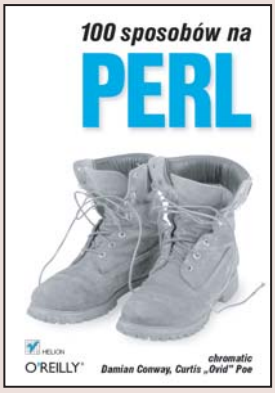
100 sposobów na Perl



Autorzy: Damian Conway, Curtis „Ovid" Poe
Tłumaczenie: Sławomir Dzieniszewski
ISBN: 83-246-0634-3
Tytuł oryginału: Perl Hacks: Tips & Tools for Programming, Debugging, and Surviving
Format: B5, stron: 320

Zbiór skutecznych rozwiązań dla programistów aplikacji internetowych

- Zwiększanie produktywności pracy
- Tworzenie interfejsów użytkownika
- Wyszukiwanie i usuwanie błędów

Perl, od swojego zaistnienia na rynku, wyewoluował od prostego narzędzia do przetwarzania tekstów i budowania raportów do formy zaawansowanego języka programowania pozwalającego na tworzenie praktycznie każdej aplikacji działającej w sieci. Mimo dość zaawansowanego „wieku", nie traci nic na popularności. W sieci pojawiają się coraz nowsze wersje, a grono programistów korzystających z Perla stale rośnie. Społeczność użytkowników tego języka skupiona wokół portalu CPAN udostępnia napisane przez siebie skrypty, wskutek czego z wieloma problemami programistycznymi można sobie poradzić, korzystając z gotowych rozwiązań lub sięgając do innych źródeł.

Dzięki książce „100 sposobów na Perl" odkryjesz mało znane i mniej typowe zastosowania tego języka. Czytając ją, dowiesz się, w jaki sposób wykorzystać Perl do różnych zadań. Nauczysz się zwiększać efektywność swojej pracy, tworzyć elementy interaktywne i przetwarzać pliki tekstowe w nietypowy sposób. Zapoznasz się z odczytywaniem danych z baz i arkuszy kalkulacyjnych, pracą z modułami oraz programowaniem obiektowym. Znajdziesz tu także informacje o testowaniu kodu, usuwaniu błędów i optymalizowaniu wydajności programów napisanych w Perlu.

- Korzystanie z biblioteki CPAN
- Automatyczne formatowanie kodu w edytorze Emacs
- Generowanie elementów graficznych
- Przetwarzanie arkuszy kalkulacyjnych
- Praca z bazami danych
- Tworzenie zestawu narzędziowego modułów
- Korzystanie z obiektów
- Testowanie kodu
- śledzenie wykonywania programu

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ PIĄTY

Sposoby na obiekty

Sposoby 43. – 50.

Jak łatwo zgadnąć, Perl też posiada obiekty. Oprócz dziwacznej na pierwszy rzut oka funkcji `bless` oraz odpowiedniego przekwalifikowania procedur, pakietów i odwołań obiektowy język Perl oferuje programiście wiele nowych opcji i rozszerza znacznie możliwości języka. Część z Czytelników zapewne korzysta z funkcji `bless`, by tworzyć (potocznie „błogosławić") obiekty tylko dlatego, że potrzebują rekordów obiektów. Warto jednak zastanowić się nad korzyściami, które może przynieść lepsza enkapsulacja (obudowywanie) danych, automatyczna serializacja danych czy wymuszanie kontroli dostępu do danych.

Im więcej programista wie na temat Perla, tym więcej będzie miał dostępnych opcji, umożliwiających tworzenie i korzystanie z wyższych poziomów abstrakcji. Następnym razem, gdy jego współpracownicy natrafią znowu na złożony problem, którego nie potrafią rozwiązać, będzie mógł zajrzeć do swej magicznej sakwy z trikami obiektowymi i uśmiechając się, uspokoić ich: „Bez obaw, za pomocą Perla można rozwiązać każdy problem".

SPOSÓB 43.

Tworzenie zamkniętych obiektów

Zadbaj o dobre obudowanie atrybutów obiektu

W Perlu 5 obsługa obiektów sprowadzona jest do minimum. Perl daje programiście wystarczające narzędzia, by umożliwić programowanie obiektowe, nie chroni go jednak przed robieniem z obiektami rzeczy nieodpowiedzialnych. Oczywiście, domyślne podejście do programowania obiektowego jest z reguły najprostsze (oraz najmniej odpowiedzialne), choć niestety nie najporządniejsze ani też nieułatwiające późniejszej obsługi kodu.

W większości przypadków obiektami będą po prostu błogosławione (tj. przemienione w obiekty za pomocą funkcji `bless`) tablice asocjacyjne, ponieważ początkującym programistom najłatwiej zrozumieć ich działanie i najłatwiej też z nich skorzystać. Niestety, czasem trudno znaleźć w nich ewentualne błędy i tak naprawdę nie oferują żadnego obudowania (enkapsulacji) dla danych. Dlatego też warto sięgnąć po specjalne, czysto obiektowe rozwiązania[1].

[1] Więcej na ten temat można znaleźć w artykule „Seven Sins of Perl OO Programing" („Siedem grzechów głównych programowania obiektowego w Perlu"), w przeglądzie *The Perl Review* 2.1, zima 2005.

Na szczęście można to łatwo naprawić.

Sposób

Obiekt przygotowany w Perlu potrzebuje dwóch rzeczy: miejsca, w którym będzie przechowywać dane instancji, oraz klasy, w której można będzie znaleźć jego metody. Błogosławiona tablica asocjacyjna (lub tablica, skalar, procedura, typeglob, etc.) przechowuje swoje dane wewnątrz obiektu, który będziemy przesyłać. Jeśli dokonamy dereferencji odwołania, to niestety będzie można odczytywać dane obiektu z dowolnego miejsca, nawet spoza klasy.

Prawidłowo przygotowany, zamknięty (ang. *inside out*) obiekt powinien przechowywać dane w innym miejscu, najczęściej w zmiennej leksykalnej, której zakres odpowiada zakresowi klasy. W przypadku użycia zmiennych leksykalnych nie będzie można (tzn. *zazwyczaj* nie będzie można — patrz „Zaglądanie za zamknięte drzwi" **[Sposób 76.]**) sięgać do danych bez użycia specjalnych metod dostępu obiektu.

Pierwsza książka Damiana Conwaya, *Object Oriented Perl* (wydawnictwo Manning, 2000), pokazywała różne sposoby przygotowywania obudowywania danych. Jego ostatnia książka, *Perl. Najlepsze rozwiązania* (wydawnictwo Helion 2006), zaleca stosowanie się do nich wszystkim programistom, którym zależy na jakości kodu. Jedna z doświadczonych weteranek Perla, Abigail, badała możliwości właściwie zbudowanych obiektów przez kilka lat. Wiele z jej spostrzeżeń i porad można znaleźć w dokumentacji modułu `Class::Std`.

