Sometimes older languages can yield hidden treasures. Guest columnist **Peter Ward** revisits Lisp and finds it invaluable for communicating with a server running Corba IDL

Life in the old dog yet

Peter Ward has been developing powerful IT systems for complex business needs for over 20 years. As a consultant for Pan Domain (**www.pandomain.co.uk**), he provides IT consultancy to large companies with challenging integration problems. Peter may be contacted at **peter-ward@bcs.org.uk**

HERE'S THE SCENARIO: ONE APPLICATION designed for client-server operation, a large repository of documents accessible by that application and an even larger audience for those documents. The audience is widely distributed, sometimes mobile, and doesn't have the luxury of a managed computing environment.

In other words, we need to provide web access from browser-equipped devices to our application server. The client-server protocol is defined in OMG IDL and the server is Corba-enabled. How do we approach this task and what tools should we use?

"Java!", "Perl!" I hear you shout. Hold on! I'm looking for the most productive environment around. To develop this code will require some exploration, to make sure that access to the server is working properly, ideally interactively. I'd like a simple solution, and any solution involving multiple technologies is not simple and likely to cause a problem at some point in the project. Also, I want to be sure that the HTML I generate is neat and well formed.

Rediscovering Lisp

I chose Common Lisp for this exercise. Many of you may never have heard of Lisp and some of you will have vague recollections of an obscure language suitable only for academic problems involving artificial intelligence. That's unfair. Yes, the roots of Common Lisp are more than 40 years old. Yes, it was and still is used for artificial intelligence work. But it's matured into a powerful and flexible tool suitable for simple and complex problems. It has a full-blown object system with the ANSI standard X3J13.

I've been using Franz Inc's Allegro Common Lisp (ACL) on Windows NT and Linux. The Linux environment is pure command line and extremely fast, but the Windows version offers the luxury of a modern integrated development environment (IDE) with debugger, inspector and GUI tools. Dedicated Unix hackers can achieve the same effects with emacs. The IDE didn't play a role in this project so I won't describe it here.

There are two ways to operate ACL with Corba servers. Our first version used Inter-Language Unification (ILU) from Xerox's Palo Alto Research Centre. It's a Corba-like system available with many language bindings, including Java, C++, Python and Common Lisp. It's free for all uses and is effectively supported by a mailing list. Installation on Windows and Linux was straightforward, as was generation of the Corba stubs using the supplied stub generator. While this was effective, we'd prefer a single-technology solution, which is why we turned to Franz Inc's Orblink add-on. This is a Corba 2.2 ORB implemented in Common Lisp. It isn't cheap but, as we shall see, integration is a snap.

In practice:

```
>(require :orblink)
```

This causes the Lisp environment to load the optional Orblink package and start an Orb with default parameters from the configuration file.

Load the IDL:

```
>(corba:idl "myinterface.idl")
```

No separate generation of stubs is necessary. This command parses the IDL and automatically defines all the necessary namespaces, classes and methods according to some straightforward rules that conform to OMG standards.

Stringing Lisp along

As with any Corba system, we have to acquire a reference to an object on the server to get started. In many installations, a naming service is used (somewhat like JNDI), but in this example we follow the common practice of publishing the reference in string form, using a URL. We grab the string reference from the web server and convert into a proper object reference (see Listing 1).

This has saved the local proxy in the variable 'server'. We can learn a little more by using describe (see Listing 2).

One method in the IDL allows us to log in to the server. The IDL is:

```
interface CoordinatorInterface {
  string login(
    in string username,
```

FACTS AT A GLANCE

- Common Lisp has matured into a powerful and flexible tool.
- Corba integration is remarkably painless.
- Lisp dynamically generates correctly formed HTML.
- A single-process model and native code outperform more complex solutions.
- Incremental development is fast and robust.

```
        in string password
    );
};
```

When this is mapped into Lisp, the method is named `op:login`. The first parameter is always the object (local proxy), followed by the other parameters.

```
>(op:login *server* "pjw" "password")
"IOR:000000000000003b494...000007"
```

This method merely returns a unique reference to a server object that represents our session. That's just how this server happens to work. Now, we can translate the string to create a local proxy and do something more interesting, such as retrieve a set of documents from the server. You can find the IDL in Listing 3.

Document retrieval

Now we can invoke one of the methods on this object, for example to retrieve a list of documents (see Listing 4), giving us a set of document details hot from the server. We need to present them in HTML through a web server. No need to leave Lisp at this point: there are at least two web servers written completely in Common Lisp. We must mention the excellent and extremely powerful cl-http from John Mallery and friends at the MIT AI labs. But for this project we are using the open sourced AllegroServe, which provides sufficient power for our purpose and also integrates well with ACL.

HTML generation with this web server is completely achieved within the Lisp language, using a set of macros that effectively extend the language to encompass HTML markup. For example, to display the details of one document using simple paragraphs within a table cell, we code:

```
(defun display-document (doc)
  (html (:tr
        (:td (title doc))
        (:td (author doc)))))
```

Tags use the macro form (:tag body). This causes the leading <tag> to be generated, the body of the macro is executed and then the trailing </tag> is generated. Hence, malformed HTML requires devious and perverse acts. Let's check the code that is generated:

```
>(display-document (first *doclist*))
"<tr><td>My document</td><td>Pete Ward</td></tr>"
```

We can now generate a containing table and apply this to all documents.

```
>(defun display-list (docs)
  (html
   (:table
     (map 'list #'display-document docs))))
```

That's a nice example of the expressive power of Lisp, taking a function that works for one instance and easily applying it to a collection using one call. Many Lisp programmers work this way, creating small functions that can be proven to work and then building up more complex blocks. Optimisation is performed later, by adding type declarations, reducing unnecessary memory allocation and profiling the code. This incremental and highly interactive approach is one reason why the environment is many times more productive than C++ or Java.

