Pragmada Software Engineering Coding Standard

Jeffrey R. Carter

Adopted 2020 November

Introduction

This standard defines the standard for all PragmAda software created or significantly modified after its adoption date given above. Code created previously may differ from this standard. Code that differs noticeably from the examples in this document does not conform to this standard, even if it does not explicitly violate any of the standard's rules.

Languages

All software must be written in Ada with all checks enabled, or in SPARK, with proof that no checks can fail, with checks disabled. New software must be written in Ada-12, unless there is a reason for it to be in an earlier version of Ada.

Editor Settings

A "line end" is a line terminator plus any immediately preceding comment.

Spacing and Indentation

There must be no tabs or other control characters in source code except line terminators. Indent 3 spaces per indentation level. Indentation must reflect the nesting of the indented code: *N* indentation levels means the code is nested in *N* surrounding constructs.

```
procedure Advance (List : in out List_Handle; Steps : in Natural := 0) is
   -- Empty
begin -- Advance
   if Steps = 0 then
      List.Current := List.Tail;
   else
      Count : for N in 1 .. Steps loop
         Move (List => List, Count => 1);

         if List.Current = null then
            List.Current := List.Tail;

            exit Count;
         end if;
      end loop Count;
   end if;
end Advance;

procedure Interlace (List           : in out List_Handle;
                     Start_Position : in     Positive;
                     Stop_Position  : in     Positive;
                     Spacing        : in     Positive)
is
...
```

Code Layout

Line Length

Lines may be up to 130 characters in length. Statements that can fit on one line should usually be on one line.

```
Account_Balance := (Fixed_Asset_Balance + Current_Asset_Balance +
                    Stock_Balance + Account_Receivable_Balance) -
                   (Total_Liability + Total_Expenses +
                    Future_Dated_Transaction);

Do_Multiple_Processing (Total_Asset_Balance_Before_Tax =>
                           Ledger_1.Total_Asset_Balance_Before_Tax,
                        Tax_Rate                       => 0.35,
                        Total_Asset_Balance_After_Tax =>
                           Ledger_1.Total_Asset_Balance_After_Tax);
```

Spacing

There must be at least one space on the left side of ':', ":=", "=>", and all binary operators, and at least one space or line end on the right. While multiple space is permitted, it is generally discouraged except to conform to other standards, such as lining things up. There must be no space between unary operators that are symbols ("+", "-") and their operands. Unary operators that are names ("abs", "not") must be separated from their operands by a single space.

Put a space between multiple parentheses of the same type. For example, "((X + Y) * Z)".

Position and Alignment

The "is" of a procedure body must be on the same line as "procedure" if that's possible; when it can't be on the same line, it must be on a line by itself, lined up with "procedure" as in the example of Interlace given above.

The "return T;" of a function declaration must be on the same line as "function" if that's possible; when it can't be on the same line, it must be on a line by itself, lined up with "function".

The "`return T is`" of a function body must be on the same line as "`function`" if that's possible; when it can't be on the same line, it must be on a line by itself, lined up with "`function`".

```
function Recurse_Available (Total  : in Integer;
                            I      : in Integer;
                            Conf   : in Integer_List;
                            Rank   : in Integer_List;
                            Lambda : in Float_List;
                            Mu     : in Float_List;
                            Bad    : in Integer)
return Float is
...
```

The "`when B is`" of an entry body must be on the same line as "`entry`" if that's possible; when it can't be on the same line, it must be on a line by itself, lined up with "`entry`".

The "`then`" of an `if` (or `elsif`) must be on the same line as "`if`" (or "`elsif`") if that's possible; when it can't be on the same line, it must be on a line by itself, lined up with "`if`" (or "`elsif`").

```
-- Short conditions:
if Stock_Balance > 0 then
   Stock_Available := True;
elsif Stock_Balance < 0 then
   Do_Critical_Reorder;
else
   Replenish_Stock;
end if;

...

-- Long conditions:
if Stock_Balance + Account_Receivable_Balance >
   Fixed_Asset_Balance + Current_Asset_Balance - Cash_In_Hand
then
   Do_Something;
elsif Current_Asset_Balance + Fixed_Asset_Balance + Total_Income <
      Total_Liability + Total_Expenses
then
   Do_Something_Nice;
else
   null;
end if;
```

The "`is record`" of a record type declaration must be on the same line as "`type`" if that's possible; when it can't be on the same line, it must be on a line by itself, lined up with "`type`". In no case can "`record`" be indented on a line of its own.

