PHP5. Obiekty, wzorce, narzędzia

Autor: Matt Zandstra
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-868-6
Tytuł oryginału: PHP 5 Objects, Patterns, and Practice
Format: B5, stron: 464

Profesjonalne techniki programowania obiektowego w PHP5

- Poznaj zasady projektowania i programowania obiektowego
- Zastosuj wzorce projektowe podczas tworzenia aplikacji
- Wykorzystaj narzędzia wspomagające pracę programisty PHP5

Wraz z rosnącą popularnością języka PHP zwiększa się również zakres jego zastosowań. Za pomocą PHP tworzy się już nie tylko proste dynamiczne witryny WWW i fora dyskusyjne, ale również rozbudowane aplikacje sieciowe, wykorzystywane często w dużych przedsiębiorstwach. Już w PHP4 zaimplementowano pewne mechanizmy ułatwiające tworzenie rozbudowanych systemów, jednak dopiero PHP5 stał się w pełni obiektowym językiem programowania pozwalającym na korzystanie z wszystkich wynikających z tego możliwości.

„PHP5. Obiekty, wzorce i narzędzia" stanowi dokładne omówienie wszystkich technik obiektowych w kontekście zastosowania ich podczas tworzenia aplikacji w PHP5. Zawiera przegląd podstawowych i zaawansowanych cech PHP5 związanych z obiektowością. Przedstawia przykłady najczęściej wykorzystywanych wzorców projektowych i zasady ich stosowania. Książka opisuje również narzędzia, które mogą okazać się bardzo przydatne podczas tworzenia rozbudowanych aplikacji, służące do tworzenia dokumentacji i kontroli wersji plików.

- Podstawowe pojęcia z dziedziny obiektowości
- Obsługa obiektów
- Wyjątki i obsługa błędów
- Projektowanie obiektowe
- Modelowanie obiektów w języku UML
- Wzorce projektowe
- Stosowanie pakietu PEAR
- Generowanie dokumentacji za pomocą PHPDocumentor
- Zarządzanie wersjami plików w systemie CVS
- Tworzenie pakietów instalacyjnych

Przekonaj się, jak potężnym narzędziem jest najnowsza wersja języka PHP

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

Spis treści

Rozdział 11.

Reprezentacja i realizacja zadań

W niniejszym rozdziale zaczniemy wreszcie działać i przyjrzymy się wzorcom projektowym, które są pomocne w wykonywaniu zadań — od interpretacji minijęzyków po hermetyzacje algorytmów.

Rozdział poświęcony będzie:

- *Wzorcowi Interpreter* — umożliwiającemu konstruowanie interpreterów minijęzyków nadające się do wbudowywania w aplikacje interfejsów skryptowych.
- *Wzorcowi Strategy* — zakładającemu identyfikowanie algorytmów stosowanych w systemie i ich hermetyzację do postaci osobnych, własnych typów.
- *Wzorcowi Observer* — tworzącemu zaczepy umożliwiające powiadamianie obiektów o zdarzeniach zachodzących w systemie.
- *Wzorcowi Visitor* — rozwiązującemu problem aplikacji operacji do wszystkich węzłów drzewa obiektów.
- *Wzorcowi Command* — obiektom poleceń przekazywanym pomiędzy częściami systemu.

Wzorzec Interpreter

Języki programowania powstają i są rozwijane (przynajmniej z początku) w innych językach programowania. PHP został na przykład „spisany" w języku C. Nic więc nie stoi na przeszkodzie, abyśmy, posługując się PHP, zdefiniowali i wykorzystywali własny język programowania. Oczywiście każdy utworzony tak język będzie powolny i dość ograniczony, ale nie oznacza to, że będzie bezużyteczny — minijęzyki są całkiem przydatne, co postaram się zademonstrować w tym rozdziale.

Tworząc interfejsy WWW (ale również interfejsy wiersza poleceń) w języku PHP, dajemy użytkownikowi dostęp do pewnego zestawu funkcji. Zawsze w takim przypadku stajemy przed wyborem pomiędzy prostotą korzystania z interfejsu a zakresem możliwości oddawanych w ręce użytkownika. Im więcej możliwości dla użytkownika, tym z reguły bardziej złożony i rozdrobniony interfejs. Bardzo pomocne jest tu staranne zaprojektowanie interfejsu i rozpoznanie potrzeb użytkowników — jeśli 90 procent z nich wykorzystuje jedynie 30 procent (tych samych) funkcji systemu, koszt udostępniania maksimum funkcjonalności może okazać się za wysoki w stosunku do efektów. Można wtedy rozważyć uproszczenie systemu pod kątem „przeciętnego" użytkownika. Ale co wtedy z owymi 10 procentami użytkowników zaawansowanych korzystających z kompletu zaawansowanych funkcji systemu? Ich potrzeby można by zaspokoić inaczej, na przykład udostępniając im wewnętrzny język programowania, w którym będą mogli odwoływać się do wszystkich funkcji systemu.

Mamy już co prawda pod ręką jeden język programowania. Chodzi o PHP. Moglibyśmy więc udostępnić go użytkownikom i pozwolić im na tworzenie własnych skryptów:

```
$form_input = "print file_get_contents('/etc/passwd');";
eval($form_input);
```

Jednakże takie rozszerzenie dostępności systemu wydaje się szaleństwem. Jeśli Czytelnik nie jest przekonany co do nonsensowności tego pomysłu, powinien przypomnieć sobie o dwóch kwestiach: bezpieczeństwie i złożoności. Kwestia bezpieczeństwa jest dobrze ilustrowana w naszym przykładzie — umożliwiając użytkownikom uzupełnianie systemu o ich własny kod w języku PHP, dajemy im w rzeczy samej pełny dostęp do serwera, na którym działa nasza aplikacja. Równie dużym problemem jest jednak złożoność — niezależnie od przejrzystości kodu aplikacji przeciętny użytkownik będzie miał problemy z jej rozszerzeniem, zwłaszcza, jeśli ma z nią kontakt jednie za pośrednictwem okna przeglądarki.

