Perl. Najlepsze rozwiązania

Autor: Damian Conway
Tłumaczenie: Grzegorz Werner
ISBN: 83-246-0127-9
Tytuł oryginału: Perl Best Practices
Format: B5, stron: 488

Zastosuj w pracy sprawdzone style i standardy kodowania

- Wypracuj czytelne konwencje nazewnicze
- Stwórz odpowiednią dokumentację kodu
- Przetestuj kod i usuń błędy

Indywidualne style kodowania, obejmujące m.in. nazywanie zmiennych, wpisywanie komentarzy i stosowanie określonych konstrukcji językowych, pomagają w rozbudowywaniu programów i usuwaniu z nich błędów. Jednak taka metoda pracy, charakterystyczna dla doświadczonych programistów, nie zawsze jest najlepsza. Dodatkowo własny styl staje się ogromną przeszkodą w przypadku pracy zespołowej – tu powinno się raczej stosować spójne standardy, dzięki którym kod będzie klarowny, niezawodny, wydajny, łatwy w konserwacji i zwięzły.

W książce „Perl. Najlepsze rozwiązania" znajdziesz ponad 250 porad programisty z 22-letnią praktyką, dotyczących pisania kodu źródłowego w Perlu. Wskazówki te obejmują układ kodu, konwencje nazewnicze, dobór struktur danych i konstrukcji sterujących, dekompozycję programu, projekt i implementację interfejsu, modularność, obiektowość, obsługę błędów, testowanie i debugowanie. Autor książki nie stara się udowodnić, że preferowane przez niego rozwiązania są jedyne i najlepsze – przedstawia jedynie sprawdzone techniki, używane przez programistów z całego świata.

- Formatowanie kodu źródłowego
- Metody określania nazw zmiennych i obiektów
- Korzystanie ze struktur sterujących
- Przygotowywanie dokumentacji
- Implementacja operacji wejścia i wyjścia
- Stosowanie wyrażeń regularnych
- Obsługa wyjątków
- Podział kodu na moduły
- Wykrywanie i usuwanie błędów z kodu

Wykorzystaj znajdujące się w tej książce wskazówki – stwórz najlepszy kod

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

Spis treści

ROZDZIAŁ 2.

Układ kodu

Większość programów należałoby wciąć
o sześć stóp w dół... i przysypać ziemią.

— Blair P. Houghton

Formatowanie. Wcięcia. Styl. Układ kodu. Bez względu na nazwę, jest to jedna z najbardziej kontrowersyjnych dziedzin dyscypliny programistycznej. O układ kodu stoczono więcej (i bardziej krwawych) wojen niż o jakikolwiek inny aspekt kodowania.

Jaka jest zatem zalecana praktyka? Czy należy posługiwać się klasycznym stylem Kernighana i Ritchie'ego (K&R)? A może zdecydować się na formatowanie BSD? Wybrać układ zalecany przez projekt GNU? A może wytyczne kodowania Slashcode?

Oczywiście, że nie! Każdy wie, że *[tutaj wstawić swój osobisty styl kodowania]* jest Jedynym Słusznym Stylem, jedynym rozsądnym wyborem, uświęconym przez *[tutaj wstawić nazwę ulubionego Bóstwa Programistycznego]* od Niepamiętnych Czasów! Każde inne rozwiązanie jest absurdalne i heretyckie, a zatem ewidentnie jest Dziełem Ciemności!!!

I właśnie na tym polega problem. Kiedy przychodzi wybrać układ kodu, trudno zdecydować, gdzie kończą się racjonalne uzasadnienia, a zaczynają zracjonalizowane nawyki.

Przyjęcie spójnych metod formatowania kodu i stosowanie ich we wszystkich programach ma fundamentalne znaczenie dla realizacji zalecanych praktyk programistycznych. Dobry układ zwiększa czytelność kodu, pomaga wykrywać błędy i sprawia, że struktura programu jest bardziej zrozumiała. Układ kodu jest ważny.

Korzyści te zapewnia jednak większość stylów kodowania — w tym cztery wspomniane wcześniej. Choć więc układ kodu jest bardzo ważny, to **konkretny** układ... jest zupełnie nieistotny!

Trzeba tylko przyjąć pojedynczy, spójny styl, który odpowiada całemu zespołowi, a następnie stosować go konsekwentnie we wszystkich programach.

Zamieszczone niżej wskazówki dotyczące układu kodu zostały starannie i świadomie wybrane spośród wielu możliwości, aby skonstruować styl, który jest spójny i zwięzły, zwiększa czytelność kodu, ułatwia wykrywanie pomyłek oraz może być z powodzeniem stosowany przez różnych programistów pracujących w wielu odmiennych środowiskach.

Nie wątpię, że niektóre z tych wskazówek wywołają sprzeciw. Prawdopodobnie gwałtowny. Każdy czytelnik musi zastanowić się, czy argumenty za odrzuceniem danej wskazówki przeważają nad argumentami za jej przyjęciem. Jeśli tak, nieprzestrzeganie tej reguły nie będzie miało znaczenia.

Stosowanie nawiasów

Stosuj nawiasy klamrowe i okrągłe w stylu K&R.

Podczas tworzenia bloków kodu należy stosować nawiasy w stylu K&R[1], tzn. umieszczać otwierający nawias klamrowy na końcu konstrukcji, która steruje blokiem. Zawartość bloku należy rozpocząć od następnego wiersza, wcinając ją o jeden poziom. Wreszcie zamykający nawias klamrowy należy umieścić w oddzielnym wierszu, na tym samym poziomie wcięcia, co konstrukcja sterująca.

Podobnie podczas pisania w nawiasie listy, która rozciąga się na wiele wierszy, należy umieścić otwierający nawias okrągły na końcu wyrażenia sterującego; elementy listy umieścić w kolejnych wierszach, wcięte o jeden poziom; zamykający nawias okrągły umieścić w oddzielnym wierszu, zmniejszając wcięcie do poziomu instrukcji wyrażenia sterującego, na przykład:

```
my @names = (
    'Damian',       # Klucz podstawowy
    'Matthew',      # Ujednoznacznienie
    'Conway',       # Ogólna klasa lub kategoria
);

for my $name (@names) {
    for my $word ( anagrams_of(lc $name) ) {
        print "$word\n";
    }
}
```

Nie należy umieszczać otwierającego nawiasu klamrowego lub okrągłego w oddzielnym wierszu, jak w stylach BSD lub GNU:

```
# Nie używać stylu BSD...
my @names =
(
    'Damian',       # Klucz podstawowy
    'Matthew',      # Ujednoznacznienie
    'Conway',       # Ogólna klasa lub kategoria
);

for my $name (@names)
{
    for my $word (anagrams_of(lc $name))
    {
        print "$word\n";
    }
}

# ani stylu GNU...
```

[1] „K&R" to Brian Kernighan i Dennis Ritchie, autorzy książki *Język C* (wyd. WNT, 1988).

```
for my $name (@names)
  {
    for my $word (anagrams_of(lc $name))
      {
        print "$word\n";
      }
  }
```

Styl K&R ma jedną oczywistą przewagę nad pozostałymi dwoma: zajmuje jeden wiersz mniej na każdy blok, co oznacza, że na ekranie widać więcej rzeczywistego kodu. Jeśli wyświetlana jest seria bloków, może to oznaczać trzy lub cztery dodatkowe wiersze na każdy ekran.

Głównym kontrargumentem na korzyść stylów BSD i GNU ma być to, że nawias otwierający[2] w oddzielnym wierszu ułatwia wizualne dopasowanie początku i końca bloku lub listy. Twierdzenie to ignoruje jednak fakt, że równie łatwo jest dopasować je w stylu K&R. Wystarczy przewijać program do góry aż do napotkania konstrukcji sterującej, a następnie przeskoczyć na koniec wiersza.

