

The Perl Review

Volume 0 Issue 4

July 1, 2002

Letters to <i>The Perl Review</i>	i
About this issue	i
Community News	ii
Short Notes	iii
Perl Golf	1
<i>Dave Hoover</i>	
Parrot bit 2: BASIC Parrot	5
<i>Clinton A. Pierce</i>	
The Facade Design Pattern	14
<i>brian d foy</i>	
Book Reviews	26

Web Access <http://www.ThePerlReview.com/> **Email** letters@theperlreview.com **Publisher** brian d foy
Editor Andy Lester **Technical Editors** Kurt Starsinic, Adam Turoff **Copy Editors** **Contributors**
brian d foy, Andy Lester, Clinton A. Pierce, Alberto Manuel Simões,

Letters

Send your letters, comments, and suggestions to letters@theperlreview.com

New TPR schedule

What happened to the June issue?
– name withheld

brian writes: *The Forces of Nature conspired against us last month. We've decided that the once-a-month schedule is a bit too much to ask from a bunch of volunteers. For the immediate future we will publish every other month which gives us more time to develop the articles and still work at our day jobs.*

About this issue

We are trying a couple of new things this issue since we are still finding our way around the magazine publishing business. So far we have concentrate on in-depth articles, but those take awhile to fully polish, so we have added a “Short Notes” section for much shorter articles. Dave Hoover has expanded the Perl Golf column to a longer article with more detail, and Clinton Pierce tells us about writing languages targeted for Parrot (even though the way he implemented BASIC had to change as Parrot did.

Perl on the web

ActiveState Active Choice Awards 2002
<http://www.activestate.com/Corporate/Awards/> – Vote for the nominees for awards in Perl

The Perl Review
<http://www.theperlreview.com> – the website for this magazine with information for readers and authors

Perl Monks
<http://www.perlmonks.org/> – Perl discussion forum

Perldoc.com
<http://www.perldoc.com> – Online, searchable Perl documentation

Use.perl
<http://use.perl.org> – Perl news and commentary

Write for TPR

Have something to say about Perl? *The Perl Review* wants first person accounts about using Perl. If you do not have much to say, you can write a “Short Note”, or if you do, you can write a full article. Want to tell everyone about a book you have read? Write a book review.

We would like to get articles or “Short Notes” on

- Perl & Ruby
- Bioinformatics
- Perl internals
- Cute Perl hacks
- Debugging Perl
- Creating modules
- *and more ...*

We also like articles aimed at Perl or programming beginners. Perl people take for granted some things that never make it into books or passed on to newbies. Do you have something new Perl programmers should know? Perhaps:

- Using templates
- Using configuration files
- Argument processing
- Deciphering documentation

You can get submission guidelines from our website, <http://www.theperlreview.com>.

Volunteer for TPR

We have not turned into a business yet, so we still rely on the generosity and availability of volunteers. If you have something to add to *The Perl Review*, send us a note!

Community News

Send us your news stories at news@theperlreview.com.

YAPC::NA

<http://www.yetanother.com>

This year YAPC::NA was at Washington University in St. Louis, June 22-26. This is the third year that Yet Another Society has hosted the low-cost conference. Rumors say that the next YAPC might be a ski vacation.

Perl 5.8.0 released

<http://use.perl.org/articles/02/04/27/1830250.shtml>
Jarkko Hientaniemi release perl5.8.0 RC1 (Release Candidate), and then RC2. The stable version is still perl5.6.1, so most people may want to wait for perl5.8.1. You can download the latest version of perl from CPAN.

MacPerl 5.8.0a2 released

<http://dev.perl.org>

Chris Nandor released a Mac Classic version of Perl 5.8.0. If you use MacOS X, this is not what you want.

MacPerl 5.8.0a2 released

<http://dev.perl.org>

Chris Nandor released a Mac Classic version of Perl 5.8.0. If you use MacOS X, this is not what you want.

Perl Conference 6 news

<http://conferences.oreilly.com/os2002>

Online registration is open for the O'Reilly Open Source Convention, July 22-26. You can still register—now with a 25% discount.

MacOS X Conference

<http://conferences.oreillynet.com/macosex2002/>

O'Reilly & Associates will host the first MacOS X conference September 30-October 3 in Santa Clara, CA. There will be Perl content.

New books

Publishers: to have us list your book, send us a note at book_reviews@theperlreview.com.

XML in a Nutshell, 2nd Edition

W. Scott Means, Elliotte Rusty Harold; O'Reilly & Associates; 0-596-00292-0; 600 pages; June 2002

Web Development with Apache and Perl

Theo Peterson; Manning; 1-930110-06-5; June 2002
Reviewed in this month's Book Reviews.

Graphics Programming with Perl

Martien Verbruggen; Manning; 1-930110-02-2; June 2002

Perl & LWP

Sean Burke; O'Reilly & Associates; 0-596-00178-9; 400 pages; July 2002

Mastering Regular Expressions, 2nd Edition

Jeffrey E. F. Friedl; O'Reilly & Associates; 0-596-00289-0; July 2002

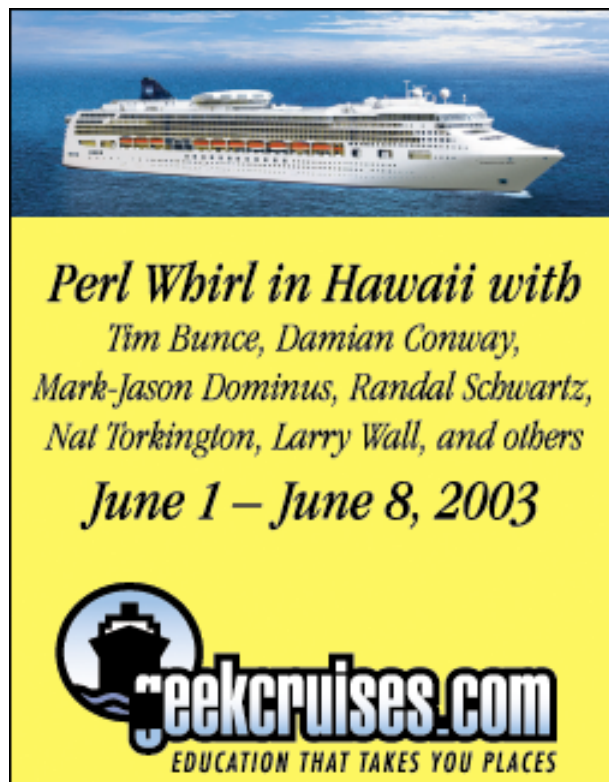
Essential Blogging

Benjamin Trott, et al.; O'Reilly & Associates; 0-596-00388-9; August 2002


Extending and Embedding Perl

Tim Jenness and Simon Cozens; Manning; 1-930110-82-0; July 2002

Would you like to review a book? Send your review to book_reviews@theperlreview.com



Perl Whirl in Hawaii with
Tim Bunce, Damian Conway,
Mark-Jason Dominus, Randal Schwartz,
Nat Torkington, Larry Wall, and others
June 1 – June 8, 2003



geekcruises.com
EDUCATION THAT TAKES YOU PLACES

Short Notes

In the spirit of the 5 minute Lightning talks run by Mark-Jason Dominus at various conferences, *The Perl Review* starts publishing “Short Notes”. If you have something that you want to show off without writing an entire article, like a cool Perl trick, a module you just released or something happening in the Perl community, send your short note, between 200 and 400 words, to short_notes@theperlreview.com.