Problemy te można wyeliminować, opracowując i udostępniając użytkownikom własny minijęzyk. Można w nim połączyć elastyczność, zredukować możliwość wyrządzania szkód przez użytkowników i równocześnie zadbać o zwartość całości.

Wyobraźmy sobie aplikację do tworzenia quizów. Autorzy mieliby układać pytania i ustalać reguły oznaczania poprawności odpowiedzi udzielanych przez uczestników quizu. Chodziłoby o to, żeby quizy toczyły się bez interwencji operatora, choć część odpowiedzi miała by być wprowadzana przez uczestników w polach tekstowych.

Oto przykładowe pytanie:

```
Ilu członków liczy banda Design Patterns?
```

Poprawnymi odpowiedziami są „cztery" albo „4". Możemy utworzyć interfejs WWW, który pozwala twórcy quizu angażować do rozpoznawania poprawnych odpowiedzi wyrażenia regularne:

```
^4|cztery$
```

Jednak od twórców quizów rzadko wymaga się biegłości w konstruowaniu wyrażeń regularnych — są oni cenieni raczej ze względu na wiedzę ogólną. Aby uprościć im życie, można więc zaimplementować przyjaźniejszy mechanizm rozpoznawania poprawnych odpowiedzi:

```
$input equals "4" or $input equals "cztery"
```

Mamy tu propozycję języka programowania obsługującego zmienne, operator o nazwie `equals` oraz operacje logiczne (`or` czy `and`). Programiści uwielbiają nadawać nazwy swoim dziełom, nadajmy więc językowi jego miano — MarkLogic. Język miałby być łatwo rozszerzalny, bo już oczyma wyobraźni widzimy postulaty zwiększenia jego możliwości. Odłóżmy chwilowo na bok kwestię analizy leksykalnej, skupiając się na mechanizmie wykorzystania języka w czasie wykonania do generowania ocen odpowiedzi. Tu właśnie zastosowanie znajdzie wzorzec Interpreter.

Implementacja

Nasz język składa się z wyrażeń (to znaczy elementów, dla których da się obliczyć wartości). Z tabeli 11.1 wynika jasno, że nawet tak prosty język jak MarkLogic musi uwzględniać wiele elementów.

Tabela 11.1. *Elementy gramatyki języka MarkLogic*

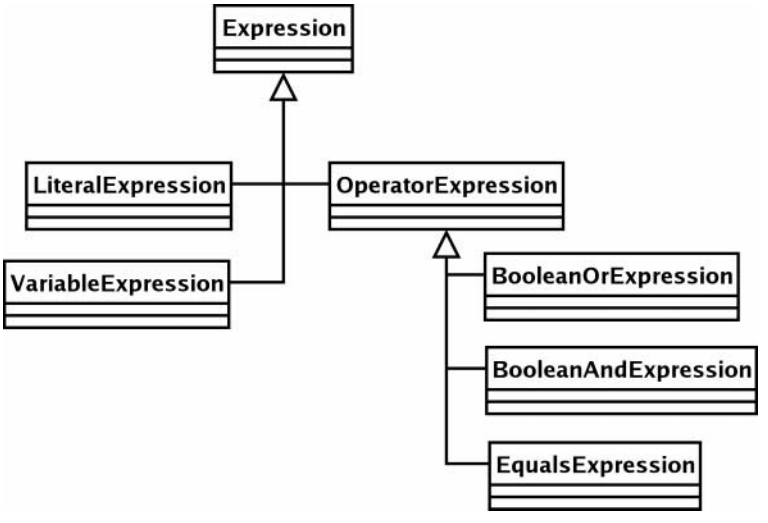
Opis	Nazwa w notacji EBNF	Nazwa klasy	Przykład
Zmienna	`variable`	`VariableExpression`	`$input`
Literał łańcuchowy	`<stringLiteral>`	`LiteralExpression`	`"cztery"`
Logiczne i	`andExpr`	`BooleanAndExpression`	`$input equals '4' and $other equals '6'`
Logiczne lub	`orExpr`	`BooleanOrExpression`	`$input equals '4' or $other equals '6'`
Test równości	`equalsExpr`	`EqualsExpression`	`$input equals '4'`

W tabeli 11.1 mamy między innymi kolumnę nazw EBNF. Cóż to za nazwy? To notacja wykorzystywana do opisu gramatyki języka. EBNF to skrót od *Extended BackusNaur Form* (rozszerzona notacja Backusa-Naura). Notacja ta składa się z szeregu wierszy (zwanych regułami produkcyjnymi), w których znajduje się nazwa i opis przyjmujący postać odniesień do innych reguł produkcyjnych (ang. *productions*) i symboli końcowych (ang. *terminals*), których nie da się już wyrazić odwołaniami do kolejnych reguł produkcyjnych. Naszą gramatykę w notacji EBNF można by zapisać następująco:

```
expr ::= operand (orExpr | andExpr)*
operand ::= ( '(' expr ')' | <stringLiteral> | variable ) (eqExpr)*
orExpr ::= 'or' operand
andExpr ::= 'and' operand
equalsExpr ::= 'equals' operand
variable ::= '$' <word>
```