Jeszcze łatwiej nacisnąć klawisz edytora, który przenosi kursor między dopasowanymi nawiasami. W edytorze *vi* jest to klawisz %. W edytorze Emacs nie ma takiego polecenia, ale łatwo je utworzyć poprzez dopisanie do pliku *.emacs* następujących wierszy[3]:

```
;; Klawisz % służy do dopasowywania różnych rodzajów nawiasów...
(global-set-key "%" 'match-paren)
(defun match-paren (arg)
  "Przechodzi do dopasowanego nawiasu, gdy kursor jest na nawiasie, w przeciwnym razie
wstawia znak %."
  (interactive "p")
  (cond ((string-match "[[{(<]" next-char) (forward-sexp 1))
      ((string-match "[\]})>]" prev-char) (backward-sexp 1))
      (t (self-insert-command (or arg 1)))))
```

Co ważniejsze, znajdowanie pasującego nawiasu rzadko jest celem samym w sobie. Zwykle jesteśmy zainteresowani nawiasem zamykającym dlatego, że chcemy ustalić, gdzie kończy się bieżąca konstrukcja (pętla `for`, instrukcja `if` lub procedura), albo dowiedzieć się, jaką konstrukcję kończy nawias zamykający. Oba te zadania są nieco **łatwiejsze** w przypadku stylu K&R. Aby znaleźć koniec konstrukcji, wystarczy spojrzeć prosto w dół, począwszy od słowa kluczowego; żeby znaleźć konstrukcję zakończoną nawiasem, wystarczy patrzeć w górę aż do napotkania słowa kluczowego.

Innymi słowy, style BSD i GNU ułatwiają dopasowanie **składni** nawiasów, a styl K&R — dopasowanie ich **semantyki**. To powiedziawszy, spieszę zapewnić, że w stylach BSD i GNU nie ma niczego złego. Jeśli czytelnik i programiści z jego zespołu uznają, że wyrównane pionowo nawiasy ułatwiają im czytanie kodu, mogą ich używać. Liczy się tylko to, aby wszyscy członkowie zespołu uzgodnili wspólny styl i konsekwentnie go stosowali.

---

[2] Dalej w tej książce słowo „nawias" będzie używane jako ogólny termin na oznaczenie czterech typów ograniczników występujących w parach: nawiasów klamrowych (`{...}`), okrągłych (`(...)`), kwadratowych (`[...]`) i trójkątnych (`<...>`).

[3] Sugerowane konfiguracje edytorów zostały zebrane w dodatku C. Można je również pobrać pod adresem *http://www.oreilly.com/catalog/perlbp*.

Słowa kluczowe

Oddzielaj słowa kluczowe struktur sterujących od nawiasu otwierającego.

Struktury sterujące regulują działanie programu, więc ich słowa kluczowe zaliczają się do najbardziej krytycznych komponentów kodu. Dlatego istotne jest, aby słowa te dobrze wyróżniały się w kodzie źródłowym.

W Perlu po większości słów kluczowych struktur sterujących natychmiast następuje nawias otwierający, co sprawia, że łatwo pomylić je z wywołaniami procedur. Warto zatem jakoś je wyróżnić. W tym celu należy wstawiać pojedynczą spację między słowem kluczowym a następującym po nim nawiasem klamrowym lub okrągłym:

```
for my $result (@results) {
    print_sep();
    print $result;
}

while ($min < $max) {
    my $try = ($max - $min) / 2;
    if ($value[$try] < $target) {
        $max = $try;
    }
    else {
        $min = $try;
    }
}
```

Bez tego odstępu trudniej zauważyć słowo kluczowe i łatwiej pomylić je z wywołaniem procedury:

```
for(@results) {
    print_sep();
    print;
}

while($min < $max) {
    my $try = ($max - $min) / 2;
    if($value[$try] < $target) {
        $max = $try;
    }
    else{
        $min = $try;
    }
}
```

Procedury i zmienne

Nie oddzielaj nazw procedur i zmiennych od następującego po nich nawiasu otwierającego.

Aby poprzednia reguła się sprawdziła, **nie należy** umieszczać spacji między nazwami procedur i zmiennych a następującymi po nich nawiasami. W przeciwnym razie łatwo będzie pomylić wywołanie procedury ze strukturą sterującą albo uznać początkową część elementu tablicy za niezależną zmienną skalarną.

Należy zatem dosuwać nazwy procedur i zmiennych do następujących po nich nawiasów okrągłych lub klamrowych:

```
my @candidates = get_candidates($marker);

CANDIDATE:
for my $i (0..$#candidates) {
    next CANDIDATE if open_region($i);

    $candidates[$i]
        = $incumbent{ $candidates[$i]{region} };
}
```

Użycie spacji niepotrzebnie utrudnia ich rozpoznanie:

```
my @candidates = get_candidates ($marker);

CANDIDATE:
for my $i (0..$#candidates) {
    next CANDIDATE if open_region ($i);

    $candidates [$i]
        = $incumbent {$candidates [$i] {region}};
}
```

Funkcje wbudowane

Nie używaj niepotrzebnych nawiasów podczas wywołania funkcji wbudowanych i „honorowo" wbudowanych.

Wbudowane funkcje Perla są faktycznie słowami kluczowymi języka, więc można je wywoływać bez nawiasów okrągłych, chyba że konieczne jest wymuszenie priorytetu.

Wywoływanie funkcji wbudowanych bez nawiasów zmniejsza zagęszczenie kodu i zwiększa jego czytelność. Brak nawiasów pomaga też odróżnić wywołania procedur od wywołań funkcji wbudowanych:

```
while (my $record = <$results_file>) {
    chomp $record;
    my ($name, $votes) = split "\t", $record;
    print 'Głosy na ',
          substr($name, 0, 10),        # Nawiasy niezbędne ze względu na priorytet
          ": $votes (verified)\n";
}
```

Niektóre importowane procedury, zwykle zawarte w modułach podstawowej dystrybucji, zaliczają się do „honorowych" funkcji wbudowanych i również mogą być wywoływane bez nawiasów. Zazwyczaj dotyczy to procedur oferujących funkcje, które powinny być w samym języku, ale nie są. Przykładem mogą być procedury `carp` i `croak` (ze standardowego modułu `Carp` — rozdział 13.), `first` i `max` (ze standardowego modułu `List::Util` — rozdział 8.) oraz `prompt` (z modułu CPAN `IO::Prompt` — rozdział 10.).

Jednakże w przypadkach, gdy konieczne jest użycie funkcji wbudowanych z nawiasami, należy zastosować regułę dotyczącą procedur, a nie słów kluczowych, tzn. nie umieszczać spacji między nazwą funkcji wbudowanej a otwierającym nawiasem okrągłym:

```
while (my $record = <$results_file>) {
    chomp( $record );
    my ($name, $votes) = split("\t", $record);
    print(
        'Głosy na ',
        substr($name, 0, 10),
        ": $votes (verified)\n"
    );
}
```

Nie należy traktować funkcji wbudowanych jak słów kluczowych (poprzez dopisanie spacji):

```
while (my $record = <$results_file>) {
    chomp ($record);
    my ($name, $votes) = split ("\t", $record);
    print (
        'Głosy na ',
        substr ($name, 0, 10),
        ": $votes (verified)\n"
    );
}
```

Klucze i indeksy

Oddzielaj skomplikowane klucze lub indeksy od otaczających je nawiasów.

Podczas uzyskiwania dostępu do elementów zagnieżdżonych struktur danych (tablic asocjacyjnych, które przechowują tablice asocjacyjne, które przechowują tablice, które przechowują jakieś elementy) łatwo o długie, skomplikowane i zagęszczone wyrażenia w rodzaju:

```
$candidates[$i] = $incumbent{$candidates[$i]{get_region()}};
```

Dotyczy to szczególnie sytuacji, w której indeksy same są zmiennymi indeksowanymi. Stłoczenie wszystkich elementów bez użycia odstępów nie poprawia czytelności takich wyrażeń. Szczególnie trudno czasem domyślić się, czy dana para nawiasów jest częścią indeksu wewnętrznego, czy zewnętrznego.