Parser guts

Alberto Manuel Simões
albie@alfarrabio.di.uminho.pt

[*Publisher’s note: We fixed an inconsistency in Alberto’s article. His code gave one answer, but the text said another. We fixed it but forgot to tell Alberto. He adds this follow-up as a correction. We apologize for the error.*]

Some of you may have wondered why in my last article (“Cooking Perl with flex”, v0i3) the answer for $1+4*2+3$ was 25, when you probably expected 12. It is all a matter of which order things happen. The calculation $(1+4) * (2+3)$ is different from $1 + (4*2) + 3$. I can correct the order to do what most people expected.

At the top of the grammar in last example I had three lines:

```
%token NUMBER NL

%left '*' '/'
%left '-' '+'
```

The first line tells the parser generator I will have tokens named NUMBER and NL. The other two lines tell the same, but with some more information. It says I have all left-associative operators. Then, I have put each one in a different line. This means that they will have different precedences. And is this where we get the difference between the result I presented and one I expected. Because the line with the multiplication and division is first, I erroneously gave higher precedence to subtraction and addition. For the expected result, I should have swapped those two lines

so the lower precedence operator came first.

```
%token NUMBER NL

%left '-' '+'
%left '*' '/'
```

I can add a higher precedence

```
%left '-' '+'
%left '*' '/'
%right '^'
```

The exponentiation $^$ has more precedence than all other operators. To make this work, I also have to add a new line to the exp production.

```
exp: exp '+' exp { return $_[1] + $_[3] }
    | exp '-' exp { return $_[1] - $_[3] }
    | exp '*' exp { return $_[1] * $_[3] }
    | exp '/' exp { return $_[1] / $_[3] }
    | exp '^' exp { return $_[1] ** $_[3] }
    | number      { return $_[1] }
    ;
```

Host with pair Networks

- Rock-Solid FreeBSD Unix Reliability
- Edit scripts and compile programs from a secure shell
- Technical, competent support
- Perl 5.6 with over 2000 installed modules
- Free Setup when you switch to us

We've got a web hosting plan waiting for you.

www.pair.com



Perl Golf

Dave Hoover, dave@redsquirreldesign.com

Abstract

Prior Perl Golf columns focused exclusively on winning solutions. A number of golfers requested coverage of a more down-to-earth solution. This month's deconstruction covers a solution near the middle of the field. May's challenge featured the Kolakoski sequence, a series of self-describing numbers.

1 The Challenge

The Kolakoski sequence is an infinite series of alternating self-describing numbers.

```
% perl kola.pl 2 3 20
22332223332233223332
```

The string's first digit (2) determines the length of the first block (22). The second digit (2) determines the length of the second block (33). The third digit determines the length of the third block. Blocks alternate between the two different digits. This continues indefinitely. Solutions need only handle at least 500 characters in the sequence, however

Programs take three numbers from the command line—two digits which determine the construction of the sequence (e.g. 2 & 3) and the length of the output string (e.g. 20).

2 The Deconstruction

Philippe 'Book' Bruhat finished with a 75 stroke Kolokoski solution which put him slightly ahead of the median 79 stroke solution. He submitted 25 working versions over 6 days, which gives us an excellent history on the development of his final solution:

```
#!/perl -l
$@.=($.=$ARGV[$_%2])x(substr$@,$_,1 or$.)for 0..500;
print substr$@,0,pop
```

When golfers test several algorithms, they better understand the options available. Novice golfers frequently trap themselves in their own cleverness hindering them when they quickly discover a working solution and become emotionally attached to the algorithm. A distinguishing factor between good and great golfers is the ability to throw away an algorithm after golfing it to its limit.

We reformatted the final solution to make it easier to read. The special variables `$@` (eval error) and `$_` (input line number) become `$string` and `$first`, respectively. Next, we reformat the `foreach` loop by breaking it down into four explicit statements with comments, and add clarifying whitespace.

```
#!/perl -l
$first = $ARGV[0];           # Copy the first argument.
foreach $i (0..500) {       # 501 iterations.
    $digit = $ARGV[$i % 2];  # Alternate digits.
    $length = (substr $string, $i, 1 # Get block length from string.
               or $first);      # If no string, use first arg.
    $string .= $digit x $length;    # Build the output string.
}
print substr $string, 0, pop;    # Output the correct length.
```

With this reformatting, the underlying algorithm becomes more recognizable. As Perl Golf progresses, veteran golfers recognize the importance of compressible algorithms and place as much value on them as they do on other techniques.

The shebang line contains the `-l` switch which causes all `print` statements to automatically append a newline. XXX: reference to earlier column.

The loop has to determine two things to output the next part of the sequence—the digit to use and the length. Philippe determines the length of the next block by reading from the specified location in the string he simultaneously builds. On the first iteration, the string is not yet defined, so `substr` returns false. The short circuit or operator returns the value of `$first` which he assigns to `$length`. On subsequent iterations, the second digit in `$string` is the length of the second block, the third digit the length of the third block, and so on until the loop finishes.

`$i % 2` is a simple way to test whether a number is odd or even. It returns 1 if `$i` is odd and 0 if even. With each iteration, `$i` will switch from odd to even, so Philippe uses it to alternate between `$ARGV[0]` and `$ARGV[1]`. This value is the next digit to use.

Once Philippe has the next digit and the length, he can add to the string. The repetition operator, `x`, repeats its left operand (the next digit) by the number its right operand (the length) specifies. He adds this to `$string`, and loops again.

Philippe admits that he is emotionally attached to the `substr` function. In this solution, both instances of `substr` take three arguments—the first argument is the string to operate on, the second is the starting point within the string, and the third is the length of the substring to extract. He uses `substr` to output the string with the correct length (from `$ARGV[2]`), which `pop` returns. A `pop` without arguments in the main body of the program removes the last element from `@ARGV` (though in subroutines it operates on `@_`) and returns it—in this case using it as the third argument to `substr`. With the same input as the first example, Philippe constructs a string 1250 characters long, although he only needs the first 20. The `substr` function extracts the first 20 characters and he never uses the rest of the string, affirming that brevity, not efficiency, was his primary concern.

