# *Gem #19: Передача данных объектов Ada в потоке XML*

**Автор: Pascal Obry, EDF R&D**

Краткое содержание: Gem Ада #19 - Передача данных объектов Ада в потоке XML.

**Давайте начнем…**

Начиная с Ada 95 было возможно передать любой объект потоком. Используя 'Input/'Output или 'Read/'Write атрибуты, любой объект (теговый или не) может быть передан потоком, используя двоичное представление. Это означает, что объекты могут быть записаны в файл или отправлены по сокету, например.

Давайте возьмем простую иерархию объектов, чтобы проиллюстрировать эту функцию. У нас будет Point (x и координата y) и Pixel (Точка с цветом).

```
package Object is

   type Point is tagged record
      X, Y : Float;
   end record;

   type Color_Name is (Red, Green, Blue);

   type Pixel is new Point with record
      Color : Color_Name;
   end record;
end Object;
```

При записывании Point или Pixel первые байты в потоке являются внешними представления тега, а затем следуют значения атрибутов объекта.

```
declare
   File : File_Type;
   P    : Point'Class := ...;
begin
   Create (File, Out_File, "streamed.data");
   Point'Class'Output (Text_Streams.Stream (File), P);
   Close (File);
end;
```

Поток будет содержать что-то как (где в угловых скобках символьный шестнадцатеричный код):

```
   <01> <00> <00> <00> <0C> <00> <00> <00> O B J E C T . P I X E L
   <9A> <99> <99> <3f> <66> <66> <06> <41> <00>
```

Тег является важной частью, так как он будет использоваться для создания соответствующего экземпляра объекта из потока.

```
P := constant Point'Class :=
       Point'Class'Input (Text_Streams.Stream (File));
```

Все хорошо! Нет, есть немного недостающей возможности. Нет никакого способа управлять, как внешний тег передан потоком. На самом деле это - последовательность, и границы (ниже и верхний) сначала выведены в поток. Эти границы - простые числа, записанные в двоичном файле.

В приведенном выше примере мы имеем четыре первых байта для нижней границы (равных 1) и четыре следующих байта для верхней границы (равных 12), а затем двенадцать байтов для полного имени полного тега OBJECT.PIXEL.

В Ada 95 нет никакого способа передать текстовое представление потоком объектов!

Но хорошей новостью является ... Ada 2005 может это сделать. Ada 2005 идет дальше, добавляя поддержку для тонкого контроля внешнего представления любых объектов. Это означает, что теперь можно создать текстовое представление такого объекта, используя атрибуты `Class'Input` и `Class'Output`.

Давайте добавим недостающие части.

Сначала начнем с атрибутов 'Read и 'Write для вывода или чтения XML-представления Point или Pixel.

```
with Ada.Streams;

package Object is

   type Point is ...

   procedure Read (S : access Root_Stream_Type'Class; O : out Point);
   for Point'Read use Read;

   procedure Write
     (S : access Root_Stream_Type'Class; O : in Point);
   for Point'Write use Write;

   type Pixel is ...

   procedure Read (S : access Root_Stream_Type'Class; O : out Pixel);
   for Pixel'Read use Read;

   procedure Write
     (S : access Root_Stream_Type'Class; O : in Pixel);
   for Pixel'Write use Write;
```

Подпрограммы Read могут быть реализованы с использованием полнофункционального XML-парсера (анализатора XML), такого как XML / Ada. Для краткости мы будем использовать две очень простые XML-ориентированные процедуры:

```
procedure Skip_Tag
  (S      : access Ada.Streams.Root_Stream_Type'Class;
   Ending : in     Character := '>');
--  Skip the next tag on stream S, returns when Ending is found

function Get_Value
  (S : access Ada.Streams.Root_Stream_Type'Class) return String;
--  Returns the current value read on stream S
```

