

# *The Perl Journal*

## Implementing the Khazad Block Cipher in Perl

Julius C. Duque • 3

## Programming Persistent Objects with *Class::PObject*

Sherzod B. Ruzmetov • 8

## Therapy Bots

Moshe Bar • 12

## Bryar: A New Weblogging Tool

Simon Cozens • 15

## PLUS

Letter from the Editor • 1

Perl News Special Report: OSCON 2003 • 2

Book Review by Russell J.T. Dyer:

*Learning Perl Objects, References, & Modules* • 19

Source Code Appendix • 21

## LETTER FROM THE EDITOR

### Perl Gets a Pony

It's one of life's little ironies that sometimes you have to go backward in order to go forward. At the recent Open Source Conference in Portland, OR, Larry Wall announced the Pony Project, an effort to port Perl 5 to run on Parrot. It's a huge undertaking, but a tremendously necessary one. An abrupt shift to Perl 6 would break millions of lines of Perl 5 code out there that are, right now, performing just fine.

So we need an interim solution like Pony that allows Parrot adoption, but will allow backward compatibility. But the Perl 5 internals are quite a house of cards. Running on Parrot will require digging around in there without bring it down around our ears. Any code base of any appreciable age (and Perl 5 surely qualifies) inevitably accumulates a healthy crust of weirdly interdependent code. Sometimes, you can't fix one thing without breaking about five things. Even the most cleverly designed systems seem not to escape this rule. So while we can pat ourselves on the back for Perl's adaptability, flexibility, and yes, cleverness, I don't envy the Pony developers their task. It ain't gonna be pretty.

To ensure Perl's future, the Pony developers will need to exhume a few skeletons. It will be a lot of work, and all of it just to get Perl past what might be termed a transitional phase. In reality though, I'd bet that quite a bit of Perl 5 code will continue to be run via Pony for many years to come. If Pony works, and all that old Perl 5 code still does its job, rewriting it to take advantage of Perl 6 features may become the ultimate back-burner project. Still, having the freedom to *not* update that old code seems like just another reason to use a language like Perl that's so committed to flexibility.

### Renew Now and Save!

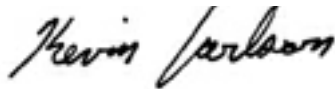
We want to thank you all for making this journal a success. To show our appreciation, we have a special offer for *TPJ* subscribers who renew their *TPJ* subscription between now and November 1.

We're offering you, for this limited time, the option to renew your *TPJ* subscription at a special low rate of \$16/year for one, or even, two years. When you do, you'll also get one year of access to BYTE.com at no extra charge. In addition to all the great Perl coverage you count on in *TPJ*, you'll get columns by Moshe Bar, Jerry Pournelle, Martin Heller, David Em, Andy Patrizio, and Lincoln Spector, plus news and views on a wide range of technology topics—and much, much more.

What's more, we'll also be providing something many of you have asked for—access to complete archives. Effective with next month's issue, you will have access to every issue of *TPJ*—from Spring 1996 to the present—at no extra charge.

A standard *TPJ* subscription is \$18/year. But if you renew now, you'll get the low rate of \$16/year, plus one year of access to BYTE.com. All current subscribers are eligible—even if you just subscribed recently, you can still take advantage of this special offer to extend your current subscription for up to two additional years!

Just visit <http://www.tpj.com/renewal/> to take advantage of this limited-time offer.



Kevin Carlson  
Executive Editor  
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to [editors@tpj.com](mailto: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: \$12.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2003 by CMP Media LLC, unless otherwise noted. All rights reserved.



## 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

### 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

#### WEB DEVELOPER

Bryan McCormick

#### WEBMASTERS

Sean Coady, Joe Lucca, Rusa Vuong

### 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 William Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, GROUP PUBLISHER TECHNOLOGY

SOLUTIONS Robert Faletta

PRESIDENT, GROUP PUBLISHER HEALTHCARE MEDIA

Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATION

TECHNOLOGY Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK

TECHNOLOGY Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA GROUP Peter Westernman

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

# Perl News

**T**he O'Reilly Open Source Convention 2003 in Oregon couldn't have come off better. Portland's river view is beautiful and the weather was never less than pleasant. Powell's Technical Books was an amazing warehouse of technical treasures. I had to constantly remind myself that anything I bought there, I would have to take home in a suitcase.

### Ponie

The big Perl news was the announcement of Ponie, the transitional Perl 5.10 that will be based on the Parrot virtual machine. During his State Of The Onion talk, Larry Wall explained how Ponie would provide a migration path from Perl 5 to Perl 6. People won't be able to convert to Perl 6 all at once, and need a way to migrate. Ponie will also "ensure the future of the millions of lines of Perl 5 code," he said.

Ponie's compatibility will be pretty slick. It will implement the current Perl internals in Parrot, which will mean compatibility with the XS API. Ponie will effectively emulate the Perl 5 virtual machine. Fotango, a London-based consultancy, is devoting 35 percent of the time of Arthur Bergman, the driving force behind Perl threads, to the Ponie effort, for at least the next two years.

Ponie: <http://www.ponicode.org/>

SOTO: <http://www.perl.com/pub/a/2003/07/16/soto2003.html>

Fotango: <http://opensource.fotango.com/>

### Wikis

The OSCON wiki was a huge success. A wiki is a web-based hypertext information store that lets anyone modify pages on the fly. The wiki was created weeks before the start of the convention, and was updated by literally hundreds of users. Some contributed only their names to the list of attendees, while others maintained elaborate lists of Birds-of-a-Feather sessions. It became the electronic equivalent of a bulletin board, but far more interactive.

Brian Ingerson has become quite an evangelist for wikis. His CGI::Kwiki has made a convert out of me, now that I've seen it in action. He's also created Test::Fit, for integrating Perl's testing framework into a wiki based on the Fit testing framework.

CGI::Kwiki: <http://search.cpan.org/dist/CGI-Kwiki/>

OSCON wiki: <http://oscon.kwiki.org>

Fit: <http://fit.c2.com/>

### Hot Topics

There was a lot of time and interest devoted to automated software testing. Besides the session I gave on the subject, Geoff Young talked about Apache::Test, Ward Cunningham and Brian Ingerson talked about Test::Fit, and Michael Schwern and chromatic's gave a half-day testing tutorial. It's a topic that will be critical to the Ponie project, since Perl 5 will be rewritten from the ground up, yet must work identically.

### TPF Auction

The Perl Foundation auction raised \$4500 for helping further Perl. Besides the expected signed copies of books, items up for bid included stuffed animals, Mark-Jason Dominus's magic pointy wizard hat, and the right to choose the color of search.cpan.org. The London.pm group loves orange, and there was an intense bidding war between the London.pm group, who wanted to change the color, and a group who wanted to keep search.cpan.org the same. When the smoke cleared, the antiorange group won the auction, but Graham Barr agreed to give in to a month of orangeness as a consolation.

### Meeting the People

Amidst all the talks and tutorials, the greatest benefit of OSCON is to meet fellow open sourcers face-to-face. These were people I knew and worked with online for years, and yet the immediacy of meeting and talking with them for a week was far more productive than a month of e-mail or IRC. This sort of mingling is one way to foster the opportunities for cross-pollination that make open-source languages so vital. In fact, I made a point of thanking Guido von Rossum and Matz, of Python and Ruby, respectively, for the innovations they were bringing to the language table for Perl to steal from. In the open-source world of language innovation, a rising tide truly does lift all boats. I was disappointed that there was only a small commons area with a dozen small tables for meeting one's peers. The next OSCON needs dedicated mingling space.

All this makes clear the importance of groups like local Perl Monks chapters, and of getting to work with your comrades-in-code. Especially interesting was my realization of the impact that London.pm has on the Perl community, specifically on Perl 6. On the flight home, I envied the London.pm group, and thought about what marvelous things could be done if my own Chicago.pm could pull together and make things happen.

—Andy Lester

# Implementing the Khazad Block Cipher in Perl

**K**hazad is a new 64-bit block cipher that accepts a 128-bit key. Invented by Paulo S.L.M. Barreto and Vincent Rijmen, it is a finalist in the New European Schemes for Signatures, Integrity, and Encryption (NESSIE) Project.

In this article, I will show you how I implemented Khazad in Perl using XS programming. Using the techniques presented here, I hope that you will also be able to implement any block cipher that you fancy (or, perhaps, just create an XS-based Perl module).

At this writing, the current version of `Crypt::Khazad` is 1.0.3. It is available on CPAN at <http://www.cpan.org/authors/id/J/JC/JCDUQUE/Crypt-Khazad-1.0.3.tar.gz>. Files mentioned in this article are available in this module distribution. Also see <http://planeta.terra.com.br/informatica/paulobarreto/KhazadPage.html>

## Using h2xs

The first thing to do is create a subdirectory where the module will be created and stored using the program `h2xs`, which converts C header files to Perl extensions. On the command line, type:

```
h2xs --omit-autoload --omit-constant --name=Crypt::Khazad
```

Refer to the `h2xs` man pages for explanation of the options `--omit-autoload` and `--omit-constant`. What's important here is the option `--name=Crypt::Khazad`. This tells `h2xs` to create a subdirectory `Crypt/Khazad/`.

The following files will be created in `Crypt/Khazad/`:

```
Changes
Khazad.pm
Khazad.xs
MANIFEST
Makefile.PL
README
ppport.h
t
```

“t,” which stands for tests, is a directory containing test scripts. For the moment, there's only one test script found here (`1.t`). All test scripts are placed in this directory and must be suffixed with `‘.t’`.

## The typemap File

Change directories (`cd`) into `Crypt/Khazad`. What is lacking in this directory is the typemap file. Create this file, and add the following line:

*Julius is a freelance network consultant in the Philippines. He can be contacted at [jcdueque@lycos.com](mailto:jcdueque@lycos.com).*

```
Crypt::Khazad T_PTROBJ
```

Note that the whitespace separating `Crypt::Khazad` and `T_PTROBJ` must be a tab.

The typemap file contains the mapping of `Crypt::Khazad`, an arbitrary data type, to its corresponding Perl value, `T_PTROBJ`, a type representing a pointer to a structure. Thus, we can set up an encryption/decryption routine like this:

```
use Crypt::Khazad;

$cipherobj = new Crypt::Khazad $key;
$ciphertext = $cipherobj->encrypt($plaintext);
$plaintext = $cipherobj->decrypt($ciphertext);
```

Given an argument, `$key`, the `new()` function will create an object, `$cipherobj`, of type `Crypt::Khazad` (the data type we just declared in the typemap file).

Essentially, `new()` clones the object `Crypt::Khazad`. In most object-oriented languages, `new()` is already a built-in function. Unfortunately, there is no equivalent function in Perl. So, we have to create one ourselves. Since we are free to implement this function in any way we want, we could just as easily name it, say, `clone()`, and we would still get the same result.

Now, comes the fun part: XS programming!

## XS Programming

Another missing file in `Crypt/Khazad/` is the heart of the module, the Khazad C code. In our case, that file is named `_khazad.c` (it's available in the `Crypt::Khazad` module distribution on CPAN). So copy this file to `Crypt/Khazad/`.

The best way to understand C code is to study its `main()` function. In `_khazad.c`, we see how `main()` uses the other functions to perform the process of encryption and decryption. (See Listing 1.)

