Industrial Strength Perl

Dave Rolsky

House Absolute Consulting

www.houseabsolute.com

dave@houseabsolute.com

OSCON 2003

Industrial Strength Perl

Industrial

- Why the word "industrial"?
- Has many bad connotations.
- Inflexible, top-heavy, slow to change ...
- But there's a word often associated with factories and industry that I have in mind.

Industrial Strength Perl

Failsafe

- Dictionary.com says "capable of compensating automatically and safely for a failure ...".
- That's a great concept for programmers.
- But I actually have a lesser goal in mind ...
- If your code can't work in all cases, then don't fsck things up **too** badly when it fails.

Industrial Strength Perl

Be Predictable

- It's better to fail **predictably** than to fail unpredictably.
- Every program you write that does anything important needs to include code paths for failure.
- To paraphrase Eric Raymond, "fail early, fail often".
- There is nothing worse than code that forges blindly ahead.
- Code like that **will** fail, eventually.
- But it will probably break something important first.

Industrial Strength Perl

"Tell Her About It"

- Of course, it's not enough to fail predictably if you don't record it.
- It's really hard to debug a failure in a production system if you don't have good logs.

Industrial Strength Perl

Some Guidelines

- Never assume you have good inputs.
- Everyone you work with is probably an idiot.
- Your customers are almost certainly idiots.
- **You** might be an idiot.
- And there is absolutely no question that **I** am an idiot.
- Therefore, your code **will** receive bad input during its lifetime in production.

Industrial Strength Perl

Some Guidelines

- It is better to fail **intentionally** rather than accidentally.
- An exception (dying) is probably a better way to fail than return values.
- In case you forgot, everyone you work with is an idiot.
- They **will** forget to check return values.
- If they want to ignore exceptions, at least they'll have to work a little harder.

Industrial Strength Perl

Some Guidelines

- Logging is good.
- In most cases, more verbose logs are gooder.
- Multiple logging outputs are often goodest.

Industrial Strength Perl

Real Code

- That was the why, now the how.
- Start with exceptions.
- Then input validation.
- Finally, logging.

Industrial Strength Perl

Die, Die, Die!

- Perl has no explicit exception mechanism.
- However, it does have the `die()` built-in function and `eval` blocks.
- If code is evaluated inside an `eval` **block**, then fatal errors are trapped.
- Except for things like seg faults, bus errors, etc.
- The first argument given to `die()` will be available outside of the `eval` block in the `$@` variable.
- The block form of `eval` is entirely different from the string form.

Very Simple Exceptions

```
eval { die "I am dead" };
warn $@ if length $@;
```

- The `$@` variable is cleared every time `eval` is used.

```
eval { die "I am dead" };
eval { $x = 1 };
warn $@ if length $@;
```

- In this case, `$@` will be an empty string.
- Note that `$@` is **always** defined.

Industrial Strength Perl

"Real" Exceptions

- Before Perl 5.005, `$@` could only contain a string.
- If the first argument to `die()` was a reference, it was stringified before being put in `$@`.
- But as of 5.005, if a reference is given to `die()`, then the actual reference will be stored in `$@`.
- So now we can do this:

```
eval { die { error => 'Bad SQL',
             sql => $sql,
             bound_vars => \@bound_vars } };
if ($@)
{
    warn "Error: $@->{error}\n";
    warn "SQL: $@->{sql}\n" if exists $@->{sql};
}
```

Industrial Strength Perl

"Realer" Exceptions

- Of course, the previous example is just begging to be made into an object.

```
eval { die SQLException->new
                ( error => 'Bad SQL',
                  sql => $sql,
                  bound_vars => \@bound_vars ) };
if ($@)
{
    warn "Error: ", $@->error, "\n";
    warn "SQL: ", $@->sql, "\n"
        if $@->isa('SQLException');
}
```

A Caveat

- The previous example called methods on $@.
- This can cause problems.
- Perl's built-in exceptions are still thrown as strings.
- So if you wrap some code in an `eval` block that divides by zero, $@ will contain a string.

Industrial Strength Perl

A Caveat

- The fix is to use `UNIVERSAL::isa()` to check that `$@` is an object first.
- It's uglier, but it won't blow up when `$@` contains a string.
- So we can do this:

```
if ($@) {
    if ( UNIVERSAL::isa( $@, 'Exception' ) ) {
        warn "Error: ", $@->error, "\n";
        warn "SQL: ", $@->sql, "\n"
            if $@->isa('SQLException');
    } else {
        warn "Error: $@\n";
    }
}
```

A Caveat

- Of course, we'd want to wrap up the ugly if-else bits into a subroutine, so we can just write something like `check_exception($@)`.

Another Caveat

- Remember that every time `eval` is used, the `$@` variable is cleared.
- So you probably should copy it to another variable before proceeding with execution.

```
if (my $exc = $@) {
    do_something();
    handle_exception($exc);
}
```

- The `do_something()` subroutine might use `eval`, or call other code that uses `eval`.
- This is doubly important if `do_something()` might invoke code in a module that is beyond your control.

