PRAGMATIC PERL

30

08/2015

pragmaticperl.com

Pragmatic Perl 30

pragmaticperl.com

Выпуск 30. Август 2015

Другие выпуски и форматы журнала всегда можно загрузить с pragmaticperl.com. С вопросами и предложениями пишите на почту editor@pragmaticperl.com.

Комментарии к каждой статье есть в html-версии. Подписаться на новые выпуски можно по ссылке pragmaticperl.com/subscribe.

Авторы статей: Наталья Савенкова, Алексей Мележик, Дмитрий Шаматрин, Владимир Леттиев

Обложка: Марко Иванык

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2015-08-28 09:08

Оглавление

1. От редактора

Теперь можно в качестве благодарности отправлять пожертвования журналу. Исключительно по собственному желанию! http://pragmaticperl.com/donations/

Друзья, журнал ищет новых авторов. Не упускайте такой возможности! Если у вас есть идеи или желание помочь, пожалуйста, свяжитесь с нами.

Приятного чтения.

■ *Вячеслав Тихановский*

2. Отчет о конференции «Perl Mama» от организатора

25-26 июля в Одессе мы провели небольшой перловый воркшоп — PerlMama, с единой целью — провести перловое событие в Украине

YAPC было давно, OSDN свернули.

У меня не было опыта в проведении конференций, но зато был опыт в организации попоек. Многие видели отчет Радецкого о конференции. Решение не пиарить событие на всяких тематических ресурсов не случайно. Чем больше людей, тем больше вероятность получить эпический фейл. Место, в котором мы проводили конференцию в Одессе известное, но т.к я там ничего не проводил ранее, надо было проверить. Проверили. Спасибо администрации хаба, все прошло прекрасно, получили ценный опыт. Когда будем проводить еще что-то, будем проводить у них.

Теперь по поводу афтерпати. В афтерпати я был уверен на все 100%. Ну, т.е. был уверен что она пройдет хорошо. Так и получилось. «Легенда» — лучшее место, где можно пообедать или выпить пива в Одессе.

Доклады постарались сделать разноплановые, сильно сложные не включали. Получилось очень даже ок. Выяснилось, что не только я занимаюсь обучением перловиков по определенной программе, а еще и господа из PortaOne, программа которых несколько сложнее для учеников, но отнимает меньше времени преподавателя. Поговорили об ElectricCommander, программном продукте ElectricCloud, в котором в качестве DSL используется Perl, которые, к тому же, выступили в роли спонсора. Рассмотрели механизм прототипов и атрибутов, проблемные аспекты unit-и-не-только тестирования. Ну и напоследок поговорили о PearlPBX и о тенденциях в мире современной разработки под веб и не только.

Порадовали шнурки для бейджей, которые получил каждый участник, они оказались весьма и весьма неплохими по качеству, а самое главное это то, что производитель шнурков отечественный. И ни один шнурок из всех не пришел в негодность.

Из негатива:

Не получилось сделать видеофиксацию докладов. Камера, которая была на это рассчитана и куплена, приехала только в понедельник. На следующий день после окончания конференции. Зато теперь она есть. В следующий раз сделаем 4к видео. Браслеты. Браслеты, как оказалось, я не шучу, XXL-размера, судя по всему, «на осень бальсой лука китайса», но при этом еле налезли на руку. Учтем тоже в следующий раз.

И вот что мы сделаем дальше. У нас есть идея и возможность реализовать гибридную конференцию, которая будет посвящена разработке вообще. Планируем начать с Ruby, Perl, Python, Erlang. И математики. Ориентировочно, если все глобально не поменяется, проведем конференцию в середине следующего июня.

За участие всем спасибо, считаю, что конференция удалась, а потому мы это повторим. И повторим не раз.

До новых встреч. Пишите письма.

■ *Дмитрий Шаматрин*

3. Работаем с legacy. Паттерн «извлечение функции» и оценка результатов

Из личного опыта рефакторинга устаревшего кода

Когда ты работаешь с унаследованным кодом, то постоянно возникают ситуации, когда просто не понимаешь, что происходит с данными. Да, ты понимаешь, что происходит что-то не то, но непонятно где. Была у меня ситуация, что в хеше что-то меняет значение, но просто беглым взглядом по коду найти это было невозможно. Эта история обогатила мой инструментарий работы с наследием, о чем я и расскажу в этой статье.

Допустим, у меня есть вот такой код, состоящий из скрипта и пары модулей:

example.pl

```
use Utils;
my $row = { user => { name => 'kate', age => 55 } };
if (check($row)) {
    say 'OK';
    # do something
}
else {
    say 'NO';
}
exit(0);
```

Utils.pm

```
package Utils;
use strict; use warnings;
use Exporter; our @ISA = qw(Exporter); our @EXPORT = qw(check);
use Uts;

sub check {
    my ($row) = shift;
    format_row($row->{user});
    return $row->{user}->{age} > 18 ? 1 : 0;
}
```

Uts.pm

```
package Uts;
use strict; use warnings;
```

```
use Exporter; our @ISA = qw(Exporter); our @EXPORT = qw(
    format_row);

my $modifiers = {
    age => 10,
    grade => 'high',
};
sub format_row {
    my ($row) = shift;
    #... много-много кода
    $row->{$_} = $modifiers->{$_} for keys %$modifiers;
    #... много-много кода
    return;
}
```

Конечно, как и всегда, код пришлось придумать. А теперь представьте, что кроме этих маленьких функций там еще по 2-3К кода вокруг, а сами функции длиной по 500 строк. Но чтение такой статьи вряд ли бы доставило удовольствие, поэтому ограничимся таким простым примером.

