Stilguide för Ada

Tommy Olsson
Linköpings universitet och Tekniska högskola
Institutionen för datavetenskap
Programvara och system
581 83 Linköping

Kodningsstil har varit en ofta förbisedd men inte desto mindre viktig aspekt på programmering. Sättet att skriva programkod på har en mycket stor betydelse för läsbarheten och förståelsen av program. Insikten om att program måste vara läsbara och förståeliga för mänskliga läsare har vuxit sig allt starkare med tiden och idag är läsbarhet en av de viktigaste aspekterna på programkod.

Denna enkla stilguide är framtagen i främsta syfte att användas i grundkurser i programmering, dels för att visa på god programmeringsstil i allmänhet och speciellt då det gäller program skrivna i Ada. För en mer komplett och omfattande stilguide för Ada hänvisas till [1], som ligger till grund för denna korta stilguide och från vilken även exemplen lånats.

1. Källkodspresentation

Den grafiska utformningen av källkoden på en sida eller skärn har stor betydelse för läsbarheten. Nedan ges både generella och specifika rekommendationer för att göra kod mer läsbar. Om man inte håller med om de specifika rekommendationerna kan man införa egna men fortfarande hålla fast vid de generella.

En helt igenom konsekvent utformning kan vara svårt att uppnå eller kontrollera manuellt. Man kan därför vilja använda ett formatteringsverktyg. Många av riktlinjerna nedan är lätta för sådana verktyg att utföra, eftersom riktlinjerna grundar sig på Adas välformade syntax. Vissa rekommendationer grundar sig däremot på semantik och är svårare för automatiska verktyg att utföra eller bibehålla.

1.1 Kodformattering

Kodformattering påverkar kods utseende, ej vad koden gör. Detta inbegriper horisontell uppdelning med mellanrum, indrag, rätning, paginering och radlängd. Den övergripande rekommendationen är att vara genomgående konsekvent i alla kompileringsenheter liksom hela projekt.

1.1.1 Horisontell uppdelning

Rekommendation

R.1 Var konsekvent i insättningen av mellanrum kring avgränsare.

R.2 Sätt in mellanrum på samma sätt som du skulle göra i vanlig prosa

Realisering

Sätt in åtminstone ett mellanrumstecken i följande fall. Normalt sätts endast ett mellanrum in, men i fall där man vill åstadkomma rätning i något avseende (se nedan) används flera mellanrum.

- före och efter följande avgränsare och tvåställiga operatorer:

```
+ - * / &
< = > /= <= >=
:= => | ..
:
<>
```

- kring sträng- och teckenlitteraler, dvs på utsidan av citattecknen (”) och apostroferna (’) som avgränsar sådana litteraler, utom i fall då detta är förbjudet.
- utanför men ej innanför parenteser.
- efter komma (,) och semikolon (;).

Sätt *ej* in mellanrum i följande situatuioner, även om det står i konflikt med ovanstående rekommendationer:

- efter operatorern plus (+) och minus (-) då de används som enställiga operatorer.
- efter ett funktionsanrop.
- innanför etikettavgränsarna (<< >>).
- före och efter exponentieringsoperatorn (**), apostrof (’) och punkt (.).
- mellan flera på varandra följande inledande eller avslutande parenteser.
- före komma (,) och semikolon (;).

I sammansatta uttryck, då överflödiga parenteser utelämnats pga beräkningsordningen för operatorer, är det tillåtet att ta bort mellanrum omkring operatorerna med högst prioritet inom uttrycket.

Exempel

```
Default_String : constant String :=
   ”Detta är den långa sträng som returneras” &
   ” standardmässigt. Den är uppdeld i flera” &
   ” källkodsrader.”;

type Signed_Whole_16 is range -2**15 .. 2**15 - 1;
type Address_Area is array (Natural range <>) of Signed_Whole_16;

Register : Address_Area (16#7FF0# .. 16#7FFF#);
Memory   : Address_Area (       0 .. 16#7FEC#);

Register(Pc) := Register(A);

X := Signed_Whole_16(Radius * Sin(Angle));

Register(Index) := Memory(Base_Address + Index * Element_Length);

Get(Value => Sensor);

Error_Term := 1.0 - (Cos(Theta)**2 + Sin(Theta)**2);

Z      := X**3;
Y      := C * X + B;
Volume := Length * Width * Height;
```

Motivering och förklaring

Det är rekommendabelt att sätta in mellanrum kring avgränsare och operatorer eftersom de vanligtvis är en- eller tvåteckenssekvenser som annars lätt försvinner bland de längre nyckelorden och identifierarna. Genom att sätta in mellanrum kring dem framstår de tydligare. Konsekvent insättning av mellanrum gör det lättare att ögna igenom källkoden.

Vissa avgränsare, som komma, semikolon, parenteser, punkt, etc., är dock kända som vanliga interpunktionstecken och det är distraherande att se dem behandlade annorkunda då det gäller insättning av mellanrum i program, jämfört med vad vi är vana vid från vanlig text. Använd därför samma konventioner för dessa avgränsare som i vanlig text (inget mellanrum före komma och semikolon, inga mellanrum innanför parenteser, etc.).

Undantag

Ett undantag från ovanstående motivering är kolon (:), som i Ada med fördel används som tabulator eller kolumnuppdelare. I detta fall är det vettigt att sätta in mellanrum både före och efter kolon och ej enbart efter, som i vanlig text.

1.1.2 Indrag

Rekommendation

R.3 Dra in och rada upp nästlade styrstrukturer, fortsättningsrader och omslutna enheter konsekvent.

R.4 Särskilj indrag för nästlade styrstrukturer och för fortsättningsrader.

R.5 Använd mellanrumstecken för indrag, ej tabuleringstecken.

Realisering

Följande indragningskonventioner rekommenderas. Observera att det är det minimala indraget som anges. Mer indrag kan behövas för att uppnå vertikal uppradning enligt rekommendationer som ges längre fram.

- använd den rekommenderade uppdelning av textavsnitt som visas i Ada Reference Manual [2].
- använd *tre* mellanrum som grundläggande indrag för nästling.
- använd *två* mellanrum som grundläggande indrag för fortsättningsrader.

Exempel

En satsetikett dras tillbaka tre mellanrum, en fortsättningsrad dras in två mellanrum:

```
begin
<<label>>                          |   <long statement with line break>
   <statement>                     |     <trailing part of same statement>
end;
```

`if`-satsen och den grundläggande slingan:

```
if <condition> then                |   <name>:
   <statements>                    |      loop
elsif <condition> then             |         <statements>
   <statements>                    |         exit when <condition>;
else                               |         <statements>
   <statements>                    |      end loop <name>;
end if;                            |
```

Slingor med **`for`**- och **`while`**-iterationsschemat:

```
<name>:                            |   <name>:
   for <scheme> loop               |      while <condition> loop
      <statements>                 |         <statements>
   end loop <name>;                |      end loop <name>;
```

Satsblock och **`case`**-satsen enligt rekommendationen i Ada Reference Manual [2]:

```
<name>:                            |   case <expression> is
   declare                         |      when <choice> =>
      <declarations>               |            <statements>
   begin                           |         when <choice> =>
      <statements>                 |         <statements>
   exception                       |      when others  =>
      when <choice> =>             |         <statements>
         <statements>              |   end case;  --<comment>
      when others =>               |
         <statements>              |
   end <name>;                     |
```

Följande **`case`** -satser sparar utrymme i jämförelse med ovanstående. Vilket du än väljer, var konsekvent.

```
case <expression> is               |   case <expression> is
when <choice> =>                   |      when <choice> => <statements>
   <statements>                    |                       <statements>
when <choice> =>                   |      when <choice> => <statements>
   <statements>                    |      when others  => <statements>
when others =>                     |   end case;
   <statements>                    |
end case;                          |
```

De olika formerna av selektiv väntan och de tidsbegränsade och villkorliga ingångsanropen:

```
select                                  |   select
   when <guard> =>                      |      <entry call>;
      <accept statement>                |      <statements>
      <statements>                      |   or
or                                      |      delay <interval>;
   <accept statement>                   |         <statements>
   <statements>                         |   end select;
or                                      |
   when <guard> =>                      |   select
      delay <interval>;                 |      <entry call>;
      <statements>                      |      <statements>
or                                      |   else
   when <guard> =>                      |      <statements>
      terminate;                        |   end select;
else                                    |
   <statements>                         |   select
end select;                             |      <triggering alternative>
                                        |   then abort
                                        |      <abortable part>
                                        |   end select;
```

accept-satsen:

```
accept <specification> do
   <statements>
end <name>;
```

Underenhet:

```
separate (<parent unit>)
<proper body>
```

Kroppar (proper bodies) för programenheter:

```
procedure <specification> is            |   package body <name> is
   <declarations>                       |      <declarations>
begin                                   |   begin
   <statements>                         |      <statements>
exception                               |   exception
   when <choice> =>                     |      when <choice> =>
      <statements>                      |         <statements>
end <name>;                             |   end <name>;
                                        |
function <specification>                |   task body <name> is
  return <type name> is                 |      <declarations>
   <declarations>                       |   begin
begin                                   |      <statements>
   <statements>                         |   exception
exception                               |      when <choice> =>
   when <choice> =>                     |         <statements>
      <statements>                      |   end <name>;
end <name>;                             |
```

Kontextklausuler för kompileringsenheter arrangeras som en tabell. Generiska formella parametrar inverkar inte på enheten själv. Funktions-, paket- och processpecifikationer använder standardindrag:

```
with <name>; use <name>;          |    function <specification>
with <name>;                      |       return <type>;
with <name>;                      |
                                  |    package <name> is
                                  |       <declarations>
                                  |    private
<compilation unit>                |       <declarations>
                                  |    end <name>;
                                  |
generic                           |    task type <name> is
    <formal parameters>           |       <entry declarations>
<compilation unit>                |    end <name>;
```

Instansiering av generiska enheter och indrag i postdeklarationer:

```
procedure <name> is               |    type ... is
   new <generic name> <actuals>   |       record
                                  |          <component list>
function <name> is                |          case <discriminant name> is
   new <generic name> <actuals>   |             when <choice> =>
                                  |                <component list>
package <name> is                 |             when <choice> =>
   new <generic name> <actuals>   |                <component list>
                                  |          end case;
                                  |       end record;
```

Indrag för post-”alignment”:

```
for <name> use
   record <mod clause>
      <component clause>
   end record;
```

Taggade typer och typutvidgning:

```
type ... is tagged
   record
      <component list>
   end record;

type ... is new ... with
   record
      <component list>
   end record;
```

Motivering och förklaring

Indrag förbättrar kodens läsbarhet därför att den ger en visuell indikation om programstrukturen. Nästlingsnivån visas tydligt genom indraget och de första och sista nyckelorden i en konstruktion kan matchas visuellt. Konsekvent indragning är mycket viktigt.