Industrial Strength Perl

Exception Modules on CPAN

- This is one area of CPAN that actually **not** overwhelmed with redundant modules.
- My favorite exception-related module on CPAN is `Exception::Class`.
- Of course, I wrote it, so I'm biased.

Exception Modules on CPAN

- There is also `Error.pm`, which is fairly popular.
- In a recent search, I found a few others.
- But the docs were not sufficient for me to figure them out.
- So we won't talk about them.

Industrial Strength Perl

Exception::Class

- This module has several main features.
- First of all, it lets you declare a hierarchy of exception classes at compile time.

```
use Exception::Class
    ( 'MyException',

      'AnotherException' =>
      { isa => 'MyException' },

      'YetAnotherException' =>
      { isa => 'AnotherException' } );
```

Industrial Strength Perl

Exception::Class

- By default, declared classes are subclasses of the `Exception::Class::Base` class.
- But you can use your own base class if you want.
- Classes can be given arbitrary "fields", for storing additional information.

```
use Exception::Class
    ( 'SQLException' =>
      { fields => [ 'sql', 'bound_values' ] }
    );

eval { SQLException->throw
           ( error => 'bad sql', sql => $sql ) };

print $@->sql;
```

Industrial Strength Perl

Exception::Class

- Each class can have an associated subroutine as shorthand

```
use Exception::Class
    ( 'SQLException' =>
      { fields => [ 'sql', 'bound_values' ],
        alias  => 'sql_error' }
    );

sql_error error => 'bad sql', sql => $sql;
```

Industrial Strength Perl

Exception::Class

- The default base class, `Exception::Class::Base`, provides a number of handy methods.
- These provide ways to get things like a stack trace, the error message, process id, etc.

Industrial Strength Perl

Error.pm

- Error.pm provides more advanced try/catch functionality than can be achieved with `eval` and `die()`.
- It also includes a base exception class, `Error::Simple`, which is similar to `Exception::Class::Base`.
- It does not provide a means to declaratively specify exception classes, so subclasses all have to be hand-coded.

Industrial Strength Perl

Error.pm

- Its try/catch syntax looks like this:

```
use Error qw(:try);

try {
    cause_an_error();
} catch Error::Foo with {
    my $error = shift;
    warn $error->text;
} catch Error::Bar with {
    my $error = shift;
    do_some_cleanup();
    throw $error; #rethrow
} otherwise {
    my $error = shift;
    panic();
} finally {
    close_some_handle();
}; # trailing semi-colon is needed
```

Industrial Strength Perl

Error.pm

- One big warning about the try/catch syntax.
- This is implemented through the use of closures.
- A try block which in turn contains other try blocks will therefore contain nested closures.
- Nested closures can cause memory leaks in Perl.
- This was definitely true in 5.6.1, and may still be a problem with 5.8.0.

Industrial Strength Perl

Parameter Validation

- Unlike with exceptions, there are indeed many redundant parameter validation modules on CPAN.
- There are also more narrowly focused modules for validating specific data types, like emails or credit card numbers.

Industrial Strength Perl

Params::Validate

- This one is my contribution to the mess.
- It can be used to validate named or positional parameters.
- Validation can be as simple as simply listing required parameters.

```
my %p = validate( @_, { foo => 1, # required
                        bar => 0, # optional
                      } );

# first 2 are required, last is optional
my @p = validate_pos( @_, 1, 1, 0 );
```

Industrial Strength Perl

Params::Validate

- Other validation options include type (scalar, array ref, etc.), class, `->can`, regex match, and callbacks.

```
my %p =
   validate
       ( @_,
         { foo => { type => SCALAR | UNDEF },
           bar => { isa => 'Bar' },
           baz => { can => 'print' },
           color =>
           { regex => qr/^(?:red|blue)$/ },
           number =>
           { callbacks =>
             { 'between 1 and 30' =>
               sub { $_[0] >= 1 && $_[0] <= 30 },
           },
         } );
```

Industrial Strength Perl

Params::Validate

- Using callbacks makes it easy to do more complex validation like checking if an email address is valid.

```
validate
   ( @_,
     { email =>
       { callbacks =>
         { 'email is valid' =>
           sub { Email::Valid->address( $_[0] ) } },
       },
       credit_card =>
       { callbacks =>
         { 'credit card is valid' =>
           sub { Business::CreditCard::validate( $_[0] ) } },
         },
       },
     );
```

Industrial Strength Perl

Params::Validate

- Finally, you can set default values for parameters.

```
my %p =
    validate
        ( @_,
          { size => { type => SCALAR,
                      default => 10 } }
        );

# $p{size} is 10 if it wasn't provided
# by the caller
```

Industrial Strength Perl

Params::Validate

- This module is fairly mature, at well over two years of age.
- Recent versions are entirely in XS.
- You can turn off validation entirely by setting an environment variable.
- But I never do this, because I don't trust myself that much.

