Kapitel 15 - Polymorphie

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Einleitung

- Bisher: Dank Vererbung akkurate Abbildung der Realität in Klassenhierarchie
- Jetzt: Polymorphie 🤔

Vererbung: Status Quo

- Zeiger einer Basisklasse wird Objekt von Kindklasse zugewiesen

```
1  Base * b = new Child(); // bzw. mit unique_ptr
2  b->do_stuff();
```

- Bisheriges Problem: Bindung der Methoden geschieht während des Übersetzens

  → „Methode `do_stuff()` aus Oberklasse `Base` wird aufgerufen"

Vererbung: Status Quo

- Bisheriges Problem: Bindung der Methoden an Objekte geschieht während des Übersetzens
- Jetzt: Polymorphie/Polymorphismus

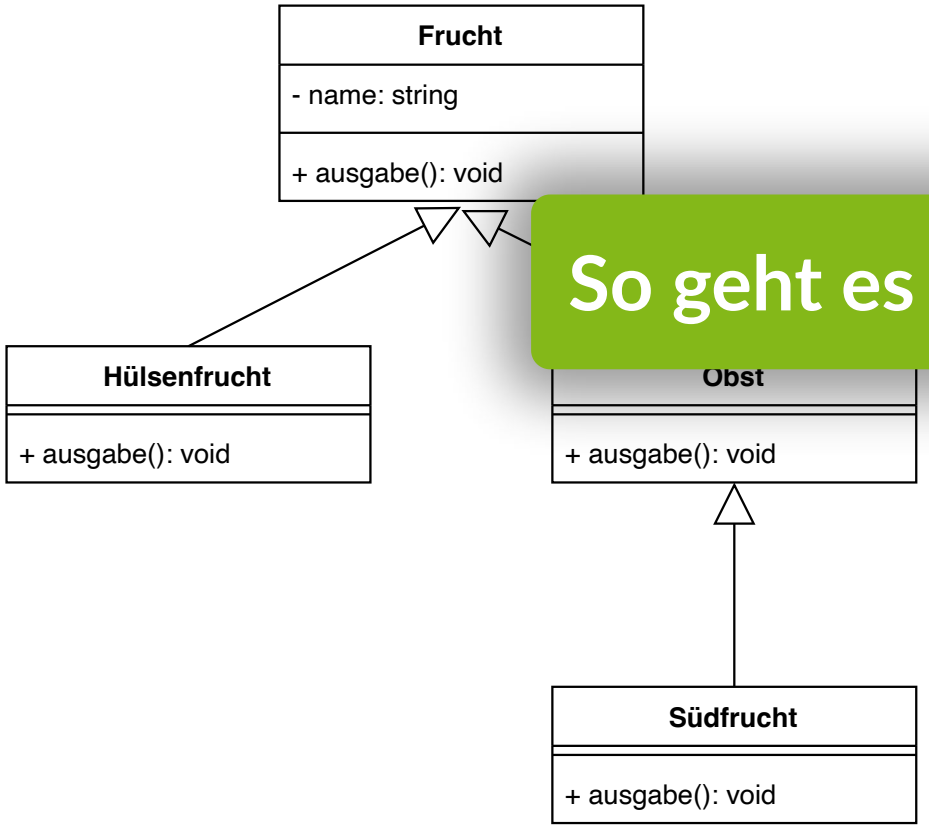
  → Vielgestaltigkeit
  - Technik zur Bindung von Methoden zur Laufzeit (dynamische Bindung)
  - Erst zur Laufzeit steht fest, welche Methode aufgerufen wird
  - Heißt daher auch *Dynamic Dispatch*

cpp Run ▶

```cpp
#include <iostream>
using std::cout, std::endl;

class Base {
public:
    void print() { cout << "Base" << endl; }
};

class Child: public Base {
public:
    void print() { cout << "Child" << endl; }
};

int main() {
    Base * b = new Child();
    b->print();
    delete b;
}
```

Beispiel - Polymorphie

- Ziel: dynamische Auswahl von `ausgabe()`



Polymorphismus bei Früchten

So geht es natürlich noch nicht!

```
1  class Frucht {
2  protected:
3      std::string name;
4  public:
5      void ausgabe();
6  };
7
   rucht: public Frucht {
   be();
11 };
12 class Obst: public Frucht {
13 public:
14     void ausgabe();
15 };
16 class Suedfrucht: public Obst {
17 public:
18     void ausgabe();
19 };
```