Undantag

Om man använder typnitt med variabel teckenbredd kommer tabuleringstecken att ge rakare uppställning av koden. Beroende på tabulatorlägen kan detta ge snabbt växande indrag i djupt nästlade konstruktioner, med kort effektiv radlängd som följd.

1.1.3 Rätning av operatorer

Rekommendation

R.6 Placera operatorer i rät linje, vilket framhäver lokal programstruktur och semantik.

Exempel

```
if Slot_A >= Slot_B then
   Temporary := Slot_A;
   Slot_A    := Slot_B;
   Slot_B    := Temporary;
end if;
-------------------------------------------------------------Numera-
tor   := B**2 - 4.0 * A * C;
Denominator := 2.0 * A;
Solution_1 := (B + Square_Root(Numerator)) / Denominator;
Solution_2 := (B - Square_Root(Numerator)) / Denominator;
-------------------------------------------------------------
X := A * B +
     C * D +
     E * F;
Y := (A * B + C) +  (2.0 * D - E) -  -- basic equation
     3.5;                            -- account for error factor
```

Motivering och förklaring

Rätning gör det lättare att se operatorernas placering och framhäver därigenom visuellt vad koden utför.

Undantag

Om rätning av operatorer innebär att en sats delas upp på två rader, speciellt om den delas på ett olämpligt ställe, kan det vara att föredra att mjuka upp dennare kommendation.

1.1.4 Rätning av deklarationer

Rekommendation

R.7 Använd rätning för att öka läsbarheten för deklarationer.

R.8 Skriv högst en deklaration per rad.

R.9 Dra in alla deklarationer i en deklarativ del lika mycket.

Realisering

För deklarationer som ej separeras av tomma rader, använd följande regler för rätning:

- placera kolon-avgränsare (:) i rät linje
- placera initieringsavgränsare (:=.) i rät linje
- då avslutande kommentarer används, placera kommentarsavgränsare (--) i rät linje
- då en deklaration blir för lång för en rad, bryt raden och lägg till en indragsnivå för fortsättningen. Ställen att föredra för radbrytningar är: (1) kommentarsavgränsaren; (2) initieringsavgränsaren; (3) kolonavgränsaren.
- för uppräkningstypsdeklarationer som ej ryms på en rad, skriv varje litteral på en egen rad och dra in ett indragningssteg. I tillämpliga fall kan semantiskt sammanhörande litteraler arrangeras per rad eller kolumn för att forma en tabell.

Exempel

Variabel- och konstantdeklarationer kan skrivas i tabellform genom uppdelning i kolumner med hjälp av symbolerna :, := och --.

```
Prompt_Column : constant        := 40;
Question_Mark : constant String := ” ? ”; -- prompt on error input
Prompt_String : constant String := ” ==> ”;
```

Om detta resulterar i rader som är för långa, kan deklarationerna skrivas ut med en del per rad och med specialanpassat indrag.

```
subtype User_Response_Text_Frame is String (1 .. 72);
-- If the declaration needed a comment, it would fit here.
Input_Line_Buffer : User_Response_Text_Frame
       := Prompt_String &
          String'(1 .. User_Response_Text_Frame'Length -
                      Prompt_String'Length => ' ');
```

Deklarationer av uppräkningslitteraler kan listas i en eller flera kolumner, som följande:

```
type Op_Codes_In_Column is
      (Push,
       Pop,
       Add,
       Subtract,
       Multiply,
       Divide,
       Subroutine_Call,
       Subroutine_Return,
       Branch,
       Branch_On_Zero,
       Branch_On_Negative);
```

eller, för att spara utrymme:

```
type Op_Codes_Multiple_Columns is
      (Push,            Pop,                Add,
       Subtract,        Multiply,           Divide,
       Subroutine_Call, Subroutine_Return,  Branch,
       Branch_On_Zero,  Branch_On_Negative);
```

eller, för att framhäva samhörighet mellan grupper av värden:

```
type Op_Codes_In_Table is
      (Push,            Pop,
       Add,             Subtract,          Multiply,    Divide,
       Subroutine_Call, Subroutine_Return,
       Branch,          Branch_On_Zero,    Branch_On_Negative);
```

En tabellformad uppställning främjar läsbarhet, genom att förhindra att namn ”gömmer” sig i en klump av deklarationer. Detta gäller för alla slags deklarationer: typer, undertyper, objekt, undantag, namngivna tal, etc.

1.1.5 Mer om rätning

Rekommendation

R.10 Placera parametermoder och parenteser i rät linje.

Realisering

Specifikt rekommenderas:

- skriv en formell parameterspecifikation per rad.
- placera parameternamn, kolon, det reserverade ordet **`in`**, det reserverade ordert **`out`**, och parametersubtyper i rät linje
- placera den första parameterspecifikationen på samma rad som underprogramnamnet, eller ingångsnamnet. Om några parametersubtyper tvingas förbi tillåten radlängd, placera den första parameterspecifikationen på en ny rad och använd samma indrag som för en fortsättningsrad.

Exempel

```
procedure Display_Menu (Title   : in     String;
                        Options : in     Menus;
                        Choice  :    out Alpha_Numerics);
```

Följande exempel visar alternativa varianter av denna rekommendation:

```
procedure Display_Menu_On_Primary_Window
      (Title   : in     String;
       Options : in     Menus;
       Choice  :    out Alpha_Numerics);
```

eller:

```
procedure Display_Menu_On_Screen (
      Title   : in     String;
      Options : in     Menus;
      Choice  :    out Alpha_Numerics
   );
```

Placering av parentesuttryck i rät linje gör komplicerade relationsuttryck tydligare:

```
if not (First_Character in Alpha_Numerics and then
        Valid_Option(First_Character))       then
```

Motivering och förklaring

Rätning gör program mer läsbara och lättare att förstå. I vissa fall, som visats ovan, får man effekten av en tabelluppställning, vilket ytterligare förenklar läsning och förståelse.

1.1.6 Tomma rader

Rekommendation

R.11 Använd tomma rader frö att gruppera logiskt sammanhörande textrader.

Exempel

```
if ... then

   for ... loop
      <statements>
   end loop;

end if;
```

```
type Employee_Record is
   record
      Legal_Name    : Name;
      Date_Of_Birth : Date;
      Date_Of_Hire  : Date;
      Salary        : Money;
   end record;
```

```
type Day is
   (Monday,   Tuesday, Wednesday, Thursday, Friday,
    Saturday, Sunday);

subtype Weekday is Day range Monday   .. Friday;
subtype Weekend is Day range Saturday .. Sunday;
```

Motivering och förklaring

Då tomma rader används med eftertanke och konsekvent, blir sammanhörande kodavsnitt mer synliga för läsaren.

1.1.7 Paginering

Rekommendation

R.12 Framhäv början av varje paket eller processpecifikation, början av varje programenhetskropp och slutet av varje programenhet.

Realisering

Specifikt rekommenderas:

- använd filprologer och andra strategiska kommentarer (en strategiska kommentar står före den enhet den beskriver).
- använd en rad med bindestreck som startar i samma kolumn som aktuellt indrag för att framhäva definitionen av nästlade enheter i en deklarativ del. Sätt in en rad med bindestreck direkt före och efter definitionen.
- om två rader med bindestreck följer direkt på varandra utelämnas den längre av de två.

Exempel

```
with Basic_Types;

package body SPC_Numeric_Types is
...
   ------------------------------------------------------------------
   function Max (Left  : in     Basic_Types.Tiny_Integer;
                 Right : in     Basic_Types.Tiny_Integer)
         return Basic_Types.Tiny_Integer is
   begin
      if Right < Left then
         return Left;
      else
         return Right;
      end if;
   end Max;

   ------------------------------------------------------------------
   function Min (Left  : in     Basic_Types.Tiny_Integer;
                 Right : in     Basic_Types.Tiny_Integer)
         return Basic_Types.Tiny_Integer is
   begin
      if Left < Right then
         return Left;
      else
         return Right;
      end if;
   end Min;

   ------------------------------------------------------------------

   use Basic_Types;

begin  -- SPC_Numeric_Types
   Max_Tiny_Integer := Min(System_Max, Local_Max);
   Min_Tiny_Integer := Max(System_Min, Local_Min);
   -- ...
end SPC_Numeric_Types;
```

Motivering och förklaring

Det är lätt att förbise delar av programenheter som inte är synliga på den aktuella sidan eller skärmen. Sidlängden hos presentationsmedier varierar stort. Genom att tydligt markera programs logiska sidgränser (med en strategisk kommentar eller med en streckad linje) möjliggör man för läsaren att snabbt kon-

trollera om alla delar av en programenhet syns. Sådan paginering gör det också enklare att snabbt skumma igenom en stor fil för att hitta en specifik programenhet.

1.1.8 Antal satser per rad

Rekommendation

R.13 Börja varje sats på en ny rad.

R.14 Skriv inte mer än en enkel sats per rad.

R.15 Dela upp sammansatta satser på flera rader.

Exempel

Skriv:

```
if End_Of_File then
   Close_File;
else
   Get_Next_Record;
end if;
```

heller än:

```
if End_Of_File then Close_File; else Get_Next_Record; end if;
```

Exceptiopnella fall:

```
Put(”A=”);   Natural_IO.Put(A);   New_Line;
Put(”B=”);   Natural_IO.Put(B);   New_Line;
Put(”C=”);   Natural_IO.Put(C);   New_Line;
```

Motivering och förklaring

En enkel sats på varje rad gör det enklare för läsaren att hitta satser och minskar risken att missa satser. Strukturen hos en sammansatt sats blir däremot tydligare då dess delar skrivs på separata rader.

