Class::Multimethods and WWW::Mechanize

Abram Hindle

Victoria Perl Mongers

abez@abez.ca

January 20, 2003

Class::Multimethods

- What does Class Multimethods do?
  - Provide Dynamic Dispatch to Perl methods.
  - Provide overloading to Perl functions and methods.
  - Provice some form of easy type checking on functions.

Why

- We can already do the type checking ourselves when we want to! Why use Class::Multimethods?
  - More maintainable.
  - Better code reuse.
  - Only functions and method we explicitly create with Class::Multimethods will use it.
  - Handles Recursion.

Multi-Dispatch

- Multi-dispatch is when at runtime a function call or method call is matched to the arguments being passed. For instance we have a subroutine and we want to it act differently based on the type of arguments passed to it, maybe we have a display function and if a window instead of a graphical context is sent to that function we have run some extra code. Multi-dispatch enables us to handle these situations while still allowing for code reuse.

Multi-Dispatch

- Our problem: We want to make a method which prints references out in a special way. An object could potentially be printed as well.
  - Solution 1: Have a big if else block or a hash with subroutine values to choose which chunk of code to run to print a reference.
  - Solution 2: Use Class::Multimethods to make multiple subroutines for each case of reference we care about.
- Solution 1 works but we'd have to write the code and it's not very maintainable. What if more Classes come along which have special ways of printing?
- Solution 2 allows for maintainability and code reuse. It also handles recursion well. The only catch is it could be slower depending on how well you're acquainted with Perl.
- Essentially this example is similar to overloading.

types.pl

```
use Class::Multimethods;
sub printRef {
        if (ref($_[0]) eq 'ARRAY') {
                print join(",",@$_[0]),"\n";
        } elsif (ref($_[0]) eq 'HASH') {
                my @out = ();
                while (my ($key,$val) = each %$_[0]) {
                        push @out, "$key => $val";
                }
                printRef(\@out);
        }
}
multimethod printRefM => (HASH) => sub {
                my @out = ();
                while (my ($key,$val) = each %$_[0]) {
                        push @out, "$key => $val";
                }
                printRefM(\@out);
        };
multimethod printRefM => (ARRAY) => sub {
                print join(",",@$_[0]),"\n";
        };
```

Overloading

- In statically typed languages it's common to have functions like:
  - connectTo(int ip);
  - connectTo(String ip);
- In Perl you have to do the type checking at the start of the subroutine but with Class::Multimethods:

```
multimethod connectTo => ('#') => sub {
        net_connect($_[0]);
}
multimethod connectTo => ('$') => sub {
        net_connect(convertToIp($_[0]));
}
```

- It's EASY! # means numeric scalar while $ means non numeric scalar!

Equality.pm

```
package Equality;
use Class::Multimethods qw(equals);
sub new {
        my $type = shift;
        $type = ref($type) || $type;
        my $self = {};
        bless $self,$type;
        return $self;
}
multimethod equals =>  (Equality,Equality) =>
                sub { return 0; };
resolve_no_match equals => sub {
                return 0;
        };
1;
```

Circle.pm

```
package Circle;
use Class::Multimethods;
use base qw(Equality);
sub new {
        my $type = shift;
        my $self = {};
        bless $self,$type;
        my %args = @_;
        $self->{radius} = $args{radius} || 1;
        return $self;
}
multimethod equals => (Circle,Circle) =>
        sub {
                return ($_[0]->{radius} == $_[1]->{radius});
        };
multimethod equals => (Circle,Oval) =>
        sub {
                return ($_[1]->{radiusx} == $_[1]->{radiusy} && $_[0]->{
                     radius} == $_[1]->{radiusx});
        };
1;
```

