helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

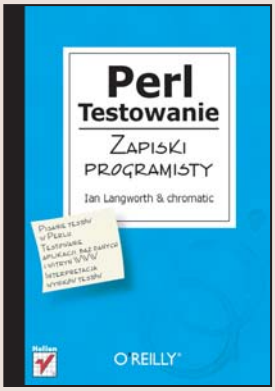
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Perl. Testowanie. Zapiski programisty



Autorzy: Ian Langworth, chromatic
Tłumaczenie: Maja Królikowska
ISBN: 83-246-0240-2
Tytuł oryginału: Perl Testing: A Developers Notebook
Format: B5, stron: 240

Testowanie aplikacji to temat najczęściej pomijany przez programistów. Testowanie nie jest tak pasjonujące jak tworzenie programów czy poznawanie nowych narzędzi. Jest jednak niezbędne. Prawidłowo przeprowadzony proces testowania może znacznie poprawić wydajność, podnieść jakość projektu i kodu, zmniejszyć obciążenia wynikające z konserwacji kodu i pomóc lepiej zaspokoić wymagania klientów, współpracowników i kierownictwa. W powszechnie uznanych metodykach projektowych testowanie, szczególnie za pomocą testów automatycznych, jest niezwykle istotnym procesem.

Książka „Perl. Testowanie. Zapiski programisty" to praktyczny przewodnik dla programistów Perla, którzy chcą poprawić jakość i wydajność tworzonych przez siebie programów. Opisuje metody tworzenia testów automatycznych, stosowania ich i interpretowania ich wyników. Przedstawia sposoby testowania pojedynczych modułów, całych aplikacji, witryn WWW, baz danych, a nawet programów stworzonych w innych językach programowania. Zawiera również informacje o tym, jak dostosować podstawowe narzędzia testujące do własnego środowiska i projektów.

- Instalowanie modułów testujących
- Pisanie testów
- Automatyzacja uruchamiania testów
- Analiza wyników testów
- Dystrybucja testów
- Testy jednostkowe
- Testowanie baz danych
- Testowanie witryn WWW i kodu HTML

Dzięki wiadomościom zawartym w tej książce można zredukować długość cyklu tworzenia oprogramowania i zdecydowanie ułatwić konserwację gotowych systemów.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Seria „Zapiski programisty" **7**

Przedmowa **13**

Rozdział 1. Początki testowania **21**

Instalowanie modułów testujących 21
Uruchamianie testów 25
Interpretacja wyników testów 28
Pisanie pierwszego testu 31
Wczytywanie modułów 34
Ulepszanie porównań w testach 38

Rozdział 2. Pisanie testów **43**

Pomijanie testów 43
Pomijanie wszystkich testów 46
Oznaczanie testów jako „do zrobienia" 48
Porównywanie prostych struktur danych 51
Złożone struktury danych 56
Testowanie ostrzeżeń 60
Testowanie wyjątków 63

Rozdział 3. Zarządzanie testami 67

Organizowanie testów 67

Sprawdzanie pokrycia kodu 71

Pisanie biblioteki testującej 78

Testowanie biblioteki testującej 81

Pisanie systemu uruchamiania z testowaniem 84

Testowanie w sieci 86

Automatyzacja uruchamiania testów 88

Rozdział 4. Dystrybuowanie testów (i kodu) 93

Testowanie plików POD 93

Testowanie pokrycia dokumentacją 95

Podpisywanie dystrybucji 98

Testowanie całych dystrybucji 101

Pozwól użytkownikowi decydować 103

Pozwól użytkownikowi decydować (ciąg dalszy) 106

Umieszczanie testów w dystrybucji modułów 107

Pobieranie wyników testów 110

Sprawdzanie poprawności Kwalitee 114

Rozdział 5. Testowanie nietestowalnego kodu 117

Zastępowanie operatorów i funkcji wbudowanych 118

Imitowanie modułów 123

Imitowanie obiektów 127

Częściowe imitowanie obiektów 133

Zastępowanie kodu 138

Zastępowanie operatorów 142

Rozdział 6. Testowanie baz danych 147

Dostarczanie testowych baz danych 147

Testowanie danych w bazie danych 151

Używanie tymczasowych baz danych 156

Imitowanie baz danych 161

Rozdział 7. Testowanie witryn WWW 167

Testowanie zaplecza aplikacji 167

Testowanie widocznej części aplikacji 173

Nagrywanie i odtwarzanie sesji przeglądarki 176

Testowanie poprawności HTML 180

Uruchamianie własnego serwera Apache 182

Testowanie za pomocą Apache-Test 185

Dystrybuowanie modułów z Apache-Test 191

Rozdział 8. Testy jednostkowe przeprowadzane za pomocą Test::Class 195

Pisanie przypadków testowych 196

Tworzenie środowiska testu 200

Dziedziczenie testów 203

Pomijanie testów przy użyciu Test::Class 206

Oznaczanie testów jako „do zrobienia" przy użyciu Test::Class 208

Rozdział 9. Testowanie całej reszty 211

Pisanie testowalnych programów 211

Testowanie programów 215

Testowanie programów interaktywnych 218

Testowanie bibliotek współdzielonych 221

Skorowidz 225

ROZDZIAŁ 2.

Pisanie testów

Perl ma bardzo bogatą składnię, ale wiele rzeczy daje się zrobić, wykorzystując tylko ułamek jego możliwości. Przykładowo: Perl oferuje ciągle zwiększającą się liczbę modułów do testowania, a także najlepszych praktyk w tym zakresie, ale wszystko to zbudowano wokół funkcji `ok()` opisanej w poprzednim rozdziale.

