

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Ruby. Wzorce projektowe

Autor: Russ Olsen

Tłumaczenie: Mikołaj Szczepaniak

ISBN: 978-83-246-1688-6

Tytuł oryginału: [Design Patterns in Ruby](#)
(Addison-Wesley Professional Ruby Series)

Format: 172x245, stron: 370



Zwiększ elastyczność swojego kodu dzięki wzorcom projektowym!

- Jak rozpocząć przygodę z językiem Ruby?
- Jak wykorzystać drzemiące w nim możliwości?
- Jak zwiększyć elastyczność tworzonego kodu za pomocą wzorców projektowych?

Stworzony w 1995 roku przez Yukihiro Matsumoto język Ruby dzięki swym unikalnym możliwościom zdobywa serca programistów na całym świecie. Cechy, które podbijają to nieufne środowisko, to między innymi prosta składnia z wbudowanymi w nią wyrażeniami regularnymi, automatyczne oczyszczanie pamięci i wiele, wiele innych. Ogromna i chętna do pomocy społeczność czyni to rozwiązanie jeszcze bardziej atrakcyjnym. Ruby pozwala na korzystanie ze wzorców projektowych – zbioru zasad i reguł prowadzących do celu w najlepszy, najszybszy i najbardziej elastyczny sposób.

Wzorce projektowe kojarzą się głównie z językami Java oraz C i C++. Książka „Ruby. Wzorce projektowe” pokazuje, że można ich z powodzeniem używać również w języku Ruby. Dowiesz się z niej, w jaki sposób wykorzystać znane wzorce, takie jak Observer, Singleton czy też Proxy. Autor przedstawi Ci również nowe wzorce, które ze względu na cechy języka Ruby mogą zostać w nim zastosowane. Jednak zanim przejdziesz do ich omawiania, Russ poprowadzi Cię przez podstawy programowania w tym języku. Nauczysz się używać między innymi pętli, instrukcji warunkowych, wyrażeń regularnych. Niewątpliwie Twoją ciekawość wzbudzi tak zwany „duck typing”, który oczywiście także został dokładnie tu omówiony. Russ Olsen dzięki swojemu wieloletniemu doświadczeniu każdy wzorec ilustruje przykładem z życia wziętym. Ułatwi Ci to przyswojenie i zastosowanie we własnych projektach przedstawionych tu wzorców.

- Podstawy programowania w języku Ruby
- Zastosowanie wzorców – takich jak Observer, Composite, Iterator, Command i wiele innych
- Wykorzystanie technik metaprogramowania do tworzenia obiektów niestandardowych
- Wykorzystanie wyrażeń regularnych
- Użycie języków dziedzicznych
- Sposób instalacji języka Ruby

Korzystaj z doświadczenia najlepszych programistów – używaj wzorców projektowych w języku Ruby!



SPIS TREŚCI

	Słowo wstępne	15
	Przedmowa	19
	Podziękowania	25
	O autorze	27
CZĘŚĆ I	WZORCE PROJEKTOWE I RUBY	29
Rozdział 1.	Budowa lepszych programów z wykorzystaniem wzorców projektowych	31
	Banda Czworoga	32
	Wzorce dla wzorców	33
	Oddzielaj elementy zmienne od elementów stałych	33
	Programuj pod kątem interfejsu, nie implementacji	34
	Stosuj kompozycję zamiast dziedziczenia	36
	Deleguj, deleguj i jeszcze raz deleguj	40
	Czy dane rozwiązanie rzeczywiście jest niezbędne	41
	Czternaście z dwudziestu trzech	43
	Wzorce projektowe w języku Ruby?	45
Rozdział 2.	Pierwsze kroki w języku Ruby	47
	Interaktywny język Ruby	48
	Przywitaj się ze światem	48
	Zmienne	51
	Typy Fixnum i Bignum	52
	Liczby zmiennoprzecinkowe	53
	W języku Ruby nie ma typów prostych	54
	Kiedy brakuje obiektów... ..	55
	Prawda, fałsz i nil	55
	Decyzje, decyzje	57
	Pętle	58

	Więcej o łańcuchach	60
	Symbole	63
	Tablice	63
	Tablice mieszające	65
	Wyrażenia regularne	65
	Nasza pierwsza klasa	66
	Operacje na zmiennych egzemplarzy	68
	Obiekt pyta: Kim jestem?	70
	Dziedziczenie, podklasy i nadklasy	71
	Opcje argumentów	72
	Moduły	73
	Wyjątki	76
	Wątki	77
	Zarządzanie odrębnymi plikami z kodem źródłowym	78
	Podsumowanie	79
CZĘŚĆ II	WZORCE PROJEKTOWE W JĘZYKU RUBY	81
Rozdział 3.	Urozmaicenie algorytmów	
	za pomocą wzorca projektowego Template Method	83
	Jak stawić czoło typowym problemom	84
	Izolowanie elementów zachowujących dotychczasową formę	85
	Odkrywanie wzorca projektowego Template Method	88
	Metody zaczepienia	89
	Gdzie się właściwie podziały wszystkie te deklaracje?	92
	Typy, bezpieczeństwo i elastyczność	93
	Testy jednostkowe nie mają charakteru opcjonalnego	95
	Używanie i nadużywanie wzorca projektowego Template Method	97
	Szablony w praktycznych zastosowaniach	98
	Podsumowanie	99
Rozdział 4.	Zastępowanie algorytmu strategią	101
	Deleguj, deleguj i jeszcze raz deleguj	102
	Współdzielenie danych przez kontekst i strategię	104
	Jeszcze raz o naszym typowaniu	106
	Obiekty Proc i bloki kodu	107
	Krótką analizą kilku prostych strategii	111
	Używanie i nadużywanie wzorca projektowego Strategy	112
	Wzorzec Strategy w praktycznych zastosowaniach	113
	Podsumowanie	114
Rozdział 5.	Jak być na bieżąco dzięki wzorcowi Observer	117
	Trzymamy rękę na pulsie	117
	Jak skuteczniej trzymać rękę na pulsie?	119
	Wyodrębnianie mechanizmu umożliwiającego obserwację	122
	Stosowanie bloków kodu w roli obserwatorów	125
	Odmiany wzorca projektowego Observer	126

