Experiences in Developing a Typical Web/Database Application

J-P. Rosen
Adalog
19-21 rue du 8 mai 1945
94110 ARCUEIL
FRANCE
+33 -1 41 24 31 40
rosen@adalog.fr

ABSTRACT

This paper describes Gesem, an application developed internally by Adalog for managing the registration to its training sessions. The application features a Web interface that uses AWS, an interface to the MySQL DBMS (over ODBC), and a local interface that uses GTK. The project explored various solutions, and identified a number of design patterns that made the development of new functionalities very straightforward. The experience gained in this project can be reused for any development in a similar environment.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *Object-oriented programming.*

General Terms

Design, Reliability, Languages.

Keywords

Ada, AWS, Data-base, GTK, web server, design patterns.

1. THE CONTEXT OF THE GESEM PROJECT

1.1 The need

Adalog (http://www.adalog.fr) provides a number of in-house trainings. This requires usual services, like managing the registration, maintaining the state of seminars (open, full, cancelled…). A special need is that several people, located in different offices, answer phone calls. It is important that when a client calls to register, the person who gets the call knows immediately about the state of the seminar.

The need was therefore to maintain a centralized database of seminars and registrations, which could be queried from many computers on the internal network. Given these constraints, developing the application as a web application seemed appropriate.

In addition, there are a number of things that need be done for the preparation of a session: prepare slides, coffee breaks, reserve the room, etc. It would be nice for the software to remind the persons in charge at an appropriate time.

Finally, Adalog makes on occasions bulk mailings to all the persons who attended a seminar in the past. A function to extract the information of all participants was therefore needed.

1.2 Constraints

In the past, the management of seminars was performed by a much older program, written in Dbase IV. There was a huge backlog of former attendees as ".dbf" files.

Although important for Adalog, the management of the seminars did not deserve the high cost of many commercial tools, notably DBMS. We wanted therefore the application to use free software components.

Finally, it was highly desirable for the application to be fully portable between GNU/Linux and Windows. Although the application was intended to run on a GNU/Linux machine, it was expected that the maintenance would be performed by the author; and given the availabilities of the said author, this meant that a lot of the work would be performed while on a train or being stuck in an airport due to delayed planes… i.e., it had to work on the author's laptop, which ran under Windows/XP. To be honest, it is generally a desirable goal in any application to be OS-independent.

In addition, we didn't want the application to be strongly tied to any particular DBMS. We felt that there were several sensible candidates, and that it would be preferable to be able to switch DBMS at any time, during the development process as well as later. Ideally, switching to a different DBMS should not require more than rewriting the body of one package.

2. A SHORT INTRODUCTION TO AWS

AWS [1] is an *Ada Web Server*, written by Pascal Obry. It is not actually a server in itself, but a set of packages that allow an application to act as a web (HTTP and HTTPS) server. Although it has many sophisticated features, the main principle is quite simple. We'll give here a rough overview; please refer to the extensive documentation that comes with AWS for more details.

2.1 Basic behavior

All it takes to make a web server is declare a variable of type AWS.Server.HTTP, and call the Start procedure on it:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda'03, December 7–11, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-476-2/03/0012…$5.00.

```
procedure Start
  (Web_Server : in out HTTP;
   Callback   : in     Response.Callback;
   Config     : in     AWS.Config.Object);
```

The Response.Callback type is a pointer to a callback function with the following profile:

```
function Service  (Request : in Status.Data)
  return AWS.Response.Data;
```

This function is called anytime a request is received, and returns the response. The type AWS.Status.Data includes the information from the request, like the URI, the parameters if any, etc. The type AWS.Response.Data has constructors like "Build" to build a response from a String, or "File" to return the content of a file as the response.

Note that at that point, the callback procedure may interpret the incoming URI any way it pleases, there is no connection to files, CGI scripts, or whatever. Of course, it *can* interpret the URI as a file name (as a conventional Web server would do), but this is just a special case.

2.2 Dispatchers

While the callback function allows any behaviour in response to a request, it is quite basic. AWS allows higher level binding of actions to URI through dispatchers.

A dispatcher is basically a tagged type, which includes a primitive operation with the following profile:

```
function Dispatch
  (Dispatcher : in Handler;
   Request    : in Status.Data)
return Response.Data;
```

Another Start procedure allows binding a dispatcher to an HTTP object:

```
procedure Start
  (Web_Server : in out HTTP;
   Dispatcher : in Dispatchers.Handler'Class;
   Config     : in AWS.Config.Object);
```

In this case, when a request comes in, the Dispatch function of the associated dispatcher is called.

It is of course possible to write one's own dispatcher, but to serve the most common cases, AWS offers a number of predefined dispatchers. Depending on the dispatcher, it is possible to associate treatments with URI patterns. Standard dispatchers provided with AWS include:

- *The URI dispatcher*. This dispatcher allows associating a call back function (or a dispatcher object) to URIs matching a given pattern. This is the basic dispatcher used to associate Ada functions to specific URIs. If an incoming URI matches several registered patterns, it will be dispatched to the one that registered *first*. Therefore it is possible to register a function *last* with the pattern ".*", which will serve as a default if no registered function matches the pattern.
- *The page dispatcher*. This dispatcher (actually a call-back function) looks for a file (from a definable directory) with a name matching the URI. It is possible to specify a specific page to return whenever there is no file corresponding to the URI, in order to display something more user-friendly than "ERROR 404".
- *The Method dispatcher*. This allows registering different dispatchers according to the method (i.e. GET, HEAD, POST or PUT) used by the URI.
- *The virtual host dispatcher*. This allows registering different dispatchers according to the host *name* used in the request. This is useful for making virtual hosts, where different logical servers, with different names, correspond to the same physical machine.

