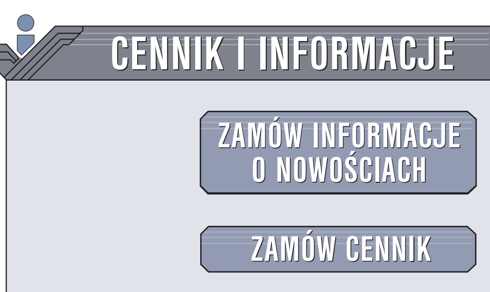

Perl. Zaawansowane programowanie. Wydanie II

Autor: Simon Cozens
Tłumaczenie: Sławomir Dzieniszewski, Mateusz Michalski
ISBN: 83-246-0231-3
Tytuł oryginału: Advanced Perl Programming, 2nd Edition
Format: B5, stron: 288

Bądź na bieżąco z najnowszymi narzędziami i technikami programowania

- Wykorzystaj możliwości szablonów
- Połącz skrypty Perla z programami napisanymi w innych językach programowania
- Przetestuj kod i popraw wydajność projektu

Perl, jeden z pierwszych języków skryptowych służących do dynamicznego generowania zawartości witryn WWW, nie traci popularności mimo dość sędziwego wieku. Ciągle wzrasta zainteresowanie tym językiem, co pewien czas powstają kolejne jego wersje. W sieci funkcjonują tysiące witryn poświęconych programowaniu w Perlu, jednak podczas pracy nad rozbudowaną aplikacją nie zawsze znajdujemy czas na poszukiwanie niezbędnych informacji. Zgromadzenie w jednym tomie opisów technik stosowanych przez doświadczonych programistów pozwoliłoby znacznie przyspieszyć pracę.

Książka „Perl. Zaawansowane programowanie. Wydanie II" to kompendium wiedzy dla wszystkich, którzy tworzą aplikacje w Perlu. Autor opisuje zmiany, jakie wprowadzono w Perlu w ciągu ostatnich lat, koncentrując się na technikach rozwiązywania konkretnych problemów, a nie na teoretycznych rozważaniach. Kładzie duży nacisk na możliwości stosowania gotowych kodów z witryn Comprehensive Perl Archive Network, w których zgromadzono setki przykładów wykorzystania Perla. Uczy efektywnej pracy i zachęca do stosowania zaawansowanych narzędzi i technik programistycznych.

- Techniki przetwarzania danych tekstowych
- Stosowanie szablonów
- Pobieranie wiadomości RSS
- Obsługa baz danych
- Korzystanie z kodowania Unicode
- Programowanie sterowane zdarzeniami
- Testowanie kodu i usuwanie błędów
- Łączenie kodu Perla z kodem C za pomocą modułu Inline

Odkryj magię Perla

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

Spis treści

Przedmowa 7

1. Techniki zaawansowane 11
Introspekcja 12
Modyfikacja modelu klas 29
Nieoczekiwany kod 34
Podsumowanie 50

2. Techniki parsowania 51
Gramatyki Parse::RecDescent 52
Parse::Yapp 74
Inne techniki parsowania 78
Podsumowanie 82

3. Szablony 83
Formaty i Text::Autoformat 84
Text::Template 88
HTML::Template 93
HTML::Mason 98
Template Toolkit 109
AxKit 115
Podsumowanie 117

4. Obiekty, bazy danych i aplikacje 119
Coś więcej niż zwykłe pliki… 119
Serializacja obiektów 120
Bazy danych obiektów 130

Obsługa baz danych 134
Zastosowania praktyczne w aplikacjach sieciowych 141
Posumowanie 147

5. Narzędzia językowe ... 149
Perl i praca z tekstem 149
Obróbka tekstów angielskich 150
Moduły do parsowania tekstów angielskich 153
Klasyfikacja i pozyskiwanie informacji 158
Podsumowanie 168

6. Perl i Unicode ... 169
Terminologia 169
Co to takiego Unicode? 171
Formaty UTF 173
Obsługa danych UTF-8 176
Moduł Encode 181
Unicode dla programistów XS 187
Podsumowanie 191

7. POE ... 193
Programowanie w środowisku sterowanym zdarzeniami 193
Elementy najwyższego poziomu — komponenty 204
Podsumowanie 211

8. Testowanie ... 213
Test::Simple 213
Test::More 215
Test::Harness 218
Test::Builder 219
Test::Builder::Tester 221
Łączenie testów z kodem 223
Testowanie jednostek kodu 224
Podsumowanie 230

9. Rozszerzanie możliwości Perla za pomocą modułu Inline ... 233
Prosty moduł Inline::C 233
Programowanie bardziej złożonych zadań z pomocą Inline::C 236
Inline::Inne moduły 249
Podsumowanie 254

10. Zabawy z Perlem ... 255
Nieczytelność 255
Just another Perl hacker 260
Golf Perla 262
Poezja Perla 264
Acme::* 265
Podsumowanie 269

Skorowidz ... 271

ROZDZIAŁ 3.

Szablony

Na grupie `comp.lang.perl` pojawił się niedawno wątek, w którym dyskutujący próbowali ustalić rytualną drogę każdego programisty Perla — indywidualne wyważanie szeroko otwartych drzwi. Na liście pokonywanych zadań znalazł się system obsługi szablonów, warstwa abstrakcji bazy danych, parser HTML, procesor argumentów wiersza poleceń oraz moduł obsługi dat i czasu.

Ciekawe, czy poniższa historia nie wyda Ci się znajoma. Musisz przygotować pewien formularz listu. Część tekstu jest stała i niezmienna, niektóre wyrażenia się zmieniają. Tworzysz więc taki mniej więcej szablon:

```
my $template = q{
    Szanowny Panie(Pani) $name,

    Otrzymaliśmy państwa zamówienie dotyczące $product. Z naszych ustaleń wynika,
    że będziemy w stanie dostarczyć go do państwa w dniu $date za cenę około $cost.

    Dziękujemy za zainteresowanie naszą ofertą,

    Acme Integrated Foocorp.
};
```

a następnie zmagasz się z koszmarnymi wyrażeniami regularnymi przy każdej linii, na przykład `s/(\$\w+)/$1/eeg`, osiągając w końcu coś, co lepiej lub gorzej działa jak powinno.

Jak to zwykle bywa z każdym projektem, dwa dni od wprowadzenia programu do użytku zmienia się specyfikacja danych i nagle Twój prosty wzorzec musi uwzględniać zastosowanie pętli dla elementów tablic, wyrażenia warunkowe czy wreszcie wykonywanie kodu Perla w samym środku wzorca. I nawet nie spostrzegasz, jak to się stało, że utworzyłeś własny język obsługi wzorców.

Jeżeli rozpoznałeś w tym siebie, nie przejmuj się. To samo przydarza się niemal wszystkim przynajmniej raz. I właśnie dlatego w CPAN znaleźć można tak wiele modułów obsługujących szablony plików tekstowych i HTML, wśród których występują tak koncepcje niewiele bardziej złożone od `s/(\$\w+)/$1/eeg`, jak i niezależne języki programowania wzorców.

Zanim przejdziemy do omawiania tych modułów, przyjrzymy się rozwiązaniu wbudowanemu w Perla — formatom.

Formaty i Text::Autoformat

Formaty są częścią Perla od wersji 1.0. Rzadko się z nich teraz korzysta, ale nadają się świetnie do uzyskania efektów, na których w wielu przypadkach zależy nam przy formatowaniu tekstów.

Formaty Perla umożliwiają utworzenie obrazu danych do wyświetlenia, a następnie wypełnienie go rzeczywistymi danymi. Przykładowo, w niedawno tworzonej aplikacji zależało mi na wyświetlaniu w oddzielnych wierszach danych o odebranych wiadomościach: identyfikatora, daty, adresu nadawcy i tematu wiadomości. Zakładając, że wiersz pomieścić może 80 kolumn, niektóre pola trzeba było ograniczyć, a inne wypełnić odstępami, tak by zajmowały nieco więcej miejsca. W czystym Perlu efekt tak sformatowanego wyjścia można osiągnąć na trzy podstawowe sposoby. Po pierwsze, korzystając z `sprintf` (lub `printf`) i `substr`:

```
for (@mails) {
    print "%5i %10s %40s %21s\n",
        $_->id,
        substr($_->received,0,10),
        substr($_->from_address,-40,40),
        substr($_->subject,0,21);
}
```

Po drugie, posługując się funkcją `pack`, o której prawie nikt nie pamięta (a która nie daje dużej kontroli przy obcinaniu łańcuchów):

```
for (@mails) {
    print pack("A5 A10 A40 A21\n",
      $_->id, $_->received, $_->from_address, $_->subject);
}
```

I w końcu po trzecie, używając formatu:

```
format STDOUT =
@<<<< @<<<<<<<<< @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< @<<<<<<<<<<<<<<<<<<<<
$_->id $_->received $_->from_address                         $_->subject
.

for (@mails) {
     write;
}
```

Osobiście uważam, że to rozwiązanie jest zgrabniejsze i bardziej intuicyjne niż dwa pozostałe. Ponadto ma tę zaletę, że formatowanie wydzielone jest poza pętlę główną, przez co kod staje się bardziej przejrzysty[1].

Formaty zawsze powiązane są z określonymi uchwytami plików; w podanym przykładzie przyjęto, że format powinien zostać zastosowany do każdego ciągu skierowanego na standardowe wyjście. Obrazkowy język formatów jest całkiem prosty — pola zaczynają się od znaków `@` lub `^`, po których pojawiają się znaki `<`, `|` lub `>` wskazujące na sposób wyrównania tekstu, odpowiednio: lewostronne, wyśrodkowane i prawostronne. Po każdej linii opisów pól pojawia się linia z wyrażeniami wypełniającymi te pola, z oddzielnym wyrażeniem dla każdego pola:

[1] Tak się złożyło, że ostatecznie nie skorzystałem z tego rozwiązania w swoim rzeczywistym programie — obszary wyświetlania poszczególnych pól musiały mieć zmienną, a nie stałą długość. Jednak w przypadkach, w których wystarcza stała szerokość pól, rozwiązanie to jest idealne.

```
format STDOUT =
Id      : @<<<<
$_->id
Date    : @<<<<<<<
$_->received
From    : @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$_->from_address
Subject : @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$_->subject

.
```

Jak do tej pory widzieliśmy przykłady tylko pól typu `@`. W przypadku formatów wieloliniowych może się okazać, że przydałoby się podzielić wyrażenie na kilka oddzielnych linii; na przykład po to, by początek treści wiadomości wyświetlać obok metadanych o tej wiadomości:

```
Id      : 1                                    Hi Simon, Thank you for the
Date    : 10/12/02                             supply of widgets that you sent
From    : fred@funglyfoobar.com                me last week. I can assure you
Subject : Widgets                              that they have all been put …
```

W takich przypadkach korzysta się z pól drugiego typu: `^`. Zaprezentowany powyżej rezultat można osiągnąć dzięki takiemu formatowi:

```
format STDOUT
Id      : @<<<<                                ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$_->id                                         $message
Date    : @<<<<<<<                             ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$_->received                                   $message
From    : @<<<<<<<<<<<<<<<<<<<<                ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$_->from_address                               $message
Subject : @<<<<<<<<<<<<<<<<<<<…                ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<…
$_->subject                                    $message

.
```

Wartości wypełniające pola `@` mogą być dowolnymi wyrażeniami Perla, natomiast wypełnieniami pól `^` mogą być tylko zwykłe skalary. Procesor formatów przy każdym napotkaniu pola `^` wyświetla tyle znaków z podanej wartości, ile może, a następnie usuwa je z początku tej wartości, przygotowując ją w ten sposób do wyświetlenia w kolejnym polu. Znak ... na końcu pola oznacza, że jeśli podana wartość będzie za długa, by zmieścić się w przewidzianym polu, powinna zostać obcięta, a na końcu pojawić się mają trzy kropki. Jeżeli do określenia wartości wypełniającej pola `^` przekazywane będą zmienne leksykalne, takie jak `$message` w powyższym przykładzie, to trzeba je zadeklarować *przed* formatem; w przeciwnym razie nie będą one w nim widoczne.

