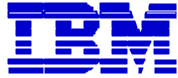
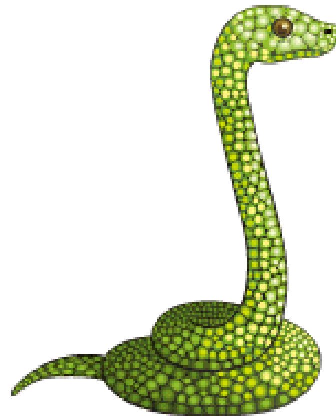


The Camel and the Snake
or "Cheat the Prophet"

Open Source Development with Perl, Python and DB2®



Bill Hilf
Sr. I/T Architect
GSMB Linux Technical Lead
mailto:billhilf@us.ibm.com

Dominique Cimafranca
I/T Specialist
SMB Asia-Pacific
mailto:cimafran@ph.ibm.com

Disclaimers and Trademarks

Copyright © 2002 IBM Corporation. All rights reserved.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS." – The IBM CORPORATION DOES NOT MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

DB2, DB2 Universal Database, IBM, MQSeries, OS/390, and RS/6000 are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc., in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product and service names may be trademarks or service marks of others.

Table of Contents

Disclaimers and Trademarks 2
Introduction 5
A brief history of Perl 6
Salient features of Perl 6
“The duct tape of the Internet” 6
Perl as Uber-Parser 7
Additional features of Perl include: 7
Modules and packages 7
Details on Perl packages 8
Object-oriented Perl 8
Bending and extending Perl 9
Perl IDEs and Perl utilities 9
Integrating Apache and Perl 11
Using mod_perl for Web application development 11
Database applications with Perl 12
Example application: GoogleSuck in mod_perl 12
Requirements and configuration 13
Build environment 15
Application walk-through 15
Presentation 17
Building GUI applications with Perl/Tk 17
Example application: GoogleSuck in Perl/Tk 19
Presentation 21
A brief history of Python 22
Salient features of Python 22
Clear and simple Syntax 22
Self-documenting 23
Interpreted language 23
Modules and packages 23
Windowing systems 23
Extending with C/C++ and Java 23
Web services support 24
Object-oriented programming in Python 24

Bending and extending Python 26
Python IDEs 26
Building GUI applications with Python 26
Python and Java 28
Integrating with Apache 29
Database applications with DB2 and Python 30
Example application: GoogleSuck in Python 30
Example application: GoogleSuck in Tk 33
Epilogue: “Keep tomorrow dark” 37
Acknowledgements 38
About the authors 38

Introduction

Freshmeat (http://freshmeat.net), as anyone who works with Linux probably knows, is a comprehensive directory of Open Source software projects from around the world. It lists over 13,000 entries and is updated many times throughout the day. Anyone who delves into Open Source should look into Freshmeat to feel the pulse of the community. So it was to Freshmeat we turned when posed with the question: "What are Open Source developers using to build their projects?"

Of the more than two dozen languages listed therein, the top ones, with more than 500 to their name, are:

C	4,100 projects
Perl	2,152 projects
C++	1,658 projects
Java™	1,464 projects
PHP	1,313 projects
Python	729 projects

C and C++, with their long and intimate history with UNIX®, Linux, and other operating systems, come as no surprise to be at the top of the list. Java™, with an independent standards body, strong vendor backing and marketing visibility, comes as no surprise either.

But how do you explain the three Ps: Perl, PHP, and Python? How do you explain that these three languages, without anything more than enthusiastic grassroots support, should be used in so many projects today?

There are many possible reasons, foremost of which are their flexibility and simplicity. Still other reasons are their portability and extensibility.

Ultimately, the reason must lie in the tools themselves. We must remember that these are languages designed by programmers for programmers; it is more than likely that they will strike a chord in that mindset common to programmers everywhere.

To help you gain a better understanding of their popularity with Open Source developers, we will take you through a tour of the important features and functions of Perl and Python. We have singled out these two because they share many common traits, such as an interpreted scripting environment, rich programming interfaces, and many different extensions.

Using these language extensions in combination with IBM's® DB2® database and with Web services provided through the Google APIs, we will build both a client and server-side application in both Perl and Python.

A brief history of Perl

Perl stands for the Practical Extraction and Report Language. As described by the Perl man page:

> "Perl is a interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. Perl is also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Perl combines some of the best features of C, sed, awk, and sh, so people familiar with those languages should have little difficulty with it. Language historians will also note some vestiges of csh, Pascal, and even BASIC|PLUS. Expression syntax corresponds quite closely to C expression syntax. If you have a problem that would ordinarily use sed or awk or sh, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then Perl may be for you. There are also translators to turn your sed and awk scripts into Perl scripts. OK, enough hype."

Perl was created by Larry Wall (who wrote the above description, which can be found at http://www.perldoc.com/perl5.6.1/pod/perl.html), and Perl 1.000 was released in 1987. Due to its flexibility and ease of programming, Perl exploded in the system administrator and Web developer communities. The "Notes" in the Perl man page sum up the reasons why Perl was so rapidly embraced:

> "The Perl motto is 'There's more than one way to do it.' Divining how many more is left as an exercise to the reader. The three principal virtues of a programmer are Laziness, Impatience, and Hubris."

Because of these tenets of flexibility and programming ease, Perl has been used to solve very different types of problems.

Perl runs on a wide variety of platforms from Linux, Macintosh, and Microsoft® to Amiga, Atari, OS/390® and Tru64 -- and dozens more. For a comprehensive list, see http://cpan.org/ports/. You can dive deeper into Perl history here: http://www.perl.org/press/history.html.

Salient features of Perl

"The duct tape of the Internet"

Perl has often been called the "duct tape of the Internet" because it has been used to quickly bring together multiple types of software and systems as well as Web applications. Moreover, the metaphor to duct tape is not so much a reflection on Perl as the perfect solution for

"plumbing" problems as it is a metaphor for Perl as an efficient tool for getting a job done. From system and network administrators to Web developers, Perl has been one of the fundamental languages used throughout the Internet infrastructure.

Perl as Uber-Parser

Perl was designed from the start as a "language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information." Because of this, Perl is a highly efficient tool for manipulating and parsing data. Additionally, through Perl's modular architecture, modules such as XML::Parser[1] can be incorporated for quick XML document parsing:

```
$p1 = new XML::Parser(Style => 'Debug');
$p1->parsefile('REC-xml-19980210.xml');
$p1->parse('<foo id="me">Hello World</foo>');
```

By browsing the vast amount of Perl modules available just for XML (http://www.perl.com/CPAN-local/modules/by-module/XML/perl-xml-modules.html), you can begin to get a flavor for the power of Perl as parsing tool – and of CPAN as a resource.

Additional features of Perl include:

- Access to a wide variety of databases like DB2, Oracle, Sybase, MySQL, PostGres, and many others through the abstract database interface called DBI
- Unicode support
- Language support for both procedural and object-oriented programming
- Ability to interface with external C/C++ libraries through XS or SWIG
- Integration with Java through JPL

Modules and packages

Perl design allows for the use of modules and packages, some of which are implemented and accessed in an object-oriented fashion.

One of the biggest advantages with Perl is the huge number of freely available Perl modules. These modules contain pre-written Perl code that helps you complete your Perl software much faster. There are hundreds of free modules available on the Comprehensive Perl Archive Network, or CPAN, a set of Internet servers mirrored throughout the world (http://www.cpan.org). In other words, don't reinvent the wheel!

A module provides a way to package Perl code for reuse. A simple way to define a package is a Perl *namespace*.

---

[1] http://search.cpan.org/search?dist=XML-Parser

Available modules include support for access to DB2 and other databases; networking protocols such as HTTP (Web), POP3 (email), and FTP (file transfers); and special Win32 modules for access to the Windows 95 or NT® operating systems. The example code below uses modules for accessing DB2, communicating with Google through Web services, and using some Apache constants for working within mod_perl.

Details on Perl packages

Perl modules provide a mechanism for alternative namespaces to protect packages from stomping on each other's variables. In fact, there's really no such thing as a global variable in Perl. The package statement declares the compilation unit as being in the given namespace. The scope of the package declaration is from the declaration itself through the end of the enclosing `block`, `eval`, `sub`, or end of file, whichever comes first. All further unqualified dynamic identifiers will be in this namespace. A package statement only affects dynamic variables--including those on which you've used `local()` -- but not lexical variables created with `my()`. Typically it would be the first declaration in a file to be included by the `require` or `use` operator. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the `main` package is assumed. That is, `$::sail` is equivalent to `$main::sail`.

Understanding `package` scope is important; to illustrate, here's an example:

```
package main;
# main is the default, and therefore unnecessary in a real package, but being
used here for clarity

$choice = "bluepill";
{
      package matrix;
      $choice = "redpill";
}
print "choice = $choice, matrix::choice = $matrix::choice \n";
```

This will print:

```
choice = bluepill, matrix::choice = redpill
```

So you can see that `$choice` is limited to the namespace of the package.

Understanding modules and packages, and using resources such as CPAN, are important for successful and efficient Perl development.

Object-oriented Perl

True to Perl's nature, you can create classes and objects without a lot of overhead and effort. As described from the "perlobj" man page entry:

1. An object is simply a reference that happens to know which class it belongs to.
2. A class is simply a package that happens to provide methods to deal with object references.
3. A module is a package that follows certain conventions.
4. A method is simply a subroutine that expects an object reference (or a package name, for class methods) as the first argument.
5. An attribute is data inside an object, typically stored in a hash.

