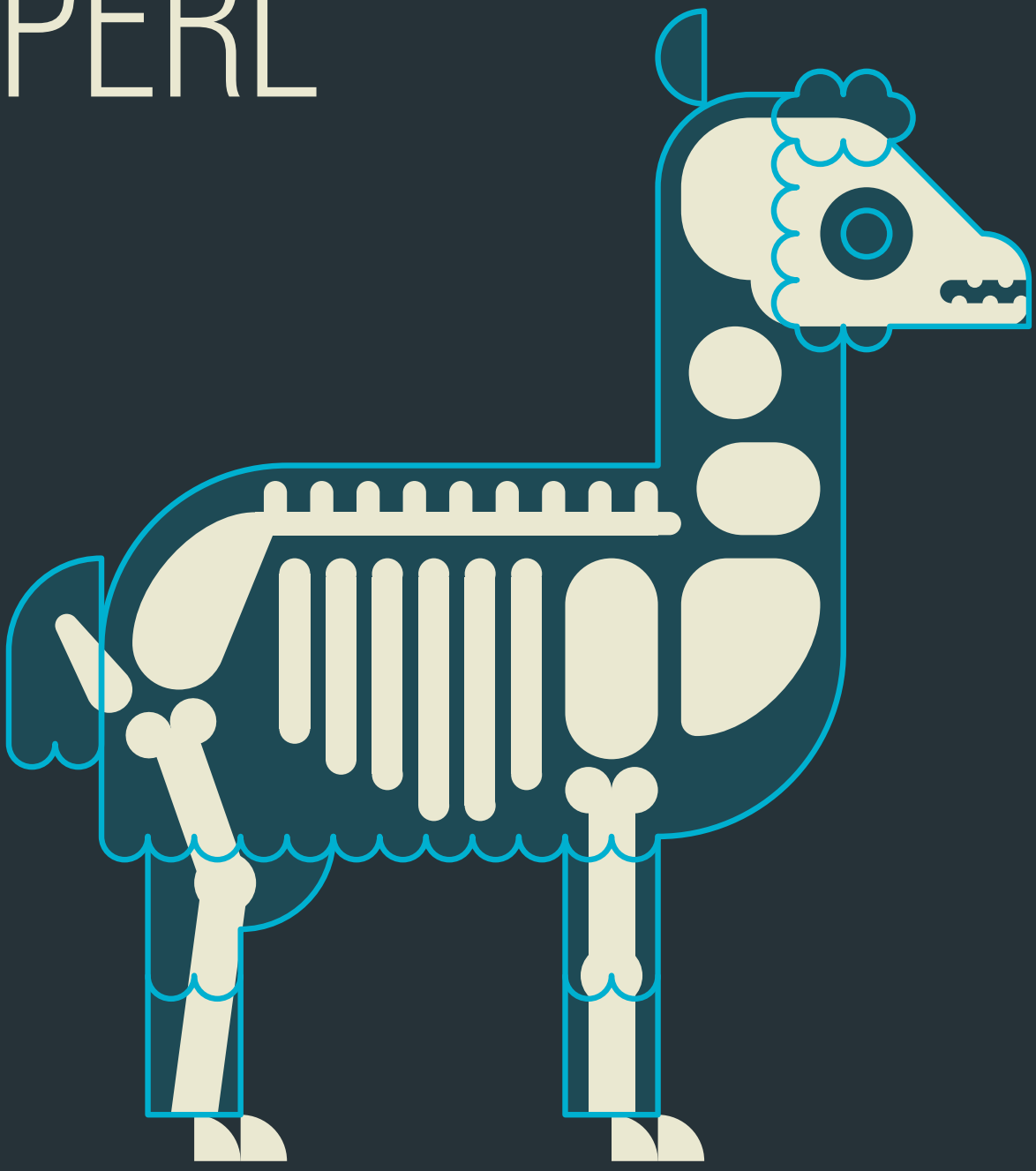
PRAGMATIC PERL

21



11/2014

pragmaticperl.com

Pragmatic Perl 21

pragmaticperl.com

Выпуск 21. Ноябрь 2014

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: Олеся Кузьмина, Дмитрий Шаматрин, Владимир Леттиев

Обложка: Марко Иваныk

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2014-11-29 16:25

Оглавление

1. От редактора

В прошлом месяце модулю DBI исполнилось 20 лет! Уверен, что каждый из Perl-программистов когда-либо использовал эту библиотеку.

Объявлена ежегодная номинация на награду White Camel за вклад в развитие Perl-сообщества. Предложить свою кандидатуру можно прямо брайану ди фою.

Традиционно в конце года проходит Perl-воркшоп Saint Perl в Санкт-Петербурге. В этом году это шестое мероприятие, и в название закралась подстрока «Perl 6» (нельзя здесь не провести параллель с недавним анонсом о том, что на конференции FOSDEM в начале следующего года будет объявлено, что Perl 6 появится в продакшене в 2015 году). Дата санкт-петербургского воркшопа выбирается максмально близкой ко дню рождения перла — 18 декабря. В планах организаторов на этот год — проведение не только дня с докладами, но и хакатона. Мероприятие пройдет 20 и 21 декабря 2014 года. Сайт конференции: event.yapcrussia.org/saintperl6/.

Для каждой статьи в интернет-версии журнала есть комментарии. Авторы постоянно следят за ними. Присоединяйтесь к обсуждению интересующих вас материалов. Комментарии можно оставлять без регистрации.

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. Тестирование в Perl. Практика

Следущая статья из цикла «Тестирование в Perl». На этот раз практические рекомендации и примеры

В прошлой статье Тестирование в Perl. Лучшие практики были перечислены особенности юнит-тестирования в Perl, как стоит писать и группировать тесты и т.д. Однако тем, кто не применял тестирование и не сталкивался с проблемами, трудно оценить и хорошие практики. На этот раз постараемся на практических примерах рассмотреть, как же тестировать классы.

В качестве задачи возьмем модуль логирования, который может логировать в stderr и в файл. Модуль должен поддерживать уровни логирования: error, warn и debug, а в качестве формата: .

Начальная структура модуля:

```
lib/
    Logger.pm
t/
    logger.t
```

Где `lib` директория с модулем, а `t` — директория с тестами.

Изначально модуль выглядит следующим образом:

```
package Logger;
use strict;
use warnings;

sub new {
    my $class = shift;

    my $self = {};
    bless $self, $class;

    return $self;
}

1;
```

А тест следующим образом:

```
use strict;
use warnings;

use Test::More;
use Logger;

subtest 'creates correct object' => sub {
    isa_ok(Logger->new, 'Logger');
};

done_testing;
```

Запускаем тесты с помощью `prove`:

```
$ prove t
t/logger.t .. ok
All tests successful.
```

На данный момент все тесты проходят. Но на самом-то деле особо ничего и не тестируется.

Вначале реализуем установку уровней логирования. Например, нам нужно, чтобы по умолчанию уровень был `error`. Пишем тест:

```
subtest 'has default log level' => sub {
    my $logger = Logger->new;

    is $logger->level, 'error';
};
```

Запускаем тесты.

```
$ prove t
t/logger.t .. 1/? Can't locate object method "level" via
   package "Logger"
```

Как видим, у логгера нет такого метода. Добавляем метод:

```
package Logger;
use strict;
use warnings;

sub new {
    my $class = shift;

    my $self = {};
    bless $self, $class;
```

```
10
11      return $self;
12  }
13
14  sub level {
15      my $self = shift;
16
17      return 'error';
18  }
19
20  1;
```

Конечно, мы не хотим, чтобы метод `level` всегда возвращал одно и тоже значение, нам нужно, чтобы оно как-то сохранялось. Напишем тест, который будет проверять, что устанавливаемое значения сохраняется.

```
1  subtest 'sets log level' => sub {
2      my $logger = Logger->new;
3
4      $logger->set_level('debug');
5
6      is $logger->level, 'debug';
7  };
```

Запускаем тесты:

```
1  $ prove t
2  t/logger.t .. 1/? Can't locate object method "set_level"
       via package "Logger"
```

Снова нет нужного метода, добавляем:

```
1  sub set_level {
2          my $self = shift;
3  }
```

Теперь метод есть, но тесты все еще не проходят:

```
1  t/logger.t .. 1/?
2      #   Failed test at t/logger.t line 22.
3      #          got: 'error'
4      #     expected: 'debug'
5      # Looks like you failed 1 test of 1.
```

Теперь метод `level` возвращает неправильное значение. Самое время реализовать сохранение.

```
1
2 ckage Logger;
3 use strict;
4 use warnings;
5
6 sub new {
7     my $class = shift;
8
9     my $self = {};
10    bless $self, $class;
11
12    return $self;
13 }
14
15 sub set_level {
16    my $self = shift;
17    my ($new_level) = @_;
18
19    $self->{level} = $new_level;
20 }
21
22 sub level {
23    my $self = shift;
24
25    return $self->{level};
26 }
27
28 1;
```

Запускаем тесты:

```
1 $ prove
2 t/logger.t .. 1/?
3     #   Failed test at t/logger.t line 14.
4     #          got: undef
5     #     expected: 'error'
6     # Looks like you failed 1 test of 1.
```

Хм, кажется, что новый тест начал проходить, но старый перестал. Вот пример того, как в будущем тесты могут выявить ошибку, которая проявится при изменении кода. Конечно, мы забыли, что по умолчанию значение уровня логирования должно быть `error`. Вернем этот функционал.

```
1 sub level {
2     my $self = shift;
3
```

```
4     return $self->{level} || 'error';
5 }
```

Теперь тесты проходят.

Далее необходимо проверить, что можно установить только разрешенные уровни логирования. Если устанавливается неизвестный уровень логирования, будем бросать исключение. Для проверок исключений воспользуемся модулем `Test::Fatal`. Новый тест будет выглядеть следующим образом:

```
1 subtest 'throws exception when invalid log level' => sub
      {
2     my $log = Logger->new;
3
4     ok exception { $log->set_level('unknown') };
5 };
```

Также нужно проверить, что все допустимые уровни логирования не бросают исключения. Чтобы не писать несколько одинаковых тестов, просто в цикле проверим каждое значение:

```
1 subtest 'not throws when known log level' => sub {
2     my $log = Logger->new;
3
4     for my $level (qw/error warn debug/) {
5         ok !exception { $log->set_level($level) };
6     }
7 };
```

Если запустить тесты, то они ожидаемо завалятся:

```
1 $ prove t
2 t/logger.t .. 1/?
3     #   Failed test at t/logger.t line 29.
4     # Looks like you failed 1 test of 1.
```

Но второй тест проходит и без какого либо кода. В данном случае мы знаем, почему так происходит, но часто бывает иначе. Пишется тест, пишется код. Запускается тест и проходит. Программист думает, что все отлично. Но на самом деле его функционал неправильный. Поэтому всегда важно перед написанием кода запустить тест и убедиться, что он не проходит.

