Ada pour développer sur Internet

Jean-Pierre Rosen
Adalog
19-21 rue du 8 mai 1945
94110 ARCUEIL
FRANCE

E-m: rosen@adalog.fr
URL: http://www.adalog.fr

Résumé

Ada permet de réaliser des applications liées à l'Internet aussi bien que n'importe quel langage. Mais le terme "application Internet" englobe des besoins différents. On distingue les services de pages Web dynamiques côté serveur (interface CGI), les animations de page côté client (Java), et l'écriture d'application autonomes utilisant Internet. Pour chacune de ces possibilités, des outils sont disponibles facilitant l'écriture d'applications en Ada.

1 Introduction

On peut bien entendu réaliser en Ada tout ce qui peut se faire dans d'autres langages. Mais en pratique, cela peut être plus ou moins facile selon les bibliothèques dont on dispose. Dans ce papier, nous présentons rapidement les différentes facettes de la programmation Internet, et faisons un panorama des services permettant de développer des applications liées à Internet en Ada, aussi bien en tant que client qu'en tant que serveur.

2 L'interface CGI

L'interface CGI [1] (*Common Gateway Interface*, à ne pas confondre avec la norme graphique *Computer Graphic Interface*) permet de créer des pages Web dynamiques, c'est à dire dont le contenu est fourni par l'exécution d'un programme et non sous la forme d'un fichier figé.

2.1 Principe de fonctionnement du CGI

Lorsqu'un serveur HTTP se voit demander une URL désignant un fichier figurant sous un répertoire spécial (souvent appelé CGI-BIN), il le considère comme un exécutable, le lance, et renvoie le produit de sa sortie standard (le `STANDARD_OUTPUT` de Ada) au client. C'est aussi simple que cela.

Enfin presque... Tout d'abord, le serveur doit connaître le type des données renvoyées par le programme. Pour cela, le texte doit commencer par une ou plusieurs lignes fournissant des indications sur le contenu de ce qui suit. Ces lignes d'en-tête sont séparées du contenu proprement dit par une ligne vide. Cela peut être, par exemple :

```
Content-type: text/html
```

pour indiquer du HTML normal, ou :

```
Location: http://monserveur.fr/reponse
```

pour indiquer que la réponse se trouve dans un autre fichier.

Ensuite, il est bien évident que les pages dynamiques n'ont d'intérêt que si elles permettent de répondre à une demande particulière du client ; il faut donc que le client passe des données à l'application... et que celle-ci soit capable de les récupérer.

Côté client, cela se fait en mettant dans l'URL, derrière le nom du fichier, des indications de la forme `Clé=Valeur`, séparées par des points d'interrogation. Dans les valeurs, tous les espaces sont remplacés par des caractères "+", et tous les caractères non alphanumériques sont représentés par leur valeur hexadécimale sous la forme "%xx".

Noter que rien n'est imposé quant à la façon dont ces paramètres sont mis par le client dans l'URL : il est parfaitement possible de les mettre directement dans une référence (balise HREF), mais en général ils proviendront d'un formulaire. Dans ce cas, le descriptif du formulaire contient l'URL du fichier CGI à activer, et c'est le logiciel client qui rajoute les noms et les valeurs des différents champs du formulaire derrière le nom du fichier.

Il existe plusieurs façons de récupérer ces valeurs côté serveur, mais nous ne présenterons ici que la plus habituelle. Avant de lancer l'application CGI, le serveur HTTP initialise des variables d'environnement avec diverses informations. La plus intéressante est `QUERY_STRING`, qui contient la chaîne de caractère suivant le premier "?" dans l'URL, c'est-à-dire tout l'ensemble des paramètres. Le logiciel n'a donc qu'à récupérer le contenu de cette variable, la décoder, et produire ce qu'il veut en sortie.

Remarquer qu'avec l'interface CGI, l'application est lancée sur le *serveur*, ce qui pourrait poser des problèmes de sécurité. C'est pour cette raison que seuls les exécutables "autorisés" (ceux du répertoire spécialement marqué dans le

serveur) peuvent être lancés de cette façon. En revanche, il n'y a aucune restriction au type d'exécutable : ce peut-être des programmes développés dans un langage conventionnel, ou des scripts en Shell, PERL, Tcl...

