

# Programmierpraktikum C und C++

## Einführung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



**Roland Kluge**

roland.kluge@es.tu-darmstadt.de

ES Real-Time Systems Lab

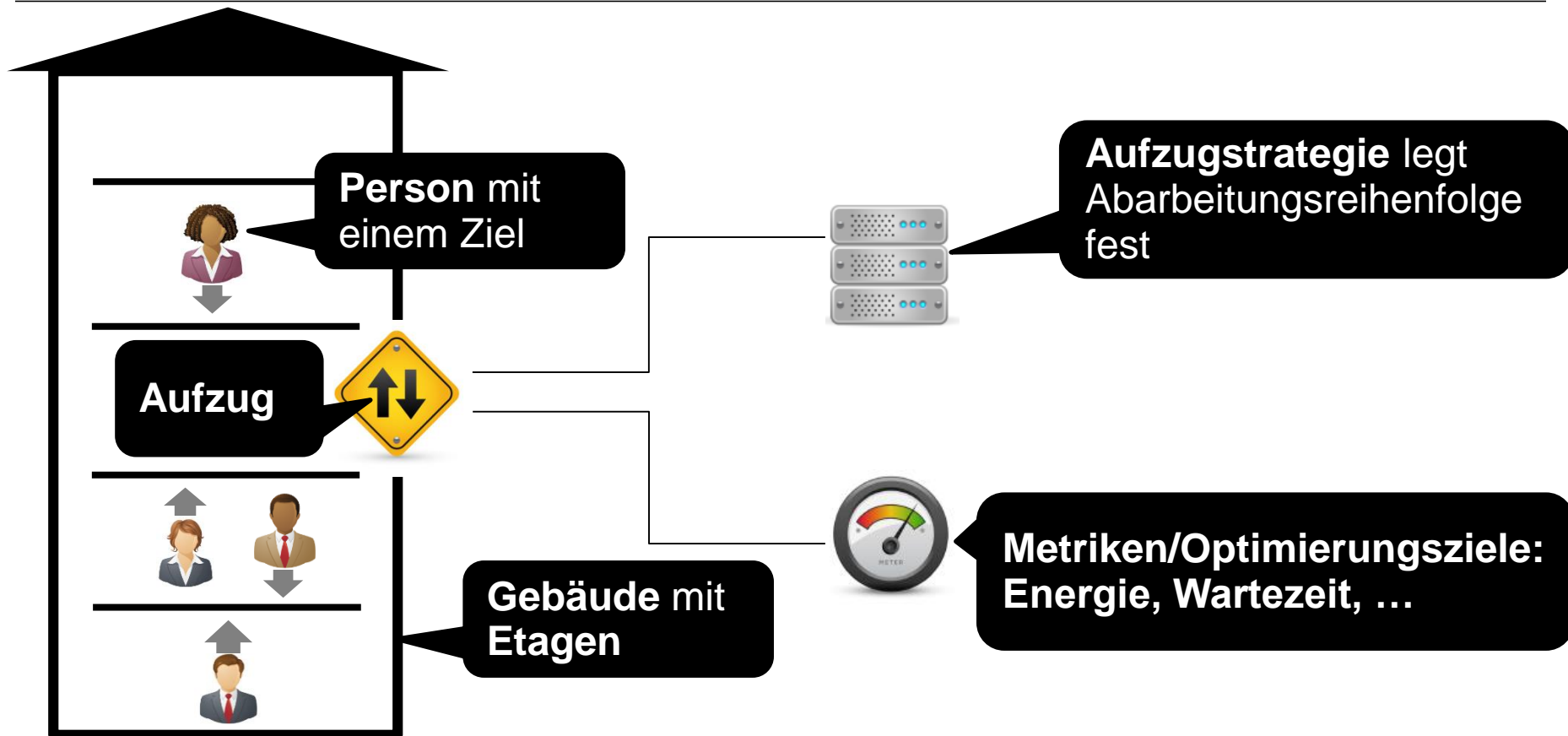
Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

