Kapitel 7

Invarianten und sichere Vererbung

Idee der Vererbung: `class U extends O ...`

bedeutet: Jedes *U* ist auch als *O* verwendbar, denn es hat mindestens dieselben Members im *O*-Subobjekt (*Typkonformanz*, “IS-A”)

Erweiterung des Typkonformanz-Begriffs auf Methodensignaturen führt zur sog. Kontravarianz (s.d.)

Jedoch: *U* kann eine *O*-Methode so umdefinieren, dass sie etwas völlig anderes macht und nicht mehr zu den anderen *O*-Methoden paßt

⇒ stärkere Forderung: *Subtyping*. Klienten-Code funktioniert auch mit *U* statt *O* (“Klientencode-Wiederverwendung”, “Liskov’s Substitutionsprinzip”, “Inclusion Polymorphism” [Cardelli])

7.1 Subtyping/Verhaltenskonformanz

⇒ für Klienten sichere Vererbung erfordert *Verhaltenskonformanz*:

1. für jede Klasse wird eine *Invariante* formuliert (Prädikat über den Instanzvariablen), die vor und nach jedem Methodenaufruf gelten muß:

$$\{Inv(C)\}\ o.m(x)\ \{Inv(C)\}$$

2. Die U-Invariante muß die O-Invariante implizieren (i.e. stärker sein):

$$Inv(U) \Rightarrow Inv(O)$$

   Beispiel:

```
class Person{
   String name;
   int alter;
   // Inv(Person) == 0 <= alter <= 150
}
class Student extends Person {
   int MatrNr;
   // Inv(Student) == (0 <= alter <= 30)
   //                 AND ... MatrNr ...
}
```

3. Für jede Methode m in O gilt:

$$Pre(O.m) \Rightarrow Pre(U.m)\ \wedge\ Post(U.m) \Rightarrow Post(O.m)$$

„Die Oberklasse verlangt mehr (hat stärkere Preconditions-/kleineren Definitionsbereich der Methoden), die Unterklasse leistet mehr (hat stärkere Postconditions)“

Dies sind strenge Forderungen, die aber absolute Sicherheit für den Klienten garantieren: *auch aus Sicht des Methodenverhaltens ist jedes U-Objekt auch ein O-Objekt*; Klientencode kann also garantiert unverändert bleiben wenn O durch U ersetzt wird

Verhaltenskonformanz: U verlangt vom Klientencode weniger, aber bietet ihm mehr als O

Dies kann nur funktionieren, wenn in die Implementierung von U erhöhter Aufwand geht

Hauptanwendung: Unterklasse *implementiert* (abstrakte) Oberklasse

Beispiel: Interface `Set` mit `insert, lookup ...` Implementierung `SortedList`, die zusätzlich `get/size` hat

$$
\begin{aligned}
Inv(Set) &\equiv true \\
Inv(SortedList) &\equiv \forall i \in 0..size() - 2 : \\
&\quad\ get(i) \leq get(i+1) \\
Pre(Set.insert(x)) &\equiv true \\
Pre(SortedList.insert(x)) &\equiv true \\
Post(Set.insert(x)) &\equiv lookup(x) = true \\
Post(SortedList.insert(x)) &\equiv lookup(x) = true
\end{aligned}
$$

In diesem einfachen Fall von Verhaltenskonformanz sind Pre/-Postkonditionen gleich, aber die Invarianten verschieden.
Bsp. mit Klassen s.u.

7.2 Inheritance/Spezialisierung

In der Praxis ist Konformanz eher selten. Häufiger ist *Spezialisierung*:

$$Pre(U.m) \Rightarrow Pre(O.m) \ \wedge \ Post(O.m) \Rightarrow Post(U.m)$$

„U verlangt vom Klientencode mehr, aber bietet ihm (funktional) weniger“

Hauptanwendung: Postconditions gleich, U bietet spezielle (zB effizientere) Implementierung, die aber nicht für alle O-Fälle funktioniert

Beispiel: `Stack` mit `push, pop, top, isempty`;
Unterklasse `BoundedStack` mit zusätzlicher Methode `isfull`

$$Pre(\texttt{Stack s.push(x)}) \equiv \texttt{true}$$
$$Pre(\texttt{BoundedStack s.push(x)}) \equiv \text{NOT } \texttt{s.isfull()}$$
$$Post(\texttt{s.push(x)}) \equiv \texttt{s.top()} = \texttt{x}$$

Es gilt $Pre(\texttt{BoundedStack s.push(x)}) \Rightarrow Pre(\texttt{Stack s.push(x)})$

