

Programmierrichtlinie

für die Programmierübungen WS 2007/2008

21. Oktober 2007, Version 499

1 Einleitung

In Wirtschaft und Forschung sind fast immer mehrere Personen mit der Bearbeitung von Programm-Quelltexten beschäftigt. Die Quelltexte müssen für verschiedene Personen mit unterschiedlichem Kenntnisstand verständlich und änderbar sein. Deshalb müssen Quelltexte, die von verschiedenen Personen erstellt wurden trotzdem einheitlich strukturiert und ausreichend kommentiert sein. Weiterhin sollen Normen befolgt werden, damit Programme mit ähnlicher Funktionalität auch ähnlich aussehen. Der Programmierer als einzelner muss sich den Standards des Teams unterordnen.

2 Formatierung

Diese Richtlinie gibt Hinweise, wie Ada-Quelltexte, die im Rahmen der Programmierübungen erstellt werden, aussehen sollen. Sie orientiert sich an den Hinweisen im Ada Reference Manual sowie dem Regelwerk, das von dem GNAT-Compiler sowie von vielen Editoren unterstützt wird.

Diese Richtlinie ist keinesfalls vollständig und es gibt viele Programmkonstrukte in Ada, die in der Richtlinie nicht einmal erwähnt werden. Im Zweifelsfall soll anhand von ähnlichen Konstrukten aus Beispiel-Quelltext entschieden werden.

2.1 Dateien

Der Quelltext wird gemäß der GNAT-Konventionen auf (möglicherweise mehrere) Dateien verteilt. Jede Datei enthält genau eine Library_Unit. Der Dateiname ist ihr vollständig expandierter Name ausschließlich in Kleinbuchstaben, wobei Punkte durch Minuszeichen ersetzt werden. Deklarationen erhalten die Dateiendung „.ads“, Implementierungen die Endung „.adb“.

Beispiele:

```
--_File:_parent-child.ads
package_Parent.Child_ is
end_Parent.Child;
```

```
--_File:_p.adb
procedure_P
is
begin
  _null;
end_P;
```

2.2 Ada-Konstrukte

Dieser Abschnitt gibt Hinweise, welche Konstrukte in Programmen wie verwendet werden sollten.

- Unterprogramme müssen kurz und übersichtlich sein. Besonders bei komplizierten Algorithmen sollte darauf geachtet werden, dass kleine überschaubare Einheiten gebildet werden. Eine maximale Länge von 50 Zeilen sollte nicht überschritten werden.
- Ein Unterprogramm sollte genau eine Aufgabe erledigen. Solche Unterprogramme sind übersichtlicher und können leichter auf geänderte Anforderungen angepasst werden.
- Es dürfen keine Pakete innerhalb von Unterprogrammen geschachtelt werden.
- Use clauses dürfen nur in lokal begrenztem Kontext eingesetzt werden. Use clauses innerhalb der Context clause, die für ein ganzes Paket gelten, sind nicht zulässig. Besonders in Quelltexten mit vielen **withs** würde es damit für den Leser schwieriger zu entscheiden von woher Deklarationen verwendet werden.
- Die Bedingungen in If-Statements sollen in der Regel positiv ausgedrückt werden, um das Verständnis zu erleichtern.

```
if A >= 7 then -- ok!  
    ...  
end if;  
if not (A < 7) then -- schwieriger zu lesen  
    ...  
end if;
```

2.3 Bezeichner

Neue Bezeichner werden stets konsistent gebildet. Bezeichner sollten entweder in deutscher oder in englischer Sprache gewählt werden (nicht gemischt!) und ihren Zweck im Programm möglichst präzise beschreiben. Bezeichner können aus mehreren zusammengesetzten Wörtern bestehen, die durch Unterstrich verbunden werden. Jedes dieser Wörter wird groß geschrieben (nur der erste Buchstabe). Gängige Abkürzungen dürfen verwendet werden. In Abkürzungen können auch mehr als nur der erste Buchstabe eines Worts groß geschrieben werden (z.B. GmbH_Name).

