mit Karosserie, Lenkrad und 4 Rädern*

Ursprung der Objektorientierung

- Objektorientierte Programmiersprachen wie Simula und Smalltalk gibt es schon seit Ende der 1960er Jahre.
- OO-Sprachen fristeten aber zunächst ein Nischendasein, die prozeduralen Programmiersprachen wie Algol-68, C und Fortran setzten sich durch.
- Die Abstraktionsmöglichkeiten der prozeduralen Sprachen sind sehr beschränkt, reichen aber für kleine Programme aus.
- Computer wurden immer leistungsfähiger, was zu steigenden Anforderungen an Software führte. Als Folge davon wurden die Programme immer komplizierter.
- Um diese Komplexität noch beherrschen zu können, müssen die Programmiersprachen andere Konzepte bereitstellen.
- Edsger W. Dijkstra: „The art of programming is the art of organizing complexity.“

Ursprung der Objektorientierung

Edsger W. Dijkstra in seiner Dankesrede zum Turing Award:

> *[The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

Im Verlauf der Software-Krise erinnerte man sich an die Konzepte der objektorientierten Programmierung und hoffte, damit den Weg aus der Krise zu finden.

Heute haben wir aspektorientierte Programmierung, agile Methoden und modellgetriebene Software-Entwicklung und trotzdem werden Budgets überzogen, Termine nicht eingehalten und Software ist immer noch fehlerhaft.

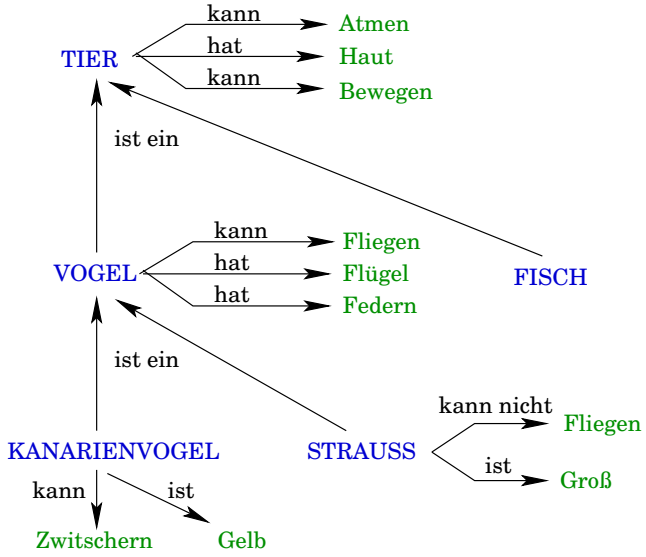
Auch das Gebiet der künstlichen Intelligenz bzw. die Entwicklung wissensbasierter Systeme forcierte die Entwicklung der objekt-orientierten Sprachen.

Wie wird Wissen im menschlichen Gehirn gespeichert? Können wir in ähnlicher Weise Wissen im Rechner speichern?

Semantische Netze stellen Wissen in einem Graphen dar:

- Knoten entsprechen Fakten bzw. Konzepten und
- Kanten entsprechen Relationen bzw. Assoziationen zwischen Konzepten.
- Sowohl Knoten als auch Kanten sind in der Regel mit Beschriftungen versehen.

Ursprung der Objektorientierung



Semantische Netze entsprechen vielleicht der Speicherung von Informationen beim Menschen. (Harmon, King, 1985)

Satz	Antwortzeit (s)
Ein Kanarienvogel ist ein Kanarienvogel.	1.0
Ein Kanarienvogel ist ein Vogel.	1.18
Ein Kanarienvogel ist ein Tier.	1.26
Ein Kanarienvogel kann zwitschern.	1.32
Ein Kanarienvogel kann fliegen.	1.38
Ein Kanarienvogel hat Haut.	1.48

Am schnellsten konnten die Informationen abgerufen werden, die spezifisch für den jeweiligen Vogel sind.

Auch die Verarbeitung von Ausnahmen scheint auf der spezifischen Ebene zu erfolgen. Auf die Frage, ob Strauße fliegen können, wurde schneller geantwortet als auf die Frage, ob Strauße atmen können.

Funktionalität erweitern

Wir wollen unsere Liste um

- eine Suchfunktion `bool contains(T val)` und
- eine Abfrage der Größe `int size()` erweitern und
- den Index-Operator überladen.

