Perl dla średnio zaawansowanych

Autorzy: Randal L. Schwartz, Brian d foy, Tom Phoenix
Tłumaczenie: Tomasz Walczak
ISBN: 83-246-0615-7
Tytuł oryginału: Intermediate Perl
Format: B5, stron: 264

Poznaj zasady programowania obiektowego w Perlu

- Utwórz i wykorzystaj moduły
- Opanuj tajniki stosowania przestrzeni nazw
- Umieść dystrybucje swoich programów w archiwum CPAN

Perl jest uniwersalnym i skutecznym językiem programowania mającym wiele zastosowań. Można wykorzystywać go do zarządzania systemami, tworzyć za jego pomocą dynamiczne witryny internetowe i manipulować danymi zgromadzonymi w tabelach baz danych. Programiści stosujący Perla twierdzą, że dzięki niemu łatwe zadania stają się jeszcze łatwiejsze, a trudne – wykonalne. Mimo iż Perl był projektowany jako język nieobiektowy, można podczas programowania w nim stosować również koncepcje obiektowe. Aby jednak używać obiektowego języka Perl, najpierw warto zrozumieć działanie pakietów, referencji, tablic asocjacyjnych, tablic, procedur i modułów.

Dzięki książce „Perl dla średnio zaawansowanych" poznasz wszystkie komponenty języka stanowiące podstawę jego obiektowych właściwości. Czytając ją, nauczysz się korzystać z modułów i referencji, manipulować złożonymi strukturami danych za pomocą pakietu Data::Dumper oraz pracować z systemem plików. Przyswoisz sobie zasady programowania obiektowego oraz dowiesz się, jak w Perlu tworzyć obiekty i usuwać je oraz budować aplikacje o skomplikowanej strukturze. Przeczytasz także o testowaniu kodu, opracowywaniu dystrybucji i umieszczaniu ich w archiwach CPAN.

- Obsługa list za pomocą operatorów
- Instalowanie modułów z archiwów CPAN
- Korzystanie z modułów
- Tworzenie tablic anonimowych i asocjacyjnych
- Wyświetlanie i przetwarzanie złożonych danych
- Obsługa plików i katalogów za pomocą referencji
- Sortowanie
- Obiekty i egzemplarze
- Wywoływanie metod
- Usuwanie obiektów
- Tworzenie dystrybucji
- Pisanie skryptów testowych
- Tworzenie własnych modułów Test::*

Poznaj obiektowe właściwości języka Perl

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

Spis treści

ROZDZIAŁ 3.

Używanie modułów

Moduły to cegiełki, z których możemy budować programy. Udostępniają one nadające się do powtórnego wykorzystania procedury, zmienne, a nawet obiektowe klasy. Zanim nauczysz się tworzyć własne moduły, pokażemy Ci niektóre gotowe moduły, które mogą Cię zainteresować. Poznasz także podstawy stosowania modułów napisanych przez innych programistów.

Dystrybucja standardowa

Perl zawiera wiele popularnych modułów. W rzeczywistości większość z ponad 50 megabajtów najnowszej dystrybucji to właśnie moduły. W październiku 1996 roku Perl 5.003_07 zawierał 98 modułów. Obecnie, na początku 2006 roku, Perl 5.8.8 ma ich 359[1]. Jest to jedna z zalet języka Perl — udostępnia on wiele elementów pozwalających małym nakładem pracy tworzyć użyteczne i złożone programy.

W niniejszej książce postaramy się pokazać, które moduły są dostępne wraz z językiem Perl (i, w większości przypadków, od której wersji języka dostępny jest dany moduł). Będziemy nazywać te moduły „modułami podstawowymi" lub napiszemy, że znajdują się one w „dystrybucji standardowej". Jeśli masz język Perl, powinieneś mieć również te moduły. Ponieważ przy pisaniu książki używaliśmy wersji Perl 5.8.7, zakładamy, że jest to bieżąca wersja języka.

Kiedy tworzysz własny kod, możesz zdecydować się na używanie jedynie modułów podstawowych, dzięki czemu będziesz miał pewność, że wszyscy użytkownicy Perla będą mieli te moduły, o ile tylko używają przynajmniej tej samej wersji co Ty[2]. Nie chcemy wdawać się w tym miejscu w spory na ten temat, głównie dlatego, że zbytnio cenimy archiwum CPAN, aby się bez niego obejść.