With the mechanics explained, we begin to return the solution to its original form by removing unnecessary white space, shortening variable names, and replacing `foreach` with its shorter synonym, `for`. We shorten the variable names `$first`, `$digit`, `$length`, and `$string` to only their first letters, `$f`, `$d`, `$l`, and `$s`.

```
#!/perl -l
$f=$ARGV[0];
for$i(0..500){
$d=$ARGV[$i%2];
$l=(substr$s,$i,1 or$f);
$s.=$d x$l;
}
print substr$s,0,pop;
```

Next, we replace the `$d` and `$l` of the string-building statement `$a.=$d x$l` with the expressions that define them. This consolidation reduces the for loop's block to one line. We discard `$i` since we can use the default index variable `$_`.

```
#!/perl -l
$f=$ARGV[0];
for(0..500){
$s.=$ARGV[$_%2]x(substr$s,$_ ,1 or$f);
}
print substr$s,0,pop;
```

Since the for loop has a single statement, we can rewrite the loop as a statement modifier.

```
#!/perl -l
$f=$ARGV[0];
$s.=$ARGV[$_%2]x(substr$s,$_ ,1 or$f)for 0..500;
print substr$s,0,pop
```

Explicitly using `@ARGV` two times should tip off a golfer that he can still make improvements. On the first pass, `$ARGV[$_%2]` is equivalent to `$ARGV[0]`, meaning we can use it to define `$f`. The parentheses around `$f=$ARGV[$_%2]` force `x` to evaluate the result of that statement. The result of an assignment is the value assigned,

```
#!/perl -l
$s.=(f=$ARGV[$_%2])x(substr$s,$_ ,1 orf)for 0..500;
print substr$s,0,pop
```

Finally, we replace the reader-friendly `$s` and `$f` with Philippe's original `$@` and `$.`.

```
#!/perl -l
$@.=(.=ARGV[$_%2])x(substr$@,$_ ,1 or$.)for 0..500;
print substr$@,0,pop
```

3 References

Kolakoski sequence – <http://mathworld.wolfram.com/KolakoskiSequence.html>

Perl Golf web site – <http://perlgolf.sourceforge.net/>

Kolakoski Perl Golf – <http://perlgolf.sourceforge.net/TPR/0/3/>

Parrot bit 2: BASIC Parrot

Clinton A. Pierce, clintp@geeksalad.org

Abstract

The Parrot assembly programming environment provides a fun, simple, and back-to-basics experience for the assembly-language programmer, and until recently did not have a complete implementation of any established high-level programming language. Being somewhat bored and looking for a challenge, I created Parrot BASIC—a full-featured interpreted BASIC capable of running an established and time-proven software base.

1 Parrot BASIC

I began work on Parrot BASIC after reading an article on Perl.com about programming in Parrot Assembler (PASM). Assembler’s always been a fun thing for me and I was immediately overcome by waves of nostalgia for programming my Atari 6502 or my first real job involving 8088 assembler, and so I dove in with the sample programs.

First, I attempted was a XML parser in PASM, which was fun while it lasted but over with far too quickly. So I set my sights on something larger—a BASIC interpreter in PASM.

BASIC is a simple language to parse, and the parser would be in PASM. BASIC is also a simple language since it has no namespaces, scopes, or complex data types. I can implement my interpreter in stages and run programs almost immediately which means that BASIC’s large library of programs (games! diversions galore!), will be ready to run.

I also decided that to parse BASIC in something like Perl, to compile to PASM would be cheating. I wanted this to be a fully-interpreted BASIC with an interactive command prompt (“Ready”) all self-contained in one Parrot bytecode file. I based the design of the interpreter roughly on designs for 8k BASIC I saw back in the early 1980s.

2 Working with Parrot

Working with Parrot was an interesting experience. As I would bumble along coding this bit or that I would stumble into parts of the project that did not quite work yet. I would post a message to the internals developer’s list, and a few hours later someone would post a bug fix. Sometimes my problems were my own fault, bugs in Parrot, or I had stumble into pieces of the project that were not quite ready for prime time yet.

Looking back at my mail to the Perl 6 Internals lists (<http://lists.perl.org/showlist.cgi?name=perl6-internals>) and IRC log files ([#parrot](http://rhizomatic.net)), the response I got from the Parrot developers was great. Eventually things got to the point where we could not describe a small and concise bug report so the developers would just fire up BASIC and play a game of wumpus until it crashed to find the bug.

It was not all joy and sunshine, though. Working on a development platform where the developers change coding standards, add opcodes, or subtly change the way existing opcodes work provided more than a few headaches. Overall though, it was a terribly rewarding experience.

3 Following Along

If you want to follow along with the discussion, you can download the Parrot distribution which includes my BASIC interpreter. To get Parrot, go to <http://dev.perl.org> and follow the instructions. Dan Sugalski gave a gentle introduction to the build process in *The Perl Review* 0.3.

After you have built Parrot, change in to the `languages/BASIC` subdirectory and type at the prompt:

```
./basic.pl
```

This is simply a harness which calls the PASM assembler with the correct options and invokes the Parrot interpreter. BASIC will take a few seconds to build and you will soon be treated to a “Ready” prompt. If you quit out of the interpreter (`^C` or `QUIT`) and want to restart it, simply type

```
../../parrot out.pbc
```

and it should start instantly without all of the re-assembly.

4 Guts of the Interpreter

The interpreter code has 5 major subsections.

- The harness, which compiles the PASM and provides the interactive prompt (`basic.pl`)
- The statement dispatcher and BASIC statement code (`basic.pasm`, `instructions.pasm`)
- The BASIC variable storage routines (`basicvar.pasm`)
- The expression evaluator (`expr.pasm`)
- Support libraries (`stackops.pasm`, `alpha.pasm`, `dumpstack.pasm`, `tokenize.pasm`)

5 Coding conventions

I began work on Parrot BASIC while the developers were still debating PASM style, and so the code presented here does not conform to all of the ‘standards’ they later chose. I used callee-save (as opposed to caller-save) and pass arguments to subroutines on the user stack instead of through registers.

I use the stack for argument passing because my assembler programming history is on stack machines (6502, 8088) and I am used to seeing the world through a series of stacks. The one register-based architecture I

used left such a bad impression on me that I may never fully recover, although this was the fault of the OS, not the register concept.

Since I use the stack to pass arguments, I can write self-contained subroutines since I do not expect registers to have anything useful in them. I manage the user stack with these guidelines:

1. Most support library calls pull a fixed number of arguments from the stack and push a fixed number back on. The caller and callee must both be careful to save and restore the same number of arguments in the same order.
2. In some cases, a variable number of elements will be passed between the caller and the callee. In this case, the top item on the stack is an integer which indicates how many items on the stack are being passed. The callee is always expected to return the stack back to this state (with the depth on top).
3. Some subroutines (expression evaluator, tokenizer, peek) take a fixed number of arguments, then deal with the stack using those arguments. In this case, the callee is expected to put the stack back together (depth on top) and then return its arguments on top of that.

At runtime, PASM uses the user stack to parse BASIC, evaluate expressions, and so on. At the same time, the use stack is also the BASIC runtime stack. I keep the runtime stack at the bottom and the working stack on top of that. For example, after processing:

```
10 FOR I=1 TO 50
20 GOSUB 100
30 NEXT I
100 LET A=90*I
110 RETURN
```

At line 100 the entire user stack would look like:

```
6          # Current stack depth
LET
A
=
90
*
I
10         # Runtime stack depth
GOS
20         # Line number of GOS
0          # Unused
0
0
0
FOR
10         # Line number of FOR
0          # Unused
0
50         # Termination expression
1          # Step
```

When the interpreter is done with line 100, the top item on the stack should be the depth of the runtime stack; each BASIC statement handler is expected to clean up after itself.

