Ersetzbarkeit und Verhalten

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

- Struktur der Typen für Ersetzbarkeit nicht ausreichend

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

  $\neq$ `void draw() // ziehe Revolver`

- Ersetzbarkeit muss Verhalten berücksichtigen

  $\rightarrow$ *Design by Contract*

Client-Server-Beziehungen

- Server bietet Dienste an, Client nutzt Dienste
- Objekt ist gleichzeitig Client und Server
- Vertrag zwischen Client und Server:
  - Client erfüllt *Vorbedingungen* eines Dienstes
  - Server erfüllt *Nachbedingungen* eines Dienstes
  - Server sichert *Invarianten* des Objekts zu
- Dienst entspricht der Ausführung einer Methode
- Vor-, Nachbedingungen, Invarianten sind *Zusicherungen*

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

Beispiel: Guthaben am Sparbuch ist immer positive Zahl

Gültigkeit der Invariante kann von Bedingungen abhängen

Beispiel: „'zuverlaessig == false' wenn Konto überzogen“

Invarianten implizieren Nachbedingungen

Ausnahme: Schreiben externer Variablen

— Client und Server verantwortlich

Beispiel zu Zusicherungen

```
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;
    }
}
```

Typen und Zusicherungen

- Zusicherungen gehören zu Typen
- Objekttyp besteht aus
  - Name von Klasse / Interface
  - Schnittstelle (= Signatur)
  - Zusicherungen
- Zusicherungen informal als Kommentare (in Java)
- Überprüfung der Zusicherungen durch Programmierer
- Änderung von Zusicherung = Typänderung
  (Auswirkungen auf andere Programmteile)

Genauigkeit von Zusicherungen

- Clients dürfen sich nur auf das verlassen, was in Schnittstellen und Zusicherungen vom Server zugesagt wird.
- Server dürfen sich nur auf das verlassen, was in Schnittstellen und Zusicherungen vom Client zugesagt wird.
- Genauigkeit durch ProgrammiererInnen bestimmbar:
  - genau: große Abhängigkeit zw. Client und Server
  - ungenau: kleine Abhängigkeit zw. Client und Server
- Tipp: keine versteckten Zusicherungen !!
- Tipp: schwache Abhängigkeiten (= wenige Zusicherungen)

Ersetzbarkeit und Verhalten

S ist nur dann Untertyp von T wenn gilt:

- Vorbedingungen in T implizieren Vorbedingungen in S
  - Vorbedingungen in Untertypen sind schwächer / gleich
  - bei Vererbung: Verknüpfung mit ODER
- Nachbedingungen in S implizieren Nachbedingungen in T
  - Nachbedingungen in Untertypen sind stärker / gleich
  - bei Vererbung: Verknüpfung mit UND
- Invarianten in S implizieren Invarianten in T
  - Invarianten in Untertypen sind stärker / gleich
  - bei Vererbung: Verknüpfung mit UND
  - extern geschriebene Variablen: Invarianten unverändert

Zusicherungen und Ersetzbarkeit (1)

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

Zusicherungen und Ersetzbarkeit (2)

```
Set s = ...;
s.insert(41);
doSomething(s);
if (s.inSet(41))
   { doSomeOtherThing(s); }
else
   { doSomethingElse(); }


SetWithoutDelete s = ...;
s.insert(41);
doSomething(s);
doSomeOtherThing(s); // s.inSet(41) always returns true
```

Zusicherungen und Ersetzbarkeit (3)

Tipp: nicht `SetWithoutDelete` verwenden, wo `Set` ausreicht

(um andere künftige Erweiterungen zu ermöglichen)

```
class SetWithDelete extends Set {
    public void delete (int x) {
        // deletes x from the set if it is there
        ...;
    }
}
```