2.3 The templates parser

In addition, AWS comes with a *template parser*. This component takes a template file, which can include variable names, conditional parts, and tables. A *tag* associates an Ada string to a name. A "Parse" function will read a file, replace all variables with the associated values, and return the result as a String.

The template parse features conditional statements (@@IF@@ … @@ELSE@@ … @@END_IF@@) that allow including only parts of a template depending on the value of some variables. An @@INCLUDE@@ feature allows including common parts (like headers) to be shared between several pages.

Vector tags allow associating several strings to a single tag. Parts of a template can be repeated (between @@TABLE@@ and @@END_TABLE@@). When a variable name that corresponds to a vector tag appears in such a construct, the associated strings will be picked in order, one for each iteration. This is very useful for building tables.

Below is an example of (a part of) a template that displays a pull-down list, allowing the user to chose information for a given year:

```
@@INCLUDE@@ head.thtml

<form method="GET" action="chrono.btns">
 <table align=center cellpadding=4>
  <tr>
   <td>Chrono de l'année</td>
   <td align="center">
    <select name="Year" size="1">
@@TABLE@@
     <option
@@IF@@ @_YEAR_SELECT_@ = @_YEAR_SELECTED_@
      selected
@@END_IF@@
     >@_YEAR_SELECT_@
     </option>
@@END_TABLE@@
    </select>
   </td>
   <td align="center">
    <input type="submit" name="Btn"
     value="Visualiser">
   </td>
  </tr>
 </table>
</form>
```

The variable YEAR_SELECT is a vector tag containing all the years, and the variable YEAR_SELECTED is a variable containing the current year. Note that the @@TABLE@@ statement will create an <option> tag for each year, but that the "selected" keyword will be included only for the chosen year. Below is how it appears on the screen:

Note that although the template parser is very useful for building parametric web pages, there is nothing in it which is specific to HTML. It can be used to process any kind of file, and we used it for example to build mail messages.

2.4 E-mail interface

AWS provides a package for sending E-mail messages. Once again, the interface is very straightforward. You declare and initialize a server object:

```
SMTP_Server : SMTP.Receiver
 := SMTP.Client.Intitalize("smtp.hostname");
```

This object can then be used from various "Send" procedures. AWS can also send MIME attachments from disk files or base64 encoded binary data.

2.5 Other services

AWS features many other services, that we did not use for Gesem. It would exceed the scope of this paper to detail them here, please refer to [1] for details. These services include:

- Session management (using cookies).
- A web *client*, allowing to retrieve pages from other sites.
- Server push
- SOAP server and client
- LDAP support
- JABBER support

Finally, AWS includes a tool for embedding resources like templates, HTML pages, images, etc. into the executable. This makes it possible to distribute an application as a single self-contained executable.

3. GESEM'S WEB INTERFACE

3.1 Gesem Filters and dispatchers

This part describes how we took advantage of dispatchers to separate the issues of controlling access to the application, providing pages as Ada code, and providing regular pages.

All the management of the Web interface is hidden into a single package called "Engine". The interface of this package only offers high-level services, like registering a page (see 3.2 below), locking the data base, or gaining write access (see 3.3 below).

3.1.1 The access control filter

For obvious reasons, we didn't want anybody on our intranet to be able to connect to the system. Forcing a login seemed overkill (the system is not connected to the Internet, so we don't need a high level of security), and would have made the system slower to get at. Since the IP address of the client is included in the request provided by AWS, we decided to allow access to the system according to the IP address.

This was done very simply, by writing a dispatcher which checks the IP address of the client. If it is not an authorized one, it returns a minimal page stating only "you are not allowed to connect to this server". Otherwise, the request is passed to the default dispatcher. This dispatcher is therefore really a filter.

Another function performed by this filter is the ability to lock the database. This is useful when maintenance is performed directly on the data base; we don't want users to be able to modify the data at the same time. When the data base is locked, any request will return a page saying that the data base is currently unavailable. We could of course stop the server, but it is more user-friendly to return a page explaining what is happening than leaving the user with a message telling that the server was not found.

Once all validity checks have been performed, the request is passed to another dispatcher, which must register itself to the access control filter. In Gesem, we used the standard AWS URI dispatcher.

3.1.2 The URI dispatcher

All pages implemented as Ada code register themselves to the URI dispatcher.

As explained above, the URI dispatcher allows a default call-back for URIs not matching any registered pattern. We used the page dispatcher (below) as the default.