2.2 L'interface Ada-CGI

Il existe un paquetage [2] pour faciliter l'écriture d'applications CGI en Ada. Ce paquetage a été écrit par David Wheeler, qui en autorise l'utilisation sans aucune restriction[1] du moment que l'on n'oublie pas de mentionner que c'est lui qui l'a écrit... ce qui est fait maintenant.

A priori, la seule chose réellement nécessaire pour écrire un programme CGI est la faculté de récupérer la valeur d'une variable d'environnement. En pratique, il faut analyser la chaîne pour séparer les différentes paires `Clé=Valeur` et décoder les espaces et les encodages hexadécimaux. Ce sont ces services qui sont fournis par le paquetage CGI, ainsi que des utilitaires facilitant la production de l'HTML en sortie.

La spécification complète du paquetage est donnée en annexe 1. Noter que tous les sous-programmes produisant des chaînes de caractères sont doublés pour retourner soit des `String`, soit des `Unbounded_String`. Les principales fonctionnalités fournies sont :

1) *La gestion des paramètres*. Les différentes fonctions `Value` permettent de récupérer la valeur associée à une clé (le paramètre `Index` désignant l'occurrence de la clé voulue, celle-ci pouvant apparaître plusieurs fois). Les fonctions `Key_Exists` pemettent de savoir si une clé est présente, et `Key_Count` le nombre d'occurrences. Il est également possible (fonctions `Key` et `Value`) de connaître la clé (et la valeur associée) se trouvant à une position donnée dans la liste. Le nombre total de paramètres est fourni par la fonction `Argument_Count`.
2) *La production d'HTML*. La procédure `Put_CGI_Header` permet de fournir la première ligne d'en-tête. Les procédures `Put_HTML_Head` et `Put_HTML_Tail` impriment les début/fin standard de document HTML. La procédure `Put_HTML_Head` sort un texte sous forme de titre standard (balises H*n*, avec *n*=1 par défaut). Enfin la procédure `Put_Error_Message` génère une page complète contenant un message d'erreur.
3) *La gestion des lignes*. Une valeur de paramètre peut s'étendre sur plusieurs lignes. La fonction `Line_Count` donne le nombre de lignes dans une valeur, et la fonction `Line_Count_Of_Value` donne le nombre de lignes d'une valeur associée à une clé. De même, les fonctions Line et Line_Of_Value permettent de récupérer la n^{ième} ligne d'une chaîne, ou d'une valeur associée à une clé.
4) *Des fonctions de plus bas niveau*. La procédure `Put_Variables` imprime (sous forme HTML) les valeurs de tous les paramètres (pour la mise au point), et la fonction `Get_Environment` permet de récupérer la valeur de n'importe quelle variable d'environnement.

Une dernière remarque : si ce paquetage ne représente pas une quantité de travail énorme, il est bien utile en affranchissant le programmeur de toute la partie fastidieuse du décodage des paramètres. De plus, c'est un bon exemple d'utilisation des nouvelles fonctionnalités Ada 95 de manipulations de chaînes de caractères.

2.3 UnCGI

Une autre façon de simplifier le décodage des paramètres CGI est d'utiliser UnCGI [3]. Ce programme agit comme un pré-processeur qui décode les associations Clés/Valeurs, crée des variables d'environnement portant le nom des clés, puis appelle un programme utilisateur... qui peut être écrit en Ada, comme dans n'importe quel autre langage.

Pour être honnête, cette interface est surtout utile avec les langages de script, pour lesquels il est plus simple d'accéder aux variables d'environnement que de faire des entrées/sorties.

3 Les applets Java

Contrairement au CGI, les applets permettent d'exécuter des programmes sur l'ordinateur du *client*, c'est à dire sur la machine qui demande l'affichage de la page Web. C'est surtout utile pour faire des animations, ou des interfaces plus sophistiquées que ce qu'autorise l'HTML standard.

