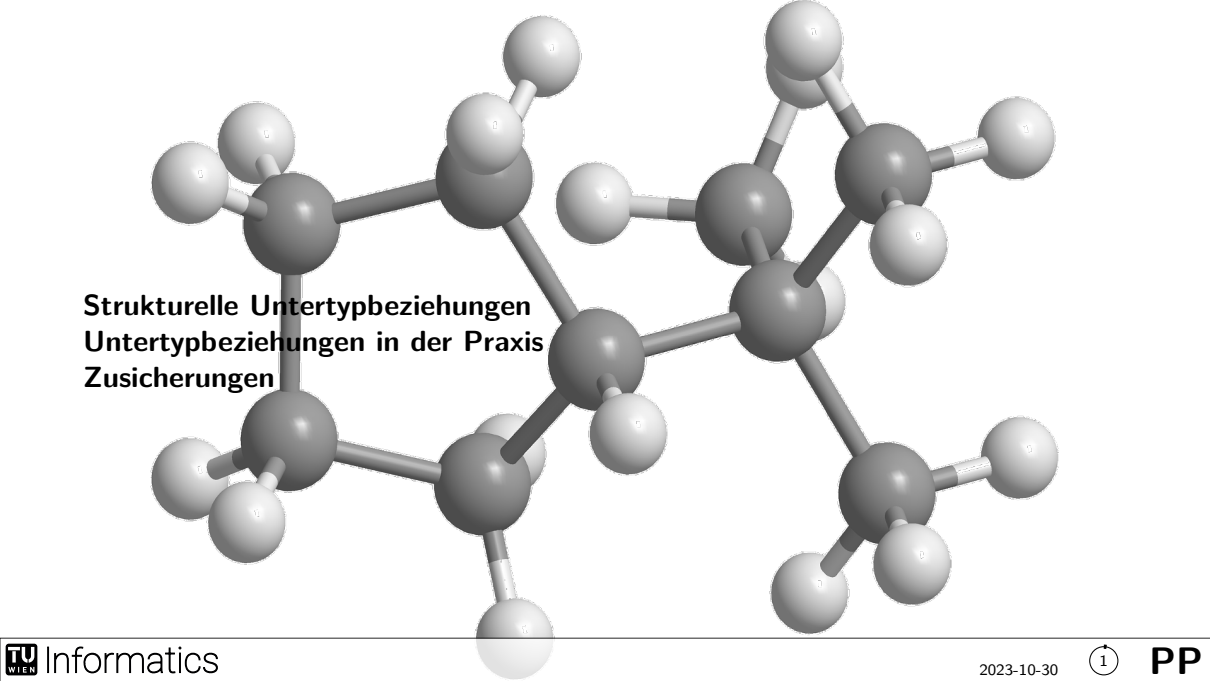


Strukturelle Untertypbeziehungen
Untertypbeziehungen in der Praxis
Zusicherungen

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 (strukturell)

Typ U ist Untertyp von Typ T wenn

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

A verhält sich zu B wie T zu U

$\forall$ 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

Ergebnistyp in U verhält sich zu Ergebnistyp in T wie T zu U

Methode in U löst nicht mehr Ausnahmen aus 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
→ A Untertyp von B — A verhält sich zu B umgekehrt wie T zu U

Ausgangsparameter:

Ergebnis wandert von aufgerufener Methode zu Aufrufer
→ B Untertyp von A — A verhält sich zu B wie T zu U

Durchgangsparameter:

gleichzeitig Ein- und Ausgangsparameter
→ A und B äquivalent

Varianz von Typen

Kovarianz: A verhält sich zu B wie T zu U

Typ von Element im Untertyp ist Untertyp des Elementtyps im Obertyp
→ Typ von Konstante, Ergebnis, Ausgangsparameter

Kontravarianz: A verhält sich zu B umgekehrt wie T zu U

Typ von Element im Untertyp ist Obertyp des Elementtyps im Obertyp
→ Typ von Eingangsparameter

Invarianz: gleichzeitig Ko- und Kontravarianz

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

$\rightarrow$ Instanz von A auch Instanz von B
$\rightarrow$ A Untertyp von B

bei Lesezugriff auf Eingangsparameter deklarierter Typ bekannt

$\rightarrow$ dabei keine Ersetzbarkeit nötig

Wann und warum Invarianz?