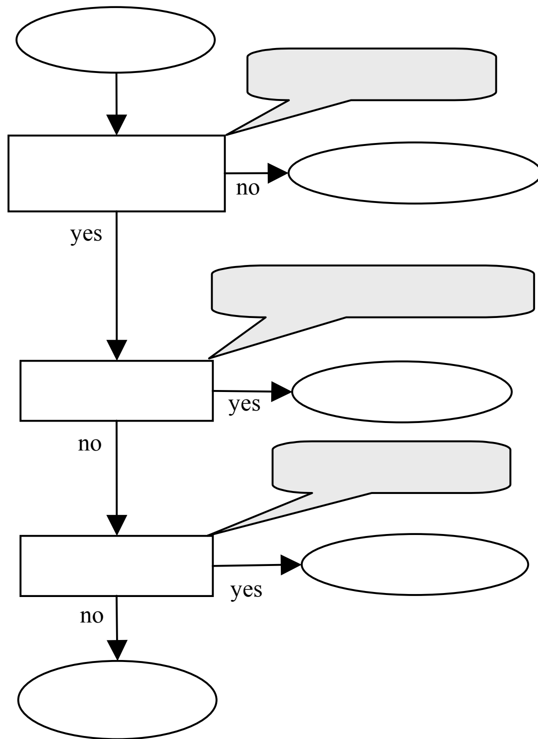
3.1.3 The page dispatcher

The page dispatcher will handle all pages not implemented as code, and therefore assumed to correspond to regular HTML files.

The default page when the URI is not recognized offers the possibility to send a mail message to the maintainer to signal the missing page (see 6.3 below, Error management). Note that this seems like overkill, since user display pages by clicking on buttons from the interface, there is therefore no reason why an unknown page would be requested. There is however no way to prevent the user from typing any inconsistent URI in the browser, so it is better to handle this case cleanly.

3.1.4 The global picture

In summary, when a request is submitted to the server, it goes through the access control filter, which checks that the client IP is authorized, and the base is not locked. It then passes the request to the URI dispatchers, which checks if the URI corresponds to a page implemented as Ada code. If not, it passes the request to the page dispatcher, which checks if the page corresponds to a regular (HTML) page. If not, our own "page not found" page is displayed. This structure is represented in the following picture:



A great benefit of this structure is that there is no difference between a page implemented as Ada code and a regular page. For example, we were able to design the home page with all buttons for expected functions. Functions that were not yet implemented simply lead to a regular page saying “Sorry, function not yet implemented”. Later, the page was implemented as an Ada function without changing the user interface.

3.2 The computed page design pattern

When we started this project, we viewed the HTML as a convenient way to provide a user interface. However, as the project matured, it became evident that designing an application for a web interface was quite different from designing a conventional GUI. The main issue is that, with a browser, the user can return to a previous page at any time with the “back” button. Unlike a regular GUI, it is not possible to guarantee that the user will follow a well defined path, or that a page will always be left by clicking either a “Cancel” or “OK” button. Another issue (discussed below) is that the user can close the browser at any time, and that the application has no way of being noticed.

As a consequence, we decided to have *no state variable in the program*. All states needed to display a page is included in the page itself (which can easily be achieved with “hidden” <input> tags). This has the benefit that we have no problem if a user bookmarks a page, and solves the re-entrancy problem by the same token: if two users are connected at the same time, we don’t need to maintain per-user information.

3.2.1 Statement of the problem

Here is a typical page (the home page for Gesem):

We see here the main elements of a computed page:

- Some fixed information (title, general layout)
- Variable information (here, the list of currently open seminars)
- Buttons

Using the template parser, it is easy to produce a page that includes variable information into a fixed template. However, buttons are an interesting problem. In HTML, a button is put into a form, and is associated to a URI that will automatically include the name of a page, plus any information in the form where the button lies. In short, from an HTML point of view, a button is linked to a *different* page, while from the programmer’s point of view, it corresponds generally to actions from the page where it resides, although sometimes a button is really a link to another page.

So, to summarize the problem, a page consists of:

- A fixed pattern
- Variable data (parameters) to be displayed
- A set of actions connected to (some) buttons in the page.

3.2.2 The Page design pattern

In order to have a uniform way of designing (computed) pages, we established the following design pattern. All pages are children of the “Pages” package. This package is empty, except for some utilities (in the private part) that are useful for building pages.

A page specification looks like this:

```
with
  AWS.Response;
package Pages.Some_Page is
  function Build (<parameters>)
    return AWS.Response.Data;
end Pages.Some_Page;
```

The package provides one or several “Build” functions that return the page built according to the parameters. For example, a page that displays information about a client will have the client handle as the parameter. Any page that needs to return a different page (as a result of clicking on a button in the page for example) does

so by returning a call to the “Build” function of the appropriate page.

The body of the package follows this general structure:

```
package body Pages.Some_Page is

  My_Name : constant String := "some_page";

  function Build (<Parameters>)
    return Response.Data is
  begin
    ...
  end Build;

  function Buttons
   (Request : in AWS.Status.Data)
  return AWS.Response.Data is
    ...
  begin
    -- Process buttons
    ...
  exception
    when Occur : others =>
      return
        Pages.Error.Build
          (Unit       => "pages." & My_Name,
           Subprogram => "Buttons",
           Occur      => Occur);
   end Buttons;

  function Page
    (Request : in AWS.Status.Data)
  return AWS.Response.Data is
    ...
  begin
    ...
  exception
    when Occur : others =>
      return
        Pages.Error.Build
          (Unit       => "pages." & My_Name,
           Subprogram => "Page",
           Occur      => Occur);
  end Page;

begin
  Engine.Register
    (My_Name, Page'Access, Buttons'Access);
end Pages.Some_Page;
```

The “Build’ function actually constructs the page from the parameters, usually by initializing tags (and vector-tags) with the appropriate information, and calling the template parser on the page template.

