

Praca z debugerem Perla

KUŚTYKANIE DO WŁAŚCIWEJ DIAGNOZY

Niektórzy użytkownicy Perla potępiają korzystanie z debugerów, lecz w pewnych wypadkach debuger jest ostatnią deską ratunku programisty. W tym miesiącu przyjrzymy się zintegrowanemu debugerowi Perla.

MICHAEL SCHILLI

Linus Torvalds nie lubi debugerów ani programistów, którzy z nich korzystają. Zdaniem Linusa, za łatwo jest skleić parę linijek kodu i dopiero za pomocą debugera zrobić z nich coś sensownego. A programiści chaotyczni raczej nie wymyślą dobrego projektu programu. To bumerang powracający w postaci oprogramowania, które trudno pielęgnować czy rozbudowywać. Ostrożne stosowanie rejestracji w dzienniku w wielu wypadkach usuwa potrzebę debugowania. Jak można przeczytać w [3], moduł *Log::Log4perl* umożliwia wbudowanie odpowiedniego poziomu debugowania w aplikację i zdalne sterowanie czynnościami debugowania.

Przebieg testowy

Czasami programista nie ma wyjścia. W końcu, co mamy robić, jeśli program reaguje w nieoczekiwany sposób, dokumentacja nie daje żadnych wskazówek, a kod (napisany, oczywiście, przez kogoś innego) jest zbyt trudny do ogarnięcia po przeczytaniu listingów? Perl ma debugera, który szybko lokalizuje błędy przez stosowanie punktów wstrzymania, czynności i punktów kontrolnych. Na Listingu 1 przedstawiono praktyczny, pięciolinijkowy skrypt o nazwie *wsrv*, który informuje, jaki serwer internetowy kryje się za określonym adresem URL. Wywołanie *wsrv http://sun.com* informuje, że firma Sun korzysta z własnej technologii: Sun Java System Web Server 6.1.

Jeśli chcemy natomiast uruchomić skrypt w debugerze, po prostu wchodzimy do wiersza poleceń za pomocą *perl -d*, a następnie podajemy pełną ścieżkę do skryptu i jego argumentów, to znaczy *perl -d wsrv http://microsoft.com*. Uruchamia się debuger i czeka na polecenia użytkownika po wyświetleniu pierwszego wiersza kodu źródłowego (który składa się faktycznie z dwóch wierszy tekstu):

```
Loading DB routines from §§
perl5db.pl version 1.27
Editor support available.
Enter h or `h h' for help, or §§
`man perldebug' for more help.
main::(wsrv:8): my $url = shift
main::(wsrv:9): §§
or die „usage $0 url”;
```

Wprowadzenie litery *n* (skrót od *next* – dalej) w zgłoszeniu powoduje uruchomienie pierwszej instrukcji bez żadnych komunikatów, która pobiera adres URL z tablicy argumentów *@ARGV* i pozostawia go w zmiennej *\$url*. Ponieważ polecenie *n* uruchamia tylko jedną instrukcję, debuger zatrzymuje się tuż przed wykonaniem następnego wiersza:

```
DB<1> n
main::(wsrv:10): §§
my (@fields) = head ($url)
main::(wsrv:11): §§
or die „Fetch failed”;
```

Następna wykonywalna instrukcja składa się z wierszy 10 i 11 w skrypcie *wsrv*. Zamiast uruchamiać wiersz w całości poleceniem *n*, użyjmy polecenia *s* (od *step* – praca krokowa), aby dokładniej zobaczyć, co się dzieje. Jak widzimy, debuger zagłębia się w funkcję *head* wywołaną w wierszu 10:

```
LWP::Simple::head §§
(.../LWP/Simple.pm:70):
70: my($url)=@_;
```

Polecenie *l* (lista) wyświetla kilka następnnych wierszy:

```
70==>my ($url) = @_;
71: _init ua ()unless$ua;
72
73: my$request= §§
HTTP::Request->§§
new (HEAD=>$url);
74: my$response§§
=$ua->request ($request);
[...]
```

Aby przejść dalej, nie wykonując faktycznie kodu, możemy po prostu jeszcze raz nacisnąć klawisz *l*. Ewentualnie można określić zakres typu *l 70+20* (20 wierszy począwszy od wiersza 70) lub *l 70-100* (wiersze od 70 do 100). Następny wykonywalny wiersz jest oznaczony znakami *==>*.

