Data::Domain

a data validation tool

laurent.dami@justice.ge.ch

What is a "data domain" ?

- term from data management
  - a set of values
    - →may be infinite
  - defined by extension (enumeration) or by intension (set of rules)

How to work with domains ?

- To put the definition into practice, we need to be able to :
  - Define a domain
    - → atomic building blocks (mostly scalar)
    - → composition operators
  - Check if a value belongs to a domain
    - → if not : explain WHY
    - → answers should be consistent over time

      validating a withdrawal from an account is not a domain operation

Why data domains ?

- when some data crosses "boundaries"
  - user form
  - database
  - parse tree
  - config. file
  - function call
- principle of defensive programming

CPAN : many modules

- Parameter checking
  → Params::Check, Params::Validate
- Data Modelling & Object-Relational Maps
  →Jifty::DBI, Alzabo, Rose::DB::Object, DBIx::Class
- HTML Form tools
  →CGI::FormBuilder, Data::FormValidator
- Business rules
  → Brick, Declare::Constraints::Simple, Data::Constraint

Terminology : "domain" often called "template" or "profile"

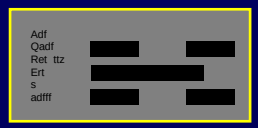
Other technologies

- database validation mechanisms
  - → reference table
  - → rules & constraints
  - → triggers
- typing (strong / dynamic)
- XML schema
- Javascript frameworks
- Parsers
- ...

Some design dimensions

- shape of data
  - scalar (string, num, date, ...)
  - array or hash
  - multi-level tree
  - objects
- shape of messages
  - single scalar
  - collection
  - multi-level tree
- Conciseness (declarative style)
- Expressiveness
- Internal dependencies (i.e. begin_date / end_date)

Scenario



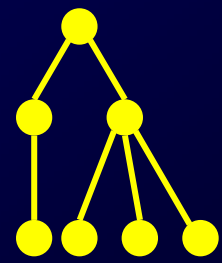
HTML form

Ajax submit

Key	Value
Foo.1.bar	
Foo.2.buz	

table

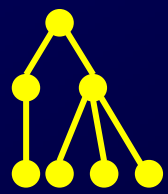
CGI::Expand



data tree

Data::
Domain::
inspect

[invalid]



JSON error messages

decorate

[valid]

process form
(➔DBIx::DataModel)

see video

Synopsis

```
my $domain = Struct(
  anInt       => Int (-min => 3,     -max => 18),
  aNum        => Num (-min => 3.33, -max => 18.5),
  aDate       => Date(-max => 'today'),
  aLaterDate => sub {
    my $context = shift;
    Date(-min => $context->{flat}{aDate})
  },
  aString     => String(-min_length => 2,
                        -optional   => 1),
  anEnum      => Enum(qw/foo bar buz/),
  anIntList   => List(-min_size => 1, -all => Int),
  aMixedList => List(Integer, String, Date),
);

my $messages = $domain->inspect($some_data);
display_error($messages) if $messages;
```

Design principles

- Do One Thing Well : just check
  - → no HTML form generation
  - → no Database schema generation
  - → no data modification (filtering, canonic form)
- return informative messages
- concise yet expressive
- extensible (OO inheritance)

Domain creation

- Object-oriented

```
my $dom = Data::Domain::String->new(
    -min        => "aaa",
    -max_length => 8,
    -regex      => qr/foo|bar/,
);
```

- Functional shortcuts

```
my $dom = String(-min => "aaa", ...);
```

- Default argument for each domain constructor

```
my $dom = String(qr/foo|bar/); # default is -regex
```

- Arguments add up constraints as "and"

Generic arguments

- -optional
  → if true, an undef value is accepted
- - name
  → name to be returned in error messages
- - messages
  → ad hoc error messages for that domain

Builtin scalar domains

- Whatever (-defined, -true, -isa, -can)
- Num, Int (-min, -max, -range, -not_in)
- Date, Time(-min, -max, -range)
- String (-regex, -antiregex, -min, -max, -range, -min_length, -max_length, -not_in)
- Enum (-values)

Builtin structured domains

- List (-items, -min_size, -max_size, -all, -any)
- Struct (-fields, -exclude)
- One_of (-options)

Example

```
use Regexp::Common;

sub Name {
  return String(-regex     => qr/^[-. [:alpha:]]+/,
                -antiregex => qr/$RE{profanity}/,
                @_)
}

my $person_dom = Struct(
      lastname  => Name,
      firstname => Name(-optional => 1),
      d_birth   => Date(-optional => 1,
                        -max      => 'today'),
);
```

Lazy Domains

- Principle
  - a coderef that returns a domain *at the time it inspects a value*
  - can look at the surrounding context (subvalues seen so far)

```
my $person_dom = Struct(
  ...
  d_birth => Date(-optional => 1,
                  -max      => 'today'),
  d_death => sub {
     my $context = shift;
      return Date(-min => $context->{flat}{d_birth});
  },
);
```

(inspiration : Parse::RecDescent)

What is in the "context"

- root
  - →top of tree
- path
  - →sequence of keys or array indices to the current node
- list
  - →ref to last array visited while walking the tree
- flat
  - →flattened hash with all keys seen so far

Example : Contextual sets

```
my $some_cities = {
   Switzerland => [qw/Genève Lausanne Bern Zurich Bellinzona/],
   France      => [qw/Paris Lyon Marseille Lille Strasbourg/],
   Italy       => [qw/Milano Genova Livorno Roma Venezia/],
};
my $domain = Struct(
   country => Enum(keys %$some_cities),
   city    => sub {
      my $context = shift;
      my $country = $context->{flat}{country};
      return Enum(-values => $some_cities->{$country});
   },
 );
```

Example : Ordered list

```
my $domain = List(-all => sub {
   my $context = shift;
   my $index   = $context->{path}[-1];
   return Int if $index == 0; # first item
   my $min = $context->{list}[$index-1] + 1;
   return Int(-min => $min);
});
```

New Domain Constructors

- by wrapping

```
sub Phone {
  String(-regex => qr/^\+?[0-9() ]+$/,
         -messages => "Invalid phone number",
         @_)
}
```

- by subclassing
  - → implement new()
  - → implement _inspect()