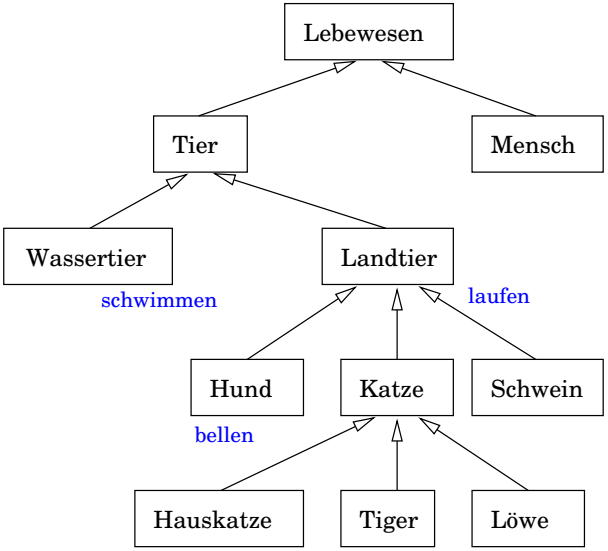
Frage: Spricht irgendetwas dagegen, diese Erweiterungen einfach in den bestehenden Code zu schreiben?

Eigentlich nicht, aber: Erweiterungen in den bestehenden Code zu schreiben ist nicht immer eine gute Idee.

- Der Source-Code der Klasse `Liste` wird immer umfangreicher und ist nur noch schwer zu verstehen/warten.
- Beim Ändern/Erweitern von bestehendem Code werden evtl. Fehler eingebaut und alte Software läuft anschließend nicht mehr.
- Der Source-Code steht in der Praxis nicht unbedingt zur Verfügung.

Wir brauchen einen Mechanismus, der uns die alte Funktionalität bereit stellt, aber dennoch Erweiterungen zulässt! Vielleicht entkommen wir so der Softwarekrise.

Hierarchische Anordnung von Klassen



Hierarchische Anordnung von Klassen

In einer solchen Hierarchie gilt:

- Alle Lebewesen einer Klasse verfügen über identische Eigenschaften.
- Die Klassen in einer tieferen Hierarchiestufe sind eine Spezialisierung der direkt übergeordneten Klasse.
- Die von einer Klasse abgeleiteten Unterklassen verfügen immer über alle Eigenschaften der Oberklasse.
- Abgeleitete Unterklassen können weitere Eigenschaften hinzufügen.

Eine solche Hierarchie ist analog zu einem evolutionärem Stammbaum, daher spricht man auch von der Vererbung von Eigenschaften.

Hierarchische Anordnung von Klassen

Obiges Konzept ist auch anwendbar auf Software-Entwicklung!

Begriffe:

- Die Klasse, die Eigenschaften weitervererbt, wird Oberklasse, Basisklasse oder Superklasse genannt.
- Klassen, die etwas erben, heißen Unterklassen, Subklassen oder abgeleitete Klassen.
- Die Oberklasse ist eine Generalisierung der Unterklassen.
- Eine Unterklasse ist eine Spezialisierung der Oberklasse.

Eine abgeleitete Klasse kann wiederum als Basisklasse für weitere Unterklassen dienen.

Typische Anwendungsfälle für Vererbung

- Es wird eine Klasse benötigt, die eine Spezialisierung oder Erweiterung einer bereits vorhandenen Klasse ist.

  hier: Wir wollen der `Liste` die Funktion `size` hinzufügen, ohne den bestehenden Code zu ändern.

- Eine Klasse vereint die Eigenschaften mehrerer vorhandener Klassen.

- Man entwickelt mehrere Klassen parallel, und man erkennt erst nachträglich, dass viel Gemeinsamkeiten vorliegen, für die man eine Basisklasse einführt.

  Wie kann das trotz guter Projektplanung passieren?
  - Software-Entwicklung häufig auf Zuruf und unter Zeitdruck.
  - Klassen entstehen in verschiedenen Projekten.

  $\rightarrow$ Refactoring

Erweitern der Funktionalität

Unsere Liste soll zusätzlich

- eine Suchfunktion `bool contains(T val)` bereit stellen,
- die Anzahl der gespeicherten Elemente ausgeben `int size()`
- und einen Zugriff auf die Elemente über den Index-Operator ermöglichen.