Kolejnym udogodnieniem związanym z formatami jest możliwość określenia nagłówka wysyłanego na początku każdej strony — Perl zlicza linie wyświetlane przez format i dzięki temu wie, kiedy zaczyna się nowa strona. Nagłówek dla określonego uchwytu pliku to format, którego nazwą jest nazwa tego uchwytu z przyrostkiem `_TOP`. Proste zastosowanie tego mechanizmu to określenie nazw kolumn dla wyświetlanych rekordów:

```
format STDOUT_TOP =
ID Received From                               Subject
===== ========== ======================================== ====================
.

format STDOUT =
@<<<< @<<<<<<<<< @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< @<<<<<<<<<<<<<<<<<<<
$_->id $_->received $_->from_address                    $_->subject
.
```

Formaty są bardzo poręczne, szczególne w sytuacjach, w których z różnymi uchwytami plików wiążemy rozmaite formaty i wysyłamy te same dane w wiele miejsc w różnej postaci. Z drugiej strony mają one kilka poważnych ograniczeń, o których trzeba pamiętać, stosując je w dużych aplikacjach.

Przede wszystkim są nieprzebranym źródłem różnych zmiennych specjalnych: `$%` oznacza bieżący numer strony formatu, `$=` to liczba linii na stronie, `$-` to liczba linii pozostałych do końca strony, `$~` nazwa bieżącego formatu wyjściowego, `$^` nazwa bieżącego formatu nagłówkowego i tym podobne. Nigdy nie pamiętam znaczenia żadnej z nich i zawsze muszę to sprawdzać w *perlvar*.

Poza tym formaty słabo radzą sobie ze zmiennymi leksykalnymi, zmianą uchwytów plików, liniami o zmiennej długości, zmianą formatu w locie itd. Ale do małych, eleganckich rozwiązań nadają się wspaniale.

Szczegóły dotyczące wbudowanych formatów Perla znaleźć można w dokumentacji *perlform*.

Text::Autoformat

Istnieje jednak inny, godny 21. wieku sposób obsługi formatowania tekstów — moduł `Text::Autoformat`. Można go stosować na dwa sposoby — do zawijania długich łańcuchów (co robi inteligentniej niż moduł `Text::Wrap` czy uniksowe polecenie `fmt`) i jako zamiennik wbudowanego języka formatów, oferuje bowiem więcej możliwości przy prostszych zasadach składniowych.

Opcja zawijania długich tekstów powiązana jest w niewielkim stopniu z szablonami, ale chyba warto wspomnieć o niej w tym miejscu.

Głównym zadaniem procedury `autoformat` jest zachowywanie struktury zawijanego tekstu; została ona stworzona z myślą o wiadomościach pocztowych (ze szczególnym uwzględnieniem cytowanych fragmentów, sygnatur itp.), ale można ją zastosować do dowolnego ustrukturyzowanego tekstu. Przykładowo, gdyby taki tekst:

```
You have:
    * a splitting headache
    * no tea
    * your gown (being worn)
      It looks like your gown contains:
        . a thing your aunt gave which you don't know what it is
        . a buffered analgesic
        . pocket fluff
```

podać na wejście `fmt`, otrzymamy dość spektakularny miszmasz:

```
You have:
    * a splitting headache * no tea * your gown
    (being worn)
      It looks like your gown contains:
        . a thing your aunt gave you which
        you don't know what it is . a buffered
        analgesic . pocket fluff
```

Funkcja `autoformat` spisze się dużo lepiej, bo z wyprzedzeniem sprawdza, jaka jest struktura formatowanego tekstu:

```
You have:
    * a splitting headache
    * no tea
    * your gown (being worn) It looks like your
      gown contains:
        . a thing your aunt gave you which you
          don't know what it is
        . a buffered analgesic
        . pocket fluff
```

Język formatów z modułu `Text::Autoformat` jest bardzo podobny do standardowo oferowanego przez Perla, tyle że jest nieco uproszczony. Po pierwsze, rozróżnienie między polami wypełnianymi, `@`, a kontynuowanymi, `^`, dokonywane jest na podstawie znaków użytych do oznaczenia zakresu pola, a nie tylko prefiksu pola. I tak to, co zapisalibyśmy do tej pory jako:

```
@<<<< @<<<<<<<<< @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< @<<<<<<<<<<<<<<<<<<<<
```

staje się teraz po prostu czymś takim:

```
<<<<< <<<<<<<<<< <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< <<<<<<<<<<<<<<<<<<<<<
```

Do oznaczenia formatów kontynuowanych służą znaki `[` i `]`, które same powtarzają się w kolejnych wierszach w miarę potrzeb:

```
Id       : <<<<<
Messagge :
         [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
```

Na tej podstawie otrzymamy taki wynik:

```
Id      :      1
Message :
        Hi Simon, Thank you for the supply of widgets that you sent me
        last week. I can assure you that they have all been put to good…
```

Jednak w odróżnieniu od wbudowanych formatów wieloliniowych linie oznaczone znakami `[` i `]` powtarzają cały format automatycznie tak długo, by wyświetlić całą zawartość zmiennej. Z tego powodu poniższy zapis nie przyniesie spodziewanego rezultatu:

```
Id      : <<<<<   [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
```

Zamiast tego rezultat będzie następujący:

```
Id      :     1   Hi Simon, Thank you for the supply of widgets that you sent
Id      :         me last week. I can assure you that they have all been put
Id      :         to good use, and have been found, as usual to be the very…
```

A efekty są jeszcze gorsze przy zastosowaniu formatów dłuższych niż jedna linia.

Dużą zaletą `Text::Autoformat` jest natomiast to, że w jego przypadku formaty są zwykłymi łańcuchami, a nie odpowiednio kompilowanymi wzorcami przeplatanymi kodem. Łańcuchy tworzące format przetwarzane są przez funkcję `form`, którą trzeba jawnie importować:

```
use Text::Autoformat qw(form);

my $format = <<EOF;
Id      : <<<<<
Date    : <<<<<<<<
From    : <<<<<<<<<<<<<<<<<<<<<
Subject : <<<<<<<<<<<<<<<<<<<<<…
```

```
EOF
my $id = 10;
my $date = "20/12/02";
my $from = "Fred Foonly";
my $subject = "Autoformatted message";
print form($format, $id, $date, $from, $subject);
```

`Text::Autoformat` pozwala także na wyjątkowo swobodne kontrolowanie dzielenia wyrazów w polach formatu tworzących wieloliniowe bloki. Można stosować różne algorytmy dzielenia, jak choćby `TeX::Hyphen` autorstwa Jana Pazdziory, używanego w pakiecie TeX Donalda Knutha. Główną wadą modułu `Text::Autoformat` jest to, że nie daje takiej kontroli nad nagłówkami i stopkami jak `write`.

Zarówno formaty Perla, jak i `Text::Autoformat` nadają się świetnie do generowania formatowanego wyjścia przypominającego stylem programy tworzone w latach 80. ubiegłego stulecia. Jednak gdy dziś mówimy o formatowaniu tekstu, to mamy na myśli raczej formularze czy listy seryjne. Przejdźmy więc do modułów lepiej dostosowanych do obsługi szablonów w takim właśnie stylu.

Text::Template

`Text::Template`, autorstwa Marka-Jasona Dominusa, to już de facto standard wśród systemów tworzenia szablonów dla zwykłego tekstu. Stosowany w nim język wzorców jest naprawdę prosty — wszystko, co umieścimy pomiędzy znakami `{` i `}`, zostanie przetworzone przez Perla, cała reszta będzie niezmieniona.

Jest to moduł zorientowany obiektowo — najpierw tworzy się obiekt szablonu z pliku, uchwytu do pliku lub łańcucha, a następnie się go wypełnia:

```
use Text::Template;
my $template = Text::Template->new(TYPE => "FILE",
                                   SOURCE => "email.tmpl");

my $output = $template->fill_in();
```

Załóżmy więc, że mamy następujący szablon:

```
Drogi {$who},
    Dziękuję Ci za moduł {modulename}, który pozwolił mi oszczedzić około
{$hours) godzin pracy w tym roku. Dzięki temu miałem możliwość rozegrania
{ int($hours*2.4) } potyczek w go i bardzo cenię sobie to, że nie straciłem
tego czasu, próżnując na IRC-u.

Z poważaniem,
Simon
```

Tworzymy obiekt szablonu, potrzebne zmienne, a następnie przetwarzamy szablon:

```
use Text::Template;
my $template = Text::Template->new(TYPE => "FILE",
                                   SOURCE => "email.tmpl");

$who = "Mark";
$modulename = "Text::Template";
$hours = 15;
print $template->fill_in();
```

Wynik będzie wyglądał tak:

```
Drogi Mark,
    Dziękuję Ci za moduł Text::Template, który pozwolił mi oszczedzić około
15 godzin pracy w tym roku. Dzięki temu miałem możliwość rozegrania
36 potyczek w go i bardzo cenię sobie to, że nie straciłem
tego czasu, próżnując na IRC-u.

Z poważaniem,
Simon
```

Zauważmy, że podstawiane zmienne — `$who`, `$modulename` i tak dalej — nie są zmiennymi `my`. Gdy się nad tym odrobinę dłużej zastanowić, staje się to oczywiste — zmienne `my` nie należałyby przecież do zakresu widoczności obowiązującego wewnątrz `Text::Template` i jako takie nie były w nim widoczne. Ma to nieco nieprzyjemne implikacje: `Text::Template` ma dostęp do zmiennych pakietowych i trzeba się nieco nagimnastykować, by bez przeszkód zastosować `use strict`.

Są dwa sposoby rozwiązania tego problemu. Pierwszy jest bardzo prosty — umieścić zmienne podstawiane w jeszcze jednym, zupełnie innym pakiecie:

```
use Text::Template;
my $template = Text::Template->new(TYPE => "FILE",
                                   SOURCE => "email.tmpl");

$Temp::who = "Mark";
$Temp:modulename = "Text::Template";
$Temp::hours = 15;
print $template->fill_in(PACKAGE => "Temp");
```

To wygląda już lepiej, ale nadal nie usatysfakcjonuje tych, którzy unikają zmiennych globalnych jak ognia. Problem ten można obejść, przekazując przenośną tablicę symboli — tablicę asocjacyjną:

```
use Text::Template;
my $template = Text::Template->new(TYPE => "FILE",
                                   SOURCE => "email.tmpl");

print $template->fill_in(HASH => {
    who = "Mark",
    modulename => "Text::Template",
    hours => 15
});
```

Pętle, tablice zwykłe i asocjacyjne

To tyle o prostych szablonach. Ponieważ `Text::Template` wykonuje wszystko, co umieścimy pomiędzy nawiasami klamrowymi jako pełnoprawny kod Perla, w szablonach możemy robić o wiele więcej. Przypuścimy na przykład, że przygotowujemy ofertę wykonania pewnych prac projektowych:

```
$client = "Acme Motorhomes and Eugenics Ltd.";
%jobs =
   ("Zaprojektowanie nowego logo" => 450.00,
    "Papier firmowy" => 300.00,
    "Przeprojektowanie strony WWW"       => 900.00,
    "Inne wydatki" => 33.75
   );
```

Możemy przygotować odpowiedni szablon, który wyręczy nas w pracy (oczywiście w pracy nad przygotowaniem oferty, nie nad zleceniem, o które się staramy):

```
{my $total=0; ''}
Do ($client}:

Dziękujemy za skorzystanie z usług Fungly Foobar Design Associates.
Oto zestawienie kosztów prac wykonanych w ramach otrzymanego zlecenia:

{
  while (my ($work, $price) = each %jobs) {
      $OUT .= $work . (" " x (50 - length $work)). sprintf("£%6.2f", $price). "\n";
      $total += $price;
  }
}

Koszt całkowity                                    {sprintf "£%6.2f",$total}

Termin płatności: 30 dni.

Z wyrażami szacunku,
Fungly Foobar
```

Co się tutaj dzieje? Najpierw tworzymy zmienną prywatną w szablonie, `$total`, którą ustawiamy wstępnie na wartość zero. Ponieważ jednak nie chcemy, aby na samym początku naszego wzorca pojawiało się 0, zapewniamy, że nasza wstawka zwróci wartość `''`, tym samym nie dodając niczego do tekstu wyjściowego. To często stosowana, przydatna sztuczka.

Następnie przechodzimy w pętli po wszystkich elementach tablicy asocjacyjnej `%jobs`. Poszczególne ceny z łatwością dodamy do zmiennej `%total`, ale chcielibyśmy przy tym dodawać wiersz do szablonu dla każdej pozycji. Chcielibyśmy wyrazić mniej więcej coś takiego:

```
{
  while (my ($work, $price) = each %jobs) {
}

{$work}                                                  £{$price}

{
      $total += $price;
  }
}
```

Jednak `Text::Template` nie działa w ten sposób: każda wstawka musi być niezależnym, poprawnym składniowo fragmentem Perla. W jaki więc sposób wpisać do szablonu wiele linii? Do tego służy magiczna zmienna `$OUT`. Użycie tej zmiennej powoduje, że jej zawartość zostanie uznana za wynik działania wstawki z kodem. Przy każdej iteracji pętli dodajemy do tej zmiennej odpowiedni tekst, który na końcu zostanie wstawiony do szablonu jako całość.