Le fournisseur d'une page HTML ignorant tout de la machine qui l'affiche, il fallait définir pour les applets un format exécutable indépendant de toute architecture matérielle. Java définit donc une *machine virtuelle*, exécutant un code machine appelé *Byte-code*. Une applet n'est donc physiquement qu'une référence à un fichier contenant du byte-code. Lors du chargement de la page, ce code est transmis à un interpréteur (interne au *browser*) qui l'exécutera. L'interpréteur aura la charge de contrôler le code téléchargé, et en particulier de limiter ou d'interdire des fonctionnalités qui pourraient être dangereuses pour la machine sur laquelle il s'exécute, comme d'effacer ou modifier des fichiers locaux !

Noter que la machine virtuelle Java est multi-tâche, ce qui est très utile pour réaliser des animations. Toutefois, les fonctionnalités de synchronisation et de communication offertes restent très élémentaires, comparées aux possibilités d'Ada.

Sun, qui est à l'origine de Java, a développé un langage spécifique, le *langage* Java, et fournit un compilateur traduisant ce langage en byte-code. Ce compilateur fait partie d'un ensemble de composants, le JDK (*Java Development Kit*), qui est téléchargeable gratuitement [4]. Le langage Java est *purement* orienté objet, c'est-à-

[1] En particulier, sans les restrictions de la licence GPL.

dire que *tout* est défini sous forme de classes, et que, à part quelques types (très) élémentaires, *toutes* les entités manipulées sont à sémantique de référence. Cet aspect est particulièrement déroutant pour les développeurs habitués à des langages conventionnels, et conduit à d'importantes inefficacités, notamment dans les manipulations de tableaux.

Un des intérêts de Java est le grand nombre de classes fournies avec l'environnement, qui permettent de développer des interfaces sophistiquées, de jouer de la musique, d'établir des liaisons internet, etc. On notera cependant que les classes de base résident sur la machine *hôte* ; comme les fonctionnalités ont été rajoutées progressivement dans les différentes versions de l'environnement, les plus récentes d'entre-elles (et souvent les plus intéressantes) risquent de ne pas fonctionner avec les *browsers* les plus anciens, ce qui impose de se limiter si l'on veut que les applets fonctionnent partout.

3.1 Principe de fonctionnement des applets Java

Une applet devant fonctionner dans une page HTML est un objet appartenant à une classe dérivée de la classe `Applet`. Cette classe fournit un certain nombre de méthodes, dont certaines devront être en général redéfinies. On trouve ainsi `Paint`, qui est chargée de redessiner le contenu de l'applet, méthode appelée par le *browser* à chaque fois qu'il est nécessaire de remettre à jour l'écran ; `init`, qui permet d'initialiser l'applet, appelée par le *browser* la première fois que la page est chargée ; `start` et `stop`, appelées par le *browser* respectivement lorsque la page devient visible ou est cachée, ce qui évite de laisser "courir" des animations lorsqu'elles ne sont pas visibles ; enfin `destroy`, qui doit libérer les ressources utilisées, qui est appelée par le *browser* lorsque la page disparaît définitivement.

3.2 Ecriture d'applets Java en Ada

A priori, rien n'impose un langage particulier pour produire le byte-code ; il existe un grand nombre de compilateurs pour des langages variés (118 selon [5]) qui génèrent du byte-code au lieu de code natif, y-compris plusieurs assembleurs. Cela ne présente pas de difficulté théorique, il ne s'agit en fait que de compilateurs croisés ayant la machine virtuelle Java pour cible.

En particulier, il est possible de générer du byte-code, et donc d'écrire des applets, à partir d'Ada[1]. Cela a été réalisé par la technologie AppletMagic d'Intermetrics [7] (aujourd'hui Averstar), que l'on retrouve dans les compilateurs basés sur ce frontal, notamment le compilateur Aonix [8]. Plus récemment, une version spéciale du compilateur GNAT (nommée JGNAT) a été développée pour la machine Java par ACT [9].

[1] Nous proposons d'appeler les applets écrites en Ada des "Adapplets". Voir la page qui leur est consacrée sur le site Adalog [6].

