

Programmierpraktikum C und C++

Vererbung und Polymorphie



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Roland Kluge

roland.kluge@es.tu-darmstadt.de

ES Real-Time Systems Lab

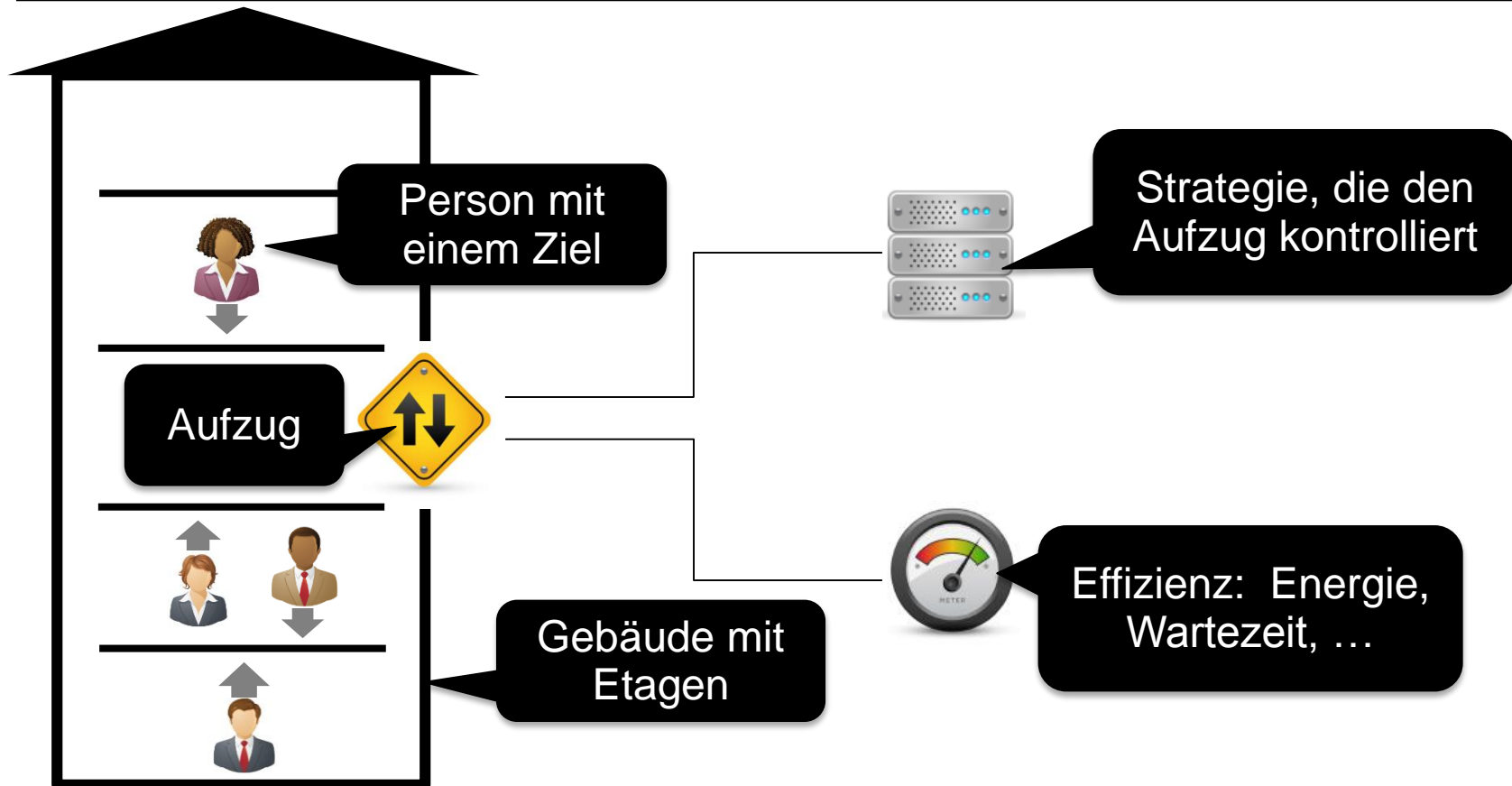
Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

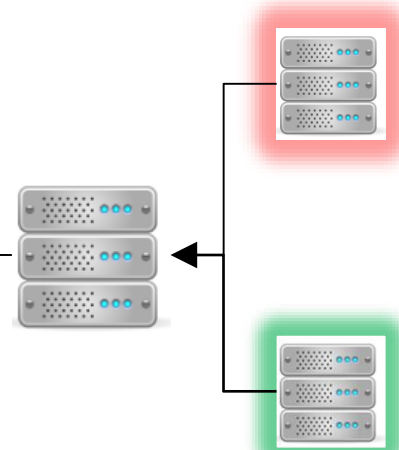
www.es.tu-darmstadt.de

Was ist Polymorphie?



Was ist Polymorphie?


Der Code im Aufzug, der die Strategie verwendet, soll sich nicht ändern, nur weil eine andere Strategie eingesetzt wird.
(Separation of Concerns)



Unterschiedliche Strategien können ergänzt und verwendet werden (**Erweiterbarkeit**). Die richtige Methode wird „magisch“ aufgerufen!



Lösung ohne Polymorphie



```
void Elevator::moveToNextFloor(int strategy){
    switch(strategy){
        case 0:
            cout << "Choose next floor to minimize energy."
                << endl;
            break;
        case 1:
            cout << "Choose next floor to minimize waiting time."
                << endl;
            break;
        // and so on ...
    }
}
```

„Dispatch“ geschieht von Hand
mit Hilfe einer „Tabelle“

Für jede neue Strategie muss die Logik
hier (und eventuell an anderen Stellen)
erweitert werden!
(Fluch des switch-case)

