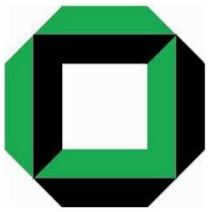
Fortgeschrittene Objektorientierung

SS 2009

Prof. Dr.-Ing. G. Snelting



Universität
Karlsruhe

Fakultät für Informatik

Vorwort

Objektorientierte Programmierung in Java wird schon im Grundstudium intensiv behandelt. Diese Hauptstudiumsvorlesung greift einerseits die programmier- und softwaretechnischen Gesichtspunkte der Objektorientierung auf, indem noch einmal vertieft auf Vererbung, Design Patterns, Event-Programmierung usw. eingegangen wird.

Die Hauptsache in dieser Vorlesung sind aber neue Sprachkonzepte, theoretische Grundlagen und Implementierungstechniken. Deshalb werden Typsysteme ebenso behandelt wie Gesichtspunkte der Compilation objektorientierter Sprachen. Zum Schluss werden Techniken der Programmanalyse vorgestellt, die eine immer größere Bedeutung nicht nur für Optimierungen, sondern insbesondere für Software-Wartung und Sicherheitsprüfungen gewinnen.

Viele der vorgestellten theoretischen Konzepte kann man streng formal einführen. Die Vorlesung ist aber in einem semi-formalen Stil gehalten, der die mangelnde Präzision von „Kästchen und Pfeilen“ vermeidet, aber dennoch nicht dem Stil einer Mathematikveranstaltung entspricht. Die ausgegebenen Folienkopien sind nicht als ausgearbeitetes Skript gedacht, sondern lediglich als Ergänzung zur Vorlesung. Wer ein Lehrbuch zur Vorlesung sucht, dem sei das Buch von Eliens empfohlen.

Prof. Dr. G. Snelting

Inhaltsverzeichnis

Literatur

- A. Eliens: Principles of Object-Oriented Software Development. 2nd edition, Pearson 2000.
- K. Bruce: Foundations of Object-Oriented Languages, MIT Press 2002.
- M. Abadi, L. Cardelli: A Theory of Objects. Springer 1996.
- J. Palsberg, M. Schwartzbach: Object-Oriented Type systems. Wiley & Sons 1994.
- E. Gamma et al.: Design Patterns. Addison-Wesley 1995.
- M. Fowler et al.: Refactoring: Improving the Design of Existing Code. Addison Wesley 1999.
- J. Gosling, B. Joy, G. Steele: The Java Language specification. Second edition, Addison Wesley 2000.
- B. Stroustrup: The C++ Programming Language. 3rd edition Addison Wesley 1997.

Kapitel 1

Einleitung

1.1 Einstiegsbeispiel zur Vererbung

Klassen sind in einer Halbordnung (*Klassenhierarchie*) angeordnet.

Hauptanwendung: *Varianten*, *Spezialisierung*

Beziehung zwischen Ober-/Unterklasse:

- Jede Methode/ Instanzvariable (Member, Slot) der Oberklasse ist auch Member der Unterklasse
- Unterklassen können Members hinzufügen
- Unterklassen können Methoden umdefinieren
- Jedes Unterklassenobjekt ist automatisch auch Oberklassenobjekt; entsprechende Zuweisungen sind erlaubt

Bsp: Kontoführung in JAVA

```
public class Account {

  protected int balance;
  protected String owner;
  protected int minimumBalance = 1000;

  protected void add(int sum) {
       balance += sum; }
  public void open(String who) {
       owner = who; }
  public void deposit(int sum) {
       add(sum); }
  public void withdraw(int sum) {
       add(-sum); }
  public boolean mayWithdraw(int sum) {
       return balance-sum>=minimumBalance; }
}
```

Die verschiedenen Varianten von Konten (Girokonto, Darlehenskonto) können als Unterklassen dargestellt werden.

```
public class CheckingAccount extends Account {
  protected int overdraftLimit = 0;
  public void setOverdraftLimit(int limit) {
    overdraftLimit = limit; }
  public void printAccountStmt() {...}
  public boolean mayWithdraw(int sum) {
    return (balance-sum) >= (minimumBalance-
      overdraftLimit); }
  }
```

```
public class LoanAccount extends Account {
  protected float interestRate = 10.0;
  protected int amortizationAmount = 0;
  public void setInterest_Rate(float rate) {
    interestRate = rate; }
  public boolean mayWithdraw(int sum) {
    return false; }
  public void withdraw (int sum) throws MyException {
  // Exception MyException muss eigentlich
  // schon in Oberklasse deklariert werden
    throw new MyException("withdraw for loan account",
     owner); }
}
```

