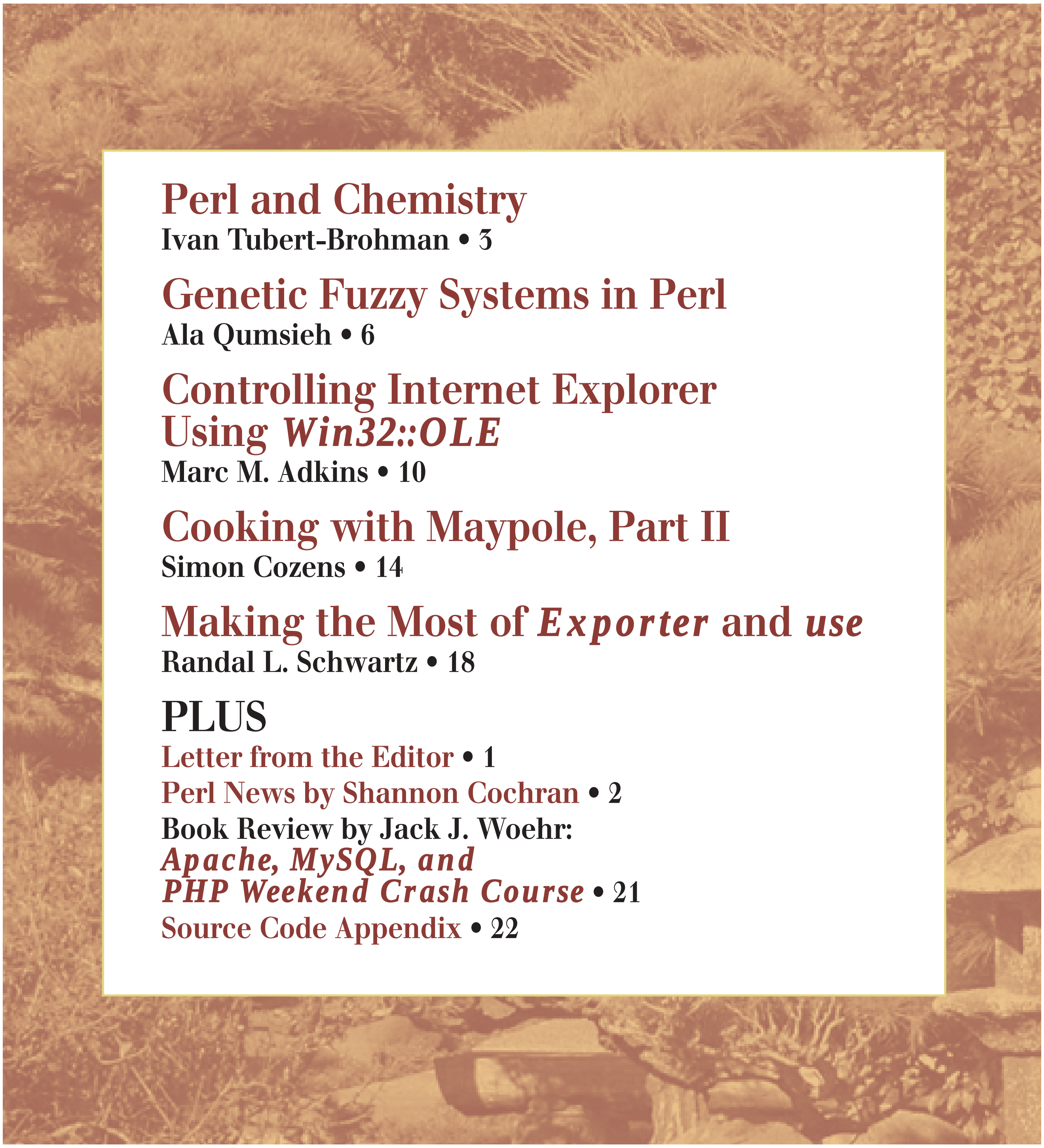
CMP
United Business Media

June 2004

The Perl Journal



LETTER FROM THE EDITOR

Keep it Simple, But Not Too Simple

Scripting languages fill an uncomfortable niche between simplistic macros and full-out application development. This can be a tough spot to call home: A scripting language has to accommodate a user base that can often have a high proportion of neophytes; it has to interact with the wide range of applications and data that everyday users are most likely to encounter; and it has to be highly extensible to avoid obsolescence. These requirements can, at times, seem mutually exclusive.

Similarly, programmers using such languages are often faced with the seemingly contradictory goals of making an application simultaneously very simple and highly functional. Of course, this is true for programmers in any language, but in languages like Perl, we're expected to do it quickly and produce applications that solve some very thorny problems. We're also expected to produce applications in spaces where commercial apps don't exist or aren't very good. Often, there's no commercial app to do the job because the problem domain is just too specialized.

Sometimes "specialized" is really just another word for "messy." Messy problems are the ones that have lots of cases that are exceptions to the rule. Messy problems are hard to encapsulate and hard to model an object around. But Perl and other scripting languages are the champions of messy situations. That's what they're bred for. To some extent, this is our fate as Perl programmers: clean up the mess.

But that's not all bad. Automating very messy processes can have huge benefits for users, precisely because messiness is what makes these processes time consuming and unpleasant when done manually. Taking away that burden can really make you someone's hero—but it does place special burdens on the programmer. It means a long development cycle, because shaking out all the weird exceptions and boundary cases can take a long time. It can mean a complex user interface, or a complex configuration file. It can require especially extensive documentation. It certainly means paying special attention to that trade-off between an application's ease of use and its power to handle everything the user might want. When that can encompass a host of niggling details, feature creep becomes more than just a delay—it can utterly slay a project.

Finding the right spot on the simplicity/functionality spectrum has effects long after the project is (supposedly) finished. It has a direct effect on your support costs. An application that is complex to use will lead to phone calls from users. On the other hand, an application that doesn't take into account the complex needs of your users will never be used.

Sometimes, the biggest challenge in automating a messy process is just getting users to realize why the problem is so messy. Questions like "Why do I have to tell the program to do that? Why can't it just figure that out for itself?" reveal that often, users don't even realize that they were making conscious decisions when doing tasks manually. If you do something enough, the various decisions involved become completely unconscious. It's easy to forget that actual input must be given at those points. Ironically, it often isn't until people are faced with an application that automates their tasks that they realize just how complex their tasks actually are.

[illegible]

Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.
THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: $18.00 for one year. Payment may be made via Mastercard or Visa; see http://www.tpj.com/. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.

CMP
United Business Media

The Perl Journal

EXECUTIVE EDITOR
Kevin Carlson

MANAGING EDITOR
Della Song

ART DIRECTOR
Margaret A. Anderson

NEWS EDITOR
Shannon Cochran

EDITORIAL DIRECTOR
Jonathan Erickson

COLUMNISTS
Simon Cozens, brian d foy, Moshe Bar, Randal Schwartz, Andy Lester

CONTRIBUTING EDITORS
Simon Cozens, Dan Sugalski, Eugene Kim, Chris Dent, Matthew Lanier, Kevin O'Malley, Chris Nandor

INTERNET OPERATIONS
DIRECTOR
Michael Calderon
SENIOR WEB DEVELOPER
Steve Goyette
WEBMASTERS
Sean Coady, Joe Lucca

MARKETING/ADVERTISING
PUBLISHER
Timothy Trickett
MARKETING DIRECTOR
Jessica Hamilton
GRAPHIC DESIGNER
Carey Perez

THE PERL JOURNAL
2800 Campus Drive, San Mateo, CA 94403
650-513-4300. http://www.tpj.com/

CMP MEDIA LLC
PRESIDENT AND CEO Gary Marshall
EXECUTIVE VICE PRESIDENT AND CFO John Day
EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner
EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING Jeff Patterson
CHIEF INFORMATION OFFICER Mike Mikos
SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz
SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro
VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson
PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletra
PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria
VICE PRESIDENT, GROUP PUBLISHER APPLIED TECHNOLOGIES Philip Chapnick
VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK MEDIA NETWORK Michael Friedenberg
VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS Paul Miller
VICE PRESIDENT, GROUP PUBLISHER ENTERPRISE ARCHITECTURE GROUP Fritz Nelson
VICE PRESIDENT, GROUP PUBLISHER SOFTWARE DEVELOPMENT MEDIA Peter Westerman
VP/DIRECTOR OF CMP INTEGRATED MARKETING SOLUTIONS Joseph Braue
CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT Shannon Aronson
CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT Michael Zane
CORPORATE DIRECTOR, PUBLISHING SERVICES Marie Myers

Shannon Cochran

Perl News

Movable Type Contest Announced

Six Apart is sponsoring a developers' contest for its Perl-based Movable Type weblog publishing platform. Entries must consist of Movable Type plugins; a total of $20,000 in prizes will be distributed to six winners. The grand prize is the winner's choice of an Apple G5 or a Dell Pentium 4.

Submissions will be judged on four criteria: broad applicability, robustness, integration, and clean design. The judges are specifically looking for plugins that take advantage of the new, "developer friendly" architecture—including a new suite of APIs—in Movable Type 3.0. The deadline for entry is July 6; details are at http://www.movabletype.org/contest.shtml.

"What Happened to Perl-QOTW?" Tom Asked Quizzically

The Perl Quiz of the Week mailing list has recently been revived after a long hiatus. Since Mark Jason Dominus had become too overwhelmed with other work to generate quizzes and summaries every week, the list now works via a community format; the quizzes and summaries are created by volunteer guest editors, and Mark Jason Dominus acts as the list administrator. The difficulty level of the quizzes varies between "regular" and "expert." Archives and subscription links to the Quiz of the Week mailing list are at http://perl.plover.com/~alias/list.cgi/0/; an associated discussion list is archived at http://perl.plover.com/~alias/list.cgi/1/.

Conferences, Committees, Chances to Contribute

YAPC, an offshoot of the Perl Foundation, has formed a new committee dedicated to encouraging YAPC conferences in Europe. "The aim is to provide a focal point, organize online registration, mailing lists, technical and conference advice and to generally support activities encouraging Perl Conferences and Workshops in Europe," the YAPC::Europe committee announced. Norbert Gruener, Ann Barcomb, Philippe Bruhat, David Lacravate, Nicholas Clark, and Richard Foley currently sit on the committee; ad-hoc members are expected to be added in the future. The YAPC::Europe web site is http://www.yapceurope.org/.

In other conference news, the Open Source Developers Conference (organized by the by the Melbourne Perl Mongers) has also issued a call for papers. Lightning talks, standard 20-minute presentations, and long (45-minute) talks will be scheduled; topics may cover business issues, implementation guides, or design aspects of open-source technologies. Deadline for proposals is June 28th. The conference will be held December 1–3 at Monash University in Melbourne. You can submit proposals at http://www.osdc.com.au/papers/call_for_papers.html.

Finally, registration is open for the sixth German Perl Workshop 2004, which will be held June 29–July 1 at the Barbara-Künkelin-Halle Schorndorf, near Stuttgart. "In contrast to the international Perl Conferences that keep growing," organizers write, "we want to retain the workshop atmosphere as much as possible. Therefore, we stay with one track only and restrict attendance to approximately 200 participants." Attendance fees are 75 Euros, with discounts for students and speakers.

Follow That Parrot

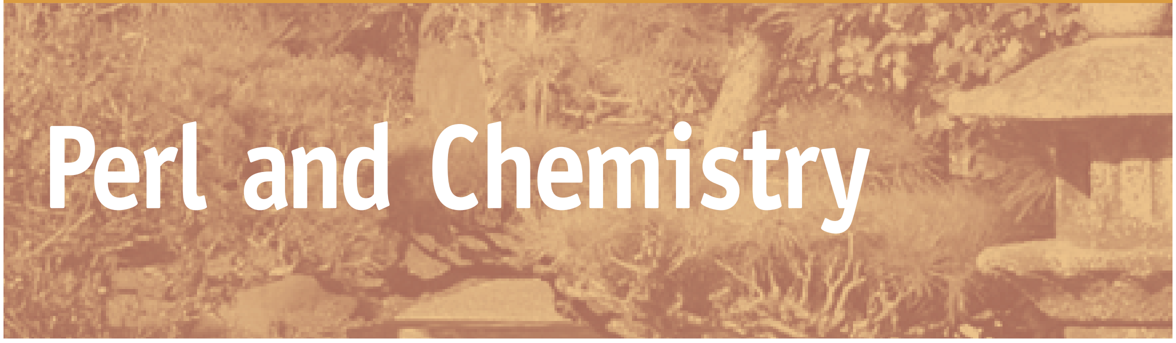
Dan Sugalski has started a new documentation effort for the Parrot project, saying "I'll try and make sure [questions] get answered, and if some kind volunteer (hint, hint…:) would like to both take questions that hit the list and add them here, as well as taking the answers as they get put up and repost 'em to the list, well…that'd be swell."

