Contract-based Programming in Ada 2012

A tutorial on how to use the Ada 2012 features for specifying detailed, checked contracts for types and subprograms[1].

Contracts document constraints on how types and subprograms behave, but unlike comments they are checked – either by when the program is compiled or on-the-fly as the program is running.

Ada 2012 contract aspects will be presented together with a set of guidelines for using contract aspects consistently. The tutorial will conclude with a live test of the guidelines on some example source text.

[1] "classes, functions, and methods" if you aren't an Ada programmer yet.

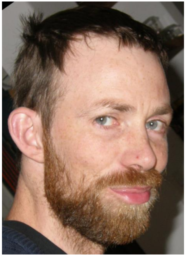
Contract-based Programming in Ada 2012
A Tutorial

Jacob Sparre Andersen

Jacob Sparre Andersen Research & Innovation

1st February 2014

Jacob Sparre Andersen



Currently:

- Independent consultant.
- Co-founder of AdaHeads K/S.
- Ada guru and software architect at AdaHeads K/S.

Background:

- PhD & MSc in experimental physics.
- BSc in mathematics.
- Has taught mathematics, physics and software engineering.
- Has worked with bioinformatics, biotechnology and modelling of investments in the financial market.

jacob@jacob-sparre.dk
www.jacob-sparre.dk

Raison d'etre for Ada

In the 1970's the US DoD noticed they had a problem with software development.

To solve it they arranged a programming language design competition with:

> *. . . three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency.*

The result was Ada (1983).

(to make a long story short)

Ada in a single slide

- Procedural programming language.
- Supports object oriented programming (encapsulation, inheritance and dynamic dispatching).
- Modular programming: Packages, child packages and individual subprograms. Generic modules.
- Separates declarations of subprograms and packages in specifications and implementations.
- Concurrent, distributed and real-time programming built in.
- Types distinguished by name (both simple and composite types).

Most recent Ada standard published by ISO in December 2012.

Contracts and aspects in Ada 2012

New in Ada 2012:

- Pre- and postconditions:
  Run-time checks at the call of and return from a subprogram.
- Type invariants:
  Run-time checks on changes to a private type.
- Dynamic subtype predicates:
  Constraints on visible subtypes.
- Static subtype predicates:
  Static constraints on visible subtypes.

Some checks in “old” Ada

- Named “primitive” types:
  No implicit conversions between different named types.
- Parameter passing directions:
  “in”, “out” or “in out” parameters to subprograms checked at compile-time.
- Range checks:
  Array indexing is checked (typically only at run-time).
- Subtypes with ranges:
  Subtype ranges are checked (typically only at run-time).
- Static coverage tests:
  Case statements are checked for full coverage at compile-time.

Preconditions

From the Ada 2012 rationale:

> *A precondition . . . is an obligation on the caller to ensure that it is true before the subprogram is called and it is a guarantee to the implementer of the body that it can be relied upon on entry to the body.*

You can only write to open files:

```
procedure Put (File : in     File_Type;
               Item : in     String)
  with Pre => (Is_Open (File));
```

Postconditions

From the Ada 2012 rationale:

> *A postcondition . . . is an obligation on the implementer of the body to ensure that it is true on return from the subprogram and it is a guarantee to the caller that it can be relied upon on return.*

The line number of a file is incremented when you write a line to it:

```
procedure Put_Line (File : in      File_Type;
                    Item : in      String)
  with Pre  => (Is_Open (File)),
       Post => (Line (File) = Line (File)'Old + 1);
```

Type invariants

Make sure that `Disc_Point` objects stay on or inside the unit circle:

```
package Places is
   type Disc_Point is private;
   --  various operations on disc points
private
   type Disc_Point is
      record
         X, Y : Float range -1.0 .. +1.0;
      end record
      with Invariant => Disc_Point.X ** 2 +
                        Disc_Point.Y ** 2 <= 1.0;
end Places;
```

Adapted from the Ada 2012 rationale.

Dynamic subtype predicates

Subtype predicates are a kind of constraints on subtypes.

This subtype of Integer can only contain primes:

```
subtype Prime is Integer range 2 .. Integer'Last
  with Dynamic_Predicate
    => (not (for some N in 2 .. Prime - 1
                 => Prime mod N = 0));
```

`Organization_URI` strings can be up to 256 characters long:

```
subtype Organization_URI is String
  with Dynamic_Predicate => (Organization_URI'Length
      <= 256);
```

Before Ada 2012 you would have to use the package `Ada.Strings.Bounded` to sort of achieve this effect.