Lösung mit Polymorphie



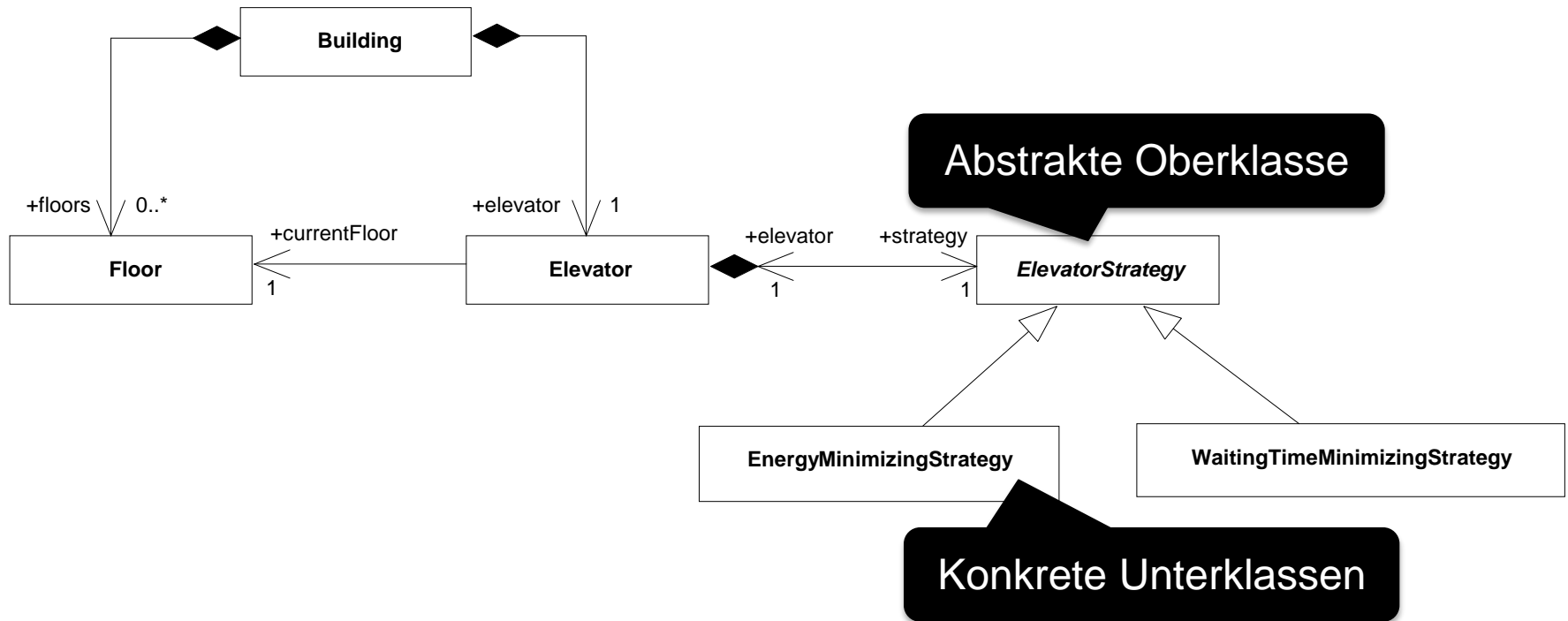
```
void Elevator::moveToNextFloor(){  
    currentFloor = strategy->next(this);  
}
```

Konkrete Strategie wird bei der Erzeugung des Aufzugs gesetzt.

Dieser Code behandelt die Strategie polymorph und muss für neue Strategien nicht verändert werden!



Aufzugsimulation (reloaded)



Intermezzo

Nochmal – was ist der Vorteil von Polymorphie?
Wie kann das so wichtig sein wenn z.B. C das nicht
unterstützt (und C doch so weitverbreitet ist)?!

Was hat Polymorphie mit Vererbung zu
tun? Geht es auch ohne Vererbung?



Floor.hpp

```
class Floor {  
public:  
    Floor(int number);  
    Floor(const Floor& floor);  
    ~Floor();  
  
    inline int getNumber() const {  
        return number;  
    }  
  
    inline void setNumber(int n) {  
        number = n;  
    }  
  
private:  
    int number;  
};
```

Kleine Methoden können
inline definiert werden!

[DE]  /Inline-Ersetzung

Floor.cpp

```
#include "Floor.hpp"  
  
Floor::Floor(int number):  
    number(number) {  
    cout << "Floor(): "  
        << "Creating floor ["  
        << number  
        << "]" << endl;  
}  
  
Floor::Floor(const Floor& floor):  
    number(floor.number) {  
    cout << "Floor(const Floor&): "  
        << "Copying floor ["  
        << floor.number  
        << "]" << endl;  
}  
  
Floor::~~Floor() {  
    cout << "~Floor(): "  
        << "Destroying floor ["  
        << number  
        << "]" << endl;  
}
```




ElevatorStrategy

Vorausdeklaration (statt `#include`), um zyklische Abhängigkeit zu vermeiden

```
ElevatorStrategy.hpp  
  
#include <boost/shared_ptr.hpp>  
#include "Floor.hpp"  
  
class Elevator;  
  
class ElevatorStrategy {  
public:  
    ElevatorStrategy();  
    ~ElevatorStrategy();  
  
    const Floor*  
    next(const Elevator *elevator) const;  
};  
  
typedef  
boost::shared_ptr<ElevatorStrategy>  
ElevatorStrategyPtr;  
  
typedef  
boost::shared_ptr<const ElevatorStrategy>  
ConstElevatorStrategyPtr;
```

In der Impl-Datei ist dies aber kein Problem!

ElevatorStrategy.cpp

```
#include "ElevatorStrategy.hpp"  
#include "Elevator.hpp"  
  
using namespace std;  
  
ElevatorStrategy::ElevatorStrategy() {  
    cout << "ElevatorStrategy(): "  
        << "Creating basic strategy"  
        << endl;  
}  
  
ElevatorStrategy::~~ElevatorStrategy() {  
    cout << "~ElevatorStrategy(): "  
        << "Destroying basic strategy"  
        << endl;  
}  
  
const Floor*  
ElevatorStrategy::next(const Elevator *elevator) const {  
    cout << "ElevatorStrategy::next(...): "  
        << "Using basic strategy ..."  
        << endl;  
  
    return elevator->getCurrentFloor();  
}
```

Sinnvolle Strategien entwickeln
wir in der Übung ☺



Elevator



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Elevator.hpp

```
#include "ElevatorStrategy.h"
#include "Floor.hpp"

class Elevator {
public:
    Elevator(const Floor*,
             ConstElevatorStrategyPtr);
    ~Elevator();

    inline const Floor* getCurrentFloor() const {
        return currentFloor;
    }

    void moveToNextFloor();

private:
    const Floor *currentFloor;
    ConstElevatorStrategyPtr strategy;
};
```

Typen ohne Namen
auch möglich

const Floor* und nicht **const Floor&**,
da der Zeiger sich ändert (aber nicht das
Objekt worauf gezeigt wird!)

