Das Ersetzbarkeitsprinzip

> U ist Untertyp von T, wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird

Dieses Ersetzbarkeitsprinzip benötigt man für

- den Aufruf einer Routine mit einem Argument, dessen Typ Untertyp des formalen Parametertyps ist;
- die Zuweisung eines Objekts an eine Variable, wobei der Objekttyp Untertyp des deklarierten Variablentyps ist.

Untertypen und Schnittstellen

Objekttyp U ist Untertyp von Objekttyp T wenn

- für jede Konstante in T (vom Typ A) existiert Konstante in U (vom Typ B), wobei B Untertyp von A
- für jede Variable in T existiert eine in U vom selben Typ
- für jede Methode in T existiert Methode in U, wobei
  - Parameteranzahlen und Parameterarten gleich
  - Eingangsparametertypen in T Untertypen der in U
  - entsprechende Durchgangsparametertypen gleich
  - Ausgangsparametertypen in U Untertypen der in T
  - Ergebnistyp in U Untertyp von Ergebnistyp in T

Ko- und Kontravarianz

Kovarianz: Typ eines Elements im Untertyp ist Untertyp des Elementtyps im Obertyp

— Konstantentypen, Ergebnistypen, Ausgangsparametertypen

Kontravarianz: Typ eines Elements im Untertyp ist Obertyp des Elementtyps im Obertyp

— Eingangsparametertypen

Invarianz: Typ eines Elements im Untertyp ist gleich dem Elementtyp im Obertyp

— Variablentypen, Durchgangsparametertypen

Beispiel für Ko- und Kontravarianz

- Annahme: B ist Untertyp von A

```
class T {
    public A meth (B par) { ... }
}
class U extends T {   // U ist Untertyp von T
    public B meth (A par) { ... }
}
```

- entspricht Bedingungen für Untertypbeziehungen

- Achtung: Kein Überschreiben in Java !!!

Wann und warum Kovarianz?

- Ersetzbarkeit nur bei lesendem Zugriff auf Konstante, Ergebnis oder Ausgangsparameter (bei Routinenaufruf)
- nur Elementtyp A im Obertyp T statisch bekannt
- Lesezugriff kann tatsächlich auf entsprechendes Element vom Typ B im Untertyp U erfolgen
- gelesener Wert soll trotzdem vom erwarteten Typ A sein

  $\Rightarrow$ jede Instanz von B soll auch Instanz von A sein

  $\Rightarrow$ B soll Untertyp von A sein
- Initialisierung Konstante, Ergebnis, Ausgangsparameter (in Routine): genauer Typ bekannt — keine Ersetzbarkeit

Wann und warum Kontravarianz?

- Ersetzbarkeit nur bei Schreibzugriff auf Eingangsparameter (bei Routinenaufruf)

- nur Parametertyp B im Obertyp T statisch bekannt

- Schreibzugriff kann tatsächlich auf entsprechenden Parameter vom Typ A im Untertyp U erfolgen

- tatsächlich geschriebener Wert soll vom Typ A sein, obwohl Werte vom Typ B geschrieben werden können

  $\Rightarrow$ jede Instanz von B soll auch Instanz von A sein

  $\Rightarrow$ B soll Untertyp von A sein

- Lesezugriff auf Eingangsparameter (in Routine): deklarierter Typ bekannt

Wann und warum Invarianz?

- Ersetzbarkeit sowohl bei schreibendem als auch lesendem Zugriff benötigt (Variable, Durchgangsparameter)
- daher sowohl Kovarianz als auch Kontravarianz benötigt
- nur Invarianz erfüllt beide Forderungen

Einschränkungen in der Praxis

- Theorie vollständig und widerspruchsfrei
  keine expliziten Untertypdeklarationen nötig

- Einschränkung in vielen praktisch verwendeten Sprachen:
  - explizite Vererbungsbeziehung vorausgesetzt
  - Vererbung entsprechend eingeschränkt

  Grund: einfach, keine zufälligen Untertypbeziehungen

- weitere Einschränkung in Java: Ergebnistypen kovariant, alle anderen Typen invariant

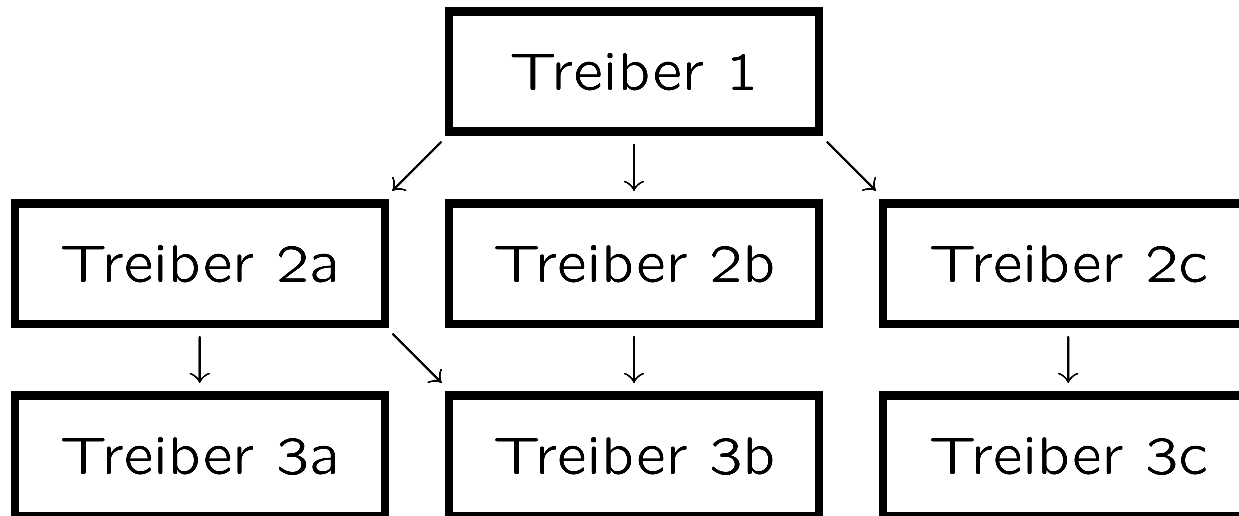
  Grund: intuitiv, unterscheidbar von Überladen