Zabezpieczenia i wykrywanie błędów

Jedną z zalet stosowania szablonów jest to, że części aplikacji niezwiązane bezpośrednio z programowaniem — wygląd stron HTML, treści listów przy korespondencji seryjnej i tym podobne — można zlecić osobom niekoniecznie umiejącym programować. Natomiast jedną z wad rozbudowanych systemów obsługi szablonów, takich jak `Text::Template`, jest to, że niewiele trzeba, by doprowadzić do rozwinięcia wzorca w postać `{ system("rm -rf /") }`. Po pierwszym takim przypadku, Ty, a przy okazji może ktoś jeszcze, będziesz zmuszony szukać nowej pracy. Musi więc istnieć jakiś sposób zabezpieczenia szablonów przed tego typu przygodami.

`Text::Template` oferuje dwa sposoby ochrony przed takimi współpracownikami… ups, miałem na myśli wpadkami. Pierwszy to zwykły mechanizm „skazy”. W trybie skazy Perl odmówi uruchomienia szablonu z pliku zewnętrznego. Zabezpiecza to przed ludźmi podmieniającymi pliki wzorców, ale w sposób całkowicie uniemożliwiający korzystanie z jakichkolwiek plików z szablonami; zamiast tego wszystkie szablony trzeba podawać jako łańcuchy.

Jeżeli mamy zaufanie do konkretnego pliku w systemie, musimy nakazać `Text::Template` przyjęcie go; służy do tego opcja `UNTAINT`:

```
my $template = new Text::Template (TYPE => "FILE",
                                   UNTAINT => 1,
                                   SOURCE => $filename);
```

Teraz będzie już można skorzystać z szablonu w pliku `$filename`, oczywiście o ile plik ten pomyślnie przejdzie sprawdzanie w trybie skazy.

Drugi sposób zabezpieczeń oferuje większy stopień szczegółowości; opcja `SAFE` pozwala na wskazanie zmiennej klasy `Safe` ograniczającej operacje wykonywane we wstawkach Perla:

```
my $compartment = new Safe; # Domyślny zestaw operacji jest bezpieczny
$text = $template->fill_in(SAFE => $compatment);
```

Osoby szczególnie przewrażliwione w zakresie bezpieczeństwa z pewnością będą chciały osiągnąć nieco więcej niż tylko ograniczenie się do domyślnego zestawu operacji zastrzeżonych.

Co będzie, gdy nie uda się coś innego? Lepiej przecież, by aplikacja nie kończyła się niespodziewanie po wystąpieniu błędu w kodzie Perla wstawionym w szablon czy zgłoszeniu błędu dzielenia przez zero. `Text::Template` domyślnie wyłapuje błędy przy wywołaniach `eval`, jednak czasami chcielibyśmy uzyskać nieco większą kontrolę nad procesem obsługi błędów. Do tego właśnie służy opcja `BROKEN`.

Dzięki opcji `BROKEN` można określić procedurę, która będzie wywoływana zawsze wtedy, gdy we wstawionym kodzie wykryty zostanie błąd składniowy czy dowolna inna nieprawidłowość. Bez opcji `BROKEN` do wyjściowego tekstu wstawiane są standardowe komunikaty o błędzie, na przykład:

```
Szanowny Panie(Pani) Program fragment delivered error ''syntax error at template
line 1'',
```

Określając procedurę `BROKEN` zyskujemy większą kontrolę nad tym, co będzie wstawione do tekstu w takiej sytuacji. W większości przypadków najsensowniejszym zachowaniem po wykryciu błędu będzie po prostu całkowite przerwanie przetwarzania szablonu. Osiągniemy to, zwracając wartość `undef` z procedury wskazanej przy opcji `BROKEN`. W takiej sytuacji `Text::Template` zwróci tyle tekstu wyjściowego, ile udało mu się do tej pory przetworzyć.

Oczywiście musi istnieć sposób wskazania, czy przetwarzanie szablonu zakończyło się powodzeniem czy może zostało przerwane przez procedurę `BROKEN`. Dokonuje się tego poprzez zwrotny argument `BROKEN_ARG`. Przekazanie `BROKEN_ARG` konstruktorowi szablonu spowoduje, że będzie on przekazany do funkcji `BROKEN`[2]. Dzięki temu możemy osiągnąć na przykład coś takiego:

[2] Udostępnienie argumentu definiowanego przez użytkownika to świetny sposób na rozszerzanie możliwości funkcji zwrotnych.

```
my $succeeded = 1;

$template->fill_in(BROKEN => \&broken_sub, BROKEN_ARG => \$succeeded);

if (!$suceeded) {
    die "Nie udało się wypełnić szablonu...";
}

sub broken_sub {
    my %params = @_;
    ${$params{arg}} = 0;
    undef;
}
```

Jak widać, funkcja zwrotna wskazana przez `BROKEN` wywoływana jest z argumentem wejściowym w postaci tablicy asocjacyjnej; argument określony przez `BROKEN_ARG` to element `arg` tej tablicy. W tym przypadku jest to po prostu referencja do zmiennej `$succeeded`; wyłuskujemy tę referencję i ustawiamy zmienną na zero, informując tym samym o wystąpieniu błędu, i zwracamy wartość `undef`, przerywając dalsze przetwarzanie szablonu.

Jeżeli ktoś ma pomysł, co zrobić z szablonem po wykryciu błędu, to może poznać treść problematycznego fragmentu kodu. `Text::Template` udostępnia taką wstawkę w elemencie `text` tablicy asocjacyjnej; jak do tej pory nie wymyśliłem jednak dla tego rozwiązania żadnego sensownego zastosowania. Pozostałe składniki tej tablicy pomocne przy wskazywaniu przyczyny błędu to: `line`, informacja o numerze linii szablonu, w której wystąpił błąd, i `error`, czyli wartość `$@` wskazująca zgłoszony błąd.

Sztuczki w Text::Template

Oznaczanie fragmentów kodu za pomocą `{` i `}` nie stanowi problemu przy większości zastosowań `Text::Template` — generowaniu seryjnej korespondencji czy e-maili. Jednak przy generowaniu tekstu, w którym znaki `{` i `}` są znaczące i występują często — na przykład stron HTML zawierających skrypty JavaScriptu czy znaczniki w TEX-u, staje się to niewygodne.

W takiej sytuacji rozwiązaniem może być poprzedzenie tych nawiasów klamrowych, które nie mają być przetwarzane jako wstawki Perla, lewym ukośnikiem:

```
if (browser == "Opera") \{
 ...
\}
```

Jak zauważył jeden z użytkowników, takie rozwiązanie jest uciążliwe przy generowaniu tekstów w TEX-u, w którym zarówno lewe ukośniki, jak i nawiasy klamrowe mają swoje znaczenie:

```
\\textit\{ {$title} \} \\dotfill \\textbf\{ \\${$cost} \}
```

Dużo zgrabniejszym rozwiązaniem byłoby wskazanie innych symboli ograniczających, unikając tym samym konieczności wstawiania co chwilę ukośników:

```
\textit{ [[[ $title ]]] } \dotfill \textbf{ [[[ $cost ]]] }
```

Tak jest o wiele przejrzyściej!

Aby osiągnąć to w `Text::Template` wystarczy skorzystać z opcji `DELIMITERS` czy to w konstruktorze, czy w metodzie `fill_in`:

```
print $template->fill_in(DELIMITERS => [ '[[[', ']]]' ]);
```

To rozwiązanie działa szybciej niż przy zastosowaniu ograniczników standardowych, ponieważ nie wymaga żadnego specjalnego przetwarzania ukośników, przy czym jest chyba oczywiste, że należy zapewnić, by wybrane symbole ograniczające nie wystąpiły nigdzie w szablonie jako literalny tekst.

Jeżeli z tego rozwiązania z jakichś powodów nie można skorzystać, to istnieje jeszcze jeden sposób, sugerowany przez Marka: anulowanie znaczenia nawiasów klamrowych poprzez zastosowanie wbudowanych w Perlu operatorów cytowania. Fragment `{ q{ Witaj! } }` zwróci łańcuch „Witaj!", który zostanie wstawiony do tekstu wyjściowego z szablonu. A więc innym rozwiązaniem na wstawienie literalnego tekstu bez konieczności anulowania znaczenia nawiasów klamrowych jest dodanie kolejnych nawiasów klamrowych!

```
{ q{

 if (browser == "Opera") { … }

} }
```

Kolejny problem wiąże się z tym, że od ciągłego wpisywania:

```
my $template = new Text::Template(...);
$template->fill_in();
```

mogą odpaść palce. Styl obiektowy jest niezastąpiony przy pracy z szablonem, który będzie wypełniany setki razy — na przykład formularzem listu — ale nie wtedy, gdy chcemy wypełnić szablon tylko raz. W takich przypadkach możemy wyeksportować z `Text::Template` procedurę `fill_in_file`. Wykonuje ona przygotowanie szablonu i jego wypełnienie jednym ruchem:

```
use Text::Template qw(fill_in_file);

print fill_in_file("email.tmpl", PACKAGE => "Q", …);
```

Zauważmy, że funkcję tę trzeba jawnie zaimportować.

HTML::Template

Formatowanie HTML-a różni się nieco od formatowania zwykłego tekstu — występują przy tym dwie główne szkoły. Pierwsze podejście, wykorzystywane w `HTML::Template`, jest podobne do metody, którą poznaliśmy przy okazji omawiania `Text::Template` — szablon jest gdzieś zapisany, a program w Perlu odczytuje go i wypełnia. Drugie podejście reprezentowane jest przez `HTML::Mason`, który omówimy jako następny. Tutaj jest na odwrót — nie uruchamia się programu, który zwracałby tekst w HTML-u; zamiast tego tworzy się plik w tym formacie, który zawiera wstawione fragmenty kodu Perla i wykonuje się go.

Aby porównać oba te rozwiązania, utworzymy tę samą aplikację za pomocą `HTML::Template`, `HTML::Mason` i `Template Toolkit` — program zbierający z różnych stron WWW nagłówki wiadomości w formacie RSS (ang. *Remote Site Summary*) i wyświetlający je na jednej stronie (podobnie jak w Amphetadesku, *http://www.disobey.com/amphetadesk/* czy Meerkacie wydawnictwa O'Reilly, *http://www.oreillynet.com/meerkat/*). RSS to format oparty na XML-u, w którym zapisywane są szczegółowe informacje o poszczególnych elementach strony; generalnie wykorzystuje się go do rozpowszechniania skrótów wiadomości z portali informacyjnych.

Zmienne i wyrażenia warunkowe

Wcześniej jednak dokonamy krótkiego przeglądu możliwości `HTML::Template`, zobaczymy, jak przekazywać mu wartości i jak uzyskać wynikowy dokument HTML.

Podobnie jak w `Text::Template` szablony umieszczane są w osobnych plikach. Szablony obsługiwane przez `HTML::Template` to zwykłe pliki HTML z kilkoma dodatkowymi znacznikami specjalnymi. Najważniejszy z nich to `<TMPL_VAR>`, który jest zastępowany zawartością zmiennej Perla. Oto przykładowa, bardzo prosta strona:

```
<html>
   <head><title>Informacje o: <TMPL_VAR NAME=PRODUCT></title></head>
   <body>
      <h1> <TMPL_VAR NAME=PRODUCT> </h1>
      <div class="desc">
           <TMPL_VAR NAME=DESCRIPTION>
      </div>
      <p class="price">Cena: $<TMPL_VAR NAME=PRICE></p>
      <hr />
      <p>Cena z dnia <TMP_VAR NAME=DATE></p>
   </body>
</html>
```

Po wypełnieniu odpowiednimi wartościami powinniśmy uzyskać mniej więcej coś takiego:

```
<html>
   <head><title>Infomracje o: Największa enchilada na świecie</title></head>
   <body>
      <h1> Największa enchilada na świecie </h1>
      <div class="desc">
           Odkryta niedawno w lasach Meksyku....
      </div>
      <p> class="price">Cena: $1504.39</p>
      <hr />
      <p>Cena z dnia 15:18 PST, 7 Mar 2005</p>
   </body>
</html>
```

Aby wypełnić szablon tymi wartościami, trzeba napisać krótki program CGI, na przykład taki:

```
use strict;
use HTML::Template;

my $template = HTML::Template->new(filename => "catalogue.tmpl");

$template->param( PRODUCT     => "Największa enchilada na świecie" );
$template->param( DESCRIPTION => $description );
$template->param( PRICE       => 1504.39 );
$template->param( DATE        => format_date(localtime) );

print "Content-Type: text/html\n\n", $template->output;
```

I znów, podobnie jak `Text::Template`, program sterujący jest bardzo prosty — załaduj szablon, wypełnij go i wyświetl. To jednak nie wszystko, co możemy osiągnąć za pomocą tego języka wzorców — istnieją jeszcze inne znaczniki, zapewniające większą elastyczność.