	Używanie i nadużywanie wzorca projektowego Observer	127
	Wzorzec Observer w praktycznych zastosowaniach	129
	Podsumowanie	130
Rozdział 6.	Budowa większej całości z części za pomocą wzorca Composite	133
	Całość i części	134
	Tworzenie kompozytów	136
	Dośkonalenie implementacji wzorca Composite z wykorzystaniem operatorów	140
	A może tablica w roli kompozytu?	141
	Kłopotliwe różnice	141
	Wskaźniki w obie strony	142
	Używanie i nadużywanie wzorca projektowego Composite	143
	Kompozyty w praktycznych zastosowaniach	145
	Podsumowanie	147
Rozdział 7.	Przeszukiwanie kolekcji z wykorzystaniem wzorca Iterator	149
	Iteratory zewnętrzne	149
	Iteratory wewnętrzne	152
	Iteratory wewnętrzne kontra iteratory zewnętrzne	153
	Nieźródlny moduł Enumerable	154
	Używanie i nadużywanie wzorca projektowego Iterator	156
	Iteratory w praktycznych zastosowaniach	158
	Podsumowanie	161
Rozdział 8.	Doprowadzanie spraw do końca za pomocą wzorca Command	163
	Eksplozja podklas	164
	Prostsze rozwiązanie	165
	Stosowanie bloków kodu w roli poleceń	166
	Rejestrowanie poleceń	167
	Wycofywanie operacji za pomocą wzorca Command	170
	Kolejkowanie poleceń	173
	Używanie i nadużywanie wzorca projektowego Command	174
	Wzorzec projektowy Command w praktycznych zastosowaniach	175
	Migracje w ramach interfejsu ActiveRecord	175
	Madeleine	176
	Podsumowanie	179
Rozdział 9.	Wypełnianie luk z wykorzystaniem wzorca Adapter	181
	Adaptory programowe	182
	Minimalne niedociągnięcia	184
	Czy alternatywą może być adaptowanie istniejących klas?	186
	Modyfikowanie pojedynczych obiektów	187
	Adaptować czy modyfikować?	188
	Używanie i nadużywanie wzorca projektowego Adapter	190
	Adaptory w praktycznych zastosowaniach	190
	Podsumowanie	191

Rozdział 10.	Tworzenie zewnętrznego reprezentanta naszego obiektu z wykorzystaniem wzorca Proxy	193
	Rozwiązaniem są pośrednicy	194
	Pośrednik ochrony	196
	Pośrednicy zdalni	197
	Pośrednicy wirtualni jako środek rozleniwiający	198
	Eliminacja najbardziej uciążliwych elementów implementacji pośredników	200
	Metody i przekazywanie komunikatów	201
	Metoda <code>method_missing</code>	202
	Wysyłanie komunikatów	203
	Bezbolesne implementowanie pośredników	203
	Używanie i nadużywanie pośredników	206
	Wzorec Proxy w praktycznych zastosowaniach	207
	Podsumowanie	208
Rozdział 11.	Doskonalenie obiektów za pomocą wzorca Decorator	211
	Dekoratory: lekarstwo na brzydki kod	212
	Dekoracja formalna	217
	Jak uprościć model delegacji zadań	218
	Dynamiczna alternatywa dla wzorca projektowego Decorator	219
	Opakowywanie metod	219
	Dekorowanie za pomocą modułów	220
	Używanie i nadużywanie wzorca projektowego Decorator	221
	Dekoratory w praktycznych zastosowaniach	222
	Podsumowanie	223
Rozdział 12.	Jak zyskać pewność, że to ten jedyny, z wykorzystaniem wzorca Singleton	225
	Jeden obiekt, dostęp globalny	225
	Zmienne i metody klasowe	226
	Zmienne klasowe	226
	Metody klasowe	227
	Pierwsza próba opracowania singletonu w Ruby	228
	Zarządzanie jedynym obiektem	229
	Upewnianie się, że istnieje tylko jeden	230
	Moduł Singleton	231
	Singletony leniwe i chciwe	232
	Konstrukcje alternatywne względem klasycznego singletonu	232
	Zmienne globalne jako singletony	232
	Klasy jako singletony	233
	Moduły jako singletony	235

	Pasy bezpieczeństwa kontra kaftan bezpieczeństwa	236
	Używanie i nadużywanie wzorca projektowego Singleton	237
	Czy singletony nie są przypadkiem	
	zwykłymi zmiennymi globalnymi?	237
	Właściwie iloma singletonami dysponujemy?	238
	Singletony i niezbędna wiedza	238
	Eliminowanie utrudnień związanych z testowaniem	240
	Singletony w praktycznych zastosowaniach	241
	Podsumowanie	242
Rozdział 13.	Wybór właściwej klasy za pomocą wzorca Factory	243
	Inny rodzaj kaczego typowania	244
	Powrót wzorca projektowego Template Method	246
	Sparametryzowane metody wytwórcze	248
	Klasy to po prostu obiekty	251
	Złe wieści: nasz program podbił świat	252
	Grupowe tworzenie obiektów	253
	Klasy to po prostu obiekty (raz jeszcze)	255
	Korzystanie z nazw	257
	Używanie i nadużywanie wzorców fabryk	258
	Wzorce fabryk w praktycznych zastosowaniach	258
	Podsumowanie	260
Rozdział 14.	Uproszczone konstruowanie obiektów	
	z wykorzystaniem wzorca Builder	263
	Budowa komputerów	264
	Klasy budowniczych polimorficznych	267
	Klasy budowniczych mogą też zapewnić	
	bezpieczeństwo tworzenia obiektów	270
	Klasy budowniczych wielokrotnego użytku	270
	Lepsza implementacja budowniczych	
	z wykorzystaniem magicznych metod	271
	Używanie i nadużywanie wzorca projektowego Builder	272
	Klasy budowniczych w praktycznych zastosowaniach	273
	Podsumowanie	274
Rozdział 15.	Łączenie systemu z wykorzystaniem interpretera	275
	Język dostosowany do realizowanego zadania	276
	Konstruowanie interpretera	277
	Interpreter odnajdywania plików	279
	Odnajdywanie wszystkich plików	279
	Wyszukiwanie plików według nazw	280
	Wielkie pliki i pliki zapisywalne	281
	Bardziej złożone operacje wyszukiwania z uwzględnieniem	
	logicznej negacji, koniunkcji i alternatywy	282

