Gliederung des Kapitels 1.4

1.4/1.5 Programmierung (und die Sprache Ada 95)

1.4.4 Polymorphie

Wie entwirft man Systeme? Zunächst erstellt man gewisse Wünsche/Forderungen und schreibt auf, welche Funktionen das System erfüllen soll.

Dann beginnt man mit Plänen, Konzepten, Strukturen und Konstruktionen, wobei man schrittweise das bisher Erstellte weiterentwickelt. Hierbei sollte man sich so spät wie möglich konkret festlegen. Dadurch bleibt der Entwurf flexibel und das System kann wesentlich leichter an künftige Änderungen und an die wechselnden Wünsche der Kunden angepasst werden.

Entscheidungen werden umrissen und eingeschränkt, aber Strukturen, konkrete Festlegungen und Datentypen von Variablen werden solange wie möglich offen gehalten. Diese Vielgestaltigkeit bezeichnet man als Polymorphie.

Polymorphie (aus dem Griechischen: *Vielgestaltigkeit*) ist z.B. aus der Biologie, aus den Grundlagen der Mathematik und aus der Linguistik bekannt. Schauen Sie in ein Lexikon!

In der Informatik ist Polymorphie ein Grundprinzip. Mit ihr lässt sich die Wiederverwendung von Programmteilen (engl.: reuse) erleichtern. Die Polymorphie äußert sich durch folgende Maßnahmen:

Möglichst lange den konkreten Datentyp von Variablen offen lassen. Man denke an unspezifizierte Feldgrenzen bei arrays (vgl. 1.3.2.3 a, Folie 77 in Kap. 1.3).

Möglichst lange die konkrete Realisierung offen lassen. Z.B. Spezifikations- und Implementierungsteil trennen.

Parametrisierung von Paketen und Unterprogrammen, um diese für möglichst viele Anwendungen einsetzen zu können (Generizität, siehe im Folgenden).

1.4.4.1: In der Logik bedeutet Polymorphie, dass zu einem Axiomensystem mehrere wesentlich verschiedene Modelle existieren.

Beispiel: Gesucht ist ein Datenbereich D mit Operationen "+" und "$*$" , so dass für alle Elemente $a, b, c \in D$ gilt:

(1) $a + b = b + a$, (2) $(a + b) + c = a + (b + c)$,
(3) $a * b = b * a$, (4) $(a * b) * c = a * (b * c)$,
(5) $a * (b + c) = (a * b) + (a * c)$,
(6) $(b + c) * a = (b * a) + (c * a)$,
(7) D besitzt unendlich viele Elemente.

Zu diesen Axiomen gibt es verschiedene mathematische Strukturen, die sie erfüllen (so genannte "Modelle"); zum Beispiel die natürlichen Zahlen $\mathbb{N}_0$, die ganzen Zahlen $\mathbf{Z}$, die rationalen Zahlen $\mathbf{Q}$, die reellen Zahlen $\mathbb{R}$ usw. Dieses Axiomensystem ist daher polymorph.

1.4.4.2: In der Programmierung spricht man in folgenden Fällen von Polymorphie:

1. Mehrfachverwendung von ***Namen*** (hierzu zählt das Überladen in Ada, siehe 1.4.2.15).

2. ***Variablen*** können je nach aktueller Umgebung Elemente verschiedener Datentypen bezeichnen oder als Werte besitzen.

Beispiele hierfür haben wir schon angedeutet (1.3.2.1); andere sind in Ada teilweise nicht zulässig. Betrachte

```
subtype natural is integer range 0..integer'Last;
subtype positive is natural range 1..natural'Last;
X: integer; Y: natural; Z : positive;
```

Y und Z werden hierbei auch als Variable des Typs integer angesehen, sind also polymorph.


```
type Vektor is array (integer range <>) of float;

procedure etwas (X, Y: in out Vektor) is
Summe: Vektor;
begin  ...  for J in Y'Range loop Summe(J) := ...  end loop;
       ...
end etwas;
```

In diesem Beispiel sind X und Y Variablen eines Datentyps, dessen Größe (in Form der Indexgrenzen) unbekannt ist. Der Datentyp von X und Y steht also erst nach der Auswertung der zugehörigen aktuellen Parameter fest. Etwas Ähnliches gilt für die lokale Variable Summe. Die Variablen X, Y und Summe kann man daher als polymorph auffassen.

In der objektorientierten Programmierung spielt diese Art der Polymorphie ein wichtige Rolle.

3. Parametrisierung von ***Typen*** eines Programms. Betrachte zum Beispiel Polynome:

Ein Polynom ist eine Abbildung p: M → M der Form
$p(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + a_1 \cdot x + a_0$,
wobei auf M eine Addition und eine Multiplikation definiert sein muss. Mathematisch ist M in der Regel ein "Ring", z.B. $\mathbb{N}_0$, $\mathbb{Z}$, $\mathbb{Q}$ oder $\mathbb{R}$. Weiterhin muss $a_n \neq 0$ sein (Ausnahme: das Null-Polynom $p(x) = 0$ für alle x); n heißt der Grad von p.

