Enkel stilguide för Ada

Tommy Olsson, tao@ida.liu.se, Institutionen för datavetenskap, Linköpings universitetet © 1999

Det här häfte innehåller grundläggande rekommendationer för hur man bör skriva program, främst då det gäller *kodningsstil*. Häftet är avsett att användas i inledande programmeringskurser baserade på Ada. Många av rekommendationerna är dock allmängiltiga.

Innehåll

1 Inledning

I programmeringens barndom var effektivitet och minnesutnyttjande A och O. Det var en naturlig följd av dåtidens datorers begränsningar med avseende på minnesstorlek och beräkningskapacitet. Idag är minne och snabbhet i många fall inget problem. I stället har underhåll och återanvändning av programvara blivit centralt och därigenom har läsbarhet och ändringsbarhet kommit att bli en viktig aspekt på programkod.

En stor del av allt programmeringsarbete utgörs av att ändra i program som någon annan skrivit. Därför ska man skriva program på ett enkelt och lättfattligt sätt. Tänk alltid på följande då du skriver program:

- Skriv för människor, inte för datorer!
- Skriv vad du menar!
- Krångla inte till din kod i onödan!

Det är förvisso lättare sagt än gjort. För att skriva bra program krävs att du kan programspråket tillräckligt bra och att du kan använda det väl, samt en hel del rutin. Läs och experimentera (kompilatorn är en bra lärare).

Vi begränsar oss i detta häfte till grundläggande delar av Ada och enklare aspekter på stil och kvalitet. För en mer uttömmande utläggning, se [1].

2 Namngivning

De bästa sättet att skriva begripliga program är att kalla saker och ting vid deras rätta namn. Namn, och därmed program, blir genom bra namn självdokumenterande och du behöver inte ha mängder av förklarande kommentarer. Hur ska man då välja namn? Följande riktlinjer ger i allmänhet bra namn.

- Namn på datatyper ska vara substantiv i singularis:

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

```
type Date is
   record
      Year  : Natural;
      Month : Positive;
      Day   : Positive;
   end record;
```

- Datakomponenter i poster (**record**) ska vara substantiv, se t.ex. `Date` ovan.
- Boolska variabler, konstanter och parametrar ska vara adjektiv eller predikatsuttryck:

```
Empty : Boolean;             -- adjektiv
```

```
List_Is_Empty : Boolean;     -- predikatsuttryck
```

- Variabler, parametrar och konstanter, som ej är boolska, ska vara substantiv i singularis:

```
Today : Day;
```

```
Examination_Date : Date;
```

- Procedurer ska vara verb:

```
procedure Get_Next_Token(The_Token : out Token);
```

```
procedure Create(The_Stack : in out Stack);
```

- Boolska funktioner ska vara predikatsuttryck:

```
function Is_Empty(The_List : in List) return Boolean;
```

```
function Is_Last_Token;
```

- Funktioner som ej är boolska ska vara substantiv:

```
function Length_Of(The_List : in List) return Natural;
```

```
function Pop(The_Stack : in Stack) return Token;
```

- Paket ska ges namn som tyder på en högre nivå av organisation än underprogram, vanligtvis substantiv eller substantivfraser.

```
package Text_Routines is
```

```
package Unbounded_Lists is
```

- Överväg att namnge generiska enheter (generiska underprogram och paket) som om de vore icke-generiska enheter. Välj generiska namn som är generellare än instansierade namn.

Det är naturligtvis enklast att namnge saker på sitt eget modersmål. Ett problem är att de flesta programspråk endast tillåter bokstäver ur det engelska alfabetet i namn. Ada tillhör ett av de få undantagen och referensmanualen [2] säger att nationella tecken ska kunna användas i namn. Att det alltid är så i praktiken ska man dock inte lita på. Hur gör man då med t.ex. svenskans å, ä och ö?

En dålig lösning är att använda a i stället för å och ä, och o i stället för ö, eftersom det lätt ger oönskade effekter. Ett program som behandlar datum kan ha variabler för år, månad och dag och kanske en funktion för att testa skottår. I så fall skulle ”år” bli ”ar” (en ytenhet), ”månad” bli ”manad” och ”skottår” bli ”skottar” (en nationalitet eller något man gör med snö). En annan lösning vore att skriva ”aa”, ”ae” och ”oe” i stället för å, ä respektive ö, men det är inte heller någon bra lösning.

Det bästa är att använda engelska. Eftersom reserverade ord och fördefinierade namn i programspråken är på engelska har detta dessutom fördelen att programmen blir språkligt enhetliga. Att hitta bra engelska namn är dock kanske inte så lätt och man bör därför ha en svensk-engelsk ordlista till hands.

Använd meningsfulla och rättvisande namn

En variabel som anger antalet bilar i ett bilregister ska naturligtvis inte kallas `X` eller `Kurt`, utan `Antal_Bilar`. I korta och lätt överblickbara slingor, t.ex., kan man ibland tillåta sig att använda enbokstavsnamn på styrvariabler (vanligtvis `I`, `J`, `K`, osv.):

```
for I in 1 .. Number loop
   Put(Vector(i));
   New_Line;
end loop;
```

Om man har nästlade slingor kan det bli svårt att hålla ordning på sådana namn och då använder man naturligtvis tydligare namn, t.ex.:

```
for Row in 1 .. Number_Of_Rows loop
   for Column in 1 .. Number_Of Columns loop
      Put(Matrix(Row, Column));
      New_Line;
   end loop;
end loop;
```

I vissa fall kan också parameternamn med viss fördel väljas korta:

```
procedure Exchange(X, Y : in out Integer) is
   Old_X : constant Integer := X;
begin
   X := Y;
   Y := Old_X;
end Exchange;
```

I det här fallet är det även (tvärt emot vad som annars är fallet) en fördel med neutrala namn som `X` och `Y`. Vad för slags variabler som det ska bytas värde på är inte intressant i proceduren `Exchange`.