Z powrotem do początku

Po przewinięciu w dół poleceniem *l*, wpisanie kropki powoduje przesunięcie listy z powrotem do miejsca, gdzie skrypt będzie wykonywany dalej. Wpisanie litery *r* (od return – powrót) nakazuje debuggerowi wykonywać kod do zakończenia bieżącej funkcji i powrotu do programu głównego. Następnie zatrzymuje się on automatycznie:

```
list context return $$
from LWP::Simple::head:
0 'text/html'
1 16144
2 1107018115
3 1107028115
4 'Microsoft-IIS/6.0'
main:(wsrv:9): print $$
„Server:$fields[4] \n”;
```

Debugger jest na tyle miły, że podaje nam wartość zwrótną funkcji *head* () po wyświetle-

niu kolejnego wykonywalnego wiersza. Jeśli interesuje nas wartość elementu tablicy *\$fields[4]*, możemy użyć polecenia *p* (od print – drukuj) w debugerze, aby wyświetlić wartość, zanim zrobi to instrukcja *print* () w programie głównym. Polecenie *p \$fields[4]* w wierszu poleceń zwraca tekst „Microsoft-IIS/6.0”, co wskazuje, że firma Microsoft także korzysta z własnej technologii serwerowej.

Aby wyświetlić zawartość całej tablicy *@fields*, moglibyśmy posłużyć się poleceniem *p @fields*, ale nie sformatowałoby ono wyników w sposób przyjazny użytkownikowi. Na szczęście, funkcja *x* debugera lepiej obsługuje bardziej złożone struktury:

```
DB<2> x @fields
0 'text/html'
1 16144
2 1107021419
3 1107031419
4 'Microsoft-IIS/6.0'
```

Ta samo dotyczy tablic asocjacyjnych, które w celu wyświetlania możemy nawet zdefiniować bezpośrednio w debugerze:

```
DB<3> %h =(donald=> 'duck',
micky =>'mouse')
DB<4> x %h
0 'donald'
1 'duck'
2 'micky'
3 'mouse'
```

Aby wyświetlać pary klucz-wartość zamiast tablicy, wystarczy przekazać odniesienie do tablicy asocjacyjnej do polecenia *x*:

```
DB<5> x \%h
0 HASH (0x837a5f8)
'donald' => 'duck'
'micky' => 'mouse'
```

Zauważmy, że numer w wierszu poleceń został zwiększony. Pojawia się on też na liście historii, dostępnej po naciśnięciu klawisza *H*:

```
DB<6> H
5: x \%h
4: x %h
3: %h=(donald=>'duck', $$
micky=>'mouse')
2: x @fields
1: p $fields[4]
```

Aby ponownie wyświetlić element *\$field[4]*, musimy tylko wpisać wykrzyknik wraz z numerem wpisu w historii *!l*. To na razie wystarczy – przejdźmy do bardziej skomplikowanych zadań.

Prawdziwy problem

Wyobraźmy sobie, że programista właśnie zakończył prace nad nowym modulem *Foo::Bar* i chce go teraz przygotować do publikacji w sieci CPAN. Pakiet zawiera plik o nazwie *Makefile.PL*, widoczny na Listingu 3, oraz plik modułu *lib/Foo/Bar.pm*, który może zawierać też dokumentację (Listing 2). Wywołanie *perl Makefile.PL* powoduje wyświetlenie dosyć mało zrozumiałego komunikatu o błędzie:

```
WARNING: Setting ABSTRACT $$
via file 'lib/Foo/Bar.pm'
failed at ExtUtils/MakeMaker.pm line 606
```