Oval.pm

```
package Oval;
use Class::Multimethods;
use base qw(Equality);
sub new {
        my $type = shift;
        my $self = {};
        bless $self,$type;
        my %args = @_;
        $self->{radiusx} = $args{radiusx} || 2;
        $self->{radiusy} = $args{radiusy} || 1;
        return $self;
}
multimethod equals => (Oval,Oval) => sub {
                return ($_[0]->{radiusx} == $_[1]->{radiusx}) &&
                        ($_[0]->{radiusy} == $_[1]->{radiusy});
        };
multimethod equals => (Oval,Circle) => sub { return $_[1]->equals($_
     [0]); };
1;
```

multimethod.pl

```
#!/usr/bin/perl
#use Contrived; #j/k
use Circle;
use Oval;
use Equality;
my $other  = Equality->new();
my %hash = (    "CircleEQ"=> Circle->new(radius=>4),
                "CircleNE"=> Circle->new(radius=>3.5),
                "Ovaleq"  => Oval->new(radiusx=>4,radiusy=>4),
                "Ovalne"  => Oval->new(radiusx=>4,radiusy=>3.5),
                "Other"   => $other,
);
foreach my $key1 (keys %hash) {
        foreach my $key2 (keys %hash) {
                print   "$key1 is ",
                        ($hash{$key1}->equals($hash{$key2}))?"equal":"not
                             equal",
                        " to $key2\n";
        }
};1;
```

Produces:

```
Other is not equal to Other
Other is not equal to CircleNE
Other is not equal to Ovalne
Other is not equal to CircleEQ
Other is not equal to Ovaleq
CircleNE is not equal to Other
CircleNE is equal to CircleNE
CircleNE is not equal to Ovalne
CircleNE is not equal to CircleEQ
CircleNE is not equal to Ovaleq
Ovalne is not equal to Other
Ovalne is not equal to CircleNE
Ovalne is equal to Ovalne
Ovalne is not equal to CircleEQ
Ovalne is not equal to Ovaleq
CircleEQ is not equal to Other
CircleEQ is not equal to CircleNE
CircleEQ is not equal to Ovalne
CircleEQ is equal to CircleEQ
CircleEQ is equal to Ovaleq
Ovaleq is not equal to Other
Ovaleq is not equal to CircleNE
Ovaleq is not equal to Ovalne
Ovaleq is equal to CircleEQ
Ovaleq is equal to Ovaleq
```

Conclusions

- Class::Multimethods provides Perl Users with easy to use Overloading and Multi-Dispatch methods.
- Reduces maintenance time.
- Class::Multimethods tries to get a closest fit which makes sense.
- You don’t have to use it.
- Checkout: http://search.cpan.org/src/DCONWAY/Class-Multimethods-1.70/tutorial.html
- (Sit down we still have WWW::Mechanize to go!)

WWW::Mechanize

- What does WWW::Mechanize do?
  - Automates interactions with Websites
  - Abstract forms on a website for you.
  - Submit forms for your scripts.
  - Maintain cookies.
  - Acts like a scriptable browser.

What?

- What is WWW::Mechanize?
  - Child of LWP::UserAgent
  - Add-on for Libwww-perl
  - Derivative of WWW::Chat
  - Your ticket to website crawling

Example 1

- Lets go to Google and click on the first link that matches cpan when we search for ”WWW::Mechanize”.
  - We see there is only 1 form on Google.
  - We see the text-field is called ”q”.

FormatHTML.pm

```
package FormatHTML;
use Exporter;
use HTML::FormatText;
use HTML::TreeBuilder;
@ISA = qw(Exporter);
@EXPORT = qw(formatHTML);
sub formatHTML {
        $tree = HTML::TreeBuilder->new->parse($_[0]);
        $formatter = HTML::FormatText->new(leftmargin => 0, rightmargin
              => 80);
        my $str = $formatter->format($tree);
        $str =~ s/\n\n/\n/g;
        return $str;
}
1;
```

mech1.pl

```
use WWW::Mechanize;
use FormatHTML;
my $agent = WWW::Mechanize->new();
$agent->get("http://www.google.ca");
$agent->form(1); #use the first form
$agent->field("q","WWW::Mechanize");
$agent->click;
$agent->follow("cpan");
print formatHTML($agent->{content}),"\n";
```