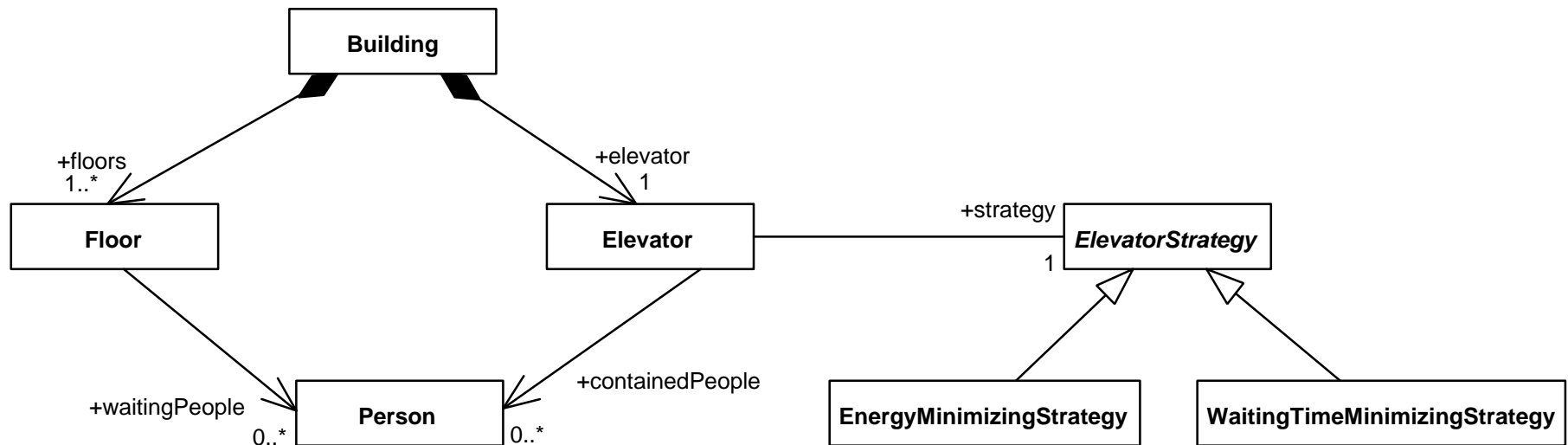
Dept. of Computer Science (adjunct Professor)

