# Rails. Zaawansowane programowanie


Autor: Brad Ediger
Tłumaczenie: Paweł Gonera
ISBN: 978-83-246-1724-1
Tytuł oryginału: Advanced Rails
Format: 168x237 , stron: 336

**Twórz zaawansowane projekty w Rails!**

- Jak zadbać o bezpieczeństwo?
- Jak zapewnić wydajność Twojej aplikacji?
- Jak stworzyć i utrzymać duży projekt w Rails?

Ruby on Rails przebojem wdarł się na rynek szkieletów aplikacji internetowych. Stworzony w architekturze MVC z wykorzystaniem popularnego języka Ruby, został entuzjastycznie przyjęty przez społeczność programistów. Główne założenia autora tego projektu, Davida Heinemeiera Hanssona, to szybkość, łatwość i przyjemność tworzenia kodu. Ruby on Rails jest dojrzałym rozwiązaniem, wykorzystywanym przez wiele firm w aplikacjach internetowych, tworzonych pod kątem ich specyficznych potrzeb. Liczba aplikacji, które powstały z wykorzystaniem tego szkieletu, świadczy o jego wysokiej jakości oraz niewątpliwie ma wpływ na wzrost popularności samego języka Ruby.

„Rails. Zaawansowane programowanie" porusza te tematy, które Wy, programiści, lubicie najbardziej! Dzięki tej książce dowiesz się, w jaki sposób wykorzystać gotowe wtyczki oraz jak stworzyć nowe. Nauczysz się stosować zaawansowane funkcje bazy danych oraz podłączać się jednocześnie do wielu baz. Po lekturze tego podręcznika bez problemu zapewnisz swojej aplikacji najwyższy poziom bezpieczeństwa, optymalną wydajność i skalowalność. Autor wskazuje tutaj również niezwykle interesujące kwestie, dotyczące projektowania dużych aplikacji, wykorzystania systemów kontroli wersji oraz utrzymywania właściwej struktury projektu.

- Przypomnienie i omówienie podstawowych elementów Ruby i Rails
- Stosowanie ActiveSupport oraz RailTies
- Zastosowanie i projektowanie wtyczek
- Zaawansowane wykorzystanie baz danych
- Uwierzytelnianie za pomocą LDAP
- Bezpieczne szyfrowanie haseł
- Bezpieczne przetwarzanie formularzy i danych użytkownika
- Zapewnienie wydajności
- Skalowanie architektury
- Wykorzystywanie usług Web
- Tworzenie wielojęzycznych aplikacji
- Zarządzanie dużymi projektami
- Używanie systemów kontroli wersji

**Poznaj wszystkie funkcje Ruby on Rails!**



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

# Spis treści

ROZDZIAŁ 1.

# Techniki podstawowe

*Do osiągnięcia niezawodności jest wymagana prostota.*
Edsger W. Dijkstra

Od pierwszego wydania w lipcu 2004 roku środowisko Ruby on Rails stale zdobywa popularność. Rails przyciąga programistów PHP, Java i .NET swoją prostotą — architekturą model-widok-kontroler (MVC), rozsądnymi wartościami domyślnymi („konwencja nad konfiguracją") oraz zaawansowanym językiem programowania Ruby.

Środowisko Rails miało przez pierwszy rok lub dwa słabą reputację z uwagi na braki w dokumentacji. Luka ta została wypełniona przez tysiące programistów, którzy korzystali ze środowiska Ruby on Rails, współtworzyli je i pisali na jego temat, jak również dzięki projektowi Rails Documentation (*http://railsdocumentation.org*). Dostępne są tysiące blogów, które zawierają samouczki oraz porady na temat programowania w Rails.

Celem tej książki jest zebranie najlepszych praktyk oraz wiedzy zgromadzonej przez środowisko programistów Rails i zaprezentowanie ich w łatwej do przyswojenia, zwartej formie. Poszukiwałem ponadto tych aspektów programowania dla WWW, które są często niedoceniane lub pomijane przez środowisko Rails.

## Czym jest metaprogramowanie?

Rails udostępnia metaprogramowanie dla mas. Choć nie było to pierwsze zastosowanie zaawansowanych funkcji Ruby, to jednak jest ono chyba najbardziej popularne. Aby zrozumieć działanie Rails, konieczne jest wcześniejsze zapoznanie się z tymi mechanizmami Ruby, które zostały wykorzystane w tym środowisku. W tym rozdziale przedstawione zostaną podstawowe mechanizmy zapewniające działanie technik przedstawianych w pozostałych rozdziałach książki.

**Metaprogramowanie** to technika programowania, w której kod jest wykorzystywany do tworzenia innego kodu, bądź dokonania introspekcji samego siebie. Przedrostek *meta* (z greki) wskazuje na abstrakcję; kod wykorzystujący techniki metaprogramowania działa jednocześnie na dwóch poziomach abstrakcji.

Metaprogramowanie jest wykorzystywane w wielu językach, ale jest najbardziej popularne w językach dynamicznych, ponieważ mają one zwykle więcej funkcji pozwalających na manipulowanie kodem jako danymi. Pomimo tego, że w językach statycznych, takich jak C# lub

Java, dostępny jest mechanizm refleksji, to nie jest on nawet w części tak przezroczysty, jak w językach dynamicznych, takich jak Ruby, ponieważ kod i dane znajdują się w czasie działania aplikacji na dwóch osobnych warstwach.

Introspekcja jest zwykle wykonywana na jednym z tych poziomów. **Introspekcja syntaktyczna** jest najniższym poziomem introspekcji — pozwala na bezpośrednią analizę tekstu programu lub strumienia tokenów. Metaprogramowanie bazujące na szablonach lub makrach zwykle działa na poziomie syntaktycznym.

Ten typ metaprogramowania jest wykorzystany w języku Lisp poprzez stosowanie **S-wyrażeń** (bezpośredniego tłumaczenia drzewa abstrakcji składni programu) zarówno w przypadku kodu, jak i danych. Metaprogramowanie w języku Lisp wymaga intensywnego korzystania z **makr**, które są tak naprawdę szablonami kodu. Daje to możliwość pracy na jednym poziomie; kod i dane są reprezentowane w ten sam sposób, a jedynym, co odróżnia kod od danych, jest to, że jest on wartościowany. Jednak metaprogramowanie na poziomie syntaktycznym ma swoje wady. Przechwytywanie zmiennych oraz przypadkowe wielokrotne wartościowanie jest bezpośrednią konsekwencją umieszczenia kodu na dwóch poziomach abstrakcji dla tej samej przestrzeni nazw. Choć dostępne są standardowe idiomy języka Lisp pozwalające na uporanie się z tymi problemami, to jednak są one kolejnymi elementami, których programista Lisp musi się nauczyć i pamiętać o nich.

Introspekcja syntaktyczna w Ruby jest dostępna za pośrednictwem biblioteki `ParseTree`, która pozwala na tłumaczenie kodu źródłowego Ruby na S-wyrażenia[1]. Interesującym zastosowaniem tej biblioteki jest Heckle[2], biblioteka ułatwiająca testowanie, która analizuje kod źródłowy Ruby i zmienia go poprzez modyfikowanie ciągów oraz zmianę wartości `true` na `false` i odwrotnie. W założeniach, jeżeli nasz kod jest odpowiednio pokryty testami, każda modyfikacja kodu powinna zostać wykryta przez testy jednostkowe.

Alternatywą dla introspekcji syntaktycznej jest działająca na wyższym poziomie **introspekcja semantyczna**, czyli analiza programu z wykorzystaniem struktur danych wyższego poziomu. Sposób realizacji tej techniki różni się w rożnych językach programowania, ale w Ruby zwykle oznacza to operowanie na poziomie klas i metod — tworzenie, modyfikowanie i aliasowanie metod; przechwytywanie wywołań metod; manipulowanie łańcuchem dziedziczenia. Techniki te są zwykle bardziej związane z istniejącym kodem niż metody syntaktyczne, ponieważ najczęściej istniejące metody są traktowane jako czarne skrzynki i ich implementacja nie jest swobodnie zmieniana.

## Nie powtarzaj się

Na wysokim poziomie metaprogramowanie jest przydatne do wprowadzania **zasady DRY** (ang. *Don't Repeat Yourself* — „nie powtarzaj się"). Zgodnie z tą techniką, nazywaną również „Raz i tylko raz", każdy element informacji musi być zdefiniowany w systemie tylko raz. Powielanie jest zwykle niepotrzebne, szczególnie w językach dynamicznych, takich jak Ruby. Podobnie jak abstrakcja funkcjonalna pozwala nam na uniknięcie powielania kodu, który jest taki sam lub niemal taki sam, metaprogramowanie pozwala nam uniknąć podobnych koncepcji, wykorzystywanych w aplikacji.

[1] *http://www.zenspider.com/ZSS/Products/ParseTree*.

[2] *http://rubyforge.org/projects/seattlerb*.

Metaprogramowanie ma na celu zachowanie prostoty. Jednym z najłatwiejszych sposobów na zapoznanie się z metaprogramowaniem jest analizowanie kodu i jego refaktoryzacja. Nadmiarowy kod może być wydzielany do funkcji; nadmiarowe funkcje lub wzorce mogą być często wydzielone z użyciem metaprogramowania.

Wzorc e projektowe definiują nakładające się obszary; wzorce zostały zaprojektowane w celu zminimalizowania liczby sytuacji, w których musimy rozwiązywać ten sam problem. W społeczności Ruby wzorce projektowe mają dosyć złą reputację. Dla części programistów wzorce są wspólnym słownikiem do opisu rozwiązań powtarzających się problemów. Dla innych są one „przeprojektowane".

Aby być pewnym tego, że zastosowane zostaną wszystkie dostępne wzorce, muszą być one nadużywane. Jeżeli jednak będą używane rozsądnie, nie musi tak być. Wzorce projektowe są użyteczne jedynie w przypadku, gdy pozwalają zmniejszać złożoność kognitywną. W Ruby część najbardziej szczegółowych wzorców jest tak przezroczysta, że nazywanie ich „wzorcami" może być nieintuicyjne; są one w rzeczywistości idiomami i większość programistów, którzy „myślą w Ruby", korzysta z nich bezwiednie. Wzorce powinny być uważane za słownik wykorzystywany przy opisie architektury, a nie za bibliotekę wstępnie przygotowanych rozwiązań implementacji. Dobre wzorce projektowe dla Ruby znacznie różnią się w tym względzie od dobrych wzorców projektowych dla C++.

Uogólniając, metaprogramowanie nie powinno być wykorzystywane tylko do powtarzania kodu. Zawsze powinno się przeanalizować wszystkie opcje, aby sprawdzić, czy inna technika, na przykład abstrakcja funkcjonalna, nie nadaje się lepiej do rozwiązania problemu. Jednak w kilku przypadkach powtarzanie kodu poprzez metaprogramowanie jest najlepszym sposobem na rozwiązanie problemu. Jeżeli na przykład w obiekcie musi być zdefiniowanych kilka podobnych metod, tak jak w metodach pomocniczych `ActiveRecord`, można w takim przypadku skorzystać z metaprogramowania.

## Pułapki

Kod, który się sam modyfikuje, może być bardzo trudny do tworzenia i utrzymania. Wybrane przez nas konstrukcje programowe powinny zawsze spełniać nasze potrzeby — powinny one upraszczać życie, a nie komplikować je. Przedstawione poniżej techniki powinny uzupełniać zestaw narzędzi w naszej skrzynce, a nie być jedynymi narzędziami.

## Programowanie wstępujące

**Programowanie wstępujące** jest koncepcją zapożyczoną z świata Lisp. Podstawową koncepcją w tym sposobie programowania jest tworzenie abstrakcji od najniższego poziomu. Przez utworzenie na początku konstrukcji najniższego poziomu budujemy w rzeczywistości program na bazie tych abstrakcji. W pewnym sensie piszemy język specyficzny dla domeny, za pomocą którego tworzymy programy.

Koncepcja ta jest niezmiernie użyteczna w przypadku `ActiveRecord`. Po utworzeniu podstawowych schematów i modelu obiektowego można rozpocząć budowanie abstrakcji przy wykorzystaniu tych obiektów. Wiele projektów Rails zaczyna się od tworzenia podobnych do zamieszczonej poniżej abstrakcji modelu, zanim powstanie pierwszy wiersz kodu kontrolera lub nawet projekt interfejsu WWW:

```
class Order < ActiveRecord::Base
  has_many :line_items

  def total
    subtotal + shipping + tax
  end

  def subtotal
    line_items.sum(:price)
  end

  def shipping
    shipping_base_price + line_items.sum(:shipping)
  end

  def tax
    subtotal * TAX_RATE
  end
end
```

# Podstawy Ruby

Zakładamy, że Czytelnik dobrze zna Ruby. W podrozdziale tym przedstawimy niektóre z aspektów języka, które są często mylące lub źle rozumiane. Niektóre z nich mogą być Czytelnikowi znane, ale są to najważniejsze koncepcje tworzące podstawy technik metaprogramowania przedstawianych w dalszej części rozdziału.

## Klasy i moduły

Klasy i moduły są podstawą programowania obiektowego w Ruby. Klasy zapewniają mechanizmy hermetyzacji i separacji. Moduły mogą być wykorzystywane jako tzw. **mixin** — zbiór funkcji umieszczonych w klasie, stanowiących namiastkę mechanizmu dziedziczenia wielobazowego. Moduły są również wykorzystywane do podziału klas na przestrzenie nazw.

W Ruby każda nazwa klasy jest stałą. Dlatego właśnie Ruby wymaga, aby nazwy klas rozpoczynały się od wielkiej litery. Stała ta jest wartościowana na **obiekt klasowy**, który jest obiektem klasy `Class`. Różni się od **obiektu `Class`**, który reprezentuje faktyczną klasę `Class`[3]. Gdy mówimy o „obiekcie klasowym", mamy na myśli obiekt reprezentujący klasę (wraz z samą klasą `Class`). Gdy mówimy o „obiekcie `Class`", mamy na myśli klasę o nazwie `Class`, będącą klasą bazową dla wszystkich obiektów klasowych.

Klasa `Class` dziedziczy po `Module`; każda klasa jest również modułem. Istnieje tu jednak niezwykle ważna różnica. Klasy nie mogą być mieszane z innymi klasami, a klasy nie mogą dziedziczyć po obiektach; jest to możliwe tylko w przypadku modułów.

## Wyszukiwanie metod

Wyszukiwanie metod w Ruby może być dosyć mylące, a w rzeczywistości jest dosyć regularne. Najprostszym sposobem na zrozumienie skomplikowanych przypadków jest przedstawienie struktur danych, jakie Ruby wewnętrznie tworzy.

[3] Jeżeli nie jest to wystarczająco skomplikowane, trzeba pamiętać, że obiekt `Class` posiada również klasę `Class`.

Każdy obiekt Ruby[4] posiada zbiór pól w pamięci:

`klass`
: Wskaźnik do obiektu klasy danego obiektu (została użyta nazwa `klass` zamiast `class`, ponieważ ta druga jest słowem kluczowym w C++ i Ruby; jeżeli nazwalibyśmy ją `class`, Ruby kompilowałby się za pomocą kompilatora C, ale nie można byłoby użyć kompilatora C++. Ta wprowadzona umyślnie literówka jest używana wszędzie w Ruby).

`iv_tbl`
: „Tablica zmiennych instancyjnych" to tablica mieszająca zawierająca zmienne instancyjne należące do tego obiektu.

`flags`
: Pole bitowe znaczników `Boolean` zawierające informacje statusu, takie jak stan śladu obiektu, znacznik zbierania nieużytków oraz to, czy obiekt jest zamrożony.

