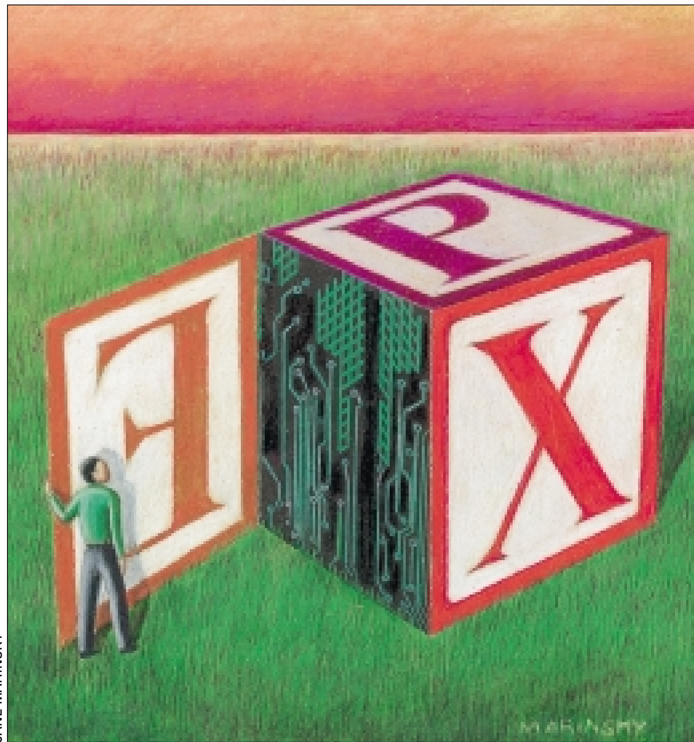
Work

by Jeffreys Copeland and Haemer



JANE MARINSKY

'Tis the gift to be simple, 'tis the gift to be free; / 'Tis the gift to come down where we ought to be
– Traditional Shaker hymn, which bears no relation to the complicated, expensive, up-in-the-air lives of your correspondents

Simpler is Better
– Dick Dunn

Simplify, Simplify

Last month, we wrote about our struggles with `Expect.pm`. To summarize our chief points:

- Expect is a wonderful tool for automating interactive programs, which lets you do lots of surprisingly interesting things.
- If you feel uncomfortable in Tcl, `Expect.pm` lets you do those things in Perl.
- `Expect.pm` offers completeness and object-orientation. It would be very useful to have an `Expect::Simple` subclass that gave you some simple calls to let you do common tasks.
- Some days, it doesn't pay to get out of bed.

To explore the third point, we began with this simple program to automate an anonymous ftp session:

```
#!/usr/bin/perl -w -s
# $Id: get_file,v 1.1 2001/04/08 ...

use File::Basename;
our $debug;
use Expect;
@ARGV == 1 or
 die "usage: $0 " .
  "ftp-site file\n";
my ($host, $path) =
 split ':', shift;
die "usage: $0 host:path"
 unless defined $path;
my ($file, $dirname) =
 fileparse $path;

my $t = 30;
my $ftp_prompt = 'ftp> ';

if ($debug) {
 $Expect::Debug=1;
 $Expect::Exp_Internal=1;
}

my $ftp =
Expect->spawn ("ftp", $host);
$ftp->expect($t, "Name") or
 die "Never got " .
  "username prompt " .
  "on $host, " .
  $ftp->exp_error() .
  "\n";

print $ftp "anonymous\r";
$ftp->expect($t, '-re',
 'Password:\s*') or
 die "Never got " .
  "password prompt " .
  "on $host, " .
  $ftp->exp_error() .
  "\n";

print $ftp $ENV{USER} .
 '@' . 'dnsdomainname' .
 "\r";
$ftp->expect($t, '-re',
 $ftp_prompt) or
 die "Never got ftp prompt " .
 "after sending username and " .
 "password, $ftp->exp_error()\n";

print $ftp "cd $dirname\r";
$ftp->expect($t, '-re',
$ftp_prompt) or
 die "Never got ftp prompt " .
 "after attempting to change " .
 "directories, ".
 $ftp->exp_error() .
 "\n";

$ftp->expect($t, '-re',
$ftp_prompt) or
 die "Never got ftp prompt " .
  "after attempting to get $file, " .
 $ftp->exp_error() . "\n";
```

```
$ftp->expect($t, '-re', "Goodbye.") or
 die "Never got 'Goodbye.', " .
  $ftp->exp_error() . "\n";

$ftp->hard_close();
```

After a little boilerplate, some argument processing, and some good housekeeping, the line beginning `my $ftp` spawns an ftp process and creates an expect object to talk to it.