И вот я запускаю свой код:

```
$ perl example.pl
NO
```

Вообще-то у меня в **$row** задано “**age => 55**” и я должна получить **OK**. Тут что-то явно не так. Сейчас всем нужно сделать вид, что этот код прочитать и понять невозможно =) Даже добавив **say Dumper $row**; вот так:

```
if (check($row)) {
    say 'OK';
    # do something
}
else {
    say 'NO';
}
say Dumper $row;
```

я вижу совсем не то, что ожидаю:

```
$VAR1 = {
          'user' => {
                      'age' => 10,
                      'name' => 'kate',
                      'grade' => 'high'
                    }
```

```
7         };
```

Посмотрим, что происходит со значением **$row->{user}->{age}**. Если бы у меня был объект класса **User**, у котого через аксессор меняются атрибуты, то все было бы куда проще. А здесь применим **tie** (http://perldoc.perl.org/perltie.html).

В этом примере я хочу отследить, что у меня происходит со скаляром. Значит мне нужен новый класс, который может быть использован **tie** для привязки к скаляру. Класс должен реализовывать методы: **TIESCALAR, STORE и FETCH**.

Вот так он выглядит: **user.pm**

```
1 package user;
2 use feature qw/say/;
3 use Carp qw/longmess/;
4 sub TIESCALAR {
5     my $class = shift;
6     return bless {}, $class;
7 }
8 sub STORE {
9     my $self = shift;
10    my $value = shift;
11    $self->{value} = $value;
12    say longmess('SET '.$value);
13 }
14 sub FETCH {
15    my $self = shift;
16    return $self->{value};
17 }
18 1;
```

Что здесь происходит: **tie** связывает указанную переменную с заданным объектом. В результате исходный код совершенно не меняется *(это огромный плюс)*, а у нас есть возможность сделать собственную реализацию методов установки и извлечения значения, добавив туда разный диагностический код.

Объект класса **user** это самый обычный объект, который будет хранить в себе значение переменной **$row->{user}->{age}** в своем атрибуте **value**. (Если этот код непонятен, тогда вам сюда http://perldoc.perl.org/perlobj.html)

Итак. Меня интересует, в каком же месте кода изменилось мое значение

$row->{user}->{age}. Я сделала класс **user**, который будет обеспечивать внутреннюю реализацию, а так же писать сообщения с информацией о том, в каком же месте кода моя переменная получает новое значение.

Carp::longmess - возвращает стек вызовов как строку.

Теперь осталось только подменить переменную объектом:

example.pl

```
use Utils;
use user;
my $row = { user => { name => 'kate', age => 55 } };
tie $row->{user}->{age}, 'user';
if (check($row)) {
...
```

Запускаем!

```
$ perl example.pl
SET 10 at Uts.pm line 13.
    Uts::format_row('HASH(0x84ecb8)') called at Utils.pm line 8
    Utils::check('HASH(0x86da78)') called at example.pl line 7

NO
```

Вот и наш виновник: **Uts.pm line 13** с чего-то вдруг установил значение 10!

В Uts.pm:13 находится вот это:

```
$row->{$_} = $modifiers->{$_} for keys %$modifiers;
```

Еть еще вот такой модуль на `Devel::Spy`, хотя он работает странновато иногда, и вообще хотелось бы больше примеров использования в документации.

Итак. У нас есть фрагмент кода, который нужно поправить. Давайте сделаем, чтобы он в цикле проверял - если ключ существует в хеше, то пропустить присвоение, а если нет - добавить в хеш. У нас вокруг этого фрагмента еще есть много-много кода-кода, поэтому применим паттерн **extract function** (Извлечение функции).

Применим автоматизацию. В perl это всегда очень забавно.

У нас есть строка кода, которую мы ходим превратить в функцию. Используем модуль **Devel::Refactor**.

```
use Devel::Refactor;
my $refactory = Devel::Refactor->new;
# описываем внутренности нашей будущей функции
my $old_code = <<'end';
$row->{$_} = $modifiers->{$_} for keys %$modifiers;
end

my ($call, $code) = $refactory->extract_subroutine('
   apply_modifiers', $old_code);
say ($call, $code);
```

И вот такой результат:

```
my () = apply_modifiers ($row, $_, $modifiers);
sub apply_modifiers {
    my $row = shift;
    my $_ = shift;
    my $modifiers = shift;

$row->{$_} = $modifiers->{$_} for keys %$modifiers;

    return ();
}
```

Конечно, не восторг, и в таком виде это не совсем подходит, но если внутри фрагмент кода обрабатывает 5-6 значений, то это сэкономит усилия.

Вопрос: зачем нужно было выносить фрагмент кода в отдельную функцию?

Ответ: рефакторинг наследия можно осуществлять двумя способами "Edit and pray" (редактируй и молись), или "Cover and modify" (Покрывай и меняй).

Мы идем по второму пути. После того, как код выделен в отдельную функцию, описываем для нее тесты, следуя методологии **TDD** (Как писать юнит-тесты уже, наверное, все знают, не будем отклоняться от темы), затем определяем новое более правильное поведение.

И вот так выглядит новый код:

```
sub format_row {
    my ($row) = shift;
    #... много-много кода
    apply_modifiers ($row, $modifiers);
```

```
    #... много-много кода
    return;
}

sub apply_modifiers {
    my $row = shift;
    my $modifiers = shift;
    for my $attr (keys %$modifiers) {
        next if exists $row->{ $attr };
        $row->{ $attr } = $modifiers->{ $attr };
    }
    return;
}
```

Она стала более читаемая, покрыта тестами, и дальнейшая модификация будет уже проще.

Но один тест все же будет. Даже не тест, а замер. Производительность важна, и если о ней не заботиться, то внезапно наступает момент, когда ее падение заметно для человека, хотя код развивался эволюционно.

Меряем:

```
use Benchmark qw(cmpthese);
use Uts;
my $modifiers = {
    age => 10,
    grade => 'high',
};

cmpthese(-1, {
    'old' => sub {
        my $row = { user => { name => 'kate', age => 55 } };
        $row->{$_} = $modifiers->{$_} for keys %$modifiers;
    },
    'new' => sub {
        my $row = { user => { name => 'kate', age => 55 } };
        Uts::apply_modifiers ($row, $modifiers);
    },
});
```

И результат:

```
$ perl bench.t
        Rate  new  old
new 590160/s   -- -18%
old 721308/s  22%   --
```

Выводы: наш код стал чище и работает правильнее, но за счет потери в производительности. Как решать эту проблему, чтобы снова не превратить код в нечитаемый кошмар, это уже другой вопрос.