Każda klasa Ruby posiada te same pola, jak również dwa dodatkowe:

`m_tbl`
: „Tablica metod" — tabela mieszająca metod instancyjnych danej klasy lub modułu.

`super`
: Wskaźnik klasy lub modułu bazowego.

Pola te pełnią ważną rolę w wyszukiwaniu metod i są ważne w zrozumieniu tego mechanizmu. W szczególności można zwrócić uwagę na różnicę pomiędzy wskaźnikami obiektu klasy: `klass` i `super`.

## Zasady

Zasady wyszukiwania metod są bardzo proste, ale zależą od zrozumienia sposobu działania struktur danych Ruby. Gdy do obiektu jest wysyłany komunikat[5], wykonywane są następujące operacje:

1. Ruby korzysta z wskaźnika `klass` i przeszukuje `m_tbl` z obiektu danej klasy, szukając odpowiedniej metody (wskaźnik `klass` zawsze wskazuje na obiekt klasowy).
2. Jeżeli nie zostanie znaleziona metoda, Ruby korzysta z wskaźnika `super` obiektu klasowego i kontynuuje wyszukiwanie w `m_tbl` klasy bazowej.
3. Ruby wykonuje wyszukiwanie w ten sposób aż do momentu znalezienia metody bądź też do osiągnięcia końca łańcucha wskaźników `super`.
4. Jeżeli w żadnym obiekcie łańcucha nie zostanie znaleziona metoda, Ruby wywołuje metodę `method_missing` z obiektu odbiorcy metody. Powoduje to ponowne rozpoczęcie tego procesu, ale tym razem wyszukiwana jest metoda `method_missing` zamiast początkowej metody.

[4] Poza obiektami natychmiastowymi (`Fixnums`, `symbols`, `true`, `false` oraz `nil`), które przedstawimy później.

[5] W Ruby często stosowana jest terminologia przekazywania komunikatów pochodząca z języka Smalltalk — gdy jest wywoływana metoda, mówi się, że jest *przesyłany komunikat*. Obiekt, do którego jest wysyłany komunikat, jest nazywany *odbiorcą*.

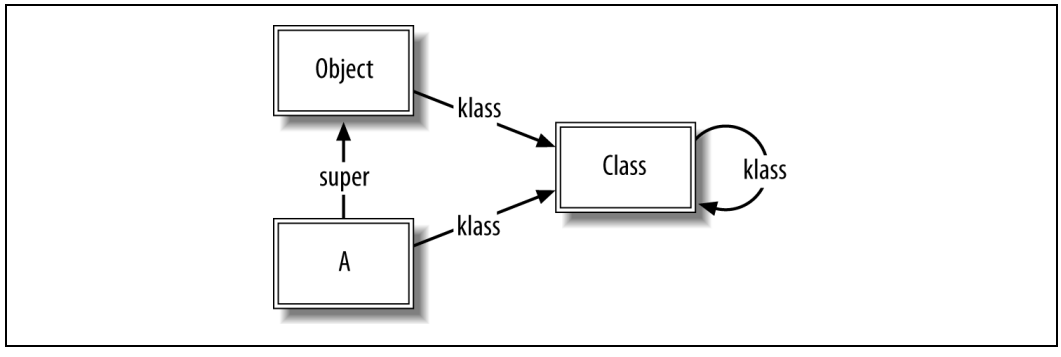
Zasady te są stosowane w sposób uniwersalny. Wszystkie interesujące mechanizmy wykorzystujące wyszukiwanie metod (*mixin*, metody klasowe i klasy singleton) wykorzystują strukturę wskaźników `klass` oraz `super`. Przedstawimy teraz ten proces nieco bardziej szczegółowo.

## Dziedziczenie klas

Proces wyszukiwania metod może być mylący, więc zacznijmy od prostego przykładu. Poniżej przedstawiona jest najprostsza możliwa definicja klasy w Ruby:

```
class A
end
```

Kod ten powoduje wygenerowanie w pamięci następujących struktur (patrz rysunek 1.1).




*Rysunek 1.1. Struktury danych dla pojedynczej klasy*

Prostokąty z podwójnymi ramkami reprezentują obiekty klas — obiekty, których wskaźnik `klass` wskazuje na obiekt `Class`. Wskaźnik `super` wskazuje na obiekt klasy `Object`, co oznacza, że `A` dziedziczy po `Object`. Od tego momentu będziemy pomijać wskaźniki `klass` dla `Class`, `Module` oraz `Object`, jeżeli nie będzie to powodowało niejasności.

Następnym przypadkiem w kolejności stopnia skomplikowania jest dziedziczenie po jednej klasie. Dziedziczenie klas wykorzystuje wskaźniki `super`. Utwórzmy na przykład klasę `B` dziedziczącą po `A`:

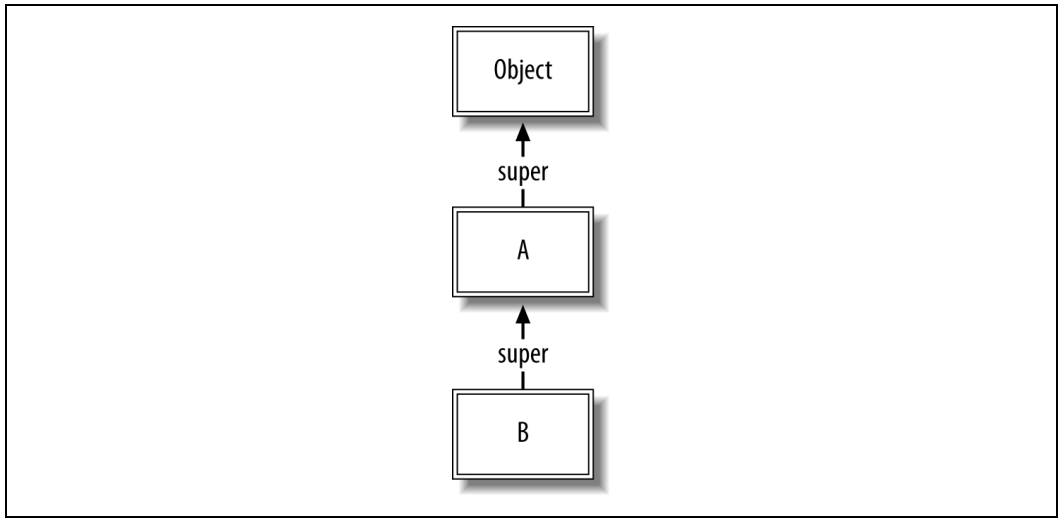
```
class B < A
end
```

Wynikowe struktury danych są przedstawione na rysunku 1.2.

Słowo kluczowe `super` pozwala na przechodzenie wzdłuż łańcucha dziedziczenia, tak jak w poniższym przykładzie:

```
class B
  def initialize
    logger.info "Tworzenie obiektu B"
    super
  end
end
```

Wywołanie `super` w `initialize` pozwala na przejście standardowej metody wyszukiwania metod, zaczynając od `A#initialize`.



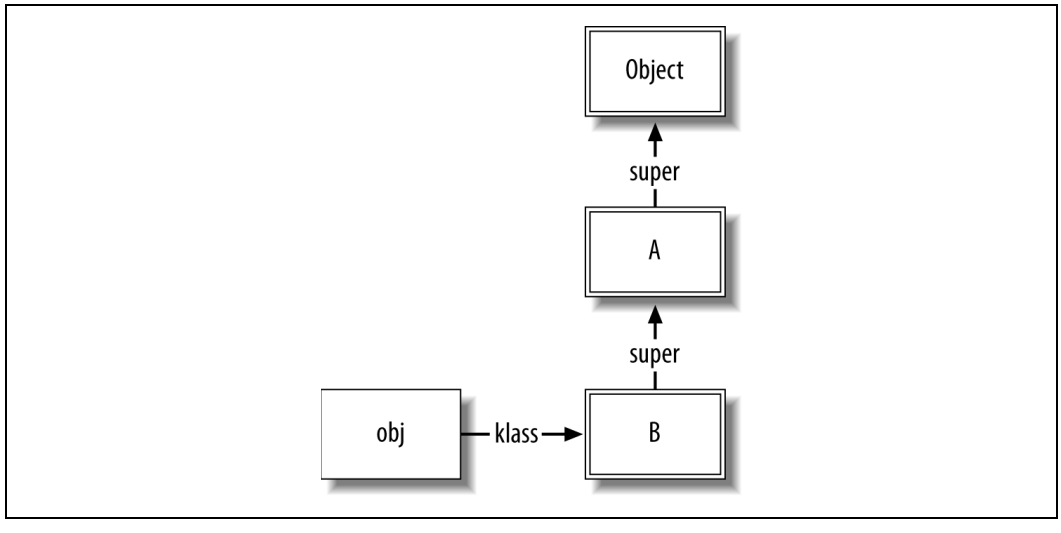

*Rysunek 1.2. Jeden poziom dziedziczenia*

## Konkretyzacja klas

Teraz możemy przedstawić sposób wyszukiwania metod. Na początek utworzymy instancję klasy `B`:

```
obj = B.new
```

Powoduje to utworzenie nowego obiektu i ustawienie wskaźnika `klass` na obiekt klasowy `B` (patrz rysunek 1.3).




*Rysunek 1.3. Konkretyzacja klas*

Pojedyncza ramka wokół `obj` reprezentuje zwykły obiekt. Trzeba pamiętać, że każdy prostokąt na tym diagramie reprezentuje instancje obiektu. Jednak prostokąty o podwójnej ramce, reprezentujące obiekty, są obiektami klasy `Class` (których wskaźnik `klass` wskazuje na obiekt `Class`).

Gdy wysyłamy komunikat do `obj`:

```
obj.to_s
```

realizowany jest następujący łańcuch operacji:

1. Wskaźnik `klass` obiektu `obj` jest przesuwany do `B`; w metodach klasy `B` (w `m_tbl`) wyszukiwana jest odpowiednia metoda.
2. W klasie `B` nie zostaje znaleziona odpowiednia metoda. Wykorzystywany jest wskaźnik `super` z obiektu klasy `B` i metoda jest poszukiwana w klasie `A`.
3. W klasie `A` nie zostaje znaleziona odpowiednia metoda. Wykorzystywany jest wskaźnik `super` z obiektu klasy `A` i metoda jest poszukiwana w klasie `Object`.
4. Klasa `Object` zawiera metodę `to_s` w kodzie natywnym (`rb_any_to_s`). Metoda ta jest wywoływana z parametrem takim jak #<B:0x1cd3c0>. Metoda `rb_any_to_s` analizuje wskaźnik `klass` odbiorcy w celu określenia nazwy klasy wyświetlenia; dlatego pokazywana jest nazwa `B`, pomimo tego, że wywoływana metoda znajduje się w `Object`.

### Dołączanie modułów

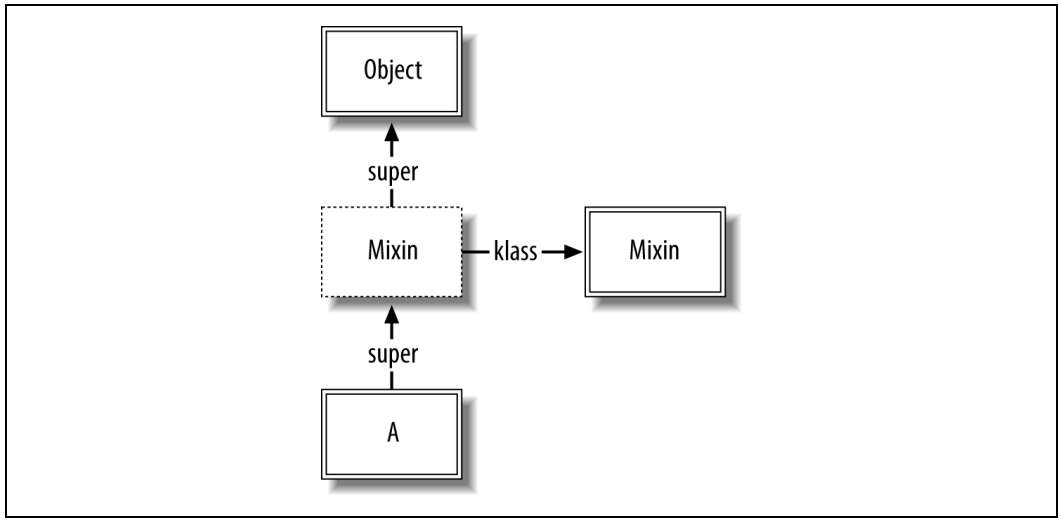
Gdy zaczniemy korzystać z modułów, sprawa stanie się bardziej skomplikowana. Ruby obsługuje dołączanie modułów zawierających `ICLASS`[6], które są pośrednikami modułów. Gdy dołączamy moduł do klasy, Ruby wstawia `ICLASS` reprezentujący dołączony moduł do łańcucha `super` dołączającej klasy.

W naszym przykładzie dołączania modułu uprościmy nieco sytuację przez zignorowanie klasy `B`. Zdefiniujemy moduł i dodamy go do `A`, co spowoduje powstanie struktur danych przedstawionych na rysunku 1.4:

```
module Mixin
  def mixed_method
    puts "Witamy w mixin"
  end
end

class A
  include Mixin
end
```

Tutaj właśnie do gry wkracza `ICLASS`. Wskaźnik `super` wskazujący z `A` na `Object` jest przechwytywany przez nowy `ICLASS` (reprezentowany przez kwadrat narysowany przerywaną linią). `ICLASS` jest pośrednikiem dla modułu `Mixin`. Zawiera on wskaźniki do tablic `iv_tbl` z `Mixin` (zmienne instancyjne) oraz `m_tbl` (metody).

[6] ICLASS jest nazwą dla klas pośredniczących, wprowadzoną przez Mauricia Fernándeza. Nie mają one oficjalnej nazwy, ale w kodzie źródłowym Ruby noszą nazwę `T_ICLASS`.

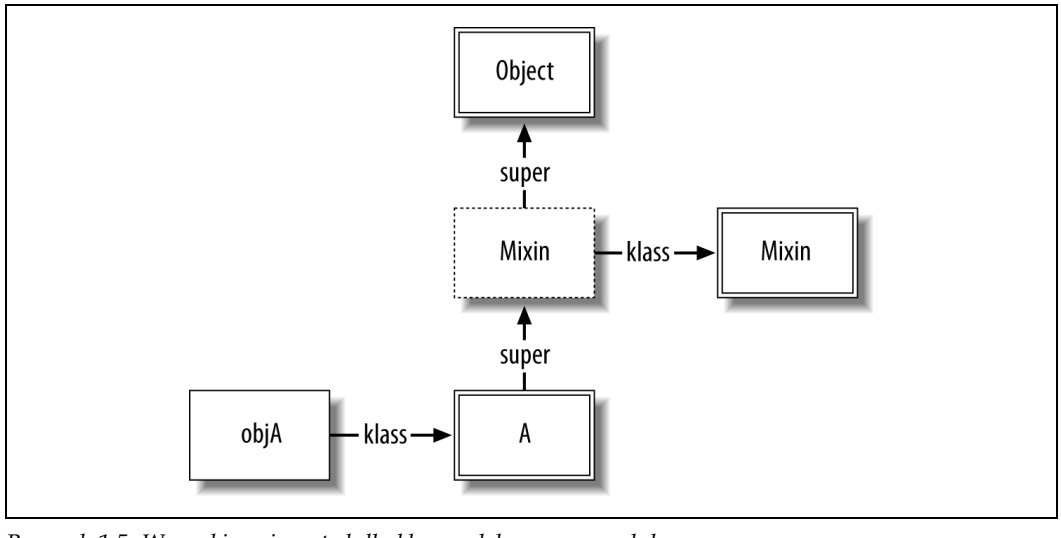



*Rysunek 1.4. Włączenie modułu w łańcuch wyszukiwania*

Na podstawie tego diagramu można łatwo wywnioskować, do czego służą nam klasy pośredniczące — ten sam moduł może zostać dołączony do wielu różnych klas; klasy mogą dziedziczyć po różnych klasach (i przez to mieć inne wskaźniki `super`). Nie możemy bezpośrednio włączyć klasy `Mixin` do łańcucha wyszukiwania, ponieważ jego wskaźnik `super` będzie wskazywał na dwa różne obiekty, jeżeli zostanie dołączony do klas mających różnych rodziców.