[1] Kiedy zapoznasz się z materiałem tej książki, powinieneś potrafić użyć modułu `Module::CoreList` do samodzielnego sprawdzenia liczby modułów. To właśnie w taki sposób określiliśmy powyższe liczby.

[2] Choć nie będziemy zgłębiać tu tego zagadnienia, moduł `Module::CoreList` zawiera listę opisującą, które moduły dostępne są w danej wersji języka Perl, a także inne dane historyczne.

Używanie modułów

Prawie każdy moduł języka Perl zawiera dokumentację, a choć możemy nie wiedzieć, jak działają mechanizmy danego modułu, nie musimy się o to martwić, jeśli wiemy, jak używać jego interfejsu. W końcu do tego właśnie służy interfejs — pozwala ukryć wszystkie szczegóły.

Na naszej lokalnej maszynie możemy wczytać dokumentację modułu, używając polecenia `perldoc`. Do polecenia należy przekazać nazwę interesującego nas modułu, a program wyświetli jego dokumentację.

```
$ perldoc File::Basename

NAME
           fileparse - split a pathname into pieces

           basename - extract just the filename from a path

           dirname - extract just the directory from a path
SYNOPSIS
                   use File::Basename;

                   ($name,$path,$suffix) = fileparse($fullname,@suffixlist)
                   fileparse_set_fstype($os_string);
                   $basename = basename($fullname,@suffixlist);
                   $dirname = dirname($fullname);
```

Dołączyliśmy początkowy fragment dokumentacji, aby pokazać najważniejszą sekcję (przynajmniej na początku nauki). Dokumentacja modułów zwykle jest zgodna ze starym formatem stron podręcznika man systemu Unix i rozpoczyna się od sekcji `NAME` oraz `SYNOPSIS` (zawierającej streszczenie).

Streszczenie przedstawia przykłady zastosowania modułu, a jeśli potrafisz obyć się bez pełnego zrozumienia, możesz używać danego modułu. Oznacza to, że możesz nie znać niektórych technik i składni języka Perl przedstawionych w streszczeniu, ale zwykle możesz po prostu posłużyć się przykładem, a kod powinien działać poprawnie.

Ponieważ Perl to mieszanka proceduralnego, funkcyjnego, obiektowego i innych rodzajów języków programowania, moduły języka Perl udostępniają interfejsy różnego typu. Będziemy używać rozmaitych modułów w nieco odmienny sposób, ale dopóty, dopóki możesz sprawdzić działanie modułu w dokumentacji, nie powinieneś mieć problemów.

Interfejsy funkcyjne

Do wczytywania modułów posłużymy się wbudowaną instrukcją `use` języka Perl. Na razie pominiemy szczegółowy opis tej techniki. Więcej dowiesz się o niej z rozdziałów 10. i 15. Na razie chcemy tylko użyć modułu. Zacznijmy od `File::Basename`, użytego już modułu z dystrybucji standardowej. Aby wczytać go do skryptu, należy wywołać poniższą instrukcję:

```
use File::Basename;
```

Kiedy to zrobimy, moduł `File::Basename` udostępni w skrypcie[3] trzy procedury: `fileparse`, `basename` i `dirname`[4]. Od tego miejsca możemy używać poniższych instrukcji:

[3] Tak naprawdę procedury zostaną zaimportowane do bieżącego pakietu, ale nie omawialiśmy jeszcze tego zagadnienia.

[4] A także procedurę narzędziową `fileparse_set_fstype`.

```
my $basename = basename( $some_full_path );
my $dirname  = dirname(  $some_full_path );
```

w taki sam sposób, jakbyśmy sami napisali procedury `basename` i `dirname` lub (prawie) jakby były one wbudowanymi funkcjami języka Perl. Te procedury pobierają ze ścieżki nazwę pliku oraz nazwę katalogu. Na przykład, jeśli zmienna `$some_full_path` ma wartość `D:\Projects\IslandRescue\plan7.rtf` (prawdopodobnie program został uruchomiony w systemie Windows), wartością zmiennej `$basename` będzie `plan7.rtf`, a wartością zmiennej `$dirname` — `D:\Projects\IslandRescue`.

Moduł `File::Basename` potrafi określić, w jakim systemie działa, dlatego jego funkcje poprawnie przetwarzają łańcuchy znaków według różnych ograniczników, jakie mogą napotkać.

