

42. Knobelaufgabe #26

$$P = NP ?$$

<http://www.claymath.org/millennium-problems/p-vs-np-problem>

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Vererbung

1003

42. Motivation

Aufgabe: Das Wörterbuch soll um die Möglichkeit erweitert werden, die verwalteten Schlüssel-Wert Paare auszugeben — entweder flüchtig auf dem Bildschirm oder dauerhaft in eine Datei.

Möglichkeit A: Wir erweitern die Klasse *SearchTree* um eine maßgeschneiderte Methode.

member *Print* : *Unit* → *Unit*

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Vererbung

1004

42. Motivation

Implementierungen

Listen

Suchlisten

Suchbäume

visuell lesbarer Text

CSV

XML

Ausgabeformate

Vorteil:

- ▶ Einfache und direkte Umsetzung der Aufgabe.

Nachteil:

- ▶ Kombinatorische Explosion von Programmieraufgaben.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Vererbung

1005

42. Motivation

Möglichkeit B: Wir überführen die Schlüssel-Wert Paare in ein einheitliches Zwischenformat, einen Mediator.

member *Keys* : *Unit* → *List* *!key*

Vorteil:

- ▶ Kombinatorische Explosion von Programmieraufgaben wird vermieden.

Nachteil:

- ▶ Zusätzlicher Speicherplatz wird benötigt: Die Daten, die bereits im Wörterbuch abgelegt sind, werden faktisch dupliziert.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Vererbung

1006

42. Motivation

Möglichkeit C: Wir ersetzen Listen durch eine alternative Darstellung von Sequenzen, die es uns erlaubt, schrittweise und *bedarfsgetrieben* ein Element nach dem anderen zu generieren.

```
member Keys : Unit → IEnumerator ⟨'key⟩
```

Der Sequenztyp *IEnumerator* firmiert unter vielen verschiedenen Namen:

- ▶ „Aufzähler“ (engl. enumerator),
- ▶ „Wiederholer“ (engl. iterator),
- ▶ „Cursor“ (engl. cursor),
- ▶ „faule Liste“ (engl. lazy list) und
- ▶ Generator.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und

aufzählbare

Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

Vertiefung

Vererbung

1007

42. Motivation: Aufzähler

Wir ersetzen eine *Datenstruktur* durch eine *Kontrollstruktur*.

Aus einem *Datum*, das alle Elemente der repräsentierten Sequenz umfasst (eine Liste), wird ein *Programm* (ein Aufzähler), das es erlaubt, alle Elemente nacheinander zu generieren.

```
type IEnumerator ⟨'elem⟩ =  
interface  
  abstract member Current : 'elem  
  abstract member MoveNext : Unit → Bool  
end
```

☞ *Current* gibt das aktuelle Element der Aufzählung zurück; *MoveNext ()* rückt zum nächsten Element vor und signalisiert, ob ein Folgeelement tatsächlich existiert.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und

aufzählbare

Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

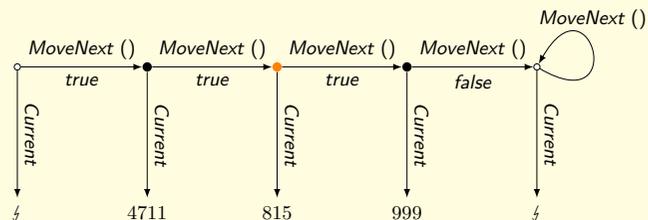
Vertiefung

Vererbung

1008

42. Motivation: Aufzähler

Bei der Verwendung der Schnittstelle *IEnumerator* muss man sich an ein striktes *Protokoll* halten.



☞ Der „Cursor“ der Aufzählung kann entweder auf ein Element zeigen oder vor dem ersten bzw. nach dem letzten Element stehen.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und

aufzählbare

Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

Vertiefung

Vererbung

1009

42. Motivation: Aufzähler

Darstellung des Intervalls *lower .. upper*.

```
let range (lower, upper) =  
  let mutable started = false  
  let mutable current = lower  
  { new IEnumerator ⟨Nat⟩ with  
    member self.Current =  
      if started then if current ≤ upper then current  
        else already-finished ()  
    else not-started ()  
    member self.MoveNext () =  
      if started then current ← current + 1  
        else started ← true  
      current ≤ upper  
  }
```

☞ Die Funktion *range* ist *defensiv programmiert*.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und

aufzählbare

Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

Vertiefung

Vererbung

1010

42. Motivation: Aufzähler

Die implizite Darstellung einer Sequenz durch ein Programm bietet die Möglichkeit, unendliche Sequenzen zu repräsentieren.

```
let squares () =
  let mutable started = false
  let mutable current = 0
  let mutable odd = 1
  { new IEnumerator <Nat> with
    member self.Current =
      if started then current
      else not-started ()
    member self.MoveNext () =
      if started then current ← current + odd
      odd ← odd + 2
      else started ← true
      true
  }
```

☞ MoveNext gibt immer true zurück; die Aufzählung terminiert nie.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische Semantik