Gdy utworzymy obiekt klasy `A`, struktury będą wyglądały jak na rysunku 1.5.

```
objA = A.new
```




*Rysunek 1.5. Wyszukiwanie metod dla klasy z dołączonym modułem*

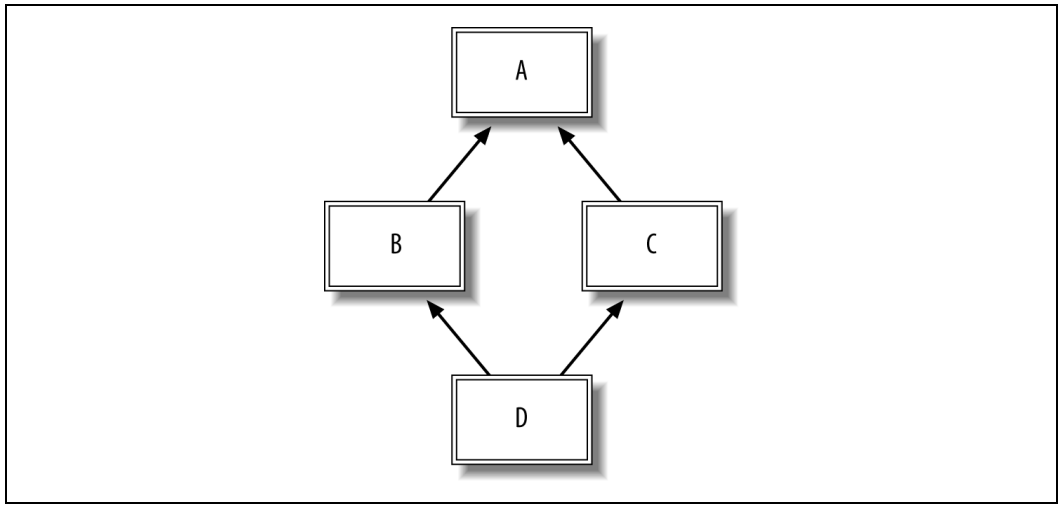
Wywołujemy tu metodę `mixed_method` z obiektu `mixin`, z `objA` jako odbiorcą:

```
objA.mixed_method
# >> Witamy w mixin
```

Wykonywany jest następujący proces wyszukiwania metody:

1. W klasie obiektu `objA`, czyli `A`, wyszukiwana jest pasująca metoda. Żadna nie zostaje znaleziona.
2. Wskaźnik `super` klasy `A` prowadzi do `ICLASS`, który jest pośrednikiem dla `Mixin`. Pasująca metoda jest wyszukiwana w obiekcie pośrednika. Ponieważ tablica `m_tbl` pośrednika jest taka sama jak tablica `m_tbl` klasy `Mixin`, metoda `mixed_method` zostaje odnaleziona i wywołana.

W wielu językach mających możliwość dziedziczenia wielobazowego występuje **problem diamentu**, polegający na braku możliwości jednoznacznego identyfikowania metod obiektów, których klasy mają schemat dziedziczenia o kształcie diamentu, jak jest to pokazane na rysunku 1.6.




*Rysunek 1.6. Problem diamentu przy dziedziczeniu wielobazowym*

Biorąc jako przykład diagram przedstawiony na tym rysunku, jeżeli obiekt klasy `D` wywołuje metodę zdefiniowaną w klasie `A`, która została przesłonięta zarówno w `B`, jak i `C`, nie można jasno określić, która metoda zostanie wywołana. W Ruby problem ten został rozwiązany przez szeregowanie kolejności dołączania. W czasie wywoływania metody łańcuch dziedziczenia jest przeszukiwany liniowo, dołączając wszystkie `ICLASS` dodane do łańcucha.

Trzeba przypomnieć, że Ruby nie obsługuje dziedziczenia wielobazowego; jednak wiele modułów może być dołączonych do klas i innych modułów. Z tego powodu `A`, `B` oraz `C` muszą być modułami. Jak widać, nie występuje tu niejednoznaczność; wybrana zostanie metoda dołączona jako ostatnia do łańcucha wywołania:

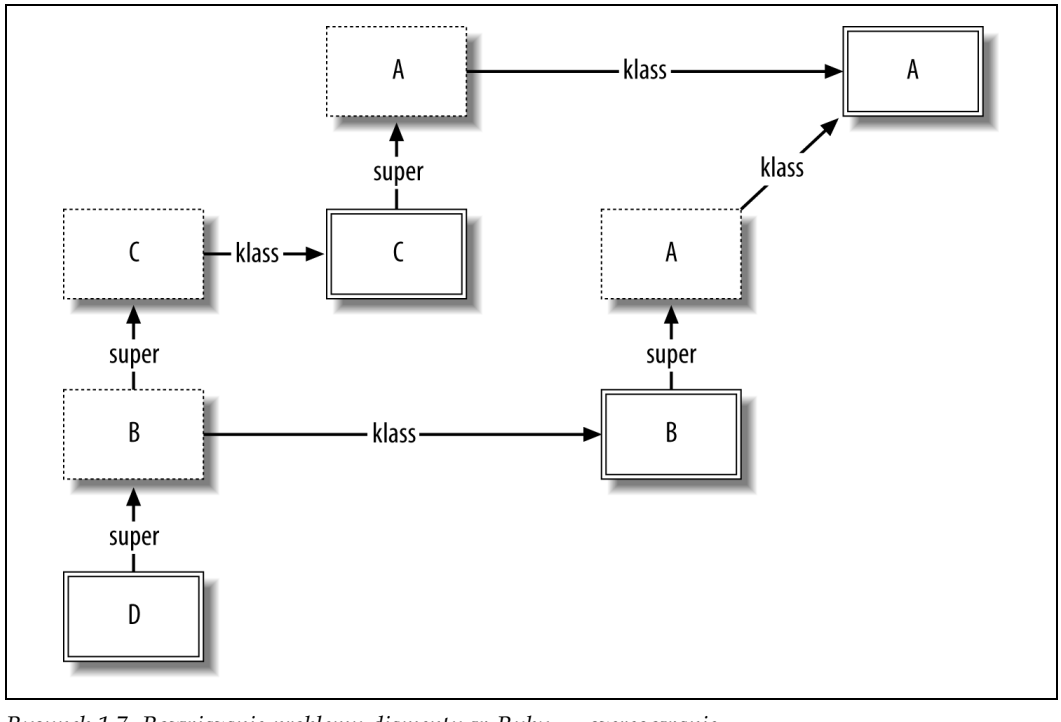
```
module A
  def hello
    "Witamy w A"
  end
end
module B
  include A
  def hello
    "Witamy w B"
  end
```

```
end

module C
  include A
  def hello
    "Witamy w C"
  end
end

class D
  include B
  include C
end

D.new.hello # => "Witamy w C"
```

Jeżeli zmienimy kolejność dołączania, odpowiednio zmieni się wynik:

```
class D
  include C
  include B
end

D.new.hello # => "Witamy w B"
```

W przypadku ostatniego przykładu, gdzie `B` został dołączony jako ostatni, diagram obiektów jest przedstawiony na rysunku 1.7 (dla uproszczenia wskaźniki od `Object` i `Class` zostały usunięte).



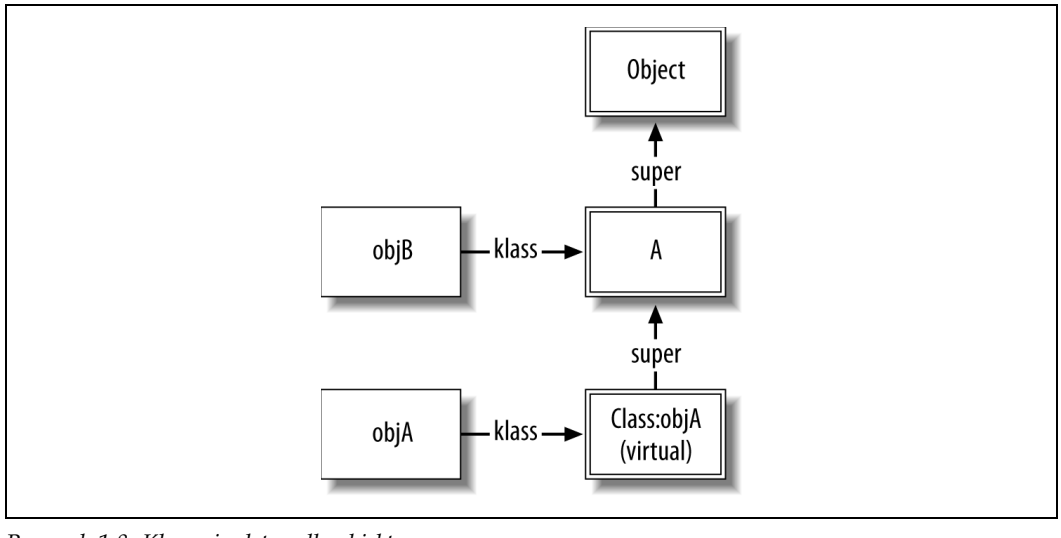

*Rysunek 1.7. Rozwiązanie problemu diamentu w Ruby — szeregowanie*

## Klasa singleton

**Klasy singleton** (również **metaklasy** lub **eigenklasy**; patrz następna ramka, „Terminologia klas singleton") pozwalają na zróżnicowanie działania obiektu w stosunku do innych obiektów danej klasy. Czytelnik prawdopodobnie spotkał się wcześniej z notacją pozwalającą na otwarcie klasy singleton:

```
class A
end

objA = A.new
objB = A.new
objA.to_s # => "#<A:0x1cd0a0>"
objB.to_s # => "#<A:0x1c4e28>"

class <<objA # Otwarcie klasy singleton dla objA
  def to_s; "Obiekt A"; end
end

objA.to_s # => "Obiekt A"
objB.to_s # => "#<A:0x1c4e28>"
```

Zapis `class <<objA` otwiera klasę singleton dla `objA`. Metody instancyjne dodane do klasy singleton funkcjonują jako metody instancyjne w łańcuchu wyszukiwania. Wynikowe struktury danych są przedstawione na rysunku 1.8.




*Rysunek 1.8. Klasa singleton dla obiektu*

Jak zwykle, obiekt `objB` jest klasy `A`. Jeżeli poprosimy Ruby o podanie typu `objA`, okaże się, że jest to również obiekt klasy `A`:

```
objA.class # => A
```

Jednak wewnętrznie obiekt ten działa nieco inaczej. Do łańcucha wyszukiwania zostaje dodana inna klasa. Jest to obiekt klasy singleton dla `objA`. W tej dokumentacji będziemy go nazywać `Class:objA`. Ruby nadaje mu podobną nazwę: `#<Class:#<A:0x1cd0a0>>`. Podobnie jak inne klasy, wskaźnik `klass` klasy singleton (niepokazany) wskazuje na obiekt `Class`.

## Terminologia klas singleton

Termin **metaklasa** nie jest szczególnie precyzyjny w określaniu klas singleton. Nazwanie klasy „meta" wskazuje, że jest ona nieco bardziej abstrakcyjna niż zwykła klasa. W tym przypadku nie ma to miejsca; klasy singleton są po prostu klasami należącymi do określonej instancji.

Prawdziwe metaklasy są dostępne w takich językach, jak Smalltalk, gdzie mamy bogaty protokół metaobiektów. Metaklasy w Smalltalku to klasy, których instancjami są klasy. W przypadku Ruby jedyną metaklasą jest `Class`, ponieważ wszystkie klasy są obiektami `Class`.

Dosyć popularnym alternatywnym terminem dla klasy singleton jest **eigenklasa**, od niemieckiego słowa *eigen* („własny"). Klasa singleton obiektu jest jego eigenklasą (własną klasą).

Klasa singleton zostaje oznaczona jako **klasa wirtualna** (jeden ze znaczników `flags` wskazuje, że klasa jest wirtualna). Klasy wirtualne nie mogą być konkretyzowane i zwykle nie są wykorzystywane w Ruby, o ile nie zadamy sobie trudu, aby ich użyć. Gdy chcieliśmy określić klasę obiektu `objA`, Ruby wykorzystywał wskaźniki `klass` i `super` w hierarchii, aż do momentu znalezienia pierwszej klasy niewirtualnej.

Z tego powodu uzyskaliśmy odpowiedź, że klasą `objA` jest `A`. Ważne jest, aby to zapamiętać — klasa obiektu (z perspektywy Ruby) może nie odpowiadać obiektowi, na który wskazuje `klass`.

Klasy singleton są tak nazwane nie bez powodu — w obiekcie może być zdefiniowana tylko jedna taka klasa. Dzięki temu możemy bez niejednoznaczności odwoływać się do klasy singleton `objA` lub `Class:objA`. W naszym kodzie możemy założyć, że klasa singleton istnieje; w rzeczywistości Ruby tworzy ją w momencie pierwszego wywołania.

Ruby pozwala na definiowanie klas singleton w dowolnych obiektach poza `Fixnum` oraz symbolach. Symbole oraz `Fixnum` są **wartościami natychmiastowymi** (dla zapewnienia odpowiedniej wydajności są przechowywane w pamięci bezpośrednio, a nie jako wskaźniki do struktur danych). Ponieważ są one przechowywane w całości, nie posiadają wskaźników `klass`, więc nie ma możliwości zmiany łańcucha wyszukiwania metod.

Można również otworzyć klasę singleton dla `true`, `false` oraz `nil`, ale zwracana będzie ta sama klasa singleton co klasa obiektu. Wartościami są obiekty singleton (jedyne instancje), odpowiednio `TrueClass`, `FalseClass` oraz `NilClass`. Gdy odwołamy się do klasy singleton dla `true`, otrzymamy `TrueClass`, ponieważ jest to jedyna możliwa instancja tej klasy. W Ruby:

```
true.class # => TrueClass
class << true; self; end # => TrueClass
true.class == (class << true; self; end) # => true
```

### Klasy singleton i obiekty klas

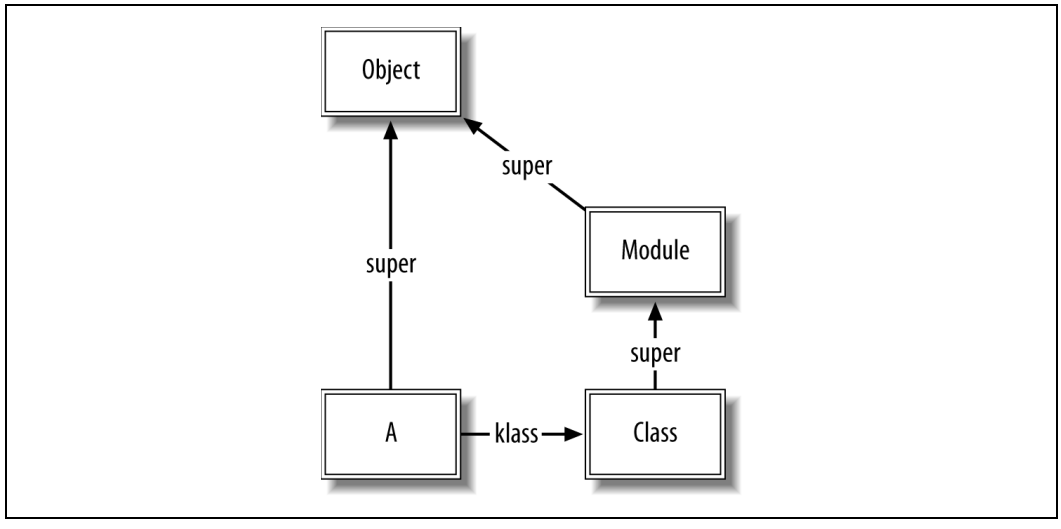
Teraz sprawy się komplikują. Należy pamiętać o podstawowej zasadzie wyszukiwania metod — na początku Ruby przechodzi po wskaźnikach `klass` i wyszukuje metody; następnie korzysta z wskaźników `super` do przeglądania łańcucha, aż do znalezienia odpowiedniej metody lub osiągnięcia końca łańcucha.

