Advanced Perl Techniques

A One Day Perl Tutorial

Dave Cross

Magnum Solutions Ltd

dave@mag-sol.com

UKUUG
26th February 2009

Advanced Perl Techniques

- Advanced level training for Perl programmers
- Turn intermediate programmers into advanced programmers
- “Modern” Perl
- Perl is not dying

Advanced Perl Techniques

- One day isn't enough time
- We'll be moving fairly fast
- Lots of pointers to other information
- Feel free to ask questions

What We Will Cover

- What's new in Perl 5.10
- Dates and times
- Testing
  - including coverage analysis
- Database access
  - DBIx::Class

What We Will Cover

- Profiling & Benchmarking
- Object oriented programming with Moose
- Templates
- MVC Frameworks
  - Catalyst

Schedule

- 09:45 – Begin
- 11:15 – Coffee break
- 13:00 – Lunch
- 14:00 – Begin
- 15:30 – Coffee break
- 17:00 – End

Resources

- Slides available on-line
  - http://mag-sol.com/train/public/2009-02/adv
- Also see Slideshare
  - http://www.slideshare.net/davorg/slideshows
- Mailing List
  - http://lists.mag-sol.com/mailman/listinfo/adv2009
- Get Satisfaction
  - http://getsatisfaction.com/magnum

Perl 5.10

Magnum
Solutions Limited
Open Source Consultancy, Development & Training

Perl 5.10

- Released 18th Dec 2007
  - Perl's 20th birthday
- Many new features
- Well worth upgrading

New Features

- Defined-or operator
- Switch operator
- Smart matching
- say()
- Lexical $_

New Features

- State variables
- Stacked file tests
- Regex improvements
- Many more

Defined Or

- Boolean expressions “short-circuit”
- `$val = $val || $default;`
- `$val ||= $default;`
- What if 0 is a valid value?
- Need to check “definedness”

-
  ```
  $val = defined $val
                 ? $val : $default;
  ```
- `$val = $default unless defined $val;`

Defined Or

- The defined or operator makes this easier
- `$val = $val // $default;`
- A different slant on truth
- Checks definedness
- Short version too
- `$val //= $default;`

Switch Statement

- Switch.pm was added with Perl 5.8
- Source filter
- Parser limitations
  - Regular expressions
  - eval
- 5.10 introduces a build-in switch statement

Given ... When

- Switch is spelled “given”
- Case is spelled “when”
- Powerful matching syntax

Given Example

-
  ```
  given ($foo) {
          when (/^abc/) { $abc = 1; }
          when (/^def/) { $def = 1; }
          when (/^xyz/) { $xyz = 1; }
          default { $nothing = 1; }
  }
  ```
- Four new keywords
  - given
  - when
  - default
  - continue

given

- `given(EXPR)`
- Assigns the result of EXPR to $_ within the following block
- Similar to `do { my $_ = EXPR; ... }`

when

- `when (EXPR)`
- Uses smart matching to compare $_ with EXPR
- Equivalent to `when ($_ ~~ EXPR)`
- ~~ is the new smart match operator
- Compares two values and “does the right thing”

default

- default defines a block that is executed if no when blocks match
- default block is optional

continue

- continue keyword falls through to the next when block
- Normal behaviour is to break out of given block once the first when condition is matched
-
```
given($foo) {
    when (/x/)
      {say '$foo contains an x'; continue }
    when (/y/)
      { say '$foo contains a y' }
    default
      { say '$foo contains no x or y' }
}
```

Smart Matching

- ~~ is the new Smart Match operator
- Different kinds of matches
- Dependent on the types of the operands
- See “perldoc perlsyn” for the full details

Smart Match Examples

- `$foo ~~ $bar; # == or eq`
- `@foo ~~ $bar; # array contains value`
- `%foo ~~ $bar; # hash key exists`
- `$foo ~~ qr{$bar}; # regex match`
- `@foo ~~ @bar; # arrays are identical`
- `%foo ~~ %bar; # hash keys match`
- Many more alternatives

say()

- say() is a new alternative to print()
- Adds a new line at the end of each call
- say($foo); # print $foo, “\n”;
- Two characters shorter than print
- Less typing

Lexical $_

- $_ is a package variable
- Always exists in main package
- Can lead to subtle bugs when not localised correctly
- Can now use `my $_` to create a lexically scoped variable called $_

State Variables

- Lexical variables disappear when their scope is destroyed

-
  ```
  sub variables {
    my $x;

    say ++$x;
  }

  variables() for 1 .. 3;
  ```

State Variables

- State variables retain their value when their scope is destroyed

-
  ```
  sub variables {
    state $x;

    say ++$x;
  }

  variables() for 1 .. 3;
  ```
- Like static variables in C

Stacked File Tests

- People often think you can do this
- `-f -w -x $file`
- Previously you couldn't
- Now you can
- Equivalent to
- `-x $file && -w _ && -f _`

Regex Improvements

- Plenty of regular expression improvements
- Named capture buffers
- Possessive quantifiers
- Relative backreferences
- New escape sequences
- Many more

Named Capture Buffers

- Variables $1, $2, etc change if the regex is altered
- Named captures retain their names
- (?<name> ...) to define
- Use new %+ hash to access them

Named Capture Example

-
  ```
  while (<DATA>) {
    if (/(?<header>[\w\s]+)
          :\s+(?<value>.+)/x) {
      print "$+{header} -> ";
      print "$+{value}\n";
    }
  }
  ```

Possessive Quantifiers

- `?+, *+, ++`
- Grab as much as they can
- Never give it back
- Finer control over backtracking
- `'aaaa' =~ /a++a/`
- Never matches

Relative Backreferences

- `\g{N}`
- More powerful version of `\1`, `\2`, etc
- `\g{1}` is the same as `\1`
- `\g{-1}` is the last capture buffer
- `\g{-2}` is the one before that

New Escape Sequences

- `\h` – Horizontal white space
- `\v` – Vertical white space
- Also `\H` and `\V`

Accessing New Features

- Some new features would break backwards compatibility
- They are therefore turned off by default
- Turn them on with the `feature` pragma
- `use feature 'say';`
- `use feature 'switch';`
- `use feature 'state';`
- `use feature ':5.10';`

Implicit Loading

- Two ways to automatically turn on 5.10 features
- Require a high enough version of Perl
- `use 5.10.0; # Or higher`
- -E command line option
- `perl -e 'say “hello”'`
- `perl -E 'say “hello”'`

Dates and Times

Dates & Times

- Dozens of date/time modules on CPAN
- Date::Manip is almost never what you want
- Date::Calc, Date::Parse, Class::Date, Date::Simple, etc
- Which one do you choose?

Perl DateTime Project

- http://datetime.perl.org/
- *"The DateTime family of modules present a unified way to handle dates and times in Perl"*
- "unified" is good
- Dozens of modules that work together in a consistent fashion

Using DateTime

-
  ```
  use DateTime;
  my $dt = DateTime->now;
  say $dt;
  # 2009-02-26T11:06:07
  say $dt->ymd;
  # 2009-02-26
  say $dt->hms;
  # 11:08:16
  ```

Using DateTime

```
use DateTime;
my $dt = DateTime->new(year  => 2009,
                       month => 2,
                       day   => 26);
say $dt->ymd('/');
# 2009/02/26
say $dt->month;      # 2
say $dt->month_name; # February
```

Arithmetic

- A DateTime object is a point in time
- For date arithmetic you need a duration
- Number of years, weeks, days, etc

Arithmetic

-
  ```
  use DateTime;
  my $dt = DateTime->new(year => 2009,
                         month => 2,
                         day => 26);
  my $two_weeks =
  DateTime::Duration->new(weeks => 2);
  $dt += $two_weeks;
  say $dt;
  # 2009-03-12T00:00:00
  ```

Formatting Output

-
  ```
  use DateTime;
  my $dt = DateTime->new(year => 2009,
                         month => 2,
                         day => 26);
  say $dt->strftime('%A, %d %B %Y');
  # Tuesday, 26 February 2009
  ```
- Control input format with DateTime::Format::Strptime

Magnum Solutions Limited
Open Source Consultancy, Development & Training

Parsing & Formatting

- Ready made parsers and formatters for popular date and time formats
- DateTime::Format::HTTP
- DateTime::Format::MySQL
- DateTime::Format::Excel
- DateTime::Format::Baby
  - the big hand is on...

