Goddard Dynamic Simulator
Ada Coding Standard

Stephen Leake

14 April 2014

1 Introduction

This document presents the coding standards used by the Goddard Dynamic Simulator project for the Ada language. There are two primary goals for these coding standards:

- Provide a common style for Ada code, so it can be comfortably read by all programmers on the project.
- Make the Ada language easier to use, by giving guidelines on using complex features.

This standard sets the policy for the appropriate use of Ada language constructs, identifier naming conventions, and general source code layout. It also suggests coding patterns for use with Ada.

In addition, it extends the unit testing standard, in areas that are unique to Ada.

The standards described in this document are either mandatory or discretionary. Mandatory standards are requirements that must be followed. These are indicated by the word "shall". Discretionary standards are guidelines that allow some judgment or personal choice by the programmer. These standards are indicated by the use of the word "should". A programmer should have a good reason when they choose to disregard a discretionary standard. Where possible, this document gives examples of acceptable deviations. A mandatory standard can be waived by the development lead for specific cases, where appropriate.

In this document, paragraphs labeled by a number in parentheses (n) are standards; other paragraphs are explanatory text.

The primary mode of reading code is assumed to be on a computer screen, with an editor that provides syntax colorization and code navigation. There will be a few times, primarily at code reviews, when code will be printed out. The coding standards reflect this bias towards on-screen viewing.

It is easier to combine code into a system, and to maintain the code, if all of the code conforms to a particular style. Since Ada is new to many programmers, and the general Ada community shares a fairly standard style, we require a particular style. See Figure 1 for an example of the style.

2 Language

1. The code shall only use Ada language constructs as defined by ISO/IEC 8652:2012(E) Information Technology – Programming Languages – Ada. An exception is allowed if a preprocessor is needed for portability; then the constructs defined by the GNAT preprocessor `gnatprep` are allowed.

   The Ada 2005 Language Reference Manual is available at

   `http://www.adaic.org/standards/ada05.html`.

2. Non-standard pragmas shall require a waiver and full documentation.

3. The non-standard GNAT pragma "Unreferenced" is granted a waiver.

4. All source files shall consist of printable ASCII characters, with no tabs or form feeds, and no trailing spaces on lines, maximum of 120 characters per line.

3 File and Package Naming Conventions

3.1 File names

1. Each source file shall contain one compilation unit.

2. The filename shall be the full Ada unit name, all lowercase, with "." replaced by "-". This is the default naming convention used by the GNU Ada compiler (GNAT). An exception is allowed when different bodies are required for different targets; in that case the filename shall be the standard file name, with a suffix indicating the target.

3. Specification files shall have an extension of `.ads`. Body files shall have an extension of `.adb`.

4. Files that contain preprocessor constructs shall have a second extension of `.gp` (for gnatprep). Thus the file `foo.ads.gp` is processed by gnatprep to produce the file `foo.ads`.

3.2 Package names

1. If unit test drivers are a child procedure of the unit package, they shall have the name `Test`. For example, the unit test driver for package `Foo` is `Foo.Test`. This gives the unit test access to the private part of the package.

2. Generic packages shall have a name that starts with `Gen_` or `Generic_`.

3. Unit tests for generic packages may either be children of a nominal instantiation of the package, or a main procedure that instantiates the package. If the latter, the name shall drop the generic prefix, and add a `_Test` suffix. So a test for `SAL.Filters.Gen_First_Order` would be named `SAL.Filters.First_Order_Test`.

4 File layout

1. The file shall be introduced by a block comment that contains a file prologue (see Figure 1). The file prologue consists of:

   Abstract A short statement that describes the purpose of the file. In a package body, the abstract should just refer to the specification.

   References List all applicable documentation references. Identify each reference to a level of detail that allows the information to be found but not to a level that will likely change. Provide version numbers when applicable.

   If a reference applies to several related packages, provide the reference only in the spec of the common parent package, to avoid duplicating information and simplify the process of updating the version.

