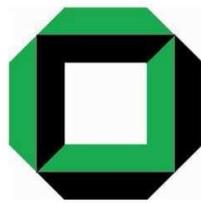


Fortgeschrittene Objektorientierung

SS 2008

Prof. Dr.-Ing. G. Snelting



Universität
Karlsruhe

Fakultät für Informatik

©2008 Universität Karlsruhe
Lehrstuhl Programmierparadigmen

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

1	Einleitung	9
1.1	Einstiegsbeispiel zur Vererbung	9
1.2	OO vs. imperative Programmierung	13
1.3	Member-/Methodenzugriff	15
1.4	Varianten des Objektbegriffs	17
1.5	OO-Sprachen	20
2	Übersicht über wichtige OO-Sprachen	21
2.1	Smalltalk	21
2.2	Java 1.5	21
2.3	C++	21
2.4	C#	21
3	Tücken der dynamischen Bindung	22
3.1	this-Pointer	22
3.2	Dynamische Bindung und Rekursion	24
3.3	Dynamische Bindung und Evolution	26
3.4	Type Casts	30
3.5	Super	31

3.6	Statische Variablenbindung	32
4	Mehrfachvererbung	34
4.1	Interface-Mehrfachvererbung	36
4.2	Multiple Subobjekte in C++	38
4.3	Subobjektgraphen	39
4.4	Static Lookup	40
4.5	Dynamische Bindung bei Rossie/Friedmann	44
4.6	Rossie/Friedmann und C++	45
5	Der vtable-Mechanismus	46
5.1	C++: Objektlayout	46
5.2	C++: Type Casts	48
5.3	C++: vtables	49
5.4	Mehrfachvererbung	50
6	Überladungen	55
6.1	Überladung und dynamische Bindung	58
6.2	Smart Pointers	59
6.3	Function Objects	63
6.4	Multimethoden	63
7	Invarianten und sichere Vererbung	68
7.1	Subtyping/Verhaltenskonformanz	69
7.2	Inheritance/Spezialisierung	71
7.3	Beispiel	72
7.4	Quadrat/Rechteck	75

7.5	Inheritance is not Subtyping	77
7.6	Vererbung vs. Delegation	78
8	Generische Klassen	81
8.1	Generische Klassen in Java	83
8.2	Wildcards	86
8.3	Generische Programmierung in C++	88
9	Inner Classes	89
9.1	Wiederholung: Iteratoren	89
9.2	Inner Classes	94
10	Event Handling	97
10.1	MVC	97
10.2	Observer in C++	99
10.3	Events in Java	103
10.4	Callbacks in C++	107
11	Refactoring	111
11.1	Die Demeter-Regel (Lieberherr 89)	111
11.2	Refactoring im Überblick	113
11.3	Beispiel: Der Videoverleih im Detail	114
11.4	Ein Refactoring-Katalog	140
11.5	Refactoring bestehenden Codes	142
12	Design Patterns	144
12.1	Das Role-Pattern	144

12.2 Wiederholung: Composite	146
12.3 Wiederholung: Strategy	148
12.4 Visitor	149
12.5 Factory	154
12.6 Abschlussbemerkung zu Design Patterns	158
13 Aspekt Orientierte Programmierung	159
13.1 Aspekte in Apache	159
13.2 Beispiel: Figurenzeichnen	162
13.3 Aspekt-orientierte Programmierung	163
13.4 AspectJ	164
13.5 Aspect Weaver	169
13.6 Typische Beispiele für Aspekte	169
13.7 Zusammenfassung	172
14 Traits and Mixins	173
14.1 Grundbegriffe der Komponententechnologie	173
14.2 Komponenten in Scala	174
14.3 Mixins	176
14.4 Beispiel 1: Symboltabellen	178
14.5 Beispiel 2: Logging	186
14.6 Beispiel 3: Visitor Pattern	188
15 Virtuelle Klassen	191