Elevator.cpp

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

#include "Elevator.hpp"

Elevator::Elevator(const Floor *currentFloor,
                   ConstElevatorStrategyPtr strategy):
    currentFloor(currentFloor), strategy(strategy) {
    cout << "Elevator(): "
         << "Creating elevator." << endl;
}

Elevator::~Elevator(){
    cout << "~Elevator(): "
         << "Destroying elevator." << endl;
}

void Elevator::moveToNextFloor(){
    cout << "Elevator::moveToNextFloor(): "
         << " Polymorphic call to strategy." << endl;

    currentFloor = strategy->next(this);
}
```

Verwendung der Strategie bleibt
gleich, egal welche konkrete
Strategie verwendet wird

Building



Building.hpp

```
#include <vector>

#include "Floor.hpp"
#include "Elevator.hpp"

class Building {
public:
    Building(int numberOfFloors,
             ConstElevatorStrategyPtr strategy);
    ~Building();

    inline int numberOfFloors() const {
        return floors.size();
    }

    inline Elevator& getElevator() {
        return elevator;
    }

private:
    std::vector<Floor> floors;
    Elevator elevator;
};
```

Strategie wird
an Elevator
weitergereicht

Building.cpp

```
#include <iostream>
using std::cout;
using std::endl;
#include <algorithm>
#include "Building.hpp"

Building::Building(int numberOfFloors,
                   ConstElevatorStrategyPtr strategy):
    floors(numberOfFloors, Floor(0)),
    elevator(&floors[0], strategy)
{
    for (int i = 0; i < numberOfFloors; i++)
        floors[i].setNumber(i);

    cout << "Building(...): "
         << "Creating building with "
         << numberOfFloors
         << " floors." << endl;

    cout << "Building(...): "
         << "Elevator is on Floor: "
         << elevator.getCurrentFloor()->getNumber()
         << endl;
}

Building::~~Building() {
    cout << "~Building(): "
         << "Destroying building." << endl;
}
```



EnergyMinimizingStrategy



TECHNISCHE
UNIVERSITÄT
DARMSTADT

EnergyMinimizingStrategy.h

```
#include "ElevatorStrategy.hpp"

class EnergyMinimizingStrategy
: public ElevatorStrategy {
public:
    EnergyMinimizingStrategy();
    ~EnergyMinimizingStrategy();

    const Floor* next(const Elevator *elevator) const;
};
```

Vererbung in C++
wird so angegeben

public-Vererbung entspricht dem
Vererbungskonzept in Java.
protected- und **private-Vererbung**
schränken die Sichtbarkeit weiter ein

EnergyMinimizingStrategy.cpp

```
#include "EnergyMinimizingStrategy.h"
#include "Elevator.hpp"
using namespace std;

EnergyMinimizingStrategy::EnergyMinimizingStrategy()
: ElevatorStrategy() {
    cout << "EnergyMinimizingStrategy(): "
        << "Creating energy minimizing strategy"
        << endl;
}

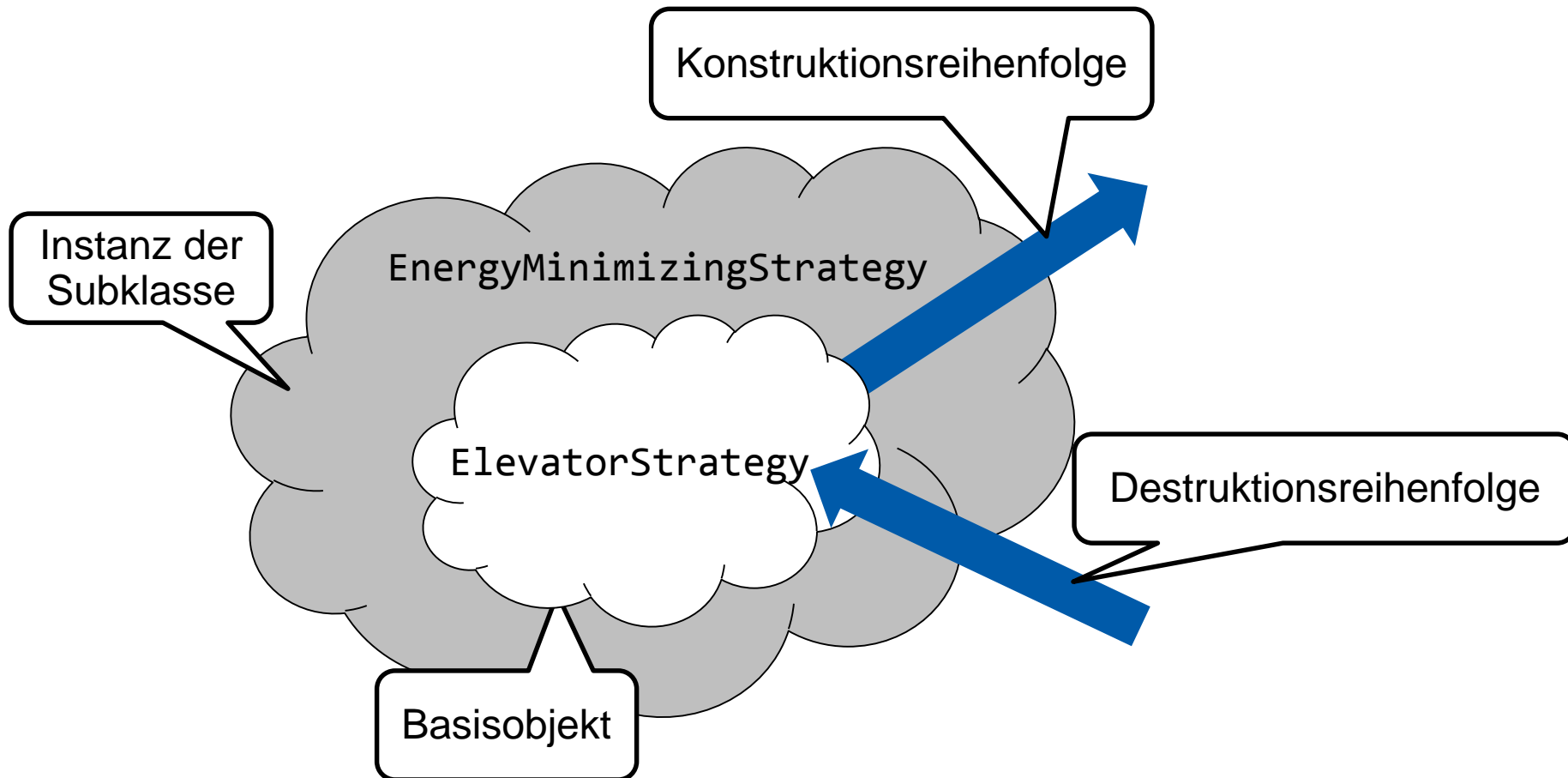
EnergyMinimizingStrategy::~EnergyMinimizingStrategy() {
    cout << "~EnergyMinimizingStrategy(): "
        << "Destroying energy minimizing strategy"
        << endl;
}

