

# Programmierpraktikum C und C++

## Vererbung und Polymorphie



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



**Anthony Anjorin**

[anthony.anjorin@es.tu-darmstadt.de](mailto:anthony.anjorin@es.tu-darmstadt.de)

ES Real-Time Systems Lab

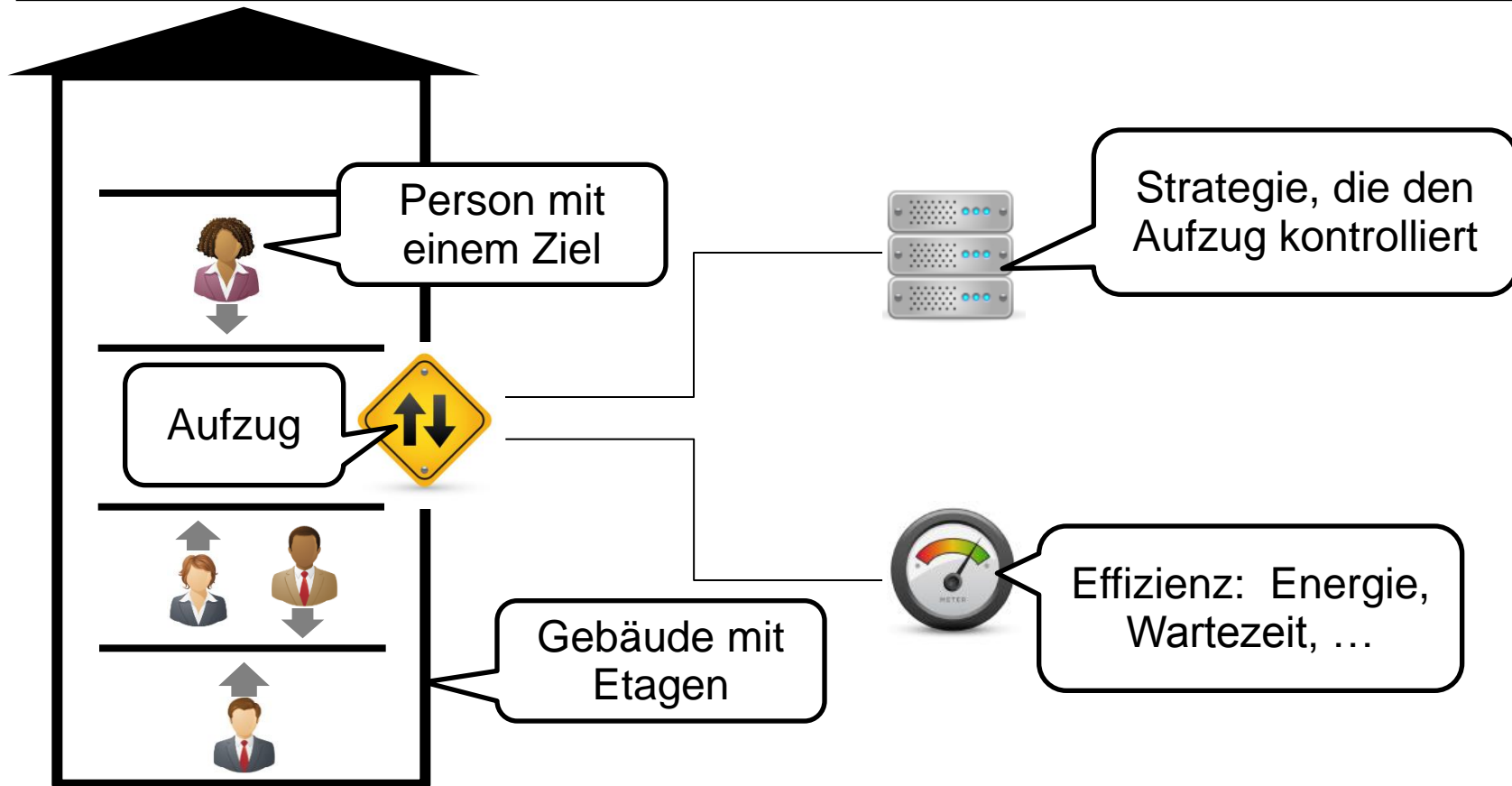
Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

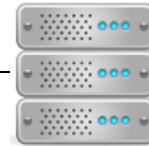
[www.es.tu-darmstadt.de](http://www.es.tu-darmstadt.de)

# Was ist Polymorphie?



# Was ist Polymorphie?

Der Code im Aufzug, der die Strategie verwendet, soll sich nicht ständig ändern, nur weil es neue Strategien gibt, bzw. die Strategie ausgetauscht wird



Konkrete, sehr unterschiedlichen 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 erweitert werden!