Static subtype predicates

Inspired by the Ada 2012 rationale:

```
procedure Seasons is
   type Months is (Jan, Feb, Mar, Apr, May, Jun,
                   Jul, Aug, Sep, Oct, Nov, Dec);
   subtype Summer is Months
     with Static_Predicate => Summer in Nov .. Dec |
                                        Jan .. Apr;
   A_Summer_Month : Summer;
begin
   A_Summer_Month := Jul;
end Seasons;
```

The compiler identifies the problem:

warning: static expression fails static predicate check on "Summer"

Coverage check

```
case Level is
   when Debug =>
      System_Message.Debug.Dial_Plan     (Message);
   when Information =>
      System_Message.Info.Dial_Plan      (Message);
   when Notice =>
      System_Message.Notice.Dial_Plan    (Message);
   when Warning =>
      System_Message.Warning.Dial_Plan   (Message);
   when Error =>
      System_Message.Error.Dial_Plan     (Message);
   when Critical =>
      System_Message.Critical.Dial_Plan  (Message);
   when Alert =>
      System_Message.Alert.Dial_Plan     (Message);
   when Emergency =>
      System_Message.Emergency.Dial_Plan (Message);
end case;
```

Guidelines – Types, constraints and invariants

Make sure your type declaration is as detailed in its constraints as possible.

- Declaring a new type or a subtype depends on what level of inter-type compatibility you want[2].
- Put an appropriate constraint on the range of values the (sub)type can have.
- Add any extra constraints as predicate (non-private types) or invariant (private types) aspects.

[2] And if there is a type to derive from.

Example – Types, constraints and invariants

Primes are integers:

```
subtype Prime is Integer;
```

... larger than 1:

```
subtype Prime is Integer range 2 .. Integer'Last;
```

... and have no other factors than 1 and the prime itself:

```
subtype Prime is Integer range 2 .. Integer'Last
  with Dynamic_Predicate
    => (not (for some N in 2 .. Prime - 1
               => Prime mod N = 0));
```

Guidelines – Subprograms and argument declarations

Make sure that you declare the arguments for your subprograms as specifically as possible.

- Select the proper direction (“**in**”, “**out**” or “**in out**”) for each of the arguments to a subprogram.
- Select as specific a (sub)type as possible for each of the arguments to a subprogram.
- Use preconditions (postconditions) to declare stronger constraints on the input (output) values than those implied by the selected subtypes.

Example – Subprograms and argument declarations

We want to be able to increment a counter by arbitrary steps. We use ("in") the value of both the counter and the step size to generate ("out") a new value for the counter:

```
procedure Increment (Counter : in out Integer;
                     Step    : in     Integer);
```

We count from zero and up (natural numbers). An increment is by one or more (positive numbers):

```
procedure Increment (Counter : in out Natural;
                     Step    : in     Positive);
```

Example – Subprograms and argument declarations

There is an upper limit (Natural'Last) to how far we can count with our selected type; and once we've incremented the counter it must be larger than zero:

```
procedure Increment (Counter : in out Natural;
                     Step    : in     Positive)
  with Pre  => (Counter < Natural'Last),
       Post => (Counter > 0);
```

We shouldn't attempt an increment so large that we go beyound the upper limit for how far we can count (Natural'Last):

```
procedure Increment (Counter : in out Natural;
                     Step    : in     Positive)
  with Pre  => (Counter < Natural'Last) and
               (Step <= Natural'Last - Counter),
       Post => (Counter > 0);
```

Guidelines – Preconditions

What are the requirements of your subprograms?

- Do some of your subprograms have some special requirements, which should be met before they can be called?
- Can a subprogram only be called once?
- Can a subprogram only be called when the system is in a specific state?

This can be documented with appropriately formulated preconditions to the subprograms.

Examples – Preconditions

If we want to write to a file, it should be open and writable:

```
   procedure Put (File : in     File_Type;
                  Item : in     Character)
     with Pre => (Is_Open (File)) and
                 (Mode (File) = Out_File or
                  Mode (File) = Append_File);
```

Initialise only once:

```
procedure Initialise
  with Pre => (State = Not_Initialised);
```

Guidelines – Postconditions from preconditions

Do post- and preconditions match for likely sequences of calls to your subprograms?

- If one subprogram should be callable after a call to another one (operating on some common data), then the precondition corresponding to the common data on the second call should be included in the promise made in the postcondition of the first call.

Example – Postconditions from preconditions

A likely call sequence: open a file; write to the file.

Here is one subprogram for writing data to a file:

```
procedure Put_Line (File : in     File_Type;
                    Item : in     String)
  with Pre  => (Is_Open (File)),
       Post => (Line (File) = Line (File)'Old + 1);
```

As "Put_Line" has the precondition "Is_Open (File)", the subprogram should have a postcondition fulfilling this to be useful:

```
procedure Open (File : in out File_Type;
                Mode : in     File_Mode;
                Name : in     String)
  with Post => (Is_Open (File)) or
               (raise Name_Error);
```

Live demonstration

Demonstrating the guidelines on an example program provided by Didier Willame (who is going to be the next speaker here).

Compile-time checking of contracts

Static (compile-time) checking of contracts and aspects is currently only implemented, where it is required by the language standard – and in some cases for static subtype predicates.

I hope that we in the future will see more cases of compilers checking contracts already at compile-time, and not just inserting the checks in the running code.

Contact

Jacob Sparre Andersen
Jacob Sparre Andersen Research & Innovation
jacob@jacob-sparre.dk
http://www.jacob-sparre.dk/

You can find my Open Source software repositories at:
http://repositories.jacob-sparre.dk/