The next three lines show a pattern that repeats itself through the rest of the program. The first of these lines looks for the string “Name” from the spawned ftp within 30 seconds. If it times out, the next two lines print an error message and exit. This at least cries out for subroutines.

We decided what we really wanted was a module that would let us write the same thing like this:

```
#!/usr/bin/perl -w -s
# $Id: goal2,v 1.1 2001/04/08 ...

use Expect::Simple;
use File::Basename;
our $debug;

@ARGV == 1 or die "usage: $0 ftp-site file\n";
my ($host, $path) = split ':', shift;
die "usage: $0 host:path" unless defined $path;
my ($file, $dirname) = fileparse $path;

timeout(30);
verbose if $debug;
spawn(cmd => 'ftp $host', ok => 'Name',
  prompt => 'ftp> ');
request(send => 'anonymous',
 ok => 'Password:\s*');
my $email =  $ENV{USER} . '@' . 'dnsdomainname';

# all unusual prompts complete
request($ENV{USER} . '@' . 'dnsdomainname');
request "cd $dirname";
request "get $file";
```

Our Subclass

The tricky part, it turns out, was finding a good mental model. The last listing above came from our fourth or fifth.

We spent about a week trying half a dozen designs that didn’t work on various simple tasks. Several of them had fundamental misunderstandings about what you can do when talking to a pty or a child process. Having our programs spawn subprocesses while trying to debug them did not exactly help, either.

Our biggest “Aha?” Instead of trying to create interfaces that waited for prompts and responded to them, like this:

```
reply(Name => 'anonymous');
```

we needed calls that issued commands and watched for acknowledgements, like this:

```
request(cmd => 'anonymous',
 ok => $password_prompt);
```

Why? Mostly because it lets our applications be in the driver’s seat. If we send a request and define the format of a proper response, we get the luxury of parsing everything in between. With a “reply” style design, we’re sending datagrams, and we have no idea what the response is without another call. We lost a lot of sleep learning that there’s a profound difference.

If you’ve been reading our columns for a while, you will realize we also like testing our code. Here is our test suite.

```
#!/usr/bin/perl -w -s
# $Id: test.pl,v 1.1 2001/04/08 ...

use Expect::Simple;
use strict;

sub print_tnp {
 print "timeout is " .
  (timeout || "undef") . "\n";
 print "prompt is " .
  (prompt || "undef") . "\n";
}

sub header {
 print "\n== @_ ==\n\n";
}
header "manipulate timeouts and prompts";
timeout(-1);
print_tnp;
timeout(10000);
prompt('foo? ');
print_tnp;
simple_reset;
print_tnp;
timeout(1);
prompt('bar! ');
print_tnp;

header "simple spawns";
verbose;
spawn(cmd => "date", timeout => 30,
 prompt => '$');
print_tnp;
spawn("date");
print_tnp;
taciturn;

header "requests";
simple_reset;
spawn("bash");
my $x = request("date");
print "$x\n";
my @x = request("ls");
foreach (@x) {
 print $_, "\n";
}
header "fancy stuff";
use File::Basename;

@ARGV = ("woodcock:/pub/foo/foo.notes");
@ARGV == 1 or die "usage: $0 ftp-site file\n";
my ($host, $path) = split ':', shift;
die "usage: $0 host:path" unless defined $path;
my ($file, $dirname) = fileparse $path;
```

```
spawn(cmd => 'ftp host', ok => 'Name',
 prompt => 'ftp> ');
request(send => 'anonymous',
 ok => 'Password:\s*');
my $email =  $ENV{USER} . '@' . `dnsdomainname`;
request(send => $email, ok => 'ftp> ');

request "cd $dirname";
request "get $file";
request "quit";
system "ls $file";
```

The `header()` commands state, simply, the intent of each section.

These test cases were an integral part of the development. Like our programs, we designed the test cases before we implemented the module and modified them as we realized we were asking the wrong questions.