Реализуем проверку списка уровней:

```
1  package Logger;
2  use strict;
3  use warnings;
4
5  use Carp qw(croak);
6  use List::Util qw(first);
7
8  sub new {
9      my $class = shift;
10
11     my $self = {};
12     bless $self, $class;
13
14     return $self;
15 }
16
17 sub set_level {
18     my $self = shift;
19     my ($new_level) = @_;
20
21     croak('Unknown log level')
22       unless first { $new_level eq $_ } qw/error warn
           debug/;
23
24     $self->{level} = $new_level;
25 }
26
27 sub level {
28     my $self = shift;
29
30     return $self->{level} || 'error';
31 }
32
33 1;
```

В качестве исключений бросается `croak` из модуля `Carp`, потому как он логичнее указывает на причину ошибки. Для поиска в списке используется `first` из модуля `List::Util`, который в отличие от обычного `grep` не будет дальше бежать по списку, а остановится при первом совпадении.

Далее напишем тест для метода `log`. Он должен печатать в stderr. Так как тестирование автоматическое, нужно как-то перехватить это вывод. Воспользуемся модулем `Capture::Tiny`, который отлич-

но справляется с этой задачей.

```
use Capture::Tiny qw(capture_stderr);

subtest 'prints to stderr' => sub {
    my $log = Logger->new;

    my $stderr = capture_stderr {
        $log->log('error', 'message');
    };

    ok $stderr;
};
```

Запускаем тест и убеждаемся, что он не проходит, и реализуем данный функционал:

```
sub log {
    my $self = shift;
    my ($level, $message) = @_;

    print STDERR $message;
}
```

Теперь тесты проходят, но, конечно, это не то, что нам нужно, нам нужно получить отформатированный вывод, для этого напишем соответствующий тест:

```
subtest 'prints formatted line' => sub {
    my $log = Logger->new;

    my $stderr = capture_stderr {
        $log->log('error', 'message');
    };

    like $stderr, qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[
       error\] message/;
};
```

Тесты не проходят:

```
$ prove t
t/logger.t .. 1/?
    #   Failed test at t/logger.t line 58.
    #                   'message'
    #     doesn't match '(?^:\d\d\d\d-\d\d-\d\d \d\d:\d\d
       :\d\d \[error\] message)'
```

Для форматирования воспользуемся `Time::Piece`. Код будет выглядеть следующим образом:

```
use Time::Piece;

sub log {
    my $self = shift;
    my ($level, $message) = @_;

    my $time = Time::Piece->new->strftime('%Y-%m-%d %T');
    print STDERR $time, " [$level] ", $message;
}
```

Теперь тесты проходят.

Понятно, что здесь тоже нужно реализовать проверку того, что передается правильный `level`, но мы переложим эту проверку на компилятор, реализовав отдельные методы для каждого уровня логирования. В итоге мы получим модифицированные тесты:

```
subtest 'prints to stderr' => sub {
    my $log = Logger->new;

    my $stderr = capture_stderr {
        $log->error('message');
    };

    ok $stderr;
};

subtest 'prints formatted line' => sub {
    my $log = Logger->new;

    my $stderr = capture_stderr {
        $log->error('message');
    };

    like $stderr, qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[
       error\] message/;
};
```

Не нужно забывать, что нужно проверить все комбинации, поэтому тесты еще раз модифицируем:

```
subtest 'prints to stderr' => sub {
    my $log = Logger->new;

```

```
    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

        ok $stderr;
    }
};

subtest 'prints formatted line' => sub {
    my $log = Logger->new;

    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

        like $stderr, qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\
           d \[$level\] message/;
    }
};
```

Реализуем подобный фукнционал, спрятав метод `log`:

```
sub error { shift->_log('error', @_) }
sub warn  { shift->_log('warn',  @_) }
sub debug { shift->_log('debug', @_) }

sub _log {
    my $self = shift;
    my ($level, $message) = @_;

    my $time = Time::Piece->new->strftime('%Y-%m-%d %T');
    print STDERR $time, " [$level] ", $message;
}
```

Глядя на код, я заметил ошибку — отсутствие перевода строки. Чтобы убедиться в том, что это ошибка, вначале напишем тест:

```
subtest 'prints to stderr with \n' => sub {
    my $log = Logger->new;

    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

        like $stderr, qr/\n$/;
```

```
    }
};
```

Теперь запускаем тесты, чтобы убедиться, что эта ошибка воспроизводится. Теперь исправляем:

```
print STDERR $time, " [$level] ", $message, "\n";
```

Если выделять отдельные ошибки в отдельные тесты, в дальнейшем можно легко проверять, что они не возвращаются при исправлении других ошибок или изменении функционала. Это защита от регрессий.

Для чего же вообще реализовывали уровни логирования? Конечно, чтобы при выставленном уровне логировать только те события, уровень которых выше текущего. Т.е. в режиме `debug` должны логироваться `error`, `warn` и `debug`. В режиме `warn` — `error` и `warn`, а в режиме `error` только `error`. Чтобы упростить тесты, составим таблицу ожидаемых значений на тестовые данные. Например, тест на то, что все логируется при нужном уровне, будет выглядеть следующим образом:

```
subtest 'logs when level is higher' => sub {
    my $log = Logger->new;

    my $levels = {
        error => [qw/error/],
        warn  => [qw/error warn/],
        debug => [qw/error warn debug/],
    };

    for my $level (keys %$levels) {
        $log->set_level($level);
        for my $test_level (@{$levels->{$level}}) {
            ok capture_stderr {
                $log->$test_level('message');
            };
        }
    }
};
```

Похожим образом будет выглядеть тест, который проверяет, что сообщения не логируются при неправильном уровне:

```
subtest 'not logs when level is lower' => sub {
```

```
    my $log = Logger->new;

    my $levels = {
        error => [qw/warn debug/],
        warn  => [qw/debug/],
    };

    for my $level (keys %$levels) {
        $log->set_level($level);
        for my $test_level (@{$levels->{$level}}) {
            ok !capture_stderr {
                $log->$test_level('message');
            };
        }
    }
};
```

Здесь нет проверки debug, потому как при этом режиме все логируется. Убеждаемся, что тесты не проходят и пишем реализацию. Во-первых, каждому уровню присваиваем свое значение, во-вторых, перед логированием проверяем, допускает ли текущий уровень логирование.

Во время реализации столкнулся с тем, что при запуске тестов непонятно, что ломается, поэтому добавил сообщения:

```
ok !capture_stderr {
    $log->$test_level('message');
}, "not log '$test_level' when '$level'";
```

Теперь сообщения стали понятнее:

```
    #   Failed test 'not log 'warn' when 'error''
    #   at t/logger.t line 109.
```

После реализации оказалось, что ломаются многие старые тесты, как, например, вот этот:

```
subtest 'prints to stderr' => sub {
    my $log = Logger->new;

    for my $level (qw/error warn debug/) {
        my $stderr = capture_stderr {
            $log->$level('message');
        };

```

```
9           ok $stderr;
10      }
11  };
```

Понятно, что нужно выставить самый высокий уровень логирования debug. Также и в других тестах.

Измененный код:

```
1  my $LEVELS = {
2      error => 1,
3      warn  => 2,
4      debug => 3
5  };
6
7  ...
8
9  sub set_level {
10     my $self = shift;
11     my ($new_level) = @_;
12
13     croak('Unknown log level')
14       unless first { $new_level eq $_ } keys %$LEVELS;
15
16     $self->{level} = $new_level;
17 }
18
19 sub _log {
20     my $self = shift;
21     my ($level, $message) = @_;
22
23     return unless $LEVELS->{$level} <= $LEVELS->{$self->
          level};
24
25     my $time = Time::Piece->new->strftime('%Y-%m-%d %T');
26     print STDERR $time, " [$level] ", $message, "\n";
27 }
```

Так как создание и инициализация логгера в каждом тесте практически одинаковая, выделим это в отдельный метод и уменьшим дублирование в тесте:

```
1  sub _build_logger {
2      my $logger = Logger->new;
3      $logger->set_level('debug');
4      return $logger;
5  }
```

На данном этапе логгер полностью протестирован. Чтобы в этом убедиться, запустим Devel::Cover:

```
$ PERL5OPT=-MDevel::Cover prove t
---------------------------- ------ ------ ------ ------ ------ ------
File                           stmt   bran   cond    sub   time  total
---------------------------- ------ ------ ------ ------ ------ ------
lib/Logger.pm                 100.0  100.0  100.0  100.0    3.4  100.0
---------------------------- ------ ------ ------ ------ ------ ------
```

Теперь необходимо реализовать логгер, который будет логировать в файл. Чтобы избежать дублирования, в данном случае воспользуемся шаблонными методами, но все зависит от конкретной задачи, и существует большое количество способов. Вначале переименуем логгер в LoggerStderr. Теперь создадим LoggerFile, который будет копией LoggerStderr, также скопируем тест. Теперь директория выглядит следующим образом:

```
lib/
    LoggerFile.pm
    LoggerStderr.pm
t/
    logger_file.t
    logger_stderr.t
```

В `logger_file.t` подправим тесты, которые проверяют, что лог записался в stderr на проверку, что запись была произведена в файл. Вместо Capture::Tiny, напишем собственную функцию, которая будет читать из файла:

```
sub _slurp {
    my $file = shift;
    return do { local $/; open my $fh, '<', $file or die
       $!; <$fh> };
}
```

Для тестирования записи в файл, будет создавать временные файлы с помощью File::Temp, и тесты будут выглядит следущим образом:

```
subtest 'prints to file' => sub {
    my $file = File::Temp->new;
    my $log = _build_logger(file => $file->filename);

```

```
    for my $level (qw/error warn debug/) {
        $log->$level('message');

        my $content = _slurp($file->filename);

        ok $content;
    }
};
```

Как видно в конструктор передается имя файла и `_build_logger` приобретает следующий вид:

```
sub _build_logger {
    my $logger = LoggerFile->new(@_);
    $logger->set_level('debug');
    return $logger;
}
```

Запускаем тесты и убеждаемся, что они не проходят. Теперь реализовываем запись в файл:

```
sub _log {
    my $self = shift;
    my ($level, $message) = @_;

    return unless $LEVELS->{$level} <= $LEVELS->{$self->
       level};

    my $time = Time::Piece->new->strftime('%Y-%m-%d %T');

    open my $fh, '>>', $self->{file} or die $!;
    print $fh $time, " [$level] ", $message, "\n";
    close $fh;
}
```

И тесты, и реализации содержат много дублирования. Вначале избавимся от дублирования в реализациях, выделив базовый класс с шаблонным методом `_print`, который будет реализован в LoggerFile и LoggerStderr. Во время рефакторинга постоянно запускаем тесты, чтобы убедиться, что ничего не сломалось.