Att använda namn som i sig är meningsfulla men som avser fel sak är förstås förödande. Om en procedur som skriver ut *alla* bilar i ett bilregister kallas för `Print_car` i stället för, som sig bör, `Print_All_Cars` eller möjligtvis `Print_Cars` förleds man.

Meningsfulla namn kan ibland innebära långa namn och det är inte bra med allt för långa namn. Försök i sådana fall hitta en kortare synonym.

Undvik förkortningar

En procedur som skriver ut alla bilar ska naturligtvis inte heta `blaha`, men den bör inte heller ha en kryptisk förkortning som `PAC` (**p**rint **a**ll **c**ars). Kalla funktionen `Print_All_Cars`. Riktlinjer:

- använd inte en förkortning om det finns en kortare synonym som kan användas
- använd inte tvetydiga förkortningar
- använd endast förkortningar som är väl accepterade i tillämpningsdomänen
- använd en konsekvent förkortningsstrategi

Egna påhittade förkortningar ska man vara mycket försiktig med. Sådana kan lätt feltolkas av andra.

Följ konventioner

I många programspråk har vissa konventioner utvecklats. Det är inte förbjudet att bryta mot dem men du ska ha goda skäl för att göra det. Till exempel finns det i Ada allmänt accepterade konventioner för vad som skrivs med stora respektive små bokstäver, att man använder understrykningstecken i namn som består av flera ord, etc.

I Ada föredrar man enhetlighet i många avseenden, även då det gäller hur man bör skriva namn:

- reserverade ord skrivs med enbart gemener
- alla andra namn skrivs med inledande versal, övriga gemena
- understrykningstecken används för att åtskilja namn som är sammansatta av flera ord och då inleds varje ingående ord med versal
- förkortningar skrivs med versaler, t.ex. som `IO` i `Ada.Text_IO`.

Referensmanualen och Rationalen för Ada 95 har många bra exempel på namngivning av olika saker.

3 Kommentarer

Syftet med kommentarer är att hjälpa en läsare att förstå koden. Felstavade, grammatiskt felaktiga, tvetydiga eller ofullständiga kommentarer motverkar detta syfte. Om det är det värt mödan att lägga till en kommentar är det också värt mödan att se till att den är korrekt. Eftersom kommentarer också måste underhållas är det viktigt att tänka på det då man ändrar i koden och det kan också vara ett argument för att inte kommentera mer än nödvändigt.

Skriv källkod med självförklarande namn och använd enkla, begripliga programstrukturer. Då reduceras behovet av kommentarer. Dåligt skriven kod är svår att förstå, underhålla och återanvända, oberoende av kommentarer.

Använd endast kommentarer för att förklara sådant som inte kan beskrivas i koden och för att framhäva det som är speciellt intressant att uppmärksamma. Upprepa aldrig sådant som direkt kan utläsas ur koden. Där kommentarer behövs ska de vara koncisa, kompletta och lätt kunna särskiljas från koden.

Att skriva bra kommentarer är svårt. Är man y som programmerare är det extra svårt att skriva bra kommentarer, eftersom t.ex. kommentarer som rör implementering ska vara skrivna för läsare som behärskar programspråket väl. Ofta ser man i nybörjarprogram kommentarer som beskriver detaljer och sådant som är uppenbart av koden, inte sällan omformulering av exakt vad koden säger. Försök tänka att du är en kunnig programmerare som ska sätta dig in ett program som någon annan har skrivit, t.ex. för att rätta fel eller göra ändringar. Skriva kommentarer utifrån vad du själv skulle vilja veta i den situationen.

Kommentarer i filhuvuden

Varje källkodsfil bör ha ett filhuvud där det framgår vem som äger filen, vem som ansvarar för filen, samt en revisionshistorik. Ett exempel hur en kommentar i ett filhuvud kan se ut:

```
--|-------------------------------------------------------------
--| Copyright (c) 1998, IDA Software.
--|
--| Ansvarig : H. Acker
--| Avdelning: Programvara och system
--|
--| Revisionshistorik:
--|   98-12-24 C. Racker
--|     Lade till funktionen Delete.
--|     Rättade ett fel i funktionen Find, som gav Constraint_Error.
--|   98-11-03 H. Acker
--|     Originalversion.
--|-------------------------------------------------------------
```

I Ada hittar vi främst två slags källkodsfiler, sådana som innehåller enhetsspecifikationer och sådana som innehåller motsvarande kroppar. Specifikationer riktar sig till användare av koden, medan innehållet i kroppar främst är av intresse för den som ska underhålla koden. Detta bestämmer vad kommentarer i respektive typ av fil ska innehålla, utöver vad som angivits ovan.

Kommentarer i filer med enhetsspecifikationer

En kommentar i en specifikationsfil är till för att hjälpa en användare att förstå hur enheten ifråga ska *användas*. Genom att läsa kommentarer och kod ska användaren i princip få reda på allt som behövs för att kunna använda enheten. Denna typ av kommentarer bör omfatta:

- enhetsnamnet eller liknande.
- en kortfattad förklaring av syftet (*vad* som görs, *inte* hur eller varför).
- en beskrivning av hur underprogram kan användas och vilka effekter de har.
- eventuella begränsningar, globala effekter, för- och eftervillkor, prestanda.
- fel som kan inträffa och hur dessa kan detekteras.

Ta *inte* med information om användning som framgår av koden. Räkna inte upp parametrar. Exempel:

```
--|-------------------------------------------------------------
--| graphlayout
--|
--| Syfte:
--|   Den här enheten beräknar positionsinformation för noder och bågar
--|   i en riktad graf. Detta grundas på en algoritm som minimerar
--|   antalet korsande bågar och framhäver den primära riktningen för
--|   bågarna i grafen.
--|
--| Effekter:
--|  - Förväntat användningssätt är:
--|    1. Anropa define för varje nod som ska ingå i grafen.
--|    2. Anropa connect för varje par av noder som ska vara förbundna
--|       med en båge.
--|    3. Anropa layout för att tilldela varje nod i grafen en position.
--|    4. Anropa position_of för att erhålla en nods position.
--|  - Operationen layout kan anropas upprepade gånger och beräknar då
--|    om positionerna för samtliga noder.
--|  - Då en nod definierats, kommer den att vara definierad till dess
--|    clear anropas för att radera hela grafen.
--|-------------------------------------------------------------
```

Hur datatyper, underprogramdeklarationer, etc., som utgör koden i specifikationsfilen kommenteras beskrivs nedan.