In Perl, an object is simply a reference. Unlike some languages, such as C++, Perl doesn't provide any special syntax for constructors. A constructor is merely a subroutine that returns a reference to something "blessed" into a class, generally the class that the subroutine is defined in. Here is a typical constructor:

```
package Foo;
sub new { bless {} }
```

The word “new” is not a language keyword, but a general practice among Perl programmers (you can call this anything you want). The {} allocates an anonymous hash containing no key/value pairs, and returns it. The `bless` function takes that reference and tells the object it references that it is now a Foo and returns the reference. This is for convenience, because the referenced object itself knows that it has been blessed, and the reference to it could have been returned directly, like this:

```
sub new {
                my $self = {};
                bless $self;
                return $self;
}
```

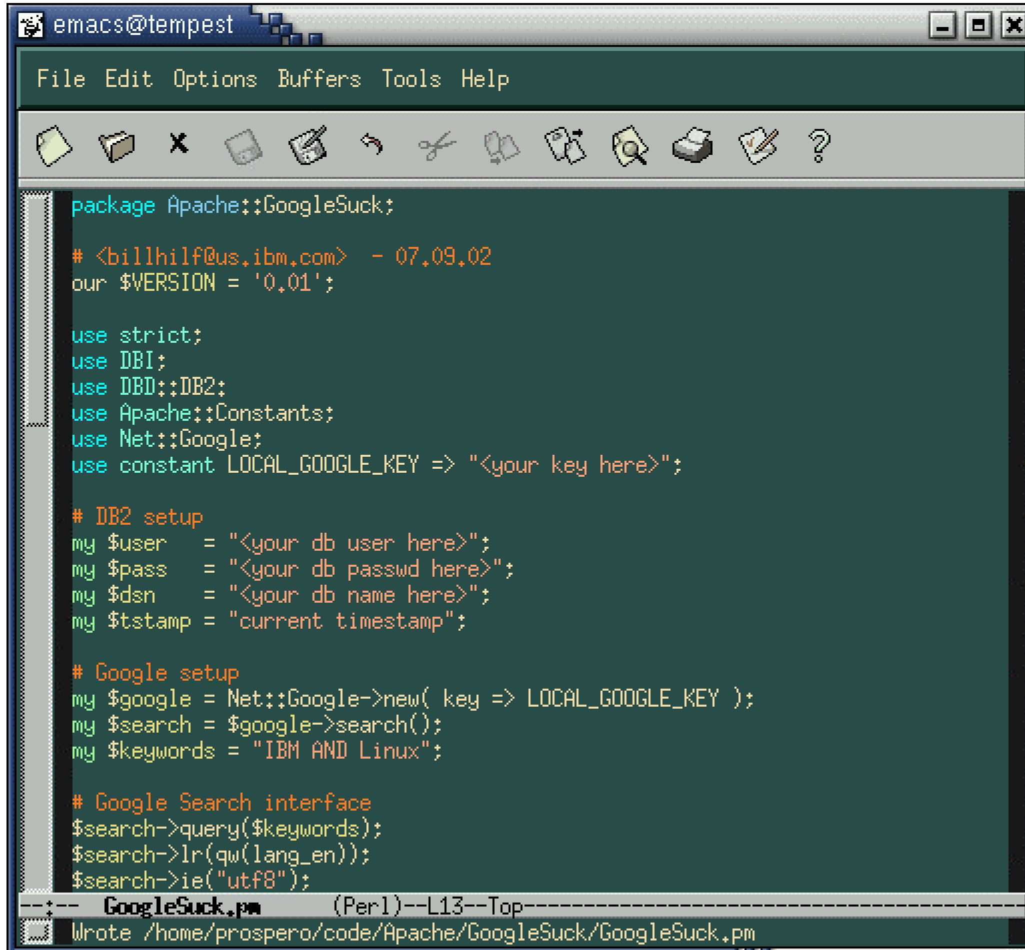
Creating objects is relatively simple in Perl. Perl does not require that an object be a special kind of data structure. In fact, you can use any existing type of Perl variable -- a scalar, an array, a hash -- as an object in Perl. Using `bless` you tell Perl that a particular object (or variable) belongs to a particular class. Object-oriented Perl programming is extremely flexible, and therefore requires a solid understanding of its use. (As any programmer will confess, flexibility is a double-edged sword). Damian Conway wrote the canonical text for OO Perl, which can be found at: http://www.manning.com/Conway/, and we would consider this essential reading before embarking on OO Perl projects.

Bending and extending Perl

Perl IDEs and Perl utilities

There are several commercial and Open Source development tools for Perl. In the Linux environment, many Perl developers use Emacs as their Perl editor of choice. A screenshot of a Perl package in Emacs is shown in Figure 1:

Figure 1. Perl package in Emacs



Emacs provides the ability to have custom language filters, which handle proper indenting, syntax, and formatting for a wide variety of popular programming languages. Related to some of the DB2 work we'll be doing later, there is an article describing Tips on Using Emacs with DB2 at http://www7b.software.ibm.com/dmdd/library/techarticle/0206mathew/0206mathew.html.

Additionally, commercial products such as ActiveState's Perl development tools (ASPN Perl, Visual Perl, Perl Dev Kit - http://www.activestate.com) and Solution Soft's Perl Builder 2.0 (http://www.solutionsoft.com/perl.htm) are robust IDE environments that run both on Linux and Windows.

There is a tremendous amount of resources available for Perl programming. We recommend that you check out O'Reilly's Perl Network at http://www.perl.com, or Learn Perl at http://learn.perl.org.

Integrating Apache and Perl

Using mod_perl for Web application development

mod_perl brings together the full power of the Perl programming language and the Apache HTTP server. You can use Perl to manage Apache, respond to requests for Web pages, and much more. As the author of CGI.pm describes mod_perl:

> "mod_perl is more than CGI scripting on steroids. It is a whole new way to create dynamic content by utilizing the full power of the Apache web server to create stateful sessions, customized user authentication systems, smart proxies and much more. Yet, magically, your old CGI scripts will continue to work and work very fast indeed. With mod_perl you give up nothing and gain so much!"
> -- Lincoln Stein at http://perl.apache.org/

However, mod_perl is *not* CGI -- it is much faster. The mod_perl environment compiles the Perl program into byte code, and those bytes are cached by mod_perl. mod_perl gives you a persistent Perl interpreter embedded in the Apache Web server. This lets you avoid the overhead of starting an external interpreter and avoids the penalty of Perl startup time, giving you very fast dynamic content.

A common question in mod_perl development is how to measure performance. There are many ways to benchmark mod_perl code; below is a simple example from the mod_perl guide (http://perl.apache.org/docs/1.0/guide/) on using the Benchmark module:

```
use Benchmark;
timethis (1_000,
      sub {
            my $x = 100;
            my $y = log ($x ** 100)  for (0..10000);
      }
);
```

…which when executed would then print:

```
timethis 1000: 25 wallclock secs (24.93 usr +  0.00 sys = 24.93 CPU)
```

For a much more detailed look at performance tuning mod_perl, see Stas Beckman's excellent article at: http://www.perl.com/pub/a/2002/05/29/mod_perl-opt.html. Additionally, an excellent resource for learning how to develop in mod_perl is *Writing Apache Modules with Perl and C*, by Lincoln Stein and Doug MacEachern (O'Reilly and Associates, ISBN 156592567, March 1999).

As you would expect from the Perl community, there are hundreds of modules written for mod_perl, everything from persistent database connections, to templating systems[2], to complete XML content delivery systems. Many high-volume Web sites, such as Slashdot and Wired Magazine use mod_perl[3].

Database applications with Perl

Perl supports a wide range of databases, both commercial and open source. These packages are written to the DBI specification. More information can be found at http://dbi.perl.org. There is a tutorial on IBM developerWorks that details how to Perl to access DB2 (http://www-106.ibm.com/developerworks/education/r-perl.

For our example applications, we need to create a DB2 table to hold the data that will be returned from the Google search. The quick table creation script shown below also illustrates the "sitescan" table structure we will be inserting into from the mod_perl and Perl/Tk applications:

```
-- To execute, become user 'db2inst1', connect to db2 and run 'db2 -tf
create_db2.sql'

CREATE TABLE sitescan
(
      id INT GENERATED ALWAYS AS IDENTITY (START WITH 1),
      title VARCHAR(255)      NOT NULL,
      url   VARCHAR(255)      NOT NULL,
      search_query      VARCHAR(255),
      -- DB2 has a 26 byte timestamp
      timestamp TIMESTAMP,
      search_time FLOAT,
      -- snippet is an HTML formatted string
      snippet     VARCHAR(255),
      summary     VARCHAR(255),
      hostname    VARCHAR(255),
      PRIMARY KEY (id)
);
```

Example application: GoogleSuck in mod_perl

The following example, called "GoogleSuck," shows how easy it is to do a fairly significant amount of work with just over sixty lines of original code. You can download the source code at the following URLs:

Perl samples
HTTP:
http://www7.software.ibm.com/vadd-bin/httpdl?1/vadc/dmdd/db2/0210hilf/PerlSamples.tgz
FTP:
http://www7.software.ibm.com/vadd-bin/ftpdl?1/vadc/dmdd/db2/0210hilf/PerlSamples.tgz

---

[2] http://www.perl.com/pub/a/2001/08/21/templating.html
[3] http://www.perl.com/pub/a/2001/10/17/etoys.html

Python samples
HTTP:
http://www7.software.ibm.com/vadd-bin/httpdl?1/vadc/dmdd/db2/0210hilf/PythonSamples.tar.gz
FTP:
http://www7.software.ibm.com/vadd-bin/ftpdl?1/vadc/dmdd/db2/0210hilf/PythonSamples.tar.gz

The intended purpose of this example application is to illustrate the power of Perl, mod_perl, Web services, DB2, and the rapid and flexible application environment available by combining best-of-breed Open Source software platforms with best-of-breed databases for Linux.

The code is rather straightforward, with the goal of the example being to build a Web application in mod_perl that uses Web services and stores data into DB2 for Linux. The Perl module `GoogleSuck.pm` does the following:

1. Uses the Net::Google module to communicate with the Google Web services (http://www.google.com/apis) via SOAP[4]
2. Extracts the data returned from a specified Google request into a data object
3. Writes the data into a DB2 database
4. Displays the results to the user's Web browser.