Vertiefung

Vererbung

1011

42. Motivation: Aufzähler

Die Funktion *from-list* : *List* <elem> → *IEnumerator* <elem> implementiert einen Repräsentationswechsel.

```
let from-list (list : List <elem>) : IEnumerator <elem> =
  let mutable started = false
  let mutable suffix = list
  { new IEnumerator <elem> with
    member self.Current =
      if started then match suffix with
        | [] → already-finished ()
        | hd :: tl → hd
      else not-started ()
    member self.MoveNext () =
      if started then match suffix with
        | [] → false
        | hd :: tl → suffix ← tl
          non-empty suffix
      else started ← true
          non-empty suffix
  }
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische Semantik

Vertiefung

Vererbung

1012

42. Motivation: Aufzählbare Objekte

☞ Viele „Dinge“ sind aufzählbar, die Elemente einer Zahlenfolge (die Primzahlen, die Catalanzahlen), die Elemente eines Containers (eines Arrays, einer Liste, eines Stacks).

Allgemeine Schnittstelle für aufzählbare Objekte:

```
type IEnumerable <elem> =
  interface
    abstract member GetEnumerator : Unit → IEnumerator <elem>
  end
```

Möglichkeit D: Die Klasse *SearchTree* implementiert die Schnittstelle.

```
interface IEnumerable <key> with
  member self.GetEnumerator () = ...
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische Semantik

Vertiefung

Vererbung

1013

42. Motivation: Aufzählbare Objekte

Syntaktischer Zucker: Die *for*-Schleife zur Linken ist eine Abkürzung für die *while*-Schleife zur Rechten.

```
for x in xs do
  body
```

```
let enumerator = xs.GetEnumerator ()
while enumerator.MoveNext () do
  let x = enumerator.Current
  body
```

☞ *for*-Schleifen geben nicht länger ein Terminierungsversprechen :-). Die Aufzählung muss ja nicht endlich sein (siehe *squares*).

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische Semantik

Vertiefung

Vererbung

1014

42. Motivation: Aufzählbare Objekte

Die beiden Schnittstellen erlauben es, eine *generische Funktion* zu schreiben, die die Elemente einer Aufzählung addiert.

```
let sum (xs : IEnumerable<Nat>) : Nat =  
  let mutable acc = 0  
  for x in xs do  
    acc ← acc + x  
  acc
```

☞ *sum* funktioniert für beliebige aufzählbare Objekte, insbesondere für Listen und Arrays.

VIII Objekte
Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Vererbung

1015

42. Motivation: Sequenzbeschreibungen

Die obigen Programme haben einen *präskriptiven* Charakter. Es wird genau detailliert,
▶ wie das nächste Element in Abhängigkeit vom aktuellen Zustand generiert und
▶ der Zustand aktualisiert wird.

Alternativ können Sequenzen *deskriptiv* mit Hilfe von *Sequenzbeschreibungen* programmiert werden. Wir geben an,

- ▶ was die Elemente der Sequenz sind.

Die Syntax kennen wir schon von Listen- und Arraybeschreibungen.

Redefinition von *squares*:

```
let squares = seq { for n in nats do yield n * n }
```

VIII Objekte
Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Vererbung

1016

42. Motivation: Sequenzbeschreibungen

Die Folge der natürlichen Zahlen selbst lässt sich entweder präskriptiv mit einem Objektausdruck oder deskriptiv mit Hilfe einer rekursiven *Wertedefinition* programmieren.

```
let rec nats =  
  seq { yield 0  
        for n in nats do  
          yield n + 1 }
```

☞ Giuseppe Peanos Beschreibung der natürlichen Zahlen wird eingefangen: Eine natürliche Zahl ist entweder 0 oder der Nachfolger $n + 1$ einer natürlichen Zahl n .

VIII Objekte
Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Vererbung

1017

42. Motivation: Sequenzbeschreibungen

Die Definition des Aufzählers für die Schlüssel des Wörterbuchs steht noch aus. Im Wesentlichen ein Inorder-Durchlauf:

```
let rec inorder = function  
  | Leaf → Seq.empty  
  | Node (l, x, r) → seq { yield! inorder l; yield x; yield! inorder r }
```

☞ *Seq.empty* generiert die leere Sequenz.

☞ *yield!* e ist eine Abkürzung für *for x in e do yield x*.

VIII Objekte
Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Vererbung

1018

42. Abstrakte Syntax

Wir erweitern Ausdrücke um Sequenzbeschreibungen.

$e \in \text{Expr} ::=$
| $\text{seq } \{se\}$

Ausdrücke:
Sequenzbeschreibungen

Innerhalb der Klammern steht ein *Sequenzausdruck*, dessen Syntax wir schon von Listen- und Arraybeschreibungen kennen.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Vererbung

1019

42. Statische Semantik

Sequenzbeschreibungen besitzen den Typ $\text{seq } \langle t \rangle$, ein Alias für den Schnittstellentyp $IEnumerable \langle t \rangle$.

$t \in \text{Type} ::=$
| $\text{seq } \langle t \rangle$

Typen:
Sequenztyp

Listen und Arrays implementieren die Schnittstelle $IEnumerable \langle t \rangle$ alias $\text{seq } \langle t \rangle$ und sind damit Untertypen von $\text{seq } \langle t \rangle$.