	Tworzenie drzewa AST	284
	Prosty analizator składniowy	284
	Interpreter bez analizatora składniowego?	286
	Analiza składniowa danych języka XML czy YAML?	287
	Generowanie złożonych analizatorów składniowych za pomocą narzędzia Racc	288
	A może wykorzystać analizator samego języka Ruby?	288
	Używanie i nadużywanie wzorca projektowego Interpreter	289
	Interpretery w praktycznych zastosowaniach	290
	Podsumowanie	291
CZĘŚĆ III	WZORCE DLA JĘZYKA RUBY	293
Rozdział 16.	Otwieranie systemów za pomocą języków dziedzinowych (DSL)	295
	Języki dziedzinowe	296
	Język DSL kopii zapasowej	297
	Czy to plik z danymi? Nie, to program!	297
	Budowa języka PackRat	299
	Łączenie opracowanych dotychczas elementów w jedną całość	300
	Krytyczne spojrzenie na język PackRat	301
	Doskonalenie języka PackRat	302
	Używanie i nadużywanie wewnętrznych języków DSL	305
	Wewnętrzne języki DSL w praktycznych zastosowaniach	305
	Podsumowanie	307
Rozdział 17.	Tworzenie niestandardowych obiektów techniką metaprogramowania	309
	Obiekty szyte na miarę metoda po metodzie	310
	Obiekty niestandardowe tworzone moduł po module	312
	Wyczarowywanie zupełnie nowych metod	313
	Rzut okiem na wewnętrzną strukturę obiektu	317
	Używanie i nadużywanie metaprogramowania	318
	Metaprogramowanie w praktycznych zastosowaniach	319
	Podsumowanie	322
Rozdział 18.	Konwencja ponad konfiguracją	323
	Interfejs użytkownika dobry dla... programistów	325
	Przewidywanie przyszłych potrzeb	326
	Oszczędźmy użytkownikom powtarzania swojej woli	326
	Udostępnianie szablonów	327
	Bramka wiadomości	327
	Wybór adaptera	329
	Ładowanie klas	331
	Dodanie prostych zabezpieczeń	333
	Ułatwianie użytkownikowi wejścia w świat naszego oprogramowania	335

	Podsumowanie przykładu bramki wiadomości	336
	Używanie i nadużywanie wzorca projektowego	
	Convention Over Configuration	337
	Wzorec Convention Over Configuration	
	w praktycznych zastosowaniach	338
	Podsumowanie	339
Rozdział 19.	Konkluzja	341
	DODATKI	345
Dodatek A	Instalacja języka Ruby	347
	Instalacja Ruby w systemie Microsoft Windows	347
	Instalacja Ruby w systemie Linux i pozostałych systemach	
	wywodzących się z systemu UNIX	348
	System Mac OS X	348
Dodatek B	Pogłębiona analiza	349
	Wzorce projektowe	349
	Ruby	350
	Wyrażenia regularne	352
	Blogi i witryny internetowe	352
	Skorowidz	353

ROZDZIAŁ 4.

Zastępowanie algorytmu strategią

W poprzednim rozdziale szukaliśmy odpowiedzi na pytanie: Jak zróżnicować wybraną część algorytmu? Jak sprawić, by trzeci krok procesu pięciokrokowego raz podejmował jedno działanie, a innym razem realizował nieco inne zadania? W rozdziale 3. przyjęliśmy rozwiązanie polegające na stosowaniu wzorca projektowego Template Method, czyli na tworzeniu klasy bazowej z metodą szablonową odpowiedzialną za ogólne przetwarzanie podklas charakterystycznych dla mechanizmów szczegółowych. Gdybyśmy raz chcieli realizować jeden wariant, by innym razem stosować mechanizm alternatywny, powinniśmy opracować dwie podklasy, po jednej dla obu scenariuszy.

Wzorzec projektowy Template Method ma, niestety, pewne wady, z których najpoważniejszą jest konieczność korzystania z relacji dziedziczenia (właśnie wokół niej opracowano ten wzorzec). Jak wiemy z rozdziału 1., stosowanie ścisłej relacji dziedziczenia niesie ze sobą wiele negatywnych konsekwencji. Niezależnie od wysiłku, jaki włożymy w rozważne projektowanie naszego kodu, podklasy będą ściśle związane ze swoją nadklasą, co w przypadku tej relacji jest zupełnie naturalne. Techniki opracowane na podstawie relacji dziedziczenia, w tym wzorzec Template Method, ograniczają więc elastyczność naszego kodu. Po wybraniu określonej wersji algorytmu (w naszym przypadku takim krokiem było utworzenie obiektu klasy HTML-Report), zmiana tego trybu na inny może się okazać niezwykle trudna. Gdybyśmy zastosowali wzorzec projektowy Template Method i zdecydowali się na zmianę formatu raportu, musielibyśmy utworzyć nowy obiekt raportu, np. obiekt klasy PlainTextReport, tylko po to, by użyć nowego formatu. Mamy inne wyjście?

DELEGUJ, DELEGUJ I JESZCZE RAZ DELEGUJ

Alternatywnym rozwiązaniem jest uwzględnienie rady sformułowanej przez Bandę Czworka i przypomnianą w rozdziale 1. tej książki: wybierajmy technikę delegowania zadań. Jaki będzie efekt rezygnacji z każdorazowego tworzenia podklas na rzecz wyodrębnienia kłopotliwego, zmienianego kodu do wyspecjalizowanej klasy? Będziemy wówczas mogli opracować całą rodzinę klas, po jednej dla każdej odmiany implementowanego mechanizmu. Poniżej przedstawiono przykład kodu odpowiedzialnego za formatowanie raportów w języku HTML przeniesionego na poziom odrębnej klasy:

```
class Formatter
  def output_report( title, text )
    raise 'Wywołano metodę abstrakcyjną'
  end
end

class HTMLFormatter < Formatter
  def output_report( title, text )
    puts('<html>')
    puts(' <head>')
    puts(" <title>#{title}</title>")
    puts(' </head>')
    puts(' <body>')
    text.each do |line|
      puts(" <p>#{line}</p> ")
    end
    puts(' </body>')
    puts('</html>')
  end
end
```

Klasę odpowiedzialną za formatowanie raportu w wersji tekstowej można by zaimplementować w następujący sposób:

```
class PlainTextFormatter < Formatter
  def output_report(title, text)
    puts("***** #{title} *****")
    text.each do |line|
      puts(line)
    end
  end
end
```

Udało nam się przenieść szczegółowy kod związany z formatowaniem danych wyników poza klasę Report, zatem jej definicja będzie teraz nieporównanie prostsza:

```
class Report
  attr_reader :title, :text
  attr_accessor :formatter
```

```

def initialize(formatter)
  @title = 'Raport miesięczny'
  @text = [ 'Wszystko idzie', 'naprawdę dobrze.' ]
  @formatter = formatter
end

def output_report
  @formatter.output_report( @title, @text )
end
end