The “Page” function extracts the parameters from the request and calls “Build”.

The “Buttons” function extracts the button name from the parameters and calls a “Build” function, either on the same page or on some other page.

The initialization part of the package calls the “Engine.Register” procedure to register the “Page” and “Buttons” functions to the URI dispatcher with the given name, and an extension of “.html” for the “Page” function, and an extension of “.btns” for the “Buttons” function.

The benefit of this design pattern is that all actions related to a page are gathered in a single package. Since all pages follow the same scheme, it is very easy to understand what a page does, and to add new pages.

3.3 Mutual exclusion

Like any application that can be used by several persons at the same time, Gesem needs some kind of mutual exclusion mechanism. However, the constraints here are quite unusual: under normal operation, it is expected that data would be read less than once a day, and modified less than once a week. This means that conflicts would be extremely rare in practice, but the issue of conflicts must still be addressed.

Our strategy was therefore to have only mutual exclusion for modifications. In normal mode, pages display informations, but do not allow modifications. There is a “Modify” button that must be pressed when the user wants to modify the content; at that point, modification right is granted to the user (based on the IP address of the user), and any attempt by another user to press a “Modify” button (even for unrelated data) will lead to a page saying “Sorry, the database is being modified, try again later”.

When a page is in “modification” mode, it shows “Validate” and “Cancel” buttons that will perform commit or rollback on the database, and release the lock.

This mechanism has an interesting property in relation to the web interface. We have no way of preventing a user from closing the web browser while he/she is holding the lock. In that case, the lock will stay associated with the user’s IP address. If the same user returns later to the same page, he/she can validate the changes and unlock the database.

As a safety feature, the control panel displays who is the owner of the lock (if any). If the database appears to be locked, it is easy to give a phone call to the person and kindly remind him/her that he/she should not leave an unfinished modification…

3.4 The e-mail interface

We took advantage of the mailing facilities offered by AWS to provide automatic mailing triggered by events or time. The recipients of the mails are stored in the database, and there is an administrative page in the program that allows to change them easily.

3.4.1 State triggered mail

Some mails are sent as a result of changes in the state of the application. For example, when a new registration is entered, or a seminar is cancelled, all interested parties are notified.

These notifications are hard-coded in the program; however the *text* of the mail is not. It is actually a template, and the result of parsing it with the template parser constitutes the body of the mail. It is therefore easy to change the content of the message, without modifying the application.

3.4.2 Time triggered mail

Other mails are to be sent a few days before a seminar starts, for example to remind the secretary to reserve the room, the presenter to prepare the slides, etc.

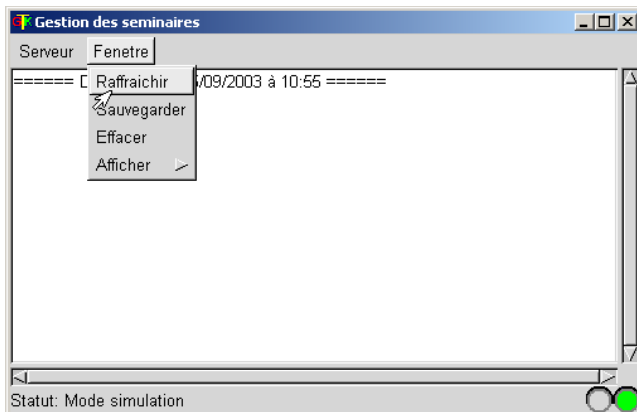
This was very easy to do, by having a local task in our mailer package that wakes up once a day, and checks if any mail is to be sent. As before, the text of the message is built by parsing a template.

4. THE LOCAL INTERFACE

Since an AWS application is a stand-alone server, it can have a local interface on the computer it runs on, while offering a web interface to the outside world.

We took advantage of this possibility to make a control-panel on the server (see below). This control panel is displayed as long as the application is running, and allows controlling the server and tracing the requests.

The interface features a trace window, which displays all requests



as they are processed. At the same time, it displays who is locking the database, and a red/green light that is red when there are any uncommitted transactions in the database. It is therefore easy to monitor what's going on, and to take appropriate actions if necessary.

The Server ("Serveur") menu in the interface allows to stop the server, lock the database, reinitialize the database (to make sure everything is consistent after a direct maintenance action on the database, f.e.). The Window ("Fenêtre") menu allows refreshing the interface (we had some problems with refreshing under Windows), clearing the trace window, or saving the trace window into a file. This latter function is very useful if any problem happens: the trace window shows the precise path followed by the client that led to the problem, as well as information about the problem itself. In the running application, we can thus save the context of the problem to a file, restart the server cleanly, and then start working on the problem.

The local interface was developed with Glade/GtkAda.

4.1 Gtk and Glade

Gtk [2] is a library for building user interfaces. It has implementations for both GNU/Linux and Windows, and an Ada binding, written by Emmanuel Briot, Joel Brobecker, Arnaud Charlet, and Nicolas Setton, developed and maintained by ACT.

Glade is a GUI design tool that allows easy visual design of user interfaces. It allows automatic generation of interfaces in many languages, including Ada.

In the case of the present project, we knew nothing beforehand about using Gtk. We simply looked at the code generated by Glade, and it was very easy to add the necessary processing in the generated templates. Glade also features a good round-trip engineering facility, which means that changes made into the generated code are automatically kept when a modification is made to the interface, and the code is generated again.