Niespecjalnie zrozumiałe, prawda? W pliku *Makefile.PL* jest używany moduł *ExtUtils::MakeMaker*, który jest czcigodnym przy-

Tabela 1: Polecenia debugera

Polecenie	Znaczenie
Kontrola przebiegu programu	
<i>n</i>	Uruchomienie następnego wiersza i zatrzymanie
<i>s</i>	Krokowe wykonanie instrukcji z następnego wiersza, zatrzymanie w podprogramie
<i>r</i>	Zakończenie bieżącej funkcji, powrót i zatrzymanie
<i>R</i>	Powrót do początku w celu ponownego uruchomienia
Wyświetlanie zmiennych	
<i>p</i>	Wyświetlenie wartości
<i>x</i>	Wyświetlenie (<i>x</i> \%tablica_asocjacyjna)
Przeglądanie źródła	
<i>l</i>	Przewijanie naprzód
<i>-</i>	Przewijanie wstecz
<i>v</i>	Wyświetlenie kodu otaczającego bieżący wiersz
<i>.</i>	Powrót do wiersza bieżącego
<i>f</i>	Zmiana na inny plik źródłowy
Nawigacja dynamiczna	
<i>c wiersz</i>	Wykonanie kodu do tego wiersza i zatrzymanie
<i>c funkcja</i>	Wykonanie kodu do tej funkcji i zatrzymanie
<i>b wiersz</i>	Ustawienie punktu wstrzymania w wierszu
<i>b funkcja</i>	Ustawienie punktu wstrzymania w funkcji
<i>b wiersz/funkcja</i>	Punkt wstrzymania z warunkiem Warunek
<i>a wiersz/funkcja czynność</i>	Punkt czynności w wierszu/funkcji
<i>w wiersz/funkcja</i>	Punkt kontrolny w zmiennej/funkcji Zmienna
<i>< Polecenie</i>	Ustawienie wstępnego zgłoszenia polecenia
<i>L</i>	Wyświetlenie punktów wstrzymania, punktów kontrolnych, czynności
<i>B/A/W</i>	Usuwanie punktów wstrzymania, punktów kontrolnych, czynności

kładem programistycznych sztuczek w Perlu, zaiste dosyć trudnych do pojęcia. Debugger pozwala nam do niego zajrzeć, jeśli wpisujemy polecenie `perl -d Makefile.PL`. Ponieważ ostrzeżenie wskazuje na wiersz 606 w pliku `ExtUtils/MakeMaker.pm` jako źródło błędu, musimy przyjrzeć się winowajcy. Polecenie `f ExtUtils/MakeMaker.pm` spowoduje, że do niego przejdziemy. Następnie musimy ustawić punkt wstrzymania w wierszu 606 (b 606), aby później debugger się tam zatrzymał. Polecenie `c` (continue – kontynuuj) nakazuje debuggerowi wykonywać program do następnego punktu wstrzymania:

```
DB<2> c
606: push $$
@{$self->{RESULT}},
$self->nicetext$$
($self->$method(%a));
```

Zamiast ustawiać punkt wstrzymania poleceniem `b 606` i wpisywać polecenie `c` w celu przejścia do tego punktu, można byłoby użyć polecenia `c 606`, aby wykonać program aż do wiersza 606. Lecz ustawienie trwałego punktu wstrzymania przyda nam się także później. Polecenie `push` widoczne w następnym wierszu kodu źródłowego dołącza wynik wywołania metody do tablicy. Warto dowiedzieć się, którą metodę wywołuje zmienna `$method`. Może nam to powiedzieć polecenie `p $method`. Wyświetla ono ciąg `post_initialize`, ale to nie daje nam szczególnie więcej informacji.