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

# Implementierung einer Aufzugsimulation



# Statische Struktur des Systems (Klassendiagramm / Metamodell)



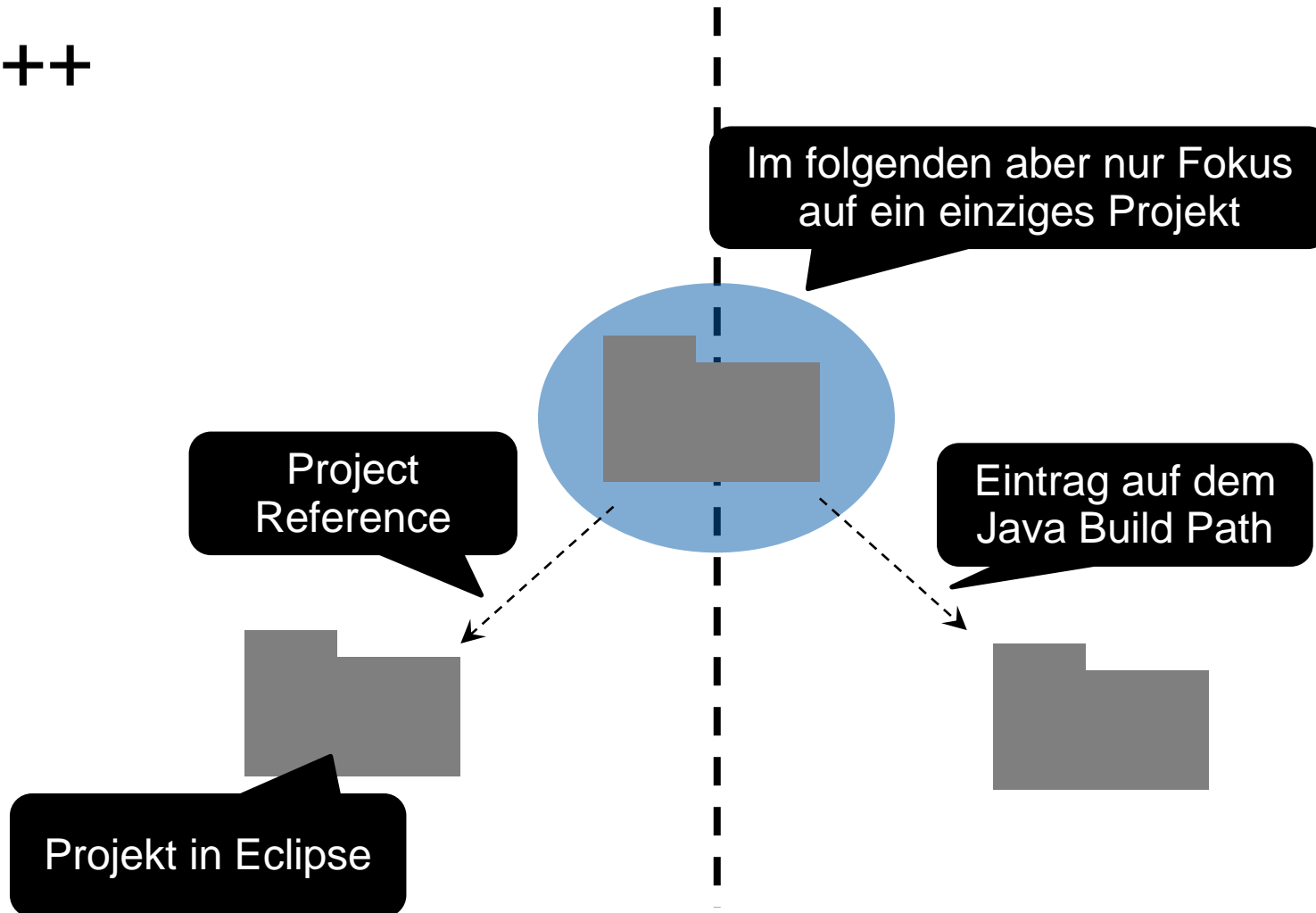


## 1. Projektabhängigkeiten

# Projektabhängigkeiten (mit Eclipse CDT)

C++

Java

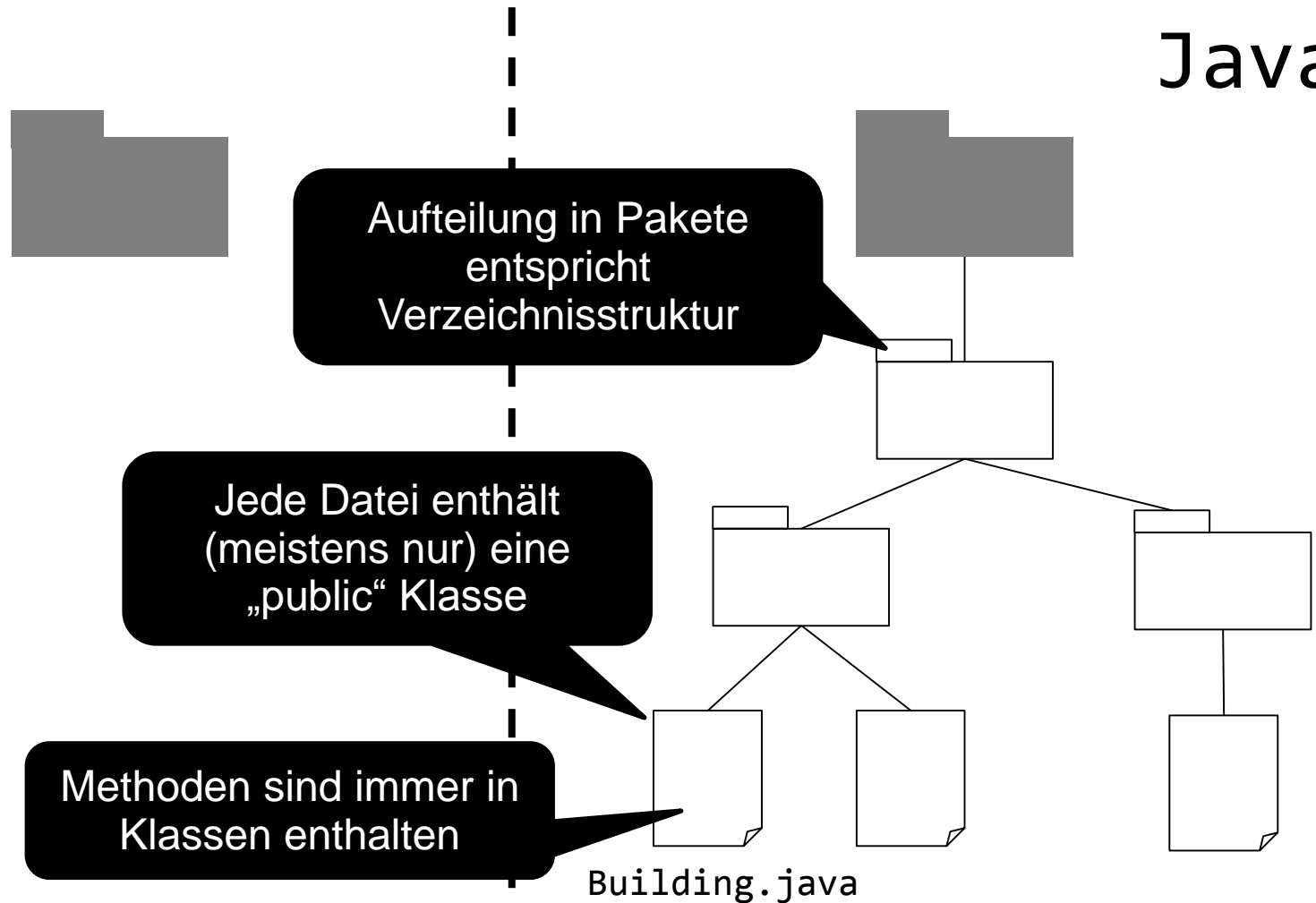




## 2. Projektstruktur

C++

Java



# Intermezzo

Ist es sinnvoll, zu verlangen, dass jede „Funktion“ in einer Klasse sein MUSS?