Since the callee was saving everything it took a bit of cleverness to call a subroutine that could modify a “global” variable used in the interpreter. Most subroutines are coded something like:

```
CHANGE_PC:  pushi
            # lots of code here, changing things, etc...
            set I23, I0  # Change the PC.
            popi
            ret          # Whoops, PC didn't get changed.
```

Which would have no effect as I23 would be reset on with the popi. Since within a callee-save subroutine I scope everything as though it were a Perl local(), I labeled this as an *unlocal*:

```
CHANGE_PC:  pushi
            # lots of code here, changing things, etc...
            set I23, I0
            save I23     # Preserve it from the popi
            popi
            restore I23  # Put it back.
            ret
```

And now I2—as it was set in the subroutine—would be preserved after the subroutine call. Sometimes this unlocal madness can be quite severe.

6 Support Libraries

A surprising amount of code in any assembler programming is not the actual opcodes themselves, but setting up calls to libraries and dealing with the results. For example, to divide a statement into tokens on the stack for processing I need functions to test for both alphanumeric sequences and whitespace.

The ISALPHA function tests for alpha-numeric characters and is typical of the types of support functions written for BASIC.

```
ISALPHA:
    pushi
    pushs
    restore S1
    ge S1, "A", UPPER    # Yes, this is ASCII.
    branch NONUP         # If you want ISO 8859 or
UPPER:  le S1, "Z", ALPHA # Unicode, you didn't want BASIC.
NONUP:
    ge S1, "a", LOWER
    branch NONLOW
LOWER:  le S1, "z", ALPHA
```

```
NONLOW:
    ge S1, "0", NUMBER
    branch NONUM
NUMBER: le S1, "9", ALPHA
NONUM:  eq S1, "_", ALPHA
        # Not A-Z0-9_
        set I1, 0
        branch LEAVE_ISALPHA
ALPHA:  set I1, 1
LEAVE_ISALPHA:
    save I1
    popi
    pops
    ret
```

The subroutine begins by pushing Parrot's Ix and Sx registers onto their appropriate stacks. This allows the subroutine to use all of the Ix and Sx registers with impunity, as long as it restores them before it returns.

The `restore S1` statement removes a string from the stack that is the character to be tested. After a series of greater-than/less-than tests, the flag variable I1 is either set to 1 (alphabetic) or 0 (non-alphabetic). I push this onto the stack, pop the Ix and Sx registers, and return from the subroutine.

To use the subroutine:

```
save S10
bsr ISALPHA
restore I0
```

Because the subroutine saves the state of the caller first, I do not need any special preparation other than to put the arguments on the stack and pull the results off. The other major library routines are:

- `TOKENIZER` – tokenize a string
- `ISALPHA`, `IS WHITE` – test a character's type
- `STRNCHR` – search for characters within strings
- `ATOI`, `ITOA`, `ISNUM` – convert strings to integers and back[1], test for numeric-ness
- `PAD`, `STRIPSPACE`, `STRIPLEADSPACE` – insert or remove excess whitespace
- `STRSTR` – search a string for a substring
- `PEEK` – fetch a value from an arbitrary depth on the user stack
- `SWAP`, `REPLACE`, `REVERSESTACK` – edit values on the user stack
- `CLEAR` – clear the user stack
- `SORTSTACK`, `NSORTSTACK` – sort the stack alphabetically, numerically.

6.1 Variable Storage

In Parrot BASIC I require storage for three different kinds of data:

- The code for the BASIC program itself
- Numeric variables
- String variables

Before May, I did this with linked lists stored in conventional string registers; unpacking and searching substrings for values in larger string registers. This was slow, but trustworthy.

In May the Parrot team completed initial work on the PerlHash and PerlArray PMC types in Parrot assembler. I scrapped the old, slow variable storage methods and replaced with PMCs. Three PMC registers (P21, P22 and P23) hold a copy of the numeric variables, string variables, and the program code respectively.

I keyed the variable PMC's on the variable name. I create multi-dimensional arrays with compound key of the variable name and the offset. I store the BASIC program itself in P23 and keyed by line number. A parallel structure of type PerlArray keeps an index of the lines in order.

From the PASM BASIC programmer's standpoint, to create a new (numeric) variable it all boils down to:

```
save 42      # Value to store
save "N2"    # Variable to create
bsr NSTORE   # A thin wrapper for VSTORE, which inserts into
```

To fetch it back later:

```
save "N2"    # BASIC variable to fetch
bsr NFETCH
restore NO    # IO now has N2's value (42)
```

With the PerlHash and PerlArray PMC types, storage in PASM got a whole lot simpler to manage.

7 Expression Evaluation

Another large component of the interpreter is the expression evaluator (`expr.pasm`). Its job is to take an expression on the stack and replace it with a value. Take the statement:

```
IF X < Y$*3/-2 THEN GOTO 50
```

The tokenizer will leave this on the stack as:

```
13      # depth
IF
X
<
Y
$
*
3
/
-
2
THEN
GOTO
50
[runtime stack below this]
```

The IF-handling subroutine will remove the IF and decrement the stack depth. Next, it pushes a stop-word onto the stack and calls EVAL_EXPR. The stop-word tells the expression evaluator when to stop.

```
restore I5      # Depth
restore S0      # "IF"
dec I5
save I5         # Put the depth back
save "THEN"     # Process until "THEN" is seen
bsr EVAL_EXPR  # True or false?
restore S0      # The return value...
```

The stack after EVAL_EXPR will contain everything from the stop-word down, and on top it will have the return value from the expression evaluation.

Expression evaluation happens in three steps:

1. The expression stack is “cooked” a bit: unary minus is glued onto its argument again (they were separated during tokenizing) and \$ is tacked onto the previous item so that string variable names are whole again. Functions (and subscripts) are re-written from RND(3*2) to RND! ~ (3 * 2) so that ~ becomes something that looks like a bind operator.
2. Next, the expression is converted from Infix notation to Postfix notation. The precedence of the operators is fixed here, parenthesis removed, and evaluation order determined. To pull this off, I had to implement a second “stack” in a string register. The expression in the IF statement above in RPN looks like: X Y\$ 3 * -2 / < . This becomes our input queue for the next step.
3. Last, the input queue is emptied one element at a time and evaluated. The nice thing about Postfix is you simply push things onto a stack from an input queue. When you find an operator on the queue, pull two items from the stack, operate on them, put one item back. When the input queue is exhausted you’re done evaluating, whatever is on the stack is your result. The RPN given would be evaluated as follows: push X on the stack, push Y\$ on the stack, push 3 on the stack, to process * pull the top two items, Y\$ and 3, multiply them and put the result back. And continue.

When functions are encountered, they pull all of the necessary items from the expression stack and put their own value back (if any). Variables (non-subscripted) are substituted during step 3 above. If there's a subscript they fall into the function-evaluation routines, it recognizes that they're not a function, and performs a variable lookup.

All the while, strings are quietly interchanged for numbers and vice-versa. This is why we can multiply Y\$ and 3 in the example above.

7.1 Running a Program

A BASIC program runs terribly simple. The interpreter fetches a line of the program, tokenizes it, grabs the command to be run—the second token—and determines if it is a valid command (`basic.pasm`). Next, it jumps to the appropriate subroutine in `instructions.pasm` and then removes the rest of the tokens from the stack and does whatever they say. It cleans up the stack, and fetches the next line. Repeat as necessary.