```

Okazuje się, że wprowadzona zmiana tylko nieznacznie komplikuje proces korzystania z klasy Report w nowym kształcie. Musimy teraz przekazywać na wejściu jej konstruktora właściwy obiekt formatujący:

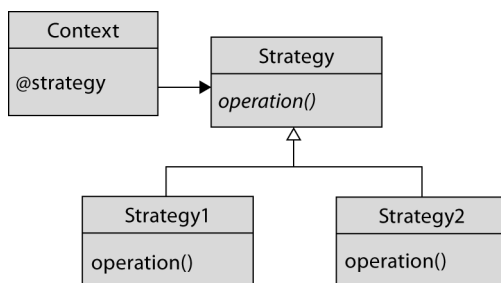
```

report = Report.new(HTMLFormatter.new)
report.output_report

```

Banda Czworka określiła technikę „wyodrębniania algorytmu do osobnego obiektu” mianem wzorca projektowego Strategy (patrz rysunek 4.1). Wzorzec Strategy opracowano w myśl założenia, zgodnie z którym warto definiować rodzinę obiektów, tzw. **strategii**, realizujących to samo zadanie w różny sposób (w naszym przypadku tym zadaniem jest formatowanie raportu). Wszystkie obiekty strategii nie tylko realizują to samo zadanie, ale też obsługują identyczny interfejs. W prezentowanym przykładzie ten wspólny interfejs ogranicza się do metody `output_report`. Skoro wszystkie obiekty strategii z zewnątrz wyglądają tak samo, użytkownik strategii (nazywany przez Bandę Czworka **klasą kontekstu** — ang. *context class*) może je traktować jak wymienne elementy. Oznacza to, że wybór strategii o tyle nie ma wielkiego znaczenia, że wszystkie te obiekty wyglądają podobnie i realizują tę samą funkcję.

RYСУNEK 4.1.
Wzorzec projektowy
Strategy



Z drugiej strony wybór strategii ma znaczenie ze względu na różnice w sposobie realizacji interesujących nas zadań. W naszym przykładzie jedna ze strategii formatowania generuje raport w języku HTML, druga generuje raport w formie zwykłego tekstu. Gdybyśmy zastosowali wzorzec Strategy w systemie obliczającym wysokość podatku dochodowego, moglibyśmy użyć jednej strategii do wyznaczania podatku płaconego przez etatowego pracownika i innej do obliczania podatku uiszczanego przez pracownika zatrudnionego na podstawie umowy o dzieło.

Wzorzec projektowy Strategy ma wiele praktycznych zalet. Przykład raportów pokazuje, że stosując ten wzorzec, możemy zapewnić skuteczniejszą izolację naszego kodu dzięki wyodrębnieniu zbioru strategii poza klasę główną. Wzorzec Strategy zwalnia klasę Report z jakiegokolwiek odpowiedzialności czy choćby konieczności składowania informacji o formacie generowanych raportów.

Co więcej, ponieważ wzorzec Strategy opiera się na relacji kompozycji i technice delegacji (zamiast na dziedziczeniu), zmiana strategii w czasie wykonywania programu nie stanowi żadnego problemu. Wystarczy wymienić obiekt strategii:

```
report = Report.new(HTMLFormatter.new)
report.output_report

report.formatter = PlainTextFormatter.new
report.output_report
```

Wzorzec projektowy Strategy ma jedną cechę wspólną ze wzorcem Template Method: oba wzorce umożliwiają nam koncentrowanie decyzji o wyborze sposobu realizacji zleczanych zadań w zaledwie jednym lub dwóch miejscach. W przypadku wzorca Template Method właściwą odmianę algorytmu wybieramy, wskazując konkretną podklasę; we wzorcu Strategy taki wybór sprowadza się do wskazania klasy strategii w czasie wykonywania programu.

WSPÓLDZIELENIE DANYCH PRZEZ KONTEKST I STRATEGIĘ

Jedną z największych zalet wzorca projektowego Strategy jest izolowanie kodu kontekstu i strategii w odrębnych klasach, co skutecznie dzieli dane pomiędzy dwa niezależne byty. Problem w tym, że musimy znaleźć jakiś sposób pokonania muru dzielącego oba byty, aby przekazywać informacje, którymi dysponuje kontekst i które są niezbędne do prawidłowego funkcjonowania strategii. Ogólnie mówiąc, mamy do wyboru dwa rozwiązania.

Po pierwsze, możemy konsekwentnie stosować dotychczasowe rozwiązanie polegające na przekazywaniu wszystkich danych niezbędnych do pracy obiektów strategii w formie argumentów (przekazywanych przez obiekt kontekstu podczas wywoływania metod obiektów strategii). Warto pamiętać, że w naszym przykładzie systemu raportującego obiekt raportu był przekazywany do obiektów formatujących za pośrednictwem argumentów wywołań metody `output_report`. Przekazywanie kompletnych danych ma tę zaletę, że pozwala skutecznie izolować kontekst od obiektów strategii. Strategie mają wspólny interfejs; kontekst korzysta tylko z tego interfejsu. Wady opisanej metody są widoczne w sytuacji, gdy musimy przekazywać mnóstwo złożonych danych, co do których nie mamy żadnych gwarancji, że rzeczywiście zostaną wykorzystane.

Po drugie, możemy przekazywać dane z kontekstu do strategii w formie referencji do samego obiektu kontekstu. Obiekt strategii może wówczas uzyskiwać niezbędne dane za pośrednictwem metod obiektu kontekstu. W naszym przykładzie systemu raportującego odpowiedni model mógłby mieć następującą postać:

```
class Report
  attr_reader :title, :text

  attr_accessor :formatter

  def initialize(formatter)
    @title = 'Raport miesięczny'
    @text = [ 'Wszystko idzie', 'naprawdę dobrze.' ]
    @formatter = formatter
  end

  def output_report
    @formatter.output_report(self)
  end
end
```

Obiekt klasy Report przekazuje strategii formatowania referencję do samego siebie; klasa formatująca może następnie uzyskiwać niezbędne dane za pośrednictwem nowych metod title i text. Poniżej przedstawiono przebudowaną klasę HTMLFormatter (przystosowaną do korzystania z referencji do obiektu klasy Report):

```
class Formatter
  def output_report(context)
    raise 'Wywołano metodę abstrakcyjną'
  end
end

class HTMLFormatter < Formatter
  def output_report(context)
    puts('<html>')
    puts(' <head>')
    puts(" <title>#{context.title}</title>")
    puts(' </head>')
    puts(' <body>')
    context.text.each do |line|
      puts(" <p>#{line}</p>")
    end
    puts(' </body>')
    puts('</html>')
  end
end
```

Chociaż opisana technika przekazywania kontekstu do strategii skutecznie upraszcza przepływ danych, warto mieć na uwadze to, że stosując wskazane rozwiązanie, zacieśniamy związki łączące kontekst i strategię. W ten sposób istotnie zwiększamy ryzyko wzajemnego uzależnienia klasy kontekstu i klas strategii.

