Gliederung des Kapitels 1.4

1.4/1.5 Programmierung (und die Sprache Ada 95)

1.4.3 Moduln

Ein Modul ist eine in sich abgeschlossene Programmeinheit, die aus Konstanten, Variablen, Datentypen und Algorithmen besteht. Moduln bilden die Bausteine, aus denen ein großes Softwaresystem zusammengesetzt wird. Man spricht dann von einem "modularen" Aufbau solcher Systeme.

Zur Schreib- und Sprechweise:

Wir sagen hier "der Módul" mit Betonung auf der ersten Silbe und dem Plural "die Móduln". Unter "das Modúl" mit Betonung auf der zweiten Silbe und dem Plural "die Modúle" verstehen wir (leicht austauschbare) physikalische Einheiten, insbesondere Hardwarebausteine.

Englisch: module.

1.4.3.1: Ein Modul sollte folgende Eigenschaften besitzen:

- Er bildet eine in sich abgeschlossene Einheit, die eine klar umrissene Aufgabe bearbeitet.
- Er hat eine genau definierte Schnittstelle nach außen (genannt "Spezifikation" oder "Interface"); nur die hier genannten Eigenschaften und Fähigkeiten sind nach außen hin sichtbar ("visibility"). Dies ist eine Art Benutzungsanweisung für alle, die diesen Modul einsetzen wollen.
- Seine Arbeitsweise ("Implementation") ist außen nicht bekannt. Er besitzt somit zwei Sichten (views): Die Außenansicht für den Benutzer (dies ist die Schnittstelle) und die Innensicht des Erstellers. Die Innensicht bleibt "gekapselt" oder nach außen "versteckt" (Prinzip des "information hiding").
- Er ist überschaubar, leicht zu testen und einfach zu warten.
- Er lässt sich in Bibliotheken aufbewahren und hierdurch leicht in beliebige Programmsysteme einbauen.

Moduln sind somit die einfachste Art, ein "Client-Server"-Modell zu realisieren:

Hierbei erstellt ein "Server" (Dienstleister, Hersteller) einen "Dienst", den er als Angebot (= Schnittstelle) nach außen bekannt macht. Wie er diese Dienstleistung tatsächlich realisiert, wird dabei nicht bekannt gegeben.

Ein "Client" (Kunde, Benutzer) benötigt eine Dienstleistung und verlässt sich darauf, dass die im Angebot eines Servers genannten Eigenschaften auch zutreffen.

Um dies Modell in die Praxis umzusetzen, benötigt man ein Netz (Internet), in dem ein Kunde die Angebote mit Hilfe von Suchmaschinen auffinden und sich mit Browsern anzeigen lassen kann. (Der Kunde "browst" und "surft" im Internet, siehe Kapitel 2.)

1.4.3.2: schematischer Aufbau eines Moduls

module <Name des Moduls> is

[with ...; use ...]	Angaben, welche anderen Moduln verwendet werden und in welcher Weise
specification ...	Angabe der nach außen sichtbaren Datentypen, Konstanten, Variablen und "Methoden" (= Funktionen, Operatoren, Unterprogramme, Unter-Module usw.)
[implementation ...]	Weitere (nach außen nicht sichtbare) Deklarationen und Programme zur Implementierung der Methoden
[begin ... end]	Anweisungen zur Initialisierung

end module [<Name des Moduls>]

1.4.3.3: Standardbeispiel "Stack" (für Zeichen)

Erinnerung: Definition 1.3.3.2: Eine lineare Liste heißt Keller oder Stapel (engl.: stack oder pushdown), wenn auf ihr genau die folgenden fünf Operationen zugelassen sind:

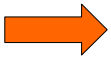
(1) "empty" = Leeren der Liste.
(2) "isempty" = Abfragen auf Leerheit der Liste.
(3) "top" = Kopieren des letzten Elements der Liste.
(4) "push" = Hinzufügen eines Elements am Ende der Liste.
(5) "pop" = Löschen des letzten Elements der Liste.

Hierin ist für einen "Kunden" bereits zu viel Information enthalten, insbesondere muss sich der Ersteller nicht auf den Begriff "Liste" festlegen. Auch braucht ein Kunde in der Regel nicht die Initialisierung "empty". Wir bieten daher an:

Angebot: Datenstruktur "Stack für Zeichen"

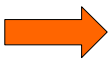
Fähigkeiten (Leistungsumfang) dieses Angebots:

Datentyp StackZ;
Push (Zeichen) aktualisiert StackZ;
Pop aktualisiert StackZ;
Top liefert als Ergebnis ein Zeichen;
Isempty liefert als Ergebnis Boolean;



specification

Umgangssprachliche Erläuterungen ("Pflichtenheft"):
Push fügt ein Zeichen an den Stack an,
Pop entfernt das zuletzt eingefügte Zeichen,
Top zeigt das zuletzt eingefügte Zeichen an,
Isempty prüft, ob kein Zeichen im Stack ist.



im Kommentarteil von specification

Wie sieht dies im Internet-Browser aus?