The new wiki is at http://www.vendian.org/parrot/wiki/bin/view.cgi/Main/HowDoIDo. It notes: "Any questions on the usage of Parrot are fair game here, from the trivial to the deep C-level stuff." So far, the questions include "How do I handle lexical variables in PIR?" and "How do I call a function in PIR?"

New Perl Editor Released

LuckaSoft has released the EngInSite Perl Editor, a Windows-based IDE specifically designed for creating, testing, and debugging Perl scripts. The "lite" version is free (though not open source). The "professional" version, which costs $49.00, adds a regex tester, a POD viewer, an integrated FTP server and a light web server, and technical support. You can find out more at http://enginsite.com/Perl.htm.

Ivan Tubert-Brohman



Perl and Chemistry

A typical day in the life of an aspiring computational chemist: I had to find a few thousand structures in the Cambridge Structural Database and figure out the statistical distribution of certain bond lengths. The software for accessing the database was available in the library and it had a feature to do exactly what I wanted. Unfortunately, it did not work as advertised. In the end, I concluded that I could export the structures as text files, which could then be opened with any molecular visualization program with which I could measure the distances that I wanted interactively. Yeah, right. Like I'm going to open a thousand files one by one and click on a bond to find out the bond length and then copy the number to a spreadsheet. I'm a Perl programmer and Perl programmers are lazy!

Luckily, I had some modules I had started writing for dealing with these types of problems. I initially wrote a 20-line script that solved it, but I still thought that that was too long. So, I continued developing the modules (which collectively I dubbed the PerlMol project) with the goal of being able to solve this kind of problem with one line of code.

This article is about how Perl can be useful for dealing with chemical information. I hope that even nonchemists may find it interesting, as it illustrates some problems of general interest in computer science and in the practice of designing and implementing Perl modules.

Chemistry 101

The idea of structure is one of the most fundamental concepts in chemistry: This is the idea that the properties of matter are determined by the arrangement of atoms in space. Throughout the history of modern chemistry, people have developed various ways of representing chemical structures, from the familiar structure diagram to mechanical models and computer-generated images. It goes without saying that computers are used every day in the chemical sciences. Therefore, one of the central issues in computational chemistry and chemical informatics has to be the representation of chemical structures in the computer. Depending on the application, one or both of the following approaches is typically used for representing molecules:

- **Molecular graphs.** A graph is a set of nodes and arcs, where the arcs connect the nodes. In a chemical graph, the nodes are the atoms and the arcs are the bonds. Molecular graphs are a subset of the more general mathematical concept of a graph; for example, the arcs do not have direction and both the arcs and the nodes are "colored." The mental model that organic chemists have of bonds, valence, rings, and functional groups is mostly based on this idea of a molecule as a graph.
- **Tridimensional vectors.** The position of each atom in space can be represented by its (*x*, *y*, *z*) coordinates. This approach has the advantage that it allows for a more realistic representation of the molecule and it has all the information required, from a rigorous physical standpoint, for the calculation of the properties of the molecule. However, it is less intuitive because it does not account for bonds and functional groups explicitly.

Ivan is a graduate student at the Yale University Chemistry department and can be contacted at ivan@tubert.org.

Common Problems in Computational Chemistry

There are already countless programs that are used for doing chemistry in the computer. These programs can:

- Draw structural diagrams.
- Calculate the energy and other properties of molecules with a basis in some physical theory, such as quantum mechanics or molecular mechanics.
- Manage databases of chemical information.
- Visualize big molecules such as proteins and DNA.
- Simulate the movement of atoms and molecules in solution.

(See http://dmoz.org/Science/Chemistry/Software/ for a more detailed categorization.)

One problem with many of these programs is that almost every one has its own file format, which makes interoperation hard. Many programs know how to read the output of some other programs, but never all. There have been open-source projects that deal specifically with the problem of file conversion, most notably OpenBabel (http://openbabel.sourceforge.net/).

Another problem is that, while there are many nice programs with glossy GUIs that allow you to visualize or modify molecules using the mouse, when you have to deal with a large number of molecules, the options are not as diverse. What is needed is a very flexible batch tool or a programming toolkit focused on dealing with molecules. Of course, there are already a few of these toolkits around. Some of these are proprietary, such as Daylight (http://www.daylight.com/), OEChem (http://www.eyesopen.com/products/toolkits/oechem.html), and ChemAxon (http://www.chemaxon.com/). Some are open source, such as the Chemistry

Development Kit (http://cdk.sourceforge.net/), OpenBabel, and JOELib (http://www-ra.informatik.uni-tuebingen.de/software/joelib/index.html). The most popular languages for these toolkits have been C++ and Java (OEChem also has a Python interface to the underlying C++ library).

I was surprised that CPAN was sorely lacking in terms of modules for chemistry. The only available modules were *Chemistry::Element*, which allows you to convert between atomic number, element symbol, and element name and store other elemental information; and *Chemistry::MolecularMass*, which calculates the mass from the molecular formula. There were no modules that actually dealt with the structure of molecules. While some of the options in other languages are not bad, I was looking for something with the simplicity and conciseness of Perl that could allow me to write "chemical one-liners" to solve small problems very quickly, without having to compile anything. Hence, PerlMol was born.

OK, enough theoretical stuff. On to the Perl.

Designing PerlMol

Molecules seem to scream that they want to be represented as objects. It is very natural to represent the molecule as an object that contains a group of atom objects and a group of bond objects. Atoms have properties such as coordinates, which are conveniently represented by vector objects. (For the vector objects, I used the *Math::VectorReal* module by Anthony Thyssen.) This leads naturally to the three most important PerlMol classes: *Chemistry::Mol*, *Chemistry::Atom*, and *Chemistry::Bond*. I like to think of them as the trinity of chemical classes. Some other toolkits (usually written in Java) tend to have a proliferation of classes in a complex hierarchy: An *Atom* is an *AtomType*, which is an *Isotope*, which is an *Element*, which is a *ChemObject*. Atoms are held by *AtomContainers* and so on. I decided to keep the flattest possible hierarchy because I do not share the philosophy that everything has to be an object. For example, I think that Perl arrays are more than powerful enough, so that container objects are not really needed (but I did keep a root class, *Chemistry::Obj*, to keep some common methods needed by the other classes).

One of the design choices was that PerlMol should "know" as little chemistry as possible, at least for the core classes. That is because there are many different—and often incompatible—ways of thinking about molecules and other chemical entities. Different programs and different file formats often assume different things. For example, it is almost always assumed that a bond connects two atoms. However, that is not entirely true because bonds that connect three atoms are known to exist, although they are not common. To be on the flexible side, atom objects have an array of atoms of unspecified length. In a way, the idea of "molecule" that is used in PerlMol should aim to be the "lowest common denominator," so that it can effectively represent different conventions. It is still possible, and indeed a good idea, to have some classes or methods that are more specific (and restrictive).

The most basic operation in PerlMol is constructing a molecule object. As you would expect, this can be done as follows:

```
my $mol = Chemistry::Mol->new;
```

That gives you a "blank molecule," to which you can add atoms and bonds with methods such as *$mol->add_atom* and *$mol->add_bond*. However, most of the time you do not want to create a molecule from scratch, but to read it from a file:

```
my $mol = Chemistry::Mol->read("myfile.mol");
```

For this to work, you need to have a module that can read that file format. The format can be specified as an option to the *read* method, or it can try to guess it automagically from the contents of the file or from the filename extension (the .mol extension in the example is used for files with the MDL molfile format). All the file I/O modules inherit a common interface from the *Chemistry::File* class and they can be used explicitly or they can all be loaded automatically (in which case, you can think of them as plug-ins).

Now you probably want to coerce some information out of the molecule, which you can easily do by calling other *Chemistry::Mol* methods, most notably *$mol->atoms*, and then the methods on the atom objects themselves. Some examples:

```
print $mol->name;
my $a1 = $mol->atoms(1);  # get the first atom
print $a1->coords;        # outputs something like [0.000, 1.679, -1.373]
if ($a1->symbol eq "Pb") {
    $a1->symbol("Au")     # Transmute lead into gold
}

# select all the carbon atoms in the molecule
my @carbons = grep { $_->symbol eq "C" } $mol->atoms;

# find carbon atoms within 2.5 Angstroms
my @close_carbons = grep { $a1->distance($_) < 2.5 } @carbons;
$mol->delete_atom(@close_carbons);   # get rid of those atoms!
```

These samples should give you an idea about how Perl lists and functions, such as *grep*, give you a lot of flexibility for selecting atoms and then playing with them. Some visualization programs give you ways of specifying atoms from a command line by using a specialized minilanguage, but here you can use the full power of Perl. There are many other methods available in the PerlMol toolkit; if you are interested, please refer to PerlMol's accompanying POD.

One of the practical problems that I found when writing PerlMol was the use of circular references. An atom object has a reference to a bond object and vice versa. Since *perl* uses a simple reference count for deciding which objects are no longer used and should be deleted, circular references cause memory leaks. There are several ways of dealing with this problem; I chose to use weak references by means of the *Scalar::Util* module by Graham Barr (there is an article on this topic by Randal Schwartz in the July 2003 issue of *TPJ*).

Pattern Matching

Everybody knows that one of the most powerful features of Perl is its use of regular expressions. You can describe the pattern that you want to match and then find its occurrences in a string. The same idea can be applied to molecules; for example, you may need to find a sulfur atom that is bound to a carbon that is bound to another carbon that is bound to an oxygen. You could use Perl's *grep* to find the first atom, then look at its neighbors, and then "walk" the molecule one atom at a time until you find a match. That is painful. What's needed is a general pattern-matching engine; for that, I wrote *Chemistry::Pattern*. A pattern is a subclass of molecule, so you can create it in a similar way:

```
my $mol  = Chemistry::Mol->read("bigmol.mol");
my $patt = Chemistry::Pattern->read("smallpatt.mol");

# find matches of the pattern in the molecule
while ($patt->match($mol)) {
    print "Atom 1 in the pattern matches atom ", $patt->atoms(1)->map_to,
          "in the molecule!\n";
}
```

What I show above is a substructure match, which is akin to a substring match. A regular expression is more powerful than that because you can define character classes and use special operators.

The *Chemistry::Pattern* engine actually has some of that flexibility, but you need to use a special language (or file type) to specify the pattern in a concise manner. The most popular language for that purpose is called SMARTS. A simple SMARTS example would be "*[F,Cl]*[!C]*", which means "either chlorine or fluorine next to any atom (*) next to anything but carbon." It is possible to specify very complex patterns using this language. (For a detailed description of the SMARTS language, see http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html.) At the time of this writing, the *Chemistry::File::SMARTS* module is still unfinished (it only implements a subset of the language), but you can use the example above as follows:

```
my $pattern = Chemistry::Pattern->parse("[F,Cl]*[!C]", format => "smarts");
```

A Molecular *grep*

One of the most common tasks in chemical informatics is to find molecules matching a pattern. A program to do this has to loop over all the files and match them against the pattern. You can write a simple "molecular *grep*" as follows:

```
1       use Chemistry::Mol;
2       use Chemistry::Pattern;
3       use Chemistry::File ':auto'; # use all available file formats
4       use strict;
5       use warnings;

6       if (@ARGV < 2) {
7           die "Usage: molgrep <smarts> <fname> ...\n";
8       }

9       my $smarts = shift;
10       my $patt = Chemistry::Pattern->parse($smarts, format => "smarts");

11       for my $fname (@ARGV) {
12           # note: fiiles may contain more than one molecule
13           my @mols = Chemistry::Mol->read($fname);
14           for my $mol (@mols) {
15               while ($patt->match($mol)) {
16                   print "File $fname matches!\n";
17               }
18           }
19       }
```

To find molecules matching the pattern just discussed, you could type:

```
molgrep '[F,Cl]*[!C]' *.mol
```

Even More Laziness: Mok

This *molgrep* is all well and good, but what if you need to extract more information from the molecule? That was the problem that I mentioned at the beginning of this article. You can just edit *molgrep* and add whatever you need in the inner loop (line 16). That sounds like needless repetition and lots of cut-and-paste whenever you want something different. Wouldn't it be nice if the loops, file reading, and the like were implicit and we only had to write line 16? That would be similar to the behavior of awk, or *perl -n*. What is needed, then, is a "molecular AWK." I decided to shorten it to Mok, which also happens to remind me of Ookla the Mok, a character from the animated series *Thundarr the Barbarian*. (If you're curious about that, see http://dmoz.org/Arts/Animation/Cartoons/Titles/T/Thundarr_the_Barbarian/.)