This is clearly intended as an example; there are many additions which could make this example much more feature-rich.

Requirements and configuration

You will need the following software packages to implement this code on your own:

- Apache 1.3.26 (available from http://httpd.apache.org)
- mod_perl 1.27 (available from http://perl.apache.org)
- DB2 UDB EE 7.2 (Trial version available from: http://www14.software.ibm.com/webapp/download/search.jsp?go=y&rs=db2udbdl&S_TACT=&S_CMP)
- Perl 5.6.1 with the following packages (all available from http://www.cpan.org):
  - DBD-DB2-0.76
  - Net-Google-0.5
  - SOAP-Lite-0.55

To integrate mod_perl and Apache, you will have to install and configure in one of the following ways:

- Via CPAN:

```
# perl -MCPAN -e shell
```

[4] A SOAP message contains an interface on the server side procedure to execute, with acceptable parameters defined within the body of the message. The server, in response, sends a SOAP message back with result information in the body of its envelope.

```
cpan> install Bundle::Apache
```

or, alternatively (in one line),

```
# perl -MCPAN -e 'install Bundle::Apache'
```

- Through a distribution package installer, such as RPM:

```
# rpm -i apache-xx.xx.rpm
# rpm -i mod_perl-xx.xx.rpm
```

- Or an APT system:

```
# apt-get install libapache-mod-perl
```

- Using the source files for each and then `make` together:

```
# cd /usr/src
# lwp-download http://www.apache.org/dist/httpd/apache_x.x.x.tar.gz
# lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
# tar xzvf apache_x.x.x.tar.gz
# tar xzvf mod_perl-x.xx.tar.gz
# cd mod_perl-x.xx
# perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
# make && make test && make install
# cd ../apache_x.x.x
# make install
```

Of course, replace your directories and version preferences where applicable.

Finally, for this example, I simply added the following to the Apache `httpd.conf` file to map the mod_perl application to the directory path `/modperl`:

```
<Location /modperl>
  SetHandler perl-script
  PerlHandler Apache::GoogleSuck
</Location>
```

Now, when users request /modperl from this Apache server, Apache::GoogleSuck will be executed.

Last but not least, make sure you have your DB2 environment configured correctly for the user who is starting Apache. In this mod_perl example, the DB2 user is 'db2inst1' so I had to source the `db2profile` file found in `/home/db2inst1/sqllib/db2profile` for the user who starts Apache. You should see various DB2 settings in your environment variables, such as `DB2INSTANCE`, `LD_LIBRARY_PATH`, and other path settings. You can check this by doing a `env | grep db2`. In a production application, this environment setup would likely be set in an Apache startup or configuration file.

Build environment

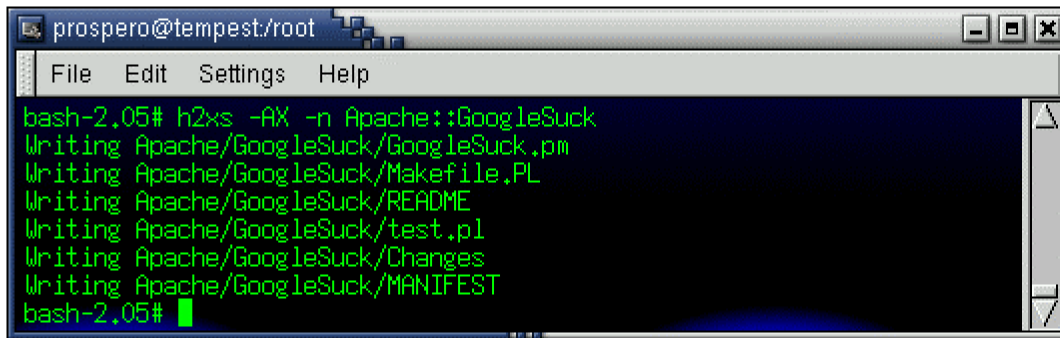
The mod_perl program GoogleSuck was built and run on an IBM x330 eServer with 256MB RAM, and (1) Pentium III 800MHz processor. The operating system was Red Hat Linux 7.2, with a 2.4.5-2 kernel.

Application walk-through

First off, there is a useful tool called h2xs, which you can use to build template files for your Apache module. This is the standard template format used for Apache modules on CPAN.

```
# h2xs -AX -n Apache::GoogleSuck
```

Figure 2. Creating templates for GoogleSuck



As you can see, h2xs created the directory structure `Apache/GoogleSuck`, as well as the template files for authoring the mod_perl Apache module.

Here are the contents of `GoogleSuck.pm` (with added comments):

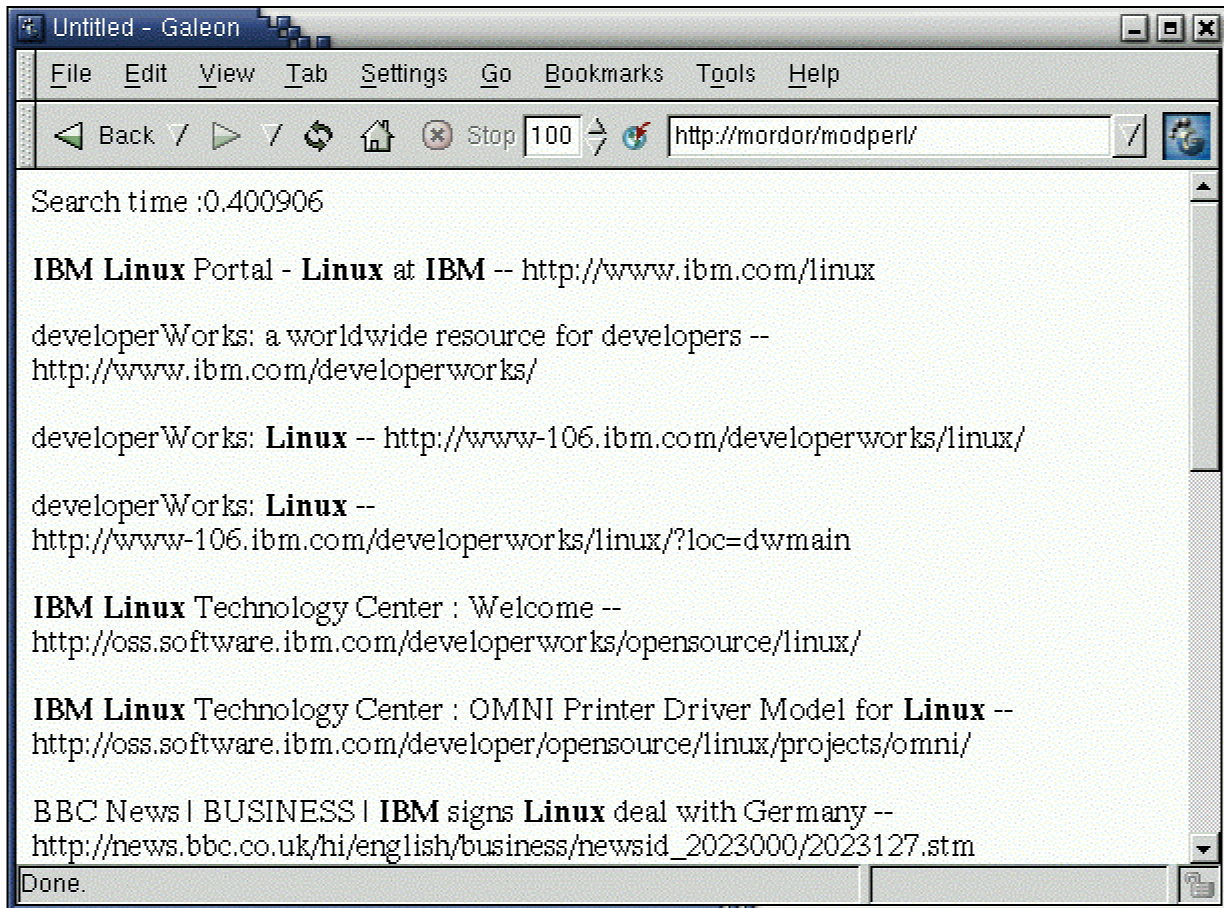
```
# declares a package, thus a package namespace
package Apache::GoogleSuck;
# restrict unsafe constructs, good Perl programming defensive measure
use strict;
# load the database interface package
use DBI;
# load the DBD::DB2 libraries for talking to DB2 through DBI
use DBD::DB2;
use Apache::Constants;
# provides the interface to the Google Web services
use Net::Google;
# Google requires a local key to manage access to their APIs - you would
insert your key here
use constant LOCAL_GOOGLE_KEY => "************";
# DB2 setup, declare the variables we'll need to connect to DB2
my $user = "db2inst1";
my $pass = "*****";  # you would insert your db password here
my $dsn  = "sample";
my $tstamp = "current timestamp";  # This allows us to insert DB2's TIMESTAMP
```

```
# Google setup, here we create a Google object and call the search method on
it, we also declare our search keywords (these would likely be form input in
a production application)
my $google = Net::Google->new(key=>LOCAL_GOOGLE_KEY);
my $search = $google->search();
my $keywords = "IBM AND Linux";

      # Google Search interface, establish the parameters of our interface
      $search->query($keywords);
      $search->lr(qw(lang_en));
      $search->ie("utf8");
      $search->oe("utf8");
      $search->starts_at(0);
      $search->max_results(15);

# DB2 work, connect to DB2
my $dbh = DBI->connect( "dbi:DB2:$dsn", $user, $pass )
  or die $DBI::errstr;
# create our SQL statement using bind variables
my $sql_stmt =
"insert into sitescan (title, url, search_query, search_time, snippet,
summary, hostname, timestamp) values (?,?,?,?,?,?,?,$tstamp)";
my $sth = $dbh->prepare($sql_stmt);

# loop through the data returned from the Google Web service...
foreach my $gr ( @{ $search->response() } ) {
    # nested loop to insert each of these results into their own row...
    for ( @{ $gr->resultElements } ) {
        $sth->execute( $_->title(), $_->URL(), $_->searchQuery(),
            $_->searchTime(), $_->snippet(), $_->summary(), $_->hostName() );
    }
}

# handler - work with Apache request object and return data to the client
sub handler {
    # we're shifting off the request object from Apache into $r
    my $r = shift;
    $r->send_http_header('text/html');
    print "<html><body> ";

    # similar to the DB2 insert above, but displaying to the browser instead
    foreach my $gr ( @{ $search->response() } ) {
        print "Search time :" . $gr->searchTime() . "<p>";

        # returns an array ref of Result objects
        # the same as the $search->results() method
        for ( @{ $gr->resultElements } ) {
            print $_->title() . " -- " . $_->URL() . "<p>";
        }
    }
    print "</body</html>";

    return OK;
}

1;
```

```
__END__
```

Presentation

Finally, here is the presentation of the GoogleSuck application results to the browser. This same information is also broken up and inserted into DB2, as described above.