For example, I implemented the GOTO (or GO TO) statement with this (somewhat over-commented) code:

```
# GOTO EXPR
# GO TO EXPR
I_GOTO: pushi
        pushes
        restore I0      # Line Number (supplied by basic.pasm)
        restore I5      # Depth
        restore S0      # Keyword "GO" or "GOTO"
        dec I5
        eq S0, "GOTO", I_NOTGO_TO
        restore S0      # Hope this is a "TO"
        dec I5
I_NOTGO_TO:
        save I5          # Stack's kosher now.
        save "REM"       # Stop-word
        bsr EVAL_EXPR    # Leaves result string on stack
        bsr ATOI         # Convert that to an integer
        restore I23      # Adjust PROGRAM COUNTER.
        bsr CLEAR        # Clean up stack
        restore I0      # Dummy value (depth of 0)
END_I_GOTO:
        save I23
        popi
        pops
        restore I23
        ret
```

The whole purpose of this routine is to set I23—the BASIC line number counter—to the new line number which is an expression.

8 Conclusion

Parrot is a charming CPU to code for, and porting BASIC to it was a gentle experience. Parrot programmers should be able to port other languages than can have small implementations (but often do not) like FORTH, Lisp, or Pascal.

9 About the Author

Clinton Pierce is the author of *Teach Yourself Perl in 24 Hours* and *The Perl Developer's Dictionary*. Currently he's writing financial software in Perl and C. For more information see <http://geeksalad.org>.

10 References

“Parrot: Some Assembly Required” by Simon Cozens
<http://www.perl.com/pub/a/2001/09/18/parrot.html>

Perl 6 Internals developer's list
<http://lists.perl.org/showlist.cgi?name=perl6-internals>

“Parrot Bits: Bit 1, The Parrot Vooms!”, *The Perl Review* 0.3, Dan Sugalski
http://www.theperlreview.com/Issues/The_Perl_Review_0_3.pdf

The Facade Design Pattern

brian d foy, comdog@panix.com

Abstract

The Facade design pattern provides an easy-to-use interface to an otherwise complicated collection of interfaces or subsystems. It makes things easier by hiding the details of its implementation.

1 Introduction

The Facade design pattern connects the code we write for applications, which do specific tasks, such as creating a report, and the low level implementation that handle the details, such as reading file, interacting with the network, and creating output. The facade is an interface that an application can use to get things done without worrying about the details. The facade decouples these layers so that they don't depend on each other, which makes each easier to develop, easier to use, and promotes code re-use.

I can use this design pattern to deal with a complex system that already exists, or one that I want to make from scratch. Several Perl modules available on the Comprehensive Perl Archive Network (CPAN) represent facades, even if they do not admit it.

2 Illustration of use

To request a simple file from a web site, I have to create a connection to the web site, request the resource using a proper HyperText Transfer Protocol (HTTP) request, receive the HTTP response, parse the response, and finally handle the data. I have to do much more work if I want to handle common web features like cookies, forms, and caching. If I want to fetch a resource from an FTP server instead of a web server, I need to handle a completely different protocol.

If I look at this problem, some immediate objects present themselves: connection, request, response, and resource. However, I only want to fetch the resource and continue on with my real work rather than deal with myriad objects to do something that is logically so simple. To code this myself in a reasonable amount of time might take a couple of screenfuls of code depending on how careful I am and how many features I decide to support.

The LWP module (Library for WWW in Perl) provides a facade for doing all of these things. I tell LWP to fetch a resource and it does the rest, including all of the protocol-specific details for HTTP, FTP, or any other protocol that LWP understands.

In code listing 1, I use `LWP::Simple` which makes fetching a web resource as simple as it can get. I do not have to specify the protocol, the connection method, or parse the response. Indeed, if I know the URL, and I can fetch the resource.

Code Listing 1: A web facade

```
1 use LWP::Simple qw(get);
2
3 my $url = 'http://www.perl.org';
4 my $data = get( $url );
```

The `LWP::Simple` module is a facade—it provides a simple interface that unifies protocol, network, and parsing aspects of the problem so that I do one thing—fetch the resource. If someone changes `LWP` or underlying implementations, I do not have to change my script and I still benefit from the improvement.

3 Focus on the task

A facade restricts the functionality, and thus the complexity, of a system by creating specialized interfaces for specific tasks. The `HTML::Parser` module is a base class, so the programmer must write a subclass that tells the parser what to do and when to do it, but it does not perform a specific task other than the parsing, such as syntax checking or data extraction. However, as in my `LWP` example, I simply want to complete a single, logical task—not program `HTML::Parser` subclasses. In reality, I like writing `HTML::Parser` subclasses, but not everyone with whom I work does, so I can create facades for them.

Facades to `HTML::Parser` do small, common tasks while they hide all of the complexity behind-the-scenes. I am the only one in the programming team who has to understand `HTML::Parser` so I can create much simpler interfaces for the rest of the team. They should not have to write an entire subclass if they only want to extract the links of an HTML document, for instance. Simple things should be simple.

If I wanted to extract references (which most people call “links”) from an HTML document, I can use the `HTML::LinkExtor` module. It handles most of the complexity for me while still giving me a reasonable amount of flexibility through a callback mechanism. In code listing 2, I use a callback to extract all of the anchor references (the `HREF` attribute from the `A` tag). I still have to do a bit of the dirty work since `HTML::LinkExtor` passes the tag name and a list of attribute–value pairs to `call_back()`. I still have to know the details of the implementation of `HTML::LinkExtor` to work with it.

Code Listing 2: `HTML::LinkExtor`

```
1 require HTML::LinkExtor;
2
3 use vars qw( @links );
4
5 sub call_back
6 {
7     my( $tag, %attr ) = @_;
8     return unless exists $attr{href};
9
10    push @links, $attr{href};
11 }
12
13 my $parser = HTML::LinkExtor->new( \%call_back, "http://www.example.com" );
14
15 $parser->parse_file("index.html");
```

I can sacrifice flexibility for convenience by using a simpler Facade, `HTML::SimpleLinkExtor`, which simply returns the references but does not have a callback mechanism. In code listing 3, I do the same thing that I do with `HTML::LinkExtor` example in code listing 2, and my fellow programmers do not know how `HTML::SimpleLinkExtor` does it. They do not need to worry about writing the callback function. They simply get the result that they need.

```
_____ Code Listing 3: HTML::SimpleLinkExtor, just HREFs _____  
1 use HTML::SimpleLinkExtor;  
2  
3 my $extor = HTML::SimpleLinkExtor->new( "http://www.example.com" );  
4 $extor->parse_file("index.html");  
5  
6 my @links = $extor->href;
```

If I want to do something different with `HTML::LinkExtor`, like extracting URLs from tags with SRC attributes, I have to modify the `callback` subroutine, or I can use a method from `HTML::SimpleLinkExtor`, a simpler facade, like I do in code listing 4.

```
_____ Code Listing 4: HTML::SimpleLinkExtor, all links _____  
1 use HTML::SimpleLinkExtor;  
2  
3 my $extor = HTML::SimpleLinkExtor->new( "http://www.example.com" );  
4 $extor->parse_file("index.html");  
5  
6 my @all_links = $extor->links;
```

In both of these examples, the facades allows programmers to focus on the task—extracting links—rather than on the programming. Since each facade provides a task-oriented interface, programmers do not spend time thinking about how the task should be completed, just as they do not think too much about HTTP or TCP/IP when they use their web browser to visit their favorite web pages.