I will not include all the Mok code here, but you can find it on CPAN. The essence of it, as you can imagine, is an *eval*. Here, we can harness the ability of Perl to compile user-supplied code dynamically.

A traditional AWK program consists of a series of patterns with associated "action blocks." For example, if you want to say "hello" whenever you see someone, you would code the following:

```
/someone/ { print "hello" }
```

I was looking for something with the simplicity and conciseness of Perl, which could allow me to write "chemical one-liners"

For the Mok language, I borrowed AWK's pattern-action format, but the code inside the block is Perl code, which is *eval*ed. The patterns are expressed as SMARTS strings. You can rewrite the molecular *grep* example from the previous section as a Mok one-liner:

```
mok '/[F,Cl]*[!C]/ { print "$file matches!\n" }' *.mol
```

All of the required *Chemistry::** modules are used implicitly. The block is executed for each molecule matching the pattern and it has several predefined variables available: *$MOL* is the current molecule, *$FILE* the current filename, *@A* are the atoms matched by the pattern, *@B* are the bonds, and so on. Now if you want to do something other than printing the filename of the matching molecule, you only have to change the inner block. To print out the *S=O* bond lengths in a group of molecules, you just write:

```
mok '/S=O/g { printf "%s: %.3f\n", $MOL->name, $B[0]->length }' *.mol
```

where the *g* flag after the pattern means "global" (i.e., give all the matches for each molecule).

Notice how we went from hours of interactive point-and-click to writing a 20-line script to something you can type at the command-line in 30 seconds!

Conclusion

In this article, I have shown how Perl can be used for dealing with some of the typical problems in chemical informatics. I mentioned some of the design and programming issues that are applicable to writing almost any object-oriented module, such as using plug-in modules for I/O and ways of dealing with circular references. I showed how creating a specialized minilanguage such as Mok can turn routine 20-line scripts into one-liners. All of this was inspired by Perl's philosophy that "easy things should be easy, hard things should be possible." I have seen other toolkits that make the hard things possible, but at the price of making the easy things complicated or verbose.

If you are interested in the PerlMol project, please visit http://www.perlmol.org/. Contributions are welcome, either with coding and testing, documentation, or by making a donation.

TPJ

Ala Qumsieh

Genetic Fuzzy Systems in Perl

In a previous article, I discussed how to use the *AI::Fuzzy-Inference* module to balance a rolling ball on a horizontal rod that is pivoted about its center. This worked because we created a set of fuzzy rules that defined the future behavior of the system based on its current state. (That article, "Fuzzy Logic in Perl," appeared in the June 2003 *TPJ*, which is available to *TPJ* subscribers in the "Archives" section of http://www.tpj.com/.) Those rules were of the form:

```
If   $posBall is far_left   AND  # ball is close to left edge of rod
     $velBall is slow       AND  # ball almost stationary
     $thRod   is medium_neg      # right edge of rod is slightly lowered
THEN $dTheta  is large_pos       # raise the right edge by large amount
```

In our system, we had three input variables—*$velBall*, *$posBall*, and *$thRod*—that defined the current velocity of the ball, the position of the ball, and the angle of the rod, respectively; and one output variable, *$dTheta*, which defined the change in the angle of the rod for the next time step. In addition, each of our input variables had five term sets associated with it, which defined the possible fuzzy sets that each variable can belong to at any instant in time. This meant that there are 5^3, or 125, different possible states that the system can be in at any one point in time; hence, 125 different rules that need to be specified to define the complete behavior of the system. And this is a relatively simple system. If we require finer control over the system, we can extend the term sets of each variable to seven, for example, which would give us a total of 343 different rules. We can see that the number of rules we have to manually define grows exponentially as we add more input variables and more term sets. Furthermore, we have to manually validate those rules by running the system and observing its behavior, then going back, figuring out which rules need to change, modifying some more, and so on. This is a very tedious process that can be quite time consuming and impractical for more complex systems.

One way to solve this problem is to use a genetic algorithm (GA) to evolve the set of rules for your system. Generally speaking, GAs are search algorithms that borrow techniques from natural genetics to guide the trek through search space. Such applications of GAs to evolve rules for fuzzy systems are referred to as "genetic fuzzy systems." My previous article covered these fuzzy systems. I will now briefly describe GAs in more detail and explain how the *AI::Genetic* module can be used to evolve the fuzzy rules for our given problem.

Ala works at NVidia Corp. as a physical ASIC designer. He can be reached at aqumsieh@cpan.org.

Introduction to Genetic Algorithms

As mentioned earlier, genetic algorithms are search algorithms that borrow techniques from natural genetics to traverse a problem's search space more efficiently. The basic idea is to maintain a population of individuals where each individual is defined by a chromosome that represents a candidate solution to the problem whose solution is being sought. Each chromosome is represented by a set of genes that directly correspond to the variables that define the state of the system to be solved. A typical GA strategy looks like this:

```
initialize GA;

while (not termination condition) do
    evaluate fitness;
    selection;
    crossover;
    mutation;
end
```

A GA typically starts off with a population of randomly generated individuals. Then it enters a loop, trying to improve upon those random solutions. During each iteration of this loop, the GA is said to evolve a generation. Evolution happens in the form of natural selection akin to what happens in nature. In each generation, the individuals are tested and rated as possible solutions to the given problem. This rating is called a "fitness value." Based on this rating, a number of individuals are then selected to undergo crossover and/or mutation. Crossover is similar to sexual reproduction in nature. Here, two individuals are selected for mating, which results in two brand new "child" individuals that share genes from both "parents." Furthermore, these child individuals might undergo mutation where one or more genes switch states. Finally, the new children are injected into the population, marking the end of the current generation. Evolution stops when a certain number of generations have passed or when some other criterion has been fulfilled; for example, when an acceptable solution has been found. Now we'll look into each of the aforementioned steps in more detail.

Fitness Evaluation

Individuals are rated by passing them on to a "fitness function" that simulates the problem to be solved, and returns a positive number that is proportional to how well the given individual's chromosomes adapt to the problem. Users of GAs quickly discover that designing the fitness function is the single most important aspect of using GAs to solve hard problems. There are two main reasons for this:

1. Most of a GA's runtime is spent in the fitness function. The fitness function has to be run once for each individual in the population. Furthermore, in each new generation, new individuals are constantly being generated and the fitness function has to be run for those to evaluate their fitness. Typically, a population would consist of hundreds, sometimes thousands, of individuals. It becomes readily apparent that efficient design of the fitness function can result in tremendous speed gains in overall runtime.
2. Individuals only adapt themselves to the fitness function. The purpose of the fitness function is to test the adaptability of each individual to the given problem. If, for any reason, the fitness function fails to test some corner case of the problem, then no chromosomes will evolve to take that corner case into account. For example, in one of my first attempts to evolve the rules to solve our problem of balancing a moving ball on a rod, I coded the fitness function such that, for each individual, it would randomly choose six different initial conditions and rate the individual based on how the system reacts under those conditions. The final solution behaved spectacularly well under a large number of initial conditions, but in some cases it failed miserably due to the fact that the fitness function did not test the most-fit individual under all possible conditions.

Selection

Selection simulates the principle of "survival of the fittest." This step basically determines which individuals get to mate with others and pass on their genes in subsequent generations. There are many different selection operators that can be used. Among the most widely used ones are "roulette wheel" and "tournament."

In the roulette wheel selection, the chance of an individual being selected is proportional to the individual's fitness. The name comes from the analogy in which each individual occupies an arc along a wheel. The extent of the arc is proportional to the individual's fitness. The wheel is then "spun" and the individual on which the wheel stops is selected.

In tournament selection, a set of individuals are randomly selected from the whole population and the fittest of those individuals is the one to be ultimately selected.

Crossover

Crossover is the actual mating between two individuals. Typically, a GA defines a crossover rate that defines the probability that two individuals will produce offspring. This probability is usually high because, in practice, we want to explore as many different solutions as possible. If mating does occur, then it will result in two new individuals that will get added to the population.

As with selection, there are many different crossover operators. "Single point" crossover works by choosing a random point along both parents' chromosomes and splitting each in two. Then the head of one is paired with the tail of the other to create two new individuals. In "two point" crossover, each chromosome is broken along two points into three parts, and the middle part of each parent is swapped with the other. In "uniform" crossover, each gene in each parent is looked at independently and randomly assigned to one of the children.

Mutation

Mutation randomly alters the state of a gene. It is a very useful operator but has to be dealt with carefully. The purpose of mutation is to avoid having all the population stuck in a local minimum. Suppose, for example, that in a certain population, the third gene of every individual has the value 0. Then, in the absence of mutation, there is no way for that gene to switch its value only by crossover since both parents will have the same value for that gene, and will hence pass it on to the offspring. Mutation allows, with a small probability, the mutation of genes, which correspond to jumps in the search space. If the jump is large enough, an individual might end up at a potentially lower minimum, yielding a better solution. The probability of a mutation happening is controlled by another variable of the GA called the "mutation rate," and is typically very small. If this value was large, then more genes will be mutated, and the offspring will look less like their parents and more like randomly generated individuals. Thus, a very delicate balance has to be struck where we allow just enough mutation to prevent getting locked up in local minima, but not too much mutation to disrupt the benefits of crossover.

GAs are search algorithms that borrow techniques from natural genetics to guide the trek through search space

Getting and Installing the Module

You can grab a copy of *AI::Genetic* from your local CPAN mirror at http://search.cpan.org/~aqumsieh/. The latest version as of this writing is 0.02. You can install it using the traditional method:

```
perl Makefile.PL
make
make test
make install
```

If you're on Windows, you can simply type *ppm install AI::Genetic* at the command prompt. Alternatively, since it's all in pure Perl, you can unpack it in any place where *perl* can find it.

Using *AI::Genetic* to Evolve Fuzzy Rules

Before we start describing the code, let's recap the problem very quickly. We are trying to come up with a set of fuzzy rules that can control a fuzzy system, allowing it to successfully balance a rolling ball on a flat rod. The rod is free to rotate about its center, and we assume a two-dimensional environment for simplicity.

I will only outline details essential to developing our application. You can find more information in the *AI::Genetic* documentation. To better understand the approach I have taken, we need to look at how I decided to represent the problem.

Representation

Our system has three input variables and one output variable. Each of these variables has five term sets:

- **input: posBall**
  Term sets: *far_left, left, center, right, far_right*
- **input: velBall**
  Term sets: *fast_neg, medium_neg, slow, medium_pos, fast_pos*
- **input: thRod**
  Term sets: *large_neg, medium_neg, small, medium_pos, large_pos*
- **output: dTheta**
  Term sets: *large_neg, medium_neg, small, medium_pos, large_pos*

This means that we have 125 possible rules that correspond to the 125 possible combinations of input states. We will use the GA to select the proper output term set for each of the input states. For example, for the state:

```
posBall = far_left

velBall = slow

thRod   = medium_pos
```

the GA should tell us what *dTheta* should be, which defines how the system will respond to balance the ball.

To represent this, I define each individual to have 125 genes. Each of these genes can assume one of the five possible values for *dTheta*. The first gene will correspond to the input state *far_left, fast_neg, large_neg*, the second to *far_left, fast_neg, medium_neg*, and so on.