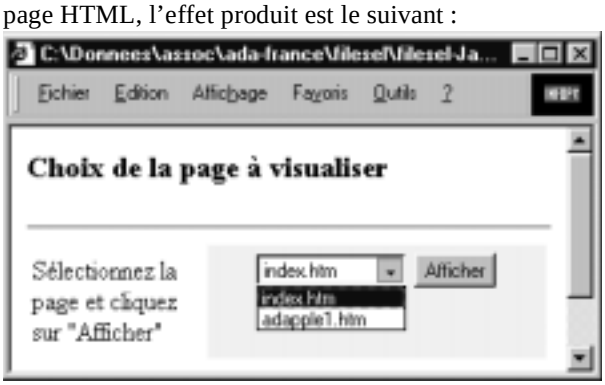
JGNAT n'ayant pas encore été diffusé à la date d'écriture de ces lignes, nous donnerons juste quelques indications concernant la technologie Intermetrics. Une classe Java correspond à un paquetage Ada contenant la déclaration d'un type étiqueté *dont le nom doit obligatoirement être celui du paquetage auquel on ajoute "_Obj"*. A part cela, il s'utilise comme n'importe que type étiqueté. Attention toutefois à un piège : Java dépendant des majuscules/minuscules, il faut que le nom du paquetage donné en tête de spécification respecte exactement la mise en majuscule de la classe Java.

Toutes les classes Java ont été interfacées et sont disponibles depuis Ada. Il existe également un paquetage `Interfaces.Java` qui assure la compatibilité des types de données et facilite la correspondance entre chaînes de caractères Java et Ada. Grâce à cela, il n'est pas plus difficile d'utiliser les classes Java que n'importe quelle autre bibliothèque pour laquelle un *binding* Ada a été défini.

Noter également que le compilateur produit des classes compilées absolument standard, et qu'il est donc possible d'utiliser des classes écrites en Ada depuis n'importe quel programme écrit en langage Java.

3.3 Exemple d'Adapplet

A titre d'exemple, nous fournissons en annexe 2 une petite applet en Ada qui affiche un menu déroulant contenant des pages web à choisir, et un bouton pour provoquer l'affichage de la page désirée. Une fois intégrée dans une page HTML, l'effet produit est le suivant :



Dans cette applet, on a juste redéfini la méthode `Init` qui met en place les éléments (menu déroulant, bouton) et la méthode `action` qui définit les traitements à effectuer en cas d'évènement concernant l'applet. Remarquer la présence d'un opérateur unaire "+" qui transforme les chaînes Ada en chaînes Java.

Des exemples d'Adapplets librement réutilisables (y compris une version plus évoluée de celle-ci) sont disponibles depuis la page des composants Adalog [10].

4 L'accès à Internet depuis Ada

Le dernier volet des applications possibles de Ada dans le monde de l'Internet concerne le développement d'applications *utilisant* Internet, telles que des logiciels de courrier électronique, des *browsers*, des transferts de

fichiers, etc. La problématique particulière à Internet, pour de telles applications, se trouve évidemment dans l'interfaçage avec le réseau.

4.1 Les ANC (*Ada Network Components*)

Il s'agit d'un ensemble de paquetages développés à l'ENST qui fournissent une interface avec les *sockets* qui constituent l'inteface système normale avec le réseau. Ces composants sont sous licence GPL "assouplie" (modèle de la bibliothèque GNAT), et sont disponibles depuis [11]. Cette interface peut être qualifiée de "moyenne" (ni mince, ni épaisse) dans la mesure où le typage a été adapté à Ada, mais où les fonctionnalités fournies correspondent exactement à l'interface socket du système.

Nous donnons à titre d'exemple, en annexe 3, la spécification du paquetage `Sockets`.

4.2 AdaJNI

Le JNI (*Java Native Interface*) est une interface permettant d'exécuter du code Java depuis un programme en C (ainsi, d'ailleurs, qu'inversement). Cela revient, en gros, à incorporer dans le programme une machine virtuelle, et à lui faire exécuter le code des classes Java que l'on souhaite utiliser.

La société Ainslie-Software [12] vend une interface appelée AdaJNI qui fournit les mêmes services à des programmes Ada. Il devient ainsi possible d'utiliser tous les services de la riche bibliothèque Java depuis un programme Ada natif. En particulier, la bibliothèque Java comporte de nombreuses classes d'interface avec le réseau, ce qui en fait une alternative intéressante pour l'écriture d'applications Internet portables.

Il existe également une version "libre" de l'interface Ada-JNI, appelée Café1815 [13], mais elle est actuellement en version Alpha. A suivre...