Alternative Calendars

- Handling non-standard calendars
- DateTime::Calendar::Julian
- DateTime::Calendar::Hebrew
- DateTime::Calendar::Mayan
- DateTime::Fiction::JRRTolkien::Shire

Calendar Examples

-
  ```
  use DateTime::Calendar::Mayan;
  my $dt = DateTime::Calendar::Mayan->now;
  say $dt->date; # 12.19.16.1.15
  ```

-
  ```
  use DateTime::Fiction::JRRTolkien::Shire
  my $dt =
    DateTime::Fiction::JRRTolkien::Shire->now;
  say $dt->on_date;
  # Mersday 24 Solmath 7473
  ```

Testing

Testing

- Never program without a safety net
- Does your code do what it is supposed to do?
- Will your code continue to do what it is supposed to do?
- Write unit tests
- Run those tests all the time

When to Run Tests

- As often as possible
- Before you add a feature
- After you have added a feature
- Before checking in code
- Before releasing code
- Constantly, automatically

Testing in Perl

- Perl makes it easy to write test suites
- A lot of work in this area over the last eight years
- Test::Simple and Test::More included in Perl distribution
- Many more testing modules on CPAN

Simple Test Program

-
  ```
  use Test::More tests => 4;

  BEGIN { use_ok('My::Object'); }

  ok(my $obj = My::Object->new);
  isa_ok($obj, 'My::Object');
  $obj->set_foo('Foo');
  is($obj->get_foo, 'Foo');
  ```

Simple Test Output

-
  ```
  $ prove -v test.t
  test....
  1..4
  ok 1 - use My::Object;
  ok 2
  ok 3 - The object isa My::Object
  ok 4
  ok
  All tests successful.
  Files=1, Tests=4,  0 wallclock secs (
  0.02 usr  0.00 sys +  0.05 cusr  0.00
  csys =  0.07 CPU)
  Result: PASS
  ```

Adding Test Names

-
  ```
  use Test::More tests => 4;
  BEGIN { use_ok('My::Object'); }

  ok(my $obj = My::Object->new,
     'Got an object');
  isa_ok($obj, 'My::Object');
  $obj->set_foo('Foo');
  is($obj->get_foo, 'Foo',
     'The foo is "Foo"');
  ```

Output With Names

-
  ```
  $ prove -v test2.t
  test2....
  1..4
  ok 1 - use My::Object;
  ok 2 - got an object
  ok 3 - The object isa My::Object
  ok 4 - The foo is "Foo"
  ok
  All tests successful.
  Files=1, Tests=4,  0 wallclock secs (
  0.02 usr  0.00 sys +  0.05 cusr  0.00
  csys =  0.07 CPU)
  Result: PASS
  ```

Using `prove`

- `prove` is a command line tool for running tests
- Runs given tests using Test::Harness
- Comes with the Perl distribution
- Command line options
  - -v verbose output
  - -r recurse
  - -s shuffle tests
  - Many more

Test Anything Protocol

- Perl tests have been spitting out “ok 1” and not “ok 2” for years
- Now this ad-hoc format has a definition and a name
- The Test Anything Protocol (TAP)
- See Test::Harness::TAP (documentation module) and TAP::Parser

TAP Output

- More possibilities for test output
  - TAP::Harness::Color
  - Test::TAP::HTMLMatrix
- Make sense of your test results

More Testing Modules

- Dozens of testing modules on CPAN
- Some of my favourites
- Test::File
- Test::Exception, Test::Warn
- Test::Differences
- Test::XML (includes Test::XML::XPath)

Writing Test Modules

- These test modules all work together
- Built using Test::Builder
- Ensures that test modules all use the same framework
- Use it as the basis of your own Test::* modules
- Who tests the testers?
- Test your Test::Builder test modules with Test::Builder::Tester

Mocking Objects

- Sometimes it's hard to test external interfaces
- Fake them
- Test::MockObject pretends to be other objects
- Gives you complete control over what they return

Testing Reactors

- You're writing code that monitors a nuclear reactor
- It's important that your code reacts correctly when the reactor overheats
- You don't have a reactor in the test environment

Testing Reactors

- Even if you did, you wouldn't want to make it overheat every time you run the tests
- Especially if you're not 100% sure of your code
- Or if you're running unattended smoke tests
- Fake it with a mock object

My::Monitor Spec

- If the temperature of a reactor is over 100 then try to cool it down
- If you have tried cooling a reactor down 5 times and the temperature is still over 100 then return an error

My::Monitor Code

-
  ```
  package My::Monitor;

  sub new {
    my $class = shift;
    my $self = { tries => 0 };

    return bless $self, $class;
  }
  ```

My::Monitor Code

-
  ```
  sub check {
    my $self = shift;
    my $reactor = shift;

    my $temp = $reactor->temperature;

    if ($temp > 100) {
      $reactor->cooldown;
      ++$self->{tries};
      if ($self->{tries} > 5) {
        return;
      }
      return 1;
  ```

My::Monitor Code

```
•       } else {
            $self->{tries} = 0;
            return 1;
        }
    }

    1;
```

Mock Reactor

- Create a mock reactor object that acts exactly how we want it to
- Reactor object has two interesting methods
- temperature - returns the current temperature
- cooldown - cools reactor and returns success or failure

monitor.t

```
• use Test::More tests => 10;

  use Test::MockObject->new;

  # Standard tests

  BEGIN { use_ok('My::Monitor'); }

  ok(my $mon = My::Monitor->new);
  isa_ok($mon, 'My::Monitor');
```

monitor.t

-
  ```
  # Create Mock Reactor Object

  my $t = 10;
  my $reactor = Test::MockObject;

  $reactor->set_bound('temperature',
                      \$t);

  $reactor->set_true('cooldown');
  ```