Jeśli indeks nie jest prostą stałą albo zmienną skalarną, lepiej umieścić spacje między wyrażeniem indeksującym a otaczającymi je nawiasami:

```
$candidates[$i] = $incumbent{ $candidates[$i]{ get_region() } };
```

Decydującymi czynnikami są tu złożoność i ogólna długość indeksu. Od czasu do czasu „rozrzedzenie" indeksu ma sens nawet wtedy, gdy **jest** on pojedynczą stałą lub skalarem. Jeśli na przykład indeks jest bardzo długi, lepiej zapisać go w taki sposób:

```
print $incumbent{ $largest_gerrymandered_constituency };
```

niż tak:

```
print $incumbent{$largest_gerrymandered_constituency};
```

Operatory

Używaj odstępów między operatorami binarnymi a ich argumentami.

Długie wyrażenia często bywają niezrozumiałe, więc nie należy jeszcze bardziej utrudniać ich interpretacji poprzez stłaczanie komponentów:

```
my $displacement=$initial_velocity*$time+0.5*$acceleration*$time**2;
my $price=$coupon_paid*$exp_rate+(($face_val+$coupon_val)*$exp_rate**2);
```

Warto dać operatorom binarnym nieco „oddechu", nawet jeśli będzie to wymagać przeniesienia kodu do następnego wiersza:

```
my $displacement
    = $initial_velocity * $time + 0.5 * $acceleration * $time**2;
my $price
    = $coupon_paid * $exp_rate + ($face_val + $coupon_val) * $exp_rate**2;
```

Ilość odstępów należy dobrać zgodnie z priorytetem operatorów, aby osoba czytająca wyrażenie mogła łatwo wyróżnić naturalne grupy elementów. Można na przykład dopisać dodatkowe spacje po obu stronach operatora + (o niższym priorytecie), aby wizualnie podkreślić wyższy priorytet dwóch wyrażeń multiplikatywnych. Z drugiej strony, można śmiało ścisnąć operator ** i jego dwa argumenty, ponieważ ma on bardzo wysoki priorytet i dłuższy, łatwo zauważalny symbol.

Jedna spacja zawsze wystarcza, kiedy do podkreślenia (albo zmiany) priorytetu używane są nawiasy:

```
my $velocity
    = $initial_velocity + ($acceleration * ($time + $delta_time));
my $future_price
    = $current_price * exp($rate - $dividend_rate_on_index) * ($delivery - $now);
```

Symboliczne operatory unarne powinny być zawsze dosunięte do argumentów:

```
my $spring_force = !$hyperextended ? -$spring_constant * $extension : 0;
my $payoff = max(0, -$asset_price_at_maturity + $strike_price);
```

Nazwane operatory binarne należy traktować tak jak funkcje wbudowane i odpowiednio oddzielać od argumentów:

```
my $tan_theta = sin $theta / cos $theta;
my $forward_differential_1_year = $delivery_price * exp -$interest_rate;
```

Średniki

Umieszczaj średnik za każdą instrukcją.

W Perlu średniki są separatorami, a nie terminatorami instrukcji, więc nie trzeba umieszczać średnika po ostatniej instrukcji w bloku. Pomimo to należy je dopisywać, nawet jeśli blok zawiera tylko jedną instrukcję:

```
while (my $line = <>) {
    chomp $line;

    if ( $line =~ s{\A (\s*) -- (.*)}{$1#$2}xms ) {
        push @comments, $2;
    }

    print $line;
}
```

Nie wymaga to wielkiego wysiłku, a końcowy średnik zapewnia dwie ważne korzyści: sygnalizuje osobie czytającej kod, że poprzednia instrukcja jest zakończona, a ponadto sygnalizuje to **kompilatorowi**. Jest to znacznie ważniejsze, ponieważ człowiek często może domyślić się, co programista miał na myśli, a kompilator odczytuje tylko to, co zostało rzeczywiście napisane.

Pominięcie końcowego średnika zwykle nie powoduje problemów podczas pisania kodu (tzn. wtedy, gdy programista przykłada uwagę do całego fragmentu kodu):

```
while (my $line = <>) {
    chomp $line;

    if ( $line =~ s{\A (\s*) -- (.*)}{$1#$2}xms ) {
        push @comments, $2
    }

    print $line
}
```

Poza średnikiem nie ma jednak niczego, co zapobiegałoby subtelnym problemom, kiedy programista później dopisze kolejne instrukcje:

```
while (my $line = <>) {
    chomp $line;

    if ( $line =~ s{\A (\s*) -- (.*)}{$1#$2}xms ) {
        push @comments, $2
        /shift/mix
    }

    print $line
    $src_len += length;
}
```

Problem w tym, że dopisany kod nie dodaje nowych instrukcji, lecz jest absorbowany przez poprzednie. Zatem powyższy fragment kodu w istocie znaczy:

```
while (my $line = <>) {
    chomp $line;

    if ( $line =~ s{\A (\s*) -- (.*)}{$1#$2}xms ) {
```

```
            push @comments, $2 / shift() / mix()
        }

        print $line ($src_len += length);
    }
```

Jest to bardzo częsta i zrozumiała pomyłka. Podczas rozbudowywania kodu z natury rzeczy skupiamy się na nowych instrukcjach, zakładając, że istniejące będą nadal działały prawidłowo. Jednakże brak końcowego średnika może sprawić, że istniejące instrukcje zostaną wchłonięte przez nową.

Reguła nie dotyczy bloków `map` lub `grep`, które zawierają tylko jedną instrukcję. W takim przypadku lepiej pominąć terminator:

```
my @sqrt_results
    = map { sqrt $_ } @results;
```

ponieważ średnik w bloku utrudnia dostrzeżenie końca pełnej instrukcji:

```
my @sqrt_results
    = map { sqrt $_; } @results;
```

Wyjątek ten nie zwiększa podatności programu na błędy, ponieważ umieszczanie kilku instrukcji w bloku `map` lub `grep` jest dość nietypowe i często świadczy o tym, że należałoby zastosować inną konstrukcję (podrozdział „Złożone odwzorowania” w rozdziale 6.).

Przecinki

Umieszczaj przecinek po każdej wartości na wielowierszowej liście.

Podobnie jak średniki pełnią funkcję separatora w blokach instrukcji, tak przecinki oddzielają wartości na liście. Oznacza to, że również je należy traktować jak terminatory.

Ponadto dopisanie końcowego przecinka (co wolno zrobić na dowolnej liście Perla) bardzo ułatwia zmianę kolejności elementów, na przykład dużo łatwiej przekształcić poniższą listę:

```
my @dwarves = (
    'Smutuś',
    'Śpioch',
    'Fajtłapek',
    'Apsik',
    'Gderek',
    'Nieśmiałek',
    'Mędrek',
);
```

na:

```
my @dwarves = (
    'Nieśmiałek',
    'Mędrek',
    'Fajtłapek',
    'Gderek',
    'Smutuś',
    'Śpioch',
    'Apsik',
);
```

Można ręcznie wycinać i wklejać wiersze, a nawet przetworzyć zawartość listy poleceniem `sort`.

Bez końcowego przecinka za elementem `'Mędrek'` zmiana kolejności listy spowodowałaby błąd:

```
my @dwarves = (
    'Nieśmiałek',
    'Mędrek'
    'Fajtłapek',
    'Gderek',
    'Smutuś',
    'Śpioch',
    'Apsik',
);
```

Oczywiście, tego rodzaju pomyłkę łatwo znaleźć i poprawić, ale czemu nie kodować w sposób, który uniemożliwia wystąpienie takiego problemu?