Debugger Perla wykonuje bieżący wiersz, kiedy wpisujemy polecenie `n` – ale nie reakcję. Polecenie `n` uruchamia wiersz, lecz nie pojawia się oczekiwane przez nas ostrzeżenie. Wygląda na to, że moduł `MakeMaker` wykonuje ten wiersz wiele razy i wyświetla błąd

Listing 1: Skrypt wsrw

```
01 #!/usr/bin/perl -w
02 #####
03 # wsrw – Serwer WWW pod danym
  adresem URL
  # Mike Schilli, 2004
  # (m@perlmeister.com)
05 # (m@perlmeister.com)
06 #####
07 use LWP::Simple;
08 my $url = shift
09 or die „usage $0 url”;
10 my (@fields) = head ($url)
11 or die „Fetch failed”;
12 print „Server: $fields[4]\n”;
```

przy `n`-tej iteracji. Nie ma problemu. Zanim wpisujemy polecenie `c` (continue), aby przejść do następnej iteracji (która znowu zatrzyma się w punkcie wstrzymania w wierszu 606), zdefiniujmy punkt czynności w tym wierszu:

```
DB<3> a 606print$$
("$method\n");
```

Dzięki poleceniu `a`, numerowi wiersza i kodowi w Perlu, debugger wyświetli zawartość zmiennej `$method` przy każdym wywołaniu wiersza 606, nawet podczas działania z pełną prędkością. Idźmy dalej, używając polecenia kontynuacji:

```
DB<4> c
606:push @
{$self->{RESULT}},
$self->nicetext$$
($self->$method(%a));
platform_constants
```

Ponieważ ustawiliśmy punkt wstrzymania w wierszu 606, debugger zatrzymuje się tu w kolejnej rundzie. Z powodu punktu czynności wyświetla bieżącą wartość zmiennej `$method`, metodę funkcyjną `platform_constants`. Na razie nic znajomego.

Poszukiwanie ostrzeżeń

Nie ma tu jeszcze nic podejrzanego. Usunmy kod wstrzymania w wierszu 606, wpisując polecenie `B 606`, i nakażmy kontynuację wykonywania programu przez debugger, wpisując `c`:

```
DB<4> B 606
DB<5> c
[...]
staticmake
test
ppd
WARNING: Setting ABSTRACT $$
via file 'lib/somehow$$
/anyway.pm' failed
```

Po wielu rundach okazuje się, że metoda `ppd` powoduje wyświetlenie ostrzeżenia. Niestety, ostatnia czynność poszła za daleko, ale bez obaw, wpisanie polecenia `R` pozwoli nam uruchomić program od początku. Teraz możemy ustawić nowy punkt wstrzymania dla modułu `ExtUtils/MakeMaker.pm` w wierszu 606, ale dodając tym razem warunek:

```
DB<5> f ExtUtils/MakeMaker.pm
DB<6> b 606 $method eq "ppd"
DB<7> c
```

Listing 2: Przykładowy moduł

```
01 =head1 NAME
02
03 Foo::Bar – Bla bla bla
04
05 =head1 SYNOPSIS
06
07 use Foo::Bar;
```

Debugger nie zatrzyma się w punkcie wstrzymania w wierszu 606, o ile zmienna `$method` nie będzie miała wartości `ppd`. Program uruchamia się, zatrzymuje, a debugger wyświetla znowu kod wiersza 606. Polecenie `p $method` potwierdza, że zachodzi określony przez nas warunek.

Możemy teraz skorzystać z polecenia `m`, aby odkryć, co robi odniesienie `$self` w wierszu 606:

```
DB<8> m $self
[...]
via MM -> ExtUtils::MM -> $$
ExtUtils::MM_Unix: $$
post_initialize $$
via MM-> ExtUtils::MM $$
->ExtUtils::MM_Unix: postamble
viaMM -> ExtUtils::MM->$$
ExtUtils::MM_Unix:ppd
```