The `HTML::SimpleLinkExtor` works for most uses, but also cannot handle complex cases at all. Reduced flexibility is the major consequences of a facade. The more restrictive facades are easier to use at the cost of flexibility. `HTML::LinkExtor` has more flexibility, but is a bit more complicated and I have to do more work to use it. The more flexible interfaces can handle more situations and respond to special cases at the cost of simplicity. Specific situations require different levels of flexibility and simplicity, and as a result, different facades, if any at all.

4 Facades promote reusability

Many programmers already use a sort of facade, although they typically call it a subroutine. Subroutines did not always exist, and they represented a pattern of their own at one time. Today most languages take subroutines for granted even though they are the foundation of re-usable code.

The abstract nature of the subroutine allows programmers not only to group program statements into logical operations behind a subroutine name, like the facades in the previous section, but they also allow

programmers to reuse that group of program statements without repeatedly typing them. Programmers can reuse and share collections of subroutines that they group in libraries which can form the interface of a facade.

What if I want to check the status of a web resource? If I went through all of the steps myself every time I needed to do this, I would have to create an HTTP request, connect to the web server, receive the response, parse the response, and check the response code against known status codes. Later, once I have coded this four or five lines of LWP code (or 15 to 20 lines of socket code) in all of my applications that need it, I will probably discover that I need to fix some of the code and to make the change in several places. If I put all of that code in a subroutine that all of my applications can share, I only have to fix things in one place and programmers using my module do not need to do anything at all.

I first ran into the link validation problem when I created a user-configurable directory of internet resources which I wanted to validate every day. I wanted to make sure that the links in the directories actually led to a web page rather than the annoying “404 Not Found” server error. I also wanted to catch dead links before a customer added them to his directory. I needed to do the same simple task in several applications, and the few lines of repeated code in each application seemed so innocent that I missed the obvious refactoring potential.

With a couple of customers using the service, I did not notice anything amiss with my validation code. It caught dead links and did not have false positives. With tens of users and a couple hundred thousand resources to validate, I discovered that not all web servers respond in the same way to certain types of HTTP requests, and that some servers even had little known bugs. One notorious server returned an HTTP error for any HEAD request¹, so I had to program some special cases. The task which I thought was simple grew much more complex, but I wanted to work on the level of a “ping”—a simple “yes” or “no” answer. Tests on small data sets did not reveal any problems in the parts per ten thousand range, but things quickly got out of hand after that.

My refactored solution was a facade. At the application level I did not care about server eccentricities, work-arounds for HTTP non-compliance, or most error recovery. I simply wanted to know if the URL actually pointed to something. I created a glorified subroutine, gave it a module name, and used it whenever I needed an HTTP response code. I uploaded the module to the Comprehensive Perl Archive Network (CPAN) as HTTP::SimpleLinkChecker. Code listing 5 shows the entire facade—a single function behind which all of the real work takes place. The facade takes care of all of the details, including all of my accrued knowledge about specific server behaviors, so that it could recognize possible problems and double check errors to a HEAD request by actually downloading the resource.

Code Listing 5: HTTP::SimpleLinkChecker

```
1 use HTTP::SimpleLinkChecker qw(check_link);
2
3 my $code = check_link("http://www.example.com");
```

If someone uses this module and decides to upgrade to a newer version when I release one, he can get the benefit of all of my improvements and enhancements without changing any of his code, while at the same time, he benefits from any enhancements to its LWP infrastructure even if he does not use the latest version of HTTP::SimpleLinkChecker. This *loose coupling* makes a programmer’s life much easier since the facade hides changes to the underlying system. Changes in other parts of the system have little or no maintenance consequences on the programmer’s application. Since the facade depends very loosely on the underlying

¹Every HTTP request specifies a method. A HEAD request asks the server for the resource’s meta data, but not the resource itself so it does not have to download potentially large amounts of data.

implementation, I can distribute it separately. Other people do not need a specific version of LWP. I make it easy for people to use so that they will use it rather than going through the pain and suffering that I did.

5 Facades as objects

A facade object acts as the gatekeeper of all method calls for the underlying implementation. Each method may in-turn act upon additional objects or classes to perform its task, but the programmer does not have to know anything about that. The facade object knows which underlying objects handles the real work and delegates parts of each task appropriately².

If I want to parse an HTML page to extract various things from the <HEAD> as well as some of the links, I can use `HTML::HeadParser` and `HTML::LinkExtor`, but at the application level that is too much detail. I am stuck thinking about HTML parsing when I should be getting my real work done. I can also create my own HTML facade that hides the two modules—or any other implementation—that I use.

In code listing 6, I wrap the interfaces to `HTML::HeadParser` and `HTML::SimpleLinkExtor` in a single interface so I only have to deal with all of them in my application.

Code Listing 6: The `HTML::SimpleExtractor` facade

```

1 package HTML::SimpleExtractor;
2
3 require HTML::HeadParser;
4 require HTML::SimpleLinkExtor;
5
6 sub new
7     {
8     my( $class, $url ) = @_;
9
10    return unless $data = LWP::Simple->get($url);
11
12    bless \$data, $class;
13    }
14
15 sub title
16     {
17     return HTML::HeadParser->new()->parse($$_[0])->header('Title');
18     }
19
20 sub links
21     {
22     HTML::SimpleLinkExtor->new()->parse($$_[0])->links;
23     }
24
25 1;
26
27 __END__

```

²Delegation represents another sort of pattern with which the Facade pattern might collaborate. Many patterns work in concert with other patterns, and although I do not discuss Delegation here, you can read about it in the documentation of Damian Conway's `Class::Delegation` or Kurt Starsinic's `Class::Delegate`

Once I have my facade in place, I can write applications that I do not need to couple to any particular module. Since my program in code listing 7 does not know that I used `HTML::HeadParser` it does not need to change if I decide to use something different for that portion of the task. The facade hides the changes.