Using *AI::Genetic*

The first thing to do is to create an *AI::Genetic* object and define its parameters:

```
use AI::Genetic;

my $ga = new AI::Genetic(
          -population => 50,
          -crossover  => 0.9,
          -mutation   => 0.05,
          -type       => 'listvector',
          -fitness    => \&fitness
         );
```

This creates an *AI::Genetic* object that has a population of 50 individuals. The crossover rate is 0.9 and the mutation rate is 0.05. The *-type* option specifies the type of chromosomes to use. *AI::Genetic* supports three types of genes:

- **bitvector.** Each gene is a bit and can be either on or off.
- **listvector.** Each gene is a string that can take one value from a specified list of possible values.
- **rangevector.** Each gene is an integer that can take a value from within a range of possible integers.

In this case, we are using *listvector* because the rules we are trying to evolve are strings. Finally, we pass on a reference to the fitness function, which I have called *fitness*. Additionally, we can specify a termination subroutine to stop evolution once we reach a certain milestone.

Next, we have to initialize the GA with random individuals. We do that via the *init()* method:

```
$ga->init([
        map [qw/large_neg
                 small_neg
                 zero
                 small_pos
                 large_pos/], 1.. 125
       ]);
```

This tells the GA that each individual is to have 125 genes, each of which can have one of the five values indicated. This will create 50 *AI::Genetic::Individual* objects with random genes.

Now we are ready to evolve our population. We do that via a call to *evolve()*:

```
for my $i (1 .. 10) {
  $ga->evolve('rouletteUniform', 50);
  my $fit = $ga->getFittest;
  print "Fittest so far has score = ", $fit->score, ".\n";
}
```

The first argument to *evolve()* is an evolution strategy to use. *AI::Genetic* comes with nine predefined strategies and allows you to create your own. A strategy defines how evolution will take place; for example, which operators to use for selection and crossover. In this case, I chose the *rouletteUniform* strategy, which uses roulette-wheel selection and uniform crossover. The *AI::Genetic* documentation explains the other available strategies and how to create your own in more detail. The second argument to *evolve()* specifies the number of generations to evolve. Note that we could have specified 500 as the second argument to *evolve()* and not used the f*or()* loop at all. However, I like to split my evolution like this so that I can get an update every 50 generations to tell me how my population is doing.

To access the fittest *AI::Genetic::Individual* object, we use the *getFittest()* method. The *AI::Genetic::Individual* class defines a number of useful methods, one of which is *score()*, which returns the fitness value of the individual. This is really where the fitness function is executed, and the result is cached such that subsequent calls to *score()* will return the cached value. If everything is going fine with our GA, then we should see the score of the fittest individual steadily increase as more generations evolve. Another useful method is *genes()*, which returns a list of the individual's genes.

Once our evolution is done, we can use the *getFittest()* method again to gain access to our best solution, and then use the *genes()* method to access the list of genes of that individual, which directly correspond to our fuzzy rules.

The Fitness Function

Our fitness function is simple in this case. It is passed a list of genes that represent the state of the output variable for each possible combination of input variables. Those genes are used to create the list of rules. Then an *AI::FuzzyInference* object is created with those rules and a few simulations are run with varying initial conditions. Each simulation is run for a maximum of 1000 time steps, or until the ball either falls to the ground, or is balanced and comes to a halt. A score is kept for each simulation and the total fitness of the individual is the average of the scores of all its simulations. Scores are assigned as follows: If the ball falls at time *T*, then the score is equal to *T*. If a ball is completely balanced (i.e., it comes to a halt on top of the rod), then the score will be equal to 1000 plus a bonus that is inversely proportional to the length of time it took to balance the ball; the faster the balancing, the bigger the bonus. If time expires without the ball falling or being balanced, then the score will be 1000.

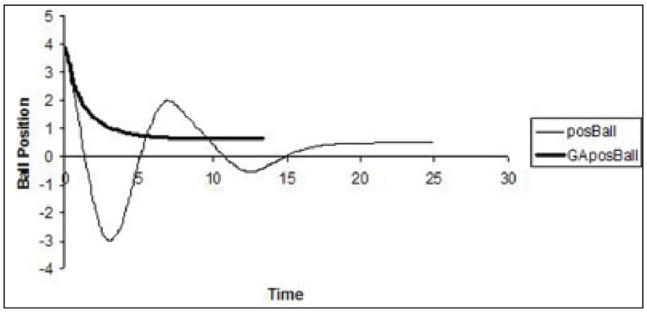


Figure 1: Position of the ball over time.

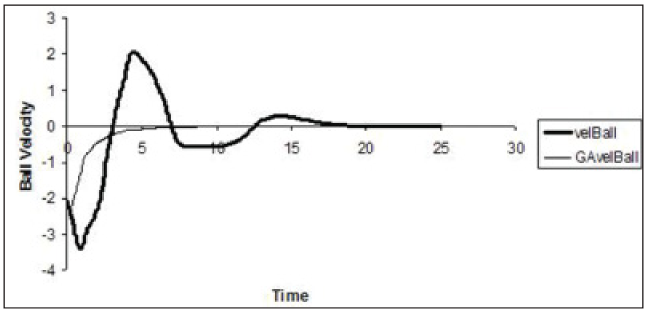


Figure 2: Velocity of the ball over time.

Results

I ran a simulation of a population of 50 individuals for 25 generations. Typically, you would have a larger population evolving for a larger number of generations, but this problem is simple enough that I can get away with a smaller population and fewer generations to save time. The fitness function tested each individual's behavior under a variety of initial conditions that took into account a large number of possible scenarios. Once the simulation ended, I took the rules defined by the best fit individual, and plugged them into the fuzzy system simulator. A typical result is shown in Figures 1 and 2.

Figure 1 shows the position of the ball plotted on the *y*-axis versus time on the *x*-axis. Two curves are shown. The thin line depicts the behavior of the system using the set of rules that I originally came up with manually by trial and error. The thicker line is the corresponding behavior using the genetically evolved rules. Figure 2 shows the corresponding velocity versus time plot. It is evident that the genetically bred rules do a much better job of balancing the ball and bringing it to a halt quicker and with virtually no oscillations.

Final Thoughts

Combining fuzzy systems with genetic algorithms is relatively easy once you understand what is going on. The example I give in this article is rather simple, but should help illustrate how genetic fuzzy systems work and what they could be used for.

One thing I would like to mention here is that I took the liberty of defining the number, shape, and extent of the term sets for each of the fuzzy linguistic variables. This process was iterative in the sense that I kept trying different combinations until I got something that worked well enough and stuck to it. An obvious extension to the system would be to allow it to evolve not only the fuzzy rules, but also the number of term sets per input variable, the shape of the term sets (triangular, trapezoidal, bell curve), and the extent of the term sets. This would make the system much more complicated to code, but would surely result in a more sophisticated and more efficient set of rules.

Finally, I would love to hear from anyone who finds my module useful or uses it in a cool application. Any other comments or suggestions are very welcome.

TPJ

Marc M. Adkins

Controlling Internet Explorer Using *Win32::OLE*

For programmers on Microsoft Windows operating systems, OLE/COM/ActiveX can be both a blessing and a curse. For the intrepid Perl programmer, it represents a flexible method of harnessing a variety of applications into new and useful configurations. In this article, I'll use the *Win32::OLE* module to demonstrate a very basic task: starting an Internet Explorer session from a Perl script, and pointing to *The Perl Journal*'s web site. The code that accomplishes that task is deceptively simple:

```
use     Win32::OLE;

my  $explorer = new Win32::OLE('InternetExplorer.Application')
                or die "Unable to create OLE object:\n  ",
                       Win32::OLE->LastError, "\n"

$explorer->Navigate("http://www.tpj.com/", 3);

sleep 3;
```

I'll try to provide some basic background on how this code does what it does, and to give you what you need to get started writing your own OLE code.

What is This OLE Stuff, Anyway?

OLE stands for Object Linking and Embedding and COM stands for Component Object Model. The history of these technologies is rich and varied. I won't go into them in depth here, as there is much available already on the subject. For our purposes, I'll oversimplify just a bit and say that OLE and COM provide the object model and the facilities for controlling Windows applications through a scripting interface. In our case, that interface is controlled from Perl.

Marc is a software developer at VersusLaw.com, and can be reached at Perl@Doorways.org.

Here's what you need to know about OLE on Windows:

- It's object oriented.
- Objects have properties (fields or member variables).
- Objects have methods (member functions).

Here are some other useful but noncritical facts:

- Objects can generate events to registered handlers.
- Objects can be written in different languages.
- Objects load either separately or in related groups.
- Objects are located and loaded via information stored in the Registry.
- Many Windows applications and much of the Windows operating system are built using OLE objects.

All of which boils down to the fact that Perl programmers can do a lot of nifty things via OLE.

Note: Some of the statements I'll make in this article are about COM in general. Because the Perl package is referred to as Win32::OLE, *I'll refer to the technology in the general sense as OLE.*

Back to the Example

Let's reexamine that previously shown snippet of code. First, import the OLE package:

```
use     Win32::OLE;
```

Then instantiate an object of that package with the registered name of the Internet Explorer application:

```
my  $explorer = new Win32::OLE('InternetExplorer.Application')
```

It's always a good idea to provide some sort of error checking:

```
    or die "Unable to create OLE object:\n  ",
           Win32::OLE->LastError, "\n"
```

Actually opening the browser window to *The Perl Journal*'s web site is almost ridiculously simple:

```
$explorer->Navigate("http://www.tpj.com/", 3);
```

But let's stop for a moment and talk about this last statement. The variable *$explorer* is of the package *Win32::OLE*. It is a proxy object, a stand-in for the actual OLE object that has been created though the arcane rules that govern such things (you do *not* want to see the C/C++ code for this, though the Visual Basic code is pretty simple).

Invoking the method *Navigate()* on the *$explorer* proxy causes an OLE method invocation to be made on the underlying Internet Explorer application object. The Internet Explorer OLE class publishes its API, enabling the *Win32::OLE* object *$explorer* to pick up the *Navigate()* call and pass it along to the real OLE object.

One of the key things to realize is that the basic *Win32::OLE* class does not have a *Navigate()* method. *Win32::OLE* is a generic class and *Navigate()* is a very specific function that only a browser would know how to perform. What *Win32::OLE* does know how to do is to query an OLE object for its API and provide "fake" methods (through the magic of Perl) that pass through to the actual OLE object.

The bottom line is that *Win32::OLE* allows you to pull OLE objects into your scripts and manipulate them as if they were regular Perl objects—just like you can do in Visual Basic, but without having to buy and learn VB.

The last statement:

```
sleep 3;
```

allows enough time for the browser to complete navigating to *TPJ*'s site. Without this, you'll get a popup dialog with an unhelpful message. If your program runs long enough after the *Navigate()* call for the browser to display completely, you won't need this statement.

The Magic Cookie

Perl classes normally provide a *new()* class method to instantiate new objects. With OLE, use *Win32::OLE::new()*, which instantiates a Perl object as a proxy for the actual OLE object. Pass a string to specify which OLE class to instantiate.

The string that determines the OLE object to be created is something of a magic cookie. If you are blessed with perfect documentation (and the patience to read it), you may have the solution at your fingertips. If your local library doesn't have the requisite grimoire, however, you have to dig for it yourself.

To illustrate, consider the string used in the example above:

```
InternetExplorer.Application
```

What can an object of that class do? Where did the string itself come from? What tools can we use to dig for this information?

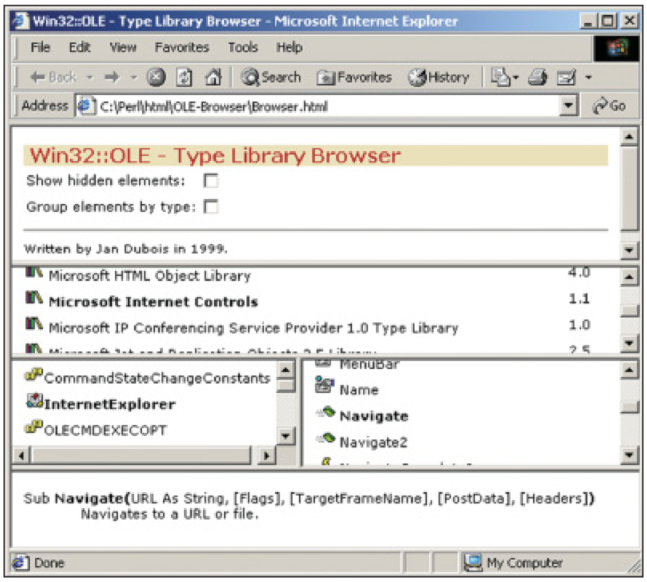
The OLE Browser

One of the most useful digging tools is the OLE Browser. It comes with the documentation in the ActiveState release of Perl. Look in the "Table of Contents" pane of the ActiveState HTML documentation for:

```
ActivePerl Components
     Windows Specific
          OLE Browser
```

Select the "OLE Browser" link and a new browser window should appear with a number of panes, as shown in Figure 1. In the largest center-ish pane, scroll down a bit and select the entry:

```
Microsoft Internet Controls
```



C:\Perl\html\OLE-Browser\Browser.html

Win32::OLE - Type Library Browser

Show hidden elements: ☐

Group elements by type: ☐

Written by Jan Dubois in 1999.