Once these passed, we knew we had a solid design and implementation.

Here is our passing module, complete with documentation.

```
#!/usr/bin/perl -w
# $Id: Simple.pm,v 1.3 2001/05/05 ...

package Expect::Simple;

use Carp;
use Expect;
require Exporter;
@ISA = qw(Expect);
@EXPORT = @Expect::EXPORT;
push @EXPORT, qw(timeout prompt verbose);
push @EXPORT, qw(taciturn simple_reset);
push @EXPORT, qw(spawn request wait_for);
@EXPORT_OK = @Expect::EXPORT_OK;
$VERSION = '0.1';
$Expect::Log_Stdout = 0;

# everything in a block just
#   for safety's sake
{
 use strict;
 # internal routine to set defaults
 #  from an argument hash
 sub set_defaults {
  my @defaults = (
   prompt => $Expect::Simple::prompt,
   timeout => $Expect::Simple::timeout,
  );
  unshift @_, "cmd" if @_ == 1;
  my %defaults = (@defaults, @_);
  $Expect::Simple::prompt = $defaults{prompt};
  $Expect::Simple::timeout = $defaults{timeout};
  %defaults;
 }

 # set timeout, return old timeout
 sub timeout {
  my $oldtimeout = $Expect::Simple::timeout;
  my $timeout = shift || $oldtimeout;
  $timeout = undef
   if defined($timeout)
    && ($timeout < 0);
  $Expect::Simple::timeout = $timeout;
  $oldtimeout;
 }

# set default prompt, return old prompt
 sub prompt {
  my $oldprompt = $Expect::Simple::prompt;
  my $prompt = shift || $oldprompt;
  $Expect::Simple::prompt = $prompt;
  $oldprompt;
 }

 # debugging info
 sub verbose {
  push @_, "1" unless @_;
  my $v = shift;
  $Expect::Log_Stdout = $Expect::Exp_Internal =
   !($v == 0 || $v =~ /off|no|false/i);
 }

 # no debugging info
 sub taciturn { verbose(0); }

 # defaults
 sub simple_reset {
  $Expect::Simple::prompt = '^.*[$>#%]\s*';
  $Expect::Simple::timeout = undef;
  taciturn;
 }

 # spawn a process
 sub spawn {
  my %args = set_defaults(@_);

  my @cmd = split /\s+/, $args{cmd};
  $Expect::Simple::handle->close()
   if (defined $Expect::Simple::handle);
  my $self =
   $Expect::Simple::handle =
   Expect->spawn(@cmd);
  return undef unless defined $self;
  $self->expect($Expect::Simple::timeout,
   -re, $args{ok} || $Expect::Simple::prompt);
 }

 # send a request, wait on an ack,
 # return the requested stuff
 sub request {
  unshift @_, "send" if @_ == 1;

  my $h = $Expect::Simple::handle;
  my @defaults = (
   timeout => $Expect::Simple::timeout,
   ok => $Expect::Simple::prompt,
  );
  my %f = (@defaults, @_);

  print $h ($f{send}, "\r") if defined $f{send};

  my $t = $f{timeout};
```

```
  $h->expect($t, '-re', $f{ok})
   or return $h->exp_error();
  my $resp = $h->exp_before();
  if ($f{send}) {
   my $leader = "^" . $f{send} . "\r\n";
  $resp =~ s/$leader//ms;
  }
  return $resp unless wantarray;
  return split /[\r\n]+/, $resp;
 }

 # wait for a particular event
 sub wait_for {
  unshift @_, "ok" if @_ == 1;
  request(@_);
 }

 simple_reset;

}

1;

=head1 NAME

Expect::Simple - Simplified Expect interface

=head1 SYNOPSIS

 use Expect::Simple
 timeout(10);
 prompt('ftp> ');
 spawn(cmd => "date", timeout => 30,
  prompt => '$');
 @x = request("ls -l");
 wait_for "->Idle";

 simple_reset;
 verbose;
 taciturn;

=head1 DESCRIPTION

This module provides a simplified
interface to C<Expect.pm>.

The following functions are
exported by this module:

=over 3

=item timeout($time)

Set timeout, return old timeout.

Default timeout is C<undef>.

=item prompt($prompt)

Set default prompt, return old prompt.
Throughout this module, a "prompt"
is something that
a call waits for from the spawned process.
In all cases, the prompt is specified
as a regular expression.

The default prompt is C<^.*[$>#%]\s*>.

=item verbose()

Turn on debugging info.
=item taciturn()

Turn off debugging info.

=item simple_reset()

Reset the prompt and timeout to default values.
Turn off debugging.

=item spawn(cmd => $process,
 timeout => $t, prompt => $p)

Spawn a process.  Overloads L<Expect::spawn>.
Guarantees a single subprocess,
waits for a prompt to check for success.
Returns C<undef> on failure.

Called with a single argument,
assumes that argument is the process
to spawn, uses the default timeout and prompt
(C<Expect::timeout> and C<Expect::prompt>).

=item request(send => $string,
 ok => $ack, timeout => $t)

Send a request to the spawned process,
wait for an acknowledgement.
Returns the entire response either
as a single string
with embedded newlines, or as an array of lines.

Uses default prompt and timeout if unspecified.

Called with a single argument,
sends that string as a request.

Appends carriage returns to the send string.

=item wait_for($prompt)

Wait for a prompt.
Returns all text leading up to the prompt.

=back

=head1 AUTHORS

Jeff Copeland <jeff@alumni.caltech.edu>

Jeffrey S. Haemer <jsh@usenix.org>

=head1 SEE ALSO
```

```
  Expect.pm

=cut
```