Ersetzbarkeit bei *schreibendem und lesendem* Zugriff

sowohl Kovarianz als auch Kontravarianz gefordert

Theorie und Praxis

Theorie vollständig und widerspruchsfrei auf Signaturen

→ für *strukturelle Typen* keine Untertypdeklarationen nötig

in der Praxis: objektorientierte Programmierung verlangt *nominale Typen*

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

weitere *Einschränkung in Java* und ähnlichen Sprachen:

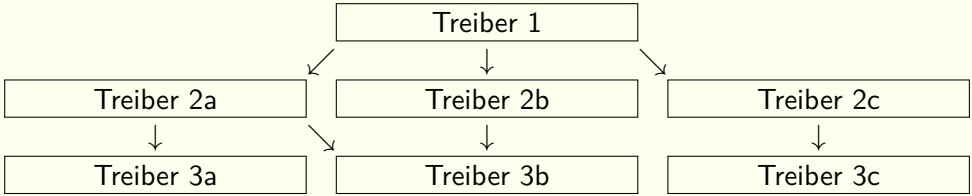
Ergebnistypen kovariant, alle anderen Typen invariant
da intuitiv und unterscheidbar von Überladen

Grenzen von Untertypbeziehungen

kovariantes Problem

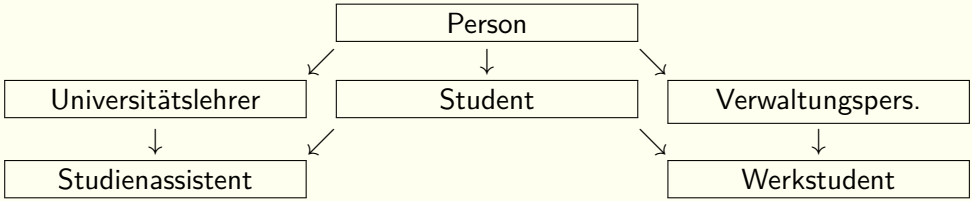
```
public class Point2D {
    protected int x, y;
    public boolean equal(Point2D p) {
        return x == p.x && y == p.y;
    }
}

public class Point3D extends Point2D {   // FALSCH
    protected int z;
    public boolean equal(Point3D p) {    // wäre kovariant
        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);}
}
```

Ersetzbarkeit und Verhalten

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

Struktur der Typen für Ersetzbarkeit nicht ausreichend

Beispiel:
```
void draw()  // zeichne Bild
void draw()  // ziehe Revolver
```

Ersetzbarkeit muss Verhalten berücksichtigen $\rightarrow$ *Design by Contract*

Client-Server-Beziehungen

Server bietet Dienste an, Client nutzt Dienste
wobei Dienst = Ausführung einer Methode

Objekt ist gleichzeitig Client und Server

Vertrag (Contract) zwischen Client und Server:

Client erfüllt *Vorbedingungen* eines Dienstes
Server erfüllt *Nachbedingungen* eines Dienstes
Server erfüllt *Invarianten* des Objekts
Client + *Server* erfüllen *History-Constraints* des Objekts

Vorbedingung (Precondition)

Verantwortlich: Client

Wann: vor Methodenaufruf

Was: hauptsächlich Eigenschaften von Argumenten
Beispiel: Argument ist Array aufsteigend sortierter Zahlen

manchmal auch (sichtbarer) Zustand des Servers
Beispiel: `abheben` nicht aufrufen wenn Konto überzogen

Nachbedingung (Postcondition)

Verantwortlich: Server

Wann: vor Rückkehr aus Methodenaufruf

Was: Eigenschaften von Methodenergebnissen sowie
Änderungen und Eigenschaften des Objektzustands

Beispiel: „Methode fügt Element (falls noch nicht vorhanden) in Menge ein. Ergebnis ist 'true' falls Element bereits vorher in Menge war."

Nachbedingung klingt oft wie Methodenbeschreibung

Invariante

Verantwortlich: Server

Wann: vor und nach Ausführung von Methoden

Was: unveränderliche Eigenschaften von Objekten und Variablen
Beispiel: Guthaben am Sparbuch ist immer positive Zahl

Gültigkeit der Invariante kann von Bedingungen abhängen
Beispiel: „'zuverlaessig == false' wenn Konto überzogen“

impliziert Nachbedingung

Ausnahme: wenn Variable von außen geschrieben werden kann
dann Client *und* Server für Invariante auf Variable verantwortlich

Server-kontrollierter History-Constraint

Verantwortlich: Server

Wann: nach Ausführung von Methoden
im Bezug auf Zustand vor Ausführung der Methoden

Was: Einschränkungen auf Veränderungen von Variablen
Beispiel: „Zählerwert erhöht sich mit jedem Methodenaufruf“

Prüfung vor Methodenausführung von Natur aus unmöglich

impliziert Nachbedingung

Ausnahme: wenn Variable von außen geschrieben werden kann
dann Client *und* Server gemeinsam verantwortlich