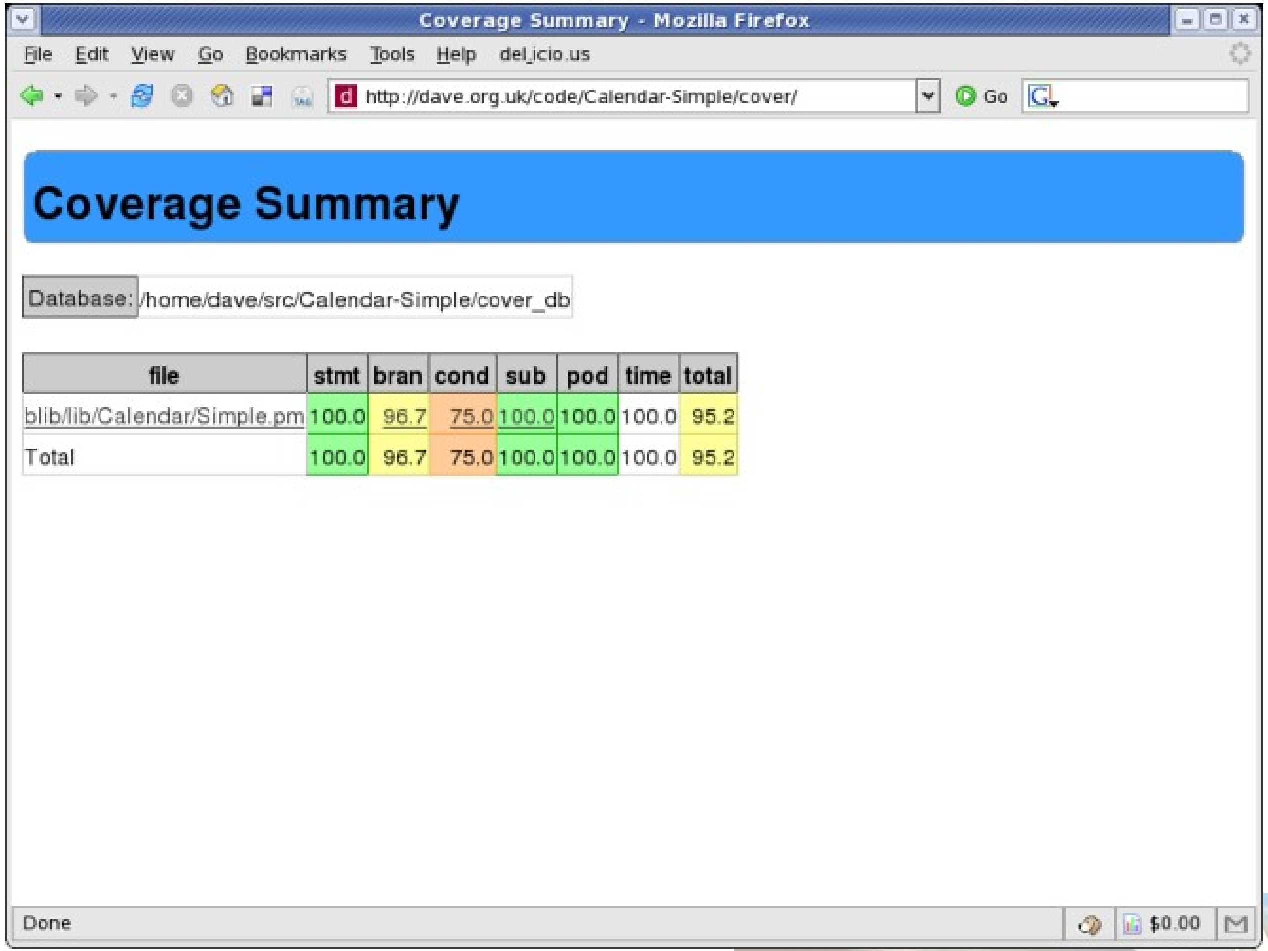
monitor.t

-
  ```
  # Test reactor
  ok($mon->check($reactor));
  $t = 120;
  ok($mon->check($reactor)) for 1 .. 5;
  ok(!$mon->check($reactor));
  ```

How Good Are Your Tests?

- How much of your code is exercised by your tests?
- Devel::Cover can help you to find out
- Deep internal magic
- Draws pretty charts
  - `HARNESS_PERL_SWITCHES= -MDevel::Cover make test`
  - `cover`

Devel::Cover Output

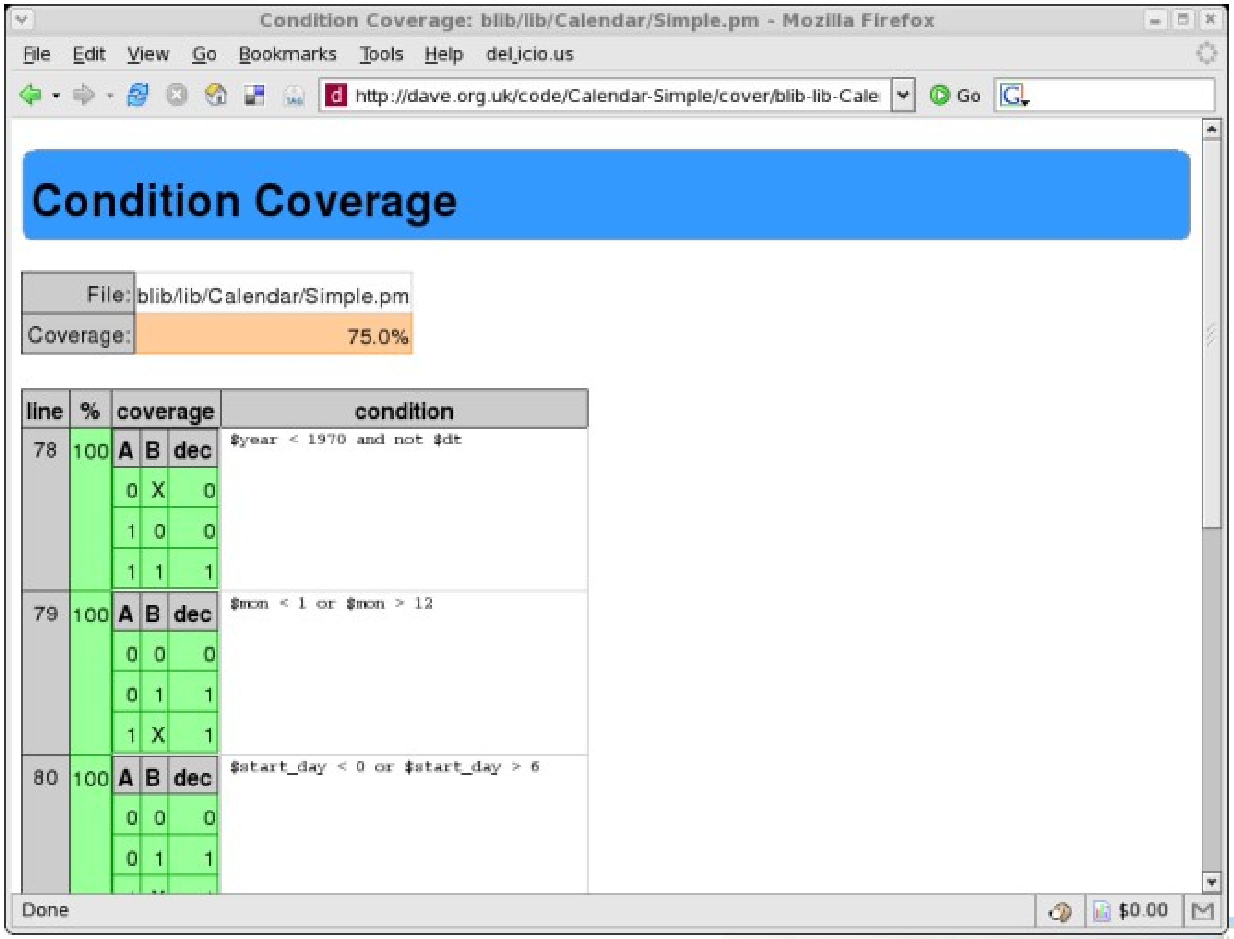

Coverage Summary - Mozilla Firefox
File Edit View Go Bookmarks Tools Help del.icio.us
http://dave.org.uk/code/Calendar-Simple/cover/
Go
Done
$0.00

Coverage Summary

Database: /home/dave/src/Calendar-Simple/cover_db

file	stmt	bran	cond	sub	pod	time	total
blib/lib/Calendar/Simple.pm	100.0	96.7	75.0	100.0	100.0	100.0	95.2
Total	100.0	96.7	75.0	100.0	100.0	100.0	95.2