To make this content available to web browsers, we have to publish a URL. This associates the URL with a Lisp function, so that the function `doc-lister` will respond when the URL `/docs` is accessed (Listing 5).

Jazzing things up

This is fine so far, but the presentation is somewhat bland. How can we take an HTML design template and blend in this dynamic content? This area still requires some work, but for the time being the approach is to take the web page design from any reasonable HTML design tool and run it through the HTML parser provided. This converts the HTML markup style into the Lisp markup style we have seen already. It's then straightforward to embed the dynamic calls.

Speed is always a concern when serving dynamically generated HTML. This solution has several points in its favour. The implementation is faithful to the layered architecture, but has allowed the middlemost four layers to be implemented in a single technology and single process space. The code is fully compiled and can reasonably be expected to outperform the equivalent Java. In the laboratory environment, with no optimisation, the code adds a perceptible but small overhead to the server calls. Although there is no built-in load balancing within the Lisp environment, this multi-tiered solution is readily scalable.

We are now awaiting the arrival of ACL version 6, which adds what is now considered essential for any serious integration project: secure sockets, XML extensions and tight integration with Java classes. Don't write off this old-timer, it's the Charlton Heston of programming languages – getting on in years but still capable of delivering the goods. ■

Links

- Franz Inc: www.franz.com
- cl-http: www.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html
- AllegroServe: http://allegroserve.sourceforge.net/
- Lisp as an Alternative to Java [Erann Gat]: www-aig.jpl.nasa.gov/public/home/gat/lisp-study.html
- ILU: ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

Listing 1: Converting a string reference to an object reference

```
>(do-http-request "http://www.myhost/myior.txt")
"IOR:000000000000003...0000007"
>(op:string_to_object corba:orb *)
#<COM/PANDOMAIN/COMMON/CORBAGEN:COORDINATORINTERFACE-PROXY
 "´¨´0_RootPOA
>(setf *server* *)
```

Listing 2: Checking the object reference using the describe function

```
>(describe *server*)
#<COM/PANDOMAIN/COMMON/CORBAGEN:COORDINATORINTERFACE-PROXY
 "´¨´0_RootPOA
   is an instance of
   #<STANDARD-CLASS COM/PANDOMAIN/COMMON/CORBAGEN:COORDINATORINTERFACE-PROXY>:
 The following slots have :INSTANCE allocation:
  JUNCTION        #<ORBLINK:CLIENTJUNCTION: [212.67.199.81:7960] Idle: 180929 sec socket: NIL
                  @ #x20cf3e9a>
  HOST             "212.67.199.81"
  PORT             7960
  IOR             #<ORBLINK::GIOP.IOP-IOR @ #x20dc86ba>
  OBJECT_KEY      "´¨´0_RootPOA
  ORIGINAL-IOR    NIL
  REPOSITORY_ID   ORBLINK.REPOSITORY_ID:|IDL:com/pandomain/common/corbagen/CoordinatorInterface:1.0|
```

Listing 3: Creating a local proxy

```
// idl2java idl2package ::corbaGen com.pandomain.engine.corbaGen
module com {
   module pandomain {
      module common {
         module corbagen {

struct DocumentArray
{
   llong_unbound_seq docIds;
   string_unbound_seq docTitles;
};
...
interface ConcurrencyInterface {

long listDocuments(out DocumentArray ownedArray);
...
};

interface CoordinatorInterface {
   string login(
      in string username,
      in octet_unbound_seq password
   );
};

         };
      };
   };
};
>(op:string_to_object corba:orb *)
#<COM/PANDOMAIN/COMMON/CORBAGEN:CONCURRENCYINTERFACE-PROXY
 "´¨´1972059480
>(setf *engine* *)
```

Listing 4: Retrieving a list of documents

```
>(op:listdocuments *engine*)
#<COM/PANDOMAIN/COMMON/CORBAGEN:DOCUMENTARRAY
 :DOCIDS #(13 18 12 0)
 :DOCTITLES #("Development Team Progress "
                "API Manual" "Test whiteboard"
                "Development Plan")
   @ #x21075022>
```

Listing 5: Associating a URL with a Lisp function

```
 ;; Make one link from id and title pair.
 (defun make-document-link (id title)
   (html
    (:h3
     ((:a href (concatenate 'string "open?docid=" (princ-to-string id)))
      (:princ title)))
    (:hr)))

  (defun list-docs-as-links (doc-array)
   (map 'list #'make-document-link
     (op:docids doc-array)
     (op:doctitles doc-array)))

 (defun doc-lister (req ent)
   (let* ((my-engine (which-engine req))
          (docs (op:listdocuments my-engine)))
 (with-http-response (req ent)
       (with-http-body (req ent)
         (html
          (:h2 "Your documents")
          (list-docs-as-links owned-docs)
          (signature req ent))))))

 (publish :path "/docs" :content-type "text/html"
          :function #'doc-lister)
```