Ist es sinnvoll, die Paketstruktur an der Verzeichnisstruktur zu binden?

Darf man in Java mehrere Klassen in einer Datei implementieren?





## C++

## Java

Implementierungsdateien mit **Funktionen** (nicht Methoden!) sind möglich und üblich

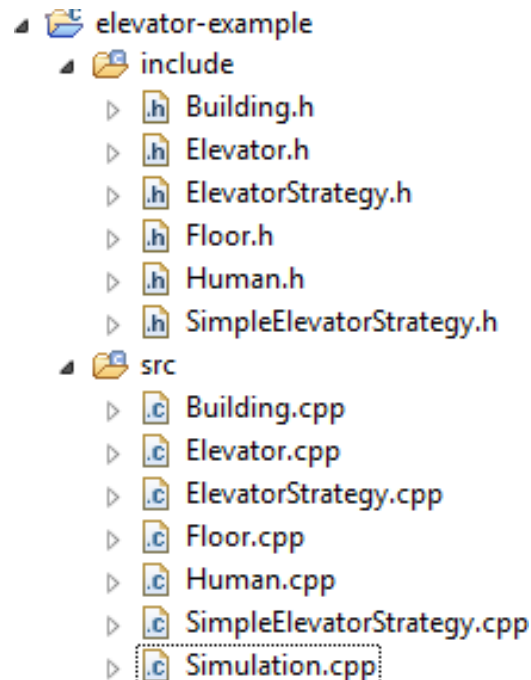
**Beliebige Verzeichnisstruktur** - hat nichts mit Sichtbarkeit zu tun

Klassen werden in **Header-** und **Implementierungsdatei** getrennt

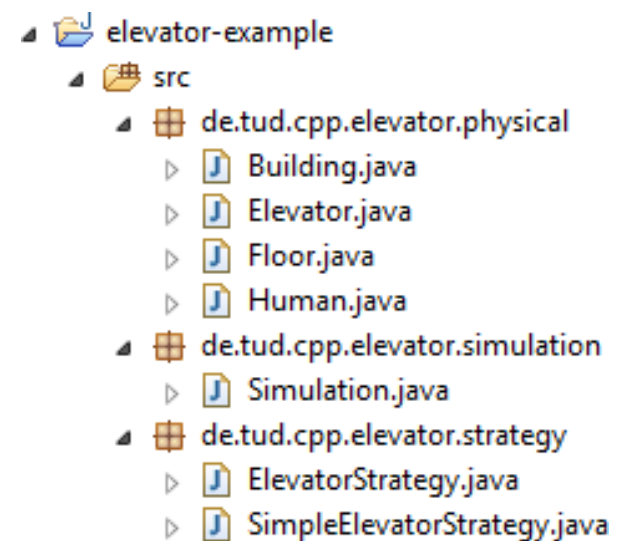
**Mehrere Klassen** können flexibel in Header/Implementierungsdateien kombiniert werden

Building.h Building.cpp

## C++



## Java



# Header und Implementierungs-Dateien



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
/*  
 * Part of the elevator simulation  
 * A Building is a container for  
 * Floors and the Elevator  
 */
```

**Kommentare** wie in Java  
(auch // möglich)

```
#ifndef BUILDING_H_  
#define BUILDING_H_
```

```
#include <vector>
```

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

**Include-Anweisungen** wie Import-  
Befehle in Java.

< ... > für Bibliotheken,  
" ... " für eigenen Code

```
class Building {  
public:  
    Building(int numberOfFloors);  
    ~Building();  
  
    void runSimulation();  
  
private:  
    std::vector<Floor> floors;  
    Elevator elevator;  
};  
  
#endif /* BUILDING_H_ */
```

**Deklaration der Klasse** ist  
wie ein Interface in Java



# Header und Implementierungs-Dateien



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

**Using-Befehle** sind wie  
statische Imports in Java  
(*cout* statt *std::cout*)

```
#include "Building.h"
```

**Header-Datei** wird eingebunden

```
Building::Building(int numberOfFloors) :
    floors(numberOfFloors, Floor()) {
    cout << "Creating building with "
         << numberOfFloors << " floors."
         << endl;
}