Devel::Cover Output

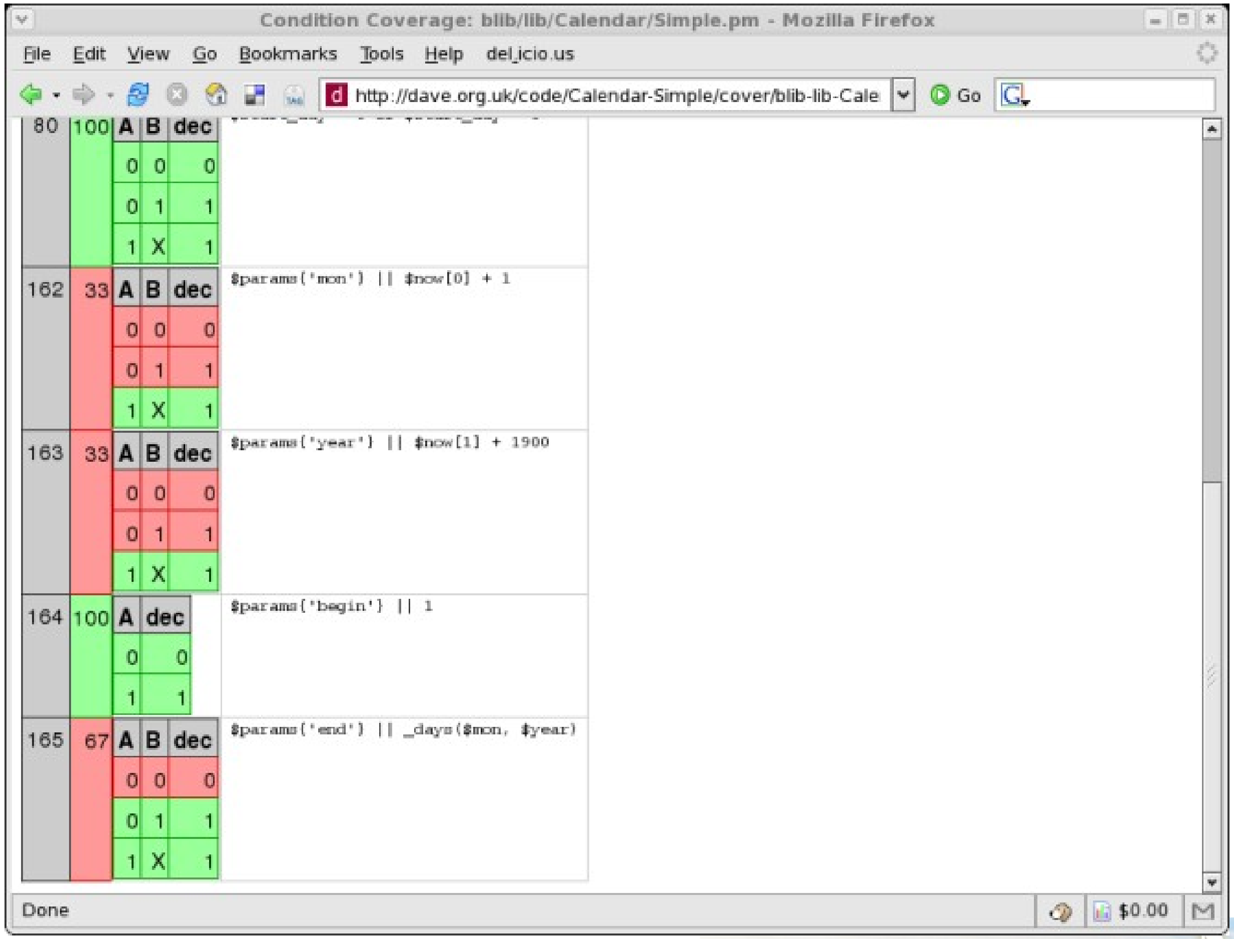

Condition Coverage: blib/lib/Calendar/Simple.pm - Mozilla Firefox
File Edit View Go Bookmarks Tools Help del.icio.us
http://dave.org.uk/code/Calendar-Simple/cover/blib-lib-Cale
Go
Done
$0.00

Condition Coverage

File:	blib/lib/Calendar/Simple.pm
Coverage:	75.0%

line	%	A	B	dec	condition
78	100	A	B	dec	`$year < 1970 and not $dt`
		0	X	0	
		1	0	0	
		1	1	1	
79	100	A	B	dec	`$mon < 1 or $mon > 12`
		0	0	0	
		0	1	1	
		1	X	1	
80	100	A	B	dec	`$start_day < 0 or $start_day > 6`
		0	0	0	
		0	1	1	

Devel::Cover Output



Condition Coverage: blib/lib/Calendar/Simple.pm - Mozilla Firefox

File Edit View Go Bookmarks Tools Help del.icio.us
http://dave.org.uk/code/Calendar-Simple/cover/blib-lib-Cale
Go
Done
$0.00

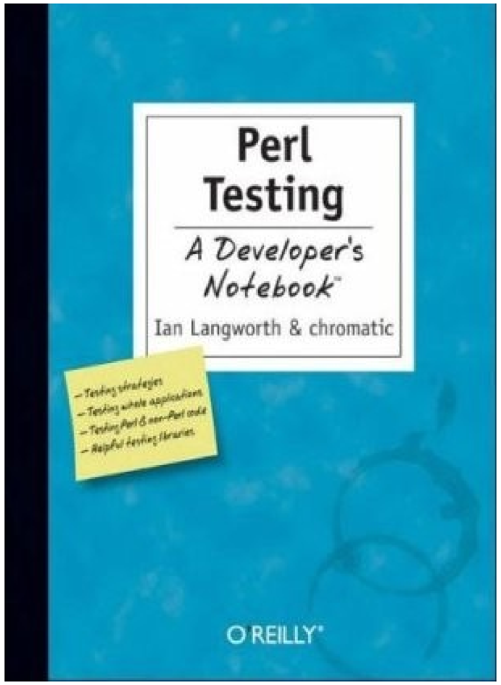
80	100	A	B	dec	[illegible]
		0	0	0	
		0	1	1	
		1	X	1	
162	33	A	B	dec	`$params{'mon'} || $now[0] + 1`
		0	0	0	
		0	1	1	
		1	X	1	
163	33	A	B	dec	`$params{'year'} || $now[1] + 1900`
		0	0	0	
		0	1	1	
		1	X	1	
164	100	A	dec		`$params{'begin'} || 1`
		0	0		
		1	1		
165	67	A	B	dec	`$params{'end'} \|\| _days($mon, $year)`
		0	0	0	
		0	1	1	
		1	X	1	

Alternative Test Paradigms

- Not everyone likes the Perl testing framework
- Other frameworks are available
- Test::Class
  - xUnit style framework
- Test::FIT
  - Framework for Interactive Testing
  - http://fit.c2.com

More Information

- Perl Testing: A Developer's Notebook (Ian Langworth & chromatic)
- perldoc Test::Tutorial
- perldoc Test::Simple
- perldoc Test::More
- perldoc Test::Builder
- etc...



Benchmarking

Benchmarking

- Ensure that your program is fast enough
- But how fast is fast enough?
- *premature optimization is the root of all evil*
  - Donald Knuth
  - paraphrasing Tony Hoare
- Don't optimise until you know what to optimise

Benchmark.pm

- Standard Perl module for benchmarking
- Simple usage

-
  ```
  use Benchmark;
  my %methods = (
      method1 => sub { ... },
      method2 => sub { ... },
  );
  timethese(10_000, \%methods);
  ```
- Times 10,000 iterations of each method

Benchmark.pm Output

```
• Benchmark: timing 10000 iterations of
  method1, method2...
   method1:  6 wallclock secs \
   ( 2.12 usr +  3.47 sys =  5.59 CPU) \
     @ 1788.91/s (n=10000)
   method2:  3 wallclock secs \
   ( 0.85 usr +  1.70 sys =  2.55 CPU) \
     @ 3921.57/s (n=10000)
```

Timed Benchmarks

- Passing `timethese` a positive number runs each piece of code a certain number of times
- Passing `timethese` a negative number runs each piece of code for a certain number of seconds

Timed Benchmarks

-
  ```
  use Benchmark;
  my %methods = (
      method1 => sub { ... },
      method2 => sub { ... },
  );

  # Run for 10,000(!) seconds
  timethese(-10_000, \%methods);
  ```

Comparing Performance

- Use `cmpthese` to get a tabular output
- Optional export

-
  ```
  use Benchmark 'cmpthese';
  my %methods = (
      method1 => sub { ... },
      method2 => sub { ... },
  );
  cmpthese(10_000, \%methods);
  ```

`cmpthese` Output

-
  ```
                      Rate method1 method2
  method1 2831802/s      --     -61%
  method2 7208959/s    155%       --
  ```
- method2 is 61% slower than method1
- Can also pass negative number to `cmpthese`