Und wenn wir schon mal dabei sind: Die Liste soll zusätzlich

- das Minimum ausgeben: `T minimum()`
- den Median ausgeben: `T median()`
- die Werte sortieren: `void sort()`

extliste.h

```
#include "liste.h"
#include "exception.h"

template <typename T>
class ExtListe: public Liste<T> {                    // Schutztyp
    bool _sorted;                          // bzw. Ableitungsmodus
    bool isEmpty() {
        return Liste<T>::_next == 0;
    }
public:
    ExtListe(int size = 8);
    void append(T val);                          // überschreiben
    int size();
    bool contains(T val);
    T operator[](int pos);
    T minimum();
    T median();
    void sort();
};
```

```
template <typename T>
ExtListe<T>::ExtListe(int size): Liste<T>(size){
     // Basisklassen-Konstruktor wird in Initialisiererliste
     // aufgerufen, nicht im Funktionsrumpf

    _sorted = false;
}

template <typename T>
bool ExtListe<T>::contains(T val) {
    for (int i = 0; i < _next; i++)
        // implizite Annahme: Templatetyp überlädt operator==
        if (Liste<T>::_values[i] == val)
            return true;

    return false;
}
```

```
template <typename T>
int ExtListe<T>::size(void) {
    return Liste<T>::_next;
}

template <typename T>
T ExtListe<T>::operator[](int pos) {
    return Liste<T>::getValueAt(pos);
}


// überschreiben der Funktionalität der Basisklasse: nach dem
// Einfügen eines Elements muss _sorted = false gesetzt werden
template <typename T>
ExtListe<T>::append(T val) {
    Liste<T>::append(val);
    _sorted = false;
}
```

Erweitern der Funktionalität

```
template <typename T>
T ExtListe<T>::minimum(void) {
    if (isEmpty())
        throw Exception("empty list");

    if (!_sorted)
        sort();
    return Liste<T>::_values[0];
}

template <typename T>
T ExtListe<T>::median(void) {
    // ..... analog zu minimum()
    return Liste<T>::_values[Liste<T>::_next / 2];
}
```

```
template <typename T>
void ExtListe<T>::sort(void) {
    for (int i = 0; i < Liste<T>::_next; i++) {
        for (int j = i + 1; j < Liste<T>::_next; j++) {
            if (Liste<T>::_values[i] > Liste<T>::_values[j]) {
                T tmp = Liste<T>::_values[i];

                Liste<T>::_values[i] = Liste<T>::_values[j];
                Liste<T>::_values[j] = tmp;
            }
        }
    }
    _sorted = true;
}
```

hier: Naives Sortierverfahren! In der Vorlesung *Algorithmen und Datenstrukturen* lernen Sie bessere Sortierverfahren kennen.

Implizite Annahmen über den Templatetyp:

- Die Vergleichsoperatoren `>` und `==` sowie
- der Zuweisungsoperator `=` müssen überladen sein!

Unsere Implementierung ist falsch:

- `_values` und `_next` sind in der Basisklasse `Liste<T>` als `private` deklariert und daher in der abgeleiteten Klasse nicht verfügbar.

→ die Attribute in der Basisklasse als `protected` deklarieren!

Anmerkungen:

- Der Konstruktor wird nicht geerbt.
- Der Destruktor wird nicht geerbt.

Eigenschaften abgeleiteter Klassen

- Alle `public`-Elemente der Basisklasse sind verfügbar.

  Übrigens: Die Menge der `public`-Elemente einer Klasse nennt man die Schnittstelle der Klasse.
- Alle `private`-Elemente bleiben auf jeden Fall nach außen verborgen, also auch einer abgeleiteten Klasse verborgen.
- Wollen wir Attribute oder Methoden zwar nach außen verbergen, aber abgeleiteten Klassen zur Verfügung stellen, müssen wir die Elemente mittels `protected` kennzeichnen.
- Weitere Elemente können der abgeleiteten Klasse hinzugefügt werden.
- Elemente der Basisklasse können überschrieben (also ersetzt), aber nicht gestrichen werden. (Das ist so nicht ganz richtig, siehe dazu den Abschnitt Schutztyp.)

→ Die abgeleitete Klasse leistet alles, was die Basisklasse kann und bietet darüberhinaus noch zusätzliche Funktionalitäten an.