Undantag

Exemplet med `Put` och `New_Line` ovan visar ett motiverat avsteg från rekommendationen. Denna gruppering av intimt sammanhörande satser på samma rad gör den strukturella relationen mellan grupperna tydlig.

1.1.9 Källkodsradlängd

Rekommendation

R.16 Håll fast vid en maximal radlängd för källkod.

Realisering

Specifikt rekommenderas:

- begränsa källkodsraders längd till 72 tecken.

Motivering och förklaring

Då kod flyttas från ett system till ett annat, kan det finnas begränsningar på poststorleken för källkodsrader, av olika skäl, både beroende på begränsningar i operativsystemen som begränsningar i utskriftsmedia.

Källkod ska ibland publiceras och A4-format, eller mindre, är inte lika överseende som en datorlistning i termer av antal användbara kolumner.

Utöver detta finns mänskliga begränsningar då det gäller läsabarhet och förståelse av källkod, som råkar ligga i intervallet 70-80 teckens kolumnbredd.

1.2 Sammanfattning

Kodformattering:

- sätt in mellanrum omkring avgränsare på ett konsekvent sätt
- sätt in mellanrum som du skulle göra i vanlig prosa
- drag in styrstrukturer, fortsättningsrader och nästlade enheter på ett konsekvent sätt
- ha olika indrag för nästlade strukturer och indrag för fortsättningsrader
- använd mellanrumstecken för indrag, ej tabulertecken
- rada upp operatorer vertikalt för att framhäva lokal programstruktur och semantik
- använd vertikal uppradning för att öka deklarationers läsbarhet
- skriv högst en deklaration per rad
- använd lika indrag för samtliga deklarationer i en deklarativ del
- rad upp parametermoder och parenteser vertikalt
- använd tomma rader för att gruppera logiska sammanhörande textrader
- framhäv början på varje paket- och processpecifikation, början av varje programenhetskropp och slutet av varje programenhet
- börja varje sats på ny rad
- skriv inte mer än en enkel sats per rad
- dela upp sammansatta sater på flera rader
- håll fast vid en maximal radlängd för källkod

2. Läsbarhet

Nedan ges rekommendationer för att använda egenskaper i Ada för att göra kod lättare att läsa och förstå. det finns mycket myter kring kommentarer och läsbarhet. Grunden för verklig läsbarhet ligger mer i namngivning och kodstruktur, än i kommentering. Att ha lika många kommentarsrader som kodrader medför ej läsbahet, utan tyder snarae på att den som skrivit koden inte förstått vad som är viktigt att förmedla.

2.0.1 Stavning

Rekommendation

R.17 Använd understrykningstecken för att separera orden i sammansatta namn.

Exempel

```
Miles_Per_Hour
Entry_Value
```

Motivering och förklaring

Då en identifierare består av flera ord är den mycket lättare att läsa om orden delas upp med understrykningstecken. Dessutom har en källkodsformatterare större kontroll då det gäller ändring av av begynnelsebokstaven i orden.

2.0.2 Tal

Rekommendation

R.18 Representera tal på ett konsekvent sätt.

R.19 Representera litteraler i ett lämpligt radix med tanke på problemet ifråga.

R.20 Använd understrykningstecken för att separera siffror på samma vis som komma och punkt (eller mellanrum för icke-decimala talbaser) skulle användas i vanlig text.

R.21 Då exponent används, skriv konsekvent exponentdelen med antingen `E` eller `e`.

R.22 I en talbas med bokstäver, skriv konsekvent bokstavstecken som antingen stora eller små bokstäver.

Realisering

- decimala och oktala tal skrivs i tregrupper med början till vänster om radixpunkten och i femgrupper med början till höger om radixpunkten.
- `E` i exponenttal skrivs alltid med stor bokstav.
- bokstavstecken som representerar siffror i talbaser större än 10 skrivs som versaler.
- hexadecimala tal skrivs i fyragrupper med början om respektive sida om radixpunkten.

Exempel

```
type Maximum_Samples     is range          1 ..  1_000_000;
type Legal_Hex_Address   is range   16#0000# ..   16#FFFF#;
type Legal_Octal_Address is range 8#000_000# .. 8#777_777#;

Avogadro_Number : constant := 6.02216_9E+23;
```

För att representera talet 1/3 som en konstant, använd:

```
One_Third : constant := 1.0 / 3.0;
```

Undvik:

```
One_Third : constant := 0.33333_33333_3333;
```

Motivering och förklaring

Konsekvent användning av versaler och gemener underlättar läsning av tal. Understrykningstecken används för att gruppera delar av tal i välkända mönster. Överensstämmelse med vanliga konventioner utgör en stor del i läsbarheten.

2.0.3 Användning av versaler och gemener

Rekommendation

R.23 Gör reserverade ord och andra programelement visuellt distinkt skilda från varandra.

Realisering

- använd gemener för alla reserverade ord.
- för alla andra identifierare, första bokstaven ska vara versal, övriga gemena, ord åtskiljs med understruktningstecken.
- använd versaler i förkortningar och initialord.

Exempel

```
...
type Second_Of_Day      is range 0 .. 86_400;
type Noon_Relative_Time is (Before_Noon, After_Noon, High_Noon);

subtype Morning   is Second_Of_Day range 0 .. 86_400 / 2 - 1;
subtype Afternoon is Second_Of_Day range Morning'Last + 2 .. 86_400;
...

Current_Time := Second_Of_Day(Calendar.Seconds(Calendar.Clock));

if Current_Time in Morning then
   Time_Of_Day := Before_Noon;
elsif Current_Time in Afternoon then
   Time_Of_Day := After_Noon;
else
   Time_Of_Day := High_Noon;
end if;

case Time_Of_Day is
   when Before_Noon =>   Get_Ready_For_Lunch;
   when High_Noon   =>   Eat_Lunch;
   when After_Noon  =>   Get_To_Work;
end case;
...
```

Motivering och förklaring

Att visuellt kunna särskilja reserverade ord gör det möjligt att fokusera på programstruktur enbart, om så önskas, och underlättar även då man ögnar igenom texten för att söka efter en viss identifierare.

Reserverade ord skrivna med gemener är mer läsbart för erfarna adaprogrammerare, för vilka reserverade ord ej behöver framträda så tydligt. Nybörjare tycker däremot ofta att reserverade ord behöver framhävas så att de därigenom lättare hittar styrstrukturerna. Det är därför inte ovanligt att läroböcker och lärares i nybörjarkurser material har reserverade ord skrivna med versaler. Referensmanualen för Ada använder feta gemener för alla reserverade ord, vilket också används här.

2.0.4 Förkortningar

Rekommendation

R.24 Använd inte en förkortning av ett långt ord som identifierare då det finns en kortare synonym.

R.25 Använd en konsekvent förkortningsstrategi.

R.26 Använd inte tvetydiga förkortningar.

R.27 För att motivera dess användning, måste en förkortning spara många tecken över hela ordet.

R.28 Använd förkortningar som är väl accepterade för tillämpningsområdet.

R.29 Underhåll en lista med accepterade förkortningar och använd enbart förkortningar från denna lista.

Exempel

Använd:

```
Time_Of_Receipt
```

istället för:

```
Recd_Time eller R_Time
```

Motivering och förklaring

Många förkortningar kan vara tvetydiga eller intetsägande om de inte ses i sitt sammanhang. Exempelvis `Temp` kan betyda antingen temporär eller temperatur. Av detta skäl bör man välja förkortningar omsorgsfullt.

Eftersom mycket långa identifierare kan störa strukturen hos program, speciellt i djupt nästlade styrstrukturer, är det en god regel att försöka hålla identifierare korta och memingsfulla. Använd korta, hela namn i alla lägen detta är möjligt.

Man kan erhålla förkortningar genom att använda **`renames`**-klausulen för långa, fullt kvalificerade namn.

2.1 Namngivningskonventioner

Välj namn som tydliggör den avsedda användningen av objekt eller en entiteter.

2.1.1 Namn

Rekommendation

R.30 Välj namn som är så självdokumenterande som möjligt.

R.31 Använd en kort synonym istället för en förkortning.

R.32 Använd namn tillhörande tillämpningsområdet men använd inte svårbegriplig jargång.

R.33 Undvik att använda samma namn för att deklarera olika slags identifierare.

R.34 Använd engelska namn. De reserverade orden och identifierarna i standardomgivningen är på engelska och språkblandning stör läsbarheten.

Exempel

I en trädvandrare är det full tillräckligt att använda namnet `Left` istället för `Left_Branch` för att i den kontexten erhålla full förståelse. Använd däremot `Time_Of_Day` istället för `TOD`.

I matematiken skrivs ofta formler med enbokstavsnamn för variablerna. Bibehåll denna konvention för matematiska ekvationer i sådana fall där detta påminner om formeln, t ex:

```
A*(X**2) + B*X + C
```

Motivering och förklaring

Ett program som följer denna rekommendation kan lättare förstås. Självdokumenterande namn medför mindre behov av förklarande kommentarer. Empiriska studier har visat att man ytterligare kan öka förståelsen om variabelnamn ej är överdrivet långa. Kontext och tillämpningsområde kan stödja i hög grad.

2.1.2 Undertypnamn[1]

Rekommendationer

R.35 Använd allmäna substantiv i singularis som identifierare för undertyper.

R.36 Välj identifierare som som beskriver ett av undertypens värden.

R.37 Överväg att använda suffix i identifierare som definierar pekartyper, delintervall eller fälttyper.

R.38 För privata typer, använd ej identifierarkonstruktioner (t ex suffix) som är unika för undertyp-identifierare.

R.39 Använd ej undertypnamn som finns i fördefinierade paket.

Exempel

```
type Day is
   (Monday,    Tuesday,   Wednesday, Thursday,  Friday,
    Saturday,  Sunday);

type Day_Of_Month    is range      0 ..    31;
type Month_Number    is range      1 ..    12;
type Historical_Year is range -6_000 .. 2_500;

type Date is
   record
      Day   : Day_Of_Month;
      Month : Month_Number;
      Year  : Historical_Year;
   end record;
```

Specifikt bör `Day` föredras istället för `Days` eller `Day_Type`.