   Design Notes List design decisions that apply to the general design of the code. Give a brief justification for each decision; for example, list the alternatives and say why they were rejected. Longer justifications may be given in a separate design document.

   Copyright Copyright or disclaimer notice. Copyright dates should be in one of the following formats: "year1, year2" or "year1 - year_n" if the years are consecutive.

2. Context clauses shall list one package per line. If a "use" clause is present, it should be on the same line as the corresponding "with".

3. Declarations should be grouped logically. A group label comment should be used to separate logical groups:

```
----------
-- Orientation operations
```

4. When there are more than five subprograms in a group, they should be listed alphabetically within the group.

5. Operations of tagged types should be grouped into "class-wide" and "dispatching" groups.

6. Operations of derived types should be grouped into "overriding" and "new" groups.

5 Style enforcing tools

Style rules are easier to follow when tools are available to enforce them, or automate using them. We define the required style in part by the tools given here. The style requirements given in the rest of this document should conform to this definition; however, when there is a conflict, the tool is correct.

1. Code shall be indented as defined by the GNU Emacs Ada mode, as distributed with Emacs (version 21.2 or higher), or as distributed by ACT, with the following settings (other settings have their default values):

   - (setq ada-when-indent 0)
   - (setq ada-label-indent 0)

2. Groups of statements, record declarations, and named parameter associations shall be aligned by the standard Emacs package `align.el`.

3. Code shall conform to the GNAT (version 3.16a or higher) style check compiler option `-gnaty3abefhiklM120nprt`. See the GNAT User's Guide for more information.

6 Comments

Comments are important in developing readable and maintainable code. Programmers should include comments whenever it is difficult to understand the code without the comments.

On the other hand, comments should not simply say the same thing the code does, as this only serves to obscure required comments. Describe why something is being done first, and only describe what is being done if it isn't obvious by the code itself.

Comments can be classified by size:

Figure 1: Example package specification file

```
-- Abstract:
--
-- General utilities for star tracker models.
--
-- References:
--
-- [1] Star Tracker Component
--
-- Design Notes:
--
-- Catalog_Type is abstract tagged to provide a common interface to different
-- catalog implementations.
--
-- Disclaimer
--
-- <standard NASA disclaimer>
--

with Ada.Numerics.Float_Random;
with Math_3_DOF;
with Math_Scalar;
package HDS.Star_Tracker is

   ----------
   -- Orientation operations

   function Aberration
     (GCI_q_ST        : in Math_3_DOF.Unit_Quaternion_Type;
      ST_Boresight    : in Math_3_DOF.Unit_Vector_Type     := Math_3_DOF.Z_Unit;
      GCI_Sun_v_Earth : in Math_3_DOF.Cart_Vector_Type;
      GCI_Earth_v_Sc  : in Math_3_DOF.Cart_Vector_Type)
     return Math_3_DOF.Unit_Quaternion_Type;
   -- Return relativistic velocity aberration quaternion ST_q_Ab.

   ---------------
   -- Star vector operations

   type Star_ID_Type is new Integer;

   Unknown : constant Star_ID_Type := -1;

   type Star_Vector_Type is record
      ID        : Star_ID_Type                := Unknown;
      Vector    : Math_3_DOF.Unit_Vector_Type := Math_3_DOF.X_Unit;
      Magnitude : Math_Scalar.Float_Type      := 0.0;
   end record;

   Unknown_Star : constant Star_Vector_Type :=
     (ID        => Unknown,
      Vector    => Math_3_DOF.X_Unit,
      Magnitude => 0.0);

end HDS.Star_Tracker;
```

- Long or block comments are those that can not fit on a single 80 character line.
- Short comments are those that can fit on a single 80 character line (note that this is shorter than the code line length limit).
- Same-line comments are small enough to include on the same line as the code that the comment supports.