Ważne jest, aby pamiętać, że *klasy są również obiektami*. Tak jak zwykłe obiekty mogą mieć klasę singleton, tak samo obiekty klas mogą również posiadać klasy singleton. Te klasy singleton, podobnie jak inne klasy, mogą posiadać metody. Ponieważ klasy singleton są dostępne za

pomocą wskaźnika `klass` z obiektu klasy, metody instancji klasy singleton są metodami klasy właściciela singletonu.

Pełny zbiór struktur danych poniższego kodu jest pokazany na rysunku 1.9.

```
class A
end
```




*Rysunek 1.9. Pełny zbiór struktur danych jednej klasy*

Klasa `A` dziedziczy po `Object`. Obiekt klasy `A` jest typu `Class`. `Class` dziedziczy po `Module`, który z kolei dziedziczy po `Object`. Metody zapisane w tablicy `m_tbl` klasy `A` są metodami instancyjnymi `A`. Co się więc stanie, gdy wywołamy metodę klasową z `A`?

```
A.to_s # => "A"
```

Stosowane są te same zasady wyszukiwania, przy użyciu `A` jako odbiorcy (należy pamiętać, że `A` jest stałą wartościowaną jako obiekt klasy `A`). Na początek Ruby korzysta ze wskaźnika `klass` pomiędzy `A` a `Class`. W tablicy `m_tbl` `Class` wyszukiwana jest funkcja o nazwie `to_s`. Ponieważ nic nie zostało znalezione, Ruby przechodzi za pomocą wskaźnika `super` z `Class` do `Module`, gdzie zostaje odnaleziona funkcja `to_s` (w kodzie natywnym, `rb_mod_to_s`).

Nie powinno być to niespodzianką. Nie ma tu żadnej magii. Metody klasowe są wyszukiwane w ten sam sposób co metody instancyjne — jedyną różnicą jest to, że odbiorcą jest klasa, a nie instancja klasy.

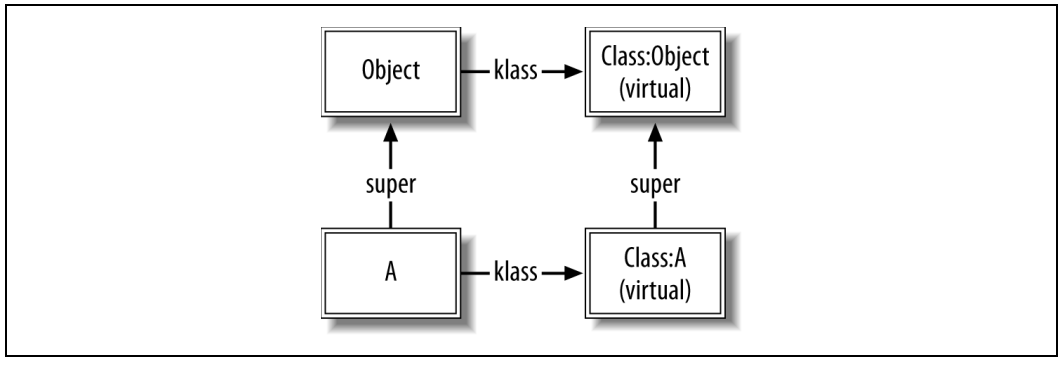
Teraz, gdy wiemy, w jaki sposób są wyszukiwane metody, możemy wnioskować, że możemy zdefiniować metodę klasową dla dowolnej klasy przez zdefiniowanie metody instancyjnej obiektu `Class` (aby wstawić go do `m_tbl` `Class`). Faktycznie — to działa:

```
class A; end
# z Module#to_s
A.to_s # => "A"

class Class
  def to_s; "Class#to_s"; end
end

A.to_s # => "Class#to_s"
```

Jest to interesująca sztuczka, ale o ograniczonej użyteczności. Zwykle chcemy zdefiniować osobne metody klasowe dla każdej z klas. W takim przypadku można wykorzystać klasy singleton dla obiektów klasowych. Aby otworzyć klasę singleton dla klasy, należy po prostu użyć nazwy klasy w notacji klasy singleton:

```
class A; end
class B; end

class <<A
  def to_s; "Klasa A"; end
end

A.to_s # => "Klasa A"
B.to_s # => "B"
```

Wynikowe struktury danych są przedstawione na rysunku 1.10. Dla uproszczenia klasa `B` jest pominięta.




*Rysunek 1.10. Klasa singleton dla klasy*

Metoda `to_s` została dodana do klasy singleton dla `A` lub `Class:A`. Teraz, gdy zostanie wywołana metoda `A.to_s`, Ruby skorzysta z wskaźnika `klass` do `Class:A` i wywoła z niej odpowiednią metodę.

W definicji metody znajduje się jeszcze jeden problem. W definicji klasy lub modułu `self` zawsze wskazuje na obiekt klasy lub modułu:

```
class A
  self # => A
end
```

Tak więc `class<<A` wewnątrz definicji klasy `A` może być zapisane jako `class<<self`, ponieważ `self` wewnątrz definicji `A` wskazuje na ten sam obiekt. Ten idiom jest używany wszędzie w Rails do definiowania metod klasowych. Poniższy przykład przedstawia wszystkie sposoby definiowania metod klasowych.

```
class A
  def A.class_method_one; "Metoda klasowa"; end

  def self.class_method_two; "Również metoda klasowa"; end

  class <<A
    def class_method_three; "Nadal metoda klasowa";
  end
end
```

```
  class <<self
    def class_method_four; "Kolejna metoda klasowa"; end
  end
end

  def A.class_method_five
    "To działa poza definicją klasy"
  end

  class <<A
    def A.class_method_six
      "Metaklasę można otworzyć poza definicją klasy"
    end
  end

  # Drukuje po kolei wyniki wywołania każdej metody.
  %w(one two three four five six).each do |number|
    puts A.send(:"class_method_#{number}")
  end

  # >> Metoda klasowa
  # >> Również metoda klasowa
  # >> Nadal metoda klasowa
  # >> Kolejna metoda klasowa
  # >> To działa poza definicją klasy
  # >> Metaklasę można otworzyć poza definicją klasy
```

Oznacza to również, że wewnątrz definicji klasy singleton — podobnie jak w każdej innej definicji klasy — `self` nadal wskazuje na obiekt definiowanej klasy. Gdy pamiętamy, że ta wartość w definicji bloku lub klasy jest wartością ostatniej wykonanej instrukcji, to wiemy, że wartością `class <<objA; self; end` jest obiekt klasa singleton `objA`. Konstrukcja `class <<objA` otwiera klasę singleton, a `self` (klasa singleton) jest zwracany z definicji klasy.

Łącząc to wszystko, możemy otworzyć klasę `Object` i dodać metodę instancyjną do każdego obiektu, który zwraca klasę singleton obiektu:

```
class Object
     def metaclass
    class <<self
       self
    end
  end
 end
```

Metoda ta tworzy podstawy metaid, o czym wkrótce.

## Brakujące metody

Po całym tym zamieszaniu `method_missing` jest dosyć prosta. Istnieje tylko jedna reguła — jeżeli cała procedura wyszukiwania metod zawiedzie, wyszukiwanie metody jest wykonywane ponownie; szukana jest tym razem metoda `method_missing` zamiast początkowej metody. Jeżeli metoda zostanie znaleziona, wywoływana jest z argumentami oryginalnej metody, z dołączoną nazwą metody. Przekazywany jest również każdy blok.

Domyślna metoda `method_missing` z `Object (rb_method_missing)` zgłasza wyjątek.

## Metaid

Autorem niewielkiej biblioteki o nazwie *metaid.rb*, wspomagającej metaprogramowanie w Ruby, jest *why the lucky stiff*. Jest ona na tyle użyteczna, aby dołączać ją do każdego projektu, w którym potrzebne jest metaprogramowanie[7]:

```
class Object
  # Ukryty singleton śledzi każdego.
  def metaclass; class << self; self; end; end
  def meta_eval &blk; metaclass.instance_eval &blk; end

  # Dodanie metody do metaklasy.
  def meta_def name, &blk
    meta_eval { define_method name, &blk }
  end

  # Definiowanie metody instancyjnej wewnątrz klasy.
  def class_def name, &blk
    class_eval { define_method name, &blk }
  end
end
```

W każdym obiekcie biblioteka ta definiuje cztery metody:

`metaclass`
: Odwołuje się do klasy singletonu odbiorcy (`self`).

`meta_eval`
: Odpowiednik `class_eval` dla klas singletonów. Wartościuje dany blok w kontekście klasy singletonu odbiorcy.

`meta_def`
: Definiuje metodę w klasie singleton odbiorcy. Jeżeli odbiorca jest klasą lub modułem, spowoduje to utworzenie metody klasowej (metody instancyjnej klasy singleton odbiorcy).

`class_def`
: Definiuje metodę instancyjną odbiorcy (który musi być klasą lub modułem).

Korzystanie z metaid jest tak proste, ponieważ zastosowano w niej znaczne uproszczenia. Przez wykorzystanie skrótu do odwoływania się i rozszerzania metaklas nasz kod staje się bardziej czytelny, ponieważ nie jest zatłoczony konstrukcjami takimi jak `class << self; self; end`. Im krótszy i czytelny jest kod realizujący daną technikę, tym bardziej prawdopodobne jest, że użyjemy go we właściwy sposób w naszym kodzie.

Poniższy przykład pokazuje zastosowanie metaid do uproszczenia naszej klasy singleton:

```
class Person
  def name; "Bob"; end
  def self.species; "Homo sapiens"; end
end
```

Metody klasowe są dodawane jako metody instancyjne klasy singleton:

```
Person.instance_methods(false)            # => ["name"]
Person.metaclass.instance_methods -
  Object.metaclass.instance_methods       # => ["species"]
```

[7] *Seeing Metaclasses Clearly*: *http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html*.

Przy użyciu metod z metaid możemy napisać nasze definicje metod w następujący sposób:

```
Person.class_def(:name) { "Bob" }
Person.meta_def(:species) { "Homo sapiens" }
```

## Wyszukiwanie zmiennych

W Ruby występują cztery rodzaje zmiennych — zmienne globalne, zmienne klasowe, zmienne instancyjne oraz zmienne lokalne[8]. Zmienne globalne są przechowywane globalnie, a zmienne lokalne są przechowywane leksykalnie, więc nie będą one przedmiotem naszej dyskusji, ponieważ nie wykorzystują systemu klas Ruby.

Zmienne instancyjne są specyficzne dla określonego obiektu. Są one prefiksowane za pomocą symbolu `@`: `@price` jest zmienną instancyjną. Ponieważ każdy obiekt Ruby ma strukturę `iv_tbl`, każdy obiekt może posiadać zmienne instancyjne.

Ponieważ każda klasa jest również obiektem, klasy również mogą posiadać zmienne instancyjne. W poniższym przykładzie kodu przedstawiony jest sposób odwołania do zmiennej instancyjnej klasy:

```
class A
  @ivar = "Zmienna instancyjna klasy A"
end

A.instance_variable_get(:@ivar) # => "Zmienna instancyjna klasy A"
```

Zmienne instancyjne są zawsze wyszukiwane na podstawie obiektu wskazywanego przez `self`. Ponieważ `self` jest obiektem klasowym `A` w definicji klasy `A ... end`, `@ivar` należy do obiektu klasowego `A`.

Zmienne klasowe są inne. Do zmiennych klasowych (które zaczynają się od `@@`) może odwoływać się każda zmienna klasowa. Zmienne klasowe mogą być również wykorzystywane w samej definicji klasy. Choć zmienne klasowe i instancyjne są podobne, nie są one tym samym:

```
class A
  @var = "Zmienna instancyjna klasy A"
  @@var = "Zmienna klasowa klasy A"

  def A.ivar
    @var
  end

  def A.cvar
    @@var
  end
end

A.ivar # => "Zmienna instancyjna klasy A"
A.cvar # => "Zmienna klasowa klasy A"
```

W tym przykładzie `@var` oraz `@@var` są przechowywane w tym samym miejscu — w tablicy `iv_tbl` klasy `A`. Są to jednak inne zmienne, ponieważ mają one inne nazwy (symbole `@` są dołączane do nazwy zmiennej przy przechowywaniu). Funkcje Ruby do odwoływania się do zmiennych instancyjnych i klasowych sprawdzają, czy przekazywane nazwy są we właściwym formacie:

[8] Istnieją również stałe, ale nie ma to w tej chwili większego znaczenia.

```
A.instance_variable_get(:@@var)
# ~> -:17:in 'instance_variable_get': '@@var' is not allowed as an instance
    variable name (NameError)
```

Zmienne klasowe są nieco mylące w użyciu. Są one współdzielone w całej hierarchii dziedziczenia, więc klasa pochodna modyfikująca zmienną klasową modyfikuje również zmienną klasową rodzica.

```
>> class A; @@x = 3 end
=> 3
>> class B < A; @@x = 4 end
=> 4
>> class A; @@x end
=> 4
```

Może to być zarówno przydatne, jak i mylące. Generalnie potrzebujemy albo zmiennych instancyjnych — które są niezależne od hierarchii dziedziczenia — albo dziedziczonych atrybutów klasy zapewnianych przez ActiveSupport, które propagują wartości w kontrolowany, dobrze zdefiniowany sposób.

## Bloki, metody i procedury

Jedną z zaawansowanych funkcji Ruby jest możliwość wykorzystywania fragmentów kodu jako obiektów. Do tego celu wykorzystuje się trzy klasy:

`Proc`
: Klasa `Proc` reprezentuje blok kodu — fragment kodu, który może być wywoływany z argumentami i może zwracać wartość.

`UnboundMethod`
: Jest ona podobna do `Proc`; reprezentuje metodę instancyjną określonej klasy (należy pamiętać, że metoda klasowa jest również metodą instancyjną obiektu klasowego, więc `UnboundMethods` może reprezentować również metody klasowe). `UnboundMethod` musi być związana z klasą przed wywołaniem.

`Method`
: Obiekty `Method` są obiektami `UnboundMethod`, które zostały związane z obiektem za pomocą `UnboundMethod#bind`. Można je również uzyskać za pomocą `Object#method`.