Długość wierszy

Używaj wierszy liczących 78 kolumn.

W świecie 30-calowych ekranów o wysokiej rozdzielczości, wygładzanych czcionek i laserowej korekcji wzroku można programować w oknie terminala szerokim na 300 kolumn.

Można, ale nie należy.

Zważywszy na ograniczenia drukowanych dokumentów, tradycyjnych ekranów VGA i oprogramowania prezentacyjnego, nierozsądnie jest formatować kod na szerokość większą niż 80 kolumn. Nawet 80-kolumnowe wiersze nie zawsze są bezpieczne ze względu na mechanizmy zawijania tekstu w niektórych terminalach, edytorach i systemach pocztowych.

Ustawienie prawego marginesu w 78. kolumnie maksymalizuje użyteczną szerokość każdego wiersza kodu, a jednocześnie gwarantuje, że wiersze będą wyświetlane w taki sam sposób na większości ekranów.

Aby ustawić prawy margines w edytorze *vi*, należy do pliku konfiguracyjnego dołączyć poniższy wiersz:

```
set textwidth=78
```

W przypadku Emacsa należy użyć poleceń:

```
(setq fill-column 78)
(setq auto-fill-mode t)
```

Kolejną zaletą takiej szerokości wiersza jest to, że każdy fragment kodu przesłany w wiadomości e-mail można przytoczyć przynajmniej raz bez zawijania wierszy:

```
From: boss@headquarters
To: you@saltmines
Subject: Proszę o wyjaśnienia

Właśnie znalazłem ten fragment kodu w Pana najnowszym module.
Czy to ma być żart!?

> $;=$/;seek+DATA,undef$/,!$s;$_=<DATA>;$s&&print||(*{q;::\;
> ;}=sub{$d=$d-1?$d:$0;s;';\t#$d#;,$_})&&$g&&do{$y=($x||=20)*($y||8);sub
```

```
> i{sleep&f}sub'p{print$;x$=,join$;,$b=~/.{$x}/g,$;}sub'f{pop||1}sub'n{substr($b
> ,&f%$y,3)=~tr,O,O,}sub'g{@_[@_]=@_;--($f=&f);$m=substr($b,&f,1);($w,$w,$m,O)
> [n($f-$x)+n($x+$f)-(${m}eq+O=>)+n$f]||$w}$w="\4O";$b=join'',@ARGV?<>:$_,$w
> x$y;$b=~s).)$&=~/\w/?O:$w)gse;substr($b,$y)=q++;$g='$i=O;$i?$b:$c=$b;
> substr+$c,$i,1,g$i;$g=~s?\d+?($&+1)%$y?e;$i-$y+1?eval$g:do{$b=$c;p;i}';
> sub'e{eval$g;&e};e}||eval||die+No.$;

Proszę natychmiast przyjść do mojego gabinetu!
Y.B.
```

Wcięcia

Używaj czterokolumnowych poziomów wcięcia.

Głębokość wcięć to sprawa dużo bardziej kontrowersyjna niż szerokość wiersza. Jeśli zapytamy czterech programistów, ile kolumn powinien liczyć każdy poziom wcięcia, otrzymamy cztery różne odpowiedzi: dwie, trzy, cztery albo osiem. Wywołamy też zażartą kłótnię.

Starożytni mistrzowie kodowania, którzy zaczynali od teletekstów i terminali sprzętowych ze stałymi przystankami tabulatora, będą nas zapewniać, że do przyjęcia są wyłącznie 8-kolumnowe wcięcia, ponieważ większość drukarek i terminali programowych nadal domyślnie wypisuje 8 kolumn na każdy znak tabulacji. Gwarantuje to jednolity wygląd kodu na każdym urządzeniu:

```
while (my $line = <>) {
        chomp $line;
        if ( $line =~ s{\A (\s*) -- ([^\n]*) }{$1#$2}xms ) {
                push @comments, $2;
        }
        print $line;
}
```

Tak (zgodzi się wielu młodszych hakerów), 8-kolumnowe wcięcia gwarantują, że kod będzie wyglądał **równie paskudnie i nieczytelnie** na każdym urządzeniu! Każdy poziom wcięcia powinien zatem liczyć nie więcej niż 2 lub 3 kolumny. Mniejsze wcięcia pozwalają zmieścić na ekranie więcej poziomów zagnieżdżenia: mniej więcej 12 poziomów przy wcięciu 2- lub 3-kolumnowym, a zaledwie 4 lub 5 poziomów przy wcięciu 8-kolumnowym. Płytsze wcięcia zmniejszają też poziomą odległość, którą musi pokonać oko. Cała lewa krawędź kodu pozostaje w zasięgu wzroku, dzięki czemu łatwiej ocenić kontekst każdego wiersza:

```
while (my $line = <>) {
  chomp $line;
  if ( $line =~ s{\A (\s*) -- ([^\n]*) }{$1#$2}xms ) {
    push @comments, $2;
  }
  print $line;
}
```

Niestety (zapłaczą starzy mistrzowie), taka metoda utrudnia dostrzeżenie wcięć programistom po trzydziestce i każdemu, kto nie ma sokolego wzroku. I w tym leży sedno problemu. Głębokie wcięcia podkreślają czytelność strukturalną kosztem kontekstowej, a płytkie — odwrotnie. Nie ma idealnego rozwiązania.

Rozsądnym kompromisem[4] jest użycie czterech kolumn na każdy poziom wcięcia. Dzięki temu starzy mistrzowie będą mogli dostrzec wcięcia, a młodzi hakerzy zagnieżdżać kod na osiem lub dziewięć poziomów[5] bez zawijania wierszy:

```
while (my $line = <>) {
    chomp $line;
    if ( $line =~ s{\A (\s*) -- (.*)}{$1#$2}xms ) {
        push @comments, $2;
    }
    print $line;
}
```

Znaki tabulacji

Wcinaj kod z wykorzystaniem spacji, a nie znaków tabulacji.

Znaki tabulacji nie nadają się do wcinania kodu, nawet jeśli przystanki tabulatora w edytorze są ustawione na cztery kolumny. Znaki tabulacji wyglądają inaczej, kiedy są drukowane na różnych urządzeniach wyjściowych, wklejane do procesora tekstu albo wyświetlane w edytorze z innymi przystankami tabulacji. Nie należy więc używać znaków tabulacji albo (co gorsza) mieszać ich ze spacjami:

```
sub addarray_internal {
»   my ($var_name, $need_quotemeta) = @_;

»   $raw .= $var_name;

»   my $quotemeta = $need_quotemeta ? q{ map {quotemeta $_} }
»   »   »   »   » :                   $EMPTY_STR
»   ··············;

····my $perl5pat
····»   = qq{(??{join q{|}, $quotemeta \@{$var_name}})};

»   push @perl5pats, $perl5pat;

»   return;
}
```

Jedynym niezawodnym, powtarzalnym i przenośnym sposobem wcinania kodu w taki sposób, aby wyglądał jednakowo w każdym środowisku, jest użycie spacji. Zgodnie z poprzednią regułą dotyczącą głębokości wcinania oznacza to cztery znaki spacji na każdy poziom wcięcia:

```
sub addarray_internal {
····my ($var_name, $need_quotemeta) = @_;

····$raw .= $var_name;

····my $quotemeta = $need_quotemeta ? q{ map {quotemeta $_} }
···················:·····················$EMPTY_STR
···················;
```

[4] Według wyników badań opublikowanych w artykule „Programming Indentation and Comprehensibility" (*Communications of ACM*, Vol. 26. No. 11, s. 861 – 867).

