

Vererbung, Polymorphie

Paul Raffer

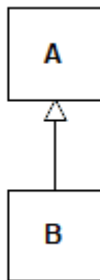
2021-04-11

Datenkapselung

| Zugriffs- modifikator | UML | Eigene Klasse | Abgeleitete Klasse | Außerhalb |
|----------------------------------|------------|--------------------------|-------------------------------|------------------|
| öffentlich | + | sichtbar | sichtbar | sichtbar |
| geschützt | # | sichtbar | sichtbar | nicht sichtbar |
| privat | - | sichtbar | nicht sichtbar | nicht sichtbar |

Vererbung

- ▶ Meist in Kombination mit Polymorphie
- ▶ Eigenschaften und Methoden der Basisklasse A werden in abgeleitete Klasse B übernommen
- ▶ Doppelter Code und Schreiarbeit werden vermieden
- ▶ UML: Pfeil von abgeleiteter Klasse zu Basisklasse



Datenkapselung im Rahmen der Vererbung

Sichtbarkeit in ...

| Basisklasse | abgeleiteter Klasse (erbt ... von Basisklasse) | | |
|--------------|--|---------------|---------------|
| | öffentlich | geschützt | privat |
| öffentlich → | öffentlich | geschützt | privat |
| geschützt → | geschützt | geschützt | privat |
| privat → | nicht vererbt | nicht vererbt | nicht vererbt |

- ▶ öffentlich: “ist-ein”-Beziehung
- ▶ geschützt: “ist-implementiert-mit”-Beziehung
- ▶ privat: “ist-implementiert-mit”-Beziehung
- ▶ Layering: “hat-ein”-/“ist-implementiert-mit”-Beziehung

Beispiel:

```
struct Point2d {  
    int x;  
    int y;  
};
```

Beispiel:

```
struct Point2d {  
    int x;  
    int y;  
};
```

Lösung ohne Vererbung:

```
struct Point3d {  
    int x;  
    int y;  
    int z;  
};
```

Beispiel:

```
struct Point2d {  
    int x;  
    int y;  
};
```

Lösung ohne Vererbung:

```
struct Point3d {  
    int x;  
    int y;  
    int z;  
};
```

Lösung mit öffentlicher Vererbung:

```
struct Point3d : Point2d {  
    int z;  
};
```

Beispiel:

```
struct Point2d {  
    int x;  
    int y;  
};
```

Lösung ohne Vererbung:

```
struct Point3d {  
    int x;  
    int y;  
    int z;  
};
```

Lösung mit öffentlicher Vererbung:

```
struct Point3d : Point2d {  
    int z;  
};
```

```
Point2d* point = new Point3d; // Möglich, aber nicht erwünscht!
```


Beispiel:

```
struct Point2d {  
    int x;  
    int y;  
};
```

Lösung ohne Vererbung:

```
struct Point3d {  
    int x;  
    int y;  
    int z;  
};
```

Lösung mit öffentlicher Vererbung:

```
struct Point3d : Point2d {  
    int z;  
};
```

Point2d* point = new Point3d; // Möglich, aber nicht erwünscht!

Lösung mit privater Vererbung:

```
struct Point3d : private Point2d {  
    int z;  
    // Eigenschaften aus Point2d wieder öffentlich machen:  
    public: using Point2d::x;  
    public: using Point2d::y;  
};
```

Schnittstellenvererbung

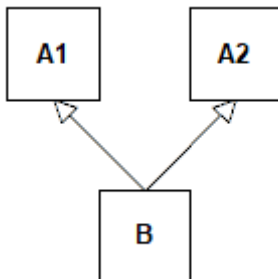
- ▶ Nur Methodensignatur, aber keine Standardimplementierung, wird vererbt
- ▶ Java: Interface
- ▶ C++: abstrakte Klasse, die nur rein virtuelle Methoden enthält

Implementierungsvererbung

- ▶ Methodensignatur und Standardimplementierung werden vererbt
- ▶ Standardimplementierung kann aber von abgeleiteter Klasse überschrieben werden