Virtuelle Methoden

- Methoden können dynamisch gebunden werden

  → Auswahl zur Laufzeit

- Schlüsselwort `virtual` in C++
- Virtuelle Methoden können überschrieben werden

  → Die Auswahl erfolgt nun für diese Methode dynamisch

```
1  class Frucht {
2  protected:
3      std::string name;
4  public:
5      virtual void ausgabe();
6  };
7
8  class Huelsenfrucht: public Frucht {
9  public:
10     void ausgabe();
11 };
12 class Obst: public Frucht {
13 public:
14     void ausgabe();
15 };
16 class Suedfrucht: public Obst {
17 public:
18     void ausgabe();
19 };
```

Beispiel - Virtuelle Methode

cpp Run ▶

```cpp
#include <iostream>

using std::cout, std::endl;

class Frucht {
public:
    virtual void ausgabe() {
        cout << "F" << endl;
    };
    virtual ~Frucht() {};
};

class Obst: public Frucht {
public:
    void ausgabe() {
```

Destruktor mit `virtual`

- Konstruktoren können **nicht** virtuell sein
- Destruktoren hingegen **sollten** virtuell sein
  - Ansonsten: statische Festlegung während des Übersetzens
  - Resultat: Aufruf des Destruktors der Basisklasse (und ggf. undefiniertes Verhalten)
- Empfehlung: Markiert Destruktor von Basisklassen **immer** als `virtual`

cpp Run

```cpp
#include <iostream>
using std::cout, std::endl;

class Base {
public:
    virtual ~Base() {
        cout << "Base" << endl;
    }
};
class Child : public Base {
public:
    ~Child() {
        cout << "Child" << endl;
    }
};

int main() {
    Base * b = new Child();
    delete b;
```

Explizites Überschreiben – `override`

- Überschreiben einer virtuellen Methode in Kindklasse mit `override` **explizit**
- Syntax: `void print() override;`
- Erzeugt Compiler-Warnung, wenn nicht überschrieben wird
- Verwendung ist (leider) nicht zwingend von C++ gefordert 😕
- Erhöht die Lesbarkeit aber ungemein. Nutzt es!

```
1  class Frucht {
2  protected:
3      std::string name;
4  public:
5      virtual void ausgabe();
6  };
7
8  class Obst: public Frucht {
9  public:
10     void ausgabe() override;
11 };
12
13 class Suedfrucht: public Obst {
14 public:
15     void ausgabe() override;
16 };
```

Das letzte Wort – `final`

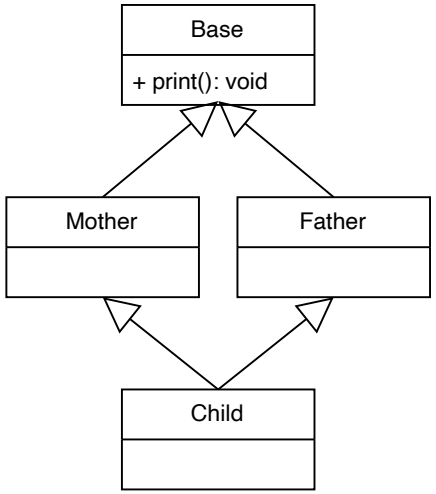
- Analog zu `override`: Schlüsselwort `final`
- Verhindert weiteres Überschrieben einer Methode
- Erzeugt Compiler-Fehler bei Versuch zu überschreiben

```
1  class Frucht {
2  protected:
3      std::string name;
4  public:
5      virtual void ausgabe();
6  };
7
8  class Obst: public Frucht {
9  public:
10     void ausgabe() final;
11 };
12
13 class Suedfrucht: public Obst {
14 public:
15     void ausgabe() override; // ERROR
16 };
```

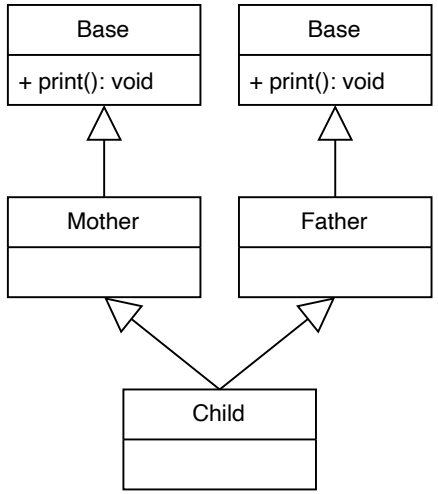
Diamond-Shape-Problem

- Das Diamond-Shape-Problem bezeichnet eine spezielle Konstellation der Vererbung
  - Zwei Eltern erben von einer gemeinsamen Klasse und aus beiden Eltern wird eine Kindklasse abgeleitet
  - Das Klassendiagramm sieht einer Raute bzw. Diamanten ähnlich (→ Name)