16 Cardelli-Typsystem	197
16.1 Typkonversionen	200
16.2 Kontravarianz	202
16.3 Kontravarianz und dynamische Bindung	205
16.4 Typkonstruktoren	206
16.5 Die Array-Anomalie in JAVA	207
17 Generizität, Abstraktion, Rekursion	210
17.1 Generische Klassen	210
17.2 Vererbung und Generizität	212
17.3 Schnitt-Typen	214
17.4 Existential Types	215
17.5 Rekursive Typen (Cook 1990)	217
18 Palsberg-Schwartzbach Typinferenz	222
18.1 Elementare Regeln	224
18.2 Typinferenz	227
18.3 Lösen von Mengungleichungssystemen	232
19 Analyseverfahren	234
19.1 Rapid Type Analysis	235
19.2 RTA als Constraint-Problem	239
19.3 Points-to Analyse	240
19.4 Points-to für OO	245
19.5 Snelling/Tip Analyse (KABA)	248

20 Ownership Types	249
20.1 Das Problem: Pointer Spaghetti	249
21 Semantik	252
21.1 Grundbegriffe	253
21.2 Semantikregeln	254
21.3 Hoare-Kalkül	256
21.4 Small Step Semantik	257
21.5 Typsicherheit	260
22 Bytecode, JVM, Dynamische Compilierung	262
22.1 Laufzeitorganisation	263
22.2 Bytecode	267
22.3 Methodenaufruf	271
22.4 Just-in-Time Compiler	275
22.5 Bytecode Verifier	277
23 Garbage Collection	278
23.1 Copy-Kollektor	279
23.2 Generational Scavenging	279
23.3 neue Verfahren	280