const Floor* EnergyMinimizingStrategy::
next(const Elevator *elevator) const{
    cout << "EnergyMinimizingStrategy::next(...): "
        << "Perform some complex calculation ..."
        << endl;

    return elevator->getCurrentFloor();
}
```

Entspricht
super()-Aufruf
in Java





Intermezzo

Wieso ist diese Reihenfolge (Konstruktoren innen nach außen, Destruktoren außen nach innen) sinnvoll?



Probelauf unserer Simulation



TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

#include "Building.hpp"
#include "ElevatorStrategy.hpp"
#include "EnergyMinimizingStrategy.hpp"

int main() {
    ElevatorStrategy *strg = new EnergyMinimizingStrategy();

    // Do something...

    ConstElevatorStrategyPtr strategy(strg);
    Building hbi(6, strategy);

    hbi.getElevator().moveToNextFloor();
}
```



Probelauf unserer Simulation



TECHNISCHE
UNIVERSITÄT
BRAUNSCHWEIG

Konstruktoren werden
richtig aufgerufen

```
ElevatorStrategy(): Creating basic strategy  
EnergyMinimizingStrategy(): Creating energy minimizing strategy
```

```
Floor(): Creating floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
~Floor(): Destroying floor [0]
```

```
Elevator(): Creating elevator.  
Building(...): Creating building with 6 floors.  
Building(...): Elevator is on Floor: 0
```



Polymorpher Aufruf hat
aber **nicht funktioniert!**

```
Elevator::moveToNextFloor(): Polymorphic call to strategy.  
ElevatorStrategy::next(...): Using basic strategy ...
```

```
~Building(): Destroying building.  
~Elevator(): Destroying elevator.
```

```
~Floor(): Destroying floor [0]  
~Floor(): Destroying floor [1]  
~Floor(): Destroying floor [2]  
~Floor(): Destroying floor [3]  
~Floor(): Destroying floor [4]  
~Floor(): Destroying floor [5]
```



Destruktor der Subklasse
wurde nicht aufgerufen!

```
~ElevatorStrategy(): Destroying basic strategy
```



Im Gegensatz zu Java ist bei C++ aus Effizienzgründen die polymorphe Behandlung von Methoden **per Default ausgeschaltet**

Es muss explizit mit dem Schlüsselwort **virtual** angegeben werden, welche Methoden polymorph zu behandeln sind

```
class ElevatorStrategy {  
public:  
    ElevatorStrategy();  
    virtual ~ElevatorStrategy();  
    virtual const Floor* next(const Elevator *elevator) const;  
};
```

Regel: Klassen mit virtuellen Methoden sollten einen **virtuellen Destruktor** besitzen!

Methoden werden als virtuell gekennzeichnet (nur im Header)

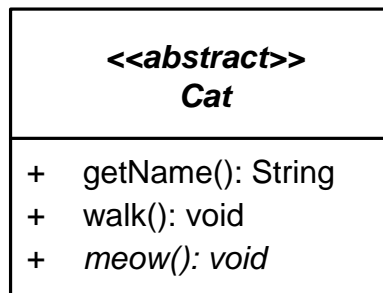
```
class EnergyMinimizingStrategy : public ElevatorStrategy {  
public:  
    EnergyMinimizingStrategy();  
    virtual ~EnergyMinimizingStrategy();  
    virtual const Floor* next(const Elevator *elevator) const;  
};
```

Dies muss nicht in Subklassen wiederholt werden, wird aber häufig der Übersicht halber gemacht

Virtual Method Table

Der Mechanismus der dynamischen Bindung

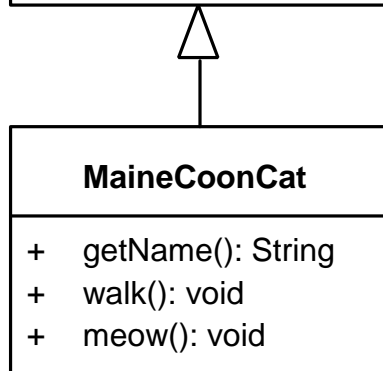
- Egal wie der Pointer auf ein Objekt deklariert ist (z.B. *ElevatorStrategy**), das Objekt behält seinen Typ (z.B. *EnergyMinimizingStrategy*).
- Jede Klasse besitzt eine „Lookup“-Tabelle (**vtable**), die jeder **virtuellen** Methode ihre Implementierung zuweist.



Methode	Implementierung
getName	Cat::getName
walk	Cat::walk
meow	NULL

Enthält standardmäßig:
Java,... : alle Methoden
C++,... : keine Methode

Falls kein Eintrag:
Verwende Methode des
Typs des Pointers.



Methode	Implementierung
getName	Cat::getName
walk	MainCoonCat::walk
meow	MainCoonCat::meow

Probelauf mit virtuellen Methoden



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
ElevatorStrategy(): Creating basic strategy  
EnergyMinimizingStrategy(): Creating energy minimizing strategy
```

```
Floor(): Creating floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
Floor(const Floor&): Copying floor [0]  
~Floor(): Destroying floor [0]
```

```
Elevator(): Creating elevator.  
Building(...): Creating building with 6 floors.  
Building(...): Elevator is on Floor: 0
```

Polymorpher Aufruf
funktioniert jetzt

```
Elevator::moveToNextFloor(): Polymorphic call to strategy.  
EnergyMinimizingStrategy::next(...): Perform some complex calculation ...
```

```
~Building(): Destroying building.  
~Elevator(): Destroying elevator.  
~Floor(): Destroying floor [0]  
~Floor(): Destroying floor [1]  
~Floor(): Destroying floor [2]  
~Floor(): Destroying floor [3]  
~Floor(): Destroying floor [4]  
~Floor(): Destroying floor [5]
```

Und alle Destruktoren werden in der
richtigen Reihenfolge aufgerufen

```
~EnergyMinimizingStrategy(): Destroying energy minimizing strategy  
~ElevatorStrategy(): Destroying basic strategy
```



Pure Virtual

ElevatorStrategy kann nicht mehr instantiiert werden.



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
class ElevatorStrategy {  
public:  
    ElevatorStrategy();  
    virtual ~ElevatorStrategy();
```

```
    virtual const Floor* next(const Elevator *elevator) const = 0;  
};
```

Methode ist hiermit **rein virtuell** – keine Default-Implementierung.