Diamond-Shape



Tatsächliche Vererbungshierarchie



- Problem: Beide Elternteile haben bei der regulären Mehrfachvererbung jeweils eine eigene Instanz von `Base`

Diamond-Shape-Problem

- Problem: Bei Aufruf ist unklar, welches `print()` verwendet werden soll → Error
  - `Mother::print(); // Dies?`
  - `Father::print(); // Oder dies?`
- Lösung: Bei der Klassendeklaration der Eltern virtuell von `Base` ableiten
  - Erzeugt eine geteilte Basisklasse anstatt zwei separaten Instanzen

cpp Run ▶

```cpp
# include <iostream>
using std::cout, std::endl;

class Base {
public:
    virtual void print() {
        cout << "Base" << endl;
    }
};

class Mother: virtual public Base {};
class Father: virtual public Base {};
class Child : public Mother, public Father {};

int main() {
    Child c;
    c.print(); // Ok mit virtueller Ableitung
}
```

Rein virtuelle Methoden

- Ebenfalls möglich: **rein virtuelle Methoden** zu deklarieren *(pure virtual)*
  - In C++: `virtual void print() = 0; // = 0 deklariert Funktion als pure`
- Bedeutung: Implementierung in **dieser** Klasse nicht vorhanden
- Folgen
  - Erzeugen einer Instanz der Basisklasse nicht möglich
  - Methode **muss** in allen abgeleiteten Klassen implementiert werden
- Klassen mit rein virtuellen Methoden heißen abstrakte Klassen
  - Häufige Abkürzung: *Abstract Base Class* (*ABC*)
  - Alternative Bezeichung: Interface (wird oft synonym genutzt)

Beispiel - rein virtuelle Methoden

- Unterschiede zu regulären Klassen
  - Abstrakte Klassen können **nicht initialisiert werden**
  - Dienen als Vorlage zur Konstruktion anderer Klassen
  - Definiert Set von Methoden, das in allen abgeleiteten Klassen verfügbar und implementiert ist

cpp Run ▶

```cpp
# include <iostream>
using std::cout, std::endl;

class Form {
public:
    virtual void print_area() = 0;
};

class Quadrat : public Form {
    double x_;
    double y_;
public:
    Quadrat(double x, double y) : x_(x), y_(y) {}
    void print_area() override {
        cout << x_ * y_ << endl;
    }
};
```

Vtables - Ein Blick hinter die Kulissen

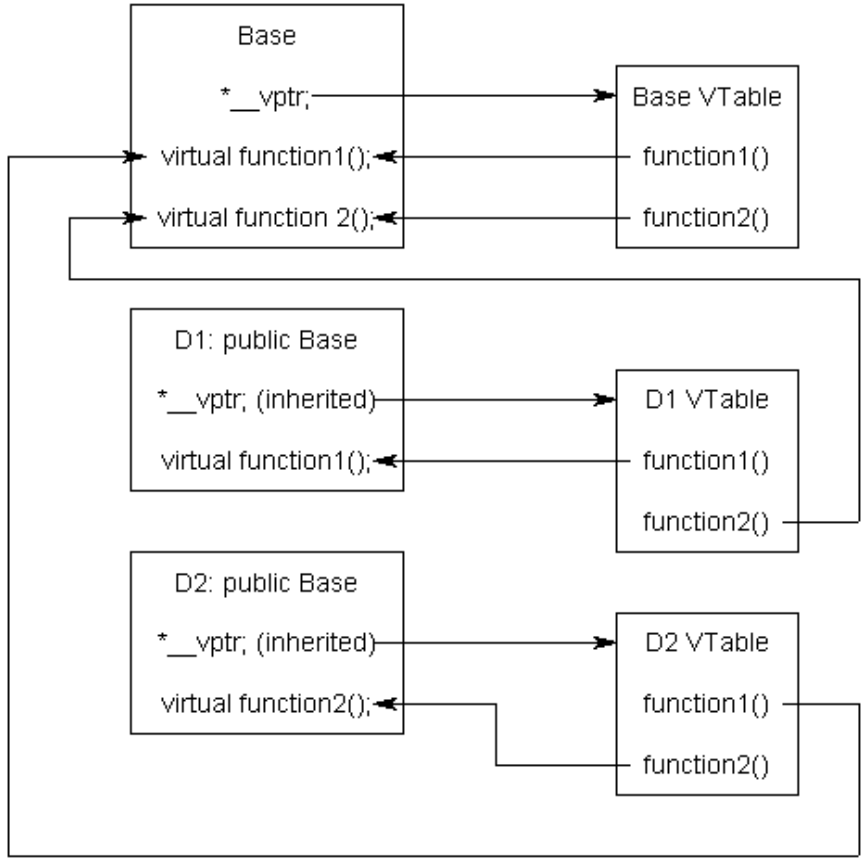
- Die wichtigsten Punkte der Polymorphie sind damit abgeschlossen ✅
- Abschließend ein Blick hinter die Kulissen: **Vtables**
- *Vtables* steht für Virtuelle Tabellen
- **Jede** Klasse mit virtuellen Funktionen bekommt eine **zusätzliche** Member-Variable