Базовый класс:

```
package LoggerBase;
use strict;
use warnings;
```

```
4
5  use Carp qw(croak);
6  use List::Util qw(first);
7  use Time::Piece;
8
9  my $LEVELS = {
10     error => 1,
11     warn  => 2,
12     debug => 3
13 };
14
15 sub new {
16     my $class = shift;
17     my (%params) = @_;
18
19     my $self = {};
20     bless $self, $class;
21
22     $self->{file} = $params{file};
23
24     return $self;
25 }
26
27 sub set_level {
28     my $self = shift;
29     my ($new_level) = @_;
30
31     croak('Unknown log level')
32       unless first { $new_level eq $_ } keys %$LEVELS;
33
34     $self->{level} = $new_level;
35 }
36
37 sub level {
38     my $self = shift;
39
40     return $self->{level} || 'error';
41 }
42
43 sub error { shift->_log('error', @_) }
44 sub warn  { shift->_log('warn',  @_) }
45 sub debug { shift->_log('debug', @_) }
46
47 sub _log {
48     my $self = shift;
49     my ($level, $message) = @_;
50
51     return unless $LEVELS->{$level} <= $LEVELS->{$self->
```

```
        level};
52
53      my $time = Time::Piece->new->strftime('%Y-%m-%d %T');
54
55      my $text = join '', $time, " [$level] ", $message, "\
          n";
56
57      $self->_print($text);
58  }
59
60  sub _print { ... }
61
62  1;
```

LoggerStderr:

```
1  package LoggerStderr;
2  use strict;
3  use warnings;
4
5  use base 'LoggerBase';
6
7  sub _print {
8      my $self = shift;
9      my ($message) = @_;
10
11     print STDERR $message;
12 }
13
14 1;
```

LoggerFile:

```
1  package LoggerFile;
2  use strict;
3  use warnings;
4
5  use base 'LoggerBase';
6
7  sub new {
8      my $self = shift->SUPER::new(@_);
9      my (%params) = @_;
10
11     $self->{file} = $params{file};
12
13     return $self;
14 }
15
```

```
16 sub _print {
17     my $self = shift;
18     my ($message) = @_;
19 
20     open my $fh, '>>', $self->{file} or die $!;
21     print $fh $message;
22     close $fh;
23 }
24 
25 1;
```

Так как появился новый класс, его тоже нужно тестировать. Как тестировать базовые классы? В самом тесте создается тестовый класс наследующий тестируемый базовый класс. А из тестов наследников удаляются дублирующие тесты. Некоторые тесты все равно останутся продублированными, но обо всем по порядку. Вначале напишем тест на базовый класс:

```
1 use strict;
2 use warnings;
3 
4 use Test::More;
5 use Test::Fatal;
6 
7 subtest 'creates correct object' => sub {
8     isa_ok(LoggerTest->new, 'LoggerTest');
9 };
10 
11 subtest 'has default log level' => sub {
12     my $logger = LoggerTest->new;
13 
14     is $logger->level, 'error';
15 };
16 
17 subtest 'sets log level' => sub {
18     my $logger = LoggerTest->new;
19 
20     $logger->set_level('debug');
21 
22     is $logger->level, 'debug';
23 };
24 
25 subtest 'not throws when known log level' => sub {
26     my $log = LoggerTest->new;
27 
28     for my $level (qw/error warn debug/) {
29         ok !exception { $log->set_level($level) };
```

```
30      }
31  };
32
33  subtest 'throws exception when invalid log level' => sub
        {
34      my $log = LoggerTest->new;
35
36      ok exception { $log->set_level('unknown') };
37  };
38
39  sub _build_logger {
40      my $logger = LoggerTest->new(@_);
41      $logger->set_level('debug');
42      return $logger;
43  }
44
45  done_testing;
46
47  package LoggerTest;
48  use base 'LoggerBase';
49
50  sub _print { }
```

Тесты, выделенные в базом тесте, удаляем из других файлов.

На данный момент и в `logger_stderr.t`, и в `logger_file.t` остались тесты, которые практически одинаковы, но сильно завязаны на реализацию. Например, в тесте:

```
1  subtest 'not logs when level is lower' => sub {
2      my $log = _build_logger();
3
4      my $levels = {
5          error => [qw/warn debug/],
6          warn  => [qw/debug/],
7      };
8
9      for my $level (keys %$levels) {
10         $log->set_level($level);
11         for my $test_level (@{$levels->{$level}}) {
12             ok !capture_stderr {
13                 $log->$test_level('message');
14             }, "not log '$test_level' when '$level'";
15         }
16     }
17 };
```

на самом деле нужно проверить, что метод _print не вызывается, а не проверять, была ли запись в stderr. Этот тест можно перенести в базовый, немного модифицировав базовый тестовый класс.

```
package LoggerTest;
use base 'LoggerBase';

sub new {
    my $self = shift->SUPER::new(@_);
    my (%params) = @_;

    $self->{output} = $params{output};

    return $self;
}

sub _print {
    my $self = shift;

    push @{$self->{output}}, @_;
}
```

Таким образом, передавая в конструктор массив $output мы сможем дальше в тесте проверить, что в него записалось. Теперь тест на проверку правильности форматировния выглядит следующим образом:

```
subtest 'prints formatted line' => sub {
    my $output = [];
    my $log = _build_logger(output => $output);

    for my $level (qw/error warn debug/) {
        $log->$level('message');

        like $output->[-1], qr/\d\d\d\d-\d\d-\d\d \d\d:\d
           \d:\d\d \[$level\] message/;
    }
};
```

и их можно удалить из дочерних классов. Также поступаем с тестами, которые не тестируют саму реализацию записи, а внутреннее поведение, реализованное в базовом классе. Финальные тесты выглядят следущим образом:

logger_base.t:

```
use strict;
```

```
use warnings;

use Test::More;
use Test::Fatal;

subtest 'creates correct object' => sub {
    isa_ok(LoggerTest->new, 'LoggerTest');
};

subtest 'has default log level' => sub {
    my $logger = LoggerTest->new;

    is $logger->level, 'error';
};

subtest 'sets log level' => sub {
    my $logger = LoggerTest->new;

    $logger->set_level('debug');

    is $logger->level, 'debug';
};

subtest 'not throws when known log level' => sub {
    my $log = LoggerTest->new;

    for my $level (qw/error warn debug/) {
        ok !exception { $log->set_level($level) };
    }
};

subtest 'throws exception when invalid log level' => sub
   {
    my $log = LoggerTest->new;

    ok exception { $log->set_level('unknown') };
};

subtest 'prints formatted line' => sub {
    my $output = [];
    my $log = _build_logger(output => $output);

    for my $level (qw/error warn debug/) {
        $log->$level('message');

        like $output->[-1],
          qr/\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d \[$level\]
              message/;
```

```
48      }
49  };
50
51  subtest 'logs when level is higher' => sub {
52      my $output = [];
53      my $log = _build_logger(output => $output);
54
55      my $levels = {
56          error => [qw/error/],
57          warn  => [qw/error warn/],
58          debug => [qw/error warn debug/],
59      };
60
61      for my $level (keys %$levels) {
62          $log->set_level($level);
63          for my $test_level (@{$levels->{$level}}) {
64              $log->$test_level('message');
65
66              ok $output->[-1];
67          }
68      }
69  };
70
71  subtest 'not logs when level is lower' => sub {
72      my $output = [];
73      my $log = _build_logger(output => $output);
74
75      my $levels = {
76          error => [qw/warn debug/],
77          warn  => [qw/debug/],
78      };
79
80      for my $level (keys %$levels) {
81          $log->set_level($level);
82          for my $test_level (@{$levels->{$level}}) {
83              $log->$test_level('message');
84
85              ok !$output->[-1], "not log '$test_level'
                    when '$level'";
86          }
87      }
88  };
89
90  sub _build_logger {
91      my $logger = LoggerTest->new(@_);
92      $logger->set_level('debug');
93      return $logger;
94  }
```

```
95
96 done_testing;
97
98 package LoggerTest;
99 use base 'LoggerBase';
100
101 sub new {
102     my $self = shift->SUPER::new(@_);
103     my (%params) = @_;
104
105     $self->{output} = $params{output};
106
107     return $self;
108 }
109
110 sub _print {
111     my $self = shift;
112
113     push @{$self->{output}}, @_;
114 }
```

logger_stderr.t:

```
1 use strict;
2 use warnings;
3
4 use Test::More;
5 use Test::Fatal;
6 use Capture::Tiny qw(capture_stderr);
7 use LoggerStderr;
8
9 subtest 'creates correct object' => sub {
10     isa_ok(LoggerStderr->new, 'LoggerStderr');
11 };
12
13 subtest 'prints to stderr' => sub {
14     my $log = _build_logger();
15
16     for my $level (qw/error warn debug/) {
17         my $stderr = capture_stderr {
18             $log->$level('message');
19         };
20
21         ok $stderr;
22     }
23 };
24
25 subtest 'prints to stderr with \n' => sub {
```

```
26      my $log = _build_logger();
27
28      for my $level (qw/error warn debug/) {
29          my $stderr = capture_stderr {
30              $log->$level('message');
31          };
32
33          like $stderr, qr/\n$/;
34      }
35  };
36
37  sub _build_logger {
38      my $logger = LoggerStderr->new;
39      $logger->set_level('debug');
40      return $logger;
41  }
42
43  done_testing;
```

logger_file.t:

```
1   use strict;
2   use warnings;
3
4   use Test::More;
5   use Test::Fatal;
6   use File::Temp;
7   use LoggerFile;
8
9   subtest 'creates correct object' => sub {
10      isa_ok(LoggerFile->new, 'LoggerFile');
11  };
12
13  subtest 'prints to file' => sub {
14      my $file = File::Temp->new;
15      my $log = _build_logger(file => $file->filename);
16
17      for my $level (qw/error warn debug/) {
18          $log->$level('message');
19
20          my $content = _slurp($file);
21
22          ok $content;
23      }
24  };
25
26  subtest 'prints to stderr with \n' => sub {
27      my $file = File::Temp->new;
```

```
28      my $log = _build_logger(file => $file);
29
30      for my $level (qw/error warn debug/) {
31          $log->$level('message');
32
33          my $content = _slurp($file);
34
35          like $content, qr/\n$/;
36      }
37  };
38
39  sub _slurp {
40      my $file = shift;
41      my $content = do { local $/; open my $fh, '<', $file
           ->filename or die $!; <$fh> };
42      return $content;
43  }
44
45  sub _build_logger {
46      my $logger = LoggerFile->new(@_);
47      $logger->set_level('debug');
48      return $logger;
49  }
50
51  done_testing;
```

В завершение реализуем фабрику, которая создает тот или иной логгер. Нужно протестировать, что возвращается нужный объект, и что при неизвестном запрашиваемом логгере бросается ошибка. Тест может выглядеть следующим образом:

```
1  use strict;
2  use warnings;
3
4  use Test::More;
5  use Test::Fatal;
6  use Logger;
7
8  subtest 'creates stderr logger' => sub {
9      my $logger = Logger->build('stderr');
10
11     isa_ok $logger, 'LoggerStderr';
12 };
13
14 subtest 'creates file logger' => sub {
15     my $logger = Logger->build('file');
16
```

```
    isa_ok $logger, 'LoggerFile';
};

subtest 'throws when unknown logger' => sub {
    ok exception { Logger->build('unknown') };
};

done_testing;
```

И сама фабрика:

```
package Logger;

use strict;
use warnings;

use Carp qw(croak);
use LoggerStderr;
use LoggerFile;

sub build {
    my $class = shift;
    my ($type, @args) = @_;

    if ($type eq 'stderr') {
        return LoggerStderr->new(@args);
    } elsif ($type eq 'file') {
        return LoggerFile->new(@args);
    }

    croak('Unknown type');
}

1;
```

Таким образом мы написали с помощью TDD-методологии простейший логгер. Надеюсь, что процесс был понятным, а если нет — спрашивайте в комментариях!