The "`=>`" of a `when` in a case statement or exception handler must be on the same line as "`when`" if that's possible; when it can't be on the same line, it must be on a line by itself, lined up with "`when`".

```
-- In order of preference:
-- Everything can fit in 130 characters:
procedure P (P1 : in T1; P2 : in T2);
function F (P1 : in T1; P2 : in T2) return T9;
...
procedure P (P1 : in T1; P2 : in T2) is
function F (P1 : in T1; P2 : in T2) return T9 is

-- Functions only: name and parameter list fit,
-- but not "return Typename;":
function F (P1 : in T1; P2 : in T2; P3 : in T3; P4 : in T4)
return T9;
...
function F (P1 : in T1; P2 : in T2; P3 : in T3; P4 : in T4)
return T9 is

-- Name and parameter list won't fit, but indented
-- parameter list alone will fit:
procedure P
   (P1 : in T1; P2 : in T2; P3 : in T3; P4 : in T4; P5 : in T5);
function F
   (P1 : in T1; P2 : in T2; P3 : in T3; P4 : in T4; P5 : in T5)
return T9;
...
procedure P
   (P1 : in T1; P2 : in T2; P3 : in T3; P4 : in T4; P5 : in T5)
is
function F
   (P1 : in T1; P2 : in T2; P3 : in T3; P4 : in T4; P5 : in T5)
return T9 is

-- Indented parameter list won't fit:
procedure P (P1 : in T1;
             P2 : in T2;
             P3 : in T3;
             P4 : in T4;
             P5 : in T5;
             P6 : in T6);
function F (P1 : in T1;
            P2 : in T2;
            P3 : in T3;
            P4 : in T4;
            P5 : in T5;
            P6 : in T6)
return T9;
...
procedure P (P1 : in T1;
             P2 : in T2;
             P3 : in T3;
             P4 : in T4;
             P5 : in T5;
             P6 : in T6)
is
function F (P1 : in T1;
            P2 : in T2;
            P3 : in T3;
            P4 : in T4;
            P5 : in T5;
            P6 : in T6)
```

```
return T9 is
```

Parentheses

Parentheses must be kept with what they enclose. Do not follow left parentheses with comments or line ends. Do not precede right parentheses with line ends. There must be at least one space or line terminator to the left of left parentheses, and at least one space or line terminator to the right of right parentheses, unless they are followed by punctuation, such as a semicolon.

Blank Lines

There must be one blank line between context clauses and the rest of the compilation unit.

There must be one blank line between compound statements and surrounding statements at the same indentation level.

There must be one blank line between transfer of control statements (`return`, `exit`, `goto`, `raise`) and surrounding statements at the same indentation level.

There must be one blank line between delay statements and surrounding statements at the same indentation level.

Do not use more than 2 multiple consecutive blank lines. Use multiple blank lines only to separate logical blocks of code that would not be clearly separated by a single blank line. Comments at the beginning of such logical blocks are recommended.

Do not put blank lines between statements at different indentation levels.

Operators and Indentation

Line up '`:`', "`:=`", "`=>`", "`in`", "`out`", and so on when they appear on adjacent lines.

Each "`when`" must be at the same indentation level as its corresponding "`exception`" or "`case`". Otherwise, a statement nested inside a case statement, for example, would be indented two indentation levels but only be nested one nesting level.

Naming Convention

Reserved words must be all lower case except when used as an attribute, in which case they may have an Initial Capital Letter (`George'Access; Martha'range`). (Although it is simpler and more consistent for reserved words to always be all lower case, many Ada-aware editors automatically capitalize them when used as an attribute, and make it difficult to obtain alternative casing, so an initial capital is allowed.)