  → Zeiger auf Tabelle mit Funktionen
- Achtung: *Vtables* sind streng genommen nicht im C++-Standard definiert
  - Snd ein Implementierungsdetail
  - Allerdings sehr weit verbreitet (De-Facto-Standard)

Vtables

- Einfache Umsetzung
  - Compiler erstellt *Vtable*
  - Fügt automatisch neuen Zeiger auf Vtable ein – hier: `__vptr`
- Zur Laufzeit: Nachschlagen in der Tabelle
- Zeiger in Tabelle verweist auf konkrete Implementierung in Klasse
- Mehrkosten: Einige Dereferenzierungen von Zeigern

  → Ziemlich schnell



© *Alex Pomeranz (learncpp.com)*

Beispiel - *Vtables* im Clang-Compiler

- Code und Analyse-Ausgabe des Clang-Compilers
  - Erzeugt mit: `clang++ -Xclang -fdump-vtable-layouts vtable.cpp`

```cpp
#include <iostream>
using std::cout, std::endl;

class Frucht {
   public:
    virtual void ausgabe() { cout << "F" << endl; }
};

class Obst : public Frucht {
   public:
    void ausgabe() override { cout << "O" << endl; }
};

class Suedfrucht : public Obst {
   public:
    void ausgabe() override { cout << "SF" << endl; }
```

```
Vtable for 'Frucht' (3 entries).
   0 | offset_to_top (0)
   1 | Frucht RTTI
       -- (Frucht, 0) vtable address --
   2 | void Frucht::ausgabe()

VTable indices for 'Frucht' (1 entries).
   0 | void Frucht::ausgabe()

Vtable for 'Obst' (3 entries).
   0 | offset_to_top (0)
   1 | Obst RTTI
       -- (Frucht, 0) vtable address --
       -- (Obst, 0) vtable address --
   2 | void Obst::ausgabe()

VTable indices for 'Obst' (1 entries).
   0 | void Obst::ausgabe()

Vtable for 'Suedfrucht' (3 entries).
   0 | offset_to_top (0)
   1 | Suedfrucht RTTI
       -- (Frucht, 0) vtable address --
```

Ausblick - Statische Reflexion

- Bisher nicht besprochen: Reflexion
- Eigenschaft vieler OOP-fähigen Programmiersprachen, z. B. Java
- Bezeichnet Fähigkeit, Informationen über sich selbst abzurufen

  → z.B. eigener Typ oder Member inkl. Namen

```java
// In Java möglich:

public class Person {
    private String name;
}
// ...

Object p = new Person();
if (p.getClass() == Person.class) {
// Weitere Verarbeitung von Person
}
```

- In C++ geht das leider (noch) nicht 😕
- Kommender Standard (C++26 oder 29) wird statische Reflexion erlauben; genauer Zeitplan fehlt aber

Zusammenfassung

- Modellierung der Realität mittels Klassenhierarchie – siehe Frucht-Beispiel
- Vorteile von Polymorphie
  - Verwaltung von Objekten einer Basisklasse
  - Dennoch Zugriff auf Verhalten der Unterklasse
- Realisierung von Polymorphie mit Schlüsselwort `virtual`
- Erzwingen der Implementierung mittels abstrakter Klassen

  → Methoden als *pure virtual* definieren
- Implementierung mittels *vtables*

Kapitel 15 - Polymorphie