Entspricht einer **abstrakten Methode** in Java.

Klasse mit rein virtuellen Methode entspricht **abstrakter Klasse** oder **Interface** in Java.

Methode kann implementiert werden, muss aber nicht.
Klasse kann dann nicht mehr instanziiert werden.



Intermezzo

Wieso sind virtuelle Methoden „teuer“?

Was bedeutet jede const-Verwendung im folgenden Ausdruck:

```
virtual const Floor* next(const Elevator *elevator) const = 0;
```

Was ist der Unterschied zwischen Zeilen (2) und (3):

1. `EnergyMinimizingStrategy strg0;`
2. `EnergyMinimizingStrategy strg1 = strg0;`
3. `EnergyMinimizingStrategy strg2(strg0);`



Copy Elision



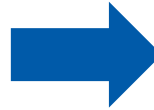
TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
class EnergyMinimizingStrategy {
public:
    inline EnergyMinimizingStrategy() {
        cout << "Constructor called" << endl;
    }

    inline EnergyMinimizingStrategy(const
    EnergyMinimizingStrategy &a) {
        cout << "Copy constructor called" << endl;
    }

    inline void operator=(
        const EnergyMinimizingStrategy &a) {
        cout << "operator= called" << endl;
    }
};

int main() {
    /*1.*/ EnergyMinimizingStrategy a;
    /*2.*/ EnergyMinimizingStrategy c = a;
    /*3.*/ EnergyMinimizingStrategy b(a);
    /*4.*/ b = a;
    /*5.*/ EnergyMinimizingStrategy d =
        EnergyMinimizingStrategy();
}
```



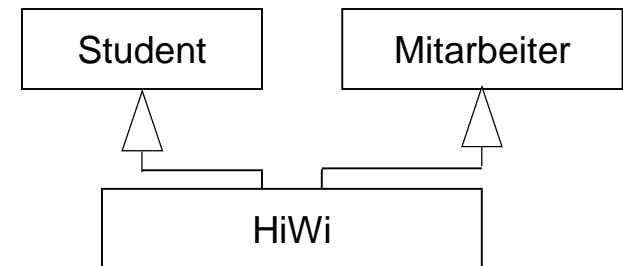