Итог: всегда лучше иметь представление какую цену вы платите за улучшение кода. А вообще рефакторинг это всегда увлекательное приключение!

Весь код и текст можно найти на https://github.com/name2rnd/pragmatic.

■ *Наталья Савенкова*

4. SWAT — простое тестирование веб-приложений

Позволяет тестировать веб-приложения используя специально подготовленную иерархию файлов

Введение

SWAT — дословно, в переводе — простое тестирование web приложений.

SWAT состоит из двух частей — DSL для написания smoke тестов и консольного клиента для запуска самих тестов.

Результат или отчет SWAT-теста выводится в TAP формате, что дает возможность переносить результаты тестов в различные системы мониторинга.

Отличительными особенностями SWAT является простота использования и минимальный набор зависимостей для установки, из которых только curl, пожалуй, является условно непредустановленным для линукс систем.

SWAT был придуман как *практическая* утилита, которой смогут воспользоваться как системные администраторы, далекие от глубин языка Perl, так и собственно разработчики web приложений.

Итак начнем описание SWAT с самого важного его компонента, а именно языка DSL для написания тестов.

DSL

DSL SWAT в первом приближении очень прост, посудите сами:

Пусть у нас есть web приложение `myapp.com` с набором http ресурсов `/hello` и `/hello/world`, требуется проверить что соответсвующие запросы к ресурсам возвращают успешный HTTP-статус `200 OK`.

С помощью SWAT это делается так:

Создать структуру проекта приложения и описать ресурсы:

```
$ mkdir  myapp # создать директорию для проекта
$ mkdir myapp/hello # описать HTTP–ресурсы
$ mkdir myapp/hello/world
```

Определить ожидаемый контент:

```
$ echo 200 OK >> myapp/hello/get.txt
$ echo 200 OK >> myapp/hello/world/get.txt
```

Согласитесь, достаточно просто? Не трудно догадаться, что SWAT во время прогона тестов сделает два запроса `GET /hello` и `GET hello/world` и проверит, что вернулся статус 200 OK.

Вот как это будет выглядеть при запуске с консоли посредством консольного клиента SWAT:

```
$ swat ./myapp http://myapp.com
/home/vagrant/.swat/reports/http://myapp.com/hello/00.t ........
# start swat for http://myapp.com//hello | is swat package 0
# swat version v0.1.19 | debug 0 | try num 2 | ignore http errors
    0
ok 1 – successfull response from GET http://myapp.com/hello
# data file: /home/vagrant/.swat/reports/http://myapp.com//hello/
   content.GET.txt
ok 2 – GET /hello returns 200 OK
1..2
ok
/home/vagrant/.swat/reports/http://myapp.com/hello/world/00.t ..
# start swat for http://myapp.com//hello/world | is swat package
   0
# swat version v0.1.19 | debug 0 | try num 2 | ignore http errors
    0
ok 1 – successfull response from GET http://myapp.com/hello/world
# data file: /home/vagrant/.swat/reports/http://myapp.com//hello/
   world/content.GET.txt
ok 2 – GET /hello/world returns 200 OK
1..2
ok
All tests successful.
Files=2, Tests=4,  0 wallclock secs ( 0.02 usr  0.00 sys +  0.02
   cusr  0.00 csys =  0.04 CPU)
Result: PASS
```

Однако, при всей кажущейся простоте SWAT обладает достаточно мощными средствами для расширения.

Основой алгоритм достаточно прост. Определили ресурсы, определили, что должно вернуться в ответе и прогнали тесты.

Однако SWAT обладает дополнительными возможностями, о которых я расскажу далее:

- использование Perl-регекспов для валидации возвращаемого контента;
- динамическая генерация проверочных утверждений;
- изменение логики проверки посредством выполнения встраиваемого perl кода.

Итак, начнем с Perl-регекспов.

Использование Perl-регекспов

Помимо обычно plain текстовых проверок в SWAT можно задавать проверки с помощью регулярных выражений. Пусть у нас есть ресурс `/version` возвращающий версию приложения в виде `version Foo.Bar.Baz` , где каждый из элементов — целое число.

Используя SWAT и регекспы нетрудно написать тест для такого случая:

```
$ cat myapp/version/get.txt

200 OK
regexp: version (\d+\.\d+\.\d+)

$ swat myapp/ 127.0.0.1
/home/vagrant/.swat/reports/127.0.0.1/version/00.t ..
# start swat for 127.0.0.1//version | is swat package 0
# swat version v0.1.19 | debug 0 | try num 2 | ignore http errors
    0
ok 1 — successfull response from GET 127.0.0.1/version
# data file: /home/vagrant/.swat/reports/127.0.0.1//version/
   content.GET.txt
ok 2 — GET /version returns 200 OK
ok 3 — GET /version returns data matching version (\d+\.\d+\.\d+)
# line found: 1.2.4
1..3
ok
```

```
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.02 usr  0.00 sys +  0.01
   cusr  0.00 csys =  0.03 CPU)
Result: PASS
```

Ну с регекспами все понятно, они сильно облегчают жизни, но иногда хочется чего-то чуть более сложного, плавно переходим к динамической генерации проверочных утверждений.

Динамическая генерация проверочных утверждений

В самом простом случае SWAT получает проверочные утверждения, считывая их из текстового файла:

```
$ cat get.txt
FOO
BAR
BAZ
regexp: number d\+
```

При таком подходе проверочные данные детерминированы на этапе запуска тестов. А что если захочется генерить эти данные динамически? Тут нам помогут SWAT-генераторы.