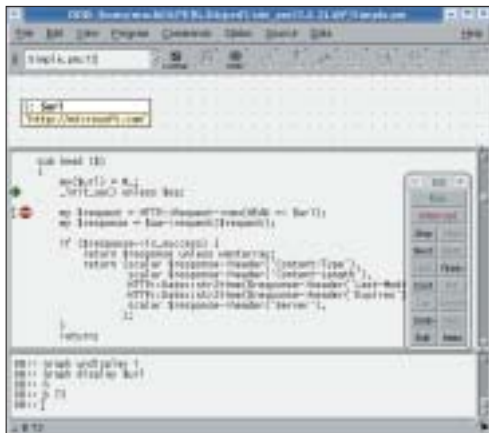
Metoda `ppd` jest oczywiście zdefiniowana w module `ExtUtils::MM_Unix`. Aby ją zanalizować, musimy kontynuować program poleceniem `c`, lecz zatrzymać się przy uruchomieniu metody `ExtUtils::MM_Unix::ppd`:

```
DB<9> c ExtUtils::MM_Unix::ppd
ExtUtils::MM_Unix::ppd$$
(ExtUtils/MM_Unix.pm:U3322):
3322: my($self) = @_;
```

Debugger jest teraz w pierwszym wierszu metody `ppd` w module `ExtUtils::MM_Unix`. Po wpisaniu polecenia `l` w celu rozejrzenia się, odkrywamy, że metoda `ppd` wywołuje metodę `parse_abstract()`:

```
DB<10> l
3322==> my($self) = @_;
3323: if ($self->$$
{ABSTRACT_FROM}){
3324: $self->{ABSTRACT}$$
= $self->parse_abstract($self->$$
{ABSTRACT_FROM})or
```

Polecenie `c parse_abstract` nakazuje debuggerowi kontynuować działanie i zatrzymać się w pierwszym wierszu metody `parse_abstract`.



Rysunek 1: Interfejs graficzny DDD zawiera debugera Perla.

```
DB<11> c parse_abstract
ExtUtils::MM_Unix::$$
parse_abstract(ExtUtils::
/MM_Unix.pm:3045):
3045: my($self,$parsefile) = @_;
```

Polecenie `l+20` wyświetla następnę 20 wierszy oraz pokazuje wyrażenie regularne, za pomocą którego moduł `Makefile.PL` pobiera streszczenie z modułu zawartego w dystrybucji: `057:nextunless/^(($package\s-\s)(.*)/`. Wpisanie polecenia `w` umożliwia nam ustawienie punktu kontrolnego dla zmiennej `$package`, które zatrzyma program po wpisaniu polecenia `c`, ilekroć zmieni się wartość zmiennej `$package`:

```
DB<2> w $package
DB<3> c
```

Listing 3: Makefile.PL

```
01 #####
02 # Makefile.PL dla Foo::Bar
03 #####
04 use ExtUtils::MakeMaker;
05
06 WriteMakefile(
07   'NAME' => 'Foo::Bar',
08   'VERSION_FROM' =>
09     'lib/Foo/Bar.pm',
10   'PREREQ_PM' => {},
11   ( $] >= 5.005
12     ? (
13       ABSTRACT_FROM =>
14         'lib/Foo/Bar.pm',
15       AUTHOR =>
16         'Ed Jones <ed@jones.com>'
17     )
18   : ()
19 ),
20 );
```

```
Watchpoint 0: packagechanged:
old value: ''
newvalue: 'Foo-Bar'
```

Wiemy teraz, co się dzieje. Metoda `parse_abstract()` poszukuje wyrażenia regularnego `/^Foo::Bar\s-\s)(.*)/`. Nazwa modułu musi znajdować się na początku wiersza, aby streszczenie zostało znalezione po jednej spacji i pauzie. Ponieważ wiersz 3 na Listingu 2 nie zawiera nazwy modułu na początku wiersza, lecz jest ona wcięta, pobranie streszczenia nie udaje się.

