Konzepte in der Übersicht

Funktion	**Abstraktion** Prozedur	generische Vereinbarung	*Programmieren im Kleinen*
Werte • Typdefinition • Ausdruck	**Speicher** • Variable • Befehl	**Bindung** • Vereinbarung • Block	
funktionale	*imperative Facette*	*Vereinbarungs-*	

Konzepte in der Übersicht

abstrakter Datentyp	**Kapselung** Objekt /Klasse	generischer Modul	
Typsysteme • Überladen • Anpassung • Polymorphie • Vererbung	**Ablaufsteuerung** • Sprung • Ausweg • Ausnahme		*Programmieren im Großen*
Funktion	**Abstraktion** Prozedur	generische Vereinbarung	*Programmieren im Kleinen*
Werte • Typdefinition • Ausdruck	**Speicher** • Variable • Befehl	**Bindung** • Vereinbarung • Block	
funktionale	*imperative Facette*	*Vereinbarungs-*	

Konzepte von Programmiersprachen
5. Kapselung

Weshalb braucht man Moduln?

D. L. Parnas (1972)

Global Variables considered harmful. CACM 15: 1053-1058

Inhalt

Moduln
- Schnittstellen
- generische Moduln
- generische Parameter
- generische Typparameter

methodisch besonders wichtige Modularten
- abstrakte Datentypen
- Objekte und Klassen

Was leisten Moduln?

Definition

ein Modul ist eine Zusammenfassung von Vereinbarungen, die für einen bestimmten Zweck getroffen werden
- der Modul *kapselt* seine Komponenten

***Abstraktion* im Großen**
- ... erlaubt die Unterscheidung:
  - — *Was* soll ein Modul leisten?
  - — *Wie* soll er implementiert werden?

Schnittstelle und Benutzung

Schnittstelle (Ada)

```
package trig is
  function sin (x : Float) return Float;
  function cos (x : Float) return Float;
end trig;
```

Benutzung

... `trig.sin (theta/2.0)` ...

beschreibt die Schnittstelle *alle wichtigen* Eigenschaften des Moduls?

Modul-Rumpf (*body*)

der Rumpf beschreibt die Realisierung der Vereinbarungen

```
package body trig is
  pi: constant Float := 3.1416...;
  function norm (x : Float) return Float is …;
  function sin (x : Float) return Float is …;
  function cos (x : Float) return Float is …;
end trig;
```

Implementierungssicht

ein Modul mit Schnittstellen in ML

```
structure trig =struct local
                      val pi = 3.1416...;
(* verborgen*)          fun norm (x : real) = ...;
                    in
(* exportiert *)        fun sin (x : real) = ...;
                      and cos (x : real) = ...;
                end   end
```

Anmerkungen

`struct` *D* **`end`** *kapselt* die Vereinbarungen *D*

`local` *L* **`in`** *E* **`end`** definiert *L* lokal für die exportierten Vereinbarungen *E*

konkrete Datentypen

Rationale Zahlen in ML

```
datatype rational = rat of (int * int);
val zero = rat(0,1); val one = rat(1,1);
fun op ++ (rat(m1,n1): rational, rat(m2,n2): rational
)                 = rat(m1*n2 + m2*n1, n1 * n2)
```

Probleme

- Redundanz
- fehlerhafte Werte
- vordefinierte Operationen sind falsch

rationale Zahlen als abstrakter Datentyp

```
abstype rational = rat of (int * int) with
  val zero = rat(0, 1);
  val one  = rat(1, 1);
  fun op // (m: int, n: int)
     = if n <> 0 then rat(m,n) else error "no rational";
  fun op ++ (rat(m1,n1): rational, rat(m2,n2): rational)
     = rat(m1*n2 + m2*n1, n1 * n2);
  fun op == (rat(m1,n1): rational, rat(m2,n2): rational)
     = (m1*n2 = m2*n1)
end
```

Beobachtungen

- nur mit `zero`, `one`, `by` und `//` können `rat`'s aufgebaut werden
- `plus` stellt sicher, daß der Nenner stets ungleich 0 bleibt
- die Redundanz der Darstellung bleibt verborgen (ist nicht *beobachtbar*)

Objekte

Definition

Ein *Objekt* ist ein Modul mit einer *verborgenen Variablen*, auf die nur mit exportierten Operationen zugegriffen werden kann.

Telefonverzeichnis in Ada

```
package dir_object is
  procedure insert (name : in Name; num: in Number);
  procedure lookup (name : in  Name; num: out Number;
                                     found: out Boolean);
end dir_object;
package body dir_object is
  type Dirptr is ...
  root: Dirptr;
  procedure insert ... is begin ... end insert;
  procedure lookup ... is begin ... end lookup;
begin…
end dir_object;
```


Benutzung des Telefonverzeichnisses

```
dir_object.insert (me, 6041);
...
dir_object.lookup (me, mynumber, ok)
```


Objektklassen

Definition

Eine *Objektklasse* ist ein Muster für das Anlegen von ähnlichen Objekten

Beispiel: Telefonverzeichnisse in Ada

```
generic package dir_class is
  procedure insert (name: in Name; num: in Number);
  procedure lookup (name: in Name; num: out Number;
                                   found: out Boolean);
end dir_class;
package body dir_class is
begin-- genau wie vorher
…
end dir_class;
```


Benutzung von Objektklassen

Instanziierung liefert Objekte

```
package homedir is new dir_class;
package workdir is new dir_class;
```