Benchmarking is Hard

- Very easy to produce lots of numbers
- Harder to ensure that the numbers are meaningful
- Compare code fragments that do the same thing

Bad Benchmarking

-
  ```
  use Benchmark qw{ timethese };
  timethese( 1_000, {
    Ordinary    => sub {
     my @results = sort { -M $a <=> -M $b }
                         glob "/bin/*";
  },
    Schwartzian => sub {
          map $_->[0],
          sort { $a->[1] <=> $b->[1] }
          map [$_, -M], glob "/bin/*";
        },
  });
  ```

What to Benchmark

- Profile your code
- See which parts it is worth working on
- Look for code that
  - Takes a long time to run, or
  - Is called many times, or
  - Both

Devel::DProf

- Devel::DProf is the standard Perl profiling tool
- Included with Perl distribution
- Uses Perl debugger hooks
- `perl -d:DProf your_program`
- Produces a data file called tmon.out
- Command line program dprofpp to view results

Sample Output

-
  ```
  $ perl -d:DProf ./invoice.pl 244
  $ dprofpp
  Total Elapsed Time = 1.173152 Seconds
    User+System Time = 0.963152 Seconds
  Exclusive Times
  %Time ExclSec CumulS #Calls sec/call Csec/c  Name
   6.02   0.058  0.067    482   0.0001 0.0001  Params::Validate::_validate
   5.09   0.049  0.114      7   0.0070 0.0163  Class::DBI::Loader::mysql::BEGIN
   4.15   0.040  0.050     10   0.0040 0.0050  Template::Parser::BEGIN
   4.15   0.040  0.166      7   0.0057 0.0237  DateTime::Locale::BEGIN
   4.05   0.039  0.094     43   0.0009 0.0022  base::import
   3.74   0.036  0.094    449   0.0001 0.0002  DateTime::Locale::_register
   3.11   0.030  0.280      4   0.0074 0.0700  DateTime::Format::MySQL::BEGIN
   2.91   0.028  0.028    170   0.0002 0.0002  Lingua::EN::Inflect::_PL_noun
   2.70   0.026  0.040      1   0.0262 0.0401  Template::Parser::_parse
   2.49   0.024  0.024   1113   0.0000 0.0000  Class::Data::Inheritable::__ANON__
   2.08   0.020  0.020     12   0.0017 0.0017  DBD::mysql::db::_login
   2.08   0.020  0.020      4   0.0050 0.0050  Template::Stash::BEGIN
   2.08   0.020  0.099      9   0.0022 0.0110  Template::Config::load
   2.08   0.020  0.067      9   0.0022 0.0074  Template::BEGIN
   2.08   0.020  0.039      4   0.0049 0.0097  Lingua::EN::Inflect::Number::BEGIN
  ```

Devel::NYTProf

- New profiling module
- Based on work from the New York Times
- Enhanced by Tim Bunce
- Pretty HTML output
  - “borrowed” from Devel::Cover
- Far more flexible
- Far more powerful

Using NYTProf

- Similar to Devel::DProf
- `$ perl -d:NYTProf ./invoice.pl 244`
- Writes nytprof.out
- `$ nytprofhtml`
- Or
- `$ nytprofcsv`

Conclusions

- Don't optimise until you know you need to optimise
- Don't optimise until you know what to optimise
- Use profiling to find out what is worth optimising
- Use benchmarking to compare different solutions

More Information

- perldoc Benchmark
- perldoc Devel::DProf
- perldoc Devel::NYTProf
- Chapters 5 and 6 of *Mastering Perl*

Object Relational Mapping

UKUUG
26th February 2009

ORM

- Mapping database relations into objects
- Tables (relations) map onto classes
- Rows (tuples) map onto objects
- Columns (attributes) map onto attributes
- Don't write SQL

SQL Is Tedious

- Select the id and name from this table
- Select all the details of this row
- Select something about related tables
- Update this row with these values
- Insert a new record with these values
- Delete this record

Replacing SQL

- Instead of

-
  ```
  SELECT *
  FROM   my_table
  WHERE  my_id = 10
  ```
- and then dealing with the prepare/execute/fetch code

Replacing SQL

- We can write

-
  ```
  use My::Object;

  # warning! not a real orm
  my $obj = My::Object->retrieve(10)
  ```
- Or something similar

Writing An ORM Layer

- Not actually that hard to do yourself
- Each class needs an associated table
- Each class needs a list of columns
- Create simple SQL for basic CRUD operations
- Don't do that

Perl ORM Options

- Plenty of choices on CPAN
- Tangram
- SPOPS (Simple Perl Object Persistence with Security)
- Alzabo
- Class::DBI
- DBIx::Class
  - The current favourite

DBIx::Class

- Standing on the shoulders of giants
- Learning from problems in Class::DBI
- More flexible
- More powerful

DBIx::Class Example

- Modeling a CD collection
- Three tables
- artist (artistid, name)
- cd (cdid, artist, title)
- track (trackid, cd, title)

Main Schema

- Define main schema class
- DB/Main.pm

-
  ```
  package DB::Main;
  use base qw/DBIx::Class::Schema/;

  __PACKAGE__->load_classes();

  1;
  ```

Object Classes

- DB/Main/Artist.pm

-
  ```
  package DB::Main::Artist;
  use base qw/DBIx::Class/;
  __PACKAGE__->load_components(qw/PK::Auto
  Core/);
  __PACKAGE__->table('artist');
  __PACKAGE__->add_columns(qw/ artistid name
  /);
  __PACKAGE__->set_primary_key('artistid');
  __PACKAGE__->has_many(cds =>
                        'DB::Main::Cd');
  1;
  ```

Object Classes

- DB/Main/CD.pm
-
```
package DB::Main::CD;
use base qw/DBIx::Class/;
__PACKAGE__->load_components(qw/PK::Auto
Core/);
__PACKAGE__->table('cd');
__PACKAGE__->add_columns(qw/ cdid artist
title year /);
__PACKAGE__->set_primary_key('cdid');
__PACKAGE__->belongs_to(artist =>
                      'DB::Main::Artist');
1;
```

Inserting Artists

```
• my $schema =
    DB::Main->connect($dbi_str);

  my @artists = ('The Beta Band',
                 'Beth Orton');

  my $art_rs = $schema->resultset('Artist');

  foreach (@artists) {
    $art_rs->create({ name => $_ });
  }
```

Inserting CDs

- Hash of Artists and CDs
-
```
my %cds = ( 'The Three EPs' =>
                      'The Beta Band',
            'Trailer Park'  =>
                      'Beth Orton');
```

Inserting CDs

- Find each artist and insert CD

-
  ```
  foreach (keys $cds) {
    my ($artist) = $art_rs->search(
                     { name => $cds{$_} }
                   );

    $artist->add_to_cds({
      title => $_,
    });
  }
  ```

Retrieving Data

- Get CDs by artist

-
  ```
  my ($artist) = $art_rs->search({
                     name => 'Beth Orton',
                   });

  foreach ($artist->cds) {
    say $_->title;
  }
  ```

Searching for Data

- Search conditions can be more complex
- Alternatives

  -
    ```
    $rs->search({year => 2006},
                {year => 2007});
    ```
- Like

  -
    ```
    $rs->search({name =>
                   { 'like', 'Dav%' }});
    ```
- Combinations

  -
    ```
    $rs->search({forename =>
                   { 'like', 'Dav%' },
                 surname => 'Cross' });
    ```

Don't Repeat Yourself

- There's a problem with this approach
- Information is repeated
- Columns and relationships defined in the database schema
- Columns and relationships defined in class definitions

Repeated Information

-
  ```
  CREATE TABLE artist (
    artistid INTEGER PRIMARY KEY,
    name     TEXT NOT NULL
  );
  ```

Repeated Information