Niektóre z symboli mają znaczenie specjalne (znane z notacji wyrażeń regularnych): na przykład gwiazdka (*) oznacza zero lub więcej wystąpień, a pionowa kreska (`|`) to to samo co w języku naturalnym „lub”. Elementy grupujemy za pośrednictwem nawiasów. W powyższym przykładzie wyrażenie (`expr`) składa się z operandu (`operand`), z którym występuje zero lub więcej wyrażeń logicznej sumy (`orExpr`) bądź logicznego iloczynu (`andExpr`). Operand może być wyrażeniem ujętym w nawiasy, ciągiem ograniczonym znakami cudzysłowu (tej reguły produkcyjnej nie ma co prawda w powyższym przykładzie) albo zmienną (`variable`). Jeśli przyzwyczaić się do ciągłego odsyłania od jednej reguły produkcyjnej do kolejnej, notacja EBNF staje się całkiem poręczna.

Na rysunku 11.1 mamy prezentację elementów gramatyki w postaci klas.



Rysunek 11.1. *Klasy wzorca Interpreter obsługujące język MarkLogic*

Jak widać, klasa `BooleanAndExpression` i jej „rodzeństwo" dziedziczą po klasie `OperatorExpression`. Wszystkie te klasy realizują bowiem operacje na obiektach wyrażeń (obiektach klasy `Expression`). Klasy `VariableExpression` i `LiteralExpression` operują wprost na wartościach.

Wszystkie obiekty hierarchii `Expression` implementują metodę `interpret()` zdefiniowaną w abstrakcyjnej klasie bazowej hierarchii, czyli właśnie w klasie `Expression`. Metoda ta oczekuje przekazania w wywołaniu obiektu klasy `Context` wykorzystywanego w roli wspólnego repozytorium danych. Każdy obiekt klasy `Expression` może składować dane w obiekcie klasy `Context`, który jest przekazywany pomiędzy obiektami hierarchii `Expression`. Aby dało się w prosty sposób wyodrębniać dane z obiektu `Context`, klasa bazowa `Expression` implementuje metodę `getKey()` zwracającą unikalny uchwyt. Zobaczmy, jak całość działa w praktyce z implementacjami abstrakcji `Expression`:

```
abstract class Expression {
    abstract function interpret(Context $context);

    function getKey() {
        return (string)$this;
    }
}

class LiteralExpression extends Expression {
    private $value;

    function __construct($value) {
        $this->value = $value;
    }

    function interpret(Context $context) {
        $context->replace($this, $this->value);
    }
}
```

```
class Context {
    private $expressionstore = array();
    function replace(Expression $exp, $value) {
        $this->expressionstore[$esp->getKey()] = $value;
    }

    function lookup(Expression $exp) {
        return $this->expressionstore[$exp->getKey()];
    }
}

$context = new Context();
$literal = new LiteralExpression('cztery');
$literal->interpret($context);
print $context->lookup($literal);
```

Zacznijmy od klasy `Context`. Jak widać, jest ona w istocie jedynie fasadą tablicy asocjacyjnej reprezentowanej składową `$expressionstore` i służącej do przechowywania danych. Metoda `replace()` klasy `Context` przyjmuje na wejście obiekt klasy `Expression`, który występuje w roli klucza tablicy asocjacyjnej, oraz wartość dowolnego typu ładującą w tablicy asocjacyjnej w parze z przekazanym kluczem. Klasa udostępnia również metodę `lookup()` umożliwiającą odczyt zapisanych w tablicy danych.

Klasa `Expression` definiuje abstrakcyjną metodę `interpret()` i konkretną metodę `getKey()`, która na podstawie bieżącego obiektu (`$this`) generuje unikalną w obrębie hierarchii etykietę. Etykieta powstaje w wyniku rzutowania wartości `$this` na typ `string`. Domyślny wynik konwersji obiektu na kontekst ciągu znaków to ciąg zawierający znakową reprezentację identyfikatora obiektu.

```
class PrintMe {}
$test = new PrintMe();
print "$test";

// wydruk:
// Object id #1
```

Metoda `getKey()` czyni z tej cechy języka narzędzie generowania klucza do tabeli asocjacyjnej. Metoda ta jest wykorzystywana w kontekście wywołań `Context::lookup()` i `Context::replace()`, konwertując argumenty typu `Expression` na ich reprezentację znakową.

Rzutowanie obiektów na ciągi na potrzeby tablic asocjacyjnych jest użyteczne, ale nie zawsze bezpieczne. W czasie pisania tej książki język PHP5 zawsze przy okazji takiego rzutowania generował ciąg z identyfikatorem obiektu. Zdaje się jednak, że docelowo konwersja ta ma uwzględniać implementację metody specjalnej `__toString()`. Oparcie wyniku konwersji na wywołaniu tej metody oznaczać będzie unieważnienie gwarancji wygenerowania unikalnego ciągu. Gwarancję tę będzie można przywrócić, jawnie implementując w klasie bazowej wykorzystywanej hierarchii metodę `__toString()` jako metodę finalną, uniemożliwiając jej przesłanianie w klasach pochodnych:

```
final function __toString() {
    return (string)$this;
}
```

Klasa `LiteralExpression` definiuje konstruktor przyjmujący wartość dowolnego typu zapisywaną w składowej `$value`. Metoda `interpret()` klasy wymaga zaś przekazania obiektu klasy `Context`. Jej implementacja sprowadza się do wywołania metody `Context::replace()` z przekazaniem w wywołaniu wartości zwracanej przez metodę `getKey()` i wartością składowej `$value`. Schemat taki będziemy obserwować w pozostałych klasach wyrażeń. Metoda `interpret()` zawsze wypisuje wyniki swojego działania za pośrednictwem obiektu `Context`.