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

void Building::runSimulation() {
    cout << "Simulation running ..." << endl;
}
```

**Methoden** werden implementiert  
(Details später)



# Intermezzo

Ist die Trennung in Header- und Impl-  
Dateien wirklich hilfreich? Oder nur nervig...



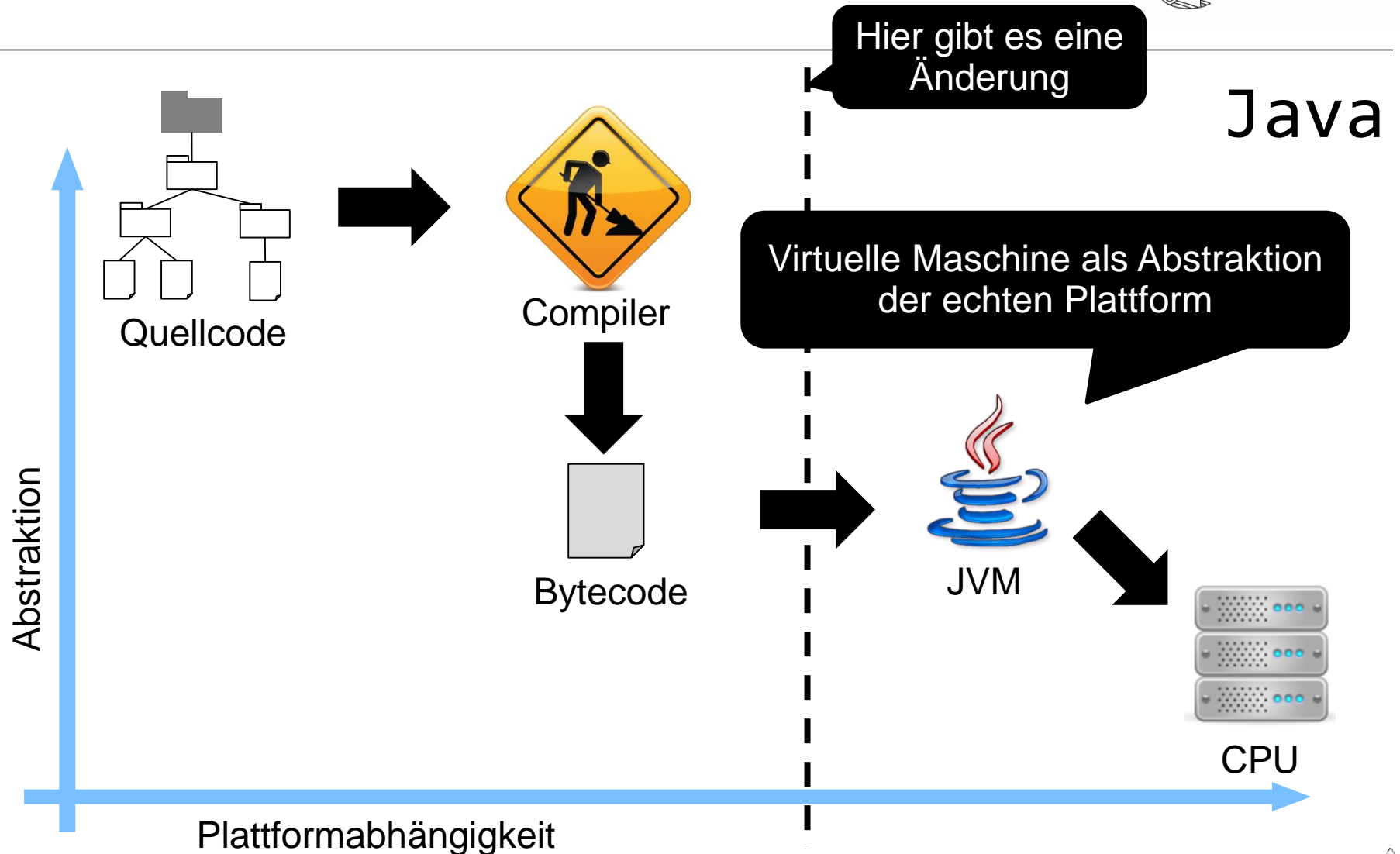


## 3. Kompilierung

# Kompilierung



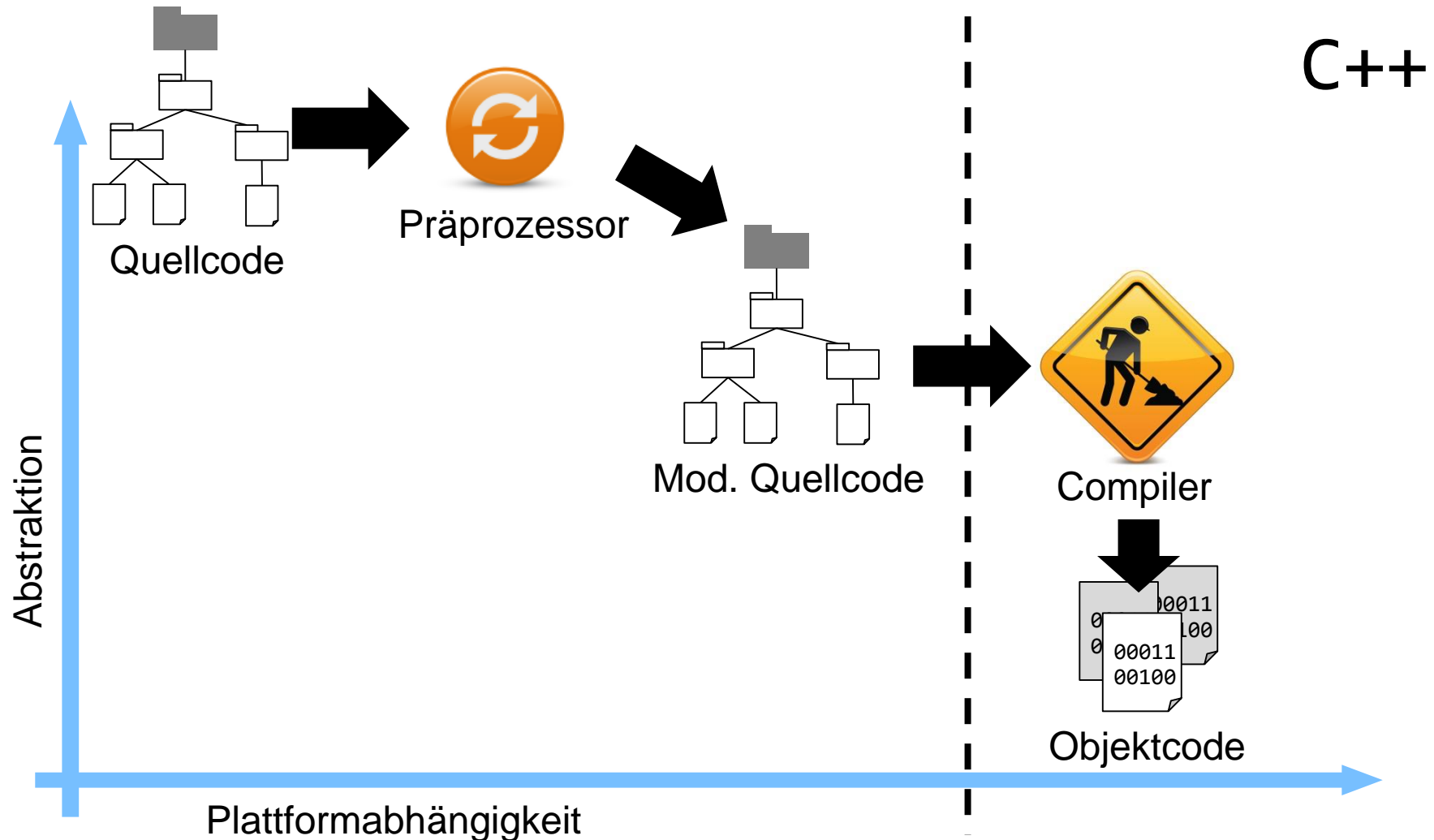
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



Coffee Cup: <http://www.iconarchive.com/show/cristal-intense-icons-by-tatice/java-icon.html>