Building GUI applications with Perl/Tk

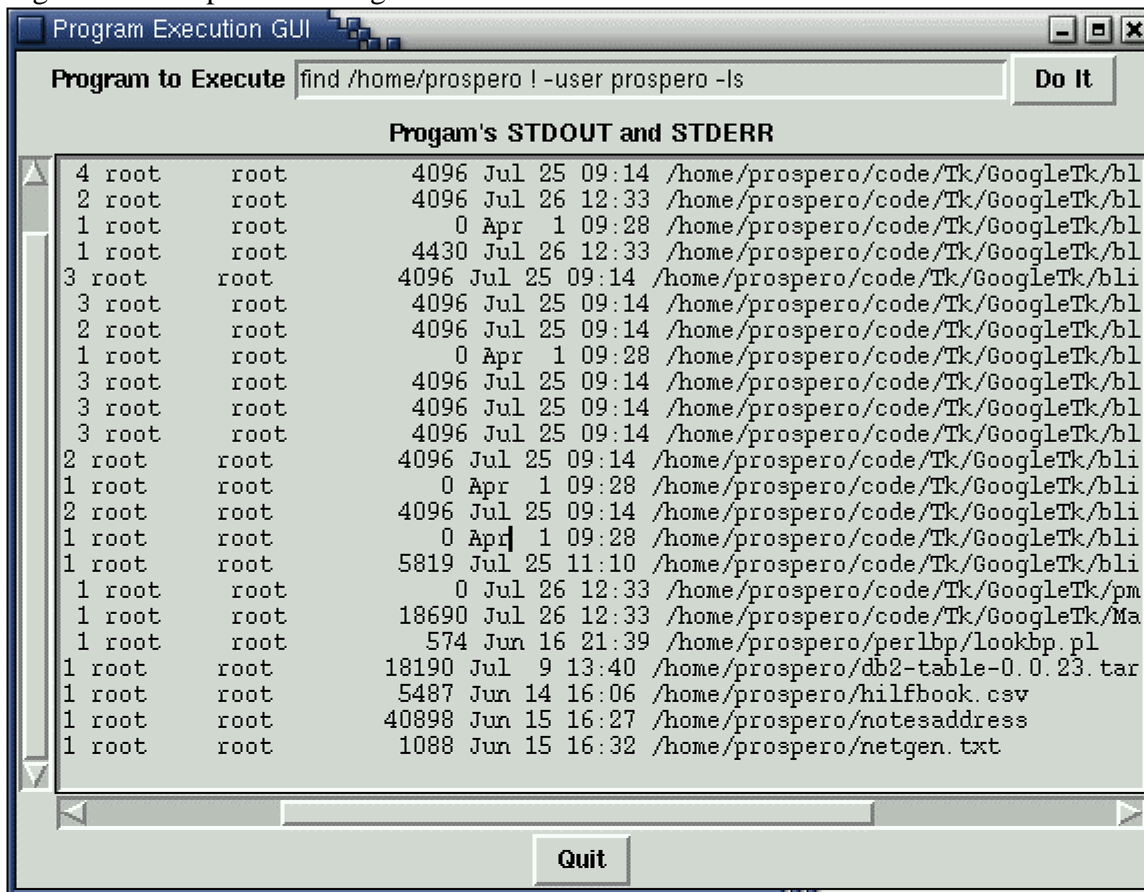
Perl/Tk is an extension to Perl for creating graphical user interfaces[5]. With Tk, Perl programs can be window-based rather than command-line based, with buttons, text entry fields, listboxes, menus, and scrollbars. Originally developed for the Tcl language, the Perl port of the Tk toolkit (aka, Tk widget set) allows Perl programmers to build graphical, event-driven applications for both Windows and UNIX. Although derived from Tcl, Perl/Tk does not require any familiarity

[5] Nick Ing-Simmons is the author of Perl/Tk. Perl/Tk is also known as pTk or ptk.

with Tcl, and with an understanding of Perl, developing GUI applications in Perl/Tk is relatively simple.

Perl/Tk is a very good solution for providing users an alternative to logging onto a Linux or Unix type of system and working via a terminal at the command line[6]. Figure 3 shows a simple GUI for using command line programs from a Perl/Tk GUI:

Figure 3. A simple GUI using Perl/Tk



The screenshot shows the user "prospero" typing the `find` command to look for files that are *not* owned by prospero in prospero's home directory.[7] A program such as this could be provided to non-privileged users who need to have certain types of command line access to a system. Alternatively, this program could be modified to replace the entry box with a fixed set of command line options, which the user could select from, as illustrated in the example Perl/Tk application below.

---

[6] There are other options for GUIs under Linux as well, such as wxWindows and the GTK+ toolkits.

[7] GUI derived from an example in Steve Lidie and Nancy Walsh's excellent book *Mastering Perl/Tk*, O'Reilly 2002.

To set up a Perl/Tk environment, it is often necessary to install Tk from CPAN, because Tk is not always included in some operating system distributions. The easiest way to do this is via CPAN:

```
[root@tempest /]# perl -MCPAN -e 'install Tk'
```

CPAN will then download and install Tk for your system (or tell you if your Tk is already installed and up to date).

Example application: GoogleSuck in Perl/Tk

To demonstrate Perl/Tk's GUI application capabilities, we've written a Tk version of GoogleSuck. The functions are almost exactly the same as the mod_perl version.

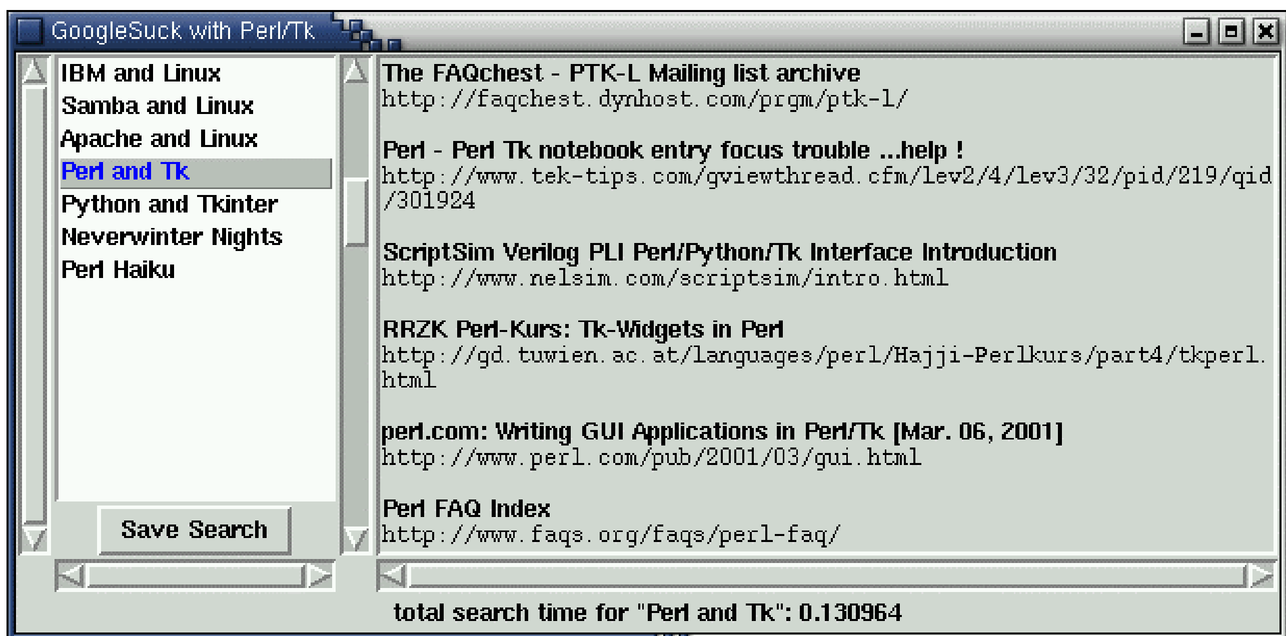
```
#!/usr/bin/perl -w

use Tk;
use DBI;
use DBD::DB2;
use Net::Google;
use constant LOCAL_GOOGLE_KEY => "****************************";
use subs qw/status/;
use strict;

# Tk setup
my $status;
my $mw = MainWindow->new;
$mw->title('GoogleSuck with Perl/Tk');
# Status window
my $sw = $mw->Label(-textvariable => \$status, -width => 100);
$sw->pack(qw/-side bottom -fill x -expand 1/);
# Label window
my $lbw = $mw->Scrolled(qw/Listbox -selectforeground blue -background
white/);
$lbw->pack(qw/side left -fill y -expand 1 -anchor w/);
# Add options to Label window
$lbw->insert('end', "IBM and Linux");
$lbw->insert('end', "Samba and Linux");
$lbw->insert('end', "Apache and Linux");
$lbw->insert('end', "Perl and Tk");
$lbw->insert('end', "Python and Tkinter");
$lbw->insert('end', "Neverwinter Nights");
$lbw->insert('end', "Perl Haiku");
# Read-Only Text window
my $tw = $mw->ROText();
$tw = $mw->Scrolled("ROText")->pack(-side => 'right');
$tw->tagConfigure('bold', -font => "Arial 12 bold");

# Setup button actions and status text
$lbw->bind('<ButtonRelease-1>' => \&google_it);
$mw->Button( -text => "Save Search", -command => \&save_results )
```

```
  ->pack( -side => "bottom", -in => $lbw );
my $help =  ' Google Options, <Button-1> Keywords, <Button-2> Stop Search';
status $help;

# Google setup
my $google = Net::Google->new(key=>LOCAL_GOOGLE_KEY);
my $search = $google->search();

# DB2 setup
my $user = "db2inst1";
my $pass = "*****";
my $dsn  = "sample";
my $tstamp = "current timestamp";

# Initiated from Label window Option selection callback
sub google_it {
    my $keywords = $lbw->get('active');
    status "Talking SOAP to Google for $keywords...";

# Google Search interface
    $search->query($keywords);
    $search->lr(qw(lang_en));
    $search->ie("utf8");
    $search->oe("utf8");
    $search->starts_at(0);
    $search->max_results(35);

# Print out results
    my (@new, $ctr, $st);
    $tw->insert("end","keywords: ".$keywords."\n\n", ['bold']);
    foreach my $gr (@{$search->results()}) {
        # Strip off the HTML from the titles in the Results object
        foreach my $nt ( $gr->title() ) {
            $nt =~ (s/<(?:[^>'"]*|(['"]).*?\1)*>//gs);
            @new = $nt;
            $ctr++;
        }
        # Pull the cleaned titles out, and display
        foreach my $i (@new) {
            $tw->insert( "end", $i . "\n", ['bold'], $gr->URL() . "\n\n" );
        }
    }
    # Pull the completion time and query terms off the Response object
    my ($time, $query);
    foreach my $r ( @{ $search->response() } ) {
        $time  = $r->searchTime();
        $query = $r->searchQuery();
    }
    status "total search time for \"$query\": $time";
}

# Initiated from 'Save Search' button callback
sub save_results {
# DB2 work
my $dbh = DBI->connect( "dbi:DB2:$dsn", $user, $pass )
      or die $DBI::errstr;
```

```
    my $sql_stmt =
"insert into sitescan (title, url, search_query, search_time, snippet,
summary, hostname, timestamp) values (?,?,?,?,?,?,?,$tstamp)";
    my $sth = $dbh->prepare($sql_stmt);
    foreach my $gr ( @{ $search->response() } ) {
        map {
            $sth->execute( $_->title(), $_->URL(), $_->searchQuery(),
                $_->searchTime(), $_->snippet(), $_->summary(),
                $_->hostName() );
        } @{ $gr->resultElements() };
    }
}

sub status {
    $status = $_[0];
    $mw->idletasks;
}

MainLoop;
```