Zugriffschutz `protected`

`protected`

- erlaubt den Zugriff auf Variablen und Methoden aus abgeleiteten Klassen, aber nicht von anderen Klassen.
- entspricht also
  - `public` für abgeleitete Klassen und
  - `private` für alle anderen Klassen.

→ Beim Entwurf der Klasse beachten, ob später Erweiterungen möglich sind.

Zugriff möglich durch ...	`private`	`protected`	`public`
eigene Memberfunktion	x	x	x
beliebige Funktion			x
befreundete Funktion	x	x	x
Memberfunktion einer beliebigen Klasse			x
Memberfunktion einer direkt abgeleiteten Klasse		x	x
Memberfunktion einer befreundeten Klasse	x	x	x

Schutztyp – Zugriffsrechte bei Ableitung

- Ableitung `public`: Die Zugriffsrechte der Basisklasse bleiben erhalten.

```
class ExtListe: public Liste
```

- Ableitung `protected`: Die `public`-Elemente der Basisklasse werden zu `protected`-Elementen der abgeleiteten Klasse.

```
class ExtListe: protected Liste
```

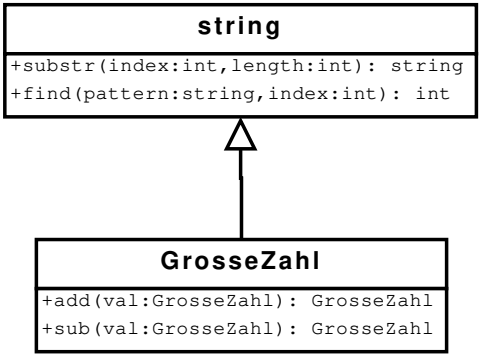
- Ableitung `private` (Standard, falls die Angabe fehlt): Die `public`- und die `protected`-Elemente der Basisklasse werden zu `private`-Elementen der abgeleiteten Klasse.

```
class ExtListe: private Liste
```

Wofür braucht man das?

Anmerkung: Im Normalfall ist nur die `public`-Ableitung sinnvoll! Aber die `private`-Ableitung wurde als Standard festgelegt.

Es sind `private`-Ableitungen vorstellbar:



- Mittels `GrosseZahl` sind beliebig lange Zahlen darstellbar.
- Durch die `private`- Ableitung verhindern wir, dass der Benutzer mit `string`-Methoden Unsinn machen kann.
- Schlechtes Design?

Schutztyp

Vererbung wird oft als ist-ein-Beziehung charakterisiert:

- Ein Tier ist ein Lebewesen.
- Ein Landtier ist ein Tier.
- Eine Katze ist ein Landtier.
- Ein Tiger ist eine Katze.

Diese Logik gilt aber nur für `public`-Ableitungen. Nur dann verfügen die Unterklassen über die Eigenschaften der Oberklasse.

Eine `GrosseZahl` ist eine Zahl, aber kein `string`, denn die `string`-Funktionalität steht nach der Ableitung nicht mehr zur Verfügung.

Konstruktoren und Destruktoren

Konstruktoren und Destruktoren werden nicht vererbt!

- Der Konstruktor der abgeleiteten Klasse ruft automatisch als erstes den Konstruktor der Basisklasse auf, entweder
  - explizit durch Angabe des Konstruktors in der Initialisiererliste
  - oder implizit durch Aufruf des Standard-Konstruktors.
- Erst wenn die Abarbeitung des Basis-Konstruktors beendet ist, wird der Rumpf des Konstruktors der abgeleiteten Klasse abgearbeitet.

Das hat zur Folge:

- Die Datenelemente der Basisklasse wurden bereits initialisiert, wenn die Abarbeitung der Befehle des Konstruktors der abgeleiteten Klasse beginnt.
- Die Methoden der Basisklasse können im Konstruktor der abgeleiteten Klasse aufgerufen werden.

- Der Aufruf der Destruktoren erfolgt in umgekehrter Reihenfolge: zuerst von der abgeleiteten Klasse, dann von der Basisklasse.
- Werden keine besonderen Maßnahmen getroffen, wird der Standard-Konstruktor der Basisklasse aufgerufen, d.h. dieser muss auch vorhanden sein!

  Soll ein anderer, selbstdefinierter Konstruktor der Basisklasse aufgerufen werden, so erfolgt dies mittels Initialisierungslisten.
- Argumente für den Konstruktor der Basisklasse werden in der Definition des Konstruktors der abgeleiteten Klassen angegeben.