-
  ```
  package DB::Main::Artist;
  use base qw/DBIx::Class/;
  __PACKAGE__->
   load_components(qw/PK::Auto Core/);
  __PACKAGE__->table('artist');
  __PACKAGE__->
   add_columns(qw/ artistid name /);
  __PACKAGE__>
   set_primary_key('artistid');
  __PACKAGE__->has_many('cds' =>
  'DB::Main::Cd');
  ```

Database Metadata

- Some people don't put enough metadata in their databases
- Just tables and columns
- No relationships. No constraints
- You may as well make each column VARCHAR(255)

Database Metadata

- Describe your data in your database
- It's what your database is for
- It's what your database does best

No Metadata (Excuse 1)

- "This is the only application that will ever access this database"
- Bollocks
- All data will be shared eventually
- People will update your database using other applications
- Can you guarantee that someone won't use mysql to update your database?

No Metadata (Excuse 2)

- "Our database doesn't support those features"
- Bollocks
- MySQL 3.x is not a database
  - It's a set of data files with a vaguely SQL-like query syntax
- MySQL 4.x is a lot better
- MySQL 5.x is most of the way there
- Don't be constrained by using inferior tools

Magnum Solutions Limited
Open Source Consultancy, Development & Training

DBIC::Schema::Loader

- Creates classes by querying your database metadata
- No more repeated data
- We are now DRY
- Schema definitions in one place
- But...
- Performance problems

Performance Problems

- You don't really want to generate all your class definitions each time your program is run
- Need to generate the classes in advance
- `dump_to_dir` method
- Regenerate classes each time schema changes

Magnum Solutions Limited
Open Source Consultancy, Development & Training

Alternative Approach

- Need one canonical definition of the data tables
- Doesn't need to be SQL DDL
- Could be in Perl code
- Write DBIx::Class definitions
- Generate DDL from those
- Harder approach
  - Might need to generate ALTER TABLE

Conclusions

- ORM is a bridge between relational objects and program objects
- Avoid writing SQL in common cases
- DBIx::Class is the currently fashionable module
- Lots of plugins
- Caveat: ORM may be overkill for simple programs

More Information

- Manual pages (on CPAN)
- DBIx::Class
- DBIx::Class::Manual::*
- DBIx::Class::Schema::Loader
- Mailing list (Google for it)

Moose

Magnum
Solutions Limited
Open Source Consultancy, Development & Training

Moose

- *A complete modern object system for Perl 5*
- Based on experiments with Perl 6 object model
- Built on top of Class::MOP
  - MOP - Meta Object Protocol
  - Set of abstractions for components of an object system
  - Classes, Objects, Methods, Attributes
- An example might help

Moose Example

-
  ```
  package Point;
  use Moose;

  has 'x' => (isa => 'Int',
              is  => 'ro');
  has 'y' => (isa => 'Int',
              is  => 'rw');

  sub clear {
    my $self = shift;
    $self->{x} = 0;
    $self->y(0);
  }
  ```

Understanding Moose

- There's a lot going on here
- `use Moose`
  - Loads Moose environment
  - Makes our class a subclass of Moose::Object
  - Turns on strict and warnings

Creating Attributes

-
  ```
  has 'x' => (isa => 'Int',
              is  => 'ro')
  ```
  - Creates an attribute called 'x'
  - Constrainted to be an integer
  - Read-only accessor

-
  ```
  has 'y' => (isa => 'Int',
              is  => 'rw')
  ```

Defining Methods

-
  ```
  sub clear {
    my $self = shift;
    $self->{x} = 0;
    $self->y(0);
  }
  ```
- Standard method syntax
- Uses generated method to set y
- Direct hash access for x

Subclassing

-
  ```
  package Point3D;
  use Moose;

  extends 'Point';

  has 'z' => (isa => 'Int');

  after 'clear' => sub {
    my $self = shift;
    $self->{z} = 0;
  };
  ```

Subclasses

- `extends 'Point'`
  - Similar to use base
  - Overwrites @ISA instead of appending
- `has 'z' => (isa = 'Int')`
  - Adds new attribute 'z'
  - No accessor function - private attribute

Extending Methods

-
  ```
  after 'clear' => sub {
    my $self = shift;
    $self->{z} = 0;
  };
  ```
- New clear method for subclass
- Called after method for superclass
- Cleaner than $self->SUPER::clear()

Creating Objects

- Moose classes are used just like any other Perl class

-
  ```
  $point = Point->new(x => 1, y => 2);
  ```

-
  ```
  $p3d   = Point3D->new(x => 1,
                        y => 2,
                        z => 3);
  ```

More About Attributes

- Use the `has` keyword to define your class's attributes
- `has 'first_name' => ( is => 'rw' );`
- Use `is` to define rw or ro
- Omitting `is` gives an attribute with no accessors

Getting & Setting

- By default each attribute creates a method of the same name.
- Used for both getting and setting the attribute
- `$dave->first_name('Dave');`
- `say $dave->first_name;`

Change Accessor Name

- Change accessor names using reader and writer

-
  ```
  has 'name' => (
    is => 'rw',
    reader => 'get_name',
    writer => 'set_name',
  );
  ```
- See also MooseX::FollowPBP

Required Attributes

- By default Moose class attributes are optional
- Change this with `required`

-
  ```
  has 'name' => (
          is       => 'ro',
          required => 1,
  );
  ```
- Forces constructor to expect a name
- Although that name could be undef

Attribute Defaults

- Set a default value for an attribute with default

-
  ```
  has 'size' => (
      is          => 'rw',
      default     => 'medium',
  );
  ```
- Can use a subroutine reference

-
  ```
  has 'size' => (
      is          => 'rw',
      default     => \&rand_size,
  );
  ```

More Attribute Properties

- `lazy`
  - Only populate attribute when queried
- `trigger`
  - Subroutine called after the attribute is set
- `isa`
  - Set the type of an attribute
- Many more

More Moose

- Many more options
- Support for concepts like delegation and roles
- Powerful plugin support
  - MooseX::*
- Lots of work going on in this area

Templating

Templating

- Many people use templates to produce web pages
- Advantages are well known
- Standard look and feel (static/dynamic)
- Reusable components
- Separation of code logic from display logic
- Different skill-sets (HTML vs Perl)

Non-Web Templating

- The same advantages apply to non-web areas
- Reports
- Business documents
- Configuration files
- Anywhere you produce output

DIY Templating

- Must be easy - so many people do it
- See perlfaq4
- How can I expand variables in text strings?

DIY Templating

-
  ```
  $text =
  'this has a $foo in it and a $bar';

  %user_defs = (
    foo  => 23,
    bar  => 19,
  );

  $text =~ s/\$(\w+)/$user_defs{$1}/g;
  ```
- Don't do that

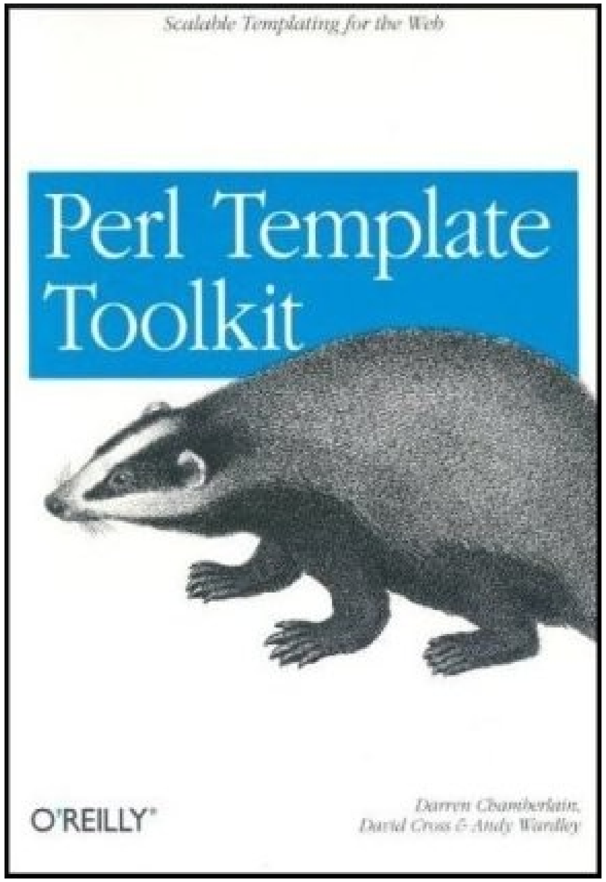
Templating Options

- Dozens of template modules on CPAN
- Text::Template, HTML::Template, Mason, Template Toolkit
- Many, many more
- Questions to consider
  - HTML only?
  - Template language
- I choose the Template Toolkit

Template Toolkit

- http://tt2.org/
- Very powerful
- Both web and non-web
- Simple template language
- Plugins give access to much of CPAN
- Can use Perl code if you want
  - But don't do that

Good Book Too!


Scalable Templating for the Web
Perl Template Toolkit
O'REILLY
Darren Chamberlain, David Cross & Andy Wardley

The Template Equation

- Data + Template = Output
- Data + Alternative Template = Alternative Output
- Different views of the same data
- Only the template changes

Simple TT Example