Bezeichner müssen so gewählt werden, dass ihre Funktion ausreichend ersichtlich wird, sollten jedoch trotzdem möglichst kurz sein. Eine Länge von 30 Zeichen sollte nicht überschritten werden. Bezeichner von Prozeduren sollten mit Verben beginnen, Bezeichner von Funktionen mit Verben oder Substantiven, Bezeichner von Paketen mit Substantiven.

Beispiele:

```
procedure Accelerate_Car;
```

```
function Size  
    (Container: Stack)  
    return Natural;
```

```
package Ada.Command_Line;
```

```
type Traffic_Light_Colors is  
    (Red, Yellow, Green);
```

```
Color: Traffic_Light_Colors;
```

2.4 Quelltext-Layout

Das Layout der Quelltexte wird mit dem Übersetzer gnatmake überprüft. Der Schalter `-gnaty<Regeln>` wird verwendet, um zu überprüfende Regeln einzuschalten. Für diese Richtlinie wurden die folgenden Regeln als verbindlich ausgewählt: `-gnaty3acefhiklmnrpt`. Quelltext sollte stets mit diesem Schalter übersetzt werden. Es darf gegen keine dieser Regeln verstoßen werden.

- 3 Die normale Einrückungstiefe beträgt 3 Leerzeichen. Lange Zeilen (siehe 2.4) werden umgebrochen und der umgebrochene Teil wird 2 Leerzeichen eingerückt bzw. an dem vorangegangenen Ausdruck ausgerichtet.

```
Value_ :=
  _ Calculation_ ( First ,
  _ Second_ * _ Third );
```

- a Die Bezeichner von Attributen (z.B. Character'Pos, Integer'Image) werden groß geschrieben.
- c – Kommentare, die die ganze Zeile einnehmen, beginnen in der gleichen Spalte wie die darüberstehende Anweisung bzw. Deklaration. Auf das „--“ folgen zwei Leerzeichen vor dem Kommentar-Text.

```
Value_ : Integer ;
-- Here is an example .
Example_ : Character ;
```

- Box-Kommentare werden wie folgt formatiert:

```
-----
-- This is a box --
-----
```

- Kommentare, die in der selben Zeile nach dem Quelltext folgen, sollten nur in seltenen Ausnahmefällen verwendet werden, z.B. um auf ansonsten überraschende Effekte hinzuweisen. Es ist besser, überraschende Effekte ganz zu vermeiden:

```
X_ := Y ; -- This creates an alias !
```

- e Nach „end“ von Unterprogrammen, und Paket-Deklarationen und -Implementierungen muss der Name wiederholt werden. Gleiches gilt für „exit“, falls dieses innerhalb einer benannten Schleife steht.

```
procedure _Name
is
begin
  _ null ;
end _Name ;
```

```
Enumeration :
loop
  _ I_ := I_ + 1 ;
  _ exit _Enumeration_ when I_ = 17 ;
end _loop_Enumeration ;
```

- f,h Die Zeichen <Form Feed>, <Horizontal Tab> und <Vertical Tab> sind in Quelltexten nicht erlaubt. Tabulatoren eignen sich im Prinzip als Zeilen-Einrückung, da jedoch die Quelltext-Darstellung stark von der eingestellten Länge der Tabulatoren abhängt und auf verschiedenen Systemen für verschiedene Benutzer häufig zu Ärgernissen führt, wird auf Tabulatoren verzichtet.

i In einem If-Statement muss das **then** in der gleichen Zeile stehen. Ist die Bedingung zu lang um in eine Zeile zu passen, so wird wie folgt umgebrochen:

```
if_One_Line_Expression_then
```

```
...  
end_if ;
```

```
if_Multi_Line_Expression  
(Parameter_1 , Parameter_2)  
then
```

```
...  
end_if ;
```

```
if  
Is_Complex_Expression_True  
then
```

```
...  
end_if ;
```

k Reservierte Wörter werden ausschließlich in Kleinbuchstaben geschrieben.