In Line 3, we see that variable `subkeys` is declared as data type `NESSIEstruct`. Lines 14–16 tell us that we have to initialize `subkeys` with a call to `NESSIEkeysetup()` before calling the functions `NESSIEencrypt()` for encryption, and `NESSIEdecrypt()` for decryption.

Now, open the file `Khazad.xs`. Initially, its content is just:

```
1 #include "EXTERN.h"
2 #include "perl.h"
3 #include "XSUB.h"
4
5 #include "ppport.h"
6
```



```
7 MODULE = Crypt::Khazad PACKAGE = Crypt::Khazad
```

Edit Khazad.xs so that it becomes:

```
1 #include "EXTERN.h"
2 #include "perl.h"
3 #include "XSUB.h"
4
5 #include "ppport.h"
6
7 #include "_khazad.c"
8
9 typedef struct khazad {
10     NESSIEstruct key;
11 }* Crypt__Khazad;
12
13 MODULE = Crypt::Khazad PACKAGE = Crypt::Khazad
```

Notice that we have now `#include-d` the C code, `_khazad.c` (line 7). What about lines 9–11, starting with `typedef struct khazad`? Because the C code declares `subkeys` as type `NESSIEstruct` (line 3 of Listing 1), we have to declare the variable `key` in Khazad.xs (line 10) as type `NESSIEstruct` as well.

But there’s a catch. When `_khazad.c` is compiled, everything is fine. But when the XS file is compiled by `xsubpp`, it complains. In fact, you are likely to get the following fatal errors:

```
Khazad.c: In function `XS_Crypt__Khazad_new':
Khazad.c:64: `Crypt__Khazad' undeclared (first use in this function)
Khazad.c:64: (Each undeclared identifier is reported only once
Khazad.c:64: for each function it appears in.)
```

The solution is to look for the `NESSIEstruct` line in `_khazad.c`, and check how `NESSIEstruct` is declared. That code snippet is as follows:

```
struct NESSIEstruct {
    u32 roundKeyEnc[R + 1][2];
    u32 roundKeyDec[R + 1][2];
};
```

The `xsubpp` compiler does not like this code. So, revise this code into this:

```
typedef struct NESSIEstruct {
    u32 roundKeyEnc[R + 1][2];
    u32 roundKeyDec[R + 1][2];
} NESSIEstruct;
```

That will keep the `xsubpp` compiler happy.

## Ensuring Crypt::CBC Compliance

`Crypt::CBC` is a module developed by Lincoln Stein to be used specifically in conjunction with block ciphers. As with all Perl modules, `Crypt::CBC` is also available from <http://search.cpan.org/>.

To be `Crypt::CBC` compliant, our block cipher must be able to return the block size it is using, as well as the length of its key, when `Crypt::CBC` asks for them. Edit Khazad.xs, adding the following lines:

```
int
keysize(...)
CODE:
    RETVAL = 16;
OUTPUT:
    RETVAL

int
```

```
blocksize(...)
CODE:
    RETVAL = 8;
OUTPUT:
    RETVAL
```

Our new Khazad.xs is transformed as shown in Listing 2.

The keyword `int` must be on a line by itself. `RETVAL` stands for “return value” and this is the data that `Crypt::CBC` needs. Read the `perlxs` man pages for more details. The `keysize()` function returns a value of 16 bytes (the 128-bit key), while `blocksize()` returns 8 bytes (the 64-bit block length).

If we have the following code snippet:

```
use Crypt::Khazad;

$cipherobj = new Crypt::Khazad $key;
$ks = $cipherobj->keysize();
$bs = $cipherobj->blocksize();
```

`$ks` should hold the value 16, and `$bs` should be 8.

To implement the `new()`, `encrypt()`, and `decrypt()` functions, we must be familiar with `perlguts` and `perlxs`.

Edit Khazad.xs. It should now look like Listing 3. You’ll want to consult the `perlguts` man pages for discussions of the following keywords:

```
SV*
STRLen
SvPOK
SvCUR
NewZ
SvPV_nolen
SvPV
newSVpv
```

A couple of important things are worth mentioning here. The `new()` function actually calls the C function `NESSIEkeysetup()`. Similarly, `encrypt()` and `decrypt()` execute the C functions `NESSIEencrypt()` and `NESSIEdecrypt()`, respectively. Just refer to the `main()` function of `_khazad.c` (Listing 1) to learn how these functions are used.

The `DESTROY()` function is special. This function is called when the object, `Crypt::Khazad`, goes out of scope and needs to be destroyed. If the `DESTROY()` function does not exist, then nothing is done, and a memory leak could occur.

## The Khazad.pm File

Delete the `Khazad.pm` generated by `h2xs`. We will make our own from scratch. Listing 4 is our new `Khazad.pm`.

It is worth noting that variable `@EXPORT_OK` (line 7) should list only those functions that we want to be visible from outside of the Khazad module. Line 7 tells us that `keysize()`, `blocksize()`, `new()`, `encrypt()`, and `decrypt()` are the only functions of Khazad that are accessible by other modules.

## The Makefile.PL File

Delete the generated `Makefile.PL`. We’ll make our own instead. The following is our hand-crafted `Makefile.PL`:

```
1 use ExtUtils::MakeMaker;
2
3 WriteMakefile(
4     'NAME' => 'Crypt::Khazad',
5     'VERSION_FROM' => 'Khazad.pm',
6     'PREREQ_PM' => {},
7     'AUTHOR' => 'Julius C. Duque',
8     'LIBS' => [],
```

```

9  'DEFINE' => '',
10 'INC' => '-I.',
11 'dist' => {'COMPRESS' => 'gzip -9f', 'SUFFIX' => 'gz'}
12 };

```

Please read the perlxsut man pages for more details.

## Writing a Test Script

Every Perl module should be accompanied by at least one test script. A sample test script is shown in Listing 5. Place this script in the “t” directory. The test script must be suffixed with a “.t.” (If you want to know more on how to make test scripts, read up on Test::More.)

## Preparing the Distribution

This is the easiest part. Just type the following:

```

perl Makefile.PL
make manifest
make dist

```

The module is now bundled as a \*.tar.gz file.

## How to Use Khazad

An example of using Khazad in a Crypt::CBC-compliant code is shown in Listing 6. The output should be:

# The Khazad Block Cipher

by Paulo S.L.M. Barreto  
and Vincent Rijmen

**K**hazad is a 64-bit block cipher that accepts a 128-bit key. The cipher is a uniform substitution-permutation network (SPN) whose inverse only differs from the forward operation in the key schedule. The overall cipher design follows the Wide Trail Strategy, favors component reuse, and permits a wide variety of implementation tradeoffs.

## Block Ciphers

Block ciphers are important elements in many cryptographic systems. They are most often used to protect the secrecy of information, but are also used for generating pseudorandom numbers and to protect the authenticity of information.

Block ciphers are the modern equivalent of the old Caesar cipher. They substitute message blocks of a fixed length with ciphertext blocks of the same length. The substitution is controlled by the key, a secret parameter that is known only to the sender and the receiver. Someone who doesn’t know the key can’t reverse the substitution.

The security of block ciphers is not based on number-theoretic problems such as factoring. Consequently, they don’t require large keys: Currently, 80-bit keys are considered enough for commercial applications, and 128-bit keys will protect even the most valuable information against the most determined attacker.

## Wide Trail Strategy

The Wide Trail Strategy was developed in the 1990s by Joan Daemen. It applies to all symmetric-cryptography algorithms, but its biggest success was the AES block cipher Rijndael, which became a standard of the US Federal Administration (FIPS-197). The strategy allows designing secure ciphers in a modular way, by specifying a number of different properties that have to be present in different components. For instance, one of the components has to achieve diffusion, another component has to achieve nonlinearity, and so on. The properties are defined in a very specific, mathematical way.

*Paulo works at Laboratório de Arquitetura e Redes de Computadores (LARC), Departamento de Engenharia de Computação e Sistemas Digitais, Escola Politécnica da Universidade de São Paulo, Brazil, and can be contacted at pbarreto@larc.usp.br. Vincent works at the Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology in Graz, Austria. He can be contacted at vrijmen@iaik.at.*

## Khazad

Khazad is a block cipher that processes its message input in blocks of 64 bits and that accepts a key of 128 bits. It is a finalist cryptographic primitive of the NESSIE project, and has been so named as a cryptic reference to J.R.R. Tolkien’s *The Lord of the Rings*.

Khazad has many similarities to the AES block cipher Rijndael. Both ciphers were designed according to the Wide Trail Strategy, and consist of a number of iterations of a transformation called a “round.” Each round is, itself, composed of certain fundamental mathematical operations: *ByteSub*, which replaces each data byte by a prescribed value; *MixColumn*, which combines different bytes within the data block and is responsible for information diffusion; and *AddRoundKey*, which mixes the data with secret information, the so-called *round subkey*, derived from the key by means of a key-schedule algorithm. The ciphers also apply an extra *AddRoundKey* before the first round, and omit *MixColumn* in the last round. The round structure of Khazad is shown in Figure 1.

```

AddRoundKey(K0)
for r ← 1 to 7 do {
    ByteSub
    MixColumn
    AddRoundKey(Kr)
}
ByteSub
AddRoundKey(K8)

```

Figure 1: Khazad round structure.

Rijndael and Khazad have in common that all the components are based on mathematical functions over finite fields, instead of the more usual integer arithmetic modulo  $2^{32}$  or floating-point arithmetic. The use of finite field mathematics allows elegant constructions and proofs of properties that are relevant for the security of the ciphers.

A primary design principle of Khazad is that all algorithm components apart from the key schedule are involutions, that is, mathematical transformations that are their own inverses. This involutory structure is important to obtain efficient implementations. It also makes encryption and decryption equivalent operations except for the key schedule, which means that encryption and decryption are equally secure. (There are ciphers where these operations have different security levels.)

TPJ

```
plaintext1 : 0123456789abcdefedcba9876543210
ciphertext : 1be2bbl7b1bfa4227e33b06cf45c2d0f942f14a5b414e41
plaintext2 : 0123456789abcdefedcba9876543210
```

## Other Crypt Modules

In addition to *Crypt::Khazad*, I have created several other block cipher modules. All of these are available on CPAN:

```
Crypt::Mistyl
Crypt::Anubis
Crypt::Noekeon
Crypt::Skipjack
Crypt::Camellia
Crypt::Square
```

## References

The following man pages are essential reading: `h2xs`, `perlxsut`, `perlx`, and `perlguts`.

The following books also got me going: *Perl 5 How-To* by M.

Glover, A. Humphreys, and E. Weiss (Waite Group; ASIN 1571690581).

*Perl Cookbook*. T. Christiansen and N. Torkington (O'Reilly & Associates; ISBN 1565922433).

## Acknowledgments

I am indebted to Marc Lehmann for his *Crypt::Twofish2* module. I used his module as a skeleton for the Khazad implementation. Many thanks also go to the inventors of Khazad, Paulo S.L.M. Barreto and Vincent Rijmen. The Khazad home page is at <http://planeta.terra.com.br/informatica/paulobarreto/KhazadPage.html>.