■ *Вячеслав Тихановский*

3. GUI-приложения на Perl с помощью wxWidgets

Рассмотрены основы написания GUI-приложений с помощью `wxWidgets`

Написание GUI-приложений на Perl ничем не отличается от написания на других динамических языках. Все зависит от качества оберток над оригинальными библиотеками и их своевременного обновления.

Почему `wxWidgets`?

`wxWidgets` является по-настоящему кросс-платформенным GUI-инструментарием. В отличие от, например, `Qt`, который тоже вроде бы является кросс-платформенным, wxWidget на запускаемых платформах использует родные библиотеки, а не пытается их эмулировать. Таким образом приложения выглядят не совсем одинаково, но совершенно привычно для своей графической оболочки.

Более того, поддержка у `wxWidgets` для Perl является полной, качественной и современной. Например, популярный редактор `Padre` тоже написан с помощью wxWidgets. Имея под рукой большой открытый и свободный проект, всегда можно подглядеть, как реализуются те или иные виджеты, можно на конкретном примере быстро научиться.

Настройка окружения

Для разработки я использую платформу GNU/Linux. Поэтому дальше приведена установка `wxWidgets` именно для моей ОС. Проще всего устанавливать библиотеки из дистрибутивных пакетов, на примере Debian:

```
1 # apt-get install libwx-perl
```

Конечно, всегда можно установить и с CPAN, но тогда придется много компилировать и можно наткнуться на несовместимость версий.

```
$ cpanm Wx
```

Первое приложение

Стандартным первым приложением будет окно на котором написано “Hello, world!”:

```
use strict;
use warnings;

use Wx;

my $app = Wx::SimpleApp->new;
my $frame = Wx::Frame->new(undef, -1, "Hello, world!");
$frame->Show;
$app->MainLoop;
```

`Wx::Frame` это окно, у которого может быть изменен размер, у него может быть статусная строка, а также меню. Т.е. это окно общего назначения. Кроме `Wx::Frame` существует также `Wx::Dialog` и другие. Для добавления визуальных элементов можно воспользоваться различными классами и методами `wxWidgets`. Например, чтобы добавить кнопку, которая закрывает приложение, напишем следующий код:

```
use strict;
use warnings;

use Wx;
use Wx::Event;

my $app    = Wx::SimpleApp->new;
my $frame  = Wx::Frame->new(undef, -1, "Hello, world!");

my $button = Wx::Button->new($frame, -1, 'Close');
Wx::Event::EVT_BUTTON($frame, $button, sub { $frame->
   Close(1) });

$frame->Show;
$app->MainLoop;
```

Все методы, события и т.д. хорошо описаны в документации wxWidgets. Не будем на этом подробно останавливаться. Рассмотрим общий подход в написании GUI-приложений.

Генераторы XML-интерфейса

Для небольшого приложения это вполне допустимо, также как и для простого Perl-скрипта можно все записать в один файл. Однако это очень неудобно для больших приложений с большим количеством окон, диалогов, сложной логикой.

Для графических приложений применяют тот же подход, что и для веб-приложений — MVC, или разделение бизнес-логики, отображения и управления. Так же можно поступить и с GUI-программой.

Для построения интерфейса лучше воспользоваться специальными программами, генерирующими XML-настройки (XRC-файлы), которые, будучи загруженными в приложении, преобразуются в графический интерфейс. Это гораздо легче поддерживать и проще изменять.

Для `wxWidgets` существует несколько таких генераторов. Наиболее полным является `wxFormBuilder`. Он позволяет создавать всевозможные элементы, указывать их расположение на окнах и т.д.

Например, XRC-файл для предыдущего приложения может выглядеть так:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<resource xmlns="http://www.wxwindows.org/wxxrc" version="2.3.0.1">
    <object class="wxFrame" name="MainFrame">
        <style>wxDEFAULT_FRAME_STYLE|wxTAB_TRAVERSAL</style>
        <size>500,300</size>
        <title></title>
        <centered>1</centered>
        <aui_managed>0</aui_managed>
        <object class="wxButton" name="CloseButton">
            <label>Close</label>
            <default>0</default>
```

```
        </object>
    </object>
</resource>
```

Теперь загрузим эту конфигурацию в приложении:

```
use strict;
use warnings;

use Wx;
use Wx::XRC;
use Wx::Event;

my $app = Wx::SimpleApp->new;

my $xrc = Wx::XmlResource->new();
$xrc->InitAllHandlers();
$xrc->Load('example.xrc');

my $frame = Wx::Frame->new;
$xrc->LoadFrame($frame, undef, 'MainFrame');

Wx::Event::EVT_BUTTON(
    $frame,
    Wx::XmlResource::GetXRCID('CloseButton'),
    sub { $frame->Close(1) }
);

$frame->Show;
$app->MainLoop;
```

Для того, чтобы достать объект `Wx::Frame`, используем метод `LoadFrame`:

```
my $frame = Wx::Frame->new;
$xrc->LoadFrame($frame, undef, 'MainFrame');
```

А для того, чтобы достать объект кнопки, вначале получаем ее `id` по имени:

```
Wx::XmlResource::GetXRCID('CloseButton')
```

А затем привязываем событие.

Всегда стоит понятным образом называть все элементы, а не оставлять что-то вроде `m_Button_1`, это сделает управляющий код суще-

ственно понятнее.

Разделение на классы

Как и любое Perl-приложение, разделение на отдельные пакеты и классы упрощает понимание кода, делает возможным тестирование отдельных классов, а также позволяет легко наследовать и менять поведение графических элементов.

Разделим наше приложение на несколько составляющих.

```
1  package MyApp;
2
3  use strict;
4  use warnings;
5
6  use base 'Wx::SimpleApp';
7
8  use Wx;
9  use Wx::XRC;
10 use MyMainFrame;
11
12 sub OnInit {
13     my $self = shift;
14
15     my $xrc = Wx::XmlResource->new();
16     $xrc->InitAllHandlers();
17     $xrc->Load('app.xrc');
18
19     my $frame = MyMainFrame->new;
20     $xrc->LoadFrame($frame, undef, 'MainFrame');
21
22     $frame->Show;
23 }
24
25 1;
```

Задача основого класса выполнить загрузку настроек, создать основное окно и показать его.

```
1  package MyMainFrame;
2
3  use strict;
4  use warnings;
5
6  use base 'Wx::Frame';
7
8  use Wx;
9  use Wx::Event;
10 use Wx::XRC;
11
12 sub new {
13     my $self = shift->SUPER::new(@_);
14
15     Wx::Event::EVT_BUTTON(
16         $self,
17         Wx::XmlResource::GetXRCID('CloseButton'),
18         sub { $self->Close(1) }
19     );
20
21     return $self;
22 }
23
24 1;
```

Задача этого класса выполнить настройку своего окна, т.е. прописать все события, в данном случае обработать нажатие на кнопку `CloseButton`.

Скрипт запуска Скрипт запуска `app.pl` является входной точкой для запуска приложения.

```
1  #!/usr/bin/env perl
2
3  use strict;
4  use warnings;
5
6  use Wx;
7  use MyApp;
8
9  MyApp->new->MainLoop;
```

С помощью такой декомпозиции каждый класс занимается своим делом, а структура приложения выглядит следующим образом:

```
app.pl
app.xrc
lib/
    MyApp.pm
    MyMainFrame.pm
```

Можно у каждого класса сделать какой-то уникальный префикс, например GUI, чтобы их отличать от обычных Perl-классов, которые будут заниматься логикой приложения.

Генераторы Perl-кода

`wxFormBuilder`, как было уже указано, может генерировать XML-представление, однако он позволяет гененировать и код. К сожалению, из коробки нет поддержки генерации Perl-кода. Однако, разработчики Padre реализовали собственный генератор Perl-кода из файла проекта wxFormBuilder. Для этого потребуются два модуля: `FBP` и `FBP::Perl`. Написав простейший скрипт для создания классов, получим на выходе дерево файлов, где каждый — это отдельное окно:

```
#!/usr/bin/env perl

use strict;
use warnings;

my ($project_file, $root_dir) = @ARGV;
die "Usage: <project_file> <dir>\n" unless $project_file
    && $root_dir;
die "'$project_file' does not exist\n" unless -e
    $project_file;
die "'$root_dir' does not exist\n"     unless -d
    $root_dir;

use File::Slurp qw(write_file);
use File::Spec::Functions qw(catfile);
use File::Path qw(make_path);
use File::Basename qw(dirname);
use FBP;
use FBP::Perl;
```

```
17
18 my $fbp = FBP->new;
19 $fbp->parse_file($project_file);
20
21 my $project = $fbp->project;
22
23 my $generator = FBP::Perl->new(project => $project);
24
25 foreach my $form ($project->forms) {
26     my $content = $generator->flatten($generator->
       frame_class($form));
27
28     write_class($form->name, $content);
29 }
30
31 foreach my $dialog ($project->dialogs) {
32     my $content = $generator->flatten($generator->
       dialog_class($dialog));
33
34     write_class($dialog->name, $content);
35 }
36
37 sub write_class {
38     my ($class_name, $content) = @_;
39
40     my $path = join('/', split /::/, $class_name) . '.pm'
       ;
41     $path = catfile($root_dir, $path);
42
43     my $dir = dirname($path);
44     make_path($dir);
45
46     write_file $path, $content;
47 }
```

Плюс этого подхода еще и в том, что нет необходимости вручную привязывать сигналы к элементам. Указав в `wxFormBuilder` название обработчика, `PBP::Perl` правильно сгенерирует соответствующий код. Вот как, например, выглядит класс основной формы:

```
1 package MyApp::FBP::MainFrame;
2
3 use 5.008005;
4 use utf8;
5 use strict;
6 use warnings;
7 use Wx 0.98 ':everything';
```