Ćwiczenia przedstawione w niniejszym rozdziale prowadzą przez zaawansowane funkcje `Test::More` i innych często używanych modułów testujących. Nauczymy się tutaj, jak i w jakim celu kontrolować uruchamianie testów oraz jak efektywnie porównywać dane wynikowe z oczekiwanymi i jak testować warunki wyjątkowe. Są to bardzo istotne techniki — budulec pozwalający na pisanie wszechstronnych zestawów narzędzi do testowania.

Pomijanie testów

Niektóre testy powinny być uruchamiane tylko w szczególnych przypadkach. Przykładem może być test połączenia z zewnętrzną usługą, bo ma on sens tylko wtedy, gdy komputer jest podłączony do internetu, lub test, który zależy od systemu operacyjnego. Poniższe ćwiczenie pokazuje, jak pominąć testy, o których wiadomo, że się nie powiodą.

Jak to osiągnąć?

Przyjmijmy, że piszemy program tłumaczący z języka angielskiego na holenderski. Klasa `Phrase` będzie przechowywać pewien tekst i będzie udostępniać konstruktor, akcesor (funkcję ustawiającą wartość zmiennej klasy) oraz metodę `as_dutch()`, zwracającą tekst przetłumaczony na język holenderski.

Następujący kod zapisz w pliku *Phrase.pm*:

```
package Phrase;
use strict;

sub new
{
   my ( $class, $text ) = @_;
   bless \$text, $class;
}

sub text
{
   my $self = shift;
   return $$self;
}

sub as_dutch
{
   my $self = shift;
   require WWW::Babelfish;
   return WWW::Babelfish->new->translate(
      source   =>   'English',
      destination => 'Dutch',
      text => $self->text(),
   );
}

1;
```

Użytkownik nie musi mieć zainstalowanego modułu do tłumaczenia `WWW::Babelfish`. Ustalamy, że funkcja `as_dutch()` w klasie `Phrase` jest opcjonalna. Jak jednak ją przetestować?

Następujący kod umieść w pliku *phrase.t*:

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

use strict;

use Test::More tests=>3;
use Phrase;
```

```
my $phrase = Phrase->new('Good morning!');
isa_ok( $phrase, 'Phrase');

is( $phrase->text(), 'Good morning!', "akcesor text() działa" );

SKIP:
{
   eval 'use WWW::Babelfish';

   skip ('WWW::Babelfish jest wymagane w funkcji as_dutch()', 1 ) if $@;

   is(
      $phrase->as_dutch,
      'Goede ochtend!',
      "udane tłumaczenie na holenderski"
     );
}
```

Gotowy test uruchom za pomocą *prove* z opcją `-v`. Jeśli masz zainstalowany moduł `WWW::Babelfish`, to *prove* wypisze, co następuje:

```
$ prove -v phrase.t
phrase....1..3
ok 1 - The object isa Phrase
ok 2 - akcesor text() działa
ok 3 - udane tłumaczenie na holenderski
ok
All tests successful.
Files=1, Tests=3,  5 wallclock secs ( 0.49 cusr +  0.06 csys =  0.55 CPU)
```

Jeśli jednak brak jest `WWW::Babelfish`, to wynik będzie inny:

```
$ prove -v phrase.t
phrase....1..3
ok 1 - The object isa Phrase
ok 2 - akcesor text() działa
ok 3 # skip WWW::Babelfish jest wymagane w funkcji as_dutch()
ok
        1/3 skipped: WWW::Babelfish jest wymagane w funkcji as_dutch()
All tests successful, 1 subtest skipped.
Files=1, Tests=3,  1 wallclock secs ( 0.09 cusr +  0.01 csys =  0.10 CPU)
```

Jak to działa?

Plik z testem zaczyna się od deklaracji `Test::More`, podobnie jak to było przy poprzednich ćwiczeniach, a potem tworzony jest przykładowy obiekt klasy `Phrase`, testowany jego konstruktor oraz akcesor `text()`.

Aby pominąć testowanie funkcji `as_dutch()` w sytuacji, gdy użytkownik nie ma zainstalowanego modułu `WWW::Babelfish`, należy użyć specjalnej

Liczba bloków kodu z etykietą SKIP może być dowolna, zależnie od potrzeb. Bloki można także zagnieżdżać, pod warunkiem, że każdy zagnieżdżony blok będzie również opatrzony etykietą SKIP.

składni. Test `as_dutch()` zawarty jest w jednym bloku kodu oznaczonym etykietą `SKIP`. Blok ten zaczyna się od próby załadowania modułu `WWW::Babelfish`.

Jeśli próba wykorzystania modułu `WWW::Babelfish` się nie powiedzie, `eval` przechwyci błąd i umieści go w zmiennej globalnej `$@`, w przeciwnym wypadku wyczyści jej zawartość. Jeśli w zmiennej `$@` jest zapisana jakaś wartość, to wykona się instrukcja umieszczona w następnym wierszu. Wykorzystano w niej kolejną funkcję eksportowaną przez `Test::More`, `skip()`. Funkcja ta ma dwa argumenty: powód pominięcia testu oraz liczbę testów do pominięcia. W naszym przypadku pomijany jest jeden test, a wyjaśnieniem jest niedostępność opcjonalnego modułu.

Test::Harness uznaje wszystkie pominięte testy za sukcesy, bo to jest zachowanie, którego oczekujemy.

Test funkcji `as_dutch()` nie został uruchomiony, ale zaliczono go do testów poprawnie zakończonych, bowiem został oznaczony jako test do pominięcia. Znaczy to, że oczekujemy, że test **nigdy** się nie powiedzie, jeśli nie zostaną spełnione pewne warunki. Jeśli `WWW::Babelfish` **byłby** dostępny, to test przebiegłby normalnie i został zaliczony do poprawnie zakończonych, tak jak to jest w przypadku każdego innego testu.

Pomijanie wszystkich testów

Poprzednie ćwiczenie pokazuje, jak można pominąć konkretny test w określonych okolicznościach. Zdarzają się jednak przypadki, w których cały **plik** z testami nie powinien być uruchamiany. Na przykład testowanie funkcjonalności specyficznej dla platformy X nie da żadnych znaczących wyników na platformie Y. Dla takiego przypadku `Test::More` także dostarcza użytecznej składni.

Jak to osiągnąć?

Zamiast podawać liczbę testów w `use()`, należy osobno użyć funkcji `plan`. Następujący kod sprawdza, czy bieżącym dniem tygodnia jest wtorek i jeśli nie jest, to wszystkie testy zostają pominięte. Zapisz go w pliku *skip_all.t*:

```
use Test::More;

if ([ localtime ]->[6] != 2)
{
```

```
    plan( skip_all => 'te testy uruchamiane są tylko we wtorki' );
}
else
{
    plan( tests=>1 );
}

require Tuesday;
my $day = Tuesday->new();
ok( $day->coat(), 'wzięliśmy nasze palto');
```

Tuesday.pm jest bardzo prosty:

```
package Tuesday;

sub new
{
    bless {}, shift;
}

# noś palto tylko we wtorki
sub coat
{
    return [ localtime ]->[6] == 2;
}

1;
```

W prawdziwym pliku testowym byłoby więcej testów; tutaj pokazujemy tylko przykład.

Uruchom plik testujący we wtorek, a zobaczysz następujący wynik:

```
$ prove -v skip_all.t
skip_all....1..1
ok 1 - wzięliśmy nasze palto
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.14 cusr +  0.02 csys =  0.16 CPU)
```

Żeby pominąć wszystkie testy, uruchom plik testujący innego dnia tygodnia:

```
$ prove -v skip_all.t
skip_all....1..0 # Skip te testy uruchamiane są tylko we wtorki
skipped
        all skipped: te testy uruchamiane są tylko we wtorki
All tests successful, 1 test skipped.
Files=1, Tests=0,  0 wallclock secs ( 0.10 cusr +  0.01 csys =  0.11 CPU)
```

Jak to działa?

Zamiast od razu podawać plan testów za pomocą przekazywania dodatkowych argumentów do `use`, w *skip_all.t* do ustalenia planu uruchomienia skryptu wykorzystana została funkcja `plan()` z modułu `Test::More`.

Jeśli bieżącym dniem tygodnia nie jest wtorek, to wywoływany jest `plan()` z dwoma argumentami: instrukcją niewykonywania jakichkolwiek testów oraz wyjaśnieniem, dlaczego tak ma się stać. Jeśli **jest** wtorek, kod zgłasza normalny plan testów i wszystko się wykonuje tak jak zwykle.

Oznaczanie testów jako „do zrobienia"

Jakkolwiek posiadanie dobrze przetestowanego podstawowego kodu ma szansę przyspieszyć powstawanie oprogramowania, to jednak czasem program znajduje się w stanie, w którym brakuje mu części funkcjonalności lub ma znane, ale jeszcze niepoprawione błędy. Wygodnie byłoby wyłapać tę informację w testach, które na pewno się nie udadzą, bowiem kod, który mają testować, jeszcze nie powstał. Na szczęście takie zadania można oznaczyć jako „do zrobienia", co sugeruje, że sprawdzenia będą wykonywane, ale zarządzanie ich wynikami będzie się odbywało w inny sposób.

Jak to osiągnąć?

Oto dobry pomysł na kod: moduł, który czyta przyszłe wersje plików. Może być naprawdę użyteczny. Nazwij go `File::Future`, a jego kod zapisz w pliku *File/Future.pm* (najpierw tworząc katalog *File*, jeśli go nie ma):

```
package File::Future;

use strict;

sub new
{
   my ($class, $filename) = @_;
   bless { filename => $filename }, $class;
}

sub retrieve
{
   # do zaimplementowania później...
}

1;
```

Konstruktor `File::Future` jako argument przyjmuje ścieżkę do pliku i zwraca obiekt. Wywołanie `retrieve()` z daną datą zwróci plik w danym terminie. Niestety, nie ma jeszcze rozszerzenia do Perla do kondensatorów strumienia. Na razie wstrzymujemy się z implementacją `retrieve()`.

Nie ma jednak sensu nie testować tego kodu. Dobrze by było wiedzieć, że robi on to, co powinien robić, bo może kiedyś, na Gwiazdkę, pojawi się w końcu moduł `Acme::FluxFS`. Co więcej, testowanie tej funkcji jest łatwe. Następujący kod zapisz w pliku *future.t*:

```
use Test::More tests=>4;
use File::Future;

my $file = File::Future->new( 'perl_testing.dn.pod' );
isa_ok( $file, 'File::Future' );

TODO: {
   local $TODO = 'continuum nie daje się jeszcze testować';

   ok( my $current = $file->retrieve( '30 stycznia 2005' ) );
   ok( my $future  = $file->retrieve( '30 stycznia 2070' ) );

   cmp_ok( length($current), '<', length($future),
      'zapewne dodamy trochę tekstu do 2070 roku' );
}
```

Uruchom test za pomocą *prove*. Wynik będzie następujący:

```
$ prove -v future.t
future....1..4
ok 1 - The object isa File::Future
not ok 2 # TODO continuum nie daje się jeszcze testować