Przeanalizujemy teraz kilka sposobów na uzyskanie obiektów `Proc` oraz `Method`. Jako przykładu użyjemy metody `Fixnum#+`. Zwykle wywołujemy ją przy pomocy uproszczonej składni:

```
3 + 5 # => 8
```

Można jednak użyć wywołania metody instancyjnej z obiektu `Fixnum`, tak samo jak innych metod instancyjnych:

```
3.+(5) # => 8
```

Do uzyskania obiektu reprezentującego metodę instancyjną można wykorzystać metodę `Object#method`. Metoda ta będzie związana z obiektem, na którym została wywołana, czyli `3`.

```
add_3 = 3.method(:+)
add_3 # => #<Method: Fixnum#+>
```

Metoda ta może być skonwertowana do `Proc` lub wywołana bezpośrednio z argumentami:

```
add_3.to_proc # => #<Proc:0x00024b08@-:6>
add_3.call(5) # => 8
#  Metoda#[] jest wygodnym synonimem dla Metoda#call.
add_3[5] # => 8
```

Istnieją dwa sposoby na uzyskanie metody niezwiązanej. Możemy wywołać `instance_method` na obiekcie klasy:

```
add_unbound = Fixnum.instance_method(:+)
add_unbound # => #<UnboundMethod: Fixnum#+>
```

Można również odłączyć metodę, która została wcześniej związana z obiektem:

```
add_unbound == 3.method(:+).unbind # => true
add_unbound.bind(3).call(5) # => 8
```

Możemy związać `UnboundMethod` z dowolnym obiektem tej samej klasy:

```
add_unbound.bind(15)[4] # => 19
```

Jednak dołączany obiekt musi być instancją tej samej klasy, ponieważ w przeciwnym razie otrzymamy `TypeError`:

```
add_unbound.bind(1.5)[4] # =>
# ~> -:16:in 'bind': bind argument must be an instance of Fixnum (TypeError)
# ~> from -:16
```

Otrzymaliśmy ten błąd, ponieważ + jest zdefiniowany w `Fixnum`; dlatego obiekt `UnboundMethod`, jaki otrzymujemy, musi być związany z obiektem, który jest `kind_of?(Fixnum)`. Gdyby metoda + była zdefiniowana w `Numeric` (z którego dziedziczą `Fixnum` oraz `Float`), wcześniejszy kod zwróciłby `5.5`.

## Bloki na procedury i procedury na bloki

Bieżąca implementacja Ruby ma wyraźną wadę — bloki nie zawsze są obiektami `Proc` i odwrotnie. Zwykłe bloki (tworzone za pomocą `do...end` oraz `{}`) muszą być dołączane do wywołania metody i nie są automatycznie obiektami. Nie można na przykład zapisać `code_ block ={puts"abc"}`. Przydają się tu funkcje `Kernel#lambda` i `Proc.new`, które konwertują bloki na obiekty `Proc`[9].

```
block_1 = lambda { puts "abc" } # => #<Proc:0x00024914@-:20>
block_2 = Proc.new { puts "abc" } # => #<Proc:0x000246a8@-:21>
```

Pomiędzy `Kernel#lambda` i `Proc.new` występuje niewielka różnica. Powrót z obiektu `Proc` utworzonego za pomocą `Kernel#lambda` powoduje zwrócenie wyniku do funkcji wywołującej; powrót z obiektu `Proc` utworzonego za pomocą `Proc.new` powoduje próbę wykonania powrotu `z` funkcji wywołującej, a jeżeli nie jest to możliwe, zgłaszany jest `LocalJumpError`. Poniżej pokazany jest przykład:

```
def block_test
  lambda_proc = lambda { return 3 }
  proc_new_proc = Proc.new { return 4 }

  lambda_proc.call # => 3
  proc_new_proc.call # =>

  puts "Nigdy nie zostanie wywołane"
end

block_test # => 4
```

Instrukcja powrotu w `lambda_proc` zwraca wartość `3`. W przypadku `proc_new_proc` instrukcja powrotu powoduje wyjście z funkcji wywołującej `block_test` — dlatego wartość `4` jest zwra-

[9] `Kernel#proc` jest inną nazwą dla `Kernel#lambda`, ale jest ona przestarzała.

cana przez `block_test`. Instrukcja `puts` nie zostanie nigdy wykonana, ponieważ instrukcja `proc_new_proc.call` spowoduje wcześniejsze zakończenie `block_test`.

Bloki mogą być konwertowane do obiektów `Proc` przez przekazanie ich do funkcji przy wykorzystaniu `&` w parametrach formalnych funkcji:

```
def some_function(&b)
  puts "Blokiem jest #{b}, który zwraca #{b.call}"
end

some_function { 6 + 3 }
# >> Blokiem jest #<Proc:0x00025774@-:7>, który zwraca 9
```

Można również zastąpić `Proc` za pomocą `&`, jeżeli funkcja oczekuje bloku:

```
add_3 = lambda {|x| x+3}
(1..5).map(&add_3) # => [4, 5, 6, 7, 8]
```

## Zamknięcia

*Zamknięcia* (ang. *closure*) są tworzone w przypadku, gdy blok lub obiekt `Proc` odwołuje się do zmiennej zdefiniowanej poza ich zakresem. Pomimo tego, że blok zawierający może wyjść z zakresu, zmienne są utrzymywane do momentu wyjścia z zakresu przez odwołujący się do nich blok lub obiekt `Proc`. Uproszczony przykład, pomimo że niezbyt praktyczny, demonstruje tę zasadę:

```
def get_closure
  data = [1, 2, 3]
  lambda { data }
end
block = get_closure
block.call # => [1, 2, 3]
```

Funkcja anonimowa (`lambda`) zwracana przez `get_closure` odwołuje się do danych ze zmiennej lokalnej, która jest zdefiniowana poza jej zakresem. Dopóki zmienna `block` znajduje się w zakresie, będzie przechowywała własną referencję do `data`, więc instancja `data` nie zostanie usunięta (pomimo tego, że funkcja `get_closure` zakończyła się). Należy zwrócić uwagę, że przy każdym wywołaniu `get_closure`, `data` odwołuje się do innej zmiennej (ponieważ jest lokalna dla funkcji):

```
block = get_closure
block2 = get_closure

block.call.object_id # => 76200
block2.call.object_id # => 76170
```

Klasycznym przykładem zamknięcia jest funkcja `make_counter`, która zwraca funkcję licznika (`Proc`), która po uruchomieniu zwiększa i zwraca ten licznik. W Ruby funkcja `make_counter` może być zaimplementowana w następujący sposób:

```
def make_counter(i=0)
  lambda { i += 1 }
end

x = make_counter
x.call # => 1
x.call # => 2

y = make_counter
y.call # => 1
y.call # => 2
```

Funkcja `lambda` tworzy zamknięcie obejmujące bieżącą wartość zmiennej lokalnej `i`. Nie tylko można odwoływać się do zmiennych, ale można również modyfikować jej wartości. Każde zamknięcie uzyskuje osobną instancję zmiennej (ponieważ jest to zmienna lokalna dla każdej z instancji `make_counter`). Ponieważ `x` oraz `y` zawierają referencje do innych instancji zmiennej lokalnej `i`, mają one inny stan.

# Techniki metaprogramowania

Po omówieniu podstaw Ruby przedstawimy kilka powszechnie stosowanych technik metaprogramowania wykorzystywanych w tym języku.

Choć przykłady są napisane w Ruby, większość z technik można wykorzystać w dowolnym dynamicznym języku programowania. W rzeczywistości wiele z idiomów metaprogramowania stosowanych w Ruby jest bezwstydnie skradzionych z języków Lisp, Smalltalk lub Perl.

## Opóźnienie wyszukiwania metod do czasu wykonania

Czasami chcemy utworzyć interfejs, którego metody są zależne od danych dostępnych w czasie wykonywania programu. Najważniejszym przykładem takiej konstrukcji są metody akcesorów atrybutów w `ActiveRecord` dostępne w Rails. Wywołania metod obiektu `ActiveRecord` (tak jak `person.name`) są modyfikowane w czasie działania na odwołania do atrybutów. Na poziomie metod klasy `ActiveRecord` oferuje niezwykłą elastyczność — wyrażenie `Person.find_all_by_user_id_and_active(42, true)` jest zamieniane na odpowiednie zapytanie SQL, a dodatkowo, jeżeli rekord nie zostanie znaleziony, zgłaszany jest wyjątek `NoMethodError`.

Umożliwia to metoda `method_missing` dostępna w Ruby. Gdy na obiekcie zostanie wywołana nieistniejąca metoda, Ruby przed zgłoszeniem wyjątku `NoMethodError` wyszukuje w klasie obiektu metodę `method_missing`. Pierwszym argumentem `method_missing` jest nazwa wywoływanej metody; pozostałe argumenty odpowiadają argumentom przekazanym do metody. Każdy blok przekazany do metody jest również przekazywany do `method_missing`. Tak więc kompletna sygnatura tej metody jest następująca:

```
def method_missing(method_id, *args, &block)
  ...
end
```

Istnieje kilka wad wykorzystywania `method_missing`:

- Jest wolniejsza niż konwencjonalne wyszukiwanie metod. Proste testy pokazują, że wyszukiwanie metod za pomocą `method_missing` jest co najmniej dwa do trzech razy bardziej czasochłonne niż konwencjonalne wyszukiwanie.
- Ponieważ wywoływana metoda nigdy faktycznie nie istnieje — jest po prostu przechwytywana w ostatnim kroku procesu wyszukiwania metod — nie może być dokumentowana lub poddawana introspekcji, jak konwencjonalne metody.
- Ponieważ wszystkie metody dynamiczne muszą przechodzić przez metodę `method_missing`, może ona znacznie urosnąć, jeżeli w kodzie znajduje się wiele miejsc wymagających dynamicznego dodawania metod.
- Zastosowanie `method_missing` ogranicza zgodność z przyszłymi wersjami API. Gdy będziemy polegać na metodzie `method_missing` przy obsłudze niezdefiniowanych metod, wprowadzenie nowych metod w przyszłych wersjach API może zmienić oczekiwania naszych użytkowników.

Dobrą alternatywą jest podejście zastosowane w funkcji `generate_read_methods` z `Active ↪Record`. Zamiast czekać na przechwycenie wywołania przez `method_missing`, `ActiveRecord` generuje implementacje dla metod odczytu i modyfikacji atrybutu, dzięki czemu mogą być one wywoływane przez konwencjonalny mechanizm wyszukiwania metod.

Jest to bardzo wydajna metoda, a dynamiczna natura Ruby pozwala na napisanie metod, które przy pierwszym wywołaniu wymieniają się na swoje zoptymalizowane wersje. Jest to używane w routingu Ruby, który musi być bardzo szybki; zastosowanie tej metody jest przedstawione w dalszej części rozdziału.

## Programowanie generacyjne — tworzenie kodu na bieżąco

Jedną z efektywnych technik, która składa się z kilku kolejnych, jest **programowanie generacyjne** — pisanie kodu tworzącego kod.

Technika ta może być zastosowana do bardzo prostych zadań, takich jak pisanie skryptu automatyzującego niektóre nudne części programowania. Można na przykład wypełnić przypadki testowe dla każdego z użytkowników:

```
brad_project:
  id: 1
  owner_id: 1
  billing_status_id: 12

john_project:
  id: 2
  owner_id: 2
  billing_status_id: 4

...
```

Jeżeli byłby to język bez możliwości zastosowania skryptów do definiowania przypadków testowych, konieczne byłoby napisanie ich ręcznie. Może to zacząć sprawiać problemy, gdy dane przekroczą masę krytyczną, a jest niemal niemożliwe, gdy przypadki testowe mają dziwne zależności w danych źródłowych. Programowanie generacyjne pozwala napisać skrypt do generowania tych przypadków użycia na podstawie danych źródłowych. Choć nie jest to idealne rozwiązanie, jest to znaczne usprawnienie w stosunku do pisania przypadków użycia ręcznie. Występuje tu jednak problem z utrzymaniem — konieczne jest włączenie skryptu w proces tworzenia oraz zapewnienie, że przypadki testowe są regenerowane w momencie zmiany danych źródłowych.

Jest to (na szczęście) rzadko, o ile w ogóle potrzebne w Ruby on Rails. Niemal w każdym aspekcie konfiguracji aplikacji Rails można stosować skrypty, co jest spowodowane w większości przez zastosowanie wewnętrznych języków specyficznych dla domeny (DSL — ang. *Domain Specific Language*). W wewnętrznym DSL można mieć do dyspozycji wszystkie możliwości języka Ruby, nie tylko określony interfejs biblioteki, jaki autor zdecydował się udostępnić.

Wracając do poprzedniego przykładu, ERb (ang. *Embedded Ruby*) znacznie ułatwia pracę. Można wstrzyknąć dowolny kod Ruby na początek pliku YAML[10] z użyciem znaczników ERb `<% %>` oraz `<%= %>`, dołączając tam dowolną logikę:

[10] Uniwersalny język formalny przeznaczony do reprezentowania różnych danych w ustrukturyzowany sposób. Słowo YAML to akronim rekursywny od słów YAML Ain't Markup Language. Pierwotnie interpretowano ten skrót jako Yet Another Markup Language. Pierwsza wersja zaproponowana została w 2001 roku przez Clarka Evansa we współpracy z Ingy döt Net oraz Oren Ben-Kiki — *przyp. red.*

```
<% User.find_all_by_active(true).each_with_index do |user, i| %>
<%= user.login %>_project:
     id: <%= i %>
     owner_id: <%= user.id %>
     billing_status_id: <%= user.billing_status.id %>

<% end %>
```

Implementacja tego wygodnego mechanizmu w `ActiveRecord` nie może być prostsza:

```
yaml = YAML::load(erb_render(yaml_string))
```

przy wykorzystaniu metody pomocniczej `erb_render`:

```
def erb_render(fixture_content)
  ERB.new(fixture_content).result
end
```

W programowaniu generacyjnym często wykorzystuje się `Module#define_method` lub `class_eval` oraz `def` do tworzenia metod na bieżąco. Technika ta jest wykorzystywana do akcesorów atrybutów; funkcja `generate_read_methods` definiuje metody do modyfikacji i odczytu jako metody instancyjne klasy `ActiveRecord` w celu zmniejszenia liczby wywołań metody `method_missing` (która jest dosyć kosztowną techniką).

## Kontynuacje

**Kontynuacje** są bardzo efektywną techniką kontroli przepływu sterowania. Kontynuacja reprezentuje określony stan stosu wywołań oraz zmiennych leksykalnych. Jest to migawka wykonana w momencie, gdy Ruby wykonuje dany fragment kodu. Niestety, w implementacji Ruby 1.8 implementacja kontynuacji jest tak powolna, że technika ta jest bezużyteczna w wielu aplikacjach. W kolejnych wersjach maszyn wirtualnych Ruby 1.9 sytuacja może się poprawić, ale nie można się spodziewać dobrej wydajności działania kontynuacji w Ruby 1.8. Jest to jednak bardzo użyteczna konstrukcja, a biblioteki WWW bazujące na kontynuacjach są interesującą alternatywą dla bibliotek takich jak Rails, więc przedstawimy tu ich zastosowanie.

Kontynuacje są tak efektywne z kilku powodów:

- Kontynuacje są po prostu obiektami; mogą być one przekazywane z funkcji do funkcji.
- Kontynuacje mogą być wywoływane z dowolnego miejsca. Jeżeli mamy referencję kontynuacji, możemy ją wywołać.
- Kontynuacje mogą być używane wielokrotnie. Można je wielokrotnie wykorzystywać do powrotu z funkcji.