1. Each line of a comment shall start with two spaces after the two hyphens, for compatibility with Emacs Ada mode.
2. Functional blocks of code should have a long comment before the actual code instead of placing a comment on each line. This comment should describe the basic purpose of the code. The comment should consist of complete English sentences.
3. A variable's units of measurement should be stated in a comment if they are not the SI standard units, as defined at http://www.bipm.org/en/si/. It is preferable to use units with no prefix, except kilograms.
4. Change log comments shall not be in the source code; use a configuration management tool instead.
5. Old versions of the code shall not be maintained in the comments; use a configuration management tool instead.
6. A special comment format shall be used to document compiler workarounds. This allows finding and fixing the workarounds when a new compiler version is released.

```
-- WORKAROUND: <compiler version> <description>
```

7. A special comment format shall be used to document an incomplete implementation:

```
-- FIXME: <description>
```

8. Another special comment format shall be used to document an implementation that could be improved:

```
-- IMPROVEME: <description>
```

7 Identifier Naming conventions

The proper naming of packages, types, subprograms and variables can increase the readability and understanding of the software. Poorly named functions and variables can add a great deal of misunderstanding and confusion. Careful consideration should be given to how others will perceive the name, and what they will think it means. An object's name should provide insight into its use and purpose.

1. Identifiers should consist of more than one character. Valid exceptions are for variables used in local indexing operations (e.g., i, j, k) and very mathematical code where the identifier matches the standard symbol used in an algorithm or equation.
2. Ada reserved words shall be all lowercase, except when used as attributes.
3. Acronyms shall be all upper case. Each project shall maintain a list of approved acronyms.
4. All other identifiers shall be mixed case, with underscores separating words. Acronyms that are part of a larger identifier shall be uppercase. This style is often called `Mixed_Case_With_Underscores`. This is the most common style for Ada code, and with an editor that does the capitalization automatically, is easier to type than `MixedCaseNoUnderscores`.
5. The names of all types not defined by the language shall end with a suffix of `_Type`. This allows using the same general name for the type and an object: `List : List_Type;`
6. Capitalization of identifiers shall be consistent among all occurrences of the identifier. The GNAT style check compiler option checks this.
7. The optional name at the end of a named construct (ie. procedures, functions) shall always be present.
8. The name of a scale factor shall use "per" rather than "to". For example, to convert from minutes to seconds, use `Seconds_Per_Minute`, not `Minutes_to_Seconds`. This makes it clear how to use the constant: `Seconds = Seconds_Per_Minute * Minutes`.

8 Indentation

1. All code shall be indented three spaces for each indentation level.
2. Statements at the same logical nesting level shall be at the same indentation level.

3. The indentation of a long or short comment shall be the same as the code it describes.

4. There should be only one statement per line.

5. Blank lines should be used between blocks of code that are functionally distinct.

6. In expressions, operators should be surrounded by blanks, except for prefix operators. This is enforced by the GNAT style checks.

7. Long lines should be broken after operators or commas.

8. Continuation lines shall be indented one level.

9 Subprograms

9.1 Specification

1. A procedure specification shall have the one of the following formats:

```
[[not] overriding] procedure Name (Arg_1 : <mode> <type>; Arg_2 : <mode> <type>);

[[not] overriding] procedure Name
  (Arg_1 : <mode> <type [:= <default>];
   Arg_2 : <mode> <type [:= <default>]);
```

2. A function specification shall have the one of the following formats:

```
[[not] overriding] function Name (Arg_1 : <mode> <type>; Arg_2 : <mode> <type>) return

[[not] overriding] function Name
  (Arg_1 : <mode> <type> [:= <default>];
   Arg_2 : <mode> <type> [:= <default>])
  return <result_type>;
```

   Ada allows the mode to be left out in functions, since it must always be “in”. However, we require that it be present, for consistency with procedures.

3. The keyword `overriding` shall be present if the subprogram is intended to override an inherited operation. The compiler has an option to enforce this.