SWAT-генераторы — это куски Perl-кода, которые возвращают ссылку на массив строк, *представляющих* проверочные утверждения. Поясню все, как обычно на примере:

Пусть у нас есть некий ресурс `GET /ABC` , возвращающий все буквы английского алфавита от A до Z :

```
$ GET /ABC

A
B
C
D
...
```

Давайте напишем для данного ресурса SWAT-тест. При решении «в лоб» можно просто перечислить все буквы английского алфавита в SWAT-файле,

но это согласитесь не очень элегантное решение. Можно также использовать Perl regexp и написать так:

```
regexp:  ^[A–Z]$
```

Уже лучше, а что если хочется соблюсти компактность кода, но при этом не мучаться с регекспами, которые в определенных случаях могут быть достаточно сложными? Давайте просто сгенерим массив проверочных утверждений:

```
$ cat ABC/get.txt

generator:  [ ('A' .. 'Z') ]
```

Это и есть SWAT-генератор. Хочется отметить, что данный пример тривиален, но ничто не останавливает вас от использования *любого валидного* Perl-кода для генерации SWAT-тестов. Пусть, мы хотим проверить что отданный нам контент - список пользователей содержится в некоторой sqlite-базе, тогда можно написать так:

```
$ cat USERS/get.txt

use DBI; \
my $dbh = DBI–>connect("dbi:SQLite:dbname=/app/data/users.db","",
    ""); \
my $sth = $dbh–>prepare("SELECT name from users"); \
$sth–>execute(); \
my $results = $sth–>fetchall_arrayref; \
[ map { $_–>[0] } @${results} ]
```

Заметили символы `\` в приведенном примере? В SWAT разрешены многострочные выражения при написании генераторов и Perl-выражений.

О Perl-выражениях и управлении проверочной логикой речь пойдет далее.

Perl-выражения и управление проверочной логикой

Perl-выражения в SWAT это просто вставки Perl-кода, которые вы можете использовать в тексте SWAT-тестов. Данный код будет выполнен в режиме `< Perl evaled string>` во время прогона тестов, приведу сначала абстрактный пример:

```
HELLO
code: print "THIS IS HELLO WORLD"
WORLD
```

Смысла в приведенном коде никакого он просто выведет на консоль строчку “THIS IS HELLO WORLD”, никак не повлияв на логику теста — будут выполнены проверки на наличие слов *HELLO* а затем *WORLD* .

Что бы понять чем же Perl-выражения могут быть полезны в SWAT, необходимо рассказать как проверочные утверждения *будут выглядеть* после обработки SWAT парсером.

Дело в том, что после парсинга, каждое SWAT-утверждение превращается в соответствующий *assert* модуля `Test::More`. Каждый, кто хоть раз писал юнит-тесты для собственных Perl-модулей, знаком с данным инструментом.

После парсинга SWAT-теста мы имеем дело по сути с тестом в формате `Test::More`, а значит мы можем использовать функции данного модуля для управления логикой теста, приведу показательные примеры:

```
RED
GREEN
code: skip("skip next check",1) # пропустить следующую проверку
   для цвета BLUE
BLUE
YELLOW
PUPRPLE
```

```
ONE
TWO
code: \
if ( $ENV{SKIP_COLORS} ) {    # пропустить следующие три проверки
    \
                              # если задана переменная окружения
                                 skip_colors \
    skip( "skip next check", 3 ) \;
}
BLUE
YELLOW
PUPRPLE
```

Обратившись к документации `Test::More`, можно найти много других интересных примеров функций, эскпортируемых данным модулем, которые могут быть нам полезны. Конечно, хочется подчеркнуть еще раз, что вы мо-

жете использовать *любой Perl-кода* при написании SWAT-генераторов и Perl-выражений, конкретные примеры можно найти на странице проекта SWAT.

Заключение

На этом я хотел бы закончить краткий экскурс в утилиту SWAT, предоставляющую DSL для написания smoke тестов web приложений. Не все фичи и тонкости раскрыты в данной статье, за полной документацией отправляйтесь на страницу проекта.

Проект достаточно молодой, но уже опробован автором на десятках web-приложений, есть большие планы по развитию и интеграции с существующими системами мониторинга, а пока, конечно же, хотелось бы получить обратную связь, вопросы, мнения и конструктивные замечания. Пишите в комменты или на почту или в гитхаб проект.

Спасибо!

■ *Алексей Мележик*

5. Обзор CPAN за июль 2015 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц

Статистика

- Новых дистрибутивов — 198
- Новых выпусков — 715

Новые модули

Coro::Multicore

Как известно, треды Coro не могут работать параллельно даже на многопроцессорных и/или многоядерных системах. Данный модуль позволяет исправить подобную ситуацию, но накладывает серьёзные ограничения на код, который может исполняться параллельно. Это могут быть только XS-функции, которые подготовлены специальным образом, и которые при работе не меняют никаких структур Perl.

Parallel::WorkUnit

Очередное прибавление в пространстве имён `Parallel`. Модуль `Parallel::WorkUnit` упрощает выполнение параллельных задач с возможностью возврата данных в родительский процесс.

```
my $wu = Parallel::WorkUnit->new();
my $pid = $wu->async( sub { 'child' }, sub { 'parent' } );

$wu->waitall();
```

Для запуска параллельной задачи модуль использует вызов `fork`, за исключением платформы windows, где используется модуль `threads`.

Perl::Tokenizer

Модуль `Perl::Tokenizer` — это попытка создания лексического анализатора Perl-кода с использованием регулярных выражений. В состав дистрибутива также входят несколько утилит:

- pfilter — фильтр кода, позволяющий, например, удалить комментарии и pod-документацию,
- pl2html — конвертация perl-кода в html-документ с подсветкой синтаксиса,
- pl2term — вывод perl-кода на консоль с подсветкой синтаксиса.

App::perlsh

Небольшая утилита `perlsh` — это очередная реализация REPL-консоли (*read-execute-print loop* — цикл чтения-выполнения-печати) для Perl. Данное приложение базируется на `Lexical::Persistence`, которое позволяет сохранять лексические переменные между циклами исполнения:

```
$ perlsh
eval: my $one = 1;
'1'

eval: my $two = 2;
'2'

eval: $one + $two
'3'
```

Attribute::Params::Validate

Модуль `Attribute::Params::Validate` для проверки параметров функций с помощью атрибутов был выведен из состава `Params::Validate` в отдельный дистрибутив. Автор не рекомендует его использовать, но если модуль кому-то интересен, автор готов передать права на сопровождение.