Identifier Names

The only characters that may appear in identifiers are `'A' .. 'Z'`, `'a' .. 'z'`, `'0' .. '9'`, and underscores. Do not use `CamelCase` or Hungarian Notation (`szFoo`). Use underscores to separate words. Use mixed case for all identifiers, a capital letter beginning every word separated by underscores. Consecutive capital letters in identifiers may only be used to represent acronyms commonly used on the project. An acronym must be a complete word of an identifier. "`Text_IO`" is acceptable, whereas "`BString`" is not.

Attributes must follow the formatting rules for identifiers above, except when they are also reserved words.

```
type Speed_Value is range 0 .. 300;

Speed          : Speed_Value;
Relative_Speed : Speed_Value;
```

Names

All constructs that can be named (loops, block statements) must be named to indicate why the construct exists. Names must be repeated wherever they are allowed (`exit`, `end`). Anonymous types, except for singleton tasks, protected objects, and anonymous access-to-subprogram parameters, must not be used.

Repeating Names

Repeat operation and block names as comments after "`begin`", "`exception`", "`private`", and "`generic`".

Useful Type-Name Suffixes

• _Handle	for private types
• _Info, _Group	for record types
• _ID, _Name	for enumeration types
• _ID	for integer types used as IDs
• _Value, _Range	for numeric types
• _List, _Set	for array types

Prefixes like "`The_`", "`An_`" and "`A_`" are not allowed when naming types and objects. No part of a name must repeat information given by the code, so type names must not have anything like a `_T[y[p[e]]]` suffix or `T_` prefix to indicate that it is a type name. Nor can names contain anything else about how they're declared, such as `_Array`, `_Package`, or `Generic_`.

Save the best name ("`List`", "`Set`", "`Person`", and so on) for parameters. Use the second-best name (such as with a suffix) for type names.

If the meaningful name differs from these guidelines, use the meaningful name. For example, a private type used as an ID would be better named `_ID` than `_Handle`.

Types

Type Extension

Type extension may only be used directly from `Ada.Finalization.[Limited_]Controlled` to obtain finalization. Types may be visibly tagged to allow `Object.Operation` notation.

Access-to-Object Types

Access-to-object types (as opposed to access-to-subprogram types) are rarely needed and must not be used unless necessary. Where access-to-object types are necessary, they must be hidden; there must be no access-to-object types in the visible part of package specifications.

Statements

`if` Statements

An `if` statement with `elsif` parts must have an `else` part, to show that all possibilities have been considered.

A short `else` part after a long `if` part is easily overlooked, so an `if` with no `elsif` parts but with an `else` part must not have the `if` part more than five lines longer than the `else` part, unless inverting the condition is less readable than the short `else`.

An `if` without an `else` part must have a comment explaining why the alternative needs no action, unless this is obvious.

`case` Statements

Other statements are nested inside `case` statements, so they must be indented one indentation level from the reserved word `case`. This implies the `when` lines of the `case` statement must not be indented.

The `when` parts of a `case` statement must appear in `"<"` order of the values. When a disjoint set is used, its parts must be in `"<"` order, and the whole set appear in `"<"` order of the first part.

```
case C is
```

```
when 'a' | 'e' | 'i' | 'o' | 'u' =>
   Process_Vowel (Vowel => C);
when 'b' .. 'd' | 'f' .. 'h' | "j' .. 'm' =>
   Process_First_Half (Letter => C);
when 'n' | 'p' .. 't' | 'v' .. 'z' =>
   Process_Second_Half (Letter => C);
when others =>
   raise Program_Error with "Invalid letter " & C;
end case;
```

Subprograms

Except for library-level, main-program procedures, every subprogram or entry body must be the completion of a declaration.

Use parameter names that will read well in calls using named notation. Use named notation for procedure and entry calls, except where parameter names are meaningless and named notation would detract from readability (`Ada.Unchecked_Deallocation` for instance).

Use positional notation for the first parameter of function calls. Remaining parameters may use positional or named notation, whichever makes the code clearer. "`Is_Member (Item, Set)`" is clear; "`Image (X, 7)`" may not be, since 7 might be the `Width` or the `Base` parameter. "`Image (X, Width => 7)`" is better.