Produces:

```
The CPAN Search Site
Home Authors Recent About Mirrors FAQ Feedback
in
Andy Lester > WWW-Mechanize-0.32

WWW-Mechanize-0.32
==================
This Release
WWW-Mechanize-0.32
 [Download] [Browse]
23 Oct 2002
Latest Release
WWW-Mechanize-0.33
 [Download] [Browse]
16 Jan 2003
Other Releases
Links
[ CPAN Testers ] [ CPAN Request Tracker ]
Special Files
Changes
MANIFEST
README

Modules
-------
WWW::Mechanize
automate interaction with websites
0.32
```

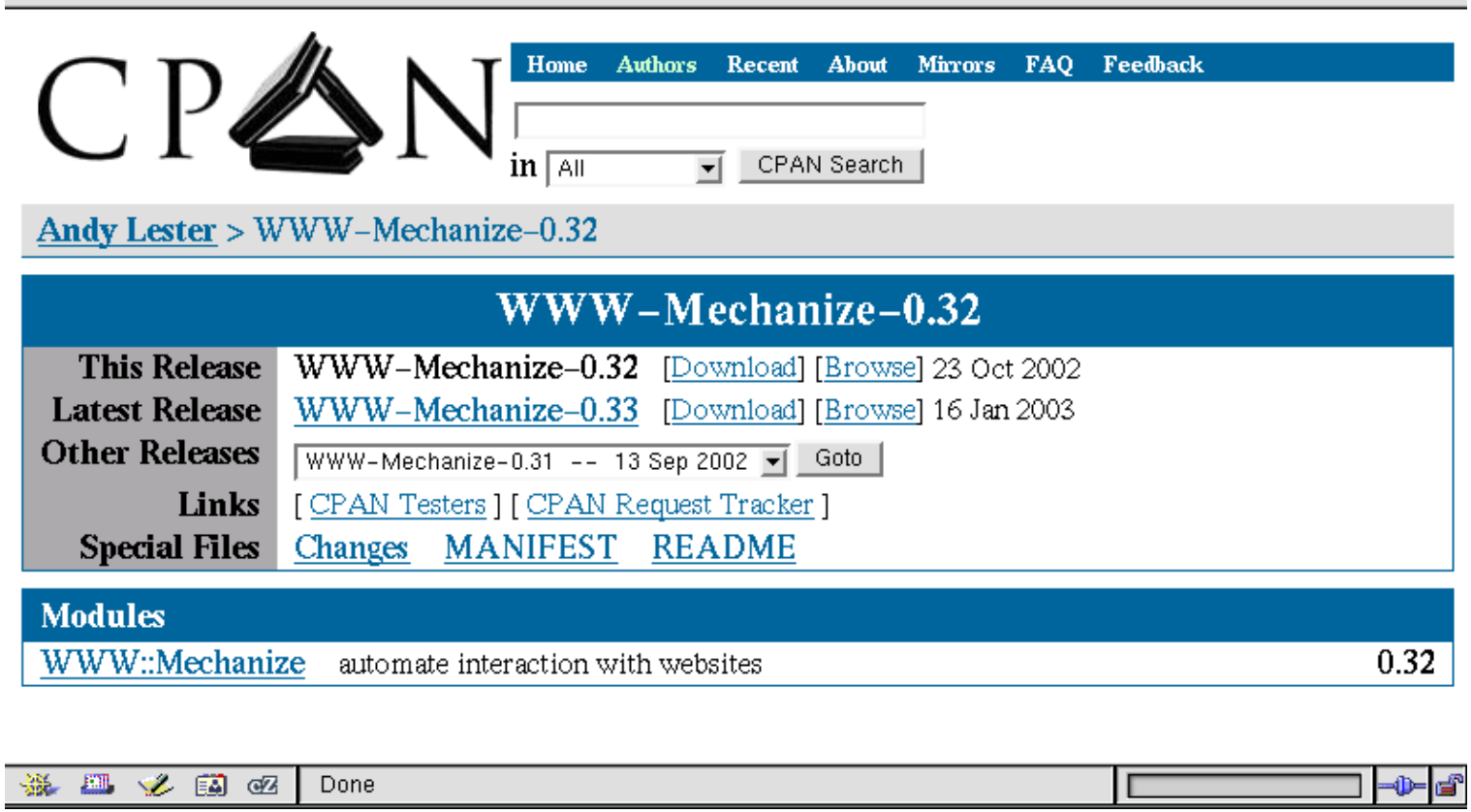

Figure 1: Mozilla Rendered Example

Review Example ”mech1.pl”

- WWW::Mechanize is object oriented, you must create a new WWW::Mechanize agent to use it.
- To get a webpage you can use the get($url) method.
- Forms are selected by order of appearance starting at 1.
- Fields have to be referenced by name
- Method ”click” will click the buttons on the form. If there is only one button just one call to click without arguments will do.
- Method ”follow” follows the first link on the page to match the arguments given.

Example 2

- Let’s query cpan for module documentation!

- **searchCPAN.pl**

```
use WWW::Mechanize;
use FormatHTML;
my $agent = WWW::Mechanize->new();
my $search = shift || "WWW::Mechanize";
$agent->get("http://search.cpan.org/");
$agent->form(1); #use the first form
$agent->field("query",$search);
$agent->click;
$agent->follow($search);
print formatHTML($agent->{content}),"\n";
```

More WWW::Mechanize

- **follow** also accepts a integer as an argument, it will click on the nth link other it will click on the link which matches the 1st argument.
- `$agent->{content}`

  gets the content of the last page visited.
- **back** method acts like a browser back button. Goes to the previous page visited (not the first page).
- **extract_links** method extracts all the links from a page (returns a list of links which are 3-tuples

  `[[$destination,$text,$name],...]`

Attribute	Comment	Type
uri	The current URI	