Die Datenstruktur AG

Die Firma
Weitere Produkte
Einsatzgebiete
StackZ-Bereiche
Leistungsumfang
Impressum
Kontakt

StackZ

Mitgeliefert werden
Push, **Pop**, **Top** und **Isempty**!
Auf Wunsch **Empty**

in den Einkaufswagen

Werden Sie bevorzugter Kunde bei uns! Betreuung, Rabatte, Sonderangebote

Was reingeht, kommt auch wieder raus!

Jetzt bestellen später zahlen

Neue Einsatzmöglich-keiten des StackZ hier

Fügen Sie mit *Push* ein Zeichen an!
Pop entfernt das letzte Zeichen!
Einmalig: *Sie können das letzte Zeichen sehen oder sogar feststellen, dass nichts mehr da ist!*

Preise, Wartung und Nachlieferungen hier

Lizenz- und Liefer-bedingungen hier

Nur zufriedene Kunden!
Referenzen hier.

Bestens bewährt, basiert auf dem LIFO-Prinzip.
Von Fachleuten empfohlen!

Einkauf fortsetzen

Kennen Sie schon QueueZ? hier ansehen

Formulierung als Modul

```
module  Stack_für_Zeichen  is
with IO; use IO        -- IO sei ein Modul, der die Ein-/ Ausgabe
                       -- von Zeichen und Texten realisiert.
specification
  type StZelle;
  type StackZ is access StZelle;
  type StZelle is  record
        inhalt: character;
        next: StackZ
  end record;
  procedure Push (value character);
  procedure Pop;
  function Top result character;
  function Isempty result Boolean;
```



```
implementation          -- vor allem: "Methoden" programmieren
   S: StackZ;
   procedure Push (A: value character) is
   p: StackZ;
   begin p := new StackZ; p.inhalt := A; p.next := S; S := p end;
   procedure Pop is
   begin if S ≠ nil then S := S.next
         else write ("Stackunterlauf"); raise Stack_Error fi  end;
   function Top result character is
   begin if S ≠ nil then result S.inhalt
         else write ("Stackunterlauf"); raise Stack_Error fi  end;
   function Isempty result Boolean is
   begin result (S = nil) end;
   procedure Empty is begin S := nil end;
begin Empty end
end module Stack_für_Zeichen;
```

Dieser Modul lässt sich nun in einem Programm wie folgt verwenden:

```
declare
        X: character;
        module  Stack_für_Zeichen  is  ...  end module;
        ...
begin ...
        Stack_für_Zeichen.Push (X);
        ...
        if not Stack_für_Zeichen.Isempty
              then  X := Stack_für_Zeichen.Top;
                    Stack_für_Zeichen.Pop  fi;
        ...
end;
```


Dieses Vorgehen eignet sich nicht, wenn man mehrere Stacks anlegen will. In diesem Fall vereinbart man den Datentyp StackZ und parametrisiert die Prozeduren.

```
module  Stacks_für_Zeichen  is
with IO; use IO;
specification
   type StZelle;
   type StackZ is access StZelle;
   type StZelle is  record
          inhalt: character;
          next: StackZ
   end record;
   procedure Push (value character, reference StackZ);
   procedure Pop (reference StackZ);
   function Top (value StackZ) result character;
   function Isempty (value StackZ) result Boolean;
```

```
implementation          -- vor allem: "Methoden" programmieren
   procedure Push (A: value character; S: reference StackZ) is
   p: StackZ;
   begin p := new StackZ; p.inhalt := A; p.next := S; S := p end;
   procedure Pop (S: reference StackZ) is
   begin if S ≠ nil then S := S.next
         else write ("Stackunterlauf"); raise Stack_Error fi  end;
   function Top (S: reference StackZ) result character is
   begin if S ≠ nil then result S.inhalt
         else write ("Stackunterlauf"); raise Stack_Error fi  end;
   function Isempty (S: reference StackZ) result Boolean is
   begin result (S = nil) end;
   procedure Empty (S: reference StackZ) is begin S := nil end;
end module Stacks_für_Zeichen;
```


Solch ein Modul lässt sich nun wie folgt verwenden:

```
declare
      X: character;
      module  Stacks_für_Zeichen  is  ...  end module;
      S1, S2, S3: StackZ;
      ...
begin ...
      Stacks_für_Zeichen.Push(X, S1);
      ...; S2 := S3; ...
      if not Stacks_für_Zeichen.Isempty(S2)
            then  X := Stacks_für_Zeichen.Top(S2);
                  Stacks_für_Zeichen.Pop(S2)  fi;
      ...
end;
```

Hier ist der Datentyp StackZ bekannt, da die Modul-Deklaration abgearbeitet wurde.