Załóżmy jednak, że w programie znajduje się już procedura `dirname`. Zostanie ona przesłonięta definicją z modułu `File::Basename`! Jeśli włączymy ostrzeżenia, zobaczymy informujący o tym komunikat. W przeciwnym razie Perl ignoruje taką sytuację.

Wybór importowanych elementów

Na szczęście możemy ograniczyć operacje wykonywane przez instrukcję `use`, podając nazwę modułu oraz listę nazw procedur. Ta lista to *lista importu*:

```
use File::Basename ('fileparse', 'basename');
```

Teraz moduł wczyta jedynie dwie powyższe procedury i pozostawi naszą procedurę `dirname` bez zmian. Oczywiście taki zapis jest dość dziwaczny, dlatego częściej używa się operatora cytowania:

```
use File::Basename qw( fileparse basename );
```

Nawet jeśli na liście znajduje się tylko jeden element, zwykle podaje się go na liście `qw()`, co zwiększa spójność i ułatwia pielęgnację. Często się zdarza, że stwierdzamy: „potrzebujemy jeszcze jednej procedury z tego modułu" i wracamy do tej listy. Łatwiej jest dodać nową procedurę, jeśli pierwsza znajduje się już na liście `qw()`.

Ochroniliśmy lokalną procedurę `dirname`, ale co zrobić, jeśli potrzebujemy także funkcjonalności tej procedury z modułu `File::Basename`? To proste. Wystarczy użyć jej, poprzedzając ją pełną nazwą pakietu.

```
my $dirname = File::Basename::dirname($some_path);
```

Lista nazw znajdująca się po instrukcji `use` nie zmienia tego, jakie procedury są zdefiniowane w pakiecie modułu (w tym przypadku jest to moduł `File::Basename`). Zawsze możemy użyć pełnej nazwy, niezależnie od zawartości listy importu, na przykład[5]:

```
my $basename = File::Basename::basename($some_path);
```

Ekstremalnym (ale niezwykle użytecznym) rozwiązaniem jest użyciu pustej listy importu, tak jak w poniższym kodzie:

```
use File::Basename()      # nie importuje niczego
my $base = File::Basename::basename($some_path);
```

[5] Nie musisz poprzedzać wywołań tych procedur znakiem ampersandu, ponieważ od przetworzenia instrukcji `use` kompilator już je zna.

Pusta lista różni się od braku listy. Pusta lista informuje: „nie należy importować żadnych elementów”, podczas gdy brak listy oznacza: „należy zaimportować domyślne elementy”. Jeśli autor modułu napisał go poprawnie, domyślne elementy to te, których potrzebujemy.

Interfejsy obiektowe

Porównajmy importowanie procedur z modułu `File::Basename` z działaniem innego modułu podstawowego, `File::Spec`. Moduł `File::Spec` służy do obsługi operacji zwykle wykonywanych na identyfikatorze pliku. Identyfikator pliku to zwykle nazwa pliku lub katalogu, ale może to być także nazwa nieistniejącego pliku — w takim przypadku nie jest to tak naprawdę nazwa pliku, prawda?

W przeciwieństwie do modułu `File::Basename` moduł `File::Spec` ma interfejs obiektowy. Wczytujemy ten moduł przy użyciu instrukcji `use`, tak samo jak wcześniej.

```
use File::Spec;
```

Jednak, ponieważ interfejs tego modułu jest obiektowy[6], powyższa instrukcja nie importuje żadnych procedur. W zamian interfejs pozwala na dostęp do funkcjonalności modułu poprzez metody klasy. Metoda `catfile` złącza listę łańcuchów znaków, używając odpowiedniego separatora katalogów:

```
my $filespec = File::Spec->catfile( $homedir{gilligan},
        'web_docs', 'photos', 'USS_Minnow.gif' );
```

Powyższy kod wywołuje metodę statycznej `catfile` klasy `File::Spec`. Ta metoda tworzy ścieżkę odpowiednią dla danego systemu operacyjnego i zwraca jeden łańcuch znaków[7]. Ze względu na składnię powyższy zapis jest podobny do ponad dwudziestu innych operacji udostępnianych przez moduł `File::Spec`.

Moduł `File::Spec` udostępnia kilka innych metod pozwalających na obsługę ścieżek plików w sposób przenośny. Więcej o zagadnieniach związanych z przenośnością kodu dowiesz się z dokumentacji *perlport*.