l Das Layout von Anweisungen und Deklarationen soll sich an den Vorschlägen des Ada Standards orientieren. Dinge, die innerhalb von umschließenden Strukturen stehen, werden eingerückt. Beispiele:

```
type_R_is  
record  
    Count : Integer ;  
    Char   : Character ;  
end_record ;  
begin  
    while_not_Done_loop  
        Work ;  
    end_loop ;  
end ;
```

m Zeilen dürfen maximal 79 Zeichen lang sein. Durch diese Einschränkung wird eine überschaubare Darstellung des Quelltexts ermöglicht und das Lesen sowie Drucken erleichtert.

n,r,p Verwendungen von Bezeichnern müssen genau so geschrieben werden wie deren Deklaration. Groß-/Kleinschreibung darf nicht variiert werden.

```
ada.Text_IO.New_Line ; --schlecht!  
Ada.Text_Io.New_Line ; --schlecht!  
Ada.Text_IO.New_Line ; --richtig!
```

t An bestimmten Stellen im Quelltext müssen (für den Compiler eigentlich überflüssige) Leerzeichen eingefügt werden, um die Lesbarkeit zu erhöhen):

– Vor und nach => müssen Leerzeichen stehen.

```
Sort_( Field => My_Field );
```

– Vor einer Box <> steht ein Leerzeichen oder eine öffnende Klammer.

```
type_Natural_Array_is_array_( Integer_range <> ) of_Natural ;
```

```
generic
```

```

type Enum is (<>);
procedure Put
  (Item: in Enum);

```

- Auf die Schlüsselwörter **abs** und **not** muss ein Leerzeichen folgen

```

if not (A and B) then
  null;
end if;

```

- Auf die unären Operatoren + und - darf kein Leerzeichen folgen

```

-17 + -B - A

```

- Vor und nach allen binären Operatoren stehen Leerzeichen. Eine Ausnahme ist der Exponentiations-Operator, dieser steht ohne Leerzeichen, z.B. (A*_B+_X**15)_(A+_3)**3

- Vor und nach Doppelpunkten stehen Leerzeichen.

- Vor und nach := stehen Leerzeichen

```

A := B;

```

- Vor der schließenden Klammer steht kein Leerzeichen. Schließende Klammern sollten am Zeilenende und nicht allein in einer Zeile stehen.

```

Complex_Procedure_Call
  ( First => 17,
    Second => 20,
    Third => 95);

```

- Vor einem Semikolon steht kein Leerzeichen.

- Vor und nach | stehen Leerzeichen

```

case A is
  when 15 | 17 =>
    null;
  when others =>
    null;
end case;

```

- (ohne automatische Prüfung) Zusätzlich ist zu beachten, dass ineinander geschachtelte Kontrollstrukturen die Lesbarkeit des Quelltexts stark verschlechtern. Tief geschachtelte Programmteile sollten durch Verwendung von Unterprogrammen umstrukturiert werden.

3 Kommentierung

Quelltexte müssen mit Kommentaren dokumentiert werden. Kommentare enthalten *keine Umlaute oder Sonderzeichen*. Kommentare werden entweder in deutscher oder in englischer Sprache verfasst, jedoch nicht gemischt. Kommentare sollen kurz aber präzise die Information bereitstellen, die der Leser benötigt um den Quelltext zu verstehen. Die Kommentare sollen keine trivialen Informationen vermitteln, sondern die Lösungsidee für das Problem erläutern oder auf Besonderheiten der Implementierung hinweisen. In längeren zusammenhängenden Quelltext-Passagen sollen Kommentare die Übersicht vereinfachen, indem sie ankündigen, an welchen Stellen einzelne Schritte des Algorithmus beginnen.