Identifieraren `Historical_Year` kan framstå som specifik, mer är i realiteten allmän, med adjektivet `historical` beskrivande intervallbegränsningen.

```
-------------------------------------------------------------------------
procedure Disk_Driver is
   -- In this procedure, a number of important disk parameters are
   -- linked.
   Number_Of_Sectors  : constant :=     4;
   Number_Of_Tracks   : constant :=   200;
   Number_Of_Surfaces : constant :=    18;
   Sector_Capacity    : constant := 4_096;

   Track_Capacity   : constant := Number_Of_Sectors  * Sector_Capacity;
   Surface_Capacity : constant := Number_Of_Tracks   * Track_Capacity;
   Disk_Capacity    : constant := Number_Of_Surfaces * Surface_Capacity;

   type Sector_Range  is range 1 .. Number_Of_Sectors;
   type Track_Range   is range 1 .. Number_Of_Tracks;
   type Surface_Range is range 1 .. Number_Of_Surfaces;

   type Track_Map   is array (Sector_Range)  of ...;
   type Surface_Map is array (Track_Range)   of Track_Map;
   type Disk_Map    is array (Surface_Range) of Surface_Map;

begin  -- Disk_Drive
   ...
end Disk_Driver;
-------------------------------------------------------------------------
```

1. Denna beteckning ansluter till konventionern i Ada Reference Manual. I allmänhet refererar ”typ” (type) till det abstrakta begreppet, medan ”undertyp” (subtype) refererar till namnet givet till det abstrakta begreppet i en konkret deklaration.

Suffixen `_Capacity`, `_Range` och `_Map` (se ovan) anger syftet med undertyperna och man slipper leta efter synomymer för sektor-, spår- och sidabstraktionerna. Utan dess suffix skulle man behöva tre olika namn per abstraktion, en för att beskriva varje begrepp som koncist namnges i suffixen. Denna rekommendation är endast tillämplig för vissa synliga undertyper. Privata typer, t ex, bör ges ett bra namn som avspeglar den abstraktion som representeras.

Motivering och förklaring

Då denna stil och den rekommenderade stilen för identifierare används, kommer koden mer nära att efterlikna engelska. Vidare är denna stil i överensstämmelse med de fördefinierade namnen i Ada, vilka t ex ej skrivs `Integers`, `Integer_Type`, `Booleans` eller `Boolean_Type`.

2.1.3 Objektnamn

Rekommendation

R.40 Använd predikatsklausuler eller adjektiv för booleska objekt.

R.41 Använd substantiv i singularis som objektidentifierare.

R.42 Välj namn som beskriver objektets värde under exekvering.

R.43 Använd substantiv (appellativ) i singularis för postkomponenter.

Exempel

Icke-booleska objekt:

```
Today           : Day;
Yesterday       : Day;
Retirement_Date : Date;
```

Booleska objekt:

```
User_Is_Available : Boolean;        -- predikatsklausul
List_Is_Empty     : Boolean;        -- predikatsklausul
Empty             : Boolean;        -- adjektiv
Bright            : Boolean;        -- adjektiv
```

Motivering och förklaring

Användning av (specific) substantiv för objekt skapar en kontext för förståelse av objektets värde, vilket är ett av de allmäna värdena som beskrivs av objektets undertypsnamn. Objektdeklarationer blir mycket engelskliknamde med denna stil, t ex kan den första deklarationen utläsas ”Today is a Day”.

Appellativa substantiv, hellre än ”specific”, används för postkomponenter därför att postobjektets namn kommer att ge en kontext för att förstå komponenten. Således förstås följande komponent som ”the year of retirement”.

```
Retirement_Date.Year
```

Genom att följa konventioner som relaterar objekttyper och sätt att uttrycka sig gör man koden mer lik vanlig text. Exempelvis behöver följande kod ingen kommentering pga. de valda namnen:

```
if List_Is_Empty then
   Number_Of_Elements := 0;
else
   Number_Of_Elements := Length_Of_List;
end if;
```

2.1.4 Namngivning av taggade typer och tillhörande paket

Utelämnas här, se [1].

2.1.5 Programenhetsnamn

Rekommendation

R.44 Använd ”action verbs” för procedurer och ingångar.

R.45 Använd predikatsklausuler för boolska funktioner.

R.46 Använd substantiv för icke-boolska funktioner.

R.47 Ge paket namn som antyder en högre nivå av organisation än underprogram. I allmänhet är dessa substantivfraser som beskriver den abstraktion som paketet står för.

R.48 Ge processer namn som låter påskina en aktiv enhet.

R.49 Använd substantiv som är beskrivande för det data som skyddas av skyddade enheter.

R.50 Överväg att namnge generiska underprogram som om de vore icke-generiska underprogram.

R.51 Överväg att namnge generiska paket som om de vore icke-generiska paket.

R.52 Väl generiska namn som generellare än de instansierade namnen.

Exempel

Exenpel på procedurnamn:

```
procedure Get_Next_Token          -- get är ett transitivt verb
procedure Create                  -- create är ett transitivt verb
```

Exenpel på funktionsnamn för boolska funktioner:

```
function Is_Last_Item             -- predikatklausul
function Is_Empty                 -- predikatsklausul
```

Exenpel på funktionsnamn för icke-boolska funktioner:

```
function Successor                -- appellativ
function Length                   -- attribut
function Top                      -- komponent
```

Exenpel på paketnamn:

```
package Terminals is               -- appellativt
package Text_Routines is           -- appellativ
```

Exenpel på skyddade object:

```
protected Current_Location is      -- data som skyddas
protected type Guardian is         -- substantiv som implicerar skydd
```

Exenpel på processnamn:

```
task Terminal_Resource_Manager is  -- appellativ som påvisar handling
```

Följande kodexempel påvisar den klarhet som blir resultatet av att använda ”part-of-speech”-namn-givningskonventioner:

```
Get_Next_Token(Current_Token);

case Current_Token is
   when Identifier =>   Process_Identifier;
   when Numeric    =>   Process_Numeric;
end case;  -- Current_Token

if Is_Empty(Current_List) then
   Number_Of_Elements := 0;
else
   Number_Of_Elements := Length(Current_List);
end if;
```

Då paket och deras underprogram namnges tillsammans, är den resulterande koden mycket beskrivande:

```
if Stack.Is_Empty(Current_List) then
   Current_Token := Stack.Top(Current_List);
end if;
```

Motivering och förklaring

Användning av dessa namngivningskonventioner skapar fårståelig kod som i stort kan läsas på naturligt språk. Då verb används för handlingar, som underprogram, och substantiv används för objekt, som de data som underprogram manipuletrar, är kod lättare att läsa och förstå.

2.1.6 Konstanter och namngivna tal

Rekommendation

R.53 Använd symboliska värden istället för litteraler, sålångt möjligt.

R.54 Använd de fördefinierade konstanterna `Ada.Numerics.Pi` och `Ada.Numerics.e` för de matematiska konstanterna `Pi` och `e`.

R.55 Använd konstanter istället för variabler för konstanta värden.

R.56 Använd en konstant när värdet är specifikt för en typ eller då värdet måste vara statiskt.

R.57 Använd namngivna tal istället för konstanter, då så är möjligt.

R.58 Använd namngivna tal för att ersätta numeriska litteraler vars typ eller kontext är verkligt universell.

R.59 Använd konstanter för objekt vars värde ej kan ändras efter elaborering.

R.60 Visa samhörighet mellan symboliska värden genom att definiera dem med statiska uttryck.

R.61 Använd linjärt oberoende mängder av litteraler.

R.62 Använd attribut som 'Firts och 'Last istället för litteraler, där så är möjligt.

Exempel

```
3.14159_26535_89793                                  -- litteral
Max_Entries : constant Integer := 400;               -- konstant
Avogadros_Number  : constant := 6.022137 * 10**23;   -- namngivet tal
Avogadros_Number / 2                                 -- statiskt uttryck
Avogadros_Number                                     -- symboliskt värde
```

Genom att deklarera `Pi` som ett namngivet tal kan det refereras symboliskt:

```
Area := Pi * Radius**2;           -- om radien är känd.
```

istället för:

```
Area := 3.14159 * Radius**2;      -- Kräver förklarande kommentar.
```

Likaså är `Ada.Character.Latin_1.Bel` mer uttryckligt än `Character'Val(8#007#)`.

Klarheten i konstantdeklarationer och deklarationer av namngivna tal kan förbättras genom att använda andra konstanter och namngivna tal, t ex:

```
Bytes_Per_Page   : constant := 512;
Pages_Per_Buffer : constant := 10;
Buffer_Size      : constant := Pages_Per_Buffer * Bytes_Per_Page;
```

är mer självförklarande och enklare att underhålla än:

```
Buffer_Size : constant := 5_120;   -- ten pages
```

Följande litteraler bör vara konstanter:

```
if New_Character  = '$' then    -- "konstant" som kan komma att ändras
...
if Current_Column = 7 then      -- "konstant" som kan komma att ändras
```

Motivering och förklaring

Användning av identifierare istället för litteraler gör uttrycks innebörd klarare, vilket minskar behovet av kommentarer. Konstantdeklarationer som består av uttryck innehållande numeriska litteraler är säkrare, eftersom de inte behöver beräknas för hand. De är också mer upplysande än enkla numeriska litteraler, eftersom det finns större möjlighet att låta förklarande namn ingå.

En konstant har en typ. Ett namngivet tal kan endast vara en universell typ: `universal_integer` eller `universal_real`. Stark typning framtvingas för konstanter men inte för namngivna tal eller litteraler. Namngivna tal tillåter kompilatorer att generera effektivare kod än för konstanter och att utföra fullständigare felkontroll vid kompilering. Om litteralen innehåller ett stort anal siffror reducerar användningen av en identifierare antalet skrivfel. Om skrivfel förekommer är de enklare att lokalisera, antingen genom inspektion eller vid kompilering.

2.1.7 Undantag

R.63 Använd ett namn som visar vilket slags problem undantaget representerar.

Exempel

```
Invalid_Name   : exception;
Stack_Overflow : exception;
```

Motivering och förklaring

Namngivning av undantag i enhetlighet med det slags problem de detekterar ökar kodens läsbarhet. Namnge undantag så precist som möjligt så att underhållaren av koden förstår varför undantaget kan komma att resas. Ett väl namngivet undantag ska vara meningsfullt för klienterna till paketet som deklarerar undantaget.

2.1.8 Konstruktorer

Rekommendation

R.64 Låt prefix som `New`, `Make` eller `Create` ingå i namn på konstruktorer (operationer för att skapa och/eller initiera objekt).