Objekte einer Unterklasse können auch als Objekte der Oberklasse verwendet werden. Denn sie haben alle Instanzvariablen und Methoden der Oberklasse. Entsprechende Zuweisungen sind erlaubt. Das umgekehrte gilt jedoch nicht!

```
Account a1 = new Account();
Account a2 = new Account();
Account a3 = new Account();
CheckingAccount a4 =
  new CheckingAccount("Albert Einstein");
LoanAccount a5 = new LoanAccount("Helmut Kohl");
a4.setOverdraftLimit(20000);
a5.setInterestRate(20);
a4.setInterestRate(20); // verboten!
a5.setOverdraftLimit(20000); // verboten!
a4.deposit(100);
a5.withdraw(500);
a1 = a4;
a2 = a5;
a1.deposit(500);
a1.withdraw(42);
```

```
if (a2.mayWithdraw(42)) { ... } ; // dyn. Bindung!
a1.setOverdraftLimit(10000); // verboten!
a2.setInterestRate(10); // verboten!
a4 = a1; // verboten!
a4 = a5; // verboten!
```

Dynamische Bindung entscheidet zur Laufzeit anhand des Objekttyps, welche Methode tatsächlich aufgerufen wird.

Statische Typisierung (falls vorhanden) garantiert, daß es eine passende Methode immer gibt

Bsp:

```
if (...) a1 = a4;
else if (...) a1 = a5;
else a1 = a2;
boolean b = a1.may_withdraw(500);
```

Die if-Bedingungen können von der Eingabe abhängen, sind also statisch nicht bekannt

Demnach weiss man nicht, ob `a1` ein Objekt vom Typ `Account`, `LoanAccount` oder `CheckingAccount` referiert

Der Compiler kann nicht entscheiden, welche `mayWithdraw` Funktion tatsächlich aufgerufen wird

Dies wird zur *Laufzeit* anhand des tatsächlich referierten Objekts entschieden

⇒ Dynamische Bindung ist etwas teurer

1.2 OO vs. imperative Programmierung

typisch: Objekte mit Variationen der Unterart

Beispiel: graphische Objekte (zB `circle`, `rectangle`) nebst Funktionen (z.B. `center`, `move`, `rotate`, `print`)

Insgesamt 8 Codevarianten:

	center	move	rotate	print
circle	c_center	c_move	c_rotate	c_print
rectangle	r_center	r_move	r_rotate	r_print

Pascal/C:

- *Records* zur Repräsentation von Objekten; *variante Records* (C: unions) für verschiedene Objektarten
- Funktionen enthalten *Fallunterscheidung* nach Objekttyp; Funktionalität ist spaltenweise zusammengefaßt

⇒ Geheimnisprinzip verletzt:
Funktionen für `circle` kennen auch Implementierung für `rectangle`

⇒ *Lokalitätsprinzip* verletzt:
neue Objektart erfordert globale Änderung aller Fallunterscheidungen und Funktionen!

OO:

- Klasse für graphische Objekte; Unterklassen für `circle` und `rectangle`
- Jede Unterklasse hat evtl. eigene Implementierung der Funktionen; Funktionalität ist zeilenweise zusammengefaßt

⇒ Geheimnisprinzip gewahrt; einfache Erweiterbarkeit durch neue Unterklassen, alte Funktionen bleiben unverändert

⇒ *Lokalitätsprinzip* ist beachtet:
neue Objektart erfordert nur Hinzufügen einer Unterklasse, der Rest des Codes bleibt unverändert!

⇒ *OO ist softwaretechnisch klar besser*

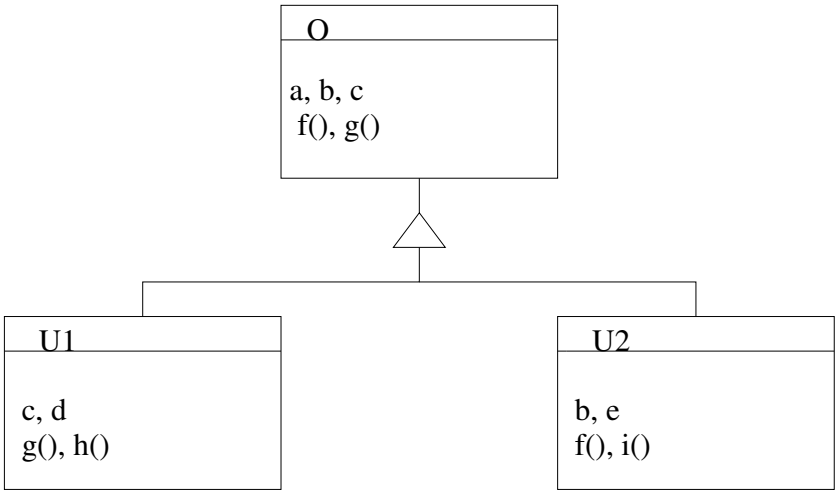
Effizienz ist gleich, da dynamische Bindung durch 1 Arrayzugriff implementiert werden kann (→ *vtable*, s.d.)