Uruchamianie sposobu

Prosta i niezbyt wyszukana implementacja obiektu, przeznaczona dla klasy rekordu przechowującego dane, może wyglądać tak:

```
# tworzymy nowy zakres dla zmiennych leksykalnych
 {
     package InsideOut::User;

     use Scalar::Util 'refaddr';

     # zmienne leksykalne służące do przechowywania danych instancji
     my %names;
     my %addresses;

     sub new
     {
         my ($class, $data) = @_;

         # błogosławimy nowy skalar, by zdobyć jego identyfikator obiektu
         bless \(my $self), $class;

         # zachowujemy dane instancji
         my $id             = refaddr( $self );
         $names{     $id }  = $data->{name};
         $addresses{ $id }  = $data->{address};

         return $self;
```

```
    }

    # metody dostępu takie jak $self->{name}, czy $self->{address} nie działają
    sub get_name
    {
        my $self = shift;
        return $names{ refaddr( $self ) };
    }

    sub get_address
    {
        my $self = shift;
        return $addresses{ refaddr( $self ) };
    }

    # wielu ludzi zapomina o tej części
    sub DESTROY
    {
        my $self = shift;
        my $id   = refaddr( $self );
        delete $names{     $id };
        delete $addresses{ $id };
    }
}

1;
```

Jak widać, zdefiniowanie zamkniętego obiektu wymaga trochę więcej pisania, niemniej kod jest teraz znacznie czystszy. Teraz możemy podklasować lub zaimplementować na nowo klasę `InsideOut::User` bez konieczności korzystania z błogosławionej tablicy asocjacyjnej — wystarczy dostosować się do interfejsu definiowanego przez tę klasę i kod powinien zadziałać.

Eksplorowanie sposobu

W sieci CPAN dostępne są trzy moduły `Class:Std`, `Class::InsideOut` i `Object::InsideOut`, które ułatwiają programistom pisanie prawidłowych, zamkniętych obiektów. Każdy z nich oferuje różne triki i użyteczne możliwości. W module `Class::Std` miłe jest to, że automatycznie tworzy metody umożliwiające dostęp do danych obiektu (ang. *accessors* — metody dostępu) i modyfikowanie (ang. *mutators* — metody modyfikujące) tych danych. Ponadto przywołuje lepsze konstruktory i destruktory obiektu oraz „umożliwia definiowanie dodatkowych informacji w atrybutach zmiennych i procedur" **[Sposób 45]**.

Korzystając z modułu `Class::Std`, przedstawioną wcześniej klasę można by napisać w następujący sposób:

```
{
    package InsideOut::User;

    use Class::Std;

    my %names     :ATTR( :get<name>    :init_arg<name>    );
    my %addresses :ATTR( :get<address> :init_arg<address> );
}
```

Kod ten automatycznie wygeneruje metody dostępu `get_name()` i `get_address()` oraz konstruktor, który pobierze początkowe wartości obiektów z odwołania do tablicy asocjacyjnej według odpowiednich kluczy tejże tablicy. Składnia tu zaprezentowana nie jest *tak elegancka*, jak składnia Perla 6, niemniej jest znacznie krótsza niż zaprezentowana wcześniej wersja napisana od podstaw w Perlu 5 — i co ważniejsze, oferuje dokładnie te same funkcje.

SPOSÓB 44.

Darmowe (prawie) serializowanie obiektów

Przechowuj dane, unikając bałaganu, nieporozumień oraz wielkich obiektów danych binarnych

Niektóre programy bezwzględnie potrzebują trwałego zapisywania danych i czasami okazuje się, że wykonywanie mapowania między obiektami a różnymi tabelami w pełni relacyjnej bazy danych jest zbyt pracochłonne. Szczególnie w tych przypadkach, gdy liczy się szybkość i łatwość edytowania danych — w takich sytuacjach trudno o lepszy interfejs do ich edytowania niż „nasz ulubiony edytor" **[Sposób 12.]**.

Zamiast ręcznego konfigurowania wszystkiego w programie i tracenia naszej cennej młodości na tworzenie idealnego schematu bazy danych lub ćwiczenia się w korzystaniu z języka XML czemu po prostu nie serializować (zapisywać na trwałe) danych z obiektów w plikach YAML?

Sposób

Jeśli korzystamy z obiektów zbudowanych na tablicach asocjacyjnych, to serializowanie danych jest bardzo proste — wystarczy utworzyć kopię tablicy asocjacyjnej i serializować ją np. w pliku:

```
use YAML 'DumpFile';

 sub serialize
 {
     my ($object, $file) = @_;
     my %data            = %$object;
     DumpFile( $file, \%data );
 }
```

Zakładamy tutaj oczywiście, że wartość przechowywana w zmiennej `$object` jest obiektem, który chcemy serializować, a zmienna `$file` określa ścieżkę do pliku, w którym obiekt ma zostać zachowany.

Jeśli natomiast korzystamy „z prawidłowo zamkniętych obiektów" **[Sposób 43.]**, to czeka nas trochę więcej pracy:

```
package Graphics::Drawable;
 {
     use Class::Std;

     my %coords_of     :ATTR( :get<coords>   :init_arg<coords>   );
     my %velocities_of :ATTR( :get<velocity> :init_arg<velocity> );
     my %shapes_of     :ATTR( :get<shape>    :init_arg<shape>    );
```

```
    sub get_serializable_data
    {
        my $self  = shift;

        my %data;

        for my $attribute (qw( coords velocity shape ))
        {
            my $method = 'get_' . $attribute;
            $data{ $attribute } = $self->$method(  );
        }

        return \%data;
    }
}
```

Teraz nasza funkcja `serialize()` będzie mogła uniknąć naruszenia obudowania danych i przywołać procedurę `get_serializable_data()`. Obiekt znajdujący się w początku układu współrzędnych — współrzędne (`0`, `0`, `0`) — poruszający się wzdłuż osi X z prędkością (`velocity`) jednej jednostki na jednostkę odległości — ruch (`1`, `0`, `0`) — oraz kształt okręgu `Circle` zostaną zserializowane w następujący sposób:

```
---
coords:
  - 0
  - 0
  - 0
shape: Circle
velocity:
  - 1
  - 0
  - 0
```

Jeśli potrzebna będzie większa liczba obiektów, wystarczy skopiować plik do nowej lokalizacji i odpowiednio go zmodyfikować. Należy tylko pamiętać, aby zachować prawidłową składnię języka YAML[2].

Przywracanie takich obiektów jest proste. Wystarczy użyć metody `LoadFile()` modułu `YAML()`:

```
use YAML 'LoadFile';

sub deserialize
{
    my ($class, $file) = @_;
    my $data           = LoadFile( $file );
    return $class->new( $data );
}
```

Jeśli nasz konstruktor klasy pobiera jako atrybut odwołanie do tablicy asocjacyjnej, której klucze odpowiadają nazwom atrybutów (tak jak to jest w klasie `Class::Std`), to w zasadzie mamy już wszystkie potrzebne elementy serializacji. Oczywiście wszystko to wymaga przygotowania pewnej fabryki obiektów, która będzie zarządzała instancjami,

[2] Co jednak jest prostsze niż ręczne pisanie prawidłowego kodu XML…

mapowała pliki i ścieżki na klasy oraz zachowywała i pobierała obiekty, że nie wspomnę o zarządzaniu błędami. Wszystkim tym może zająć się moduł `Class::Storage-Factory`, dostępny w sieci CPAN.