R.65 Använd namn som är påvisar deras innebörden för barnpaket som innehåller konstruktorer.

Realisering

- Namnge ett barnpaket som innehåller konstruktorer <*något*>`.Constructor`.

Exempel

```
function Make_Square (Center : Cartesian_Coordinates;
                      Side   : Positive)
  return Square;
```

Motivering och förklaring

Inkludering av ett ord som `New`, `Make` eller `Create` i ett konstruktornamn, gör dess syfte tydligt. Man kan vilja begränsa användningen av prefixet New till konstruktorer som returnerar ett pekarvärde, eftersom prefixet antyder intern användning av en allokator.

Att placera alla konstruktorer i ett barnpaket, även om de returnerar pekarvärden, är en användbar organisationsprincip.

2.2 Kommentarer

2.2.1 Allmäna kommentarer

Rekommendation

R.66 Gör koden så klar som möjligt för att reducera behovet av kommentarer.

R.67 Upprepa aldrig information i en kommentar som redan är tillgänglig i koden.

R.68 Då en kommentar behövs, gör den koncis och fullständig.

R.69 Använd korrekt språk och stavning i kommentarer.

R.70 Gör kommentarer visuellt distinkta från koden.

R.71 För sådana kommentarer detta är intressant för, utforma dessa så att informationen automatiskt kan extraheras av ett verktyg.

Motivering och förklaring

2.2.2 Filhuvuden

Rekommendation

R.72 Inför ett filhuvud i varje källkodsfil.

R.73 Placera information ägarskap, ansvar och historik för filen i filhuvudet.

Realisering

- Skriv en copyright-notis i filhuvudet.
- Ange författarens namn och avdelning i filhuvudet.
- Inför en revisionshistorik i filhuvudet, med en sammanfattning av varje ändring, datum och namn på personen som gjort ändringen.

Exempel

```
------------------------------------------------------------------------
--      Copyright (c) 1991, Software Productivity Consortium, Inc.
--      All rights reserved.

-- Author: J. Smith
-- Department:System Software Department

-- Revision History:
--   98-01-12 H. Acker
--     - Added function Size_Of to support queries of node sizes.
--     - Fixed bug in Set_Size which caused overlap of large nodes.
--   7/1/91 M. Jones
--     - Optimized clipping algorithm for speed.
--   6/25/91 J. Smith
--     - Original version.
------------------------------------------------------------------------
```

Motivering och förklaring

2.2.3 Filhuvuden för programenhetsspecifikationer

2.2.4 Filhuvuden för programenhetskroppar

2.2.5 Datakommentarer

Rekommendation

R.74 Kommentera alla datatyper, objekt och undantag, såvida deras namn ej är självförklarande.

R.75 Kommentera komplexa, pekarbaserade datastrukturer.

R.76 Kommentera relationer som underhålls mellan dataobjekt.

R.77 Utelämna kommentarer som endast upprepar informationen i namnet.

Exempel

Objekt kan grupperas efter syfte och kommenteras enligt:

```
...
--------------------------------------------------------------------
-- Current position of the cursor in the currently selected text
-- buffer, and the most recent position explicitly marked by the
-- user.
-- Note:  It is necessary to maintain both current and desired
--        column positions because the cursor cannot always be
--        displayed in the desired position when moving between
--        lines of different lengths.
--------------------------------------------------------------------

Desired_Column : Column_Counter;
Current_Column : Column_Counter;
Current_Row    : Row_Counter;
Marked_Column  : Column_Counter;
Marked_Row     : Row_Counter;
```

Villkoren under vilka ett undantag reses bör skommenteras:

```
--------------------------------------------------------------------
-- Exceptions
--------------------------------------------------------------------

Node_Already_Defined : exception;   -- Raised when an attempt is made
                                    --|   to define a node with an
                                    --|   identifier which already
                                    --|   defines a node.
Node_Not_Defined     : exception;   -- Raised when a reference is
                                    --|   made to a node which has
                                    --|   not been defined.
```

Ett mer omfattande exempel, omfattande flera poster och pekartyper som används för att utforma en komplex datastruktur:

```
--------------------------------------------------------------------
-- These data structures are used to store the graph during the
-- layout process. The overall organization is a sorted list of
-- "ranks," each containing a sorted list of nodes, each containing
-- a list of incoming arcs and a list of outgoing arcs.
-- The lists are doubly linked to support forward and backward
-- passes for sorting. Arc lists do not need to be doubly linked
-- because order of arcs is irrelevant.
-- The nodes and arcs are doubly linked to each other to support
-- efficient lookup of all arcs to/from a node, as well as efficient
```

```
-- lookup of the source/target node of an arc.
-----------------------------------------------------------------------

type Arc;
type Arc_Pointer is access Arc;

type Node;
type Node_Pointer is access Node;

type Node is
   record
      Id       : Node_Pointer;   -- Unique node ID supplied by the user.
      Arc_In   : Arc_Pointer;
      Arc_Out  : Arc_Pointer;
      Next     : Node_Pointer;
      Previous : Node_Pointer;
   end record;

type Arc is
   record
      ID     : Arc_ID;           -- Unique arc ID supplied by the user.
      Source : Node_Pointer;
      Target : Node_Pointer;
      Next   : Arc_Pointer;
   end record;

type Rank;
type Rank_Pointer is access Rank;

type Rank is
   record
      Number     : Level_ID;     -- Computed ordinal number of the rank.
      First_Node : Node_Pointer;
      Last_Node  : Node_Pointer;
      Next       : Rank_Pointer;
      Previous   : Rank_Pointer;
   end record;

First_Rank : Rank_Pointer;
Last_Rank  : Rank_Pointer;
```

Motivering och förklaring

Det är mycket användbart med kommentarer som förklarar syftet, strukturen och semantiken för datastrukturerna. Många underhållare tittar först på datastrukturerna då de ska försöka förstå implementeringen av en enhet.

En annan fördel med att kommentera datadeklarationerna är att en uppsättning kommentarer kan ersätta flera omgångar med som annars kan behövas på flera ställen i koden där data manipuleras.

Det är viktigt att dokumentera undantag och under vilka omständigheter de kan resas, speciellt då undantag deklareras i en paketspecifikation. Läsaren har inget annat sätt att utröna den exakta innebörden av undantaget (utan att läsa koden i paketkroppen). Det är bättre att kommentera undantaget i allmäna termer istället för att räkna upp alla underprogram som kan generera undantaget; en sådan lista är svårare att underhålla.

2.2.6 Satskommentarer

Rekommendation

R.78 Minimera kommentarer inbäddade bland satser.

R.79 Använd endast kommentarer för att förklara delar av koden som ej är självklar.

R.80 Kommentera sådant som avsiktligt utelämnats i koden.

R.81 Använd ej kommentarer för att parafrasera koden.

R.82 Använd ej kommentarer för att förklara avsides befintlig kod, som t ex koden i ett underprogram som anropas av den aktuella enheten.

R.83 Där kommentarer är nödvändiga, gör dem visuellt distinkta från koden.

Exempel

Följande är ett exempel på illa kommenterad kod:

```
...
-- Loop through all the strings in the array Strings, converting
-- them to integers by calling Convert_To_Integer on each one,
-- accumulating the sum of all the values in Sum, and counting them
-- in Count.  Then divide Sum by Count to get the average and store
-- it in Average. Also, record the maximum number in the global
-- variable Max_Number.

for I in Strings'Range loop
   -- Convert each string to an integer value by looping through
   -- the characters which are digits, until a nondigit is found,
   -- taking the ordinal value of each, subtracting the ordinal value
   -- of '0', and multiplying by 10 if another digit follows.  Store
   -- the result in Number.
   Number := Convert_To_Integer(Strings(I));
   -- Accumulate the sum of the numbers in Total.
   Sum := Sum + Number;
   -- Count the numbers.
   Count := Count + 1;

   -- Decide whether this number is more than the current maximum.
   if Number > Max_Number then
      -- Update the global variable Max_Number.
      Max_Number := Number;
   end if;
end loop;
-- Compute the average.
Average := Sum / Count;
```

Följande har förbättrats, genom att ej upprepa saker i kommentarer som är uppenbara i koden, genom att ej beskriva detaljerna om vad som föregår i Convert_To_Integer, genom att den felaktiga kommentaren till satsen som ackumulerar summan tagits bort, och genom att de få återstående kommentarerna gjorts mer visuellt distinkta från koden:

```
Sum_Integers_Converted_From_Strings:
   for I in Strings'Range loop
      Number := Convert_To_Integer(Strings(I));
      Sum := Sum + Number;
      Count := Count + 1;

      -- The global Max_Number is computed here for efficiency.
      if Number > Max_Number then
         Max_Number := Number;
      end if;
   end loop Sum_Integers_Converted_From_Strings;

Average := Sum / Count;
```

Motivering och förklaring

Förbättringarna i exemplet ovan är inte enbart i form av antalet kommentarer reducerats, utan även i form av att antalet meningslösa kommentarer reducerats.

Kommentarer som parafraserar eller förklarar uppenbara aspekter på koden har inget värde. De innebär är ett slöseri av energi för författaren att skriva och för underhållaren att uppdatera. Därför slutar det ofta med att sådana kommentarer blir felaktiga. De skräpar ned i koden och döjler de få viktiga aspekterna.

En kommentar som beskriver vad som försiggår inuti en annan enhet bryter mot principen om döljande av information. Den är irrelevanta för den anropande enheten och utelämnas hellre ifall den anropade enheten någon gång kommer att ändras på ett sätt som avspeglar sig i kommentaren. I annat fall får man stora problem med att underhålla kommentarer som beskriver vad som försiggår i andra delar av koden.

Fördelen med att göra kommentarer visuellt distinkta från koden är att detta gör det lättare att ögra igenom koden och de få viktiga kommentarerna framstår tydligare.

2.2.7 Markörkommentarer

Rekommendation

R.84 Använd pagineringsmarkörer för att märka ut gränser mellan programenheter.

R.85 Upprepa enhetsnamnet i en kommentar för att markera **`begin`** i en paketkropp, underprogramkropp, processkropp eller ett block, ifall **`begin`** föregås av deklarationer.

