Vererbung

1. Grundideen
2. Verwendung und Auswirkungen, Vor- und Nachteile
3. Dimensionen des Vererbungskonzeptes
   - Vererbungsstruktur
   - Qualität
   - Granularität
4. Ausblick

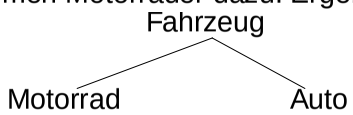
Was ist Vererbung?

Vererbung (*inheritance*) ist ein Mechanismus, neue Klassen aus existierenden Klassen zu definieren. Eine aus einer bestehenden Klasse (= **Oberklasse**) abgeleitete **Unterklasse** erbt alle Instanzvariablen und Operationen (Spezifikation und Implementierung) der Oberklasse. Eine Unterklasse kann

1. neue Instanzvariable und/oder Operationen definieren
2. Implementierung geerbter Operationen überschreiben
3. geerbte Operationen durch eigenen Code ergänzen

Was ist Vererbung (2)

- Im folgenden verwendetes Synonym für Vererbung: **Subklassenbildung**
- Aber: Die durch den Begriff “Subklassenbildung” vorgegebene Richtung entspricht nicht immer der Entstehungsreihenfolge
- Beispiel:
  Verwalte in einem neuen System zuerst Informationen über Autos
- Später kommen Motorräder dazu. Ergebnis:



Heuristiken für Subklassenbildung

Erzeugung von neuen Klassen aus anderen zum Zwecke der: (Einteilung nach Budd, 97)

- Spezialisierung
- Spezifikation
- Konstruktion
- Generalisierung
- Erweiterung
- Einschränkung
- Abänderung

Verwendung von Vererbung

Spezialisierung

- semantische Beziehung zwischen dem Verhalten von Ober- und Unterklasse
- Subklasse ist ein Spezialfall und verhält sich entsprechend der Oberklasse (Substitutsprinzip)
- "*Is-a*" Beziehung aus der AI-Literatur (semantische Netze, Frames): "a Triangle *is a* geometric object".
  nAber: dort oft vereinfachte Sichtweise, weil statisch
- Unterscheidung zu Teil-Beziehungen (*has-part, is-part-of*): Motor/Kolben

Verwendung von Vererbung

Spezifikation

- Festlegung einer gemeinsamen Schnittstelle für alle Subklassen (evt. nicht einmal Erweiterung durch Unterklasse)
- Sonderfall der Spezialisierung, aber Subklassen sind nicht Verfeinerung eines konkreten Typs
- Oberklasse stellt (unvollständige) abstrakte Spezifikation dar (⇒ **Abstrakte Klasse**). *Beispiel*: Klasse Collection in Smalltalk
- Zusammenhang mit *Typen* (später)

Verwendung von Vererbung

Konstruktion

- Funktionalität der Oberklasse wird übernommen, aber evt. das Interface geändert, z.B. neue Namen für existierende Methoden.

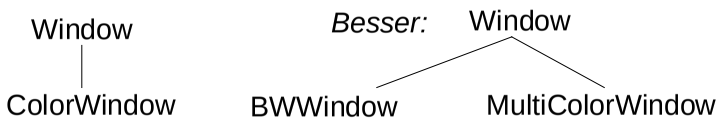
Bsp: Symboltabelle in Compiler. *Schlüsselwert* - Name des Symbols*, Eintrag* - ein Record, der den Typ/die Prozedur beschreibt

Zusätzliche Methoden: z.B. Überprüfung semantischer Programminformation bei Änderungen, Zugriff auf bestimmte Felder im Record.

Verwendung von Vererbung

Generalisierung "nach unten"

- Verallgemeinerung der Funktionalität einer unveränderbaren Klasse (widerspricht der Spezialisierung)



- tritt meistens auf, wenn Design stark datenorientiert
- nur wenn nicht vermeidbar

Verwendung von Vererbung

Erweiterung

- Hinzufügen komplett neuer, unabhängiger Eigenschaften
- Beispiel 1: “Suche nach Präfix” für Collections, die Strings enthalten
- Beispiel 2: Listenattribut (z.B. Pointer ‘next’) für Personen, die ich in einer Liste halten möchte

Verwendung von Vererbung

Einschränkung des Verhaltens

- Abschaffung von Operationen
- Beispiel: Stack als Subklasse von Liste

⇒ *widerspricht Spezialisierung*

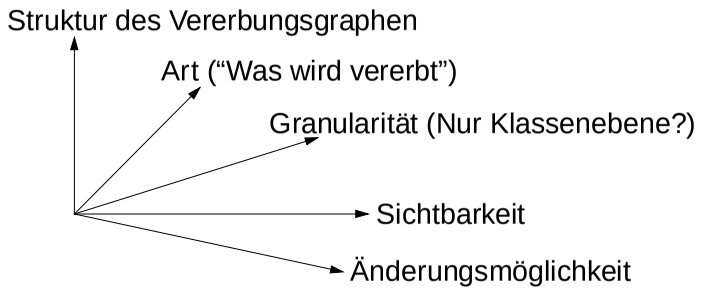
eigentlich zu vermeiden

Abänderung/Varianten

- 2 Klassen, die ähnlich sind, aber keine klare Unterordnung definiert haben (*Bsp.*: Maus - Digitizer)
- Besser: abstrakte Oberklasse einführen, gemeinsames Verhalten dorthin “ausfaktorisieren”

Wie funktioniert Vererbung?

- Vererbung ist nicht eindeutig definiert
- Unterscheidung nach verschiedenen Dimensionen



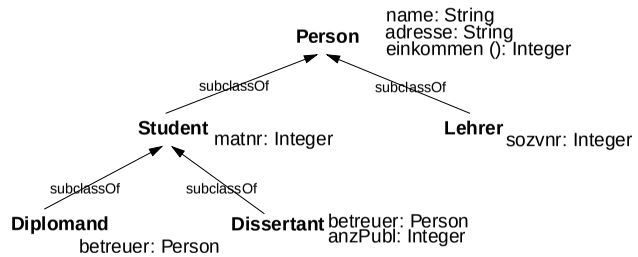
Vererbungsstruktur

Welche Einschränkungen gelten für die Struktur des Vererbungsgraphen?

- Einfache Vererbung
- Mehrfache Vererbung
- Wiederholte Vererbung

Einfache Vererbung

Jede Klasse hat genau eine direkte Oberklasse (mit Ausnahme der Wurzelklasse).



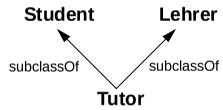
➥ *Vererbungshierarchie (inheritance hierarchy)*

Mehrfache Vererbung

Bei der **Mehrfachvererbun**g (*multiple inheritance*) erbt eine Klasse Instanzvariablen und Operationen von mehreren direkten Oberklassen.

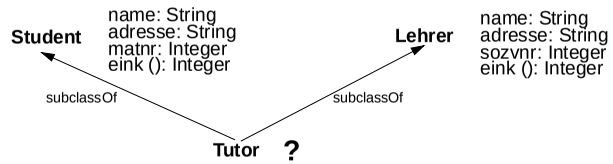
Das Verhalten der Klasse ist eine *Kombination der Verhalten* der Oberklassen.

➥ *Vererbungsgraph (DAG, inheritance lattice)*



Namenskonflikte bei Mehrfachvererbung

Beispiel:



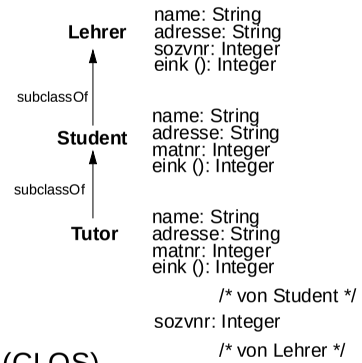
verschiedene Lösungsstrategien:

1. Linearisierung des Vererbungsgraphen
2. Umbenennen
3. Qualifizieren / Überschreiben
4. Auswählen
5. “Meet” Strategie

Linearisierung des Vererbungsgraphen

zu einem Vererbungspfad ohne Duplikate. Weitere Behandlung wie mit einfacher Vererbung.