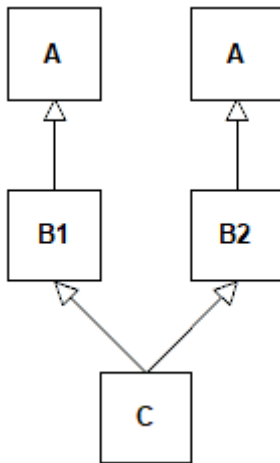
Mehrfachvererbung

- ▶ Eine abgeleitete Klasse erbt von mehreren Basisklassen
- ▶ Mehrfachinterfacevererbung problemlos möglich
- ▶ Mehrfachimplementierungsvererbung führt oft zu fehleranfälligem und unübersichtlichem Code



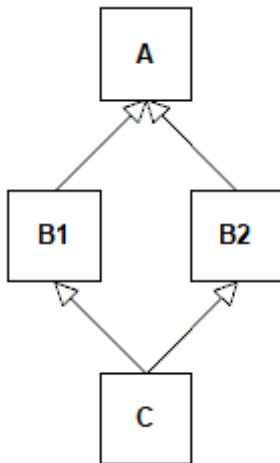
Diamond-Problem

- ▶ Eine abgeleitete Klasse erbt über mehr als einen Pfad von derselben Basisklasse
- ▶ Eigenschaften und Methoden werden mehrfach vererbt



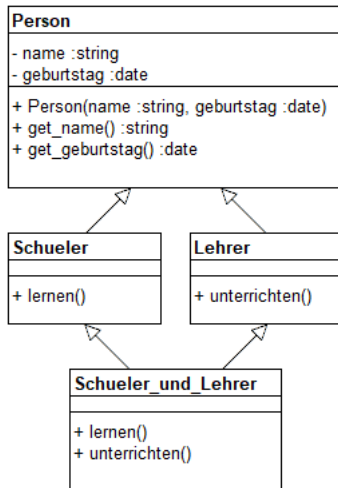
C++-Lösung: Virtuelle Vererbung

- ▶ B1 und B2 erben von A virtuell
- ▶ abgeleitete Klassen teilen sich eine gemeinsame Instanz



Beispiel: Schüler/Lehrer

- ▶ Schueler und Lehrer erben von Person
- ▶ Es gibt auch Schüler, die anderen Schülern Nachhilfe geben



Abstrakte Klassen

- ▶ Von abstrakten Klassen können keine Objekte erzeugt werden
- ▶ Es wird von solchen Klassen nur geerbt und von der abgeleiteten Klasse ein Objekt erzeugt.

Abstrakte Klassen

- ▶ Von abstrakten Klassen können keine Objekte erzeugt werden
- ▶ Es wird von solchen Klassen nur geerbt und von der abgeleiteten Klasse ein Objekt erzeugt.

Java:

```
abstract class Abstrakte_klasse {  
    public:  
        void virtuelle_methode();  
};
```

C++:

```
class Abstrakte_klasse {  
    public:  
        virtual void rein_virtuelle_methode() = 0;  
};
```

Abstrakte Klassen

- ▶ Von abstrakten Klassen können keine Objekte erzeugt werden
- ▶ Es wird von solchen Klassen nur geerbt und von der abgeleiteten Klasse ein Objekt erzeugt.

Java:

```
abstract class Abstrakte_klasse {  
    public:  
        void virtuelle_methode();  
};
```

C++:

```
class Abstrakte_klasse {  
    public:  
        virtual void rein_virtuelle_methode() = 0;  
};
```

```
Abstrakte_klasse* = new Abstrakte_klasse{};
```

Abstrakte Klassen

- ▶ Von abstrakten Klassen können keine Objekte erzeugt werden
- ▶ Es wird von solchen Klassen nur geerbt und von der abgeleiteten Klasse ein Objekt erzeugt.