kleiner Nachteil: wenn eine neue Fkt für alle Objektarten hinzukommen soll, müssen alle Klassen geändert werden

aber solche „universellen“ Erweiterungen seltener als „artspezifische“ Erweiterungen

1.3 Member-/Methodenzugriff

Darstellung von Vererbung im UML-Diagramm:



Verdeckung/Redefinition: Sei `x: O; y: U1; z: U2`

⇒ `x` sieht `O::a, O::b, O::c, O::f(), O::g()`
`y` sieht `O::a/b, U1::c/d, O::f(), U1::g(), U1::h()`
`z` sieht `O::a/c, U2::b/e, U2::f(), O::g(), U2::i()`

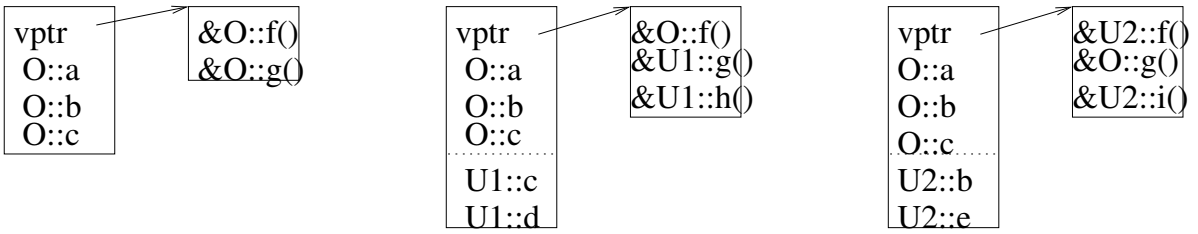
Zugriff auf verdeckte Oberklassenmembers:
C++: `y.O::c, y.O::g()` ;
Java: `super.c, super.g()`
auch `((O)y).c`

Jedoch nicht `((O)y).g()` bzw `x=y; x.g()` !!

Upcasts schalten nicht die dynamische Bindung ab!!

Dieses Verhalten wird verständlich, wenn man die Implementierung betrachtet.

Implementierung: Objektlayout/Methodentabellen (Grundprinzip C++):



- Jedes **new** erzeugt neues Objekt; Methodentabellen gibt es nur einmal
- Im Objekt kommen zuerst ererbte Instanzvariablen (sog. Subobjekt), dann die eigenen
- jedes Objekt enthält Zeiger auf Methodentabelle
- Methodentabelle enthält einen Eintrag für alle (auch ererbte) Methoden
- für jede Methode wird Einsprungadresse gespeichert
- Methoden-Redefinitionen werden durch geänderte Tabelleneinträge dargestellt
- globale Invariante: verschiedene redefinierte Varianten einer Methode haben stets dieselbe Position in der Methodentabelle

⇒ dynamische Bindung kann in konstanter Zeit (1 Arrayzugriff+ 1 Indirektion) realisiert werden

1.4 Varianten des Objektbegriffs

OO-Sprachen unterscheiden sich in 4 Dimensionen:

1. *Objekte* als modulare Berechnungsagenten:

   Variabilität: Sind Objekte ADOs? Wie stark wird das Geheimnisprinzip unterstützt? Können Objekte verteilt und nebenläufig aktiv sein?

2. *Typen* als Invarianten über Variablen (Klassifikation von Ausdrücken):

   Variabilität: Werden elementare Daten (Integers etc) und Objekttypen unterschieden? Statische oder dynamische Typisierung?

3. *Delegation* als Resourcen-Wiederverwendung:

   Variabilität: Gibt es Klassen/Vererbung oder nur objektspezifische „Vorgängerobjekte“? Wiederverwendung nur durch ererbte Oberklassenmethoden, oder volle dynamische Kontrolle von Methodendelegation?