Ausgabe:

- 1 Constructor called
- 2 Copy constructor called
- 3 Copy constructor called
- 4 operator= called
- 5 Constructor called

Mit `-fno-elide-constructors` wird
tatsächlich kopiert.

- Zu erwarten ist, dass bei (5.) zunächst ein Objekt mittels Default-Konstruktor angelegt und dann mittels `operator=` überschrieben wird – C++ ist da schlauer ☺.



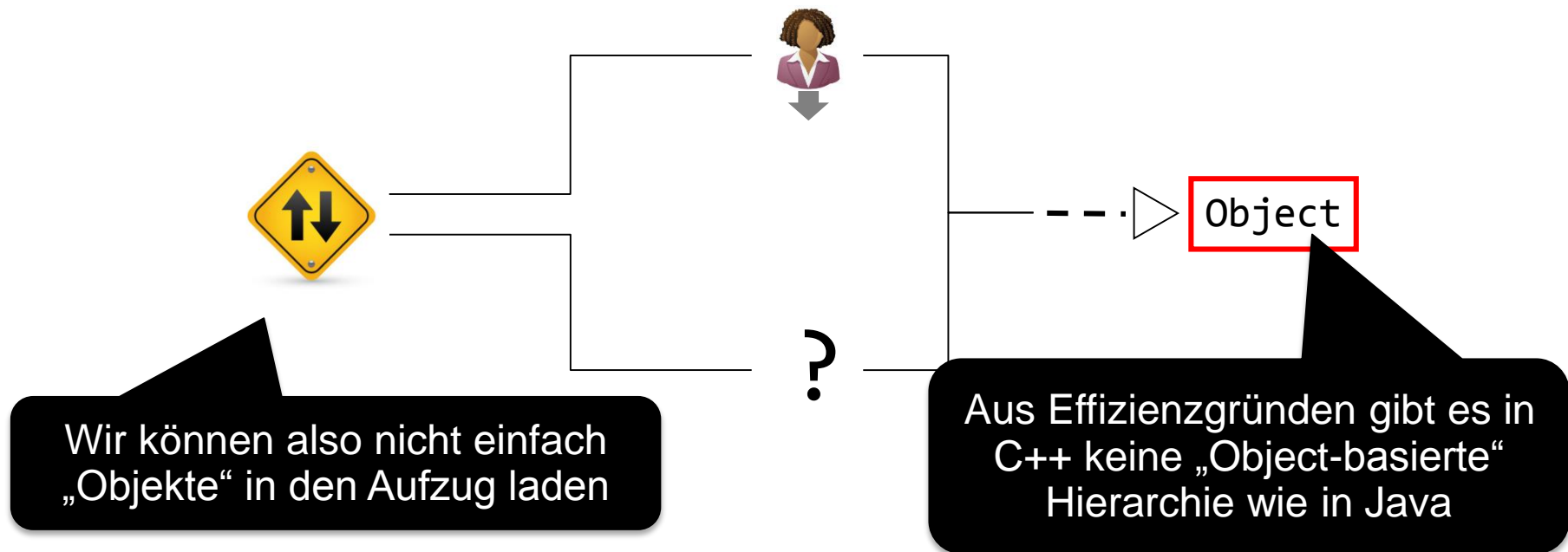


MEHRFACHVERERBUNG



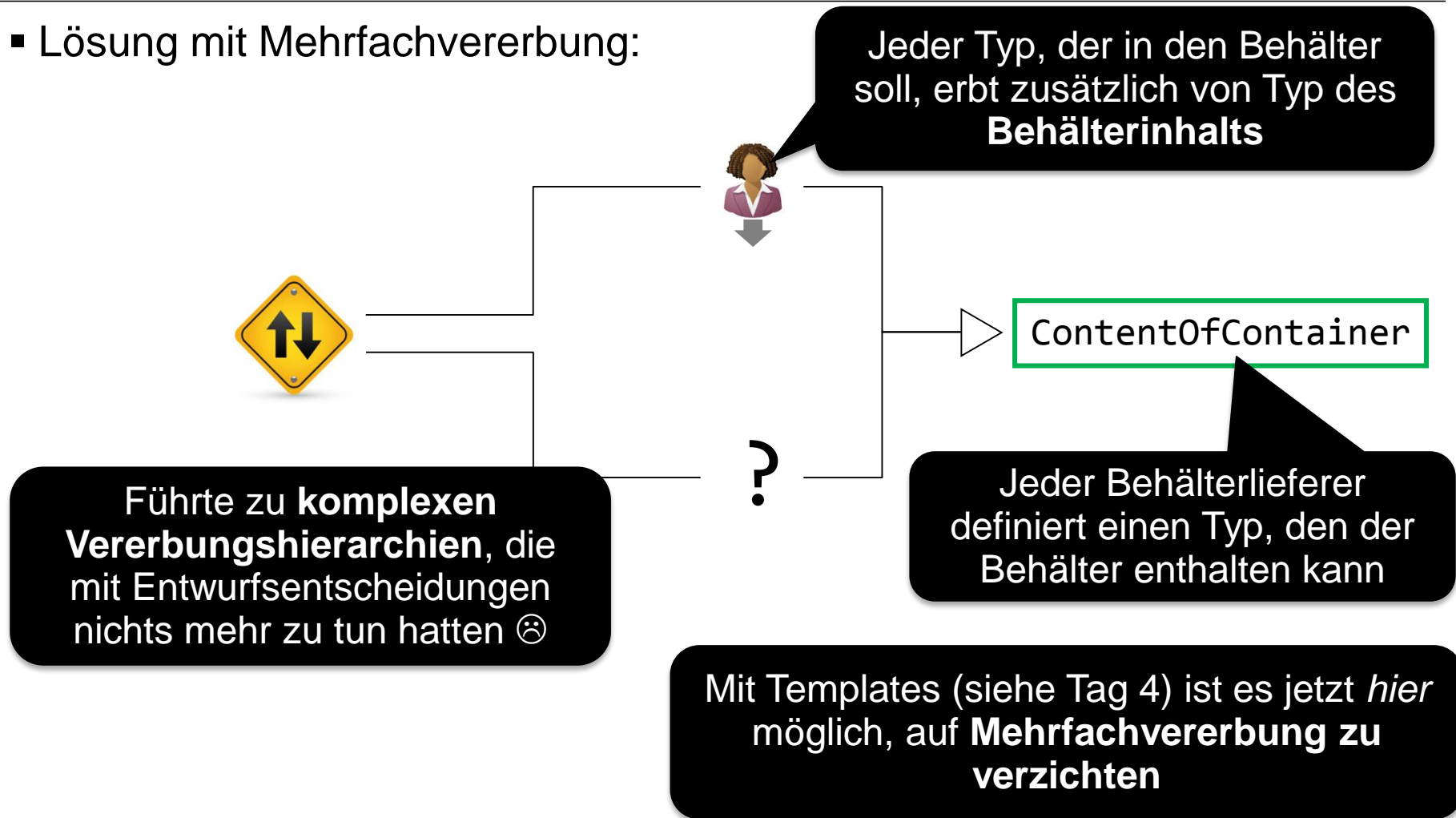
Mehrfachvererbung: Historie

- Ursprünglich als Lösung für **Containerproblem**: Wir wollen Objekte unterschiedlicher Art in den Aufzug (Container) laden.



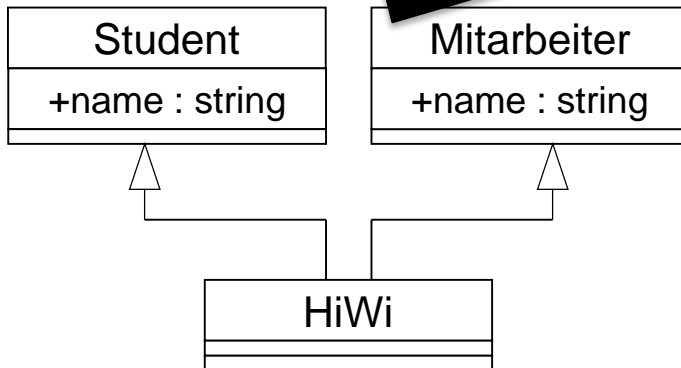
Mehrfachvererbung: Nicht mehr so relevant!

- Lösung mit Mehrfachvererbung:



Schnittstellen- und Implementierungsvererbung

Wenn weitere Oberklassen *pure virtual* sind (enthalten nur ***pure virtual* Methoden**), dann ist Mehrfachvererbung überhaupt kein Problem



Dies entspricht der Verwendung von **Interfaces** in Java!

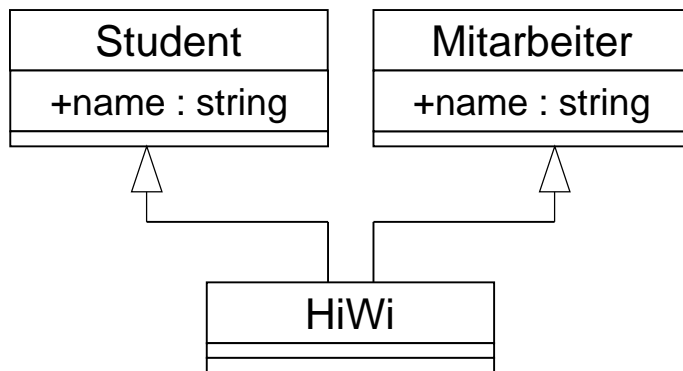
Wird aber von mehreren Oberklassen wirklich **Implementierung** geerbt, so kann das zu Problemen führen...



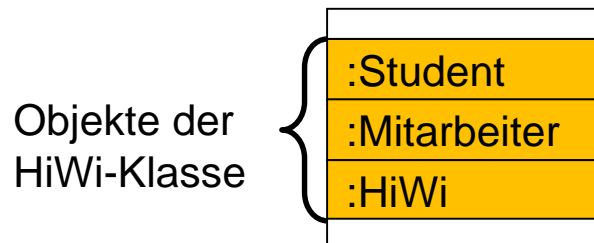
Implementierungsvererbung: Konflikte

▪ Mehrfachvererbung kann zu Mehrdeutigkeit führen

Attribute und Methoden einer Oberklasse sind Bestandteil der Unterklasse (außer private-Elemente)



Speicher