5 Et si cela ne suffit pas...

Si malgré tout les outils précédants ne répondent pas à vos besoins, rappelons qu'il est possible d'utiliser en Ada n'importe quelle bibliothèque interfacée avec le langage C.

L'adaptation d'une bibliothèque C peut être grandement facilitée par l'utilisation de l'outil C2ADA [14] qui traduit (presque) automatiquement un fichier d'en-tête C (".h") en spécification Ada. Il est également possible d'utiliser des services fournis sous forme de DLL ou d'utiliser l'interface COM de Microsoft ; tous les renseignements à ce sujet peuvent être obtenus depuis l'excellent site AdaPower [15], ou la fameuse "page Ada" de Pascal Obry [16].

Et n'oubliez pas que si vous avez eu un besoin qui vous a conduit à créer une nouvelle interface avec une bibliothèque existante, d'autres pourraient bien avoir le même besoin... Pensez à mettre votre travail à disposition sur Internet ; de nombreux sites sont prêts à héberger le fruit de vos efforts !

6 Conclusion

Les mêmes outils sont aujourd'hui disponibles pour développer des applications Internet avec Ada qu'avec les autres langages. On doit cet (heureux) état de fait aux nouvelles facilités d'interfaçage apportées par Ada 95, et au développement de l'Internet lui-même, qui permet de mettre le travail de chacun à disposition de tous.

Même pour ce type d'application, où le C est traditionnellement utilisé, l'alternative est disponible ; le choix du langage doit donc être guidé par les critères habituels de génie logiciel, maintenance, etc. Encore faut-il savoir, et faire savoir, que le choix existe !

Webliographie

[1] **CGI** : http://hoohoo.ncsa.uiuc.edu/cgi

[2] **Ada-CGI** : http://goanna.cs.mit.edu.au/~dale/cgi/cgi.htm

[3] **UnCGI** : http://www.midwinter.com/~koreth/uncgi.html

[4] **JDK** : http://java.sun.com/products/jdk/1.2/

[5] **Langages pour la Java machine** : http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

[6] **Adapplets**: http://pro.wanadoo.fr/adalog/adapple1.htm

[7] **AppletMagic** : http://www.appletmagic.com

[8] **Aonix** : http://www.aonix.fr

[9] **ACT** :http://www.act-europe.fr

[10] **Composants Adalog** : http://pro.wanadoo.fr/adalog/compo1.htm

[11] **Ada Network Components** : http://www.inf.enst.fr/ANC

[12] **Ainslie Software** : http://www.ainslie-software.com

[13] **Café 1815** : http://www.acenet.com.au/~gbull/

[14] **C2ADA** : http://www.inmet.com/~mg/c2ada/c2ada.html

[15] **AdaPower** : http://www.AdaPower.com

[16] **Pascal Obry** : http://ourworld.compuserve.com/homepages/pascal_obry/ada.html