W prezentowanym kodzie nie zabrakło przykładowego kodu użytkującego klasy konkretyzującego obiekty klas `Context` i `LiteralExpression` (z wartością „cztery"), a następnie przekazującego obiekt `Context` do wywołania `LiteralExpression::interpret()`. Metoda ta zapisuje parę klucz-wartość w obiekcie `Context`, z którego można ją później wyodrębnić wywołaniem `lookup()`.

Zdefiniujmy pozostałe klasy symboli końcowych naszej gramatyki. Klasa `VariableExpression` jest już nieco bardziej złożona:

```
class VariableExpression extends Expression {
    private $name;
    private $val;

    function __construct($name, $val=null) {
        $this->name = $name;
        $this->val = $val;
    }

    function interpret(Context $context) {
        if (!is_null($this->val)) {
            $context->replace($this, $this->val);
            $this->val = null;
        }
    }

    function setValue($value) {
        $this->val = $value;
    }

    function getKey() {
        return $this->name;
    }
}

$context = new Context();
$myvar = new VariableExpression('input', 'cztery');
$myvar->interpret($context);
print $context->lookup($myvar);
// wydruk:
// cztery

$newvar = new VariableExpression('input');
$newvar->interpret($context);
print $context->lookup($newvar);
// wydruk:
// cztery

$myvar->setValue("pięć");
```

```
$myvar->interpret($context);
print $context->lookup($myvar);
// wydruk:
// pięć

print $context->lookup($newvar);
// wydruk:
// pięć
```

Klasa `VariableExpression` przyjmuje przy konstrukcji parę wartości: nazwę zmiennej i jej wartość. Udostępnia też metodę `setValue()`, aby użytkownicy mogli przypisywać do zmiennych nowe wartości.

Metoda `interpret()` sprawdza przede wszystkim, czy składowa `$val` obiektu ma wartość niepustą. Jeśli tak, wartość ta jest zapisywana w kontekście, po czym do składowej `$val` przypisywana jest wartość pusta, na wypadek, gdyby metoda `interpret()` została ponownie wywołana po tym, jak inny egzemplarz `VariableExpression` o tej samej nazwie zmienił wartość zapisaną w kontekście. W rozszerzeniach języka trzeba by przewidzieć operowanie na obiektach `Expression`, tak aby zmienna mogła zawierać wyniki testów i operacji. Na razie jednak taka implementacja `VariableExpression` jest wystarczająca. Zauważmy, że przesłoniliśmy w niej implementację `getKey()`, tak aby wartość klucza konstytuowana była nie identyfikatorem egzemplarza klasy, a ciągiem zapisanym w składowej `$name`.

Wyrażenia operatorów w naszym języku każdorazowo operują na dwóch obiektach `Expression` (obsługujemy bowiem wyłącznie operatory dwuargumentowe). Zasadne jest więc wyprowadzenie ich ze wspólnej klasy bazowej. Oto klasa `OperatorExpression`:

```
abstract class operatorExpression extends Expression {
    protected $l_op;
    protected $r_op;

    function __construct(Expression $l_op, Expression $r_op) {
        $this->l_op = $l_op;
        $this->r_op = $r_op;
    }

    function interpret(Context $context) {
        $this->l_op->interpret($context);
        $this->r_op->interpret($context);
        $result_l = $context->lookup($this->l_op);
        $result_r = $context->lookup($this->r_op);
        $this->doOperation($context, $result_l, $result_r);
    }

    protected abstract function doOperation(Context $context,
                                            $result_l,
                                            $result_r);
}
```

Klasa `OperatorExpression` to klasa abstrakcyjna. Implementuje co prawda metodę `interpret()`, ale definiuje również abstrakcyjną metodę `doInterpret()`.

Konstruktor oczekuje przekazania dwóch obiektów klasy `Expression`: `$l_op` i `$r_op`, do których referencje zapisywane są w zabezpieczonych składowych obiektu.

Implementacja metody `interpret()` rozpoczyna się od wywołania `interpret()` na rzecz obu operandów (jeśli pamiętasz poprzedni rozdział, zauważysz tu zapewne zastosowanie wzorca Composite). Po ewaluacji operandów metoda `interpret()` musi pozyskać zwracane przez nie wartości. Odwołuje się do niech za pośrednictwem metody `Context::lookup()` wywoływanej dla obu składowych. Dalej następuje wywołanie `doInterpret()`, o którego wyniku decyduje jednak implementacja w klasach pochodnych.

Spójrzmy na jej implementację w klasie `EqualsExpression`, porównującej dwa obiekty klasy `Expression`:

```
class EqualsExpression extends OperatorExpression {
    protected function doOperation(Context $context, $result_l, $result_r) {
        $context->replace($this, $result_l == $result_r);
    }
}
```

Klasa `EqualsExpression` implementuje jedynie metodę `doOperation()`, w ramach której porównuje wartości operandów przekazanych z metody `interpret()` klasy nadrzędnej, a wynik porównania umieszcza w przekazanym obiekcie `Context`.