JESZCZE RAZ O KACZYM TYPOWANIU

Przykładową aplikację generującą raporty, którą postugiwaliśmy się w tym rozdziale, zbudowano zgodnie z zaleceniami Bandy Czworka odnośnie do wzorca projektowego Strategy. Nasza rodzina strategii formatowania składa się z „abstrakcyjnej” klasy bazowej `Formatter` oraz dwóch podklas: `HTMLFormatter` i `PlainTextFormatter`. Prezentowany model nie najlepiej pasuje do języka Ruby, ponieważ klasa `Formatter` w praktyce nie realizuje żadnych działań — istnieje tylko po to, by definiować wspólny interfejs wszystkich klas formatujących. Wspomniany problem w żaden sposób nie wpływa na formalną poprawność naszego kodu — tak napisany program po prostu działa. Z drugiej strony takie rozwiązanie jest sprzeczne z przyjętą w języku Ruby filozofią tzw. kaczego typowania. Skoro klasy `HTMLFormatter` i `PlainTextFormatter` implementują wspólny interfejs przez zgodne definiowanie metody `output_report`, wprowadzanie sztucznego tworu (właśnie w formie klasy pozbawionej działań) potwierdzającego ten fakt nie ma większego sensu.

Możemy wyeliminować z naszego projektu klasę bazową `Formatter` za pomocą zaledwie kilku uderzeń w klawisz *Delete*. Ostatecznie nasz kod będzie miał następującą postać:

```
class Report
  attr_reader :title, :text
  attr_accessor :formatter

  def initialize(formatter)
    @title = 'Raport miesięczny'
    @text = [ 'Wszystko idzie', 'naprawdę dobrze.' ]
    @formatter = formatter
  end

  def output_report()
    @formatter.output_report(self)
  end
end

class HTMLFormatter
  def output_report(context)
    puts('<html>')
    puts(' <head>')
    # Wyświetla pozostałe dane tego obiektu...

    puts(" <title>#{context.title}</title>")
    puts(' </head>')
    puts(' <body>')
    context.text.each do |line|
      puts(" <p>#{line}</p>")
    end
    puts(' </body>')
    puts('</html>')
  end
end
```

```

class PlainTextFormatter
  def output_report(context)
    puts("***** #{context.title} *****")
    context.text.each do |line|
      puts(line)
    end
  end
end
end

```

Jeśli porównamy ten kod z poprzednią wersją, przekonamy się, że rezygnacja z klasy bazowej `Formatter` nie wymagała zbyt wielu dodatkowych zmian. Dzięki dynamicznej kontroli typów nadal możemy generować prawidłowe raporty. Chociaż obie wersje działają zgodnie z założeniami, do świata Ruby dużo lepiej pasuje model pozbawiony klasy bazowej `Formatter`.

OBIEKTY PROC I BLOKI KODU

Okazuje się, że eliminacja klasy bazowej nie jest jedynym sposobem dostosowania wzorca projektowego `Strategy` do charakteru języka programowania Ruby. Zanim jednak zrobimy kolejny krok, musimy wyjaśnić jeden z najciekawszych aspektów języka Ruby: bloki kodu i obiekty `Proc`.

Jako użytkownicy obiektowych języków programowania spędzamy mnóstwo czasu na myśleniu o obiektach i sposobach ich skutecznego łączenia. Warto jednak zwrócić uwagę na pewną asymetrię występującą w typowym postrzeganiu obiektów. Z reguły nie mamy problemu z wyodrębnianiem danych poza obiekt — możemy na przykład uzyskać wartość zmiennej `@text` obiektu raportu i przekazywać ją dalej niezależnie od pozostałych składowych tego obiektu. Z drugiej strony zazwyczaj przyjmujemy, że nasz kod jest ściśle i nierozdzielnie związany z poszczególnymi obiektami. Takie przekonanie oczywiście nie znajduje potwierdzenia w praktyce. Co w takim razie powinniśmy zrobić, aby wyodrębnić fragmenty kodu poza nasz obiekt i przekazywać je dalej tak jak zwykle obiekty? Okazuje się, że język Ruby oferuje z myślą o podobnych operacjach gotowe mechanizmy.

W języku Ruby `Proc` jest obiektem reprezentującym fragment kodu. Najpopularniejszym sposobem konstruowania obiektu `Proc` jest stosowanie metody `lambda`:

```

hello = lambda do
  puts('Witaj')
  puts('Jestem wewnątrz obiektu Proc.')
end

```

W języku Ruby fragment kodu zawarty pomiędzy słowem kluczowym `do` a słowem `end` określa się mianem **bloku kodu**¹. Metoda `lambda` zwraca nowy obiekt `Proc`, czyli

¹ Stosuje się też terminy **domknięcia** (ang. *closure*) oraz **lambdy** (stąd nazwa naszej metody tworzącej obiekt `Proc`).

kontener obejmujący cały kod zdefiniowany pomiędzy słowami `do` i `end`. W przedstawionym przykładzie obiekt `Proc` jest wskazywany przez zmienną `hello`. Kod reprezentowany przez obiekt `Proc` możemy wykonać, wywołując metodę `call` (trudno sobie wyobrazić lepszą nazwę). Jeśli w przedstawionym przykładzie wywołamy metodę `call` obiektu `Proc`:

```
hello.call
```

otrzymamy komunikaty:

```
Witaj
Jestem wewnątrz obiektu Proc.
```

Wyjątkowo cennym aspektem obiektów `Proc` jest zdolność do funkcjonowania w otaczającym je środowisku. Oznacza to, że wszelkie zmienne widoczne w momencie tworzenia obiektu `Proc` pozostają dla tego obiektu dostępne także w czasie wykonywania programu. Na przykład w poniższym fragmencie kodu istnieje tylko jedna zmienna `name`:

```
name = 'Jan'
proc = Proc.new do
  name = 'Maria'
end

proc.call
puts(name)
```

Kiedy wykonamy ten kod, zmiennej `name` w pierwszej kolejności zostanie przypisany łańcuch `"Jan"`, po czym (po wywołaniu obiektu `Proc`) wartość tej zmiennej zostanie zastąpiona przez łańcuch `"Maria"`.