Annexe 1 : spécification du paquetage CGI

```
with Ada.Strings.Unbounded;  use  Ada.Strings.Unbounded;

package CGI is
-- This package is an Ada 95 interface to the "Common Gateway Interface" (CGI).
-- This package makes it easier to create Ada programs that can be
-- invoked by World-Wide-Web HTTP servers using the standard CGI interface.
-- CGI is rarely referred to by its full name, so the package name is short.
-- General information on CGI is available at "http://hoohoo.ncsa.uiuc.edu/cgi/".

-- Developed by (C) David A. Wheeler (wheeler@ida.org) June 1995.
-- This is version 1.0.

-- This was inspired by a perl binding by Steven E. Brenner at
--   "http://www.bio.cam.ac.uk/web/form.html"
-- and another perl binding by L. Stein at
--   "http://www-genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html"
-- A different method for interfacing binding Ada with CGI is to use the
-- "Un-CGI" interface at "http://www.hyperion.com/~koreth/uncgi.html".

-- This package automatically loads information from CGI on program start-up.
-- It loads information sent from "Get" or "Post" methods and automatically
-- splits the data into a set of variables that can be accessed by position or
-- by name.  An "Isindex" request is translated into a request with a single
-- key named "isindex" with its Value as the query value.

-- This package provides two data access methods:
-- 1) As an associative array; simply provide the key name and the
--    value associated with that key will be returned.
-- 2) As a sequence of key-value pairs, indexed from 1 to Argument_Count.
--    This is similar to Ada library Ada.Command_Line.
-- The main access routines support both String and Unbounded_String.

-- See the documentation file for more information and sample programs.

function Parsing_Errors return Boolean;  -- True if Error on Parse.
function Input_Received return Boolean;  -- True if Input Received.
function Is_Index       return Boolean;  -- True if an Isindex request made.
  -- An "Isindex" request is turned into a Key of "isindex" at position 1,
  -- with Value(1) as the actual query.

type CGI_Method_Type is (Get, Post, Unknown);

function CGI_Method return CGI_Method_Type;  -- True if Get_Method used.

-- Access data as an associative array - given a key, return its value.
-- The Key value is case-sensitive.
-- If a key is required but not present, raise Constraint_Error;
-- otherwise a missing key's value is considered to be "".
-- These routines find the Index'th value of that key (normally the first one).
function Value(Key : in Unbounded_String; Index : in Positive := 1;
               Required : in Boolean := False) return Unbounded_String;
function Value(Key : in String; Index : in Positive := 1;
               Required : in Boolean := False) return String;
function Value(Key : in Unbounded_String; Index : in Positive := 1;
              Required : in Boolean := False) return String;
function Value(Key : in String; Index : in Positive := 1;
               Required : in Boolean := False) return Unbounded_String;

-- Was a given key provided?
function Key_Exists(Key : in String; Index : in Positive := 1) return Boolean;
```

```
function Key_Exists(Key : in Unbounded_String; Index : in Positive := 1) return Boo-
lean;

-- How many of a given key were provided?
function Key_Count(Key : in String) return Natural;
function Key_Count(Key : in Unbounded_String) return Natural;


-- Access data as an ordered list (it was sent as Key=Value);
-- Keys and Values may be retrieved from Position (1 .. Argument_Count).
-- Constraint_Error will be raised if (Position < 1 or Position > Argument_Count)
function Argument_Count return Natural;          -- 0 means no data sent.
function Key(Position : in Positive) return Unbounded_String;
function Key(Position : in Positive) return String;
function Value(Position : in Positive) return Unbounded_String;
function Value(Position : in Positive) return String;

-- The following are helpful subprograms to simplify use of CGI.

function My_URL return String; -- Returns the URL of this script.

procedure Put_CGI_Header(Header : in String := "Content-type: text/html");
-- Put CGI Header to Current_Output, followed by two carriage returns.
-- This header determines what the program's reply type is.
-- Default is to return a generated HTML document.

procedure Put_HTML_Head(Title : in String; Mail_To : in String := "");
-- Puts to Current_Output an HTML header with title "Title".  This is:
--   <HTML><HEAD><TITLE> _Title_ </TITLE>
--   <LINK REV="made" HREF="mailto:  _Mail_To_ ">
--   </HEAD><BODY>
-- If Mail_To is omitted, the "made" reverse link is omitted.

procedure Put_HTML_Heading(Title : in String; Level : in Positive);
-- Put an HTML heading at the given level with the given text.
-- If level=1, this puts:  <H1>Title</H1>.

procedure Put_HTML_Tail;
-- This is called at the end of an HTML document. It puts to Current_Output:
--   </BODY></HTML>

procedure Put_Error_Message(Message : in String);
-- Put to Current_Output an error message.
-- This Puts an HTML_Head, an HTML_Heading, and an HTML_Tail.
-- Call "Put_CGI_Header" before calling this.

procedure Put_Variables;
-- Put to Current_Output all of the CGI variables as an HTML-formatted String.

function Line_Count (Value : in String) return Natural;
-- Given a value that may have multiple lines, count the lines.
-- Returns 0 if Value is the empty/null string (i.e., length=0)

function Line_Count_of_Value (Key : String) return Natural;
-- Given a Key which has a Value that may have multiple lines,
-- count the lines.  Returns 0 if Key's Value is the empty/null
-- string (i.e., length=0) or if there's no such Key.
-- This is the same as Line_Count(Value(Key)).

function Line (Value : in String; Position : in Positive)
               return String;
-- Given a value that may have multiple lines, return the given line.
```

```
   -- If there's no such line, raise Constraint_Error.

   function Value_of_Line (Key : String; Position : Positive)
                           return String;
   -- Given a Key which has a Value that may have multiple lines,
   -- return the given line.  If there's no such line, raise Constraint_Error.
   -- If there's no such Key, return the null string.
   -- This is the same as Line(Value(Key), Position).

   function Get_Environment(Variable : in String) return String;
   -- Return the given environment variable's value.
   -- Returns "" if the variable does not exist.

   end CGI;
```