Beispiel: **Tutor** subclassOf **Student**, **Lehrer**



Probleme:

- systemweit eindeutige Namen für spätere Vererbung
- willkürliche Auswahl und daher nicht vorgesehene Vererbungsbeziehung (**Student** subclassOf **Lehrer**)

Sprachen: Flavors, CommonLoops (CLOS)

Umbenennen der geerbten Information

Instanzvariable und Operationen werden in der erbenden Klasse umbenannt.

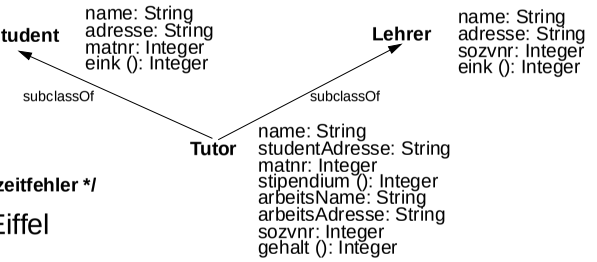
Beispiel: **Tutor** subclassOf

Student rename **adresse** as **studentAdresse**, **eink** as **stipendium**

Lehrer rename **adresse** as **arbeitsAdresse**, **name** as **arbeitsName**

Problem:
Polymorphismus und dynamisches Binden führt zu Typfehler:

```
s: Student;
t: Tutor;
...
s := t;
s.adresse /* führt zu Laufzeitfehler */
```



Sprachen: Eiffel

Qualifizieren / Überschreiben

Instanzvariable und Operationen werden bei ihrer Verwendung mit der Klasse **qualifiziert**, von der ihre Spezifikation (und Implementierung) genommen werden soll.

Beispiel: **Tutor** : public **Student**, public **Lehrer**

```
public: gesamt_eink() : Integer     /* neue Operation */
        { return Student::eink + Lehrer::eink };
```

Operationen werden in erbender Klasse **überschrieben**.

Beispiel: **Tutor** : public **Student**, public **Lehrer**

```
public: eink() : Integer            /* überschriebene Operation */
        { return Student::eink + Lehrer::eink };
```

Sprachen: C++, O_2, Eiffel (nur Überschreiben)

Entwurfsrichtlinie für Qualifizieren / Überschreiben

Überschreiben ist bessere Lösung als Qualifizieren

- Auflösen der Mehrdeutigkeit
- Erbende Klasse erbt eindeutige Bezeichner

Auswählen

In der erbenden Klasse wird angegeben, von welcher Oberklasse die Definitionen von mehrfach vorhandenen Instanzvariablen und Operationen geerbt werden.

Beispiel:

add class **Tutor** : inherits **Student**, **Lehrer**

...

method **eink**from **Lehrer**;

Systeme: O_2

"Meet" Strategie

von Cardelli (1984);
definiert eine eindeutige Semantik für die mehrfache Vererbung von Instanzvariablen in streng getypten Sprachen; Instanzvariable haben einen Tupeltyp.

$T_1 = [a_1:t_1, a_2:t_2, a_3:t_3]$, $T_2 = [a_3:t_{3'}, a_4:t_4]$

T_1 *meet* T_2 ist der *größte gemeinsame Teiler*, wobei die Subtypbeziehung zur Berechnung des *meet* benutzt wird.

$T_3 = T_1$ *meet* $T_2 = [a_1:t_1, a_2:t_2, a_3:t_3$ *meet* $t_{3'}, a_4:t_4]$

Beispiel: Erwachsen = [30..150], JungErwachsen = [19..40]
Erwachsen *meet* JungErwachsen = [30..40]

Nachteil: nicht auf Operationen anwendbar!

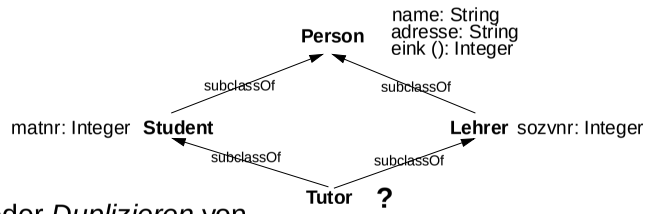
Zusammenfassung - Namenskonflikte lösen

	Umbenennen	Überschreiben	Qualifizieren	Auswählen	Linearisierung
Instanzvar.	Eiffel	Eiffel, O_2	O_2, C++		CLOS
Operationen	Eiffel	Eiffel, O_2, C++	O_2, C++	O_2	CLOS

Wiederholte Vererbung

Ein Spezialfall der mehrfachen Vererbung ist die **wiederholte Vererbung** (*repeated inheritance*), bei der eine Klasse eine Oberklasse mehrmals über verschiedene Pfade erbt.

➥ *Vererbungsverband (inheritance lattice)*



Problem:
Teilen oder *Duplizieren* von Instanzvariablen und Operationen der Wurzelklasse?

Mögliche Lösungen (1)

in C++:

- Instanzvariable und Operationen werden *n* mal über *n* Pfade von der Wurzelklasse vererbt.

Lösung 1: Qualifizieren und Überschreiben

```
Student : public Person;
Lehrer: public Person;
Tutor: public Student, public Lehrer; /* name und adresse 2x von Person geerbt */
        public: anschrift () : String { return Student::adresse };
```

Lösung 2: Def. der Wurzelklasse als *virtuelle Basisklasse.*

```
Student : public virtual Person;
Lehrer: public virtual Person;
Tutor: public Student, public Lehrer; /* name und adresse 1x von Person geerbt */
        public: anschrift () : String { return adresse };
```

Mögliche Lösungen (2)

Lösung 2 in C++ (cont'd):

Achtung:

Wurzelklasse muß von allen direkten Unterklassen als virtuelle Basisklasse definiert sein; sonst Mehrfachvererbung.

Student : public virtual **Person**;
Lehrer: public virtual **Person**;
WiHi: public **Person**;
Tutor: public **Student**, public **Lehrer**, public **WiHi**;
/* name und adresse **1x** von **Person** via **Student** und **Lehrer** geerbt */
/* name und adresse **1x** von **Person** via **WiHi** geerbt */
...
public: **anschrift1** () : String { return **Student::adresse** };
public: **anschrift2** () : String { return **WiHi::adresse** };

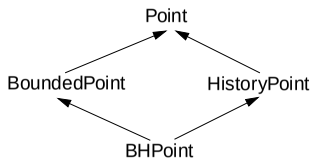
Mögliche Lösungen (3)

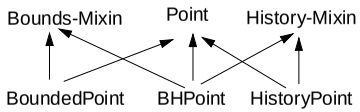
in Eiffel:

- Instanzvariable und Operationen werden einmal vererbt, falls sie in *keinem* Vererbungspfad umbenannt wurden.
- Sonst:
  n Duplikate für *n* Vererbungspfade → Umbenennen.

Mögliche Lösungen (3) - Mixins

Eine ***Mixin***-Klasse spezifiziert ein singuläres, in sich abgeschlossenes Verhalten. Entworfen um *vererbt* (***mixed-in***) zu werden, *nicht* um *instanziert* zu werden.

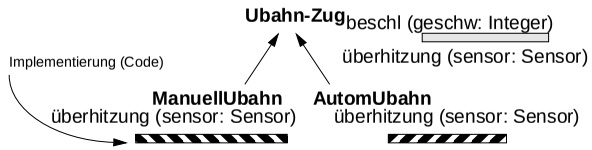




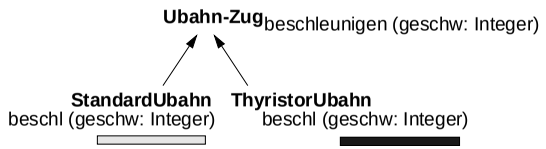
Sprachen: CLOS

Was bringt Multiple Inheritance?