Capstone

`Capstone` — это обвязка к одноимённому фреймворку дизассемблера, поддерживающего множество аппаратных архитектур. Модуль можно использовать для создания дизассемблера и анализа бинарных исполняемых файлов.

experimentals

Модуль `experimentals` позволяет включить все существующие в данный момент экспериментальные возможности. То есть вместо записи:

```
use v5.22;
use experimental qw(
    fc           postderef    current_sub say          regex_sets
    unicode_eval state        array_base  postderef_qq switch
    smartmatch   lexical_subs bitwise     signatures
       lexical_topic
    evalbytes    refaliasing  unicode_strings          autoderef
);
```

Достаточно просто:

```
use v5.22;
use experimentals;
```

Keyword::Declare

`Keyword::Declare` позволяет задавать новые синтаксические конструкции языка с довольно сложной логикой. Используется keyword API Perl, поэтому для работы модуля требуется Perl ≥ 5.12. Синтаксический разбор кода происходит на этапе компиляции. Например, так можно создать новый оператор цикла `loop`:

```
use Keyword::Declare;

keyword loop (Int $count?, Block $block) {
    defined $count ?
        "for (1..$count) $block" :
        "while (1) $block";
}

```

```
loop 10 {
    ...
}
```

Dios

`Dios` — это ещё одна реализация ООП для Perl, которая основана на `Keyword::Declare` и технике *изнаночных* (inside-out) объектов. Синтаксически всё очень похоже на Perl6/Moose, но есть и некоторые уникальные особенности. Пример, класса:

```
use Dios;

# Indetity наследуется от Trackable
class Identity is Trackable {

    # Общие(разделяемые) переменные для всех объектов класса
    shared Num %!allocated_IDs; # приватная и только для чтения
    shared Num $.prev_ID is rw; # публичная для чтения и записи

    # Каждый объект получает свою копию этих атрибутов
    has Num $.ID     = _allocate_ID();  # Инициализация функцией
    has Str $.name //= '<anonymous>';   # Инициализация значением
        по умолчанию
    has Passwd $!passwd;                # Приватная переменная

    # Функция (без объекта)
    func _allocate_ID() { ... }

    # Метод (получает объект $self и доступ к атрибутам)
    method identify ($pwd) {
       return "$name [$ID]" if $pwd eq $passwd;
    }

    # Деструктор (субметод не наследуется)
    submethod DESTROY {
        say "Пока, $name!";
    }
}
```

Perl-ToPerl6

В соответствии с названием данный модуль предназначен для конвертации Perl 5 кода в код Perl 6. В состав дистрибутива входит утилита perlmogrify, которая имеет множество опций и может выполнить трансформацию как одного файла, так и дерева исходных кодов. Впрочем, работоспособность полученного Perl 6 кода не гарантируется.

AnyEvent::InfluxDB

`AnyEvent::InfluxDB` — это асинхронный клиент для набирающей популярность базы данных InfluxDB, предназначенной для хранения всевозможных метрик, событий и аналитики.

B::Utils1

Модуль `B::Utils1` — это форк `B::Utils`, который создал Рейни Урбан, чтобы исправить проблемы модуля в связи с несовместимыми изменениями в Perl 5.22. Поскольку Рейни (как и Марк Леманн) за словом в карман не лезет, то конфликт интересов в конечном счёте привёл к его бану в рассылке perl5-porters.

Обновлённые модули

File::Slurper 0.006

Вменяемый модуль для чтения содержимого файлов `File::Slurper` обновлён до версии 0.006. Автор указывает, что с модуля снят статус экспериментального и он может заменить небезопасный `File::Slurp`.

Sisimai 4.1.27

Обновлён модуль `Sisimai`, который предназначен для анализа почтовых сообщений об отказах в доставке (bounced mail). В новой версии исправлено несколько ошибок и добавлен метод `dump()`, позволяющий обработать сразу весь почтовый ящик с сообщениями формата mailbox или maildir и вернуть json-строку с результатом.

Devel::Cover 1.20

Обновлён модуль `Devel::Cover`, позволяющий проверить покрытие кода тестами. Исправлена работа модуля на Perl 5.22.0, также добавлены тесты для проверки работы с Perl 5.23.0.

Net::FTPSSL 0.29

Вышел новый релиз клиента защищённой передачи данных по FTP `Net::FTPSSL`. В новой версии реализована возможность повторного использования сессии, что сокращает задержку при повторных подключениях.

RPerl 1.000007

В июле состоялся первый мажорный релиз оптимизирующего компилятора RPerl. RPerl позволяет преобразовать Perl программу в C++ код, который затем компилируется с помощью `Inline::CPP` в бинарный модуль. Итоговый исполняемый файл в некоторых случаях работает на несколько порядков быстрее оригинальной программы. RPerl накладывает ограничения на используемые конструкции языка Perl.

Sidef 0.0900001

Sidef — это новый язык программирования реализованный на Perl. Sidef вобрал в себя лучшие идеи Ruby, Go, Perl 6 и JavaScript. Проект активно разви-

вается и регулярно выпускает релизы на CPAN.

HTTP::Message 6.10

После трёх лет забвения стали выходить обновления модуля `HTTP::Message`, в которых исправлены многие ошибки, накопившиеся в багтрекере за эти годы. Новый сопровождающий модуля *Karen Etheridge*.

Convert::UUlib 1.5

В новой версии модуля `Convert::UUlib` — обвязки к библиотеке uulib исправлена ошибка с переполнением буфера.

Plack::Middleware::ETag 0.05

Обновлён модуль `Plack::Middleware::ETag`, предназначенный для автоматического добавления заголовка ETag в ответ http-сервера. Исправлена ошибка в формировании значения ETag на основе номера inode файла: раньше вместо значения inode по ошибке использовалось значение битов прав доступа.