$\overline{\text{List } \langle t \rangle} \preceq \text{seq } \langle t \rangle$

$\overline{\text{Array } \langle t \rangle} \preceq \text{seq } \langle t \rangle$

☞ Typregeln, siehe Skript.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Vererbung

1020

42. Dynamische Semantik

Die Sequenzbeschreibung $\text{seq } \{se\}$ wird in den Ausdruck $\llbracket se \rrbracket$ übersetzt, wobei $\llbracket se \rrbracket$ wie folgt definiert ist.

$\llbracket \text{yield } e \rrbracket = \text{Seq.singleton } e$
 $\llbracket \text{yield! } e \rrbracket = e$
 $\llbracket se_1; se_2 \rrbracket = \text{Seq.append } (\llbracket se_1 \rrbracket) (\llbracket se_2 \rrbracket)$
 $\llbracket \text{if } e \text{ then } se \rrbracket = \text{if } e \text{ then } \llbracket se \rrbracket \text{ else } \text{Seq.empty}$
 $\llbracket \text{for } x \text{ in } e \text{ do } se \rrbracket = \text{Seq.collect } (\text{fun } x \rightarrow \llbracket se \rrbracket) e$

☞ Wir beschränken uns auf die Definition der Konkatenation.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Vererbung

1021

42. Dynamische Semantik

Konkatenation von Aufzählern:

```
let append-enumerators (first : IEnumerable <'elem>)
                      (second : IEnumerable <'elem>) =
  let mutable first-active = true
  { new IEnumerable <'elem> with
    member self.Current =
      if first-active then first.Current
      else second.Current
    member self.MoveNext () =
      if first-active then first-active <- first.MoveNext ()
      first-active || second.MoveNext ()
      else second.MoveNext ()
  }
```

☞ Wir merken uns, ob der erste Aufzähler aktiv ist. Wenn nicht, werden die Nachrichten an den zweiten Aufzähler delegiert.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische
Semantik

Vertiefung

Vererbung

1022

42. Dynamische Semantik

VIII Objekte
Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Vererbung

Konkatenation von aufzählbaren Objekten:

```
let append (first : IEnumerable ('elem))
           (second : IEnumerable ('elem)) =
  { new IEnumerable ('elem) with
    member self.GetEnumerator () =
      append-enumerators (first.GetEnumerator ())
                        (second.GetEnumerator ())
  }
```

1023

42. Vertiefung: Testen

VIII Objekte
Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Vererbung

Harry Hacker hat einen neuen Sortieralgorithmus entwickelt: *lightning-sort*. Lisa Lista schlägt vor, das Programm vor dem produktiven Einsatz systematisch zu testen.

☞ Sequenzbeschreibungen eignen sich wunderbar für die Generierung von Testdaten.

```
for n in 0..10 do
  for xs in permutations [1..n] do
    if lightning-sort xs <> List.sort xs then
      raise (Panic "Harry!")
```

Die Funktion *permutations* : *List* 'a' → *seq* (List 'a') generiert systematisch alle Permutationen der angegebenen Liste.

Summa summarum werden 4.037.914 Tests durchgeführt.

Wie lassen sich die Permutationen einer Liste systematisch generieren?

1024

42. Permutationsalgorithmen

VIII Objekte
Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Vererbung

- ▶ *Peano Entwurfsmuster*: Problem der Größe n wird auf Problem der Größe $n - 1$ zurückgeführt.
- ▶ *Permutieren*: die Problemgröße entspricht der Anzahl der zu permutierenden Elemente.
- ▶ *Permutieren durch Einfügen*:
 - ▶ Lege das erste Element zur Seite,
 - ▶ bilde alle Permutationen der restlichen Elemente,
 - ▶ füge das erste Element in jede Permutation nacheinander an allen möglichen Positionen ein.
- ▶ Fokussiert auf die Eingabe.
- ▶ *Permutieren durch Auswählen*:
 - ▶ Wähle nacheinander alle Elemente aus,
 - ▶ bilde jeweils alle Permutationen der restlichen Elemente,
 - ▶ setze das entfernte Element vor jede korrespondierende Permutation.
- ▶ Fokussiert auf die Ausgabe.

1025

42. Permutieren durch Einfügen

VIII Objekte
Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Vererbung

Beispiel: Wir wollen alle Ziffern der Zahl 1234 permutieren.

Wir entfernen die erste Ziffer, die ①, und berechnen die $3! = 6$ Permutationen der restlichen Ziffern: 234, 324, 342, 243, 423, 432. In jede dieser Permutationen müssen wir ① an allen 4 Positionen einfügen.

```
①234 2①34 23①4 234①
①324 3①24 32①4 324①
①342 3①42 34①2 342①
①243 2①43 24①3 243①
①423 4①23 42①3 423①
①432 4①32 43①2 432①
```

Insgesamt erhalten wir $3! \cdot 4 = 4! = 24$ Permutationen.