Kontynuacje są często przedstawiane jako „strukturalne `GOTO`". Z tego powodu powinny być traktowane z taką samą uwagą jak każda inna konstrukcja `GOTO`. Kontynuacje mają niewielkie lub żadne zastosowanie w kodzie aplikacji; powinny być ukryte w bibliotece. Nie uważam, że należy chronić programistów przed nimi samymi. Chodzi o to, że kontynuacje mają większy sens przy tworzeniu abstrakcji niż przy bezpośrednim wykorzystaniu. Gdy programista buduje aplikację, powinien myśleć o „zewnętrznym iteratorze" lub „koprocedurze" (obie te abstrakcje są zbudowane za pomocą kontynuacji), a nie o „kontynuacji".

Seaside[11] jest biblioteką WWW dla języka Smalltalk, która bazuje na kontynuacjach. Są one wykorzystywane w Seaside do zarządzania stanem sesji. Każdy z użytkowników odpowiada kontynuacji na serwerze. Gdy zostanie odebrane żądanie, wywoływana jest kontynuacja i wykonywany jest dalszy kod. Dzięki temu cała transakcja może być zapisana jako jeden strumień kodu pomimo tego, że może ona składać się z wielu żądań HTTP. Biblioteka ta korzysta z tego, że kontynuacje w Smalltalku są serializowalne; mogą być one zapisane do bazy danych lub w systemie plików, a następnie po odebraniu żądania pobierane i ponownie wywoływane. Kontynuacje w Ruby nie są serializowalne. W Ruby kontynuacje są tylko obiektami pamięciowymi i nie mogą być transformowane do strumienia bajtów.

Borges (*http://borges.rubyforge.org*) jest prostym przeniesieniem Seaside 2 do Ruby. Główną różnicą pomiędzy Seaside a Borges jest to, że biblioteka Borges musi przechowywać wszystkie bieżące kontynuacje w pamięci, ponieważ nie są one serializowalne. To znaczne ograniczenie uniemożliwia stosowanie biblioteki Borges do aplikacji WWW o jakimkolwiek obciążeniu. Jeżeli w jednej z implementacji Ruby powstanie mechanizm serializowania kontynuacji, ograniczenie to zostanie usunięte.

Efektywność kontynuacji przedstawia poniższy kod przykładowej aplikacji Borges, która generuje listę elementów z magazynu dostępnego online:

```
class SushiNet::StoreItemList < Borges::Component

  def choose(item)
    call SushiNet::StoreItemView.new(item)
  end

  def initialize(items)
    @batcher = Borges::BatchedList.new items, 8
  end

  def render_content_on(r)
    r.list_do @batcher.batch do |item|
      r.anchor item.title do choose item end
  end

    r.render @batcher
      end

  end # class SushiNet::StoreItemList
```

Większość akcji jest wykonywana w metodzie `render_content_on`, która korzysta z `BatchedList` (do stronicowania) w celu wygenerowania stronicowanej listy łączy do produktów. Jednak cała zabawa zaczyna się od wywołania `anchor`, w którym jest przechowywane wywołanie do wykonania po kliknięciu odpowiedniego łącza.

Nie ma jednak zgody, w jaki sposób wykorzystywać kontynuacje do programowania WWW. HTTP został zaprojektowany jako protokół bezstanowy, a kontynuacje dla transakcji WWW są całkowitym przeciwieństwem bezstanowości. Wszystkie kontynuacje muszą być przechowywane na serwerze, co zajmuje pamięć i przestrzeń na dysku. Wymagane są również *dołączające sesje* do kierowania wywołań użytkownika na ten sam serwer. W wyniku tego, jeżeli jeden z serwerów zostanie wyłączony, wszystkie jego sesje zostają utracone. Najbardziej popularna aplikacja Seaside, DabbleDB (*http://dabbledb.com*) wykorzystuje kontynuacje w bardzo małym stopniu.

[11] *http://seaside.st*.

## Dołączenia

**Dołączenia** zapewniają kontekst dla wartościowania w kodzie Ruby. Dołączenia to zbiór zmiennych i metod, które są dostępne w określonym (leksykalnym) punkcie kodu. Każde miejsce w kodzie Ruby, w którym są wartościowane instrukcje, posiada dołączenia i mogą być one pobrane za pomocą `Kernel#binding`. Dołączenia są po prostu obiektami klasy `Binding` i mogą być one przesyłane tak jak zwykłe obiekty:

```
class C
  binding # => #<Binding:0x2533c>
  def a_method
    binding
  end
end
binding # => #<Binding:0x252b0>
C.new.a_method # => #<Binding:0x25238>
```

Generator rusztowania Rails zapewnia dobry przykład zastosowania dołączeń:

```
class ScaffoldingSandbox
  include ActionView::Helpers::ActiveRecordHelper
  attr_accessor :form_action, :singular_name, :suffix, :model_instance

  def sandbox_binding
    binding
  end

  # ...
end
```

`ScaffoldingSandbox` to klasa zapewniająca czyste środowisko, na podstawie którego generujemy szablon. ERb może generować szablon na podstawie kontekstu dołączeń, więc API jest dostępne z poziomu szablonów ERb.

```
part_binding = template_options[:sandbox].call.sandbox_binding
# ...
ERB.new(File.readlines(part_path).join,nil,'-').result(part_binding)
```

Wcześniej wspomniałem, że bloki są zamknięciami. Dołączenie zamknięcia reprezentuje jego stan — zbiór zmiennych i metod, do których posiada dostęp. Dołączenie zamknięcia można uzyskać za pomocą metody `Proc#binding`:

```
def var_from_binding(&b)
  eval("var", b.binding)
end

var = 123
var_from_binding {} # => 123
var = 456
var_from_binding {} # => 456
```

Użyliśmy tutaj tylko obiektu `Proc` jako obiektu, dla którego pobieraliśmy dołączenie. Poprzez dostęp do dołączeń (kontekstu) tych bloków można odwołać się do zmiennej lokalnej `var` przy pomocy zwykłego `eval` na dołączeniu.

## Introspekcja i ObjectSpace — analiza danych i metod w czasie działania

Ruby udostępnia wiele metod pozwalających na zaglądanie do obiektów w czasie działania programu. Dostępne są metody dostępu do zmiennych instancyjnych, ale ponieważ łamią one zasadę hermetyzacji, należy ich używać z rozwagą.

```
class C
  def initialize
    @ivar = 1
  end
end

c = C.new
c.instance_variables              # => ["@ivar"]
c.instance_variable_get(:@ivar)   # => 1

c.instance_variable_set(:@ivar, 3) # => 3
c.instance_variable_get(:@ivar)    # => 3
```

Metoda `Object#methods` zwraca tablicę metod instancyjnych wraz z metodami typu singleton zdefiniowanymi w obiekcie odbiorcy. Jeżeli pierwszym parametrem `methods` jest `false`, zwracane są wyłącznie metody typu singleton.

```
class C
  def inst_method
  end

  def self.cls_method
  end
end

c = C.new

class << c
  def singleton_method
  end
end

c.methods - Object.methods # => ["inst_method", "singleton_method"]
c.methods(false) # => ["singleton_method"]
```

Z kolei metoda `Module#instance_methods` zwraca tablicę metod instancyjnych klasy lub modułu. Należy zwrócić uwagę, że `instance_methods` jest wywoływana w kontekście klasy, natomiast `methods` — w kontekście instancji. Przekazanie wartości `false` do `instance_methods` powoduje, że zostaną pominięte metody klas nadrzędnych:

```
C.instance_methods(false) # => ["inst_method"]
```

Do analizy metod klasy `C` możemy wykorzystać metodę `metaclass` z metaid:

```
C.metaclass.instance_methods(false) # => ["new", "allocate", "cls_method", "superclass"]
```

Z mojego doświadczenia wynika, że metody te są zwykle wykorzystywane do zaspokojenia ciekawości. Z wyjątkiem bardzo niewielu dobrze zdefiniowanych idiomów rzadko zdarza się, że w kodzie produkcyjnym wykorzystywana jest refleksja metod obiektu. Znacznie częściej techniki te są wykorzystywane we wierszu poleceń konsoli do wyszukiwania dostępnych metod obiektu — zwykle jest to szybsze od sięgnięcia do podręcznika.

```
Array.instance_methods.grep /sort/ # => ["sort!", "sort", "sort_by"]
```

### ObjectSpace

`ObjectSpace` to moduł wykorzystywany do interakcji z systemem obiektowym Ruby. Posiada on kilka przydatnych metod modułu, które ułatwiają operacje niskopoziomowe.

- Metody usuwania nieużytków: `define_finalizer` (konfiguruje metodę wywołania zwrotnego wywoływaną bezpośrednio przed zniszczeniem obiektu), `undefine_finalizer` (usuwa to wywołanie zwrotne) oraz `garbage_collect` (uruchamia usuwanie nieużytków).
- `_id2ref` konwertuje ID obiektu na referencję do tego obiektu Ruby.
- `each_object` pozwala na iterację po wszystkich obiektach (lub wszystkich obiektach określonej klasy) i przekazuje je do bloku.

Jak zawsze, duże możliwości wiążą się ze znaczną odpowiedzialnością. Choć metody te mogą być przydatne, mogą również być niebezpieczne. Należy korzystać z nich rozsądnie.

Przykład prawidłowego zastosowania `ObjectSpace` znajduje się w bibliotece `Test::Unit`. W kodzie tym metoda `ObjectSpace.each_object` jest wykorzystana do wyliczenia wszystkich istniejących klas, które dziedziczą po `Test::Unit::TestCase`:

```
test_classes = []
ObjectSpace.each_object(Class) {
  | klass |
  test_classes << klass if (Test::Unit::TestCase > klass)
}
```

Niestety `ObjectSpace` znacznie komplikuje niektóre z maszyn wirtualnych Ruby. W szczególności wydajność JRuby znacznie spada po aktywowaniu `ObjectSpace`, ponieważ interpreter Ruby nie może bezpośrednio przeglądać sterty JVM w poszukiwaniu obiektów. Z tego powodu JRuby musi śledzić obiekty własnymi mechanizmami, co powoduje powstanie znacznego narzutu czasowego. Ponieważ ten sam mechanizm można uzyskać przy wykorzystaniu metod `Module.extend` oraz `Class.inherit`, pozostaje niewiele przypadków, w których zastosowanie `ObjectSpace` jest niezbędne.

## Delegacja przy wykorzystaniu klas pośredniczących

**Delegacja** jest odmianą kompozycji. Jest podobna do dziedziczenia, ale pomiędzy komponowanymi obiektami pozostawiona jest pewna „przestrzeń" koncepcyjna. Delegacja pozwala na modelowanie relacji „zawiera", a nie „jest". Gdy obiekt deleguje operację do innego, nadal istnieją dwa obiekty, a nie powstaje jeden obiekt będący wynikiem zastosowania hierarchii dziedziczenia.

Delegacja jest wykorzystywana w asocjacjach `ActiveRecord`. Klasa `AssociationProxy` deleguje większość metod (w tym `class`) do obiektów docelowych. W ten sposób asocjacje mogą być ładowane z opóźnieniem (nie są ładowane do momentu odwołania do danych) przy użyciu całkowicie przezroczystego interfejsu.

### DelegateClass oraz Forwardable

Standardowa biblioteka Ruby posiada mechanizmy dedykowane dla delegacji. Najprostszym jest `DelegateClass`. Przez dziedziczenie po `DelegateClass(klass)` oraz wywołanie w konstruktorze `super(instance)` klasa deleguje wszystkie wywołania nieznanych metod do przekazanego obiektu `klass`. Jako przykład weźmy klasę `Settings`, która deleguje wywołania do `Hash`:

```
require 'delegate'
class Settings < DelegateClass(Hash)
  def initialize(options = {})
    super({:initialized_at => Time.now - 5}.merge(options))
  end

  def age
    Time.now - self[:initialized_at]
  end
end

settings = Settings.new :use_foo_bar => true

# Wywołania metod są delegowane do obiektu
settings[:use_foo_bar] # => true
settings.age # => 5.000301
```

Konstruktor `Settings` wywołuje `super` w celu ustawienia delegowanego obiektu na nowy obiekt `Hash`. Należy zwrócić uwagę na różnicę pomiędzy kompozycją a dziedziczeniem — jeżeli zastosowalibyśmy dziedziczenie po `Hash`, `Settings` *byłby* obiektem `Hash`, natomiast w tym przypadku `Settings` *zawiera* obiekt `Hash` i deleguje do niego wywołania. Taka relacja kompozycji zapewnia większą elastyczność, szczególnie gdy obiekt do wydelegowania może zmieniać się (funkcja zapewniana przez `SimpleDelegator`).

Biblioteka standardowa Ruby zawiera również interfejs `Forwardable`, za pomocą którego poszczególne metody, a nie wszystkie niezdefiniowane metody, mogą być delegowane do innych obiektów. `ActiveSupport` w Rails zapewnia te same funkcje poprzez `Module#delegate`, a dodatkowo jej API jest znacznie jaśniejsze:

```
class User < ActiveRecord::Base
  belongs_to :person

  delegate :first_name, :last_name, :phone, :to => :person
end
```

# Monkeypatching

W Ruby wszystkie klasy są otwarte. Każda klasa i obiekt mogą być modyfikowane w dowolnym momencie. Daje to możliwość rozszerzania lub zmieniania istniejących funkcji. Takie rozszerzanie może być zrealizowane w bardzo elegancki sposób, bez modyfikowania oryginalnych definicji.

Rails w znacznym stopniu korzysta z otwartości systemu klas Ruby. Otwieranie klas i dodawanie kodu jest nazywane **monkeypatching** (termin zapożyczony ze społeczności języka Python). Choć brzmi to odpychająco, termin ten został użyty w zdecydowanie pozytywnym świetle; technika ta jest uważana za niezwykle przydatną. Niemal wszystkie wtyczki do Rails wykonują w pewien sposób monkeypatching rdzenia Rails.

## Wady techniki monkeypatching

Technika monkeypatching ma dwie główne wady. Przede wszystkim kod jednej metody może być rozsiany po kilku plikach. Najważniejszym tego przykładem jest metoda `process` z `ActionController`. Metoda ta w czasie działania jest przechwytywana przez metody z pięciu różnych plików. Każda z tych metod dodaje kolejną funkcję: filtry, obsługę wyjątków, komponenty i zarządzanie sesją. Zysk z rozdzielenia komponentów funkcjonalnych na osobne pliki przeważa rozdmuchanie stosu wywołań.

Inną konsekwencją rozsiania funkcji jest problem z prawidłowym dokumentowaniem metod. Ponieważ działanie metody `process` może zmieniać się w zależności od załadowanego kodu, nie istnieje dobre miejsce do umieszczenia dokumentowania operacji dodawanych przez każdą z metod. Problem ten wynika z powodu zmiany identyfikacji metody `process` wraz z łączeniem ze sobą metod.

## Dodawanie funkcji do istniejących metod

Ponieważ Rails wykorzystuje filozofię rozdzielania problemów, często pojawia się potrzeba rozszerzania funkcji istniejącego kodu. W wielu przypadkach chcemy „dokleić" fragment do istniejącej funkcji bez wpływania na kod tej funkcji. Ten dodatek nie musi być bezpośrednio związany z oryginalnym przeznaczeniem funkcji — może zapewniać uwierzytelnianie, rejestrowanie lub inne zagadnienia przekrojowe.

Przedstawimy teraz kilka zagadnień związanych z problemami przekrojowymi i dokładnie wyjaśnimy jeden (łączenie metod), który zdobył największe zainteresowanie w społeczności Ruby.