Beispiele für Zusicherungen: Java API Dokumentation

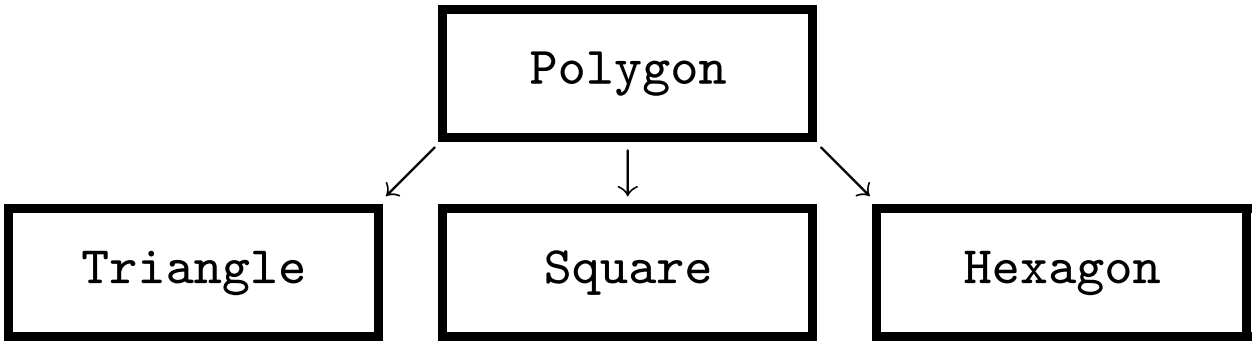
(siehe z.B. `http://download.oracle.com/javase/6/docs/api/`)

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

Abstrakte Klassen




```
abstract class Polygon {
    public abstract void draw(); // draw polygon on screen
}

class Triangle extends Polygon {
    public void draw() { // draw a triangle on the screen
        ...;
    }
}
```

Verwendung abstrakter Klassen

- Implementierungen oft in Klassen ohne Unterklassen
- abstrakte Klassen (und Interfaces) eher stabil
- Parametertypen sollen stabil sein

  $\Rightarrow$ sie sollen nicht mehr ausdrücken, als nötig
  (keine unnötigen Abhängigkeiten)
- abstrakte Klassen leicht hinzuzufügen
  (z.B. verwendbar als Parametertypen für jeden Bedarf)

Interfaces

```
interface X {
    static final double PI = 3.14159d;
    double fooX();
}

interface Y {
    double fooY();
}

interface Z extends X, Y {
    double fooZ();
}
```

Klassen implementieren Interfaces

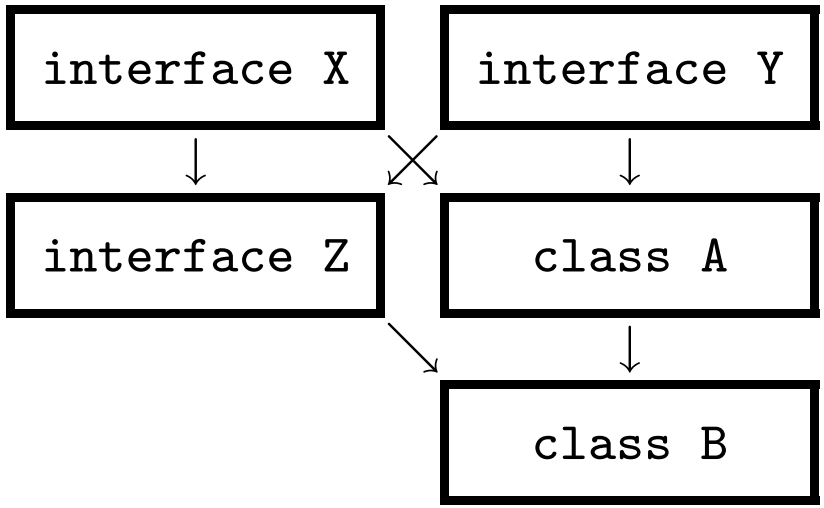
```
class A implements X, Y {
    public double foo() { return PI; }
    public double fooX() { return factor * PI; }
    public double fooY() { return factor * fooX(); }
    protected double factor = 2.0d;
}

class B extends A implements Z {
    public double fooY() { return 3.3d * foo(); }
    public double fooZ() { return factor / fooX(); }
}
```

Unterschiede zu abstrakten Klassen

- Schlüsselwort `interface` statt `abstract class`
- alles implizit public — daher `public` nicht nötig
- alle Methoden abstrakt — daher `abstract` nicht nötig
- keine Variablen, nur Konstanten (`static final ...`)
- `static` und `final` in Methodendeklarationen verboten
- nach `extends` können mehrere Namen von Interfaces stehen — Mehrfachvererbung

Interfaces und Untertyprelationen

Interfaces für Untertyprelationen gut geeignet



Abstrakte Klassen gegenüber Interfaces nur dann zu bevorzugen, wenn Code vererbt werden soll

Zusicherungen auf abstrakten Methoden besonders wichtig!