Deklarationen, die für andere Programmteile zur Verfügung gestellt werden, sollen mit Kopfkomentaren besonders erläutert werden. Der Leser soll den Nutzen dieser Deklarationen verstehen,

ohne deren Implementierung oder Verwendung im Detail untersuchen zu müssen. Kopfkommatare werden wie in den nachfolgenden Abschnitten beschrieben aufgebaut. Texte zwischen spitzen Klammern <...> werden als Platzhalter für auszufüllenden Text verwendet.

3.1 Kopfkomentar für Dateien

Quelltext-Dateien erhalten einen Kommentar am Beginn der Datei. Dieser Kommentar enthält generelle Informationen über den Inhalt der Datei.

```
--UUFILE:UUUU<Dateiname>
--
--UUPROJECT:UProgrammieruebungen,UUebungsblattU<Nummer>
--UUVERSION:U<Revisionsnummer>
--UUDATE:UUUU<DatumUderUletztenUÄnderung>
--UUAUTHOR:UU<NameUoderULoginnameUdesUAutors>
--
-----
--
--UU<ART DER PROGRAMMEINHEIT>U<NameUderUProgrammeinheit>
--
--UU<ProsabeschreibungUdesUZwecksUbeiUSpezifikationen>
--UU<ErläuterungUderUImplementierungUbeiUBodies>
--
```

Bemerkungen

- Die Punkte <PROJECT>, <VERSION>, <DATE> und <AUTHOR> können durch ein Versionsverwaltungssystem (z.B. Subversion oder CVS) automatisch ausgefüllt werden.
- <ART DER PROGRAMMEINHEIT> kann sein FUNCTION, PROCEDURE, PACKAGE oder PACKAGE BODY, davor kann jeweils GENERIC stehen.

3.2 Kopfkomentar für Unterprogramme

Alle Unterprogramme mit Ausnahme des Hauptprogramms erhalten einen Kopfkomentar, dessen Aufbau nachfolgend beschrieben wird. Das Hauptprogramm benötigt keinen eigenen Kommentar, weil seine Funktion im Datei-Kopfkomentar erfolgt.

```
--UUFUNCTIONU<Name>
--
--UU<Prosatext>
--
--UUPARAMETERS:
--UU+U<ersterUName>:U<Beschreibung>
--UU+U<zweiterUName>:U<Beschreibung>
--UU...
--UURETURNS:U<Beschreibung>
--UURAISES:U<Exception-Name>U<GrundUfürUPropagation>
--UUUUUUUUUUUUUUUU...
--UUPRE:U<formalUbeschriebeneUVorbedingung>
--UUPOST:U<formalUbeschriebeneUNachbedingung>
--UUCOMPLEXITY:U<KomplexitätUinU0-Notation>
```

Bemerkungen

- FUNCTION kann durch PROCEDURE ersetzt werden. Beides kann auch mit GENERIC stehen.

- Der <Prosatext> erläutert die Funktionsweise des Unterprogramms und kann Hinweise auf ein mögliches Einsatzszenario geben. Einsatzszenarien lassen sich meist besser in einem allgemeinen Teil innerhalb des Datei-Kopfkommentars erläutern.
- PARAMETERS sollte weggelassen werden, falls das Unterprogramm höchstens einen Parameter hat und dessen Funktion anhand des <Prosatext> ausreichend klar wird. Kommentare wie „Das Objekt“ sind für den Leser nicht hilfreich.
- In dem RAISES-Abschnitt werden alle Ausnahmen, die von dem Unterprogramm propagiert werden können aufgelistet. In dem <Grund für Propagation> wird genau beschrieben, unter welchen Umständen das Unterprogramm diese Ausnahme propagiert. Dabei sollte falls möglich ein Bezug auf die Vorbedingung hergestellt werden.
- Im PRE-Abschnitt wird eine Vorbedingung (bool'scher Ausdruck) angegeben, die zur korrekten Ausführung des Unterprogramms vor jedem Aufruf erfüllt sein muss. Oftmals ist in der Praxis die Vorbedingung eines Unterprogramms sehr komplex und formal schwierig darstellbar. In diesem Fall sollte der Abschnitt weggelassen werden. Unvollständige Vorbedingungen sind nicht hilfreich. Der <Prosatext> oder der RAISES-Abschnitt sollten Auskunft darüber geben was geschieht falls das Unterprogramm trotzdem aufgerufen wird.
- Im POST-Abschnitt wird eine formale Bedingung angegeben, die nach Rückkehr des Unterprogramms stets gilt, falls vor dem Aufruf die Vorbedingung gegolten hatte. Die Angabe ist oftmals in der Praxis schwierig. In diesem Fall sollte der Abschnitt weggelassen werden.
- Im COMPLEXITY-Abschnitt wird das Laufzeitverhalten des Unterprogramms im schlimmsten Fall angegeben. Es kann die $O(f(n))$ Notation verwendet werden. Es muss explizit angegeben werden, auf welches n sich die Angabe bezieht. In der Praxis ist die Angabe von Komplexitäten häufig schwierig. Sie darf weggelassen werden.