4. In some cases, the GNAT runtime code does not have `overriding` where it should, resulting in a compile-time style error. The fix is to add `pragma Style_Checks (Off, );` on the entity that causes the problem, or to surround the line of code with `pragma Style_Checks (Off); .. pragma Style_Checks (On);`

5. All multi-line parameter lists shall have the format given by the Emacs Ada mode align function.

6. Each subprogram declaration shall be followed by a block comment giving a description of the subprogram, including its purpose and how each argument is used. For simple subprograms with clear argument names, the descriptive comment can be empty. For a very complex subprogram, the description should just reference separate documentation.

9.2 Body

1. A subprogram body shall have one of the following formats:

```
[[not] overriding] procedure Name (Arg_1 : <mode> <type>; Arg_2 : <mode> <type>)
is
   Local_Var_1 : <type> [:= <default>];
begin
   <sequence_of_statements>
end Name;

[[not] overriding] procedure Name
  (Arg_1 : <mode> <type [:= <default>];
   Arg_2 : <mode> <type [:= <default>]);
is
   Local_Var_1 : <type> [:= <default>];
begin
   <sequence_of_statements>
end Name;
```

   Putting the "is" on a separate line cleanly separates the parameter list from the local variable list.

2. The keywords `not overriding` shall be present if they are on the corresponding specification. The compiler has an option to enforce this.

3. A function body shall have one of the following formats:

```
function Name (Arg_1 : <mode> <type>; Arg_2 : <mode> <type>) return result_<type>
is
   Local_Var_1 : <type> [:= <default>];
```

```
begin
   <sequence_of_statements>
end Name;

function Name
  (Arg_1 : <mode> <type> [:= <default>];
   Arg_2 : <mode> <type> [:= <default>])
  return <result_type>
is
   Local_Var_1 : <type> [:= <default>];
begin
   <sequence_of_statements>
end Name;
```

9.3 Call

1. Argument lists that do not fit on one line should have one argument per line, starting on the line after the subprogram name:

   ```
   Name (Arg_1 => Arg_1_Value, Arg_2 => Arg_2_Value);

   Name
     (Arg_1 => Arg_1_Value,
      Arg_2 => Arg_2_Value);
   ```

   One point to remember is that indentation should never depend on the length of a name.

2. If there are many arguments, and named association is not needed, they may be grouped on one line:

   ```
   Name
     (Arg_1, Arg_2, Arg_3, Arg_4, Arg_5, Arg_6,
      Arg_7, Arg_8, Arg_9, Arg_A, Arg_B, Arg_C);
   ```

3. Named association should be used whenever the association of value to parameter might be ambiguous or confusing.

   In particular, if two or more arguments have the same type, named association should be used.

4. A useful style is to group little-used parameters with defaults at the end of a parameter list, and only use named association for them when they are not defaulted:

   ```
   Name
     (Arg_1, Arg_2,
      Verbosity => 2);
   ```

10 Access types

It is often difficult to decide whether to use an 'in' (or 'in out') parameter of a type, an 'access' parameter of type, or an 'in' parameter of an access type, particularly for class-wide types, since they almost always require an access type at some point. Then it seems that a lot of clients need `.all` if the parameter is 'in'. However, in practice, when you change from 'access' to 'in' for a complex subprogram, particularly if it passes that parameter on to others, it is often not clear whether you are adding or deleting more `.all` uses.

On the other hand, some simple subprograms (typically Set and Get) will *only* ever be called with an object that was directly declared to be of a named access type, and have only very simple references to the parameter in their body. In that case it is clearly better to use a few `.all` in the simple body than in every call, so the subprogram should use the named access type.

Using an access parameter (either an explicit type or anonymous type) only when strictly required by the body makes it clearer when access types are actually needed, which helps when considering how to redesign a subprogram to add a feature.

There is one situation where an 'in' parameter is required; when converting to a type lower in a derived type hierarchy. For example:

```
declare
  Owner : constant access Modules.Module_Type'Class := Symbols.Owner (Symbol);
begin
  if Owner.all in Modules.Hardware.Module_Type'Class then
    Modules.Hardware.Checked_Write_Command (Modules.Hardware.Module_Type'Class (Owner.all),

  ...
```

In this case, `Checked_Write_Command` must be declared with an 'in' parameter of the class-wide type. If it had been declared as an 'access' parameter, the type conversion would not be possible, since there is no type name that matches. An 'in' parameter of an access type would not be dispatching.