Jeśli już mamy te wszystkie narzędzia — i aby móc odtworzyć obiekt, potrzebujemy tylko danych z publicznego interfejsu obiektu (atrybutów konstruktora i danych dostępnych za pomocą metod dostępu) — to serializowanie do pliku YAML lub innego czysto tekstowego formatu (może np. JSON?) jest szybkie, wygodne i prawie nic nie kosztuje.

SPOSÓB 45.

Umieszczanie dodatkowych informacji w atrybutach

Opatrz swoje zmienne i procedury paroma dodatkowymi informacjami

Procedury i zmienne są dość oczywiste. Owszem, można przesyłać odwołania do nich lub zmienić je w procedury i zmienne anonimowe, a następnie robić z nimi różne dziwne rzeczy, niemniej tak czy siak mamy niewielki wpływ na to, co Perl będzie z nimi robić.

Najlepszym rozwiązaniem jest przydanie im atrybutów. Wspomniane atrybuty są to małe fragmenty danych podczepiane do zmiennych lub procedur. Z ich pomocą można skłonić Perla, by uruchomił dowolny kod, jaki będzie nam potrzebny. Daje to naprawdę nieograniczone możliwości.

Sposób

Załóżmy, że przygotowaliśmy klasę i chcielibyśmy udokumentować przeznaczenie każdej z jej metod. Niektóre języki dostarczają w tym celu krótkich *łańcuchów dokumentujących* (ang. *docstrings*) — komentarzy, które można oglądać, przywołując metody klasy. Komentarze Perla są raczej nudne, niemniej można osiągnąć prawie taki sam efekt, opatrując metody odpowiednimi atrybutami procedur.

Rozważmy klasę `Counter`, której celem jest dostarczenie domyślnego konstruktora zliczającego liczbę utworzonych obiektów. Korzystając z atrybutu `Doc` oferowanego przez moduł `Attribute::Docstring`, można przygotować następującą klasę:

```
package Counter;

 use strict;
 use warnings;

 use Attribute::Docstring;

 our $counter :Doc( 'licznik wszystkich nowych obiektów Foo' );

 sub new :Doc( 'konstruktor obiektu Foo' )
 {
     $counter++;
     bless {  }, shift;
 }
```

```
sub get_count :Doc( 'zwraca licznik dla wszystkich obiektów foo' )
{
    return $counter;
}

1;
```

Prototyp pojawia się zaraz po nazwie procedury i jest poprzedzony dwukropkiem. W przeciwnym razie wyglądałby zupełnie jak wywołanie funkcji. Argumentem (jedynym) tak definiowanego atrybutu jest łańcuch dokumentujący.

Uruchamianie sposobu

Najprostszym sposobem tworzenia atrybutów i korzystania z nich jest użycie pomocy modułu `Attribute::Handlers`. Umożliwia on pisanie procedur nazywanych od atrybutów, które chcemy zadeklarować. Implementacja pakietu `Attribute::Docstring` jest następująca:

```
package Attribute::Docstring;

use strict;
use warnings;

use Scalar::Util 'blessed';
use Attribute::Handlers;

my %doc;

sub UNIVERSAL::Doc :ATTR
{
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;
    return if $symbol eq 'LEXICAL';

    my $name                  = *{$symbol}{NAME};
    $doc{ $package }{ $name } = $data;
}

sub UNIVERSAL::doc
{
    my ($self, $name) = @_;
    my $package       = blessed( $self ) || $self;

    return unless exists $doc{ $package }{ $name };
    return               $doc{ $package }{ $name };
}

1;
```

Aby atrybut `Doc` dostępny był wszędzie, moduł definiuje procedurę `UNIVERSAL::Doc`. Procedura ta sama w sobie również posiada atrybut `:ATTR`, który opisuje ją jako procedurę obsługującą atrybuty.

W przypadku każdej procedury lub zmiennej deklarującej atrybut `Doc`, procedura `UNIVERSAL::Doc` otrzymywać będzie kilka informacji. Tutaj najważniejsze są pakiet zawierający procedurę, symbol — za pomocą którego, wykorzystując dostęp typeglob,

będzie można pobrać nazwę — i dane przypisane atrybutowi. W klasie `Counter` procedura obsługująca atrybuty otrzyma nazwę pakietu, czyli `Counter`, oraz typeglob z nazwą symbolu `new`, gdy tylko Perl skończy kompilowanie metody `new()`. Następnie zachowa dane atrybutu (sam łańcuch dokumentujący) w tablicy asocjacyjnej, której kluczami są najpierw nazwa pakietu, a następnie nazwa symbolu.

Z uwagi na różnice w sposobie, w jaki Perl traktuje zmienne leksykalne i zmienne globalne, procedura nie będzie w stanie wiele zrobić, jeśli otrzyma symbol leksykalny (tzn. gdy zmienna `$symbol` będzie `LEXICAL`). Tego rodzaju zmienne i procedury są prywatnymi zmiennymi i procedurami pakietu, więc i tak nie warto ich w ten sposób dokumentować.

Podobna metoda `doc()` działa na każdej klasie i obiekcie, tak więc wywołanie `Counter->doc( 'new' )`, jak również `$counter->doc( 'get_count' )`, zwróci łańcuch dokumentujący dla odpowiedniej metody podanej jako argument. Po prostu odszuka łańcuch dokumentujący metodę w odpowiednim pakiecie i zwróci go.

Eksplorowanie sposobu

Jednym z potencjalnych usprawnień opisanego sposobu jest dodanie do nazwy odpowiedniej „pieczęci" (ang. *sigil*), aby zapobiec wzajemnemu zapisywaniu łańcuchów dokumentujących zmienną o nazwie `$count` i metodę `count()`. Wymagać to będzie wprowadzenia zmian w procedurze `UNIVERSAL::doc()`, by zmienna `$name` zawierała odpowiednią pieczęć (lub nie, jeśli pieczęcią identyfikującą opatrzona ma być metoda).

Kolejna możliwość polega na pobraniu procedury `UNIVERSAL::Doc()` z modułu `UNIVERSAL` i dołączeniu jej do kodu klasy — zamiast importowania jej do pakietu (klasy), który korzysta z tego modułu. W ten sposób zdejmujemy obciążenia z modułu `UNIVERSAL` kosztem jednak zaśmiecania kodu przywołujących go klas. Czasem taka wymiana jest opłacalna, czasem nie.

Atrybuty mogą mieć długość kilku wierszy, niemniej trzeba niestety korzystać w tym celu ze składni heredocs.

SPOSÓB 46.

Upewnianie się, że metody są prywatne dla obiektów

Dowiedz się, jak niewielkim wysiłkiem wymuszać enkapsulację metod

Perl oferuje bardzo wszechstronne narzędzia programowania obiektowego, umożliwiające tworzenie i emulację praktycznie dowolnych rodzajów obiektów lub systemów klas. Mechanizm programowania obiektowego w Perlu jest również bardzo permisywny i nie oferuje bazowo żadnej kontroli dostępu do danych obiektu. Dowolny zewnętrzny kod może w każdej chwili wykorzystać lub podprowadzić metody obiektu i ich metody nadrzędne, używając ich w innej klasie. Może również przywoływać rzekomo prywatne metody obiektu wbrew intencjom programisty, który pisał jego kod.