Oznacza to, że ostatecznie Ruby wyświetli łańcuch `"Maria"`. Z myślą o programistach, którym konstrukcja `do-end` wydaje się zbyt rozbudowana, Ruby oferuje krótszą składnię złożoną tylko z nawiasów klamrowych. Poniżej przedstawiono przykład użycia tej składni do utworzenia naszego obiektu `Proc`:

```
hello = lambda {
  puts('Witaj, jestem wewnątrz obiektu Proc')
}
```

Wielu programistów języka Ruby przyjęło konwencję, zgodnie z którą konstrukcję `do-end` stosuje się do bloków wielowierszowych, a nawiasy klamrowe stosuje się do bloków jednowierszowych². W tej sytuacji najlepsza wersja prezentowanego przykładu będzie miała następującą postać:

```
hello = lambda {puts('Witaj, jestem wewnątrz obiektu Proc')}
```

² Uczciwość nakazuje wspomnieć o istotnej różnicy dzielącej obie konstrukcje. W wyrażeniach języka Ruby nawiasy klamrowe mają wyższy priorytet od pary słów kluczowych `do` i `end`. Krótko mówiąc, nawiasy klamrowe są ściślej związane z wyrażeniami. Specyfika nawiasów klamrowych jest widoczna dopiero wtedy, gdy zrezygnujemy z opcjonalnych nawiasów okrągłych.

Obiekty Proc pod wieloma względami przypominają metody. Podobnie jak metody, obiekty Proc nie tylko reprezentują fragmenty kodu, ale też mogą zwracać wartości. Obiekt Proc zawsze zwraca ostatnią wartość wyznaczoną w danym bloku — aby zwrócić wartość z poziomu takiego obiektu, wystarczy się upewnić, że jest ona wyznaczana przez jego ostatnie wyrażenie. Wszelkie wartości zwracane przez obiekty Proc są przekazywane do kodu wywołującego za pośrednictwem metody call. Oznacza to, że kod w postaci:

```
return_24 = lambda {24}
puts(return_24.call)
```

wyświetli wartość:

```
24
```

Dla obiektów Proc można też definiować parametry, choć składnia tego rodzaju definicji jest dość nietypowa. Zamiast otaczać listę parametrów tradycyjnymi nawiasami okrągłymi, należy je umieszczać pomiędzy dwoma symbolami |:

```
multiply = lambda {|x, y| x * y}
```

Kod w tej formie definiuje obiekt Proc otrzymujący dwa parametry, wyznaczający ich iloczyn i zwracający otrzymaną wartość. Aby wywołać obiekt Proc z parametrami, wystarczy je przekazać na wejściu metody call:

```
n = multiply.call(20, 3)
puts(n)
n = multiply.call(10, 50)
puts(n)
```

Po wykonaniu tego kodu otrzymamy następujące wartości:

```
60
500
```

Możliwość przekazywania bloków kodu jest na tyle przydatna, że twórcy Ruby zdecydowali się zdefiniować z myślą o tej technice specjalną, skróconą konstrukcję składniową. Jeśli chcemy przekazać blok kodu na wejściu metody, wystarczy go dołączyć do jej wywołania. Tak wywołana metoda może następnie wykonać ten blok kodu za pomocą słowa kluczowego yield. Poniżej zdefiniowano przykładową metodę wyświetlającą komunikat, wykonującą otrzymany blok kodu i wyświetlającą kolejny komunikat:

```
def run_it
  puts("Przed wykonaniem yield")
  yield
  puts("Po wykonaniu yield")
end
```

A tak może wyglądać wywołanie metody run_it. Warto pamiętać, że w ten sposób dopisujemy blok kodu na koniec wywołania naszej metody:

```

run_it do
  puts('Witaj')
  puts('Przybywam do Ciebie z wnętrza bloku kodu')
end

```

Kiedy doklejamy blok kodu do wywołania metody (jak w powyższym przykładzie), wspomniany blok (który w rzeczywistości ma postać obiektu Proc) jest przekazywany do tej metody w formie dodatkowego, niewidocznego parametru. Słowo kluczowe `yield` powoduje więc wykonanie tego parametru. W wyniku wykonania powyższego kodu na ekranie zostaną wyświetlone następujące komunikaty:

```

Przed wykonaniem yield
Witaj
Przybywam do Ciebie z wnętrza bloku kodu
Po wykonaniu yield

```

Jeśli przekazany blok kodu sam otrzymuje jakieś parametry, należy je przekazać za pośrednictwem słowa kluczowego `yield`. Oznacza to, że poniższy kod:

```

def run_it_with_parameter
  puts("Przed wykonaniem yield")
  yield(24)
  puts("Po wykonaniu yield")
end

run_it_with_parameter do |x|
  puts('Witaj z wnętrza bloku kodu')
  puts("Parametr x ma wartość #{x}")
end

```

wyświetli następujące komunikaty:

```

Przed wykonaniem yield
Witaj z wnętrza bloku kodu
Parametr x ma wartość 24
Po wykonaniu yield

```

W niektórych przypadkach warto zdefiniować wprost parametr reprezentujący blok kodu — w takim przypadku blok przekazany na wejściu naszej metody ma postać argumentu przypominającego typowe parametry. W tym celu należy umieścić specjalny parametr na końcu listy parametrów tej metody. Tak przekazany parametr (poprzedzony znakiem `&`) zostanie przypisany obiektowi Proc utworzonemu na podstawie bloku kodu dołączonego do wywołania danej metody. Oznacza to, że odpowiednikiem przedstawionej powyżej metody `run_it_with_parameters` będzie następująca konstrukcja:

```

def run_it_with_parameter(&block)
  puts("Przed wykonaniem yield")
  block.call(24)
  puts("Po wykonaniu yield")
end

```

Symbol & można z powodzeniem stosować także w przeciwnym kierunku. Gdybyśmy umieścili obiekt Proc w zmiennej i chcieli go przekazać na wejściu metody oczekującej bloku kodu, moglibyśmy przekonwertować ten obiekt do postaci wymaganego bloku, poprzedzając odpowiedni parametr właśnie znakiem &:

```
my_proc = lambda {|x| puts("Parametr x ma wartość #{x}")}
run_it_with_parameter(&my_proc)
```

KRÓTKA ANALIZA KILKU PROSTYCH STRATEGII

Co bloki kodu i obiekty Proc mają wspólnego ze wzorcem projektowym Strategy? Najkrócej mówiąc, strategię można postrzegać jako blok wykonywalnego kodu, który „wie”, jak zrealizować określone zadanie (np. formatowanie tekstu) i który opakowano w formie obiektu. Przytoczona definicja brzmi znajomo — obiekt Proc bywa określany właśnie jako fragment kodu opakowany w formie obiektu.