Implementację klas wyrażeń wieńczą klasy wyrażeń logicznych — `BooleanOrExpression` i `BooleanAndExpression`:

```
class BooleanOrExpression extends OperatorExpression {
    protected function doOperation(Context $result_l, $result_r) {
        $context->replace($this, $result_l || $result_r);
    }
}

class BooleanAndExpression extends OperatorExpression {
    protected function doOperation(Context $result_l, $result_r) {
        $context->replace($this, $result_l && $result_r);
    }
}
```

Zamiast sprawdzania równości aplikujemy tu operatory operacji logicznej sumy (w `BooleanOrExpression`) bądź logicznego iloczynu (`BooleanAndExpression`). Wynik operacji jest przekazywany do metody `Context::replace()`.

Mamy już bazę kodu wystarczającą do wykonania prezentowanego wcześniej fragmentu kodu naszego minijęzyka. Oto on:

```
$input equals "4" or $input equals "cztery"
```

Powyższe wyrażenie możemy odwzorować w hierarchii `Expression` w sposób następujący:

```
$context = new Context();
$input = new VariableExpression('input');
$statement = new BooleanOrExpression(
    new EqualsExpression($input, new LiteralExpression('cztery')),
    new EqualsExpression($input, new LiteralExpression('4')),
);
```

Konkretyzujemy tu zmienną o nazwie `input`, ale wstrzymujemy się z przypisaniem jej wartości. Następnie tworzymy obiekt wyrażenia sumy logicznej `BooleanExpression` operującego na wynikach dwóch porównań realizowanych obiektami `EqualsExpression`. W pierwszym porównaniu uczestniczą: obiekt wyrażenia wartości (`ValueExpression`) przechowywanej w `$input` z obiektem ciągu znaków (`LiteralExpression`) zawierającym ciąg „cztery"; w drugim porównywany jest ten sam obiekt wartości `$input` z ciągiem znaków „4".

Po takim rozpracowaniu wyrażenia z przykładowego wiersza kodu możemy przystąpić do obliczenia wartości zmiennej `input` i uruchomienia mechanizmu oceny:

```
foreach (array("cztery", "4", "52") as $val) {
    $input->setValue($val);
    print "$val:\n";
    $statement->interpret($context);
    if ($context->lookup($statement)) {
        print "Znakomita odpowiedź\n\n";
    } else {
        print "Do oślej ławki\n\n";
    }
}
```

Mamy tu trzykrotne uruchomienie tego samego kodu, dla trzech różnych wartości zmiennej wejściowej. Za pierwszym razem ustawiamy tymczasową zmienną `$val` na „cztery", przypisując ją następnie do obiektu `VariableExpression` za pośrednictwem metody `setValue()`. Dalej wywołujemy metodę `interpret()` na rzecz szczytowego obiektu `Expression` (obiektu `BooleanOrExpression` zawierającego referencję do pozostałych obiektów wyrażeń uczestniczących w instrukcji). Spójrzmy na sposób realizacji tego wywołania:

- Obiekt `$statement` wywołuje metodę `intepret()` na rzecz składowej `$l_op` (pierwszego obiektu klasy `EqualsExpression`).
- Pierwszy z obiektów `EqualsExpression` wywołuje z kolei metodę `interpret()` na rzecz *swojej* składowej `$l_op` (referencji do obiektu `VariableExpression` przechowującego wartość „cztery").
- Obiekt `VariableExpression` zapisuje swoją bieżącą wartość do wskazanego obiektu klasy `Context` (wywołaniem `Context::replace()`).
- Pierwszy z obiektów `EqualsExpression` wywołuje metodę `interpret()` na rzecz *swojej* składowej `$r_op` (referencji do obiektu `LiteralExpression` inicjowanego wartością „cztery").
- Obiekt `LiteralExpression` rejestruje właściwą dla siebie parę klucz-wartość w obiekcie kontekstu.
- Pierwszy z obiektów `EqualsExpression` odczytuje wartości `$l_op` („cztery") i `$r_op` („cztery") z obiektu kontekstu.
- Pierwszy z obiektów `EqualsExpression` porównuje odczytane w poprzednim kroku wartości i rejestruje wynik porównania (`true`) wraz z właściwym sobie kluczem w obiekcie kontekstu.

- Po powrocie w górę drzewa obiektów następuje wywołanie metody `interpret()` na rzecz składowej `$r_op` obiektu `$statement`. Wartość tego wywołania (tym razem `false`) obliczana jest identycznie jak dla pierwszego obiektu `$l_op`.
- Obiekt `$statement` odczytuje wartości swoich operandów z obiektu kontekstu i porównuje je za pośrednictwem operatora `||`. Suma logiczna wartości `true` i `false` daje `true` i taka wartość jest ostatecznie składowana w obiekcie kontekstu.

Cały ten proces to zaledwie pierwsza iteracja pętli. Oto wynik wykonania wszystkich trzech przebiegów:

```
cztery:
Znakomita odpowiedź

4:
Znakomita odpowiedź

52:
Do oślej ławki
```

Być może zrozumienie tego, co dzieje się w powyższym kodzie, wymagać będzie kilkukrotnej lektury opisu — znów mamy bowiem do czynienia z pomieszaniem pomiędzy drzewami klas a hierarchiami obiektów. Klasy wyrażeń tworzą hierarchię dziedziczenia `Expression`, ale równocześnie obiekty tych klas są w czasie wykonania formowane w strukturę drzewiastą. Należy jednak pamiętać o rozróżnieniu obu hierarchii.