1026

42. Permutieren durch Einfügen

Die Hilfsfunktion *insertions* fügt ein Element an allen Positionen einer Liste ein.

```
let rec insertions x = function
| []      → seq {yield [x]}
| xs & (y :: ys) → seq {yield (x :: xs)
                        for zs in insertions x ys do
                          yield y :: zs}

let rec permutations = function
| []      → seq {yield []}
| x :: xs → seq {for ys in permutations xs do
                  for zs in insertions x ys do
                    yield zs}
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische Semantik

Vertiefung

Vererbung

1027

42. Permutieren durch Auswählen

Beispiel: Es wird nacheinander die erste Ziffer der Ausgabe ausgewählt: zuerst ①, dann ②, dann ③ und schließlich ④. Für jede Auswahl werden die $3! = 6$ Permutationen der restlichen Ziffern bestimmt und jeweils an die erste Ziffer angehängt.

```
①234 ①243 ①324 ①342 ①423 ①432
②134 ②143 ②314 ②341 ②413 ②431
③214 ③241 ③124 ③142 ③421 ③412
④231 ④213 ④321 ④312 ④123 ④132
```

Wiederum erhalten wir insgesamt $4 \cdot 3! = 4! = 24$ Permutationen.

☞ Permutieren durch Auswählen generiert die Permutationen in *lexikographischer Reihenfolge*.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische Semantik

Vertiefung

Vererbung

1028

42. Permutieren durch Auswählen

Die Hilfsfunktion *deletions* gibt neben dem ausgewählten Element zusätzlich die Liste der restlichen Elemente zurück.

```
let rec deletions = function
| []      → Seq.empty
| x :: xs → seq {yield (x, xs)
                  for (y, ys) in deletions xs do
                    yield (y, x :: ys)}

let rec permutations = function
| []      → seq {yield []}
| xs     → seq {for (y, ys) in deletions xs do
                  for zs in permutations ys do
                    yield y :: zs}
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische Semantik

Vertiefung

Vererbung

1029

42. Lösung Knobelaufgabe #19

Die *unendliche* Folge

```
0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 5 ...
```

ist invariant unter der folgenden Transformation.

- ▶ Jedes Element wird um 1 erhöht.
- ▶ An den Anfang und zwischen je zwei Elemente wird eine 0 gesetzt.

Direkte Umsetzung mit Hilfe einer rekursiven *Wertdefinition*:

```
let rec carry =
seq {yield 0
     for i in carry do
       yield i + 1
     yield 0}
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische Semantik

Vertiefung

Vererbung

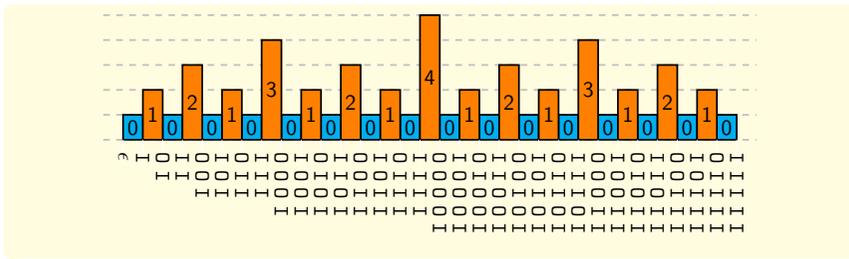
1030

42. Lösung Knobelaufgabe #19

Die *unendliche* Folge

0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 5...

heißt *Übertragsequenz* (engl. binary carry or ruler sequence). Das n -te Element entspricht dem Exponenten der höchsten 2-Potenz, die n teilt.



☞ Siehe A007814 in der On-Line Encyclopedia of Integer Sequences (kurz OEIS, <http://oeis.org/>).

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und

aufzählbare

Objekte

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

Vererbung

Vererbung

1031

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und

aufzählbare

Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1032

It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing (...) is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing.

— Christopher Strachey

43. Implementierungsvererbung

Wie Schnittstellen durch „Vererbung“ verbreitert werden können, so lassen sich auch Klassen um zusätzliche Funktionalität erweitern.

```
type TrustMeGold (seed : Nat) =
  class
    inherit TrustMe (seed)
    member self.Clear () : Nat =
      let amount = self.Balance
      self.Withdraw amount
      amount
  end
```

☞ *Implementierungsvererbung*: *TrustMeGold* ist eine *Unterklasse* (engl. subclass) der Klasse *TrustMe*, der *Oberklasse* (engl. subclass) bzw. *Basisklasse* (engl. base class).

☞ *Einfachvererbung*: Eine Klasse darf nur von *einer* Oberklasse erben.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und

aufzählbare

Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1033

43. Redefinition

☞ Vererbung kann auch verwendet werden, um Verhalten zu ändern (Alternative zu Delegation).

```
type TrustMeStudent (seed : Nat, limit : Nat) =
  class
    inherit TrustMe (seed)
    let limit = min limit 1000
    override self.Withdraw (amount : Nat) =
      if amount > limit then raise Limit
      else base.Withdraw amount
  end
```

☞ *override* zeigt an, dass sich über die bereits bestehende Definition der Methode *Withdraw* hinweggesetzt wird (engl. override).

☞ In der Redefinition von *Withdraw* wird auf die ursprüngliche Definition mit *base.Withdraw* zugegriffen.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und

aufzählbare

Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1034

43. Virtuelle Methoden

Die Basisklasse muss allerdings Reimplementierungen zulassen.