Annexe 2 : Exemple d'applet Java en Ada

```
   ------------------------------------------------------------------------------
   -- Adapplet filesel                                                         --
   -- (C) Copyright 1998 ADALOG                                                --
   ------------------------------------------------------------------------------
   with Java.Lang, Java.Applet.Applet, Java.Awt.Event;
   use  Java.Lang, Java.Applet.Applet;

   -- Clauses for private part only:
   with Java.Awt.Button, Java.Awt.Choice;
   package filesel is
      type Filesel_Obj is new Applet_Obj with private;

      procedure init   (Obj  : access Filesel_Obj);
      function  action (Obj  : access Filesel_Obj;
                        Evt  : Java.Awt.Event.Event_Ptr;
                        What : Object_Ptr)
        return Boolean;

   private
      use Java.Awt.Button, Java.Awt.Choice;

      type Filesel_Obj is new Applet_Obj with
         record
            Download_Button : Button_Ptr;
            Format_List     : Choice_Ptr;
         end record;
   end filesel;

   with Java.Awt.Component, Java.Awt.Color;
   with Java.Applet.AppletContext;
   with Java.Net.URL;
   with Interfaces.Java; use Interfaces.Java;
   package body filesel is

      procedure init (Obj : access Filesel_Obj) is
         use Java.Awt.Component, Java.Awt.Color;

         Junk : Component_Ptr;
      begin
         setBackground (Obj, new_Color(255, 255, 128));

         Obj.Format_List := new_Choice;
         addItem (Obj.Format_List, +"index.htm");
         addItem (Obj.Format_List, +"adapple1.htm");
         Junk := add (Obj, +"WEST", Component_Ptr (Obj.Format_List));
```

```
      Obj.Download_Button := new_Button (+"Afficher");
      Junk := add (Obj, +"EAST", Component_Ptr (Obj.Download_Button));

      Validate (Obj);
   end Init;

   function action (Obj  : access Filesel_Obj;
                    Evt  : Java.Awt.Event.Event_Ptr;
                    What : Object_Ptr)
     return Boolean
   is
      use Java.Net.URL, Java.Applet.AppletContext;
   begin
      if Evt.Target = Object_Ptr (Obj.Download_Button) then
         showDocument (getAppletContext(Obj),
                       new_URL (getDocumentBase(Obj),
                                getSelectedItem (Obj.Format_List)));
      end if;

      return True;
   end action;

end filesel;
```