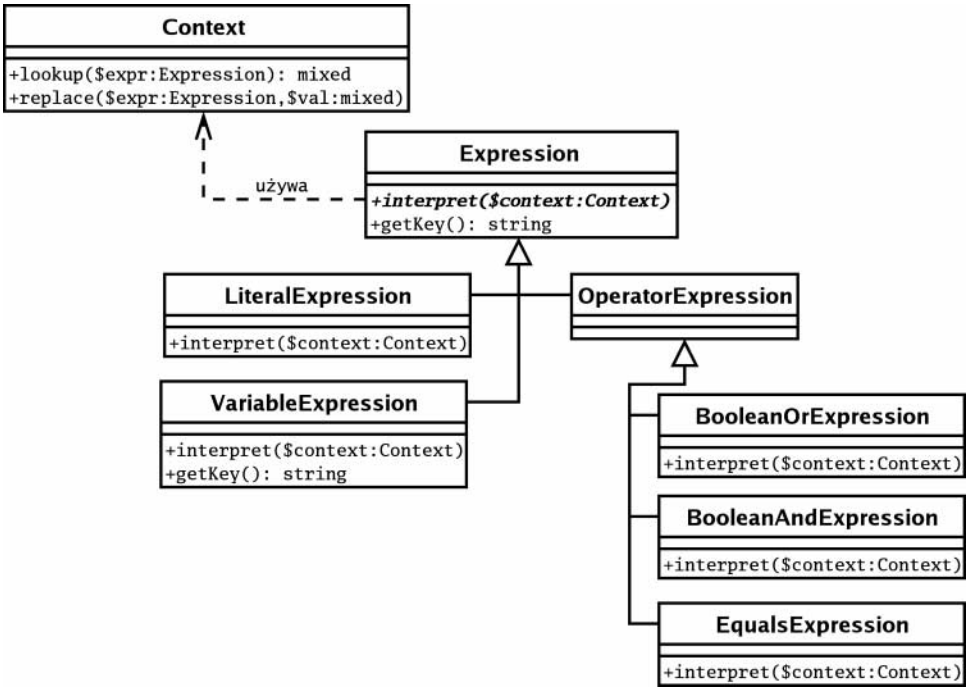
Kompletny diagram klas dla tego przykładu prezentowany jest na rysunku 11.2.

Ciemne strony wzorca Interpreter

Po ułożeniu rdzenia hierarchii klas wzorca Interpreter jego rozbudowa jest już dość prosta, odbywa się jednak przez tworzenie coraz to nowych klas. Z tego względu wzorzec Interpreter najlepiej stosować do implementacji języków stosukowo uproszczonych. W obliczu potrzeby pełnoprawnego języka programowania należałoby raczej skorzystać z gotowych narzędzi przeznaczonych do analizy leksykalnej i implementacji własnej gramatyki.

Dalej, klasy wzorca Interpreter często realizują bardzo podobne zadania, warto więc pilnować, aby nie dochodziło w nich do niepotrzebnego powielania kodu.

Wiele osób, przymierzając się do pierwszego w swoim wykonaniu wdrożenia wzorca Interpreter, rozczarowuje się odkryciem faktu, że wzorzec ten nie obejmuje analizy leksykalnej. Oznacza to, że nie wystarczy on do implementacji gotowego mechanizmu skryptowego rozszerzania aplikacji. Przykładowa implementacja analizy leksykalnej mocno uproszczonego języka prezentowana jest w dodatku B.



Rysunek 11.2. *Wdrożenie wzorca Interpreter*

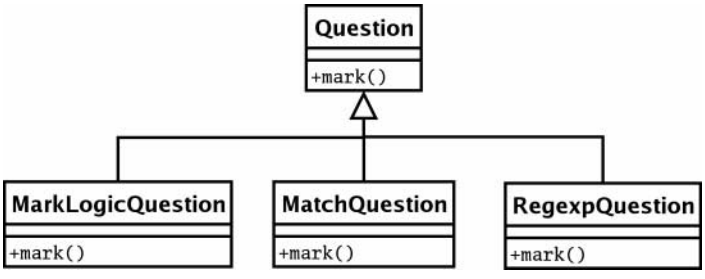
Wzorzec Strategy

Klasy często obciążane są nadmierną liczbą zadań. To zrozumiałe: niemal zawsze tworzymy je z myślą o kilku podstawowych funkcjach. W trakcie kodowania okazuje się, że niektóre z tych funkcji trzeba zmieniać w zależności od okoliczności. Oznacza to konieczność podziału klasy na podklasy. I zanim się ktokolwiek obejrzy, projekt zostaje rozdarty przeciwnymi nurtami.

Problem

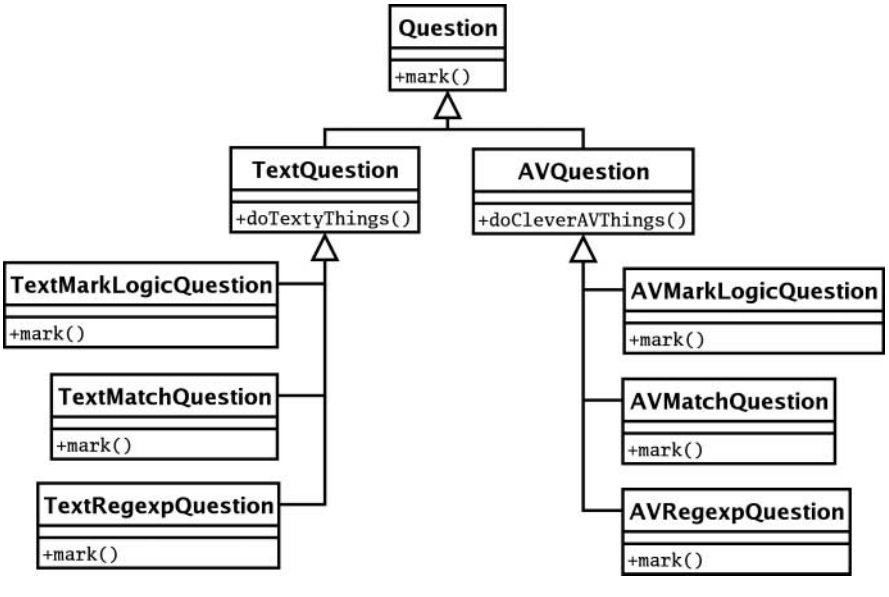
Ponieważ zdołaliśmy ostatnio opracować implementację miniaturowego języka oceny, trzymajmy się przykładu z quizami. Quizy nie mogą się obejść bez pytań, skonstruujemy więc klasę `Question` (pytanie) i wyposażymy ją w metodę oceny — `mark()`. Wszystko w porządku, dopóki nie pojawi się potrzeba obsługiwania różnych mechanizmów oceniania.