4.2 Issues with tasking

The only difficult issue we encountered with using Gtk was related to tasking. AWS uses different tasks to manage simultaneous connections. Since the interface traces the name of pages as they are displayed, it was necessary for several tasks to send messages to the interface at the same time. Unfortunately, since Gtk has been developed in C, it is not multi-tasking friendly. Although the Ada binding provides a locking mechanism, it only allows one task to manage windows, and prevents any other task from interacting with the GUI. This was a big problem for this application, since the tracing mechanism meant that many tasks needed to add text to the trace window, for example.

We decided to adopt the classical structure: the main program would call the GUI's main loop, and exiting this main loop would terminate the program. When other tasks need to update the display, they drop a message into a mail box and then schedule an idle-action. An idle-action is a call-back function which is automatically invoked when the GUI is idle. Of course, this function is invoked by the task in charge of the GUI, i.e. the one that holds the lock on the GUI! In our case, the idle-action will pick up the message from the mail box and update the GUI accordingly. This way, there is effectively only one task interacting with GTK..

5. THE DATABASE INTERFACE

5.1 Choosing a database

In addition to our constraint of having a free DBMS, we wanted it to be reliable; although our application is not very demanding as far as efficiency is concerned, reliability was important, since losing data about people registered to a seminar would put us in a bad position. Possible candidates included PostgresSQL and mySQL, but PostgresSQL is not available under Windows, so eventually we chose mySQL.

As a side note, a nice feature of mySql is that it provides a utility called "mysqldump" which dumps a whole database as SQL statements. It is therefore extremely easy to move data to another database, should we ever choose to do so.

5.2 Choosing an interface

We had several options for interfacing to mySQL from Ada:

- Use GNADE [3] with the native interface to mySQL;
- Use ODBC and GNADE with the ODBC interface;
- Use ODBC and a direct binding to it.

We rapidly decided to use ODBC, because it would ensure that our application was not dependent on any special feature of mySQL, and would allow us to change the DBMS without changing the application. mySQL provides ODBC drivers for both Windows and GNU/Linux. Moreover, with ODBC, other applications (notably Excel) can access the database, making it easier when we need to correct some inconsistencies in the database manually.

Our first move was to use GNADE, written by Micahel Erdman. GNADE is an implementation of the ISO SQL standard [4]. It uses the so-called "embedded SQL" approach, where SQL statements are mixed with Ada code. A pre-processor reads the

mixed code and transforms all embedded SQL statements into calls to the DBMS binding. A typical example of embedded SQL looks like this (example borrowed from [3]):

```
Delare_ERROR : exception;
Open_ERROR   : exception;

EXEC SQL DECLARE BEGIN
  empno : SQL_Standard.INTEGER;
  ...
EXEC SQL END-EXEC

EXEC SQL
  WHENEVER SQLERROR raise Declare_Error;

EXEC SQL
  DECLARE emp_cursor CURSOR FOR
    SELECT empno, name INTO :empno, :name
      FROM employees
      WHERE manager = :to_find;

EXEC SQL
  WHENEVER SQLERROR raise Open_ERROR;

EXEC SQL OPEN emp_cursor;

loop
  EXEC SQL FETCH emp_cursor;

  exit when SQLCODE in
     SQL_STANDARD.SQL_ERROR;
  Put_Line(name);
end loop;
```

However, we had to move back for several reasons:

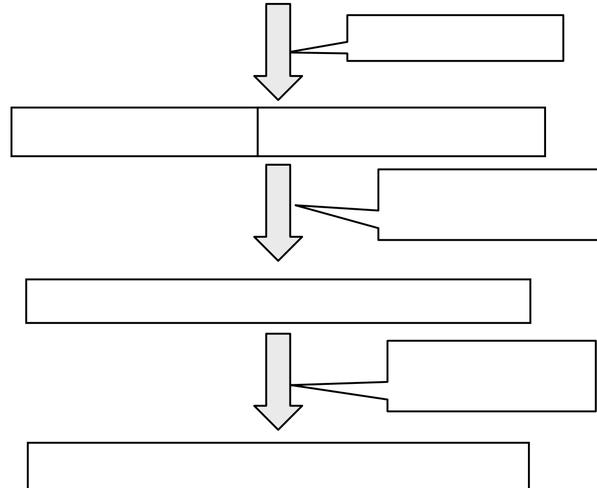
- GNADE is not yet a totally mature product, and we experienced some difficulties with it.
- The preprocessor idea is not very popular in the Ada world, and complicates the building process, requiring makefiles, etc.
- Having a foreign embedded syntax in Ada code drives Ada tools, and especially the Ada mode of emacs, completely crazy. We lose automatic indentation, syntax highlighting, etc. In practice, this was felt to be a high nuisance.

We then tried to use the binding to ODBC which is provided with GNADE without the preprocessor, i.e. we wrote Ada code directly that was roughly what the preprocessor would have produced. However, the interface was really designed for use with the preprocessor, and was not very convenient to use directly. Moreover, it meant that we had to drag in the whole GNADE library, while we were really only using a small part of it.