```

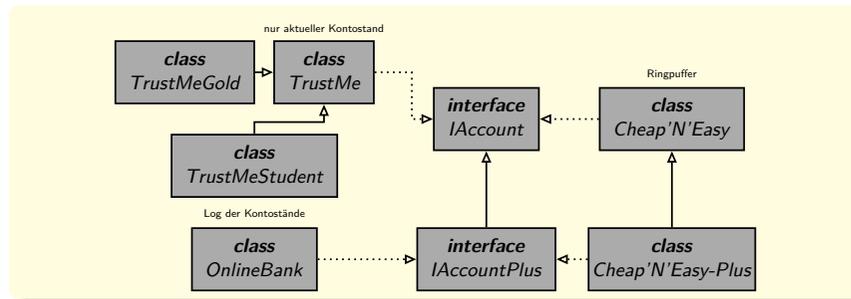
type TrustMe (seed : Nat) =
  class
  ...
  abstract member Withdraw : Nat → Unit
  default self.Withdraw (amount : Nat) =
    funds ← monus (funds, amount)
  ...
end
    
```

- ☞ Neben der Signatur von *Withdraw* wird eine *Standarddefinition* (engl. default definition) angegeben — diese ist optional.
- ☞ Eine abstrakte Methode mit einer Standarddefinition heißt auch *virtuelle Methode* (engl. virtual method).

VIII Objekte
Ralf Hinze
Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Vererbung
Grundlagen
Abstrakte Klassen
Delegation versus Vererbung
Paradigmen

43. Klassendiagramm

Die Klassenhierarchie kann vollkommen unabhängig von der Schnittstellenhierarchie entwickelt werden.



☞ Programme, siehe Skript.

VIII Objekte
Ralf Hinze
Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Vererbung
Grundlagen
Abstrakte Klassen
Delegation versus Vererbung
Paradigmen

43. Entwurfsmuster

Entwurfsmuster existieren in verschiedenen „Größenordnungen“.

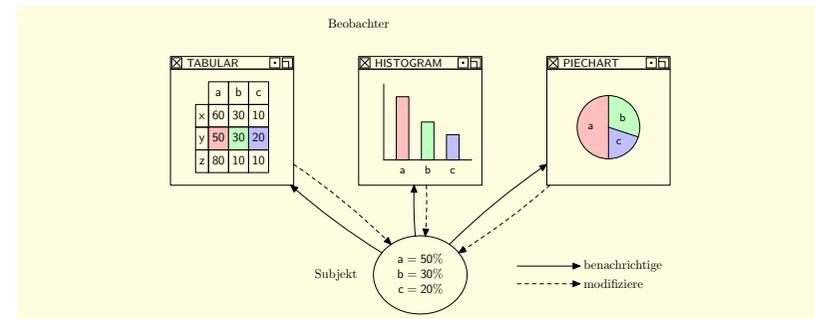
- ▶ Entwurfsmuster im Kleinen (small-scale): Organisation und Entwurf einzelner Funktionen oder Gruppen von Funktionen: Peano, Leibniz, Struktur.
- ▶ Entwurfsmuster im Größeren (medium-scale): Organisation und Entwurf von größeren Programmteilen oder Bibliotheken: zum Beispiel „Beobachter“ (engl. observer)
- ▶ Entwurfsmuster im Großen (large-scale): Organisation und Entwurf der Architektur eines Programms oder Softwaresystems.

Im Folgenden schauen wir uns zwei Beispiele für „medium-scale“ Entwurfsmuster an: Beobachter und Schablonenmethode.

VIII Objekte
Ralf Hinze
Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Vererbung
Grundlagen
Abstrakte Klassen
Delegation versus Vererbung
Paradigmen

43. Beobachter-Entwurfsmuster

▶ *Beispiel:* Trennung von Daten und deren Repräsentation.



- ▶ Subjekt hat beliebig viele Beobachter; ändert sich der Zustand des Subjekts, sollen alle Beobachter darüber informiert werden.
- ▶ *Lose Kopplung:* Das zu beobachtende Objekt soll von den Beobachtern unabhängig bleiben, ihre Schnittstelle nicht kennen.
- ▶ *Vorteil:* Ein Aspekt lässt sich unabhängig von den anderen ändern.

VIII Objekte
Ralf Hinze
Objekte
Untertypen
Klassen
Aufzähler und aufzählbare Objekte
Vererbung
Grundlagen
Abstrakte Klassen
Delegation versus Vererbung
Paradigmen

43. Beobachter-Entwurfsmuster: Demo

Wir illustrieren das Beobachter-Entwurfsmuster am Beispiel eines beobachtbaren Zählers.

Zähler mit einer einfachen „Fortschrittsanzeige“:

```
let steps = counter ()
do steps.Attach (fun n → if n % 1000 = 0 then putchar '*')
let rec fibonacci (n : Nat) : Nat =
  steps.State ← steps.State + 1
  if n ≤ 1 then n
  else fibonacci (n - 1) + fibonacci (n - 2)
```

Nach jeweils 1000 Schritten wird ein Asteriskus '*' ausgegeben.