```
8
9  our $VERSION = '0.01';
10 our @ISA     = 'Wx::Frame';
11
12 sub new {
13     my $class  = shift;
14     my $parent = shift;
15
16     my $self = $class->SUPER::new(
17         $parent,
18         -1,
19         '',
20         wxDefaultPosition,
21         [ 500, 300 ],
22         wxDEFAULT_FRAME_STYLE | wxTAB_TRAVERSAL,
23     );
24
25     $self->{CloseButton} = Wx::Button->new(
26         $self,
27         -1,
28         "Close",
29         wxDefaultPosition,
30         wxDefaultSize,
31     );
32
33     Wx::Event::EVT_BUTTON(
34         $self,
35         $self->{CloseButton},
36         sub {
37             shift->OnClick(@_);
38         },
39     );
40
41     my $bSizer1 = Wx::BoxSizer->new(wxVERTICAL);
42     $bSizer1->Add( $self->{CloseButton}, 0, wxALL, 5 );
43
44     $self->SetSizer($bSizer1);
45     $self->Layout;
46
47     return $self;
48 }
49
50 sub OnClick {
51     warn 'Handler method OnClick for event CloseButton.
          OnButtonClick not implemented';
52 }
53
54 1;
```

Как же использовать эти классы? Добавлять в них функционал не стоит, потому как они автоматически генерируются из файла проекта. Наиболее правильным решением будет унаследоваться от сгенерированного класса и там уже имплементировать нужный код. Кроме того, имеет смысл генерировать классы с определенным префиксом, чтобы было понятно, какие классы автоматические, а какие написаны вручную.

Вот так выглядит реализация обработчика события `OnClick` в дочернем классе:

```
1  package MyApp::MainFrame;
2
3  use strict;
4  use warnings;
5
6  use base 'MyApp::FBP::MainFrame';
7
8  sub OnClick {
9      my $self = shift;
10
11     $self->Close(1);
12 }
13
14 1;
```

Проект с использованием сгенерированных классов выглядит следующим образом:

```
1 app.pl
2 lib/
3     MyApp/
4         FBP/
5             MainFrame.pm
6         MainFrame.pm
7     MyApp.pm
8 generate.pl
9 simple.fbp
```

`simple.fbp` это файл проекта wxFormBuilder, а `generate.pl` — скрипт для генерации.

Модуль `MyApp.pm` придется тоже написать вручную, но его код предельно прост:

```
1 package MyApp;
```

```
2
3 use strict;
4 use warnings;
5
6 use base 'Wx::SimpleApp';
7
8 use MyApp::MainFrame;
9
10 sub OnInit {
11     my $self = shift;
12
13     my $frame = MyApp::MainFrame->new;
14
15     $frame->Show;
16 }
17
18 1;
```

A `app.pl` остается тем же, что и в случае использования XML-представления.

Разделение логики

Как уже упоминалось, при написании графических приложений разделяется собственно представление и логика самого приложения. Идеально, если приложение будет реализовано таким образом, что замена графической библиотеки другой, или же веб-приложением, или консольным вариантом не повлечет каких-либо или, как минимум, никаких серьезных изменений.

Например, если в приложении необходимо сделать запрос к веб-сайту для получения каких-то данных, стоит выделить это в отдельный класс, не зависящий от графической библиотеки. Это позволит гораздо проще его протестировать стандартными средствами. В случае тестирования графических программ это сделать гораздо сложнее.

Впрочем, тестировать графические интерфейсы тоже можно и нужно, как например, с помощью Selenium тестируют и веб-приложения. Для X11 есть набор инструментов Xnee, который

позволяет записывать сценарии, а затем выполнять их в автоматическом режиме.

Блокировки

Если писать серьезные приложения на `wxWidgets`, в скором времени вы столкнетесь с тем, что при выполнении требующих времени задач активное окно блокируется и как будто замирает. Все дело в блокировке. Во время выполнения вашего кода `wxWidgets` ждет, пока он закончится.

Стандартным способом решения этой проблемы является использование тредов (потоков, нитей), когда задачи, которые необходимо выполнить на заднем фоне, выполняются в отдельном потоке. В этом случае окно не блокируется, а после завершения задачи основное окно получает ответ и обрабатывает его.

Кроме потоков возможно использование асинхронных фреймворков по типу `POE`. На Linux возможно использование `wxGTK`, а вместе с ним `AnyEvent`. В этом случае придется писать событийно-ориентированный код со всеми преимуществами и недостатками последнего.

Также возможно выполнение фоновых задач в отдельном процессе с помощью `wxExecute`, но это больше подходит для запуска сторонних приложений.

Выполнение задач в отдельном потоке

Самый распространенный и простой способ выполнить задачу в фоновом режиме — использовать потоки. Чтобы использовать потоки с `Wx`, необходимо, во-первых, убедиться, что `perl` собран с поддержкой `threads`:

```
1 perl -V | grep threads
```

А во-вторых, подключить прагму `threads` ДО подключения `Wx` (это стоит сделать также в скрипте запуска приложения):

```
use threads;
use Wx;
```

Например, при нажатии на кнопку, мы хотим выполнить какой-то код и оповестить главное приложение о результате. Скелет такого приложения будет выглядеть следующим образом:

```
package MyApp::MainFrame;

use strict;
use warnings;
use threads;
use threads::shared;

use base 'MyApp::FBP::MainFrame';

my $DONE_EVENT : shared = Wx::NewEventType;

sub OnClick {
    my $self = shift;

    Wx::Event::EVT_COMMAND($self, -1, $DONE_EVENT, \&done
       );

    threads->create(\&work, $self);
}

sub work {
    my $handler = shift;

    # do something

    my @result : shared = ...;

    my $threvent = Wx::PlThreadEvent->new(-1, $DONE_EVENT
       , \@result);
    Wx::PostEvent($handler, $threvent);

    threads->detach;
    threads->exit;
}

sub done {
    my $self = shift;
    my ($event) = @_;
```

```
37
38      my $result = $event->GetData;
39
40      # do something
41      ...
42  }
43
44  1;
```

Переменные с тегом `shared` необходимы для обмена данными между потоками, так как потоки выполняются одновременно, и использование обычных переменных небезопасно. `Wx` сам проследит, что переменные объявлены как `shared`, обезопасив таким образом программиста.

В этом приложении при нажатии на кнопку регистрируется обработчик завершения события, далее создается поток для выполнения работы. Когда поток завершается, он сигнализирует главному приложению о своих результатах. Главное приложение в обработчике, зарегистрированном ранее, получает данные и производит какие-то действия для отображения результата.

Проектируя код таким образом, т.е. исполняя код в фоновом режиме, можно писать обычные Perl-модули совершенно независимо от графической библиотеки.

Пример приложения

Для примера, напишем приложение, которое получает от MetaCPAN десять последних загруженных модулей и отображает их в виде списка.

Модель

Вначале реализуем модель, т.е. основу приложения. В нашем случае это модуль, который будет получать последние добавленные дистрибутивы. Он может выглядеть следующим образом:

```
1  package MyApp::ModulesFetcher;
2
3  use strict;
4  use warnings;
5
6  use JSON ();
7  use LWP::UserAgent;
8
9  my $METACPAN = 'http://api.metacpan.org/v0';
10
11 sub new {
12     my $class = shift;
13
14     my $self = {};
15     bless $self, $class;
16
17     return $self;
18 }
19
20 sub fetch {
21     my $self = shift;
22
23     my $ua = LWP::UserAgent->new;
24
25     my $response = $ua->post(
26         "$METACPAN/release/_search",
27         Content => JSON::encode_json(
28             {
29                 query => {
30                     filtered => {
31                         query  => {"match_all" => {}},
32                         filter => {
33                             term => {'release.authorized'
                                    => \1},
34                         }
35                     }
36                 },
37                 fields => ['distribution'],
38                 size => 10,
39                 from => 0,
40                 sort => [{date => 'desc'}],
41             }
42         )
43     );
44
45     return () unless $response->is_success;
46
```

```
47      my $res = JSON::decode_json($response->
           decoded_content);
48
49      my @distributions = @{$res->{hits}->{hits}};
50
51      return map { $_->{fields}->{distribution} }
           @distributions;
52  }
53
54  1;
```

Более подробно о том, как работать с API MetaCPAN, можно почитать в документации. Там же есть множество примеров.

Отображение

Графическое приложение будет представлять собой одно единственное окно с кнопкой `Fetch` и списком. При нажатии на кнопку в отдельном потоке создаем объект класса `ModulesFetcher` и возвращаем результат. Главное графическое приложение обрабатывает результат и заполняет список.

Скелет всего приложения уже описывался нами выше, поэтому здесь покажем только пример использования стороннего модуля:

```
1  package MyApp::MainFrame;
2
3  use strict;
4  use warnings;
5  use threads;
6  use threads::shared;
7
8  use base 'MyApp::FBP::MainFrame';
9
10 use MyApp::ModulesFetcher;
11
12 my $DONE_EVENT : shared = Wx::NewEventType;
13
14 sub OnClick {
15      my $self = shift;
16
17      Wx::Event::EVT_COMMAND($self, -1, $DONE_EVENT, \&done
           );
18
```

```
19      threads->create(\&work, $self);
20  }
21
22  sub work {
23      my $handler = shift;
24
25      my $fetcher = MyApp::ModulesFetcher->new;
26      my @modules = $fetcher->fetch;
27
28      my @result : shared = @modules;
29
30      my $threvent = Wx::PlThreadEvent->new(-1, $DONE_EVENT
            , \@result);
31      Wx::PostEvent($handler, $threvent);
32
33      threads->detach;
34      threads->exit;
35  }
36
37  sub done {
38      my $self = shift;
39      my ($event) = @_;
40
41      my $modules = $event->GetData;
42
43      my $list_ctrl = $self->{ModuleListCtrl};
44      $list_ctrl->ClearAll;
45
46      foreach my $module (reverse @$modules) {
47          $list_ctrl->InsertStringItem(0, $module);
48      }
49
50  }
51
52  1;
```

В методе `work` вызывается метод `fetch` на объекте модели, затем результат преобразуется в `shared`-переменную, и создается сигнал главному окну с передачей результата.

В методе `done` запрашиваем полученные данные у события с помощью метода `GetData` и затем в цикле добавляем их в объект списка. Перед этим не забываем очистить старые данные.

Выводы

Писать графические приложения на Perl можно вполне успешно. С помощью библиотеки `wxWidgets` это еще и просто. Правильно разделяя модель и отображение, можно добиться переносимости между различными графическими библиотеками, а также упростить реализацию.

Если читателям интересны подобные статьи, в планах есть написание аналогичного примера с помощью библиотеки `Gtk`. Жду ваших комментариев.

■ *Олеся Кузьмина*

4. Еще немного об асинхронном программировании на Anyevent

В журнале уже были статьи, посвященные асинхронному программированию на Perl. Статьи, бесспорно, весьма достойные и интересные, однако, к сожалению, они отвечают на вопрос “как?”, тогда как для лучшего понимания материала они должны отвечать на еще один вопрос “зачем?”.

У меня много спрашивают касательно `AnyEvent`, причем очень часто спрашивают такие вещи, которые, в принципе, на `AnyEvent` написать нельзя.

Я задался целью узнать, почему так происходит. Я считаю, что так происходит потому, что большинство руководств и обучающих материалов написаны для тех людей, которые знают, что такое асинхронное программирование и как оно работает. Я же постараюсь в рамках цикла статей рассказать о сути асинхронного программирования вообще, но примеры будут, естественно, на Perl.

И да, если у вас возникают какие-либо вопросы, мне можно написать, совершенно беспроблемно, на электронную почту, и я постараюсь объяснить непонятные моменты и отвечу на все вопросы.

Итак, поехали.

Историческая справка

Для того, чтобы понять, что оно такое и с чем его едят, нам необходим теоретический минимум. Забегая вперед, скажу, что абсолютной асинхронности, как таковой, вообще не существует. Существует асинхронность относительная.

Программирование вообще очень забавная вещь. Оно позволяет клиентам решать такие проблемы, о которых они не даже не догадывались, теми способами, о которых они тоже даже не догады-

ваются. Особенность программирования заключается в том, что это полностью искусственное знание, эмпирическим путем получить которое практически нереально, т.к. с окружающим миром оно имеет мало общего.

При классическом подходе к программированию обозначается цель, строится алгоритм, затем он декомпозируется на кусочки поменьше, и записывается реализация на определенном языке программирования. Как правило, приложение проектируется определенным образом. Я это написал лишь потому, что прежде чем я продолжу, я хотел бы ответить на вопрос: “Можно ли сделать так, чтобы мое приложение стало асинхронным без переписывания?”. Короткий ответ: “Нет”.

Если приложение проектировалось под синхронную модель, ничего с этим не сделаешь, асинхронным оно, увы, от замены пары строчек кода не станет.

Лирическое отступление

Синхронное программирование

Основой синхронного программирования есть одно простое слово: “Последовательность”. Все действия выполняются строго последовательно, в том порядке, в котором они записаны в исходном коде программы. В настольных приложениях, например, может использоваться многопоточный подход. Он используется для того, чтобы приложение могло делать несколько вещей одновременно. Например, один поток приложения считает сумму, а второй выводит надоедливые рекламные баннеры.

Процессор, например, двухъядерный, сферический и в вакууме, умеет одновременно выполнять *две* задачи и не задачей более. Но приложений запущено много. Как? А штука в том, что есть такой замечательный механизм — переключение контекста и очередь. Все задачи выстраиваются в очередь и процессор их обрабатывает по *две* одновременно. Программа, как правило, даже если выглядит

так:

```
for (my $i = 0; $i <= 10; $i++) {
    print "OMFG!!111\n";
}
```

То вовсе не обязательно, но вероятно, что все итерации цикла будут выполнены в рамках одной задачи. А переключение происходит настолько быстро, что зрительно это увидеть вообще нереально. Если мы напишем приложение, которое будет просто ждать пять секунд, а потом печатать “Hello world”, например, вот так:

```
sleep 5;
print "Hello world!\n";
```

То мы просто *ничего* не делаем пять секунд, а строчка “Hello world” будет напечатана только после того, как эти самые пять секунд пройдут.

Асинхронное программирование

Самым важным и основой основ асинхронного программирование является понятие блокировки. Для того, чтобы объяснить, что такое блокировка, рассмотрим простейший пример приложения, которое получает при помощи GET-запроса данные с узла http://pragmaticperl.com и немного теории:

```
#!/usr/bin/env perl
use strict;
use warnings;
use LWP::UserAgent;

my $url = "http://pragmaticperl.com/";
my $content = get_content();

sub get_content {
    my $url = shift;
    my $ua = LWP::UserAgent->new();

    my $response = $ua->get($url);
    return $response->content();
}
```

И еще рассмотрим такой пример:

```
#!/usr/bin/env perl
use strict;
use warnings;

print hello_world();

sub hello_world {
    return "Hello world!\n";
}
```

Итак, у нас есть два приложения и две функции. Это `get_content` и `hello_world`. Между ними есть очень важное отличие: результат работы функции `get_content` может быть разным, сервер упал, или интернета нет, например. Т.е. при вызове с одинаковыми параметрами несколько раз функция *может* вернуть разный результат. Функция же `hello_world` *всегда* возвращает “Hello world”. Такие функции имеют определенные названия. Итак, `get_content` — функция с побочными эффектами.

Функцией с побочными эффектами называется такая функция, которая может возвращать разный результат при вызове с одинаковыми параметрами, например, взаимодействуя с внешним миром.

Чистые функции же — такие функции, которые всегда возвращают одинаковый результат при вызове с одинаковыми параметрами. Например, синус — чистая функция, т.к. синус 90 градусов величина постоянная.

Так вот, асинхронное программирование позволяет реализовывать эффективные и производительные схемы только над функциями с побочными эффектами. Только. С. Побочными. Эффектами. `get_content` работает следующим образом. Он запрашивает внешний ресурс и ожидает его. Т.е. приложение не выполняет никакой полезной нагрузки, а занимается простоем. И вот здесь на помощь приходит асинхронное программирование. Пока мы ждем результатов извне, мы можем что-то сделать полезное.

Вообще, асинхронное выполнение можно организовать несколькими способами:

- Многопоточность
- Многопроцессность
- Событийность

Мы рассмотрим событийно-ориентированное программирование, как наиболее дешевую модель с точки зрения потребления ресурсов. Итак, еще одно теоретическое отступление:

Все мы знаем функцию `map`. `map` — функция высшего порядка. Функцией высшего порядка называется такая функция, которая принимает в качестве аргументов или возвращает в качестве результата другую функцию. В Perl функцию высшего порядка написать элементарно. Например:

```
mySub(sub {
    print "Hello world!\n";
});

sub mySub {
    my $sub = shift;
    $sub->();
}
```

В этом примере `mySub` — функция высшего порядка, т.к. принимает функцию в качестве первого и единственного аргумента. Кстати, `PSGI`-приложение тоже является функцией высшего порядка, т.к.

```
return sub {
    ...
}
```

Так вот, событийное программирование — это функции высшего порядка, да и еще с побочными эффектами. Есть такая штука — событийная машина, она генерирует события. И еще одно теоретическое отсутпление:

`callback`, он же обратный вызов, это действие, которое выполняется по возникновению какого-либо события. Например, запросить данные с `google.com`, а по получению данных вывести их на стандартный вывод.

Далее. Асинхронное событийно-ориентированное программирование — это функции высшего порядка с побочными эффектами, обратными вызовами поверх событийной машины.

Событийная машина работает следующим образом. У нее есть несколько стадий работы:

1. Запуск событийной машины.
2. Регистрация событий, наблюдателей.
3. Блокировка.
4. Выполнение.

Если хотя-бы одного элемента нет — ничего не будет. И вот теперь я могу объяснить, что такое блокировка и почему это важно.

Блокировка — особое состояние цикла событий. В этом состоянии цикл событий перестает принимать *новые* события, ожидая какого-либо события. Особенность его в том, что все события, которые были зарегистрированы ранее, выполняются. Т.е. весь код, который написан до строчки с блокировкой будет работать, там будут регистрироваться события, если надо, они будут выполняться и т.д; но вот дальше строчки с блокировкой выполнение не пойдет, т.к. ждем. В свою очередь блокировка является сигналом того, что можно начинать обрабатывать события. Безусловно, можно заблокировать весь цикл сразу, что он будет просто ждать, но тогда это классический синхронный подход и это явно не наш случай.

Суть асинхронного программирования. Суть асинхронного программирования в максимальном использовани имеющихся ресурсов и уменьшении времени простоя в целом, а, как следствие, суммарное время выполнения уменьшается.

Идем в бар

Например, возьмем в качестве примера два простых бара. В первом баре все синхронно, а во втором, соответственно, асинхронно. В первом баре бармен может наливать по одному стакану пива за раз. Не

важно, что есть свободные краны, не важно, что от бармена требуется поставить стакан, открыть кран и ждать. Это синхронный подход. Асинхронный подход — бармен ставит стакан, открывает кран, а пока стакан наливается, бармен ставит рядом, к другому крану, еще один стакан. И открывает чипсы, например. При синхронном подходе бармен будет делать все эти действия последовательно. И если каждое действие будет занимать 9 секунд, то в первом случае бармен потратит 27 секунд, как минимум, для того, чтобы выполнить все необходимые манипуляции. В асинхронном баре же эти действия займут, как минимум, 9 секунд, т.к. бармен не ждет, а что-то делает в фоне. Соответственно, мы выигрываем во времени, но проигрываем по нагрузке. За все надо платить, в асинхронном баре бармен больше устает, но у него и зарплата больше, которая зависит от количества налитых стаканов пива и открытых чипсов. Как-то так, да.

Негатив

Любой подход не лишен недостатков. Всякая асинхронщина не исключение. Так получилось, что форкнуть работающую событийную машину задача весьма нетривиальная, потому, как правило, асинхронные решения работают в один процесс и в один поток. Все вроде бы ок, но потенциальная проблема здесь следующая: Если какая-то чистая функция будет выполняться долго, то все будет ждать. Чистые функции не прерываются событийной машиной и не могут работать одновременно с циклом событий. Если неправильно спроектировать асинхронное приложение, то блокировок можно наделать везде и оно будет, как говорят про vim, бибикать и все портить. Отсюда следует очень важный факт, который игнорируется многими.

Важнейший абзац. Нельзя просто так взять и вернуть данные из callback. Они существуют в своем мирке со своими правилами, область видимости функции. Использовать глобальные переменные можно, но не всегда, а там где можно — нежелательно. Единственный способ получить данные в другом месте, вне callback — их туда отправить. Или же заблокироваться и поднять на уровень выше. В правильном асинхронном приложении данные внутри callback не должны влиять на данные, которые находятся на уровень выше. Ис-

ключение — чистые данные и чистые функции. Иногда ссылки на данные. Но очень осторожно.

Сложность отладки. Выношу отдельным абзацем. Если я скажу, что отлаживать большие асинхронные приложения сложно — я, наверное, буду не очень прав. Отлаживать асинхронные приложение вообще ад. Особенно, если оно неправильно спроектировано. К счастью, некоторые проблемы призван решить этот цикл статей. Существует еще особая уличная магия в виде блокировки внутри в callback с передачей параметров, мы рассмотрим этот вариант в следующих статьях цикла.

Потребление ресурсов. В виду того, что асинхронное приложение, если оно правильно спроектировано, практически не простаивает почем зря, существенно возрастает нагрузка на процессор. Если у вас слабое железо — профита не будет.

Резюмирую

Асинхронное решение вам подходит, если:

- приложение активно взаимодействует с внешним миром;
- приложение производит мало вычислительной работы;
- приложение постоянно работает с какими-либо БД;
- приложение чего-то ждет;
- приложение — HTTP-сервер.

Асинхронное решение вам не подходит, если:

- много матана у вас в приложении;
- лениво разбираться с асинхронными решениями;
- мало ресурсов.

Несколько советов:

- Если не надо блокироваться — не блокируйся.
- Если не знаешь, а надо ли здесь колбэк, оно не надо.
- Три действия по 30 мс каждое лучше, чем одно по 90 мс.
- Сломаться может где-угодно. Программирование на побочных эффектах требует перестраховываться везде.
- Если думаешь, нужна проверка или нет, оно же не сломается — проверка нужна, оно сломается.
- Порядок выполнения колбэков, если они не вложенные, определяет событийная машина, и она лучше программиста понимает, что надо делать.
- Не полагайся на порядок выполнения колбэков в приложении, он может всегда измениться.

Событийно-ориентированное программирование на Perl. AnyEvent.

Про AnyEvent уже ранее писалось в рамках данного журнала, потому на данном этапе я буду краток. AnyEvent — фреймворк, универсальный интерфейс для построения асинхронных приложений. Грубо говоря, он представляет собой DSL для написания асинхронных приложений, абстрагируясь от событийной машины и цикла событий. Т.е. AnyEvent позволяет с легкостью заменить один цикл событий другим, а приложение, если оно не использовало специфические возможности цикла событий, запустится без изменений кода. AnyEvent — DBI of event loop programming. Более подробно теорию по AnyEvent мы рассмотрим в следующих статьях. Сейчас же займемся практикой.

Немного кода

Как я и обещал, примеры кода будут на Perl. Для того, чтобы почувствовать вкус, мы напишем простое приложение для получения контента двух страниц с просторов интернета. Традиционно, первое приложение — синхронное, чтобы увидеть логику, второе приложение — переписанное на асинхронный лад первое.