TPJ

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

### Listing 1

```
1 int main(void)
2 {
3     struct NESSIEstruct subkeys;
4     u8 key[KEYSIZEB];
5     u8 plain[BLOCKSIZEB];
6     u8 cipher[BLOCKSIZEB];
7     u8 decrypted[BLOCKSIZEB];
8
9     int v;
10
11     printf("Test vectors -- set 1\n");
12     printf("=====\n");
13
14     NESSIEkeysetup(key, &subkeys);
15     NESSIEencrypt(&subkeys, plain, cipher);
16     NESSIEdecrypt(&subkeys, cipher, decrypted);
17
18     /***** more lines follow *****/
```

### Listing 2

```
1 #include "EXTERN.h"
2 #include "perl.h"
3 #include "XSUB.h"
4
5 #include "ppport.h"
6
7 #include "_khazad.c"
8
9 typedef struct khazad {
10     NESSIEstruct key;
11 }* Crypt__Khazad;
12
13 MODULE = Crypt::Khazad PACKAGE = Crypt::Khazad
14
15 int
16 keysize(...)
17 CODE:
18 RETVAL = 16;
19 OUTPUT:
20 RETVAL
21
22 int
23 blocksize(...)
24 CODE:
25 RETVAL = 8;
26 OUTPUT:
27 RETVAL
```

### Listing 3

```
1 #include "EXTERN.h"
2 #include "perl.h"
3 #include "XSUB.h"
4
5 #include "ppport.h"
6
```

```
7 #include "_khazad.c"
8
9 typedef struct khazad {
10     NESSIEstruct key;
11 }* Crypt__Khazad;
12
13 MODULE = Crypt::Khazad PACKAGE = Crypt::Khazad
14
15 int
16 keysize(...)
17 CODE:
18 RETVAL = 16;
19 OUTPUT:
20 RETVAL
21
22 int
23 blocksize(...)
24 CODE:
25 RETVAL = 8;
26 OUTPUT:
27 RETVAL
28
29 Crypt::Khazad
30 new(class, rawkey)
31 SV* class
32 SV* rawkey
33 CODE:
34 {
35     STRLEN keyLength;
36     if (! SvPOK(rawkey))
37         croak("Error: Key must be a string scalar!");
38
39     keyLength = SvCUR(rawkey);
40     if (keyLength != 16)
41         croak("Error: Key must be 16 bytes long!");
42
43     Newz(0, RETVAL, 1, struct khazad);
44     NESSIEkeysetup(SvPV_nolen(rawkey), &RETVAL->key);
45 }
46
47 OUTPUT:
48 RETVAL
49
50 SV*
51 encrypt(self, input)
52 Crypt::Khazad self
53 SV* input
54 CODE:
55 {
56     STRLEN blockSize;
57     unsigned char* intext = SvPV(input, blockSize);
58     if (blockSize != 8) {
59         croak("Error: Block size must be 8 bytes long!");
60     } else {
61         RETVAL = newSVpv("", blockSize);
62         NESSIEencrypt(&self->key, intext, SvPV_nolen(RETVAL));
63     }
64 }
65
66 OUTPUT:
67 RETVAL
68
```

```

69  SV*
70  decrypt(self, input)
71      Crypt::Khazad self
72      SV* input
73      CODE:
74      {
75          STRLEN blockSize;
76          unsigned char* intext = SvPV(input, blockSize);
77          if (blockSize != 8) {
78              croak("Error: Block size must be 8 bytes long!");
79          } else {
80              RETVAL = newSvPv("", blockSize);
81              NESSIEdecrypt(&self->key, intext, SvPV_nolen(RETVAL));
82          }
83      }
84
85      OUTPUT:
86          RETVAL
87
88      void
89      DESTROY(self)
90          Crypt::Khazad self
91          CODE:
92          Safefree(self);

```

#### Listing 4

```

1  package Crypt::Khazad;
2
3  use strict;
4  use warnings;
5  require Exporter;
6
7  our @EXPORT_OK = qw(keysize blocksize new encrypt decrypt);
8  our $VERSION = '1.0.0';
9  our @ISA = qw(Exporter);
10
11  require XSLoader;
12  XSLoader::load('Crypt::Khazad', $VERSION);
13
14  # Preloaded methods go here.
15
16  1;
17
18  __END__
19
20  =head1 NAME
21
22  Crypt::Khazad - Crypt::CBC-compliant block cipher
23
24  =head1 ABSTRACT
25
26  Put abstract here.
27
28  =head1 SYNOPSIS
29
30      use Crypt::Khazad;
31      $cipher = new Crypt::Khazad $key;
32      $ciphertext = $cipher->encrypt($plaintext);
33      $plaintext = $cipher->decrypt($ciphertext);
34
35  =head1 DESCRIPTION
36
37  Put description here.
38
39  =head1 COPYRIGHT AND LICENSE
40
41  Put copyright notice here.
42
43  =cut

```

#### Listing 5

```

1  use diagnostics;
2  use strict;
3  use warnings;
4  use Test::More tests => 2;
5  BEGIN {
6      use_ok('Crypt::Khazad')
7  };
8
9  BEGIN {
10     my $key = pack "H32", "80000000000000000000000000000000";
11     my $cipher = new Crypt::Khazad $key;
12     my $plaintext = pack "H16", "0000000000000000";
13     my $ciphertext = $cipher->encrypt($plaintext);
14     my $answer = unpack "H*", $ciphertext;
15     is("49a4ce32ac190e3f", $answer);
16 };

```

#### Listing 6

```

1  #!/usr/local/bin/perl
2
3  use diagnostics;
4  use strict;
5  use warnings;
6  use Crypt::CBC; # CBC automatically loads Khazad for us
7
8  my $key = pack "H32", "00112233445566778899aabbccddeeff";
9  my $IV = pack "H16", "0102030405060708";
10
11  my $cipher = Crypt::CBC->new({'key' => $key,
12                                'cipher' => 'Khazad',
13                                'iv' => $IV,
14                                'regenerate_key' => 1,
15                                'padding' => 'standard',
16                                'prepend_iv' => 0
17                                });
18
19  my $plaintext1 = pack "H32", "0123456789abcdeffedcba9876543210";
20  print "plaintext1 : ", unpack("H*", $plaintext1), "\n";
21
22  my $ciphertext = $cipher->encrypt($plaintext1);
23  print "ciphertext : ", unpack("H*", $ciphertext), "\n";
24
25  my $plaintext2 = $cipher->decrypt($ciphertext);
26  print "plaintext2 : ", unpack("H*", $plaintext2), "\n";

```

TPJ

## Renew Now & Save!

### Plus A Special Offer for Current *TPJ* Subscribers!

The time for subscription renewal is **NOW**—and we have a special offer for **all** current subscribers!

- You can lock in next year's subscription (and the year beyond that, too!) at the low rate of \$16/year by renewing **now**!
- Have access to the complete *TPJ* archives—Spring 1996 to the present! (Effective September 1, 2003.)
- And by renewing between now and November 1, 2003, you also get one year of access to **BYTE.com** at **no extra charge**!

That's a savings of nearly \$20—and you get *BYTE* columns by Moshe Bar, Jerry Pournelle, Martin Heller, David Em, Andy Patrizio, and Lincoln Spector, plus features on a wide range of technology topics.

Currently, all new subscriptions to *The Perl Journal* are \$18/year. But to show our appreciation for your support in our first year of publication, we're making this special limited-time offer available to current *TPJ* subscribers who renew between now and November 1.

**Don't miss out on a single issue or this special offer! Renew now!**

**One low price! One great deal!**

**<http://www.tpj.com/renewal/>**

***The Perl Journal***



# Programming Persistent Objects with *Class::PObject*

Ever wished you could just forget about database-specific routines to manage data files, BerkelyDB, MySQL, PostgreSQL, and the like, and use pure Perl for managing your application data? Or have you ever written a program with a specific database in mind, and had to port the application to another database and wished there was an easier way?

The philosophy behind persistent objects addresses just this issue—to represent persistent data (otherwise stored on disk) as a software object, and provide methods and behaviors for manipulating its content.

## What is a Persistent Object?

A persistent object can be thought of as another way of representing data on disk. It may help to think of a persistent object as a single row of a database table. It can look something like Example 1.

The whole record represents an object, and each column of the record represents attributes of the object. You use object methods from within your programs to create or access the data without having to run any database-specific queries.

What is the advantage? By treating the real data as an object, we achieve a higher level of database abstraction. This allows the development of platform-independent code, since our programs don't really care anymore how the data is stored and retrieved from the disk. It's up to the object drivers to perform these tasks transparently. Data, on the other hand, can be stored in plain files, BerkelyDB, Comma Separated Values (CSV), MySQL, or any other database system for which an object driver is available.

This design also makes development and maintenance easier, because maintainers will not need to learn implementation-specific database queries, but will only work with software objects. *Class::PObject* provides this framework.

## Programming Style

The programming style of *Class::PObject* resembles that of standard *Class::Struct*, and allows us to create objects dynamically. *Class::PObject* imports a single function, *pobject()*, which takes arguments describing the class attributes.

In this section, we will tackle a real-world task: managing user accounts in a web project.

Let's try to think of a user as an object, and decide what attributes a single user can have. The minimum required attributes

for a user might be *name*, *password*, and *email*. We'll stick to these three attributes for the sake of simplicity, and try to extend them later. The user can also have some behaviors. The most important of those behaviors is the ability to identify himself to our web site, a process also known as "login."

## Declaring a Class

In this section, we will build a *User* class. There are two ways of declaring a class using *Class::PObject*: inside a dedicated .pm file or in an inline declaration. Inline declaration is the easiest, and you can declare it in anywhere, even inside your programs. Here is an example of declaring a *User* class using the inline syntax:

```
pobject User => {  
    columns => ['id', 'name', 'password', 'email']  
};
```

Unfortunately, objects created this way will not be accessible to other programs because they are embedded inside your program. A better way is to define objects in their dedicated class files. To do this, create a *User.pm* file, with the following content:

```
package User.pm  
use strict;  
use Class::PObject;  
pobject {  
    columns => ['id', 'name', 'password', 'email']  
};  
1;
```

Although it might be hard to believe, the aforementioned examples create a fully functional *User* class. The *columns* in the aforementioned class definitions are attributes of the object that will be stored into disk. In other words, think of it as columns of a record stored in a database table. (Another example *User.pm* file is available online at <http://www.tpj.com/source/>.)

user:			
id	name	password	email
1	Sherzod	secret	sherzodr@handalak.com

Example 1: Imaginary *User* table representing a persistent object.

*Sherzod is a student at Central Michigan University. He is the author of several CPAN libraries, including Class::PObject. He can be reached at sherzodr@cpan.org.*

## The Generated *object*

The generated *object* provides the following methods:

- ***new()***—Constructor for creating a new object. Calling this method without any arguments will create an empty *User* object.
- ***load([\$id] [/[%terms]] [, %args])***—Constructor for loading object(s) from disk.
- ***save()***—For storing the changes made to the object back to disk.
- ***remove()***—Removing the object from the disk.
- ***remove\_all()***—Static class method for removing all the objects of the same type from the disk.

It also creates accessor methods for each of the declared columns. These methods carry the same name as column names. You can change the value of the column by passing an argument. To access the current value of the column, you call it without an argument. For example, to get the user's name, we can say:

```
my $name = $user->name();
```

To rename the user, we say:

```
$user->name("Sherzod");
```

## Create a New Object

Now let's create a new user account:

```
use strict;
require User;
my $user = new User();
$user->name('Sherzod');
$user->password('secret');
$user->email('sherzodr@handalak.com');
$user->save();
```

Notice we're loading the *User* class and creating an instance of a *User* object using the *new()* constructor. This will create an empty user. In consecutive lines, we are defining the user's attributes. When we're done, we save the object. *save()* will flush all the in-memory data to disk.