R.86 För långa eller omfattande nästlade if- och case-satser, markera slutet av satsen med en kommentar som sammanfattar villkoret som styr satsen.

R.87 För långa eller omfattande nästlade if-satser, markera else-delen med en kommentar som sammanfattar villkoret som styr denna del av satsen.

Exempel

```
if    A_Found then
   ...
elsif B_Found then
   ...
else  -- A and B were both not found
   ...
   if Count = Max then
      ...
   end if;
   ...
end if;  -- A_Found
------------------------------------------------------------------------
package body Abstract_Strings is
   ...
   ------------------------------------------------------------------------
   procedure Concatenate (...) is
   begin
      ...
   end Concatenate;
   ------------------------------------------------------------------------

   ...
begin  -- Abstract_Strings
   ...
end Abstract_Strings;
------------------------------------------------------------------------
```

Motivering och förklaring

Markörkommentarer framhäver strukturen hos koden och gör den enklare att ögna igenom.

2.3 Användning av typer

Stark typning främjar pålitlighet hos programvara. Typdefinitionen för ett objekt definierar alla tillåtna värden och operationer och möjliggör för kompilatorn att kontrollera och identifiera potentiella fel under kompilering. Dessutom möjliggör typreglerna för kompilatorn att generera kod för att under körning kontrollera brott mot begränsningar. Användning av dessa egenskaper hos adakompilatorn möjliggör tidig och mer komplett feldetektering än vad som är möjligt för svarare typade språk.

2.3.1 Deklaration av typer

Rekommendationer

R.88 Begränsa intervallet för skalära typer så mycket som möjligt.

R.89 Sök information om möjliga värden i tillämpningen.

R.90 Återanvänd ej några av unbdertypnamnen i paketet Standard.

R.91 Använd undertypdeklarationer för att förbättra programs läsbarheten.

R.92 Använd härledda typer och undertyper i samklang.

Exempel

```
subtype Card_Image is String (1 .. 80);

Input_Line : Card_Image := (others => ' ');

-- restricted integer type:
type    Day_Of_Leap_Year     is                  range 1 .. 366;
subtype Day_Of_Non_Leap_Year is Day_Of_Leap_Year range 1 .. 365;
```

Med följande deklaration säger programmeraren ”jag har inte den blekaste aning om hur många” men det aktuella intervallet kommer att dyka upp begravd i koden eller som en systemparameter.

```
Employee_Count : Integer;
```

Motiv och förklaringar

Undantag

Det finns fall där man ej har ett visst beroende till något intervall av numeriska värden. Sådan situationer dyker ofta upp, t ex, för fältindex (t ex en lista vars storlek ex är fixerad av någon speciell semantik).

2.3.2 Uppräkningstyper

Rekommendationer

R.93 Använd uppräkningstyper istället för numeriska koder.

R.94 Endast om det är absolut nödvändigt, använd representationsklausuler för att möta krav för externa enheter.

Exempel

Använd:

```
type Color is (Blue, Red, Green, Yellow);
```

hellre än:

```
Blue   : constant := 1;
Red    : constant := 2;
Green  : constant := 3;
Yellow : constant := 4;
```

och lägg till följande om nödvändigt:

```
for Color use (Blue   => 1,
               Red    => 2,
               Green  => 3,
               Yellow => 4);
```

Motivering och förklaring

2.4 Sammanfattning

Stavning

•

Namngivningskonventioner

•

Kommentarer

•

Användning av typer

•

3. Referenser

[1] Christine Ausnit-Hood, Kent A. Johnson, Robert G. Pettit IV, Steven B. Opdahl (Eds.). (1997) Lecture Notes in Computer Science; Vol. 1344, *Ada 95 Quality and Style*, Springer.

[2] Ada Reference Manual.

4.

Kompileringsmodellen för Ada bygger på idéen om ett *programbibliotek*, vilket innehåller de programenheter som utgör ett adasystem. Enheterna i programbiblioteket utgörs av sk. *biblioteksenheter*, vilka kan vara paket och underprogram, även generiska sådana. Uppdelning av adaprogram på källkodsfiler är därför i princip av sekundär betydelse men i praktiken kan det finnas praktiska eller kompileringstekniska aspekter av vikt. Om man arbetar direkt mot källkodsfiler är det lämpligt att följa programbibliotekets och enheternas struktur.

- varje enhetsspecifikation skrivs på en egen fil
- varje enhetskropp skrivs på en egen fil

Filnamn (ej inräknat filtypen) väljs utifrån enhetsnamnen, vilket innebär filen med en ehets specifikation har samma namn som filern med dess kropp. Filtypen väljs utifrån om filen innehåller en specifikation eller en kropp. Ett rekommenderat val av filtyp är

- `ads` för specifikationer
- `adb` för kroppar

4.1 Kommentarer

Skriv alla kommentarer som om läsaren är datorkunnig och har grundläggande kunskaper om Ada. Utgå däremot ifrån att läsaren vet nästan inget om vad programmet ska göra. Tänk på att ett kodavsnitt som förefaller intuitivt för dig då du skriver det, kanske inte är det för en annan läsare eller för en själv senare. Dokumentera i enhetlighet med detta.

R.95 Varje fil som innehåller källkod ska dokumenteras inledningsvis där dess namn och innehåll framgår.

Kommentar: Källkod måste kommenteras och det bör göras på ett kompakt och lätt lokaliserbart sätt. En *strategisk* kommentar skrivs före en enhet och beskriver enhetens egenskaper. En *taktisk* kommentar beskriver vad en enskild rad i koden gör och bör, om möjligt, placeras sist på raden. Med systematisk och väl genomtänkt namngivning av enheter, variabler etc. och genom väl strukturerad kod blir behovet av kommentarer i koden litet.

R.96 Varje enhet ska ha en strategisk kommentar som beskriver dess roll.

R.97 Kommentera efter hand som olika programdelar färdigställs.

R.98 Se till att kommentarer är korrekta. Var kortfattad men fullständig.

R.99 Uppdatera kommentarer då ändringar görs i koden.

5. Namngivning

Namngivning är mycket viktigt för läsbarheten. Genom att tillämpa en systematiska namngivning underlättas identifieringen av olika namngivna entiteter och förståelsen av koden.

R.100 Ge namn en semantisk mening. Använd normalt inte enbokstavsnamn (ett tillåtet undantag från den regeln kan vara t ex styrvariabeln i en `for`-sats, om det bara är fråga om att stega igenom ett antal heltalsvärden).

R.101 Försök begränsa namn till en rimlig längd, men eftersträva normalt hela ord.

R.102 Namn inleds med stor bokstav och skrivs för övrigt med små bokstäver, t ex `Put`.

R.103 I namn sammansatta av flera ord, ska orden separeras med ett understrykningstecken och första bokstaven i respektive ord ska skrivas med stor bokstav, t ex `Put_Line`.

6. Kodningsstil

Regler och rekommendationer som kan röra kodningsstil kan även finnas under andra avsnitt.

6.1 Reserverade ord

Reserverade ord ska skrivas med genomgående små bokstäver. Det finns exempel på motsatsen, t ex den gamla referensmanualen för Ada 83 och vissa läroböcker, men detta får anses vara en föråldrad stil som medför sämre läsbarhet.

6.2 Format

R.104 Skriv en deklaration per rad, t.ex. en objektdeklaration eller en enkel sats.

R.105 Indrag ska systematiskt avspegla det logiska nästlingsdjupet och ska vara antingen 3 eller 4 positioner (ej blandat).

R.106 Olika kodstycken (t ex delbearbetningar) avgränsas med en tomrad eller en kommentar.

6.3 Tomrum

Tomrum spelar en viktig roll för att öka läsbarheten hos program. Grundregeln är att använda tomrum i enhetlighet med hur mellanrum sätts in och styckesuppdelning görs i vanlig text. Använd omsorgsfullt tomrum för att dela upp och dra uppmärksamhet till olika delar av program.

R.107 Sätt in *ett* mellanrumstecken efter varje komma i en parameterlista, men ej före.

R.108 Sätt in *ett* mellanrumstecken före och efter varje tvåställig operator.

R.109 Sätt *ej* in mellanrumstecken omedelbart efter en vänsterparentes eller före en högerparentes.

R.110 Sätt *ej* in mellanrumstecken före semikolon.

R.111 Sätt in *ett* mellanrumstecken före en vänsterparentes, utom före en parameterlista.

6.4 Programstruktur allmänt

Ada har en väl genomtänkt och systematisk syntax, vilket bland annat avspeglar sig i att alla konstruktioner, som kan ha det, har en syntaktisk avslutning. Detta förenklar inlärning och kodning samt gör program pålitligare. De sammansatta satserna är bra exempel på detta, och där Ada skiljer sig från många andra språk. If-satsen, t ex, har en tvågrenad form, som inleds med **`if`** (inledningsmarkör), de två grenarna separeras med **`else`** (uppdelningsmarkör) och satsen avslutas med **`end if`** (avslutningsmarkör).

En enhets eller sats' inlednings- och avslutningsrad (markörer) ska skrivas med samma indrag från vänstermarginalen, enligt gällande logiska djup. Eventuella uppdelningsmarkörer som ingår skrivs med samma indrag som inlednings och avslutningsraderna. Övrig kod dras in motsvarande sitt logiska djup i förhållande till enheten eller satsen (vilket är minst ett steg). För en funktion med en tvågrenad if-sats innebär detta följande struktur.

```
function Max(                X, Y : in Integer) return Integer is
   Temp : Integer;
begin
   if X > Y then
      Temp := X;
   else
      Temp := Y;
   end if;

   return Temp;
end Max;
```

Funktionen inleds med **`function`** och avslutas med **`end`** och funktionsnamnet, **`begin`** är en delar upp funktionen i en deklarativ del från en satsdel. Deklarationen i den deklarativa delen, liksom satserna i

funktionens satsdel är indragna ett steg. Tilldelningssatserna i if-satsen är sedan i sin tur indragna ett steg inom if-satsen.

6.5 Meddelanden och ledtexter.

R.112 Var inte nedlåtande.

R.113 Försök inte vara humoristisk.

R.114 Var informativ men kortfattad.

R.115 Ange specifika inmatningsval i ledtexter, i förekommande fall.

R.116 Ange standardval i ledtexter, i förekommande fall.

6.6 Utmatning

R.117 Rubricera all utmatning tydligt.