Presentation

Here's the Perl/Tk GUI when executed:



Certainly, this could be enhanced and extended much further, including integrating non-blocking requests, or to use this GUI as an interface with the mod_perl, server-side application. We've kept these limited for the sake of clarity and brevity.

That covers mod_perl and Perl/Tk for server- and client-side programming. Now, let's look at the same client and server applications in Python.

A brief history of Python

Python is an interpreted, interactive, object-oriented programming language. At its most basic, Python can be thought of as a high-level scripting language; behind this simplicity is a powerful set of features, including modules, exceptions, dynamic typing, very high-level dynamic data types, and classes.

Python was invented in 1989 by Guido van Rossum, then working for CWI, which is a mathematics and computer research institute in the Netherlands. It had its roots in ABC, another interpreted programming language, with some features adapted from Modula-2+, and Modula-3. Version 1.0 was publicly released in 1991.

Van Rossum wrote Python initially as a system administration tool for Amoeba, a distributed operating system. From these roots, Python has come a long way. It has been used for a variety of projects such as system management scripts, scientific modelling, rapid application development, CGI scripting, XML processing, software testing, multimedia, graphics modelling, and games development. The list of applications continues to grow.

The latest release version is Python 2.2.1. Van Rossum, now working for Digital Creations, continues to guide its development in true Open-Source fashion. Python contributors abound, and there is an annual conference devoted to the language and the community.

Python currently runs on many brands of UNIX, including Linux. It also runs on MacOS, MS-DOS, Windows 9x/Windows NT/2000, OS/2, and PalmOS. Its portability has been extended further with Jython, an interpreter written in Java.

Why is it called Python? Van Rossum provides the answer at http://www.python.org/cgi-bin/faqw.py?req=show&file=faq01.002.htp:

> "Apart from being a computer scientist, I'm also a fan of "Monty Python's Flying Circus" (a BBC comedy series from the seventies, in the -- unlikely -- case you didn't know). It occurred to me one day that I needed a name that was short, unique, and slightly mysterious. And I happened to be reading some scripts from the series at the time... So then I decided to call my language Python."

Salient features of Python

Clear and simple Syntax

Simplicity is the key feature of Python; it's an easy language to learn and code written in it is easy to maintain. Python does away with statement terminators, and tags such as the semi-colon and curly braces in Java and C; instead, it enforces indentation as a way to block off groups of statements, and it relies on carriage returns to delineate statements. This minimalism extends in

subtle ways throughout most of the language. Generally, Python code is two to ten times shorter than its equivalent in C or Java.

Self-documenting

Incredible as it sounds, Python is a self-documenting language. For instance, programmers can look into variables and methods provided by any object in Python by using the `dir()` function. This feature makes it very easy for beginning programmers to pick up the nuances of the language; it is also invaluable for more advanced developers, who no longer need to reference manuals for quick mnemonic hints. Additional descriptions can be built into each function, module, class, or program to improve on the quality of the documentation.

Interpreted language

Python is primarily an interpreted language, and this makes it ideal for prototyping and rapid application development. Programmers do not need to compile their applications, thus decreasing development time. Also, because of this, a Python program cannot cause a segmentation fault, either because of a bug or bad input; instead, the interpreter simply raises an exception.

This does not mean, though, that Python programs cannot be compiled. Internally, Python source code is always translated into "byte code" representation, as in Java. However, this process takes place automatically.

Developers looking to distribute their Python applications without having to include the interpreter can make use of the `freeze.py` script included with the Python tools. This creates a single binary file that includes the interpreter and whatever modules are required. This approach does limit the binary to the same type of platform in which the application was created.

Modules and packages

In Python, a module is a collection of classes, variables, and methods written as Python code. A package is a collection of modules grouped in a directory or a directory tree.

Similar to Perl's CPAN, the Vaults of Parnassus (http://www.vex.net/parnassus) lists hundreds of modules for almost every task imaginable. There are modules for networking protocols such as HTTP, POP3, FTP, and SOAP; there are modules for integrating with windowing systems; and there are also modules for interfacing with ActiveX, COM, and DCOM. Some developers have written modules for DB2 Universal Database™ and MQSeries®. The list is huge.

Windowing systems

With the appropriate modules, Python is able to support windowing systems natively, which means you can prototype and develop native GUI applications quickly. The standard GUI system in Python is the Tk library, but it also supports Gtk+, wxWindows, and Qt.

Extending with C/C++ and Java

Python is extensible. Developers can create built-in modules containing functions, variables, exceptions and even new types in C. Using compatibility features, this capability can be extended into C++.

This extensibility works both ways. Developers can execute Python statements, evaluate expressions, and call object methods from C.

Jython (http://www.jython.org) is a Java implementation of the Python interpreter. It can be used to connect Java components. Running in Jython, Python code can import and use Java classes as if they were Python modules. This leads to several interesting possibilities in application development. For instance, the Jakarta Tomcat Web server can host Jython applications called from JSPs or servlets.

That said, we must also point out that some differences do exist between Jython and the more common C-based implementation. Some of these differences are trivial, such as floating point representation; and some differences are significant, such as the availability of extension modules. For more details on these differences, see http://jython.sourceforge.net/docs/differences.html

Web services support

Python has a good interface for handling XML and Web services, as provided by various modules, both native and added-on. The XML parser is quite easy to use because the object model used by Python mirrors that used for XML documents, particularly the DOM.

XML support for Python has evolved somewhat over the years, so developers do need to make some distinctions. Python 1.5.2 used an XML module called xmllib, which provided both simple validation and an event-driven parser. Python 2.0 introduced a new hierarchy of modules and packages for working with DOM and SAX, as well as an interface to the generic Expat parser. xmllib, however, is still available in newer versions of Python, though its use has been deprecated in favor of the newer packages.

Python also has SOAP and XML-RPC modules, so developing Web services application in Python is fairly easy. Additionally, DB2 provides superior support for Web services, more developer articles on this subject can be found at: http://www7b.software.ibm.com/dmdd/zones/webservices/

Object-oriented programming in Python

Python has strong object-oriented capabilities. The details of this implementation remain largely hidden when Python is used as a scripting tool: beginning developers can use Python as a functional programming language. But when object-oriented programming is appropriate, Python shines.

The class mechanisms in Python derive from C++ and Modula-3. As per the Python tutorial: "Classes can inherit from multiple base classes (multiple inheritance), a derived class can override the methods of its base class or classes, and a method can call the methods of a base class with the same name. Objects can contain an arbitrary amount of private data."

Java and C++ programmers would probably be surprised to learn that there is no concept of private variables or functions in classes. Instead, Python relies on the programmer not to break the definition of the class.