Der Typ "Polynom" hat also die Parameter n und M:

```
array [0..n] of M.
```

Schreibweise (mit subtype exponent is integer range 0..max):

```
type polynom (n: exponent; type ring) is
    record A: array (0..n) of ring end record;
```


Diesen parametrisierten Typ kann man dann im Programm benutzen, um konkrete Variablen zu deklarieren:

```
P : polynom(n =>4, ring => float);     oder mit Initialisierung:

Q : polynom(4, integer) := (2, 0, 6, 5, 3);
```

Dies bezeichnet das Polynom (A[0] sei 2 usw.):
$q(x) = 3 \cdot x^4 + 5 \cdot x^3 + 6 \cdot x^2 + 2$.

In Ada sind parametrisierte Typen erlaubt, allerdings sind hierbei keine Typen als Parameter vorgesehen:

```
    type polynom (n: maxexp) is
    record A: array (0..n) of integer; end record;
Q: polynom := (4, (2, 0, 6, 5, 3));
```

Typen als Parameter in anderen Typen kann man in Ada leicht durch parametrisierte Pakete realisieren.

4. Ein ***Unterprogramm*** oder ein ***Paket*** heißt polymorph, wenn mindestens eine Spezifikation eines Datentyps oder wenn der Datentyp eines formalen Parameters oder des Ergebnisses polymorph ist. Bekannte Beispiele für solche Polymorphie sind Sortierverfahren, Integrale usw.

Diese Form von vielfacher Verwendbarkeit von Programmeinheiten (vor allem Unterprogramme und Moduln) bezeichnet man als Generizität. In Ada legt man hiermit ein Schema an, das spätestens bei der Deklaration zugehöriger Variablen durch konkrete Datentypen und Konstanten ausgefüllt werden muss. Das Schlüsselwort ist "generic", hinter dem die Typen und ggf. zusätzliche Funktionen auflistet sind; anschließend folgt dann die jeweilige Programmeinheit.

Ein einfaches Beispiel ist das Vertauschen (1.1.2.5). Man könnte hier schreiben (über die Schreibweise kann man sich natürlich streiten):

```
procedure Tausch (type T; A, B: reference T) is
H: T;
begin  H:=A; A:=B; B:=H  end;
```

und im Falle

```
X, Y: character;
```

könnte man dann die Inhalte von X und Y vertauschen:

```
Tausch (character, X, Y);
```

In der Programmierung ist ein beliebiger Polymorphismus zwar erwünscht, aber er ist schwer zu implementieren. Daher sollte man sich auf den Polymorphismus beschränken, der möglichst "zur Compilezeit" bereits beseitigt werden kann.

Statt Tausch (character, X, Y); sollte man daher eine für Character zugeschnittene Prozedur aus dem "Prozedurschema" Tausch erzeugen und hiermit weiterarbeiten:

```
procedure TauschChar (A, B: reference character) is
    new Tausch (T => character; A, B: reference T);
...
TauschChar (X, Y);
```

Ada arbeitet in dieser Weise.

1.4.4.3: Generizität in Ada

Unter Generizität versteht man vor allem den obigen Punkt 4: die Parametrisierung von Programmeinheiten mit Datentypen, Unterprogrammen und Moduln. Beispiel:

Die Datenstruktur "Liste" lässt sich unabhängig von den in ihr verwalteten Daten erklären und sollte sich in einer allgemeinen Form als Bibliotheks-Modul vorhanden sein. Ein Programm sollte seine konkreten Listen zu Beginn hieraus ableiten, indem der Modul importiert ("with") und durch den aktuell benötigten Datentyp konkretisiert wird.

Wir erläutern dies an Ada-Beispielen und beginnen mit dem Vertauschen zweier Inhalte.

Der Spezifikationsteil und der Implementierungsteil müssen bei generischen Einheiten in Ada prinzipiell getrennt angegeben werden. Der variabel gehaltene Bereich wird mit "generic" eingeleitet:

```
generic
    type element is private;
procedure Tausch (A, B: in out element);

procedure Tausch (A, B: in out element) is
H: element;
begin  H:=A; A:=B; B:=H;  end Tausch;
```


Bei folgender Deklaration

```
X, Y: integer; C, D: Boolean;
```

kann man die generische Prozedur Tausch dann wie folgt konkretisieren:

```
procedure BoolTausch is new Tausch (Boolean);
procedure IntTausch is new Tausch (integer);
... IntTausch(X, Y); ... BoolTausch(C, D); ...
```

Wegen des Überladens kann man in Ada auch schreiben:

```
procedure Austausch is new Tausch (Boolean);
procedure Austausch is new Tausch (integer);
... Austausch (X, Y); ... Austausch (C, D); ...
```

