

# **Fortgeschrittene objektorientierte Programmierung**

(185.211, VL 2.0)

Franz Puntigam  
Institut für Computersprachen

`franz@complang.tuwien.ac.at`

`http://www.complang.tuwien.ac.at/franz/foop.html`

Freitag, 12. 3. – 7. 5. 2010, 10<sup>00</sup> bis 12<sup>00</sup> Uhr, EI 3A

---

# Ersetzbarkeit

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

Ersetzbarkeit gegeben wenn

- Eingangstypen kontravariant
- Variablen-, Durchgangstypen invariant
- Konstanten-, Ergebnis-, Ausgangstypen kovariant
- Methoden in U verhalten sich wie die in T

---

# Ersetzbarkeit und Verhalten

Methoden in U verhalten sich wie die in T wenn

- von Client zu erfüllende Zusicherungen (Vorbedingungen) in U nicht stärker als jene in T
- von Server zu erfüllende Zusicherungen (Nachbedingungen, Invarianten) in U nicht schwächer als jene in T
- von Client und Server beeinflusste Zusicherungen (alle Arten von Zusicherungen, nur manchmal bei Invarianten akzeptabel) in U und T äquivalent
- Methoden in U werfen nicht mehr Exceptions als die in T

---

# Ausdrückbarkeit von Zusicherungen

- theoretisch alle Zusicherungen formal ausdrückbar (Logik, Algebra), Beziehungen zwischen ihnen oft statisch prüfbar
- praktisch vieles nicht ausdrückbar und Beziehungen nicht statisch prüfbar da
  - mangelhafte Sprachunterstützung (Kommentare informal → in der Regel mehrdeutig)
  - Client fehlt nötige Information über Objektzustand (Info. des Servers wegen data hiding nicht zugreifbar)
  - sich Objektzustand auf unvorhersehbare Weise ändern kann (Nebenläufigkeit, Aliasing-Probleme)

---

# Zusicherungen: Beispiel (1)

```
class IntSet {
    public boolean find (int x) { ... }
    // true wenn x in Menge, sonst false
    public void insert (int x) { ... }
    // x unmittelbar nach Einfuegen in Menge
    ...
}
...
IntSet set = new IntSet();
set.insert(1);      // jetzt ist 1 sicher in set
boolean a = set.find(1);
do_something_not_using_set();
boolean b = set.find(1);
```

---

## Zusicherungen: Beispiel (2)

- „unmittelbar nach“ nicht eindeutig
- häufige Interpretation: solange kein „delete“ (möglicherweise in Untertyp definiert) auf Menge ausgeführt, bleibt Element in Menge
- obwohl „set“ direkt initialisiert könnte Konstruktor Alias auf Menge eingeführt haben und „do something not using set“ Menge ändern → „b“ vielleicht „false“
- nebenläufiger Thread könnte Element gleich nach „insert“ löschen → „a“ und „b“ vielleicht „false“

---

## Zusicherungen: Beispiel (3)

- möglicher Ausweg: mit „find“ vergewissern, dass Element in Menge (Vorbedingung oder normaler Code)
  - was machen wenn Überprüfung fehlschlägt?
  - Alias-Probleme ausschließen (aufwendig)
  - bei Nebenläufigkeit atomare Aktion (aufwendig)
- ⇒ unerwartete Seiteneffekte vermeiden, z.B. in Konstruktor  
(= einzig wirklich praktikabler Weg)

---

# Zusicherungen: History Constraints

- Clients haben oft mehr Information über Objektzustand als durch Abfragen des Zustands feststellbar (data hiding)
- wichtiges Beispiel: *history properties*  
(wann wird was aufgerufen – Reihenfolge oft nicht beliebig)
- *history constraints* legen Einschränkungen auf history fest  
Bsp.: „unlock“ aufrufbar, wenn vorher „lock“ aufgerufen
- history constraints ähneln Invarianten, aber Clients sind eher für Einhaltung verantwortlich (Aufrufreihenfolge)
- history constraints in Untertypen können mehr Aufrufreihenfolgen erlauben als in Obertypen



---

# Zusicherungen: Fazit

- es gibt keine einfache Lösung, da vollständiges Ausschließen aller Fehlermöglichkeiten viel zu aufwendig wäre
- einschätzbar programmieren, Tricks vermeiden, und darauf verlassen, dass Programmierer sich „normal“ verhalten
- design rules (oft spezifisch für Firma oder Projekt)
- gesamte verfügbare Information nutzen (history)
- oberstes Gebot: !! unnötige Abhängigkeiten vermeiden !!  
Z.B.: keine vermeidbaren Zusicherungen, keine unnötigen Aufrufe und Parameter, keine unnötig sichtbaren Variablen und Methoden, wohlüberlegte Parametertypen, Code nur tief in Klassenhierarchie (= Vererbung vermeiden)