Przypuśćmy na przykład, że dysponujemy zdjęciem największej enchilady na świecie — to z pewnością jest coś wartego pokazania na stronie WWW. Jednak nie mamy zdjęć wszystkich produktów znajdujących się w naszej bazie danych; chcielibyśmy więc, by obrazki pokazy-

wały się tylko przy tych pozycjach, których zdjęcia rzeczywiście mamy. Możemy więc dodać do szablonu coś takiego:

```
<TMPL_IF NAME=PICTURE_URL>
<div class="photo">
    <img src="<TMP_VALUE NAME=PICTURE_URL>" />
</div>
</TMPL_IF>
```

Oznacza to, że jeśli PICTURE_URL będzie miało wartość logicznie prawdziwą — to znaczy, jeśli rzeczywiście przypiszemy mu rzeczywisty URL — to wstawimy znacznik <div> dla zdjęcia. Ponieważ znaczniki <TMPL_...> nie są de facto prawdziwymi znacznikami HTML, a jedynie przetwarzanymi przez HTML::Template, nic nie stoi na przeszkodzie umieszczania ich wewnątrz innych znaczników HTML, tak jak zrobiliśmy to tutaj z <img scr="...">.

Oczywiście jeśli nie mamy odpowiedniego zdjęcia, możemy w jego miejsce wstawić zdjęcie zastępcze — efekt ten osiągniemy, korzystając choćby z pseudoznacznika <TMPL_ELSE>:

```
<div class="photo">
<TMPL_IF NAME=PICTURE_URL>
   <img erc="<TMP_VALUE NAME=PICTURE_YURL>" />
<TMPL_ELSE>
   <img src="http://www.mysite.com/images/noimage.gif" />
</TMPL_IF>
</div>
```

Zauważmy, że o ile każdemu <TMPL_IF> musi odpowiadać znacznik </TMPL_IF>, to w przypadku <TMPL_ELSE> nie ma znacznika kończącego.

Ale być może przedstawione rozwiązanie to niepotrzebne komplikowanie sprawy, przecież w tym przykładzie wystarczyłoby zapewnić domyślną wartość dla PICTURE_URL, a to możemy osiągnąć, stosując atrybut DEFAULT w <TMPL_VALUE>:

```
<div class="photo">
   <img src="
<TMPL_VALUE NAME=PICTURE_URL
            DEFAULT="http://www.mysite.com/images/noimage.gif">
   "/>
</div>
```

Sprawdzanie poprawności

Wiele osób ma, niebezpodstawne, obawy co do wpływu takiego niewybrednego nadużywania SGML-a na mechanizmy sprawdzające poprawność kodu szablonów (choć z drugiej strony, wiele osób, niestety, w ogóle *nie* dba o walidację HTML-a). Co więcej, użytkownicy edytorów korzystających przy sprawdzaniu poprawności z DTD mogą się zastanawiać, w jaki sposób bezproblemowo umieszczać takie pseudoznaczniki w swoich dokumentach.

HTML::Template oferuje rozwiązanie tego problemu; zamiast zapisywać te znaczniki tak, jakby były zwykłymi znacznikami HTML-a, można je zapisywać w formie komentarzy, na przykład:

```
<!-- TMPL_IF NAME=PICTURE_URL -->
<div class="photo">
   <img src="<!-- TMP_VALUE NAME=PICTURE_URL -->" />
</div>
<!-- /TMPL_IF -->
```

Pętle

Jeżeli nasz przykład z RSS ma mieć szanse zadziałania, to musimy się dowiedzieć, w jaki sposób powtórzyć pewne operacje w pętli dla wielu elementów — skrótów wiadomości w naszym zestawieniu. W tym celu `HTML::Template` dostarcza znacznik `<TMPL_LOOP>`, który umożliwia potraktowanie zmiennej jak tablicy. Na przykład poniższy kod:

```
<ul>
<TMPL_LOOP NAME=STORIES>
    <li> Źródło <TMPL_VAR NAME=FEED_NAME>: <TMPL_VAR NAME=STORY_NAME> </li>
</TMPL_LOOP>
</ul>
```

po dostarczeniu odpowiednich struktur danych spowoduje powtórzenie komunikatu dla wszystkich elementów w tablicy `STORIES`, co prowadzi do następującego efektu końcowego:

```
<ul>

    <li> Źródło Slashdot: NASA Finds Monkeys on Mars </li>

    <li> Źródło use.perl: Perl 6 Release Predicted for 2013 </li>

</ul>
```

Trik oparto na tym, że przekazana tablica zawierała tablice asocjacyjne, a każda z nich zawierała odpowiednie nazwy zmiennych:

```
$template->param(STORIES => [
 { FEED_NAME => "Slashdot", STORY_NAME => "Nasa Finds Monkeys on Mars" },
 { FEED_NAME => "use.perl", STORY_NAME => "Perl 6 Release Predicted for 2013" }
]);
```

Gromadzenie wiadomości RSS

Dysponując taką wiedzą, bez problemu utworzymy aplikację zbierającą wiadomości RSS; najpierw zgromadzimy wszystkie interesujące nas komunikaty, następnie je posortujemy i wstawimy do struktury danych odpowiedniej do iteracji za pomocą `<TMPL_LOOP>`.

Wiadomości RSS pobierzemy, korzystając z modułów `LWP` i `XML::RSS`. W naszym przykładzie założymy, że dysponujemy wystarczająco dużą ilością pamięci podręcznej, dzięki czemu nie będzie problemów z cyklicznym pobieraniem wiadomości; w prawdziwym programie praktyczniejszym rozwiązaniem byłoby zapisywanie odebranych fragmentów w formacie XML do plików o stałych nazwach i przed ponownym pobraniem wiadomości z Sieci sprawdzanie, jak długo pliki znajdują się już na dysku.

Tworzenie naszej aplikacji gromadzącej rozpoczniemy od napisania małego programu w Perlu pobierającego i porządkującego wiadomości:

```
#!/usr/bin/perl

use LWP::Simple;
use XML::RSS;
my @stories;
while (<DATA>) {
    chomp;
    my $xml = get($_) or next;
    my $rss = XML::RSS->new;
    eval { $rss->parse($xml) }; next if $@;
```

```
    for my $item (@{$rss->{'items'}}) {
        push @stories, {
             FEED_NAME  => $rss->channel->{'title'},
             FEED_URL   => $rss->channel->{'link'},

             STORY_NAME => $item->{'title'},
             STORY_URL  => $item->{'link'},
             STORY_DESC => $item->{'description'}
             STORY_DATE => $item->{'dc'}->{'date'}
        }
    }
}

@stories = sort { $b->{STORY_DATE} cmp $a->{STORY_DATE} } @stories;

__DATA__
http://slashdot.org/slashdot.rss
http://use.perl.org/perl-news-short.rdf
http://www.theregister.co.uk/tonys/slashdot.rdf
http://blog.simon-cozens.org/blosxom.cgi/xml
http://www.oreillynet.com/~rael/index.rss
```

Musimy teraz zaprojektować szablon, do którego przekażemy listę wiadomości. Osobiście kiepski ze mnie projektant stron, dlatego tak bardzo lubię szablony. Wystarczy że utworzę coś, co z grubsza odda zarys spodziewanego efektu, i przekażę to komuś z darem tworzenia ciekawej warstwy prezentacyjnej. Oto taki prosty i szybki wzorzec:

```
<html>
  <head> <title> Wiadomości <title> </head>
  <body>
     <h1> Wiadomości zebrane o <TMPL_VAR TIME> </h1>

     <TMPL_LOOP STORIES>
        <table border="1">
          <tr>
           <td>
             <h2>
               <a href="<TMPL_VAR STORY_URL>"> <TMPL_VAR STORY_NAME> </a>
             </h2>
             <p> <TMPL_VAR STORY_DESC> </p>
             <hr>
             <p> <i> Źródło
                 <a href='<TMPL_VAR FEED_URL>"> <TMPL_VAR FEED_NAME> </a>
             </i> </p>
           </td>
          </tr>
        </table>
     </TMPL_LOOP>
  </body>
</html>
```

(Zauważmy, że używam krótkich form przy zapisywaniu pseudoznaczników; wszędzie tam, gdzie nie prowadzi to do niejednoznaczności, można z powodzeniem używać zapisu `NAZWA_ZMIENNEJ` zamiast `NAME=NAZWA_ZMIENNEJ`.)

Wystarczy jeszcze kilka drobnych zmian w programie sterującym sprowadzających się do przekazania wygenerowanej tablicy do `HTML::Template`:

```
#!/usr/bin/perl

use LWP::Simple;
use XML::RSS;
```

```
use HTML::Template;

my @stories;

while (<DATA>) {
    chomp;
    my $xml = get($_) or next;
    my $rss = XML::RSS->new;
    eval { $rss->parse($xml) }; next if $@;
    for my $item (@{$rss->{'items'}}) {
        push @stories, {
             FEED_NAME  => $rss->channel->{'title'},
             FEED_URL   => $rss->channel->{'link'},

             STORY_NAME => $item->{'title'},
             STORY_URL  => $item->{'link'},
             STORY_DESC => $item->{'description'},
             STORY_DATE => $item->{'dc'}->{'date'}
        }
    }
}

my $template = HTML::Template->new(filename => "aggregator.tmpl");

$template->param(STORIES => [
    sort {$b->{STORY_DATE} cmp $a->{STORY_DATE} } @stories
                    ] );
$template->param( TIME=> scalar localtime );

delete $_->{STORY_DATE} for @stories;

print "Content-Type: text/html\n\n", $template->output;

__DATA__
http://blog.simon-cozens.org/blosxom.cgi/xml
http://slashdot.org/slashdot.rss
http://use.perl.org/perl-news-short.rdf
http://theregister.co.uk/tonys/slashdot.rdf
http://www.oreillynet.com/~rael/index.rss
```

Po użyciu `STORY_DATE` do określania kolejności wiadomości musimy ją usunąć, ponieważ `HTML::Template` „nie lubi”, gdy w pętli pojawia się zmienna, z której nie korzystamy w szablonie.

Wystarczy umieścić to na serwerze obsługującym CGI, a otrzymamy tani i sprawny klon Amphetadesku.

HTML::Mason

Jedną z największych wad `HTML::Template` jest to, że w pewnym stopniu zmusza nas do przeplatania ze sobą części prezentacyjnej i logiki programu, a tego przecież powinniśmy uniknąć dzięki zastosowaniu szablonów. Przykładowo, ostatni przykład dość trudno analizować, bo znaczniki HTML-a i zmiennych mieszają się ze sobą, zacierając znaczenie poszczególnych elementów. Lepszy byłby więc system, który umożliwiałby jeszcze większą abstrakcję poszczególnych składników funkcjonalnych szablonu — w tej roli świetnie sprawdza się `HTML::Mason`.

Jak już wspomniałem, `HTML::Mason` to system obsługi szablonów działający na odwrotnej zasadzie niż pozostałe. Określenie system obsługi szablonów pasuje do niego tak samo dobrze

jak system abstrakcji komponentów służących do budowania stron HTML z małych elementów logicznych, nadających się do wielokrotnego użytku. Zanim przejdziemy do tworzenia aplikacji zabierającej komunikaty RSS, przyjrzymy się, jak korzystać z `HTML::Mason`.

Podstawowe komponenty

W Masonie wszystko jest komponentem. Oto prosty przykład wykorzystania takich komponentów. Załóżmy, że mamy trzy pliki: *test.html* (przykład 3.1), *Header* (przykład 3.2) i *Footer* (przykład 3.3).