#     Failed (TODO) test (future.t at line 10)
not ok 3 # TODO continuum nie daje się jeszcze testować

#     Failed (TODO) test (future.t at line 11)
not ok 4 - zapewne dodamy trochę tekstu do 2070 roku # TODO continuum nie
daje się jeszcze testować

#     Failed (TODO) test (future.t at line 13)
#     '0'
#         <
#     '0'
ok
All tests successful.
Files=1, Tests=4,  1 wallclock secs ( 0.13 cusr +  0.01 csys =  0.14 CPU)
```

Jak to działa?

W odróżnieniu od testów pominiętych, te oznaczone jako „do zrobienia" są naprawdę uruchamiane. Jednakże, inaczej niż w przypadku normalnych testów, system uruchamiania z testowaniem interpretuje je jako udane, nawet jeśli nie uda się ich

W pliku testowym dla `File::Future` zaznaczono testowanie pobierania dokumentów z przyszłości jako niedokończoną, ale planowaną funkcjonalność.

Żeby oznaczyć zestaw testów jako „do zrobienia", trzeba je umieścić w bloku z etykietą `TODO`, podobnie jak to było w przypadku bloku `SKIP` w podrozdziale „Pomijanie testów" wcześniej w tym rozdziale. Zamiast jednak wykorzystywać funkcję podobną do `skip()`, trzeba użyć zmiennej `$TODO` i przypisać do niej powód, dla którego testy nie powinny się udać.

Warto zauważyć w podanym wyniku, że `Test::More` oznaczył testy jako TODO (do zrobienia), podając powód, dla którego nie są uruchamiane. Testy się nie udają, ale ponieważ w pliku testowym zapisano, że jest to zachowanie oczekiwane, system uruchamiania z testowaniem traktuje je jako zakończone sukcesem.

A co...

...się stanie, jeśli testy się powiodą? Jeśli na przykład test sprawdza jakiś błąd, który zostanie przez kogoś poprawiony w trakcie naprawiania czegoś innego, to co się zdarzy?

Jeśli testy oznaczone jako TODO („do zrobienia") naprawdę zakończą się sukcesem, to system uruchamiania z testowaniem zgłosi, że niektóre testy niespodziewanie się udały:

```
$ prove -v future-pass.t
future-pass....1..4
ok 1 - The object isa File::Future
ok 2 # TODO continuum nie daje się jeszcze testować
ok 3 # TODO continuum nie daje się jeszcze testować
ok 4 # TODO continuum nie daje się jeszcze testować
ok
        3/4 unexpectedly succeeded
All tests successful (3 subtests UNEXPECTEDLY SUCCEEDED).
Files=1, Tests=4,  0 wallclock secs ( 0.11 cusr +  0.03 csys =  0.14 CPU)
```

Jest dobrze. Można teraz przenieść udane testy poza blok `TODO` i uznać je za pełnoprawne testy, które powinny zawsze kończyć się sukcesem.

Porównywanie prostych struktur danych

Funkcja `is()` z modułu `Test::More` sprawdza, czy dwie wartości skalarne są równe, ale co zrobić z bardziej skomplikowanymi strukturami takimi jak listy lub listy list? Dobre testy często wymagają sprawdzenia, czy dwie struktury są naprawdę identyczne w sensie równości każdego z ich elementów. Pierwsze rozwiązanie, które przychodzi do głowy, to funkcja rekursywna lub kilka zagnieżdżonych pętli. Warto się jednak wstrzymać — `Test::More` i inne moduły do testowania zrobią to lepiej swoimi funkcjami porównującymi.

Jak to osiągnąć?

Następujący kod zapisz w pliku *deeply.t*:

```
use Test::More tests => 1;
my $list1 =
[
   [
      [ 48, 12 ],
      [ 32, 10 ],
   ],
   [
      [ 03, 28 ],
   ],
];
   my $list2 =
[
   [
      [ 48, 12 ],
      [ 32, 11 ],
   ],
   [
      [ 03, 28 ],
   ],
];
```

Uruchom go za pomocą `prove -v`, a zobaczysz następujące komunikaty diagnostyczne:

```
$ prove -v deeply.t
deeply....1..1
not ok 1 - równoważność egzystencjalna

#     Failed test (deeply.t at line 25)
#     Structures begin differing at:
#          $got->[0][1][1] = '10'
#     $expected->[0][1][1] = '11'
```

```
# Looks like you failed 1 test of 1.
dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
deeply.t       1   256     1    1 100.00%  1
```

Jak to działa?

Przykładowy test porównuje dwie listy list za pomocą funkcji `is_deeply()` wyeksportowanej przez moduł `Test::More`. Warto zwrócić uwagę na różnicę między listami. Ponieważ w drugiej tablicy umieszczono wartość `11` tam, gdzie w pierwszej jest `10`, to test zakończył się porażką.

Wynik testu pokazuje też różnicę pomiędzy `$list1` a `$list2`. Jeśli w strukturach danych występuje wiele różnic, to `is_deeply()` wypisze tylko pierwszą. Co więcej, jeśli w jednej ze struktur danych brakuje elementu, to `is_deeply()` też to pokaże.

A co...

...gdy musimy zobaczyć różnice, a nie podobieństwa pomiędzy strukturami danych?

`Test::Differences` eksportuje funkcję `eq_or_diff()`, która pokazuje wyniki podobne do uniksowego *diff*, ale dla struktur danych. Plik *differences.t* jest zmodyfikowaną wersją poprzedniego pliku testowego i wykorzystuje tę funkcję:

```
use Test::More tests => 1;
use Test::Differences;

my $list1 =
[
   [
      [ 48, 12 ],
      [ 32, 10 ],
   ],
   [
      [ 03, 28 ],
   ],
];
my $list2 =
[
```

```
    [
        [ 48, 12 ],
        [ 32, 11 ],
    ],
    [
        [ 03, 28 ],
    ],
];
eq_or_diff( $list1, $list2, 'opowieść o dwóch referencjach' );
```

Uruchomienie tego pliku za pomocą *prove* daje wynik prezentowany poniżej. Wiersze diagnostyczne, które zaczynają się i kończą gwiazdką (*), oznaczają różnice między strukturami danych.

