Why JAR?

ANTWERP 94

In 1993-1994 I wrote a `J/DDE/FoxPro` based database system to store `J` words. This system turned out so well I gave a poster about it at *APL94* in Antwerp Belgium.

WHY DID I DO THIS?

J System Issues

In 1994 the J programming environment was very simple. It was very clear to me that if you wanted to use J for serious applications the native J environment would need to be supplemented. For example:

1. Prior to J2 the system didn't even have an embedded editor.
2. There was no blob support. `jfiles` and `kfiles` didn't exist.
3. Symbol table size was very limited and J tended to crash when you had to many objects in the workspace, i.e.: there was a strong incentive to load only what you needed and nothing else.

Code & Configuration Management Issues

From years of *APL* programming and working with other languages and tools I knew that maintaining one, and only one, source for a given code unit *pays huge dividends over time.*

How huge? Just ask the companies paying the bills on all these Y2K and Euro projects. If all this Y2K gunk had been developed using a dictionary based approach. Fixing Y2K problems would consist of roughly these steps:

1. lookup relevant code, data, documentation, test cases, and build procedures in the master database.
2. make necessary changes to code, data, ...
3. generate *N* new systems from master database
4. run regression tests from master database
5. generate new system documentation from database
6. release Y2K *patch*.

Instead of a massive global time and money consuming mess Y2K shrinks to the *trivial* problem *it should be*!

So I put together JDict (JAR's ancestor). The JDict FoxPro database server has proven very stable. After 1995 I made no changes to the server. I did make minor changes to J client software to accommodate changes to J .

I have tried on at least three occasions to replace JDict with an ODBC or Delphi OLE-Automation based system. My *APL97* presentation demonstrated the results of one such effort. With ODBC and Automation I could:

1. never match, or even come close to, native FoxPro performance.

With each hardware advance JDict has become more useful. The DDE system would run quite nicely on a 33MHZ 386 with 8 megabytes of memory and still execute common tasks in less than 2 seconds. On current 166 to 300 MHz machines typical operations take a good 10 to 20 milliseconds.

Performance is not why I'm building JAR.

WHY AM I BUILDING JAR?

J Has Changed - A Lot!

The J programming environment and system have greatly improved. All the limitations that guided my thinking when I built JDict have vanished. J now has an integrated development environment, supports large symbol tables, (2048 names per locale), no longer crashes when hundreds, or even thousands of names are in the workspace, supports blobs, is *much faster* and has all the necessary Windows GUI tools needed to build a decent dictionary browser.

The introduction of locale paths, numbered locales, (objects), and indirect locale names invalidates many of the assumptions I made about how J systems would be structured.

Finally, J file extensions have changed to avoid clashes with Javascript. The old JDict system embeds these file extensions in #DEFINE's. Changing the extension requires a recompilation of the FoxPro server application. I don't want to open up that Pandora's box.

The Software World has changed.

Software is always changing—*sometimes for good reasons*. Since I finished JDict all of the software JDict uses has changed.

1. FoxPro has moved through three major releases since 1994. I doubt, given Microsoft's sterling compatibility record, that the JDict server would even run on later versions of FoxPro and frankly I have no interest in finding out.
2. JDict generates structured documentation in an old LaTeX 2.09 format. LaTeX2e is now the standard. I would have to rewrite the FoxPro LaTeX code generator, which I always considered the ugliest code in the system, to handle LaTeX2e.
3. The web has exploded on to the scene. This makes HTML and XML support a necessity.

4. System interprocess interfaces have mutated. DDE was always to primitive to be taken seriously and even Microsoft has abandoned it for other methods. J can be turned into a very nice automation server but it cannot, as of release 4.0x, function as an automation client. This time around I'm using native J methods and avoiding all this interprocess nonsense.

I Made some JDict design Mistakes

By using `JDict` I have discovered some flaws in the system's design that need fixing.

1. The `LaTex` document generator was built before the

   ```
   n : 0
     ...
   )
   ```

   Multiline J format stabilized. It needs to be modified to display user defined ranks on explicit functions, e.g.

   ```
   foo =: 3 : 0 "0 1
      ...
    )
   ```

2. Test cases are second class citizens. **One of the fundamental lessons I learned from `JDict` is that test cases are more important than the objects they test.** The original `JDict` stored test cases but provided little support for them. Also, the basic database design mistakenly keyed test cases to words. Experience has shown that test cases are separate animals that may, or may not, be tied to particular words. JAR treats test cases with the same respect that it affords words.
3. My cross-referencing tools do not stop at locale, (or object), boundaries. Locale boundaries should be treated like black boxes just like system primitives. Respecting and recognizing interfaces is the foundation of modular software.
4. No way to store certain types of word definition scripts: Cliff Reiter's `numbers.ijs` is a good example of a script that cannot be easily saved in `JDict`.
5. Poor noun storage—loses precision on floating point types and cannot store anything larger than 20K. 20K is an absurdly low limit imposed by the underlying DDE support of FoxPro and J. To store large objects various ugly kludges (swap files, DDE loops) must be used. Oddly this limit has not been suffocating because the typical J word is a verb with a mean size of 180 characters.
6. Cannot open more than one dictionary at a time—no dictionary paths. Trying to stuff all words into one dictionary has proven unworkable. During system development you need some clean mechanism for separating experimental changes from stable definitions. A series of dictionaries linked by a search path can easily provide this facility.

7. The asynchronous nature of DDE makes it difficult to control all server operations from J without risking time-outs or worse. I chose to live with this limitation because fixing it would have forced me to give up J's immediate execution mode which is the main reason anyone is using J.
8. The FoxPro server uses a large runtime support file that makes it difficult to distribute and support the system. If it's not on CD-ROM forget it.
9. The FoxPro browser was always to wide open. You can enter nonsense at anytime in any one of the display tables. The JAR browser will be built on a strict look but don't touch basis.

I WANT A MORE FLEXIBLE TOOL

The main reason I am building a second generation dictionary is that I want a more flexible development tool. FoxPro is an excellent high performance database system but it typically requires a lot of code to perform tasks that can be trivially dealt with in one or two lines of J. The same can be said for my ODBC, Visual Basic, PowerBuilder and Delphi efforts. Performance was the only *rational*[1] reason I ever used these tools. *The J system has steadily closed the performance gap and is now fast enough that sacrificing the language's awesome expressiveness for machine cycles no longer makes sense.*

[1] There are many *irrational* reasons for using particular software tools. The number one braindead reason for using a particular tool is that the customer has a corporate standard. A corporate standard may make sense just don't bet any large sums on it!.