

The Perl Journal

Inversion of Control in Perl

Stevan Little • 3

Making Servers Dynamically Configurable

Stephen B. Jenkins • 7

Keeping Up with the World

Simon Cozens • 11

Geolocation in Perl

brian d foy • 15

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

Beginning Perl • 17

Source Code Appendix • 19

LETTER FROM THE EDITOR

Return to the Web

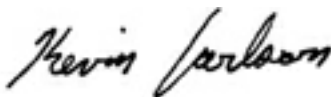
It seems that since the dot-com implosion, there has been a tendency to forget about the Web's potential for economic growth. Two years ago, incestuous circular advertising links between web sites and declining click-through rates led many to pronounce online advertising dead. Many online ad sales staff lost their jobs, and companies shelved online investment until something better than the banner ad came along.

But it appears there's life in the banner ad yet. According to *The New York Times*, online advertising in 2004 is expected to reach \$9.7 billion, or 3.7 percent of U.S. advertising spending, and is expected to grow 19 percent in 2005. Online news sites, in particular, can't keep up with demand and are looking for new ways to provide enough pages to fill ad inventory. That just might put a few online ad execs back in the work force.

It also has tremendously positive repercussions for Perl programmers and others who write the code that delivers those online page views. To acquire enough online ad-space inventory, media companies are buying up each other's most popular sites and adding content to their portfolios. All of this new content must be delivered dynamically, which means lots of lines of content-management code behind those web sites. Add to this the need to reorganize and repackage this content in new ways to maximize its reach and penetration, and you have a significant new burden in content-management chores.

Companies will undoubtedly rely as much as they can on off-the-shelf, turnkey systems to meet this burden. But they're going to need programmers. Anyone who has tried to deploy a turnkey content-management product for a media organization of any size knows this. Such systems rarely work right out of the box. They need to integrate with existing systems, and often need extra bits of functionality before they really meet the business plan. Even if it does fit the bill without modification, that rarely lasts for long. It's almost inevitable that two years down the road, the business plan will look very different, and so will the required functionality. The only way to meet these needs with any kind of agility is with programmers on staff to stitch together the pieces of content-management functionality on the fly.

The question that should be on the minds of these programmers is whether media companies will realize this fact, or instead spend another couple of years bumbling around with systems that cost more than the yearly salaries of a team of programmers and don't produce results like a team of programmers. There is perhaps reason to hope—an admittedly unscientific search on Craigslist.org for jobs in the SF Bay Area with "Perl" in the description currently turns up 281 hits, a big improvement over two years ago. The Web is not slowing down, nor is it settling into any kind of groove. Without programmers on staff, large content-management systems, whether proprietary or open-source, lock companies into ways of doing business that quickly become obsolete. There's nothing so agile as the human brain—the media industry should make sure they retain a few if they want their content management to be capable of the gymnastics necessary to make money online.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2005 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Wyser

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, Brian d'Joy, Moshe Bar, Andy Lester

CONTRIBUTING EDITORS

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

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Michael Goodman

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403

650-513-4300. <http://www.tpj.com/>

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING

Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT/GROUP DIRECTOR INTERNET BUSINESS

Mike Azara

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

PRESIDENT, CHANNEL GROUP Robert Faletra

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED TECHNOLOGIES

Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK COMPUTING

ENTERPRISE ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE DEVELOPMENT

MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING SOLUTIONS

Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES Marie Myers

Perl News

Perl 6 Grammar Ready for Hacking

Luke Palmer and Patrick R. Michaud have unveiled their work on a formal grammar for Perl 6, and thrown the doors open to contributions from volunteers. “Perl 6 is a huge language, so the task seems better done incrementally by the community,” Luke wrote on the Perl 6-compiler mailing list (<http://www.mail-archive.com/perl6-compiler@perl.org/msg00135.html>). “It’s written in a top-down fashion so far (because that’s how my brain works with grammars), but feel free to work on it bottom-up, left-right (though recall that we have to make an LL grammar :-), inside-out, whatever. Let’s just get rules written. Also, document your work as much as possible. We won’t be accepting new rules without explanation unless they are **really** trivial...Most of all, hack liberally! Don’t be afraid to change stuff around. These patches are easy to write, and they can be as small or as large as you feel necessary. It’s also helpful to make suggestions even if you don’t have a patch.”

“When there’s a doubt about how to do something,” Patrick added, “we’ll follow whatever is written in the most recent Synopsis/Apocalypse/whatever, and if it’s a major language design issue, we’ll kick it back to perl6-language for discussion...We’re still working out some of the details of parsing, including operator precedence. As a result some things may be handwavy at first—that’s normal.”

A subversion repository has been set up at <https://svn.perl.org/perl6>; the Perl 6 grammar is currently archived under `grammar/trunk/Grammar.perl6`, although Patrick notes that “this and the overall repository structure will certainly change over time.” Patches to the grammar can be contributed to the perl6-compiler list.

Passage of the Pumpking

Hugo van der Sanden has stepped down as pumpking of perl-5.10; “As work continues to demand more of my time, it has become clear that I will not be able to devote the time required for the role for the foreseeable future. This also means that I do not expect to be able to undertake the major reworking of the regexp engine I had planned, though I still hope to find time to implement some smaller improvements,” he announced on the perl5-porters mailing list (<http://www.nntp.perl.org/group/perl.perl5.porters/97896>). Rafael Garcia-Suarez has been crowned the new pumpking.

Adam Kennedy Awarded Perl Foundation Grant

The latest grant from the Perl Foundation goes to Adam Kennedy to complete his work on PPI, a round-trip document parser for Perl source code. “That is,” Adam wrote in his grant application, “it parses a contiguous chunk of Perl source as a document and not as code, and is able to perfectly reassemble a parsed Perl document back into the original file (including whitespace and comments).”

Perl’s dynamic grammar makes it very difficult—if not impossible—to truly parse. Characters with multiple uses such as “/”, which can be either a divide operator or the beginning of a

regular expression, along with the potential ambiguity of function signatures, give rise to the saying that “the only thing that can parse Perl is Perl.” Or, as the PPI documentation (<http://search.cpan.org/perldoc?PPI>) puts it, “A true or ‘real’ parser needs information that cannot be found in the immediate vicinity. In fact, this information might not even be in the same file. It might also not be able to determine this without the prior execution of a *BEGIN* {} block. In other words, to parse Perl, you must also execute it, or if not it, everything that it depends on for its grammar.”

PPI, therefore, no longer stands for *Parse::Perl::Isolated*: “In acknowledgment that someone may some day come up with a valid solution for the grammar problem, it was decided to leave the *Parse::Perl* namespace free. The purpose of this parser is not to parse Perl code, but to parse Perl documents. In most cases, a single file is valid as both. By treating the problem this way, we can parse a single file containing Perl source isolated from any other resources, such as the libraries upon which the code may depend, and without needing to run an instance of Perl alongside or inside the the parser (a possible solution for *Parse::Perl* that is investigated from time to time).”

Possible applications for PPI include automatically generating documentation; applying quality metrics or performing structural analysis; refactoring code; or changing the layout or presentation of code without changing its text. PPI is currently available in beta form from CPAN; The Perl Foundation’s \$3500 grant will support Adam for two months of work as he finishes the final version.

Calls for Proposals

The sixth International Free Software Forum, fisl6.0, is seeking speakers: “Submission of Perl-related proposals is much welcome,” according to conference organizer Flavio Soibelmann Glock. Lectures will be 60 minutes long, including discussion time. The International Free Software Forum is scheduled for June 1–4 in Porto Alegre, Brazil; submissions are due by February 28. The call for papers is at <https://fisl.softwarelivre.org/papers/index.en.html>.

YAPC North America has also released a call for participation. Lightning talks, standard 20-minute talks, long and extra-long talks will be offered, along with half-day or full-day tutorials. “Submissions regarding all Perl topics are welcome,” the organizers state, “but they must be technical topics aimed at the benefit of the attendees (not marketing talks aimed at the benefit of some product or service associated with the presenter).” YAPC::NA::2005 will be held in Toronto, Canada on June 27–29. The deadline for proposals is April 18. See <http://yapc.org/America/cfp-2005.shtml> for details.

YAPC::EU is gearing up as well; the theme this year is “Perl Everywhere,” so talks about Perl on varied platforms are especially encouraged. The conference will take place in Braga, Portugal from August 31 to September 2. Proposals are due by May 15, and can be submitted through the wiki at <http://braga.yapceurope.org/index.cgi?TalkProposals>.

Inversion of Control In Perl

Inversion of Control (IoC) is the very simple idea of releasing control of some part of your application over to some other part of your application or an outside framework.

IoC is a common paradigm in GUI frameworks, whereby you give up control of your application flow to the framework and install your code at callback hooks within the framework. For example, take a very simple command-line interface: The application asks a question, the user responds, the application processes the answer and asks another question, and so on until it is done. Now consider the GUI approach for the same application: The application displays a screen and goes into an event loop, users actions are processed with event handlers and callback functions. The GUI framework has inverted the control of the application flow and relieved your code of having to deal with it.

If you hang around enough Java programmers, chances are you have heard the phrase “inversion of control” thrown around recently. IoC is also sometimes referred to as “dependency injection” or the “dependency injection principle,” and many people confuse the two. IoC and dependency injection are not the same, and in fact, the concepts behind dependency injection are actually just an example of IoC principles in action (in particular, as they relate to your application’s dependency relationships). IoC is also sometimes referred to as the “hollywood principle” because of the “don’t call us, we’ll call you” approach of things like callback functions and event handlers.

Despite its current overhyped buzzword status, IoC is nothing new. It is a time-tested concept that most programmers use and encounter on a daily basis. And while the ideas behind the current dependency injection hype have been around for several years, they are now reaching a level of maturity where their widespread use is becoming practical.

Howard Lewis Ship, the creator of the HiveMind IoC Framework, once referred to dependency injection as being the inverse of garbage collection. With garbage collection, you hand over the details of the destruction of your objects to the garbage collector. With dependency injection, you are handing over control of object creation, which also includes the satisfaction of your dependency

relationships. In his opinion, just as many programmers today would never want to go back to manual memory management—soon programmers will forget the days when they had to manage the creation of their own objects.

This is just the tip of the IoC iceberg, though. If you expand the idea of managed object creation, it becomes clear that it can be used for much more than just creating objects and handling dependency relationships. It becomes much simpler to do a number of things that would otherwise complicate your design; for instance, wrapping your objects in AOP-like method call tracers/proxies, silently substituting mock objects for real ones during testing, or exerting transparent and fine-grained control over the lifecycle of your objects. But all this is heavily in the abstract, so let’s get down to something more concrete.

IOC in Action

I recently released an IoC framework to CPAN called *IOC*, which is available at <http://search.cpan.org/~stevan/IOC/>. It is still a work in progress, and its current focus is on object creation and managing dependency relationships. Future directions will include more work with object proxies and AOP-style technologies. The following is a discussion of how you can use *IOC* in its current form.

Containers

The central part of just about any IoC framework is the container. A container’s responsibilities are roughly to dispense objects, to handle the resolution of said object’s dependency relationships, and to manage the lifecycles of the objects.

Let’s start with dispensing objects. First, we must create a container for our objects to live in and give it a name:

```
my $c = IOC::Container->new('MyApplication');
```

Next, we need to add a component to that container and give it a name. In the *IOC* framework, we use an *IOC::Service* to wrap and manage the lifecycle of our components:

```
$c->register(IOC::Service->new('logger'  
=> sub { My::Logger->new() }));
```

Now, if we want an instance of our logger component, we simply ask the container for it:

Stevan is the Senior Developer at Infinity Interactive, a small NYC consultancy specializing in building LAMP applications for a number of corporate clients. Stevan can be contacted at stevan@iinteractive.com.

```
my $logger = $c->get('logger');
```

Pretty simple.

Dependency Management

Dependency management is also quite simple and is easily shown with an example. But first, let's create another component for our container—a database connection:

```
$c->register(IOC::Service->new(
  'db_conn' => sub {
    DBI->connect('dbi:mysql:test', '', '')
  }
));
```

Now, let's add an authenticator to our container. The authenticator requires both a database connection and a logger instance in its constructor:

```
$c->register(IOC::Service->new(
  'authenticator' => sub {
    my $c = shift;
    My::Authenticator->new(
      $c->get('db_conn'), $c->get('logger')
    );
  }
));
```

As you can see, the first argument to our service subroutine is actually our container instance. Through this, we can then resolve the authenticator's dependency relationships.

Lifecycle Management

The default lifecycle for `IOC::Service` components is that of a Singleton, which means each time we ask for, say, the logger, we will get the same instance back. There is also another option for lifecycle management that we call “prototype.” It is worth noting that this is not the same as prototype-based OO and should not be confused with that. Here is an example of how we would use the prototype lifecycle:

```
$c->register(IOC::Service::Prototype->new(
  'db_conn' => sub {
    DBI->connect('dbi:mysql:test', '', '')
  }
));
```

Now, each time we request a new database connection from our container, we will get a new one. Being able to change between the different lifecycles by simply changing the service wrapper will come in handy as your application grows. Extending this idea, it is possible to see how you could create your own custom service objects to manage your specific lifecycle needs, such as a pool of database connections.

Unit Testing with Mock Objects

We have now decoupled the creation of our authenticator from the creation of both our logger instance and our database connection. We have already seen that the respective lifecycles of each component are irrelevant to their sibling components and can be varied as needed. And since nothing is referred to directly, only by name through the container, we are free to change the details and/or implementation of our components and to do something like substitute mock components.

In the unit tests for our authenticator, we might want to use the mock DBI driver `DBD::Mock`. To do this, we would only have to change the details of our `db_conn` service and none of our other components would be the wiser:

```
$c->register(IOC::Service->new(
  'db_conn' => sub {
    DBI->connect('dbi:Mock:mysql', '', '')
  }
));
```

We could also substitute our own mock logger—one that would check the expected log statements without needing to parse a log

*A container's responsibilities
are roughly to dispense objects,
to handle the resolution of
said object's dependency
relationships, and to manage the
lifecycles of the objects*

file. In abstracting away the details of the creation of an object, IoC makes the use of mock objects a much simpler process than it might otherwise be.

Debugging with Proxies

As we have seen with lifecycle management and switching in mock objects in our unit tests, we can do a number of things to a component behind the scenes and not have it affect our other components. The `IOC` framework also offers another means of doing this through its `IOC::Proxy` module.

`IOC::Proxy` implements a (mostly) transparent proxy package around a particular instance of a component. Through this proxy, we can actually capture each method call and do pretty much anything we like. The most obvious usage is for logging method calls for debugging purposes. Here is an example of a basic debugger proxy:

```
my $p = IOC::Proxy->new({
  on_wrap => sub {
    my ($p, $object_being_wrapped,
      $proxy_package) = @_;
    warn(">>> wrapping $object_being_wrapped\n"
      with $proxy_package);
  }
  on_method_call => sub {
    my ($p, $method_name, $full_method_name,
      $method_args) = @_;
    warn(">>> $method_name called with [" .
      (join ", " => @$method_args) .
      "] dispatching to $full_method_name\n");
  }
});
```

In order for this to be useful, we need to install the proxy on a component. There are two ways of accomplishing this. Either we can associate our proxy with a component at registration time using the `registerWithProxy` method, like so:

```
$c->registerWithProxy(IOC::Service->new(
    'authenticator' => sub { ... }, $p
));
```

or we can add the proxy to a component by name, using the *addProxy* method:

```
$c->addProxy('authenticator', $p);
```

Howard Lewis Ship once referred to dependency injection as being the inverse of garbage collection

The result is that when you ask for our authenticator component, you will actually be given back a proxy object, which is almost indistinguishable from the real component. The proxy object actually does quite a lot to cover its tracks and will respond as expected to *isa* and *can* (including *UNIVERSAL::isa* and *UNIVERSAL::can*) and will not leave any information to be found in the output of caller. It will even handle overloaded operators and AUTOLOAD correctly. The only place in which the proxy object is easily detectable is when it is passed to *ref*.

This is just the base for the proxying possibilities. There are plans in the works for creating more specialized proxy objects and these will likely show up in future releases of *IOC*.

IOC in Detail

I have shown how to use *IOC* to manage components lifecycles, their dependencies, and even how the *IOC* framework itself can be used to help unit testing and debugging. All of this can be done with a fairly high degree of decoupling and only a small configuration overhead. For the most part, these represent the most common and basic usage of the *IOC* framework, but as your application grows, there are other, deeper parts of the *IOC* framework that might come in handy.

Service Creation/Injection Styles

So far, I have shown the default way of creating a service or component, using an anonymous subroutine. But this is not the only way to go about this. Those who have encountered IoC in the Java world may be familiar with the idea that there are three “types” of IoC/dependency injections: constructor injection, setter injection, and interface injection. In *IOC*, we support both constructor and setter injection. However, I decided that interface injection was not only too complex, but highly java-specific, and the concept did not adapt itself well to Perl. The three injections styles *IOC* supports are as follows.

Block injection. This is *IOC*’s default style of injection. It is not one of the official three types (mostly because it’s not possible in Java), although it can be found in a few Ruby IoC frame-

works, and “Block injection” is my own term for it. It could be viewed as a compound injection style, in that it can be used to mix constructor and setter injection. Take this example:

```
my $s = IOC::Service->new('app' => sub {
    my $c = shift;
    my $app = My::Application->new($c->get('authenticator'));
    $app->setLogger($c->get('logger'));
    $app->setDatabaseConnection($c->get('db_conn'));
    return $app;
});
```

The *My::Application* object requires an authenticator in its constructor, and then a logger instance and a database connection are added through setter methods. This allows for a great deal of flexibility with component creation and especially tends to work best when retrofitting an application with *IOC*.

Constructor injection. With constructor injection, the container calls the object’s constructor and feeds it the required arguments. This promotes what is called a “good citizen” object, or an object that is completely initialized upon construction. This is the style used and popularized by the Pico Container Java Framework. *IOC* supports this style through the *IOC::Service::ConstructorInjection* and *IOC::Service::Prototype::ConstructorInjection* modules. Here are some examples of how constructor injection is done in *IOC*:

```
my $s = IOC::Service::ConstructorInjection->new(
    'db_conn' => (
        'DBI', 'connect', [ 'dbi:mysql',
                           'user', '*****' ]
    )
);
```

Dependencies are managed somewhat differently with constructor injection than they are with the default block injection. Here is an example of how that looks:

```
my $s = IOC::Service::ConstructorInjection->new(
    'authenticator' => (
        'My::Authenticator', 'new', [
            IOC::Service::ConstructorInjection
                ->ComponentParameter('db_conn'),
            IOC::Service::ConstructorInjection
                ->ComponentParameter('logger'),
        ]
    )
);
```

As you can see, the class method *ComponentParameter* is used as a placeholder for any dependencies that need to be resolved at component creation time. There are advantages to this style, one of which is that it promotes the “good citizen” object pattern. However, it is easy to see how a constructor could get very messy as the number of dependencies are increased.

Setter Injection. *IOC* also supports setter injection. The idea behind setter injection is that for each component dependency, a corresponding *setter* method is created. This style has been popularized by the Spring Java Framework. *IOC* supports this style with its *IOC::Service::SetterInjection* and *IOC::Service::Prototype::SetterInjection* modules. Here is an example of how setter injection is done with *IOC*:

```
my $s = IOC::Service::SetterInjection->new(
    'app' => sub {
        'My::Application', 'new' => [
```

```

    { setAuthenticator      => 'authenticator' },
    { setLogger            => 'logger'          },
    { setDatabaseConnection => 'db_conn'        },
  ]
}
);

```

As you can see, for each *setter* method, we provide a component name to be passed to it. This style has its advantages, but one obvious problem is the public nature of the setters.

Hierarchal Containers

So far, I have shown basic containers that only have a single level of components. As your application grows larger, it may become useful to have a more hierarchal approach to your containers. *IOC::Container* supports this behavior through its many subcontainer methods. Listing 1 is an example of how we might rearrange the previous examples using subcontainers.

We have introduced the *IOC::Container* method *find* in this example. This method can be used to find a component that is stored outside of the current components container and it supports a basic path-like syntax for climbing your container hierarchy.

Global Container Registry

IOC provides a global container registry through the module *IOC::Registry*. *IOC::Registry* is a Singleton and can be used to store and then access all of your containers and services; it also includes the ability to search for services and containers within your hierarchy. You would likely initialize your registry Singleton in the same configuration file where you created all your containers. So utilizing the code from Listing 1, your registry might look like this:

```

my $r = IOC::Registry->new();
$r->registerContainer($app_c);

```

Now, from anywhere else in your code, you could do something like this:

```

# get the singleton instance
my $r = IOC::Registry->instance();

# and now try to find a particular service

```

```

my $s = $r->searchForService('laundry')
|| die "Could not find the laundry service";

# or for a particular container
my $s = $r->searchForService('database')
|| die "Could not find the database container";

```

You can also address services and containers by paths, much like the *find* method of the *IOC::Container* object. That would look like this:

```

my $db_conn = $r->locateService('/database/connection');

```

The registry object can serve as a central point for all your object dispensing needs, and reduce a sometimes complex collection of Singletons and object factories.

The Future of IOC

IOC is still very much a work in progress. Its current version as of this writing is 0.11, but no doubt that will change as I am taking the “release early/release often” approach to this module. Here are some directions *IOC* may take:

- **Handling cyclical dependencies.** Currently *IOC* does not handle cyclical dependencies. It can catch the cyclical dependency and will throw an exception when it is found. I am currently looking for a way to overcome this problem and hope to have it resolved soon.
- **XML or YAML configuration.** I have been experimenting with both XML- and YAML-based configuration files, which could be used to automatically create containers.
- **Dependency visualization.** As an application grows, it can become more and more difficult to visualize all the interdependencies within it. I can see a real use for having the ability for *IOC* to visualize those dependencies for you using some kind of graph drawing tool like *GraphViz*.
- **More proxies.** The current *IOC::Proxy* object is just the beginning of what can be done using this AOP-like approach. I have a number of ideas for all sorts of funky proxy objects.

TPJ

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

Listing 1

```

my $app_c = IOC::Container->new('app');

my $db_c = IOC::Container->new('database');
$db_c->register(IOC::Service->new('dsn' => sub { 'dbi:mysql:test' }));
$db_c->register(IOC::Service->new('username' => sub { 'user' }));
$db_c->register(IOC::Service->new('password' => sub { '*****' }));
$db_c->register(IOC::Service->new('connection' => sub {
    my $c = shift;
    return DBI->connect(
        $c->get('dsn'),
        $c->get('username'),
        $c->get('password'));
    });
});

$app_c->addSubContainer($db_c);

my $log_c = IOC::Container->new('logging');
$log_c->register(IOC::Service->new('log_file', sub {
    '/var/log/app.log'
}));
$log_c->register(IOC::Service->new('logger', sub {
    My::Logger->new((shift)->get('log_file'))
}));

```

```

$app_c->addSubContainer($log_c);

my $sec_c = IOC::Container->new('security');
$sec_c->register(IOC::Service->new('authenticator' => sub {
    my $c = shift;
    My::Authenticator->new(
        $c->find('../database/connection'),
        $c->find('../logging/logger')
    );
}));

$app_c->addSubContainer($sec_c);

$app_c->register(IOC::Service->new('app' => sub {
    my $c = shift;
    my $app = My::Application->new($c-
>find('/security/authenticator'));
    $app->setLogger($c->find('/logging/logger'));
    $app->setDatabaseConnection($c->find('/database/connection'));
    return $app;
}));

```

TPJ

Making Servers Dynamically Configurable

In the course of my job at the Aerodynamics Laboratory, I typically have to write two or three TCP/IP servers a year. They can range from simple, single-client status applications to complex forking daemons servicing multiple clients that are critical to the execution of our experiments. Regardless of size, all of these servers have two things in common: They are expected to run unattended for long periods, and they need to have their configurations changed from time to time. By configuration, I mean things such as verbose mode (on or off), error reporting (off, to a log file, or via e-mail), and current working directory. I've found that the best way to provide for these types of changes is to make the servers dynamically configurable, rather than the traditional method of requiring the application to be stopped and restarted with new command-line or configuration-file values. Because most of my code is running in a web-based environment already (see "Reference" for a previous article I've written about this environment), it makes sense for me to use the Web for this task as well. And while I'm going to all this trouble, I may as well add some status-reporting abilities.

A Simple Example

Because it's easier to explain this type of concept by example, I've included complete programs for a simple server and a command-line client to test it, as well as some snippets of code for a CGI configuration editor. I'll explain the server and test client first, then move on to the CGI program. Please note that in order to keep this example as simple as possible, I've decided to ignore several important issues that I'll discuss after the basics have been covered.

The Server

The program shown in Listing 1 (available electronically at <http://www.tpj.com/source/>) is a simple server that returns a date and

time string when a connecting client sends it a *time* message. After the usual preamble to *use* the pragmas and modules, I create an anonymous hash and populate it with the default configuration settings. I then set up the server socket and block, waiting for clients to connect. When that occurs, the *nummess* counter is incremented, and the incoming message is compared to the three types of requests that the server understands: *time*, *status*, and *config*. If the client is requesting the time, a string is created and sent off, the socket is closed, and the message and its time are recorded in the configuration hash. The other two types of requests are more complex as well as more interesting.

If the client is requesting the current status of the server, *Data::Dumper* is used to serialize the master configuration hash into a single string; this will allow clients to reconstruct the hash just by using *eval* on the *Dumper* output. The configuration string is sent to the client, and as before, the socket is closed and the message and its time are recorded.

If the client is requesting a configuration change, each key of the configuration hash is used to search the incoming message for *key=value* pairs. If the two keys match, the configuration hash is updated with the new value. The configuration message and its time are stored in their own location in the master hash. Once again, the socket is closed and the message and its time are recorded.

Lastly, if the message doesn't contain an understandable request, the server responds to the client with a polite and informative reply.

The Test Client

The test client shown in Listing 2 (available electronically at <http://www.tpj.com/source/>) is much simpler than the server because the user is expected to do most of the work.

After the appropriate *use* statements, the client's TCP/IP information is initialized, and a *while* loop is used to block waiting for input from STDIN. When a command is entered, a connection is made to the server and the user's input is sent. The reply is read and displayed, the socket is closed, and the *while* loop waits for more input.

Stephen is the senior programmer/analyst at the Aerodynamics Laboratory of the Institute for Aerospace Research, National Research Council of Canada. For more information, go to <http://www.erudil.com/>.

Putting Them Together

The easiest way to see how everything works is to run the server and client programs in separate terminal or command windows. Try entering *time* and *status* requests in the client to ensure that it and the server are running properly. You should see the request echoed in the server window and the response in the client window. Next, enter a configuration change request such as *config*

The biggest complication for this technique in the real world is security

verbose=0 defpath=/foo/bar/. You can use a *status* request or the server's console output to verify your changes. That's it. You now have a dynamically configurable server, although it's rather awkward to use with the test client.

The Web-Based Client

Since the reason for adding dynamic configuration to the server was to make life simpler, the next step is to replace the command-line client with an easy-to-use web-based client. Because of space limitations and all the boilerplate code needed for CGI programs, I'll only show snippets of the most important parts of the web client. The complete program, as well as the server and test client programs, is available at <http://www.tpj.com/source/>.

In this first code block, I check to see if the form is being submitted. If not, the program calls *getconfig()*.

```
#
# check for submit button
#
if( param('Submit') ) {
    &handlesubmit($remotehost,$port); # this never
                                     # returns
}

#
# otherwise, get the current config
#
my $config = &getconfig($remotehost,$port);
```

In this routine, a connection is made to the time server and the current configuration structure is requested with a *status* command:

```
sub getconfig {
    my $remotehost = shift;
    my $port       = shift;
    #
    # connect to the server and get the serialized
    # configuration structure
    #
```

```
my $socket = IO::Socket::INET->new(
    PeerAddr  => $remotehost,
    PeerPort  => $port,
    Proto     => "tcp",
    Type      => SOCK_STREAM )
or &htmllexit("<h2>Couldn't connect to
             server.pl : $@\n</h2>");

print $socket "status\n";
my $status = <$socket>;
chomp $status;
close($socket);

#
# turn the Dumper output back into a data structure
#
if( $status =~ /^(\$VAR1[^\;]*)$/ ) {
    # keep -T happy
    $status = $1;
} else {
    &htmllexit("<h2>ERROR - untaint of status
              message failed\n</h2>");
}
my $VAR1;
eval( $status );
if( $@ ) {
    &htmllexit("<h2>ERROR - eval of status message
              failed:$@\n</h2>");
}

return $VAR1;
}
```

After the reply is read and the socket closed, the string is untainted and then *eval*ed to recreate the configuration hash. Control then returns to the main block where the hash data is munged into an HTML table, with the read-only parameters as text strings and the modifiable parameters as HTML elements:

```
my $verbosehtml = checkbox(
    -name      => 'verbosefl',
    -label     => "",
    -checked   => $config->{'verbosefl'}
);

my $defpathhtml = textfield(
    -name      => 'defpath',
    -size      => 60,
    -maxlength => 132,
    -value     => $config->{'defpath'}
);

my $html = "";

$html .= start_form( -method => 'POST' );

$html .= qq{
<h2>Configuration as of $localtime: </h2>
<table cellpadding=2>
<tr><th align="left">Server ID String</th>
    <td>$config->{'serverstr'}</td>
</tr>
<tr><th align="left">Server Start Time</th>
    <td>$startstr</td>
</tr>
<tr><th align="left">Number of Messages</th>
    <td>$config->{'nummess'}</td>
```

```

</tr>
<tr><th align="left">Last Config</th>
  <td>$confstr</td>
</tr>
<tr><th align="left">Last Message</th>
  <td>$messstr</td>
</tr>
<tr><th align="left">Verbose Flag</th>
  <td>$verbosehtml</td>
</tr>
<tr><th align="left">Current default file
  path</th>
  <td>$defpathhtml</td>
</tr>
</table>
};

```

Submit and Reset buttons are created and added to the form, and the whole thing is sent to the user's browser to appear as it does in Figure 1.

After the user has modified the configuration parameters and submitted the form, the CGI client is executed again—this time calling *handlesubmit*():

```

sub handlesubmit {
    my $remotehost    = shift;
    my $port          = shift;

    #
    # build the config string from the HTML elements
    #
    my $configstr = "config ";

    foreach my $option ( qw( defpath ) ) {
        # would normally have several options
        $configstr .= "$option=$_ "
            if( $_ = param($option) );
    }

    foreach my $flag ( qw( verbosefl ) ) {
        # would normally have several flags
        $configstr .= param($flag)
            ? "$flag=1 " : "$flag=0 ";
    }

    #
    # set up the connection to the server
    # and send the config string
    #

```

```

my $socket = IO::Socket::INET->new(
    PeerAddr    => $remotehost,
    PeerPort    => $port,
    Proto       => "tcp",
    Type        => SOCK_STREAM )
or &htmxexit("<h2>Couldn't connect to
server : $@\n</h2>");
print $socket "$configstr\n";
close($socket);

&htmxexit("<h3>Submitted:</h3>
<h2>$configstr</h2>");
}

```

*It's so easy that you can even
allow knowledgeable end users to
make the changes in your absence*

This routine converts the form element information into a string of *key=value* pairs, with the *config* command prepended. The CGI client once again connects to the time server, sending it the configuration change request. Finally, the configuration string is displayed in the user's browser as a confirmation message.

The Real World

In the real world, things are quite a bit more complex, of course. All of the socket I/O must have appropriate timeout handling, and the server's responses to configuration messages should reflect their validity. The biggest complication for this technique in the real world, however, is security. The messages need much more thorough taint checking and the server should only allow clients from trusted IP addresses to connect. (In my case, I have a small range of addresses belonging to a private network protected by a local firewall for the wind tunnel control room, all hidden behind a corporate firewall.) You cannot rely solely on the security built into Apache to protect your system, since anyone can bypass it completely by creating a simple program like the test client.

To improve both security and robustness, I usually restrict the values of the most sensitive configuration parameters by setting up hashes whose keys are the acceptable values. That way I can easily check the incoming values like this:

```

$pathgoodvalue = ( '/tmp'    => 1,
                   '/usr/tmp' => 1 );

#
# put code here to read the config request string
# and isolate the value
#

if exists $pathgoodvalue{$requestvalue} {

```

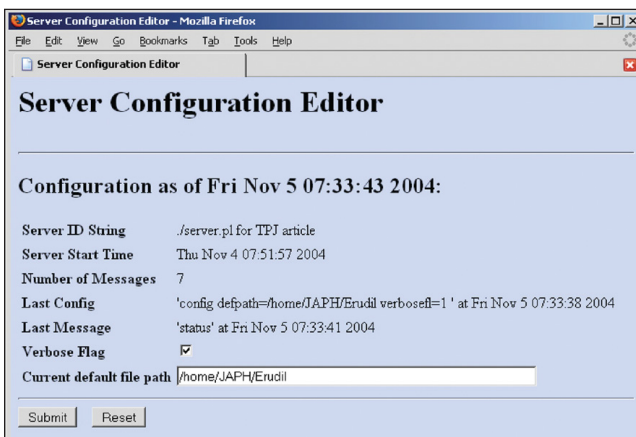


Figure 1: CGI client web page.

If the hashes containing the acceptable values are placed in a module, they can be included in the CGI program where their keys can easily be used in the creation of HTML select elements.

Although the *eval* of the *Data::Dumper* output in the web client may set off warning bells in the mind of some CGI programmers, this is one part of the process that is actually not as dangerous as it first appears. For this to be a security risk, someone would have had to compromise the network behind the firewall, steal the server computer's IP address, and then mimic the server program's behavior. If that has happened, there are much bigger things to worry about than a corrupted CGI form. (If you're really uncomfortable with the *eval*, you could use either *Storeable* or *YAML* to serialize and reconstruct the hash, since neither of these methods requires runtime code evaluation.)

For the four servers that I've written most recently, I created separate CGI programs for status reporting and configuration modification. This way, I can use standard Apache security techniques to allow everyone to view the status information, but only allow select people in our IT group to modify the configuration. See Figure 2 for an example of the web pages for one of my real-world servers.

The corresponding master configuration hash for this server is:

```
my $config = { 'serverstr' => "$0 using Perl $)",
               'starttime' => time,
               'numevents' => 0,
               'quietfl'   => $opt_q,
               'verbosefl' => $opt_v,
               'dddofff'   => $opt_x,
               'sleeptime' => 1,
               'maxsleeps' => 60,
               'maxevents' => 100,
               'fatalmail' => $mailaddrs[0],
               'history'   => [],
             };
```

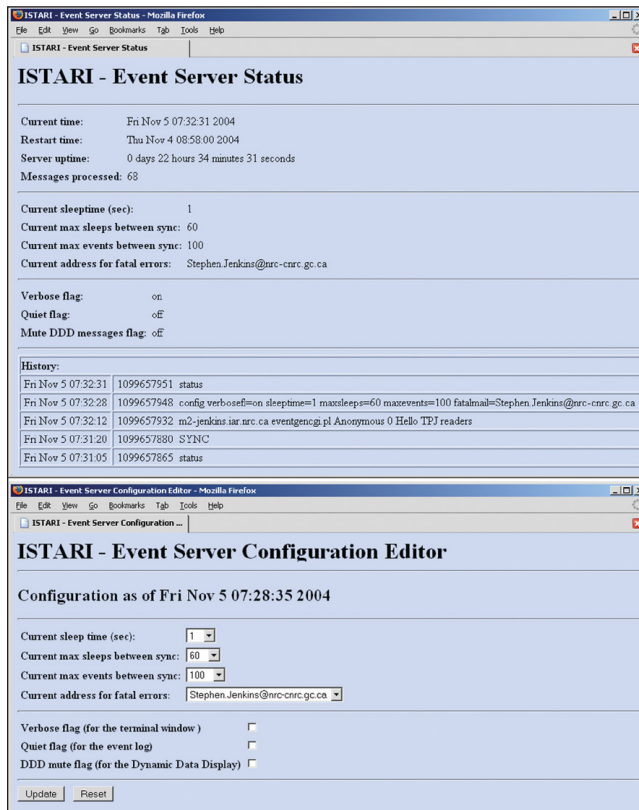


Figure 2: Real-world status and configuration web pages.

A few of my servers have one additional complexity. They are daemons that fork off child processes that may stay alive for days or even weeks. In order to propagate the configuration changes to them, the serialized master hash must be sent from the parent process to each of the children via a pipe. The child processes can then update their own copies of the configuration data.

Conclusion

Using this method to make your servers dynamically configurable is simple to implement, and by allowing changes to be made easily and quickly, it exhibits two of the three great virtues: laziness and impatience. In fact, it's so easy that you can even allow knowledgeable end users to make the changes in your absence, reducing telephone or pager interruptions during your time off. The only drawback I have found to this technique is the possibility of security problems; you will have to examine your own environment to decide if the benefits outweigh the potential risks.

Acknowledgment

I would like to thank Paul Fenwick of Perl Training Australia for his assistance preparing this article.

Reference

Jenkins, S.B. "A Web-Based Environment to Support Aerodynamic Testing," *IEEE Aerospace and Electronic Systems Magazine*, January 2004, pg. 3.

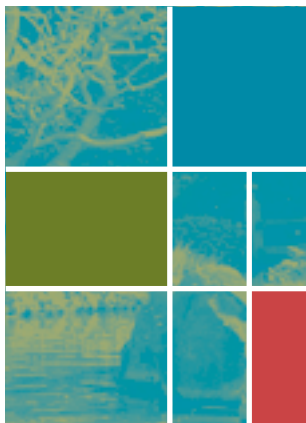
TPJ

Fame & Fortune Await You!

Become a **TPJ** author!

The *Perl Journal* is on the hunt for articles about interesting and unique applications of Perl (and other lightweight languages), updates on the Perl community, book reviews, programming tips, and more.

If you'd like share your Perl coding tips and techniques with your fellow programmers – *not to mention becoming rich and famous in the process* – contact Kevin Carlson at kcarlson@tpj.com.



Keeping Up With the World

Simon Cozens

I love the holiday period—a chance to go back and see family, and to get away from it all in an isolated cottage in rural Wales. Of course, there's a downside to this: being in an isolated cottage in rural Wales. I still like to keep up with what's going on in the outside world, with the latest tech news and what people are saying on their blogs. Unfortunately, there's only dial-up available and that is time-metered, so I'm very much on a bandwidth budget.

Normally, I'd use an RSS reader. We've met RSS before—it's a way for sites to publish a machine-readable file of their latest news and items, so we can suck down these XML files and see if there's anything we haven't seen yet. Unfortunately, these XML files can get quite big, averaging around 10K, and if you're following a lot of them, that soon adds up. One possible solution is to run a console-based RSS reader on my server, and use a text-mode interface to that: This drastically cuts down the amount of data I need to download.

One last stipulation—when I get back from my holiday and back to bandwidth and my desktop, I don't want to lose track of what I've already read; I want to be able to carry on where I left off. At college, I use NetNewsWire, a Mac RSS client, so I want my text-based reader to read the NNW history format as well. Naturally, there aren't many of these around; several come close, but nothing was quite right. So I did the obvious thing: I wrote one. It's called “press,” from “Perl RSS.”

We'll look at how I did it this month and next: This month, we'll concentrate on getting a basic RSS aggregator working, and next month, we'll see how to interface that to NetNewsWire.

Scanning CPAN

The first port of call for any application like this is CPAN. What modules can we find that will do the work for us? I already knew about *POE::Component::RSSAggregator*, which does most of the job: It polls news sites, downloads their RSS, and alerts the POE event loop if there are any new headlines. Next, we need a UI to display it all: Curses is the right idea, but it's a bit low level, so I

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 pumping. Simon can be reached at simon-cozens.org.

decided to check out *Curses::UI*. We'll need to format the stories contained in the RSS files, which are usually in HTML, for display as plain text, so we pick up *HTML::FormatText* as well. Finally, for dealing with the Macintosh property list files used by NetNewsWire, we can grab the aptly named *Mac::PropertyList*.

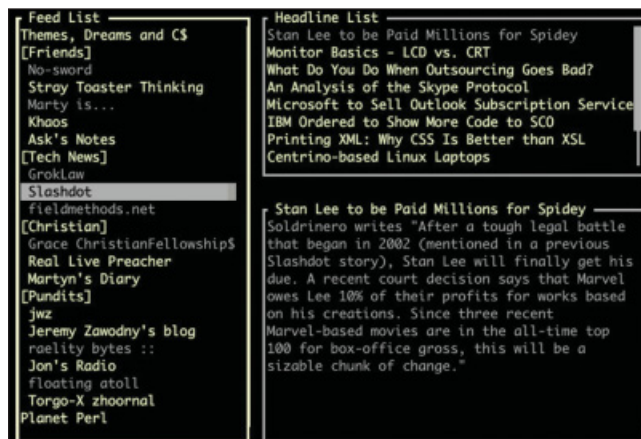
If it sounds like we've got most of the application done already, we have—the finished product was just over 200 lines of code.

Hacking the UI

Next, I got myself familiar with *Curses::UI* by creating a mock-up of the application. *Curses::UI* allows you to create “widgets,” such as windows, list boxes, text areas, and so on, and stick them all together in a nice object-oriented way. So for instance, I wanted the display to have three panels, just like NetNewsWire (and many other readers), one for the list of the feeds I'm following, one for the headlines in an individual feed, and one for the story behind the headline. Figure 1 is a picture of what we're aiming toward.

I started by creating the three windows and adding them to the main *Curses::UI* object:

```
my $cui = new Curses::UI(-color_support => 1);
my $win1 = $cui->add( "feedlist", "Window",
    -titlereverse => 0, -htmltext => 1,
```



```

-border => 1, -bfg=>"blue",
-title => "Feed List", -width => 30);

my $win2 = $cui->add( "headlist", "Window",
  -titlereverse => 0, -htmltext => 1,
  -border => 1, -bfg=>"blue",
  -title => "Headline List",
  -x => ($win->width+1), -height => 10);

my $win3 = $cui->add( "datawin", "Window",
  -titlereverse => 0, -htmltext => 1,
  -border => 1, -bfg=>"blue",
  -x => ($win->width+1),
  -y => ($win2->height+1) );

```

Each call to *cui->add* is followed by a name for the widget so that we can get at it later, the name of the widget (here we're starting with Windows), and then some options—we don't want the title to be reversed, we want to be able to use simple markup in titles, we want a border, and so on. We declare the sizes and positions of the latter two windows relative to the first, so that if we change the width of that, everything else will still be in the right place. Incidentally, the first line of options is always the same, so we factor that out to avoid repeating code:

```

@mywindow = ( "Window",
  -titlereverse => 0, -htmltext => 1,
  -border => 1, -bfg=>"blue");
my $win1 = $cui->add( "feedlist", @mywindow, ...);
my $win2 = $cui->add( "headlist", @mywindow, ...);

```

Now we can place some widgets inside the windows—a “Listbox” widget in each of the feed and headline lists, and a “TextViewer” in the data window:

```

my $feedbox = $win1->add('feedlistbox', "Listbox",
  -vscrollbar => 1);
my $headbox = $win2->add('headlistbox', "Listbox",
  -vscrollbar => 1);
my $viewer = $win3->add('data', "TextViewer");

```

And now we can tell the main Curses event loop to start:

```
$cui->mainloop;
```

When we run this, we get presented with a nice three-panel interface, two empty lists, and...no way to quit. Oops. We'd better add some simple navigation. First, we tell *Curses::UI* that the “^C” and “q” keys can be used to quit:

```
$cui->set_binding(sub { exit }, $_[0]) for "\cC", "q";
```

Next, we'll tell each window that the tab key can be used to move to the next window, just as we'd expect. We do this by moving the focus to the next window in sequence:

```

$win1->set_binding(sub { $win2->focus }, "\cI");
$win2->set_binding(sub { $win3->focus }, "\cI");
$win3->set_binding(sub { $win1->focus }, "\cI");

```

Now our interface looks a bit better, and we can start thinking about how we're going to implement the logic.

What's POE?

This application is going to have multiple sources of input, and they could happen at any time: The user could hit a key on the keyboard and we'd have to update the screen, or an updated RSS

feed could come in and we'd possibly have to deal with adding new headlines to the display. We also want to fire off HTTP requests occasionally to ask for new news. All of these things are called “events,” and we're now entering the world of event-driven programming.

In an ordinary, procedural program, we'd have to deal with the user's keystrokes, then send off the HTTP request and block there until we got a response, and the user couldn't use the application's UI until we cede control back to her and wait for another keystroke. This leads to a horrible user experience. In the event world, we have a main event loop that watches for things happening. We tell the event loop that we want to schedule web requests periodically. When something happens, such as a keystroke or a response from the HTTP request, the event loop calls a routine. In this sense, the event loop is a bit like an operating system—it looks after scheduling events and dispatching them to the appropriate bit of code that's currently listening for them.

POE is one such event loop, and an award-winning one at that. Part of the beauty of POE is that major chunks of event-generating and event-responding code are packaged up as “components,” like our *POE::Component::RSSAggregator*. There are also POE components that act as HTTP servers, IRC clients, watch for changes to files, or even control MP3 players. POE can look fiendish if you've never seen it before, but we'll explain it as we go along.

Subclassing CPAN

While I'm a big fan of doing as much as possible with CPAN modules, I'm not going to pretend that CPAN modules are always a perfect fit for the job in hand. However, they're usually pretty good, and if they're not a perfect fit, you can usually get some use out of them by subclassing and bending them to your will.

So with the current program, there are two slight mismatches that we need to fix. First, *POE::Component::RSSAggregator* uses *XML::RSS::Feed* objects, which are great, but unfortunately they assume that every time a new article appears, the old articles aren't new any more; this is perfectly good behavior if you're writing a news ticker where you only want to display each new item once. However, we're writing a news reader, and we want articles to stay new until the user has read them. So we get subclassing!

Looking at *XML::RSS::Feed*, we find that every time an XML file is parsed, it calls *_mark_all_headlines_seen*, and this method puts the ID of the headline object in the *rss_headline_ids* hash. So we create a new class that doesn't *_mark_all_headlines_seen* automatically, but does allow us to specify manually when a headline has been read:

```

package XML::RSS::Feed::Manual;
use base 'XML::RSS::Feed';
sub _mark_all_headlines_seen {}
sub mark_read {
  my ($self, $head) = @_;
  $self->{rss_headline_ids}{$head->id} = 1;
}

```

The second mismatch is that we want to use POE as our event loop so that *POE::Component::RSSAggregator* can post events about new articles. Unfortunately, we're also using *Curses::UI*, which has its own event loop. There's a Curses loop for POE, *POE::Wheel::Curses*, but we still need to glue it all together. So we look at *Curses::UI's* *mainloop* and see how it works:

```

sub mainloop ()
{
  my $this = shift;

```

```
# Draw the initial screen.
$this->focus(undef, 1); # 1 = forced focus
$this->draw;
doupdate();

# Infinite event loop.
for(;;)
{
    $this->do_one_event
}
}
```

We also find that *do_one_event* reads a key from the keyboard, unless *\$this->{-feedkey}* is set to a pending keystroke. *POE::Wheel::Curses* also reads a key from the keyboard, so we can use that as our main loop, then have it feed the key that it has just read into *\$cui->{-feedkey}* and call *do_one_event*: This will cause *Curses::UI* to dispatch the key to the appropriate widget and do the right thing. So we convert our dummy UI to use POE:

```
POE::Session->create(inline_states => {
    _start => sub {
        my ($heap) = $_[HEAP];
        $heap->{console} =
            POE::Wheel::Curses->new(
                InputEvent => "got_keystroke"
            );
        $cui->focus(undef, 1);
        $cui->draw;
        Curses::doupdate();
    },
    got_keystroke => sub {
        $cui->{-feedkey} = $_[ARG0];
        $cui->do_one_event;
    }
});
POE::Kernel->run;
```

If POE is like an operating system, then *POE::Session* objects are its processes. An operating system with no processes is boring, so we create a new one. This session will respond to two events: the *_start* event is called when the session begins, and the *got_keystroke* will be called every time *POE::Wheel::Curses* sees a keystroke. Inside the *start* event, we set up *POE::Wheel::Curses*, and put its data on the “heap”—this is just a storage area that the POE kernel sets up for us, and means that the Curses handler is going to stick around for the whole of the application. We also tell the Curses wheel what event to fire when a key is pressed, and then we copy in the initialization code from *Curses::UI*'s *mainloop*.

When a key is pressed, we feed the key into the *Curses::UI* object and run one event. Once we're all set up, we tell the POE kernel to run, and now we can test that our application correctly dispatches keystrokes from POE to *Curses::UI*.

Adding the News

So far so good, but now we need to think about the RSS handling part. For this month, we'll assume that we have a static list of feeds, like so:

```
my %feeds = (
    "http://planet.perl.org/rss10.xml" =>
        "Planet Perl",
    "http://slashdot.org/slashdot.rss" =>
        "Slashdot",
    "http://interglacial.com/~sburke/torgo_x_upo.rss"
```

```
    => "Torgo-X zhoornal",
    # ...
);
```

Now we'll create our *XML::RSS::Feed::Manual* objects:

```
my @values;
my %labels;
while (my ($rss, $name) = each %feeds) {
    my $feed = XML::RSS::Feed::Manual->new(
        rss => $rss, name => $name
    );
    push @values, $feed;
    $labels{$feed} = $name;
}
```

We now have a list of feed objects and a hash that turns the feed object into a name. We can use these as the values and labels of our list box:

```
$feedbox->values(\@values);
$feedbox->labels(\%labels);
```

This means that the feed list will be full of items labeled according to the name of the feed: We will see a list “Planet Perl,” “Slashdot,” and so on, as we might expect. However, when one of these items is selected, we can call a method on the list box and get back the underlying *XML::RSS::Feed::Manual* object for that item.

For instance, as we select each feed in the feed list, we want the list of headlines to change. To do this, we add an *onselchange* handler to the feed list:

```
my $feedbox = $win->add('feedlistbox',
    "Listbox", -vscrollbar => 1,
    -onselchange => \&select_feed );
```

Now as we cursor up and down in that box, *select_feed* is called:

```
sub select_feed {
    my $feed = $feedbox->get_active_value;
    my @headlines = $feed->headlines;
    $headbox->values(\@headlines);
    $headbox->labels({
        map { $_ => $_->headline } @headlines
    });
    $headbox->layout_content->draw(1);
}
```

As mentioned, we can ask the feed box for the currently selected feed object. As this changes, we ask the feed for its headline objects, which are in the *XML::RSS::Headline* class. In the same way as with the feed box, we fill the headline box with these objects, and then give each object a label that is the human-readable headline. Since we have changed the data in the headline box, we need to force it to redraw, which we do with an incantation stolen from the *Class::UI::Dialog::Filebrowser*.

Next, as we select a headline, we want to display the story in the data window. So, in exactly the same way, we attach a handler to the headlines box:

```
my $headbox = $win2->add('headlistbox', "Listbox",
    -vscrollbar => 1,
    -onselchange => \&select_story );
```

And this does much the same thing, looking at the headline object this time instead of the feed object:


```

sub select_story {
    my $head = shift->get_active_value;
    if ($head) {
        $viewer->text(
            HTML::FormatText->format_string(
                $head->description,
                lm=>0, rm => $viewer->width -2
            )
        );
        my $feed = $feedbox->get_active_value;
        if (!$feed->seen_headline($head->id)) {
            $feed->mark_seen($head);
        }
    } else { $viewer->text("Nothing selected"); }
    $viewer->layout_content->draw(1);
}

```

We take the description from the headline object and format that up as plain text, putting the result in the viewer. Now we need to tell the feed that this item has been read—this currently doesn't make any difference, since we don't distinguish between read and unread items in the aggregator yet. But we will.

We're very nearly there. The only thing we don't have is data. But once we plug in *POE::Component::RSSAggregator*, we'll get that automatically. So in our *session _start* event, we add the following code:

```

$heap->{rssagg} =
    POE::Component::RSSAggregator->new(
        alias => "rssagg"
    );
$kernel->post('rssagg', add_feed => $_)
    for @values;

```

This is all we need to do. Having told the aggregator about the feed objects that we want to watch, it will arrange with the POE scheduler to update them for us. When the program starts, the aggregator component will fire off a set of HTTP requests; when it sees a response to one of them, it will create the appropriate headline objects in the feed. The next time the user selects a new feed, the new headlines will already be there. Then the aggregator schedules another bunch of requests, 10 minutes later. We don't need to do any work.

Finishing Touches

The whole point of this exercise was for me to be able to quickly and easily check RSS news with a minimum of bandwidth. Unfortunately, at present, I have to check every single feed since there's no visual indication of which feeds have new items, or even which items are new and which have been seen before.

Here's one subroutine that will help:

```

sub bolden_news {
    my $labels = $feedbox->labels;
    for my $elem (@values) {
        if ($elem->late_breaking_news and $labels
            ->{$elem} !~ /<bold>/) {
            $labels->{$elem} = "<bold>".$labels
                ->{$elem}."</bold>"
        } elsif (!$elem->late_breaking_news) {
            $labels->{$elem} =~ s{</?bold>}{}g;
        }
    }
    $feedbox->labels($labels);
    $feedbox->layout_content->draw(1);
}

```

The key here is the *late_breaking_news* method on a feed object. It returns True if the feed contains any articles we haven't yet marked. We look at each of the feeds in the feed box; if there are some new articles, and the label doesn't have bold tags around it (this is the simple markup we were talking about when we created the windows), then we add some tags. If we've read everything in this feed, then we take the bold tags out again.

We can do the same trick for headlines in the headlines list:

```

sub select_feed {
    my $feed = $feedbox->get_active_value;
    my @headlines = $feed->headlines;
    $headbox->values(\@headlines);
    $headbox->labels({ map {
        my $head = $_->headline;
        $_ => (!$feed->seen_headline($_->id)
            ? "<bold>$head</bold>" : $head
        ) $feed->headlines });
    $headbox->layout_content->draw(1);
}

```

If we haven't seen this headline before, we "bolden" it. Now we just need to arrange for *bolden_news* and *select_feed* to be called when the *POE::Component::RSSAggregator* sees new news:

```

my ($kernel, $heap, $session) = @_[KERNEL, HEAP, SESSION];
...
$heap->{rssagg} = POE::Component::RSSAggregator->new(
    alias    => 'rssagg',
    callback => $session->postback("handle_feed"),
);

sub handle_feed { bolden_news(); select_feed() }

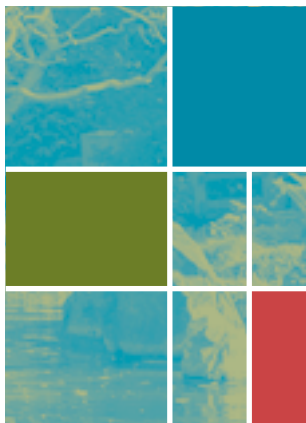
```

When a new news item comes in, the aggregator asks the session to call the *handle_feed* subroutine; in this, *bolden_news* will highlight the feed that the new item is in, and *select_feed* will highlight the unread items.

Now we have a useful news aggregator that will keep track of new news articles for us while requiring minimal bandwidth. Next month, we'll see how to pull the feeds out of the NetNewsWire preferences and how to read and store history so that the information about which articles we've read becomes persistent.

TPJ





Geolocation in Perl

brian d foy

Many web server log analyzers now support “geolocation,” meaning that they can turn a host name or IP address into a point on the globe. With geolocation, instead of looking at a bunch of numbers, I can look at maps. Using the *Geo::IP* module and the databases from MaxMind (<http://www.maxmind.com/>), both of which are freely available, I can add this feature to my programs, too.

Turning IP addresses into locations is not perfect, but as long as I understand how IP numbers are assigned and split up, I can put my geolocation results into perspective. I need to know a little about how IP numbers are assigned, so I can interpret the results and judge the accuracy.

Starting from the top, the Internet Corporation for Assigned Names and Numbers, better known as ICANN (<http://www.icann.org/>), is responsible for IP address allocation. It gives out large chunks for addresses to Regional Internet Registries (RIR) that cover certain parts of the globe. So far, these are the American Registry of Internet Addresses (ARIN), the Asia Pacific Network Information Centre (APNIC), Latin American and Caribbean Internet Addresses Registry (LACNIC), and RIPE Network Coordination Centre (RIPE). Another registry, African Network Information Center (AfriNIC), is on the way, too.

Each of the RIRs hand out chunks of their own IP space to major organizations, which then do the same to smaller organizations, and so on until one of those IP numbers gets to my cable modem in my apartment.

Knowing that, to figure out where any IP address is on the globe means I just have to track it through that process until I get the resolution I want. I may want to stop at the country level, or I might want to go even further. For this article, I’m going to stop at the country level.

There is a problem, though: Organizations don’t have to respect these lines that we’ve overlaid on the globe. For example,

brian has been a Perl user since 1994. He is founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and has been a featured speaker at The Perl Conference, Perl University, YAPC, COMDEX, and Builder.com. Contact brian at comdog@panix.com.

America Online (AOL) has a very large IP allocation, but it’s a multinational company. It can use its allocation as it likes. Indeed, that’s one of the major differences in the free and paid versions of the MaxMind GeoIP database: The paid version can figure out which AOL addresses are not in the United States. In the free version of the database, all AOL addresses show up as the United States.

Now that I know why my results won’t be perfect, I can move on to the task. Before I install the *Geo::IP* module, I need to get a database for it. The free version of the MaxMind database identifies the country of the IP address with 93 percent accuracy. Various levels of the paid version can get me to 98 percent country accuracy for \$50, or for \$370, down to the city with longitude and latitude. Considering how much work I would have to do to compile all this myself, and how much the local burger joint pays me to make fries, I think these prices are a bargain.

I’m going to stick with the free version of the database for this article, though. I get the free database from the MaxMind developer section (http://www.maxmind.com/app/geoip_country). Once I unzip the archive, I end up with a single file, *GeoIP.dat*, which I move to its default location, */usr/local/share/GeoIP/GeoIP.dat*.

Once I have the database in place, I can install the *Geo::IP* module either by installing it with CPAN.pm or downloading it and installing it by hand. It’s at <http://search.cpan.org/dist/Geo-IP>. I give it a whirl with a simple script using my own IP address (at least, the one I have today):

```
% perl -MGeo::IP -le \  
'print Geo::IP->new->country_name_by_addr(  
    "24.12.70.45" )'  
  
United States
```

Now I want to apply this to a bunch of IP addresses, and web-server access logs are a great source of those. I’m going to use a web access log in the Apache format, although that doesn’t really matter—I just need the IP address. Each line shows the start of a line from one of my logs. It starts with an IP number, followed by some whitespace, then some other fields that I don’t need for this task. In the real world, I’ll probably have something else that completely parses the log.

```
#!/usr/bin/perl

while( <> )
{
    my( $ip ) = split; # just the first field

    $Seen{ $ip }++;
}

foreach my $ip ( sort keys %Seen )
{
    printf "%6d %s\n", $Seen{ $ip }, $ip;
}
```

I get as output a long list of IP numbers and the count of how many times my web server responded to a request from that address. Those aren't the numbers of unique visitors or any other massaged number, they are just the number of requests my web server had to respond to from that IP address:

```
4 128.112.148.25
5 130.160.141.180
4 168.122.194.108
6 194.7.234.34
2 203.144.187.62
```

That's not all that interesting, though. I don't really care about IP numbers, and I want to see which countries people are in. I'll never be able to really figure out which country the pair of eyes is in because nothing stops me from logging into a computer halfway around the world and browsing the Web from there. Knowing that, I'm willing to accept the results.

```
#!/usr/bin/perl

use Geo::IP;

my $gip = Geo::IP->new();

while( <> )
{
    my( $ip ) = split;

    my $country = $gip->country_name_by_addr( $ip );

    $Countries{ $country }++;
}

foreach my $ip ( sort keys %Countries )
{
    printf "%6d %s\n", $Countries{ $ip }, $ip;
}
```

I get a columnar listing of the number of times an IP address from a country hit my web server:

```
346 Belgium
51 Canada
22 China
446 France
302 Germany
21 Taiwan
157 Thailand
1233 United States
```

That's still not good enough for me. Why should I settle for text when I can see a picture? Douwe Osinga provides a little ser-



Figure 1: Visiting countries displayed in red.

vice that can make a map that colors each country that I've visited (<http://douweosinga.com/projects/visitedcountries/>). I can just as easily turn that into a colored map of each country that has visited me. He uses a long URL with a query string that has a two-digit country code.

The *Geo::IP* module can handle this, too. I need the country code instead of the country name, so I use the *Geo::IP's* *country_code_by_addr()* method instead of *country_name_by_addr()*. Once I have all the country codes, I don't print them in a report—I join them without any characters in between them and use that as the URL query string. Because I want to redirect to a web page, I have the script output a CGI header that redirects to the external URL:

```
#!/usr/bin/perl

use Geo::IP;

my $gip = Geo::IP->new();

while( <> )
{
    my( $ip ) = split;

    my $country = $gip->country_code_by_addr( $ip );

    $Countries{ $country }++;
}

my $query_string = join "", keys %Countries;

my $url =
"http://douweosinga.com/projects/
visitedcountries/colormap" .
"?visited=" . $query_string;

print "Status: 302 Moved
Temporarily\nLocation: $url\n\n";
```

The URL returns a GIF image that is a map of the world with the visiting countries colored red (see Figure 1).

I started with IP addresses and ended up with a map of the world, using only a freely available Perl module, a free geolocation database, and a free mapping service. I could get much more fancy than this, too. With finer-resolution databases, I can get down to the city level and combine that with longitude and latitude data to make dots on a map. I don't have to stick to web access logs either: I could use geolocation to identify users as they come into my web site, or any other service I might provide. However I decide to use it, *Geo::IP* makes it easy.



Beginning Perl

Jack J. Woehr

Beginning Perl, Second Edition, is a Perl 5.6 self-tutorial introductory book gone Perl 5.8. It's a great book for computer-science students and programmers who want to jump into Perl at the intermediate level. Programming beginners may well be baffled: This is not an introduction to computer science itself.

Executive summary: If you're a working coder and don't know Perl, or perhaps want a refresher course (Perl comes upon one in fits, like Lewis Carroll's passion for chess), *Beginning Perl* is one of the more business-like and thorough-going expositions on the bookshelf, as well as being neatly written and eminently readable.

Breadth of coverage is good in *Beginning Perl*, running right up through object-oriented Perl and modules. Necessarily concise, the coverage is anything but superficial; rather, it is most nerdy and rich in insights, alternatives, and tips. The final two chapters, by a mysterious, unwritten, but apparently universal law of computer technical publishing, are dedicated to "on beyond," in this instance CGI and DBI.

Beginning Perl is a quality production in both accuracy and layout. The table of contents, source code for the examples, and forums are found online at <http://www.apress.com/book/bookDisplay.html?bID=344>. Amusingly, the book does not contain this URL, instead referring the reader seeking source code to Apress's home page, which features a search engine incapable of finding the term "Beginning Perl" and that had to be fed the ISBN number to find the book.

The second edition was prepared by James Lee, the competent clean-up batter who brought in the 5.8 coverage. However, experienced programmers may themselves be baffled by the earnest but mumbled and equivocal coverage of Perl 6. Experts among us can sometimes compile and even occasionally run Perl 6 at this writing in early 2005. I've done it myself, submitting patches along the way, and I'm still embarrassed by how much effort it took.

Simon Cozens, fellow *TPJ* writer, departing master of perl.org, and author of *Advanced Perl Programming* (soon out in second

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

Beginning Perl, Second Edition

*James Lee, Simon Cozens,
and Peter Wainwright*

Apress, 2004
600 pp., \$39.99
ISBN 159059391X

edition) and something like 70 CPAN modules, was the original author of *Beginning Perl*. Let's see what he had to say about the new edition of his book:

"I was asked if I would like to write *Beginning Perl 2*. I said yes. I was then told that the publishers 'would prefer to find an author who can carry revisions of the book from 5.6 to the present and beyond for Perl 6.' I said that writing a Perl 6 book at this point was insane, but I took the hint that they were going to be using my first edition but getting another author to update it."

I chatted recently with Simon, who is retiring from Perl (but not from *TPJ*) and heading off to be an overseas Christian missionary.

TPJ: You seem to sort of validate my thesis that a good liberal arts education is a sound basis for technical pursuits.

Simon Cozens: Hmm, interesting. I think it certainly helps for communicating technical ideas.

TPJ: Are you a C.S. Lewis fan?

SC: I am, yes. I have all his works behind me at the moment.

TPJ: The “hard” stuff too? For example, *The Great Divorce*, *The Problem of Pain*, and so forth, one assumes.

SC: Yep.

TPJ: Lewis definitely was an influence for me. Do you suppose that somehow that sort of influence affects programming style?

SC: Hmm, tricky one. It’s certainly not conscious. Maybe there’s something about the Oxford tutorial style that both he and I are a product of that leads to particular clarity of logical expression. But I suspect that’s overanalyzing; I originally learned to program from my father, who’s as much of an engineer as they come.

TPJ: So your Japanese is pretty good?

SC: Yep. That was my first degree. *Beginning Perl* was written while I was living in Japan.

TPJ: How about *Advanced Perl Programming*?

SC: *Advanced Perl Programming* is a complete reworking of the O’Reilly book. It’s more of a rewrite than a second edition; I’ve kept no material [from the original].

TPJ: Are you cutting that loose for your mission or are you going to keep in touch with the publisher via e-mail?

SC: Oh, I’ll still be in touch with the editor. We’ve put the whole thing in Subversion and we’re editing it together. But yeah, I do want to clear the decks as soon as I can.

TPJ: So you can become a moth in the flame of metaphysical speculation?

SC: Well, so I can preach to people about something other than Perl for a while.

TPJ: Is this a hiatus or do you really feel you’ll not work in coding anymore?

SC: I really am going out to full-time missionary work, so in that sense, I’m never going to be a professional programmer again. I suspect I will end up doing the occasional bits of coding to make things easier for myself or to bring in additional income if support gets tight, but I’m very much permanently phasing out of the open-source community.

TPJ: You’re positively Byronic, you know...

SC: It’s liberating! And, Larry Wall was going to do it, but didn’t make it. So that gets me one up on him.

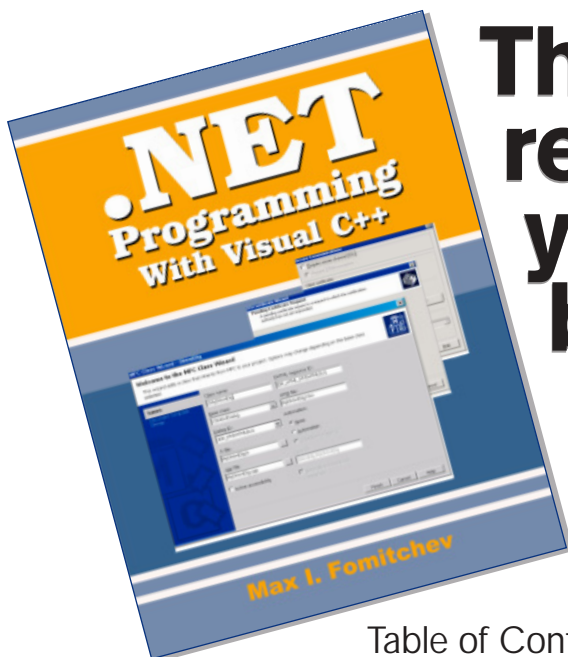
TPJ: Any last thoughts to hurl over the wall into the Perl community before you fold your tent and vanish into the desert?

SC: I think one failure of mind that people involved in programming can have is to think that the world revolves around their sphere of operation; that, say, everyone in the world ought to care about software patents or programming freedom or good style, or that everyone in programming should be using Perl, or whatever. Programming in Perl becomes an end in itself. What I’m doing is opening the door to the rest of the world out there; if you program in Perl, do it to make that world out there a better place. Perl can be a good tool to sculpt the world with, but don’t confuse Perl and the world.

TPJ: Well have a wonderful life, be a blessing to all you meet.

SC: Thanks, I’m sure it’s going to be an experience.

TPJ



The .NET resource you’ve been waiting for!



- Delivered in PDF format.
- Packed with C++ code examples.
- Thousands of lines of source code.
- A complete reference to the .NET Framework

Table of Contents and sample chapter available at:
<http://www.ddj.com/dotnetbook/>

Get your copy now!

Available via **download** for just **\$19.95**
or
on **CD-ROM** for only **\$24.95** (plus s/h).

Source Code Appendix

Stevan Little “Inversion of Control in Perl”

Listing 1

```
my $app_c = IOC::Container->new('app');

my $db_c = IOC::Container->new('database');
$db_c->register(IOC::Service->new('dsn' => sub { 'dbi:mysql:test' }));
$db_c->register(IOC::Service->new('username' => sub { 'user' }));
$db_c->register(IOC::Service->new('password' => sub { '*****' }));
$db_c->register(IOC::Service->new('connection' => sub {
    my $c = shift;
    return DBI->connect(
        $c->get('dsn'),
        $c->get('username'),
        $c->get('password'));
}));

$app_c->addSubContainer($db_c);

my $log_c = IOC::Container->new('logging');
$log_c->register(IOC::Service->new('log_file', sub {
    '/var/log/app.log'
}));
$log_c->register(IOC::Service->new('logger', sub {
    My::Logger->new((shift)->get('log_file'))
}));

$app_c->addSubContainer($log_c);

my $sec_c = IOC::Container->new('security');
$sec_c->register(IOC::Service->new('authenticator' => sub {
    my $c = shift;
    My::Authenticator->new(
        $c->find('../database/connection'),
        $c->find('../logging/logger')
    );
}));

$app_c->addSubContainer($sec_c);

$app_c->register(IOC::Service->new('app' => sub {
    my $c = shift;
    my $app = My::Application->new($c->find('/security/authenticator'));
    $app->setLogger($c->find('/logging/logger'));
    $app->setDatabaseConnection($c->find('/database/connection'));
    return $app;
}));
```

TPJ