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, 14^{00} bis 16^{00} Uhr, EI 3A

Ersetzbarkeit

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

Ersetzbarkeit gegeben wenn

- Eingangsparametertypen kontravariant
- Variablen-, Durchgangsparametertypen invariant
- Konstanten-, Ergebnis-, Ausgangsparam.typen kovariant
- U wirft nicht mehr Exceptions als T
- Methoden von U verhalten sich wie die von T

Ersetzbarkeit und Verhalten

Methoden von U verhalten sich wie die von 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

Ausdrückbarkeit von Zusicherungen

- theoretisch alle Zusicherungen formal ausdrückbar (Logik, Algebra), Beziehungen oft statisch prüfbar
- praktisch vieles nicht ausdrückbar und Beziehungen nicht statisch prüfbar da
  - mangelhafte Sprachunterstützung (Kommentare informal $\rightarrow$ in der Regel mehrdeutig)
  - Client fehlt nötige Information über Objektzustand
  - 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);
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 $\rightarrow$ „b“ vielleicht „false“
- nebenläufiger Thread könnte Element gleich nach „insert“ löschen $\rightarrow$ „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)

$\Rightarrow$ unerwartete Seiteneffekte vermeiden (z.B. in Konstruktor)

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 Untertyp kann mehr Aufrufreihenfolgen erlauben als in Obertyp

Zusicherungen: Fazit

- es gibt keine einfache Lösung, da vollständiges Ausschließen aller Fehlermöglichkeiten viel zu aufwendig
- 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 möglichst 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
  $\rightarrow$ gute Programme ohne Kommentare lesbar
- gut gewählte Namen machen Programmierer einschätzbar
- Benennung von Typen hat semantische Bedeutung (nicht nur für Programmierer sondern auch für Compiler)

Anonyme Typen: Beispiel

zwei 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() }
```

zufällige Übereinstimmungen möglich, nicht „fälschungssicher“

Anonyme und benannte Typen

	anonym	benannt
Typäquivalenz	gleiche Struktur	gleicher Name
Subtyping	implizit	explizit
Verwendung	einfach	komplizierter
Verhaltensabstraktion	nein	ja
Lesbarkeit	mittelmäßig	besser
Plug & Play	besser	problematisch
Namenskonflikte	optimistisch	pessimistisch

Arten von Namenskonflikten

- unterschiedliche Dinge mit gleichem Namen
  - beide nicht dort definiert, wo Namen aufeinander treffen
    $\rightarrow$ je nach Sprache lokal umbenennen oder qualifizieren
  - mindestens ein Ding lokal definiert
    $\rightarrow$ umbenennen, möglicherweise globale Auswirkungen
- gleiche Dinge mit unterschiedlichen Namen (Strukturen)
  - beide Namen nicht dort definiert, wo Gleichheit nötig
    $\rightarrow$ größere Umstrukturierungen nötig (bzw. Wrapper)
  - mindestens ein Name lokal definiert
    $\rightarrow$ 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
  $\rightarrow$ 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#, 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 Obertyps mehrerer (nicht änderbarer) bestehender Typen