```
1 #!/usr/bin/env perl
```

```
use strict;
use warnings;
use LWP::UserAgent;

my $urls = ['http://justnoxx.name/one.html', 'http://
    justnoxx.name/two.html'];
my $ua = LWP::UserAgent->new();

for my $url (@$urls) {
    print "GET request: $url\n";
    my $content = $ua->get($url)->content();
    print "Content: $content\n";
}
print "Done.\n";
```

Вывод данной программы будет иметь вид:

```
GET request: http://justnoxx.name/one.html
Content: one

GET request: http://justnoxx.name/two.html
Content: two

Done.
```

Совершенно очевидно и, к тому же, логично, что порядок работы программы не изменится. Допустим, если первый запрос будет занимать 10 с, а второй 1 с, то суммарно время работы скрипта будет примерно равно 11 секундам. Если второй запрос будет занимать 10 секунд, а первый 1, то суммарное время будет равно примерно 11 секундам. От перестановки слагаемых сумма не меняется. В данном случае асинхронное решение нам подходит идеально. Т. к:

- Мы работаем с внешним миром.
- У нас мало вычислений.
- Приложение много простаивает.

Асинхронное приложение будет выглядеть примерно так:

```
#!/usr/bin/env perl
use strict;
use warnings;

```

```
# Подключаем AE
use AnyEvent;
# Асинхронный юзер-агент
use AnyEvent::HTTP;

my $urls = ['http://justnoxx.name/one.html', 'http://
    justnoxx.name/two.html'];

# $cv — переменная состояния
my $cv = AnyEvent->condvar();
# количество запросов
my $count = 0;

for my $url (@$urls) {
    # вывод нам поможет понять, как именно, а что более
       важно, в каком порядке, выполняется приложение
    print "New event: GET => $url\n";
    # это очень важная строка. my $guard; $guard будет
       пояснено далее в тексте статьи
    my $guard; $guard = http_get(
        $url, sub {
            # см. предыдущий коммент
            undef $guard;
            my ($content, $headers) = @_;
            print "Content of $url: $content\n";
            $count++;
            # если количество успешных запросов равно
               размеру списка URLов, отправляем данные, на
                уровень выше.
            $cv->send("Done.\n") if $count == scalar
               @$urls;
        },
    );
}

# блокировка и ожидание. Собственной персоной.
my $result = $cv->recv();

print "$result\n";
```

А вывод вот так:

```
New event: GET => http://justnoxx.name/one.html
New event: GET => http://justnoxx.name/two.html
Content of http://justnoxx.name/two.html: two

Content of http://justnoxx.name/one.html: one

```

```
Done.
```

А теперь самое важное. Отличия, как и, самое главное, почему это работает.

Я выше писал, что приложение сначала набирает задания в очередь (`http_get`), а потом начинает выполнять. Это можно увидеть в выводе. Сначала задачи ставятся в очередь, а затем выполняются. Обратите внимание на то, что: `GET => http://justnoxx.name/one.html` был поставлен в очередь раньше, но первым выполнился `GET => http://justnoxx.name/two.html`.

`$cv` — переменная состояния. Если вызывать у нее метод `recv` — никакие новые события в очередь поступать не будут, т.к. цикл событый решит, и решит весьма справедливо, что дальнейшая работа без данных невозможна. `recv` ожидает данных, которые можно отправить при помощи метода send этой же переменной. В данном приложении, как только данные получены, блокировка снимается и приложение завершается. Однако, есть тут подводный камень. В одной области видимости может существовать только одна переменная состояния. Иначе цикл событий начинает конкретно паниковать, не понимая, что ему делать. Еще. Нельзя вызывать метод `recv` без `send`, или без событий. Вызов в скрипте `–>recv`, если это единственный вызов во всем скрипте вообще, приведет к массе ошибок и/или загрузке процессора на 100%, зависит от цикла событий.

```
my $guard; $guard = ... undef $guard;
```

Подобный прием не редкость. Он используется для того, чтобы сборщик мусора не утилизировал наш `callback` (увеличиваем счетчик ссылок на 1). Если мы убереи `undef $guard`, ничего работать не будет. Иногда говорят что мы “замыкаем переменную”. Кстати, я писал про замыкания ранее. Мы можем сделать иначе — объявить в глобальной области видимости переменную-массив и потом при помощи push поместить туда `$guard`, и тогда наш `callback` не будет утилизирован.

Затем мы выполняем send-метод, который отправляет на уровень выше результат (“Done.” в нашем случае), который принимает `–>recv`. Затем программа завершается.

И наконец, если мы уберем `$cv->recv();`, ничего работать не будет, т.к. нет блокировки. В принципе, для начала достаточно. В следующей статье я остановлюсь более подробно на этих моментах.

В следующей статье цикла мы рассмотрим теорию по `AnyEvent`, понятие относительной асинхронности, таймеры и концепцию наблюдателей.

Я надеюсь, что данная статья была интересна. До новых встреч.

■ *Дмитрий Шаматрин*

5. Обзор CPAN за октябрь 2014 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 242
- Новых выпусков — 899

Новые модули

- Proc::Fork::Control

Модуль `Proc::Fork::Control` представляет ещё один простой подход к созданию и управлению дочерними процессами. Модуль также позволяет контролировать максимальное количество создаваемых процессов и имеет встроенную процедуру для демонизации процессов.

- Mojo::Pg

Себастьян Ридель представил новый модуль `Mojo::Pg` для работы с базой данных PostgreSQL в веб-приложениях на фреймворке Mojolicious. Сам по себе модуль является тонкой надстройкой к `DBD::Pg`, которая автоматически контролирует все соединения с базой данных, кэшируя их для повторного использования, а также производит сброс соединений, если форкается новый процесс, позволяя прозрачно работать с одним объектом модуля во всех процессах.

`Mojo::Pg` позволяет выполнять как блокирующиеся, так и асинхронные запросы к базе данных, с тем лишь ограничением, что запросы выполняются последовательно в рамках одного соединения.

- Net::DNS_A

`Net::DNS_A` — это обёртка к функции стандартной библиотеки C `getaddrinfo_a`, которая является асинхронным аналогом функции `getaddrinfo`. Модуль позволяет выполнять асинхронное разрешение DNS-имён.

- Authen::SCRAM

`Authen::SCRAM` — это реализация современного стандарта RFC5802 SCRAM — механизма хранения данных и протокола аутентификации посредством пароля.

- Net::HTTP::Knork

Более легковесная реализация спецификации SPORE для описания REST API интерфейсов. В отличие от `Net::HTTP::Spore`, не имеет зависимости от `Moose` и возвращает результаты в виде объекта на базе стандартного класса `HTTP::Response`.

- Tree::BK

Реализация структуры данных BK-дерева для выполнения поиска с неточным соответствием. Подобная структура часто используется для проверки правописания слов по словарю.