Przykład 3.1. Plik test.html

```
<& /Header &>
<p>
  Hello World
</p>
<& /Footer &>
```

Przykład 3.2. Komponent Header

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
      <title>Some Web Application</title>
      <link rel=stylesheet type="text/css" href="nt.css">
  </head>

<body>
```

Przykład 3.3. Komponent Footer

```
    <hr>
    <div class="footer">
      <address>
         <a href="mailto:webmaster@yourcompany.com">webmaster@yourcompany.com<'a>
      </address>
    </div>
  </body>
</html>
```

`HTML::Mason` buduje stronę, dołączając komponenty wskazane przez znaczniki `<&` i `&>`. Przy tworzeniu pliku *test.html* Mason najpierw wstawi treść komponentu *Header*, umieszczonego w bieżącym katalogu generowanego dokumentu, potem pozostały kod HTML i na końcu komponent *Footer*.

Komponenty mogą odwoływać się do innych komponentów. Jak do tej pory nie zrobiliśmy nic ponad dołączanie plików obsługiwane przez serwer.

Podstawowe mechanizmy dynamiczne

Gdzie tu więc szablony? Do stron Masona można je dodać na trzy sposoby. Oto pierwszy z nich, mała modyfikacja komponentu *Footer*.

```
<hr>
<div class="footer">
  <address>
     <a href="mailto:webmaster@yourcompany.com">webmaster@yourcompany.com</a>
 </address>
```

```
      Generated: <%scalar localtime %>
    </div>
  </body>
</html>
```

Po umieszczeniu kodu Perla pomiędzy znacznikami `<% ... %>` wynik przetworzonego wyrażenia zostanie wstawiony do wynikowej strony HTML.

Wiemy więc, jak wstawić proste wyrażenia, ale co z rzeczywistą logiką Perla? I na to jest sposób: pojedynczy znak `%` umieszczony na początku linii powoduje, że Mason interpretuje całą linię jako kod Perla. Dzięki temu można uzyskać dostęp do zawartości tablicy asocjacyjnej w sposób zaprezentowany w przykładzie 3.4.

Przykład 3.4. Komponent Hashdump

```
<table>
  <tr>
     <th> key </th>
     <th>value</th>
  </tr>

% for (keys %hash) {
   <tr>
     <td> <% $_%> </td>
     <td> <% $hash{$_} %> %> </td>
   </tr>
% }
</table>

<%ARGS>
%hash => undef
</%ARGS>
```

Analizując ten przykład, warto zwrócić uwagę na trzy rzeczy. Po pierwsze na to, w jaki sposób można przeplatać zwykłego HTML-a z logiką, za pomocą składni `% ...`, i obliczanymi wyrażeniami Perla, za pomocą `<% ... %>`. Znak `%` ma specjalne znaczenie tylko na początku linii i jako część znacznika `<% ... %>`; `%` w `%hash` to zwykły zapis w Perlu.

Druga rzecz warta podkreślenia to sposób przekazania wartości tablicy asocjacyjnej do komponentu. Do tego służy sekcja `<%ARGS>` — w niej znajduje się deklaracja argumentów przekazywanych do komponentu. A jak je przekazać? Oto przykład wyrażenia wywołującego komponent Hashdump:

```
% my %foo = ( one => 1, two => 2 );

<& /Hashdump, hash => %foo &>
```

Mieliśmy więc przykład deklaracji zmiennej typu `my` wewnątrz komponentu, przekazania nazwanego parametru do innego komponentu i odbioru tego parametru przez komponent, który z niego korzysta. Jeśli przekażemy do komponentu parametry innego typu niż zadeklarowane w sekcji `<%ARGS>` tego komponentu (w tym przypadku przekazaliśmy akurat tablicę asocjacyjną dla parametru `%hash`), Mason będzie próbował zrobić z tym coś sensownego, ale prościej unikać potencjalnych problemów, przekazując dane właściwego typu.

Bloki Perla

Istnieje jeszcze jeden, ostatni, sposób dokładania logiki Perla do komponentów, jednak w opisywanej formie stosuje się go rzadko. W przypadku długich sekcji kodu Perla niewygodne

staje się umieszczanie na początku każdej linii znaku %. Zamiast tego można umieścić całą taką sekcję wewnątrz bloku <%PERL ... /%PERL>.

W praktyce często spotyka się za to blok <%INIT ... /%INIT>. Można go umieścić w dowolnym miejscu komponentu, przyjęło się jednak zapisywać go na końcu, tak by nie mieszał się z resztą HTML-a. Jednak bez względu na to, gdzie go umieścimy, jego zawartość zostanie wykonana zawsze jako pierwsza, przed całą pozostałą treścią komponentu. Jest to dobre miejsce do deklarowania i inicjalizowania zmiennych (tak przy okazji: Mason wymusza stosowanie `use strict`…) oraz przeprowadzania wszelkich złożonych obliczeń, które mają być wykonane przed rozpoczęciem wyświetlania strony.

Pozostało jeszcze wspomnieć o kolejnym rzadko wykorzystywanym bloku: <%ONCE> ... </%ONCE>. Jest on wykonywany tylko raz na początku. Można o nim myśleć jak o odpowiedniku bloku `BEGIN` Perla.

Program gromadzący wiadomości RSS

Po takim wprowadzeniu najwyższy czas przystąpić do składania naszego zbieracza RSS. Przykład prezentowany w tej sekcji zaczerpnięty jest z kodu, który pisałem na użytek pewnego portalu sieciowego. Warto zaznaczyć, że realizacja projektu zajęła mi około dwóch, trzech godzin. W założeniach miał on obsługiwać logowanie użytkowników, spersonalizowane listy wiadomości, indywidualnie określane zasady ich sortowania i tym podobne. Mimo że nie zrealizowałem w tym czasie wszystkiego, wydaje mi się, że efekt tej niespełna trzygodzinnej pracy jest wart rozważenia w tym miejscu[3].

Na początku zastanówmy się, jaki powinien być układ strony głównej. Moim zdaniem dobrym rozwiązaniem byłby podział na dwie kolumny, tak jak pokazano to na rysunku 3.1. Lewa kolumna będzie zawierała zaproszenie do zalogowania do portalu oraz listę dostępnych typów wiadomości. Poszczególnych kategorie wiadomości umieścimy w oddzielnych folderach reprezentowanych przez katalogi w systemie plików. W prawej kolumnie wyświetlone będą ulubione wiadomości zalogowanego użytkownika, wiadomości z wybranego folderu, o ile któryś z nich został kliknięty, bądź domyślny zestaw wiadomości w każdym innym przypadku.

Przystąpmy więc do tworzenia witryny. Przed wszystkim przygotujemy nagłówek i stopkę, tak by na wstępie pozbyć się nudnych części dokumentu HTML — prezentują je przykłady, odpowiednio, 3.5 i 3.6.

Przykład 3.5. Komponent Header

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html lang="en">

<head>
<title> My Portal </title>
<link rel="stylesheet" type="text/css" href="/stylesheets/portal.css">
</head>
<body class="pagetable">

<img src="/images/portal-logo.gif" id="toplogo">
<h1>My Portal</h1>
```

[3] Oczywiście zachęcam do samodzielnej implementacji wszystkich wymienionych elementów — będzie to świetne ćwiczenie programowania z wykorzystaniem `HTML::Mason`.

Rysunek 3.1. Portal z wiadomościami RSS

Przykład 3.6. Komponent Footer

```
</body>
</html>
```

Teraz wykorzystamy próbkę magicznych możliwości Masona: zamiast ręcznego dodawania nagłówka i stopki oddzielnie do każdej strony użyjemy komponentu *autohandler*, dodawanego automatycznie do wszystkich stron. Jego treść prezentuje przykład 3.7.

Przykład 3.7. Komponent Autohandler

```
<& /header &>
<% $m->call_next %>
<& /footer &>
```

Strony przetwarzane przez Masona są w tle poddawane działaniu pewnych *komponentów obsługujących* (ang. *handlers*), będących pozostałością po modułach tego typu z `mod_perl` Apache. I w rzeczywistości zmienna `$m` użyta w tym przykładzie to request object Masona, będący odpowiednikiem request objectu z Apache[4].

Mason wywołuje w pierwszej kolejności automatyczne komponenty obsługujące (ang. *autohandlers*), obsługujące wszystkie żądania; następne w kolejności są dhandlery (ang. *dhandlers*), obsługujące poszczególne URI, i na końcu wywoływane są zwykłe handlery Masona obsługujące przetwarzanie żądanej strony. Powyższy przykład prezentuje najprostszą, a przy tym najczęstszą postać automatycznego komponentu obsługującego: wywołanie komponentu nagłówkowego (*header*), przekazanie wywołania do następnego handlera w łańcuchu handlerów Masona i ostatecznie wywołanie komponentu stopki (*footer*). W ten sposób zapewniliśmy, że na każdej stronie pojawią się nagłówek i stopka.

[4] Do request objectu Apache'a można się odwołać w Masonie za pośrednictwem zmiennej `$r`.

Zastanówmy się teraz, jaka powinna być zawartość pliku *index*. Jak wspomniałem, skorzystamy z układu dwukolumnowego, tak jak w przykładzie 3.8.

Przykład 3.8. Plik index.html

```
<table>

<tr>
<td valign="top">
<& /LoginBox &>
<& /Directories &>

<%INIT>
$open = ($open = ~ /(\w+)/) ? $1 : '';
</%INIT>

</td>
<td width=4> </td>

<td width='100%'>

%# Czy jesteśmy zalogowani?
% if (0) {
<& /LoggedInPane &>
%} elsif ($open) {
<& /DirectoryPane, open => $open &>
%} else {
<& /StandardPane &>
%}
</td>

</table>

<%ARGS>
$open => undef
</%ARGS>
```

Zgodnie z obietnicą w lewej kolumnie umieściliśmy obszar logowania i katalog wiadomości. Prawa strona może przyjąć jeden z trzech stanów: wygląd dla użytkownika zalogowanego (na razie otoczony niespełnionym warunkiem w `if`, ponieważ obsługa użytkowników zostanie dodana jako przyszłe rozszerzenie), zawartość konkretnego katalogu otwartego przez użytkownika lub domyślny wygląd dla odwiedzających główną stronę serwisu[5].

A co z wartością zmiennej `%open`? Komponenty w Masonie mogą przyjmować argumenty albo poprzez CGI, albo przekazywane z innych komponentów. W tym przypadku komponent najwyższy w hierarchii, *index.html*, otrzyma parametry poprzez CGI — oznacza to, że wywołanie URL-a w postaci *http://www.oursite.com/rss/index.html?open= News* spowoduje ustawienie zmiennej `$open` na wartość `News`. Komponent *DirectoryPane* otrzymuje swój argument z *index.html* — przekazujemy mu uzyskaną wartość `$open`.

Ponieważ `$open` wskazywać będzie później na określony katalog w serwerze WWW, musimy uwiarygodnić jego poprawność, aby uniknąć ataków związanych z przeglądaniem katalogów przeprowadzanych przez przekazywanie zapytań w postaci `open=../../..`. W tym celu posługujemy się blokiem `<% INIT %>`, w którym zastępujemy przekazany parametr pierwszym

[5] Wynika z tego, że wszystkie żądania będą przechodziły przez *index.html* (co często się zdarza) — moglibyśmy więc umieścić w nim bezpośrednio także nagłówek i stopkę, jednak wykorzystanie autohandlera jest bardziej przejrzyste i zgodne z przyjętymi konwencjami kodowania.

słowem z odebranego łańcucha. Jeżeli nie pojawią się w nim żadne litery, ustawiamy wartość parametru na łańcuch pusty, dzięki czemu pozostała część kodu potraktuje go tak, jakby nie został wybrany żaden katalog.

Nasza strona będzie składała się z wielu wydzielonych obszarów o różnych nazwach i kolorach, dlatego utworzymy kilka komponentów ułatwiających rysowanie ramek. Każda ramka będzie miała definiowany przez użytkownika kolor, tytuł i opcjonalne łącze tytułowe. Jak uczy doświadczenie, najlepiej jest rozdzielić tworzenie ramki na komponenty rozpoczynający i kończący ramkę. Komponent rozpoczynający, przedstawiony w przykładzie 3.9, tworzy tabelę wewnątrz tabeli.