Kommentarer i filer med enhetskroppar

En kommentar i en fil med en enhetskropp är till för att hjälpa en underhållare att förstå implementeringen, även val som gjort mellan olika implementeringsalternativ. Dokumentera alla beslut som tagits vid implementeringen, så att en underhållare kan undvika att göra om samma misstag som du gjort. En viktig typ av information är en beskrivning av varför en viss lösning som övervägts inte skulle fungera. Portabilitetsaspekter hör också hemma här. Den här typen av kommentarer bör omfatta:

- enhetens namn eller liknande
- förklaringar av *hur* och *varför* enheten utför sin uppgift, inte vad den gör
- korta sammanfattningar av komplexa algoritmer som används
- anledningar till påtagliga eller kontroversiella implementeringsbeslut
- förkastade implementeringsalternativ, tillsammans med orsaken till detta
- förutsedda framtida förändringar

Ta inte med sådan information som framgår av koden. Exempel på kommentar till enhetskropp:

```
--|-------------------------------------------------------------
--| Graphlayout
--|
--| Implementeringsnoteringar:
--|   Denna enhet använder en heuristisk algoritm för att minimera
--|   antalet korsande bågar. Den uppnår inte alltid det verkliga
--|   minimum som kan räknas ut teoretiskt. Den uppnår dock ett nästan
--|   perfekt resultat på förhållandevis kort tid. För detaljer om
--|   algoritmen, se …
--|
--| Portabilitetsaspekter:
--|   - Standardbiblioteket Math används för beräkning av koordinater.
--|   - 32-bitars heltal krävs.
--|   - Inga operativsystemspecifika rutiner anropas.
--|
--| Förutsedda ändringar:
--|   - Coordinate_Type skulle kunna ändras från heltal till flyttal
--|     med liten arbetsinsats. Åtgärder har vidtagits för att ej vara
--|     beroende av heltalsaritmetiska egenskaper.
--|-------------------------------------------------------------
```

Hur underprogramdefinitioner, etc., som utgör koden i enhetskroppen, kommenteras beskrivs nedan.

Kommentarer till underprogram

Ett underprogram kan delas upp i en specifikation och en kropp. Specifikationen riktar sig till användare av underprogrammet, medan kroppen är av intresse för den som ska underhålla koden.

Kommentarer till underprogramspecifikationer

En kommentar till en underprogramsspecifikation ska beskriva syftet med underprogrammet och vad det gör. Om det är helt uppenbart vad ett underprogram gör och om specifikationen är självförklarande behöver man inte kommentera enligt ovan, utan kan t.ex. låta en övergripande kommentar vara nog.

```
--|-------------------------------------------------------------
--| Define
--|
--| Syfte:
--|   Den här proceduren definierar en nod i den aktuella grafen.
--| Undantag:
--|   Node_Already_Defined
--|-------------------------------------------------------------
procedure Define(New_Node : in Node);
```

```
--|------------------------------------------------------------------
--| Layout
--|
--| Syfte:
--|   Den här proceduren tilldelar de definierade noderna koordinat-
--|   positioner.
--| Undantag:
--|   Inga.
--|------------------------------------------------------------------
procedure Layout();

--|------------------------------------------------------------------
--| Position_Of
--|
--| Syfte:
--|   Den här funktionen returnerar positionen för den angivna noden.
--|   Om ingen position ännu tilldelats noden returneras den förvalda
--|   koordinatpositionen (0,0).
--| Undantag:
--|   Node_Not_Defined.
--|------------------------------------------------------------------
function Position_Of(The_Node : in Node) return Position;
```

Kommentarer till underprogramkroppar

En underprogramkropp ska ha kommentarer som riktar sig till en underhållare av koden.

```
--|------------------------------------------------------------------
--| Define
--|
--| Implementeringsnoteringar:
--|   Den här proceduren lagrar en nod i den ordinära Graf-datastruk-
--|   turen, inte i Fast_Graph-datastrukturen, därför att …
--|------------------------------------------------------------------
procedure Define(New_Node : in Node) is
   …
end Define;

--|------------------------------------------------------------------
--| Layout
--|
--| Implementeringsnoteringar:
--|   Den här proceduren kopierar Graf-datastrukturen (optimerad för
--|   snabb slumpmässig åtkomst) till Fast_Graph-datastrukturen
--|   (optimerad för snabb sekventiell iteration), utför Layout och
--|   kopierar sedan tillbaka till Graf-datastrukturen. Denna teknik
--|   infördes som en optimering då algoritmen befanns för långsam
--|   (snabbheten förbättrades genom detta med en faktor 10).
--|------------------------------------------------------------------
procedure Layout() is
   …
end Layout;

--|------------------------------------------------------------------
--| Position_Of
--|------------------------------------------------------------------
function Position_Of(The_Node : in Node) return Position is
   …
end Position_Of;
```

Om en funktion är enkel och koden självförklarande behöver man inte ha någon kommentar av ovanstående slag.

Kommentarer till data

Kommentarer till data ska förklara syfte med, struktur hos och semantik för datastrukturer. Att förstå hur data lagras och hur olika data hänger ihop kan vara ett första steg i att förstå övriga detaljer i ett program. Riktlinjer:

- kommentera datatyper, variabler och konstanter, såvida inte deras namn är självförklarande.
- förklara alltid hur komplicerade, pekarbaserade datastrukturer byggs upp (datatyperna enbart säger ofta inte allt om struktur i sådana fall).
- beskriv eventuella samband mellan olika dataelement.