Microsoft HTML Object Library 4.0
Microsoft Internet Controls 1.1
Microsoft IP Conferencing Service Provider 1.0 Type Library 1.0

CommandStateChangeConstants
InternetExplorer
OLECMDEXECOPT

MenuBar
Name
Navigate
Navigate2

Sub **Navigate**(URL As String, [Flags], [TargetFrameName], [PostData], [Headers])
Navigates to a URL or file.

Figure 1: OLE Browser showing InternetExplorer.Navigate.

In the next pane down on the left, a list of names and icons will appear. If you scroll down you should find Internet Explorer. Select this link and the pane to the right should fill with more names and icons.

The pane on the left shows classes. The pane on the right shows the methods, properties, and events that are published by the selected class on the left. As you select methods, properties, and events the final pane on the bottom shows details about these entities.

Notice that the *InternetExplorer* class has an *Application* method. Using a dotted notation to express this, we have "InternetExplorer.Application," our magic cookie, which is in this case a class name and method name of the class. In other cases the magic cookie will be the library name dotted with the class name. In the case of Microsoft Word, this would be "Word.Application" where "Application" is a class name instead of a method name. Why the difference? Well, for one thing, the class structure exported by an OLE application or component is up to the designer of the application. The interface will reflect the needs and predilections of the designer.

In this case, there is also the lack of a document object—this would be different with a big MDI application such as Word or Excel. A web browser is stateless, so some of the normal API structure collapses, resulting in a nonstandard magic cookie.

Frankly, without detailed documentation, finding the magic cookie is often a matter of trial and error. Using the OLE browser is one way to find what you need, but you may also need to know a bit about the Registry. It's time to break out your Registry editor to look for available OLE class names.

The Registry

The Registry editor is available by bringing up the Run… dialog from the Start menu. Enter "regedt32.exe" and press OK (on some versions of Windows, there may be a different name for the executable, such as "regedt.exe"). We're only going to use the Registry editor for browsing—for gosh sakes, don't *change* anything unless you know what you're doing. Because all of Windows is controlled from the Registry, it is possible to do some real damage (e.g., you could make Windows forget how to find drivers and libraries it needs to boot).

The Registry editor comes up with several subwindows that describe different Registry trees. The HKEY_CLASSES_ROOT tree has a lot of entries. These include a lot of file suffixes and some file-type descriptors, and also a larger number of what you can think of as OLE class names (I'm glossing over a lot of details here that aren't particularly relevant to our task).

For example, you can find the following:

```
InternetExplorer.Application
Shell.Application
Word.Application
```

I'm using the first one for the example in this article. The second one (Shell.Application) can be used to do things like create shortcuts. The last one allows scripting of Microsoft Word.

Note that not all of the OLE classes can be created directly. Many of them are classes that can only be instantiated by an *Application* object. The entries are also mixed in with a lot of other stuff that you don't care about, like the suffixes and suffix classes I mentioned earlier.

Nevertheless, you can often find the magic cookie you need by searching this part of the registry. But then what? How do you get from there back to the OLE Browser? Let's focus on InternetExplorer.Application. Find that in the Registry Editor under HKEY_CLASSES_ROOT. Now double-click on that entry to expand it. Select the CLSID key under it and look at the value in the right-hand pane as shown in Figure 2:

```
{0002DF01-0000-0000-C000-000000000046}
```

This is a magic number known as a "GUID." Each OLE class will have one of these, and this is the true name of the class. Never mind where it came from—Microsoft says it's unique across all time and space. So this arcane incantation is what really connects everything together. One of the side effects of this is that you can use the CLSID in the *Win32::OLE::new()* statement to instantiate the object. This is slightly faster during execution, but much less readable in the code.

Now scroll HKEY_CLASSES_ROOT back up to its CLSID entry and double-click to expand that into a really big list of GUIDs. Look up the one GUID for InternetExplorer.Application and double-click it. Select the key TypeLib and in the right-hand pane find:

```
{EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}
```

(See Figure 3.) Okay, you're almost there now. Scroll HKEY_CLASSES_ROOT down to its TypeLib entry and double-click to expand that into yet another really big list of GUIDs. Look up the type library CLSID you just found and double-click it. Under that find a key "1.1" that, when selected, says "Microsoft Internet Controls" in the right-hand pane (your library may be a different version number); see Figure 4.

The name you just found is the name you looked up in the OLE Browser. If you look in the top pane of the OLE Browser, it says that, after all, it's the "Win32::OLE - Type Library Browser." What you have done is tracked through the Registry from the name of the class you want to instantiate to the name of the type library (*TypeLib*, remember?) in which that class resides. Now you can look it up in the OLE Browser and find out what methods, properties, and events it has.

The string that determines the OLE object to be created is something of a magic cookie

Of course, that's *way* too much work, so you can use the "typeLib.pl" script in Listing 1. Just feed it a class name from the registry and it will look up the rest of the data. It will actually search for data using the argument string as a pattern, so you can search on things like "InternetExplorer" and get:

```
InternetExplorer:
  InternetExplorer.Application:
    CLSID:   {0002DF01-0000-0000-C000-000000000046}
    TypeLib: {EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}
    Library: Win32::TieRegistry=HASH(0x1b4cb5c)
      1.1\ => Microsoft Internet Controls
  InternetExplorer.Application.1:
    CLSID:   {0002DF01-0000-0000-C000-000000000046}
    TypeLib: {EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}
    Library: Win32::TieRegistry=HASH(0x1d10134)
      1.1\ => Microsoft Internet Controls
```

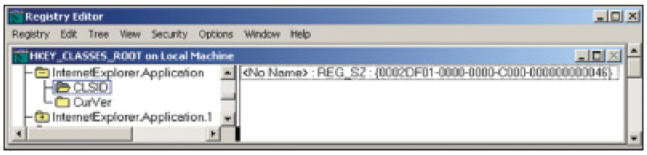
Note that there is a "generic" entry and a version-numbered entry. This is because the underlying model behind OLE provides a way to support multiple versions of OLE libraries on the same machine.

The bad news is that all of this still won't guarantee that you find the type library name. It is something of a black art. Sometimes you just have to break down and read the relevant documentation or go searching the Internet for an appropriate incantation.

A Contrived Example

Now for a contrived example using OLE to start Internet Explorer. For this, I'm going to introduce the *HTTP::Daemon* package that comes standard with Perl.

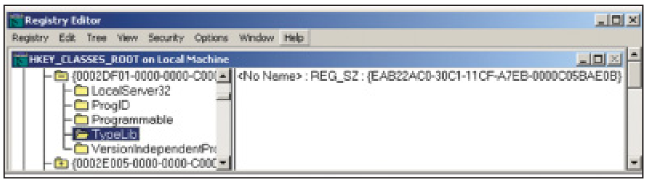
Actually, this isn't such a contrived example, at least for me. I'm really lazy and GUI code takes a while to write. So whenever I can, I use a command line interface, but sometimes I need


Registry Editor
Registry Edit Tree View Security Options Window Help
HKEY_CLASSES_ROOT on Local Machine

InternetExplorer.Application
CLSID
CurVer
InternetExplorer.Application.1

<No Name> : REG_SZ : {0002DF01-0000-0000-C000-000000000046}

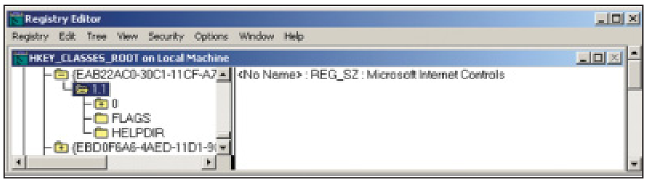
Figure 2: Registry Editor showing InternetExplorer.Application CLSID.


Registry Editor
Registry Edit Tree View Security Options Window Help
HKEY_CLASSES_ROOT on Local Machine

{0002DF01-0000-0000-C00
LocalServer32
ProgID
Programmable
TypeLib
VersionIndependentPr
{0002E005-0000-0000-C00

<No Name> : REG_SZ : {EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}

Figure 3: InternetExplorer.Application type library CLSID.


Registry Editor
Registry Edit Tree View Security Options Window Help
HKEY_CLASSES_ROOT on Local Machine

{EAB22AC0-30C1-11CF-A7
1.1
0
FLAGS
HELPDIR
{EBD0F6A6-4AED-11D1-9

<No Name> : REG_SZ : Microsoft Internet Controls

Figure 4: InternetExplorer.Application type library name.

something slightly more interactive. Web pages are a kind of GUI, and HTML is much simpler than most GUI toolkits—which is where *HTTP::Daemon* comes in.

The example program, asparagus.pl (see Listing 2), is an extremely simple server that shows the time as constructed by the *Acme::Time::Asparagus* module available via CPAN or the ActiveState PPM repository. It's just a bit of fluff, but it shows the basic mechanism I've used on a number of occasions. It's a relatively simple way to handle forms and/or different pages, providing a quick-and-dirty GUI without a lot of hassle.

The use of *Win32::OLE* to start Internet Explorer when the server is executed is just a small tweak to save some mouse clicks. Since the server stays running, the sleep statement is not needed. Usually, I add a form or link that can be used to shut down the server, making the whole thing kind of self contained.

Other Things You Can Do With OLE

Starting Internet Explorer, while simple and easy to illustrate, is not exactly the best demonstration of the power of OLE, in the sense that the object model is shallow and not really standard. Other real-world examples are much more impressive. It is possible to drive Word, Excel, PowerPoint, and virtually any OLE-enabled application. I've created Excel documents, used Word to convert Word documents to HTML, and completely disassembled PowerPoint documents using this mechanism.

The object model for fully OLE-enabled functional applications is more robust. An application has multiple documents, which are objects in their own right. Each document then contains many hierarchical levels of objects, each of which may have its own properties, methods, and events. The internal structure of the document becomes easily accessible.

The result is that Perl, the perennial glue program, becomes that much more useful. Using OLE, you can harness an even greater variety of applications to ever more peculiar purposes.

TPJ

(Listings are also available online at http://www.tpj.com/source/.)