3.3 Weitere Kopfkommentare

Für alle Deklarationen, die anderen Programmeinheiten zur Verfügung gestellt werden, sollten Kopfkommentare erstellt werden mit folgendem Aufbau:

```
--_<ART>_<Name >
--
--_<Prosatext >
```

<ART> könnte z.B. sein: TYPE, SUBTYPE, VARIABLE, EXCEPTION, PACKAGE, GENERIC PACKAGE, TASK, PROTECTED UNIT, ...

Der <Prosatext> soll deutlich machen, warum eine Deklaration dasteht und in welchem Zusammenhang sie verwendet wird. Auch sollte auf besondere Voraussetzungen aufmerksam gemacht werden, die zur Verwendung erfüllt sein müssen, z.B.:

```
--_TYPE_List
--
--_Stellt_eine_lineare_Liste_zur_Verfuegung._Objekte_muessen
--_mit_'Create'_initialisiert_werden.
type_List_is_private;
```

Quelltext-Beispiel

```

-- FILE:      stacks.ads
--
-- PROJECT: Programmierrichtlinie
-- VERSION: $Revision: 29 $
-- DATE:      $Date: 2006-10-25 10:49:37 +0200 (Wed, 25 Oct 2006) $
-- AUTHOR:    $Author: keulsn $
--
-----
--
-- GENERIC PACKAGE Stacks
--
-- Provides an abstract data type for a dynamic stack of elements of
-- any non-limited data type.
--
with Ada.Finalization;

generic
  -- Data type for elements of the stack data structure.
  type Element_Type is private;
package Stacks is

  -- TYPE Stack
  --
  -- Stack-objects can be declared as local variables. They are
  -- default-initialized to an empty stack. No manual
  -- initialization is necessary. Automatic finalization is
  -- provided.
  type Stack is limited private;

  -- EXCEPTION Stack_Empty
  --
  -- Raised whenever an operation is executed on an empty stack and
  -- that operation requires the stack to contain an element.
  Stack_Empty : exception;

  -- FUNCTION Is_Empty
  --
  -- Returns 'True' if and only if the stack 'Container' is
  -- empty. Returns 'False' otherwise.
  --
  -- RETURNS: Size (Container) = 0
  -- COMPLEXITY: 0 (1)
  function Is_Empty
    (Container : in Stack)
    return Boolean;

  -- FUNCTION Size
  --
  -- Returns the number of elements currently stored on the stack
  -- 'Container'.
  --