At this point, don't worry about how and where the above data is being stored.

## Loading Previously Stored Objects

All the objects should have a unique ID that distinguishes them from other objects of the same type. This ID is a primary key attribute of the object, called "*id*" by default. *Class::PObj* will make sure that every object will be assigned a unique, autoincrementing ID.

The most efficient way of loading an object is through its ID. We pass it to the *load()* method:

```
my $user = User->load($id);
```

You can also load objects by specifying a set of terms. These terms can be passed to *load()* as the first argument in the form of hash reference. Consider the following example, which is loading a user account with *name* "Sherzod."

```
my $user = User->load({name=>"Sherzod"});
```

We can now perform some modifications on this user. Say, we want to change the password:

```
$user->password('top_secret');
$user->save();
```

None of the modifications to the object will be flushed to disk unless we call *save()*.

## Loading Multiple Objects

If you call the *load()* method in array context, it will return a list of all the results matching your terms. If no terms are given, all the objects of the same type will be returned:

```
my @users = User->load();
```

Elements of *@users* hold all the *User* objects. Suppose we wanted to generate a list of all the users in our database:

```
for my $user ( @users ) {
    printf("[%03d] - %s <%s>\n", $user->id, $user->name, $user->email);
}
```

In addition to terms, you can pass the second argument to *load()* to perform such actions as sorting on a specific column, limiting the number of results returned, or returning spliced result sets:

```
# to return first 5 results ordered by 'name' column:
my @users = User->load(undef, {sort=>'name', limit=>5});

# to return results 10 through 25, sorting by 'name' in
# descending order:
my @users = User->load(undef, {sort => 'name', direction=> 'desc',
                             offset => 10, limit => 25 });
```

## Removing the Object

To delete the data an object is associated with, you can call the *remove()* method on that object. It is a nonreversible action, so use it only when you really mean it. The following example removes the user account with *id* 10:

```
my $user = User->load(10);
$user->remove();
```

and the following sample of code removes all the available *Users*:

```
my @users = User->load();
for my $user ( @users ) {
    $user->remove();
}
```

For most object drivers, removing all the data at once is a lot more efficient than removing them one by one, as we did in the aforementioned example. For this reason, *Class::PObj* also provides a *remove\_all()* method, which does exactly what its name claims:

```
User->remove_all();
```

Note that it is a static class method, and is not associated with any specific object. So saying something like this is not very intuitive:

```
my $user = User->load(); # <-- returns any one User object
$user->remove_all();     # <-- huh?
```

## Extending the Object's Interface

For some objects, accessor methods are not all you need. Many objects also need behaviors. A behavior is an action that an object can perform on its data. For example, an imaginary *Window* object could support behaviors such as *open()*, *close()*, *move()*, and so on. *authenticate()*, on the other hand, makes perfect sense for our *User* object. It's a procedure to be performed when a user submits a login form on a web page. The task of the *authenticate()* method is to validate the user's submitted *name* and *password* fields to the ones stored in the database. If they match, the user can enter.

To support this behavior, we need to open our *User.pm* file we created earlier, and define the *authenticate()* method:

```

package User;
use strict;
use Class::PObject;
pobject {
    columns => [
        'id', 'name', 'password', 'email'
    ]
};
sub authenticate {
    my $class = shift; # <-- removing Class name from @_
    my ($name, $passwd) = @_;
    return $class->load({name=>$name, password=>$passwd});
}
1;
__END__

```

Our *authenticate()* accepts two arguments, *\$name* and *\$passwd*, and returns a matching object. If no such account can be found, it will return *undef*. We now can use this new feature from within our programs like so:

```

# we take the form data using standard CGI.pm
my $name      = $cgi->param('name');
my $password  = $cgi->param('password');
# we then attempt to authenticate the user:
my $user = User->authenticate($name, $password);
unless ( defined $user ) {
    die "couldn't authenticate. Username and/or password are incorrect\n";
}

```

*authenticate()* is a class method, which makes sense, since we use it to retrieve a valid user object.

There is much more than that *Class::PObject* offers. For the details, you should always consult the latest library manual, available from your CPAN mirror.

## On Object Drivers

So far we have only discussed the programming syntax of *Class::PObject*. Now let's talk about how and where all the data is stored. Objects created using *Class::PObject* rely on object drivers for mapping objects into physical data.

The *User* class we have been working with so far was built using the most basic declaration. We didn't even tell it what driver to use, where and how the data should be stored, and so forth. In such cases, *Class::PObject* falls back to default driver, "file." To use any other drivers, you should define the *driver* class attribute. *datasource* can be defined to pass arguments for drivers:

```

package User;
pobject {
    columns => ['id', 'name', 'password', 'email'],
    driver => 'mysql',
    datasource => {
        DSN => 'DBI:mysql:users',
        UserName => 'shertzodr',
        Password => 'secret'
    }
};

```

In the following section, we'll give a brief overview of various drivers available for *Class::PObject*, and how they map the object data to disk files.

## File Driver

The *file* driver is a default driver used by *Class::PObject*. If *datasource* is provided, the driver will interpret it as a directory that objects will be stored in. If it's missing, it will create a folder in

your system's temporary directory. The name of the folder will be the lowercased version of the class name, with nonalphanumerics replaced with underscores ("\_").

The driver stores each object as a separate file, and uses *Data::Dumper* to serialize the data. The name of the file is in the form of "obj%05d.cpo," where "%05d" will be replaced with the object ID, with zeros padded if necessary.

As the number of objects in your database grows big, it will get less efficient and slower to manipulate objects in this form, since there will be lots of *open/close* and *eval()* calls.

---

*Class::PObject will make sure that every object will be assigned a unique, autoincrementing ID*

---

## CSV Driver

*csv* stores objects of the same type in a single file in CSV (Comma Separated Values) format. These objects can easily be loaded to your favorite spreadsheet application, such as Excel, or loaded to an RDBMS such as MySQL or PostgreSQL. It uses the *DBI* and *DBD::csv* libraries.

*csv* creates all the objects in your system's temporary folder unless you explicitly define *datasource* to be a *hashref* with the following keys:

- **Dir**—Directory where the object will be stored.
- **Table**—Name of the file that will hold this particular object. If you omit the *Table* name (recommended), it will default to the name of the object, nonalphanumerics underscored, and lowercased.

Here's an example using the *csv* driver:

```

package User;
pobject {
    columns => ['id', 'name', 'password', 'email'],
    driver => 'csv',
    datasource => {
        Dir => 'data/'
    }
};

```

## MySQL Driver

The *mysql* driver can be used for storing objects in MySQL tables. Unlike the aforementioned database drivers, you first need to set up the table structure to be able to use the *mysql* driver. In other words, each *pobject* represents a single database table. Each column of the table represents an attribute of the object and is declared in the *columns* array.

Consider the following object configuration for our *User* class using the *mysql* driver:

```

package User;
pobject {
    columns => ['id', 'name', 'password', 'email'],
    driver => 'mysql',
    datasource => {
        DSN => "dbi:mysql:users",

```

```

    UserName => "sherzodr",
    Password => "secret"
}
};

```

You can optionally provide *Table* inside *datasource* if you want to store the user data in a different table.

Now, let's create a table for storing the above object. Remember, you should create at least all the columns mentioned in the *columns* class attribute. Another thing to remember is to make *id* as an *AUTO\_INCREMENT* column. Consider the following table schema, which is designed for storing the just defined *User* objects:

```

CREATE TABLE user (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  name VARCHAR(30) NOT NULL,
  password CHAR(32) NOT NULL,
  email VARCHAR(80) NOT NULL,
  PRIMARY KEY(id)
);

```

Now, when we say:

```

my $u = new User();
$u->name('sherzodr');
$u->email('sherzodr@handalak.com');
$u->password('secret');
$u->save();

```

our MySQL table would then contain the following data as shown in Example 2.

```

mysql> select * from user;
+----+-----+-----+-----+
| id | name  | password | email                |
+----+-----+-----+-----+
| 1  | sherzodr | secret  | sherzodr@handalak.com |
+----+-----+-----+-----+

```

**Example 2:** Actual *User* table created in MySQL using the *mysql* object driver with *Class::PObject*.

## Conclusion

Persistent objects allow you to forget the messy details of data storage, and to treat your data like you would any other software object. I hope this article gives you enough of an understanding of creating persistent objects with *Class::PObject* for you to get started using them in your own code. (For more example code for this article, see <http://www.tpj.com/source/>.)

## References

**perlobj**—The Standard Perl manual of Perl objects.

**Class::PObject**—The official manual for *Class::PObject*. The latest features will be available through your CPAN mirror. <http://search.cpan.org/perldoc?Class::PObject/>.

**Class::DBI**—Another utility for programming persistent objects.

**Class::Struct**—The Standard Perl library for automating creation of simple classes using C++'s *struct*-like syntax.

TPJ

# Renew Now & Save!

## Plus A Special Offer for Current *TPJ* Subscribers!

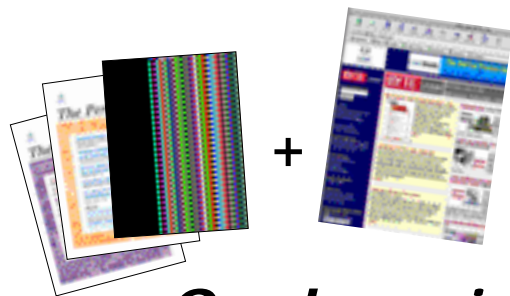
The time for subscription renewal is **NOW**  
—and we have a special offer for **all** current subscribers!

- You can lock in next year's subscription (and the year beyond that, too!) at the low rate of \$16/year by renewing **now**!
- Have access to the complete *TPJ* archives—Spring 1996 to the present! (Effective September 1, 2003.)
- And by renewing between now and November 1, 2003, you also get one year of access to **BYTE.com** **at no extra charge**!

That's a savings of nearly \$20—and you get **BYTE** columns by Moshe Bar, Jerry Pournelle, Martin Heller, David Em, Andy Patrizio, and Lincoln Spector, plus features on a wide range of technology topics.

Currently, all new subscriptions to *The Perl Journal* are \$18/year. But to show our appreciation for your support in our first year of publication, we're making this special limited-time offer available to current *TPJ* subscribers who renew between now and November 1.

**Don't miss out on a single issue or  
this special offer! Renew now!**



**= One low price!  
One great deal!**

<http://www.tpj.com/renewal/>

*The Perl Journal*



# Therapy Bots

I have always liked the ideas of bots (Internet robots) wading through the Internet and working for us humans. Things get especially interesting when you give bots the gift of artificial intelligence (AI).

For many people, AI is still synonymous with Eliza, the therapist-feigning program written by Josef Weizenbaum in the mid '60s. Weizenbaum's Eliza was an attempt at fooling the Turing Test. The test itself is fairly complex, but it is usually given in this simplified form:

A person sits in front of two terminals (screens with keyboards). One of these terminals lets the person communicate with a real flesh-and-blood human, the other with a computer program. The person can type any question to either/both of the two entities at the other end of the terminals—there is no restriction on what the person can ask! If, after a certain period (say, an hour), the person can't tell which terminal leads to the human and which to the computer, then the computer program could be referred to as intelligent.

Now, I often have to use IRC chat to manage my OpenSource project openMosix (see <http://www.openmosix.org/>). People usually have quite serious and intelligent discussions in our channel. But if you have ever logged into one of the more general chats, you might have noticed the almost total absence of intelligence in certain discussions. So, while computers have not really become more intelligent (in any substantial form) in these last 40 years, it seems that certain layers of society have collectively become less intelligent. One might be inclined to suspect that a computer might soon pass the Turing Test just because the human on the other side is more stupid than the program.