Bardziej typowy moduł obiektowy — Math::BigInt

Abyś się nie rozczarował nieobiektowym wyglądem modułu `File::Spec`, który nie ma obiektów, przyjrzyjmy się następnemu modułowi podstawowemu, `Math::BigInt`, który służy do obsługi liczb całkowitych przekraczających wbudowane możliwości języka Perl[8].

```
use Math::BigInt;

my $value = Math::BigInt->new(2);    # rozpoczynamy od 2

$value->bpow(1000);                  # przyjmuje wartość 2**1000

print $value->bstr( ), "\n";         # wyświetla wynik
```

[6] Jeśli potrzebny jest interfejs funkcyjny, można użyć instrukcji `File::Spec::Functions`.

[7] W systemie Unix tym łańcuchem znaków może być na przykład `/home/gilligan/web_docs/USS_Minnow.gif`. W systemie Windows separatorami katalogów są zwykle lewe ukośniki. Ten moduł pozwala w łatwy sposób napisać przenośny kod (przynajmniej ze względu na specyfikacje plików).

[8] Na zapleczu język Perl jest ograniczony architekturą, w której działa. Jest to jedno z niewielu miejsc, gdzie trzeba zwracać uwagę na sprzęt.

Także ten moduł nie wymaga importowania żadnych jednostek. Cały jego interfejs składa się z metod statycznych, takich jak metoda `new`, którą należy wywoływać wraz z nazwą klasy, aby utworzyć jej egzemplarz. Następnie można wywoływać przy użyciu tych egzemplarzy metody egzemplarza, takie jak `bpow` czy `bstr`.

Archiwum CPAN

Archiwum CPAN (ang. *Comprehensive Perl Archive Network*) to wynik wspólnej pracy wielu wolontariuszy. Wielu z nich początkowo prowadziło własne małe (lub duże) witryny FTP o języku Perl, jeszcze zanim narodził się internet. Pod koniec 1993 roku zjednoczyli oni swe wysiłki dzięki liście dyskusyjnej *perl-packrats* i zdecydowali, że przestrzeń dyskowa stała się na tyle tania, że można powielić te same informacje na wszystkich witrynach, zamiast dążyć do specjalizacji każdej z nich. Ten pomysł dojrzewał przez rok, aż w końcu Jarkko Hietaniemi utworzył fińską witrynę FTP jako źródło, z którego wszystkie serwery „lustrzane" mogą pobierać codzienne lub nawet cogodzinne aktualizacje.

Część pracy nad tym projektem wymagała ponownego uporządkowania i zorganizowania odrębnych archiwów. Zostały utworzone miejsca dla binariów języka Perl przeznaczonych dla architektur niebazujących na systemie Unix, dla skryptów oraz dla samego kodu źródłowego w języku Perl. Jednak największym i najciekawszym elementem archiwum CPAN są moduły.

Moduły w archiwum CPAN są zorganizowane w drzewo dowiązań symbolicznych według hierarchicznych, funkcjonalnych kategorii. Odnośniki wskazują na katalog autora, w którym to katalogu znajdują się potrzebne pliki. Obszar zawierający moduły zawiera także indeksy mające zwykle format łatwy do przetworzenia przy użyciu języka Perl, na przykład dane wyjściowe z modułu `Data::Dumper` w przypadku szczegółowego indeksu modułów. Oczywiście wszystkie te indeksy są automatycznie generowane na podstawie baz danych głównego serwera, do czego służą inne programy w języku Perl. Często odzwierciedlanie archiwum CPAN z jednego serwera na inny odbywa się przy użyciu wiekowego już programu `mirror.pl`, napisanego w języku Perl.

Od skromnych początków w postaci kilku serwerów „lustrzanych" archiwum CPAN rozrosło się do ponad 200 publicznych archiwów dostępnych we wszystkich zakątkach internetu. Wszystkie serwery są archiwizowane i aktualizowane przynajmniej raz dzienie, a niektóre nawet co godzinę. Niezależnie od tego, w którym miejscu świata jesteś, znajdziesz w pobliżu serwer CPAN, z którego możesz pobrać najnowsze zasoby.

Niezwykle użyteczna strona CPAN Search (*http://search.cpan.org*) stanie się prawdopodobnie Twoim ulubionym interfejsem. Dzięki tej witrynie możesz wyszukiwać moduły, sprawdzać ich dokumentację, przeglądać dystrybucje i kontrolować raporty testerów CPAN i wykonywać wiele innych operacji.