At that point, we discovered the ODBC binding [5] written by Sune Falck. This binding consists of a very thin layer (Iodbc) that simply interfaces to C, and a thicker layer (Odbc2) which is more Ada oriented. It was much simpler to use, but suffered from being still quite low level in some places (especially as it uses addresses to pass back query results). It was however relatively easy to add another layer on top of it to hide the ugly details. This package (DB_Interface) manages the connection, and processes SQL requests without dependences to ODBC itself.

The DB_Interface itself provides operations for requests that simply return a single string. It has a generic child (DB_Interface.Cursor) for requests that return tables. This child is instantiated with a string containing the SQL request, and provides iterator operations over the resulting table, with access functions to return the values of the various columns.

The final structure of the interface is described in the figure below:



5.3 Implementing objects over the DBMS

We did not want the core of the application to deal directly with SQL, since having SQL statements spread all over the place would have made the application much more difficult to maintain. In general, it is better to design a structure such that programmers who are not fluent in SQL can add new features to the application.

Each object of the application (like clients, seminars, orders, etc.) is implemented as a package which is a child of the “Objects” package. Each object provides functionalities dealing with the object, and only this object, and translates the requests into SQL queries.

Each instance of an object is identified by a unique handle (actually a primary key in the data base). Operations dealing with persistence, like creating a new object, setting its value, etc. share a lot of commonality. We designed a generic package (“Data_Manager”) that automatically provides these operations. The specification (slightly simplified) of this package is as follows:

```
with Globals, Objects;
use Globals;
pragma Elaborate (Objects);

generic
   type Data is private;
   Data_Name : String;
   Columns   : String;

   with function Image (Item : Data)
      return Array_Of_Unbounded;
   with function Value (Item : Array_Of_Unbounded)
      return Data;
package Data_Manager is
   pragma Elaborate_Body;

   Max_Columns : constant Positive := 20;
   function  Uncommitted return Boolean;
```

```
   type Handle is private;
   Null_Handle : constant Handle;
   function To_String (Source : Handle)
      return String;
   function To_Handle (Source : String)
      return Handle;
   Bad_Handle : exception;

   procedure Set (The_Handle : Handle;
                  Item       : Data);
   function  Get (The_Handle : Handle) return
Data;
   function  Create (Item : Data) return Handle;
   procedure Delete (The_Handle : Handle);

private
   …
end Data_Manager;
```

The generic parameters are the type of data to store in the database, a string identifying the column names in the database, and two functions allowing to convert the data to and from arrays of unbounded strings. At the database level, all data are managed as strings, and this allows reconstructing the proper data from its string representation.

Although all objects are managed through their handles, note that the package provides "To_String" and "To_Handle" functions to convert handles to and from a string representation. This is very important in our case, because it allows passing handles to (and getting handles from) web pages as parameters.

An object package instantiates the data manager, and only operations that are special to an object need to be written when a new object is created. A typical object uses the following design pattern:

```
with Globals, Data_Manager, AWS.Templates;
use Globals;
pragma Elaborate (Data_Manager);
package Objects.Abstraction is

   type Data is
      record
         …
      end record;

   function Image (Item : Data)
      return Array_Of_Unbounded;
   function Value (Item : Array_Of_Unbounded)
      return Data;
   package Manager is new Data_Manager
     (Data      => Data,
      Data_Name => "my_data",
      Columns   => "col1, col2, col3");

   subtype Handle is Manager.Handle;

   type List is array (Positive range <>)
      of Handle;

   function Associations (Item : Handle)
      return AWS.Templates.Translate_Table;
   function Associations (Item : List)
      return AWS.Templates.Translate_Table;
   function Extract (Param : AWS.Parameters.List)
      return Data;

 -- Other operations on Abstraction.Data

end Objects.Abstraction;
```

In addition to the database (persistence) operations provided by the data manager, each object features "Associations" operations that associate template tags with the fields of the data, or vector tags in the case of a list of data, and conversely an "Extract" function that reconstructs the data from the parameters of a web page.

In a sense, this structure allows easy conversions between three representations of the data:

- As an Ada record, for computing
- As tags, for displaying in web pages
- As a database table.

This principle is nice, but does not allow covering all the needs, since sometimes we need information involving more than one object. For example, we needed the list of all clients who attended a seminar or asked for a documentation.. Is it a property of the seminar, the documentations, or the client? Putting such a function in either object would have created dependencies across objects, something that we wanted to avoid, except when it was structurally required (an order refers to the ordering client for example, therefore there must be a dependence from "Objects.Order" to "Objects.Client"). All requests that are not logically part of an object are gathered in another package, called "Queries". This way, we keep cross-dependencies to a minimum, and we guarantee that all SQL statements are within the "Objects" hierarchy plus the "Queries" package, and nowhere else.

Another issue was raised when we felt the need to have operations that affected all objects. For example, we needed to have some checks performed on objects whenever we had a commit or rollback operation on the database (for example, to check if some objects had uncommitted changes).