-
  ```
  use Template;
  use My::Object;
  my ($id, $format) = @ARGV;
  $format ||= 'html';
  my $obj = My::Object->new($id)
    or die;
  my $tt  = Template->new;
  $tt->process("$format.tt",
               { obj => $obj },
               "$id.$format")
    or die $tt->error;
  ```

html.tt

```
• <html>
    <head>
      <title>[% obj.name %]</title>
    </head>
    <body>
      <h1>[% obj.name %]<h1>
      <p><img src=“[% obj.img %]” /><br />
      [% obj.desc %]</p>
      <ul>
      [% FOREACH child IN obj.children -%]
        <li>[% child.name %]</li>
      [% END %]
    </body>
  </html>
```

text.tt

-
  ```
  [% obj.name | upper %]

  Image: [% obj.img %]
  [% obj.desc %]

  [% FOREACH child IN obj.children -%]
    * [% child.name %]
  [% END %]
  ```

Adding New Formats

- No new code required
- Just add new output template
- Perl programmer need not be involved

Equation Revisited

- Data + Template = Output
  - Template Toolkit
- Template + Output = Data
  - Template::Extract
- Data + Output = Template
  - Template::Generate

Catalyst

UKUUG
26th February 2009

MVC Frameworks

- MVC frameworks are a popular way to write applications
  - Particularly web applications
- Model
  - Data storage & data access
- View
  - Data presentation layer
- Controller
  - Business logic to glue it all together

MVC Examples

- Ruby on Rails
- Django (Python)
- Struts (Java)
- CakePHP
- Many examples in most languages
- Perl has many options

MVC in Perl

- Maypole
  - The original Perl MVC framework
- CGI::Application
  - Simple MVC for CGI programming
- Jifty
  - Developed and used by Best Practical
- Catalyst
  - Currently the popular choice

Catalyst

- MVC framework in Perl
- Building on other heavily-used tools
- Model uses DBIx::Class
- View uses Template Toolkit
- These are just defaults
- Can use anything you want

Simple Catalyst App

- Assume we already have model
  - CD database from DBIx::Class section
- Use `catalyst.pl` to create project

-
  ```
  $ catalyst.pl CD
  created "CD"
  created "CD/script"
  created "CD/lib"
  created "CD/root"
  ... many more ...
  ```

What Just Happened?

- Catalyst just generated a lot of useful stuff for us
- Test web servers
  - Standalone and FastCGI
- Configuration files
- Test stubs
- Helpers for creating models, views and controllers

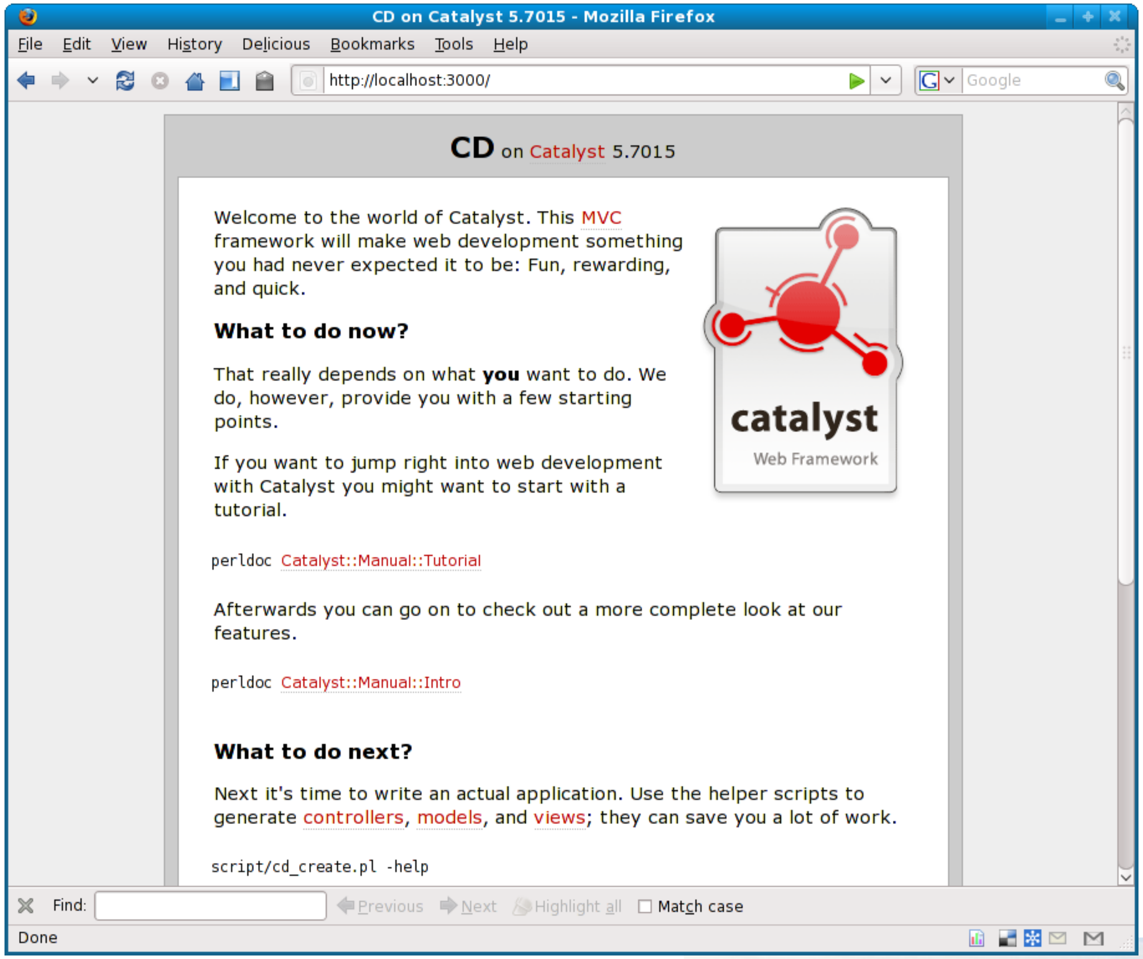
A Working Application

- We already have a working application
-
```
$ CD/script/cd_server.pl

... lots of output

[info] CD powered by Catalyst 5.7015
You can connect to your server at
http://localhost:3000
```
- Of course, it doesn't do much yet

Simple Catalyst App


CD on Catalyst 5.7015 - Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
http://localhost:3000/
Google

CD on Catalyst 5.7015

Welcome to the world of Catalyst. This MVC framework will make web development something you had never expected it to be: Fun, rewarding, and quick.

What to do now?

That really depends on what **you** want to do. We do, however, provide you with a few starting points.

If you want to jump right into web development with Catalyst you might want to start with a tutorial.