Stacks_für_Zeichen.Empty(S1) ist nicht erlaubt, da diese Prozedur nicht in der Spezifikation aufgeführt ist.

In diesem Beispiel haben wir den Datentyp StackZ nach außen bekannt gemacht. Dies ist aber nicht nötig. Der Ersteller des Moduls könnte den Stack auch durch ein array [1..flex] of character implementieren, wobei eine Indexvariable sich die Stelle merkt, wo das oberste (= zuletzt eingefügte) Zeichen steht.

Will man die Implementierung des Datentyps offen lassen, muss man seine Definition vor dem Benutzer verstecken. Man würde dann unter "specification" nur schreiben:

```
type StackZ
```

oder sogar diese Information weglassen und die Deklaration vollständig in den Implementierungsteil verschieben.

Datentypen, die man dem Benutzer zwar bekannt gibt, aber dessen genaue Definition man ihm nicht mitteilt, bezeichnet man als "private Typen" des Moduls.

with und use können in allen Deklarationsteilen stehen.

Erläuterung des Sprachelements with:
with M bedeutet, dass an dieser Stelle die Deklaration des Moduls M hineinkopiert wird. Alles,was in dessen Spezifikation steht, kann ab hier über die Punktnotation "M. ---" benutzt werden.

Erläuterung des Sprachelements use:
use M bedeutet, dass ab dieser Stelle für einen Namen Funk, der in M vereinbart wird, die Punktnotation "M.Funk" durch Funk ersetzt werden kann, dass also die "Qualifizierung" (= der Zugriff auf die in M deklarierten Größen) ohne "M." erfolgen darf. Für Namenskonflikte ist der Programmierer selbst verantwortlich, wobei das Überladen erlaubt ist.

1.4.3.4 Moduln in Ada ("Pakete")

Das Sprachelement lautet hier package und man spricht von Paketen statt von Moduln.

Der Spezifikationsteil und der Implementierungsteil werden voneinander getrennt. Der Spezifikationsteil, der früher als der Implementierungsteil im Programm stehen muss, hat die Form

package <Name des Pakets> is
<Folge von einfachen Deklarationen> end <Name des Pakets>;

Der Implementierungsteil heißt "body" und hat die Form

package body <Name des Pakets> is
<Folge von Deklarationen> end <Name des Pakets>;

Alle Teile der Spezifikation, die im Body programmiert werden, müssen wörtlich dort wieder vorkommen, s.u.

Besteht die Spezifikation nur aus Datentyp- und Konstanten-Deklarationen, so entfällt der Implementierungsteil.

Die Deklaration privater Typen erfolgt im Spezifikationsteil als "is private"; die Struktur wird anschließend durch das gleiche Schlüsselwort private vor dem Benutzer versteckt.

In Ada sind mit einem Datentyp (auch einem private-Datentyp) stets folgende Operationen verbunden:

- Gleichheit ("="),
- Ungleichheit ("/="),
- Zuweisung (":=").

Will man auch diese Operationen nicht für einen Datentyp des Moduls zulassen, so muss man ihn als limited private in der Spezifikation deklarieren.

Das Standardbeispiel 1.4.3.3 StackZ lässt sich leicht nach Ada übertragen (Stack_Error sei als exception hier sichtbar):

```
package  SfZ  is
with Ada.Text_IO; use Ada.Text_IO;
type StZelle;
type StackZ is access StZelle;
type StZelle is  record
                 inhalt: character;
                 next: StackZ;
                 end record;
procedure Push (A: in character; S: in out StackZ);
procedure Pop (S: in out StackZ);
function Top (S: in StackZ) return character;
function Isempty (S: in StackZ) return Boolean;
end SfZ;
```



```
package body  SfZ  is
   procedure Push (A: in character; S: in out StackZ) is
   p: StackZ;
   begin p := new StackZ; p.inhalt := A; p.next := S; S := p; end;
   procedure Pop (S: in out StackZ) is
   begin  if S /= null then S := S.next;
        else put ("Stackunterlauf"); raise Stack_Error; end if;  end;
   function Top (S: in StackZ) return character is
   begin  if S /= null then return S.inhalt;
        else put ("Stackunterlauf"); raise Stack_Error; end if;  end;
   function Isempty (S: in StackZ) return Boolean is
   begin return (S = null); end;
end SfZ;
```