As proof that we were where we wanted to be, we spent a few minutes writing a Perl version of a program we'd written long ago in Expect, which times the progress of print jobs by watching a telnet session to a printer.

```
#!/usr/bin/perl -w
# $Id: click2clunk,v 1.2 2001/05/05 ...

use Expect::Simple;
use strict;

sub parse_opts {
 use Getopt::Long;
 use Pod::Usage;
 my ($help, $man, $printer);

 my $usage = "usage: $0 [-p printer] file ...";

 GetOptions('printer=s' => \$printer,
  'help|?' => \$help, man => \$man)
  or pod2usage(2);
  pod2usage(1) if $help;
  pod2usage(-exitstatus => 0,
          -verbose => 2) if $man;

 @ARGV && ($printer ||= $ENV{PRINTER})
     or pod2usage(2);
 -r or die "can't read $_\n" foreach @ARGV;
 $printer;
}

my $printer = parse_opts;

my $printcmd = ">/dev/null 2>&1 " .
 /rd/local/bin/hpsend -h $printer";

spawn(cmd => "telnet $printer",
  prompt => '->Idle');
foreach (@ARGV) {
 system "$printcmd $_";
 $? == 0 or die "$printcmd: $@";
 wait_for "Starting job";
 my $t = time;
 wait_for "->Idle";
 print "$_: ", time - $t, "\n";
}
=head1 NAME

click2clunk -- time a print job

=head1 SYNOPSIS

click2clunk [--printer printername]
  file [file ...]

=head1 DESCRIPTION

=over 2

Times print jobs using Minolta-QMS
remote console.

=back

=head1 OPTIONS AND ARGUMENTS

=over 2

=item I<--printer|-p>

The printer to use.
If unset, uses C<$PRINTER>.

=item I<file>

The file(s) to print.

=back

=head1 AUTHORS

Jeff Copeland <jeff@alumni.caltech.edu>

Jeffrey S. Haemer <jsh@usenix.org>

=head1 SEE ALSO

  Expect::Simple

=cut
```

An earlier version replaced a platoon of interns with stopwatches and blistered thumbs.

Finishing up

Whew! It took us two months, but we now have a nice, reasonably intuitive interface to Expect through Perl.

Next time, we'll think about parsing XML. In the meantime, we leave you with a non-programming exercise for the reader. Assuming you were a Russian cosmonaut, and had a ready supply of potatoes on board your space station, how would you build a still in zero-gravity so you could have vodka? (Which leads to the punchline, "hey, don't try to dock the supply ship if you've been drinking.") We thank our friend Jason Zions, standards geek extraordinare, for this one.

Until next time, happy trails.

Jeffrey Copeland (copeland@alumni.caltech.edu) *is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer (jsh@usenix.org) *works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at http://alumni.caltech.edu/~copeland/work *or alternately at* ftp://ftp.cpg.com/pub/Work.