Grenzen von Untertypbeziehungen

```
class Point2D {
    protected int x, y;
    public boolean equal (Point2D p) {
        return x == p.x && y == p.y;
    }
}
class Point3D extends Point2D {  // Achtung: FALSCH
    protected int z;
    public boolean equal (Point3D p) {
        return x == p.x && y == p.y && z == p.z;
    }
}
```

Untertypen — Codewiederverwendung

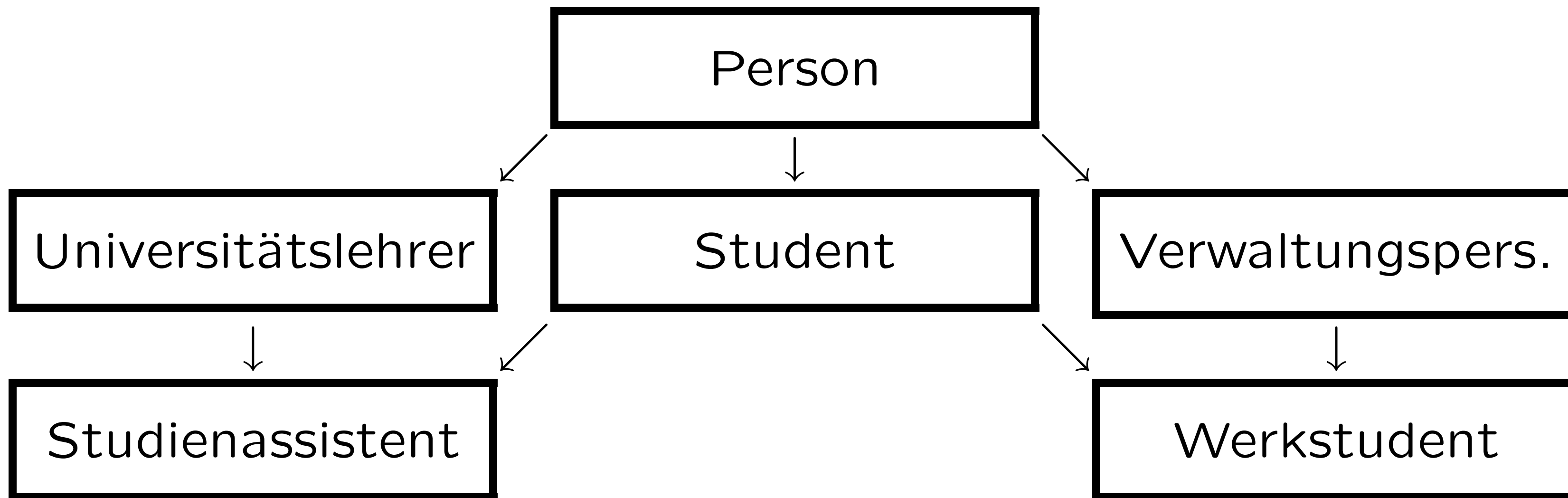
Software-Generationen und -Versionen



Typen sollen unverändert bleiben, Erweiterungen möglich

Untertypen — Codewiederverwendung

Wiederverwendung innerhalb eines Programms



Typen sollen stabil sein — vor allem weiter oben

Dynamisches Binden (1)

```
class A {
    public String foo1() { return "foo1A"; }
    public String foo2() { return fooX(); }
    protected String fooX() { return "foo2A"; } }
class B extends A {
    public String foo1() { return "foo1B"; }
    protected String fooX() { return "foo2B"; } }
class DynamicBindingTest {
    public static void test(A x) {
        System.out.println(x.foo1());
        System.out.println(x.foo2()); }
    public static void main(String[] args) {
        test(new A); test(new B); }
}
```

Dynamisches Binden (2)

Aufruf:

```
java DynamicBindingTest
```

Bildschirmausgabe:

```
foo1A
foo2A
foo1B
foo2B
```

Beispiel mit Switch

```
public void gibAnredeAus (int anredeArt, String name) {
    String str;
    switch(anredeArt) {
        case 1:     // weiblich
            str = "S.g. Frau "; break;
        case 2:     // maennlich
            str = "S.g. Herr "; break;
        default:    // unbekannt
            str = "S.g. ";
    }
    System.out.print (str + name);
}
```

Beispiel ohne Switch

```
class Adressat {
    protected String name;
    public void gibAnredeAus() {
        System.out.print("S.g. " + name);
    } ... }
class WeiblicherAdressat extends Adressat {
    public void gibAnredeAus() {
        System.out.print ("S.g.Frau " + name);
    } }
class MaennlicherAdressat extends Adressat {
    public void gibAnredeAus() {
        System.out.print ("S.g. Herr " + name);
    } }
```