req	The current request object	[HTTP::Request]
res	The response received	[HTTP::Response]
status	The status code of the response	
ct	The content type of the response	
base	The base URI for current response	
content	The content of the response	
forms	Array of forms found in content	[HTML::Form]
form	Current form	[HTML::Form]
links	Array of links found in content	

Table 1: WWW::Mechanize's get method sets the above attributes per call

Forms

- Forms are already parsed for you.
- If you don't know the name of the first input on the webpage you can use the HTML::Form object to get the input name.

```
sub getFirstInputName {
        my $agent = shift;
        my $form = $agent->{form};
        foreach my $input ($form->inputs) {
                if ($input->type =~ /text/) {
                        return $input->name;
                }
        }
        return undef;
}
```

Tips

- It's often easier to parse HTML with all the tags removed e.g. use my formatHTML command
- To get cookies to work you have to set a cookie-jar:
```
$agent->cookie_jar({file=>"$ENV{HOME}/.cookies.txt"});
```
- You can set the name of the user agent passed to server easily:
```
$agent->agent(["Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.3a) Gecko
    /20021126"]);
```
- Since a WWW::Mechanize object is a LWP::UserAgent you can do anything a LWP::UserAgent can! So if you need to post to a form and you know the parameters, just do it!
```
use HTTP::Request;
use HTTP::Request::Common;
use LWP::UserAgent;
$ua = LWP::UserAgent->new;
$r = $ua->request(POST "http://www.allmusic.com/cg/amg.dll",[ P=>"amg"
    , sql=>"air", opt1=>"1", ]);
print $r->content;
```

References

- I recommend getting help from these resources:
  - perldoc WWW::Mechanize
  - perldoc LWP::UserAgent
  - perldoc HTTP::Request::Common
  - perldoc LWP
  - perldoc Class::Multimethods
  - CPAN (of course)
  - Conway, Damian
    http://search.cpan.org/src/DCONWAY/Class-Multimethods-1.70/tutorial.html