```
perldoc Catalyst::Manual::Tutorial
```

Afterwards you can go on to check out a more complete look at our features.

```
perldoc Catalyst::Manual::Intro
```

What to do next?

Next it's time to write an actual application. Use the helper scripts to generate controllers, models, and views; they can save you a lot of work.

```
script/cd_create.pl -help
```

Next Steps

- Use various helper programs to create models and views for your application
- Write controller code to tie it all together
- Many plugins to handle various parts of the process
  - Authentication
  - URL resolution
  - Session handling
  - etc...

Create a View

-
  ```
  $ script/cd_create.pl view Default TT
   exists "/home/dave/training/cdlib/CD/script/../
  lib/CD/View"
   exists "/home/dave/training/cdlib/CD/script/../
  t"
  created "/home/dave/training/cdlib/CD/script/../
  lib/CD/View/Default.pm"
  created "/home/dave/training/cdlib/CD/script/../
  t/view_Default.t"
  ```

Remove Default Message

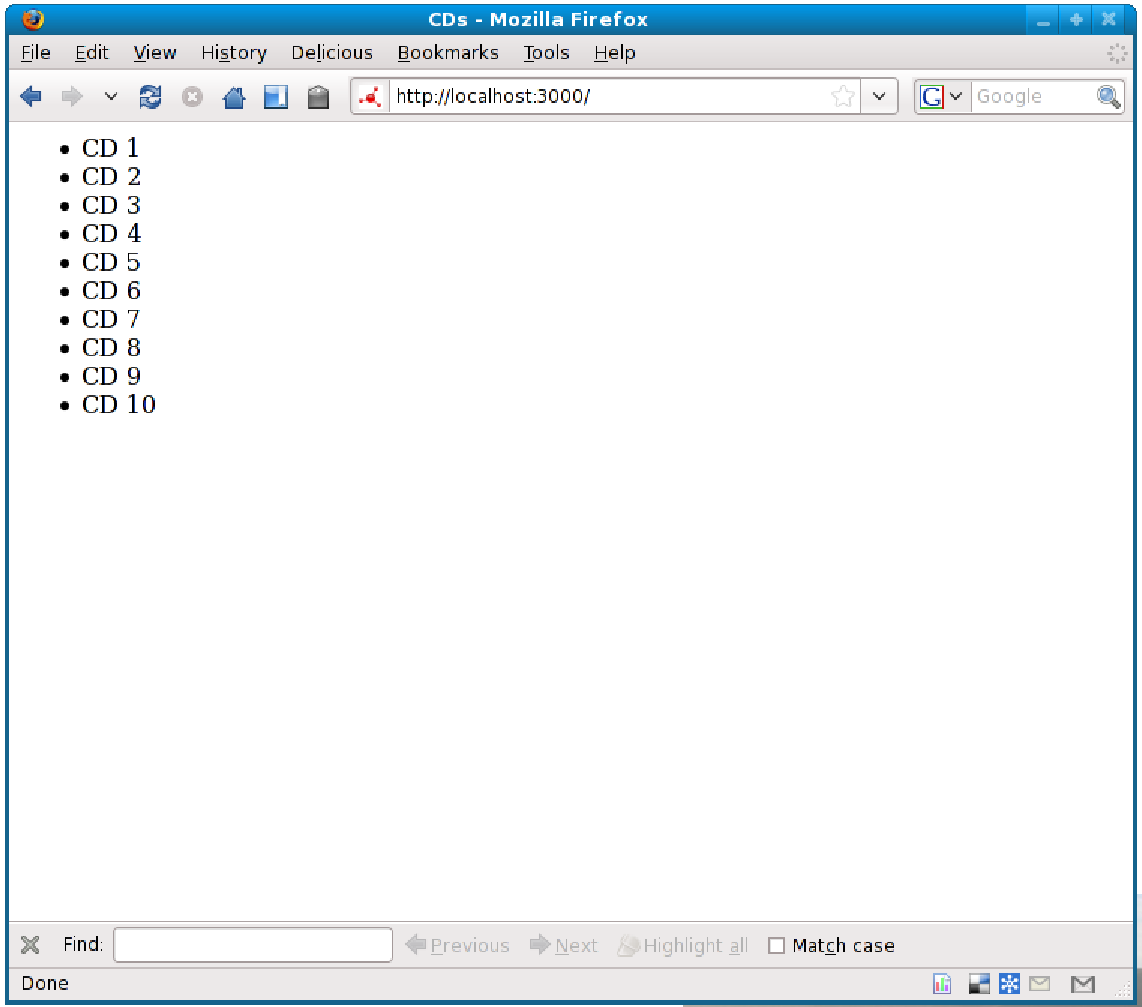
- In lib/CD/Controller/Root.pm
-
```
sub index :Path :Args(0) {
    my ( $self, $c ) = @_;

    # Hello World
    $c->response_body($c->welcome_message);
}
```
- Remove response_body line
- Default behaviour is to render index.tt
- Need to create that

index.tt

- root/index.tt
-
```
<html>
  <head>
    <title>CDs</title>
  </head>
  <body>
    <ul>
[% FOREACH cd IN [ 1 .. 10 ] %]
      <li>CD [% cd %]</li>
[% END %]
    </ul>
  </body>
</html>
```

New Front Page



- CD 1
- CD 2
- CD 3
- CD 4
- CD 5
- CD 6
- CD 7
- CD 8
- CD 9
- CD 10

Adding Data

- Of course that's hard-coded data
- Need to add a model class
- And then more views
- And some controllers
- There's a lot to do
- I recommend working through a tutorial

Easier Catalyst

- A lot of web applications do similar things
- Given a database
- Produce screens to edit the data
- Surely most of this can be automated
- It's called CatalystX::ListFramework::Builder
- (Demo)

CatX::LFBuilder

- Does a lot of work
- On the fly
- For every request
- No security on table updates
- So it's not right for every project
- Very impressive though

Conclusions

- There's a lot to bear in mind when writing a web app
- Using the right framework can help
- Catalyst is the most popular Perl framework
- As powerful as any other framework
  - In any language
- Lots of work still going on
- Large team, active development

Further Information

Further Information

- Some suggestions for places to go for further information
- Web sites
- Books
- Magazines
- Mailing lists
- Conferences

London Perl Mongers

- http://london.pm.org/
- Mailing list
- Regular meetings
  - Both social and technical
- London Perl Workshop
- Many other local Perl Monger groups
  - http://pm.org/

Web Sites

- use Perl;
  - Perl news site
  - Also journals
- perl.com
  - O'Reilly run site
  - High quality articles

Web Sites

- Perl Monks
  - Best web site for Perl questions
  - Many Perl experts
- The Perl directory
  - http://perl.org/
  - Lists of many Perl-related sites

Books

- Some recent Perl books
- *Perl Best Practices* - Damian Conway
- *Advanced Perl Programming* - Simon Cozens
- *Perl Hacks* - chromatic, Conway & Poe
- *Intermediate Perl* - Schwartz, foy & Phoenix
- *Mastering Perl* - brian d foy

More Books

- *Higher Order Perl* - Mark-Jason Dominus
- *Minimal Perl* - Tim Maher
- *Pro Perl Debugging* - Richard Foley & Joe McMahon
- *Perl & LWP* - Sean M Burke
  - Updated online edition
  - http://lwp.interglacial.com/
- See http://books.perl.org/

Magazines

- The Perl Review
  - http://www.theperlreview.com/
- Randal's monthly columns
  - Linux Magazine
  - SysAdmin

Mailing Lists

- Many mailing lists devoted to Perl topics
- See http://lists.cpan.org/

Conferences

- The Open Source Convention
  - San Diego 20-24 July 2009
- YAPC
  - Pittsburgh 22-24 June 2009
  - Lisbon 3-5 August 2009
  - Brazil, Asia, Israel, Australia
- One-Day Perl Workshops
- See http://yapc.org/

That's all folks

- Any questions?