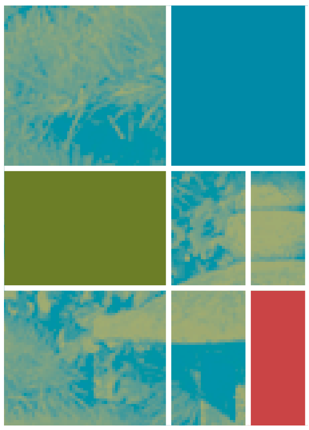
Listing 1

```
use     strict;
use     warnings;

use     Win32::TieRegistry;

die "usage:  typeLib <classname>\n"
    unless @ARGV;

my  $classes = $Registry->{"Classes\\"};

die "* Unable to get HKEY_ROOT_CLASSES from Registry\n"
    unless UNIVERSAL::isa($classes, 'Win32::TieRegistry');

for my $class (@ARGV) {
    print "$class:\n";

    for my $key (keys %$classes) {
        next unless $key =~ /$class/;

        my  $clsid = $classes->{$key}->{"CLSID\\\\"};

        $key =~ s/\\+$//;
        print "  $key:\n";

        unless ($clsid) {
            print "    * No class ID\n";
            next;
        }

        print "    CLSSID:   $clsid\n";

        my  $typid = $Registry->{"Classes\\CLSID\\$clsid\\TypeLib\\\\"};

        unless ($typid) {
            print "    * No type library ID\n";
            next;
        }

        print "    TypeLib:  $typid\n";

        my  $typlb = $Registry->{"Classes\\TypeLib\\$typid\\"};

        print "    Library:  $typlb\n";

        unless ($typlb) {
            print "    * No type library key\n";
            next;
        }

        print "      $_ => $typlb->{$_}->{'\\'}\n" for keys %$typlb;
    }
}
```

Listing 2

```
use     strict;
use     warnings;
use     Acme::Time::Asparagus;
use     HTTP::Daemon;
use     Win32::OLE;

my  $port = 9183;   # just pick one

# Generate the HTML page with the vegetable time:
sub timePage
{
    my  $conn = shift;
    $conn->send_response(<<VEGETABLE);
200 OK

<html>
<body>
  <h2>Vegetable Clock</h2>
  <p><em>At the tone the time will be:</em>&nbsp;
     <strong>@{[veggietime]}</strong></p>
</body>
</html>
VEGETABLE
}

# Main program, start Internet Explorer first:
my  $explorer = new Win32::OLE('InternetExplorer.Application', 'Quit')
                or die "Unable to create OLE object:\n  ",
                       Win32::OLE->LastError, "\n";

$explorer->Navigate("http://localhost:$port/", 3);

# Start up a tiny, single-function HTTP server:
my  $daemon = new HTTP::Daemon(LocalPort => $port)
              or die "Unable to create daemon\n";

while (my $conn = $daemon->accept) {
    while (my $rqst = $conn->get_request) {
        timePage($conn) if $rqst->method eq 'GET';
    }
    $conn->close;
    undef $conn;
}
```

TPJ



Cooking with Maypole, Part II

Simon Cozens

We began our gastronomic adventures with Maypole last time, when we constructed a recipe collection and a way to index and investigate the current state of food stocks in the house. Now we're going to combine the two concepts, and search for recipes that fit what we've got available to eat.

First, though, we'll take a brief look at how Maypole works and what it actually does.

How Maypole Works

In Part I, we saw Maypole primarily in terms of putting an interface onto an existing data structure and applying templates to this. However, to think of it like this is to miss the flexibility and extensibility of Maypole as a web application framework.

Perhaps the best way to think of Maypole is as a tool for mapping a URL onto an "action," where an action is specified as a method call and a template. So the URL "/recipe/view/12" is asking for the *view* action to be performed on a *recipe* class, with argument "12." Practically, this means that the *view* method will be called on the *Larder::Recipe* class on the object representing the 12th row in the table, and then the *view* template used to display the results.

This process is carried out by the gradual fleshing out of a "Maypole request object"—analogous to an Apache *request* object but at a much higher level. As well as containing the means to communicate with the web server (such as an *Apache::Request* object), it begins with the configuration and some idea of the path requested: /recipe/view/12.

Next, it decomposes the path down to its components:

```
{ table => "recipe",
  action => "view",
  args => [ 12 ]
}
```

Then it associates the table with the *Larder::Recipe* class and calls the *view* method. Finally, the output from the templating stage gets added to the request, and the request is sent back to the front-end *Maypole* class (usually *Apache::MVC* or *CGI::Maypole*) for eventual output to the browser.

Avoiding Rotten Tomatoes

We'll begin our extension of the Larder application by adding an action to the contents of our larder that are in imminent danger of going off. To help us do this, we'll write a utility method in the *Larder::Contents* class called *ripe_ food*, which returns all the objects that need to be eaten. This is pure *Class::DBI* for the time being.

First, SQL dates are a bit of a pain to do anything sensible with, so we have *Class::DBI* automatically inflate them to *Time::Piece* objects:

```
Larder::Contents->has_a(use_by => 'Time::Piece',
     inflate => sub { Time::Piece->strptime(shift, "%Y-%m-%d") },
     deflate => 'ymd',
       );
```

Now, every time we call *use_by*, we get a *Time::Piece* object. This doesn't really affect our display because *Time::Piece* stringifies nicely.

We can now look for those *Contents* objects that have a *use_by* date under five days away:

```
use Time::Seconds;
use Time::Piece;

sub ripe_food {
  my $class = shift;
  my $deadline = localtime + 5 * ONE_DAY;
  grep { $_->use_by <= $deadline } $class->retrieve_all;
}
```

(If more efficiency is required, we could do the searching in the SQL by using *Class::DBI::AbstractSearch*, but my larders aren't big enough to warrant it.)

To turn this method into an action that can be called from the Web, we need to create an "Exported" method that places these objects into the Maypole request object:

```
sub must_eat :Exported {
  my ($self, $r) = @_;
  $r->{objects} = [ $self->ripe_food ];
}
```

We create a template in contents/must_eat, and we can now view these items from the URL /contents/must_eat. In order to suggest recipes that use these up, we need to link ingredients to recipes.

Categories and Ingredients

Our data loader in Part I looked like this:

```
use Larder;
use XML::Simple;
use File::Slurp;
```

Simon is a freelance programmer and author, whose titles include Beginning Perl *(Wrox Press, 2000) and* Extending and Embedding Perl *(Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumpking. Simon can be reached at simon@simon-cozens.org.*

```
for my $recipe (<xml/*>) {
  my $xml = read_file($recipe);
  my $structure = XMLin($xml, ForceArray => 1)->{recipe}->[0];
  my $name = $structure->{head}->[0]->{title}->[0];
  my @ingredients = @{$structure->{ingredients}[0]{ing}};
  my @cats = @{$structure->{head}[0]{categories}[0]{cat}};
  Larder::Recipe->find_or_create({
    name => $name,
    xml => $xml
  });
}
```

Now we're going to extend this, and our *Larder* class, to support the linkages between recipes and categories, and recipes and their ingredients. *Class::DBI* makes it easy for us to do this: We tell it the name of the accessor we want, the name of the mapping class, and the name of the accessor in that class that returns what we want.

In our case, we first have to tell *Class::DBI* about the relationship between the ingredient table and the food table:

```
Larder::Ingredient->has_a(food => "Larder::Food");
```

and vice versa:

```
Larder::Food->has_many(ingredients => "Larder::Ingredient");
```

So, in our case, we want the *Larder::Recipe* class to have an accessor called *ingredients*, which uses *Larder::Ingredient* to get a list of ingredients for a recipe and calls *food* on each one to return a *Larder::Food* object. The code for that looks like this:

```
Larder::Recipe->has_many(ingredients => [ Larder::Ingredient => "food" ]);
```

Similarly, we can get from a *Larder::Food* object to the recipes that contain it:

```
Larder::Ingredient->has_a(recipe => "Larder::Recipe");
Larder::Food->has_many(recipes => [ Larder::Ingredient => "recipe" ]);
```

And all the same for categories. Of course, we could do this a little more easily by using my module *Class:DBI::Loader::Relationship*. This uses *Class::DBI::Loader* (which Maypole also uses; this is no coincidence) to express the relationships between classes in more natural terms. It's not as powerful, but it is easier to remember:

```
Larder->config->{loader}->relationship(
  "A recipe has categories via categorizations"
);
```

With our relationships set up, we can tell the loader to associate a recipe with its categories and ingredients:

```
for (@categories) {
  my $cat = Larder::Category->find_or_create({ name => $_ });
  $recipe->add_to_categories({ category => $cat });
}

for (@ingredients) {
  my ($ing, $amt) = ($_->{item}[0], $_->{amt}[0]);
  my $ingredient = Larder::Food->find_or_create({ name => $ing });
  my $quantity = $amt->{qty}[0]. " " . $amt->{unit}[0];
  $recipe->add_to_ingredients({ food => $ingredient,
                  quantity => $quantity   });
  }
}
```

Now we can create a template that suggests some recipes to go with our moribund ingredients:

```
<h1> You need to use up some food! </h1>

[% FOR content = contents %]
  <h2> [% content.food.name %] </h2>

  <P> Needs to be used up by [% content.use_by %] </P>

  <P>
  Some recipes you could make with this:
  <P>
  <UL>
     [% FOR recipe = content.food.recipes %]
    <LI> <A HREF="/recipe/view/[% recipe.id %]"> [% recipe %]</A>
    [% END]
  </UL>
[% END %]
```

Of course, these aren't necessarily the best recipes for the job; ideally, we're going to find the recipes that use up as many of the ingredients as possible. We can't do this with a plain database search. My initial plan was to gather up all the potential recipes, then score them based on the ingredients that they use and the immediacy of getting rid of the ingredient.

But then an even more interesting technology came along.

Plucene

Plucene is a Perl port of the Java Lucene search engine, a Jakarta project (see http://jakarta.apache.org/lucene/ for more information). Rather than a standalone search tool, it is a library with which you can construct your own indexing and searching tools. The easiest interface to it is through the Perl module *Plucene::Simple*, which we'll use for indexing the recipes.

Plucene works in terms of "documents," which are a little like pages in a book. When you're building an index to a book, the index will relate a word or phrase from the page (the index term) to a page number—it doesn't directly relate the word to the entire contents of the page, or the index would be exponentially longer than the book itself! Instead, the reader is responsible for turning the page number into the original page contents. If we're indexing a book with Plucene, we might create documents like this:

```
@documents = (
1 => {
  chapter_title => "Preface",
  text => "We have many emotions as we ..."
},
3 => {
  chapter_title => "Ethos",
  text => "We have made fundamental assumptions  ..."
},
...
);
```

We create a *Plucene::Simple* object that represents the index:

```
use Plucene::Simple;
my $index = Plucene::Simple->open("/home/simon/ow_book/index");
```

And we can add the documents:

```
$index->add(@documents);
$index->optimize;
```

The optimization step defragments the index once we've finished adding a lot of data at once. To run a search, we open the index again and call its search method:

```
use Plucene::Simple;
my $index = Plucene::Simple->open("/home/simon/ow_book/index");
my @results = $index->search("we");
```

Now a search for "*we*" would return "*1*" and "*3*," and we could narrow down our search by looking for "we chapter_title:Ethos," which would return only page 3. It's assumed that we have an easy way of turning the ID, "*3*," into the full text of the book's page.

However, we're not indexing a book, but a set of recipes. In this case, our documents are going to look like this:

```
52 => {
  title => "Aioli",
  categories => "Salads Condiment Classic",
  ingredients => "Garlic Mayonnaise",
}
```

where "*52*" is the ID of the recipe in the recipe table. We can, of course, look this up again by doing *Larder::Recipe->retrieve(52)*. We're going to skip the process of indexing the directions part of the recipe because we're only interested in searching for particular ingredients at the moment. So, once again, we edit our recipe loader script, which currently has this at the end of the loop:

```
Larder::Recipe->find_or_create({
  name => $name,
  xml => $xml
});
```

We need to keep hold of that recipe's ID and build up our hash of things to index:

```
my $recipe = Larder::Recipe->find_or_create({
    name => $name,
    xml => $xml
  });
my $hash = {
  title => $name,
  categories => (join " ", @categories),
  ingredients => (join " ", map { $_->{item}[0] } @ingredients)
};
$index->add( $recipe->id => $hash );
```

Finally, we optimize the index outside of the loop once everything has been added:

```
$index->optimize;
```

Now we have, in addition to our database of recipes, an index by which we can look for entries in that database. How does this help us find good recipes for food that's going off?

Locating the Best Recipe

Once our index has been built up, we can now start searching for recipes by their ingredients, and Plucene automatically makes sure that those recipes that match "better"—that is, use more of the ingredients—are returned first. So, for instance, if we have some carrots, bacon, and mushrooms to use up, we can create a simple test search script like this:

```
use Plucene::Simple;
use Larder;
my $index = Plucene::Simple->open("pl_index");
for ($index->search("carrots bacon mushrooms")) {
  my $r = Larder::Recipe->retrieve($_);
  print $r->name,"\n", join ", ", map { $_->name } $r->ingredients;
  print "\n\n";
}
```

And we'll be given a list of recipes that contain any of those ingredients, but starting with the best matches:

```
24 Hour Vegetable Salad
Iceberg lettuce, Mushrooms, Peas, Carrots, Egg whites, Bacon,
Cheese, Fat-free mayonnaise, Lemon Juice

Beef Burgundy
Mushrooms, Onions, Butter, Bacon, Sirloin Steaks, Flour,
Burgundy, Beef Broth, Bay leaf, Garlic, Ground Thyme, Carrots,
Salt And Pepper, Noodles, Chopped Parsley

Bacon Supper Snack
Gammon, Tomatoes, Stuffing, Butter, Mushrooms, Soft White
Bread Crumbs, Salt and pepper, Mixed Herbs, Egg

All-In-One-Breakfast
Whole Wheat Bread, Butter, Mushrooms, Tomatoes, Cheese, Bacon

...
```

The last two recipes shown here (and there were many more) don't contain all three ingredients, but do contain two; as we carry on down the list, we get less and less specific.