DBD::mysql 4.032

Состоялся релиз новой версии драйвера СУБД MySQL `DBD::MySQL`. В новой версии появился новый атрибут `mysql_enable_utf8mb4`, который схож с `mysql_enable_utf8`, но добавляет поддержку 4-байтовых символов Юникода (эмодзи и другие). Также добавлена поддержка передачи атрибутов подключения `mysql_conn_attr` для MySQL ≥ 5.6.6.

Encode 2.76

Обновлён модуль Encode. В новой версии исправлена давняя ошибка в таблице кодировки KOI8-U для символа BULLET OPERATOR (раньше он имел неверное наименование BULLET).

■ *Владимир Леттиев*

6. Интервью с Филиппом Брухатом

Филипп Брухат (Philippe Bruhat) — автор модулей на CPAN, подборки секретных операторов perlsecret, сооснователь конференций YAPC::EU

Когда и как научился программировать?

Первый компьютер мне достался когда я был подростком (где-то около 13 лет). Это был T07/70 (французский микрокомпьютер). Единственным доступным языком был Microsoft BASIC. Помню, что набирал бесконечные страницы программ из французского еженедельника Hebdogiciel, где были распечатки для всех компьютеров того времени (Oric, Sinclair, Commodore и т.д.).

Все выпуски можно почитать онлайн! http://www.abandonware-magazines.org/affiche_mag.php?mag=7&page=1

Также припоминаю, что большинство программ были в виде шестнадцатеричных дампов с загрузчиком. До меня не сразу дошло, почему там только буквы от A до F, и почему некоторые последовательности встречаются чаще, чем другие.

В конце концов я купил Assembler, но примеры в книге были неправильными, и я не сильно продвинулся.

Какой редактор испльзуешь?

Я пользуюсь vim, хотя мой `.vimrc` довольно скуден. Когда я начал использовать Unix, помню, что нужно было выбрать между Emacs и Vi, и кто-то мне сказал, что неважно что установлено в системе, но там всегда должен быть `vi`, поэтому выучить его было полезно. Так я и подсел.

Годами я пользовался vim не зная многих команд. В 1999 я купил vi-кружку через интернет и многое выучил прямо с нее. Но настоящим прорывом было когда мой друг, увидев ту самую кружку, сказал, что перед любой `:`-командой можно вставить «адрес» и она будет выполнена на соотвующем участке файла.

Кружку до сих пор можно купить через cafepress: http://www.cafepress.com/geekcheat.11507711

Шпаргалка, напечанная на кружке, выглядит следующим образом: http://f2.org/image/archive/vi-mug-detail.gif

Я часто пользуюсь терминалом. Помню как допечатывал последние символы на пятом соревновании по обфусцированному Perl через `screen`-сессию на настоящем VT320, подключенном к моему компьютеру через серийный порт. Этот терминал до сих пор пылится у меня на полке.

Моим первым настоящим email-клиентом в Unix был Elm (моим первым email-аккаунтом был аккаунт на VMS-машине, но я написал всего лишь дюжину писем прежде чем перейти на Unix и никогда об этом не пожалел). Потом я перешел на Pine, также немного сидел на Eudora (когда застрял на Windows 95 на несколько месяцев). Я перешел на mutt в ноябре 2002 года и до сих пор использую ее как свой основной email-клиент.

Когда и как познакомился с Perl?

Моей первой Perl-программой был счетчик посещений. Я его даже не написал, а просто скопировал. Позже (когда начала расти посещаемость моего сайта), смутно припоминаю, что добавлял что-то вроде блокировки файлов, чтобы счетчик не повреждался.

Другой программой, которую я «написал», была гостевая книга (вот это были дни). Я хорошо помню одну магическую Perl-строку. С 99.9% уверенностью могу сказать, что взял ее из Архива Мэтта, и это строкой была:

```
1 $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
```

Гостевая книга до сих пор доступна в Интернет Архиве http://web.archive.org/web/19970705232331/www2.ec-lille.fr/~book/goldenbook.htm и среди глупых сообщений есть первый контакт с женщиной, которая в дальнейшем стала мне женой. Кому нужны сервисы знакомств, если есть Perl и CGI?

С какими другими языками интересно работать?

Хотя я и согласен с популярной мудростью, что должно знать несколько языков программирования и использовать подходящий для конкретной задачи, мне немного стыдно признаться, что я использую Perl практически для всего.

Я писал несколько быстрых прототипов на Bash, но затем обычно переписы-

вал все на Perl, как только программа становилась более сложной.

Я писал несколько простых PostScript-программ, когда мне нужен был больший контроль на тем, что печатается на бумаге. Также с гордостью могу заявить, что однажды написал целую презентацию на PostScript, и некоторые слайды использовали функцию `rand` для отрисовки графиков. Также у меня был `Makefile` для пересборки слайдов (в основном для замены констант), чтобы они нормально выглядели независимо от разрешения проектора (4:3 или 16:9).

Также я написал как минимум одну нетривиальную программу на Befunge.

Очевидно, чтобы работать с современным веб, мне пришлось выучить немного JavaScript (jQuery рулит). Также я написал несколько строк на Python, так как это основной язык используемый в моей команде (у нас очень странное приложение, где основная работа сделана на Python, а чокнутые места на Perl).

У меня несколько книг о языках, которые хотелось бы выучить (Haskell, Erlang, Lisp), но так как Perl удовлетворяет все мои нужды, у меня нет времени и мотивации учить новые языки программирования.

Что мне действительно нравится, так это история компьютеров. У меня много книг об истории Unix и интернета, всегда ищу «классику». Одной из последних книг, которую я прочел от корки до корки, была «О стиле программирования» (“Elements of Programming Style”), второе издание, Керниган и Плагер (1978). Что означает, что я выучил немного PL/I и Фортран пока читал ее. До этой книги мне понравилась «Жизнь в Unix» (“Life with Unix”) Дона Либса (1989). И, конечно же, «Пособие для ненавидящих UNIX» (“Unix-Haters Handbook”), Гарфинкела, Вайса и Страсманна (1994).