Exempel på kommentering av data:

```
--|-------------------------------------------------------------------
--| Dessa datastrukturer används för att lagra grafen under layout-
--| processen. Den övergripande organisationen är en sorterad lista
--| av ranker, var och en innehållande en sorterad lista av noder,
--| var och en innehållande en lista av inkommande bågar och en lista
--| av utgående bågar.
--|
--| Rank- och nodlistorna är dubbellänkade för att tillåta att listan
--| genomlöps i båda riktningarna under sorteringspassen. Båglistorna
--| behöver ej vara dubbellänkade eftersom bågordningen är oväsentlig.
--|
--| Noderna och bågarna är dubbellänkade till varandra för att tillåta
--| effektiv uppsökning av alla bågar till/från en nod, liksom effektiv
--| uppsökning av en båges start/slutnod.
--|-------------------------------------------------------------------------

type Arc;

type Arc_Pointer is access Arc;

type Node;

type Node_Pointer is access Node;

type Node is
   record
      ID       : Node_Pointer;   -- unik nodidentitet, anges av användaren
      Arc_In   : Arc_Pointer;
      Arc_Out  : Arc_Pointer;
      Next     : Node_Pointer;
      Previous : Node_Pointer;
   end record;

type Arc is
   record
      ID     : Arc_ID;          -- unik bågidentitet, anges av användaren
      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;      -- beräknat ordningstal för ranken
      First_Node : Node_Pointer;
      Last_Node  : Node_Pointer;
      Next       : Rank_Pointer;
      Previous   : Rank_Pointer;
    end record;

First_Rank : Rank_Pointer;
Last_Rank   : Rank_Pointer;
```

Kommentarer till satser

Kommentarer till satser bör endast förekomma för att dokumentera kod som ej är portabel, beroende av miljön eller på något sätt ”tricksig”. Den här typen av kommentarer ska tydligt synas i koden. Riktlinjer:

- använd minimalt med kommentarer bland satser
- använd kommentarer endast för att förklara kodavsnitt som inte är uppenbara
- kommentera sådant som normalt borde förekomma i ett visst sammanhang men som av någon anledning medvetet utelämnats
- använd inte kommentarer som bara är en omskrivning av kod
- använd inte kommentarer för att beskriva kod på annan plats, t.ex. för att beskriva en funktion som anropas av aktuell kod

4 Användning av variabler och konstanter

Använd symboliska konstanter och namngivna datatyper

Genom symboliska konstanter och namngivna datatyper blir program både lättare att läsa och enklare att ändra i, eftersom man ofta bara behöver ändra på ett ställe.

- använd symboliska konstanter i stället för literaler, då så är möjligt.
- använd de fördefinierade konstanterna `Ada.Numerics.Pi` och `Ada.numerics.e` för de matematiska konstanterna Pi och e.
- använd attribut som `'First` och `'Last` i stället för literaler, då så är möjligt.

Exempel:

```
3.14159_26525_89739                              -- litteral
Hash_Table_Size  : constant Integer := 997;      -- konstant
Avogadros_Number : constant := 6.022137 * 10**23; -- namngivet tal

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

Begränsa variablers räckvidd

En variabel ska definieras ”så nära där den används som möjligt” och det inte ska gå att komma åt den på andra ställen i programmet.

- använd parametrar och returvärden för att överföra värden till och från enheter.
- en variabel som endast behövs i en viss enhet ska deklareras i enheten.

Återanvänd inte variabler

Om du i en funktion först behöver arbeta med antal bilar och sen med antal ägare, ska du definiera två variabler kallade `Number_Of_Cars` och `Number_Of_Owners`). Du vinner inget på att bara definiera en variabel (kallad t. ex. `Number`_Of_Cars_Or_Owners) och använda den för bägge ändamålen. Såna saker (att inte använda mer minne än som behövs) fixar kompilatorn åt dig.

5 Val av selektionssatser

Valet mellan **if** och **case** är vanligtvis inget problem. En **if**-sats styrs av ett villkorsuttryck som kan formuleras fritt. En **case**-sats kan bara användas när styrningen grundas på en diskret variabels värde.

Om man kan välja en **case**-sats bör man göra det. Den garanterar ju att alla tänkbara värden som styruttrycket kan anta har ett motsvarande läge **case**-satsen, om inte annat **others**. Adas **case**-sats kan ju också skrivas mycket elegant med värdeuppräkning, värdeintervall och även undertypsnamn som lägen.

6 Val av repetitionssatser

I Ada finns endast en **sats** för repetition, **loop**-satsen. Denna är en enkel sluten slinga. För att konstruera vanliga typer av repetitioner finns iterationsschemana **for** och **while**, samt **exit**-satsen med vilken man kan hoppa ur **loop**-satsen.

Principiellt brukar man tala om *räknarstyrda* respektive *villkorsstyrda* slingor. En räknarstyrd slinga stegar igenom en följd av diskreta värden, t.ex. heltalen från 1 till 100 eller tecknen från ’A’ till ’Z’. En villkorsstyrd sling styrs av ett villkor, vilket testas i varje varv. Villkorsstyrda slingor kan delas upp i två typer, *förtestade* och *eftertestade*.

Med **for** konstruerar du *räknarstyrda* slingor. Med **while** konstruerar du *förtestade* villkorsstyrda slingor. Med **exit** kan du konstruera *eftertestade villkorsstyrda* slingor och även slingor som ska avbryta ”mitt i”.

Slingor ska vara korta. Det är svårt att få överblick över en slinga som sträcker sig över många rader kod. Dela upp innehållet i komplicerade slingor med hjälp av underprogram.

Räknarstyrd repetition

En räknarstyrd slinga stegar igenom en följd av diskreta värden. Antalet varv i slingan är känt när man går in i slingan, genom ett startvärde (t.ex. 1) och ett slutvärde (t.ex. 100). Dessa värden kan bestämmas av uttryck som beräknas då slingan initieras, men under själva repetitionen kan inte slutvärdet ändras. För att styra repetitionen används en styrvariabel, som stegas från startvärdet till slutvärdet med steget 1 (eller motsvarande).