# Kompilierung für C/C++ I

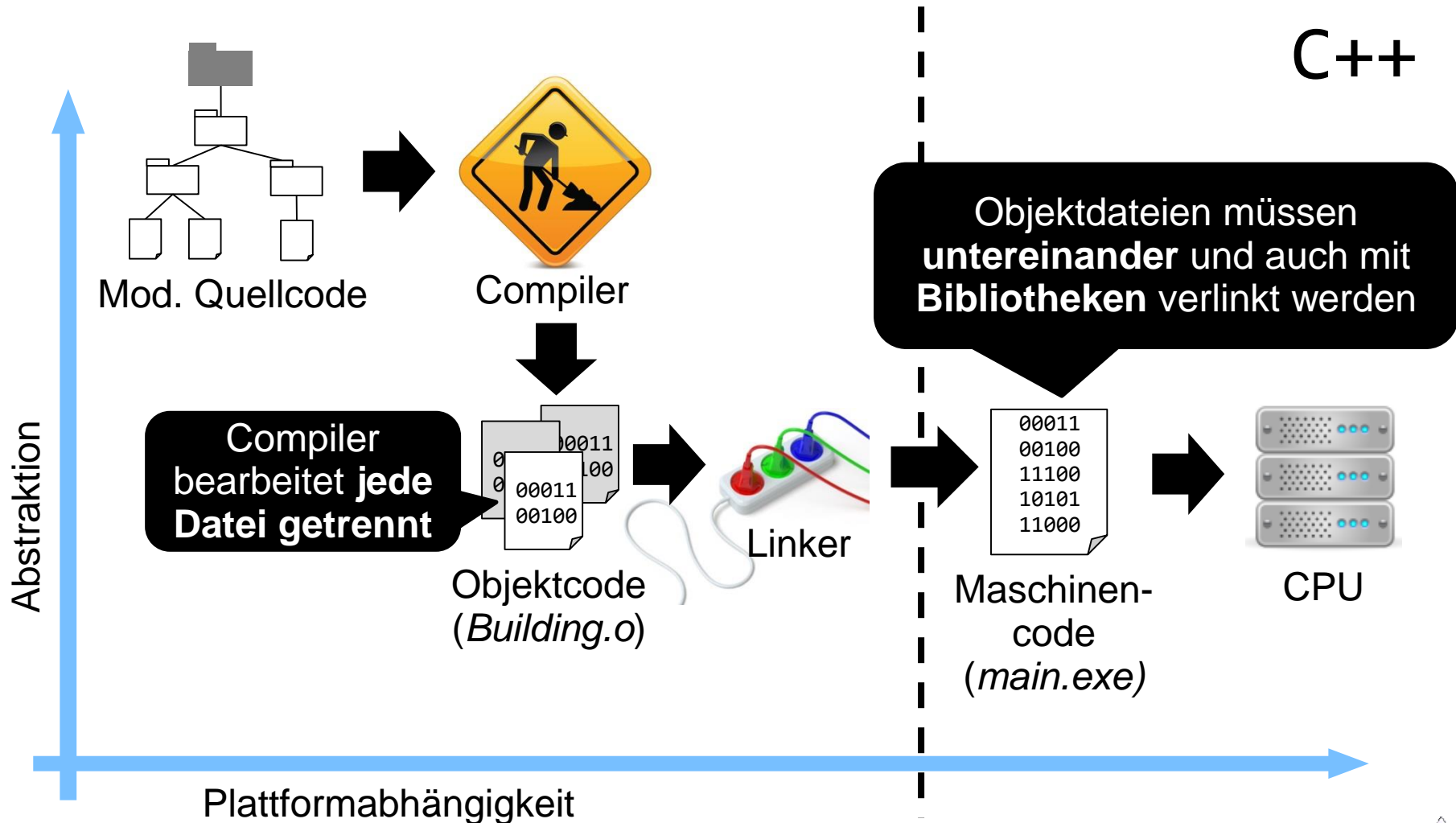




# Kompilierung für C/C++ II



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



# Was genau macht der Präprozessor?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
#ifndef BUILDING_H_
#define BUILDING_H_

#include <vector>

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

**Include Guard:**  
Schützt davor, dass *Building.h*  
mehrmals eingebunden wird

Diese Konvention macht es  
möglich, ohne Bedenken immer  
**alle benötigten Header überall**  
**einbinden** zu können