### Podklasy

W tradycyjnym programowaniu obiektowym klasa może być rozszerzana przez dziedziczenie po niej i zmianę danych lub działania. Paradygmat ten działa dla większości przypadków, ale ma kilka wad:

- Zmiany, jakie chcemy wprowadzić, mogą być niewielkie, co powoduje, że tworzenie nowej klasy jest zbyt skomplikowane. Każda nowa klasa w hierarchii dziedziczenia powoduje, że zrozumienie kodu jest trudniejsze.
- Może być konieczne wprowadzenie serii powiązanych ze sobą zmian do klas, które nie są ze sobą w inny sposób związane. Tworzenie kolejnych podklas może być przesadą, a dodatkowo spowoduje rozdzielenie funkcji, które powinny być przechowywane razem.
- Klasa może być już używana w aplikacji, a my chcemy globalnie zmienić jej działanie.
- Można chcieć dodawać lub usuwać operacje w czasie działania programu, co powinno dawać globalny efekt. (Technika ta zostanie przedstawiona w pełnym przykładzie w dalszej części rozdziału).

W tradycyjnym programowaniu obiektowym funkcje te mogą wymagać zastosowania skomplikowanego kodu. Kod nie tylko będzie skomplikowany, ale również znacznie ściślej związany z istniejącym kodem lub kodem wywołującym go.

### Programowanie aspektowe

**Programowanie aspektowe** (AOP — ang. *Aspect-oriented Programming*) jest jedną z technik, które mają za zadanie rozwiązać problemy z separacją zadań. Prowadzonych jest dużo dyskusji na temat stosowania AOP w Ruby, ponieważ wiele z zalet AOP można osiągnąć przez użycie metaprogramowania. Istnieje propozycja implementacji AOP bazującej na przecięciach w Ruby[12], ale zanim zostanie ona dołączona do oficjalnej wersji, mogą minąć miesiące lub nawet lata.

[12] *http://wiki.rubygarden.org/Ruby/page/show/AspectOrientedRuby*.

W AOP bazującym na przecięciach, przecięcia te są czasami nazywane „przezroczystymi podklasami", ponieważ w modularny sposób rozszerzają funkcje klas. Przecięcia działają tak jak podklasy, ale nie ma potrzeby tworzenia instancji tych podklas, wystarczy utworzyć instancje klas bazowych.

Biblioteka Ruby Facets (*facets.rubyforge.org*) zawiera bibliotekę AOP bazującą na przecięciach, zrealizowaną wyłącznie w Ruby. Posiada ona pewne ograniczenia spowodowane tym, że jest napisana wyłącznie w Ruby, ale jej użycie jest dosyć jasne:

```
class Person
  def say_hi
    puts "Cześć!"
  end
end

cut :Tracer < Person do
  def say_hi
    puts "Przed metodą"
    super
    puts "Po metodzie"
  end
end

Person.new.say_hi
# >> Przed metodą
# >> Cześć!
# >> Po metodzie
```

Jak widać, przecięcie `Tracer` jest przezroczystą podklasą — gdy tworzymy instancję `Person`, jest ona modyfikowana przez przecięcie `Tracer` i „nie wie" ona nic o jego istnieniu. Możemy również zmienić metodę `Person#say_hi` bez konieczności modyfikacji naszego przecięcia.

Z różnych powodów techniki AOP w Ruby nie przyjęły się. Przedstawimy teraz standardowe metody radzenia sobie z problemami separacji w Ruby.

## Łączenie metod

Standardowym rozwiązaniem tego problemu w Ruby jest **łączenie metod** — nadawanie istniejącej metodzie synonimu i nadpisywanie starej definicji nową treścią. W nowej treści zwykle wywołuje się starą definicję metody przez odwołanie się do synonimu (odpowiednik wywołania `super` w dziedziczonej, nadpisywanej metodzie). W efekcie tego można modyfikować działanie istniejących metod. Dzięki otwartej naturze klas Ruby można dodawać funkcje do niemal każdego fragmentu kodu. Oczywiście trzeba pamiętać, że musi być to wykonywane rozważnie, aby zachować przejrzystość kodu.

Łączenie metod w Ruby jest zwykle realizowane za pomocą standardowego idiomu. Załóżmy, że mamy pewną bibliotekę kodu, która pobiera obiekt `Person` poprzez sieć:

```
class Person
  def refresh
    # (pobranie danych z serwera)
  end
end
```

Operacja trwa przez pewien czas, więc chcemy go zmierzyć i zapisać wyniki. Wykorzystując otwarte klasy Ruby, możemy po prostu otworzyć klasę `Person` i dodać kod rejestrujący do metody `refresh`:

```
class Person
  def refresh_with_timing
    start_time = Time.now.to_f
    retval = refresh_without_timing
    end_time = Time.now.to_f
    logger.info "Refresh: #{"%.3f" % (end_time-start_time)} s."
    retval
end

  alias_method :refresh_without_timing, :refresh
  alias_method :refresh, :refresh_with_timing
end
```

Możemy umieścić ten kod w osobnym pliku (być może razem z pozostałym kodem pomiarowym) i jeżeli dołączymy ten plik za pomocą `require` po oryginalnej definicji `refresh`, kod pomiarowy będzie w odpowiedni sposób dodany przed wywołaniem i po wywołaniu oryginalnej metody. Pomaga to w zachowaniu separacji, ponieważ możemy podzielić kod na kilka plików, w zależności od realizowanych zadań, a nie na podstawie obszaru, jaki jest modyfikowany.

Dwa wywołania `alias_method` wokół oryginalnego wywołania `refresh` pozwalają na dodanie kodu pomiarowego. W pierwszym wywołaniu nadajemy oryginalnej metodzie synonim `refresh_without_timing` (dzięki czemu otrzymujemy nazwę, poprzez którą będziemy się odwoływać do oryginalnej metody z wnętrza `refresh_with_timing`), natomiast w drugim nadajemy naszej nowej metodzie nazwę `refresh`.

Ten paradygmat użycia dwóch wywołań `alias_method` w celu dodania funkcji jest na tyle powszechny, że ma w Ruby swoją nazwę — `alias_method_chain`. Wykorzystywane są tu dwa argumenty: nazwa oryginalnej metody oraz nazwa funkcji.

Przy użyciu `alias_method_chain` możemy połączyć dwa wywołania `alias_method` w jedno:

```
alias_method_chain :refresh, :timing
```

## Modularyzacja

Technika monkeypatching daje nam ogromne możliwości, ale zaśmieca przestrzeń nazw modyfikowanej klasy. Często można osiągnąć te same efekty w bardziej elegancki sposób, przez modularyzację dodatku i wstawienie modułu w łańcuch wyszukiwania klasy. Wtyczka Active Merchant, której autorem jest Tobias Lütke, wykorzystuje to podejście w pomocy do widoków. Najpierw tworzony jest moduł z metodami pomocniczymi:

```
module ActiveMerchant
  module Billing
    module Integrations
      module ActionViewHelper
        def payment_service_for(order, account, options = {}, &proc)
          ...
        end
      end
    end
  end
end
```

Następnie, w skrypcie wtyczki *init.rb*, moduł jest dołączany do `ActionView::Base`:

```
require 'active_merchant/billing/integrations/action_view_helper'
ActionView::Base.send(:include,
  ActiveMerchant::Billing::Integrations::ActionViewHelper)
```

Oczywiście, znacznie prościej byłoby bezpośrednio otworzyć `ActionView::Base` i dodać metodę, ale ta metoda pozwala wykorzystać zaletę modularności. Cały kod Active Merchant znajduje się w module `ActiveMerchant`.

Metoda ta ma jedną wadę. Ponieważ w dołączonym module metody są wyszukiwane według własnych metod klasy, nie można bezpośrednio nadpisywać metod klasy przez dołączenie modułu:

```
module M
  def test_method
    "Test z M"
  end
end

class C
  def test_method
    "Test z C"
  end
end

C.send(:include, M)
C.new.test_method # => "Test z C"
```

Zamiast tego powinniśmy utworzyć nową nazwę w module i skorzystać z `alias_method_chain`:

```
module M
  def test_method_with_module
    "Test z M"
  end
end

class C
  def test_method
    "Test z C"
  end
end

# W przypadku wtyczki te dwa wiersze znajdą się w init.rb.
C.send(:include, M)
C.class_eval { alias_method_chain :test_method, :module }

C.new.test_method # => "Test z M"
```

# Programowanie funkcyjne

Paradygmat **programowania funkcyjnego** skupia się na wartościach, a nie efektach ubocznych wartościowania. W odróżnieniu od programowania imperatywnego styl funkcjonalny operuje na wartościach wyrażeń w sensie matematycznym. Aplikacje oraz kompozycje funkcyjne korzystają z koncepcji pierwszej klasy, a zmieniający się stan (który oczywiście istnieje na niskim poziomie) jest niewidoczny dla programisty.

Jest to dosyć zdradliwa koncepcja i jest ona często nieznana nawet doświadczonym programistom. Najlepsze porównania są zaczerpnięte z matematyki, z której korzysta programowanie funkcyjne.

Rozważmy równanie matematyczne $x = 3$. Znak równości w tym wyrażeniu wskazuje na równoważność: „$x$ jest równe 3”. Dla porównania, wyrażenie `x = 3` w Ruby ma zupełnie inną naturę. Znak równości oznacza przypisanie: „przypisz 3 do x”. Najważniejsza różnica polega na

tym, że języki programowania funkcyjnego określają, *co* należy policzyć, natomiast języki programowania imperatywnego zwykle definiują, *jak* to policzyć.

## Funkcje wysokiego poziomu

Kamieniami węgielnymi programowania funkcyjnego są oczywiście funkcje. Głównym sposobem wpływu paradygmatu programowania funkcyjnego na główny kierunek rozwoju programowania w Ruby jest użycie **funkcji wysokiego poziomu** (nazywanych również **funkcjami pierwszej kategorii**, choć te dwa terminy nie są dokładnie tym samym). Funkcje wysokiego poziomu są funkcjami działającymi na innych funkcjach. Zwykle wymagają jednej lub więcej funkcji jako argumentów lub zwracają funkcję.

W Ruby funkcje są obsługiwane zwykle jako obiekty wysokiego poziomu; mogą być one tworzone, zmieniane, przekazywane, zwracane i wywoływane. Funkcje anonimowe są reprezentowane jako obiekty `Proc` tworzone za pomocą `Proc.new` lub `Kernel#lambda`:

```
add = lambda{|a,b| a + b}
add.class # => Proc
add.arity # => 2

# Wywołanie Proc za pomocą Proc#call.
add.call(1,2) # => 3

# Składnia alternatywna.
add[1,2] # => 3
```

Najczęstszym zastosowaniem bloków w Ruby jest użycie ich razem z iteratorami. Wielu programistów, którzy przeszli na Ruby z bardziej imperatywnych języków, zaczyna pisać kod w następujący sposób:

```
collection = (1..10).to_a
for x in collection
  puts x
end
```

Ten sam fragment kodu napisany zgodnie z duchem Ruby korzysta z iteratora, `Array#each` i przekazania wartości do bloku. Jest to druga natura doświadczonych programistów Ruby:

```
collection.each {|x| puts x}
```

Ta metoda jest odpowiednikiem utworzenia obiektu `Proc` i przekazania go do `each`:

```
print_me = lambda{|x| puts x}
collection.each(&print_me)
```

Przykłady te mają na celu pokazanie, że funkcje są obiektami pierwszej kategorii i mogą być traktowane tak jak inne obiekty.

## Moduł Enumerable

Moduł `Enumerable` w Ruby udostępnia kilka przydatnych metod, które mogą być dołączane do klas, które są „wyliczalne”, czyli można na nich wykonać iterację. Metody te korzystają z metody instancyjnej `each` i opcjonalnie metody <=> (porównanie lub „statek kosmiczny”). Metody modułu `Enumerable` można podzielić na kilka kategorii.

### Predykaty

Reprezentują one właściwości kolekcji, które mogą przyjmować wartości `true` lub `false`.

`all?`
: Zwraca `true`, jeżeli dany blok zwraca wartość `true` dla wszystkich elementów w kolekcji.

`any?`
: Zwraca `true`, jeżeli dany blok zwraca wartość `true` dla dowolnego elementu w kolekcji.

`include?(x), member?(x)`
: Zwraca `true`, jeżeli `x` jest członkiem kolekcji.

### Filtry

Metody te zwracają podzbiór elementów kolekcji.

`detect`, `find`
: Zwraca pierwszy element z kolekcji, dla którego blok zwraca wartość `true` lub `nil`, jeżeli nie zostanie znaleziony taki element.

`select`, `find_all`
: Zwraca tablicę elementów z kolekcji, dla których blok zwraca wartość `true`.

`reject`
: Zwraca tablicę elementów z kolekcji, dla których blok zwraca wartość `false`.

`grep(x)`
: Zwraca tablicę elementów z kolekcji, dla których `x=== item` ma wartość `true`. Jest to odpowiednik `select{|item| x === item}`.

### Transformatory

Metody te przekształcają kolekcję na inną, zgodnie z jedną z kilku zasad.

`map`, `collect`
: Zwraca tablicę składającą się z wyników danego bloku zastosowanego dla każdego z elementów kolekcji.

`partition`
: Odpowiednik `[select(&block), reject(&block)]`.

`sort`
: Zwraca nową tablicę elementów z kolekcji posortowanych przy użyciu bloku (traktowanego jako metoda <=>) lub własnej metody <=> elementu.

`sort_by`
: Podobnie jak `sort`, ale wartości, na podstawie których jest wykonywane sortowanie, są uzyskiwane z bloku. Ponieważ porównywanie tablic jest wykonywane w kolejności elementów, można sortować według wielu pól przy użyciu `person.sort_by{|p| [p.city, p.name]}`. Wewnętrznie metoda `sort_by` korzysta z transformacji Schwartza, więc jest znacznie bardziej efektywna niż `sort`, jeżeli wartościowanie bloku jest kosztowne.

`zip(*others)`
: Zwraca tablicę krotek zbudowanych z kolejnych elementów `self` i `others`:

```
puts [1,2,3].zip([4,5,6],[7,8,9]).inspect
# >> [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Gdy wszystkie kolekcje są tej samej wielkości, `zip(*others)` jest odpowiednikiem `([self]+ ↪others).transpose`:

```
puts [[1,2,3],[4,5,6],[7,8,9]].transpose.inspect
# >> [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Gdy zostanie podany blok, jest on wykonywany raz dla każdego elementu tablicy wynikowej:

```
[1,2,3].zip([4,5,6],[7,8,9]) {|x| puts x.inspect}
# >> [1, 4, 7]
# >> [2, 5, 8]
# >> [3, 6, 9]
```

### Agregatory

Metody te pozwalają na agregowanie lub podsumowanie danych.

`inject(initial)`
: Składa operacje z kolekcji. Na początku inicjowany jest akumulator (pierwsza wartość jest dostarczana przez `initial`) oraz pierwszy obiekt bloku. Zwrócona wartość jest używana jako akumulator dla kolejnych iteracji. Suma kolekcji jest często definiowana w następujący sposób:

```
module Enumerable
  def sum
inject(0){|total, x| total + x}
  end
end
```

Jeżeli nie zostanie podana wartość początkowa, pierwsza iteracja pobiera pierwsze dwa elementy.

`max`
: Zwraca maksymalną wartość z kolekcji przy użyciu tych samych procedur co w przypadku metody `sort`.

`min`
: Podobnie jak `max`, ale zwraca wartość minimalną w kolekcji.

### Pozostałe

`each_with_index`
: Podobnie jak `each`, ale korzysta z indeksu zaczynającego się od `0`.

`entries`, `to_a`
: Umieszcza kolejne elementy w tablicy, a następnie zwraca tablicę.

Metody modułu `Enumerable` są bardzo przydatne i zwykle można znaleźć metodę odpowiednią do tego, co potrzebujemy. Jeżeli jednak nie znajdziemy odpowiedniej, warto odwiedzić witrynę Ruby Facets (*http://facets.rubyforge.org*).