The listing below shows an example of basic class in Python, as taken from Van Rossum's tutorial presentation (found at http://www.python.org/doc/essays/ppt).

```
class Stack:
      """A well-known data structure"""   #documentation string
      def __init__(self):                 #constructor
            self.items=[]
      def push(self, x):
            self.items.append(x)          #the sky is the limit
      def pop(self):
            x=self.items[-1]              #what happens if it's empty?
            del   self.items[-1]
            return x
      def empty(self):
            return len(self.items)==0     #Boolean result
```

Using classes in Python is straightforward. With the class listed above as an example, here are the ways in we can instantiate and use classes.

To create a class,

```
x=Stack()
```

In Python, you do not use the “new” operator to instantiate a class.

To use the methods of the instance, use the dot operator:

```
x.empty()                # -> returns 1 because, yes, it's empty
x.push(1)                # [1] this is the content of the stack
x.empty()                # -> returns 0 because, no, it's not empty
x.push("hello")          # [1, "hello"]
x.pop()                  # "hello"   [1]
```

Similarly, the values stored by instance variables can be retrieved using the dot operator:

```
x.items                  # returns [1]
```

Inheritance in Python is also straightforward. The class is defined using the parent class or classes as parameters. The listing below shows how this is done.

```
class FancyStack(Stack):
        """stack with added ability to inspect inferior stack items"""

        def peek(self, n):
                """peek(0) returns top; peek(-1) returns items below that"""
                size=len(self.items)
                assert 0 <= n < size
                return self.items[size-1-n]
```

Bending and extending Python

One of the reasons for the longevity and popularity of Python is its extensibility. It is for this same reason that Python has made its way into such a diverse range of applications. In this section, we look more closely at some of these extensions.

Python IDEs

Python has a number of integrated development environments built for it, in both open source and commercial flavors. In fact, the standard IDE ships with the Python tools, a small package called IDLE, written by Van Rossum.

IDLE is a small but functional environment that provides code highlighting, a class browser, and a debugger. The debugger provides breakpoints, stepping, and variable watches. It doesn't have a visual development environment, but for most other jobs, it does the trick.

Most open source code editors commonly support Python code highlighting, at the very least. Among these editors are the staple Vim and Emacs.

Other free IDEs include PythonWin, MacPython IDE, and Boa Constructor. PythonWin and MacPython IDE are specific to their platforms. Boa Constructor, still under heavy development, runs on multiple platforms and features visual development using wxWindows widgets.

Commercial IDEs currently include: Komodo, BlackAdder, WingIDE, and PythonWorks Pro.

A nice article by David Mertz, appearing in IBM developerWorks, takes a closer look at these IDEs. You can read it at http://www-106.ibm.com/developerworks/linux/library/l-pide.

Building GUI applications with Python

Python works with several windowing systems. The standard system, included with most versions of Python, is the Tk library, which provides a number of commonly used widgets.

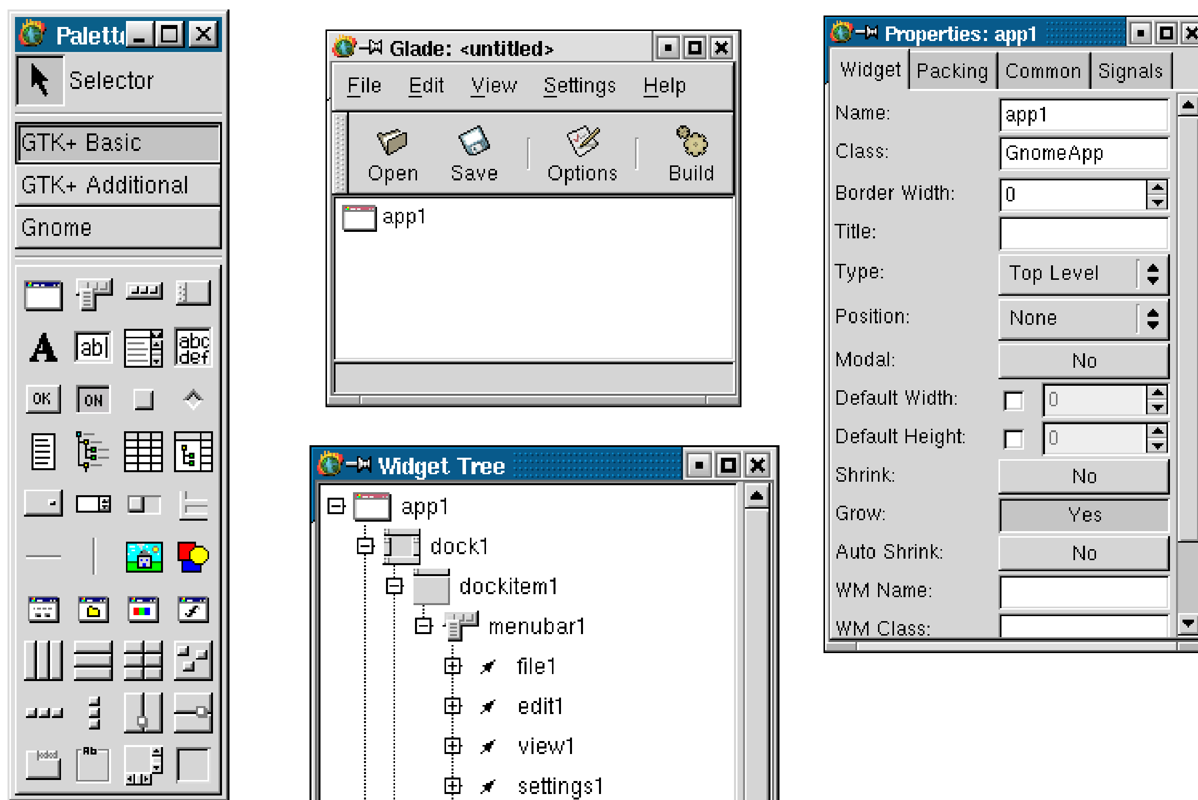
The basic Tk widget set is rather limited. Some developers have put together extensions to Tk, which add more widgets. Other windowing systems offer a bit more, and we will examine some of them briefly. If you want to learn more about Python and Tk, visit http://www.pythonware.com/library/tkinter/introduction/index.htm

A standard Red Hat distribution ships with Gtk+, otherwise known as the GIMP Toolkit. Gtk+ offers a better widget selection than Tk. The native programming language for Gtk+ applications is C, but a Python binding, written by James Henstridge, exists. You can get more information on the binding from his Web site at http://www.daa.com.au/~james.

The Gtk+ methods in Python closely match their C equivalents, which is a good thing because the Gtk+ documentation is all written in C. You can build a GUI application in a manner similar to the Tk example above.

Gtk+ deserves special mention because it is the basis for two additional GUI-related features. First, it is the basis for the Gnome UI, which is essentially a collection of commonly used objects built using Gtk+ widgets. Some examples of these widgets include dialog boxes, font selection, calendars, editing windows, and file selectors. Second, there is a development tool called Glade, which allows you to assemble your GUI elements visually.

Figure 4. Gtk+ GUI features



Glade can output code in C/C++, Perl, Ada, and Eiffel, but not in Python. However, it does produce an XML file that describes the objects, events, and placement of the application windows. Python can read in this XML file using a module called libglade, and thus make all the widgets available to your Python code. It's not a full-fledged visual development tool, because you still have to code the events behind the scenes, but it does open up possibilities for quick-and-dirty prototyping and creation of simple visual applications.

There is a good tutorial presentation on using Python with Gtk+ and Glade, written by Brian Kelly, available at http://www.bioreason.com/~brian/GladeBase.html

That's not all, though. A standard Red Hat distribution also comes with the Qt Designer. Qt is another GUI toolkit available across multiple platforms, and Qt designer is a visual development tool[8]. As with Glade, its primary target is C/C++. A utility called PyQt creates a Python class out of the window elements; you can then instantiate the class from a Python program and override the methods therein.

Yet another windowing system, and one that is gaining popularity, is wxPython, built around the wxWindows toolkit with some elements from Tk. It also provides cross-platform capabilities. It is by far the best-documented windowing system for Python. You can download it from http://www.wxpython.org.

All in all, the GUI extensions to Python offer exciting possibilities for developers looking to develop quickly on Linux.

Python and Java

Jython is an implementation of Python written in Java. Jython, like its predecessor, Python, is certified as 100% Pure Java. It is distributed as a Java class file, and requires a highly compliant JDK to run properly.

Jython is useful for embedded scripting. Java programmers can add the Jython libraries to their system to allow end users to write simple or complicated scripts that add functionality to the application. Python code executed in Jython is dynamically compiled to Java bytecodes, resulting in better performance.

The sample Jython session demonstrates how Java classes can be instantiated and called from Python code.

```
>>> import javax.swing as swing
>>> win = swing.JFrame("Welcome to Jython")
>>> win.size = (200, 200)
>>> win.show()
```

[8]Qt is available in both free and commercial editions. Although Qt Free Edition is under GPL, developers concerned with the licensing may wish to read up on the specifics and history at http://www.trolltech.com/developer/licensing/index.html and http://www.trolltech.com/developer/faqs/simple.html.

In the example above, Jython will create a swing JFrame and display it on screen. This example and more are given at http://www.onjava.com/pub/a/onjava/2002/03/27/jython.html

Integrating with Apache

With Python, you have several options for building an interactive Web application. On the extreme side, you could build a Web server completely in Python, using modules that come with the standard package. This gives you better control of the application and is much more compact, but comes at the expense of speed and features. Developers writing small Web-based system administration programs would probably take this path.

A more conventional approach would be to execute the Python code as a CGI script. A CGI module comes with the standard Python package, and you can use this module to parse CGI variables. This is simple to implement; you don't need to recompile your Web server, but again, it comes at the expense of speed.