Przykład 3.9. Komponent BoxTop

```
<table bgcolor="#777777" cellspacing=0 border=0 cellpadding=0>
<tr><td> rowspan=2></td>
<td valign=middle align=left bgcolor="<%$color%>">
 
<font size=-1 color="#ffffff">
<b>
<% $title_href && "<a href=\"$title_href\">"|n%>
<%$title |n %>
<% $title_href && "</a>" |n %>
</b></font></td>
<td rowspan=2> </td></tr>
<tr><td colspan=2 bgcolor="#eeeeee" valign=top align=left width=100%>
<table cellpadding=2 width=100%><tr><td>

<%ARGS>
$title_href => undef>
$title => undef

$color => "#000099"
</%ARGS>
```

Zwróćmy uwagę na dyrektywę `|n` pojawiającą się przy końcu niektórych sekcji kodu Perla. Ma ona za zadanie wyłączenie stosowanego domyślnie w Masonie przekształcania znaków na kody HTML-a. Przykładowo, przekazując wartość dla zmiennej `$title_href`, oczekujemy, że wiersz:

```
<% $title_href && "</a>" %>
```

doprowadzi do wyniku zawierającego ciąg `</a>`. Jednak Mason będzie się starał zastąpić znaki kodami, co doprowadzi do postaci `</a>` — musimy więc to wyłączyć.

Kod domykający ramkę, pokazany w przykładzie 3.10, jest dużo prostszy i sprowadza się właściwie do zakończenia rozpoczętej tabeli.

Przykład 3.10. Komponent BoxEnd

```
</td></tr></table>

</td></tr>
<tr><td colspan=4> </td></tr>
</table>
```

Jako przykład zastosowania tych komponentów utworzymy ramkę logowania, co demonstruje komponent z przykładu 3.11.

Przykład 3.11. Komponent LoginBox

```
<% BoxTop, title=>"Login" &>
<small>Log in to Your portal:</small><br/>
<form>
<ul>
<li> Barcode: <input name="barcode">

<li> Password: <input name="password">

</ul>

</form>
<& BoxEnd &>
```

Po przetworzeniu przez Masona otrzymamy w wyniku następujący kod HTML:

```
<table bgcolor="#777777" cellspacing=0 border=0 cellpadding=0>
<tr><td> rowspan=2></td>
<td valign=middle align=left bgcolor="#000099">
 
<font size=-1 color="#ffffff">
<b> Login </b></font></td>
<td rowspan=2> </td></tr>
<tr><td colspan=2 bgcolor="#eeeeee" valign=top align=left width=100%>
<table cellpadding=2 width=100%><tr><td>
<small>Log in to Your Portal:</small><br/>
<form>
<ul>
<li> Barcode: <input name="barcode">
<li> Password: <input name="password">

</ul>

</form>
</td></tr></table>

</td></tr>
<tr><td colspan=4> </td></tr>
</table>
```

Musimy teraz podjąć kilka decyzji odnośnie układu naszej strony. Jak wspomniałem, wiadomości będziemy przechowywać w systemie plików z podziałem na różne katalogi. Każda pobrana grupa wiadomość będzie oddzielnym komponentem Masona wyświetlanym przez komponent biblioteczny, który nazwiemy *RSSBox*. Komponent *Directories* będzie ramką zawierającą listę kategorii; kliknięcie jednej z nich spowoduje wyświetlenie wszystkich należących do niej wiadomości. Ponieważ każda kategoria jest oddzielnym katalogiem, możemy utworzyć listę w sposób pokazany w przykładzie 3.12.

Przykład 3.12. Komponent Directories

```
<& /BoxTop, title=> "Resources" &>

<ul>
<%$Portal::dirs%>
</ul>
<& /BoxEnd &>

<%ONCE>
    my $root = "/var/portal";
```

```
    for my $top (grep { -d $_ } glob("$root*")) {

        $top =~ s/$root//;
        $Portal::dirs .= qq{
            <li><a href="/?open=$top">$top</a>
        } unless $top =~ /\W/;
    }
</%ONCE>
```

Działa to w następujący sposób: podczas uruchamiania serwera przeglądane są wszystkie podkatalogi głównego katalogu naszego portalu, po czym usuwana jest z nich nazwa katalogu podstawowego (w tym przypadku */var/portal/*) po to, by zamienić je na łącza używane w naszej aplikacji. Przykładowo, katalog o nazwie */var/portal/News* zostanie zamieniony na łącze */?open=News*. Łącze to spowoduje powrót na stronę główną, gdzie wartość parametru `open` wymusi zaprezentowanie komponentu *DirectoryPane*, który wyświetli wiadomości z wybranego katalogu. Kod pomija wszelkie katalogi, których nazwy nie zawierają liter, dzięki czemu wygenerowane łącza bez problemu przejdą wszystkie sprawdzenia dokonywane na parametrze `open`.

Przejdźmy więc do implementacji komponentu katalogów, *DirectoryPane*. Wiemy, że będziemy przeglądać katalog zawierający zbiór plików będących komponentami Masona. Będziemy chcieli dynamicznie dołączyć każdy z tych plików, tak by zbudować katalog wiadomości.

Dynamiczne wywoływanie komponentów umożliwia metoda `comp` request objectu Masona, `$m`; jest to perlowa wersja znacznika `<& comp &>` dołączającego komponenty. Ostatecznie komponent obsługujący katalogi przyjmie postać pokazaną w przykładzie 3.13.

Przykład 3.13. Komponent DirectoryPane

```
<%ARGS>
$open
</%ARGS>

% for (grep {-f $_} glob( "/var/portal/$open/*") ) {
% s|/var/portal/||;

<% $m->comp($_) %>
% }
```

Najpierw odbieramy nazwę katalogu do otwarcia. Następnie przeglądamy wszystkie pliki w tym katalogu, usuwając nazwę katalogu głównego (idealnym rozwiązaniem byłoby podanie go w pliku konfiguracyjnym) i wywołujemy komponenty o uzyskanych nazwach. Oznacza to, że gdyby istniał katalog *Technology*, zawierający następujące pliki:

```
01-Register
02-Slashdot
03-MacNews
04-LinuxToday
05-PerlDotCom
```

to wywołanie `<& /DirectoryPane, open => "Technology" &>` będzie miało efekt taki jak zapisanie wszystkiego oddzielnie:

```
<& /Technology/01-Register   &>
<& /Technology/02-Slashdot   &>
<& /Technology/03-MacNews    &>
<& /Technology/04-LinuxToday &>
<& /Technology/05-PerlDotCom &>
```

Widok standardowy, opisany w przykładzie 3.14, pojawia się wtedy, gdy nie zostanie wybrany żaden katalog. Będzie prezentował te wiadomości, które uznamy za domyślne:

Przykład 3.14. Komponent StandardPane

```
<& /BoxTop, title=> "Hello!", color => "dd2222"&>
Welcome to your portal! From here you can subscribe to a wide range of
news and alerting services; if you long in, you can customize this home
page.
<& /BoxEnd &>

<& /Weather/01-Oxford &>
<& /Technology/02-Slashdot &>
<& /News/01-BBC &>
<& /People/03-Rael &>
...
```

Cóż więc będą zawierały poszczególne pliki wiadomości? Jak wspomniałem, będą korzystały z komponentu *RSSBox*, przekazując URL do wiadomości i ewentualnie kolor, maksymalną liczbę wiadomości czy ich zbiorową nazwę. Poza tym będą przekazywały parametr informujący, czy wyświetlane mają być tylko nagłówki i odnośniki do każdego elementu RSS czy może także szersze opisy. Przykładowo, */News/01-BCC* wygląda tak:

```
<7 /RSSBoxd, URL =>"http://www.newsisfree.com/HPE/xml/feeds/60/60.xml",
Color =>"#dd0000" &>
```

a odnośnik do bloga Raela Dornfesta tak:

```
<& /RSSBox, URL => "http://www.orillynet.com/~rael/index.rss",
Color=> "#cccc00"  Title => "Rael Dornfest", Full => 0 &>
```

Jak się za chwilę okaże, całe piękno takiego modularnego systemu objawia się najpełniej w tym, że możemy tworzyć komponenty, które będą robiły coś innego niż proste pobieranie wiadomości RSS.

Ale najpierw uzupełnijmy nasz portal, pisząc komponent *RSSBox*, wykorzystywany przez komponenty utworzone wcześniej. Najpierw skorzystamy z bloku `ONCE` do załadowania wszystkich potrzebnych modułów:

```
<%ONCE>
use XML::RSS;
use LWP::Simple;
</%ONCE>
```

Następnie pobieramy argumenty, określając odpowiednie wartości domyślne:

```
<%ARGS>
$URL
$Color => "#0000aa"
$Max => 5
$Full => 1
$Title => undef
</%ARGS>
```

Zanim zaczniemy cokolwiek wyświetlać na stronie, załadujemy wiadomości ze wskazanego źródła i przeanalizujemy za pomocą modułu `XML::RSS`. Wywołujemy metodę `cache_self` Masona, która spowoduje, że komponent ten będzie korzystał z pamięci podręcznej dla danych wyjściowych; jeżeli w ciągu 10 minut wystąpi próba dostępu do tego samego URL-a, zaprezentowane zostaną dane z pamięci podręcznej:

```
<%INIT>
return if $m->cache_self(key => $URL, expires_in => '10 minutes');
my $rss = new XML::RSS;
eval { $rss->parse(get($URL));};
my $title = $Title || $rss->channel('title');
</%INIT>
```

Doszliśmy wreszcie do końca. Postać całego komponentu prezentuje przykład 3.15.

Przykład 3.15. Komponent RSSBox

```
<%ONCE>
use XML::RSS;
use LWP::Simple;
</%ONCE>

<%ARGS>
$URL
$Color => "#0000aa"
$Max => 5
$Full => 1
$Title => undef
</%ARGS>

<%INIT>
my $rss = new XML::RSS;
eval { $rss->parse(get($URL));};
my $title = $Title || $rss->channel('title');
my $site = $rss->channel('link');
</%INIT>

<BR>
<& BoxTop, color => $Color, title => $title, title_href = $site &>

    <dl class="rss">
% my $count = 0;
% for (@{$rss->{items}}) {
    <dt class="rss">
    <a href="<% $_->{link} %>"> <% $_->{title} %> </a>
    </dt>
% if ($Full) {
    <dd> <% $_->{description} %> </dd>
% }

% last if ++$count >= $Max
% }

    </dl>
<& /BoxEnd &>
```

Komponent nie jest bardzo skomplikowany; dla każdej pobranej wiadomości tworzony jest odnośnik, tytuł i, opcjonalnie, opis. Działanie przerywamy po przetworzeniu maksymalnej dopuszczalnej liczby wiadomości.

Demonstruje to potężne możliwości tkwiące w Masonie. Jak już pisałem, całkowity czas tworzenie tej witryny był nie dłuży niż kilka godzin. Cały kod zajmuje wyraźnie mniej niż 200 linii. Dodatkowo, jak wspomniałem, zyskujemy możliwość dołączania komponentów, które nie będą obsługiwały tylko wiadomości RSS. Przykładowo, żadna witryna nie rozgłasza wiadomości pogodowych z okolicy Oksfordu, ale istnieje witryna publikująca te dane w ściśle określonym formacie. Oznacza to, że przykładowy komponent *Weather/01-Oxford* nie będzie wcale wywoływał komponentu *RSSBox*, ale uzyska informacje w następujący sposób:

```
<%INIT>
use LWP::Simple
my @lines = grep /Temperature|Pressure|humidity|^Sun|Rain/,
            split /\n/,
            get('http://www-atm.physics.ox.ac.uk/user/cfinlay/now.htm');
</%INIT>

<br>
<& /BoxTop, title => "Oxford Weather", color => "#dd00dd" &>

<ul>
% for (@lines) {
 <li> <% $_%> </li>
% }
</ul>

<& /BoxEnd &>
```

Jest to dobre podsumowanie opisu Masona — system prosty, łatwo rozszerzalny i oferujący ogromne możliwości.

Oczywiście Mason oferuje wiele innych możliwości — zbyt wiele, by opisać je wszystkie w tym miejscu. Fantastyczna książka *Embedding Perl in HTML with Mason* (*http://www.masonbook.com*) autorstwa Dave'a Rolskiego i Kena Williamsa opisuje go szczegółowo, nie pomijając zagadnienia instalowania i uruchomienia Masona na serwerze WWW. Wiele informacji można znaleźć także na stronie domowej projektu (*http://www.masonhq.com*).

Template Toolkit

Rozwiązania omawiane do tej pory były dedykowane głównie dla programistów Perla — kod Perla był osadzany w tym czy innym medium — Template Toolkit Andiego Wardleya (*http://www.template-toolkit.org*) jest nieco inny. Komponenty, pętle, wywołania metod, elementy struktur danych i inne są w nim opisywane za pomocą własnego języka wzorców. Dzięki temu jest łatwy do opanowania przez projektantów stron, którzy niekoniecznie znają tajniki perlowej strony danej aplikacji[6], ale pracują nad jej stroną prezentacyjną. Zgodnie z tym, co mówi sama dokumentacja, język Template Toolkitu należy postrzegać jako zbiór dyrektyw określających sposób wyświetlania danych, a nie ich obliczania.