Java:

```
abstract class Abstrakte_klasse {  
public:  
    void virtuelle_methode();  
};
```

C++:

```
class Abstrakte_klasse {  
public:  
    virtual void rein_virtuelle_methode() = 0;  
};
```

```
Abstrakte_klasse* = new Abstrakte_klasse{};
```

```
class Abgeleitete_klasse : public Abstrakte_klasse {  
public:  
    virtual void rein_virtuelle_methode() override { /*...*/ }  
    // ...  
};
```

```
Abstrakte_klasse* = new Abgeleitete_klasse{};
```

Entgeltige Klassen

- ▶ Kann keine Basisklasse sein

Polymorphie

- ▶ Gleiches Interface für Objekte von verschiedenen Typen
- ▶ Gegenteil: Monomorphie

| | universell unendlich viele Typen eine Implementierung | Ad-hoc endliche Anzahl an Typen unterschiedliche Implementierungen |
|--|--|---|
| dynamisch Laufzeit langsamer | Inklusionspolymorphie/ Vererbungspolymorphie | |
| statisch Kompilzeit schneller | parametrische Polymorphie | Überladung, Coercion |

- ▶ Statisch: es steht zur Kompilzeit fest, welche Funktion aufgerufen wird
- ▶ Dynamisch: es wird erst zur Laufzeit bestimmt, welche Funktion aufgerufen wird

universelle Polymorphie

- ▶ Gleiches Interface für unendlich viele Typen
- ▶ Eine Implementierung
- ▶ “echte Vielgestaltigkeit”
- ▶ Inklusionspolymorphie
- ▶ Vererbungspolymorphie
- ▶ Parametrische Polymorphie

Inklusionspolymorphie

- ▶ Liskovsches Substitutionsprinzip ist erfüllt
 - ▶ Objekt des Typen A kann problemlos, durch Objekt des Typen B ersetzt werden

Vererbungspolymorphie

- ▶ Dynamisch
- ▶ Kann Inklusionspolymorphie ausdrücken
- ▶ Sollte auch das Liskovsche Substitutionsprinzip befolgen, muss aber nicht
- ▶ Virtuelle Methoden

Beispiel1: Fahrzeug (Vererbungspolymorphie)

```
class Fahrzeug {  
    // ...  
    public: virtual void rechts_abbiegen() = 0;  
    // ...  
};  
  
class Auto {  
    // ...  
    public: virtual void rechts_abbiegen()  
    {  
        // Implementierung fuer Autos  
    }  
    // ...  
}  
  
class Fahrrad {  
    // ...  
    public: virtual void rechts_abbiegen()  
    {  
        // Implementierung fuer Fahrraeder  
    }  
    // ...  
};  
  
auto main() -> int  
{  
    Fahrzeug* fahrzeug = new Auto;  
    fahrzeug->rechts_abbiegen();  
}
```

Beispiel1: Fahrzeug (Mehrfache Auswahl)

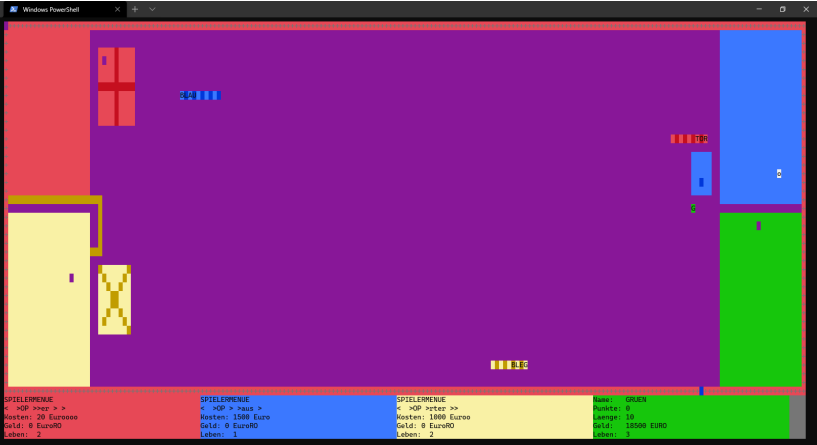
```
enum Fahrzeugtyp {
    Auto,
    Fahrrad,
};

struct Fahrzeug {
    Fahrzeugtyp typ;
    // ...
};

void rechts_abbiegen(Fahrzeug* fahrzeug)
{
    switch (fahrzeug->typ) {
        case Auto:
            // Implementierung fuer Autos
            break;
        case Fahrrad:
            // Implementierung fuer Fahrraeder
            break;
        default:
            // Fehler: Ungueltiger Typ!
            break;
    }
}

auto main() -> int
{
    Fahrzeug fahrzeug{Auto};
    rechts_abbiegen(&fahrzeug);
}
```