Załóżmy, że mamy zaimplementować ocenę wedle języka MarkLogic, ocenę na podstawie prostego dopasowania odpowiedzi i ocenę z dopasowaniem przy użyciu wyrażeń regularnych. W pierwszym podejściu moglibyśmy zróżnicować projekt pod kątem tych mechanizmów oceny, jak na rysunku 11.3.



Rysunek 11.3. *Definiowanie klas pochodnych wedle strategii oceny*

Całość będzie się sprawdzać dopóty, dopóki ocena pozostanie jedynym zmiennym aspektem hierarchii. Wyobraźmy sobie jednak, że zażądano od nas dodatkowo obsługi różnego rodzaju pytań: czysto tekstowych i opartych na materiale audiowizualnym. Powstaje problem uwzględnienia dwóch kierunków zmian w jednym drzewie dziedziczenia — patrz rysunek 11.4.



Rysunek 11.4. *Wyróżnianie klas pochodnych według dwóch kryteriów podziału*

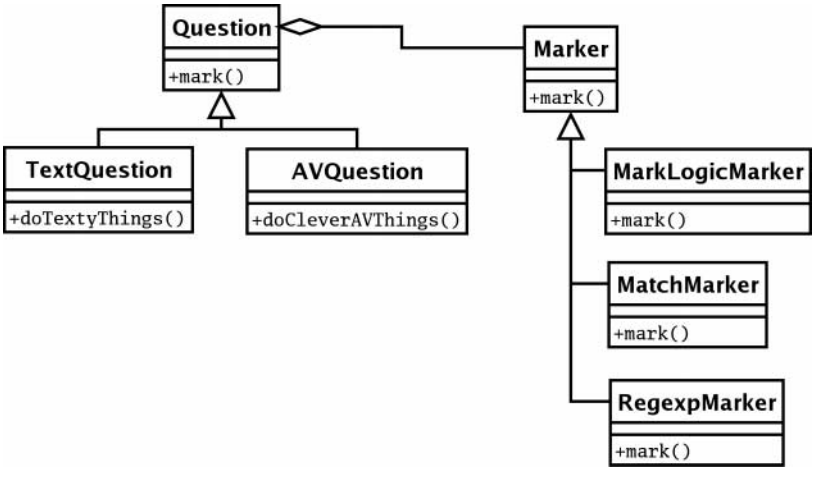
Nie tylko doszło do podwojenia (niemal) liczby klas w hierarchii, ale i do powielenia kodu. Nasza logika oceniania jest bowiem powielona w obu podgałęziach hierarchii dziedziczenia.

Jeśli kiedykolwiek staniesz w obliczu powielania algorytmu w równoległych gałęziach hierarchii dziedziczenia (powielania tak przez wydzielanie klas pochodnych, jak i rozbudowywanie instrukcji warunkowych), powinieneś rozważyć wyodrębnienie algorytmu do jego własnego typu.

Implementacja

Wzorzec Strategy (strategia), podobnie jak cała gama najlepszych wzorców, łączy prostotę z wielkimi możliwościami. Kiedy klasy muszą obsługiwać wielorakie implementacje interfejsu (u nas są to wielorakie mechanizmy oceny), wzorzec ten zakłada zaniechanie rozbudowywania oryginalnej hierarchii klas, zalecając wyodrębnienie owych implementacji do osobnego typu.

Odnosząc to do naszego przykładu, powiedzielibyśmy, że najlepiej byłoby wyodrębnić osobny typ mechanizmu oceny — `Marker`. Nową strukturę projektu ilustruje rysunek 11.5.



Rysunek 11.5. *Wyodrębnienie algorytmów do osobnego typu*

To kolejny znakomity przykład wdrożenia jednej z podstawowych zasad projektowych promowanych przez Bandę Czworga (i nie tylko), a mówiącej o wyższości kompozycji nad dziedziczeniem. Definiując i hermetyzując algorytmy oceny, redukujemy liczbę pochodnych w hierarchii dziedziczenia i zwiększamy równocześnie elastyczność systemu. Możemy go bowiem w dogodnych momentach uzupełniać o następne strategie oceny bez konieczności wprowadzania jakichkolwiek zmian w klasach hierarchii `Question`. Wszystkie klasy tej hierarchii mają do swojej dyspozycji egzemplarz klasy `Marker`, a interfejs klas udostępnia metodę oceny `mark()`. Szczegóły implementacji są dla wywołującego tę metodę zupełnie nieistotne.