Code Listing 7: `HTML::SimpleExtractor`

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  use HTML::SimpleExtractor;
5
6  my $html = HTML::SimpleExtractor->new('http://www.example.com');
7
8  my $title = $html->title;
9  my $links = $html->links;
10
11 $ = "\n\t";
12
13 print "$title\n\t@links\n";
14
15 __END__
```

In this case, I can change any of the details involved with fetching and parsing the HTML to something smarter and more efficient later. This can be quite expedient when the responsibility for the facade and the application belong to different programmers or teams, or when it would take much longer to fully implement the facade than to finish any of the applications. I can create something that works today even if it is not the best implementation then I can incrementally change and improve it as time allows.

Even though my example is very simple-minded, I get the job done. I hide the two modules behind the facade, and I can immediately use `HTML::SimpleExtractor` in my applications. Once I have more time to devote to the module, I can change how it does its job while all of my applications that use it stay the same. None of my application code depends on the specifics of the implementation, and may not even know that the implementation has changed.

6 Ad hoc facades

Most frequently the work programmers seem to do involves an established base code that has entrenched itself into the work flow of their organizations, and the further away they are from the creation time of this code, the more difficult it is to maintain or learn, especially if it is as sparsely documented as most such code I have seen. New team members can have an especially difficult time learning a byzantine code base which ends up strangling the work flow.

A facade can gradually fix this without an immediate or complete rewrite of the old code. Since a facade provides a unified interface to an complex, underlying system, it can also hide years of improvements, multitudes of styles, and unforeseen problems in the original code. A new interface that connects various legacy subsystems of the old code provides a way for programmers to replace the functionality later while creating new applications with the new interface. New programmers do not need to learn the entire system if one of the old salt programmers create a facade for them.

As more and more applications use the new facade, and hopefully fewer and fewer applications use the old

code base as programmers gradually replace them, the application base moves towards something much more maintainable since the application code does not rely on the underlying facade implementation. Programmers can re-implement portions of that at their leisure without breaking applications.

Typically, these sorts of facades pull together a particular way of doing things in a particular context such as a special business need or workflow. I might have several closely related applications, and as I develop them in parallel parts of them start to look the same because they do similar things and use the same resources. Such an application's first few lines might look like code listing 8 which uses several Perl workhorse modules.

Code Listing 8: Using several modules

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  require cgi-lib.pl;
5  use DBI;
6  use HTML::Parser;
7  use HTML::TreeBuilder;
8  use LWP;
9  use Text::Template;
10
11 # my Perl implementation here
12 __END__
```

I can refactor those applications and use a facade to contain all of the details about how I do the work. I want the facade to represent the task, not the method. If I use certain modules by local policy, then I only have to enforce that policy behind the facade rather than in every application. For this suite of hypothetical applications I create a module I call Tsunami³—the name I give my fictional product. Instead of all of the modules I use in code listing 8, including the notoriously old `cgi-lib.pl`, I only have to use one module, as in code listing 9. This also means that other team members, by policy if not practice, only use one module too. If, for some reason, policy changes so that Tsunami should use `CGI.pm` instead of `cgi-lib.pl`, I only have to change one file. If I improve Tsunami, everybody benefits.

Code Listing 9: Using one module

```
1  #!/usr/bin/perl
2
3  use Tsunami;
4
5  my $wave = Tsunami->new(...);
6
7  $wave->fetch('http://www.example.com');
8
9  my $title = $wave->title();
10
11 my $txt   = $wave->as_text();
12
13 print $txt;
14
15 __END__
```

³a deluge of code

7 A priori facades

If I have the luxury of prior thought and planning, and I know that some parts of the system resist planning, I can use a facade to present an application programming interface, and build up the rest of the stuff as I find out more and more about the problem.

Once I go through the object-oriented analysis process and have identified the objects, I can also identify the different ways that the programmers will use those objects. For instance, if I want to create an application to send messages between two computers, I know that I need a network object and a message object. These objects make it easy to deal with related sets of information my program must maintain.

I want to create a application that can “chat” with another, meaning that the two applications can send messages back and forth between each other. I can use a facade to represent a simple interface, with `send()` and `receive()` methods. I might need more later, but in this contrived example I pretend that I do not know everything this application might have to do because the specifications are fuzzy and the scope of the problem scares the project planners (in this case, me).

When I start programming, nothing works because I have not written any code to represent the objects, and at that level I need all of the objects for the other ones to do their part. Since a facade hides these objects, it also hides their absence as well. I can get something in place quickly, just as in my `HTML::SimpleExtractor` example in code listing 6, and be on my way.

As with all new programming I start, the first thing I do is write a test suite. Since I have no objects to test, all of the tests should fail, which is my first real test—tests fail when they should.

I create my module workspace with `h2xs`, which creates a `t` directory for test files⁴. In my test file I add the test in code listing 10. Since the `send()` method should die with an internal error message, I check to see that it does. So far the module `Chat.pm` is the template that `h2xs` created for me.

Code Listing 10: Test an unimplemented method

```
1 eval {
2     Chat->send('Come here Mr. Watson, I need you');
3 };
4 print $@ ? 'not ' : '', "ok\n";
```

One step beyond that I want to test that the parts of the interface exist, so I need an interface. I need to create the `Chat.pm` module. In code listing 11 I have a minimal module which has one class method, `send()`. I have not really implemented it yet, so I call the `die()` function if a program calls the method.

Code Listing 11: `Chat.pm`

```
1 package Chat;
2
3 use vars qw( $UNIMPLEMENTED );
4 $UNIMPLEMENTED = "Not implemented!";
5
6 sub send    { die $UNIMPLEMENTED }
7
8 1;
```

⁴see `Test::Harness` for details about testing

I expect the test from code listing 10 to fail because `send()` calls the `die()` function, but I can modify the test to see if I get the right message in `$@`. In code listing 12, the test succeeds even though the `send()` uses `die()`, since that is the behavior I expect—part of the interface now exists.

Code Listing 12: Test an unimplemented method

```
1 eval {
2     Chat->send('Come here Mr. Watson, I need you');
3 };
4 print "$@" eq "$Chat::UNIMPLEMENTED ?" : 'not ', "ok\n";
```

A similar test for an undefined method should give me a different sort of error. In code listing 13 I test to see if the `receive()` method exists, and if it does, then something is wrong. I have not defined `receive()` yet.

Code Listing 13: Test a non-existent method

```
1 eval {
2     Chat->receive('I am on my way');
3 };
4 print "$@" ? : 'not ', "ok\n";
```

Although I still do not have any objects, I can change my `send()` method to take arguments. In code listing 14 the `send()` method takes a message and a recipient argument, although I have not said anything about what they are. I have, however, made progress on the interface even though I still do not have anything to actually do the work.

Code Listing 14: `Chat::send()` with arguments

```
1 sub send
2     {
3     my( $class, $message, $recipient ) = @_;
4
5     return unless defined $message and defined $recipient;
6
7     return 1;
8     }
```

My test for `send()` changes to make sure it does the right thing for different argument lists. In code listing 15 I add three tests for different numbers of arguments. Only the call to `send()` with the right number of arguments should succeed. The other tests check for failure when `send()` should fail.

Code Listing 15: `send()` with different numbers of arguments

```
1 # should fail -- no recipient
2 eval {
3     Chat->send('Come here Mr. Watson, I need you');
4 };
5 defined "$@" ? not_ok() : ok();
6
```

```
7 # should fail -- no message or recipient
8 eval {
9     Chat->send();
10 };
11 defined $$ ? not_ok() : ok();
12
13 # should succeed
14 eval {
15     Chat->send('Come here', 'Mr. Watson');
16 };
17 defined $$ ? not_ok() : ok();
```

This process continues as I add more to the interface and as I implement the objects that will actually do the work. In the mean time, I have done useful work that has gotten me towards my goal, and I have created a suite of tests to help me along the way. The implementation does not concern me too much at this point because I can easily change it later. The rest of the project depends on the facade. Only the facade knows about the implementation, so everything else, including applications and tests, do not have to wait for the complete implementation to start work.

Once I progress far enough to have Message and Recipient objects, if I decide I need them, I can change my send() method to use them. In code listing 16 I check each argument to ensure that they belong to the proper class. Each object automatically inherits the isa() method from UNIVERSAL, the base class of all Perl classes, and returns TRUE if the object is an instance of the class named as the argument, or a class that inherits from it. All of my tests still do the same thing.

Code Listing 16: Argument checking

```
1 sub send
2     {
3     my( $class, $message, $recipient ) = @_;
4
5     return unless $message->isa('Chat::Message') and
6         $recipient->isa('Chat::Recipient');
7
8     return 1;
9     }
```

If later I change the objects or their behavior, I have not wasted too much time. My facade and its tests still work. Any application I have written does not need to change significantly. The facade handles the details and the interactions between the various objects. At the same time, other programmers can start to use the interface to create applications. The programs will not work until everything is complete, of course, but the programmers have a jump start on the process because they can code and test before everything is in place. They essentially work in parallel, instead of serially, with the programmers implementing the objects and the facade. The creation of classes is not a work flow bottleneck since the facade decouples the application and lower level implementations.

8 Perl modules which are facades

Several modules exhibit a Facade design pattern, although their authors may not have thought about design patterns or facades in particular. A good design stays out of the way and does not draw attention to itself. A good facade keeps most of us truly ignorant about whatever is behind it.

Most of the modules on CPAN with “Simple” in their name use the facade design pattern, including LWP::Simple, HTTP::SimpleLinkChecker, and HTML::SimpleLinkExtor which I used for examples.

8.1 LWP

The LWP family of modules act as a facade with a unified interface to various network protocols including HTTP, HTTPS, FTP, NNTP, and even the local filesystem. It can handle some protocol specific details too.

8.2 DBI

The DBI module provide a simple interface to several database servers or file formats. I can query several types of server or file formats with the same interface while DBI—actually the appropriate DBD—handles the connection, query, response, and other tasks. The DBI interface even allows me to change the database server behind the scenes (from SQL Server to postgresql, perhaps) without changing much more than the DBI->connect statement. I do not need to worry about the connection implementation, protocols, or data format.

8.3 Tied classes

Perl’s tie functionality is a type of facade. What looks like a normal Perl scalar, array, or hash stands-in for a possibly complex object behind the scenes. The most impressive use of this sort of facade, in my opinion, is the Win32::TieRegistry module, which represents the quite complex Microsoft Windows registry as a tied hash. This module even sees the following as equivalent:

```
\$Registry->{'LMachine/Software/Perl'}  
\$Registry->{'LMachine'}->{'Software'}->{'Perl'}
```

I do not have to know very much, if anything, about the Registry. I do need to know how I name a key, but I do not need to know how it is stored or accessed since I already understand Perl hashes and references. This module takes what I already know and lets me use it instead of learning low-level vendor interfaces.

This facade also allows me to develop on foreign platforms. I can reimplement the code so that I can have a Unix version of the Windows registry, for instance. This fake version of the Registry allows me to use all of the great tools and setups I already have on my unix accounts while targeting Windows platforms. If my application had to use explicit calls to the Windows programming interface I would not be able to do that.

9 Conclusion

The Facade design pattern provides a simple interface to a complex collection of modules or code. The Facade design pattern removes the details of the task from the application layer which allows me to focus on the task rather than the details, as well as allowing me to change implementations without changing my applications. This makes code more modular, maintainable, and easier to use. It decouples application code from the low level implementation which allows parallel development at different levels.

10 References

You can read more about design patterns in *Design Patterns*, Erich Gamma, Richard Helm, Ralph Johnson, Jon Vlissides, Addison Wesley, 1995.

Test suites for Perl modules usually use `Test::Harness`, although I find actual test files easier to understand. I have some simple tests in the `test.pl` of `HTML::SimpleLinkExtractor` distribution, or the `t` directory of `HTTP::SimpleLinkChecker`.

All real modules in this article are in the Comprehensive Perl Archive Network (CPAN)
<http://search.cpan.org>

11 About the author

brian d foy is the publisher of *The Perl Review* and the author of several modules available on CPAN, including `HTML::SimpleLinkExtor` and `HTTP::SimpleLinkChecker`. He teaches and writes on Perl topics in between doing real work.

Book Reviews

Debugging Perl

*Martin C. Brown; McGraw Hill; 0-07-212676-0;
March 2002; 225 pages
reviewed by Andy Lester*

Debugging Perl is an unfortunate title for Martin C. Brown's book. It belies the breadth of coverage of advanced topics in Perl that lie within. The author covers debugging, testing, optimizing and stretching Perl, along with a heavy dose of internals for good measure.

This is not a book for beginners. Brown starts right in with 20 pages on the parsing and execution of a Perl program. I was surprised to see Perl internals so early, but later chapters often rely on these details.

Although the book presents plenty of information, it lacks an overarching organization. Each of the 14 chapters covers a discrete topic, such as error trapping, code optimization, or debuggers, but within the chapter, the author haphazardly scatter informational tidbits. For example, the chapter "Program Design" includes such disparate concepts as dispatch tables, the two-parameter form of the `bless` function, and writing POD, but the author offers nothing to hold the bits together.

This approach also means that much of the information lacks the necessary detail. For example, Brown tells us not to use a certain `foreach` construct because it eats space: "Although this has been optimized in recent versions of Perl, it's still best avoided if possible". Which versions of Perl? And why? He never tells us.

The book's strength is in its discussion of actual debugging techniques. Sections on the signal handlers `_DIE_` and `_WARN_` and the importance of the caller function are strong, if a bit short. I was especially glad to see a section on writing to Windows 2000 system logs, since Win32 system so often get short shrift in Perl books.

Unfortunately I found a number of factual errors in the text. In the discussion of the `ref` function, Brown leaves out `LVALUE`, `IO`, `IO::Handle`, `Regexp` return values and, most importantly, any other user-defined

class, and `false` if the parameter is not a reference at all.

Production errors also mar the book. A table for the open function's modes appears on the page in the wrong place, in a block of text relating to the `sprintf` function. I cannot imagine how an editor or proofreader could have missed such a glaring error.

Overall, I cannot recommend purchasing the book new, although adding a used or remaindered copy to your bookshelf might be worthwhile.

Web Development with Apache and Perl

*Theo Petersen; Manning; 1-930110-06-5;
June 2002; 410 pages
reviewed by brian d foy*

I have not been satisfied by any book that talked about web servers since net.Genesis published *Build A Web Site* in 1995, but things were simpler then. Manning Publications has the latest effort in a saturated market with Theo Petersen's *Web Development with Apache and Perl*. Despite what some publishers think, web technology has not changed that much. If you want a book on web development, look for something that says Lincoln Stein on the cover and give this one a pass.

The book's scope is a bit wide, and since it's also only 400 pages, no topic ever gets the attention it deserves. I could forgive this if the author referred the reader to the many excellent resources on the web, or even in the software packages he discusses, but he leaves the reader wanting for more. The bibliography is a collection of his favorite resources rather than a reading list of relevant material. Check out the books in he lists and buy those instead, although I think he meant to recommend *HTML: The Definitive Guide* rather than *Dynamic HTML*, the mostly dead web technology.

I typically blame most of these sorts of problems on the editor who should have made sure the book was reviewed and fact-checked, but curiously, this book appears to be editor-less. Manning apparently only lists the copy-editor in its books—the copy is passable, but almost everything else needs help.