The third approach is to incorporate Python capabilities into the Web server, in this case, Apache.[9]

In a fashion similar to mod_perl, Python also has mod_python. mod_python was written by Gregory Trubetskoy, and is available at http://www.modpython.org. We will look in some detail at this module.

mod_python requires at least Python 1.5.2 and Apache 1.3. Apache 1.3.20 or higher is recommended; Apache 2.0 is not yet supported. mod_python can be compiled and statically linked into Apache. It can also be dynamically loaded as a DSO (Dynamic Shared Object), assuming the web server was compiled with this option. As a DSO, mod_python is dynamically loaded at runtime. Installation and configuration instructions for both methods are clearly explained in the mod_python documentation.

Apache processes requests in phases, and are implemented by handlers. Typically, the handlers are written in C; mod_python allows the handlers to be written in Python.

The listing below shows how you can write one such simple handler:

```
# mptest.py
from mod_python import apache

def handler(req):
      req.send_http_header()
      req.write("Hello World!")
      return apache.OK
```

[9]And yet another approach, also based on Python, is Zope. Zope is a content-management system and web server, maintained by Digital Creations, where Guido van Rossum currently works. It is available from http://www.zope.org

This file is stored in a directory that is marked in the Apache configuration file as processing Python handlers. req is a Request object: it contains all the information about the request, such as the IP of the client, the headers, and the URI. The response to the client is also done through this object.

Simple as the handler is, mod_python provides an even simpler mechanism for processing requests. The alternative is the standard mod_python handler called Publisher. This handler allows access to functions and variables within a module via URLs. Here is the example given in the documentation. If you have the following module, called hello.py :

```
""" Publisher example """
def say(req, what="NOTHING"):
      return "I am saying %s" % what
```

a URL http://www.mysite.com/hello.py/say would return "I am saying NOTHING." A URL http://www.mysite.com/hello.py/say?what=hello would return "I am saying hello."

Database applications with DB2 and Python

Python supports a wide range of databases, both commercial and open source. These packages are written to the Python DBI specification, currently at version 2.0. More information can be found at http://www.python.org/topics/database/.

The Python interface to DB2 was written by Man-Yong (Bryan) Lee and is available for download at ftp://people.linuxkorea.co.kr/pub/DB2/. It works with DB2 UDB v7.1 and v7.2, and requires Python 2.0 or later. It supports cursors as well as BLOBs. Below are sample sessions, as taken from the documentation:

```
>>> import DB2
>>> conn = DB2.connect(dsn='sample', uid='db2inst1', pwd='ibmdb2')
>>> curs = conn.cursor()
>>> curs.execute('SELECT * FROM staff')
>>> one_row = curs.fetchone()
>>> many_rows = curs.fetchmany(3)
>>> many_rows = curs.fetchall()
```

Example application: GoogleSuck in Python

To show Python in action, we will recreate the GoogleSuck example given earlier. The equipment used in developing and testing this application was a Pentium II-350 with 256MB RAM, running Red Hat 7.2 on a 2.4.7-10 kernel. For the database, we used DB2 UDB v7.2. We used the Python2-2.1.1-2.rpm package that came with the distribution.[10]

[10]Red Hat 7.2 ships with both Python 1.5.2 and Python 2.1. The default is Python 1.5.2. To work these examples, we linked /usr/bin/python to the python2 interpreter, because the Python 1.5.2 on Red Hat 7.2 did not have the

The following modules were also installed:

- Apache 1.3.26, downloaded from http://www.apache.org.
- mod_python 2.7.8, downloaded from http://www.modpython.org
- DB2-0.995, downloaded from ftp://ftp.linuxkorea.co.kr
- pyGoogle 0.5.2, downloaded from http://diveintomark.org

The pyGoogle module, written by Mark Pilgrim, performs a similar function to the Perl equivalent. The SOAP module is included with this package. The site provides other Web services interfaces, including that to Amazon.

Here is the code in Python:

```
# import the required modules
import google
import DB2

def main():
	# define constants
	db_uid="db2inst1"
	db_pwd="ibmdb2"
	db_dsn="test"
	tstamp="current timestamp"  # this allows us to insert DB2's TIMESTAMP

	# Google setup, get your own key
	go_license_key="**************"
	go_keywords="IBM AND Linux"
	go_language="lang_en"
	go_inputencoding="utf8"
	go_outputencoding="utf8"
	go_start=0
	go_maxResults=10

	# Perform the actual Google search, using all the options we specified
	data=google.doGoogleSearch(go_keywords,\
		start=go_start,\
		maxResults=go_maxResults,\
		language=go_language,\
		inputencoding=go_inputencoding,\
		outputencoding=go_outputencoding,\
		license_key=go_license_key
		)

	# Connect to the database; if connection fails, print message and exit
	try:
		conn=DB2.connect(dsn=db_dsn, uid=db_uid, pwd=db_pwd)
	except:
```

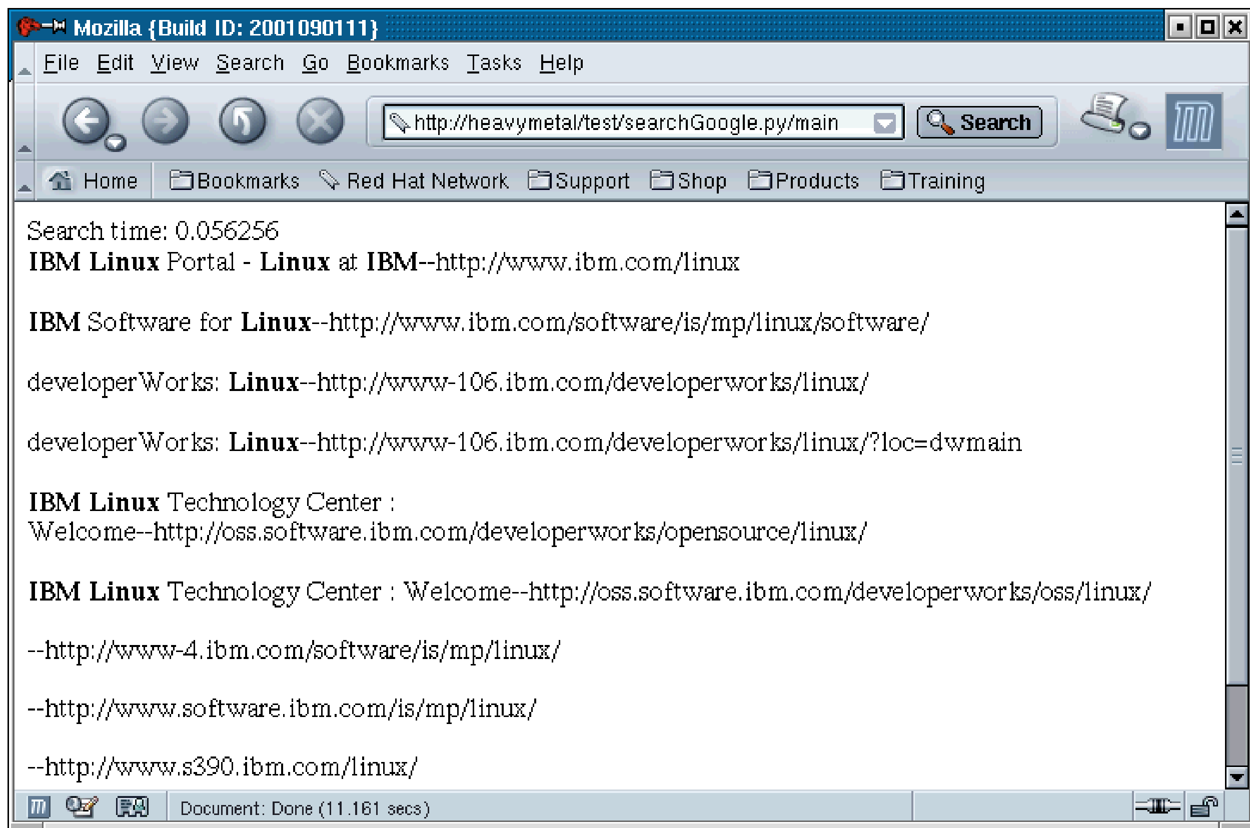
---

pydist module. The DB2 and pyGoogle modules also required the later version. In hindsight, this was not a good idea, because this broke some of the system administration routines of Red Hat. The implementation in Red Hat 7.3 is somewhat better, and mod_python is now included.

```
            print "Error opening database"
            sys.exit(1)

      curs=conn.cursor()

      sql_template="insert into sitescan (title, url, search_query,\
      search_time,\
            hostname) values \
            ('%s', '%s', '%s', %s, '%s')"
      search_query=data.meta.searchQuery
      search_time=data.meta.searchTime

      # Cycle through the results and insert them into the database
      for i in data.results:
            sql_statement=sql_template % (i.title, i.URL, search_query,\
            search_time, \
                  i.hostName)
            curs.execute(sql_statement)

      # And we're done with the database, so we close
      conn.close()

      # Now, we output the results as an HTML file
      # mod_python's Publisher handler takes care of all the other details
      # so all we need to do is to build the output string containing HTML
      s="Search time: " + str(search_time) + "<BR>"
      for i in data.results:
            s=s+i.title+"--"+i.URL+"<BR><BR>"

      # ...and we return it
      return s

# The lines below are not necessary, but convenient.
# With them, we can do command-line testing of the script.  When we do call
# the script from a browser, the lines are ignored
if __name__=="__main__":
      print main()
```

The output is given below. Please take note of the URL used to call the module. The file was saved as `searchGoogle.py`; the URL calls the file name, followed by the function, which in this case is called main().

Figure 5. Output from GoogleSuck Python Application



Example application: GoogleSuck in Tk

To demonstrate Python's GUI application capabilities, we've written a Tk version of GoogleSuck. The functions are almost exactly the same as the mod_python version.