- Steuerung eines U-Bahn-Systems

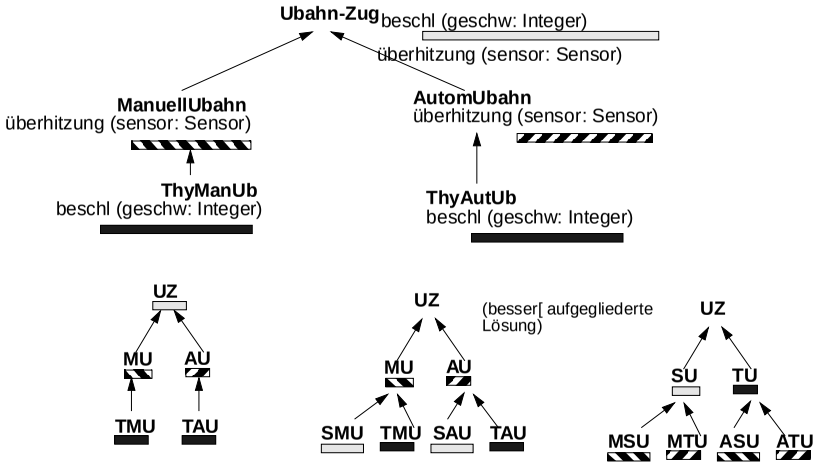


- Einführung eines neuen Zugtyps



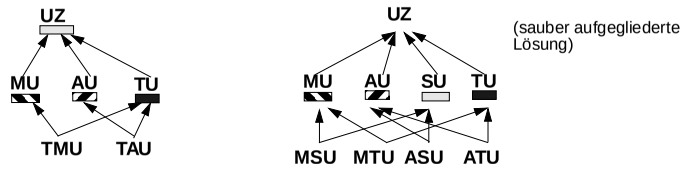
Lösungsansätze (1)

- Lösung mit Single Inheritance

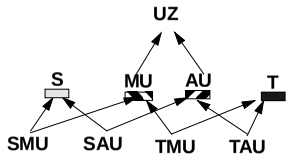


Lösungsansätze (2)

- Lösung mit Multiple Inheritance

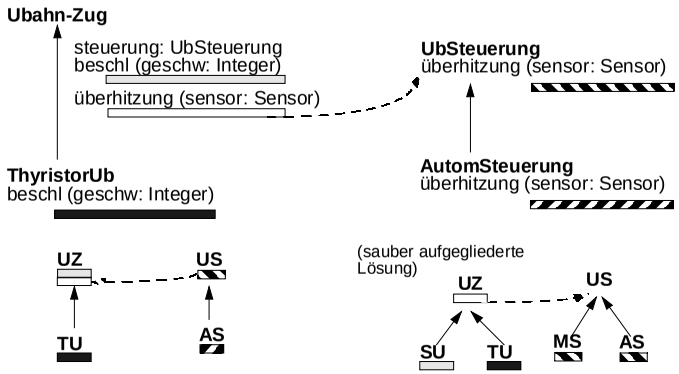


- Lösung mit Mixins



Lösungsansätze (3)

- Lösung mit Zusammensetzung (Composition)



Part-of-Beziehung, über Instanzvariable implementiert

Art der Vererbung

Was wird eigentlich vererbt? Code, Spezifikation,beides?
Welche Beziehung zw. Klassen wird dadurch ausgedrückt?

- Implementierungsvererbung
  ➥ Subklassen-Beziehung
- Spezifikationsvererbung
  ➥ Subtypen-Beziehung
- Spezialisationsvererbung
  ➥ Spezialisierung (“isA-Beziehung”)

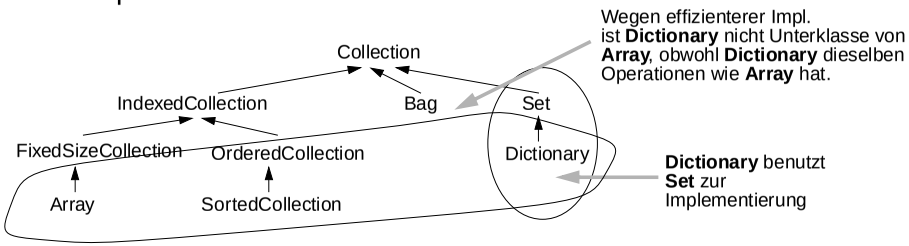
Implementierungsvererbung (1)

Vererbung ermöglicht, eine Klasse *C* als Modifikation oder Erweiterung bestehender Klassen $C_1..C_n$ zu definieren.

Motivation: Code-sharing (Code-Vererbung)

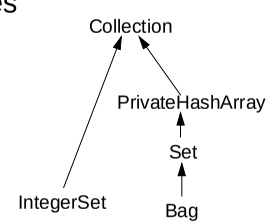
Wiederverwendung für den **Implementierer**

Beispiel: Teile der Smalltalk-Klassenhierarchie



Implementierungsvererbung (2)

- An Smalltalk angelehntes hypothetisches Beispiel
- Set und Bag sind ähnlich, also Subklassen
- PrivateHashArray: separate Definition des Hashalgorithmus für leichtere Erweiterung (z.B. Dictionary)
- IntegerSet: Bitstring-Implementierung, daher keine Subklasse von Set



Implementierungsvererbung (3)

Implementierungsvererbung zwischen Klasse *C* und Oberklassen $C_1..C_n$ impliziert nicht Subtyp-Beziehung

> d.h., *C* versteht nicht notwendigerweise alle Operationen, die in $C_1..C_n$ bekannt sind.

Beispiel aus der Smalltalk-Klassenhierarchie:

- **FixedSizeCollection** ist Subklasse von **IndexedCollection**.
- **IndexedCollection** erbt Methode **add**: von **Collection**
- **add**: wird in **FixedSizeCollection** mit dem body "self **shouldNotImplement**" überschrieben.

Implementierungsvererbung (4)

Die nur teilweise Vererbung von Instanzvariablen und Operationen von Oberklassen in Unterklassen bezeichnet man auch als

> *selektive (= partielle) Vererbung*

- Sehr flexibel, aber sehr komplexe Klassenhierarchien möglich.

Sprachen:
Smalltalk (Überschreiben mit **#shouldNotImplement**),
CLOS
C++

Zusammensetzung

Codevererbung oft verwendet für *Benutzung* des Codes von $C_1..C_n$ (Oberklassen) durch *C* (Unterklasse). Zur *Konstruktion* von *C* wird sowohl der Code als auch die interne Repräsentation von $C_1..C_n$ benutzt.

```
class SymbolTable : Dictionary {
        addConstant(c); memoryUsage(s);.. };
```

Zusammensetzung (auch: Client/Server-Prinzip):

Realisierung über Instanzvariable.

```
class SymbolTable {
        d : Dictionary;
        addConstant(c); { d->add(c); }
        memoryUsage(s); { (d->at(s))->memoryUsage
        size(); { d->size() }; .. };
```


Zusammensetzung

Vorteile der Zusammensetzung:

- Vererbung von Operationen, die nicht gebraucht werden, ist ausgeschlossen - eindeutig definiertes Interface
- Änderung der Implementierung ist möglich ohne Änderung der Vererbungsbeziehungen
- Vererbung kann der Implementierung einer semantischen Beziehung zwischen Unter- und Oberklassen vorbehalten werden
- Dictionary-Semantik ist über die Schnittstelle klar definiert

Zusammensetzung

Nachteile der Zusammensetzung:

- Für jede in der Klientenklasse benötigte und in der Anbieterklasse vorhandene Operation muß Spez. und Impl. in der benutzenden Klasse geschrieben werden (Aufruf der benötigten Operation in der Anbieterklasse), daher mehr Code und aufwendiger zu entwickeln
- Wenn große Ähnlichkeit gegeben ist, kann das für die Mehrheit der Methoden der Fall sein
- Längerer Code eventuell schwerer verständlich
- Eine Aufrufebene mehr - langsamerer Code

Der Yo-yo-Effekt

Durch wiederholtes Überschreiben einer Operation in Unterklassen und Aufruf derselben (überschriebenen) Operation durch Qualifikation in den Oberklassen kann es zu komplexen Ausführungsfolgen kommen, die mehrere Klassen in der Klassenhierarchie betreffen.