Konstruktoren und Destruktoren

```
class Date {
    int _day, _month, _year;
    ....
    Date(int d, int m, int y) { .... }
    static Date getCurrentDate();
    ....
};

class DateTime : public Date {
    int _hour, _minute, _seconds;
    ....
    DateTime(int d, int m, int y, int ho, int mi, int se)
              : Date(d, m, y) {
        ....
    }
    static DateTime getCurrentDateTime();
    ....
};
```

Ausgabe? Was fällt bei der Zuweisung an `_alter` auf?

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

class Vater {
protected:
    int _alter;

public:
    Vater() {
        cout << "Standard-Konstruktor Vater\n";
        _alter = 35;
    }
    ~Vater() {
        cout << "Destruktor Vater" << endl;
    }
};
```

```
class Sohn: public Vater {
public:
    Sohn(int alter) {
        cout << "Konstruktor Sohn" << endl;
        _alter = alter;
    }
    ~Sohn() {
        cout << "Destruktor Sohn" << endl;
    }
};

int main(void) {
    Sohn s(12);
    return 0;
}
```

Ausgabe? Was fällt bei der Zuweisung an `_alter` auf?

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

class Mutter {
protected:
    int _alter;
public:
    Mutter() {
        cout << "Standard-Konstruktor Mutter\n";
        _alter = 30;
    }
    Mutter(int alter) {
        cout << "Konstruktor Mutter\n";
        _alter = alter;
    }
    ~Mutter() {
        cout << "Destruktor Mutter\n";
    }
};
```

```
class Tochter: public Mutter {
public:
    Tochter(int alter): Mutter(alter) {
        cout << "Konstruktor Tochter\n";
    }
    ~Tochter() {
        cout << "Destruktor Tochter\n";
    }
};

int main(void) {
    Tochter t(12);
    return 0;
}
```

Ausgabe? Was fällt bei der Zuweisung an `_alter` auf?

Syntax der Initialisierungslisten

```
Klasse::Klasse(parliste)
        : Basisklasse1(parliste1), Basisklasse2(parliste2),
          var1(paramX), var2(paramY), ... {
    ....
}
```

Mehrfachvererbung in C++ möglich: Klasse erbt von mehreren Basisklassen.

- `iostream` ist von `istream` und `ostream` abgeleitet.
- Wir könnten unsere Klasse `DateTime` von einer Klasse `Date` und von einer Klasse `Time` ableiten.

Wird hier nicht weiter betrachtet: Java, C# und andere Sprachen verzichten bewusst auf Mehrfachvererbung.

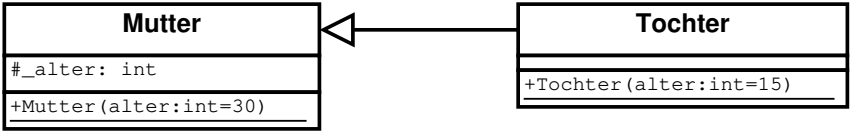
Konstruktoren und Destruktoren

- Wird für eine Basisklasse kein Konstruktor in der Initialisierungsliste angegeben, wird deren Standard- Konstruktor aufgerufen (der auch vorhanden sein muss!)
- Membervariablen werden in der Reihenfolge ihrer Deklaration in der Klasse initialisiert, denn der Nutzer kennt nur die Header-Datei, aber nicht die Implementierung!
- Anweisungen im Konstruktor werden zuletzt ausgeführt.
- Konstruktoren, Destruktoren, `operator=` sowie Freunde werden nicht vererbt!

Wir können Standardwerte für Parameter definieren:

```
Mutter(int alter = 30) {
    cout << "Konstruktor Mutter\n";
    _alter = alter;
}

Tochter(int alter = 15) :
        Mutter(alter) {
    cout << "Konstruktor Tochter\n";
}
```



Templates vs. Vererbung

In Java: Alle Klassen sind implizit Spezialisierung von der Klasse `Object`, auch selbstdefinierte Typen!

Wäre das in C++ auch so, könnte man unsere Liste auch ohne Templates wie folgt definieren:

```
class Liste {
protected:
    int _next, _size;
    Object *_values;

public:
    void append(Object value);
    Object getValueAt(int pos);
    ....
};
```

Vorteile? Nachteile?