# Lösung mit Polymorphie



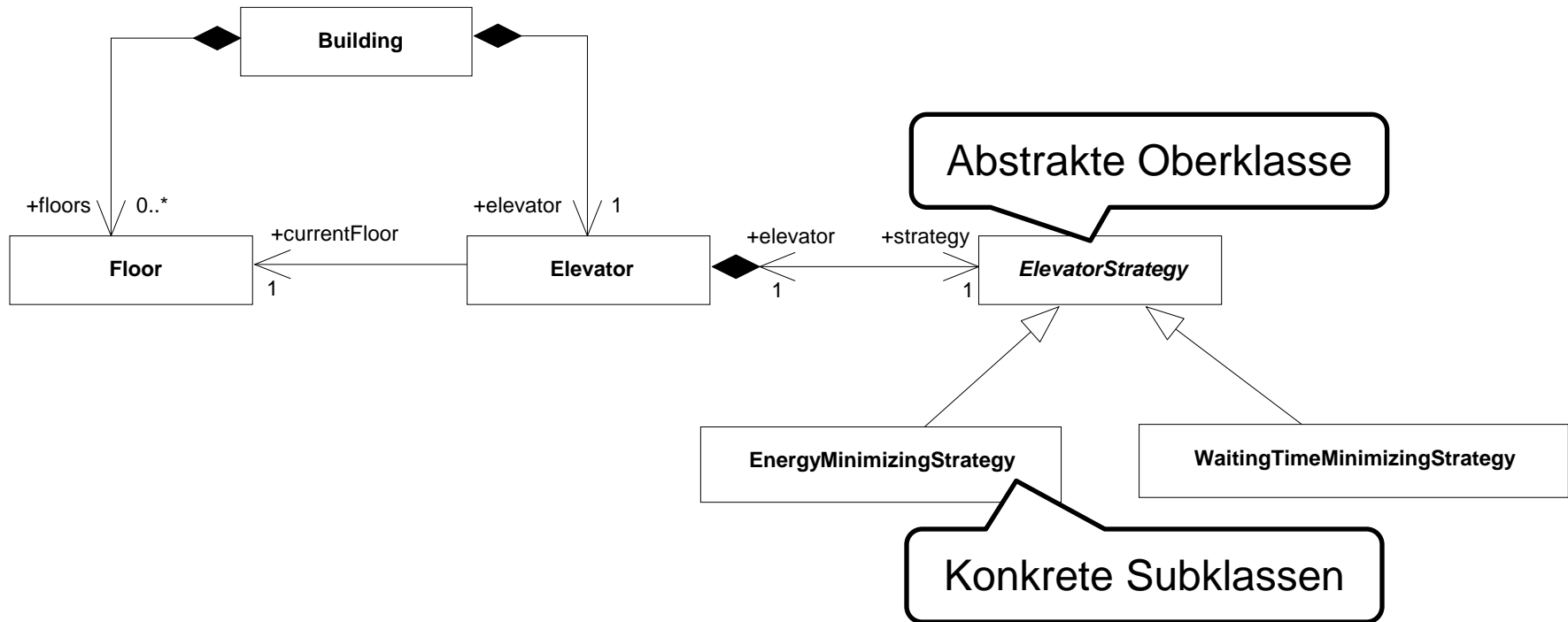
```
void Elevator::moveToNextFloor(){  
    currentFloor = strategy->next(this);  
}
```

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)



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?



```
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!

```
#include "Floor.h"  
  
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



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Vorwärtsreferenz für Elevator um  
zyklische Abhängigkeit aufzulösen

```
#include <boost/shared_ptr.hpp>
#include "Elevator.h"

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!

```
#include "ElevatorStrategy.h"
#include "Elevator.h"

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

```
#include "ElevatorStrate
#include "Floor.h"
```

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

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

```
void moveToNextFloor();
```

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

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

Typen ohne Namen  
auch möglich

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

```
#include "Elevator.h"
```

```
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



Strategie wird  
an Elevator  
weitergereicht

```
#include <vector>

#include "Floor.h"
#include "Elevator.h"
```

```
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;
};
```

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

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

Vererbung in C++  
wird so angegeben

```
#include "ElevatorStrategy.h"

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

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

Public-Vererbung entspricht dem  
Vererbungskonzept in Java.  
Protected- und Private-Vererbung  
schränken die Sichtbarkeit weiter ein

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

Entspricht  
**super**-Aufruf  
in Java

```
EnergyMinimizingStrategy::EnergyMinimizingStrategy()
: ElevatorStrategy() {
```

```
    cout << "EnergyMinimizingStrategy(): "
          << "Creating energy minimizing strategy"
          << endl;
```

```
}
```