Wróćmy teraz do naszego przykładowego systemu formatującego raporty, gdzie zastosowanie strategii w formie obiektu Proc okazuje się wyjątkowo proste. Zmiany, które musimy wprowadzić w kodzie klasy Report, ograniczają się do poprzedzenia parametru przekazywanego na wejściu metody initialize symbolem & i zmiany nazwy wywoływanej metody z output_report na call:

```
class Report
  attr_reader :title, :text
  attr_accessor :formatter

  def initialize(&formatter)
    @title = 'Raport miesięczny'
    @text = [ 'Wszystko idzie', 'naprawdę dobrze.' ]
    @formatter = formatter
  end

  def output_report
    @formatter.call( self )
  end
end
```

Z nieco inną sytuacją mamy jednak do czynienia w przypadku klas odpowiedzialnych za właściwe formatowanie. Musimy teraz stworzyć obiekty Proc zamiast egzemplarzy naszych wyspecjalizowanych klas formatujących:

```
HTML_FORMATTER = lambda do |context|
  puts('<html>')
  puts(' <head>')
  puts(" <title>#{context.title}</title>")
  puts(' </head>')
  puts(' <body>')
  context.text.each do |line|
```

```

    puts(" <p>#{line}</p>" )
  end
  puts(' </body>' )
end

```

Skoro dysponujemy już kodem formatującym w formie obiektu Proc, możemy przystąpić do tworzenia raportów. Ponieważ dysponujemy obiektem Proc, a konstruktor klasy Report oczekuje bloku kodu, tworząc nowy obiekt tej klasy musimy poprzedzić wspomniany obiekt Proc znakiem &:

```

report = Report.new &HTML_FORMATTER
report.output_report

```

Po co w ogóle zajmujemy się problemem implementacji strategii w formie obiektu Proc? Jednym z powodów jest brak konieczności definiowania specjalnych klas dla poszczególnych strategii — wystarczy opakować kod w ramach obiektu Proc. Co więcej, stosując tę technikę możemy tworzyć strategię niemal z niczego, przekazując bloki kodu na wejściu istniejącej metody. Możemy zaimplementować na przykład mechanizm formatujący raporty tekstowe w formie następującego bloku kodu:

```

report = Report.new do |context|
  puts("***** #{context.title} *****")
  context.text.each do |line|
    puts(line)
  end
end

```

Programistom nieprzyzwyczajonym do tego rodzaju konstrukcji bloki kodu mogą się wydawać dziwaczne. Z drugiej strony powinniśmy pamiętać, że proponowana technika implementacji wzorca Strategy umożliwia zastąpienie klasę kontekstu, klasę bazową strategii i wiele konkretnych strategii (wraz ze wszystkimi niezbędnymi obiektami) pojedynczą klasą kontekstu i kilkoma blokami kodu.

Czy to oznacza, że powinniśmy raz na zawsze zapomnieć o strategiach implementowanych w postaci odrębnych klas? Nie do końca. Strategie w formie bloków kodu zdają egzamin tylko wtedy, gdy ich interfejs jest stosunkowo prosty i obejmuje zaledwie jedną metodę. Trudno się temu dziwić, skoro call jest jedyną metodą, którą możemy wywoływać dla obiektów Proc. Jeśli implementowane strategie wymagają interfejsu złożonego z większej liczby metod, nie mamy wyboru — musimy skorzystać z tradycyjnych klas. Jeśli jednak interfejs strategii jest prosty, koniecznie powinniśmy rozważyć użycie bloków kodu.

UŻYWANIE I NADUŻYWANIE WZORCA PROJEKTOWEGO STRATEGII

Najczęstszą przyczyną błędnego implementowania wzorca projektowego Strategy jest niewłaściwe definiowanie interfejsu pomiędzy kontekstem a strategiami. Musimy pamiętać, że naszym celem jest wyodrębnienie kompletnego, spójnego i mniej lub

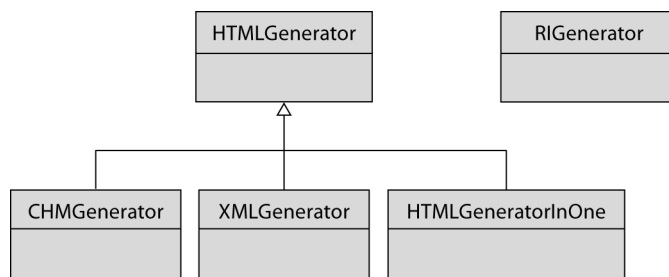
bardziej autonomicznego zadania poza obiekt kontekstu i delegowanie go do obiektu strategii. Powinniśmy zwracać szczególną uwagę zarówno na szczegóły interfejsu łączącego kontekst ze strategią, jak i na związki występujące pomiędzy obiema stronami tej relacji. Stosowanie wzorca Strategy będzie bezcelowe, jeśli zwiążemy kontekst z pierwszą strategią na tyle mocno, że uzupełnienie projektu o drugą, trzecią i kolejne strategie okaże się niemożliwe.

WZORZEC STRATEGIY W PRAKTYCZNYCH ZASTOSOWANIACH

Narzędzie rdoc (dołączane do dystrybucji języka Ruby) zawiera wiele mechanizmów opracowanych na podstawie klasycznego, zaproponowanego przez Bandę Czworka i opartego na klasach wzorca projektowego Strategy. Zadaniem tego narzędzia jest generowanie dokumentacji na podstawie kodu źródłowego. rdoc oferuje możliwość dokumentowania zarówno programów napisanych w Ruby, jak i programów opracowanych w C i (ratunku!) programów stworzonych w języku FORTRAN. Narzędzie rdoc wykorzystuje wzorec Strategy do obsługi poszczególnych języków programowania — każdy analizator składniowy (właściwy dla języków C, Ruby i FORTRAN) ma postać strategii stworzonej z myślą o innych danych wejściowych.