4. *Abstraktion* als Schnittstellenmechanismus:

   Variabilität: statische Interfaces/Zugriffsrechte vs dynamische Zugriffskontrollen/Protokolle; Geheimnisprinzip vs Zugriffsrechte für Klassen

Beispiele:

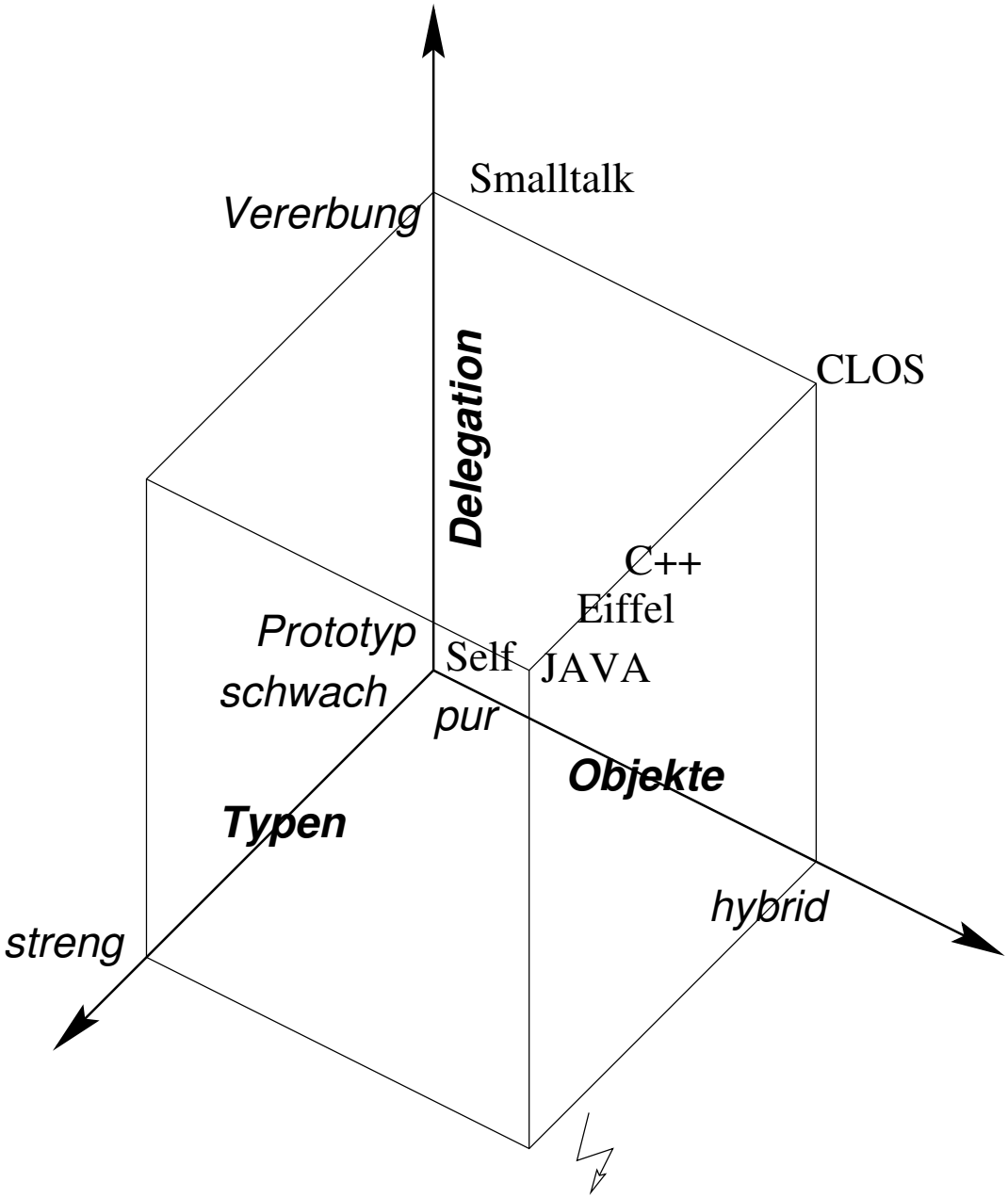
- offene Objekte: Self; geschlossene Objekte/ Zugriffsrechte: Java, C++
- dynamische Typisierung: Smalltalk/Self; statische Typisierung: Java/C++
- pure OO-Sprachen (Smalltalk, Self): alles ist ein Objekt (auch Integers, Klassen, Anweisungen)
- hybride OO-Sprachen (C++, Eiffel, Java): Mischung mit traditionellen Datentypen (Effizienz!)
- Klassen als Objekte: Metaklassen, reflexive Klassen (Smalltalk)
- Objekte als Prototypen/Exemplare: es gibt keine Klassen, nur Oberobjekt-Verweise (Self)

Bem 1: starke Typisierung impliziert nicht Geheimnisprinzip!