Here is the listing:

```
#!/usr/bin/env python

from Tkinter import *
from string import *
import google
import DB2

# Define some constants
db_uid="db2inst1"
db_pwd="ibmdb2"
```

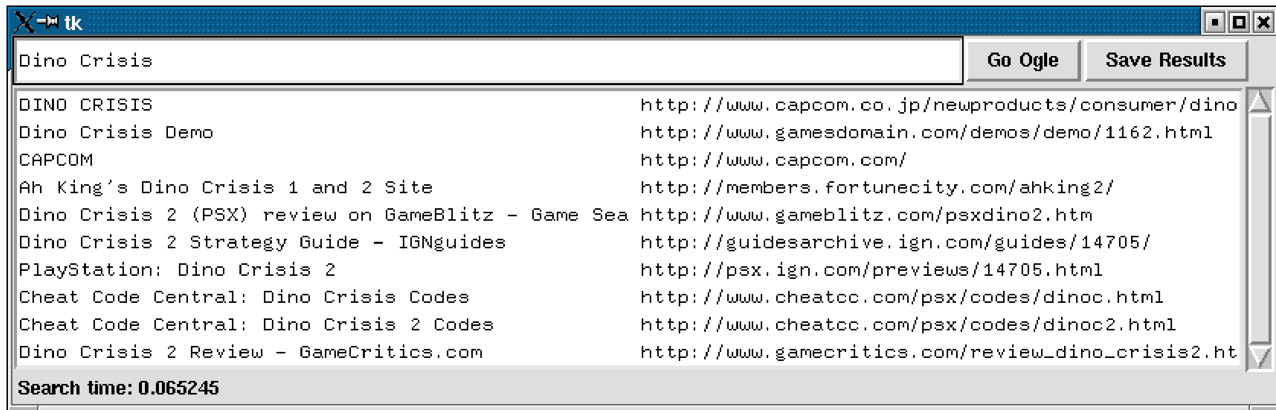
```
db_dsn="test"

# Google setup
go_license_key="***************"
go_language="lang_en"
go_inputencoding="utf8"
go_outputencoding="utf8"
go_start=0
go_maxResults=10


def add_entry(event):
      go_keywords=entry.get()

      # define data as a global - just an expedient
      global data
      data=google.doGoogleSearch(go_keywords,\
                start=go_start,\
                maxResults=go_maxResults,\
                language=go_language,\
                inputencoding=go_inputencoding,\
                outputencoding=go_outputencoding,\
                license_key=go_license_key
                )

      # clear the listbox with every new query
      listbox.delete(0, END)

      # show the results
        for i in data.results:
            title=lstrip(i.title+" "*100) # Pad the title with spaces
            url=lstrip(i.URL+" "*100)     # Pad the URL with spaces

            title=replace(title, "<b>", "")     # Remove the ugly tags
            title=replace(title, "</b>", "")

            title=replace(title, "'", "") # Remove the apostrophes
            url=replace(url,"'", "")

            listbox.insert(END, title[0:50] + " " + url[0:70])

      # show the search time in the status box
      label["text"]="Search time: " + str(data.meta.searchTime)

      # show the Save Results button; if there is no query, button is hidden
      button2.pack(side=LEFT)


def search_save(event):

      global data

      # Connect to the database; if connection fails, print a message, exit
      try:
              conn=DB2.connect(dsn=db_dsn, uid=db_uid, pwd=db_pwd)
      except:
```

```
                print "Error opening database"

                sys.exit(1)

        curs=conn.cursor()

        sql_template="insert into sitescan (title, url, search_query,\
              search_time,\
              hostname) values \
              ('%s', '%s', '%s', %s, '%s')"
              search_query=data.meta.searchQuery
          search_time=data.meta.searchTime

         # Cycle through the results and insert them into the database
         for i in data.results:
               sql_statement=sql_template % (i.title, i.URL, search_query,\
                           search_time, \
                           i.hostName)

                  curs.execute(sql_statement)

        conn.close()

        # Update the status bar
        label["text"]="Search saved"


root=Tk()

frame1=Frame(root)
frame1.pack(expand=1, fill=BOTH)

entry=Entry(frame1, width=77, background="white", font="terminal")
entry.insert(0, "<Type your Google query here>")
entry.pack(side=LEFT, fill=Y)

button1=Button(frame1)
button1["text"]="Go Ogle"
button1.bind("<Button>", add_entry)
button1.pack(side=LEFT)

button2=Button(frame1)
button2["text"]="Save Results"
button2.bind("<Button>", search_save)

frame2=Frame(root)
frame2.pack(expand=1, fill=BOTH)

scrollbar=Scrollbar(frame2, orient=VERTICAL)
listbox=Listbox(frame2, height=10, width=100, yscrollcommand=scrollbar.set,
background="white", font="terminal")
scrollbar.config(command=listbox.yview)
listbox.pack(side=LEFT)
scrollbar.pack(side=LEFT, fill=Y)

label=Label(root)
```

```
label["text"]="Enter a query"
label.pack(side=LEFT)

root.mainloop()
```

Figure 6 shows the application, when it is run:

Figure 6. GoogleSuck application in Tk



Epilogue: "Keep tomorrow dark"

In our introduction, we cited several possible reasons for the popularity of Perl and Python (and also PHP) as Open Source development languages: their flexibility, simplicity, power, speed of development, and extensibility, among other things. We even conjectured that the answer may lie in the design of the languages themselves. It is our hope that this brief tour, along with the resources we've listed, have given you enough information to come to your own decision.

But there may be an even more fundamental reason for their popularity, something not rooted in the cold logic of technology or market research. We refer to a quirk of human nature. It is easy to understand but hard to explain. The writer and philosopher GK Chesterton probably best summed it up in the opening lines of his short novel The Napoleon of Notting Hill (House of Stratus Inc., January 2001, ISBN 0755100174):

> "The human race, to which so many of my readers belong, has been playing at children's games from the beginning, and will probably do it till the end, which is a nuisance for the few people who grow up.
>
> "And one of the games to which it is most attached is called, 'Keep tomorrow dark,' and which is also named (by the rustics in Shropshire, I have no doubt) 'Cheat the Prophet.' The players listen very carefully and respectfully to all that the clever men have to say about what is to happen in the next generation. The players then wait until all the clever men are dead, and bury them nicely. Then they go and do something else. That is all. For a race of simple tastes, however, it is great fun."

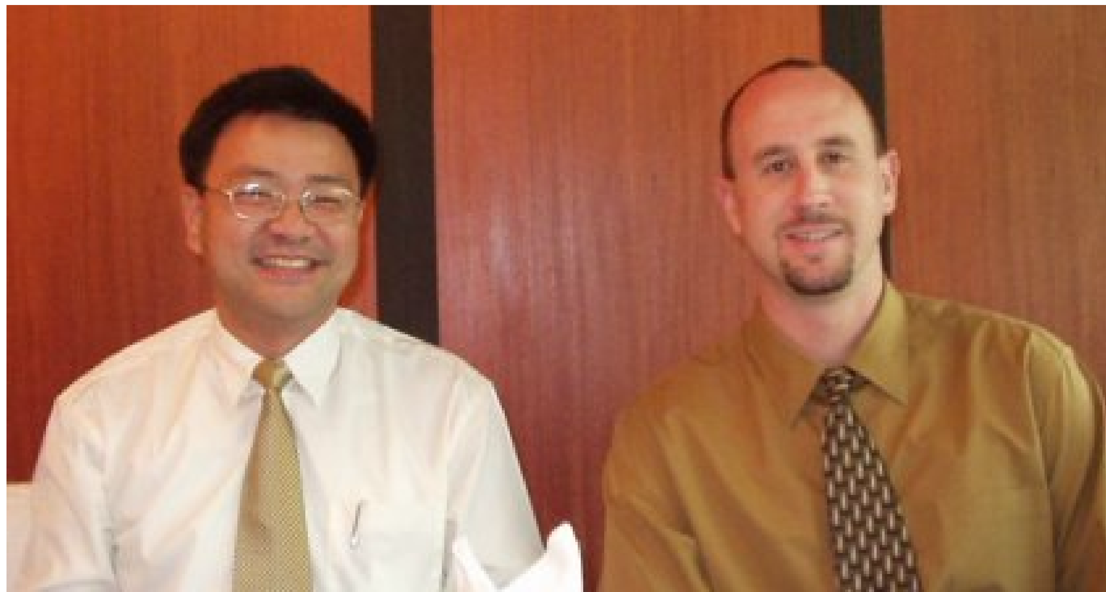
Our point is: People -- programmers especially -- will always find a way to "Cheat the Prophet" (whether they are vendors, market analysts, committees, writers, or multi-billionaires). In this wonderful dynamic called Open Source, they have found the perfect way to do that.

Acknowledgements

We would like to gratefully acknowledge and thank our technical reviewers:

Perrin Harkins, William Yu, and Miguel Paraz

About the authors



Dominique Cimafranca and Bill Hilf (the authors, in Korea)

Bill Hilf (billhilf@us.ibm.com)

Bill Hilf is a Sr. I/T Architect for IBM's Emerging and Competitive Markets team. Bill has been involved with Open Source technologies as a developer, architect, and evangelist for over seven years. Before joining IBM, Bill was the Sr. Director of Engineering for eToys, and an architect for CNET's sites, such as News.com. Bill holds a Masters degree from Chapman University and serves as a board member and instructor for the California State University, Fullerton, UNIX and Linux educational programs.

Dominique Cimafranca (cimafran@ph.ibm.com)

Dominique Gerald Cimafranca is an IT Specialist for IBM Asia-Pacific's Emerging and Competitive Markets team. In this role, he works on Linux, Digital Media, and Life Sciences opportunities. Prior to this, he also worked for IBM's NetGen and e-Business teams. He is responsible for the first RS/6000® SP-based ISP installation in Asia. He has been working with IBM since 1997 and is co-author of a firewall RedBook.

Before joining IBM, Dominique worked with Digital Equipment Corporation as a firewall and Internet consultant. He has been involved in customer engagements in over 18 countries.