```
class Building {
public:
    Building(int numberOfFloors);
    ~Building();

    void runSimulation();

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

#endif /* BUILDING_H_ */
```

Der Präprozessor kann viel mehr,  
aber seine Verwendung für C++-  
Programme (über das gezeigte  
hinaus) ist **weder notwendig**  
**noch zu empfehlen**



# Spaß mit dem Präprozessor

Do not try this at home! ☺



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## ▪ Keyword *return* neu definieren:

```
#define return DoSomeStackCheckStuff, return
```

Hoffentlich erinnert sich da später noch jemand dran...

## ▪ Auswertung von Ausdrücken zur Compile-Zeit:

```
/* Force a compilation error if condition is true */  
#define BUILD_BUG_ON(condition) ((void)sizeof(char[1 -  
2*!!(condition)]))
```

Angeblich im Linux-Kernel verwendet, um Asserts zur Compile-Zeit durchzuführen

Quelle: <http://stackoverflow.com/questions/599365/what-is-your-favorite-c-programming-trick>



# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Stimmt es wirklich, dass Java  
„plattformunabhängig“ ist und C++ nicht?

Ist es möglich, dass man erfolgreich  
kompilieren aber nicht linken kann? Wie?

Ist der Präprozessor wirklich „böse“? Wieso?  
Ist dies bei allen Sprachen der Fall?





## 4. Systemstart

Viele Beispiele der Vorlesung sind im SVN-Repository.