Beispiel: vordefinierte Variablen *self* und *super* in Smalltalk.

- *self* zeigt immer auf das Objekt, das aktueller Empfänger einer Nachricht ist.
- *super* zeigt immer auf die direkte Oberklasse jener Klasse, deren Code gerade ausgeführt wird.

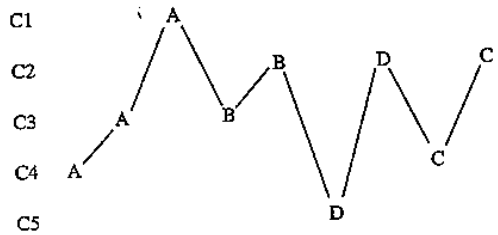
(*Ref*: D. Taenzer et al.: Object-Oriented Software Reuse: The Yo-yo Problem, JOOP, Sept/Okt 1989.)

Der Yo-yo-Effekt - Beispiel

Class	Method A	Method B	Method C	Method D
C1	implements sends *self B & C*			
C2		implements sends *self D*	implements	implements
C3	refines sends *super A*	refines sends *super B*		
C4	refines sends *super A*		refines sends *super C*	
C5				refines sends *super D*

subclassOf

Implementierung und Überschreiben der Methoden A bis D unter Verwendung von *self* und *super*



Ausführungsfolge von Operationen A bis D in den Klassen C1 bis C5 aufgrund des Sendens der Nachricht A an eine Instanz der Klasse C5

Das Yo-yo-Problem?

1. Das Yo-yo-Problem ist ein Problem der Betrachtung des Codes (Code-Vererbung)
2. Werden vererbte Methoden in erster Linie auf *Spezifikationsebene* oder *Spezialisierungsebene* (“is-A”) betrachtet, verschwindet das Problem.

Beispiel: Methode **#do:** in der Klasse Collection und Unterklassen. Bis zu 30 Implementierungen in verschiedenen Klassen, aber keine Verständnisprobleme aufgrund klarer Spezifikation.

Spezifikationsvererbung (1)

- Spezifikation = Beschreibung des Verhaltens mit eingeschränkter Ausdruckskraft
- Typen: eine Möglichkeit der Spezifikation in Programmen
- Analog zur Subklassenbildung für Typen: Subtypbildung
- **Wichtig**: Spezifikation ist natürlich nicht vollständig
  - Es ist nicht garantiert, daß der Code das Richtige tut, auch wenn die Spezifikation erfüllt ist
  - *Meistens* beschränkt sich die Spezifikation ohnehin auf die Signatur der Operationen

Typkonzept

Ein **Typ** (*type*) spezifiziert das Verhalten einer Menge von Datenwerten (in konventionellen Sprachen) oder Objekten (in oo Systemen).

Vorteile von getypten Systemen:

1. *disziplinierte Programmierung*: jede Variable hat Typ
2. *Typüberprüfung* so weit wie möglich zur Übersetzungszeit
3. *Dokumentation*: jeder Typ beschreibt das Verhalten eines Teil des Programmes (siehe Smalltalk-Namenskonventionen als Ersatz für fehlende Typisierung!)

Arten von Typen (strukturell)

- atomare (primitive) Typen:

  konkrete Datentypen (oft auch “printable” genannt, weil sie direkt dargestellt werden können):
  integer, char, real, bool, ...

- komplexe Typen:

  Typkonstruktoren: tupel, list, set, array, ...
  (sollen orthogonal sein, d.h., Reihenfolge und Anzahl der Anwendung ist irrelevant)

Subtyp und Typersetzbarkeit

Ein Typ *S* ist **Subtyp** (*subtype*) von Typ *T* (*S* ≤ *T*), wenn *S* dasselbe Verhalten wie *T* und optional zusätzliches hat.

Wenn *S* ≤ *T*, dann kann an jeder Stelle, an der eine Instanz von *T* erwartet wird, eine Instanz von *S* verwendet werden.

➥ Prinzip der Typersetzbarkeit

Subtypbeziehungen ergeben sich aus einer formalen Spezifikation der wesentlichen Eigenschaften eines Objekts:

- Struktur (bei abstrakten Datentypen an sich nicht nötig!)
- Operationen

Typüberprüfung (1)

Statisch getypte Sprachen (*statically typed languages*) sind Sprachen, bei denen der Typ jedes Ausdrucks zur Übersetzungszeit bekannt ist
➞ statische (zur Übersetzungszeit) Typüberprüfung!

Streng getypte Sprachen (*strongly typed languages*) sind Sprachen, bei denen der Typ jedes Ausdrucks nicht unbedingt zur Übersetzungszeit bekannt sein muß, aber jeder Ausdruck typkompatibel ist
➞ statische und dynamische (zur Laufzeit) Typüberprüfung!

Schwach getypte Sprachen (*weakly typed languages*) sind Sprachen, bei denen der Typ jedes Ausdrucks erst zur Laufzeit bekannt ist ➞ dynamische Typüberprüfung!

Warum ist die Subtyp-Beziehung relevant?

Ziel: maximale Wiederverwendung für den Benutzer, der neue Systeme spezifiziert / entwirft.

➥ dazu brauchen wir *Typersetzbarkeit* !

Beispiel: einEmpfänger.nachricht (aCollection, aMagnitude)

jedes Objekt, das anstelle von aCollection bzw. aMagnitude verwendet werden kann, ist ein gültiger Parameterwert:
für aCollection kann stehen Set, Dictionary, String
für aMagnitude kann stehen Char, String, Integer.

Subtyp-Beziehung (≤) ist eine *semantische Beziehung* zwischen Typen, die *Typersetzbarkeit* und *Wiederverwendung von Spezifikation* erlaubt.
=> Voraussetzung für Polymorphismus bei strong typing

Objekttyp und Subtypenbildung

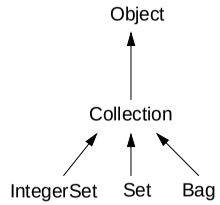
- Der Typ eines Objekts umschreibt dessen Verhalten und Struktur (Spezifikation)
- Eine Subtypenbeziehung zwischen Objekten ist daher abhängig von ihrer beider Verhalten und Struktur

➥ Subtypbeziehung hängt im Prinzip von den Attributen (Instanzvariablen) und den Operationen ab (“Typ” von Variablen und “Typ” von Operationen)

➥ In der Praxis (Datenkapselung) vor allem die *Operationen* wichtig!

Beispiel zur Subtypenbildung

- Sind die verschiedenen Beispielklassen gegenseitig ersetzbar?
- IntegerSet: zu eingeschränkt vom Elementtyp her
- Set, Bag: unterstützen nicht spezielle Operationen v. IntegerSet (z.B. sum)



- Bag: funktioniert bei Eintragen und Löschen anders als Set (und umgekehrt)!
- Auf Typebene keine Unterordnung
- Klasse PrivateHashArray sollte eigentlich nicht sichtbar sein, da nur für Implementierungszwecke

Beispiel (2)

- Spezifikation des Verhaltens kann in Teilbereiche zerlegt werden:
  - eine Klasse - mehrere Typen
  - mehrere Klassen mit demselben Typ
- Beispiel für Collections:
  - Iteration (#do:, #select:, etc.)
  - Hinzufügen/Entfernen (#add:, #remove:)
  - Sortierreihenfolge (1-n, benutzerdefiniert, zufällig)
- Achtung: das ist eine spezielle Betrachtungsweise, wird nicht von Typsystemen unterstützt!