[5] Nie należy jednak tego robić! Jeśli potrzeba więcej niż czterech lub pięciu poziomów zagnieżdżenia, kod niemal na pewno powinien zostać przeniesiony do procedury lub modułu (rozdziały 9. i 17.).

```
····my $perl5pat
········= qq{(??{join q{|}, $quotemeta \@{$var_name}})};

····push @perl5pats, $perl5pat;

····return;
}
```

Zauważmy, że powyższa reguła nie zabrania używać klawisza *Tab* do wcinania kodu; wymaga tylko, aby wynikiem naciśnięcia tego klawisza było coś innego niż znak tabulacji. Łatwo to osiągnąć w nowoczesnych edytorach, które można skonfigurować tak, aby przekształcały znaki tabulacji w spacje, na przykład użytkownicy edytora *vim* mogą umieścić poniższe dyrektywy w swoim pliku *.vimrc*:

```
set tabstop=4        "Poziom wcięcia co cztery kolumny"
set expandtab        "Przekształcanie wszystkich wpisanych znaków tabulacji w spacje"
set shiftwidth=4     "Wcinanie i usuwanie wcięć o cztery kolumny"
set shiftround       "Wcinanie i usuwanie wcięć do najbliższego przystanku tabulatora"
```

W pliku inicjalizacyjnym *.emacs* należy natomiast napisać (tryb „cperl”):

```
(defalias 'perl-mode 'cperl-mode)

;; czterokolumnowe wcięcia w trybie cperl
'(cperl-close-paren-offset -4)
'(cperl-continued-statement-offset 4)
'(cperl-indent-level 4)
'(cperl-indent-parens-as-block t)
'(cperl-tab-always-indent t)
```

Byłoby najlepiej, gdyby kod nie zawierał ani jednego znaku tabulacji. W układzie kodu należy przekształcać je w spacje, natomiast w literalnych łańcuchach trzeba posługiwać się symbolem `\t` (rozdział 4.).

Bloki

Nigdy nie umieszczaj dwóch instrukcji w tym samym wierszu.

Jeśli w jednym wierszu znajdują się dwie lub więcej instrukcji, każda z nich staje się mniej zrozumiała:

```
RECORD:
while (my $record = <$inventory_file>) {
    chomp $record; next RECORD if $record eq $EMPTY_STR;
    my @fields = split $FIELD_SEPARATOR, $record; update_sales(\@fields);$count++;
}
```

Oszczędność miejsca na ekranie zapewniają już nawiasy w stylu K&R; warto wykorzystać to miejsce do zwiększenia czytelności kodu przez umieszczenie każdej instrukcji w oddzielnym wierszu:

```
RECORD:
while (my $record = <$inventory_file>) {
    chomp $record;
    next RECORD if $record eq $EMPTY_STR;
    my @fields = split $FIELD_SEPARATOR, $record;
    update_sales(\@fields);
    $count++;
}
```

Wskazówka ta dotyczy nawet bloków `map` i `grep`, które zawierają więcej niż jedną instrukcję. Należy pisać:

```
my @clean_words
    = map {
          my $word = $_;
          $word =~ s/$EXPLETIVE/[DELETED]/gxms;
          $word;
      } @raw_words;
```

a nie:

```
my @clean_words
    = map { my $word = $_; $word =~ s/$EXPLETIVE/[DELETED]/gxms; $word } @raw_words;
```

Akapity

Dziel kod na akapity.

Akapit to zbiór instrukcji, które realizują pojedyncze zadanie; w literaturze jest to seria zdań przekazujących jedną ideę, a w programowaniu — seria instrukcji odpowiadających jednej fazie algorytmu.

Kod trzeba dzielić na sekwencje, które realizują pojedyncze zadanie. Między kolejnymi sekwencjami należy umieszczać puste wiersze. Aby jeszcze bardziej ułatwić konserwację kodu, na początku każdego akapitu powinno się dopisywać jednowierszowy komentarz wyjaśniający przeznaczenie danej sekwencji:

```
# Przetwarzamy rozpoznaną tablicę...
sub addarray_internal {
    my ($var_name, $needs_quotemeta) = @_;

    # Buforujemy oryginał...
    $raw .= $var_name;

    # Na żądanie konstruujemy kod przytaczający metaznaki...
    my $quotemeta = $needs_quotemeta ?  q{map {quotemeta $_} } : $EMPTY_STR;

    # Rozwijamy elementy zmiennej, łączymy je za pomocą operacji OR...
    my $perl5pat = qq{(??{join q{|}, $quotemeta \@{$var_name}})};

    # Na żądanie wstawiamy kod diagnostyczny...
    my $type = $quotemeta ? 'literał' : 'wzorzec';
    debug_now("Dodaję $var_name (jako $type)");
    add_debug_mesg("Wypróbowuję $var_name (jako $type)");

    return $perl5pat;
}
```

Akapity są przydatne, ponieważ ludzie potrafią skupić się tylko na kilku informacjach jednocześnie[6]. Akapity grupują powiązane informacje, dzięki czemu wynikowa „porcja" może zmieścić się w ograniczonej pamięci krótkotrwałej. Dzięki akapitom fizyczna struktura tekstu od-

[6] Ideę tę przedstawił w 1956 roku George A. Miller w artykule „Magiczna liczba siedem, plus minus dwa" (*The Psychological Review*, 1956, Vol. 63, s. 81 – 97).

zwierciedla i podkreśla strukturę logiczną. Komentarze na początku akapitów wzmacniają ten podział, jawnie podsumowując przeznaczenie każdego fragmentu[7].

Zauważmy jednak, że komentarze mają tu drugorzędne znaczenie. Kluczowe są pionowe odstępy między akapitami. Bez nich czytelność kodu znacznie się zmniejsza, nawet w razie zachowania komentarzy:

```
sub addarray_internal {
    my ($var_name, $needs_quotemeta) = @_;
    # Buforujemy oryginał...
    $raw .= $var_name;
    # Na żądanie konstruujemy kod przytaczający metaznaki...
    my $quotemeta = $needs_quotemeta ?  q{map {quotemeta $_} } : $EMPTY_STR;
    # Rozwijamy elementy zmiennej, łączymy je za pomocą operacji OR...
    my $perl5pat = qq{(??{join q{|}, $quotemeta \@{$var_name}})};
    # Na żądanie wstawiamy kod diagnostyczny...
    my $type = $quotemeta ? 'literał' : 'wzorzec';
    debug_now("Dodaję $var_name (jako $type)");
    add_debug_mesg("Wypróbowuję $var_name (jako $type)");
    return $perl5pat;
}
```

Instrukcje else

Nie stłaczaj instrukcji `else`.

„Stłoczona" instrukcja `else` wygląda tak:

```
} else {
```

A „niestłoczona" tak:

```
}
else {
```

Stłoczone instrukcje `else` pozwalają oszczędzić jeden wiersz na każdą alternatywę, ale ostatecznie zmniejszają czytelność kodu, zwłaszcza sformatowanego w stylu K&R. Stłoczona instrukcja `else` nie znajduje się w jednej linii ani z kontrolującą ją instrukcją `if`, ani z własnym nawiasem zamykającym. To przesunięcie utrudnia wizualne dopasowanie poszczególnych komponentów konstrukcji `if`-`else`.

Co ważniejsze, instrukcja `else` definiuje alternatywny tryb postępowania, kiedy zaś jest stłoczona, rozróżnienie to staje się mniej wyraźne. Znika niemal pusty wiersz z nawiasem klamrowym zamykającym instrukcję `if`, co zmniejsza wizualny odstęp pomiędzy blokami `if` i `else`. Takie ściśnięcie bloków kłóci się z ich wewnętrznym układem, zwłaszcza jeśli są one podzielone na akapity w sposób opisany w poprzednim podrozdziale.