The natural place for such general procedures is in the "Objects" package itself. The body of the package would then have to **with** all its children (all the objects) in order to call the corresponding procedures. But this would mean that each time a new object is added to the system, the body of the "Objects" package needs to be modified. To avoid this, we used the ambassador paradigm (described in [6]) from the ESCADRE project [7]. The "Objects" package defines an abstract "ambassador" tagged type:

```
type Ambassador is abstract tagged null record;

function Is_Committed (Item : Ambassador)
  return Boolean is abstract;
procedure On_Commit (Item : Ambassador)
  is abstract;
procedure On_Roll_Back (Item : Ambassador)
  is abstract;

procedure Register (Item : Ambassador'Class);
```

This type has a primitive operation for each of the services that are to be dispatched to all objects, plus a primitive to register any object belonging to the class of Ambassadors. Each object package will (locally) derive a type from Ambassador, and create one instance of it which will be registered (by elaboration code of

the package) to the “Objects” package. The “Objects” package just maintains a list of ambassadors, and when a global request is made, it just calls the corresponding primitive operation on all ambassadors. In practice, this is done by the “Data_Manager” generic package, so we don’t really have to bother about it when we design new objects.

5.4 Other issues

When the program starts, it opens a connection to the database, and keeps the connection open all the time. It would seem more logical to open the connection when a user comes in, and to close it when the user leaves; however, this is incompatible with a web interface, because we never know that a user has left. Moreover, the user can bookmark pages, or use the “Back” button on the browser, allowing to access data in a very unstructured way. This resulted in making it very difficult to decide when to open or close the connection. By having the connection always open, we avoid the problem.

But we discovered a quite unexpected issue, due to the very low service rate of the server. If a database connection stays open for several hours without any activity, the database server thinks that the client has died, and closes the connection. It is therefore necessary to “ping” the database every now and then to keep the connection alive.

This was easily done by introducing a task in the high level database interface (package “DB_Interface”) that wakes up at 4 hours period, and sends a dummy request to the database.

6. OTHER FUNCTIONALITIES

6.1 Bulk mailing

Bulk mailing consists basically in an extraction of clients that match certain criteria: sometimes we want to mail all our clients, sometimes the continuous education services only. It is therefore typically a (quite sophisticated) SQL query.

The result of the query is simply put into a file in CSV (comma separated values) format, which is a text format that can be read by any spreadsheet or database application that we know of. It is therefore both easy to produce (we just need Text_IO) and easy to manipulate.

6.2 Logging

Finally, we have a logging package that registers into a simple text file the main events in the database, like creating a seminar, registering new clients, etc. Since AWS provides us with the IP address of the client, we have a table mapping this IP address to actual persons, and we are able to log the event together with the time and the person who was responsible for it.

The log can be viewed using a regular page. It is therefore easy to trace who changed some important state in the database, and when.

6.3 Error management

We wanted the program to be very fault tolerant, since it runs on our computer, and nobody can reset it while we are on vacation (or presenting a paper at a SIGAda conference!). We therefore established the following error policy:

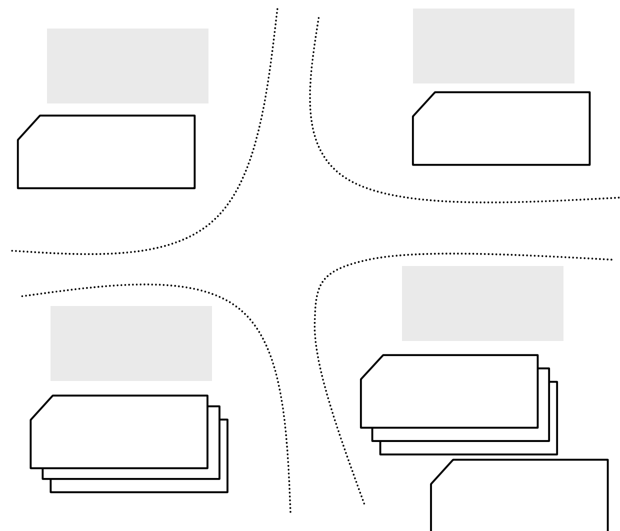
- When an error is detected in any subprogram except a page, it traces the cause of the problem to the local interface, and raises (or propagates) an exception. If the detected error was not already an exception, an exception message is added to identify the context of the problem. Since all subprograms are initially called from pages, the exception will eventually be propagated to a page subprogram.
- All pages have a catch-all exception handler, which returns (remember that pages are functions) a call to “Pages.Error.Build”, i.e. they display an error page. The parameters of “Pages.Error.Build” include an “Exception_Occurrence” parameter.
- The error page rolls back the database (to return to a safe state), and displays a page explaining that a problem has occurred. This page has a text entry where the user can add more information, and a “Send” button that sends a mail to the maintainer of the program, with all the information about the error (including the context from the exception occurrence and comments by the user) .

Using this strategy, no information about the problem is lost, while allowing the application to return to a safe state and continue.

7. LESSONS LEARNED

7.1 On the general structure of the application

The whole application, excluding AWS and the ready-to-use software components (but including the code generated by Glade) is about 10_000 raw lines (4460 semi-colons). It is nicely partitioned into subsystems, as pictured in the following figure:



We have completely separated the pages (which deal only with high level objects), the DBMS interface which is completely hidden below the objects layer, the local user interface, and the management of the HTTP link. This structure proved very effective, since adding functionalities or modifying the application are straightforward and follow well defined and unified design patterns.

7.2 On portability

All of AWS, GTK/Glade, MySQL and ODBC are available on both GNU/Linux and Windows. None of these software components showed differences between the two systems.

The net effect is that exactly the same code is used on both operating systems. In the first times, we used to test the program on both systems; we don't do that any more. Whenever a modification or improvement is needed, we usually develop and test it on a Windows laptop, then copy the modified sources to the GNU/Linux desktop, recompile and bring directly into service without any further testing.