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!

```
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:

	<code>center</code>	<code>move</code>	<code>rotate</code>	<code>print</code>
<code>circle</code>	<code>c_center</code>	<code>c_move</code>	<code>c_rotate</code>	<code>c_print</code>
<code>rectangle</code>	<code>r_center</code>	<code>r_move</code>	<code>r_rotate</code>	<code>r_print</code>

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*

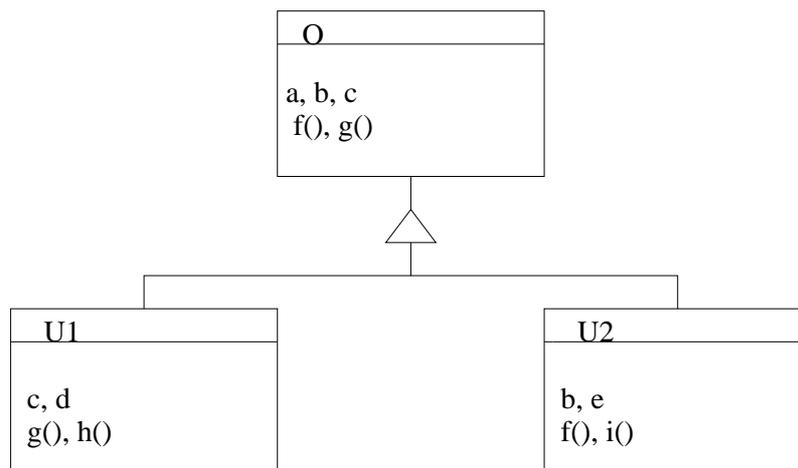
Effizienz ist gleich, da dynamische Bindung durch 1 Arrayzugriff implementiert werden kann (→ *vtable*, s.d.)

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 $x: O$; $y: U1$; $z: U2$

$\Rightarrow x$ 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()`

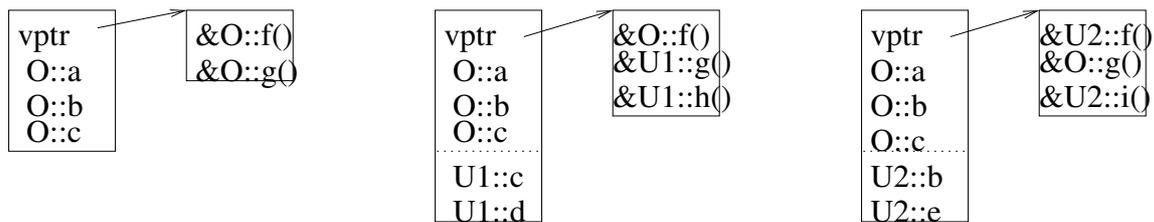
auch `((O)y).c`

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 Einsprungsadresse 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 Ressourcen-Wiederverwendung:

Variabilität: Gibt es Klassen/Vererbung oder nur objekt-spezifische „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:

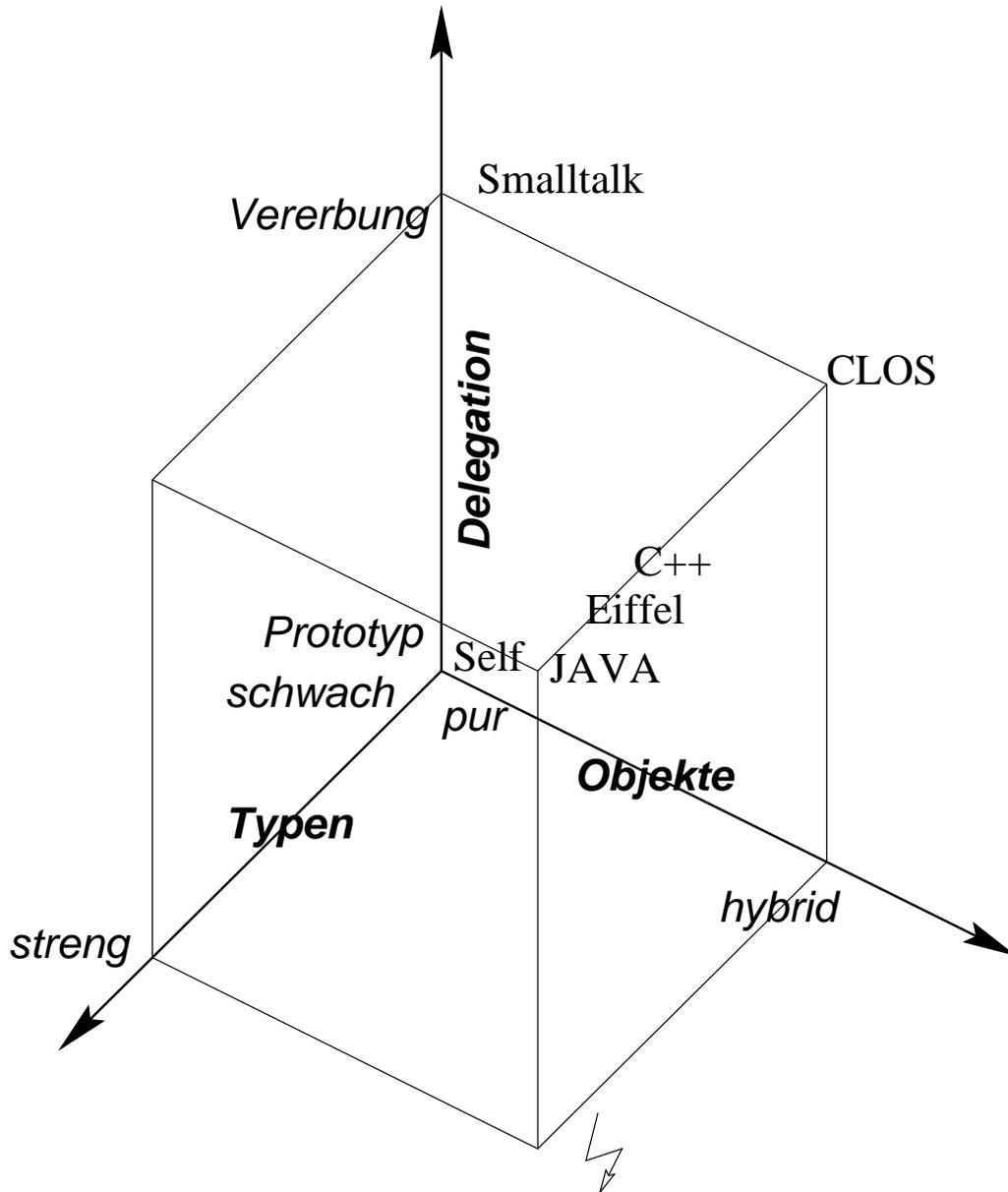
- offene Objekte: Self; geschlossene Objekte/ Zugriffsrechte: Java, C++
- dynamische Typisierung: Smalltalk/Self; statische Typisierung: Java/C++
- pure OO-Sprachen (Smalltalk, Self): alles ist ein Objekt (auch Integers, Klassen, Anweisungen)
- hybride OO-Sprachen (C++, Eiffel, Java): Mischung mit traditionellen 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: objektbasierte 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 Würfecken vergeben :-)

1.5 OO-Sprachen

Historische Vererbungsbeziehungen:

