If you can't read this, move closer!

BUILD EASILY EXTENSIBLE PERL PROGRAMS

(AKA PLUGINS GALORE!)

Ask Bjørn Hansen
Develooper LLC
Portland, OSCON 2005

Not Extensible

- I made an smtp server in Perl
- People loved hacking it
- How not to extend a program

```
if ($config->{something}) {
  $self->some_feature;
  return if $self->but_it_failed;
}
elsif (...) {
  ...
}
```

More than one way to do it

- Among others ...
  - Hooks
  - Delegation
  - Mix-ins
- All useful
  - Use, mix & match as appropriate

Hooks

- The plugin registers with the program
  - “I can read image/png files”
  - “Tell me when the user enters a URL”
  - “Call me when a new connection is opened”
- Fine control
- Clear API
- Can be a pain to put in hook calls everywhere it’s (possibly) needed

Delegation

- The program tells the plugin/extension/ module
  - “You go take care of this stuff”
  - “you take care of the database connection”
  - “you parse the XML”
- Not so good for small tweaks and add-ons
- Inheritance is / can be a variation of this

Mix-ins

- Add methods into existing base classes
- Good for adding *new* stuff to frameworks
- Not so good for changing how something works
- Usually not good for application plugins

Class::DBI::Plugin

- Class::DBI::Plugin will pull method_name into the Class::DBI namespace

```
Package Class::DBI::Plugin::MyPlugin;
use base 'Class::DBI::Plugin';

  sub init {
    my $class = shift;
    $class->columns( TEMP => ... );
  }

  sub method_name : Plugged {
    my $class = shift;
    ...
  }

  sub this_method_is_not_exported {}
```

CGI::Application plugins

- Just exports methods into your namespace

```
package CGI::Application::Plugin::Session;
require Exporter;

@EXPORT = qw(session);
sub import { goto &Exporter::import }

sub session {
    my $self = shift;
    ...
```

Multiple Inheritance

- No encapsulation, no privacy
- Hard to keep separation between your plugin and everything else
- Too easy to shoot yourself in the foot
- Don’t do it.
- Use delegation
- (Catalyst is using MI and they are happy)

qpsmtpd

- Started in 2001 for perl.org
- “The mod_perl of email”
- Super flexible smtp daemon
- Receive SMTP only
- Used by perl.org, lists.mysql.com, apache.org, several of the Really Big spamtraps

qpsmtpd v0.10 (late 2001)

- Make it **easy** to extend and change the program
- I knew mod_perl and the Apache API all too well
  - sharing some concepts from there
  - “Hooks” and return codes

- qpsmtpd is MIT/X licensed; so borrow away!

Dog food

- Move as much as possible into plugins
- Only keep the basic functionality in the application
- By lines of code qpsmtpd is 50% "core" and 50% plugins – and we rarely add to the core
  - Except for making it easier to create more powerful plugins

The first plugin

- Print a funny message when the client disconnects

```
sub quit_hook {
  my $qp = shift->qp;

  my $fortune = '/usr/games/fortune';
  return DECLINED unless -e $fortune;

  my @fortune = `$fortune -s`;
  @fortune = map { chop; s/^/  \/ /; $_ } @fortune;

  $qp->respond(221, $qp->config('me')
                    . " closing connection.", @fortune);

  return DONE;
}
```

quit plugin in action

- edit `config/plugins`
  add `plugins/quit_fortune`
- `telnet localhost 25`

```
...
> QUIT
< 221-mary.develooper.com closing connection.
< 221-  / Nobody said computers were
< 221   / going to be polite.
Connection closed by foreign host.
```

Plugin mechanisms

- Clear API
- Separation of concerns
- `if ($foo_on) { run_foo() }`
  sucks because it's too intermingled!
- Change the internals without changing the plugins
- Make a plugin without understanding all (or much!) of the internals
- Make it easy (and fun!) to write plugins

Another plugin example

- Make a note of the clients country for use by other plugins ... This is the full plugin! (apart from POD)

```
my $geoip = Geo::IP->new(GEOIP_STANDARD);

sub hook_connect {
  my ($self) = @_;

  my $country = $geoip->country_code_by_addr(
                   $self->qp->connection->remote_ip
                );

  $self->qp->connection->notes('geoip_country', $country);
  $self->log(LOGNOTICE, "GeoIP Country: $country");

  return DECLINED;
}
```

Load plugins – 1

- Convert plugin name “queue/postfix” to a package name (Qpsmtpd::Plugin::queue::postfix)

```
sub load_plugins {
  ...
  for my $plugin_line (@plugins) {
    my $plugin_name = $plugin;
    $plugin =~ s/:\d+$//;       # after this point, only used for filename

    # Escape everything into valid perl identifiers
    $plugin_name =~ s/([^A-Za-z0-9_\/])/sprintf("_%2x",unpack("C",$1))/eg;

    # second pass cares for slashes and words starting with a digit
    $plugin_name =~ s{
                      (/+)       # directory
                      (\d?)      # package's first character
                     }[
                       "::" . (length $2 ? sprintf("_%2x",unpack("C",$2)) : "")
                      ]egx;

    my $package = "Qpsmtpd::Plugin::$plugin_name";
```

Load plugins – 2

- Call the compilation method
- Create a new object of the plugin we just loaded and compiled!

```
    # don't reload plugins if they are already loaded
    unless ( defined &{"${package}::plugin_name"} ) {
      Qpsmtpd::Plugin->compile($plugin_name,
        $package, "$dir/$plugin", $self->{_test_mode});
    }

    my $plug = $package->new();
    push @ret, $plug;
    $plug->_register($self, @args);
  }

  return @ret;
}
```

Wrap and compile the plugin into its own package

- Read the plugin file ...

```
sub compile {
    my ($class, $plugin, $package, $file, $test_mode) = @_;

    my $sub;
    open F, $file or die "could not open $file: $!";
    {
      local $/ = undef;
      $sub = <F>;
    }
    close F;
```

Compile plugin 2

- Wrap the plugin into the package
- Compile it with eval
- Now it's Qpsmtpd::Plugin::geoip; inheriting from Qpsmtpd::Plugin!

```
my $line = "\n#line 0 $file\n";
my $eval = join(
                "\n",
                "package $package;",
                'use Qpsmtpd::Constants;',
                "require Qpsmtpd::Plugin;",
                'use vars qw(@ISA);',
                'use strict;',
                '@ISA = qw(Qpsmtpd::Plugin);',
                "sub plugin_name { qq[$plugin] }",
                "sub hook_name { return shift->{_hook}; }",
                $line,
                $sub,
                "\n", # last line comment without newline?
               );

eval $eval;
die "eval $@" if $@;
```

Wrapped Plugin

- Let's look at it without the noise

```
package Qpsmtpd::Plugin::geoip;
use Qpsmtpd::Constants;
require Qpsmtpd::Plugin;
use vars qw(@ISA);
use strict;
@ISA = qw(Qpsmtpd::Plugin);
sub plugin_name { qq[$plugin] }
sub hook_name { return shift->{_hook}; }
#line 0 plugins/geoip
... here goes the plugin code ...
```

register() vs Method names ($hook ."_hook")

- We used to have the plugin call register (a Qpsmtpd::Plugin method) to activate hooks for the plugin

```
sub register {
  shift->register_hook("data", 'method');
}
```

- Now you can just name the method appropriately and it'll be called automatically

register the named methods

```
sub _register_standard_hooks {
  my ($plugin, $qp) = @_;

  for my $hook (@hooks) {
    my $hooksub = "hook_$hook";
    $hooksub  =~ s/\W/_/g;
    $plugin->register_hook( $hook, $hooksub )
      if ($plugin->can($hooksub));
  }
}
```

_register_hook

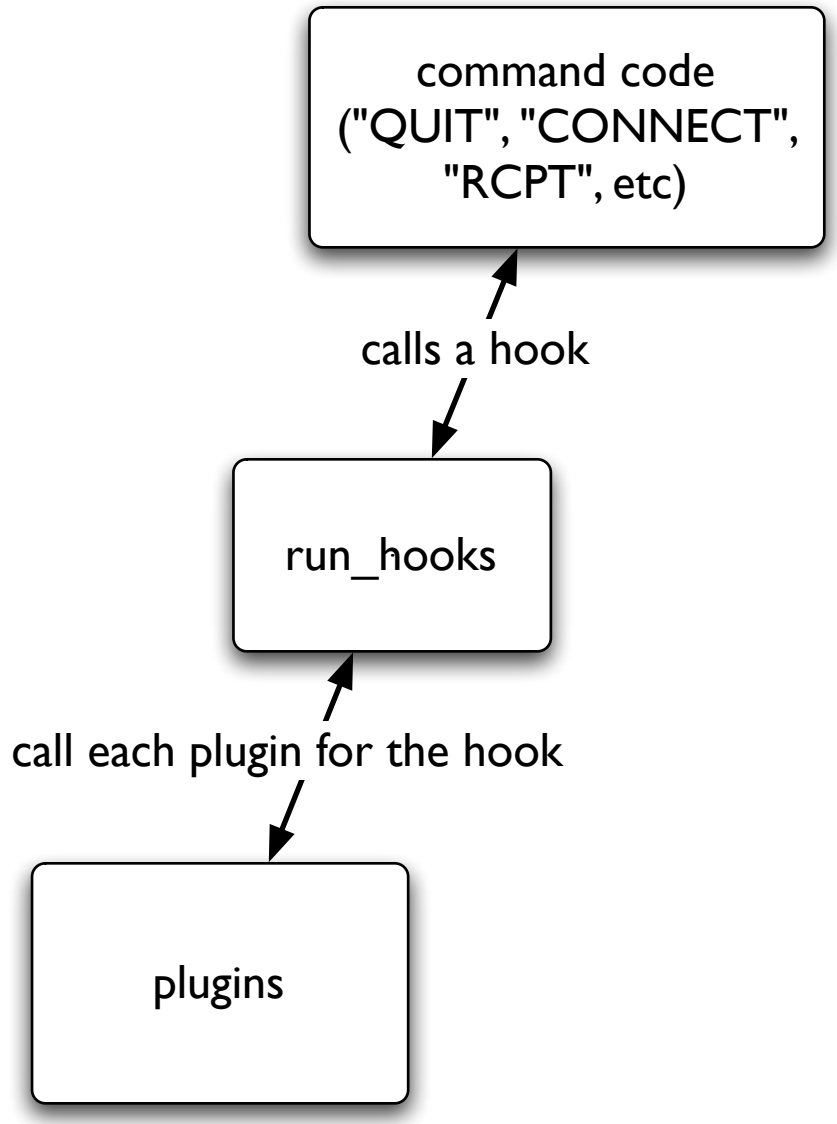
- In the Qpsmtpd module, called from Qpsmtpd::Plugin

```
sub _register_hook {
  my $self = shift;
  my ($hook, $code, $unshift) = @_;

  my $hooks = $self->{hooks};
  if ($unshift) {
    unshift @{$hooks->{$hook}}, $code;
  }
  else {
    push @{$hooks->{$hook}}, $code;
  }
}
```

How we run the plugins

- “command code” or event trigger calls a hook
- run_hooks runs the plugins
- each plugin return a status code



Call a plugin

- The quit command runs all configured “quit” hooks
- Return the default message unless a hook said it was “done”

```
sub quit {
  my $self = shift;
  my ($rc, $msg) = $self->run_hooks("quit");
  if ($rc != DONE) {
    $self->respond(221, $self->config('me')
    . " closing connection. Have a wonderful day.");
  }
  $self->disconnect();
}
```

Complex Command

RCPT TO: <ask@perl.org>

sub rcpt { ... }

```
if ($rc == DONE) {
  return 1;
}
elsif ($rc == DENY) {
  $msg ||= 'relaying denied';
  $self->respond(550, $msg);
}
elsif ($rc == DENYSOFT) {
  $msg ||= 'relaying denied';
  return $self->respond(450, $msg);
}
elsif ($rc == DENY_DISCONNECT) {
    $msg ||= 'delivery denied';
    $self->log(LOGINFO, "delivery denied ($msg)");
    $self->respond(550, $msg);
    $self->disconnect;
}
elsif ($rc == DENYSOFT_DISCONNECT) {
  $msg ||= 'relaying denied';
  $self->log(LOGINFO, "delivery denied ($msg)");
  $self->respond(421, $msg);
  $self->disconnect;
}
elsif ($rc == OK) {
  $self->respond(250, $rcpt->format . ", recipient ok");
  return $self->transaction->add_recipient($rcpt);
}
else {
  return $self->respond(450, "No plugin decided if relaying is allowed");
}
return 0;
```

run_hooks

- Run all plugins configured for a hook
  - Until one of them says DENY/OK/DONE/...

```
sub run_hooks {
 my ($self, $hook) = (shift, shift);
  my $hooks = $self->{hooks};
  if ($hooks->{$hook}) {
    my @r;
    for my $code (@{$hooks->{$hook}}) {
      eval { (@r) = $code->{code}->($self, $self->transaction, @_); };
        $@ and $self->log(LOGCRIT, "FATAL PLUGIN ERROR: ", $@)
           and next;
      last unless $r[0] == DECLINED;
    }
    $r[0] = DECLINED if not defined $r[0];
    return @r;
  }
  return (0, '');
}
```

run_hooks 2

- Actually, that was a lie
- run_hooks is more complicated now
- Supports logging hooks (uh, deep recursion!)
- Split into run_hooks and run_hook
- qpsmtpd runs (optionally) in an event driven framework so it support continuations

Run hooks on hooks!

- More complexity from run_hooks we skipped!
- You can “hook” a plugin into return values from other hooks!

```
if ($r[0] == DENY
      or $r[0] == DENYSOFT
      or ... )
   {
     $r[1] = "" if not defined $r[1];
     $self->log(LOGDEBUG, "Plugin " . $code->{name}....);
                   . ", hook $hook returned "
                   . return_code($r[0])
                   . ", $r[1]"
                );
     $self->run_hooks("deny", $code->{name}, $r[0], $r[1])
       unless ($hook eq "deny");
   }
 else { ... $self->run_hooks(“ok”) ... }
```

Inherit from another plugin!

- Change how plugins work!
- Change return codes, run plugins conditionally

```
sub init {
  my ($self, $qp) = @_;
  $self->isa_plugin('greylisting');
}

sub hook_rcpt {
  my $self = shift;
  my $c = $self->qp->connection->notes('geoip_country') ||'';
  $self->SUPER::hook_rcpt( @_ )
    if $c eq 'cn';  # only do greylisting on hosts from China
}
```

THANK YOU

- Now go build programs with plug-ins!
- Questions?
- qpsmtpd
  - http://smtpd.develooper.com/
  - http://svn.perl.org/qpsmtpd/
- http://develooper.com/talks/
- ask@develooper.com or ask@perl.org