Beispiel2: Snake



Beispiel2: Snake

- ▶ Unterschiedliche Arten von Gebäuden (Kanonen, Mauern, Geldlager, Krankenhäuser, ...)
- ▶ Unterschiedliche Arten von Punkten (normale Punkte, Geld, Leben, ...)

Beispiel2: Mein Code (nicht nachmachen!)

```
// ...
```

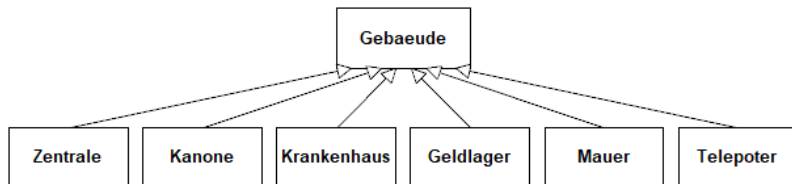
```
gebaeude_verschieben(spielfeld , spieler , spieler.at(s).gebaeude.zentrale , s , v);  
gebaeude_verschieben(spielfeld , spieler , spieler.at(s).gebaeude.kanone , s , v);  
gebaeude_verschieben(spielfeld , spieler , spieler.at(s).gebaeude.krankenhaus , s , v);  
gebaeude_verschieben(spielfeld , spieler , spieler.at(s).gebaeude.geldlager , s , v);  
gebaeude_verschieben(spielfeld , spieler , spieler.at(s).gebaeude.mauer , s , v);  
gebaeude_verschieben(spielfeld , spieler , spieler.at(s).gebaeude.teleporter , s , v);
```

```
// ...
```

```
gebaeude_gameover(spielfeld , spieler , spieler.at(sp).gebaeude.kanone , s , sp);  
gebaeude_gameover(spielfeld , spieler , spieler.at(sp).gebaeude.krankenhaus , s , sp);  
gebaeude_gameover(spielfeld , spieler , spieler.at(sp).gebaeude.geldlager , s , sp);  
gebaeude_gameover(spielfeld , spieler , spieler.at(sp).gebaeude.mauer , s , sp);  
gebaeude_gameover(spielfeld , spieler , spieler.at(sp).gebaeude.teleporter , s , sp);
```

```
// ...
```

Beispiel2: Mit Vererbungspolymorphie



```
// ...
```

```
for (auto & g : spieler.at(s).gebaeude) {  
    // Welchen Typ g hat, und somit auch welche Methode 'verschieben'  
    // aufgerufen wird, wird erst zur Laufzeit bestimmt.  
    g.verschieben(spielfeld, spieler, s, v);  
}
```

```
// ...
```

```
for (auto & g : spieler.at(sp).gebaeude) {  
    g.gameover(spielfeld, spieler, s, sp);  
}
```

```
// ...
```

Kreis-Ellipse-Problem

- ▶ Problem:
 - ▶ Basisklasse 'Form' hat die Methoden 'zeichnen' und 'flaeche'
 - ▶ Ellipsen und Kreise sollen erstellt werden können
 - ▶ 'Kreis' erbt von 'Ellipse' erbt von 'Form'
 - ▶ Bei Kreisen können nicht beide Dimensionen unabhängig voneinander skaliert werden
 - ▶ Liskovsches Substitutionsprinzip nicht erfüllt
- ▶ Lösungsvorschläge:
 - ▶ Fehler bei Größenänderung
 - ▶ Ellipse erbt von Kreis ab
 - ▶ Keine Klasse Kreis
 - ▶ Keine Vererbungsbeziehung zwischen Ellipse und Kreis
 - ▶ Einführen neuer Basisklasse
- ▶ Keiner der Lösungsvorschläge ist ideal
- ▶ Nicht jede "ist-ein"-Beziehung sollte durch öffentliche Vererbung dargestellt werden!