```
Mini) fibonacci 20
*****val it = 987
Mini) steps.State
val it = 21891
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1039

43. Beobachter-Entwurfsmuster: Beobachter

Ein *Beobachter* ist eine Prozedur, die einen Zustand verarbeitet.

```
type Observer ('state) = 'state → Unit
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1040

43. Beobachter-Entwurfsmuster: Subjekt

Das Subjekt, das wir beobachten wollen, wird durch eine *abstrakte Klasse* realisiert (die Eigenschaft *State* ist abstrakt).

```
[< AbstractClass >]
type Subject ('state) () =
  class
    let mutable observers = []
    abstract member State : 'state with get, set
    member self.Attach (observer : Observer ('state)) =
      observers ← observer :: observers
      observer self.State // notify observer
    member internal self.Notify () =
      for observer in observers do
        observer self.State
  end
```

☞ Die interne Methode *Notify* ist nicht für den Endbenutzer gedacht, sondern wird lediglich Unterklassen zur Verfügung gestellt.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1041

43. Abstrakte Klassen

Das sogenannte *Attribut* (engl. attribute) [*< AbstractClass >*] kennzeichnet die Klasse *Subject* als abstrakt: nicht alle Methoden sind implementiert.

Auf diese Weise wird sichergestellt, dass nicht aus Versehen die Bereitstellung einer *default*-Methode versäumt wird.

```
[< AbstractClass >]
type Subject ('state) () =
  class
    ...
    abstract member State : 'state with get, set
    ...
  end
```

☞ Eine abstrakte Klasse hat keine Instanzen.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1042

43. Beobachter-Entwurfsmuster: Zähler

Eine abstrakte Klasse kann durch eine Unterklasse konkretisiert werden oder durch einen Objektausdruck.

```
let counter () =  
  let mutable n = 0  
  { new Subject <Nat> () with  
    member self.State  
      with set k = n ← k  
        self.Notify ()  
      and get () = n  
  }
```

Der Zähler ist ein Subjekt vom Typ *Nat*, der die Eigenschaft *State* mit entsprechenden Getter und Setter-Methoden konkretisiert.

☞ Wir beobachten nur schreibende Zugriffe (via *Notify*).

43. Schablonen-Entwurfsmuster

- ▶ *Beispiel*: Der Ablauf eines Zwei-Personen-Spiels lässt sich in eine feste *Schablone* pressen: Die Spieler sind abwechselnd am Zug; ein Spieler verliert, wenn keine Zugmöglichkeiten mehr existieren.
- ▶ *Idee*: Skelett eines Algorithmus festlegen, aber die Teilschritte variabel halten.
- ▶ So besteht die Möglichkeit, einzelne Schritte des Algorithmus zu verändern oder zu überschreiben, ohne dass die zu Grunde liegende Struktur des Algorithmus modifiziert werden muss.

Wir illustrieren das Schablonen-Entwurfsmuster am Beispiel von *neutralen Zwei-Personen-Spielen*.

43. Schablonen-Entwurfsmuster: Demo

Das bekannteste neutrale Zwei-Personen-Spiel ist *Nim*: Von einem Haufen Streichhölzer werden in einem Spielzug 1–3 Hölzer entfernt.

```
Mini> (Nim 10).Play ()  
||||| |||||  
number of matches: 1  
||||| |||||  
I take 1  
||||| |||  
number of matches: 3  
|||||  
I take 1  
||||  
number of matches: 2  
||  
I take 2  
I win
```

☞ Der menschliche Spieler eröffnet den Reigen und entfernt 1 Streichholz. Nach drei Runden gewinnt der Rechner („I win“).

43. Schablonen-Entwurfsmuster

Die variablen Bestandteile des Algorithmus werden mit abstrakten Methoden modelliert. Der Algorithmus selbst ist eine konkrete Methode, die aber von den abstrakten Bestandteilen abhängt.

```
[< AbstractClass >]  
type TwoPlayerGame () =  
  class  
    abstract member Finished : Bool  
    abstract member Position : Unit → Unit  
    abstract member Move : Bool → Unit  
    abstract member Winner : Bool → Unit  
    member self.Play () =  
      let mutable turn = false  
      while not self.Finished do  
        self.Position ()  
        turn ← not turn  
        self.Move turn  
        self.Winner turn  
    end
```

43. Schablonen-Entwurfsmuster

```
type Nim (n : Nat) =
  inherit TwoPlayerGame ()
  let mutable matches = n
  let announce i = putline ("I take " ^ show i); i
  override self.Finished =
    matches = 0
  override self.Position () = putline ...
  override self.Move (human : Bool) : Unit =
    let i =
      if human then
        checked-query ("number of matches",
          both (is-nat, both (is-greater 0, is-less 4)))
      else
        announce (max 1 (matches % 4))
    matches ← matches - i
  override self.Winner human =
    putline ((if human then "you" else "I") ^ " win")
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1047

43. Kritik

I fear the the new object-oriented systems may suffer the fate of LISP, in that they can do many things, but the complexity of the class hierarchies may cause them to collapse under their own weight.
— Bill Joy

Von den unzähligen Sprachfeatures, die wir im Laufe der Vorlesung eingeführt haben, ist Vererbung das wohl kontroverseste.

inheritance breaks abstraction

In vielen Fällen lässt sich Implementierungsvererbung durch einfachere und bewährte Sprachfeatures ersetzen.

