Tagesprogramm

Grundlagen der Ersetzbarkeit

Untertypbeziehungen und Wiederverwendung

Dynamisches Binden

Ersetzbarkeitsprinzip

U ist Untertyp von T wenn
Instanz von U überall verwendbar wo Instanz von T erwartet

benötigt für

Argument dessen Typ Untertyp des formalen Parametertyps

Zuweisung `x = y;` wobei Typ von `x` Untertyp des deklarierten Typs von `y`

Untertypen und Schnittstellen

Typ U ist Untertyp von Typ T wenn

$\forall$ Konstante in T (Typ A) $\exists$ Konstante in U (Typ B): B Untertyp von A

$\forall$ Variable in T (Typ A) $\exists$ Variable in U (Typ B): A und B äquivalent

$\forall$ Methode in T $\exists$ Methode in U:

- Parameteranzahlen und Parameterarten gleich
- Parametertypen passen zueinander
- Ergebnistyp in U Untertyp von Ergebnistyp in T
- Methode in U wirft nicht mehr Exceptions als die in T

Untertypen nach Parameterarten

Methodenparameter in Obertyp T vom deklarierten Typ A und
Methodenparameter in Untertyp U vom deklarierten Typ B

Eingangsparameter:

Argument wandert von Aufrufer zu aufgerufener Methode
$\rightarrow$ A Untertyp von B

Ausgangsparameter:

Ergebnis wandert von aufgerufener Methode zu Aufrufer
$\rightarrow$ B Untertyp von A

Durchgangsparameter:

gleichzeitig Ein- und Ausgangsparameter
$\rightarrow$ A und B äquivalent

Varianz von Typen

Kovarianz:

Typ von Element im Untertyp ist Untertyp des Elementtyps im Obertyp

→ Typ von Konstante, Ergebnis, Ausgangsparameter

Kontravarianz:

Typ von Element im Untertyp ist Obertyp des Elementtyps im Obertyp

→ Typ von Eingangsparameter

Invarianz:

Typ von Element im Untertyp ist äquivalent zu Elementtyp im Obertyp

→ Typ von Variable, Durchgangsparameter

Beispiel für Varianz

Annahme: variante Parametertypen erlaubt (nicht in Java)

```
class T {
    public T meth(U p) { ... }
}
class U extends T {   // U ist Untertyp von T
    public U meth(T p) { ... }
}   // in Java ueberladen, nicht ueberschrieben
```

entspricht Bedingungen für Untertypbeziehungen

ACHTUNG: funktioniert in Java nicht so

Wann und warum Kovarianz?

Ersetzbarkeit bei **Lesezugriff** auf Konstante, Ergebnis, Ausgangsparameter

nur Elementtyp A im Obertyp T statisch bekannt

Lesezugriff kann tatsächlich auf Element vom Typ B in U erfolgen

gelesener Wert soll vom erwarteten Typ A sein

→ Instanz von B auch Instanz von A

→ B Untertyp von A

bei Schreiben von Konstante, Ergebnis, Ausgangspar. genauer Typ bekannt

→ dabei keine Ersetzbarkeit nötig

Wann und warum Kontravarianz?

Ersetzbarkeit bei **Schreibzugriff** auf Eingangsparameter

nur Parametertyp A im Obertyp T statisch bekannt

Schreibzugriff kann tatsächlich auf Parameter vom Typ B in U erfolgen

geschriebener Wert vom Typ B obwohl Werte vom Typ A schreibbar

→ Instanz von A auch Instanz von B

→ A Untertyp von B

bei Lesezugriff auf Eingangsparameter deklarierter Typ bekannt

→ dabei keine Ersetzbarkeit nötig

Wann und warum Invarianz?

Ersetzbarkeit bei **schreibendem und lesendem** Zugriff

sowohl Kovarianz als auch Kontravarianz gefordert

Aufgabe: Explizite Untertypbeziehungen

Regeln für Untertypbeziehungen sind durch die Theorie vollständig und unumstößlich vorgegeben. Sie sind nicht verhandelbar.

Suchen Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Fragen:

1. Kann man Untertypbeziehungen aus den vorgegebenen Regeln ableiten und dadurch auf das explizite Anschreiben von Untertypbeziehungen verzichten?

   Wann ja, nennen Sie eine solche objektorientierte Programmiersprache.