Используя эти подпрограммы, реализация 'Read и 'Write проста. Вот реализация для точки:

```
procedure Read (S : access Root_Stream_Type'Class; O : out Point) is
begin
   Skip_Tag (S); O.X := Float'Value (Get_Value (S)); Skip_Tag (S, ASCII.LF);
   Skip_Tag (S); O.Y := Float'Value (Get_Value (S)); Skip_Tag (S, ASCII.LF);
end Read;
```

```
procedure Write (S : access Root_Stream_Type'Class; O : in Point) is
begin
   String'Write (S, "   <x>"  & Float'Image (O.X) & "</x>" & ASCII.LF);
   String'Write (S, "   <y>"  & Float'Image (O.Y) & "</y>" & ASCII.LF);
end Write;
```

Последний недостающий элемент - это передача тега. Мы хотим, чтобы тег был просто: <point> и <pixel> (без привязки и просто имя объекта вместо полного имени, префиксного вложенным именем пакета). Чтобы установить правильное имя тега, мы используем атрибут External_Tag:

```
package Object is

   type Point is ...
   for Point'External_Tag use "point";

   type Pixel is ...
   for Pixel'External_Tag use "pixel";
```

Затем мы хотим подключить собственную XML-ориентированную реализацию атрибутов Class'Input и Class'Output. Это необходимо только для корневого типа. Point:

```
package Object is

   type Point is ...
   for Point'External_Tag use "point";

   procedure Class_Output
    (S : access Ada.Streams.Root_Stream_Type'Class; O : in Point'Class);
   for Point'Class'Output use Class_Output;

   function Class_Input
     (S : access Ada.Streams.Root_Stream_Type'Class) return Point'Class;
   for Point'Class'Input use Class_Input;
```

Процедура Class_Output должна вывести открытый тег XML, вывести сам объект, а затем закрыть тег XML. Довольно просто сделать; следующий код:

```
procedure Class_Output
  (S : access Ada.Streams.Root_Stream_Type'Class; O : in Point'Class) is
begin
   --  Write the opening tag
   Character'Write (S, '<');
   String'Write (S, Ada.Tags.External_Tag (O'Tag));
   String'Write (S, '>' & ASCII.LF);

   --  Write the object, dispatching call to Point/Pixel'Write
   Point'Output (S, O);

   --  Write the closing tag
   Character'Write (S, '<');
   Character'Write (S, '/');
   String'Write (S, Ada.Tags.External_Tag (O'Tag));
   String'Write (S, '>' & ASCII.LF);
end Class_Output;
```

И теперь последняя часть используется `Ada.Tags.Generic_Dispatching_Constructor`, которая создаст объект из потока с учетом тега объекта. Это должно быть полностью противоположно процедуре `Class_Output`. Открывается XML-тег открытия, затем объект с использованием `Generic_Dispatching_Constructor` и, наконец, закрывающий тег XML.

```
function Class_Input
  (S : access Ada.Streams.Root_Stream_Type'Class) return Point'Class
is
   function Dispatching_Input is
      new Ada.Tags.Generic_Dispatching_Constructor
        (T           => Point,
         Parameters  => Ada.Streams.Root_Stream_Type'Class,
         Constructor => Point'Input);
   Input     : String (1 .. 20);
   Input_Len : Natural := 0;
begin
   --  On the stream we have , we want to get "tag_name"
   --  Read first character, must be '<'
   Character'Read (S, Input (1));
   if Input (1) /= '<' then
      raise Ada.Tags.Tag_Error with "starting with " & Input (1);
   end if;

   --  Read the tag name
   Input_Len := 0;
   for I in Input'range loop
      Character'Read (S, Input (I));
      Input_Len := I;
      exit when Input (I) = '>';
   end loop;

   --  Check ending tag
   if Input (Input_Len) /= '>'
    or else Input_Len <= 1
   then -- Empty tag
      raise Ada.Tags.Tag_Error with "empty tag";
   else
     Input_Len := Input_Len - 1;
   end if;

   declare
      External_Tag : constant String := Input (1 .. Input_Len);
      O            : constant Point'Class := Dispatching_Input
                       (Ada.Tags.Internal_Tag (External_Tag), S);
      --  Dispatches to appropriate Point/Pixel'Input depending on
      --  the tag name.
   begin
      --  Skip closing object tag
      Skip_Tag (S); Skip_Tag (S, ASCII.LF);
      return O;
   end;
end Class_Input;
```

На этом этапе код, показанный в начале, будет работать без изменений. Тот факт, что объект передается потоком с использованием представления XML, прозрачен для пользователей пакета Object.