Bem 2: objekt*basierte* Sprachen: Objekte/Module, aber keine Vererbung (ADA, Modula2, Visual Basic)

Bem 3: Interessant wäre nähere Betrachtung der dynamisch typisierten Sprachen wie Smalltalk oder der prototypbasierten, klassenfreien Sprachen wie Self (Ungar 87), aber darauf müssen wir verzichten (vgl. Eliens bzw Abadi/Cardelli)

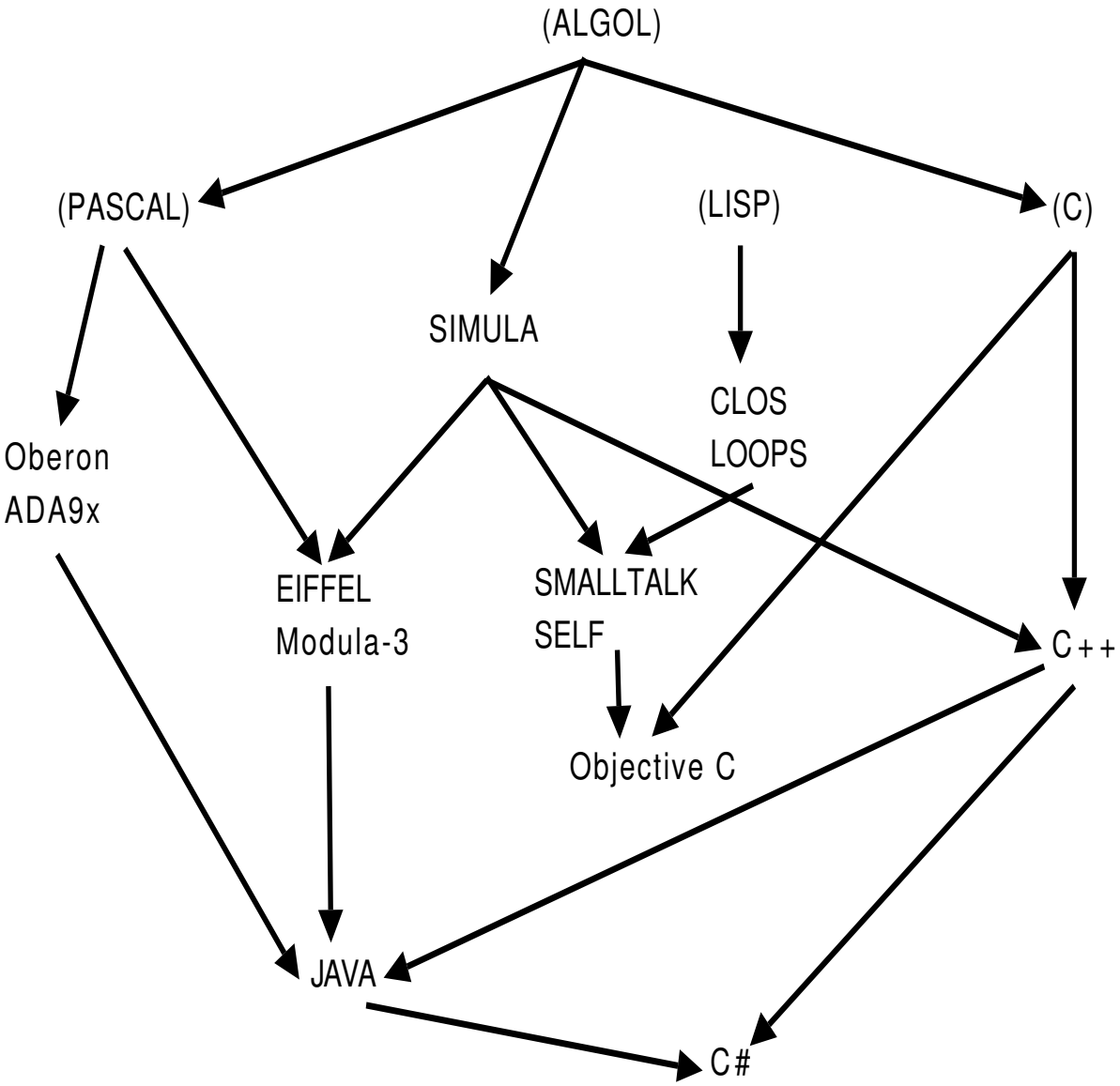
Darstellung von 3 Dimensionen:



Es sind noch nicht alle Würfelecken vergeben :-)

1.5 OO-Sprachen

Historische Vererbungsbeziehungen:



Kapitel 2

Übersicht über wichtige OO-Sprachen

2.1 Smalltalk

... Extraskript ...

2.2 Java 1.5

... Extraskript ...

2.3 C++

... Extraskript ...

2.4 C#

... Extraskript ...