Regeln zur Subtypenbildung (1)

1. *atomare (primitive) Datentypen:*

   Typ = Menge von Objekten ➞ Subtyp = Untermenge

   Beispiel:

   Typ T = Integer,

   Subtypen T_1 = [1..100], T_2 = Prim, T_3 = Even

   Hülleneigenschaft ist erfüllt in T_3 (z.B.: Even + Even -> Even), nicht in T_1 und T_2. Möglich bei Redefinition der Parameter (z.B.: Prim + Prim -> Integer)

Regeln zur Subtypenbildung (2)

2. *Komplexe Typen - Tupeltypen (records):*

$T_1 = [a_1:t_1, a_2:t_2, .., a_n:t_n] \leq T_2 = [a_1:t'_1, a_2:t'_2, .., a_m:t'_m]$,

$\Leftrightarrow t_i \leq t'_i$ für $1 \leq i \leq m$ und $m \leq n$

Beispiel:

type **Person** = [name : String, alter : [0..120]]

type **Lehrer** = [name : String, alter : [20..60], gehalt : Integer] ≤ type **Person**

3. *Komplexe Typen - Sammeltypen (Collections):*

$T_1 = \{t_1\} \leq T_2 = \{t'_1\} \Leftrightarrow t_1 \leq t'_1$

Beispiel: {**Lehrer**} ≤ {**Person**} ⇔ **Lehrer** ≤ **Person**

Regeln zur Subtypenbildung (3)

4. *Operationen:*

➥ *syntaktische Prüfung*

Signatur (op_1)= ($T_1 \rightarrow T_2$) ≤ Signatur(op_2)= ($T_3 \rightarrow T_4$)?

- Name muß gleich sein:
  "+" kann nie einen Subtyp von "-" haben.
- Kovarianzregel
- Kontravarianzregel
- (strenge) Nonvarianz

Kovarianz

$T_1 \leq T_3 \wedge T_2 \leq T_4 \Rightarrow (T_1 \rightarrow T_2) \leq (T_3 \rightarrow T_4)$

Beispiele:

+ (Nat → Nat) ≤ + (Int → Int)

(steuern(eink: [0..10000]) → [0..5000]) ≤ (steuern(eink:Int) →Int)

- intuitiv, aber verletzt statische Typüberprüfung von dynamisch gebundenen, polymorphen Variablen:

p: Person; q: Pensionist;
p := q;
print(p.steuern(5 000 000)); ✕

Kontravarianz

$T_3 \leq T_1 \wedge T_2 \leq T_4 \Rightarrow (T_1 \rightarrow T_2) \leq (T_3 \rightarrow T_4)$

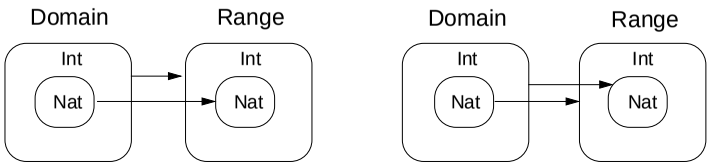
+ (Int → Nat) ≤ + (Nat → Int)

(steuern(eink: Int) → [0..5000]) ≤ (steuern(eink:[0..10000]) →Int)

Beispiel:

pers: Person; pens: Pensionist;
pers := pens;
print(pens.steuern(5 000 000));
print(pers.steuern(10 000));

Kovarianz versus Kontravarianz (1)



Kovarianzregel:

Nat -> Nat ≤ Int -> Int

Kontravarianzregel:

Int -> Nat ≤ Nat -> Int

Systeme:

Eiffel (Parameter), Sather, O_2

Trellis/Owl,Ada9X

Eiffel (Assertions)

Nonvarianz

$T_1 = T_3 \wedge T_2 \leq T_4 \Rightarrow (T_1 \rightarrow T_2) \leq (T_3 \rightarrow T_4)$

Beispiele:

+ (Int → Nat) ≤ + (Int → Int)

(steuern(eink: [Int]) → [0..5000]) ≤ (steuern(eink:Int) →Int)

Nur Ergebnis wird eingeschränkt!

Beispiel: C++

Strenge Nonvarianz:

$T_1 = T_3 \wedge T_2 = T_4 \Rightarrow (T_1 \rightarrow T_2) \leq (T_3 \rightarrow T_4)$

Alles bleibt gleich!

Beispiel: Simula

Regeln zur Subtypenbildung (4)

Operationen (Forts.):

➥ *semantische Prüfung*

mit Vor- und Nachbedingungen sowie invarianten Bedingungen für Operationen

Invariante Bedingungen werden pro Klasse/Typ definiert (d.h., sie sind für alle Operationen einer(s) Klasse/Typs gleich) und müssen immer erfüllt sein.

Regeln zur Subtypenbildung (5)

- *Kovarianzregel:*

$op_1 \leq op_2 \Leftrightarrow$

Signatur $(op_1) \leq_{kovariant}$ Signatur$(op_2) \wedge$
Vorbedingung$(op_1) \Rightarrow$ Vorbedingung$(op_2) \wedge$
Nachbedingung$(op_1) \Rightarrow$ Nachbedingung$(op_2) \wedge$
Invar.Bedingung$(op_1) \Rightarrow$ Invar.Bedingung(op_2)

- *Kontravarianzregel:*

$op_1 \leq op_2 \Leftrightarrow$

Signatur $(op_1) \leq_{kontravariant}$ Signatur$(op_2) \wedge$
Vorbedingung$(op_2) \Rightarrow$ Vorbedingung$(op_1) \wedge$
Nachbedingung$(op_1) \Rightarrow$ Nachbedingung$(op_2) \wedge$
Invar.Bedingung$(op_1) \Rightarrow$ Invar.Bedingung(op_2)

Beispiel (Kovarianz)

Stack mit zusätzlicher Zählvariable (*counter*):

StatisticStack ≤ Stack

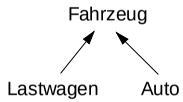
StatisticStack.pop ≤ Stack.pop
StatisticStack.push ≤ Stack.push

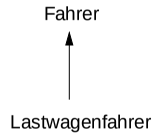
Kovarianz:
Vorbed.(Stack.pop) = not(self.empty)
Vorbed.(StatisticStack.pop) = not(self.empty) and counter > 0.

Nachbed.(Stack.push) = not(self.empty)
Nachbed.(StatisticStack.push) = not(self.empty) and counter > 0.

Beispiel (2)

- Kovarianzproblematik tritt häufig in der Praxis auf
- Beispiel:
  2 parallele Vererbungshierarchien
- Operation:
  Fahrzeug.registriere(f: Fahrer)





Lastwagen.registriere(f: Lastwagenfahrer)

Kovarianz versus Kontravarianz (2)

Kovarianz	*Kontravarianz*
+ intuitiv und unterstützt daher Realweltmodellierung	-
+ vereinfachtes und intuitives Überschreiben von Op.	-
+ intuitives Überschreiben der Vor-/Nach-/invar. Bedingungen	-
- statische Typüberprüfung nicht vollkommen	+

Lösungsvorschlag:
(1) Explizite Typumwandlung der Argumenttypen (widerspricht dem Geist des Polymorphismus)
(2) Multidispatch der Methoden, d.h., auch Argumenttypen beim Methoden-Lookup beachten

Regeln zur Subtypenbildung (6)

5. *Objekttypen:*

$OT_1 \leq OT_2 \Leftrightarrow$?

Verschiedene Interpretationen möglich**:**

1. basierend auf Kovarianz oder Kontravarianz
2. basierend auf Spezifikation ohne / mit Impl.