- ▶ *Schablonen-Entwurfsmuster:*
eine Schablone ist eine *Funktion höherer Ordnung*;
- ▶ *Beobachter-Entwurfsmusters:*
Objektconstructoren höherer Ordnung.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1048

43. Schablonen-Entwurfsmuster — da capo

Eine Schnittstelle für Zwei-Personen-Spiele:

```
type ITwoPlayerGame =
  interface
    abstract member Finished : Bool
    abstract member Position : Unit → Unit
    abstract member Move : Bool → Unit
    abstract member Winner : Bool → Unit
  end
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1049

43. Schablonen-Entwurfsmuster — da capo

Das Schablonen-Entwurfsmuster lässt sich alternativ mit Hilfe einer Funktion umsetzen, die mit einem Spielobjekt parametrisiert ist.

```
let play (game : ITwoPlayerGame) =
  let mutable turn = false
  while not game.Finished do
    game.Position ()
    turn ← not turn
    game.Move turn
  game.Winner turn
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1050

43. Beobachter-Entwurfsmuster — da capo

Eine Schnittstelle für „Zustände“:

```
type IState ⟨'state⟩ =  
  interface  
    abstract member State : 'state with get, set  
  end
```

43. Beobachter-Entwurfsmuster — da capo

Wir parametrisieren den Zähler mit dem Beobachter:

```
let counter (observer : Nat → Unit) =  
  let mutable n = 0  
  { new IState ⟨Nat⟩ with  
    member self.State  
      with set k = n ← k  
        observer n  
      and get () = n  
    }  
let steps = counter (fun n → if n % 1000 = 0 then putchar '*')
```

☞ Der Objektkonstruktor *counter* ist eine *Funktion höherer Ordnung*.

☞ „Low cost“ Version; vollständige Version, siehe Skript.

43. Delegation versus Vererbung

Um Objekte kompositional zu definieren, haben wir zwei grundlegende Techniken kennengelernt:

- ▶ *Delegation*;
- ▶ *Unterklassen*: Implementierungsvererbung.

☞ Im Allgemeinen ist Delegation die bessere Wahl. Warum? Um die Gründe zu verstehen, müssen wir etwas ausholen ...

☞ Es ist wichtig, den Mechanismus des Nachrichtensendens im Zusammenspiel von Klassen und Unterklassen genau zu verstehen.

43. Vererbung: Schrittzähler

```
type StepCounter () =  
  class  
    let mutable n = 0  
    abstract Inc : Unit → Unit  
    default self.Inc () =  
      putline "Inc: base class"  
      n ← n + 1  
    abstract Step : Int → Unit  
    default self.Step k =  
      putline "Step: base class"  
      for i in 1..k do  
        self.Inc ()  
  end
```

☞ *Inc* und *Step* sind beide virtuelle Methoden.

43. Vererbung: Paritätszähler

```
type ParityCounter () =  
  class  
    inherit StepCounter ()  
    let mutable even = true  
    override self.Inc () =  
      putline "Inc: subclass"  
      base.Inc ()  
      even ← not even  
    override self.Step k =  
      putline "Step: subclass"  
      base.Step k  
      even ← even = (k % 2 = 0)  
    member self.Parity = even  
  end
```

👁️ Der Paritätszähler hält nach, ob die Gesamtzahl der Schritte gerade oder ungerade ist.

VIII Objekte

Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und
aufzählbare
Objekte
Vererbung
Grundlagen
Abstrakte Klassen
Delegation versus
Vererbung
Paradigmen

1055

43. Vererbung: Demo

Senden wir einem Paritätszähler die Nachricht `Step 3`, ergibt sich eine interessante Nachrichtenkaskade.

```
Mini) let counter = ParityCounter ()  
Mini) counter.Step 3  
Step: subclass  
Step: baseclass  
Inc: subclass  
Inc: baseclass  
Inc: subclass  
Inc: baseclass  
Inc: subclass  
Inc: baseclass  
Mini) counter.Parity  
true
```

👁️ *Problem:* Jede Erhöhung wird doppelt gezählt!

VIII Objekte

Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und
aufzählbare
Objekte
Vererbung
Grundlagen
Abstrakte Klassen
Delegation versus
Vererbung
Paradigmen

1056

43. Vererbung: Paritätszähler — da capo

Lösung: Wir dürfen `Step` nicht redefinieren.

```
type ParityCounter () =  
  class  
    inherit StepCounter ()  
    let mutable even = true  
    override self.Inc () =  
      putline "Inc: subclass"  
      base.Inc ()  
      even ← not even  
  
    member self.Parity = even  
  end
```

VIII Objekte

Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und
aufzählbare
Objekte
Vererbung
Grundlagen
Abstrakte Klassen
Delegation versus
Vererbung
Paradigmen

1057

43. Vererbung: Schrittzähler — da capo

Problem gelöst? Nicht ganz ...

