Warum Programme Verträge schließen sollten

RALF HINZE

Institut für Informatik, Lehrstuhl Softwaretechnik, Universität Freiburg
Georges-Köhler-Allee, Gebäude 079, 79110 Freiburg i. Br.

Email: ralf@informatik.uni-bonn.de
Homepage: http://www.informatik.uni-bonn.de/~ralf

4. Juli 2005

(Die Folien sind online verfügbar: .../~ralf/talks.html#T41.)

Überblick

Bisher: grundlegende Konzepte der objekt-orientierten Programmierung (Java).

- Klasse:
  - Attribut (instance variable, field),
  - Methode (instance method, function).
- Vererbung:
  - Redefinition (overriding),
  - dynamische Bindung (late binding).

Heute: Methode zur Entwicklung korrekter objekt-orientierter Software (Eiffel).

Lernziele

- Zusicherungen und Invarianten lesen können. ☞ verstehen
- Definieren können, wann eine Klasse korrekt ist.
- Ein tieferes Verständnis für das Konzept der Vererbung erlangen.
  ☞ anwenden
- Klassen nach der Methode „Design by Contract“ entwickeln können.

Korrektheit

Ein wesentliches Qualitätsmerkmal von Software ist Zuverlässigkeit:

- Korrektheit: die Software tut, was sie soll.
- Robustheit: die Software kommt mit unerwarteten Situationen zurecht.

Was bedeutet Korrektheit? Ist das Programm

$$x := y + 5$$

korrekt?

Spezifikation

☞ Weder noch! Die Frage macht keinen Sinn. Korrektheit ist relativ:

> Ein Programm ist korrekt bezüglich einer Spezifikation.

Das obige Programm erfüllt zum Beispiel die Spezifikation,

$$\{\, y > 7 \,\}\; x := y + 5 \;\{\, x \geqslant 11 \,\}$$

aber auch die folgende Spezifikation,

$$\{\, \mathit{true} \,\}\; x := y + 5 \;\{\, x \neq y \,\}$$

Syntax und Semantik von Spezifikationen

Allgemein: eine Anweisung A lässt sich mit einem Hoare-Tripel spezifizieren:

$$\{\, P \,\}\ A\ \{\, Q \,\}$$

wobei P und Q Zusicherungen sind: P ist die Vorbedingung, Q die Nachbedingung. Eine Zusicherung ist im wesentlichen ein Boolescher Ausdruck.

☞ Bedeutung des Hoare-Tripels (totale Korrektheit):

Wenn P erfüllt ist, dann terminiert A in einem Zustand, der Q erfüllt.

Sind P und Q gegeben, dann ist $\{\, P \,\}\ A\ \{\, Q \,\}$ eine Job-Beschreibung für A.

Zum Knobeln: Ist der Job $\{\, \mathit{false} \,\}\ A\ \{\, Q \,\}$ attraktiv? Oder $\{\, P \,\}\ A\ \{\, \mathit{true} \,\}$?

Vor- und Nachbedingungen

Zusicherungen können zur Spezifikation von Funktionen und Klassen verwendet werden.

Idee: die Spezifikation in den Programmtext integrieren; die Implementierung ausgehend von der Spezifikation entwickeln.

feature *routine* **is require** *pre*
do *body*
ensure *post* **end**

Unterscheidung zwischen dem Was? und dem Wie?: die Spezifikation beschreibt, was die Software machen soll; die Anweisungen geben an, wie das geschieht.

feature *sqrt* $(x : \text{INTEGER}) : \text{INTEGER}$ **is**
require $x \geqslant 0$
do ...
ensure $Result^2 \leqslant x$ **and** $x < (Result + 1)^2$ **end**

Vor- und Nachbedingungen — Klassen

```
class STACK[G]
  feature
    count : INTEGER
    top : G                  is require not empty
                               do        ...
                             end
    empty : BOOLEAN          is do       ... end
    push(x : G)              is do       ... end
                               ensure    not empty
                                         top = x
                                         count = old count + 1
    remove                   is require not empty
                               do        ...
                               ensure    count = old count − 1
                             end
  end
```

Verträge — „Design by contract“

Das Paar aus Vorbedingung und Nachbedingung entspricht einem Vertrag zwischen der Routine und ihren Klienten („Design by Contract“).

```
class CLASS
  ...
  feature routine is require pre
                    do      body
                    ensure  post end
```

Klasse CLASS: „Wenn Sie garantieren, *routine* nur dann aufzurufen, wenn *pre* gilt, dann erledige ich alles notwendige, damit *post* erfüllt wird.“

Verträge — Rechte, Pflichten, Vertragsbruch

☞ Ein Vertrag regelt Rechte und Pflichten.

Partei	Pflichten	Rechte
Klient	erfülle Vorbedingung	aus der Nachbedingung
Lieferant	erfülle Nachbedingung	aus der Vorbedingung

Die Rechte des einen sind die Pflichten des anderen.

☞ Wird eine Zusicherung zur Laufzeit verletzt, ist die Software fehlerhaft.

Wird die Vorbedingung verletzt, liegt der Fehler beim Klienten.
Wird die Nachbedingung verletzt, liegt der Fehler beim Lieferanten.