```

```

-- COMPLEXITY: 0 (1)
function Size
  (Container : in Stack)
  return Natural;

-- FUNCTION Top
--
-- Returns the current top element on a stack.
--
-- RETURNS: the element currently on top of the stack
-- RAISES: Stack_Empty if the stack is currently empty
-- PRE: not Is_Empty (Container)
-- COMPLEXITY: 0 (1)
function Top
  (Container : in Stack)
  return Element_Type;

-- PROCEDURE Push
--
-- Pushes one element on top of 'Container'.
--
-- PARAMETERS:
-- + Container: a stack
-- + Element: new element to push on top of the stack
-- POST: not Is_Empty (Container) and
--       Top (Container) = Element
-- COMPLEXITY: 0 (1)
procedure Push
  (Container : in out Stack;
   Element   : in      Element_Type);

-- PROCEDURE Pop
--
-- Removes the top element from 'Container'.
--
-- RAISES: Stack_Empty, if Is_Empty (Container)
-- PRE: not Is_Empty (Container)
-- COMPLEXITY: 0(1)
procedure Pop
  (Container : in out Stack);

-- PROCEDURE Clear
--
-- Removes all elements from 'Container'.
--
-- POST: Is_Empty (Container)
-- COMPLEXITY: 0 (Size (Container))
procedure Clear
  (Container : in out Stack);

```

private

```

type Stack_Node;

type Stack_Node_Access is access Stack_Node;

-- Type 'Stack' is a linear list. This type is used for nodes in
-- that list.
type Stack_Node is
  record
    -- Data element in this node
    Element : Element_Type;

    -- Link to next node in the list
    Next    : Stack_Node_Access := null;
  end record;

-- Stack datatype. This is an anchor to store the top element and
-- the size. 'Stack_Node's are allocated as needed.
type Stack is new Ada.Finalization.Limited_Controlled with
  record
    -- The top node on the stack, null if and only if the stack
    -- is empty.
    Top    : Stack_Node_Access := null;

    -- Number of elements on the stack.
    Size : Natural := 0;
  end record;

-- PROCEDURE Finalize
--
-- Redefinition to provide storage deallocation on finalization
procedure Finalize
  (Container : in out Stack);

end Stacks;

```

```
-- FILE:      stacks.adb
--
-- PROJECT: Programmierrichtlinie
-- VERSION: $Revision: 29 $
-- DATE:      $Date: 2006-10-25 10:49:37 +0200 (Wed, 25 Oct 2006) $
-- AUTHOR:    $Author: keulsn $
--
```

```
-----
with Ada.Unchecked_Deallocation;
```

```
package body Stacks is
```

```
-- Instantiate a deallocator procedure for nodes on the stack
```

```
procedure Free is new Ada.Unchecked_Deallocation
  (Object => Stack_Node,
   Name   => Stack_Node_Access);
```

```
function Is_Empty
  (Container : in Stack)
return Boolean
is
begin
  return Container.Top = null;
end Is_Empty;
```

```
function Size
  (Container : in Stack)
return Natural
is
begin
  return Container.Size;
end Size;
```

```
function Top
  (Container : in Stack)
return Element_Type
is
begin
  -- If 'Container' is empty then 'Container.Top' must not be
  -- dereferenced.
  if Container.Top = null then
    raise Stack_Empty;
  else
    return Container.Top.Element;
  end if;
end Top;
```

```
procedure Push
  (Container : in out Stack;
```

```

    Element    : in    Element_Type)
is
begin
    Container.Top := new Stack_Node'
        (Element => Element,
         Next    => Container.Top);
    Container.Size := Container.Size + 1;
end Push;

procedure Pop
(Container : in out Stack)
is
    Top_Node : Stack_Node_Access := Container.Top;
begin
    -- If 'Container' is empty then 'Top_Node' must not be
    -- dereferenced.
    if Top_Node = null then
        raise Stack_Empty;
    else
        Container.Top := Top_Node.Next;
        Container.Size := Container.Size - 1;
        -- deallocate old top node
        Free (Top_Node);
    end if;
end Pop;

procedure Clear
(Container : in out Stack)
is
    Next : Stack_Node_Access;
begin
    while Container.Top /= null loop
        -- Remove all nodes
        -- 'Container.Size' remains unmodified while inside the loop!
        Next := Container.Top.Next;
        Free (Container.Top);
        Container.Top := Next;
    end loop;
    -- Adjust 'Container's size
    Container.Size := 0;
end Clear;

procedure Finalize
(Container : in out Stack)
is
begin
    -- Free all nodes
    Clear (Container);
end Finalize;

end Stacks;

```