Industrial Strength Perl

Getargs::Long

- Can validate based on type or class.

```
my ( $foo, $bar ) =
    getargs( @_, 'foo=ARRAY', 'bar=Some::Class' );
```

- Can specify some parameters as optional.

```
my ( $foo, $bar ) =
    xgetargs( @_,
              foo => 'ARRAY',
              bar => [ 'Some::Class' ] );
```

Industrial Strength Perl

Getargs::Long

- Defaults can be specified as well.

```
my ( $foo, $bar ) =
    xgetargs( @_,
              foo => [ 'i', 23 ], # 'i' = integer
              bar => [ 'Some::Class' ] );
```

Industrial Strength Perl

Getargs::Long

- There is no way to integrate other types of validation.
- It also has no active maintainer, but the original author is willing to give maintainership to someone else.

Industrial Strength Perl

Data::FormValidator

- This module has only a little to do with HTML forms.
- The basic usage is similar to `Params::Validate` and `Getargs::Long`.
- Parameters can be defined as required or optional.

```
my $results =
   Date::FormValidator->check
       ( %params,
         { required  => [ 'foo', 'bar' ],
           optional  => [ 'baz' ] } );
```

- A parameter which has a value only containing whitespace is considered to be missing.

Industrial Strength Perl

Data::FormValidator

- Each parameter can be given one or more constraints.

```
my $results =
   Date::FormValidator->check
       ( %params,
         { required  => [ 'foo', 'bar', 'email' ],
           constraints => { email => 'email' }
         } );
```

- The distribution comes with some pre-defined constraints, but it is easy to integrate your own custom constraints.

Industrial Strength Perl

Data::FormValidator

- Parameters can be given defaults.
- Values for parameters can be transformed by filters.
- As with constraints, the distribution provides some filters, but custom filters are possible as well.
- It is also possible to specify dependencies between parameters, so that if one is present, others become required.
- Validation results are returned as an object.

Other Parameter Validation Modules

- There are a number of other parameter validation modules on CPAN.
- `Class::Contract` - design-by-contract in Perl
- `Data::Validator::Item`
- `Params::Check`
- And probably many more I didn't notice!

Industrial Strength Perl

Logging

- Surprisingly, there aren't an overwhelming number of general-purpose logging modules on CPAN.
- By general purpose, I mean a module designed to provide one API for multiple logging outputs.

Industrial Strength Perl

Log::Dispatch

- No surprise, we'll start with the one I wrote.
- `Log::Dispatch` lets you create a single dispatch object which holds multiple outputs.
- There are output modules for files, email, the screen, etc.
- Each output can have a minimum and maximum accepted log level.

Industrial Strength Perl

Log::Dispatch

- A simple example:

```
my $dispatch = Log::Dispatch->new;
$dispatch->add( Log::Dispatch::File->new
                    ( name => 'debug',
                      file => 'debug.log',
                      min_level => 'debug' ) );

$dispatch->add( Log::Dispatch::File->new
                    ( name => 'error',
                      file => 'error.log',
                      min_level => 'error' ) );

$dispatch->add( Log::Dispatch::Screen->new
                  ( name => 'screen',
                    min_level => 'debug' ) );

$dispatch->log( level => 'info', message => 'information' );
```

Industrial Strength Perl

Log::Dispatch

- Filters can be defined for each output, as well as for the dispatcher itself.

```
my $dispatch = Log::Dispatch->new;
$dispatch->add
    ( Log::Dispatch::File->new
          ( name => 'debug',
            file => 'debug.log',
            min_level => 'debug',
            callbacks =>
            sub { my %p = @_;
                  $p{message} =
                      "[$p{level}] $p{message}" }
          ) );
```

Industrial Strength Perl

Log::Dispatch

- Writing new outputs is very easy, and several are available on CPAN.
- `Log::Dispatch::Config`, written by Tatsuhiko Miyagawa, allows you to configure logging in a text file.
- This is similar to log4j.

Industrial Strength Perl

Log::Log4Perl

- A complete log4j implementation in Perl.
- Can be configured from a text file or via Perl code.
- Can log one message to multiple outputs.
- Has a `sprintf()`-like pattern language for specifying how log messages should be formatted.
- Comes with some output modules, and can also use `Log::Dispatch` output modules.

Industrial Strength Perl

Log::Log4Perl

- Definitely the most powerful logging module on CPAN.
- But it is also the most complex.
- Fortunately, it is well-documented.

Industrial Strength Perl

Log::Agent

- Unlike the other two modules we've seen, `Log::Agent` provides a procedural interface to logging.
- Output can be sent to three different channels.
- There are three channels, "debug", "error", and "output".
- Outputs are further categorized by log level.

Industrial Strength Perl

Log::Agent

- Example:

```
logerr "error"; # uses default outputs

logconfig
    ( -driver =>
      Log::Agent::Driver::File->make
          ( -channels    =>
            { 'error'  => "$0.err",
              'output' => "$0.out",
              'debug'  => "$0.dbg",
            } ) );

# now goes to the $0.err file
logerr "another error";
```

Log::Agent

- It also possible to do filtering of the messages on a per-output basis.
- And of course, you can add your own output modules.