В заключение, для краткости, код as-is не выводит соответствующие XML-документы, так как отсутствует XML-заголовок и не учитывается существование нескольких корневых узлов. Доработка этих недостатков остается, как упражнение, для читателя.

**Связанный исходный код**

```
with Ada.Streams;
```

```
package Object is

   use Ada.Streams;

   --  Point  --

   type Point is tagged record
      X, Y : Float;
   end record;

   procedure Class_Output
     (S : access Ada.Streams.Root_Stream_Type'Class; O : in Point'Class);
   for Point'Class'Output use Class_Output;

   function Class_Input
     (S : access Ada.Streams.Root_Stream_Type'Class) return Point'Class;
   for Point'Class'Input use Class_Input;

   for Point'External_Tag use "point";

   procedure Write
     (S : access Root_Stream_Type'Class; O : in Point);
   for Point'Write use Write;

   procedure Read (S : access Root_Stream_Type'Class; O : out Point);
   for Point'Read use Read;

   procedure Display (O : in Point);

   --  Pixel  --

   type Color_Name is (Red, Green, Blue);

   type Pixel is new Point with record
      Color : Color_Name;
   end record;

   for Pixel'External_Tag use "pixel";

   procedure Write
     (S : access Root_Stream_Type'Class; O : in Pixel);
   for Pixel'Write use Write;

   procedure Read (S : access Root_Stream_Type'Class; O : out Pixel);
   for Pixel'Read use Read;

   overriding procedure Display (O : in Pixel);

end Object;

with Ada.Tags.Generic_Dispatching_Constructor;
with Ada.Text_IO;
with Ada.Text_IO.Text_Streams;

package body Object is

   use Ada.Text_IO;

   ----------------------------------------- XML parsing helper functions

   procedure Skip_Tag
     (S      : access Ada.Streams.Root_Stream_Type'Class;
      Ending : in     Character := '>');
   --  Skip the next tag on stream S, returns when Ending is found
```

```
function Get_Value
  (S : access Ada.Streams.Root_Stream_Type'Class) return String;
--  Returns the current value read on stream S

--------------
-- Skip_Tag --
--------------

procedure Skip_Tag
  (S      : access Ada.Streams.Root_Stream_Type'Class;
   Ending : in     Character := '>')
is
   C : Character;
begin
   loop
      Character'Read (S, C);
      exit when C = Ending;
   end loop;
end Skip_Tag;

---------------
-- Get_Value --
---------------

function Get_Value
  (S : access Ada.Streams.Root_Stream_Type'Class) return String
is
   Buffer : String (1 .. 100);
   K      : Positive := Buffer'First;
begin
   loop
      Character'Read (S, Buffer (K));
      exit when Buffer (K) = '<';
      K := K + 1;
   end loop;
   return Buffer (1 .. K - 1);
end Get_Value;

----------------------------------------- Point'Class

------------------
-- Class_Output --
------------------

procedure Class_Output
  (S : access Ada.Streams.Root_Stream_Type'Class; O : in Point'Class) is
begin
   --  Write the opening tag <tag_name>
   Character'Write (S, '<');
   String'Write (S, Ada.Tags.External_Tag (O'Tag));
   String'Write (S, '>' & ASCII.LF);

   --  Write the object, dispatching call to Point/Pixel'Write
   Point'Output (S, O);

   --  Write the closing tag </tag_name>
   String'Write (S, "</");
   String'Write (S, Ada.Tags.External_Tag (O'Tag));
   String'Write (S, '>' & ASCII.LF);
end Class_Output;

-----------------
-- Class_Input --
```

```
-----------------

function Class_Input
  (S : access Ada.Streams.Root_Stream_Type'Class) return Point'Class
is
   function Dispatching_Input is
      new Ada.Tags.Generic_Dispatching_Constructor
        (T           => Point,
         Parameters  => Ada.Streams.Root_Stream_Type'Class,
         Constructor => Point'Input);
   Input     : String (1 .. 20);
   Input_Len : Natural := 0;
begin