```
$ prove -v differences.t
differences....1..1
not ok 1 - opowieść o dwóch referencjach

#     Failed test (differences.t at line 24)
# +----+-----------+-----------+
# | Elt|Got        |Expected   |
# +----+-----------+-----------+
# |   0|[          |[          |
# |   1|  [        |  [        |
# |   2|    [      |    [      |
# |   3|      48,  |      48,  |
# |   4|      12   |      12   |
# |   5|    ],     |    ],     |
# |   6|    [      |    [      |
# |   7|      32,  |      32,  |
# *   8|      10   |      11   *
# |   9|    ]      |    ]      |
# |  10|  ],       |  ],       |
# |  11|  [        |  [        |
# |  12|    [      |    [      |
# |  13|      3,   |      3,   |
# |  14|      28   |      28   |
# |  15|    ]      |    ]      |
# |  16|  ]        |  ]        |
# |  17|]          |]          |
# +----+-----------+-----------+
# Looks like you failed 1 test of 1.
dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test   Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
differences.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

...z porównywaniem dwóch napisów wiersz po wierszu?

Funkcja `eq_or_diff()` z `Test::Differences` pokazuje różnice między napisami wielowierszowymi. Następujący przykład zapisz w pliku *strings.t*. Sprawdza on równoważność dwóch tekstów przy użyciu wspomnianej funkcji:

```
use Test::More tests => 1;
use Test::Differences;

my $string1 = <<"END1";
Lorem ipsum dolor sit
amet, consectetuer,
adipiscing elit.
END1

my $string2 = <<"END2";
Lorem ipsum dolor sit
amet, facilisi
adipiscing elit.
END2

eq_or_diff( $string1, $string2, 'Czy są takie same?' );
```

Uruchomienie go za pomocą *prove* daje poniższy wynik:

```
$ prove -v strings.t
strings....1..1
not ok 1 - Czy są takie same?

#     Failed test (strings.t at line 16)
# +---+----------------------+----------------------+
# | Ln|Got                   |Expected              |
# +---+----------------------+----------------------+
# |  1|Lorem ipsum dolor sit |Lorem ipsum dolor sit |
# *  2|amet, consectetuer,   |amet, facilisi        *
# |  3|adipiscing elit.      |adipiscing elit.      |
# +---+----------------------+----------------------+
# Looks like you failed 1 test of 1.
dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
-----
strings.t      1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

Diagnostyka przypomina uzyskaną z *differences.t*. Różniące się wiersze zostały oznaczone gwiazdkami.

...z porównywaniem danych binarnych?

W przypadku niektórych różnic dobrze jest móc porównać specjalne sekwencje znaków. Taką funkcjonalność w postaci zestawu funkcji do porównywania i testowania napisów, które nie są czystym tekstem lub są szczególnie długie, udostępnia moduł `Test::LongString`.

Plik *longstring.t* jest zmodyfikowanym plikiem *strings.t* i różni się od niego wykorzystaniem `is_string()`:

```
use Test::More tests => 1;
use Test::LongString;

my $string1 = <<"END1";
Lorem ipsum dolor sit
amet, consectetuer,
adipiscing elit.
END1

my $string2 = <<"END2";
Lorem ipsum dolor sit
amet, facilisi
adipiscing elit.
END2

is_string( $string1, $string2, 'Czy są takie same?' );
```

Uruchomienie *longstring.t* za pomocą *prove* daje następujący wynik:

```
$ prove -v longstring.t
longstring....1..1
not ok 1 - Czy są takie same?

#     Failed test (longstring.t at line 16)
#          got: ..."sit\x{0a}amet, consectetuer,\x{0a}adipiscing
elit.\x{0a}"...
#       length: 59
#     expected: ..."sit\x{0a}amet, facilisi\x{0a}adipiscing
elit.\x{0a}"...
#       length: 54
#     strings begin to differ at char 29
# Looks like you failed 1 test of 1.
dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test  Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
longstring.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

Test::LongString eksportuje także kilka innych wygodnych funkcji do testowania napisów. Funkcje te dają podobne informacje diagnostyczne. Więcej informacji na ten temat znajduje się w dokumentacji modułu.

\x{0a} jest jednym ze sposobów reprezentacji znaku nowego wiersza.

Diagnostyczne wyjście z funkcji `is_string()` modułu `Test::LongString` wyświetla w specjalny sposób znaki niedrukowalne (`\x{0a}`), prezentuje długość napisów (59 i 54) oraz pozycję pierwszego znaku różniącego oba napisy.

Złożone struktury danych

Wraz ze wzrostem złożoności struktur danych używanych w kodzie komplikują się też testy. Ważna staje się możliwość weryfikacji tego, co tak naprawdę składa się na strukturę danych, a nie tylko proste porównywanie do istniejącej struktury. **Można** sprawdzać każdy element z osobna, przechodząc przez wszystkie poziomy zagnieżdżonych list czy tablic asocjacyjnych. Na szczęście moduł `Test::Deep` pozwala na poprawienie wyglądu kodu odpowiadającego za testowanie skomplikowanych struktur oraz dostarcza sensownych komunikatów o błędach.