Graficzny interfejs użytkownika

Jeśli wolimy interfejs graficzny w stylu „wskaz i kliknij”, możemy połączyć program `Data Display Debugger (ddd)` z perlowym zapleczem. Program `ddd` jest dołączany do wszystkich ważniejszych dystrybucji Linuksa. Następujące polecenie powoduje wywołanie skryptu `wsrv` w programie `ddd` w celu zbadania strony głównej firmy Microsoft:

```
ddd -perl wsrv $$
http://microsoft.com.
```

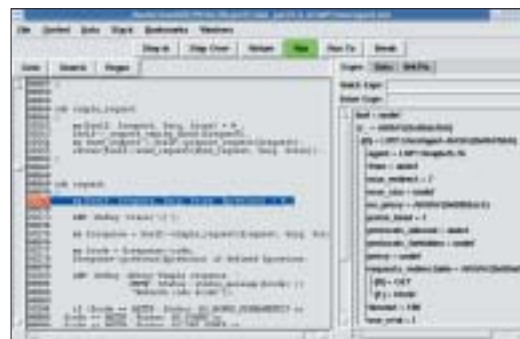
Na Rysunku 1 widać działanie interfejsu graficznego. Ułatwia on ustawianie punktów wstrzymania czy wyrażeń kontrolnych (takich jak skalar `$url` na Rysunku 1). Podobny efekt można uzyskać używając polecenia `<` w deburgerze Perla, lecz niektórzy wolą ładnie sformatowane ekrany graficzne. Zamiast nich można używać komercyjnego środowiska graficznego IDE `Komodo`, a także darmowego `Ptkdb`, który jest dostępny z sieci CPAN:

```
perl -MCPAN -e $$
'install(Tk,Devel::ptkdb)'
perl -d:ptkdb wsrv $$
http://microsoft.com
```

Rysunek 2 przedstawia iterację `Ptkdb` po metodzie `request` w pakiecie `LWP::UserAgent`. Kolumna z prawej strony przedstawia atrybuty obiektu `LWP::UserAgent`, który został przekazany do metody.

Śledzenie

Na koniec mała sztuczka służąca do wyświetlania każdego wierszu kodu wykonywanego przez skrypt. Zmienna środowiskowa `PERLDB_OPTS` steruje śledzeniem debugera. Wystarczy dodać ją do wiersza polecenia przed wywołaniem debugera:



Rysunek 2: Graficzny debugger Ptkdb jest oparty na interpreterze Perl/Tk i łatwo go zainstalować z sieci CPAN.

```
PERLDB_OPTS="NonStop=1$$
AutoTrace=1 frame=2 "$$
perl -d Sprogram
```

Opcja `AutoTrace` przełącza debugera w tryb śledzenia, w którym wyświetla on każdy wiersz kodu źródłowego przed jego wykonaniem. Opcja `NonStop` nakazuje debugerowi nie zatrzymywać się w celu pobrania danych od użytkownika na początku lub końcu. Parametr `frame=2` dodaje komunikaty wejścia i wyjścia w momencie wchodzenia do podprogramów i wychodzenia z nich. Jeśli potrzebujemy także informacji o przekazywanych parametrach i wartościach zwracanych z podprogramów, należy określić parametr `frame=4`. Wreszcie opcja `Perl -S` powoduje poszukiwanie skryptu w całej zmiennej `$PATH`, a nie tylko w katalogu bieżącym.

Każda kolejna dystrybucja Perla zawiera krótkie wprowadzenie do sztuki debugowania. Polecenie `perldoc perldebtut` wyświetla stronę podręcznika. Strona `perldebug` zawiera dokładniejsze informacje, a jeśli naprawdę chcemy dostać się do „wnętrza” debugera, zajrzyjmy do `perldebguts`. Poza tym warto kupić sobie książkę [2], aby mieć najlepszy leksykon dotyczący debugera Perla. ■

INFO

- [1] Listingi do tego artykułu: <http://www.linux-magazin.com/Magazine/Downloads/54/Perl>
- [2] Richard Foley, „Perl Debugger Pocket Reference”: O'Reilly 2004
- [3] Michael Schilli, „Retire your Debugger, log smartly with Log::Log4perl”: <http://www.perl.com/pub/a/2002/09/11/log4perl.html>
- [4] Peter Scott and Ed Wright: „Perl Debugged”, Addison-Wesley 2001