Stłaczanie sprawia też, że instrukcja `else` nie zajmuje należnego jej miejsca po lewej stronie wiersza, co utrudnia zlokalizowanie słowa kluczowego podczas przeglądania kodu. Natomiast niestłoczona instrukcja `else` poprawia pionowy podział kodu i ułatwia identyfikację słowa kluczowego:

[7] **Znaczenie**, a nie **działanie**. Komentarze przed akapitami mają wyjaśniać, do czego służy kod, a nie parafrazować realizowane przez niego operacje.

```
if ($sigil eq '$') {
    if ($subsigil eq '?') {
        $sym_table{ substr($var_name,2) } = delete $sym_table{$var_name};

        $internal_count++;
        $has_internal{$var_name}++;
    }
    else {
        ${$var_ref} = q{$sym_table{$var_name}};

        $external_count++;
        $has_external{$var_name}++;
    }
}
elsif ($sigil eq '@' && $subsigil eq '?') {
    @{ $sym_table{$var_name} }
        = grep {defined $_} @{$sym_table{$var_name}};
}
elsif ($sigil eq '%' && $subsigil eq '?') {
    delete $sym_table{$var_name}{$EMPTY_STR};
}
else {
    ${$var_ref} = q{$sym_table{$var_name}};
}
```

Porównajmy to ze stłoczoną instrukcją `else` lub `elseif`, która zaciemnia wewnętrzny podział bloków na akapity i zmniejsza widoczność słów kluczowych:

```
if ($sigil eq '$') {
    if ($subsigil eq '?') {
        $sym_table{ substr($var_name,2) } = delete $sym_table{$var_name};

        $internal_count++;
        $has_internal{$var_name}++;
    } else {
        ${$var_ref} = q{$sym_table{$var_name}};

        $external_count++;
        $has_external{$var_name}++;
    }
} elsif ($sigil eq '@' && $subsigil eq '?') {
    @{$sym_table{$var_name}}
        = grep {defined $_} @{$sym_table{$var_name}};
} elsif ($sigil eq '%' && $subsigil eq '?') {
    delete $sym_table{$var_name}{$EMPTY_STR};
} else {
    ${$var_ref} = q{$sym_table{$var_name}};
}
```

Wyrównanie pionowe

Wyrównuj pionowo powiązane ze sobą elementy.

Innym, dobrze znanym sposobem grupowania pokrewnych informacji (i sygnalizowania relacji logicznych przez układ fizyczny) są tabele. Podczas formatowania kodu często warto rozmieścić dane w sposób przypominający tabelę. Jednolite wcięcia mogą sugerować równoważną strukturę, użycie lub przeznaczenie.

Przykładowo inicjalizatory zmiennych nieskalarnych są znacznie czytelniejsze, kiedy ułoży się je w kolumny z wykorzystaniem dodatkowych odstępów. Poniższe inicjalizatory tablicy zwykłej i asocjacyjnej są bardzo czytelne właśnie dzięki układowi tabelarycznemu:

```
my @months = qw(
    Styczeń      Luty      Marzec
    Kwiecień     Maj       Czerwiec
    Lipiec       Sierpień  Wrzesień
    Październik  Listopad  Grudzień
);

my %expansion_of = (
    q{it's}    => q{it is},
    q{we're}   => q{we are},
    q{didn't}  => q{did not},
    q{must've} => q{must have},
    q{I'll}    => q{I will},
);
```

Przekształcenie ich w listy pozwala zaoszczędzić kilka wierszy, ale znacznie zmniejsza ich czytelność:

```
my @months = qw(
    Styczeń Luty Marzec Kwiecień Maj Czerwiec Lipiec Sierpień  Wrzesień
    Październik Listopad Grudzień
);

my %expansion_of = (
    q{it's} => q{it is}, q{we're} => q{we are}, q{didn't} => q{did not},
    q{must've} => q{must have}, q{I'll} => q{I will},
);
```

Podobny układ warto stosować w sekwencjach ustawiających wartości pokrewnych zmiennych. Lepiej wyrównać operatory przypisania:

```
$name   = standardize_name($name);
$age    = time - $birth_date;
$status = 'aktywny';
```

niż pisać w taki sposób:

```
$name = standardize_name($name);
$age = time - $birth_date;
$status = 'aktywny';
```

Wyrównanie jest jeszcze ważniejsze podczas przypisywania wartości elementom tablicy asocjacyjnej lub zwykłej. W takich przypadkach klucze (lub indeksy) należy ułożyć w kolumnie, a otaczające je nawiasy klamrowe (lub okrągłe) również powinny być wyrównane:

```
$ident{ name   } = standardize_name($name);
$ident{ age    } = time - $birth_date;
$ident{ status } = 'aktywny';
```

Układ tabelaryczny wyróżnia klucze elementów, a zatem podkreśla cel każdego przypisania. Bez tego uwagę przyciąga „kolumna” przedrostków `$ident`, przez co dużo trudniej odróżnić nazwy kluczy:

```
$ident{name} = standardize_name($name);
$ident{age} = time - $birth_date;
$ident{status} = 'aktywny';
```

Wyrównanie samych operatorów przypisania jest lepsze niż zupełny brak wyrównania, ale nie aż tak czytelne jak wyrównanie zarówno kluczy, jak i operatorów:

```
$ident{ name }   = standardize_name($name);
$ident{ age }    = time - $birth_date;
$ident{ status } = 'aktywny';
```

Dzielenie długich wierszy

Dziel długie wyrażenia przed operatorem.

Kiedy wyrażenie na końcu instrukcji jest zbyt długie, często dzieli się je zaraz po operatorze i kontynuuje od nowego wiersza wcięte o jeden poziom:

```
push @steps, $steps[-1] +
    $radial_velocity * $elapsed_time +
    $orbital_velocity * ($phase + $phase_shift) -
    $DRAG_COEFF * $altitude;
```

Operator na końcu wiersza ma pełnić funkcję znacznika kontynuacji — sygnalizować, że instrukcja ciągnie się dalej w następnym wierszu.

Używanie operatora jako znacznika kontynuacji wydaje się doskonałym pomysłem, ale jest z tym pewien problem: ludzie rzadko patrzą na prawą stronę kodu. Większość wskazówek semantycznych — takich jak słowa kluczowe — pojawia się po lewej stronie. Co ważniejsze, wskazówki strukturalne, na przykład wcięcia, również znajdują się z lewej strony (więcej informacji na ten temat można znaleźć w ramce „Na lewo patrz"). Oznacza to, że wcinanie kolejnych wierszy wyrażenia w rzeczywistości wywołuje fałszywe wrażenie podstawowej struktury, które trzeba skorygować poprzez prześledzenie całego wiersza aż do prawego marginesu.

Na lewo patrz

Lewa krawędź kodu jest najbardziej wyróżniającym się miejscem, ponieważ — podobnie jak język polski — Perl jest zasadniczo językiem typu „od lewej do prawej", a w takich językach lewa część wyrażenia jest szczególnie istotna.

Na początku wyrażenia czytelnik jest „świeży"; nie musi pamiętać niczego, co nastąpiło wcześniej. Natomiast na końcu wyrażenia pamięć krótkotrwała jest wypełniona, a czytelnik skupia uwagę na interpretacji całego wiersza albo w ogóle traci koncentrację.

Lingwiści nazywają ten efekt „problemem wagi końcowej" i odradzają zachowywanie ważnych informacji na sam koniec:

> *Ponieważ po długiej nocy spędzonej na programowaniu w przerażającym śnie przyszły do mnie potępione dusze odpowiedzialne za ANSI C++, uciekłem z krzykiem.*

Jeśli informacja ta zostanie umieszczona na początku, łatwiej będzie zwrócić na nią uwagę, mimo że reszta zdania nieco się rozmyje:

> *Uciekłem z krzykiem, ponieważ potępione dusze odpowiedzialne za ANSI C++ przyszły do mnie w przerażającym śnie po długiej nocy spędzonej na programowaniu.*

Oczywiście, można zaprojektować język programowania, w którym ważne informacje są umieszczone na szarym końcu — przykładami mogą być Forth i PostScript — ale, na szczęście, Perl nie jest takim językiem.