---

# Bedeutung von Namen

- Namen abstrahieren Verhalten von Klassen bzw. Methoden und Eigenschaften von Variablen
- darin ähneln Namen (informalen) Zusicherungen
- Intuition kommt hauptsächlich von Namen; Kommentare nur nötig, wenn Intuition nicht ausreicht → gute Programme auch ohne Kommentare lesbar
- gut gewählte Namen machen Programmierer einschätzbar (schlechte Namen → kein Vertrauen → unnötiger Code)
- Benennung von Typen hat semantische Bedeutung (nicht nur für Programmierer, sondern auch für Compiler)

---

# Strukturelle – nominale Typen: Beispiel

zwei *strukturelle* (= anonyme) Typen in Untertyprelation:

```
{ String name; String adresse() }  
{ String name; String adresse(); int matrnr }
```

Namen von Members zur Benennung von Typen:

```
{ String name; String adresse(); void isPerson() }  
{ String name; String adresse(); void isPerson();  
  int matrnr; void isStudent() }
```

in der Theorie hauptsächlich strukturelle Typen untersucht,  
aber praktisch meist *nominale* (= benannte) Typen verwendet

nominale Typen durch strukturelle Typen simulierbar, aber  
zufällige Übereinstimmungen möglich, nicht „fälschungssicher“

---

# Strukturelle und nominale Typen

	strukturell	nominal
Typäquivalenz	gleiche Struktur	gleicher Name
Subtyping	implizit	explizit (Vererbung)
Verwendung	einfach	komplizierter
Plug & Play	besser	schlechter
Namenskonflikte	möglich	leichter möglich
zufällige Beziehungen	möglich	unwahrscheinlich
Lesbarkeit	schlechter	besser
Verhaltensabstraktion	nein	ja

---

---

# Arten von Namenskonflikten

- unterschiedliche Dinge mit gleichem Namen
  - beide nicht dort definiert, wo Namen aufeinander treffen
    - je nach Sprache lokal umbenennen oder qualifizieren
  - mindestens ein Ding lokal definiert
    - umbenennen, möglicherweise globale Auswirkungen
- gleiche Dinge mit unterschiedlichen Namen (Strukturen)
  - beide Namen nicht dort definiert, wo Gleichheit nötig
    - größere Umstrukturierungen nötig (bzw. Wrapper)
  - mindestens ein Name lokal definiert
    - lokale Definition entfernen

---

# Compiler berücksichtigen Typnamen

```
class SortedList<A extends Comparable<A>>
```

- Eigenschaft „sortiert“ kaum direkt prüfbar, sondern nur durch Klassenzugehörigkeit
- jede Instanz von `SortedList<T>` durch einen Konstruktor in `SortedList` (oder Unterklasse) erzeugt – kein Schummeln bei benannten Typen
- wenn `SortedList` und Unterklassen sicherstellen, dass Inhalt jeder Instanz sortiert, dann darf man sich darauf verlassen

---

# Final Klassen

```
class UnsortedList<A extends Comparable<A>>  
    extends SortedList<A>
```

- Schummeln durch Erzeugen von Unterklassen (die Ersetzbarkeitsprinzip verletzen) doch möglich
- final Klassen machen Schummeln unmöglich
- final Klassen als Parametertypen führen oft zu unnötigen Abhängigkeiten  
→ vermeiden, außer wenn Schummeln sehr gefährlich
- Programmiersprache Sather zeigt, dass es sinnvoll sein kann, wenn alle nicht-abstrakten Klassen final sind

---

# Generizität mit anonymen Schranken

```
class SortedList<A extends Comparable<A>>
```

- sinnlos, dass A von Comparable erben muss  
ausreichend, wenn A Vergleichsoperation bereitstellt  
(Comparable impliziert keine weiteren Eigenschaften)
- Ada, C++ (implizit), ... erlauben anonyme Schranken

```
generic
```

```
  type T is private;
```

```
  with function compare(x,y: T) returns Boolean
```

```
package SortedList ...
```



---

## Wo Typnamen manchmal stören

- bei Schranken für gebundene Generizität
- beim Zusammenfügen oder Austauschen von Softwarekomponenten (Plug & Play)
- beim nachträglichen Einfügen eines gemeinsamen Ober-typs mehrerer (nicht änderbarer) bestehender Typen

---

# Arten gebundener Generizität

- F-gebundene Generizität:  $S \leq T\langle S \rangle$

Beispiel in Java: `Integer ≤ Comparable<Integer>`

kann mit binären Methoden umgehen

funktioniert nur eingeschränkt über mehrere Ebenen

- higher-order subtyping (matching):

Def.:  $S < \# T$  wenn  $S\langle U \rangle \leq T\langle U \rangle$  für alle  $U$

Matching unterstützt binäre Methoden direkt

als Schranke bei Generizität typsicher (C++ concepts)

allgemein muss  $U$  für statische Typsicherheit bekannt sein