Dieser Modul lässt sich nun in einem Ada-Programm zum Beispiel wie folgt verwenden:

```
declare
        X: character;
        package  SfZ  is  ...  end SfZ;
        S1, S2: StackZ;  -- S1 und S2 werden in Ada mit null ini-
                         -- tialisiert, da StackZ ein access-Typ ist
begin   ...
        SfZ.Push (X, S1);
        S2 := S1;
        if S1 = S2 then ... end if;
        ...
        if not SfZ.Isempty (S2)
            then S1.inhalt := SfZ.Top (S2); SfZ.Pop (S2); end if;
        ...
end;
```


Hinweis:

Die Prozedur

```
procedure Push (A: in character; S: in out StackZ) is
p: StackZ;
begin p := new StackZ; p.inhalt := A; p.next := S; S := p; end;
```

lässt sich in Ada wesentlich kürzer mit Hilfe der Initialisierung beim Anlegen eines neuen StackZ-Elements schreiben:

```
procedure Push (A: in character; S: in out StackZ) is
begin  S := new StackZ'(A,S);  end;
```

Die Initialisierung erfolgt durch '(...). Rechts vom Zuweisungszeichen wird die next-Komponente des neuen Elements auf den alten Wert von S gesetzt, die Komponente "inhalt" erhält den Wert von A und durch die Wertzuweisung wird dieses neue Element anschließend das oberste Stackelement.

Man kann die Definition des Datentyps StackZ verbergen:

```
package  SfZ  is
with Ada.Text_IO; use Ada.Text_IO;
type StackZ is private;
procedure Push(A: in character; S: in out StackZ);
procedure Pop(S: in out StackZ);
function Top(S: in StackZ) return character;
function Isempty(S: in StackZ) return Boolean;
private
    type StZelle;
    type StackZ is access StZelle;
    type StZelle is   record
                      inhalt: character;
                      next: StackZ;
                      end record;
end SfZ;
```


Man braucht den Implementierungsteil (package body) hierbei nicht zu ändern, auch das sonstige Programm nicht, sofern von dort nicht auf die einzelnen Komponenten von S1, S2, ... zugegriffen wird.

Außerhalb des Implementierungsteils sind dann Zugriffe wie

S1.inhalt oder S2.next

nicht zugelassen, allerdings sind S1 = S2 oder S1 := S2 noch erlaubt.

Schreibt man statt `type StackZ is private;`
die Zeile `type StackZ is limited private;`
so sind außerhalb des Implementierungsteils auch keine Vergleiche auf Gleicheit und Ungleichheit sowie Zuweisungen an Variablen vom Typ StackZ mehr zugelassen. Variablen des Datentyps können dann nur noch mit Hilfe der Operationen des Pakets manipuliert, gespeichert und ausgegeben werden.

1.4.3.5: Modul besitzen "Zustände". Hierunter versteht man die aktuellen Werte von veränderlichen Informationen; in der Programmierung ist dies die aktuelle Menge der Werte aller Programmvariablen sowie die Position, an der man sich im Programm befindet. Ein Zustand ist bei Programmen also im Wesentlichen der aktuelle Speicherinhalt.

Moduln sind (beliebig große) Einheiten, die zum einen Speicherstrukturen (in Form von Datentypen und Variablen) mit Algorithmen ("Methoden") zur Manipulation ihrer Zustände anbieten und zum anderen ihren Zustand zwischen zwei Benutzungen nicht verändern.

Vgl. Folien 117/118: Der dortige Modul Stack_für_Zeichen besitzt Zustände (= Speicherinhalte der Variablen S). Dagegen bietet der Modul Stacks_für _Zeichen auf den Folien 120/121 nur Datentypen und Methoden an.

Unterprogramme unterliegen dem Keller-Prinzip und werden im lokalen Speicher des Programms nach dem LIFO-Prinzip verwaltet; verlässt man sie, so endet auch die Lebensdauer aller lokalen Variablen.

Dagegen bleiben die Inhalte der Variablen eines Moduls erhalten, wenn man ihn verlässt. Verwendet man den Modul wieder, so besitzt er anfangs den gleichen Zustand, in dem er das letzte Mal verlassen worden ist.

Moduln geben ihren Zustand und ihre Arbeitsweise nach außen nur zum Teil bekannt (Prinzip des "information hiding"). Es gibt daher zwei Sichten: die in der Spezifikation festgelegte Außenansicht (partial view) für den Benutzer/ Kunden und die durch die Implementierung gegebene Innenansicht (full view) des Erstellers/Anbieters.