Parameter Modes

Parameter modes must be specified for all subprograms and entries in Ada-12. In earlier versions of Ada, all function parameters have mode `in`, so modes may be omitted for functions. Align "`:`" and parameter modes when parameters are not on the same line.

```
procedure Dynamic_Sensor_Fusion (N             : in     Integer;
                                 F             : in     Integer;
                                 Value         : in out Object;
                                 Bounds        : in out Reading;
                                 Data          : in out Reading_List;
                                 Print_Details : in     Integer;
                                 Result        :    out Result_Value)
is
   Cardinality : Natural := 0;
   Factor      : Integer := 0;
   ...
```

`Use` Clauses

Avoid all unnecessary "`use`" of packages. There must be no "`use`" clauses in package specifications or context clauses. In bodies, "`use [all] type`" is acceptable. Acceptable "`use`" clauses are limited to standard packages and packages widely used in the project, limited in scope (to small subprograms, entry bodies, or block statements), apply to 3 or more references to the package, and leave it clear where the abbreviated names are declared.

```
George : declare
   use Stack;
begin -- George
   ...
   Push (Item => M);
   ...
   N := Pop;
   ...
exception -- George
when Error =>
   Put (Item => "Stack manipulation error.");
when others =>
   Put (Item => "Something else went wrong with Stack.");
end George;
```

Do not "`with`" or "`use`" ancestor packages. Their specifications are available without a context clause and directly visible without a `use` clause.

Transfer-of-Control Statements

Loop Control

Conditions should be as easy to read and understand as possible. Positive logic (`More_Values_Exist`) is easier to read than negative logic (`not End_Of_Values`), and exit conditions are more intuitive than continuation conditions. Use "`exit`" to control loops when the exit condition is more readable than the continuation condition; use "`while`" when the continuation condition is more readable than the exit condition. Use

```
All_Lines : loop
   exit All_Lines when End_Of_File; -- Positive logic
```

not

```
All_Lines : while not End_Of_File loop -- Negative logic
```

and

```
All_Lines : while More_Lines_Exist loop
```

not

```
All_Lines : loop
   exit All_Lines when not More_Lines_Exist;
```

but since the exit condition is more intuitive than the continuation condition, consider

```
function No_More_Lines return Boolean is (not More_Lines_Exist);
...
All_Lines : loop
   exit All_Lines when No_More_Lines;
```

Goto Statements

Use "`goto`" only for mechanical implementation of state-transition diagrams.

Comments

Comment package specifications such that a reader need not see the body to understand what the package provides and how to use the package. Comment subprograms and entries after the subprogram or entry declaration.

Put a space after the "--" of a comment. Multiple spaces may be used to line things up in consecutive comments.

Empty declarative regions must contain a comment indicating that the region is deliberately empty;

```
-- Empty
```

is sufficient.

Comments must be correct English and begin with a capital letter.

Comments must add value. Comments that repeat things that are obvious from the code do not add value. Comments must add information that is not apparent from the code. Things are obvious when they are obvious to any competent software engineer.

```
X := X + 1; -- Increment X. (This comment does not add value)

-- If Item.Group is outside the desired range of groups,
-- all following items' groups will also be outside
-- the range, so we can terminate the iteration now
if Item.Group > Last_Group then
   Continue := False;

   return;
end if;
```

Miscellaneous

Name packages to be descriptive and show logical groupings of declarations; avoid ambiguous names such as "utilities", "helpers", and the like. If operations are long, consider making them separate.

Do not use `'range` unnecessarily:

```
if Something in Number then
```

as opposed to

```
if Something in Number'range then
```

There must be no variables in package specifications except protected objects.

Declarations must be grouped to show logical relationships. Within such logical groups, group similar kinds of declarations: named numbers, constants, types and subtypes, variables. Separate different kinds of declarations by a blank line.

There must be only one name per declaration and one parameter name per parameter declaration. There must be only one unit name per "`with`" statement, one unit name per "`use`" statement, and one subtype name per "`use type`" statement.

Everything in a package specification must be needed by the specification, must be needed by the client to use the package, or must be in the private part and needed by multiple descendants of the package. Use "`private with`" for units that are only needed in the private part.

There must be no unit named in a context clause that is not referenced in the compilation unit where the context clause appears, except for library-level, main-program procedures, which may `with` packages that are not referenced but provide essential functionality through their elaboration.