Jak to osiągnąć?

Następujący kod zapisz w pliku *cmp_deeply.t*:

```
use Test::More tests => 1;
use Test::Deep;

my $points =
[
   { x => 50, y => 75 },
   { x => 19, y => -29 },
];

my $is_integer = re('^-?\d+$');

cmp_deeply( $points,
   array_each(
      {
         x => $is_integer,
         y => $is_integer,
      }
   ),
   'obydwa zestawy punktów powinny być liczbami całkowitymi' );
```

Plik *cmp_deeply.t* uruchom za pomocą *prove* z wiersza poleceń. Wynikiem jest jeden udany test:

```
$ prove cmp_deeply.t
cmp_deeply....ok
```

```
All tests successful.
Files=1, Tests=1,  1 wallclock secs ( 0.25 cusr +  0.02 csys =  0.27 CPU)
```

Jak to działa?

Funkcja `cmp_deeply`, podobnie jak większość innych funkcji testujących, przyjmuje dwa lub trzy argumenty: strukturę danych do przetestowania, jej oczekiwaną zawartość oraz opcjonalny opis testu. Drugi argument, oczekiwane dane, to specjalna struktura testowa w formacie zawierającym specjalne funkcje `Test::Deep`.

Plik testowy rozpoczyna się od utworzenia wyrażenia regularnego za pomocą funkcji `re()` z modułu `Test::Deep`. W funkcji `re()` deklaruje się, że dane muszą pasować do podanego wyrażenia regularnego. Jeśli zamiast tego użyjemy bezpośrednio wyrażenia regularnego, to `Test::Deep` uzna, że oczekujemy właśnie takiego wyrażenia w charakterze danych, a nie że dane mają do niego pasować.

Funkcja re() pozwala także wykonywać sprawdzenia na danych, które pasują do wyrażenia. Więcej informacji na ten temat można znaleźć w dokumentacji Test::Deep.

Funkcja `array_each()` z modułu `Test::Deep` tworzy główną strukturę testową. Żeby test zakończył się sukcesem, `$points` musi być tablicą, a każdy jej element musi być zgodny ze strukturą testową przekazaną do funkcji `array_each()`.

Przekazanie odwołania do tablicy asocjacyjnej jako struktury testowej oznacza zadeklarowanie, że każdy element testowanej tablicy musi być tablicą asocjacyjną i że wartości w nim przechowywane muszą być zgodne z wartościami w strukturze testowej. W pliku *cmp_deeply.t* tablica asocjacyjna ma tylko dwa klucze, `x` oraz `y`, więc testowana struktura też musi mieć tylko te dwa klucze. Dodatkowo obydwie wartości muszą pasować do wyrażenia regularnego utworzonego przez `re()`.

Diagnostyka `Test::Deep` jest naprawdę bardzo użyteczna, gdy mamy do czynienia z dużymi strukturami danych. Zmień teraz `$points` tak, aby wartością `y` w pierwszej tablicy asocjacyjnej była litera `Q`, co jest niepoprawne z punktu widzenia struktury testowej. Zmieniony plik zapisz w pliku *cmp_deeply2.t*:

```
use Test::More tests => 1;
use Test::Deep;

my $points =
```

```
[
   { x => 50, y => 75 },
   { x => 19, y => 'Q' },
];

my $is_integer = re('^-?\d+$');

cmp_deeply( $points,
   array_each(
      {
         x => $is_integer,
         y => $is_integer,
      }
   )
);
```

cmp_deeply2.t uruchom za pomocą `prove -v`. Funkcja `cmp_deeply()` zgłosi niepowodzenie, pokazując następujące komunikaty diagnostyczne:

```
$ prove -v cmp_deeply2.t
cmp_deep2....#     Failed test (cmp_deeply2.t at line 12)
# Using Regexp on $data->[1]{"y"}
#    got : 'Q'
# expect : (?-xism:^-?\d+$)
# Looks like you failed 1 test of 1.
dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test   Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
cmp_deeply2.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
```

W komunikatach diagnostycznych określono dokładnie, która część struktury danych nie przeszła testu. Znajduje się tam też wyjaśnienie, że wartość `Q` nie pasuje do wyrażenia regularnego `$is_integer`.

A co...

...zrobić, jeśli niektóre wartości w strukturze mogą się zmieniać?

Żeby nie sprawdzać niektórych wartości, trzeba zamiast wyrażenia regularnego użyć funkcji `ignore()`. Następujący przykład pozwala się upewnić, że każda tablica asocjacyjna w tablicy jako klucze wykorzystuje zarówno `x` jak i `y`, ale nie jest sprawdzana wartość `y`:

```
array_each(
   {
      x => $is_integer,
```

```
        y => ignore(),
    }
);
```

...zrobić, jeśli niektóre klucze w strukturach danych mogą się zmienić?

Przypuśćmy, że chcemy się upewnić, że każda tablica asocjacyjna zawiera **przynajmniej** klucze `x` oraz `y`. Funkcja `superhashof()` zapewnia, że klucze i wartości im przypisane znajdują się w sprawdzanej tablicy asocjacyjnej, ale pozwala też na inne klucze i wartości:

```
array_each(
   superhashof(
      {
         x => $is_integer,
         y => ignore(),
      }
   )
);
```

W podobny sposób funkcja `subhashof()` z modułu `Test::Deep` zapewnia, że dana tablica asocjacyjna może zawierać niektóre albo wszystkie klucze podane w testowej tablicy asocjacyjnej, ale żadnych innych.

Warto o tym pomyśleć jako o nadzbiorach i podzbiorach.