Zgodnie z powszechnie przyjętą w społeczności programistów Perla konwencją metody, których nazwy zaczynają się od znaku podkreślenia, należy traktować jako metody prywatne i nie próbować ich pokrywać, przywoływać spoza klasy ani też używać innych trików i sztuczek programistycznych, które umożliwiłyby obejście ich prywatności. Jest to rozsądna zasada, niemniej należy pamiętać, że jest to tylko konwencja, niewymuszana w żaden sposób przez konstrukcję języka. Nadal wiec można przywoływać prywatne obiekty w niewłaściwy sposób, obojętnie, czy przez przypadek, czy też celowo.

Na szczęście istnieją lepsze (lub przynajmniej niefrasobliwe) sposoby ukrywania metod.

Sposób

Najprostszy sposób, gwarantujący, że procedury będą w momencie kompilowania traktowane jak metody, to skorzystanie z „atrybutów procedur" **[Sposób 45.]**. Moduł `Class::HideMethods` dodaje do metod atrybut Hide, który czyni je niedostępnymi i prawie niemożliwymi do wywołania spoza programu:

```
package Class::HideMethods;

use strict;
use warnings;
use Attribute::Handlers;

my %prefixes;

sub import
{
    my ($self, $ref)      = @_;
    my $package           = caller(  );
    $prefixes{ $package } = $ref;
}

sub gen_prefix
{
    my $invalid_chars = "\0\r\n\f\b";

    my $prefix;

    for ( 1 .. 5 )
    {
        my $char_pos = int( rand( length( $invalid_chars ) ) );
        $prefix     .= substr( $invalid_chars, $char_pos, 1 );
    }

    return $prefix;
}

package UNIVERSAL;

sub Private :ATTR
{
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;

    my $name    = *{ $symbol }{NAME};
    my $newname = Class::HideMethods::gen_prefix( $package ) . $name;
    my @refs    = map { *$symbol{ $_ } } qw( HASH SCALAR ARRAY GLOB );
    *$symbol    = do { local *symbol };
```

```
    no strict 'refs';
    *{ $package . '::' . $newname } = $referent;
    *{ $package . '::' . $name    } = $_ for @refs;
    $prefixes{ $package }{ $name }  = $newname;
}

1;
```

Aby ukryć metodę, kod ten zastępuje symbol metody nowym pustym typem globalnym typeglob. Zabieg ten usuwa jednak wszystkie zmienne o tej samej nazwie, więc odpowiedni kod kopiuje je najpierw z symbolu i zapisuje w nowym, pustym symbolu. Widać tutaj, jak można „usuwać" dane z typu globalnego typeglob.

Uruchamianie sposobu

Korzystanie z tego modułu nie jest trudne. Wewnątrz naszej klasy wystarczy zadeklarować leksykalną tablicę asocjacyjną, która przechowywać będzie sekretne nowe nazwy metod. Należy przesłać ją do wiersza instrukcji `use Class::HideMethods` używającej pomocniczego modułu:

```
package SecretClass;

 my %methods;
 use Class::HideMethods \%methods;

 sub new            { bless {  }, shift }
 sub hello :Private { return 'hello'   }
 sub goodbye        { return 'goodbye' }

 sub public_hello
 {
     my $self  = shift;
     my $hello = $methods{hello};
     $self->$hello(  );
 }

 1;
```

Należy pamiętać, aby przywoływać wszystkie prywatne metody, używając składni `$przywołujący->$nazwa_metody` i odpowiedniej ukrytej nazwy metody.

Aby upewnić się, czy to zabezpieczenie działa, wykonajmy kilka testów próbujących przywołać metody z zewnętrznego kodu:

```
use Test::More tests => 6;

my $sc = SecretClass->new(  );
isa_ok( $sc, 'SecretClass' );

ok( ! $sc->can( 'hello' ),        'hello(  ) powinna być ukryta'              );
ok( $sc->can( 'public_hello' ),   'public_hello(  ) powinna być dostępna'     );
is($sc->public_hello(  ),
    'hello', '... i powinna móc przywoływać hello(  )' );
ok( $sc->can( 'goodbye' ),        'goodbye(  ) powinna być dostępna '         );
is($sc->goodbye(  ), 'goodbye',    '... i powinna dać się przywoływać'        );
```

Nawet podklasy zdefiniowanej klasy nie są w stanie przywołać jej metod bezpośrednio. Jak widać, udało się uzyskać całkiem dobry poziom prywatności!

Jak ten sposób działa

Wewnętrznie Perl wykorzystuje tzw. tablice symboli (ang. *symbol tables*) do przechowywania wszystkiego, co opatrzone jest nazwą — zmiennych, procedur, metod, klas i pakietów. Robi tak dla wygody ludzi programistów. Teoretycznie nie powinno Perla interesować, jaką metoda ma nazwę, może przywoływać ją zarówno za pomocą nazwy, jak i poprzez odwołanie czy też luźny opis.

Po części jest to prawda, a po części nie. Tylko dla *parsera* Perla ważne są nazwy. Akceptowalne identyfikatory powinny zaczynać się od litery alfabetu albo znaku podkreślenia i zawierać jeden lub więcej znaków alfanumerycznych lub znaków podkreślenia. Gdy już parser Perla dokona analizy programu, sprawdzi, jakimi symbolami dysponuje, w sposób podobny do tego, w jaki przegląda wartości przechowywane w tablicy asocjacyjnej. Jeśli uda nam się przekonać Perla, aby odszukał symbol zawierający teoretycznie nieprawidłowe znaki, z ochotą to zrobi.

Na szczęście istnieje więcej niż jeden sposób przywoływania metod. Jeśli mamy skalar zawierający nazwę metody (który można zdefiniować jako łańcuch zawierający praktycznie dowolne znaki, niekoniecznie poprawny identyfikator) lub odwołanie do samej metody, to Perl przywoła metodę dla elementu wywołującego. To połowa całego triku.

Druga z wykorzystywanych magicznych sztuczek polega na usunięciu symbolu z tablicy symboli, obecnego tam pod swoją niesekretną nazwą. Bez tego użytkownicy mogliby ominąć sekretną nazwę i przywoływać teoretycznie ukrytą metodę wprost.

Jednak skoro prawdziwa nazwa jest niewidoczna, to sama klasa musi w jakiś sposób odnajdywać swoje prywatne metody, by móc je przywoływać. Właśnie temu służy leksykalna tablica asocjacyjna `%methods`, która nie jest normalnie widoczna spoza samej klasy (lub przynajmniej spoza zawierającego ją pliku).

Eksplorowanie sposobu

Sprytniejsza wersja tego kodu mogłaby nawet obyć się bez wykorzystywania tablicy asocjacyjnej `%methods` w klasie, której metody są ukrywane. Być może wykorzystując klauzulę `constant`, by przechowywać nazwy metod w sposób bardziej odpowiedni.