Lepszym rozwiązaniem jest dzielenie długich wierszy **przed** operatorem. Dzięki temu każdy wiersz kontynuowanego wyrażenia będzie zaczynał się od operatora, co w kodzie Perla jest niezwykłe. Kiedy osoba czytająca kod będzie przesuwać wzrok wzdłuż lewej krawędzi, natychmiast zauważy, że wcięty wiersz jest dalszym ciągiem poprzedniego.

Bardzo istotne jest również wcięcie drugiego i następnych wierszy. Kontynuowanych wierszy nie należy przesuwać do następnego poziomu wcięcia, ale do początkowej kolumny wyrażenia, do którego należą, tzn. że nie należy pisać tak:

```
push @steps, $steps[-1]
    + $radial_velocity * $elapsed_time
    + $orbital_velocity * ($phase + $phase_shift)
    - $DRAG_COEFF * $altitude
    ;
```

lecz tak:

```
push @steps, $steps[-1]
             + $radial_velocity * $elapsed_time
             + $orbital_velocity * ($phase + $phase_shift)
             - $DRAG_COEFF * $altitude
             ;
```

Układ ten ma dodatkową zaletę — dwa argumenty instrukcji `push` są wizualnie oddzielone, dzięki czemu łatwiej je odróżnić.

Jeśli wyrażenie rozciąga się na wiele wierszy, warto umieścić końcowy średnik w oddzielnym wierszu i w tej samej kolumnie, od której rozpoczyna się kontynuowany tekst. Kiedy czytelnik będzie przesuwał wzrok wzdłuż operatorów rozpoczynających kolejne wiersze, napotkanie samotnego średnika wyraźnie zasygnalizuje mu, że wyrażenie dobiegło końca.

Wyrażenia nieterminalne

Wyodrębniaj długie wyrażenia ze środka instrukcji.

Poprzednia wskazówka dotyczy tylko sytuacji, w której długie wyrażenie jest ostatnim elementem instrukcji. Jeśli występuje ono w środku instrukcji, lepiej wyodrębnić je w oddzielne przypisanie zmiennej, na przykład:

```
my $next_step = $steps[-1]
              + $radial_velocity * $elapsed_time
              + $orbital_velocity * ($phase + $phase_shift)
              - $DRAG_COEFF * $altitude
              ;
add_step( \@steps, $next_step, $elapsed_time);
```

zamiast:

```
add_step( \@steps, $steps[-1]
              + $radial_velocity * $elapsed_time
              + $orbital_velocity * ($phase + $phase_shift)
              - $DRAG_COEFF * $altitude
              , $elapsed_time);
```

Dzielenie wyrażeń według priorytetu

Zawsze dziel długie wyrażenie na operatorze o najniższym priorytecie.

Jak pokazują przykłady w poprzednich dwóch podrozdziałach, podczas dzielenia wyrażenia na wiele wierszy każdy wiersz powinien rozpoczynać się od operatora o niskim priorytecie. Dzielenie wierszy na operatorach o wyższym priorytecie może sprawić, że nieuważny czytelnik błędnie zinterpretuje obliczenia. Poniższy układ na przykład może zasugerować, że dodawania i odejmowania zachodzą przed mnożeniami:

```
push @steps, $steps[-1] + $radial_velocity
             * $elapsed_time + $orbital_velocity
             * ($phase + $phase_shift) - $DRAG_COEFF
             * $altitude
             ;
```

Jeśli konieczne jest podzielenie wiersza na operatorze o wysokim priorytecie, należy wciąć dalszy ciąg wiersza o jeden poziom względem początku wyrażenia:

```
push @steps, $steps[-1]
             + $radial_velocity * $elapsed_time
             + $orbital_velocity
                 * ($phase + $phase_shift)
             - $DRAG_COEFF * $altitude
             ;
```

Dzięki tej strategii podwyrażenia o wyższym priorytecie pozostają wizualnie „blisko siebie".

Przypisania

Dziel długie instrukcje przed operatorem przypisania.

Często długą instrukcją, która wymaga podzielenia, jest przypisanie. W takich przypadkach można zastosować poprzednią regułę, ale prowadzi ona do kodu nieestetycznego i nieczytelnego:

```
$predicted_val = $average
               + $predicted_change * $fudge_factor
               ;
```

Tego rodzaju instrukcje lepiej dzielić przed operatorem przypisania, pozostawiając w pierwszym wierszu tylko nazwę zmiennej. Następny wiersz należy wciąć o jeden poziom i umieścić w nim operator przypisania, który będzie pełnił funkcję znacznika kontynuacji:

```
$predicted_val
    = $average + $predicted_change * $fudge_factor;
```

Metoda ta często pozwala zmieścić prawą stronę przypisania w jednym wierszu, tak jak w powyższym przykładzie. Jeśli jednak wyrażenie po prawej stronie nadal jest za długie, należy podzielić je ponownie na operatorze o niskim priorytecie, w sposób zasugerowany w poprzedniej wskazówce:

```
$predicted_val
    = ($minimum + $maximum) / 2
      + $predicted_change * max($fudge_factor, $local_epsilon);
```

Inną strategią jest dzielenie długich instrukcji za operatorem przypisania:

```
$predicted_val =
    $average + $predicted_change * $fudge_factor;
```

Metoda ta ma jednak opisany wcześniej problem: uniemożliwia wykrycie kontynuacji wiersza bez prześledzenia go aż do prawego marginesu kodu, a „nienacechowane" wcięcie drugiego wiersza może wprowadzić w błąd nieuważnego czytelnika. Problem czytelności staje się szczególnie dotkliwy, kiedy zmienna, której przypisywana jest wartość, sama jest długa:

```
$predicted_val{$current_data_set}[$next_iteration] =
    $average + $predicted_change * $fudge_factor;
```

a właśnie w takich sytuacjach przypisanie zwykle wymaga podziału. Dzielenie wiersza przed operatorem przypisania ułatwia identyfikację długich przypisań, ponieważ operator pozostaje blisko wyrażenia:

```
$predicted_val{$current_data_set}[$next_iteration]
    = $average + $predicted_change * $fudge_factor;
```

Operator trójkowy

Formatuj w kolumny wyrażenia z kaskadowymi operatorami trójkowymi.

Operator trójkowy zachęca do tworzenia szczególnie długich wyrażeń. Ponieważ elementy ? oraz : tego operatora mają bardzo niski priorytet, prosta interpretacja reguły dzielenia długich wyrażeń nie sprawdza się w tym przypadku, bowiem prowadzi do instrukcji w rodzaju:

```
my $salute = $name eq $EMPTY_STR ? 'Customer'
           : $name =~ m/\A((?:Sir|Dame) \s+ \S+)/xms ? $1
           : $name =~ m/(.*), \s+ Ph[.]?D \z/xms ? "Dr $1" : $name;
```

które są bardzo nieczytelne.

Serię operatorów trójkowych najlepiej ułożyć w dwóch kolumnach:

```
            # Kiedy klient ma na nazwisko..          Tytułujemy go...
my $salute = $name eq $EMPTY_STR                    ? 'Customer'
           : $name =~ m/\A((?:Sir|Dame) \s+ \S+) /xms ? $1
           : $name =~ m/(.*), \s+ Ph[.]?D \z     /xms ? "Dr $1"
           :                                            $name
           ;
```

Innymi słowy, należy dzielić serię operatorów trójkowych przed każdym dwukropkiem, wyrównując dwukropki z operatorem poprzedzającym pierwszy warunek. Dzięki temu testy utworzą kolumnę. Następnie trzeba wyrównać znaki zapytania w taki sposób, aby możliwe wyniki operatora trójkowego również tworzyły kolumnę. Na koniec należy wciąć ostatni wynik (niepoprzedzony znakiem zapytania) tak, żeby również znalazł się w kolumnie wyników.