Instalowanie modułów z archiwum CPAN

Zainstalowanie prostego modułu z CPAN nie powinno sprawiać trudności — wystarczy pobrać archiwum z dystrybucją modułu, rozpakować je i umieścić moduł w odpowiednim katalogu. W poniższym kodzie używamy instrukcji `wget`, ale możesz używać także innych narzędzi.

```
$ wget http://www.cpan.org/.../HTTP-Cookies-Safari-1.10.tar.gz
$ tar -xzf HTTP-Cookies-Safari-1.10.tar.gz
$ cd HTTP-Cookies-Safari-1.10s
```

Następnie możesz wybrać jedno z dwóch rozwiązań (opisujemy je szczegółowo w rozdziale 16.). Jeśli znajdziesz plik o nazwie `Makefile.PL`, możesz uruchomić poniższą sekwencję poleceń w celu skompilowania, przetestowania i zainstalowania kodu źródłowego:

```
$ perl Makefile.PL
$ make
$ make test
$ make install
```

Jeśli nie masz uprawnień do instalowania modułów w katalogach globalnych[9], możesz nakazać zainstalowanie ich w innej ścieżce, używając argumentu `PREFIX`:

```
$ perl Makefile.PL PREFIX=/Users/home/Ginger
```

Aby Perl szukał modułów w tym katalogu, możesz ustawić zmienną środowiskową `PERL5LIB`. Perl doda wtedy określone katalogi do listy katalogów, w których wyszukuje modułów.

```
$ export PERL5LIB=/Users/home/Ginger
```

Możesz także użyć dyrektywy `lib`, aby dodać katalog do ścieżki wyszukiwania modułów, jednak to rozwiązanie nie jest równie wygodne, ponieważ wymaga wprowadzania zmian w kodzie. Ponadto na innych komputerach, na których możesz chcieć uruchomić dany kod, potrzebny moduł może znajdować się w innym katalogu.

```
#!/usr/bin/perl
use lib qw(/Users/home/Ginger);
```

Cofnijmy się nieco. Jeśli znajdziesz plik `Build.PL` zamiast pliku `Makefile.PL`, powinieneś wykonać te same operacje. W przypadku dystrybucji udostępnianych z tym pierwszym plikiem należy używać modułu `Module::Build` do kompilowania i instalowania kodu. Ponieważ moduł `Module::Build` nie jest modułem podstawowym języka Perl[10], musisz go zainstalować, zanim będziesz mógł zainstalować potrzebną dystrybucję.

```
$ perl Build.PL
$ perl Build
$ perl Build test
$ perl Build install
```

Aby zainstalować moduł w prywatnym katalogu przy użyciu modułu `Module::Build`, należy dodać parametr `--install_base`. Informowanie interpretera Perl o tym, gdzie znajdują się moduły, odbywa się w taki sam sposób jak poprzednio.

```
$ perl Build.PL --install_base /Users/home/Ginger
```

Czasem w dystrybucji znajdują się oba pliki — `Makefile.PL` i `Build.PL`. Co należy wtedy zrobić? Można użyć dowolnego z nich. Możesz wybrać ten, który lepiej znasz.

---

[9] Te katalogi ustawia administrator w czasie instalowania języka Perl. Można je wyświetlić, używając instrukcji `perl -V`.

[10] Przynajmniej na razie. Planowane jest dołączenie go do modułów podstawowych od wersji 5.10 języka Perl.

Ustawianie ścieżki w odpowiednim momencie

Perl wyszukuje moduły, przeszukując katalogi znajdujące się w specjalnej tablicy języka Perl, `@INC`. Instrukcja `use` jest wykonywana na etapie kompilacji, dlatego sprawdzanie ścieżki prowadzącą do modułu, `@INC`, również ma miejsce na etapie kompilacji. Może to spowodować niepoprawne działanie programu, które trudno zrozumieć, o ile nie weźmie się pod uwagę zawartości tablicy `@INC`.