Podejście to nie jest *kompletnym* rozwiązaniem problemu kontroli dostępu do wewnętrznych metod i zmiennych obiektu, przynajmniej w tym sensie, że dostęp do nich mógłby być blokowany przez sam język. Nadal bowiem można znaleźć obejście przedstawionego tu zabezpieczenia. Na przykład można by było przejrzeć tablicę symboli pakietu, szukając zdefiniowanego kodu. Jednym ze sposobów zapobieżenia takim sprytnym sztuczkom jest zrezygnowanie z zapisywania metod z powrotem w tablicy symboli pod zmienionymi nazwami. Zamiast tego należy zupełnie usunąć metody z tabeli symboli i przechowywać odpowiednie odwołania w leksykalnej pamięci podręcznej (ang. *cache*) dla metod.

W ten sposób uda się nam powstrzymać większość zdeterminowanych ludzi. Jednak ludzie *naprawdę* zdeterminowani wiedzą, że moduł `PadWalker` z sieci CPAN umożliwia im „wykorzystywanie zmiennych leksykalnych poza ich normalnym zakresem" **[Sposób 76.]**… Niemniej każdy, kto zada sobie aż tyle wysiłku, mógłby również bez większego wysiłku sprawić, by zamiast ładowania modułu `Class::HideMethods` nasz pakiet ładował coś innego, co nie usunie symboli ukrytych metod. Mimo to nadal jednak trudno będzie przywołać metody obiektu przez przypadek lub nawet celowo bez odrobiny porządnego główkowania. Prawdopodobnie jest to również najlepsze zabezpieczenie, jakie można przygotować w Perlu 5.

SPOSÓB 47.

Autodeklarowanie argumentów metod

Wiesz kim jesteś, więc nie ma powodu, by się powtarzać

Oferowane przez Perl narzędzia programowania obiektowego są bardzo wszechstronne, głównie z uwagi na swoją prostotę i minimalizm. Czasami jest to korzystne: umożliwia programistom tworzenie skomplikowanych systemów obiektowych nawet na bazie bardzo skromnego zestawu narzędzi. Kiedy indziej jednak nawet najprostsze rzeczy programuje się w bólach.

Mimo iż programista nie musi zawsze przywoływać w metodach kod je wywołujący za pomocą zmiennej `$self`, to jednak bardzo często stawać będzie przed koniecznością deklarowania argumentu określającego kod wywołujący i zarządzania nim oraz innymi argumentami. Jest to dość kłopotliwe — niemniej można temu zaradzić. Oczywiście, można by skorzystać z „profesjonalnego filtru kodu źródłowego" **[Sposób 94.]**, aby z jego pomocą odłożyć na bok argument `$self` i przetwarzać tylko pozostałe argumenty z listy. To jednak raczej zbyt rozbudowane narzędzie, by używać go do usuwania takiej drobnej niewygody. Istnieje inny, lepszy sposób.

Sposób

Rozwiązanie tego problemu bez wykorzystywania filtrów kodu źródłowego wymaga rozwiązania trzech problemów. Po pierwsze potrzebny nam będzie jakiś sposób oznaczania procedury jako metody, ponieważ nie wszystkie procedury *są* metodami. Po drugie, aby zachowane zostały reguły dobrego programowania, rozwiązanie to powinno być kompatybilne z deklaracją `strict`. Po trzecie wreszcie, powinien istnieć jakiś sposób umożliwiający dodawanie właściwych operacji, by można było definiować wartości zmiennej `$self` i innych argumentów.

Rozwiązanie pierwszego problemu jest proste: wystarczy skorzystać z „atrybutu procedury" **[Sposób 45.]** o nazwie `Method`. Rozwiązanie trzeciego również nie jest trudne z pomocą modułu „`B::Deparse`" **[Sposób 56.]** oraz instrukcji `eval`. Natomiast drugi wymaga pewnego zachodu…

Na szczęście jednak wszystkie problemy można rozwiązać w jednym, dość krótkim module:

```
package Attribute::Method;

 use strict;
 use warnings;

 use B::Deparse;
 use Attribute::Handlers;

 my $deparse = B::Deparse->new(  );

 sub import
 {
     my ( $class, @vars ) = @_;
     my $package          = caller(  );

     my %references       =
     (
         '$' => \undef,
         '@' => [  ],
         '%' => {  },
     );

     push @vars, '$self';

     for my $var (@vars)
     {
         my $reftype                 = substr( $var, 0, 1, '' );

         no strict 'refs';
         *{ $package . '::' . $var } = $references{$reftype};
     }
 }

 sub UNIVERSAL::Method :ATTR(RAWDATA)
 {
     my ($package, $symbol, $referent, undef, $arglist) = @_;

     my $code                 = $deparse->coderef2text( $referent );
     $code                    =~ s/{/sub {\nmy (\$self, $arglist) = \@_;\n/;

     no warnings 'redefine';
     *$symbol                 = eval "package $package; $code";
 }

 1;
```

Wszystkie zmienne, włączając w to zmienną `$self`, powinny być zmiennymi leksykalnymi w obrębie swoich metod, bowiem w przeciwnym razie mogą się zdarzyć nieprzyjemne rzeczy, gdy spróbujemy przywoływać jedną metodę spod drugiej. Na przykład możemy niechcący zapisać nowymi wartościami jakąś zmienną globalną. Procedura obsługująca atrybut `Method` pobiera skompilowany kod, dokonuje jego analizy i wstawia słowo kluczowe `sub` oraz wiersz zajmujący się obsługą argumentów przed resztą kodu. Wszystkie argumenty dla atrybutu muszą być nazwami zmiennych leksykalnych o zakresie ograniczonym do danej metody.

Skompilowanie tego kodu z instrukcją `eval` przygotuje nową, anonimową procedurę, którą kod wstawia następnie do tablicy symboli zaraz po wyłączeniu ostrzeżeń `Subroutine %s redefined` (procedura `%s` została przedefiniowana).

Uruchamianie sposobu

W kodzie każdej klasy, dla której nie chce nam się na okrągło deklarować i pobierać wciąż tych samych argumentów, można napisać:

```
package Easy::Class;

 use strict;
 use warnings;

 use Attribute::Method qw( $status );

 sub new :Method
 {
     bless { @_ }, $self;
 }

 sub set_status :Method( $status )
 {
     $self->{status} = $status;
 }

 sub get_status :Method
 {
     return $self->{status};
 }

 1;
```

Dla każdej metody oznaczonej atrybutem `:Method` argument określający kod przywołujący `$self` mamy teraz zadeklarowany niejako za darmo. Ponadto dla każdej metody opatrzonej tym atrybutem, parametryzowanej przez listę nazw zmiennych, otrzymamy również te zmienne.

Warto również zwrócić uwagę na magiczne sztuczki zastosowane w procedurze `import()`, jak również na listę przesyłanych jej argumentów. W ten właśnie sposób omijany jest test badający poprawność kodu włączany przez deklarację `strict`. Gdybyśmy zamiast tego użyli tylko struktur `refs` i `subs`, to nie musielibyśmy nawet przesyłać modułowi `Attribute::Method` listy interesujących nas zmiennych.