Another situation requires a different solution. If a derived type hierarchy has a parallel named access type hierarchy, it may be necessary to use anonymous access to class-wide parameters:

```
package GDS.Threads.Distribute is

   procedure Add_Communicate_1
     (Chassis   : in     Chassis_ID_Type;
```

```
      Direction : in     Direction_Type;
      Symbol    : access constant GDS.Symbols.Symbol_Type'Class);
   --  Store Symbol in the Communicate_1 list.
end;

package body GDS.Modules.Executor.Wrapper is

   Time_Sync_Valid : GDS.Symbols.Times.Symbol_Access_Type := ...;

   procedure Build_Remote_Lists (Module : in Module_Type)
   is begin
       Add_Communicate_1 (Module.Chassis_ID, TX, Time_Sync_Valid);
   end;
end;
```

In this case, since `Add_Communciate_1` is storing Symbol in a list, we would like to specify the exact named access type used in the list. However, that is `GDS.Symbols.Symbol_Access_Type`; a class-wide pointer to the root of the hierarchy. On the other hand, `Time_Sync_Valid` is declared with a lower-level named access type, to avoid run-time checks.

If `Add_Communicate_1` were declared with `GDS.Symbols.Symbol_Access_Type`, an explicit type conversion would be required at the call in `Build_Remote_Lists`. Using an anonymous access type avoids that explicit conversion.

1. Use an 'in' or 'in out' parameter of a non-access type, unless an access type is required by the body, even if that means most clients will need `.all` to call the subprogram.

   Exceptions to this rule are allowed for simple subprograms (for example Set and Get) that will *only* ever be called with an object that was directly declared to be of a named access type, and have only very simple references to the parameter in their body. In that case, the subprogram should use the named access type. Each such subprogram must be clearly labeled with a comment explaining the exception.

2. If an 'access' parameter is used, 'constant' must also be specified if the value is not changed by the subprogram.

3. Use a named access type parameter when the pointer will be stored in a list or other structure; then use exactly the same access type as used in the list (except as noted in the next rule). Otherwise use an anonymous access type.

   Using a named access type as the parameter type forces the compile and run-time checks to be done at the point of the call, making it easier to understand when there is a problem.

4. If a subprogram would use a named access type by the previous rule, but will typically be passed a named access type lower in the hierarchy, use an anonomous access type equivalent to the root named access type. Be sure to include `constant` if applicable.

11 Statements

11.1 Variable declarations

1. Each variable declaration shall declare only one variable. For example:

```
One_List     : List_Type;
Another_List : List_Type;
```

   not:

```
One_List, Another_List : List_Type; -- wrong!
```

2. A declared variable shall be initialized only when an algorithm requires that the variable have an initial value.

11.2 if Statements

1. An if statement shall have one of the following formats:

```
if condition then
   statement;
[elsif condition then
   statement;]
else
   statement;
end if;

if long-condition
   continued-condition
then
   statement;
[elsif
   long-condition
then
   statement;]
else
   statement;
end if;
```

The else clause is optional. These formats are checked by the GNAT style check.

11.3 Short cut Boolean operators

The short cut Boolean expressions are `or else` and `and then`. They prevent execution of the right expression if the left expression is true or false, respectively. These should only be used when the right expression will raise an exception if executed.

Programmers are often tempted to use the short cut expressions to "optimize" the code. The compiler is far better at optimizing than the most programmers, and optimizing at this level is a waste of programmer time until timing measurements prove it is necessary. It is better to use the presence of a short cut operator to indicate a possible exception.

11.4 case Statement

1. A case statement shall have the following format:

```
case expression is
when constant_expression_1 =>
    statements;
when constant_expression_2 =>
    statements;
end case;
```

2. A `when others` clause shall only be present if the case statement is not exhaustive. Most case statements should be exhaustive, so the compiler will warn you when a value is added to the case expression type.