Podobnie jak w Masonie kompilacja, pamięć podręczna i szablony są obsługiwane w tle. Jednak inaczej niż Mason, Template Toolkit został pomyślany jako ogólny, łatwo rozszerzalny system obsługi wyświetlania i formatowania danych. Przykładowo, można go wykorzystać do dynamicznego generowania dokumentów PDF zawierających wykresy tworzone na podstawie danych z bazy danych, a wszystko to tylko z wykorzystaniem standardowych wtyczek i filtrów języka Template Toolkitu.

Zanim jednak przejdziemy do omawiania złożonych przypadków, przyjrzyjmy się bardzo prostym zastosowaniom Template Toolkitu. W najprostszych przypadkach przypomina w działaniu `Text::Template`. Tworzymy obiekt szablonu, przekazujemy mu dane i wskazujemy szablon do przetworzenia:

[6] I nie wykazują chęci jej poznania!

```
use Template;
my $template = Template->new():
my $variables = {
    who        => "Andy Wardley",
    modulename => "Template Toolkit",
    hours      => 30,
    games      => int(30*2.4)
};
$template->process("thankyou.txt", $variables);
```

Tym razem szablon wygląda tak:

```
Drogi [% who %],
    Dziękuję Ci za moduł [% modulename %], który pozwolił mi oszczedzić około
[% hours %] godzin pracy w tym roku. Dzięki temu miałem możliwość rozegrania
[% games %] potyczek w go i bardzo cenię sobie to, że nie straciłem
tego czasu, próżnując na IRC-u.

Z poważaniem,
Simon
```

Przetworzony tekst szablonu pojawi się oczywiście na standardowym wyjściu. Zauważmy jednak, że zmienne umieszczone pomiędzy `[%` i `%]` nie są zmiennymi Perla poprzedzanymi przy zapisie znakiem, do którego jesteśmy przyzwyczajeni; są to zmienne Template Toolkitu. Zmienne w tym systemie mogą być bardziej złożone niż tylko proste skalary — prosta, zwarta składnia umożliwia dostęp do złożonych struktur danych, nawet do obiektów Perla. Powróćmy do przykładu z wycenianiem prac nad nowym logo dla pewnej firmy. Tym razem jednak użyjemy nieco innej struktury danych:

```
my $invoice = {
    client => "Acme Motorhomes and Eugenics Ltd.",
    jobs => [
     { cost => 450.00, description => "Zaprojektowanie nowego logo" },
     { cost => 300.00, description => "Papier firmowy" },
     { cost => 900.00, description => "Przeprojektowanie strony WWW" },
     { cost =>  33.75, description => "Inne wydatki" }
    ],
    total => 0
};

$invoice->{total} += $_->{cost} for @{$invoice->{jobs}};
```

Jak zaprojektować szablon, który poradzi sobie z obsługą tego typu danych? Oczywiście musimy w pętli przeglądać poszczególne prace umieszczone w anonimowej tablicy i pozyskiwać związane z nimi informacje. Oto jak to zrobimy:

```
Do [% client %]

Dziękujemy za skorzystanie z usług Fungly Foobar Design Associates.
Oto zestawienie kosztów prac wykonanych w ramach otrzymanego zlecenia:

[% FOREACH job = jobs %]
     [% job.description %]  : [% job.cost %]
[% END %]

Koszt całkowity                                          $[% total %]

Termin płatności: 30 dni.

Z wyrażami szacunku,
Fungly Foobar
```

Jak widać, składnia jest inspirowana Perlem — możemy użyć wyrażenia `foreach` w celu przejrzenia wszystkich elementów listy, a odwoływać się do każdego z nich w poszczególnych iteracjach z wykorzystaniem lokalnej zmiennej `job`. Operator kropki jest odpowiednikiem operatora `->` z Perla — dokonuje dereferencji odwołań do tablic zwykłych i asocjacyjnych, a poza tym można nim wywoływać metody obiektów.

W naszym przykładzie ukryty jest pewien mankament: opis każdego elementu może być innej długości, przez co efekt końcowy może być mało przejrzysty[7]. Czy można coś na to poradzić? Jest to świetne pole do popisu dla użytecznego mechanizmu *filtrów* Template Toolkitu.

Filtry

Filtry w Template Toolkit przypominają nieco filtry uniksowe — są to krótkie procedury przyjmujące na wejście pewne dane, przekształcające je i odsyłające z powrotem. I podobnie jak filtry uniksowe dołącza się je do wyjścia szablonu za pomocą symbolu potoku (`|`).

W naszym przypadku wystarczy użyć filtru o znaczącej nazwie `format`, który formatuje dane wejściowe w sposób podobny do działania `printf`:

```
[% job.description | format("%60s") %]  : [%job.cost %]
```

Tym samym poprawiliśmy reprezentację danych wygenerowanych przez procesor obsługi szablonów — `job.descritpion` jest najpierw zamieniane na rzeczywisty opis, a po tym poddawane filtracji. Filtry można jednak zakładać na całe bloki szablonu. Przykładowo, jeśli chcielibyśmy zamienić tekst wyjściowy na HTML, moglibyśmy posłużyć się filtrem `html_entity` zamieniającym wszystkie znaki specjalne na odpowiadające im kody:

```
[% FILTER html_entity %]
Termin płatności: < 30 dni.
[% END %]
```

Tekst ten zostanie zamieniony na `Termin płatności: < 30 dni.`

To przykład kolejnego bloku dostępnego w Template Toolkit; do tej pory zetknęliśmy się z blokami `FOREACH` i `FILTER`. Dostępny jest poza tym blok `IF/ELSIF/ELSE`:

```
[% IF delinquent %]
   Infomracje w naszej bazie wskazują, że jest to drugie wezwanie do zapłaty. Prosimy
o NIEZWŁOCZNE uregulowanie zaległości.
[% ELSE %]
   Termin płatności: < 30 dni.
[% END %]
```

Inne interesujące filtry to: `upper`, `lower`, `ucfirst` i `lcfirst` zmieniające wielkość liter w tekście, `uri` do oznaczania znaków specjalnych w URI, `eval` do przekazywania tekstu na inny poziom przetwarzania szablonu i `perl_eval`, traktujący wyjście jako kod Perla, wykonujący go i dodający rezultat do wyjścia szablonu. Zestawienie wszystkich dostępnych filtrów, z przykładami użycia, znaleźć można w dokumentacji `Template::Manual::Filters`.

Wtyczki

Filtry stanowią interfejs do prostej funkcjonalności Perla — wbudowanych funkcji, takich jak `eval`, `uc`, `sprintf`, czy prostego zastępowania fragmentów tekstu — natomiast *wtyczki* (ang.

[7] Przemilczeliśmy ten problem zupełnie przy opisie `Text::Template`. Ciekawe czy to zauważyłeś?

plugins) umożliwiają dostęp do bardziej złożonych funkcji. Zazwyczaj używa się ich w celu dodania do języka formatującego funkcjonalności oferowanej przez określone moduły Perla.

Przykładowo, wtyczka `Template::Plugin::Autoformat` pozwala na korzystanie z możliwości modułu `Text::Autoformat` przy automatycznym formatowaniu tekstu. Podobnie jak w przypadku modułów Perla do ładowania wtyczek służy dyrektywa `USE` procesora formatującego. Spowoduje ona wyeksportowanie procedury `autoformat` i powiązanego z nią filtru o tej samej nazwie:

```
[% USE autoformat(right=78) %]
                                         [% address | autoformat %]
```

W ten sposób zapewniliśmy, że adres zostanie wyświetlony w odpowiednio przygotowanym bloku w prawej części strony.

Wyjątkowo przydatną wtyczką jest moduł `Template::Plugin::XML::Simple`, parsujący pliki XML za pomocą `XML::Simple` i umożliwiający manipulowanie wynikową strukturą danych z wnętrza szablonu. W poniższym przykładzie dyrektywa `USE` posłużyła do prostego zwrócenia wartości:

```
[% USE document = XML.Simple("app2ed.xml") %]
```

Od tej pory dysponujemy strukturą danych utworzoną na podstawie zawartości dokumentu w formacie XML. Strukturę tę możemy przeglądać, przechodząc do poszczególnych elementów, podobnie jak robiliśmy to w punkcie „Parsowanie XML-a" w rozdziale 2.:

```
Autor tej książki to
[% document.bookinfo.authorgroup.author.firstname # 'Simon'  %]
[% document.bookinfo.authorgroup.author.surname   # 'Cozens' %]
```

Napisanie wtyczki tego typu jest niezwykle proste, a jak się okaże, będziemy musieli to zrobić dla potrzeb naszego przykładu z wiadomościami RSS. Najpierw tworzymy nowy moduł o nazwie `Tamplate::Plugin::Whatever`, gdzie `Whatever` będzie nazwą wtyczki obowiązującą w obrębie języka szablonów. Moduł ten będzie ładował moduły, z których będziemy korzystali. Musimy przy tym dziedziczyć po `Template::Plugin`. Spróbujmy więc napisać interfejs do modułu `Data::BT::PhoneBill` Toniego Bowdena, obsługującego zapytania do systemu rachunków w British Telecom.

```
package Template::Plugin::PhoneBill;
use base 'Template::Plugin';
use Data::BT::PhoneBill;
```

W miejscu wywołania dyrektywy `USE` z nazwą naszej wtyczki powinna być przekazywana nazwa pliku z informacjami o rachunku, którą zamienimy na odpowiedni obiekt. Napiszmy więc zapewniającą to metodę `new`:

```
sub new {
    my ($class, $context, $filename) = @_;
    return Data::BT::PhoneBill->new($filename);
}
```

`$context` to obiekt przekazywany przez Template Toolkit reprezentujący kontekst, w którym procedura jest wywoływana. I to właściwie wszystko, co potrzeba do utworzenia wtyczki — można ewentualnie dodać jeszcze sprawdzenia, czy plik o podanej nazwie istnieje i czy da się go poprawnie przeanalizować, ale samo sedno budowy wtyczki jest takie jak pokazałem.

Po utworzeniu wtyczki możemy dobrać się do informacji o rachunku w taki sam sposób, jak robiliśmy to ze strukturą danych pozyskaną z `XML::Simple`:

```
[% USE bill = PhoneBill("mybill.txt") %]

[% WHILE call = bill.next_call %]
Call made on [% call.date %] to [% call.number %]...
[% END %]
```

Interesującą rzeczą wartą podkreślenia w tym miejscu jest fakt, że w przypadku korzystania z wtyczki `XML.Simple` dostęp do elementów struktury danych odbywał się za pomocą operatora kropki: `document.bookinfo` i tak dalej. W tym przypadku oznaczało to przeglądanie zawartości tablic asocjacyjnych; te same operacje zapisane w Perlu wyglądałyby tak: `$document->{bookinfo}->{authorgroup}->{author}....` W ostatnim przykładzie korzystaliśmy z dokładnie tego samego operatora kropki, ale tym razem oznaczał on wywołanie metod: `call.date` przekłada się w Perlu na `$call->date`. Jednak dla piszącego szablon wszystko wygląda identycznie. Taka abstrakcja rzeczywistych struktur danych jest jedną z wielkich zalet Template Toolkitu.

Komponenty i makra

Przy omawianiu systemu `HTML::Mason` wychwalałem możliwość podziału szablonu na wiele komponentów, które można było dołączać wielokrotnie z określonymi wartościami parametrów. Nie powinno więc być zaskoczeniem, że dokładanie tę samą funkcjonalność oferuje Template Toolkit.

Do dołączania komponentów do szablonu służy dyrektywa `INCLUDE`. Przykładowo, możemy w bardzo podobny sposób jak robiliśmy to w `HTML::Mason`, napisać bibliotekę rysującą ramkę pokazaną w przykładzie 3.16.