Gelegentlich gibt es noch (Verhaltens-)*Kovarianz*:

$$Pre(O.m) \Rightarrow Pre(U.m) \ \wedge \ Post(O.m) \Rightarrow Post(U.m)$$

und (Verhaltens-)*Kontravarianz*:

$$Pre(U.m) \Rightarrow Pre(O.m) \ \wedge \ Post(U.m) \Rightarrow Post(O.m)$$

7.3 Beispiel

Eine Klasse `Bank` erlaubt das Geld-Abheben von einem Konto mittels einer Methode `abheben`. Sie bietet einen Dispo-Kredit an, der als negativer Wert dargestellt wird - der Betrag, um den das Konto überzogen werden darf.

```
class Bank {
   int konto;
   int dispo;

   public Bank(int k, int d) {
      konto = k;
      dispo = d;
   }
   public void abheben(int betrag, Date datum) {
      konto -= betrag;
      ...
   }
}
```

Die Vorbedingung für `abheben` ist

$$Pre(\texttt{Bank b.abheben(int betrag, Date datum)}) \equiv$$
$$(\texttt{konto} - \texttt{betrag}) \geq \texttt{dispo} \wedge \texttt{dispo} \leq 0$$

die Nachbedingung ist

$$Post(\texttt{Bank b.abheben(int betrag, Date datum)}) \equiv$$
$$(\texttt{konto'} = \texttt{konto} - \texttt{betrag})$$

Eine Subklasse von Bank ist WeihnachtBank, die im Dezember den Dispokredit um 1000 Euro erhöht. Die Vorbedingung ist hier

Pre(WeihnachtBank k.abheben(**int** betrag,Date datum)) $\equiv$
(datum.month() $=$ Dezember) $\Rightarrow$
(konto $-$ betrag $\geq$ (dispo -1000) $\wedge$ dispo ≤ 0
(datum.month() $\neq$ Dezember) $\Rightarrow$
(konto $-$ betrag $\geq$ dispo $\wedge$ dispo ≤ 0

die Nachbedingung ist

$Post$(WeihnachtBank k.abheben(**int** betrag,Date datum)) $\equiv$
(konto' $=$ konto $-$ betrag)

```
class WeihnachtBank extends Bank {
   public WeihnachtBank(int k, int d) {
      konto = k;
      dispo = d;
   }
   public void abheben(int betrag, Date datum) {
      konto -= betrag;
      ...
   }
}
```

WeihnachtBank erfüllt die Verhaltenskonformanz:

- wenn Vorbedingung für abheben der Klasse Bank gilt, dann auch für abheben von WeihnachtBank (Übung: nachrechnen!)
- wenn Nachbedingung von abheben in WeihnachtBank gilt, dann auch für abheben in Bank

Eine andere Bank namens KeinDispoBank gewährt keinen Dispokredit. Die Vorbedingung ist hier

$$Pre(\texttt{KeinDispoBank r.abheben(int betrag,Date datum)}) \equiv (\texttt{konto} - \texttt{betrag} \geq 0)$$

die Nachbedingung ist

$$Post(\texttt{KeinDispoBank r.abheben(int betrag,Date datum)}) \equiv (\texttt{konto'} = \texttt{konto} - \texttt{betrag})$$

```
class KeinDispoBank extends Bank {
   public KeinDispoBank(int k) {
      konto = k;
      dispo = 0;
   }
   public void abheben(int betrag, Date datum) {
      konto -= betrag;
      ...
   }
}
```

Hier ist die Verhaltenskonformanz nicht erfüllt:

- Vorbedingung für abheben von KeinDispoBank folgt nicht aus Vorbedingung für abheben von Bank

KeinDispoBank ist also eine Spezialisierung von Bank.

Objekte von `KeinDispoBank` können daher nicht ohne weiteres als Objekte von `Bank` verwendet werden:

```
class Client {
   void abheben (Bank bank, int betrag, Date datum) {
      bank.abheben(1000, datum);
   }

   void main() {
      Bank b = new Bank(100, 1000);
      Bank w = new WeihnachtBank(100, 1000);
      Bank k = new KeinDispoBank(100);

      // Diese beiden Aufrufe werden
      // problemlos bearbeitet.
      abheben(b, 1000, 24.12.06);   // konto = -1000
      abheben(w, 1000, 24.12.06);   // konto = -1000

      // Nach diesem Aufruf hat der Benutzer
      // sein Konto ueberzogen!
      abheben(k, 1000, 24.12.06)    // konto = -1000
   }
}
```

7.4 Quadrat/Rechteck

Bekanntes Beispiel: Klassen Rechteck und Quadrat mit `x, y, setx(x), sety(y), area()`

In der Mathematik gilt: Ein Quadrat ist ein Rechteck (IS-A). Also `Square extends Rectangle`

Ist das verhaltenskonformant?