Parametrische Polymorphie (TODO)

- ▶ Statisch

Ad-hoc-Polymorphie

- ▶ Gleiche Schnittstelle für begrenzte Anzahl an bestimmten Typen
- ▶ Eine Implementierung pro Typ
- ▶ Coercion
- ▶ Überladung

Coercion

- ▶ Statisch
- ▶ Implizite Typumwandlung

```
auto main() -> int
{
    // Implizite Umwandlung von int{5} zu double{5.0}
    // Output: 8.2
    std::cout << 5 + 3.2;
}
```

Coercion - Konvertierungskonstruktor

```
class Bruch {
    int zaehler_;
    int nenner_;
    // ...
};

// Konvertierungskonstruktor (int to Bruch)
Bruch::Bruch(int zaehler = 0, int nenner = 1)
    : zaehler_{zaehler}, nenner_{nenner} {}

void print(Bruch const & bruch)
{
    std::cout << '(' << bruch.zaehler() << '/' << bruch.nenner() << ')';
}

auto main() -> int
{
    // implizite Cast von int{42} zu Bruch{42, 1}
    // Output: (42/1)
    print(42);
}
```

Coercion - Konvertierungsoperator

```
Bruch::operator double() // Konvertierungsoperator (Bruch zu double)
{
    return double{this->zaehler_} / double{this->nenner_};
}

auto main() -> int
{
    // Implizite Umwandlung von Bruch{1, 4} zu double{0.25}
    // Output: 4.5
    std::cout << Bruch{1, 4} + 4.25;
}
```

Überladung

- ▶ Statisch
- ▶ Mehrere Funktionen haben den gleichen Namen
- ▶ Operatorüberladung

```
void reset(int& i)
{
    i = 0;
}

void reset(std::vector<int>& vec)
{
    for (int& i : vec) {
        reset(i);
    }
}
```

Quellen

- ▶ FSST-Mitschrift der 2. und 3. Klasse
- ▶ <https://www.youtube.com/watch?v=7EmboKQH81M>
- ▶ https://www.ics.uci.edu/~jajones/INF102-S18/readings/05_stratchey_1967.pdf
- ▶ <https://invidious.snopyta.org/watch?v=7PfNo-FMIf0>
- ▶ <https://invidious.snopyta.org/watch?v=uTxRF5ag27A>
- ▶ <https://soundcloud.com/lambda-cast>
- ▶ [https://de.wikipedia.org/wiki/Polymorphie_\(Programmierung\)](https://de.wikipedia.org/wiki/Polymorphie_(Programmierung))
- ▶ <https://lec.inf.ethz.ch/ifmp/2019/slides/lecture14.handout.pdf>
- ▶ <https://courses.cs.washington.edu/courses/cse331/11wi/lectures/lect06-subtyping.pdf>
- ▶ [https://de.wikipedia.org/wiki/Vererbung_\(Programmierung\)](https://de.wikipedia.org/wiki/Vererbung_(Programmierung))
- ▶ <https://wiki.haskell.org/Polymorphism>
- ▶ <http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf>
- ▶ C++ Templates: The Complete Guide
- ▶ C++: Das umfassende Handbuch
- ▶ Grundkurs C++
- ▶ C++: Die Sprache der Objekte
- ▶ Exceptional C++: 47 engineering puzzles, programming problems, and solutions
- ▶ Effektiv C++. 50 Specific Ways to Improve Your Programs and Designs
- ▶ <https://invidious.snopyta.org/watch?v=HddFGPTAmtU>
- ▶ <https://www.bfilipek.com/2020/04/variant-virtual-polymorphism.html>
- ▶ <https://de.wikipedia.org/wiki/Diamond-Problem>
- ▶ <https://stackoverflow.com/questions/66983156/is-coercion-static-or-dynamic-polymorphism>