Many people would find this dangerous; and to be honest, this is not a life-critical application! But the point is that in our experience, we *never* exhibited a difference in behaviour, including when we were chasing bugs: bugs behaved exactly the same on both systems.

To state it shortly: this application is 100% portable, full stop. It owes a lot to Ada and the nice software components we used. This does not mean that all Ada programs are portable, but it is a proof by example that is *possible* to have 100% portability with Ada, even for such an application that would look at first as very OS dependent.

7.3 Exceptions and reliability

We used exceptions extensively in this program. Many assumptions are checked, and raise exceptions when they failed. Carefully placed exception handlers ensure that errors are caught at an appropriate place, and allow the system to return to a consistent state.

The net result is that the server never stops, even in the face of programming errors. This was a main goal of the project, since the server must run permanently.

We wanted to stress this achievement here, because many people are afraid of exceptions. Of course, this program does not need the high level of confidence of life-critical software, but we claim that obtaining 100% availability would have been very difficult to achieve without the nice exception mechanism provided by Ada

8. USING APACHE OR A DEDICATED WEB SERVER?

At this point, it may be worth discussing, in the light of this experiment, the benefits and drawbacks of developing an independent web server, as allowed by AWS, versus the more conventional approach of using CGI, scripts, and a general Web server like Apache.

8.1 Ease of changes

Apache is a dedicated, general-purpose, web server. The application itself is made of pages and scripts that can easily be changed without affecting the server. When you update some script, it has no effect on other pages served by the same server.

With AWS, the whole application is a program. If the application changes, the whole program has to be recompiled. To activate the new version, it is necessary to stop and restart the server, thus creating a short time of unavailability. Note that AWS provides services for storing states between runs, so that any user connected when the server is stopped can restart at the same point, therefore mitigating this drawback.

If a single AWS server is used for several applications, for example using the virtual hosting facility (see 2.2 above), then all applications will experience a short breakdown while the server is restarted.

Note however that this does not apply if you just need to change the *aspect* of a page, but not the *content*. In this case, you update a *template*, which is a file external to the server.

8.2 Concurrency

Concurrency, and especially mutual exclusion, is an important feature of a web application. With a regular server, each request runs as a separate process. Mutual exclusion is generally dealt with by having heavy-weight mechanisms, like lock files.

Since an AWS application is a single executable, it can have global locks implemented by protected types. It is therefore easy to have a light-weight mutual exclusion mechanism.

Although we didn't have this need in our application, it is also much easier to make several users communicate through the usual tools of Ada tasking.

Finally, within a single application, it is very easy to have periodic events (like the automatic mailer), with a simple task. This would require cron or similar system level application with an Apache application, and would not be integrated in the application itself.

8.3 Half-web application

In the case of Gesem, the web interface is the main goal of the application, and we took advantage of the "single program" approach to feature a local panel displaying trace information and managing the application as a whole.

But there is another class of applications, where the local interface would be the main goal, while still providing the possibility to interact with it remotely through the web. Imagine for example an experiment, where the computer manages various devices; with AWS, it can be designed as a regular application, while allowing the experiment to be managed remotely through a web interface. This would be very difficult to do with an Apache based interface.

8.4 Installation

We are discussing here the case of a Web application intended to be widely available, like START [8].

As Apache is a very general server, a lot of things can be parameterized. Having the parameters right for Apache usually requires extensive knowledge of how it works, plus administrator rights. Note that defaults parameters for Apache under GNU/Linux may vary according to the distribution, therefore making difficult to have applications that run correctly in all contexts.

On the other hand, an AWS application is simply a regular executable, with a set of data files that can even be integrated in the executable itself. The executable can run with normal user's privileges, so the application can run out-of-the-box (or out-of-the-download !).

9. CONCLUSION

In this paper, we have discussed the design choices and showed the design patterns that were used in developing our Gesem application. Although Gesem is not a particularly demanding application, especially as the workload is concerned, it is quite typical of any application that must simultaneously provide a local and a remote user interface, while manipulating data from a database.

Of course, the final structure emerged from a trial-and-error process. But it is now well established, and can be reused for any project of the same type. With the experience acquired, Adalog is now ready to develop web application for its clients in a very cost-effective manner.

As a side effect of this development, several reusable components were designed, like the abstract interface over ODBC, or the data manager that adds persistence to objects. It is our intent that these components will eventually be made available from Adalog's free components page (http://www.adalog.fr/compo1.htm).

10. ACKNOWLEDGMENTS

Much of the development work of Gesem was performed by two Adalog trainees, Jérôme Burlando and Bertrand Carlier, whose work is gratefully acknowledged.

Thanks to Pascal Obry for useful comments on an earlier version of this paper.

11. REFERENCES

[1] AWS. http://libre.act-europe.fr/aws/

[2] Gtk: http://www.gtk.org

[3] GNADE. http://gnade.sourceforge.net/

[4] SQL. ISO/IEC 9075 :1999

[5] AdaODBC. http://www.adapower.com/reuse/iodbc.html

[6] Manuel du concepteur ESCADRE v5.0. http://escadre.cad.etca.fr/ESCADRE/v5.0/documentation/s_design50/s_design50.pdf (in French)

[7] ESCADRE. http://escadre.cad.etca.fr (in French)

[8] START, Submission Tracking And Review Toolset, http://www.softconf.com/START.