I decided to test this assertion by writing a Perl program to log into IRC channels and pretend to be a human, while I watch the resulting comical onslaught from my IRC chat client.

There are some ready-made bots in Perl out there on the Internet, but all the fun is in writing these things and seeing them run. There are also plenty of Perl modules which manage connections to IRC chats. So, if you have a favorite one, use that instead of the raw `IO::Socket` connections, which I prefer to use because they give me more control over what is happening with the connection.

---

*Moshe is a systems administrator and operating-system researcher and has an M.Sc. and a Ph.D. in computer science. He can be contacted at [moshe@moelabs.com](mailto:moshe@moelabs.com).*

So let's write a simple IRC bot. Usually, I start things with establishing a connection. Something like this:

```
#!/usr/bin/perl

use IO::Socket;

$sock = IO::Socket::INET->new(
    PeerAddr => 'irc.freenode.net',
    PeerPort => 6667,
    Proto => 'tcp' ) or die "could establish a connection";
```

Any public IRC server can be used here. I use `irc.freenode.net`. By standard, all connections go to port 6667-7000. If you have problems, try to find a different server on that network. To make things easier, you can make the `PeerAddr` a variable, which is specified by an argument from the command line.

Once we have established a connection to a server, we need to login to the IRC server. We use the `$sock` handle to communicate with the server. If you use a `telnet irc.freenode.net 6667`, you will find out what the server sends you upon a newly established connection:

```
bash-2.05a$ telnet irc.freenode.net 6667
Trying 65.39.204.5...
Connected to earth.peeringolutions.com.
Escape character is '^'.
NOTICE AUTH :*** Looking up your hostname...
NOTICE AUTH :*** Checking ident
NOTICE AUTH :*** No identd (auth) response
NOTICE AUTH :*** Found your hostname
```

The key here is the line with `NOTICE AUTH` in it. This is when we need to login to the irc server. To do this we send:

```
NICK bots_nick
USER bots_ident 0 0 :bots name
```

Remember to send a line break after the `bots_nick` and a line break at the end. So, in the `while` loop, we will add something like this:

```
while($line = <$sock>){
    print $line;
```

```

if($line =~ /(NOTICE AUTH).*(checking ident)/i){
    print $sock "NICK MosheBar\nUSER bot 0 0 :I am human\n";
    last;
}
}

```

If you run a simple program with the aforementioned code, you will almost certainly notice the server closing the connection after a while. Why? Because many servers will ask for a ping to make

## Chatbot::Eliza is a simple implementation of Weizenbaum's Eliza therapist

sure the client is active. Most ready-made bots you find on the Internet will stumble on this. To handle server-side ping requests, again, we need to analyze the IRC chat behavior with a telnet session. Notice that the server will ask for a ping before it asks about NICKSERV registration/identification, so we need to stop this loop after it mentions NICKSERV.

In the following code section, notice the numbers in the last *if* statement. They help us identify when we are ready to send our nickname incantation.

```

while($line = <$sock){
    print $line;
    #use next line if the server asks for a ping
    if($line =~ /^PING/){
        print $sock "PONG : " . (split(/ :/, $line))[1];
    }
    if($line =~ /(376|422)/i){
        print $sock "NICKSERV :identify nick_password\n";
        last;
    }
}

```

The `NICKSERV :identify nick_password<` line is a good example of an *irc* command. This is just a simple *irc* command.

The command is `NICKSERV` and the arguments are `identify nick_password` where `nick_password` is the actual password for

this nick. The line ends in a line break and all *irc* commands are in upper case. When there is a colon before something, it means it is a multiple word argument (it has spaces in it). This is how we will handle the possible ping and the *nickserv* identification.

Once we are done with identification and login, it's time to join a channel. You'll notice the IRC server printing out many lines of text. You join a channel by doing something like this:

```
print $sock "JOIN #channel\n";
```

Notice, there is no colon before `#channel`. This is because it does not have any spaces in it. And the `JOIN` command is in all caps. For a full list of commands try reading a tutorial on the IRC protocol.

All we need to do now is read the messages users send to the channel and respond to them. The format of a *priv\_msg* is as follows:

```

:nick!ident@hostname.com PRIVMSG #channel :the line of text
It is wise to separate them into variables.
:nick!$hostname $type $channel :$text

```

Let's not forget that most IRC servers will ping us from time to time, and we must reply with a PONG and the same characters we were pinged with. The routine in Listing 1 will handle that.

### Breathing Life into the Bot

Once we have the IRC chat part of our bot working, it is now time to imbue it with some intelligence. A good start is the Perl module *Chatbot::Eliza*, which provides a simple implementation of Weizenbaum's Eliza therapist. Listing 2 is a small program that shows how to use the module for interactive therapy.

It is slightly funnier to watch Eliza be its own therapist. Listings 3 and 4 show the server and client, respectively.

As you can see, the `$eliza->transform($ans)` line uses the module's capabilities to get an answer from Eliza based on a text string. We can use this capability with our bot to answer lines written by users in IRC channels. We split the chat messages into their components, and all we have to do is:

```
$msg = $eliza->transform($text)
```

and we have a reply ready to send back to the IRC channel.

I leave it to the reader to complete a bot with the components described in this article.

I have let the bot loose on a few IRC channels and sometimes it is really hard to distinguish which species, the human or the computer, is more intelligent. It seems Weizenbaum never anticipated that humans could fall so low when using new communication media.

TPJ

### Listing 1

```

while ($line = <$sock>) {
    ($command, $text) = split(/ :/, $line); # $text is the stuff from the
                                           #ping or the text from the
server
    if ($command eq 'PING'){
        #while there is a line break - many different ways to do this
        while ( (index($text, "\r") >= 0) || (index($text, "\n") >= 0) ){
            chop($text);
        }
        print $sock "PONG $text\n";
        next;
    }
    #done with ping handling

    ($nick,$type,$channel) = split(/ /, $line); #split by spaces

    ($nick,$hostname) = split(/!/, $nick); #split by ! to get nick and
                                           #hostname separate

    $nick =~ s/://; #remove ':'s
    $text =~ s/://;

```

```

#get rid of all line breaks. Again, many different ways of doing this.
$/ = "\r\n";
while($text =~ m#$/#){ chop($text); }

#end of parsing, now for actions
}

```

### Listing 2

```

Server.pl
#!/usr/bin/perl -w
use Chatbot::Eliza;
$SIG{INT} = sub { print "\nreceived signal!!! \n\n\n";
print "Do you wish to interrupt this program? [Y/n]: ";
$ans = <STDIN>;
if ($ans eq 'n' or $ans eq 'N') {exit 0};
};
$| = 1;

my $bot = Chatbot::Eliza->new;
$bot->command_interface();

```

### Listing 3

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

use strict;
use Chatbot::Eliza;
use IO::Socket::INET;

print ">> Therapy Server Program <<\n";
my $response;
my $eliza;
# Create a new socket
my $lsocket = IO::Socket::INET->new(
    LocalPort => 1111,
    Proto => 'tcp',
    Listen => 1,
    Reuse => 1
) or die "Cannot open socket $!\n";

my $socket = $lsocket->accept;
$eliza = new Chatbot::Eliza;
my $def_msg="\nReceiving message from client....\n";
print "Eliza is ready. What's troubling you? ";
my $text = '';
while( 1 )
{
    $socket->recv( $text , 256 );
    if( $text ne '' )
    {
        print "\npatient: $text \n" ;
        sleep 4;
        my $resp = $eliza->transform($text);
        $socket->send( $resp);
        print "\ndoctor: $resp \n";
        sleep 2;
    }
    # If client message is empty exit
    else
    {
        print "Client has exited!\n";

        exit 1;
    }
}
```

### Listing 4

```
#!/usr/bin/perl -w
use IO::Socket;
use Chatbot::Eliza;
$remote = IO::Socket::INET->new(
    Proto => "tcp",
    PeerAddr => "localhost",
    PeerPort => "1111",
)
or die "cannot connect to therapy port at localhost";

my $msg = "Are you my therapist?";
my $ans = " ";
my $eliza = Chatbot::Eliza->new;
while (1) {

    $remote->send($msg);
    $remote->recv($ans, 256);
    $msg = $eliza->transform($ans);
}
```

TPJ



# Renew Now & Save!

## Plus A Special Offer for Current *TPJ* Subscribers!

The time for subscription renewal is NOW  
—and we have a special offer for all current subscribers!

- You can lock in next year's subscription (and the year beyond that, too!) at the low rate of \$16/year by renewing **now!**
- Have access to the complete **TPJ** archives—Spring 1996 to the present! (Effective September 1, 2003.)
- And by renewing between now and November 1, 2003, you also get one year of access to **BYTE.com** at no extra charge!

Currently, all new subscriptions to *The Perl Journal* are \$18/year. But to show our appreciation for your support in our first year of publication, we're making this special limited-time offer available to current **TPJ** subscribers who renew between now and November 1.

**Don't miss out on a single issue or this special offer!  
Renew now!**

That's a savings of nearly \$20—and you get **BYTE** columns by Moshe Bar, Jerry Pournelle, Martin Heller, David Em, Andy Patrizio, and Lincoln Spector, plus features on a wide range of technology topics.



+ = **One low price!  
One great deal!**

<http://www.tpj.com/renewal/>

*The Perl Journal*



# Bryar: A New Weblogging Tool

*Simon Cozens*

In my last article, I mentioned that I maintain a *blosxom*-based weblog. *blosxom* is a very clever bit of Perl written by Rael Dornfest, and is designed to be simple, self-contained, and easy to get up and running. I like its simplicity, I like the way it uses flat text files, I like the basic idea of it, but there are some things about it I don't actually like. So after my last article got me thinking about blogging, I decided the time was right to write my own blog software. That weekend, I heard someone speak about "blossoms and briars," and a new project was born.

A lot of what I've been doing recently might be categorized as reinventing wheels—first the *Email::* project, and now this. But sometimes, it's necessary to reinvent wheels. If we didn't, to paraphrase Gary Burquist, this would be The Cobol Journal, we'd have leeches all over our bodies, we would be listening to just 8-track tapes. Wheel reinvention is the only way we get smoother wheels. But we don't do so lightly and we don't do it just for the sake of it. When we're reinventing wheels, it's important to think about what problems we're trying to solve and how we're going to improve on the existing technology.

So, while the basic ideas of *blosxom* were great, it had two big problems. First, the code is monolithic, difficult to follow, and difficult to extend; if, for instance, you want to shift from flat files to a database-backed storage system, you're basically out of luck and reduced to tearing up major parts of the program. Second, because the software is self-contained, the templates for critical parts of the output live in a *DATA* section at the end of *blosxom.cgi* (at least for the 1.x series of *blosxom*; I'm told things get slightly better in the new 2.x series, but I was already fed up with *blosxom* by this point). Editing these to customize the output can be awkward, and having done so makes upgrading difficult.

It's also important to realize that these things aren't defects in *blosxom* at all; they come about because *blosxom* is confirming to a particular set of design goals, and they are goals that I agree with and think are suitable for most people. Keep it simple, self-contained, easy to install, and easy to use. But I also felt that I had outgrown the boundaries of what I could do with *blosxom*, and so it was time for something new.

---

*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-cozens.org](mailto:simon-cozens.org).*

Let's first look at a user's perspective on the result of all this deliberation, and then we'll take a look at how some of it was done; in the next article, we'll take a deeper look at the construction of Bryar, and how it can be extended. Along the way, we'll learn a little about blogging with Bryar, program design, finding files with *File::Find::Rule*, and efficient list handling.

## Using Bryar

Installing Bryar is relatively simple. Not quite as simple as installing *blosxom*, but close. First, we have to grab all the modules we need from CPAN, and the best way to do this is with the *CPANPLUS* Perl module:

```
perl -MCPANPLUS -e 'install Bryar'
```

Once you've got *CPANPLUS* up and running—an essential component of any serious Perl site—the aforementioned command should install the Bryar code and its dependencies, and it should also tell you at some point:

You probably want to run *bryar-newblog* in a likely home for your blog once we've finished installing.

Let's do that now.