Wir betrachten nun ein anderes Beispiel in Ada mit einem Datentyp "item", auf dem eine Ordnung ">" definiert sein möge. Wir sortieren ein Feld aus max Komponenten mittels Bubble Sort:

1.4.4.4: Sortieren (leider kein ganz korrektes Ada, siehe später):

```
generic
   max: Positive;
   type T is private;
   type VektorT is array (1..max) of T;
procedure SORT (A: in out VektorT);

procedure SORT (A: in out VektorT) is
procedure Austausch is new Tausch (T);
weiter: Boolean := true;
begin
while weiter loop  weiter := false;
   for I in 1..max-1 loop
   if A(I) > A(I+1)
      then weiter := true; Austausch(A(I), A(I+1)); end if;
   end loop; end loop;
end SORT;
```

Nicht erlaubt

Wir machen hieraus ein neues Paket.

Korrekt, aber wir definieren es hier nochmals lokal.

Dieser Operator ist bekannt zu machen

Hinweise: Diese Formulierungen sind in Ada nicht zulässig, da zum einen der Wert von Max nicht innerhalb derselben Deklaration bereits verwendet werden darf und da zum anderen der Operator ">" bekannt gegeben werden muss. Letzteres geschieht durch Angabe mit "with"; dieser Operator muss später für den konkreten Datentyp, der T ersetzt, angegeben werden.
Wir modifizieren also die Sortierprozedur:

```
generic
   Max: Positive;
   type T is private;
   with function ">"(L,R:T) return Boolean;
package Sortpaket is
   type VektorT is array (1 .. Max) of T;
   procedure Sort (A: in out VektorT);
end Sortpaket;
```

```
package body Sortpaket is
  generic   type Element is private;
  procedure Tausch (A, B: in out Element);
  procedure Tausch (A, B: in out Element) is
    H: Element; begin  H:=A; A:=B; B:=H; end Tausch;
  procedure Sort(A: in out VektorT) is
    Weiter: Boolean := True;
    procedure Austausch is new Tausch(T);
  begin
    while Weiter loop
      Weiter := False;
      for I in 1..Max-1 loop
        if A(I)>A(I+1) then Weiter := True; Austausch(A(I), A(I+1));
        end if;
      end loop;
    end loop;
  end Sort;
end Sortpaket;
```


Verwendung im weiteren Verlauf des Programms (dies wurde nicht verifiziert, bitte selbst in einem Programm prüfen!):

```
        -- Im Deklarationsteil:
with Sortpaket; with Ada.Integer_Text_IO;
package SortpaketInt is new Sortpaket(500, integer, ">");
R: SortpaketInt.VektorT;
        -- Im Anweisungsteil:
...                     -- Fülle das Feld R mit ganzen Zahlen
SortpaketInt.Sort(R);
...
```

Hinweis: Wenn Sie im package SortpaketInt die Operation ">" durch "<" ersetzen, so wird absteigend sortiert!

Hinweise zur Syntax für generische Unterprogramme und Pakete:

```
generic_declaration ::=  generic_subprogram_declaration |
                         generic_package_declaration

generic_subprogram_declaration ::=
        generic_formal_part  subprogram_specification ;

generic_package_declaration ::=
        generic_formal_part  package_specification ;

generic_formal_part ::=
    generic {generic_formal_parameter_declaration | use_clause}
```



```
generic_formal_parameter_declaration ::=
    formal_object_declaration  |
    formal_type_declaration   |
    formal_subprogram_declaration  |
    formal_package_declaration

formal_subprogram_declaration ::=
    with subprogram_specification [is subprogram_default] ;

formal_package_declaration ::=
    with package defining_identifier is new
    generic_package_name  formal_package_actual_part ;

formal_type_declaration ::=
    type defining_identifier[discriminant_part] is
                                    formal_type_definition ;

formal_package_actual_part ::= (<>) | [generic_actual_part]
```

Anderes Beispiel: Wir erweitern das Standardbeispiel 1.4.3.4 (Stack):

```
generic   type item is private;
package  Stack  is
type StZelle;
type RefStZelle is access StZelle;
type StZelle is  record inhalt: item; next: RefStZelle; end record;
procedure Push(A: in item; S: in out RefStZelle);
procedure Pop(S: in out RefStZelle);
function Top(S: in RefStZelle) return character;
function Isempty(S: in RefStZelle) return Boolean;
end Stack;

package body  Stack  is
  procedure Push (A: in item; S: in out RefStZelle) is
      begin  S := new RefStZelle'(A,S);  end;
  ...
end Stack;
```


Verwendung dieses Pakets:

```
package Zeichenstack is new stack(character);

A: character;
X: Zeichenstack.RefStZelle;

use Zeichenstack;

begin   ...
        Push(A,X);  ...
        if isempty(X) then Push('C',X);
        else Pop (X); end if;
        ...
end;
```