Что, по-твоему, является самым большим преимуществом Perl?

Простой ответ — CPAN, с CPAN Testers на втором месте, но это не мой ответ.

Perl удивительный язык программирования, но самым большим успехом Ларри было привлечение сообщества.

По-моему, самым большим преимуществом Perl являются его люди.

Я не считал себя Perl-программистом пока не присоединился с парижским Perl-монгерам в апреле 1999. С тех пор я им являюсь.

INGY в 2004 году дал мое любимое Perl-интервью. До сих пор у меня бегут муражки по коже когда я читаю последний параграф, последнее предложение интервью: «Если глубоко копнуть, то меня держит Perl-сообщество. Эти люди как странная связанная семья. Perl привлекает людей определенного типа; это мои люди».

Интервью INGY: http://osdir.com/Article1534.phtml

Что, по-твоему, является самой важной особенностью языков будущего?

Возможность использовать несколько ядер процессора без необходимости fork и IPC. И затем хорошая система обмена данными поверх всего для возможности дальнейшего роста мощности. «Большие данные» (“Big data”) последнее время стало популярным словом; я знаю нескольких людей, которые занимаются подобным на работе, и ясно, что с технической стороны, любой достаточно успешный проект сталкивается с одной проблемой: масштабирование.

Успех, если вы работаете с интернетом, означает, что что бы вы не делали, это может вырасти на несколько порядков за короткое время. Если вы не хотите утонуть в своем собственном успехе, вам понадобятся технологии для соответствия росту. В некоторых областях моей работы (очень успешная интернет-компания) мы достигли лимита Perl 5.

Что думаешь о Perl 6?

Я достаточно давно в сообществе, так что я застал, когда Perl 6 только был объявлен. Я помню волнение, когда писались RFC. Помню как слышал о том, что кружки разбивались о стены. Помню, как разработку Parrot возглавил Дэн Сугальски.

Я пытался учить Perl 6, когда только появился Pugs, но дальше, чем несколько примеров, не продвинулся.

Я следил за прогрессом Perl 6 издалека, зная, что однажды мне придется его выучить. Мне нравится, что Ларри представляет Perl 5 и Perl 6 сестрами в семье Perl-языков. (Хотя, почитайте про семьи в его выступлении “State of

the Onion" в 2006 году http://www.perl.com/pub/2006/09/21/onion.html)

Мне нравится лого Perl 6, бабочка, и ее имя, Камелия. (И как имя пересекается с верблюдом Perl 5: camel, Amelia). Также мне нравится, что лого отталкивает многих программистов-мужиков. Хотя не думаю, что Ларри стоит представлять язык как привлекательный для семилетних девочек, потому как это только подкрепляет стереотипы с которыми он хочет бороться.

Многие годы я хотел поехать на FOSDEM, но никак не мог втиснуть конференцию в свое расписание. Наконец-то в этом году поехал (хотя я провел меньше 24 часов в Брюсселе, включая время, которое потратил, чтобы туда добраться) только для того, чтобы услышать объявление Ларри о том, что Perl 6 выходит в 2015 году.

Несколько недель назад я снова начал пробовать программировать на Perl 6, на этот раз с четкой целью: портировать один из моих Perl 5-модулей на Perl 6, и написать о том как опытный Perl 5-программист пишет реальный модуль на Perl 6.

Что ты делаешь для организации YAPC::EU и почему?

Я один из сооснователей YAPC Europe Foundation. В 2003 году сразу после YAPC Europe в Париже, организаторы прошлых четырех конференций (Лондон, Амстердам, Мюнхен и Париж) собрались вместе в баре обсудить будущее.

YAPC Europe стало жить своей жизнью, и не было необходимости в людях из YAS (Еще Одно Общество, неприбыльная организация, которая в последствии стала TPF), которые были за океаном и несколькими часовыми поясами, для помощи европейскому сообществу в организации конференций.

Команда из Парижа завела платежную систему, подключенную к нашему сайту (который был переписан через год для первого французского воркшопа, сайт в дальнейшем начал называться "Act"), и мы хотели сделать это доступным и для других Perl-конференций, так что работа и деньги, которые шли через платежную систему не были утеряны.

Так как команда была интернациональная, организация была зарегистрирована в Нидерландах как Dutch Stichting, с банковским счетом во франции (чтобы сохранить один и тот же банк для платежной системы, которую ис-

пользовали в Париже). А нашим первым председателем был немец.

У YAPC Europe Foundation (http://yapc.eu/) есть две роли: выбор команды для организации следующей конференции YAPC Europe; и управление платежной системой. Платежная система предоставляется бесплатно для Perl-сообщества, так как YEF оплачивает все счета. Весь бюджет YEF складывается из пожертвований и тратится на оплату онлайн-сервисов и предоставляется как начальная сумма для разных Perl-конференций http://www.yapceurope.org/finance/kickstart.html.

В 2008 году я занял роль казначея, так как у предыдущего не было достаточно свободного времени. Так что я тот, кто отправляет деньги участников организаторам. С 2005 года YEF пропустило через себя €350,000 от участников и спонсоров к организаторам 45 Perl-конференций.

Где сейчас работаешь, сколько времени проводишь за написанием Perl-кода?

С марта 2007 года я работаю в Booking.com в основном удаленно из Лиона. С тех пор компания выросла на порядок (и даже больше) в разных направлениях.

Я пишу на Perl большинство времени (я все еще разработчик) в команде, где большинство коллег моляться на Python. Мой текущий проект использует Moo (в течение некоторого времени я использовал одновременно Moose и Moo, но затем Moose полностью убрал), Dancer и DBIx::Class. Так как у проекта есть веб-интерфейс, я также немного пишу на JavaScript.

Стоит ли советовать молодым программистам учить сейчас Perl?

Не могу найти точной цитаты Ларри Уолла, но он как-то объяснял, что Perl вероятнее может быть последним языком, чем первым.