```
class Student { public: string name; };
class Mitarbeiter { public: string name };
```

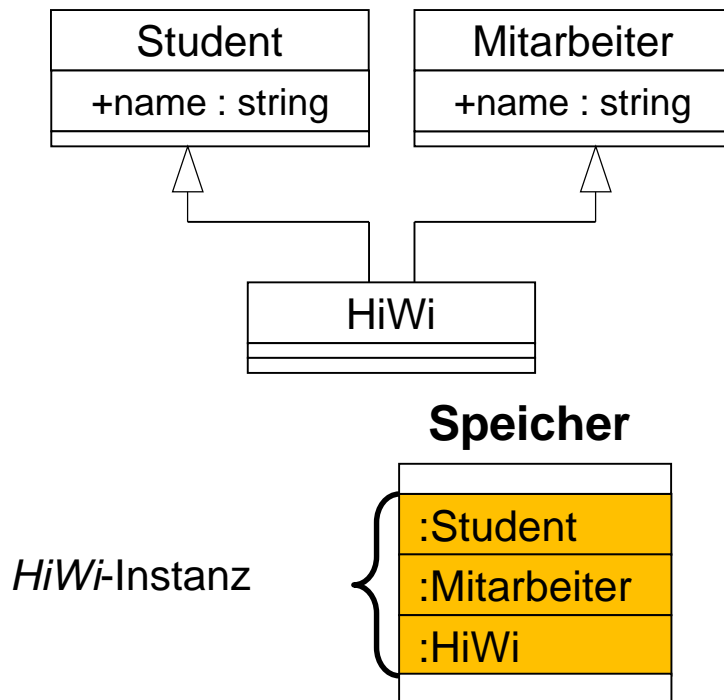
```
class HiWi : public Student, public Mitarbeiter
{ ... }
```

```
HiWi* h = new HiWi();
h->name = "Christian";
```

Namenskonflikt!
Keine eindeutige
Zuweisung ...

Implementierungsvererbung: Konflikte

- Auflösung der Mehrdeutigkeit durch Verwendung des vollständigen Namens (Scope-Operator)



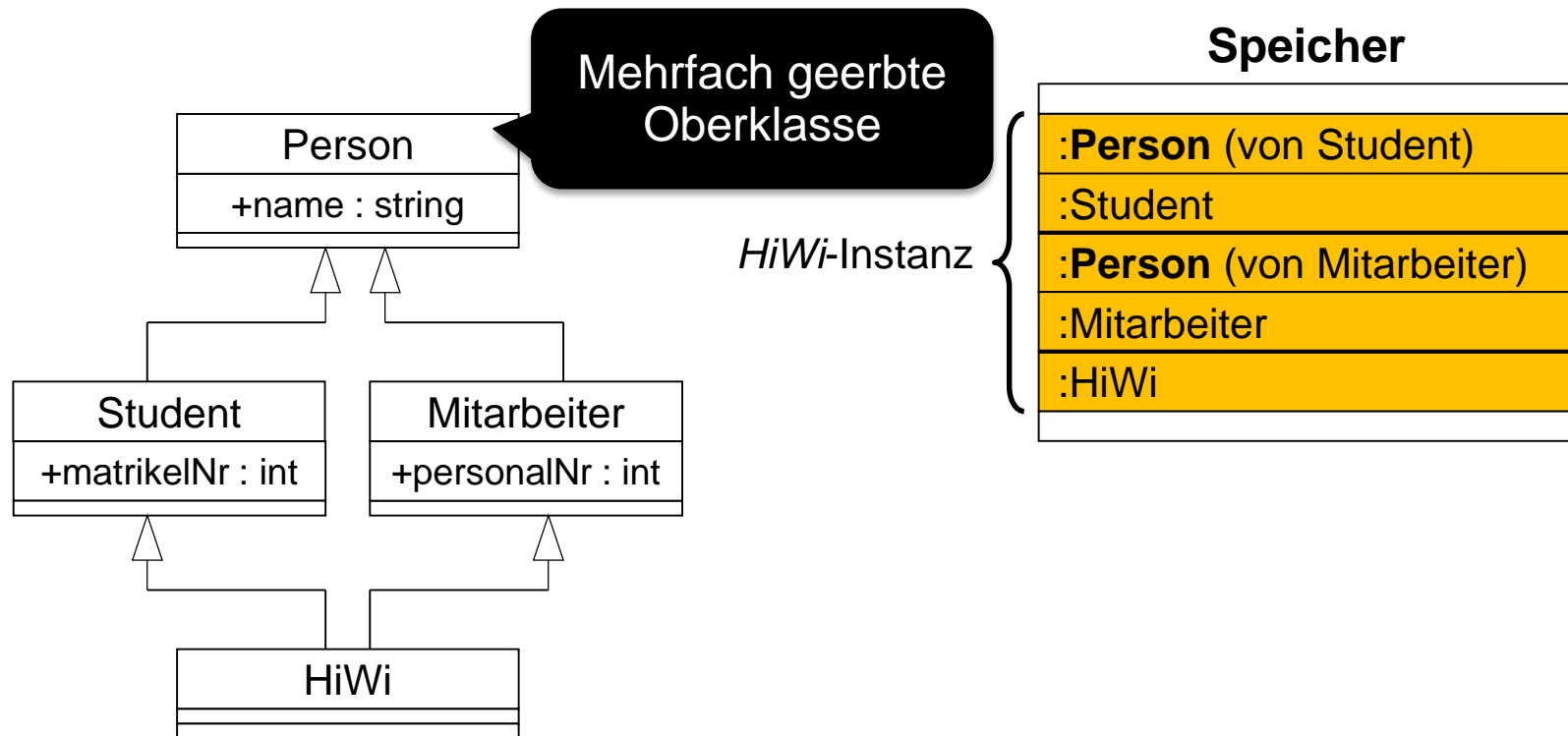
```
class Student { public: string name; };  
class Mitarbeiter { public: string name };
```