Subtypbeziehung ist ausschließlich über die Schnittstelle der Objekttypen (= Spezifikation der Operationen) definiert

ermöglicht unterschiedliche Implementierung und interne Struktur von Typen und Subtypen
z.B.: **Dequeue** ≤ **Stack**, aber **Dequeue** ist als variables Array und **Stack** als doppelt verkettete Liste implementiert.

Subtypbeziehung ist über die Schnittstelle und die interne Struktur (Typen der Instanzvariable) der Objekttypen definiert

Die 4 wünschenswerten Eigenschaften

1. Typersetzbarkeit
2. Statische Typüberprüfung
3. Veränderbarkeit
4. Subtypenbildung durch Spezialisierung

Aber: nicht alle 4 sind gleichzeitig erfüllbar

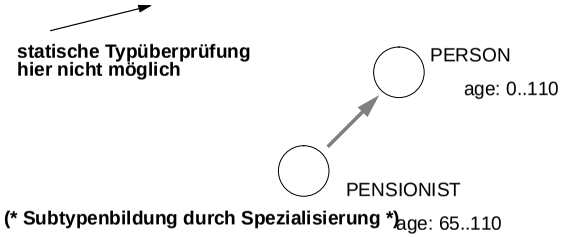
Die 4 unverträglichen Eigenschaften

p: PERSON;

r: PENSIONIST;

p :=r ; **(* Typersetzbarkeit *)**

p.alter_eintragen(30) **(* Veränderbarkeit *)**;



Andere Sicht der Unverträglichkeit

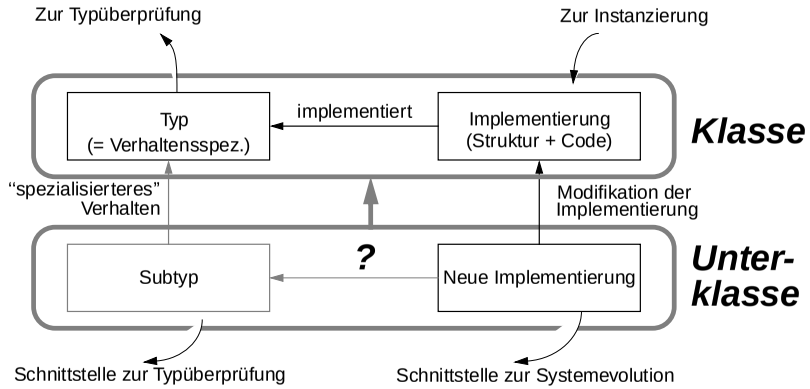
Typ PERSON	Typ PENSIONIST ≤ PERSON
alter: A;	alter: A’;
read_alter():A;	read_alter():A’;
write_alter(A);	write_alter(A’);

Statische Typüberprüfung (Kontravarianz):

Erweiterung des Inputs von write_alter ⇒ A ≤ A’
Einschränkung des Outputs von read_alter ⇒ A’ ≤ A

⇒ A=A’ (keine Einschränkung möglich)

Typ ≠ Klasse



Diese unterschiedlichen Intentionen machen es problematisch, Subtypen mit Unterklassen gleichzusetzen. Bei der Codevererbung will man/frau maximale Modifizierbarkeit gegebener Klassen. Bei der Subtypenbildung soll der Subtyp substituierbar für den Supertyp sein, was der Modifikationsmöglichkeit starke Einschränkungen auferlegt.

Typ und Klasse

- in typisierten Sprachen mit nur einer Vererbungshierarchie sind Subtypbeziehungen meist als Teilmenge der Vererbungsbeziehungen def.. Beispiel: C++, Eiffel
- Typanforderungen können übergangen werden: Nichtvererben von Nachrichten, explizite Typumwandlung
- Ausweichen durch zweite Art des Polymorphismus: Parametrisierte Klassen (s.u.)
- Entsprechung vonTypspezifikation in schwach getypten Sprachen: *Protokolle*

  Ein Protokoll ist eine Menge von zusammengehörigen Operationen, die auf einem Objekt ausgeführt werden können

Protokolle (z.B. Smalltalk)

- In Sprachen ohne typisierte Variablen können Subtypbeziehungen unabhängig von der Klassenhierarchie auftreten.
- *Protokoll* : Eine Menge von Messages mit bestimmter Funktionalität

  Beispiel 1: Das Protokoll von **OrderedCollection** ist eine Obermenge des Protokolls von **Array** ⇒ "Subtypbeziehung" unabh. v. Vererbung
- *Protokoll* einer Klasse: Messages dieser Klasse.

Protokolle (2)

- Manchmal Unterstützung bestimmter Teilprotokolle

  Beispiel 2: Sowohl **Array** als auch **Dictionary** unterstützen das Protokoll für Zugriff über Schlüsselwerte (#at:, #at:put:). Trotzdem weder Subklasse noch *global gesehen* Subtyp (Array wächst nicht, Dictionary nicht geordnet)
- d.h., es gibt ein Protokoll (“Zugriff über Schlüssel”), das die Messages #at:, #at:put: umfaßt, aber nicht einer Klasse fix zugeordnet ist

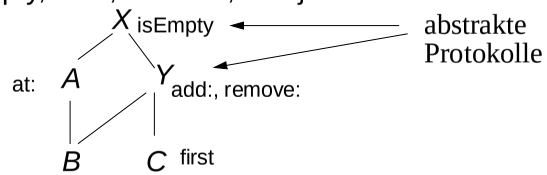
  ⇒ detaillierte Protokoll-Analyse zum Auffinden einer expliziten **Protokoll**- oder **Interface-Hierarchie** [Cook92]

Protokoll-Analyse

- Automatische Feststellung der Protokoll-Hierarchie
- Klassen mit überschneidenden Protokollen: Existenz von “abstrakten” Protokollen

Beispiel:

A = { isEmpty, at: },
B = { isEmpty, at:, add:, remove: },
C = { isEmpty, add:, remove:, first }.



Parametrisierte Klassen (1)

Eine **parametrisierte Klasse** ist eine Klasse mit mindestens einem formalen Klassenparameter.

Beispiel (in Eiffel; ähnlich *class templates* in C++):

```
class Stack[T];
...;
class IntStack[Integer] inherits Stack[T};
```

Die Klasse `Stack` ist parametrisiert mit ihrer Elementklasse. In der Subklasse `IntStack` wird der Parameter an eine aktuelle Klasse (`Integer`) gebunden. Durch die Parametrisierung werden eine Menge unterschiedlicher STACK-Klassen erzeugt, die alle durch dieselbe Impl. realisiert sind.

Smalltalk: Unnötiges Konzept, da keine Einschränkung auf fixen Elementtyp

Parametrisierte Klasse (2)

Abhängig vom Klassenparameter unterscheidet man:

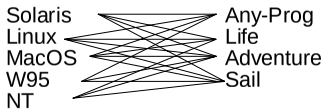
- uneingeschränkte Parametrisierung:
  z.B.: `class Stack[T];` für `T` kann jede Objektklasse als aktueller Parameter verwendet werden
- eingeschränkte Parametrisierung:
- z.B.: `class Stack[T ≤ GeometrischesObjekt];` für `T` kann jede Objektklasse als aktueller Parameter verwendet werden, die zumindest das Verhalten von `GeometrischesObjekt` erfüllt, d.h., ein Subtyp von `GeometrischesObjekt` ist.

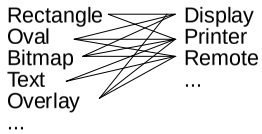
Vorteil: Lokalisierung der Spezifikation von Verhalten!

Multipolymorphismus

- Normale Annahme: ein Parameter bestimmt bei Dynamischem Binden die Auswahl der Implementierung
- Manchmal hängt die Implementierung einer Funktion von mehreren Parametern ab.