   --  On the stream we have <tag_name>, we want to get "tag_name"
   --  Read first character, must be '<'
   Character'Read (S, Input (1));
   if Input (1) /= '<' then
      raise Ada.Tags.Tag_Error with "Starting with " & Input (1);
   end if;

   --  Read tag
   Input_Len := 0;
   for I in Input'range loop
      Character'Read (S, Input (I));
      Input_Len := I;
      exit when Input (I) = '>';
   end loop;

   --  Check ending tag
   if Input (Input_Len) /= '>'
     or else Input_Len <= 1
   then -- Empty tag
      raise Ada.Tags.Tag_Error with "empty tag";
   else
      Input_Len := Input_Len - 1;
   end if;

   declare
      External_Tag : constant String := Input (1 .. Input_Len);
      O            : constant Point'Class := Dispatching_Input
                        (Ada.Tags.Internal_Tag (External_Tag), S);
      --  Dispatches to appropriate Point/Pixel'Input depending on
      --  the tag name.
   begin
      --  Skip closing object tag
      Skip_Tag (S, ASCII.LF);
      return O;
   end;
end Class_Input;

----------------------------------------- Point

-------------
-- Display --
-------------

procedure Display (O : in Point) is
begin
   Put_Line ("*** A point");
   Point'Output (Text_Streams.Stream (Current_Output), O);
end Display;

----------
-- Read --
```

```
   ----------

   procedure Read (S : access Root_Stream_Type'Class; O : out Point) is
   begin
      Skip_Tag (S); O.X := Float'Value (Get_Value (S)); Skip_Tag (S,
ASCII.LF);
      Skip_Tag (S); O.Y := Float'Value (Get_Value (S)); Skip_Tag (S,
ASCII.LF);
   end Read;

   -----------
   -- Write --
   -----------

   procedure Write (S : access Root_Stream_Type'Class; O : in Point) is
   begin
      String'Write (S, "   <x>"  & Float'Image (O.X) & "</x>" & ASCII.LF);
      String'Write (S, "   <y>"  & Float'Image (O.Y) & "</y>" & ASCII.LF);
   end Write;

   ---------------------------------------- Pixel

   -------------
   -- Display --
   -------------

   overriding procedure Display (O : in Pixel) is
   begin
      Put_Line ("*** A pixel");
      Pixel'Output (Text_Streams.Stream (Current_Output), O);
   end Display;

   ----------
   -- Read --
   ----------

   procedure Read (S : access Root_Stream_Type'Class; O : out Pixel) is
   begin
      Read (S, Point (O));
      Skip_Tag (S);
      O.Color := Color_Name'Value (Get_Value (S));
      Skip_Tag (S, ASCII.LF);
   end Read;

   -----------
   -- Write --
   -----------

   procedure Write (S : access Root_Stream_Type'Class; O : in Pixel) is
   begin
      Write (S, Point (O));
      String'write
        (S, "   <color>"
           & Color_Name'Image (O.Color) & "</color>" & ASCII.LF);
   end Write;

end Object;

with Ada.Text_IO;
with Ada.Text_IO.Text_Streams;

with Object;

procedure Main is
```

```
   use Ada;
   use Ada.Text_IO;
   use Object;

   File : Text_IO.File_Type;

begin
   --  Write some objects

   declare
      P  : Point := (6.8, 0.1);
      CP : Pixel := (1.2, 8.4, Red);
   begin
      Create (File, Out_File, "data");
      Point'Class'Output (Text_Streams.Stream (File), CP);
      Point'Class'Output (Text_Streams.Stream (File), P);
      Close (File);
   end;

   --  Read them back

   Open (File, In_File, "data");

   declare
      P1 : constant Point'Class :=
             Point'Class'Input (Text_Streams.Stream (File));
      P2 : constant Point'Class :=
             Point'Class'Input (Text_Streams.Stream (File));
   begin
      P1.Display; New_Line;
      P2.Display; New_Line;
   end;

   Close (File);
end Main;
```

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

**Обсуждение…**

1 Комментарии

Кристоф Грин
29 ноября 2007 г.

К сожалению, в тексте есть некоторые опечатки :-(
К счастью, исходный код в порядке :-)