```
# mkdir -p /opt/blog
# chown simon:simon /opt/blog
% cd /opt/blog
% bryar-newblog
Setting up a Bryar blog in this directory
```

Done. Now you want to probably customize 'bryar.conf'. You should probably also customize *template.html*, *head.html* and *foot.html*. Then point your browser at *bryar.cgi*, and get blogging!

Now, just as with *blosxom*, we need to tell the web server to serve up that directory and treat *bryar.cgi* as a CGI script and the directory's index. Once we've done that, we can point to our browser at the relevant location—perhaps <http://localhost/blog/> and we should see something like Figure 1.

It's not pretty, but it works. As the message says, we should probably customize *template.html*, *head.html*, and *foot.html* to make it a bit prettier. Figure 2 shows what my blog looks like after a tiny bit of customization—deliberately constructed to make it very similar to *blosxom*'s default format. By default,



Bryar uses Template Toolkit to format posts, so regular readers of this column should know how to deal with the HTML files generated by bryar-newblog.

To make blog posts, we simply add files called *something.txt* in our data directory:

```
Title

<P>HTML goes here</P>
```

Bryar automatically picks these up, sorts them in date order, and formats them appropriately. In fact, I use a little script to ensure that I have a unique post ID every time I want to make a blog entry; here's my *bin/blog* command:

```
#!/usr/bin/perl
my $path = "/opt/blog/" . shift() . "/";
$blog[$_]++ for map { /(\d+)\.txt/ } <$path/*txt>;
system("vim", $path . @blog . ".txt");
```

The first line of code decides where to look for posts. The second is a sneaky way to find a new post number. My posts are all entered numerically (6065.txt, and so on). This line looks at all of the ".txt" files and builds an array of currently existing posts; if the highest numbered post we have currently is 1234.txt, it will set *\$blog[1234]* to 1. It'll also assign to a bunch of other elements of that array, but we don't care about those; we only care about the highest number in existence at the moment.

If *\$blog[1234]* is the highest numbered element, *@blog* itself will have 1235 elements, and thanks to the wonderful Mr. Cantor and his diagonal theorem, post 1235 is guaranteed not to be in use at the moment. So we start an editor on */opt/blog/1235.txt*—a brand new blog post.

For those of you who have embraced the RSS generation, Bryar can also generate an RDF file by adding "/xml" to the end of the URL. The POD documentation to *Bryar.pm* details all the other things that can be done with Bryar URLs.

Now that we have an idea of what Bryar does, let's have a look at how it does it.



Figure 1: Default Bryar blog layout.



Figure 2: Bryar blog after customization.

## Basic Design

I designed Bryar to have four major areas of operation; it turns out that there are possibly some more things it needs to do, but these can be worked out over time. By splitting Bryar's operation into these distinct areas, we enable it to be highly customizable by letting people replacing any or all of the classes that implement these operations.

The four things that Bryar has to do can be described as "interface," "retrieval," "collation," and "formatting." But since I didn't think in those terms when I was designing the code, we'll call them the *Frontend*, *DataSource*, *Collector*, and *Formatter* classes.

## Wheel reinvention is the only way we get smoother wheels

**Frontend**—The *Frontend* class deals with the interaction between Bryar and the outside world. As with *blosxom*, the primary interface mechanism is through the Common Gateway Interface; the URL and other options are determined from the execution environment, the final output is written to standard output, and so on. We can equally conceive of a component which implements *Bryar::Frontend* as a *mod\_perl* handler, for instance, or as a stand-alone program taking options from the command line.

**DataSource**—Blog entries have to come from somewhere; the *DataSource* class finds entries, finding only those entries fulfilling particular criteria, and turns them into a set of *Bryar::Document* objects that abstract postings away from the data source and into a common interface. Since we're emulating *blosxom*, the default *DataSource* class reads all the .txt files in a given data directory, and uses filesystem properties such as the last-modified time and the owner as post metadata. We could also take blog entries from a database and we'll show an example of this later in the article.

**Collector**—The *Collector* class has the job of interpreting the options obtained from the *Frontend*, turning them into a search query, and asking the *DataSource* class for all the documents that fulfill the query. For instance, we might return the last 20 posts, (the default operation) or the posts in a particular month or day, or posts containing a given phrase. This is possibly the most "stationary" class in Bryar, since it's difficult to imagine a good reason for changing the default behavior.

**Renderer**—Once the *Collector* has decided on a set of *Bryar::Document* objects to be displayed, they are handed off to the *Renderer* class. As mentioned earlier, we use the Template Toolkit by default, but there's no reason why we couldn't create *Renderer* classes that use *HTML::Template* or *HTML::Mason*.

Bryar's operation can be summarized in the flowchart shown in Figure 3.

Other satellite classes that turned out to be useful were *Bryar::Config*, which encapsulates the configuration, and *Bryar::Comment*, a subclass of *Bryar::Document* used for encapsulating comments on a blog posting.

We'll take a detailed look at two of these areas, those that are most likely to be customized—the data source and the front end.

## Data Sources

As I've mentioned, the job of the *DataSource* class is to turn our raw data into *Bryar::Document* objects. There are three methods we need to provide in order to do this: *all\_documents* should

return absolutely everything, *search* should return those documents matching specified search criteria, and *add\_comment* should record a comment against an article.

As it turns out, if we implement *all\_documents*, we don't need to implement *search* but can instead inherit from *Bryar::Data-Source::Base*. This base class provides a very dumb search facility that looks at every single *Bryar::Document* and sorts out those that match the search terms. This is very inefficient, though, since it's usually faster to do some kind of searching at the data-source level—for instance, if our posts are stored in a SQL database, we might as well use the capabilities of SQL SELECT to find posts within a certain time period or for a particular ID, rather than plough through individual posts.

Similarly, if we implement *search*, we don't actually have to implement *all\_documents*, as nothing in Bryar calls it (yet). On the other hand, it may be useful to do so—both for completeness, and as a warm up for writing *search*. It's also conceivable that, in the future, one might add extensions to, say, format an entire journal for printing as a PDF file.

Let's start by looking at *all\_documents* in our flat file data source. With the obscene comments removed, this looks like:

```
sub all_documents {
    my ($self, $bryar) = @_;
    croak "Must pass in a Bryar object"
        unless UNIVERSAL::isa($bryar, "Bryar");

    my $where = cwd;
    chdir($bryar->{config}->datadir);

    my @docs = map { $self->make_document($_) }
        File::Find::Rule->file()
            ->name("*.txt")
            ->maxdepth($bryar->{config}->depth)
            ->in(".");

    chdir($where);
    return @docs;
}
```

We're called as a class method and passed a *Bryar* object; this stores the configuration and current state of the blog, and essentially draws everything together. First, we need to determine where our data files live; this is stored in the Bryar config, which we access via *\$bryar->{config}*. Once we've found this, we change to that directory and start looking for files. I've used Richard Clamp's wonderful *File::Find::Rule* module which, as we'll see later, turns out to be almost purpose built for what we're about to do.

*File::Find::Rule* is a nicer way of finding files than the usual *File::Find*. The idea is that we chain together rules that describe what we're looking for, and finally tell it where to look. This acts a little like the UNIX *find(1)* command. So in this case, we look for all the *\*.txt* files in the current directory and subdirectories below, up to a maximum depth specified in the blog configuration. This allows us to have entries categorized into subdirectories included in the blog. We call the helper function *make\_document* on each one, which does the heavy work of turning a filename into a *Bryar::Document* object, and we return all these results.

*make\_document* is fairly uninteresting, being concerned with extracting metadata from a file; we saw in my last article how to do this for blossom-style files. Now we're warmed up and ready to go, let's move on to the more interesting *search* method, which stretches *File::Find::Rule* a little more.

## Finding Files By Rule

The *search* method takes a Bryar object as before, but it also takes a hash of things to look for. In particular, *id* finds a document with a particular ID; this is used to provide permanent links for articles, especially in the case of articles referenced from an RDF feed. *since* and *before* look for documents after and before par-

ticular UNIX epoch times; *contains* finds documents containing a particular word, and *subblog* looks for documents in a given sub-blog or category. Finally, the *limit* parameter is used to set a maximum number of entries to return. For instance, by default, the front page of a blog will show the 20 most recent articles.

Thanks to *File::Find::Rule*, we can actually construct a search that covers all but one of these search terms in one single call. Naturally, the more specific we can make a single search, the fewer passes over the data we need to do, and therefore, the more efficient our data source class turns out to be.

The first few lines of *search* should be reasonably obvious:

```
sub search {
    my ($self, $bryar, $params) = @_;
    croak "Must pass in a Bryar object"
        unless UNIVERSAL::isa($bryar, "Bryar");

    my $was = cwd;
    my $where = $bryar->{config}->datadir."/";
    if ($params{subblog}) { $where .= $params{subblog}; }
    chdir($where);
```

We find our data directory as before, and this time, if we're given a subdirectory name to look in: We start searching from there instead of the document root.

Now we start putting our query together. The most obvious thing to look for is the document ID. If we're trying to find blog post number 89, then it's quite dull to look for *\*.txt* and extract *"89.txt"*—we might as well just go straight for *"89.txt"*:

```
my $find = File::Find::Rule->file();
if ($params{id}) { $find->name("$params{id}.txt") }
    else { $find->name("*.txt") }
```

As before, we restrict our search to a maximum depth:

```
$find->maxdepth($bryar->{config}->depth);
```

And now the clever stuff starts. *File::Find::Rule* allows us to specify bounds for the last-modified time of a file, something that will help us to find entries within a given period:

```
if ($params{since}) { $find->mtime(">".$params{since}) }
if ($params{before}) { $find->mtime("<".$params{before}) }
```

Notice what's happening here—we're simply modifying the *\$find* object by adding constraints to it. It isn't hitting the filesystem at all yet, it's simply building up a data structure that determines how to perform the search.

Finally, we can look for items that contain a particular word or phrase; *File::Find::Rule* allows us to grep through the contents of a file using the aptly named *grep* method:

```
if ($params{content}) { $find->grep(qr/\bQ$params{content}\b/i) }
```

Are you comfortable with that regular expression? It says that we're looking for a word break, then the literal contents of *\$params{content}*, then another word break. The word breaks are there to ensure that a search for "pie" only finds articles that talk about "pie," and not those which talk about being "occupied" or similar; the *\Q* and *\E* make sure that we don't treat *\$params{content}* as a regular expression, but rather as literal text.

Why don't we let the users search using regular expressions? Although this would undeniably be powerful, we want to keep this relatively simple. Since some data sources, such as a SQL back end, won't support searching by regular expression, so we deliberately restrict the search to make it as efficient as possible.

## Setting a Limit

Now we are ready to actually launch our search and grovel around the filesystem. The only search term we have not dealt with is

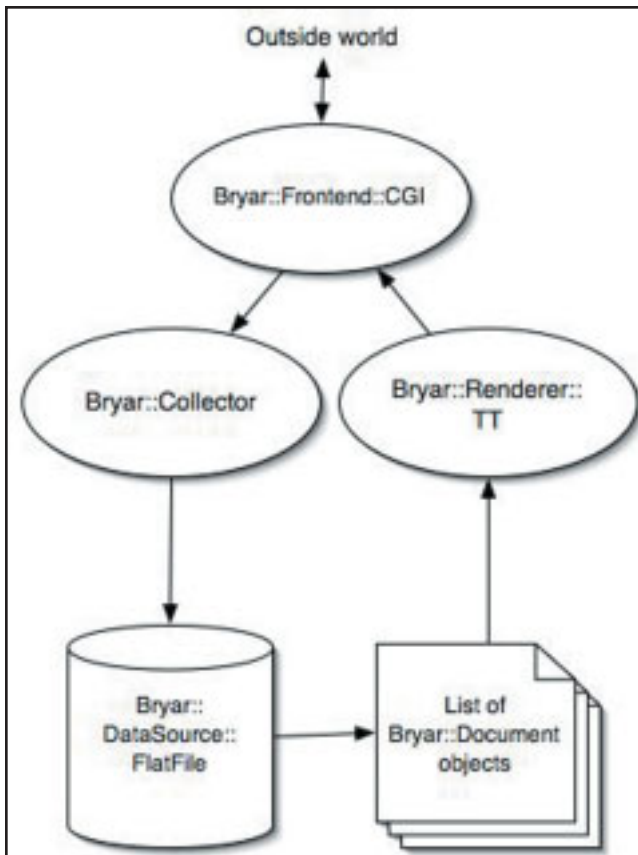


Figure 3: Flowchart of Bryar's operation.

*limit*, restricting the number of documents returned. The problem with this is that we can't force *File::Find::Rule* to return the results in any particular order. When we want at most 20 blog entries, we actually want the most recent 20, not just the first 20 we come across on the disk; these may not turn out to be the same. So, to implement *limit*, we need to do a little trickery. Let's first dispatch the case where there is no limit given:

```

if (!$params{limit}) {
    @docs = map { $self->make_document($_) } $find->in(".");
}

```

We simply start the search in the current directory, turn all the found files into documents, and we're done. That was easy. When there's a limit, we have to be a bit more careful:

```

@docs = map { $self->make_document($_) }
(
    map { $_->[0] }
    sort { $b->[1] <=> $a->[1] }
    map { [$_, ((stat $_)[9]) ] }
    $find->in(".")
) [0..$params{limit}-1];

```

What's going on here? Well, the core of it is the same as the unlimited case—we find the files in the current directory and turn some of them into *Bryar::Documents*. In the middle, though, we first have a well-known Perl sort technique, the Schwartzian transform:

```

@sorted_files =
    map { $_->[0] }
    sort { $b->[1] <=> $a->[1] }

```

```

map { [$_, ((stat $_)[9]) ] }
@files;

```

This is the most efficient way to sort a list of files newest-first. We could, of course, just say this:

```

@sorted_files = sort {
    my $a_modification = (stat $a)[9];
    my $b_modification = (stat $b)[9];
    $b_modification <=> $a_modification;
} @files;

```

Unfortunately, this makes two *stat* calls for every comparison, and every student of algorithms knows that Quicksort uses roughly  $n \log n$  comparisons. This means that, for 100 files, we have about 1000 system calls. This isn't great. Instead, we use the Schwartzian transform to reduce that to only 100 *stats*—one per file—can't get much better than that. For more details of how the Schwartzian works, see <http://www.cpan.org/doc/FMTEYEWTK/sort.html>.

Now that we have the sorted list of filenames, we use a slice to pull out the required number, and then pass these on to *make\_document*:

```

@docs = map { $self->make_document($_) }
@sorted_files[0..$params{limit}-1];

```

However, as we've seen, most of the heavy lifting has already been done by *File::Find::Rule*, so this turns out to be a reasonably efficient search routine, given that we're using a filesystem-based data source.

## Where to from Here?

We've said that *Bryar::DataSource::FlatFile* is reasonably efficient. But of course, we can make the whole process of retrieving posts even more efficient. In our next article, we'll look at how we can implement the same *DataSource* interface with a SQL back end using *Class::DBI* to retrieve posts from a database, and how this affects the way we can search for posts.

We'll also look at extending some of the other parts of Bryar as well. It's a tribute to Bryar's component-based design that we'll be able to turn it from a CGI-based flat-file blog tool using *Template::Toolkit* to a database-backed Apache-mod\_perl blog formatted by *HTML::Mason*. Tune in next time to find out how we do it—and in the meantime, happy blogging!

TPJ







# Learning Perl Objects, References, & Modules

*Russell J.T. Dyer*

**R**andal Schwartz, the author and coauthor of several columns and books on Perl, has written a new book with his seminar partner, Tom Phoenix. *Learning Perl Objects, References & Modules* is a more advanced book on learning Perl. If you enjoyed Schwartz's immensely popular book, *Learning Perl* (a.k.a, the Llama book), and you're ready to move up to a higher level in your Perl training, then you'll want to get a copy of this book. The Alpaca book (that's the signature O'Reilly cover animal for this book—it's a close relative to the Llama) will be of particular interest to you if you've been working with Perl by yourself and you would like to be able to work for an organization with many Perl programmers. This book will help you develop the skills that you will need to work in a large group on extensive, lengthy programs.

## Style & General Content

As with all of Schwartz's writings, the Alpaca book is written in a clear and relaxed style, with lighthearted examples. Although it's somewhat advanced, the lessons are easy for intermediate Perl programmers to follow and learn. This is not a book to be rushed, though, nor would I recommend jumping around in it. But if you pace yourself and read it from beginning to end—especially if you work through the exercises—you'll find it to be a fairly easy read and will be surprised at how much your Perl coding will improve. What's important is to experiment with the concepts presented.

*Learning Perl Objects, References & Modules* is composed of three major sections, not in the order of the title. The first section is on references. Typically, a reference is a memory address for where a data structure like a hash or an array is contained. They can be very useful in speeding up scripts. Instead of moving the actual data around within a program, one can simply capture the memory address or rather a reference to the data (which is only a small string of characters) and pass it around. The second section of Schwartz's new book is on object-oriented programming (OOP) with Perl. OOP is particularly useful for organizations with many programmers, writing thousands of lines of code. The third section deals with creating one's own Perl modules, either for private use or to contribute to CPAN. Private modules can assist

---

*Russell is a Perl programmer, MySQL developer, and web designer living and working on a consulting basis in New Orleans. He is also an adjunct instructor at a local college where he teaches Linux and other open-source software. He can be reached at [russell@dyerhouse.com](mailto:russell@dyerhouse.com).*

*Learning Perl Objects,  
References & Modules*  
Randal Schwartz with Tom Phoenix  
O'Reilly & Associates  
\$34.95  
ISBN: 0-596-00478-8

greatly in streamlining code. As you can see already just in this overview, the Alpaca book focuses on creating powerful, efficient code, while being mindful of other programmers who may use it.

## References Section

The first two chapters discuss fundamental (albeit advanced) concepts involved in writing large Perl programs with several other programmers. For one thing, it gives suggestions on watching out for unnecessary redundancy in your code. To eliminate redundancy, Schwartz discusses using subroutines within a program; to be able to reuse subroutines among many programs, he recommends libraries to house them. The elimination of the bulk of a program's clutter makes way for the use of references for further fine tuning and clean up.

The third chapter introduces the concept of references and illustrates how to create them. With references, one can build and manipulate complex data structures. For instance, suppose you have a program with a set of hashes that contain information on your company's clients. One hash contains a client's account number, name, address, telephone number, and so forth. Suppose also that you have several such hashes—one for each client. You could put all of these hashes together into an all encompassing hash by using references to each hash:

```
%clients = ('AC1234' => \%AC1234, 'AC1235' => \%AC1235, ...)
```

With only references to each client data hash contained in the values of each hash pair (the keys are account numbers), passing around a set of nested hashes like this poses no significant drain on resources. Chapter Four demonstrates how to



dereference complex data structures involving references. It's effortless and it doesn't require a *loop* statement. For instance, to extract the telephone number of the client with the account number of AC1234 from the nested hashes, one would enter `$clients{'AC1234'} -> {'telephone'}`. Clean as can be!

Of course, it can be difficult at first to plod through complex data structures with references. It doesn't take long before confusion sets in if you're not meticulous. When this happens, it's hard to know where the trouble lies. In Chapter Five, Schwartz shows how to use a few debugging tools to display the values contained in complex data structures in an organized manner at key points in a program. He covers the debugger, the *Data::Dumper* module, and the *Storable* module for this purpose.

In Chapter Six, Schwartz suggests referencing subroutines. This was a bizarre notion to me when I first read it. However, it can be quite slick. For instance, suppose you have a program in which you want to run one of 10 different subroutines, depending upon which of the 10 possible values the user selects. Normally, I would string together a long *if/elsif* statement. Instead, you could just put a reference to each subroutine in a hash and add something like this to your code to call the appropriate subroutine and to pass any parameters to the subroutine selected:

```
$sub_routines{$selection} -> ($parameter)
```

What could be smoother than that? The last chapter on references, Chapter Seven, elaborates on how to sort data with references and with recursively defined data.

## Objects Section

Most object-oriented programming tutorials tend to bog themselves down with conceptual differences between OOP and

nonOOP. Since this isn't a beginner's book, and since Schwartz seems to prefer a more economical style of writing, he just tells the reader how OOP is done in a clutter-free, straightforward manner. This is not to say that you need to understand OOP concepts before you can benefit from this section. Chapter Eight does walk the reader through the essential elements of OOP with Perl. It's just that some prior exposure to OOP, even if only through *CGI.pm*, will make it clearer. Chapter Nine goes into greater detail on object classes and methods. This includes understanding objects and not treating them in more linear ways, while remembering object inheritance. Schwartz also recommends bringing references into objects to make for still more powerful and efficient code.

Chapters Ten and Eleven cover some housekeeping aspects of objects. Chapter Ten explains to the reader how to deal with objects that become inactive in the course of programs, but are still referenced. These are known as "destroyed objects." Chapter Ten also covers "metavariables" (that is, objects in a hash). Even the minor chapters contain goodies for leveraging your code.

Chapter Eleven deals with more advanced object-oriented programming. For when objects fail, a default method can be established, a *UNIVERSAL* method. To catch failures before they can happen, there is the *isa* and the *can* methods. Schwartz discusses these and the *AUTOLOAD* method, which is used to reduce the drain on resources by little-used objects.

## Modules Section

When most of us think of modules, we tend to think of either the modules that come with Perl (that is, *DBI.pm* and *CGI.pm*), or the thousands of modules found on the CPAN. Most of us don't consider creating our own modules for private use. Chapter Twelve explains how to build modules and how to use them effectively. It expounds on object-oriented modules in particular; it details their intricacies and how they should be called upon. Schwartz advises on building well-constructed modules for others to be able to utilize easily and intuitively.

If you create a module that you think would be of use to others outside of your organization, Chapter Thirteen will show you how to prepare it for public distribution. This includes creating the *Makefile.PL*, the *README* file, the *MANIFEST* file, and the documentation. This also includes setting up the module for users to be able to run *make test* and *make install*. In Chapter Fourteen, Schwartz goes over how to test a module before making it public. He suggests using a few other modules: *Test::Harness*, *Test::Simple*, and *Test::More*. Chapter Fifteen provides information on the final step in making a module public: joining CPAN and uploading a new module.

## Conclusion

Although a slender book of only about 200 pages, the *Alpaca* book is probably the best Perl book to come along in a while. If you're an intermediate Perl programmer, after working through this book thoroughly, you should break through the barrier and become an advanced Perl programmer. It really is an outstanding book that will improve the quality of your code by leaps after reading each chapter. Every serious Perl fan should buy a copy.

TPJ

## Renew Now & Save!

### Plus A Special Offer for Current *TPJ* Subscribers!

The time for subscription renewal is NOW  
—and we have a special offer for **all** current subscribers!

- You can lock in next year's subscription (and the year beyond that, too!) at the low rate of \$16/year by renewing **now**!
- Have access to the complete *TPJ* archives—Spring 1996 to the present! (Effective September 1, 2003.)
- And by renewing between now and November 1, 2003, you also get one year of access to **BYTE.com** at **no extra charge**!

That's a savings of nearly \$20—and you get *BYTE* columns by Moshe Bar, Jerry Pournelle, Martin Heller, David Em, Andy Patrizio, and Lincoln Spector, plus features on a wide range of technology topics.

Currently, all new subscriptions to *The Perl Journal* are \$18/year. But to show our appreciation for your support in our first year of publication, we're making this special limited-time offer available to current *TPJ* subscribers who renew between now and November 1.

**Don't miss out on a single issue or  
this special offer! Renew now!**

**One low price! One great deal!**

**<http://www.tpj.com/renewal/>**

*The Perl Journal*

---

# Source Code Appendix

---

## Julius C. Duque “Implementing the Khazad Block Cipher in Perl”

### Listing 1

```
1  int main(void)
2  {
3      struct NESSIEstruct subkeys;
4      u8 key[KEYSIZEB];
5      u8 plain[BLOCKSIZEB];
6      u8 cipher[BLOCKSIZEB];
7      u8 decrypted[BLOCKSIZEB];
8
9      int v;
10
11     printf("Test vectors -- set 1\n");
12     printf("=====\n");
13
14     NESSIEkeysetup(key, &subkeys);
15     NESSIEencrypt(&subkeys, plain, cipher);
16     NESSIEdecrypt(&subkeys, cipher, decrypted);
17
18     /***** more lines follow *****/
```

### Listing 2

```
1  #include "EXTERN.h"
2  #include "perl.h"
3  #include "XSUB.h"
4
5  #include "ppport.h"
6
7  #include "_khazad.c"
8
9  typedef struct khazad {
10     NESSIEstruct key;
11     }* Crypt__Khazad;
12
13     MODULE = Crypt::Khazad    PACKAGE = Crypt::Khazad
14
15     int
16     keysize(...)
17     CODE:
18     RETVAL = 16;
19     OUTPUT:
20     RETVAL
21
22     int
23     blocksize(...)
24     CODE:
25     RETVAL = 8;
26     OUTPUT:
27     RETVAL
```

### Listing 3

```
1  #include "EXTERN.h"
2  #include "perl.h"
3  #include "XSUB.h"
4
5  #include "ppport.h"
6
7  #include "_khazad.c"
8
9  typedef struct khazad {
10     NESSIEstruct key;
11     }* Crypt__Khazad;
12
13     MODULE = Crypt::Khazad    PACKAGE = Crypt::Khazad
14
15     int
16     keysize(...)
17     CODE:
18     RETVAL = 16;
19     OUTPUT:
20     RETVAL
21
22     int
23     blocksize(...)
24     CODE:
25     RETVAL = 8;
26     OUTPUT:
27     RETVAL
28
```

```

29 Crypt::Khazad
30 new(class, rawkey)
31     SV* class
32     SV* rawkey
33     CODE:
34     {
35         STRLEN keyLength;
36         if (! SvPOK(rawkey))
37             croak("Error: Key must be a string scalar!");
38
39         keyLength = SvCUR(rawkey);
40         if (keyLength != 16)
41             croak("Error: Key must be 16 bytes long!");
42
43         Newz(0, RETVAL, 1, struct khazad);
44         NESSIEkeysetup(SvPV_nolen(rawkey), &RETVAL->key);
45     }
46
47     OUTPUT:
48     RETVAL
49
50 SV*
51 encrypt(self, input)
52     Crypt::Khazad self
53     SV* input
54     CODE:
55     {
56         STRLEN blockSize;
57         unsigned char* intext = SvPV(input, blockSize);
58         if (blockSize != 8) {
59             croak("Error: Block size must be 8 bytes long!");
60         } else {
61             RETVAL = newSVpv("", blockSize);
62             NESSIEencrypt(&self->key, intext, SvPV_nolen(RETVAL));
63         }
64     }
65
66     OUTPUT:
67     RETVAL
68
69 SV*
70 decrypt(self, input)
71     Crypt::Khazad self
72     SV* input
73     CODE:
74     {
75         STRLEN blockSize;
76         unsigned char* intext = SvPV(input, blockSize);
77         if (blockSize != 8) {
78             croak("Error: Block size must be 8 bytes long!");
79         } else {
80             RETVAL = newSVpv("", blockSize);
81             NESSIEdecrypt(&self->key, intext, SvPV_nolen(RETVAL));
82         }
83     }
84
85     OUTPUT:
86     RETVAL
87
88 void
89 DESTROY(self)
90     Crypt::Khazad self
91     CODE:
92     Safefree(self);

```

## Listing 4

```

1 package Crypt::Khazad;
2
3 use strict;
4 use warnings;
5 require Exporter;
6
7 our @EXPORT_OK = qw(keysize blocksize new encrypt decrypt);
8 our $VERSION = '1.0.0';
9 our @ISA = qw(Exporter);
10
11 require XSLoader;
12 XSLoader::load('Crypt::Khazad', $VERSION);
13
14 # Preloaded methods go here.
15
16 1;
17
18 __END__
19

```

```

20 -headl NAME
21
22 Crypt::Khazad - Crypt::CBC-compliant block cipher
23
24 -headl ABSTRACT
25
26 Put abstract here.
27
28 -headl SYNOPSIS
29
30     use Crypt::Khazad;
31     $cipher = new Crypt::Khazad $key;
32     $ciphertext = $cipher->encrypt($plaintext);
33     $plaintext = $cipher->decrypt($ciphertext);
34
35 -headl DESCRIPTION
36
37 Put description here.
38
39 -headl COPYRIGHT AND LICENSE
40
41 Put copyright notice here.
42
43 =cut

```

## Listing 5

```

1 use diagnostics;
2 use strict;
3 use warnings;
4 use Test::More tests => 2;
5 BEGIN {
6     use_ok('Crypt::Khazad')
7 };
8
9 BEGIN {
10     my $key = pack "H32", "80000000000000000000000000000000";
11     my $cipher = new Crypt::Khazad $key;
12     my $plaintext = pack "H16", "0000000000000000";
13     my $ciphertext = $cipher->encrypt($plaintext);
14     my $answer = unpack "H*", $ciphertext;
15     is("49a4ce32ac190e3f", $answer);
16 };

```

## Listing 6

```

1 #!/usr/local/bin/perl
2
3 use diagnostics;
4 use strict;
5 use warnings;
6 use Crypt::CBC;    # CBC automatically loads Khazad for us
7
8 my $key = pack "H32", "00112233445566778899aabbccddeeff";
9 my $IV = pack "H16", "0102030405060708";
10
11 my $cipher = Crypt::CBC->new({ 'key' => $key,
12     'cipher' => 'Khazad',
13     'iv' => $IV,
14     'regenerate_key' => 1,
15     'padding' => 'standard',
16     'prepend_iv' => 0
17 });
18
19 my $plaintext1 = pack "H32", "0123456789abdefedcba9876543210";
20 print "plaintext1 : ", unpack("H*", $plaintext1), "\n";
21
22 my $ciphertext = $cipher->encrypt($plaintext1);
23 print "ciphertext : ", unpack("H*", $ciphertext), "\n";
24
25 my $plaintext2 = $cipher->decrypt($ciphertext);
26 print "plaintext2 : ", unpack("H*", $plaintext2), "\n";

```

## Moshe Bar “Therapy Bots”

### Listing 1

```

while ($line = <$sock>) {
    ($command, $text) = split(/ :/, $line); # $text is the stuff from the
                                           # ping or the text from the server
    if ($command eq 'PING'){
        #while there is a line break - many different ways to do this
        while ( (index($text, "\r") >= 0) || (index($text, "\n") >= 0) ){
            chop($text);
        }
        print $sock "PONG $text\n";
        next;
    }
}

```



```

}
#done with ping handling

($nick,$type,$channel) = split(/ /, $line); #split by spaces

($nick,$hostname) = split(/!/, $nick); #split by ! to get nick and
                                     #hostname sepearate
$nick =~ s/://; #remove ':'s
$text =~ s/://;

#get rid of all line breaks. Again, many different ways of doing this.
$/ = "\r\n";
while($text =~ m#$/#){ chomp($text); }

#end of parsing, now for actions
}

```

## Listing 2

```

Server.pl
#!/usr/bin/perl -w
use Chatbot::Eliza;
$SIG{INT} = sub { print "\nreceived signal!!! \n\n\n";
print "Do you wish to interrupt this program? [Y/n]: ";
$sans = <STDIN>;
if ($sans eq 'n' or $sans eq 'N') {exit 0};

};
$| = 1;

my $bot = Chatbot::Eliza->new;
$bot->command_interface();

```

## Listing 3

```

#!/usr/bin/perl -w

use strict;
use Chatbot::Eliza;
use IO::Socket::INET;

print ">> Therapy Server Program <<\n";
my $response;
my $eliza;
# Create a new socket
my $lsocket = IO::Socket::INET->new(
    LocalPort => 1111,
    Proto => 'tcp',
    Listen => 1,
    Reuse => 1
) or die "Cannot open socket $!\n";

my $socket = $lsocket->accept;
$eliza = new Chatbot::Eliza;
my $def_msg="\nReceiving message from client....\n";
print "Eliza is ready. What's troubling you? ";
my $text = '';
while( 1 )
{
    $socket->recv( $text , 256 );
    if( $text ne '' )
    {
        print "\npatient: $text \n" ;
        sleep 4;
        my $resp = $eliza->transform($text);
        $socket->send( $resp);
        print "\ndoctor: $resp \n";
        sleep 2;
    }
    # If client message is empty exit
    else
    {
        print "Client has exited!\n";

        exit 1;
    }
}
}

```

## Listing 4

```

#!/usr/bin/perl -w
use IO::Socket;
use Chatbot::Eliza;
$remote = IO::Socket::INET->new(
    Proto => "tcp",
    PeerAddr => "localhost",
    PeerPort => "1111",
)
or die "cannot connect to tharapy port at localhost";

```

---

```
my $msg = "Are you my therapist?";
my $ans = " ";
my $eliza = Chatbot::Eliza->new;
while (1) {

    $remote->send($msg);
    $remote->recv($ans, 256);
    $msg = $eliza->transform($ans);
}
```

***TPJ***