Beispiele:





Lösung (1)

```
<Rectangle> displayOn: aPort
    aPort isMemberOf: DisplayPort
        ifTrue: [ “code for Displaying on DisplayPort” ].
    aPort isMemberOf: PrinterPort
        ifTrue: [ “code for Displaying on PrinterPort”].
    ...
```

- Akzeptable Lösung für Einführung neuer graphischer Objekte, aber keine Unterstützung für neue Ports
- Alternative: Dynamisches Binden über mehr als 1 Parameter (**Multimethoden**, **Generic Functions**). Beispiel: CLOS, Dylan

```
(defmethod display ((go graphicalObject) (p port))
    body )
```

Lösung (2)

- Multimethoden können über beliebig viele Parameter dispatchen (in diesem Beispiel nur 2)

```
(defmethod display((go graphicalObject) (p port)) body )
(defmethod display((go Rectangle) (p displayPort)) body )
(defmethod display((go Oval) (p displayPort)) body )
(defmethod display((go Bitmap) (p displayPort)) body )
...
(defmethod display((go Rectangle) (p printer)) body )
(defmethod display((go Rectangle) (p remote)) body )
```

- Saubere Zerlegung auf Schnittstellenebene
- von Sprache und Typsystem unterstützt

Multimethoden (2)

- Problem1: Klasse als Organisationskriterium verliert an Bedeutung
- Problem 2: Eindeutige Auswahl ist nicht gewährleistet

```
(defmethod display((go graphicalObject) (p port)) body )
(defmethod display((go Rectangle) (p port)) body )
(defmethod display((go graphicalObject) (p printer)) body )
```

Aufruf: `(display aRectangle aPrinter)`

Ausweg: Festlegung von Prioritäten, z.B.:

- In der Vererbungshierarchie näherliegende Methoden
- zuerst gelistete Parameter zählen mehr
- Angabe von Methoden für Prioritätenberechnung

Ausweg ohne Multimethoden (1)

- Doppeltes Dispatching

<Rectangle> displayOn: aPort
    aPort displayRectangle: self

<Oval> displayOn: aPort
    aPort displayOval: self
... analoge Definition für jede Klasse

<DisplayPort> displayRectangle: aRect
    “Code für Display...”

<DisplayPort> displayOval: anOval
    “Code für Display...”

...

<Printer> displayRectangle: aRect
    “Code für Display...”

...

Ausweg ohne Multimethoden (2)

- Verwendbar für beliebige Tiefe (d.h., Dispatching über beliebig viele Argumente)
- Beibehaltung des OO Stils - Definitionen auf Schnittstellenebene, leichte Erweiterbarkeit (nur eine Seite ist massiv betroffen)
- Lösung nicht eindeutig: Dispatching könnte auch über Ports gehen, d.h.,

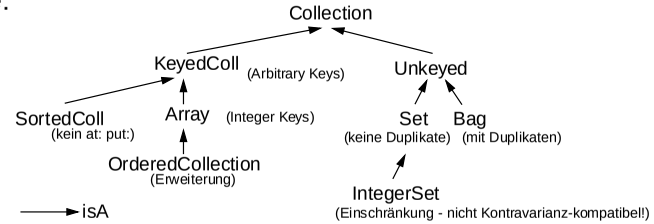
<Printer> display: aGraphicalObject
    aGraphicalObject displayOnPrinter: self

...

Spezialisationsvererbung (1)

modelliert klassische monotone *isA-Beziehung* der semantischen Datenmodellierung und Wissensrepräsentation.
Logische (intuitive) Beziehung zw. Unter- und Oberklassen.

Beispiele:
Car is*A* Vehicle, Dog *isA* Mammal *isA* Animal,
Beispiel für einen Ausschnitt aus der Smalltalk-Klassenhierarchie:



Spezialisation (3)

Vergleiche Smalltalk-Klassenhierarchie

- Physische Sicht (unterstützt von der Sprache)
  ➥ Implementierungshierarchie, für den Implementierer
- Logische Sicht:
  ➥ isA-Hierarchie, für den Benutzer

Beide Hierarchien sind sinnvoll.

Realisierung separater Hierarchien bisher nur in experimentellen Systemen

(Anm: “Hierarchie” wird hier als Synonym für “Vererbungsgraph” gebraucht - diese Diskussion gilt genauso für Systeme mit Multiple Inheritance.)

Spezialisation (4)

Die *isA-Beziehung* wird speziell
in der *Daten- und Objektmodellierung* zur Kategorisierung von Instanzen und zur Untermengenbildung verwendet.

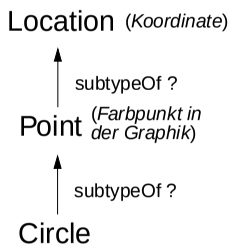
Zitat aus Wegner/Zdonik, S.65:

> “It is the subset relation rather than the subtype relation that corresponds to classical isA relation.”

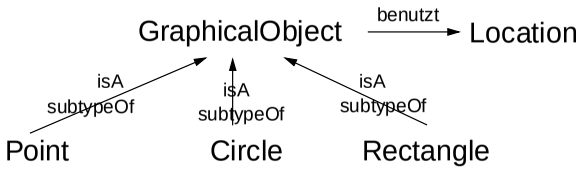
Allg.: isA ≠ subclass ≠ subtype

isA = subtype ? Manchmal !

Schlechter Entwurf:



Besserer Entwurf:



Entwurfsrichtlinien für Vererbung

- Vermeide Vererbung ausschließlich für Wiederverwendung von Code und interner Repräsentation (besser ist Konstruktion). Diese Form der Vererbung unterstützt **nicht** Wartbarkeit und modulare Wiederverwendung
- Überprüfe, ob eine isA-Beziehung Sinn hat
- Vermeide die Vererbung von Instanzvar. und Op., die in der Unterklasse nicht benötigt werden (lose Kopplung)
- Versuche, sinnvolle Abstraktionen zu finden, und diese als abstrakte Oberklassen zu implementieren (z.B.: *GraphicalObject* ist abstrakte Oberklasse von allen graphischen Objekten)

Granularität

Muß Vererbung immer zwischen Klassen stattfinden?

- Vererbung auf Klassenebene
- Vererbung auf Objektebene

Vererbung auf Klassenebene

- bisher betrachtete Systeme sind klassen/typ-basiert.
- Vererbung ist für eine *Menge von Objekten* definiert, nicht für ein einzelnes Objekt.
- Instanzen erfragen das Verhalten bei der instanziierenden Klasse
- interne Struktur in der Klassendefinition festgelegt, wird beim Erzeugen der Instanz angelegt

Delegation (1)

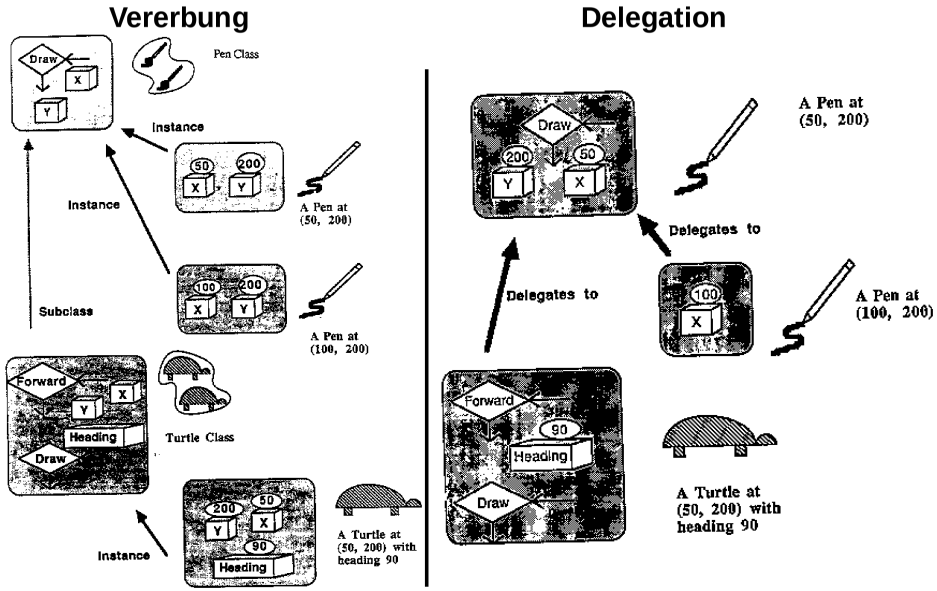
Vererbung auf Instanzenebene.