Ten specjalny układ zmienia nieczytelną sekwencję operatorów trójkowych w prostą tabelę wyszukiwania — dla danego warunku w pierwszej kolumnie użyć odpowiedniego wyniku z drugiej.

Układu tabelarycznego można użyć nawet wtedy, gdy instrukcja zawiera tylko jeden operator trójkowy:

```
my $name = defined $customer{name} ? $customer{name}
         :                           'Sir or Madam'
         ;
```

Dzięki temu kolejni programiści będą mogli łatwiej dodawać do tabeli kolejne możliwości. Ideę tę zbadamy dokładniej w podrozdziale „Operatory trójkowe w układzie tabelarycznym" w rozdziale 6.

Listy

Umieszczaj długie listy w nawiasach okrągłych.

Przecinek jest operatorem tylko w kontekście skalarnym; na listach jest separatorem elementów. Dlatego przecinki na listach wielowierszowych lepiej traktować jak terminatory. Co więcej, łatwo pomylić wielowierszową listę z sekwencją instrukcji, ponieważ wizualna różnica między przecinkiem a średnikiem jest niewielka.

Ze względu na możliwość nieporozumień warto jasno oznaczyć listę wielowierszową jako listę. Jeśli więc konieczne jest podzielenie listy na wiele wierszy, należy umieścić ją w nawiasie okrągłym. Nawias otwierający podkreśla fakt, że następujące po nim wyrażenie jest listą, a nawias zamykający jednoznacznie wskazuje, że lista dobiegła końca.

Podczas formatowania instrukcji zawierającej wielowierszową listę należy umieścić nawias otwierający w tym samym wierszu, w którym znajduje się poprzednia część instrukcji. Następnie trzeba podzielić listę po każdym przecinku, umieszczając jednakową liczbę elementów w każdym wierszu i wcinając te wiersze o jeden poziom w stosunku do instrukcji. Nawias zamykający należy umieścić na tym samym poziomie wcięcia, na jakim jest instrukcja, na przykład:

```
my @months = qw(
    Styczeń      Luty      Marzec
    Kwiecień     Maj       Czerwiec
    Lipiec       Sierpień  Wrzesień
    Październik  Listopad  Grudzień
);

for my $item (@requested_items) {
    push @items, (
        "Zupełnie nowy $item",
        "W pełni odnowiony $item",
        "Sfatygowany stary $item",
    );
}

print (
    'Przetwarzam ',
    scalar(@items),
    ' elementów o ',
    time,
    "\n",
);
```

Warto pamiętać, że ostatni element na liście również powinien być opatrzony przecinkiem, choć nie jest to wymagane syntaktycznie.

Podczas pisania wielowierszowych list zawsze należy używać nawiasów okrągłych (w stylu K&R), umieszczać tyle samo elementów w każdym wierszu i pamiętać, że w kontekście listy przecinek nie jest operatorem, więc reguła dzielenia przed operatorem w tym przypadku nie obowiązuje. Innymi słowy, nie należy pisać tak:

```
my @months = qw( Styczeń  Luty  Marzec  Kwiecień  Maj  Czerwiec  Lipiec  Sierpień
                 Wrzesień  Październik  Listopad  Grudzień
               );

for my $item (@requested_items) {
    push @items, "Zupełnie nowy $item"
               , "W pełni odnowiony $item"
               , "Sfatygowany stary $item"
               ;
}

print 'Przetwarzam '
    , scalar(@items)
    ,' elementów o '
    , time
    , "\n"
    ;
```

W podrozdziale „Cienkie przecinki”, w rozdziale 4. podano inne przesłanki do umieszczania list w nawiasie okrągłym.

Zautomatyzowane formatowanie

Wymuszaj stosowanie wybranego stylu kodowania.

Na dłuższą metę najlepiej wyćwiczyć siebie i innych członków zespołu w konsekwentnym, racjonalnym i czytelnym stylu kodowania, takim jak zasugerowany wcześniej. Czasem brakuje jednak na to czasu lub sił. Wtedy można pójść na kompromis i wybrać narzędzie do formatowania kodu, które trzeba będzie stosować przed przekazaniem programu do przejrzenia albo publicznej prezentacji.

Doskonałe narzędzie do formatowania kodu Perla, to *perltidy*. Jest ono dostępne bezpłatnie w witrynie SourceForge pod adresem *http://perltidy.sourceforge.net* i oferuje szeroką gamę opcji konfiguracyjnych, które pozwalają określić wcięcia, układ ograniczników bloków, wyrównywanie kolumn oraz pozycjonowanie komentarzy.

Za pomocą programu *perltidy* można przekształcić poniższy kod:

```
if($sigil eq '$'){
    if($subsigil eq '?'){
        $sym_table{substr($var_name,2)}=delete $sym_table{locate_orig_var($var_name)};
        $internal_count++;$has_internal{$var_name}++
    } else {
        ${$var_ref} =
            q{$sym_table{$var_name}}; $external_count++; $has_external{$var_name}++;
```

```
}} elsif ($sigil eq '@'&&$subsigil eq '?') {
    @{$sym_table{$var_name}} = grep
        {defined $_} @{$sym_table{$var_name}};
} elsif ($sigil eq '%' && $subsigil eq '?') {
delete $sym_table{$var_name}{$EMPTY_STR}; } else
{
${$var_ref}
=
q{$sym_table{$var_name}}
}
```

w coś bardziej czytelnego:

```
if ( $sigil eq '$' ) {
    if ( $subsigil eq '?' ) {
        $sym_table{ substr( $var_name, 2 ) }
            = delete $sym_table{ locate_orig_var($var_name) };
        $internal_count++;
        $has_internal{$var_name}++;
    }
    else {
        ${$var_ref} = q{$sym_table{$var_name}};
        $external_count++;
        $has_external{$var_name}++;
    }
}
elsif ( $sigil eq '@' && $subsigil eq '?' ) {
    @{ $sym_table{$var_name} }
        = grep {defined $_} @{ $sym_table{$var_name} };
}
elsif ( $sigil eq '%' && $subsigil eq '?' ) {
    delete $sym_table{$var_name}{$EMPTY_STR};
}
else {
    ${$var_ref} = q{$sym_table{$var_name}};
}
```

Zauważmy, że zmodyfikowana wersja spełnia reguły formatowania podane w niniejszym rozdziale. Aby osiągnąć taki rezultat, należy skonfigurować plik *.perltidyrc* w następujący sposób:

```
-l=78       # Maksymalna szerokość wiersza: 78 kolumn
-i=4        # Poziom wcięcia: 4 kolumny
-ci=4       # Wcięcie kontynuowanego wiersza: 4 kolumny
-st         # Wyniki na STDOUT
-se         # Błędy na STDERR
-vt=2       # Maksymalne ścieśnienie w pionie
-cti=0      # Bez dodatkowego wcinania nawiasów zamykających
-pt=1       # Średnie ścieśnienie nawiasów okrągłych
-bt=1       # Średnie ścieśnienie nawiasów kwadratowych
-sbt=1      # Średnie ścieśnienie nawiasów klamrowych
-bbt=1      # Bez spacji przed przecinkami
-nsfs       # Bez zmniejszania wcięcia długich łańcuchów w cudzysłowie
-nolq
-wbb="% + - * / x != == >= <= =~ !~ < > | & >= < = **= += *= &= <<= &&= -=
      /= |= >>= ||= .= %= ^= x="
            # Podział przed wszystkimi operatorami
```

Nakaz formatowania kodu z wykorzystaniem określonego narzędzia pozwala też uniknąć obiekcji, kłótni i wygłaszania dogmatów nieodłącznie związanych z każdą dyskusją na temat układu kodu. Jeśli całą pracę wykonuje program *perltidy*, programiści mogą przyjąć nowe zalecenia praktycznie bez wysiłku. Wystarczy, że ustawią w edytorze makro, które na żądanie „uporządkuje” kod.