Людям определенно стоит изучить Perl, в дополнение к другим языкам в их арсенале. Все говорят, что стоит знать больше, чем один язык, и лучше всего делать это когда учишься программировать. Учителя должны преподавать как можно больше языков, как можно больше парадигм (императивное, декларативное, структурное, функциональное, ООП-программирование и т.д.), так чтобы студенты могли понять разницу и схожесть между ними.

И затем студенты могут узнать про Perl и смешать все воедино.

Вопросы от читателей

Как решил собрать в одном месте все секретные операторы Perl?

Я долгое время тусовался в рассылке Fun with Perl, и именно там Хосе Кастро начал спрашивать про названия популярных секретных операторов для своей книги OGSOP (Обфускация, Гольф и Секретные операторы в Perl). Я придумал название «детская коляска» для `@{[]}`, и оно мне так понравилось, что я захотел убедиться, что другие люди тоже будут использовать это название. Единственным способом добиться этого, было написать документацию самому…

Также в 2008-2009 гг. я сделал несколько докладов о секретных операторах в Perl на разных Perl-конференциях. Так само собой я начал их собирать.

У меня была заготовка для “perlsecret” (по аналогии с “perlop”) в одной из директорий (самая старая версия была от 2010 года) и однажды я решил довести ее до ума. Другой моей целью было включение этого списка в документацию Perl, но, как оказалось, упоминание “Goatse” не очень вписывалось в официальную документацию.

Git::Repository использует команду git, есть ли какое-то преимущество в использовании libgit2?

Это большая проблема, что у git нет библиотеки libgit. Это значит, что для управления git-репозиторием приходится использовать git (почти как только perl может отпарсить Perl).

Дело даже не в скорости: git очень быстр, и время на создание fork все равно будет тратится при использовании shell-скриптов. Программа, использующая libgit, будет вероятнее быстрее, чем shell-скрипт выполняющий git-команды, но я сомневаюсь, что другая библиотека (такая как libgit2) будет быстрее, чем git, выполнять git-операции.

Git::Repository оборачивает репозитории в простые объекты, которые предоставляют контекст для запуска git-команд. Для меня преимущество использования Git::Repository в том, что я могу работать со многими репозиториями не волнуясь о текущей рабочей директории. Также модуль поддерживает

все 128+ git-команд и работает с потоком данных (любого размера), который производится git.

На CPAN есть обертки над libgit2 (Git::Raw), но я не использовал их. Читал где-то, что в libgit2 объекты могут быть сохранены не только в git-объектах или git-пакетах. Это может быть полезным при работе с git-объектами крупных масштабов (как, например, это делает GitHub).

Почему розовый?

Это длинная история.

Я одеваю розовые вещи на Perl-конференции уже довольно давно. Я вынуждал некоторые конференции (и не только французские) печатать розовую футболку специально для меня. У меня также одна из двух розовых футболок Dancer. На YAPC в Пизе, Ник Кларк и BinGOs пародируя меня тоже одели розовые футболки. На работе некоторые коллеги требуют, чтобы я носил что-то розовое, или хотя бы с требовательным тоном спрашивают почему я не одел розовое сегодня.

Есть история почему я ношу розовые футболки. В 2001 году во время конференции YAPC Europe в Амстердаме, небольшая группа парижских Perl-монгеров подумало: «почему бы не организовать YAPC в Париже?». Затем мы встретились с Кевином Лензо (из YAS) и амстердамскими организаторами, чтобы поговорить о следующей YAPC. К нашему удивлению, была еще одна команда, хотевшая провести следующую YAPC. Так случилось, что команда из Мюнхена была первой. И у нас было два года, чтобы подготовить свою YAPC.

В 2002 году аукционы была до сих пор популярны на YAPC-конференциях. В течение аукциона в конце мюнхенской конференции продавалась множество смешных вещей (опкод Perl 6, оптимизация Perl 5, штаны Шверна, полуголая борьба на руках между Шверном и Дэмианом Конвейем…). У меня были свежи воспоминания про «драку» внутри London.pm во время аукциона в Амстердаме по поводу даты проведения ежемесячных социальных встреч, и мне захотелось инициировать что-нибудь подобное и здесь. Я выставил в качестве лота цвет футболки организаторов на следующей YAPC в Париже (все знали, что это будем мы, не было других вариантов).

Моя идея была в том, что Леон Брокард выставит свой любимый цвет (оран-

жевый), а фракция London.pm будет отстивать какой-нибудь другой цвет и цена за лот будет расти. Увы, оказалось, что британцы любят прикалываться над французами больше, чем над собой… Грег МакКарол выставил розовый и очень быстро он дошел до €100. Некоторые из парижской команды очень быстро стали на меня злиться…

Я решил превратить шутку в маркетинговый ход и сделал розовый официальным цветом конференции. Сайт был розовым (см. http://conferences.mongueurs.net/ye2003/coming_soon.html и http://conferences.mongueurs.net/ye2003/), флаеры были розовыми, футболки организаторов (все двенадцать) были розовыми.

Через год на аукционе в Белфасте (тут есть и своя история), я понял, что розовый это «наш цвет» (фрацузская YAPC), и в дальнейшем стал использовать розовый как отличительный цвет (так как никто из французов больше не носил розовые футболки). И так я делаю уже более десяти лет. Таким образом обычно люди и узнают меня на Perl-мероприятиях.

Мораль такова, что не обязательно быть гениальным программистом, чтобы тебя узнавали на Perl-конференции (я не претендую ни на гениальность, ни на известность), достаточно сделать что-то глупое и придерживаться этого.

Вообще мне нравится носить розовое, так как это в каком-то смысле способ противостоять гендерным стереотипам. Я рад, что мои дочери думают, что розовый это любимый цвет папы (на самом деле нет), вместо того, что это девчачий цвет (Цвета не должны быть принадлежать разным полам! Каждый может одевать то, что ему хочется!).

■ *Вячеслав Тихановский*