Annexe 3 : Spécification du paquetage Sockets (ANC)

```
------------------------------------------------------------------------------
--                                                                          --
--                          ADASOCKETS COMPONENTS                           --
--                                                                          --
--                              S O C K E T S                               --
--                                                                          --
--                                 S p e c                                  --
--                                                                          --
--                         $ReleaseVersion: 0.1.3 $                         --
--                                                                          --
--  Copyright (C) 1998  École Nationale Supérieure des Télécommunications  --
--                                                                          --
--   AdaSockets is free software; you can  redistribute it and/or modify    --
--   it  under terms of the GNU  General  Public License as published by    --
--   the Free Software Foundation; either version 2, or (at your option)    --
--   any later version.   AdaSockets is distributed  in the hope that it    --
--   will be useful, but WITHOUT ANY  WARRANTY; without even the implied    --
--   warranty of MERCHANTABILITY   or FITNESS FOR  A PARTICULAR PURPOSE.    --
--   See the GNU General Public  License  for more details.  You  should    --
--   have received a copy of the  GNU General Public License distributed    --
--   with AdaSockets; see   file COPYING.  If  not,  write  to  the Free    --
--   Software  Foundation, 59   Temple Place -   Suite  330,  Boston, MA    --
--   02111-1307, USA.                                                       --
--                                                                          --
--   As a special exception, if  other  files instantiate generics  from    --
--   this unit, or  you link this  unit with other  files to produce  an    --
--   executable,  this  unit does  not  by  itself cause  the  resulting    --
--   executable to be  covered by the  GNU General Public License.  This    --
--   exception does  not  however invalidate any  other reasons  why the    --
--   executable file might be covered by the GNU Public License.            --
--                                                                          --
--   The main repository for this software is located at:                    --
--       http://www-inf.enst.fr/ANC/                                        --
--                                                                          --
------------------------------------------------------------------------------
```

```
with Ada.Streams;
with Interfaces.C;
package Sockets is
   type Socket_FD is tagged private;
   --  A socket

   type Socket_Domain is (AF_INET);
   --  AF_INET: Internet sockets (yes, should be PF_INET, but they hold the
   --  same value)

   type Socket_Type is (SOCK_STREAM, SOCK_DGRAM);
   --  SOCK_STREAM: Stream mode   (TCP)
   --  SOCK_DGRAM:  Datagram mode (UDP, Multicast)

   procedure Socket
     (Sock   : out Socket_FD;
      Domain : in Socket_Domain := AF_INET;
      Typ    : in Socket_Type   := SOCK_STREAM);
   --  Create a socket of the given mode

   Connection_Refused : exception;

   procedure Connect
     (Socket : in Socket_FD;
      Host   : in String;
      Port   : in Positive);
   --  Connect a socket on a given host/port. Raise Connection_Refused if
   --  the connection has not been accepted by the other end.

   procedure Bind
     (Socket : in Socket_FD;
      Port   : in Positive);
   --  Bind a socket on a given port

   procedure Listen
     (Socket     : in Socket_FD;
      Queue_Size : in Positive := 5);
   --  Create a socket's listen queue

   type Socket_Level is (SOL_SOCKET, IPPROTO_IP);

   type Socket_Option is (SO_REUSEADDR, IP_MULTICAST_TTL,
                          IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP,
                          IP_MULTICAST_LOOP);

   procedure Setsockopt
     (Socket  : in Socket_FD'Class;
      Level   : in Socket_Level := SOL_SOCKET;
      Optname : in Socket_Option;
      Optval  : in Integer);
   --  Set a socket option

   generic
      Level   : Socket_Level;
      Optname : Socket_Option;
      type Opt_Type is private;
   procedure Customized_Setsockopt (Socket : in Socket_FD'Class;
                                    Optval : in Opt_Type);
   --  Low level control on setsockopt

   procedure Accept_Socket (Socket     : in Socket_FD;
```

```
                           New_Socket : out Socket_FD);
   --  Accept a connection on a socket

   Connection_Closed : exception;

   procedure Send (Socket : in Socket_FD;
                   Data   : in Ada.Streams.Stream_Element_Array);
   --  Send data on a socket. Raise Connection_Closed if the socket
   --  has been closed.

   function Receive (Socket : Socket_FD;
                     Max    : Ada.Streams.Stream_Element_Count := 4096)
     return Ada.Streams.Stream_Element_Array;
   --  Receive data from a socket. May raise Connection_Closed

   procedure Receive (Socket : in Socket_FD'Class;
                      Data   : out Ada.Streams.Stream_Element_Array);
   --  Fill data from a socket. Raise Connection_Closed if the socket has
   --  been closed before the end of the array.

   type Shutdown_Type is (Receive, Send, Both);

   procedure Shutdown (Socket : in Socket_FD;
                       How    : in Shutdown_Type := Both);
   --  Close a previously opened socket

   ---------------------------------
   -- String-oriented subprograms --
   ---------------------------------

   procedure Put (Socket : in Socket_FD'Class;
                  Str    : in String);
   --  Send a string on the socket

   procedure New_Line (Socket : in Socket_FD'Class;
                       Count  : in Natural := 1);
   --  Send CR/LF sequences on the socket

   procedure Put_Line (Socket : in Socket_FD'Class;
                       Str    : in String);
   --  Send a string + CR/LF on the socket

   function Get (Socket : Socket_FD'Class) return String;
   --  Get a string from the socket

   function Get_Line (Socket : Socket_FD'Class) return String;
   --  Get a full line from the socket. CR is ignored and LF is considered
   --  as an end-of-line marker.

private
   ....
end Sockets;
```