Benutzung

```
homedir.insert (me, 6041);
workdir.insert (me, 8715);
workdir.lookup (me, mynumber, ok)
```


Telefonverzeichnis als abstrakter Datentyp

```
package dir_type is
  type Directory is limited private;
  procedure insert
    (dir: in Directory; name: in Name; num: in Number);
  procedure lookup (dir : in Directory; name: in  Name;
                    num: out Number; found: out Boolean);
private
  type Directory is …;
end dir_type;
package body dir_type is
  procedure insert … is begin … end insert;
  procedure lookup … is begin … end lookup;
  …
end dir_type;
```

Vergleich von Klassen und abstrakte Datentypen

Benutzung von abstrakten Datentypen

```
use dir_type;
homedir, workdir : Directory;
insert (homedir, me, 6041);
insert (workdir, me, 8715);
lookup (workdir, me, mynumber, ok);
```

Benutzung von Klassen

```
homedir.insert (me, 6041);
workdir.insert (me, 8715);
workdir.lookup (me, mynumber, ok)
```


Parameterisierte Moduln (*generics*)

Generische Moduln sind Abstraktionen von Moduln (Vereinbarungen)

ein generischer Modul abstrahiert von Vereinbarungen (einem Modul):

```
generic package dir_class is …
end dir_class;
package body dir_class is …
end dir_class;
```

Instanziierung

Die *Instanziierung* eines generische Moduls elaboriert den Modulrumpf und liefert Vereinbarungen (ein neuer Modul):

```
package homedir is new dir_class;
```

der logische Schritt

generische Moduln können auch *parameterisiert* werden

mit Konstanten, Variablen, Funktionen, Prozeduren und *Typen*

generische Parameter (Konstanten)

```
generic
  capacity : in Positive;
package queue_class is
  procedure append (item : in  Character);
  procedure remove (item : out Character);
end queue_class;
package body queue_class is
  items : array (1..capacity) of Character;
  size, front, rear: Integer range 0..capacity;
  procedure append (item: in  Character) is …;
  procedure remove (item: out Character) is …;
begin -- initialization
end queue_class;
```


Instanziierung (Konstanten)

Die *Instanziierungen* erzeugen Schlangen verschiedener Kapazitäten:

```
package line_buffer is new queue_class(120);
package terminal_buffer is new queue_class(80);
```


generische Parameter (Typen)

```
generic
  capacity : in Positive;
  type ITEM is private;
package queue is
  procedure append (i: in  ITEM);
  procedure remove (i: out ITEM);
end queue;
```

Instanziierung

```
type Transaction is record … end record;
package audit_trail is
  new queue(120,Transaction);
```


Was macht Typ-Parameter *besonders*?

```
package body queue is
  items : array (1..capacity) of ITEM;
  size, front, rear: Integer range 0..capacity;
  procedure append (i: in  ITEM) is
  begin
    …; items(rear) := i; …;
  end append;
  procedure remove (i: out ITEM) is
  begin
    …; i := items(front); …;
  end remove;
begin
  front := 1; rear:= 0;                    -- initialization
end queue;
```


Das Besondere an generischen Typparametern

„Datentypen bestehen aus Wertemenge und *Operationen*“

- Welche *Operationen* gehören zum formalen generischen Typparameter *F*?
- Welche Operationen werden im Modulrumpf benutzt?
- Welche Operationen gehören zum aktuellen generischen Typparameter *A*?

Forderung

- Für den formalen Typ *F* müssen alle benötigten Operationen angefordert werden

  `type F is` *Spezifikationen der geforderten Operationen für T*`;`

- Im Rumpf dürfen *höchstens* die geforderten Operationen benutzt werden
- Der aktuelle Typparameter *A* muss *mindestens* die geforderten Operationen haben

weitergehende Forderung

Spezifikation der geforderten *Eigenschaften* der geforderten Operationen

Weshalb geht das nicht so einfach?

generische Typparameter in Ada

Spezifikation des generischen Typparameters (Ada)

```
type ITEM is private;
```

besagt, dass `ITEM` eine *Wertzuweisung* `:=` und *Gleichheitsabfrage* `=` haben soll.

Überprüfung des Rumpfes

Im Rumpf wird nur die Zuweisung benutzt

… `items(rear) := i;` …

… `i := items(front);` …

Überprüfung der Instanziierungen

```
type Transaction is record … end record;
package audit_trail is new queue(120,Transaction);
```

Jeder Verbund-Typ hat Zuweisung und Gleichheit

generische Funktions- und Typparameter

```
generic
  type ITEM is private;
  type SEQUENCE is array (Integer range <>) of ITEM;
  with function prec(x,y: ITEM) return Boolean;
package sorting is
  procedure sort (s: in out SEQUENCE);
  procedure merge (s1, s2: in SEQUENCE; s: out SEQUENCE);
end sorting;
package body sorting is
  procedure sort (s: in out SEQUENCE) is
    begin … if prec(s(j), s(i)) then …
    end sort;
  procedure merge(s1, s2: in SEQUENCE; s: out SEQUENCE)
is
  begin … end merge;
end sorting;
```


Instanziieren von Funktions- und Typparametern

die generischen Parameter hängen von einander ab:

Das muß bei der Instanziierung berücksichtigt werden

```
type FloatSequence is array (Integer range <>) of Float;
package ascending is
  new sorting(Float,FloatSequence, "<=");
package decending is
  new sorting(Float,FloatSequence, ">=");
```