Einige Zeit später optimiert Harry Hacker die Klasse `StepCounter`:

```
type StepCounter () =  
  class  
    let mutable n = 0  
    abstract Inc : Unit → Unit  
    default self.Inc () =  
      putline "Inc: base class"  
      n ← n + 1  
    abstract Step : Int → Unit  
    default self.Step k =  
      putline "Step: base class"  
      n ← n + k  
  end
```

VIII Objekte

Ralf Hinze

Objekte
Untertypen
Klassen
Aufzähler und
aufzählbare
Objekte
Vererbung
Grundlagen
Abstrakte Klassen
Delegation versus
Vererbung
Paradigmen

1058

43. Vererbung: Demo

Wiederholen wir jetzt die obige Interaktion, ergibt sich das folgende Bild.

```
Mini) let counter = ParityCounter ()
Mini) counter.Step 3
Step: baseclass
Mini) counter.Parity
true
```

☞ Die Parität wird überhaupt nicht aktualisiert!

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1059

43. Vererbung: Fazit

☞ Die Änderung der Basisklasse zieht nicht-lokale Änderungen der Unterklassen nach sich — eine Klasse kann viele Unterklassen besitzen und diese können über verschiedene Module verstreut sein.

☞ Die Unterklasse *ParityCounter* lässt sich nicht ohne intime Kenntnis der Oberklasse *StepCounter* korrekt implementieren.

Die Kapselung, die eine Klasse vornimmt, wird durch virtuelle Methoden und Vererbung aufgeweicht:

inheritance breaks abstraction

☞ Das Problem tritt bei der Delegation nicht auf ...

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1060

43. Delegation: Schrittzähler

```
type IStepCounter =
  abstract member Inc : Unit → Unit
  abstract member Step : Int → Unit
```

```
let step-counter () =
  let mutable n = 0
  { new IStepCounter with
    member self.Inc () =
      putline "Inc: delegatee"
      n ← n + 1
    member self.Step k =
      putline "Step: delegatee"
      for i in 1..k do
        self.Inc ()
  }
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1061

43. Delegation: Paritätszähler

```
type IParityCounter =
  inherit IStepCounter
  abstract member Parity : Bool
```

```
let parity-counter () =
  let basic = step-counter () // delegatee
  let mutable even = true
  { new IParityCounter with
    member self.Inc () =
      putline "Inc: delegator"
      basic.Inc ()
      even ← not even
    member self.Step k =
      putline "Step: delegator"
      basic.Step k
      even ← even = (k % 2 = 0)
    member self.Parity = even
  }
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1062

43. Delegation: Demo

```
Mini) let counter = parity-counter ()
Mini) counter.Step 3
Step: delegator
Step: delegatee
Inc: delegatee
Inc: delegatee
Inc: delegatee
Mini) counter.Parity
false
```

☞ Die Parität wird korrekt nachgehalten.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1063

43. Delegation: Schrittzähler — da capo

Einige Zeit später optimiert Harry Hacker den Konstruktor *step-counter*:

```
let step-counter () =
  let mutable n = 0
  { new IStepCounter with
    member self.Inc () =
      putline "Inc: delegatee"
      n ← n + 1
    member self.Step k =
      putline "Step: delegatee"
      n ← n + k
  }
```

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1064

43. Delegation: Demo

Die Kontrollausgaben ändern sich:

```
Mini) let counter = parity-counter ()
Mini) counter.Step 3
Step: delegator
Step: delegatee
Mini) counter.Parity
false
```

☞ Die Parität wird weiterhin korrekt nachgehalten.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1065

43. Delegation: Fazit

Wie erklären sich die Unterschiede zwischen den beiden Ansätzen?

- ▶ *Vererbung*: Es existiert genau *ein* Objekt, der Paritätszähler, der aber Verhalten von der Basisklasse erbt. Mit dem Akt der Vererbung werden die Methoden kompliziert miteinander verflochten — via Nachrichten an die Basisklasse und via Selbstnachrichten.
- ▶ *Delegation*: Es existieren *zwei* Objekte, der Paritätszähler und der Schrittzähler, die in nachvollziehbarer Weise miteinander interagieren: Einer delegiert Arbeit an den anderen.

☞ *Fazit*: virtuelle Methoden und Unterklassen sollten mit Bedacht eingesetzt werden.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und
aufzählbare
Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1066

43. Paradigmen: objektorientierte Programmierung

obiectare

entgegenwerfen; preisgeben; vorwerfen, vorhalten.

As if my mother, instead of saying: „Please bring a chair to the table“, had said „please, send a message to an instance of the class chair, which is a subclass of class furniture, to be brought to the very instance of the class table, being also a subclass of class furniture.“
— László Böszörményi

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1067

43. Paradigmen

To know another language is to have a second soul.
— Charlemagne (742–814)

You can never understand one language until you understand at least two.
— Ronald Searle (1920–)

Überblick:

- ▶ *funktionale Programmierung*: deskriptiv, problemnah, werte-fokussiert;
- ▶ *imperative Programmierung*: präskriptiv, maschinennah, effekt-fokussiert;
- ▶ *objektorientierte Programmierung*: Programmierung im Großen.

☞ Die Paradigmen konkurrieren nicht; sie ergänzen sich sinnvoll.

VIII Objekte

Ralf Hinze

Objekte

Untertypen

Klassen

Aufzähler und aufzählbare Objekte

Vererbung

Grundlagen

Abstrakte Klassen

Delegation versus

Vererbung

Paradigmen

1068