Zu jedem Objekt (*extension object*) existiert ein **prototypisches Objekt** (*Prototype*, *prototypical object*), von dem das Verhalten und die interne Struktur *incl. der Werte* geerbt wird. (H. Liebermann, in OOPSLA'86)

Vererbung durch **Delegation** (*delegation*):

- eine *Nachricht* wird entweder *selbst beantwortet* (Operation ist beim Objekt selbst definiert), oder an das prototypische Objekt *delegiert*.
- Neue Nachrichten werden an das ursprüngliche Empfängerobjekt gesendet (analog *self* in klassenb. Sys.).

Vererbung gegen Delegation (1)



Vererbung und Delegation (2)

- Vererbung wird zur Übersetzungszeit / Klassengenerierungszeit festgelegt
- Delegation wird zur Laufzeit festgelegt: entweder bei Objektgenerierung oder beim Nachrichtenempfang

Welches Konzept ist besser?

➥ Abhängig von der Anwendung!

1. Vererbung für Produktionssysteme, laufzeiteffizient, stabil, Typüberprüfung möglich
2. Delegation für experimentelle Systeme, Prototypen, dynamisch, flexibel (Sprache SELF - Stanford, Sun)

Delegation (2)

Auch als *dynamische Vererbung* (da Änderung der Vererbungsbeziehungen zur Laufzeit elegant darstellbar und leicht zu implementieren ist) oder *Objektvererbung* (Gegensatz zu Klassenvererbung) bezeichnet

Ziel:

Klassenvererbung (incl. Typüberprüfungsmöglichkeit) und eine Form der Delegation (flexible Objektmodellierung) im selben System unterstützen.

Mögliche Lösung:

Typspezialisierung ***und*** *Objektspezialisierung* im selben System unterstützen.

Typspezialisierung

auf Typebene

1. Vererbung von Strukturdefinitionen
2. Vererbung von Operationen
3. exklusive Klassifikation

Nachteile:

1. Objektevolution schwierig (tritt auf, wenn Objekte ihren Typ ändern, z.B.: Angestellter wird Pensionist)
2. Mehrfachvererbung notwendig, um exklusive Klassifikation zu umgehen (aufwendig! Bei *n* Oberklassen 2^n mögliche Unterklassen)

Objektspezialisierung (1)

auf Instanzebene

1. Vererbung von Strukturdefinition und -ausprägungen
2. Vererbung von Operationen
3. dynamische Klassifikation, inklusiv

Vorteile:

1. Objektevolution leicht handhabbar
2. weniger Typen notwendig

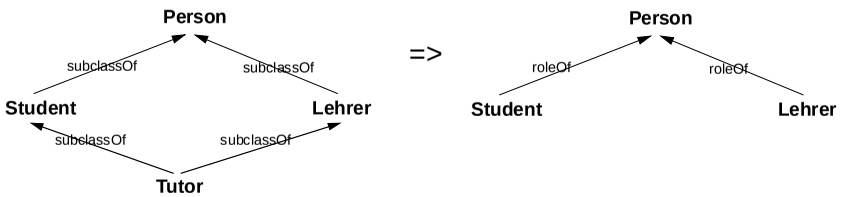
Nachteile:

1. Laufzeit?
2. Übersicht?

Objektspezialisierung (2)

Mögliche Realisierung:

durch spezielle *roleOf*-Beziehung. Ein Realweltobjekt ist durch Instanzen verschiedener Typen (Merke: mit verschiedenen Oid's!) im System repräsentiert, die durch eine *roleOf*-Beziehung verbunden sind und eine *Rollenhierarchie* bilden. Ein Rollenobjekt ist von dem durch die *roleOf*-Beziehung assoziierten Objekt **existenzabhängig**.



Objektspezialisierung (3)

Achtung: Rollen und Mixins nicht verwechseln! Mixins sind Konstrukte auf Klassenebene, ohne Laufzeitdynamik!

Vorteilhafte Eigenschaften der *Modellierung mit Rollen* sind:

- *multiple-type membership*: Realweltobjekt ist gleichzeitig durch Instanzen verschiedener Typen im System repräsentiert.
- *multiple instantiation*: Objekt kann gleichzeitig mehrmals Instanz desselben Typs sein (z.B.: Student an verschiedenen Universitäten)
- *object evolution*: Der Typ eines Objekts kann sich ändern.

Vorteile der Vererbung (1)

1. Wiederverwendung:
   Instanzvariable und Operationen einer Klasse in Unterklassen verwendbar
2. Konsistente Schnittstellen/Spezifikation:
   dasselbe Verhalten hat denselben Namen
3. Modellierungshilfe:
   Kategorisierung/Spezialisierung als zentrale Modellierungstätigkeit

- Abwägen ist wichtig

Vorteile der Vererbung (2)

4. Softwarekomponenten
   standardisierte Bibliotheken
   - aber mit Varianten
   - mit Erweiterungsmöglichkeit
5. Inkrementelle Entwicklung:
   (vom Allgemeinen zum Speziellen oder umgekehrt)
6. Rapid Prototyping:
   rasche Entwicklung durch Weiterentwicklung existierender Klassen(hierarchien)

Nachteile der Vererbung (1)

1. Dynamisches Binden etwas langsamer
   - Aber: Polymorphismus ersetzt Case-Anweisungen
   - Und: Polymorphismus ohne Vererbung bedeutet redundanten Code
2. Größeres Programm
   - Behauptung: Code wird vererbt (und ist Teil des Programms), wird aber nicht immer benötigt
   - Grundsätzl. Problem der Wiederverwendung - Frage der Tools, nicht des Ansatzes (auch bei Libraries)
3. Problematik der Klassenverwaltung bei Teamarbeit und in verteilten Systemen

Referenzen

1. P. Wegner, S. Zdonik: ***Inheritance as an Incremental Modification Mechanism***, Proc. ECOOP'88, Springer LNCS, 1988.
2. W.LaLonde, J. Pugh: ***Subclassing*** ≠ ***Subtyping*** ≠ ***IsA***, Journal of OOP (JOOP), Jan.1991.
3. L.A. Stein, H. Liebermann, D. Ungar: ***A Shared View of Sharing: The Treaty of Orlando***, Object-Oriented Concepts, Databases, and Applications, W. Kim & L.H. Lochovsky (Hrsg.), ACM Press, 1989.
4. J. Micallef: ***Encapsulation, Reusability and Extensibility in OOPLs***, Journal of OOP, April/Mai 1988.
5. S. Khoshafian, R. Abnous: ***Object-Orientation: Concepts, Languages, Databases, User Interfaces*** (- Kapitel über Vererbung), Wiley, 1990.
6. A. Snyder: ***Inheritance and the Dev. of Encapsulated SW Comp.,*** Research Directions in ooP, Shriver & Wegner (Hrsg.), MIT Press, 1987.

Referenzen

7. S. Keene, ***Object-Oriented Programming in Common Lisp - a Programmer's Guide to CLOS***, Addison-Wesley, 1989.
8. D. Ingalls: ***A Simple Technique for Handling Multiple Polymorphism***, Proc. OOPSLA '86, ACM Press.
9. W. Cook: ***Interfaces and Specifications for the Smalltalk-80 Collection Classes***, Proc. OOPSLA '92, ACM Press.