## Enumerator

Ruby zawiera również mało znany moduł biblioteki standardowej, `Enumerator`. (Ponieważ jest to biblioteka standardowa, a nie podstawy języka, konieczne jest użycie frazy `require "enumerator"`).

Moduł `Enumerable` zawiera wiele enumeratorów, które mogą być użyte dla dowolnego obiektu wyliczalnego, ale posiadają one jedno ograniczenie — wszystkie te iteratory bazują na metodach instancyjnych. Jeżeli będziemy chcieli skorzystać z innego iteratora niż `each`, jako podstawy dla `map`, `inject` lub innych funkcji z `Enumerable`, można skorzystać z modułu `Enumerator` jako łącznika.

Metoda `Enumerator.new` posiada sygnaturę `Enumerator.new(obj, method, *args)`, gdzie *obj* jest obiektem do enumeracji, *method* jest bazowym iteratorem, a *args* to dowolne argumenty przekazywane do iteratora. Można na przykład napisać funkcję `map_with_index` (odmiana `map`, która przekazuje obiekt z indeksem do bloku):

```
require "enumerator"
module Enumerable
  def map_with_index &b
    enum_for(:each_with_index).map(&b)
  end
end

puts ("a".."f").map_with_index{|letter, i| [letter, i]}.inspect
# >> [["a", 0], ["b", 1], ["c", 2], ["d", 3], ["e", 4], ["f", 5]]
```

Metoda `enum_for` zwraca obiekt `Enumerator`, którego każda z metod działa podobnie do `each_with_index` z oryginalnego obiektu. Ten obiekt `Enumerator` został wcześniej rozszerzony o metody instancyjne z `Enumerable`, więc możemy po prostu wywołać na nim `map`, przekazując odpowiedni blok.

`Enumerator` dodaje również do `Enumerable` kilka przydatnych metod. Metoda `Enumerable#each_slice(n)` iteruje po fragmentach tablicy, po `n` jednocześnie:

```
(1..10).each_slice(3){|slice| puts slice.inspect}
# >> [1, 2, 3]
# >> [4, 5, 6]
# >> [7, 8, 9]
# >> [10]
```

Podobnie `Enumerable#each_cons(n)` porusza „oknem przesuwnym" o rozmiarze `n` po kolekcji, o jeden element na raz:

```
(1..10).each_cons(3){|slice| puts slice.inspect}
# >> [1, 2, 3]
# >> [2, 3, 4]
# >> [3, 4, 5]
# >> [4, 5, 6]
# >> [5, 6, 7]
# >> [6, 7, 8]
# >> [7, 8, 9]
# >> [8, 9, 10]
```

Enumeracje zostały usprawnione w Ruby 1.9. Moduł `Enumerator` stał się częścią podstawowego języka. Dodatkowo iteratory zwracają automatycznie obiekt `Enumerator`, jeżeli nie zostanie do nich przekazany blok. W Ruby 1.8 do mapowania wartości tablicy mieszającej wykorzystywany był następujący kod:

```
hash.values.map{|value| ... }
```

Na podstawie tablicy mieszającej tworzona była tablica wartości, a następnie mapowanie było realizowane na tej tablicy. Aby pominąć krok pośredni, można użyć obiektu `Enumerator`:

```
hash.enum_for(:each_value).map{|value| ... }
```

W ten sposób mamy obiekt `Enumerator`, którego każda z metod działa identycznie jak metoda `each_value` z klasy `hash`. Jest to zalecane w przeciwieństwie do tworzenia potencjalnie dużych tablic, które są za chwilę zwalniane. W Ruby 1.9 jest to domyślne działanie, o ile nie zostanie przekazany blok. Znacznie to upraszcza nasz kod:

```
hash.each_value.map{|value| ... }
```

# Przykłady

## Zmiany funkcji w czasie działania

Przykład ten łączy kilka technik przedstawionych w tym rozdziale. Wracamy do przykładu `Person`, w którym chcemy zmierzyć czas działania kilku kosztownych metod:

```
class Person
  def refresh
    # ...
  end

  def dup
    # ...
  end
end
```

Nie chcemy pozostawiać całego kodu pomiarowego w środowisku produkcyjnym, ponieważ wprowadza on dodatkowy narzut czasowy. Jednak najlepiej pozostawić sobie możliwość włączenia tej opcji w czasie rozwiązywania problemów. Napiszemy więc kod, który pozwoli dodawać i usuwać funkcje (w tym przypadku kod pomiarowy) w czasie pracy programu bez modyfikowania kodu źródłowego.

Najpierw napiszemy metody otaczające nasze kosztowne metody poleceniami pomiarowymi. Jak zwykle, wykorzystamy metodę monkeypatching do dołączenia metod pomiarowych z innego pliku do `Person`, co pozwala oddzielić kod pomiarowy od funkcji logiki modelu[13]:

```
class Person
  TIMED_METHODS = [:refresh, :dup]
  TIMED_METHODS.each do |method|
    # Konfiguracja synonimu _without_timing dla oryginalnej metody.
    alias_method :"#{method}_without_timing", method

    # Konfiguracja metody _with_timing method, która otacza kod poddawany pomiarowi.
    define_method :"#{method}_with_timing" do
      start_time = Time.now.to_f
      returning(self.send(:"#{method}_without_timing")) do
        end_time = Time.now.to_f

        puts "#{method}: #{"%.3f" % (end_time-start_time)} s."
      end
    end
  end
end
```

[13] W tym przykładowym kodzie wykorzystana została interpolacja zmiennych w literałach symboli. Ponieważ symbol jest definiowany z wykorzystaniem ciągu w cudzysłowach, interpolacja zmiennych jest tak samo dozwolona jak w innych zastosowaniach ciągu w cudzysłowach — symbol `:"sym#{2+2}"` jest tym samym co `:sym4`.

Aby włączać lub wyłączać śledzenie, dodajemy do `Person` metody singleton:

```
class << Person
  def start_trace
    TIMED_METHODS.each do |method|
      alias_method method, :"#{method}_with_timing"
    end
  end

  def end_trace
    TIMED_METHODS.each do |method|
      alias_method method, :"#{method}_without_timing"
    end
  end
end
```

Aby włączyć śledzenie, umieszczamy każde wywołanie metody w wywołaniu metody pomiarowej. Aby je wyłączyć, po prostu wskazujemy metodę z powrotem na oryginalną metodę (która jest dostępna tylko przez jej synonim `_without_timing`).

Aby skorzystać z tych dodatków, po prostu wywołujemy metodę `Person.trace`:

```
p = Person.new
p.refresh # => (...)

Person.start_trace
p.refresh # => (...)
# -> refresh: 0.500 s.

Person.end_trace
p.refresh # => (...)
```

Gdy mamy teraz możliwość dodawania i usuwania kodu pomiarowego w czasie pracy, możemy udostępnić ten mechanizm w aplikacji; możemy udostępnić administratorowi lub programiście interfejs do śledzenia wszystkich lub wybranych funkcji bez konieczności ponownego uruchomienia aplikacji. Podejście takie ma kilka zalet w stosunku do dodawania kodu rejestrującego dla każdej funkcji osobno:

- Oryginalny kod jest niezmieniony, może on być modyfikowany lub ulepszany bez wpływania na kod śledzący.
- Po wyłączeniu śledzenia kod działa dokładnie tak samo jak wcześniej, ponieważ kod śledzący jest niewidoczny w śladzie stosu. Nie ma narzutu wydajnościowego po wyłączeniu śledzenia.

Istnieje jednak kilka wad kodu, który sam się modyfikuje:

- Śledzenie jest dostępne tylko na poziomie funkcji. Bardziej szczegółowe śledzenie wymaga zmiany lub łatania oryginalnego kodu. W kodzie Rails jest to rozwiązywane przez korzystanie z małych metod o samoopisujących się nazwach.
- Po włączeniu śledzenia zapis stosu staje się bardziej skomplikowany. Przy włączonym śledzeniu zapis stosu dla metody `Person#refresh` zawiera dodatkowy poziom — `#refresh_with_timing`, a następnie `#refresh_without_timing` (oryginalna metoda).
- Podejście takie może zawieść przy użyciu więcej niż jednego serwera aplikacji, ponieważ synonimy funkcji są tworzone w pamięci. Zmiany nie są propagowane pomiędzy serwerami i zostaną wycofane, gdy proces serwera zostanie ponownie uruchomiony. Jednak może to być niewielki problem; zwykle nie profilujemy całego ruchu w obciążonym środowisku produkcyjnym, a jedynie jego fragment.

## Kod sterujący Rails

Kod sterujący jest prawdopodobnie najbardziej skomplikowanym koncepcyjnie kodem w Rails. Kod ten podlega kilku ograniczeniom:

- Segmenty ścieżek mogą przechwytywać wiele części adresu URL:
  - Kontrolery mogą być dzielone za pomocą przestrzeni nazw, więc ścieżka `":controller/:action/:id"` może odpowiadać adresowi URL `"/store/product/edit/15"` kontrolera `"store/product"`.
  - Ścieżki mogą zawierać segmenty `path_info`, które pozwalają na podział wielu segmentów URL: ścieżka `"page/*path_info"` może odpowiadać adresowi URL `"/page/products/top_products/15"` z segmentem `path_info` przechwytującym pozostałą część URL.
- Ścieżki mogą być ograniczane przez warunki, które muszą być spełnione, aby dopasować ścieżkę.
- System ścieżek musi być dwukierunkowy; działa on w przód w celu rozpoznawania ścieżek i w tył do ich generowania.
- Rozpoznawanie ścieżek musi być szybkie, ponieważ jest wykonywane dla każdego żądania HTTP. Generowanie ścieżek musi być niezwykle szybkie, ponieważ może być wykonywane dziesiątki razy na żądanie HTTP (po jednym na łącze wychodzące) podczas tworzenia strony.

Nowy kod `routing_optimisation` w Rails 2.0 (*actionpack/lib/action_controller/routing_optimisation.rb*), którego autorem jest Michael Koziarski, rozwiązuje problem złożoności sterowania w Rails. W nowym kodzie zoptymalizowany został prosty przypadek generowania nazwanych ścieżek bez dodatkowych `:requirements`.

Ponieważ szybkość jest wymagana zarówno przy generowaniu, jak i rozpoznawaniu, kod sterujący modyfikuje się w czasie działania. Klasa `ActionController::Routing::Route` reprezentuje pojedynczą ścieżkę (jeden wpis w *config/routes.rb*). Metoda `Route#recognize` sama się modyfikuje:

```
class Route
  def recognize(path, environment={})
    write_recognition
    recognize path, environment
  end
end
```

Metoda `recognize` wywołuje `write_recognition`, która przetwarza ścieżkę i tworzy jej skompilowaną wersję. Metoda `write_recognition` nadpisuje definicję `recognize` za pomocą tej definicji. W ostatnim wierszu oryginalnej metody `recognize` wywoływana jest metoda `recognize` (która została zastąpiona przez jej skompilowaną wersję) z oryginalnymi argumentami. W ten sposób ścieżka jest skompilowana w pierwszym wywołaniu `recognize`. Wszystkie kolejne wywołania korzystają ze skompilowanej wersji i nie wykonują ponownie parsowania i wykonywania kodu kierującego.

Poniżej przedstawiona jest metoda `write_recognition`:

```
def write_recognition
  # Tworzenie struktury if do pobrania parametrów, o ile występują.
```

```
    body = "params = parameter_shell.dup\n#{recognition_extraction * "\n"}\nparams"
    body = "if #{recognition_conditions.join(" && ")}\n#{body}\nend"

    # Budowanie deklaracji metody i jej kompilacja.
    method_decl = "def recognize(path, env={})\n#{body}\nend"
    instance_eval method_decl, "generated code (#{__FILE__}:#{__LINE__})"
    method_decl
  end
```

Zmienna lokalna `body` jest wypełniana skompilowanym kodem ścieżki. Jest ona umieszczona w deklaracji metody nadpisującej `recognize`. Dla domyślnej ścieżki:

```
map.connect ':controller/:action/:id'
```

metoda `write_recognition` generuje następujący kod:

```
def recognize(path, env={})
  if (match = /(long regex)/.match(path))
    params = parameter_shell.dup
    params[:controller] = match[1].downcase if match[1]
    params[:action] = match[2] || "index"
    params[:id] = match[3] if match[3]
    params
  end
end
```

Metoda `parameter_shell` zwraca domyślny zbiór parametrów związanych ze ścieżką. W treści tej metody wykonywane są testy przy użyciu wyrażenia regularnego, wypełniana i zwracana jest tablica `params`, o ile parametry zostaną wykryte przez wyrażenie regularne. Jeżeli nie zostaną wykryte, metoda zwraca `nil`.

Po utworzeniu treści metody jest ona wartościowana w kontekście ścieżki, przy użyciu `instance_eval`. Powoduje to nadpisanie metody `recognize` określonej ścieżki.

# Propozycje dalszych lektur

Świetnym wprowadzeniem do wewnętrznych mechanizmów Ruby jest *Ruby Hacking Guide*, którego autorem jest Minero AOKI. Pozycja ta została przetłumaczona na angielski i jest dostępna pod adresem *http://rhg.rubyforge.org*.

Kilka kolejnych artykułów technicznych na temat Ruby można znaleźć na witrynie Eigenclass (*http://eigenclass.org*).

*Evil.rb* jest biblioteką pozwalającą na dostęp do wnętrza obiektów Ruby. Pozwala ona na zmianę wewnętrznego stanu obiektów, śledzenie i przeglądanie wskaźników `klass` oraz `super`, zmianę klasy obiektu i powodowanie niezłego zamętu. Należy korzystać z niej rozważnie. Biblioteka jest dostępna pod adresem *http://rubyforge.org/projects/evil/*. Mauricio Fernández przedstawia możliwości biblioteki Evil w artykule dostępnym pod adresem *http://eigenclass.org/*
*↪hiki.rb?evil.rb+dl+and+unfreeze*.

Jamis Buck w dokładny sposób przedstawia kod sterujący Rails, jak również kilka innych złożonych elementów Rails — pod adresem *http://weblog.jamisbuck.org/under-the-hood*.

Jednym z najłatwiejszych do zrozumienia i najlepiej zaprojektowanych fragmentów Ruby, z jakim miałem do czynienia, jest Capistrano 2, którego autorem jest również Jamis Buck.

Capistrano ma nie tylko jasne API, ale jest również niezwykle dobrze napisany. Zagłębienie się w szczegółach Capistrano jest warte poświęconego czasu. Kod źródłowy jest dostępny za pośrednictwem Subversion pod adresem *http://svn.rubyonrails.org/rails/tools/capistrano*.

Książka *High-Order Perl* (Morgan Kaufman Publishers), której autorem jest Mark Jason Dominus, była rewolucją przy wprowadzaniu koncepcji programowania funkcyjnego w języku Perl. Gdy książka ta została wydana, w roku 2005, język ten nie był znany ze wsparcia programowania funkcyjnego. Większość przykładów z tej książki została przetłumaczona dosyć wiernie na Ruby; jest to dobre ćwiczenie, jeżeli Czytelnik zna Perl. James Edvard Gray II napisał swoją wersję *High-Order Ruby*, dostępną pod adresem *http://blog.grayproductions.net/categories/highe*
*↪rorder_ruby*.

Książka *Ruby Programming Language*, autorstwa Davida Flanagana i Yukihiro Matsumoto (O'Reilly), obejmuje zarówno Ruby 1.8, jak i 1.9. Została ona wydana w styczniu 2008 roku. Jej część poświęcono technikom programowania funkcyjnego w Ruby.