R.118 Presentera all information för användaren på ett enhetligt sätt.

7. Satser

7.1 Blocksats

Blocksatsen har, i jämförelse med många andra språk, en begränsad användning, eftersom samtliga sammansatta satser i Ada har en syntaktisk avslutning. Block kan namnges och bör syntaktiskt utformas enligt följande. Satsnamnets indrag är lite besvärligt, halvt indrag rekommenderas.

```
blocknamn:
   declare
      lokala deklarationer
   begin
      satser
   exception
      undantagshanterare
   end blocknamn;
```

Den kanske vanligaste användningen av blocksatsen är för att placera in undantagshantering, i vilket fall ej **`declare`**-delen ingår (den är valfri). Denna variant av blocksatsen betecknas ”frame”.

7.2 `If`-satsen

De tre varianterna av **`if`**-satser ska skrivas enligt följande syntax (**`else`** i den tredje varianten med **`elsif`** är valfritt).

```
if uttryck then                    if uttryck then                    if uttryck then
   satser                             satser                             satser
end                                elsif uttryck then                 else
                                      satser                             satser
                                end if;                            elseif …
                                                                      …
                                                                   else
                                                                      satser
                                                                   end if;
```

Uttrycket i if-satserna ska vara ett villkorsuttryck eller boolevariabel.

R.119 Om samtliga uttryck i en **`elsif`**-konstruktion beror av samma variabel bör en **`case`**-sats användas i stället.

R.120 **`if`**-satser bör ha maximal 7 grenar och ett maximalt nästlingsdjup av 3.

Kommentar: Antalet saker som människan klarar av att hantera samtidigt begränsas normalt till 7 2.

7.3 `case`-satsen

`case`-satsen ska skrivas enligt följande syntax. En väljarlista kan bestå av ett värde, en uppräkning av värden separerade med ’|’, ett intervall (*startvärde* **`..`** *slutvärde*) eller namnet på en undertyp till typen för styruttrycket (motsvarar ett intervall).

```
case diskret_uttryck is
   when väljarlista =>
      satser
   when väljarlista =>
      satser
   when others =>
      satser
end case;
```

case-satsen kräver att samtliga värden för styruttryckets typ täcks av väljarlistorna eller genom att ett avslutande **others**-alternativ finns, i vilket värden som ej täcks av värdelistorna hanteras.

7.4 loop-satsen

loop-satser skrivs enligt följande syntax. **loop**-satser kan namnges och detta kan användas för att tydligt markera en **loop**-sats start och slut och även för att låta slingnamnet dokumentera slingans syfte. Annars är slingnamns främsta användning ihop med **exit**-satsen, se nedan, för att i nästlade slingstrukturer ange vilken slinga som man vill avbryta med en viss **exit**-sats.

slingnamn:
    loop
        satser
    end loop *slingnamn*;

För att styra **loop**-satsen finns tre möjligheter, dels **exit**-satsen och dels två till **loop**-satsen hörande sk. *iterationsscheman*, **for** och **while**. **exit**-satsen beskrivs längre ner.

Iterationsschemana är ett slags tillägg till **loop**-satsen och placeras före inledningsmarkören **loop**. **for**-iterationsschemat gör **loop**-satsen till en *räknarstyrd slinga*, och finns i två varianter, en uppräknande (**in**) och en nedräknande (**in reverse**).

slingnamn:
    for *styrvariabel* **in** *startvärde* **..** *slutvärde* **loop**
        satser
    end loop *slingnamn*;

slingnamn:
    for *styrvariabel* **in reverse** *startvärde* **..** *slutvärde* **loop**
        satser
    end loop *slingnamn*;

R.121 **For**-schemat ska användas för repetitioner där antalet varv kan bestämmas då satsen inleds, dvs. då *startvärde* och *slutvärde* kan bestämmas då exekveringen går in i **for**-satsen.

Kommentar: I annat fall används **while** eller **do**-**while**.

while-iterationsschemat ska skrivas enligt följande syntax. Styruttrycket är ett booleuttryck.

slingnamn:
    while *styruttryck* **loop**
        satser
    end loop *slingnamn*;

R.122 **while**-schemat bör endast väljas då styruttrycket på ett naturligt sätt kan förtestas. Eftertestade slingor skrivs företrädesvis med en avslutande **exit**-**when**-sats sist i slingan.

Kommentar: Genom detta undviker man ologiska initieringar av variabler som ingår i uttrycket, vilket ofta är ett tecken på att do-while bör väljas istället.

7.5 exit-satsen

exit-satsen innebär att programmet hoppar ur en omslutande **loop**-sats då den utförs. **exit**-satsen har flera former. Den enklaste formen är enbart **exit;**, och denna kan kombineras vanligtvis med en **for**-sats som testar ett avbrottsvillkor och utför exit-satsen om detta är uppfyllt. Som framgår av exemplet t.v. nedan är detta en klumpig konstruktion och av det skälet finns en speciell **exit**-**when**-variant, se exemplet t.h. nedan. Slingnamnets användning i **exit**-satserna är valfritt och om inget sådant anges, avses den närmast omslutande **loop**-satsen.

slingnamn:
    loop
        …
        if *villkor* **then** **exit** *slingnamn* **when** *villkor*;

slingnamn:
loop
…
exit *slingnamn* **when** *villkor*;

```
            exit slingnamn;                                    …
        end if;                                        end loop slingnamn;
        …
    end loop slingnamn;
```

7.6 goto-satsen

R.123 Använd i aldrig **`goto`** för normal programstyning.

Kommentar: **`goto`** bryter den sekvensiella exekveringen på ett sätt som ofta gör koden svår att förstå. Dessutom finns begränsningar för när **`goto`** kan användas; det är t.ex. ej tillåtet att hoppa förbi en sats som initierar ett lokalt objekt som har en konstruktör.

7.7 return-satsen

R.124 Eftersträva endast en **`return`**-sats i en funktion, på sista raden i funktionen. Om påtaglig förenkling kan erhållas genom flera **`return`**-satser, kan det tillåtas.

Kommentar: Endast en **`return`**-sats stödjer regeln att varje konstruktion ska ha endast en utgång (och endast en ingång).

8. Uttryck

Ada har en relativt begränsad uppsättning operatorer och uttryck kan inte skrivas lika komplicerat som i vissa andra språk.

R.125 Använd parenteser för att förtydliga beräkningsordningen för operatorer och deluttryck i sammansatta uttryck.

9. Klasser

Klasser bör definieras i enlighet följande:

Exempel:
```
class Derived : private Base
{
   public:
      Derived ();                 // standardkonstruktör
      Derived (const Derived&);   // kopieringskonstruktör
      ...
   protected:
      ...
   private:
      ...
};
```

R.126 `public`-, `protected`- och `private`-avdelningarna ska deklareras i den ordningen.

Kommentar: Genom denna deklarationsordning kommer det som är av intresse för användaren först i klassdefinitionen, nämligen `public`-deklarerade medlemmar. Därefter följer i `protected`-delen sådant som är av intresse om man avser att härleda från klassen. Sist `private`-delen som är av minst intresse allmänt.

R.127 Inga medlemsfunktioner ska *definieras* i klassdefinitionen.

Kommentar: Medlemsfunktioner som definieras i klassdefinitionen blir automatiskt ***inline***-funktioner, vilket ibland kan vara önskvärt. Att specificera medlemsfunktioner i klassdefinitionen gör dock lätt att klassdefinitionen blir omfattande och svårare att läsa.Dessutom strider det mot principen att dölja information. Det är därför att föredra att skriva separata `inline`-deklarerade specifikationer för medlemsfunktionerna, vilket också har vissa fördelar. Dels kan en inline-funktion lätt göras om till en vanlig funktion.

9.1 Åtkomsträttighet

R.128 Specificera ej datamedlemmar i en klass som `public` eller `protected`. I gränsytor mot andra språk, som t.ex. C, kan det dock vara nödvändigt att definiera poster (`struct`) med `public`-specificerade datakomponenter.

Kommentar: Att specificera en datamedlem som `public` innebär att användare av klassen fritt kan operera på en sådan datamedlem. Detta strider mot den grundläggande principen i objektorienterad programmering att data ska vara inkapslade och enbart kunna manipuleras på ett kontrollerat sätt, dvs. via klassens medlemsfunktioner. Om `public`-data undviks kan dess interna representation ändras utan att detta påverkar användarens kod.

Användning av `protected`-variabler i en klass rekommenderas ej, eftersom sådana variabler blir synliga i härledda klasser. Namnen på sådana variabler kan då ej ändras, eftersom härledda klasser kan vara beroende av namnen. Om en härledd klass måste komma åt data i föräldraklassen, kan detta lösas genom att göra ett speciell `protected`-gränsyta i basklassen med funktioner som returnerar `private`-data. Detta behöver ej påverka effektiviteten om funktionerna definieras som ***inline***-funktioner.

9.2 *Inline*-funktioner

R.129 Åtkomstfunktioner bör vara `inline`-funktioner.

9.3 Operatoröverlagring

R.130 Använd operatoröverlagring sparsamt och på ett enhetligt sätt.

R.131 När två operatorer är varandras motsats (t ex `==` och `!=`) är det lämpligt att definiera båda.

10. Funktioner

10.1 Funktionsparametrar

R.132 Använd inte ospecificerade funktionsargument (elips, `...`);

R.133 Undvik funktioner med många argument.

R.134 Om en funktion lagrar en pekare till ett objekt som åtkoms via en parameter, ska parametern vara av typen pekare. I annat fall ska referens användas.

R.135 Använd konstanta referenser (`const &`) för värdeanrop, såvida ej parametertypen är av en fördefinierad typ eller en pekare.

10.2 Funktionsöverlagring

R.136 Alla variationer av ett överlagrat funktionsnamn bör ha samma semantik.

10.3 Allmänt

R.137 Undvik långa och invecklade funktioner.

11. Variabler och konstanter

11.1 Variabler

R.138 Variabler ska deklareras med minsta möjliga räckvidd.

R.139 Varje variabel ska deklareras i en separat deklarationssats.

R.140 Varje deklarerad variabel ska ges ett värde innan den används, helst redan i samband med deklarationen.

11.2 Konstanter

R.141 Undvik numeriska värden i koden, använd symboliska konstanter istället. Vissa numeriska värden har en väletablerad och klar mening i ett program och kan då få användas direkt.