Eksplorowanie sposobu

Czy to rozwiązanie jest lepsze niż zastosowanie filtrów kodu źródłowego? Oczywiście, jego składnia nie jest tak czysta. Z drugiej strony, rozwiązania oparte na atrybutach przeważnie są mniej wrażliwe niż filtrowanie kodu źródłowego. Przede wszystkim nie uniemożliwiają korzystania z innych filtrów kodu źródłowego lub innych atrybutów. Ponadto praktycznie nigdy nie zawodzą — jeśli nawet nasze procedury będą zawierać błędy, to Perl zaraportuje je w czasie kompilacji, z punktu widzenia oryginalnego kodu, zanim jeszcze przywoła procedurę obsługującą atrybuty. Technika ta sprawdza się najlepiej w klasach posiadających kilka lub więcej metod, z których każda wymaga przesłania jej takich samych argumentów.

Kolejnym *możliwym* rozwiązaniem tego problemu jest przepisanie na nowo drzewa operacji (ang. *optree*) dla odwołań do kodu (co wymaga modułu `B::Generate` i *wiele* cierpliwości), by dodać operacje przypisujące argumenty do odpowiednich zmiennych. Oczywiście, trzeba będzie wstawić zmienne leksykalne do PAD-a powiązanego z CV, jednak Czytelnicy, którzy wiedzą, co to znaczy, będą również zapewne wiedzieli, jak to zrobić.

Wyszukiwanie i rezerwowanie wszystkich zmiennych leksykalnych, które nasze metody zamykają, nie wypada tak źle w porównaniu z innymi metodami. Patrz „Zaglądanie za zamknięte drzwi" **[Sposób 76.]**.

Alternatywny sposób rozwiązania tego problemu można znaleźć w module `Sub::MicroSig`, napisanym przez Richarda Signesa.

SPOSÓB 48.

Kontrola dostępu do zdalnych obiektów

Wymuszaj kontrolę dostępu w swoich obiektach

Jeśli chodzi o podejście do kontroli dostępu i prywatności, język Perl jest bardzo uprzejmy i mało restrykcyjny. Czasami jest to zaletą — nie musimy na przykład długo zastanawiać się, co i jak należy ukryć. Jest to również korzystne, gdy *musimy* szybko przejrzeć kod przygotowany przez kogoś innego.

Innym razem z kolei ważniejsze podczas programowania mogą się okazać względy bezpieczeństwa — szczególnie wtedy, kiedy nasz program będzie musiał stawić czoła dzikiemu, groźnemu światu zewnętrznemu. Nawet wtedy, kiedy musimy wystawić nasz kod na niebezpieczeństwa internetu, nie chcielibyśmy przecież, by każdy mógł z nim robić, co mu się żywnie podoba.

Liczne moduły i narzędzia programistyczne, takie jak `SOAP::Lite`, ułatwiają usługom WWW sięganie do prostych, starych obiektów Perla. Tutaj pokażę, jak zabezpieczyć odrobinę kod programów.

Sposób

Po pierwsze, należy zdecydować, jakiego rodzaju operacje dany obiekt powinien obsługiwać. Weźmy standardowy przykład składnicy danych obsługiwanej przez internet. Potrzebna nam będzie możliwość pobierania elementu, wstawiania elementu, aktualizowania elementu oraz usuwania go. Następnie należy zidentyfikować typy dostępu: dla potrzeb tworzenia, odczytywania, zapisywania i usuwania.

Można by oczywiście przechowywać w kodzie lub w pliku konfiguracyjnym listę mapującą wszystkie wymienione sposoby dostępu na odpowiednie metody obiektów wykorzystywanych w naszym programie (systemie zarządzania składnicą). To jednak byłoby idiotycznie skomplikowane — w końcu pracujemy w Perlu! Zamiast tego lepiej skorzystać z „atrybutów procedur" **[Sposób 45.]**.

```
package Proxy::AccessControl;

 use strict;
 use warnings;

 use Attribute::Handlers;

 my %perms;

 sub UNIVERSAL::perms
 {
     my ($package, $symbol, $referent, $attr, $data) = @_;
     my $method                                      = *{ $symbol }{NAME};

     for my $permission (split(/\s+/, $data))
     {
         push @{ $perms{ $package }{ $method } }, $permission;
     }
 }

 sub dispatch
 {
     my ($user, $class, $method, @args) = @_;

     return unless $perms{ $class }{ $method } and $class->can( $method );

     for my $perm (@{ $perms{ $class }{ $method } })
     {
       die "Potrzebne uprawnienia '$perm\n'" unless $user->has_permission(
       $perm );
     }

     $class->$method( @args );
 }

 1;
```

Deklarowanie uprawnień jest proste:

```
package Inventory;

use Proxy::AccessControl;

sub insert :perms( 'create' )
{
    my ($self, $attributes) = @_;
    # …
}

sub delete :perms( 'delete' )
{
    my ($self, $id) = @_;
    # …
}

sub update :perms( 'write' )
{
    my ($self, $id, $attributes) = @_;
    # …
}
```

```
sub fetch :perms( 'read' )
{
    my ($self, $id) = @_;
    # …
}
```

Można także łączyć uprawnienia i w ten sposób dopasowywać je do naszych potrzeb:

```
sub clone :perms( 'read create' )
{
    my ($self, $id, $attributes) = @_;
    # …
}
```

Pakiet `Proxy::AccessControl` dostarcza procedury obsługującej atrybuty `perms`, która rejestruje dzieloną spacjami listę uprawnień dla każdej zaznaczonej metody. Dostarcza również metody `dispatch()` — pełniącej funkcję bariery chroniącej system pomiędzy nadchodzącymi żądaniami przesyłanymi przez kontroler (ang. *controller*) a obiektami Perla, które będą obsługiwać żądania.

Jedyną rzeczą, która pozostała do zrobienia (poza napisaniem kodu tworzącego logikę biznesową programu), jest sprawienie, aby nasz kontroler przesyłał wszystkie dane za pośrednictwem metody `Proxy::AccessControl::dispatch()`. Funkcja ta wymaga przesłania jej trzech parametrów. Pierwszy parametr `$user` reprezentuje w pewnym sensie możliwości dostępu, które ma zewnętrzny użytkownik. (Nasz kod powinien umożliwiać tworzenie i uwierzytelnianie tego obiektu). Parametry `$class` i `$method` identyfikują odpowiednio klasę i wywoływaną metodę, jeśli użytkownik ma uprawnienia do jej wywołania.

Eksplorowanie sposobu

Metoda `dispatch()` jest dość prostą metodą pośredniczącą. Być może warto byłoby przygotować odpowiednich specjalnych pośredników (proxy), dopasowanych do konkretnych usług WWW lub protokołów używanych przez zdalne obiekty. Za kulisami mogłyby one pobierać tylko jeden dodatkowy parametr (obiekt użytkownika) i dla każdej metody pośredniczącej dostarczałyby własnej implementacji proxy wykonującej odpowiednie testy dostępu, a następnie odpowiednio przekazującej dalej lub odrzucającej żądania.

Ponadto można by również rozszerzyć kontrolę dostępu, by nie ograniczała się tylko do sprawdzania uprawnień. Można na przykład kontrolować dostęp do obiektów w zależności od liczby równolegle wykonywanych prób sięgania do nich, fazy księżyca, rodzaju zdalnego systemu operacyjnego, pory dnia czy jakiegokolwiek innego parametru. Każde dane, które można umieścić w atrybucie, będą akceptowalne.

SPOSÓB 49.