Client-kontrollierter History-Constraint

Verantwortlich: Client

Wann: vor Methodenaufrufen

Was: Einschränkungen auf der Reihenfolge von Aufrufen
Beispiele: zuerst `initialize`, dann andere Methoden aufrufbar;
nach jedem Aufruf von a ein Aufruf von b nötig

Server kennt Aufrufreihenfolge oft nicht, Clients schon

ein Client oder *alle* Clients bestimmen Reihenfolge

Zusammenhang mit Synchronisation (Koordination)

Beispiel zu klassischen Zusicherungen

```
public class Konto {
    public int guthaben;
    public int ueberziehungsrahmen;
    // guthaben >= -ueberziehungsrahmen

    // einzahlen addiert summe zu guthaben; summe >= 0
    public void einzahlen(int summe)
        { guthaben = guthaben + summe; }

    // abheben zieht summe von guthaben ab;
    // summe >= 0; guthaben+ueberziehungsrahmen >= summe
    public void abheben(int summe)
        { guthaben = guthaben - summe; }
}
```

Beispiel zu History-Constraints

```
public class Transaction {
    private int started = 0, committed = 0, aborted = 0;
    // number of corresponding transactions
    // can only increase one by one    (server-controlled)
    // started >= committed + aborted          (invariant)

    // for each invocation of start() returning x, either
    // commit(x) or abort(x) must be invoked exactly once
    public int start() { ...; return started++; }
    public void commit(int x) { ...; committed++; }
    public void abort(int x)  { ...; aborted++; }
}
```

Typen und Zusicherungen

Zusicherungen gehören zu nominalen Typen

Objekttyp besteht aus	Schnittstelle (= Signatur) Name von Klasse oder Interface Zusicherungen

Änderung von Zusicherung = Typänderung
$\rightarrow$ Auswirkungen auf andere Programmteile

Genauigkeit von Zusicherungen

Genauigkeit durch Programmierer(inn)en bestimmbar:

- genau: große Abhängigkeit zwischen Client und Server — schlecht
- ungenau: kleine Abhängigkeit zwischen Client und Server — gut

Tipp: keine versteckten Zusicherungen

Tipp: schwache Abhängigkeiten (= wenige Zusicherungen)

Ersetzbarkeit und Verhalten (klassisch)

U ist nur dann Untertyp von T wenn gilt:

Vorbedingungen in Untertypen sind *schwächer oder gleich*

bei Vererbung: Verknüpfung mit *oder*

Nachbedingungen in Untertypen sind *stärker oder gleich*

bei Vererbung: Verknüpfung mit *und*

Invarianten in Untertypen sind *stärker oder gleich*

aber wenn Variable von außen schreibbar, dann Invarianten *gleich*

bei Vererbung: Verknüpfung mit *und* (wenn nicht gleich)

Ersetzbarkeit und History-Constraints

U ist nur dann Untertyp von T wenn gilt:

Wenn *Server-kontrollierte History-Constraints* in T verhindern,
dass eine Variable vom Zustand X in den Zustand Y kommt,
dann müssen dies auch Constraints in U tun.

> U kann Zustandsänderungen stärker einschränken als T
> aber wenn Variable von außen schreibbar, dann *gleiche* Constraints

Jede von *Client-kontrollierten History-Constraints* in T erlaubte Aufrufreihenfolge muss auch in U erlaubt sein.

> TraceSet(T) $\subseteq$ TraceSet(U)
> Menge aller Clients betrachten, nicht nur einen einzelnen Client

Zusicherungen und Ersetzbarkeit – Beispiel

```
public class Set {
    public void insert(int x)
        // inserts x into set iff not already there
        // x is in set immediately after invocation
        { ... }
    public boolean inSet(int x)
        // returns true if x is in set, otherwise false
        { ... }
    ...
}

public class SetWithoutDelete extends Set {
    // elements in the set always remain in the set
}
```

Beispiel fortgesetzt

```
Set s = ...;
s.insert(42);
doSomething(s);
if (s.inSet(42)) { doSomeOtherThing(s); }
else { doSomethingElse(); }
```

```
SetWithoutDelete s = ...;
s.insert(42);
doSomething(s);
doSomeOtherThing(s); // s.inSet(42) always returns true
```

Faustregeln zu Zusicherungen

Zusicherungen sollen

- stabil sein (vor allem an Wurzel der Typhierarchie)
- keine unnötigen Details festlegen
- explizit im Programm stehen
- unmissverständlich formuliert sein
- während Programmentwicklung ständig überprüft werden

Kommentare sollen ...