What we're doing, then, is using the built-in scoring techniques of a standard search engine to find the best recipes for us—web search engines are all about finding the most appropriate pages relating to the user's terms; we're using the same mechanism to find the most appropriate lunch relating to what's in the cupboard.

Now we can turn our "must eat" page into a page that helps us search for the best recipes to eat:

```
sub must_eat :Exported {
  my $index = Plucene::Simple->open("recipe_index");
  my @ripe = $self->ripe_food;
  $r->{objects} = \@ripe;
  my @terms = map { '"'. $_->name. '"' } @ripe;
  my @results = map { Larder::Recipe->retrieve($_) }
      $index->search(join " ", @terms);
  $r->{template_args}{recipes} = \@results;
  $r->{template_args}{highlight} = { map { $_->name => 1 } @ripe };
}
```

Notice that we surround our ingredient names in double quotes—Plucene understands the concept of phrase matches, as one would expect from a search engine: "*fish fingers*" searches for recipes containing fish fingers, whereas fish fingers (no quotes) will search for recipes that make use of both fish and fingers. (One can only hope that neither of those searches will turn up any hits in your recipes.) When we've retrieved the results from our search engine and turned them into recipe objects, we add them to our set of arguments to the template. We also add a hash of the ingredient names we're looking for—this will help us highlight the ingredients when we're producing a summary of the recipe. So, for instance, we want our page to look like Figure 1.

The associated template would go as follows:

```
<h2> You need to use up some food! </h2>
<P>
The following food is getting a bit ripe:
</P>
<UL>
```

```
[% FOR content = contents %]
  <LI> [% content.food.name %]
[% END %]
</UL>

<H2> Suggested recipes </H2>
<P>
These recipes will help you use up those ingredients:
</P>
```

Now we look at each recipe in our search results:

```
[% FOR recipe = recipes %]
  <h3> <A HREF="/recipe/view/[%recipe.id%]"> [% recipe.name %] </a> </h3>
  Requires:
  <p>
  [% FOR ingredient = recipe.ingredients;
     SET name = ingredient.food.name;
```

And now for each ingredient, we can show their names and check whether or not to highlight them:

```
     '<span class="searchresult">' if highlight.$name;
     name;
     '</span>' if highlight.$name;
     END;
  %]
  </p>
[% END %]
```

Hooray—now we not only know which recipes will use up the dying ingredients, but also which ones will include the most of them at once. There's one final touch we can add to our application before we head off to the kitchen—a sense of urgency.

If something needs to be used by today, we want a recipe that uses it today. Let's change *ripe_food* so that it returns us a hash of ingredients and a score representing the need to eat them:

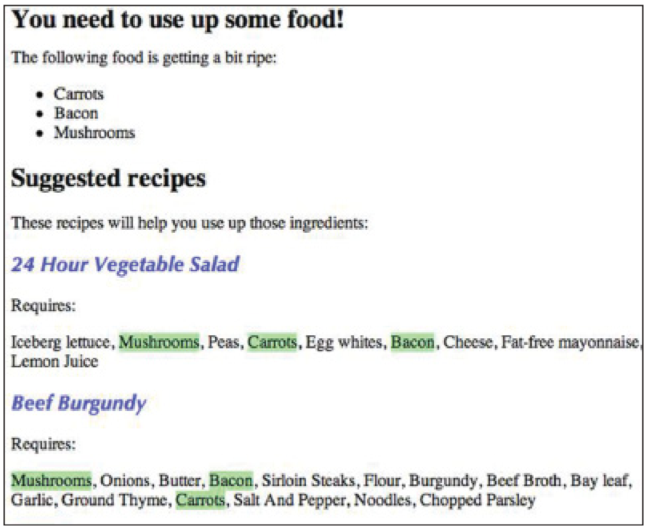
```
sub ripe_food {
  my $class = shift;
  my $deadline = localtime + 5 * ONE_DAY;
  map { 5 - int(($deadline - $_->use_by) / ONE_DAY) }
  grep { $_->use_by <= $deadline } $class->retrieve_all;
}
```

Now if we have some cheese that really needs to be eaten today and some ham that has two days to go, we get:

```
( Cheese => 5, Ham => 3 )
```

We want Plucene to score up recipes that contain cheese, relative to those that contain ham. We can do this using a boost factor in the search term. Plucene allows us to search for *"Cheese"*^5 *"Ham"*^3—now it tries to find recipes that have both ham and cheese in it, then those that contain cheese, then those that contain ham. With a list of 10 or 20 ingredients to get rid of, this is more or less guaranteed to give us recipes that use up the widest range of the most desperate ingredients first, giving us the most economical ways to clean out our cupboards. We'll need to modify the *must_eat* action to understand the list returned:

```
sub must_eat :Exported {
  my $index = Plucene::Simple->open("recipe_index");
  my @ripe = $self->ripe_food;
  $r->{objects} = [];
  my @terms;
  while (my ($obj, $score) = splice(@ripe, 0, 2)) {
    push @{$r->{objects}}, $obj;
```



You need to use up some food!

The following food is getting a bit ripe:

- Carrots
- Bacon
- Mushrooms

Suggested recipes

These recipes will help you use up those ingredients:

24 Hour Vegetable Salad

Requires:

Iceberg lettuce, Mushrooms, Peas, Carrots, Egg whites, Bacon, Cheese, Fat-free mayonnaise, Lemon Juice

Beef Burgundy

Requires:

Mushrooms, Onions, Butter, Bacon, Sirloin Steaks, Flour, Burgundy, Beef Broth, Bay leaf, Garlic, Ground Thyme, Carrots, Salt And Pepper, Noodles, Chopped Parsley

Figure 1: Suggesting recipes to use up food.

```
    push @terms, '"'. $obj->name. '"^'. $score;
    $r->{template_args}{highlight}{$obj->name}++;
  }

  my @results = map { Larder::Recipe->retrieve($_) }
      $index->search(join " ", @terms);
  $r->{template_args}{recipes} = \@results;
}
```

We're using a list of pairs instead of a real hash because the "keys" are objects, and Perl doesn't let us use objects as hash keys—they store just fine, but as they are stored, they get stringified and we can't use them as objects again when we retrieve from the hash. The technique of using

```
while (my ($key, $value) = splice(@list, 0, 2)) {
```

where you'd normally expect

```
while (my ($key, $value) = each %hash) {
```

is quite a common one you can use where you'd like to use objects as hash keys.

With this in place, Plucene scores each ingredient according to its freshness, in a nice simple way that frees us from having to think up a complicated algorithm to do the job. That's code reuse! And now, what's in the fridge?

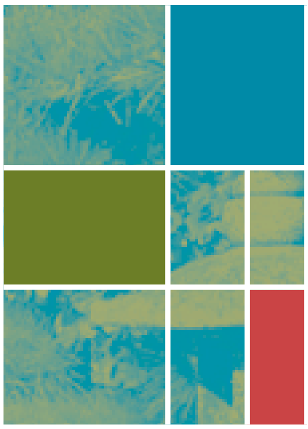
Happy Cooking

I hope you've enjoyed our two-part foray into cooking with Perl; we've covered quite a lot of ground on the way. This time we've focused particularly on Maypole and showed how to turn it from a simple front-end to databases into a web application framework on which to base more complex applications. We've also taken a look at Plucene, a pure Perl search engine that allows us to index and search through all kinds of data—including recipes!

If you want to find out more about Maypole, there's a growing set of documentation at http://maypole.simon-cozens.org/docs/, including a large manual with several examples of real-life applications. Plucene can be downloaded from CPAN, and there's a longer introduction to it at http://www.perl.com/pub/a/2004/02/19/plucene.html. Finally, for a load of great recipes in RecipeML, try http://dsquirrel.tripod.com/recipeml/indexrecipes2.html.

Bon appetit!

TPJ



Making the Most of *Exporter* and *use*

Randal L. Schwartz

Unless this is your first day of programming in Perl, I think it would be safe to say that you've used *use* at least once, and probably with some module that imports subroutines into your current package namespace. Let's take a look at precisely how that works.

As an example, let's look at the *LWP::Simple* module. By default, having:

```
use LWP::Simple;
```

at the top of my program means I can safely invoke *get*, *head*, *getprint*, *getstore*, and *mirror*, as well as a series of *HTTP::Status* constants, treating them as if I had written the program directly. I can narrow down that namespace pollution with a direct specification of things to import:

```
use LWP::Simple qw(mirror RC_OK RC_NOT_MODIFIED);
```

And now I get only the *mirror* routine and its two likely return statuses.

These invocations act as if I had included a *BEGIN* block wrapped around a *require* operation followed by a method call. For example, that latter invocation looks like this:

```
BEGIN {
  require LWP::Simple;
  LWP::Simple->import(qw(mirror RC_OK RC_NOT_MODIFIED));
}
```

Notice first that because it's a *BEGIN* block, we're talking about an operation that happens at compile time, not deferred until runtime. As the package name of *LWP::Simple* is fed to *require*, it is converted into an appropriate directory/subdirectory path for the architecture running the program, with a .pm extension automatically tacked on, as if we had said:

```
require "LWP/Simple.pm";
```

on a UNIX architecture machine.

Randal is a coauthor of Programming Perl, Learning Perl, Learning Perl for Win32 Systems *and* Effective Perl Programming, *as well as a founding board member of the Perl Mongers (perl.org). Randal can be reached at merlyn@stonehenge.com.*

Because we're using *require*, the *@INC* path is automatically consulted to locate the file. By default, this includes the system directories where Perl was installed and the current directory. We can alter the *@INC* path permanently by recompiling Perl, or temporarily through a combination of setting the *PERL5LIB* environment variable, adding a *-I* command-line switch, or preceding this use with either a *use lib* setting or something else that alters *@INC* at compile time.

Also, because we're using require, the .pm file must have a *true* value as the last expression evaluated. This is typically provided with a simple:

```
1;
```

at the end of the file; although, if you were being perverse, you could change that to:

```
"this line intentionally left true";
```

or something else as long as it was *true*.

Up to this point, we simply have a *require* operator that runs at compile-time instead of runtime, with a consistent name-management scheme to find the file. But the final step of *use* is the implicit method call, and this is where we get our subroutine name imports.

A typical module brought in with *use* will define subroutines in the package namespace appropriate to that module; for example, *LWP::Simple* defines *LWP::Simple::get*, *LWP::Simple::mirror*, and so on. The method call to import within *LWP::Simple*'s class causes these names to be aliased to their same name within the invoker's namespace (typically *main*). This is the result of *LWP::Simple* inheriting from *Exporter*, a core Perl module that provides a relatively flexible and smart import method to handle the aliasing.

This means that *LWP::Simple*, like most modules, begins with something like:

```
package LWP::Simple;
use base qw(Exporter);
```

or equivalently, but backward compatible for fairly old Perl versions:

```
package LWP::Simple;
use vars qw/@ISA/;
```

```
require Exporter;
@ISA = qw(Exporter);
```

But now we've just moved the problem around a bit. The author of *LWP::Simple* doesn't have to write *import*, but *Exporter::import* still needs to know what to do. The *Exporter::import* routine looks at the *@EXPORT* and *@EXPORT_OK* variables in the current package to determine the permitted exports.

For example, *LWP::Simple* starts with something like:

```
use vars qw(@EXPORT @EXPORT_OK);
@EXPORT = qw(get head getprint getstore mirror);
@EXPORT_OK = qw($ua);
```

The final step of use is the implicit method call, and this is where we get our subroutine name imports

This means that, by default, the five subroutines are imported into the caller's namespace. However, any specific list of imports must come from all six names (everything in both *@EXPORT* and *@EXPORT_OK*). In truth, the exports of *HTTP::Status* are also added to *@EXPORT*, but let's ignore them for a second.