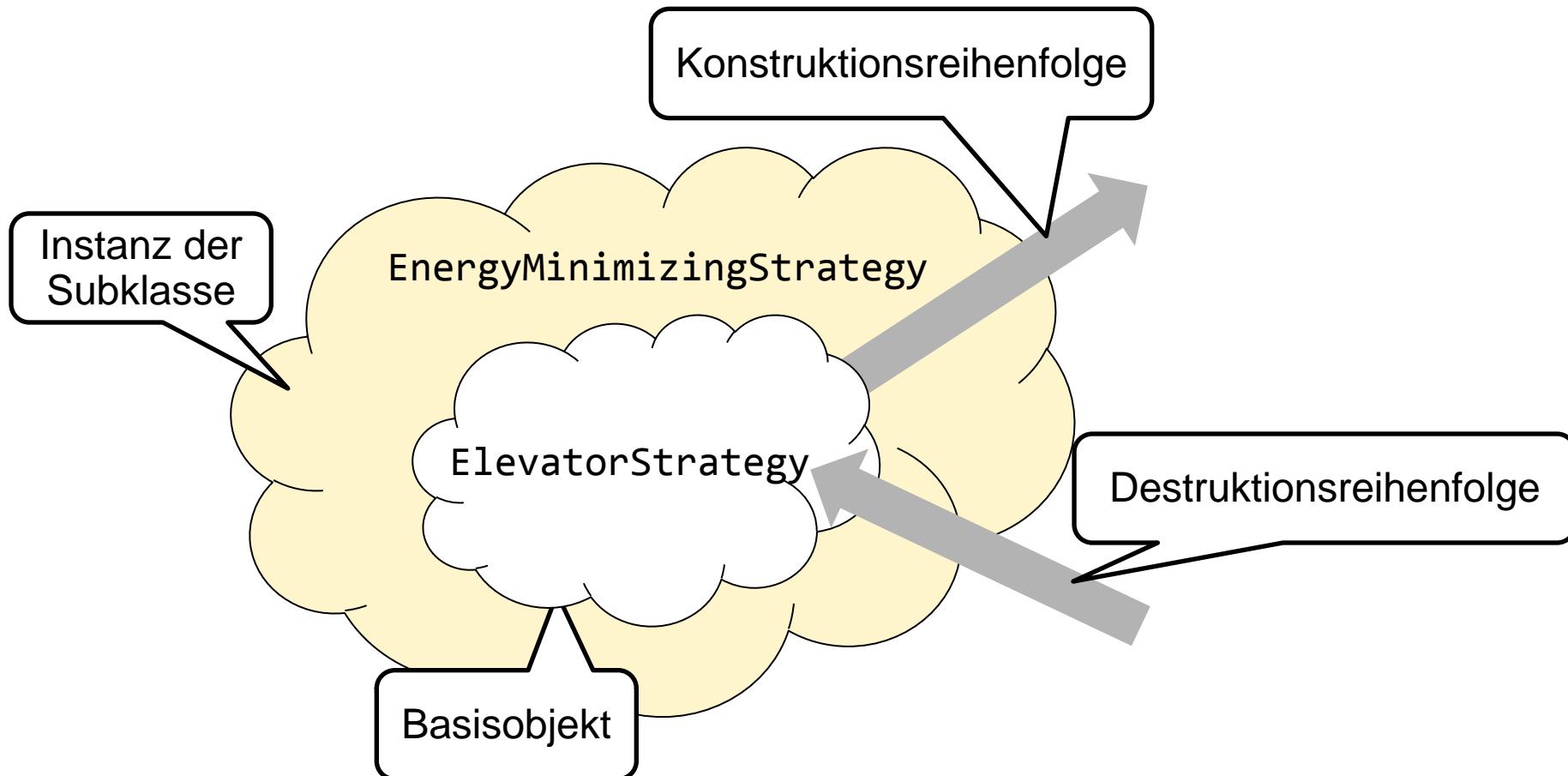
```
EnergyMinimizingStrategy::~~EnergyMinimizingStrategy() {
    cout << "~EnergyMinimizingStrategy(): "
          << "Destroying energy minimizing strategy"
          << endl;
```

```
}
```

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

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





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



# Probelauf unserer Simulation



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

#include "Building.h"
#include "ElevatorStrategy.h"
#include "EnergyMinimizingStrategy.h"

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: Eine Klasse mit einer virtuellen Methode sollte 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

# Probelauf mit virtuellen Methoden

```
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

```
class ElevatorStrategy {  
public:  
    ElevatorStrategy();  
    virtual ~ElevatorStrategy();  
  
    virtual const Floor* next(const Elevator* elevator) const = 0;  
};
```

Methode ist hiermit  
„pur virtuell“

-----

Entspricht einer abstrakten Methode in Java.

Klasse mit einer puren virtuellen Methode entspricht einer abstrakten Klasse in Java.

Methode kann implementiert werden, muss aber nicht.

Klasse kann aber nicht mehr instanziiert werden.



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);`