...ze sprawdzaniem zawartości tablicy, gdy nie da się przewidzieć kolejności jej elementów?

Moduł `Test::Deep` dostarcza funkcji `bag()`, która dokładnie to robi. Zapisz następujący kod w pliku *bag.t*:

```
use Test::More tests => 1;
use Test::Deep;

my @a = ( 4, 89, 2, 7, 1 );

cmp_deeply( \@a, bag(1, 2, 4, 7, 89 ) );
```

Po uruchomieniu *bag.t* widać, że test kończy się powodzeniem. Funkcja `bag()` jest tak często wykorzystywana w plikach testowych, że `Test::Deep` zawiera też funkcję `cmp_bag()`. Plik *bag.t* można również napisać następująco:

```
use Test::More tests => 1;
use Test::Deep;

my @a = ( 4, 89, 2, 7, 1 );

cmp_bag( \@a, [ 1, 2, 4, 7, 89 ]);
```

Więcej informacji

Niniejszy podrozdział zawiera jedynie krótki przegląd modułu `Test::Deep`. Moduł ten dostarcza dalszych funkcji porównujących do testowania obiektów, metod, zbiorów (nieuporządkowanych tablic o unikalnych elementach), zmiennych logicznych i odwołań. Informacje na ten temat znajdują się w dokumentacji `Test::Deep`.

Testowanie ostrzeżeń

Testowania nie wymagają jedynie niepotrzebne fragmenty kodu. Jeśli program w niektórych sytuacjach zgłasza ostrzeżenia i są one ważne, to trzeba przetestować, czy występują tylko wtedy, gdy są oczekiwane. Moduł `Test::Warn` zawiera użyteczne funkcje do wyłapywania i sprawdzania ostrzeżeń.

Jak to osiągnąć?

Następujący kod należy zapisać w pliku *warnings.t*:

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

sub add_positives
{
   my ( $l, $r ) = @_;
   warn "pierwszy argument ($l) jest ujemny" if $l < 0;
   warn "drugi argument ($r) jest ujemny" if $r < 0;
   return $l + $r;
}

warning_is { is( add_positives( 8, -3), 5 ) }
   "drugi argument (-3) jest ujemny";

warnings_are { is( add_positives( -8, -3), -11) }
   [
      'pierwszy argument (-8) jest ujemny',
      'drugi argument (-3) jest ujemny'
   ];
```

Między pierwszym a drugim argumentem wszystkich funkcji testujących z modułu Test::Warn nie ma przecinka, bo ich prototypy zamieniają normalnie wyglądające bloki kodu na odwołania do podprogramów.

Uruchom ten plik za pomocą *prove*, a otrzymasz następujący wynik:

```
$ prove -v warnings.t
warnings....1..4
ok 1
ok 2
```

```
ok 3
ok 4
ok
All tests successful.
Files=1, Tests=4,  0 wallclock secs ( 0.19 cusr +  0.04 csys =  0.23 CPU)
```

Jak to działa?

W pliku testowym znajduje się deklaracja i test prostej funkcji `add_positives()`. Funkcja dodaje do siebie dwie liczby i generuje ostrzeżenie, jeśli którakolwiek z tych liczb jest mniejsza od zera.

Argumentem `warning_is()` jest blok kodu, który będzie uruchamiany, oraz tekst oczekiwanego ostrzeżenia. Podobnie jak inne funkcje testujące, ma ona jeszcze trzeci, opcjonalny argument, który jest opisem testu. Wywołanie `add_positives()` z dwoma argumentami mniejszymi od zera powoduje wygenerowanie dwóch ostrzeżeń. Żeby przetestować taką sytuację, trzeba użyć funkcji `warnings_are()` z modułu `Test::Warn`. Zamiast pojedynczego napisu drugim argumentem `warnings_are()` jest odwołanie do tablicy napisów generowanych przez ostrzeżenia.

A co…

…zrobić, gdy ostrzeżenia nie da się wyrazić konkretnym napisem?

Moduł `Test::Warn` eksportuje również funkcję `warning_like()`, której podaje się odwołanie do wyrażenia regularnego zamiast do konkretnego napisu. Argumentem drugiej funkcji, `warnings_like()`, jest nienazwana tablica wyrażeń regularnych, a nie pojedyncze takie wyrażenie. Dzięki tym funkcjom można skrócić plik *warnings.t*:

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

sub add_positives
{
   my ( $l, $r ) = @_;
   warn "pierwszy argument ($l) jest ujemny" if $l < 0;
   warn "drugi argument ($r) jest ujemny" if $r < 0;
   return $l + $r;
}

warning_like { is( add_positives( 8, -3), 5 ) } qr/ujemny/;

warnings_like { is( add_positives( -8, -3 ), -11 ) }
```

```
    [ qr/pierwszy.*ujemny/, qr/drugi.*ujemny/ ];
```

...jeśli chcemy dowieść, że nie ma żadnych ostrzeżeń w pewnym bloku kodu?

To jest dobry test w sytuacji, gdy `add_positives()` zostaje wywołane z dwoma liczbami naturalnymi. Aby się upewnić, że blok kodu nie generuje żadnych ostrzeżeń, trzeba użyć funkcji `warnings_are()` z modułu `Test::Warn` z pustą tablicą nienazwaną jako argumentem:

```
warnings_are { is( add_positives( 4, 3), 7 ) } [];
```

...zrobić, żeby się upewnić, że w kodzie nie jest generowane żadne ostrzeżenie?

Do tego służy moduł `Test::NoWarnings`, który w trakcie działania testu sprawdza, czy pojawiły się jakieś ostrzeżenia. `Test::NoWarnings` na końcu dodaje test, w którym się upewnia, że nie było ostrzeżeń.