Verträge — Zuständigkeit

☞ Der Lieferant zieht aus der Vorbedingung einen Nutzen:

- Die Routine muss nur funktionieren, wenn die Vorbedingung erfüllt ist.
- Ist sie verletzt, ist die Routine nicht an die Nachbedingung gebunden: sie kann einen beliebigen Wert zurückgeben, sie kann in eine Endlosschleife gehen, sie kann einen Programmabsturz verursachen.

☞ Insbesondere ist es unnötig, sogar unvereinbar mit der Entwurfsmethodik, die Vorbedingung in der Routine explizit zu testen.

Prinzip der Irredundanz:
Unter keinen Umständen darf die Vorbedingung im Rumpf einer Routine getestet werden.

Weniger ist mehr: Zusicherungen regeln die Zuständigkeit, vermeiden somit redundante Tests und verringern im Endeffekt die Komplexität von Software.

Invarianten

Vor- und Nachbedingungen beschreiben die Eigenschaften einer einzelnen Routine.

Globale Eigenschaften einer Klasse werden durch Invarianten beschrieben.

class STACK[G]
  feature
    ...
  invariant
    $0 \leqslant count$
    $empty = (count = 0)$
  end

Routinen, die ein Objekt der Klasse STACK erzeugen, müssen die Invarianten etablieren; alle anderen Routinen müssen die Invarianten erhalten.

Auch im täglichen Leben gibt es Invarianten: „Verlassen Sie die Gemeinschaftsküche so, wie Sie sie vorzufinden wünschen“.

Korrektheit einer Klasse

Wann ist eine Klasse korrekt?

- Für jede Routine, die ein Objekt erzeugt, muss gelten:

$$\{\, pre \,\}\ body\ \{\, Inv \wedge post \,\}$$

- Für alle anderen Routinen muss gelten:

$$\{\, Inv \wedge pre \,\}\ body\ \{\, Inv \wedge post \,\}$$

Vererbung

Mittels Vererbung lässt sich die Funktionalität einer Klasse erweitern.

class MAXSTACK[G → COMPARABLE] **inherit** STACK[G]
  feature
    max : G
    ...
  invariant
    $max = representation.maximum$
  end

Jede Instanz von MAXSTACK ist auch Instanz von STACK. Deswegen: MAXSTACK erbt die Invariante von STACK.

Allgemein: die Invariante einer Klasse ist die logische Konjunktion der Invarianten aller Oberklassen.

Vererbung — Redefinition

☞ Die Implementierung von *push* und *remove* sind nicht mehr korrekt: sie verletzen die Invariante von MAXSTACK.

☞ Die Funktionen *push* und *remove* müssen neu definiert werden.

```
class MAXSTACK[G → COMPARABLE] inherit STACK[G]
                                 redefine push remove
  ...
  feature
    push(x : G) is do Precursor(x)
                    if x > max then max := x end
                end
  ...
```

Zum Knobeln: wie muss *remove* erweitert werden?

Vererbung — dynamische Bindung

Eiffel verwendet (wie Java) das Prinzip der dynamischen Bindung.

$s : \mathrm{STACK}[\mathrm{INTEGER}];$
$\ldots$
$s.push(4711);$

Ist s zur Laufzeit an ein Objekt der Klasse MAXSTACK gebunden, so wird die $push$-Methode aufgerufen, die in MAXSTACK neu definiert wurde.

Warum ist das die richtige Methode? Nur diese Methode garantiert die Konsistenz des Objekts!

☞ Statische Bindung (in C++ die Voreinstellung) ist keine Alternative, sondern nur eine Optimierung (wenn sie den gleichen Effekt wie dynamische Bindung hat).

Vererbung — Gegenbeispiel

Nicht jede Spezialisierung ist korrekt:

```
class BOUNDEDSTACK[G] inherit STACK[G]   -- FALSCH
  feature
    capacity : INTEGER
  invariant
    count ⩽ capacity
  end
```

Die akkumulierte Invariante lautet: $0 \leqslant count$ **and** $count \leqslant capacity$.

Vererbung — Gegenbeispiel — Fortsetzung

☞ Die Implementierung von $push$ verletzt die Invariante. Die Routine benötigt eine stärkere Vorbedingung:

$push(x : \mathrm{G})$ **is require** $count < capacity$ -- FALSCH
...

Aber: die Redefinition darf die Vorbedingung nur abschwächen, nicht verstärken.

Regel für Redefinitionen:
Redefinierte Routinen dürfen die Vorbedingung der Vorgängerin nur abschwächen bzw. die Nachbedingung verstärken.

Zusammenfassung und Ausblick

Zusammenfassung:

- Verträge regeln die Zuständigkeit.
- Verträge sind Bestandteil des Programmtextes (Ariane 5-Fehlschlag).
- Unterklassen dürfen das Verhalten verbessern, nicht aber willkürlich verändern.
- Summa summarum:
  „Design by Contract“ hilft, korrekte Software zu entwickeln.

Ausblick:

- Monitoring von Zusicherungen (testen, debuggen).
- Wenn eine Routine den Vertrag nicht erfüllen kann: Fehlerbehandlung.

Literatur

- C. A. R. Hoare. An Axiomatic Basis for Computer Programming, Communications of the ACM, 12(10), 576–580, 1969.
- Bertrand Meyer. Applying „Design by Contract“. Computer (IEEE), 25(10), 40–51, Oktober 1992.
- Bertrand Meyer. Object-oriented Software Construction. 2. Auflage. Prentice-Hall, 1997.