Przygotowywanie naprawdę polimorficznych obiektów

Buduj klasy, opierając się na tym, co będą robić, a nie na tym, skąd dziedziczą

Wiele przewodników i książek poświęconych programowaniu stara się przekonać Czytelnika, że centralnym elementem programowania obiektowego jest dziedziczenie. Co nie jest prawdą.

Znacznie ważniejszy jest *polimorfizm*. Oznacza to, że gdy przywołujemy procedurę `log()` na obiekcie, który potrafi zapisywać w dzienniku swój wewnętrzny status, obiekt zapisze ten status, a nie to, że dziedziczy ze znajdującej się gdzieś jakiejś abstrakcyjnej klasy `Logger` czy też wylicza dla nas jakiś abstrakcyjny dziennik. Perl 6 ułatwia ten rodzaj programowania, udostępniając role. W Perlu 5 natomiast można albo zbudować polimorfizm samemu, albo skorzystać z modułu `Class::Trait`, by rozbić złożone operacje na bardziej naturalne, nazwane grupami metod.

Brzmi to okropnie abstrakcyjnie — niemniej, jeśli mamy złożony problem, który *jesteśmy* w stanie odpowiednio rozłożyć na czynniki, to możemy bez trudu napisać odpowiedni kod i uzyskać efekt polimorfizmu.

Sposób

Wyobraźmy sobie, że budujemy aplikację zaopatrzoną w odpowiedni, dokonujący abstrakcji model obiektów, perspektywę (ang. *view*) i kontroler. Przygotowaliśmy wiele typów służących do zwracania danych — standardowy obiekt dla języka XHTML, odpowiednio przykrojony dla języka XHTML używanego przez urządzenia przenośne i obiekty zwracające dane w formacie Ajax lub JSON dla usług WWW opartych na REST, jak również przyjazne dla oprogramowania opiekującego się interfejsem użytkownika.

Każda możliwa perspektywa posiada odpowiadającą jej klasę perspektywy. Jak do tej pory, cały projekt ma sens. Teraz pojawia się jednak pytanie, w jaki sposób ustalać, z której perspektywy należy korzystać, gdy kod aplikacji będzie otrzymywał i rozpatrywał kolejne żądania? Co gorsza, jeśli mamy wiele perspektyw, to w jaki sposób zbudować odpowiednie klasy bez popadania w szaleństwo, próbując rozważać wszystkie możliwe kombinacje?

Jeśli zdecydujemy się na drobne oszustwo i zadeklarujemy perspektywy jako cechy (ang. *traits*), to możliwe będzie zastosowanie ich na obiektach tworzących model i, co za tym idzie, odpowiednie zarządzanie danymi.

Oto przykład modelu, z którego dziedziczą dwie konkretne klasy wuja `Uncle` i bratanka `Nephew`:

```
package Model;

 sub new
 {
     my ($class, %args) = @_;
     bless \%args, $class;
```

```
}

sub get_data
{
    my $self = shift;
    my %data = map { $_ => $self->{$_} } qw( imie profesja wiek );
    return \%data;
}

1;
```

Perspektywy są również bardzo proste:

```
package View;

use Class::Trait 'base';

package TextView;

use base 'View';

sub render
{
    my $self = shift;
    printf( "Nazywam się %s.  Moja profesja to %s i mam %d lat.\n",
        @{ $self->get_data(  ) }{qw( imie profesja wiek )} );
}

package YAMLView;

use YAML;
use base 'View';

sub render
{
    my $self = shift;
    print Dump $self->get_data(  );
}

1;
```

Tekstowa perspektywa wyświetla ładnie sformatowany łańcuch tekstu, podczas gdy perspektywa YAML zwraca serializowaną wersję struktury danych. Teraz klasa kontrolera musi tylko utworzyć odpowiedni model obiektowy i zanim przywoła procedurę `render()`, zastosować go na odpowiedniej perspektywie:

```
# wykorzystujemy model i oglądamy klasy

# tworzymy odpowiednie obiekty modelu
my $uncle  = Uncle->new(
    imie => 'Robert', profesja => 'Wuj', wiek => 50
);
my $nephew = Nephew->new(
    imie => 'Jakub', profesja => 'Agent Chaosu', wiek => 3
);

# stosujemy odpowiednie perspektywy
Class::Trait->apply( $uncle,  'TextView' );
Class::Trait->apply( $nephew, 'YAMLView' );
```

```
# wyświetlamy wyniki
$uncle->render(  );
$nephew->render(  );
```

Uruchamianie sposobu

Kod ten wyświetla następujące informacje:

```
Nazywam się Robert.  Pracuję jako Wuj i mam 50 lat.
---
imie: Jakub
profesja: Agent Chaosu
wiek: 3
```

Eksplorowanie sposobu

Jeśli nawet za pomocą ról i cech (ang. *traits*) można byłoby osiągnąć tylko tyle, to i tak byłyby one już niezmiernie użyteczne. Oferują nam jednak znacznie więcej! Moduł `Class::Traits` dostarcza metodę `does()`, którą można wykorzystać, by sprawdzić możliwości obiektu. Zakładając, że możemy otrzymać obiekt, który ma już wbudowaną perspektywę (na przykład model debugowania), należy przywołać metodę `does`, aby upewnić się, czy naprawdę posiada on już perspektywę:

```
Class::Trait->apply( $uncle, $view_type ) unless $uncle->does( 'View' );
```

Ponadto cechy wcale nie muszą dziedziczyć z bazowej cechy. Jeśli cały kod korzystający z obiektów i klas za pomocą cech będzie wykonywał testy za pomocą metody `does()`, a nie za pomocą metody `isa()` Perla, to będziemy mogli korzystać z cech, które będą robić to, co trzeba, niepowiązanych żadnymi relacjami ani stosunkiem dziedziczenia z żadną inną cechą.

Przydaje się to szczególnie w przypadku modeli i perspektyw, które korzystają z pośredników (proxies) lub wykonują zapisywanie w dziennikach.

SPOSÓB 50.

Automatyczne generowanie metod dostępu

Nie pisz dłużej metod dostępu ręcznie

Jedną z zalet Perla jest oszczędzanie programistom pracy. Nie oznacza to oczywiście, że nie trzeba w ogóle pracować, ale że można swoją pracę wykonać przy minimum włożonego wysiłku. W końcu nikt nie ma ochoty po raz kolejny wpisywać tego samego kodu. Niech komputer się tym zajmie.

Metody dostępu (ang. *accessors*) i metody modyfikujące (ang. *mutators*), lub inaczej, metody pobierające (ang. *getters*) i ustawiające (ang. *setters*) dane, są właśnie przykładem takiego kodu. Oto prosty obiektowy moduł:

```
package My::Customer;

 use strict;
 use warnings;
```

```
sub new { bless {  }, shift }

sub first_name
{
    my $self            = shift;
    return $self->{first_name} unless @_;
    $self->{first_name} = shift;
    return $self;
}

sub last_name
{
    my $self           = shift;
    return $self->{last_name} unless @_;
    $self->{last_name} = shift;
    return $self;
}

sub full_name
{
    my $self = shift;
    return join ' ', $self->first_name(  ), $self->last_name(  );
}

1;
```

Oraz prosty program, który z niego korzysta:

```
my $cust = My::Customer->new(  );
$cust->first_name( 'Jan' );
$cust->last_name( 'Publiczny' );
print $cust->full_name(  );
```

I wyświetla tekst `Jan Publiczny`.