Na przykład, załóżmy, że masz własny katalog `/home/gilligan/lib` i umieściłeś własny moduł `Navigation::SeatOfPants` w pliku `/home/gilligan/lib/Navigation/SeatOfPants.pm`. Kiedy spróbujesz wczytać ten moduł, Perl nie będzie potrafił go znaleźć.

```
use Navigation::SeatOfPants
```

Perl wyświetli informację, że nie może znaleźć modułu w tablicy `@INC`, a następnie wyświetli wszystkie znajdujące się w niej katalogi.

```
Can't locate Navigation/SeatOfPants.pm in @INC (@INC contains: …)
```

Możesz wpaść na rozwiązanie polegające na dodaniu katalogu zawierającego moduł do tablicy `@INC` przed wywołaniem instrukcji `use`. Jednak nawet po dodaniu tej instrukcji:

```
unshift @INC, '/home/gilligan/lib';   # nie działa
use Navigation::SeatOfPants;
```

kod nie zadziała. Dlaczego? Ponieważ metoda `unshift` jest wywoływana w czasie wykonywania programu, długo po próbie wywołania instrukcji `use` mającej miejsce na etapie kompilacji. Te dwie instrukcje są blisko siebie w przestrzeni, ale nie w czasie. To, że napisałeś je jedna po drugiej, nie oznacza jeszcze, iż zostaną wykonane w takiej kolejności. Chcesz zmienić zawartość tablicy `@INC` przed wywołaniem instrukcji `use`. Jednym z rozwiązań jest dodanie bloku `BEGIN` wokół instrukcji `push`:

```
BEGIN { unshift @INC, '/home/gilligan/lib'; }
use Navigation::SeatOfPants;
```

Teraz blok `BEGIN` zostanie skompilowany i wykonany w czasie kompilacji, dzięki czemu ustawi odpowiednią ścieżkę dla instrukcji `use`.

Jednak to rozwiązanie jest nieeleganckie i może wymagać o wiele więcej wyjaśnień, niż jesteś skłonny przedstawić, szczególnie gdy odbiorcą jest programista, który w późniejszym czasie ma dbać o kod. Możesz zastąpić cały powyższy bałagan prostą dyrektywą, której używałeś już wcześniej:

```
use lib '/home/gilligan/lib';
use Navigation::SeatOfPants;
```

W powyższym rozwiązaniu dyrektywa `lib` przyjmuje jeden lub więcej argumentów i dodaje je na początek tablicy `@INC`, podobnie jak robi to instrukcja `unshift`[11]. To rozwiązanie działa, ponieważ jest wykonywane na etapie kompilacji, a nie w czasie wykonywania programu. Dlatego wszystko jest gotowe do wykonania instrukcji `use` występującej bezpośrednio poniżej.

Ponieważ dyrektywa `use lib` prawie zawsze używa ścieżki zależnej od systemu, jest to rozwiązanie tradycyjne i zalecamy umieszczanie tej instrukcji na początku pliku. Dzięki temu można ją łatwiej znaleźć i poprawić, kiedy chcesz przenieść plik do nowego systemu lub zmienić

[11] Instrukcja `use lib` ponadto dodaje zależną od architektury bibliotekę za żądaną biblioteką, przez co jest to bardziej wartościowe rozwiązanie niż przedstawiony wcześniej odpowiednik.

nazwę katalogu zawierającego lib. Oczywiście możesz całkowicie wyeliminować instrukcję use lib, jeśli zainstalujesz moduł w standardowej lokalizacji opisanej w tablicy @INC, ale nie zawsze jest to wygodne.

Instrukcja use lib nie znaczy „użyj danej biblioteki", ale raczej „użyj danej ścieżki do znalezienia bibliotek (i modułów)". Zbyt często spotykamy się z kodem podobnym do poniższego:

```
use lib '/home/gilligan/lib/Navigation/SeatOfPants.pm'; # ŹLE
```

Następnie programista zastanawia się, dlaczego instrukcja nie dodała definicji. Pamiętaj, że instrukcja use lib jest wykonywana na etapie kompilacji, dlatego poniższe rozwiązanie także nie jest poprawne:

```
my $LIB_DIR = '/home/gilligan/lib';
…
use lib $LIB_DIR;   # BŁĄD
use Navigation::SeatOfPants;
```

Oczywiście język Perl przetwarza deklarację zmiennej $LIB_DIR na etapie kompilacji (dlatego nie pojawi się błąd, jeśli użyjesz instrukcji use strict, natomiast próba wywołania use lib powinna zakończyć się niepowodzeniem), ale rzeczywiste przypisanie wartości /home/gilligan/lib/ ma miejsce dopiero w czasie wykonywania programu. Niestety, także w tym przypadku jest to zbyt późno.