2. Warum ist es in Sprachen wie Java nötig, Untertypbeziehungen explizit in den Programmcode zu schreiben?

Zeit: 3 Minuten

Theorie und Praxis

Theorie vollständig und widerspruchsfrei auf Signaturen

→ für **strukturelle Typen** keine Untertypdeklarationen nötig

in der Praxis: **nominale Typen**

→ explizite Vererbungsbeziehung vorausgesetzt,
→ Vererbung eingeschränkt,
→ „zufällige“ Untertypbeziehungen verhindert

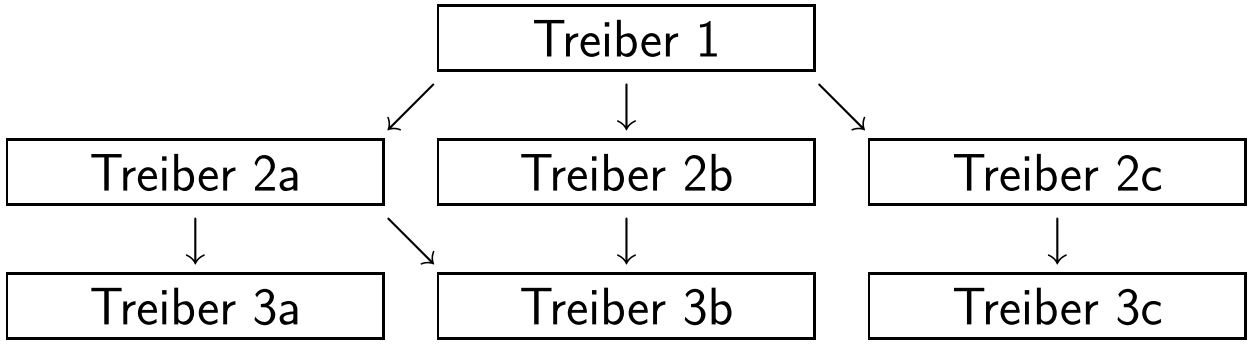
weitere **Einschränkung** in Java und ähnlichen Sprachen:

Ergebnistypen kovariant, alle anderen Typen invariant
da intuitiv und unterscheidbar von Überladen

Grenzen von Untertypbeziehungen

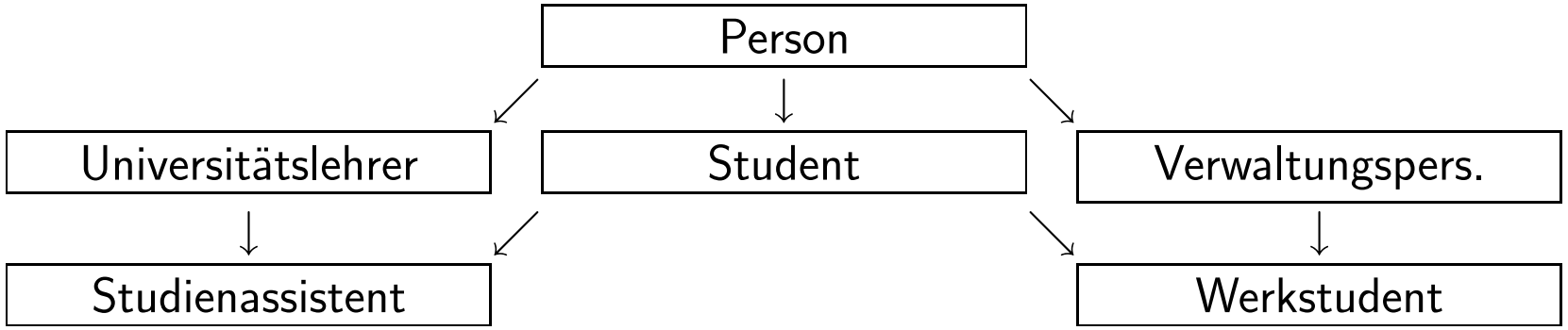
```
public class Point2D {
    protected int x, y;
    public boolean equal(Point2D p) {
        return x == p.x && y == p.y;
    }
}

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

Code-Wiederverwendung über Generationen



Typen sollen **unverändert** bleiben, Erweiterungen möglich

Code-Wiederverwendung im Programm



Typen sollen **stabil** sein – vor allem weit oben in Typhierarchie

Abstr. Beispiel für dynamisches Binden

```
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()); }
}
```

Beispiel mit Switch

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

Beispiel ohne Switch

```
public class Adressat {
    protected String name;
    public void gibAnredeAus()
        { System.out.print("S.g. " + name); }
    ... // Konstruktoren und weitere Methoden
}

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

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