```
1    use Tree::BK;
2    use utf8;
3
4    my $tree = Tree::BK->new();
5    $tree->insert_all(qw(Валя Вася Варя Катя Маша));
6    $tree->find('Ваня', 1); # Валя, Вася, Варя
7    $tree->find('Ваня', 2); # Валя, Вася, Варя, Катя
```

- Panda::Lib

Модуль `Panda::Lib` является коллекцией оптимизированных высокоскоростных функций, подходящих для использования как в Perl-, так и в XS-коде. В основном это функции для работы с хэшами и массивами: объединение, клонирование, сравнение.

Обновлённые модули

- perl 5.18.4

Выпущен четвёртый релиз исправлений для предпоследней стабильной ветки perl-5.18. Релиз 5.18.3 был выпущен с ошибкой и просуществовал лишь несколько часов. В новом релизе исправлено несколько ошибок, в том числе ошибка сегментации памяти в `Digest::SHA` и утечка памяти на платформе Win32 при использовании `system` или обратных кавычек.

- Mojo::Redis 1.00

Первый и последний мажорный релиз модуля для работы с Redis в веб-приложениях Mojolicious `Mojo::Redis` объявляет модуль устаревшим и рекомендует переходить на на `Mojo::Redis2`.

- ExtUtils::MakeMaker 7.00

Вышел новый мажорный релиз самой старой и по-прежнему популярной системы для сборки перл-модулей EUMM. Релиз содержит множество исправлений, например, наконец-то появилась возможность указывать каталог для инсталляции, имя которого содержит пробелы, поддерживаются опции в кодировке UTF-8, а также файлы и каталоги с символами в кодировке UTF-8. На платформе Windows появилась поддержка для GNU Make.

- autovivification 0.13

Обновлён модуль прагмы `autovivification`, позволяющий отключать автовивификацию в вашем коде. Автор обращает внимание, на то, что требуется тщательно протестировать этот релиз, поскольку потенциально возможно зависание на этапе компиляции. Новая версия должна заметно снизить общее замедление на этапе компиляции, которое вносит использование данной прагмы.

- JSON::PP 2.27300

Новый релиз модуля `JSON::PP` содержит исправление ошибки при разборе JSON на старых версиях perl ≤ 5.8.6.

- Panda::Export 2.1.0

`Panda::Export` — это альтернатива для модуля Exporter, написанная на C, которая позволяет на порядок ускорить импорт констант и функций в процессе компиляции. В новом релизе добавлена поддержка создания констант списком и увеличена скорость создания и импорта констант.

- Mojolicious 5.56

В этом месяце традиционно вышло множество релизов веб-фреймворка `Mojolicious`, включающие множество различных изменений и улучшений. Важным стоит отметить релиз 5.48, в котором было сделано исправление серьёзной уязвимости, позволяющей проводить инъекцию параметров. Всё это привело к поломке обратной совместимости: методы, которые раньше возвращали разные результаты в зависимости от контекста, теперь разделены, например, params/all_params, cookie/all_cookies и т.д. Подробности об атаке можно почитать здесь

В отличие от Mojolicious, другие фреймворки ограничились менее радикальными мерами: в Dancer добавили метод `param_array`, а Catalyst ограничился упоминанием в документации, что метод `params` устарел и требует осторожного использования.

- Syntax::Feature::Try 1.00

Вышел первый мажорный релиз реализации оператора try/catch/finally для Perl. В отличие от многих других реализаций он может обрабатывать ошибки по их ISA, поддерживает несколько catch-блоков и реализован на основе keyword/parser API (требует perl ≥ 5.14), не требует обязательной точки с запятой после завершающей фигурной скобки. В новом релизе была добавлена поддержка субтипов `Moose::Util::TypeConstraints` при обработке в catch.

- IO::Socket::SSL 2.002

Новый мажорный релиз `IO::Socket::SSL` можно было смело переименовать в `IO::Socket::TLS` поскольку поддержка последней версии протокола SSL 3.0 по умолчанию отключена. Это связано с публикацией информации об уязвимости в протоколе SSL 3.0 — POODLE. Также по умолчанию подключаются опции для улучшения PFS в протоколах шифрования.

- CSS::Sass 3.0.1

Новый мажорный релиз байндинга к библиотеке libsass для компиляции .sass/.scss-файлов. Новая версия соответствует последней мажорной версии библиотеки libsass.

- Web::Scraper 0.38

Неожиданно после двухлетнего перерыва обновился модуль для разбора HTML-страниц с помощью XPath-выражений `Web::Scraper`. В обновлённой версии включены давно забытые в баг-трекере улучшения в документации и фильтры на основе регулярных выражений. Молодец, Миягава, лучше поздно, чем никогда.

■ *Владимир Леттиев*

6. Интервью с Еленой Большаковой

Елена Большакова (Liruoko) — Perl-программист, математик по образованию, соавтор проекта http://perltrap.com.

Как и когда научились программировать?

Поздно и случайно. В девятом классе на информатике был какой-то условный алгоритмический язык, переменные-циклы-условия-блок-схемы. Программы писали в тетрадках, компьютеров, которые могли бы выполнять эти программы, не было. Совсем не интересно.

Летом услышала рассказ одной девочки, как она писала программу для рисования маятника, и как это было сложно. Хотя казалось бы, что такого? Уравнение движения известно из физики.

В десятом классе (уже в другой школе) были компьютеры и начался Бейсик: line, circle, goto. Ага! — теперь я тоже могу делать мультики. Осталась после уроков и запрограммировала качающийся маятник. Кстати, самым сложным оказалось подобрать интервалы перерисовки кадров, чтобы картинка не моргала.

С этого момента программировать стало интересно, вроде как читать Жюль Верна.

После школы — факультет ВМиК МГУ, и там все было серьезно: алгоритмы, структуры данных, оценки сложности, доказательства корректности, методы оптимизации. У меня был собственный sql-сервер, компилятор С-подобного языка в байт-код и его интерпретатор, шелл, Элиза-подобная программа-собеседник. В общем, все было очень здорово.

Но к пятому курсу меня посетила идея, что программирование — слишком сиюминутное занятие: сегодня ты программируешь протокол X под архитектуру Y, а завтра протокол устарел и архитектура мертва. А мне хотелось вечности. И я пошла в аспирантуру на мехмат, «делать математику». Кстати, доказала красивые результаты и защитила диссертацию.

Какой редактор используете?

Vim. Он везде есть и одинаково удобен и при локальном использовании, и на удаленных серверах.

Кстати, а почему не спрашиваете про клавиатуру? Для комфортной работы с текстом хорошая клавиатура важна почти так же, как хороший редактор. У меня две любимые клавиатуры: IBM Model M и Code Keyboard.

Как и когда познакомились с Perl?

Когда была в аспирантуре, знакомый системный администратор рассказывал про свои скрипты и про Perl — язык, похожий на человеческий. Язык, в котором одно и то же можно написать по-разному, и где можно сказать: сделай или умри.

Потом однажды весной наткнулась в Википедии на перловый однострочник, в котором коротким регулярным выражением, состоящим почти из одних единиц, выводился бесконечный список простых чисел. Вот это да! Хочу знать больше. Тот же знакомый сисадмин (к тому времени мы поженились) подсунул Camel Book. Книга очень понравилась с первых страниц, я решила, что это мое, и написала резюме в Яндекс. Тем же летом я вышла на работу в Директ, Perl-разработчиком.

С какими другими языками интересно работать?

Нравится C — он прямой и честный. Еще очень забавный Forth: минималистичный до крайности, но при этом настоящий промышленный язык, не brainfuck какой-нибудь.

Но на самом-то деле мне интересно не с языками работать, а задачи решать. Сейчас мне хватает достойных задач, а вообще было бы интересно заняться чем-нибудь с формальными грамматиками. Или со статистикой. Еще очень увлекательная область — вероятностные структуры данных.

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

Perl очень пластичный и выразительный, идеально подходит для трансляции мыслей в машинно-читаемый формат. И на нем очень легко начать писать. Я слышала такой рассказ от коллеги: «Друг поставил Perl на компьютер, я наугад написал ‘print “hello” ’ — а оно заработало. Так я и попался».

Еще на Perl можно писать с разной степенью аккуратности, и это удобно. Иногда надо быстро набросать скрипт, хотя бы и грязноватый, а иногда такой набросок превращается потом в продакшен-систему. Perl хорошо подходит для подобных превращений. Главное — и это очень важно — в каждый момент понимать, насколько педантичный код уместен прямо сейчас. Об этом моя статья про постепенную автоматизацию.

И, конечно, сообщество. Perl собирает вокруг себя замечательных людей и специалистов.

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

Давайте расскажу страшилку: в будущем новые языки будут появляться чаще и умирать быстрее. Новая мобильная или социальная платформа — новый язык, ушла платформа — ушел язык. Никаких стабильных версий, каждые полгода базовый синтаксис радикально меняется. Динамика и драйв, революция каждый день.

Что думаете о будущем Perl?

Судьба Perl зависит от сообщества, то есть и от нас с вами. Это большая ответственность, но это и здорово!

Помогает ли математическое образование в программировании?

А что мы считаем «математическим образованием»? Вызубрить тысячу теорем, сдать пятьдесят зачетов и тридцать экзаменов, а потом забыть все, как страшный сон? По-моему, такое образование не помогает само по себе, но работает как отбор: смог? осилил? Значит, силён.

Если же действовать не по стандартному студенческому «знал, сдал,

забыл», а полноценно участвовать в математических исследованиях: работать в научном семинаре, вникнуть в современные результаты в какой-нибудь области, построить контрпример к актуальной гипотезе, доказать что-нибудь новенькое, опубликоваться в хорошем журнале — да, такой математический background полезен для программистов.

Знаете, у компьютерщиков в запасе есть негодные оправдания: «Работает ведь! Так что неважно, насколько аккуратно сделано, эффективно ли, надежно ли». У математиков же нет никаких машин, которые бы воспринимали доказательства. Хочешь утвердить новый результат — убеди людей, как три тысячи лет назад, как у древних греков. Вот эту культуру обсуждения, поиска слабых мест, контраргументов полезно освоить и разработчику.

Вас всегда можно увидеть на различных Perl-мероприятиях, чем они вас привлекают?

Интересно слушать людей из других компаний и проектов.

У нас у всех есть общее — Perl — но задачи и проблемы очень разные. Во время некоторых докладов я думаю «да с таким подходом мы и недели не выжили бы», на других — «смотри-ка, и у них так же», а иногда — «о, вот это интересно, надо изучить подробнее».

В целом, чужой опыт помогает принимать лучшие решения в своей работе.

Расскажите про http://perltrap.com.

Давным-давно мне попалась книжка Стива Уэллина «Как не надо программировать на C++». Там приводятся фрагменты программ, и в каждом есть проблема — иногда более заковыристая, иногда менее. К каждой проблеме дается серия подсказок, обычно довольно ехидных, и наконец объяснение.

Мне кажется, эта книга — про самую суть отладки. Ведь в конечном счете все сводится к тому, что ты смотришь на исходный код и в какой-то момент понимаешь, что именно с ним не так. Навороченные дебаггеры и многочасовое пошаговое отслеживание данных —

подготовка к моменту озарения, но сам инсайт всегда связан с внимательным анализом кода.

Книга Уэллина дает отличную возможность попрактиковаться в таком критическом чтении. Это интересно и полезно — и для самоконтроля, и для код-ревью.

Потом, уже в Яндексе, я подумала, что с удовольствием читала бы такую книгу-задачник про Perl, только ее почему-то никто не писал. Если никто другой, тогда, может быть, я? К тому же нашлись коллеги, которым тоже понравилась идея, мы зарегистрировали имя, сделали сайт и стали выкладывать задачи. Вот и все.

Как известно, вы работаете в Яндексе уже продолжительное время. Сколько времени проводите за написанием Perl-кода?

Меньше, чем хотелось бы.

Сейчас я руковожу группой инфраструктурной разработки в Директе. С одной стороны, порядочное время занимает планирование, проектирование, ревью, консультации, а с другой — у нас специфические задачи. Мы обеспечиваем разработчиков, администраторов, тестировщиков, менеджеров эффективными и надежными инструментами. Для этого не требуется много кода, но всегда требуется тщательный анализ, чтобы направить усилия самым выгодным образом.

По крупному счету все хорошо, но бывает, я чувствую себя как в анекдоте: «Дома говорю, что иду на работу, на работе пишу, что задерживаюсь, а сам в парк на скамеечку, достаю ноутбук, и программирую, программирую...»

Стоит ли советовать молодым программистам учить сейчас Perl?

Обязательно надо показывать молодым программистам, какие их задачи легко решаются с помощью Perl.

На уровне однострочников `perl -lane '$sum{$F{3}} += $F[1] END{print "$_ $sum{$_}" for keys %sum}'` Perl нужен всем, наравне с grep-ом и make-ом.

А делать ли Perl основной специальностью или оставить инструментом на подхвате — каждый потом разберется сам.

Вопросы от читателей

Будут ли еще модули на CPAN?

Если буду писать что-то общественно-полезное — буду выкладывать.

Согласны ли со мнением, что вы «самый крутой перловик Яндекса»? :)

А почему только Яндекса? ^_^

Если серьезно, я считаю своей самой сильной стороной юзабилити command-line-инструментов, я умею делать так, чтобы разработчики и администраторы ошибались реже. А еще я могу брать что-нибудь сложное — программу, систему, идею, текст, процесс — и делать эту вещь проще. Вот!

Почему перестали преподавать?

Это не так. Просто перешла от серийного обучения в аудитории к штучному кураторству над стажерами в Яндексе. И еще пишу статьи для PragmaticPerl ^_^

Как устроиться работать в Яндекс?

- Во-первых, захотеть.
- Во-вторых, отправить резюме.
- В-третьих, хорошо показать себя на собеседованиях.

Первые два пункта просты: если нет желания — то и говорить не о чем, а резюме — механическая работа. Что касается собеседований… В разных командах и для разных позиций требования отличаются, но определенно полезно логически мыслить, уметь работать над задачей, которая не решается с первого взгляда, чувствовать сложность алгоритомов, понимать unix-like-системы — форки, процессы, сокеты, все такое. Не помешают базы данных.

Удачи!

Спасибо журналу «Pragmatic Perl» за приглашение, это было неожиданно и приятно.

■ *Вячеслав Тихановский*