```
for I in 1 .. Last_Value loop
   Put(I, Width => 8);
   New_Line;
end loop;
```

Villkorsstyrd repetition

Villkorsstyrd repetition används vanligtvis då man inte på förhand vet hur många gånger en repetition ska utföras, utan det bestäms under själva repetitionen. Villkorsstyrda repetitioner kan delas upp i två typer, *förtestade* och *eftertestade*.

Förtestad villkorsstyrd repetition

I en förtestad repetition kan det tänkas att satserna i slingan inte ska utföras någon enda gång, och testet om repetitionen ska fortsätta görs då allra först. Ett typiska exempel där denna typ av repetition används är vid läsning av filer. Filer kan dels vara tomma och dels vet vi ofta inte hur mycket data som finns i en fil, utan får läsa till filen tar slut. Exempel:

```
while not End_Of_File(Data_File) loop
   Get(Data_File, X);
   Sum = Sum + X;
end loop;
```

Ett annat typiskt fall där förtestade slingor används är stegning genom länkade listor. Dessa kan dels vara tomma och dels lagrar vi ofta inte längden på dem utan får testa när listan tar slut. Exempel:

```
P : List_Node_Ptr := List;

while P /= null loop
   Put(P.Data); New_Line;
   P := P.Next;
end loop;
```

Eftertestad villkorsstyrd repetition

I en eftertestad repetition ska satserna i slingan utföras minst en gång och testet om fler varv ska utföras görs sist i slingan. I Ada finns ingen specifik sats eller något iterationsschema för detta (som t.ex.

Pascals **repeat**-**until** eller C/C++:s **do**-**while**). I Ada använder man en **loop**-sats med en **exit**-sats allra sist:

```
loop
   <satser>
   exit when <villkor>;
end loop;
```

Loop-and-a-half

I samband med t.ex. interaktiv inmatning dyker ofta följande konstruktion upp, där villkor för om slingan ska avbrytas av logiska skäl hamnar inne i slingan; det finns satser både före och efter. Denna konstruktion är så vanlig att den fått ett namn som på engelska är ”*loop-and-a-half*”:

```
loop
   Put(”Skriv ett heltalsvärde större än 0: ”);
   Get(Value);

   exit when Value > 0;

   Put_Line(”Felaktigt värde, ” & Integer’Image(Value) & ”, försök igen!”);
end loop;
```

Detta sätt, att med **exit**-satsen hoppa ur en slinga får anses helt legitimt. Det framgår tydligt att man ska leta efter slingstyrning inuti slingan, genom att slingan inleds med enbart **loop**. Däremot ska du vara restriktiv med att kombinera ett iterationsschema (**for** eller **while**) med **exit**. Du ska ha goda skäl för att göra det.

7 Uppdelning i underprogram

Dela upp program i små, enkla underprogram. Det är ett utmärkt sätt att erhålla hanterbara program och också att begränsa olika variablers räckvidd.

Varje underprogram bör göra *en* ”sak”. Denna ”enda sak” kan dock vara komplicerad och kräva många rader kod. All kod behöver inte ligga i underprogrammet i fråga; dela upp i flera (hjälp)underprogram!

Det ska helst gå att beskriva vad ett underprogram gör i en mening, t.ex. ”denna procedur sorterar heltal i ett fält i stigande ordning”. Om beskrivningen blir lång eller komplicerad har antagligen för mycket, som kanske egentligen inte hör dit, lagts in i underprogrammet. Dela i så fall upp det i beståndsdelar.

Undvik att upprepa kod. Dels blir det onödigt mycket att läsa (och skriva), och dels måste du komma ihåg att ändra på alla ställen om du ska ändra något i koden. Samla kod som upprepas på flera ställen i ett program i ett underprogram, som kan anropas från de ställen där koden ska utföras.

8 Strukturerad programmering

Använd strukturerad programmering. Detta betyder att du ska använda de konstruktioner för repetition och val (**loop**, **if**, **case** och andra sammansatta satser) som finns i språket, och helt undvika hopp med **goto** (det finns i princip ingen anledning att använda **goto** i handskrivna Ada-program).

En del programmerare tycker att man ska vara försiktig med **return**-satsen, eftersom den fungerar som ett hopp ut ur ett underprogram. Varje **return**-sats blir en utgång och ju fler utgångar, desto svårare att överblicka hur ett underprogram beter sig. Varje underprogram borde alltså endast returnera en gång, allra sist. Det sätt man kan behöva koda för att åstadkomma det kan dock ibland ge invecklad kod.

9 Formatering av kod

Kodformatering påverkar kodens utseende, inte vad koden gör. Formatering gäller t.ex. hur många steg man bör göra indrag från vänstermarginalen, hur man bör använda mellanrumstecken på enskilda rader, hur ska man bör sätta in tomma rader, hur långa rader man maximalt bör skriva, hur man bör placera de reserverade ord som ingår i olika konstruktioner.

Det finns olika stilar som utvecklats för olika språk, alla med sina för- och nackdelar. Syftet med dessa är att göra program läsbara, genom att framhäva programstrukturen och genom att göra enskilda rader lätta att läsa. Programstruktur kan du framhäva med tomma rader mellan kodavsnitt och med indrag från vänstermarginalen. Lättlästa rader får du genom att sätta in mellanrum på ett genomtänkt sätt.

En fördel med Ada är att dess syntax är så väldesignad och regelbunden att det inte finns speciellt mycket att diskutera om hur bra adakod bör skrivas. Referensmanualen för Ada 95 [2] visar hur kod bör skrivas och [1] behandlar ingående hur man skriver kod för hög kvalitet och med god stil. För övrigt:

- Var konsekvent!
- Om du ändrar i ett program som någon annan skrivit, följ den stil som använts i programmet.

Det finns program som formaterar kod, åtminstone så att alla rader får korrekt indrag. Inget program kan dock göra ett lika bra jobb som du själv kan!

Indrag från vänstermarginalen

Indrag från vänstermarginalen använder du för att framhäva den logiska strukturen i program. Många anser att tre positioner är ett optimalt indrag och att man generellt bör hålla sig mellan 2-4 steg. Här är några rekommendationer:

- Var konsekvent då du gör indrag av nästlade styrstrukturer, fortsättningsrader och inbäddade enheter (en enhet som deklareras i den deklarativa delen av en annan enhet).
- Särskilj nästlade styrstrukturer från fortsättningsrader genom olika indrag; använd tre steg för de förra, två steg för de senare.
- Skriv satsetiketter, dvs namn på satser (<*namn*>:) och hopplägen (<<*etikett*>>), på föregående indragsnivå i förhållande till den sats de relaterar till.

Här följer exempel på hur indrag görs för olika satser mm. enligt ovanstående riktlinjer och vad som rekommenderas i [2].

Posttyper skrivs på följande sätt:

```
type <namn> is
   record
      <komponent>
      <komponent>
   end record;
```

Satsettiketter (dras tillbaka tre steg) och fortsättningsrader (dras in två steg):

```
begin                                  <lång sats med radbrytning>
<<etikett>>                              <fortsätter med två teckens indrag>
   <sats>
end;
```

If-satsen och den enkla slingan (namngiven; namnet dras tillbaka tre steg) med **exit**-sats :

```
if <villkor> then                      <namn>:
   <satser>                            loop
elsif <villkor> then                      <satser>
   <satser>                               exit <namn> when <villkor>;
else                                      <satser>
   <satser>                            end loop;
end if;
```

Namngivna slingor (namnen dras tillbaka tre steg) med iterationsschema, **for** respektive **while**:

```
<namn>:                                <namn>:
   for <iterationsshema> loop          while <villkor> loop
      <satser>                            <satser>
   end loop <namn>;                    end loop <namn>;
```

Blocksatsen (namngiven) och **case**-satsen, den senare varianten för mycket korta sats-listor:

```
<namn>:
   declare
      <deklarationer>
   begin
      <satser>
   exception
      when <val> =>
         <satser>
      when others =>
         <satser>
   end <namn>;
```

```
case <uttryck> is
   when <val> =>
      <satser>
   when <val> =>
      <satser>
   when others =>
      <satser>
end case;

case <uttryck> is
   when <val>  => <satser>
   when <val>  => <satser>
   when others => <satser>
end case;
```

Enhetsspecifikationer (korta funktionsspecifikationer behöver inte brytas före **return**):

```
procedure <specifikation>;

function <specifikation>
  return <typ>
```

```
package <namn> is
   <deklarationer>
private
   <deklarationer>
end <namn>;
```

Enhetskroppar (**exception**-delen i respektive enhet är valfri, liksom satsdelen i paketkroppen):

```
procedure <specifikation> is
   <deklarationer>
begin
   <satser>
exception
   when <val> =>
      <satser>
end <namn>;
```

```
package body <namn> is
   <deklarationer>
begin
   <satser>
exception
   when <val> =>
      <satser>
end <namn>;
```

```
function <specifikation>
  return <typ> is
   <deklarationer>
begin
   <satser>
exception
   when <val> =>
      <satser>
end <namn>;
```

Kontextklausuler (**with**, för paket även **use**) för kompileringsenheter (t.ex. en underprogramspecifikation eller en paketkropp enligt ovan) skrivs i tabellform ovanför enhetsdeklarationen:

```
with <namn>; use <namn>;
with <namn>;
<kompileringsenhet>
```

Generiska enhetsspecifikationer (de generiska formella parametrarna skrivs ovanför själva enheten och skymmer därigenom ej denna) och instansdeklarationer:

```
generic
   <formella parametrar>
<generisk enhet>
```

```
package <namn>
   is new <generisk enhetsnamn> <argument>
```

Var noggrann! Det är förödande för läsbarheten om inte programrader har korrekt indrag, i förhållande till varandra och till det logiska nästlingsdjupet. Abetar man i ett verktyg för programmering finns vanligtvis stöd för att erhålla korrekt indrag då man ska skriva en ny rad.

Formatering av rader

Skriv normalt endast en ”sak” per rad, t.ex. en enkel sats eller en deklaration. Det är ofta svårt att läsa multipla deklarationer och rader med flera satser. Räta upp koden på sammanhängande rader enligt följande, för att göra koden lättare att läsa:

```
Values : array (1 .. 100) of Float;
Sum    : array (1 .. 10)  of Float;
Mean   : array (1 .. 10)  of Float;
Number : Integer := 0;
X      : Float;
```

Långa rader delar du upp genom att välja så vettiga brytpunkter som möjligt:

- i långa uttryck, t.ex. i en **if**-sats, gör du radbrytning mellan deluttryck; om brytningen bör göras före eller efter en operator kan bero på typen av uttryck. Dra in så att deluttrycken på de raderna börjar i samma position.
- långa parameterlistor delar du lämpligen upp genom att skriva *en* parameter per rad (den första på samma rad som enhetsnamnet) och bryta efter varje komma. Gör indrag av raderna så att parametrarnas namn börjar i samma position och rikta även upp de kolon som följer.
- bryt efter komma i långa argumentlistor anrop. Dra in så att argumenten får samma vänstermarginal.

Insättning av mellanrum

Använd mellanrum för att öka läsbarheten på enskilda rader. Ett exempel på dåligt skriven kod:

```
if x*y>1 and then(z+2*w<0 or else z+2*w>10)then
```

Denna oläsliga gröt blir betydligt mer lättläst om du använder mellanrum för att separera namn och operatorer:

```
if x * y > 1 and then (z + 2 * w < 0 or else z + 2 * w > 10) then
```

Sätt in ett mellanrumstecken på följande ställen (flera mellanrum kan bli aktuellt att sätta in om du vill räta upp koden på ett visst sätt i vissa situationer):

- före och efter följande avgränsare och tvåställiga (binära) operatorer:

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

- på utsidan om avgränsarna för stränglitteraler (`"`) och teckenliteraler (`'`), utom där detta förhindras av annan regel (t.ex. följande)
- på utsidan om men ej innanför parenteser
- efter komma (`,`) och semikolon (`;`)

Sätt *ej* in något mellanrumstecken på följande ställen, även då det står i konflikt med ovanstående rekommendationer:

- efter plus- (`+`) och minustecken (`-`) som används som enställiga (unära) operatorer
- efter ett funktionsanrop
- innanför satsetikettavgränsarna (`<<` och `>>`)
- före och efter exponentieringsoperatorn (`**`), apostrof (`'`) och punkt (`.`)
- mellan direkt på varandra följande inledande eller avslutande parenteser
- före komma (`,`) och semikolon (`;`)

När redundanta parenteser utelämnas p.g.a. operatorprioritetsregler, kan som ett alternativ mellanrum utelämnas runt de operatorer i uttrycket som har högst prioritet, t.ex. runt * i följande exempel:

```
if x*y > 1 and then (z + 2*w < 0 or else z + 2*w > 10) then
```

Insättning av tomma rader

Tomma rader använder du för att dela upp längre kodavsnitt i delmoment eller beståndsdelar, t.ex.:

- mellan enheter som är skrivna på samma fil, för att tydligt visa var de börjar och slutar
- mellan olika delar i en satsdel, för att t.ex. påvisa olika delmoment som utförs

Var systematisk med antalet tomma rader i olika sammanhang. Överdriv inte antalet! Inuti satsdelar finns det sällan anledning att ha mer än *en* tom rad då man vill göra en uppdelning. Se också till att inga helt omotiverade tomma rader finns; det ger fel information och förvirrar läsaren.

10 Modularisering och abstraktion

De här avsnittet handlar mer om programmeringsmetodik än om kodningsstil men det är också av stort intresse att ta upp i sammanhanget.

Moduluppdelning

Att modularisera innebär att dela upp en stor uppgift i flera mindre delar. En fördel med modularisering är att det är lättare att lösa många små uppgifter var för sig än att lösa en stor uppgift. Om man gör uppdelningen på ett bra sätt, kan man ofta lösa varje del utan att behöva tänka på de andra delarna.

En annan fördel med uppdelning i mindre delar är att man kan testa varje del för sig. Då varje del fungerar som den ska, kan man slå ihop alla delarna till ett (förhoppningsvis) fungerande program.

Ytterligare en fördel med modularisering är att man kan återanvända en modul, t.ex. genom att använda den till flera olika saker i samma program eller att i framtiden använda den i andra program.

Varje modul ska vara starkt sammanhållen och kopplingen mellan moduler ska vara svag. Att en modul är starkt sammanhållen betyder att den gör *en* sak och gör denna fullständigt. Denna sak kan vara komplicerad och bestå av delmoment, t.ex. kan en moduls gränssnitt bestå av flera underprogram, som var och en utför en viss del. Att en modul är svagt kopplad innebär att den ej är beroende av hur andra moduler är implementerade.[1]

En modul kan typiskt utgöras av en uppsättning relaterade underprogram, eller en eller flera datatyper med tillhörande operationer (underprogram). Datatyp(er) och underprogramspecifikationer placeras i en paketspecifikation, medan underprograndefinitionerna och annat som inte en användare behöver känna till placeras i den tillhörande paketkroppen.

Abstrahera

Abstraktion avser, i detta sammanhang, att dölja detaljer. Det kan gälla både data och bearbetningar. En procedur, t.ex., döljer alla detaljer i sin kropp. I ett anrop ser vi bara procedurens namn och de argument som motsvarar dess parametrar.

I vissa fall kan vi dölja data men ändå operera på data med hjälp av tillhörande operationer (underprogram). Datatyper där vi lyckats dölja och/eller förhindra åtkomst till implementeringen kallas ibland för abstrakta datatyper. En abstrakt datatyp kan ses som ett skal. Innanför skalet finns alla detaljer, t. ex. hur data lagras och hur de underprogram som opererar på objekt av datatypen ifråga är skrivna. Utanför skalet varken vill man eller får man bry sig om dessa detaljer. Paketkonstruktionen i Ada gör detta möjligt (även skyddade typer och processer kan användas för att abstrahera).

En stack är en rak datastruktur, i vilken man kan lägga till (”push”) och ta bort (”pop”) element i dess ena ända (”sist in, först ut”). En stack kan implementeras med antingen ett fält eller med en länkad lista. Detta kan man i Ada dölja genom att kapsla implementeringen i ett paket.

1. De engelska begreppen för sammanhållning och koppling är *cohesion* resp. *coupling*.

Definitionen av en datatyp `Stack` med tillhörande operationer kan se ut enligt följande:

```
package Stacks is

   type Stack is private;

   procedure Push(The_Item : in Float; Onto_Stack : in out Stack);
   function  Pop(The_Stack : in out Stack) return Float;
   function  Is_Empty(The_Stack : in Stack) return Boolean;

private
   …
end Stacks;
```

En fältimplementation kan se ut enligt följande:

```
…
private

   Stack_Size : constant := 100;

   type Structure is array (Positive range <>) of Float;

   type Stack is
      record
         Element : Structure(1 .. Stack_Size);
         Top     : Natural := 0;
      end record;

end Stacks;
```

I paketkroppen skrivs underprogramkropparna:

```
package body Stacks is

   procedure Push(The_Item : in Float; Onto_Stack : in out Stack) is
   begin
      …
   end Push;

   …
end Stacks;
```

Om stacken i stället implementerats med en länkad lista kan det se ut enligt följande:

```
…
private
   type Stack_Node;

   type Stack_Node_Ptr is access Stack_Node;

   type Stack is
      record
         Top : Stack_Node_Ptr;   -- initieras per definition till null
      end record;

end Stacks;
```

Stack skulle alternativt kunna definieras genom härledning från `Stack_Node_Ptr`, dvs:

```
type Stack is new Stack_Node_Ptr;
```

Man behöver inte ge den fullständiga deklarationen av `Stack_Node` förrän i paketkroppen:

```
package body Stack is

   type Stack_Node is
      record
         Elements : Float;
         Next     : Stack_Node_Ptr;
      end record;
   …

end Stacks;
```

I en enhet som ska använda en stack anger man enheten `Stacks` i en kontextklausul och kan sedan deklarera och använda en stack enligt följande:

```
with Stacks; use Stacks;

procedure Test is

   S : Stack;
   X : Float;
   …
begin
   …
   Push(The_Item => X, Onto_Stack => S);
   …
   if not Is_Empty(The_Stack => S) then
      X = Pop(The_Stack => S);
      …
end Test;
```

Om `Stack` är implementerad med ett fält eller med en länkad lista påverkar i princip ej användningen av stackvariabeln `S`. Fullständigt oberoende av implementering är dock svårt att uppnå, t.ex. har ju ett fält en begränsad storlek (100 element i detta fall) medan en länkad stack kan växa så länge mer dynamiskt minne kan erhållas. För fältimplementeringen skulle man därför kunna tänka sig att även en operation `is_full` kunde finnas. Enhetlighet är dock möjlig om man använder sig av undantag (exception) för att hantera fallet ”stacken är full”.

Om vi gör paketet `Stacks` till ett generisk paket genom att parametrisera ut elementtypen och stackstorleken får vi en högre abstraktionsnivå och en betydligt mer återanvändbar enhet.

```
generic
   type Element is private;
   Stack_Size : Positive := 100;
package Stacks is
…
```

En sådan enhet kan instansieras för alla datatyper som har tilldelning definierad.

11 Optimering

En del programmerare lägger ner stor tid på att skriva ”effektiva” program, dvs (oftast) program som går snabbt att köra. Vi brukar kalla detta att man (källkods)optimerar sitt program. Om man anstränger sig för att skriva snabba program har man infört en extra svårighet, förutom att få programmet att fungera. Programmet blir lätt krångligt och svårläst, eftersom man ofta ägnar sig åt olika ”tricks och fix” för att öka snabbheten. I Ada har vi inte samma möjligheter att skriva obskyra program som t.ex. finns i C.

Det är i de flesta fall onödigt att optimera och även i de fall där det är motiverat är det lätt att man optimerar på fel sätt. Det finns två grundläggande regler för optimering:

1. Gör det inte!
2. Gör det inte *än*!

Oftast behöver man inte optimera. Om man trots allt måste göra det (när programmet *är* för långsamt) bör man vänta i det längsta med optimering, så att man vet *att*, *vad* och *hur* man ska optimera!

Se först till att programmet är korrekt (att det fungerar) och robust (att det klarar felaktiga och konstiga inmatningar utan att bete sig felaktigt eller krascha), samt att koden är lättbegriplig. *Optimera inte!*

Om du fortfarande tycker att programmet går för långsamt, låt kompilatorn försöka generera effektivare kod. Kompilera med optimeringsflagga. *Optimera inte själv, låt kompilatorn göra det!*

Om det i alla fall går för långsamt, kan du börja tänka på att optimera för hand. Gör följande:

1. Ta reda på *var* du ska optimera, dvs var i programmet går det åt mest tid? Det finns analysverktyg som visar hur ofta och hur långvarigt olika delar av ett program används.
2. Optimera på *hög nivå* (algoritmer och datastrukturer). Om du byter en enkel sorteringsalgoritmmot en mer avancerad kanske körtiden minskar avsevärt. Om du byter en enkel sökstruktur mot en hash-tabell kan kanske tiden som går åt för sökningar minskas påtagligt.
3. Först i sista hand ska du ägna dig åt lågnivåoptimering, dvs ”tricks och fix” på detaljnivå. Sådant kan du göra när du redan har effektiva datastrukturer och algoritmer, programmet ändå är för långsamt och du tagit reda på var det behöver optimeras.

Även ett enkelt program kan göras så långsamt och minneskrävande att det blir helt oanvändbart, om man väljer olämpliga datastrukturer och långsamma algoritmer. Du bör redan från början försöka välja datastrukturer och algoritmer som passar för problemet. Lågnivåeffektiviteten är dock mindre viktig, och hanteras oftast bäst av kompilatorn.

12 Flyttbarhet

Ett flyttbart (portabelt) program är ett som du kan flytta till en annan dator än det skrevs på, med ingen eller liten arbetsinsats. Att ha ett så standardiserat språk som Ada är förstås en fördel i många avseenden men det är ingen garanti för att programmen är flyttbara. Vissa saker i språket anges uttrycklingen vara implementationsberoende, andra är inte helt specificerade och kan då variera för olika implementationer. Exempel på sådant som är implementationsberoende:

- storlek och komplexitet hos program kan begränsas av kompilatorn, t.ex. antal satser som kan nästlas, antal parentesnivåer i uttryck, antal namn som kan deklareras i ett block, antal tecken i namn, antal parametrar, variablers minnesstorlek, antal värden i en uppräkningstyp, antal tecken i en sträng, radlängd i källkoden, etc. Vanligtvis är begränsningarna inte speciellt besvärande.
- beräkningsordning i sammansatta uttryck.

Kod som du själv skriver och som du vet är implementationsberoende ska du om möjligt hålla isär från annan kod. Det gör det enklare att anpassa koden om programmet ska flyttas till en annan dator.

13 Sammanfattning

- Använd meningsfulla och rättvisande namn – undvik förkortningar!
- Använd inte ”globala” variabler i onödan – återanvänd inte variabler!
- Använd symboliska konstanter och namngivna datatyper!
- De viktigaste kommentarerna är de översiktliga och förklarande!
- Kommentera lagom mycket – kommentera rätt saker!
- Använd strukturerad programmering!
- Slingor, **if**-satser och underprogram ska vara korta!
- Modularisera och abstrahera – dela upp program i enheter och på filer!
- Formatera program så att de blir lättlästa!
- Följ konventioner!

14 Referenser

[1] Ausnit-Hood C., Johnson K. A, Pettit IV R. G., Opdahl S.B. (Eds.): Ada 95 Quality and Style. (1995). Springer Verlag.

[2] Ada 95 Reference Manual. Internationell standard ISO/IEC 8652:1995.

Elektroniska kopior av dessa dokument finns på webben, se t.ex. www.adahome.com.