12. Pekare och dynamisk minneshantering

12.1 Dynamisk minneshantering

R.142 Tilldela ej minne och förvänta att någon annan ska återlämna det.

13. Undantag

R.143 Använd undantag (*exceptions*) enbart för verkligt exceptionella händelser, hantering av fel eller påtagligt ovanliga eller viktiga situationer.

R.144 Hantera varje undantag på lämplig designnivå.

14. Kodmallar

Dokumentation är ett viktig inslag i framställning av programvara. Här behandlas den form av dokumentation som utgörs av kommentarer i källkoden.

Källkodskommentarer kan delas in i strategiska och taktiska kommentarer:

- en *strategisk kommentar* placeras före ett kodavsnitt, t.ex. före en moduldeklaration eller en funktionsdefinition, och beskriver vad det beskrivna kodavsnittet fyller för funktion.
- en *taktisk kommentar* rör vanligtvis en enskild rad kod. En taktisk kommentar placeras med fördel sist på raden om detta är möjligt, i annat fall före raden ifråga.

Strategiska kommentarer vänder sig i första hand till användare av en modul eller en funktion, medan tatktiska kommentarer vänder sig till den som ska underhålla koden. För mycket taktiska kommentarer gör lätt koden oläslig och bör undvikas, och kan ofta undvikas genom den strategiska kommenteringen, väl valda namn och välskriven kod för övrigt. Endast påtagligt komplicerad kod bör kommenteras taktiskt.

Nedan visas mallar för olika slags filer, inkluderingsfiler, *inline*-definitionsfiler och implementeringsfiler, samt för strategiska kommentarer för funktioner. Att standardisera sättet att kommentera kan göra det möjligt att med lämpliga verktyg automatiskt generera annan dokumentation, t.ex. manualsidor.

14.1 Inkluderingsfiler

```
//--|------------------------------------------------------------------
//--| IDA Programvaruproduktion AB (u.p.a.)
//--|------------------------------------------------------------------
//--| IDENTIFIERING
//--|
//--| Filnamn:      Foo.hh
//--| Enhetsnamn:   Foo
//--| Typ:          Moduldeklaration
//--| Revision:     2.1
//--| Skriven av:   H. Acker
//--|
//--|------------------------------------------------------------------
//--| BESKRIVNING
//--|
//--| Denna modul ...
//--|------------------------------------------------------------------
//--| REVISIONSBERÄTTELSE
//--|
//--| Revision  Datum   Förändringar
//--|
//--| 1         970319  Ursprungsversion
//--| 1.1       970407  ...
//--| ...
//--| 2.0       970821  ...
//--|
//--|------------------------------------------------------------------

#ifndef FOO_HH
#define FOO_HH

//--|------------------------------------------------------------------
//--| REFERERADE BIBLIOTEK OCH MODULER
//--|------------------------------------------------------------------

#include <*.hh>
#include "*.hh"

...

//--|------------------------------------------------------------------
//--| VILLKORLIG INKLUDERING AV INLINE-DEFINITIONSFILEN
//--|
//--| Vid normal kompilering inkluderas inline-definitionsfilen
//--| "Foo.icc" här. Vid kompilering för avlusning inkluderas filen
//--| i stället i "Foo.cc". Detta styrs i kompileringskommandot med
//--| preprocessorflaggan "-D" och "DEBUG" som argument, dvs. "-DDEBUG".
//--|------------------------------------------------------------------

#ifndef DEBUG
  #include "Foo.icc"
#endif

//--|------------------------------------------------------------------
//--| SLUT PÅ FILEN Foo.hh
//--|------------------------------------------------------------------

#endif
```

14.2 Inline-definitionsfiler

```
//--|------------------------------------------------------------------
//--| IDA Programvaruproduktion AB (u.p.a.)
//--|------------------------------------------------------------------
//--| IDENTIFIERING
//--|
//--| Filnamn:     Foo.icc
//--| Enhetsnamn:  Foo
//--| Typ:         Inline-definitioner hörande till modul Foo
//--| Revision:    2.1
//--| Skriven av:  H. Acker
//--|
//--|------------------------------------------------------------------
//--| BESKRIVNING
//--|
//--| Denna definitionsfil...
//--|------------------------------------------------------------------
//--| REVISIONSBERÄTTELSE
//--|
//--| Revision  Datum   Förändringar
//--|
//--| 1         970319  Ursprungsversion
//--| 1.1       970407  ...
//--| ...
//--| 2.0       970821  ...
//--|
//--|------------------------------------------------------------------

//--|------------------------------------------------------------------
//--| FUNKTION f(...)
//--| ...
//--|------------------------------------------------------------------

template <class T>
inline
int
Foo<T>::f(...)
{
   ...
}

//--|------------------------------------------------------------------
//--| FUNKTION ...
//--| ...
//--|------------------------------------------------------------------

template <class T>
inline
...

//--|------------------------------------------------------------------
//--| SLUT PÅ FILEN Foo.icc
//--|------------------------------------------------------------------
```

14.3 Implementeringsfiler

```
//--|------------------------------------------------------------------
//--| IDA Programvaruproduktion AB (u.p.a.)
//--|------------------------------------------------------------------
//--| IDENTIFIERING
//--|
//--| Filnamn:     Foo.cc
//--| Enhetsnamn:  Foo
//--| Typ:         Definitioner hörande till modul Foo, ej inline
//--| Revision:    2.1
//--| Skriven av:  H. Acker
//--|
//--|------------------------------------------------------------------
//--| BESKRIVNING
//--|
//--| Denna implementeringsfil...
//--|------------------------------------------------------------------
//--| REVISIONSBERÄTTELSE
//--|
//--| Revision  Datum    Förändringar
//--|
//--| 1         970319   Ursprungsversion
//--| 1.1       970407   ...
//--| ...
//--| 2.0       970821   ...
//--|
//--|------------------------------------------------------------------

//--|------------------------------------------------------------------
//--| REFERERADE BIBLIOTEK OCH MODULER
//--|------------------------------------------------------------------

#include ”Foo.hh”

//--|------------------------------------------------------------------
//--| VILLKORLIG INKLUDERING AV INLINE-DEFINITIONSFILEN
//--|
//--| Vid kompilering för avlusning inkluderas inline-definitionsfilen
//--| ”Foo.icc” här, samtidigt som inline-deklarationerna avlägsnas.
//--| Vid normal kompilering inkluderas filen i stället i ”Foo.hh”.
//--| Detta styrs i kompileringskommandot med preprocessorflaggan ”-D”
//--| och ”DEBUG” som argument, dvs. ”-DDEBUG”.
//--|------------------------------------------------------------------

#ifdef DEBUG
   #define inline
   #include ”Foo.icc”
   #undef inline
#endif
```

```
//--|------------------------------------------------------------------
//--| LOKALA DEFINITIONER
//--|------------------------------------------------------------------

//--|------------------------------------------------------------------
//--| BESKRIVNING
//--| ...
//--|------------------------------------------------------------------

static ...     // ...

...

//--|------------------------------------------------------------------
//--| LOKALA FUNKTIONER (DEKLARATIONER)
//--|------------------------------------------------------------------

void f(...);   // ...

...

//--|------------------------------------------------------------------
//--| KONSTRUKTÖR Foo()
//--| ...
//--|------------------------------------------------------------------

template <class T>
Foo<T>::Foo()
{
   ...
}

//--|------------------------------------------------------------------
//--| LOKALA FUNKTIONER (DEFINITIONER)
//--|------------------------------------------------------------------

//--|------------------------------------------------------------------
//--| FUNKTION fie(...)
//--| ...
//--|------------------------------------------------------------------

void
fie(...)
{
   ...
}

//--|------------------------------------------------------------------
//--| SLUT PÅ FILEN Foo.cc
//--|------------------------------------------------------------------
```

14.4 Klassbeskrivningar

Nedan visas hur klasser kan dokumenteras i inkluderingsfilen.

```
//--|------------------------------------------------------------------
//--| KLASS X
//--|------------------------------------------------------------------
//--| BASKLASSER
//--|
//--| ...
//--|------------------------------------------------------------------
//--| BESKRIVNING
//--|
//--| ...
//--|------------------------------------------------------------------
//--| TILLSTÅND
//--|
//--| ...
//--|------------------------------------------------------------------
//--| KONSTRUKTÖRER
//--|
//--| ...
//--|------------------------------------------------------------------
//--| OPERATIONER
//--|
//--| ...
//--|------------------------------------------------------------------
//--| DATAMEDLEMMAR
//--|
//--| ...
//--|------------------------------------------------------------------
//--| REVISIONSBERÄTTELSE
//--|
//--| Revision  Datum   Förändringar
//--|
//--| 1         970319  Ursprungsversion
//--| ...
//--|------------------------------------------------------------------
```

```
class X
{
   friend ...
   ...
   public:
    ...
   protected:
    ...
   private:
    ...
}; // X
```

14.5 Funktionsbeskrivningar

I filmallarna ovan har strategiska kommentarer för funktioner markerats med t.ex.:

```
//--|------------------------------------------------------------------
//--| FUNKTION funktionsnamn
//--| ...
//--|------------------------------------------------------------------
```

I realiteten ska en fullständig funktionsbeskrivning finnas, vilken kan se ut enligt följande.

```
//--|------------------------------------------------------------------
//--| FUNKTION fie(int i, ...)
//--|------------------------------------------------------------------
//--| BESKRIVNING
//--|
//--| Denna funktion...
//--|------------------------------------------------------------------
//--| INDATA
//--|
//--| ...
//--|------------------------------------------------------------------
//--| UTDATA
//--|
//--| ...
//--|------------------------------------------------------------------
//--| SIDOEFFEKTER
//--|
//--| ... (helst inga)
//--|------------------------------------------------------------------
//--| UTNYTTJAR
//--|
//--| Modul:     iostream
//--| Modul:     ...
//--| Funktion:  fum
//--| Funktion:  ...
//--|------------------------------------------------------------------
//--| REVISIONSBERÄTTELSE
//--|
//--| Revision   Datum   Förändringar
//--|
//--| 1          970319  Ursprungsversion
//--| ...
//--|------------------------------------------------------------------
```

```
void
fie(int i, ...)
{
   ...
} // fie
```

[3]