Program przedstawiony na poniższym listingu, *nowarn.t*, testuje funkcję `add_positives()` i wykorzystuje `Test::NoWarnings`. Licznik testów zmienił się i uwzględnia dodatkowy test:

```
use Test::More tests => 3;
use Test::NoWarnings;

sub add_positives
{
   my ( $l, $r ) = @_;
   warn "pierwszy argument ($l) jest ujemny" if $l < 0;
   warn "drugi argument ($r) jest ujemny" if $r < 0;
   return $l + $r;
}

is( add_positives( 4, 6 ), 10 );
is( add_positives( 8, -3),  5 );
```

Drugi test generuje ostrzeżenie, które będzie przechwycone i zapamiętane przez `Test::NoWarnings`. Uruchomienie testu wyświetli informacje diagnostyczne na temat wszystkich ostrzeżeń, które wystąpiły, i testów, które je wygenerowały.

```
nowarn....1..3
ok 1
ok 2
not ok 3 - no warnings

#     Failed test (/usr/lib/perl5/5.8.6/Test/NoWarnings.pm at line 45)
```

```
# There were 1 warning(s)
#       Previous test 1 ''
#       drugi argument (-3) jest ujemny at nowarn.t line 8.
#  at nowarn.t line 8
#       main::add_positives(8, -3) called at nowarn.t line 13
#
# Looks like you failed 1 test of 3.
dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 3
        Failed 1/3 tests, 66.67% okay
Failed Test Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
nowarn.t       1   256     3    1  33.33%  3
Failed 1/1 test scripts, 0.00% okay. 1/3 subtests failed, 66.67% okay.
```

Testowanie wyjątków

Czasem coś się nie udaje. Nic nie szkodzi, niejednokrotnie najlepszym pomysłem na obsługę nienaprawialnego błędu jest jego zgłoszenie i pozostawienie wyższym warstwom kodu decyzji, co z nim zrobić. Jeśli postępujemy w taki sposób, to zachowanie to musi być przetestowane. Jak zwykle jest gotowy moduł, który to ułatwia. `Test::Exception` dostarcza funkcji testujących, czy dany blok kodu zgłasza (lub nie zgłasza) wyjątki, których oczekujemy.

Jak to osiągnąć?

Przypuśćmy, że jesteśmy zadowoleni z `add_positives()` omówionego w podrozdziale „Testowanie ostrzeżeń", ale nasi współpracownicy nie używają tej funkcji poprawnie. Wywołują funkcję z ujemnymi argumentami i ignorują ostrzeżenia, a potem mają pretensje, że ich kod nie działa prawidłowo. Lider naszego zespołu zaproponował, żeby zmienić funkcję tak, by nie **tolerowała** ujemnych liczb i zgłaszała wyjątek, jeśli na jakąś trafi. Jak to teraz przetestować?

Następujący kod zapisz w *exception.t*:

```
use Test::More tests => 3;
use Test::Exception;
use Error;

sub add_positives
{
```

Między pierwszym a drugim argumentem wszystkich funkcji testujących z Test::Exception nie ma przecinków.

```
    my ( $l, $r ) = @_;
    throw Error::Simple("pierwszy argument ($l) jest ujemny") if $l < 0;
    throw Error::Simple("drugi argument ($r) jest ujemny") if $r < 0;
    return $l + $r;
}

throws_ok { add_positives( -7,  6 )} 'Error::Simple';
throws_ok { add_positives(  3, -9 )} 'Error::Simple';
throws_ok { add_positives( -5, -1 )} 'Error::Simple';
```

Plik uruchom za pomocą *prove*:

```
$ prove -v exception.t
exception....1..3
ok 1 - threw Error::Simple
ok 2 - threw Error::Simple
ok 3 - threw Error::Simple
ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.13 cusr +  0.02 csys =  0.15 CPU)
```

Jak to działa?

Wywołanie funkcji `throws_ok()` zapewnia nas, że `add_positives()` zgłasza wyjątek typu `Error::Simple`. Funkcja `throws_ok` wykonuje sprawdzenie `isa()` dla przechwyconych przez siebie wyjątków. W związku z tym można więc podać dowolną klasę nadrzędną dla zgłaszanego wyjątku. Ponieważ wyjątki dziedziczą po klasie `Error`, to można, na przykład, zamienić wszystkie wystąpienia `Error::Simple` w *exception.t* na `Error`.

A co...

...gdy trzeba się upewnić, że kod nie zgłasza żadnych wyjątków?

Należy wykorzystać funkcję `lives_ok()` z modułu `Test::Exception`.

Żeby się upewnić, że funkcja `add_positives()` nie zgłasza wyjątków, gdy ma argumenty będące liczbami naturalnymi, należy dodać dodatkowy test sprawdzający:

```
use Test::More tests => 4;
use Test::Exception;
use Error;

sub add_positives
{
   my ( $l, $r ) = @_;
   throw Error::Simple("pierwszy argument ($l) jest ujemny") if $l < 0;
```

```
    throw Error::Simple("drugi argument ($r) jest ujemny") if $r < 0;
    return $l + $r;
}

throws_ok { add_positives( -7,  6 )} 'Error::Simple';
throws_ok { add_positives(  3, -9 )} 'Error::Simple';
throws_ok { add_positives( -5, -1 )} 'Error::Simple';
lives_ok  { add_positives(  4,  6 )} 'tutaj nie ma wyjątku!';
```

Jeśli w bloku kodu zgłaszany jest wyjątek, to funkcja `lives_ok()` wyświetli informację o nieudanym teście. W przeciwnym wypadku test zakończy się sukcesem.