Oczywiście, gdyby to naprawdę był obiekt reprezentujący klienta, musiałby robić coś więcej. Na przykład można by było określać wypłacalność kredytową klienta, tożsamość głównego sprzedawcy, który go obsługuje, itd.

Jak widać, metody pobierające imię `first_name` i nazwisko `last_name` są prawie identyczne. Nowe metody dostępu również zapewne będą bardzo podobne. Czy nie dałoby się tego jakoś zautomatyzować?

Sposób

W sieci CPAN dostępnych jest wiele modułów, które potrafią sobie poradzić z tym zadaniem, każdy w odrobinę inny sposób. Tutaj przedstawię dwa przykłady takich modułów — jeden najbardziej wszechstronny, a drugi narzucający programiście najmniejsze ograniczenia.

Class::MethodMaker

Jednym z najstarszych takich modułów jest `Class::MethodMaker`, po raz pierwszy opublikowany w 1996 roku. Posiada bardzo bogaty zestaw funkcji i choć jego dokumentacja pozostawia trochę do życzenia, z samego modułu korzysta się z łatwością. Aby przekonwertować kod wcześniejszego pakietu `My::Customer`, należy napisać:

```
package My::Customer;

 use strict;
 use warnings;

 use Class::MethodMaker[
     new    => [qw( new )],
     scalar => [qw( first_name last_name )],];

 sub full_name
 {
     my $self = shift;
     return join ' ', $self->first_name(  ), $self->last_name(  );
 }
```

Konstruktor, jak widać, jest bardzo prosty, ale co stało się z metodami `first_name` i `last_name`? Argumenty przesłane modułowi `Class::MethodMaker` polecają mu utworzyć dwa komplety metody dostępu i metody modyfikującej dla dwóch wartości skalarnych. Niemniej, mimo iż kod ten wygląda prawie identycznie, ma jednak znacznie większe możliwości.

Załóżmy, że chcemy sprawdzić, czy ktoś już definiował wartość zmiennej `first_name`, czy też ma przypisaną wartość `undef`:

```
print $cust->first_name_isset(  ) ? 'true' : 'false';
```

Nawet jeśli zmienna imienia `first_name` ma wartość `undef`, metoda `first_name()` zwróci wartość `true()`. Oczywiście, czasami wygodniej byłoby, aby zmienna miała status zmiennej jeszcze nieokreślonej, nawet jeśli wcześniej przypisana była już jej jakaś wartość. To również da się zrobić:

```
$cust->first_name( 'Ozymandias' );
print $cust->first_name_isset(  ) ? 'true' : 'false'; # prawda - true
$cust->first_name_reset(  );
print $cust->first_name_isset(  ) ? 'true' : 'false'; # fałsz - false
```

Class::BuildMethods

Moduł `Class::MethodMaker` obsługuje również tablice, tablice asocjacyjne i wiele innych użytecznych funkcji. Niemniej wymaga, aby w obiektach korzystać z błogosławionych (za pomocą funkcji `bless`) tablic asocjacyjnych. Prawdę powiedziawszy, większość modułów z sieci CPAN, które tworzą metody dostępu, przyjmuje jakieś założenia na temat wewnętrznej struktury naszych obiektów. Jednym z nielicznych wyjątków jest moduł `Class::BuildMethods`.

Moduł `Class::BuildMethods` umożliwia programiście budowanie metod dostępu dla tworzonej klasy niezależnie od tego, czy oparta jest ona na błogosławionej tablicy asocjacyjnej, odwołaniu do tablicy, wyrażeniu regularnym, czy jeszcze czymś innym. Osiąga to, „sięgając po trik wykorzystywany przy tworzeniu prawidłowo zamkniętych obiektów” **[Sposób 43.]**. Typowy kod tworzący z jego pomocą klasę będzie wyglądał mniej więcej tak:

```
package My::Customer;

 use strict;
 use warnings;

 use Class::BuildMethods qw(
   first_name
   last_name
 );

 # Warto zauważyć, że jeśli wolimy, możemy użyć odwołania do tablicy
 sub new { bless [  ], shift }

 sub full_name
 {
     my $self = shift;
     return join ' ', $self->first_name(  ), $self->last_name(  );
 }

 1;
```

Z klasy tej korzysta się taka samo jak z każdej innej. Wewnętrznie indeksuje ona wartości metod dostępu według adresu obiektu. Standardowo automatycznie zajmuje się niszczeniem obiektu, niemniej pozwala też programiście zrobić to ręcznie, jeśli potrzebować będzie jakiegoś specjalnego zachowania w metodzie `DESTROY` (takiego jak na przykład zdjęcie wcześniej założonych blokad).

Konstrukcja modułu `Class::BuildMethods` jest bardzo prosta. Podobnie jak większość innych modułów zajmujących się generowaniem metod dostępu dla obiektów, dostarcza programiście kilku wygodnych funkcji, jednak tylko wtedy, gdy chodzi o domyślne wartości i sprawdzanie danych:

```
use Class::BuildMethods
  'imie',
  gender => { default  => 'mezczyzna' },
  age    => { validate => sub
  {
      my ($self, $age) = @_;
      carp 'Nie możesz studiować, jeśli jesteś istotą niższą'
          if ( $age < 18 && ! $self->is_emancipated(  ) );
  }};
```

W tym kodzie związana z płcią metoda `gender()` zwróci wartość `male` (mężczyzna), chyba że ustawimy w zmiennej jakąś inną wartość. Związana z wiekiem metoda `age()` demonstruje natomiast, jak przygotować elastyczny mechanizm sprawdzania wartości. Ponieważ metoda `validate()` wskazuje do odwołania do procedury, a nie dostarcza specjalnych procedur sprawdzających wartości, przyjęte przez autora założenia co do sposobu sprawdzania kodu nie będą nas w żadnym stopniu ograniczać.

Moduł `Class::BuildMethods` zawsze zakłada, że metody ustawiające wartości (metody modyfikujące) zawsze będą wymagały tylko jednego argumentu, programista musi więc pamiętać, aby w razie czego przesyłać im odwołania do tablic i tablic asocjacyjnych.

Ponadto moduł ten nie obsługuje tworzenia metod klasy (tj. metod statycznych). Ograniczenia te oznaczają, że zaprezentowany tu kod nie w każdej sytuacji będzie spełniał potrzeby programisty, ale nasz przykładowy moduł miał z założenia być bardzo porosty. Dokumentację modułu można przeczytać i zrozumieć za jednym posiedzeniem.

Uruchamianie sposobu

Automatyczne generowanie metod dostępu stanowi dla programisty masę pracy. Wyzwala jego umysł z monotonii powtarzalnej pracy. Doświadczony programista Perla wie bowiem, że w programowaniu inteligentne lenistwo polega na tym, że każda zaoszczędzona minuta, której nie stracimy, zajmując się nudnymi szczegółami, może zostać poświęcona na rozwiązywanie naprawdę poważnych problemów.