Inv(`Rectangle`) ≡ `true`, Inv(`Square`) ≡ (`x` = `y`).
Es gilt Inv(`Square`) ⇒ Inv(`Rectangle`)

- Jedoch Pre(`R.setx(x)`) ≡ `true`, Pre(`Q.setx(x)`) ≡ (`x` = `Q.gety()`), und erforderliche Implikation für Preconditions gilt nicht. Nimmt man für Unterklasse als `setx`-Precondition auch `true` (weil man sonst gar nicht Seitenlänge ändern kann), kann Invariante `Q.x` = `Q.y` zerstört werden und damit auch Verhaltenskonformanz.
- Die Alternative `Rectangle` **`extends`** `Square` ist nicht vermittelbar, da gegen 2500 Jahre mathematische Tradition, und außerdem auch nicht verhaltenskonformant, da die erforderliche Implikation der Invarianten nicht gilt:

Fügt man in `Rectangle` `y` neu hinzu und erbt `x` aus `Square`, dann ist für die Flächenberechnung wiederum die Verhaltenskonformanz zerstört, da $Post$(`R.area()`) ⇒ $Post$(`Q.area()`) nicht gilt, denn `Q.area()` = `Q.x`^{2}, `R.area()` = `R.x` · `R.y`, und `R.x` = `R.y` muss nicht gelten

Also wie nun? gar keine Vererbungsbeziehung ist auch nicht vermittelbar.

Vorschlag des Dozenten: `Square` **`extends`** `Rectangle` sowie zusätzliche Exceptions, falls Preconditions verletzt sind (⇝ Programming by Contract, s.u.)

Übung: Ist Complex Unterklasse von Real oder umgekehrt? Diskutieren Sie Verhaltenskonformanz!

7.5 Inheritance is not Subtyping

Vererbung = Konformanz ?

Man unterscheidet:

- Inheritance (Implementierung-Vererbung): Wiederverwendung von Methodenimplementierungen

  führt typischerweise zu Spezialisierung

- Subtyping (Subtyp-Vererbung): Wiederverwendung von Klienten-Code! jeder Code, der mit O-Objekten arbeitet, kann auch mit U-Objekten arbeiten

  führt zu Verhaltenskonformanz

Durch Inheritance ist die Klassenimplementierung billiger, aber es entstehen Folgekosten beim Klienten (Absturz, ...)

durch Subtyping ist der Klientencode billiger (Lokalitätsprinzip für neue Unterklassen ist voll gewahrt), aber Unterklasse teurer ($U.m()$ verlangt weniger, aber leistet mehr!)

Wiederverwendung ist nie umsonst ...

⇒ eigentlich Trennung von Klassen und Typen notwendig; aber in Java, Eiffel, ... nicht möglich. Jedoch:

- Java hat Trennung Interfaces/Klassen. Subtyping typischerweise durch Interface-Vererbung.
- In C++: `private` Inheritance: kein Subtyp!
- SATHER erlaubt explizite Trennung

7.6 Vererbung vs. Delegation

Subtyping ist für den Klientencode sicher, Spezialisierung potentiell unsicher
⇒ evtl. Ersetzung von Vererbung durch *Delegation*

Delegation: Weiterreichen eines Aufrufs an referenziertes Objekt statt eigener Implementierung (Standardbeispiel: Adapter)

Wir betrachten 3 Stack-Implementierungen: `Stack` mit verketteter Liste, `BoundedStack` mit statischem Array (nicht verhaltenskonformant, zusätzliche Methode `isfull`), `SuperStack` mit dynamischem Array (verhaltenskonformant)

1. Konformanter Fall: Delegation statt `SuperStack` **`extends`** `Stack`

```
class Stack {
  Superstack s;
  public void push(Object x) { s.push(x); }
}
```

Nachteil: zusätzlicher Aufruf, keine dynamische Bindung. Unsinnig.

2. Spezialisierungsfall: Delegation statt `BoundedStack extends Stack`

```
class Stack {
  BoundedStack s;
  public void push(Object x) {
    if (s.isfull())
      throw new StackOverflow();
    s.push(x);
  }
}
```

empfohlen von "Radikalen", die Vererbung nur zulassen, wenn sie verhaltenskonformant ist

Vorteil: zusätzliche Precondition wird geprüft
Nachteil: keine volle Verhaltenskonformanz, keine dynamische Bindung

Der Dozent bevorzugt deshalb Vererbung mit zusätzlicher Prüfung der Precondition:

```
class BoundedStack extends Stack {
  int c = -1; int[] a = new int[42];
  public void push(Object x) {
    if (this.isfull()) {
        throw new StackOverflow();}
    c++; a[c] = x;
  }
  public boolean isfull() {return c==41;}
}
```

3. Amputationsfall:

```
class Stack extends BoundedStack {
  public void isfull() {
    throw new IllegalStackOP();
  }
}
```

entspricht Pre(`Stack.isfull(x)`) = `false`; Postconditions sind gleich
⇒ Amputation ist spezielle Spezialisierung!

(Ansonsten ist `Stack extends BoundedStack` verhaltenskonformant, aber unsinnig ...)