```
class HiWi : public Student, public Mitarbeiter  
{ ... }
```

```
HiWi* h = new HiWi();  
h->Student::name = "Christian";  
h->Mitarbeiter::name = "Mark";
```

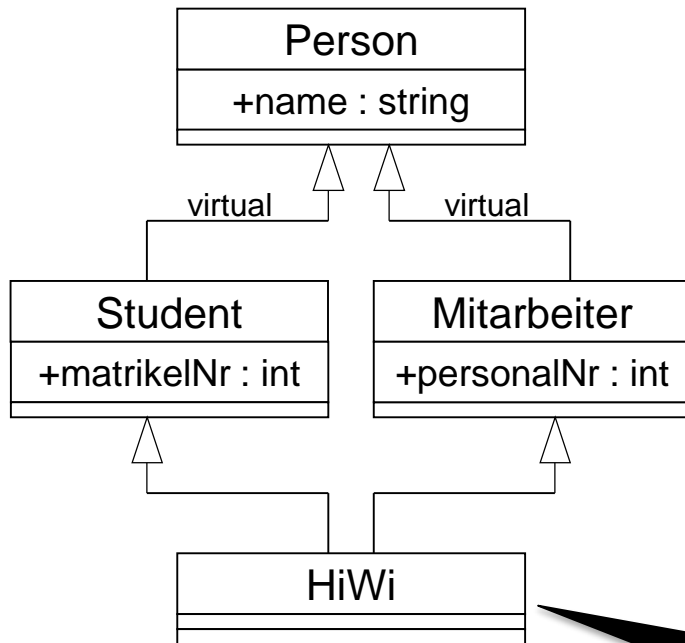
Scope-Operator

- Mehrfach geerbte Oberklassen führen auch zur unnötigen Bindung von Speicher



▪ Lösung: Mehrfach geerbte Oberklassen nur einmal einbinden

Schlüsselwort **virtual** ermöglicht virtuelle Oberklassen / Vererbung



```
class Person { public: string name; };
class Student : virtual public Person { ... };
class Mitarbeiter : virtual public Person { ... };

class HiWi : public Student, public Mitarbeiter { ... }

HiWi* h1 = new HiWi();
H1->name = „Max“; // eindeutig (nur 1x vorhanden)
```

Aber: Die *virtual*-Deklaration findet nicht an der Stelle statt, an der sie nötig wird (*HiWi*)!

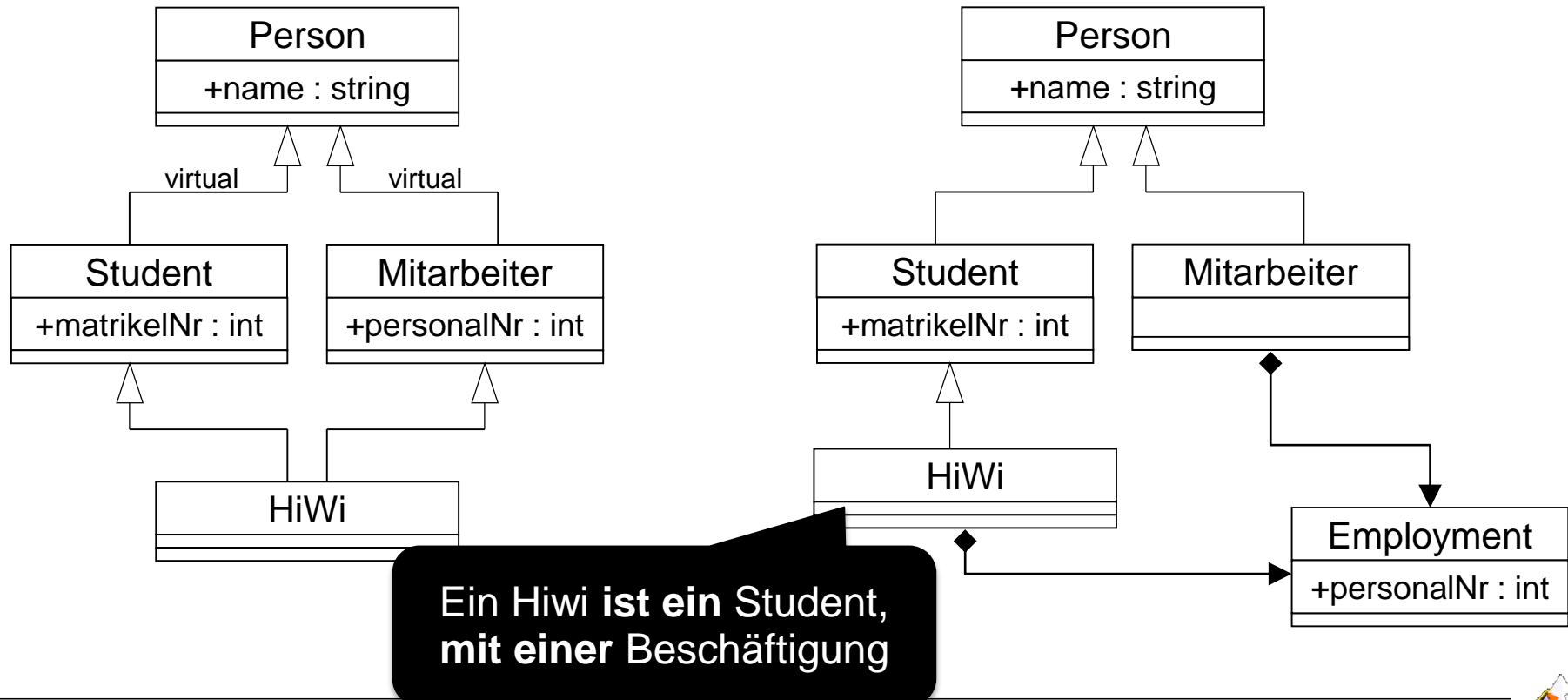
Implementierungsvererb.: Schlechtes Design?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

▪ Mehrfachvererbung kann auf „schlechtes“ Design hindeuten

Gemeinsamkeiten sollen explizit extrahiert bzw. das Design vereinfacht werden



Intermezzo

Also – Mehrfachvererbung: Ja oder nein?



Mehrfachvererbung: Ja oder Nein?

1. **Schnittstellenvererbung** sinnvoll, nützlich (Design!) und zumeist unproblematisch (vgl. Interfaces in Java)

Aber auch Interfaces können zu Problemen führen!

2. **Implementierungsvererbung** problematisch und zu vermeiden (Komposition vorziehen)