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 erfüllt *Invarianten* des Objekts
  - Client + Server erfüllt *History-Constraints* des Objekts
- Dienst = Ausführung einer Methode

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/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: Variable kann von außen geschrieben werden

→ Client UND Server für Invariante darauf verantwortlich

Server-kontrollierter History-Constraint

Verantwortlich: Server

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

Was: Einschränkungen auf der Veränderung von Variablen
Beispiel: Zählerwert erhöht sich mit jedem Methodenaufruf

Prüfung vor Methodenausführung von Natur aus unmöglich

impliziert Nachbedingung

Ausnahme: Variable kann von außen geschrieben werden
$\rightarrow$ 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
nach jedem Aufruf von A einer von B nötig

Server kennt Reihenfolge 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 / Interface
  - Zusicherungen
- in Java Zusicherungen nur informal als Kommentare
- Überprüfung nur durch Programmierer
- trotzdem: Änderung von Zusicherung = Typänderung
  $\rightarrow$ Auswirkungen auf andere Programmteile

Genauigkeit von Zusicherungen

- Client verlässt sich nur darauf, was vom Server zugesagt
- Server verlässt sich nur darauf, was vom Client zugesagt
- Genauigkeit durch Programmierer(inn)en 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 (1)

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
  - bei Vererbung: Verknüpfung mit UND
  - von außen schreibbare Variable $\rightarrow$ gleiche Invarianten

Ersetzbarkeit und Verhalten (2)

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
  - von außen schreibbare Variable $\rightarrow$ 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 einzelnen Client

Zusicherungen und Ersetzbarkeit (1)

```
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
}
```

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)

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

Beispiele für Zusicherungen: Java API Dokumentation

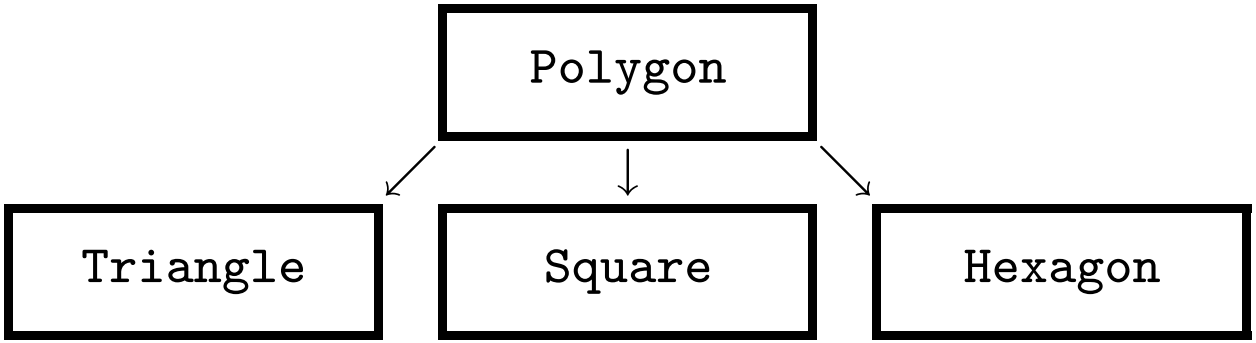
(siehe z.B. `http://download.oracle.com/javase/7/docs/api/`)

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

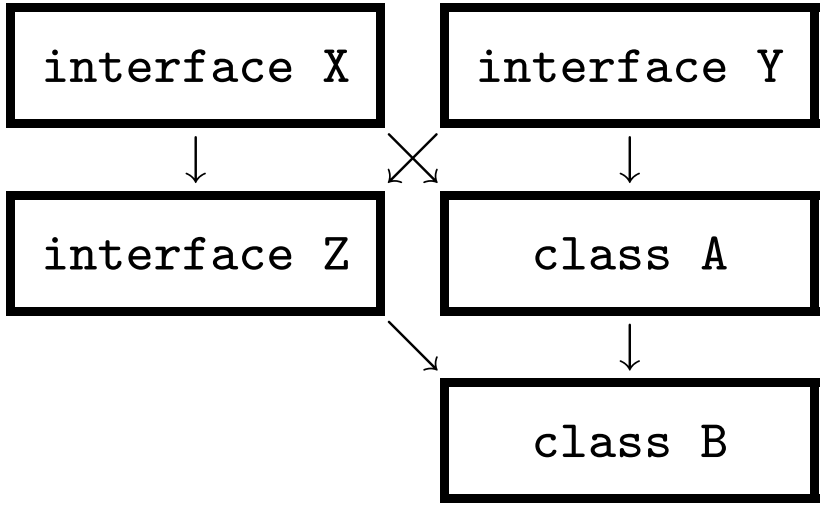



```
public abstract class Polygon {
    public abstract void draw(); // draw polygon on screen
}
public class Triangle extends Polygon {
    public void draw() { ... }  // draw triangle on screen
}
```

Zusicherungen auf abstrakten Methoden besonders wichtig!

Interfaces

Interfaces für Untertyprelationen sehr gut geeignet



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

Zusicherungen auf Interfaces besonders wichtig!

Abstr. Klassen u. Interfaces verwenden

- Implementierungen oft nur 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 bzw. Interfaces leicht hinzuzufügen
  (z.B. verwendbar als Parametertypen für jeden Bedarf)
- Zusicherungen auf abstrakten Einheiten besonders wichtig