```
//=====
// Name: elevator-example-lecture.cpp
//=====
```

```
#include "Building.h"
```

```
// int main(int argc, char** argv)
int main() {
    Building building(3);
    building.runSimulation();
}
```

**Main-Funktion** entspricht  
Main-Methode in Java  
(Argumente auch möglich  
aber nicht nötig)

**Kein Rückgabewert nötig** (implizit 0 für  
„alles ordnungsgemäß durchgelaufen“),  
zumindest bei *gcc*



# Intermezzo



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

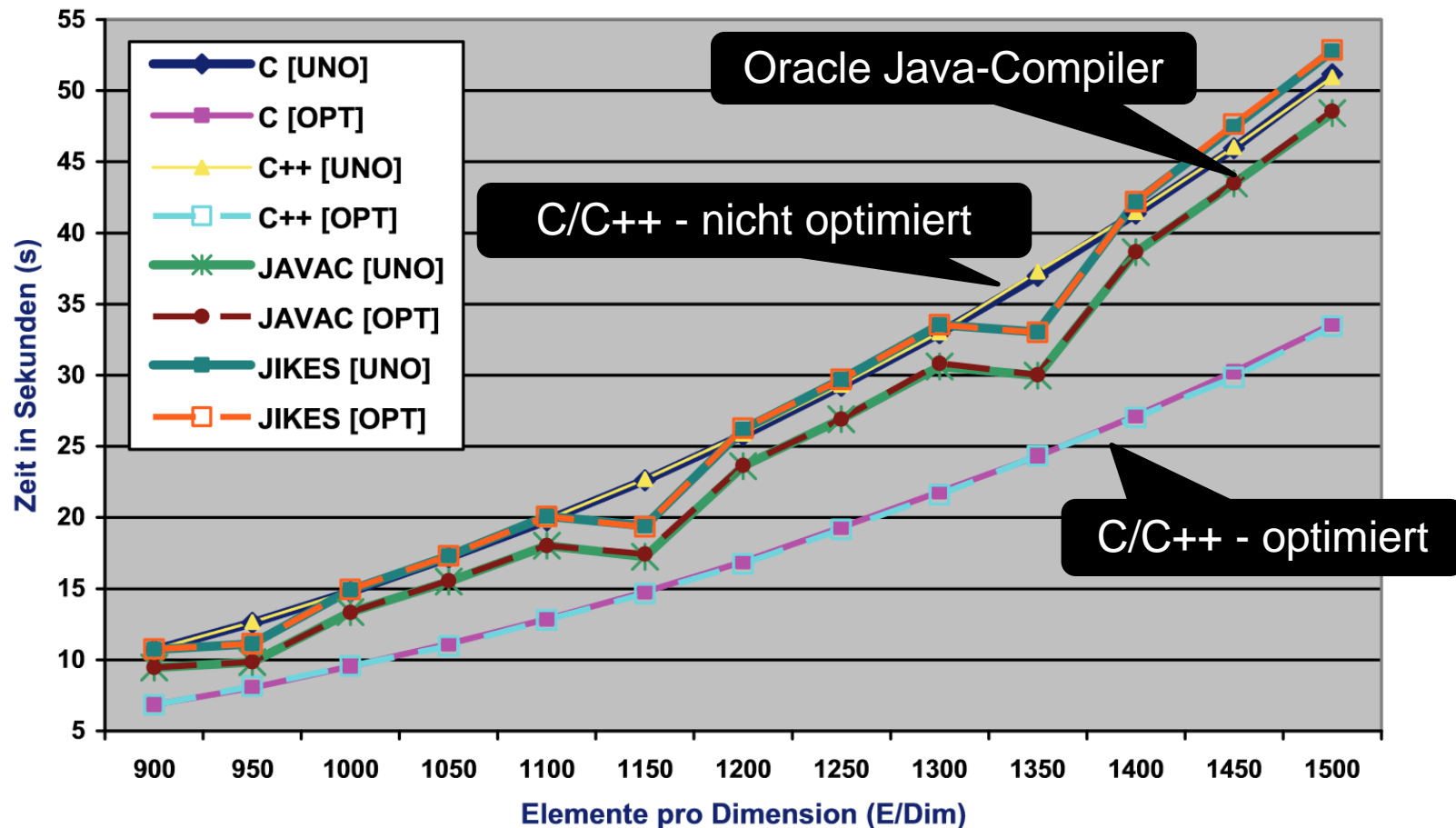


**Java vs. C++: Stärken und Schwächen?**



# Laufzeitunterschied zwischen Java und C++

## Beispiel Matrixmultiplikation



Manuel Prager: Laufzeitvergleiche für die Implementierung von Algorithmen in Java und C/C++  
Hochschule Neubrandenburg, 2010