Dlatego trzeba umieścić instrukcje w bloku BEGIN lub zdecydować się na inną operację wykonywaną na etapie kompilacji — na ustawienie wartości stałej przy użyciu instrukcji use constant:

```
use constant LIB_DIR => '/home/gilligan/lib';
…
use lib LIB_DIR;
use Navigation::SeatOfPants;
```

Gotowe. Ponownie działa, przynajmniej dopóty, dopóki określenie biblioteki nie zależy od wyników jakichś obliczeń. (Kiedy to się skończy? Niech ktoś przerwie to szaleństwo!) To rozwiązanie powinno działać poprawnie w około 99 procentach przypadków.

Obsługa zależności modułu

Widziałeś już, że jeśli spróbujesz zainstalować dany moduł przy użyciu Module::Build, musisz najpierw zainstalować sam moduł Module::Build. Jest to stosunkowo łagodny przykład bardziej ogólnego problemu z zależnościami, którego nie rozwiążą wszystkie kokosy na wyspie rozbitków. Może zajść potrzeba zainstalowania kilku innych modułów, które same wymagają następnych modułów.

Na szczęście dostępne są odpowiednie narzędzia. Moduł CPAN.pm jest częścią dystrybucji standardowej od wersji Perl 5.004. Ten moduł udostępnia interaktywną powłokę służącą do instalowania modułów.

```
$ perl -MCPAN -e shell
cpan shell -- CPAN exploration and module installation (v1.76O1)
ReadLine support available (try 'install Bundle::CPAN')

cpan>
```

Aby zainstalować moduł oraz jego zależności, należy wywołać instrukcję install i podać nazwę modułu. Wtedy moduł CPAN.pm obsłuży wszystkie zadania: pobranie, rozpakowanie,

skompilowanie, przetestowanie i zainstalowanie modułu, wykonując te operacje rekurencyjnie dla wszystkich zależności.

```
cpan> install CGI::Prototype
```

Wymaga to nieco zbyt wiele pracy, dlatego brian utworzył skrypt `cpan`, który także jest dostępny w Perlu. Wystarczy podać listę modułów, które chcesz zainstalować, a skrypt wykona wszystkie potrzebne operacje.

```
$ cpan CGI::Prototype HTTP::Cookies::Safari Test::Pod
```

Następne narzędzie, `CPANPLUS`, to zupełnie nowa wersja modułu `CPAN.pm`, która jednak na razie nie wchodzi w skład dystrybucji standardowej.

```
$ perl -MCPANPLUS -e shell
CPANPLUS::Shell::Default -- CPAN exploration and modules installation (v0.03)
*** Please report bugs to <cpanplus-bugs@lists.sourceforge.net>.
*** Using CPANPLUS::Backend v0.049.
*** ReadLine support available <try 'i Term::ReadLine::Perl').

CPAN Terminal>
```

Aby zainstalować wybrany moduł, należy użyć polecenia `i`.

```
CPAN Terminal> i CGI::Prototype
```

Moduł `CPANPLUS` także związany jest ze skryptem, który ułatwia jego używanie. Nazwa tego skryptu to `cpanp`. Jeśli przekażesz do skryptu opcję `i` oraz listę modułów, skrypt zainstaluje je, podobnie jak instrukcja przedstawiona powyżej.

```
$ cpanp i CGI::Prototype HTTP::Cookies::Safari Test::Pod
```

Ćwiczenia

Rozwiązania poniższych ćwiczeń znajdziesz w punkcie „Rozwiązania ćwiczeń z rozdziału 3.”, w dodatku A.

Ćwiczenie 1. (25 minut)

Wczytaj listę plików znajdujących się w bieżącym katalogu i przekształć ich nazwy na identyfikatory pełnej ścieżki. Do pobrania bieżącego katalogu nie używaj powłoki ani żadnego zewnętrznego programu. Wystarczą do tego moduły `File::Spec` oraz `Cwd`, oba dostępne w języku Perl. Wyświetl każdą ścieżkę, dodając cztery odstępy z jej przodu oraz znak nowego wiersza na końcu, podobnie jak zrobiłeś to w pierwszym ćwiczeniu w rozdziale 2. Czy potrafisz powtórnie wykorzystać fragment tamtego rozwiązania do wykonania tego zadania?

Ćwiczenie 2. (35 minut)

Przetwórz numer ISBN znajdujący się z tyłu tej książki (8324606157). Zainstaluj moduł `Business::CPAN` z archiwum CPAN i użyj go do pobrania z tego numeru kodu kraju oraz kodu wydawcy.