Przykład 3.16. Komponent BoxTop

```
<table bgcolor="#777777" cellspacing=0 border=0 cellpadding=0>
<tr>
  <td rowspan=2></td>
  <td valign=middle align=left bgcolor="[% color %]">
     
   <font size=-1 color="#ffffff">
   <b>
   [% IF title_href %]
      <a href="[% title_href %]"> [% title %] </a>
   [% ELSE %]
      [% title %]
   [% END %]
   </b>
   </font>
  </td>
  <td rowspan=2> </td>
</tr>
<tr>
  <td colspan=2 bgcolor="#eeeeee" valign=top align=left width=100%>
  <table cellpadding=2 width=100%>
    <tr><td>
```

i identycznie jak w `HTML::Mason` dołączyć ten komponent do szablonu z określeniem parametrów lokalnych:

```
[% INCLUDE boxtop
           title = "Login"
           ...
%]
```

W Template Toolkit dostępna jest jeszcze inna metoda abstrakcji często używanych komponentów — dyrektywa `MACRO`. Możemy zdefiniować makro rozwijane na dowolny kod Template Toolkitu; zacznijmy po prostu od dołączania komponentu rysującego za pomocą dyrektywy `INCLUDE`:

```
[% MACRO boxtop INCLUDE boxtop %]
[% MACRO boxend INCLUDE boxend %]
```

Dzięki temu narysowanie ramki będzie możliwe w taki mniej więcej sposób:

```
[% boxtop(title="My Box") %]
   <P> Hello, people! </P>
[% boxend %]
```

Zamiast korzystać z pliku komponentu i dyrektywy `INCLUDE` możemy przypisać nazwę makra do całego bloku dyrektyw Template Toolkitu.

```
[% MACRO boxtop BLOCK %]
<table bgcolor="#777777" cellspacing=0 border=0 cellpadding=0>
<tr>
  ...
[% END %]

[% MACRO boxend BLOCK %]
</td></tr></table>

</td></tr>
<tr><td colspan=4> </td></tr>
</table>
[% END %]
```

Możemy wreszcie zdefiniować całą bibliotekę podręcznych makr i dołączać ją później za pomocą pojedynczej dyrektywy `INCLUDE`, zamiast obsługiwać wiele oddzielnych plików z komponentami.

Załóżmy, że utworzyliśmy taką bibliotekę zawierającą pokazane wyżej dwa makra rysujące ramkę i przejdźmy do wykorzystania zdobytej wiedzy — obsługi wiadomości RSS.

Program zbierający wiadomości RSS

Przed przystąpieniem do pisania naszej aplikacji przeglądamy listę dostępnych wtyczek dla Template Toolkitu i z radością stwierdzamy, że istnieje taka, która może się nam przydać: `Template::Plugin::XML::RSS`, korzystająca z `XML::RSS`. Nasz zachwyt nie potrwa jednak długo — jak się okazuje, wymaga ona przekazania nazwy pliku, a nie adresu URL czy łańcucha danych w formacie XML. Niezbyt odpowiada nam wizja zapisywania plików, które trzeba będzie potem parsować.

Utworzymy więc własną podklasę `Template::Plugin::XML::RSS` przyjmującą adresy URL na wejście:

```
package Template::Plugin::XML::RSS::URL;
use base 'Template::Plugin::XML::RSS';
use LWP::Simple;

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

return $class->fail('Niepoprawny adres URL') unless $url;
```

```
my $url_data = get($url)
  or return $class->fail("Nie udało się pobrać danych z $url");

my $rss = XML::RSS->new
  or return $class->fail('Nie udało się utworzyć XML::RSS');

eval { $rss->parse($url_data) } and not $@
  or return $class->fail("Błąd przy parsowaniu $url: $@");

  return $rss;
}

1;
```

Możemy teraz utworzyć odpowiednik komponentu *RSSBox*, który wykorzystywaliśmy w Masonie:

```
[% MACRO RSSBox(url) USE rss = XML.RSS.URL(url) %]
[% box_top(title = rss.channel.title, title_href = rss.channel.link) %]

<dl class="rss">
[% FOREACH item = news.items %]
    <dt clas="rss">
       <a href="[% item.link  %]"> [% item.title %] </a>
    [% IF full %]
       <dd> [% item.description %] </dd>
    [% END %]
    </dt>
[% END %]
</dl>
[% box_end %]
[% END %]
```

Podstawową różnicą pomiędzy tym kodem a przykładem w Masonie jest to, że tutaj wszystko obsługiwane jest jawnie — projektant szablonu ma dostęp do całego procesu pobierania i parsowania danych RSS. Nie ma tu żadnego fragmentu w Perlu. Poza tym kod wynikowy jest zwarty, przejrzysty, łatwy do czytania i zrozumienia. Gdy dysponujemy takim makrem, ramkę zawierającą wiadomości RSS wyrażoną w HTML-u uzyskamy za pomocą jednego, prostego wywołania:

```
[% RSSBox("http://slashdot.org/slashdot.rss") %]
```

Od tego miejsca skonstruowanie pozostałej części aplikacji sprowadza się tylko do odpowiedniego zaprojektowania szablonu; wszystko, co związane z jest Perlem, zostało ładnie „obudowane”.

AxKit

Wspominam o szablonie AxKit (*http://www.axkit.org*) przy okazji omawiania systemów obsługi szablonów, choć w porównaniu z modułami poznanymi do tej pory ma on nieco inny charakter; nie jest to zwykły system szablonów, jest to pełnoprawny XML-owy serwer aplikacji dla Apache. AxKit stosuje się najczęściej do zamiany „w locie” dokumentów XML na HTML przy dostarczaniu ich odbiorcom w Sieci.

Jednak dzięki XSP (ang. *Extensible Server Pages*), opracowanym w ramach projektu Apache Cocoon, AxKit można wykorzystywać jako łatwo rozszerzalny system obsługi wzorców. Podstawą działania XSP jest to, że określone znaczniki XML powodują wykonanie wskazanych

procedur Perla. Na podstawowym poziomie można wykorzystać znaczniki po prostu do wydzielenia kodu Perla:

```
<p>
Good
<xsp:logic>
if ((localtime) [2] >= 12) {
    <i>Afternoon</i>
}
else {
    <i>Morning</i>
}
</xsp:logic>
</p>
```

AxKit pozwala na dowolne przeplatanie danych XML-owych z kodem Perla. Ponieważ analizuje XML-a, wie, że `<i>Afternoon</i>` to dane, a nie kod w Perlu, i odpowiednio je potraktuje. Oznacza to poza tym, że osoby dobrze znające XML-a znajdą sposób na zapewnienie poprawności otrzymanego HTML-a z osadzonym XSP. A ponieważ AxKit traktuje wszystko jako XML, tworzone dokumenty HTML muszą być poprawnie sformatowane, gdyż w przeciwnym razie nie uzyskamy żadnego rezultatu na jego wyjściu.

AxKit nie ogranicza nas jednak tylko do tak podstawowego poziomu; XSP pozwala na tworzenie w Perlu bibliotek znaczników. Przykładowo, biblioteka `AxKit::XSP::ESQL` stanowi „obudowę" dla bibliotek DBI. Biblioteki znaczników definiują własne przestrzenie nazw w XML-u i umieszczają nowe znaczniki w ich obrębie. W dokumencie XML importowanie bibliotek znaczników odbywa się poprzez deklarację wykorzystania odpowiedniej przestrzeni nazw:

```
<xsp:page
     language="perl"
     xmlns:xsp="http://apache.org/xsp/core/v1"
     xmlns:esql="http://apache.org/xsp/SQL/v2"
>
```

Od tej pory można używać znaczników `<esql: ...>` w dokumencie:

```
<esql:connection>
<esql:driver>Pg</esql:driver>
<esql:dburl>dbname=rss</esql:dburl>
<esql:username>www</esql:username>
<esql:password></esql:password>
<esql:execute-query>
  <esql:query>
    select description, url, title from feeds
  </esql:query>
  <esql:results>
    <ul>
    <esql:row-results>
       <li>
        <a>
         <xsp:attribute name="href">
             <esql:get-string column="url"/>
         </xsp:attribute>
         <esql:get-string column="name"/>
        </a> - <esql-get-string column="description"/>
       </li>
    </esql:row-results>
    </ul>
  </esql:results>
```

```
    <esql:no-results> <p> Nie uzyskano żadnego wyniku! </p> </esql:no-results>
  </esql:execute-query>
  </esql:connection>
```

Wysyłamy zapytanie SQL i zamieniamy wynik na listę w HTML-u. Jedyny fragment, który może nie być oczywisty, to miejsce wykorzystania atrybutu, `<xsp:attribute>`. Kluczem do zrozumienia jest tu zdanie sobie sprawy z tego, że dokument przetwarzany przez AxKit musi być w 100% poprawnym dokumentem w formacie XML. W przypadku `HTML::Template` czy `HTML::Mason` nie mieliśmy problemów z zapisami w stylu `<a href="<TMPL_VAR URL>">` czy `<a href="<% $url |n%>">`, czyli umieszczaniem znaczników wewnątrz innych znaczników.

AxKit przetwarza cały dokument jako XML, a dopiero potem poddaje go transformacjom. W podanych wyżej przykładach znaczniki zostałyby zinterpretowane jako całkowicie poprawne (choć dla nas bezsensowne) wartości atrybutów `<TMPL_VAR URL>` oraz `<% $url|n>` i na tym zakończyłoby się ich przetwarzanie. Co gorsza, nie możemy sobie nawet pozwolić na zapis w postaci `<a href=<esql:get-string column="url"/>>`, bo nie jest to poprawne wyrażenie w XML-u.

Dlatego posługujemy się niewielką sztuczką. Na warstwie XSP wymuszamy zapisanie odpowiedniej wartości atrybutu `href` dla znacznika `<a>`, co nastąpi już po przeanalizowaniu składni dokumentu, która przebiegnie gładko i bez zakłóceń.

Istnieje wiele innych bibliotek znaczników pełniących te same funkcje co wtyczki Template Toolkitu i udostępniające autorowi dokumentu XML wysokopoziomową funkcjonalność Perla; na przykład moja własna biblioteka `AxKit::XSP::ObjectTaglib` pozwala programiście na łatwe przekształcenie dowolnego modułu obiektowego w bibliotekę znaczników.

Nie będziemy za pomocą AxKita implementować aplikacji gromadzącej wiadomości RSS — jest to bowiem kompletny system obsługi dokumentów XML. Z tego względu całą pracę można w nim załatwić na poziomie arkuszy stylów XSTL, co praktycznie eliminuje konieczność stosowania Perla.

Osoby zainteresowane tym projektem znajdą więcej informacji w książce *Perl and XML* (wydawnictwa O'Reilly) i na stronie domowej AxKita, *http://www.axkit.org*.

Podsumowanie

W niniejszym rozdziale dokonaliśmy przeglądu kilku z dostępnych systemów obsługi szablonów wykorzystywanych najczęściej w połączeniu z Perlem: począwszy od prostych formatów — `sprintf` i podobnych — poprzez `Text::Template` i `HTML::Template`, aż do bardziej wyrafinowanych rozwiązań w postaci `HTML::Mason` i Template Toolkit.

Pominęliśmy jednak odpowiedź na jedno ważne pytanie: którego z nich należy używać? Odpowiedź, jak zwykle, zależy częściowo od tego, co trzeba zrobić, a częściowo od upodobań programisty.

Po pierwsze, należy rozważyć różnice pomiędzy systemami opartymi na Perlu, jak `Text::Template` i `Text::Autoformat`, a działającymi na nieco innej zasadzie modułami w stylu `HTML::Masona`. Jeżeli głównym zadaniem programu będzie generowanie pewnych danych w oparciu o szablony, jak w przypadku aplikacji WWW, to prawdopodobnie powinieneś się skłaniać w stronę rozwiązań typu `HTML::Mason` czy Template Toolkit.

Należy poza tym wziąć pod uwagę to, kto będzie tworzył szablony i czy będą one przetwarzane bezpośrednio przez kod Perla. Template Toolkit, AxKit i `HTML::Template` to przykłady systemów starających się ukryć przed autorem szablonów wszelkie zawiłości Perla, natomiast `HTML::Mason` zmusza go do intensywnego wykorzystywania tego języka.

Po drugie, pozostaje jeszcze zagadnienie osobistych upodobań. Ja osobiście nie przepadam zbytnio za `HTML::Template`, za to podoba mi się filozofia działania Masona. Zdaję sobie sprawę z potężnych możliwości oferowanych przez AxKit, ale wiele razy frustrowała mnie konieczność zapewniania czystości formatu XML. W miarę używania coraz bardziej podoba mi się Template Toolkit, choć i tak wolę Masona, głownie dlatego, że mam w nim więcej doświadczenia.

Każdy może mieć inne zapatrywania na te sprawy. I bardzo dobrze — podobnie jak wiele innych rzeczy w Perlu, wszystko można zrobić na wiele sposobów.