12 Miscellaneous

1. Named association shall be used for subprogram parameters and aggregates whenever the parameter order is not fully determined by the types. For example, if a subprogram has two parameters of type Integer, named association must be used to distinguish them.

   For example, `Ada.Strings.Fixed` declares the Index function:

```
function Index
  (Source   : in String;
   Pattern  : in String;
```

```
Going    : in Direction := Forward;
Mapping  : in Maps.Character_Mapping := Maps.Identity)
return     Natural;
```

It is easy to confuse the Source and Pattern parameters, so named association must be used in a call:

```
Comma := Index (Source => "hello, world", Pattern => ",");
```

2. Named association shall be used for all generic instantiations.

3. A use clause may be given for a package only in a package or subprogram body, not in a package specification or among the `with` statements preceeding the package body. The scope of the use clause should be limited as much as reasonable.

   Use clauses make for more compact code, with the tradeoff of making it more difficult to understand where identifiers are declared. In package specifications, it is more important to understand the relationship with other packages, so use clauses are not used. In bodies, it is more important to allow for compact code, which makes it easier to understand the control flow.

   For a subprogram body, a good rule of thumb is if a package name would appear in more than 4 statements, a use clause is appropriate.

4. Every thread shall have a catch all exception handler either in the main loop or at the top level in the task body, that prints an error message to `Ada.Text_IO.Standard_Error`, so the user will know if the thread is terminating.

5. The Ada 2005 construct `raise <exception_name> with "message";` is prefered over the Ada 95 `Ada.Exceptions.Raise_Exception`.

6. The extended return statement shall only be used in a function whose result type is an anonymous access type.

   The Ada 2005 extended return statement declares the return object in a nested scope (different than the function scope). This is only necessary when the function result type is an anonymous access type; it allows the accesibility level of the result object to be that required at the point of call, rather than that of the function.

   Thus use of the extended return statement indicates an accesibility level issue; use in other contexts would be confusing.

7. The keyword `overriding` preceding a subprogram declaration should be present if it is legal.

   This indicates the subprogram must override some inherited operation. This gives an error if the operation on the parent type is changed, alerting the programmer to change the overriding operation as well.

8. A subprogram declaration of the form `procedure ... is null;` is prefered over an actual null body.

9. `'Floor` By default, elisp and C truncate on conversion from floating point to integer; Ada rounds. `'Floor` can be used to obtain the C or elisp behavior.

   Another possible use for `'Floor` is to model an analog to digital converter; the integer output is the highest integer that corresponds to a voltage that is less than the actual voltage. However, `'Floor` should not be used for this purpose, because it is too harsh. Round off error can cause a number that should be an exact integer to be slightly less, and floor drops it to one whole integer less. This causes significant errors in the model. Rounding avoids the problem, because we don't care whether half-counts are exact in an analog to digital converter model; converters are only accurate to half a bit anyway.

13 Importing C code

1. Some C functions have both an "out" parameter and a non-void function result. To import these into Ada, use the GNAT-specific pragma `Import_Valued_Procedure`. This must be used in conjuction with pragma `Import`. For example:

```
procedure M1553_Read_Sa_Control_Buffer
  (Status    :    out Status_Type;
   Device    : in     Interfaces.Unsigned_16;
   Buffer_Id : in     Interfaces.Unsigned_32;
   Buffer    :    out Sa_Control_Buffer_Type);
pragma Import (C, M1553_Read_Sa_Control_Buffer);
pragma Import_Valued_Procedure (M1553_Read_Sa_Control_Buffer, "m1553_read_sa_control
```

14 Unit Tests

See 582 Branch Unit Test Standard for the base unit testing standard. This section presents unit test standards that are unique to Ada.

1. Fixed point types shall have unit tests that verify the range.

   Fixed point types are useful for matching packet formats, but it is easy to get the range wrong. Unit tests should show that the type allows the expected range of values.