So, if I ask for:

```
use LWP::Simple;
```

I get the default list (everything in *@EXPORT*). If I ask for just *get* and *mirror* explictly:

```
use LWP::Simple qw(get mirror);
```

then that's all I get. If I ask for something not in the list:

```
use LWP::Simple qw(head tail);
```

then *Exporter::import* aborts because *tail* is neither *@EXPORT* nor *@EXPORT_OK*.

We can completely skip the invocation of import with an empty set of parenthesis:

```
use LWP::Simple ();
```

The *$ua* member of *@EXPORT_OK* here is actually a package variable, initially defined as *$LWP::Simple::ua*, but available for import into my namespace. As a rule of thumb, it's generally better to export behavior (subroutines or objects) rather than data (variables) because once your data format is "out there," it's hard to change it for future releases.

What if we want "all the default, plus this one" or "all the default, except for that one"? It turns out that *Exporter::import* is actually fairly flexible. We can use:

```
use LWP::Simple qw(:DEFAULT $ua);
```

to mean "all the default and *$ua*." Similarly, we can use:

```
use LWP::Simple qw(:DEFAULT !getprint !getstore);
```

to mean "all the default without these two listed names." Additionally, we can use regular expressions, such as:

```
use LWP::Simple qw(:DEFAULT !/^get/);
```

to mean "all the default without any name beginning with get." An expression that begins with *!* appearing at the beginning of the list implies a *:DEFAULT* ahead of it, so we can write that last one more simply as:

```
use LWP::Simple qw(!/^get/);
```

We could even ask for:

```
use LWP::Simple qw(/^/);
```

to mean "everything that can possibly be exported" (because the regular expression of */^/* matches all possible strings).

We can define additional shortcut keywords such as *:DEFAULT* if subsets of our export list are frequently desired. For example, if the *get* routines are frequently selected or deselected as a group, we can group them as the *:GET* group like so:

```
package LWP::Simple;
use base qw(Exporter);

use vars qw(@EXPORT @EXPORT_OK %EXPORT_TAGS);
@EXPORT = qw(get head getprint getstore mirror);
@EXPORT_OK = qw($ua);
%EXPORT_TAGS = ("GET" => [qw(getprint getstore)]);
```

And now we can include them or exclude them easily:

```
use LWP::Simple qw(:GET);
```

which brings in only *getprint* and *getstore*, or:

```
use LWP::Simple qw(!:GET);
```

which gives us the default list, minus *getprint* and *getstore*, or even:

```
use LWP::Simple qw(/^/ !:GET);
```

which is all possible exports, except for *getprint* and *getstore*.

If */^/* is a bit mystical to explain to your users, you can help them along with an *:ALL* tag:

```
%EXPORT_TAGS = (
  "ALL" => [@EXPORT, @EXPORT_OK],
  "GET" => [qw(getprint getstore)]
);
```

which permits us to say:

```
use LWP::Simple qw(:ALL);
```

and get all possible exports and:

```
use LWP::Simple qw(:ALL !:GET);
```

to get everything except the *getprint* and *getstore* routines.

You can also go the other way, defining your symbols primarily in *%EXPORT_TAGS*, and then adding them to the *@EXPORT* and *@EXPORT_OK* variables:

```
package My::Package;
use base qw(Exporter);
use vars qw(%EXPORT_TAGS);
```

Another example of a very complex import *routine is the core CGI module*

```
%EXPORT_TAGS = (
  "CORE" => [qw(core_enable core_disable)],
  "FIRE" => [qw(halt_and_catch_fire)],
  "VARS" => [qw($core_heat $core_fire)],
);
Exporter::export_tags('CORE', 'FIRE'); # default list
Exporter::export_ok_tags('VARS'); # optional list
```

Another feature of *Exporter* and *use* is the ability to check version numbers. If we include a version number before the (optional) import list, the *$VERSION* variable of the package will be checked to ensure that it is not smaller than the requested version. For example, if *My::Package* is defined as:

```
package My::Package;
use base qw(Exporter);
use vars qw($VERSION @EXPORT @EXPORT_OK);
$VERSION = 1.02;
@EXPORT = ...;
@EXPORT_OK = ...;
```

and I try to invoke it with:

```
use My::Package 1.05;
```

I will get a complaint that the version number is not sufficient. This feature is provided by the *Exporter::require_version* method, and compares the numbers as floating-point values. If you want a different sort of comparison, you should write your own.

Because *Exporter* exports a half-dozen routines into the current namespace, you might want to narrow it down to just *import*:

```
package My::Package;
use Exporter qw(import);
```

and now we don't have to worry about other name collisions. However, we just broke version checking, as described above, so be careful with this.

So far, we've seen most of the standard things you can do with *Exporter::import*. But we can also write our own *import* routine to do nonstandard tasks. For example, the *use lib* module takes the import list and adds it to *@INC*. The code is roughly something like:

```
package lib;
sub import {
  unshift @INC, @_;
}
```

which simply prepends the arguments passed to *lib::import* to the *@INC* path. (The actual *lib* module also deals with architecture-specific path additions.)

Another example of a very complex *import* routine is the core CGI module, which supports a version of colon-prefixed keywords as well as some other flags to trigger various actions.

You can, of course, create your own import routines as well:

```
package My::Package;

sub import {
  my $filename = shift;
  open MY_LOG, ">>$filename" or die;
}
```

and now use this as:

```
use My::Package qw(/my/log/file);
```

The indicated filename will be opened as the (package based) filehandle for logging.

Well, I think this about wraps it up for now. Hopefully, you've seen a few new ways to use your import list. Until next time, enjoy!

TPJ

Apache, MySQL, And PHP Weekend Crash Course

Jack J. Woehr

It's an interesting concept. Spend the weekend, from Friday evening through Sunday afternoon, in thirty 30-minute sessions on a subject and master it. In this case, the subjects are: the Apache web server, the MySQL database, the PHP hypertext preprocessor, and the integration of the three. *Apache, MySQL, and PHP Weekend Crash Course is* a structured learning seminar in a book. The author, Steven Schafer, is humane enough to recommend that the reader "take some time to relax after each session to let the information sink in and prepare for the next session." Kind of like the SATs, but the SATs are shorter.

Apache, MySQL, and
PHP Weekend Crash Course
Steven M. Schafer
John Wiley & Sons, 2004
504 pp., $24.99
ISBN 0-7645-4320-2

The book targets a reader who is familiar with programming and web serving but may be utterly novice to the specific tools being taught. Everything is laid out for that reader, starting with the web sites from which to download the tools and pointers for choosing versions and avoiding pitfalls in the download-and-installation process. Windows and Linux are the two targeted operating systems, but the structure is such that a reasonably savvy reader could "translate" the advice for BSD, Solaris, OS X, and the like. The book is artfully planned, designed, and laid out so that the minimum of visual confusion and page bloat are incurred in the presentation of the obligatory Windows screen shots. The best aspects of this book are the ordering and pacing of the subject matter, and the tasteful presentation, leaving Windows readers feeling fully at home while not drowning the Linux reader in Windows details.

Per the prescribed program, Friday evening is getting and installing the three components and learning to configure Apache with PHP. Saturday morning is more Apache configuration, setting up MySQL and security issues with the two. Saturday afternoon is lots of MySQL and PHP. Saturday evening is heavy HTML and PHP. Sunday morning is good practice with PHP/MySQL and debugging the two. Sunday afternoon is projects. The deuce is in the details; I've just offered here the summary of a summary of a summary of the 400+ page book. Ambitious as it is, *Apache, MySQL, and PHP Weekend Crash Course* is a masterfully executed encapsulation that epitomizes the problem domain.

That said, the promotional premise of this publication—a Monday presentation after the weekend brainbath—is absurd. *Apache, MySQL, and PHP Weekend Crash Course* is of great value if this most popular scripting-webserver-with-database vertical free-software stack is of critical interest to you. You will indeed learn the overall concept of why you would wish to integrate these three tools, along with the basics of installation, configuration, and programming of the resulting system. You will, however, most likely spend a good deal more calendar time than a weekend and more than 30 hours screen time working through the structured learning. It's almost plausible that one could allow one's intellect to be primed with this amount of information in the source of a weekend's forced march through the book. If, however, the material is truly novel to the reader, there's going to be a lot of backtracking to half-remembered points and many long pauses just to let the eyes adjust and to internalize even that beginner's portion of the deep subject matter that this book serves up. Okay, in a week or two, you might be ready for that Monday presentation, but after the presentation, the real learning begins. SQL in 30 minutes? I've been struggling with it since 1996. Apache and MySQL? Each has its own bookshelf. PHP? Welcome to learning the ins and outs of another text-processing language vastly more feature-laden than m4 and with the concomitant loss of theoretical focus.

It comes down to how you define learning: as an exhausting cram for the test review at the end of each section or as genuinely absorbing both the broad concept and the particular detail. Those skilled at the former ace the API-specific job interviews, then spend a lot of time wandering across the terrain writing, unwriting, and rewriting oft-tangled code; those enamored of the latter sort of learning have the firmer grasp and the steadier hand on the keyboard—they can't be pushed faster than they can tread surely and also tend to state their qualifications more modestly and diffidently in the interview. I've hired both types of programmers: I admire the energy of the crammers and treasure the depth of the absorbers. It takes all kinds to keep the team on its toes.

The book's companion web site is http://www.wiley.com/compbooks/schafer/. The site presents the samples files, errata (yes, there are a few), and a practice test done in Flash. The Wiley product site for the book, presenting the table of contents, an excerpt in PDF form, and reviews of the book is found at http://www.wiley.com/WileyCDA/WileyTitle/productCd-0764543202.html.

TPJ

Jack J. Woehr is an independent consultant and team mentor practicing in Colorado. He can be contacted at http://www.softwoehr.com/.

Source Code Appendix

Marc M. Adkins "Controlling Internet Explorer Using *Win32::OLE*"

Listing 1

```
use     strict;
use     warnings;

use     Win32::TieRegistry;

die "usage:  typeLib <classname>\n"
    unless @ARGV;

my  $classes = $Registry->{"Classes\\"};

die "* Unable to get HKEY_ROOT_CLASSES from Registry\n"
    unless UNIVERSAL::isa($classes, 'Win32::TieRegistry');

for my $class (@ARGV) {
    print "$class:\n";

    for my $key (keys %$classes) {
        next unless $key =~ /$class/;

        my  $clsid = $classes->{$key}->{"CLSID\\\\"};

        $key =~ s/\\+$//;
        print "  $key:\n";

        unless ($clsid) {
            print "    * No class ID\n";
            next;
        }

        print "    CLSSID:   $clsid\n";

        my  $typid = $Registry->{"Classes\\CLSID\\$clsid\\TypeLib\\\\"};

        unless ($typid) {
            print "    * No type library ID\n";
            next;
        }

        print "    TypeLib:  $typid\n";

        my  $typlb = $Registry->{"Classes\\TypeLib\\$typid\\"};

        print "    Library:  $typlb\n";

        unless ($typlb) {
            print "    * No type library key\n";
            next;
        }

        print "      $_ => $typlb->{$_}->{'\\'}\n" for keys %$typlb;
    }
}
```

Listing 2

```
use     strict;
use     warnings;

use     Acme::Time::Asparagus;
use     HTTP::Daemon;
use     Win32::OLE;

my  $port = 9183;   # just pick one

# Generate the HTML page with the vegetable time:
sub timePage
{
    my  $conn = shift;
    $conn->send_response(<<VEGETABLE);
200 OK

<html>
<body>
  <h2>Vegetable Clock</h2>
  <p><em>At the tone the time will be:</em>&nbsp;
     <strong>@{[veggietime]}</strong></p>
</body>
</html>
VEGETABLE
}
```

```
# Main program, start Internet Explorer first:
my  $explorer = new Win32::OLE('InternetExplorer.Application', 'Quit')
                 or die "Unable to create OLE object:\n  ",
                        Win32::OLE->LastError, "\n";

$explorer->Navigate("http://localhost:$port/", 3);

# Start up a tiny, single-function HTTP server:
my  $daemon = new HTTP::Daemon(LocalPort => $port)
               or die "Unable to create daemon\n";

while (my $conn = $daemon->accept) {
    while (my $rqst = $conn->get_request) {
        timePage($conn) if $rqst->method eq 'GET';
    }
    $conn->close;
    undef $conn;
}
```

TPJ