Narzędzie rdoc oferuje też użytkownikowi wybór formatu wyjściowego — możemy wybrać jedną z wielu odmian języka HTML, język XML bądź jeden z formatów wykorzystywanych przez polecenie ri języka Ruby. Jak nietrudno odgadnąć, każdy z tych formatów wyjściowych także jest obsługiwany przez wyspecjalizowaną strategię. Relacje łączące poszczególne klasy strategii programu rdoc dobrze ilustrują ogólne podejście do problemu dziedziczenia stosowane w języku Ruby. Związki klas reprezentujących poszczególne strategie przedstawiono na rysunku 4.2.

RYSUNEK 4.2.
Klasy generujące
narzędzia rdoc



Jak widać na rysunku 4.2, istnieją cztery powiązane ze sobą strategie formatowania danych wyjściowych (określanych w programie rdoc mianem generatorów) i jedna strategia autonomiczna. Wszystkie cztery powiązane strategie generują dokumentację w podobnym formacie — znane wszystkim konstrukcje `<coś tam>` otoczone nawiasami

ostrymi </co>am>³. Ostatnia strategia generuje dane wyjściowe na potrzeby polecenia `ri` języka Ruby, które nie przypominają ani kodu XML-a, ani kodu HTML-a. Jak wynika z diagramu UML przedstawionego na rysunku 4.2, relacje łączące poszczególne klasy odzwierciedlają szczegóły implementacyjne: klasy generujące dokumentację w formatach HTML, CHM i XML z natury rzeczy współdzielą znaczną część kodu, stąd decyzja twórców `rdoc` o zastosowaniu relacji dziedziczenia. Klasa `RIGenerator` generuje zupełnie inne dane wynikowe (w formacie całkowicie niezwiązanym z rodziną języków XML i HTML). Twórcy `rdoc` nie zdecydowali się na współdzielenie jednej nadklasy przez wszystkie klasy generatorów tylko dlatego, że wszystkie te klasy implementują ten sam interfejs. Ograniczyli się jedynie do zaimplementowania właściwych metod w opisanych klasach.

Okazuje się, że z dobrym przykładem wykorzystania obiektu `Proc` w roli lekkiej strategii mamy do czynienia na co dzień — takie rozwiązanie zastosowano w przypadku tablic. Jeśli chcemy posortować zawartość tablicy języka Ruby, wywołujemy metodę `sort`:

```
a = ['ryszard', 'michał', 'jan', 'daniel', 'robert']
a.sort
```

Metoda `sort` domyślnie sortuje obiekty składowane w tabeli, stosując „naturalny” porządek. Co w takim razie powinniśmy zrobić, jeśli chcemy użyć innego schematu porządkowania elementów? Jak należałoby na przykład posortować łańcuchy według długości? Wystarczy przekazać strategię porównywania obiektów w formie bloku kodu:

```
a.sort { |a,b| a.length <=> b.length }
```

Metoda `sort` wywołuje nasz blok kodu za każdym razem, gdy będzie zmuszona porównać dwa elementy sortowanej tablicy. Nasz blok powinien zwracać wartość 1, jeśli pierwszy element jest większy od drugiego; wartość 0, jeśli oba elementy są równe, i wartość -1, jeśli większy jest drugi element. Nieprzypadkowo działanie tego kodu przypomina zachowanie operatora `<=>`.

PODSUMOWANIE

Wzorzec projektowy Strategy reprezentuje nieco inną, opartą na delegowaniu zadań propozycję rozwiązywania tego samego problemu, który jest również rozwiązywany przez wzorzec Template Method. Zamiast upychania zmiennych części algorytmu w podklasach, implementujemy każdą z wersji tego algorytmu w odrębnym obiekcie. Możemy następnie urozmaicać działania tego algorytmu przez wskazywanie obiektowi kontekstu różnych obiektów strategii. Oznacza to, że o ile jedna strategia na przykład generuje raport w formacie HTML, o tyle inna może generować ten sam raport

³ Format CHM jest odmianą HTML-a wykorzystywaną do generowania plików pomocy firmy Microsoft. Chciałbym też podkreślić, że to kod narzędzia `rdoc` — nie ja — sugeruje, że XML jest odmianą języka HTML.

w formacie PDF; podobnie, jedna strategia może wyznaczać wysokość podatku odprowadzanego przez pracownika etatowego, by inna wyliczała podatek należny od pracownika zatrudnionego na podstawie umowy o dzieło.

Mamy do dyspozycji kilka rozwiązań dotyczących przekazywania niezbędnych danych pomiędzy obiektem kontekstu a obiektem strategii. Możemy albo przekazywać wszystkie dane w formie parametrów wywoływanych metod obiektu strategii, albo po prostu przekazywać obiektowi strategii referencję do całego obiektu kontekstu.

Bloki kodu języka Ruby, które w istocie mają postać kodu opakowywanego w ramach tworzonego na bieżąco obiektu (a konkretnie obiektu Proc), wprost doskonale nadają się do błyskawicznego konstruowania prostych obiektów strategii.

Jak się niedługo przekonamy, wzorzec projektowy Strategy przypomina (przynajmniej na pierwszy rzut oka) wiele innych wzorców. Wzorzec Strategy wymusza stosowanie obiektu (nazywanego kontekstem), który odpowiada za realizację określonego zadania. Okazuje się jednak, że wykonanie tego zadania wymaga od kontekstu skorzystania z pomocy innego obiektu (określanego mianem strategii). Z bardzo podobnym modelem mamy do czynienia w przypadku wzorca projektowego Observer (obserwatora), gdzie obiekt realizujący pewne zadania kieruje wywołania do drugiego obiektu, bez którego jego funkcjonowanie byłoby niemożliwe.

Okazuje się, że jedyną różnicą dzielącą oba te wzorce jest intencja programisty. Celem wzorca projektowego Strategy jest dostarczenie obiektowi kontekstu innego obiektu, który „wie”, jak wykonać określoną wersję algorytmu. Zupełnie inny cel przyświeca programiście stosującemu wzorzec Observer — otóż stosujemy go wtedy, gdy... Chyba powinniśmy te rozważania przenieść do innego rozdziału (tak się składa, że będzie to kolejny rozdział).