Oto hierarchia `Question` wyrażona kodem źródłowym:

```
abstract class Question {
    protected $prompt;
    protected $marker;

    function __construct($prompt, Marker $marker) {
        $this->marker = $marker;
        $this->prompt = $prompt;
    }

    function mark($response) {
```

```
        return this->marker->mark($response);
    }
}

class TextQuestion extends Question {
    // operacje charakterystyczne dla prezentacji pytań w formie tekstowej...
}

class AVQuestion extends Question {
    // operacje charakterystyczne dla prezentacji pytania z materiałem audiowizualnym...
}
```

Szczegóły implementacyjne rozróżniające klasy `TextQuestion` i `AVQuestion` pozostawiłem wyobraźni Czytelnika. Najważniejsze z naszego punktu widzenia funkcje tych klas zdefiniowane zostały bowiem w klasie bazowej `Question`, która ponadto przechowuje w składowej `$marker` obiekt oceny (obiekt klasy `Marker`). Kiedy następuje wywołanie metody `Question::mark()` z argumentem reprezentującym odpowiedź uczestnika quizu, realizacja wywołania w klasie `Question` polega na oddelegowaniu wywołania do odpowiedniej metody obiektu `Marker`.

Zdefiniujmy klasę obiektów `Marker`:

```
abstract class Marker {
    protected $test;

    function __construct($test) {
        $this->test = $test;
    }

    abstract function mark($response);
}

class MarkLogicMarker extends Marker {
    private $engine;
    function __construct($test);
        parent::construct($test);
        // $this->engine = new MarkParse($test);
    }

    function mark($response) {
        // return $this->engine->evaluate($response);
        // na razie działa "na niby":
        return true;
    }
}

class MatchMarker extends Marker {
    function mark($response) {
        return ($this->test == $response);
    }
}

class RegexpMarker extends Marker {
    function mark($response) {
        return (preg_match($this->test, $response));
    }
}
```

W implementacji klas hierarchii `Marker` niewiele jest elementów zaskakujących — w rzeczy samej niewiele z nich wymaga w ogóle jakiegokolwiek komentarza. Zauważmy jedynie, że obiekty klasy `MarkLogicMarker` są przystosowane do korzystania z analizatora leksykalnego, którego kod jest prezentowany w dodatku B. Jednak na potrzeby tego przykładu możemy ten aspekt klasy pominąć, więc metoda `MarkLogicMarker::mark()` realizuje na razie ocenę „na pół gwizdka", zwracając za każdym razem `true`. Najważniejsza w tej hierarchii jest definiowana nią struktura, nie zaś szczegóły implementacji poszczególnych strategii ocen. Struktura ta ma zaś umożliwiać przełączanie mechanizmu oceny pomiędzy obiektami hierarchii `Marker` bez uszczerbku dla klasy `Question`, która się do tego mechanizmu odwołuje.

Wciąż pozostaje oczywiście kwestia podjęcia decyzji co do zastosowania jednego z konkretnych obiektów hierarchii `Marker`. Problem ten rozwiązuje się w praktyce na dwa sposoby. Pierwszy polega na wyborze strategii oceny na etapie układania quizu przez jego autora, a wybór sprowadza się do zaznaczenia odpowiedniego pola w formularzu. Drugi sposób to rozróżnianie mechanizmu oceny na podstawie struktury ciągu określającego bazę oceny. Jeśli baza wyrażona jest prostą odpowiedzią, wybierany jest mechanizm prostego porównania-dopasowania (`MatchMarker`):

```
pięć
```

Wybór mechanizmu MarkLogic sygnalizowany jest znakiem dwukropka poprzedzającego wyrażenie oceny:

```
:$input equals 'pięć'
```

Z kolei ocena na podstawie dopasowania wyrażenia regularnego wybierana jest w przypadku rozpoznania w ciągu wzorca odpowiedzi znaków ukośników ograniczających wyrażenie:

```
/pi../
```

Oto kod ilustrujący stosowanie poszczególnych klas:

```
$markers = array(new RegexpMarker("/pi../")),
                 new MatchMarker("pięć"),
                 new MarkLogicMarker('$input equals "pięć"'));

foreach ($markers as $marker) {
    print get_class($marker)."\n";
    $question = new TextQuestion("Ile boków ma pięciobok?", $marker);
    foreach(array("pięć", "cztery")) as $response) {
        print "\todpowiedź: $response";
        if ($question->mark($response)) {
            print "wyśmienita odpowiedź\n";
        } else {
            print "pomyłka\n";
        }
    }
}
```

Konstruujemy powyżej trzy obiekty strategii oceny, z których każdy jest następnie wykorzystywany do konstrukcji obiektu pytania `TextQuestion`. Następnie każdy z takich obiektów jest konfrontowany z dwoma przykładowymi odpowiedziami.

Klasa `MarkLogicMark` jest w swej obecnej postaci jedynie makietą, a jej metoda `mark()` każdorazowo zwraca wartość `true`. Oznaczony komentarzem kod da się jednak uruchomić w połączeniu z przykładową implementacją analizatora leksykalnego prezentowaną w dodatku B; można też ją przystosować do współpracy z analizatorami autorstwa osób trzecich.

Oto wynik wykonania powyższego kodu:

```
RegexpMarker:
    odpowiedź: pięć - wyśmienita odpowiedź
    odpowiedź: cztery - pomyłka
MatchMarker:
    odpowiedź: pięć - wyśmienita odpowiedź
    odpowiedź: cztery - pomyłka
MarkLogicMarker:
    odpowiedź: pięć - wyśmienita odpowiedź
    odpowiedź: cztery - wyśmienita odpowiedź
```

Klasa `MarkLogicMarker` jest w tej chwili jedynie atrapą. Jej metoda oceny daje zawsze wartość `true`, przez co w powyższym kodzie obie udzielone odpowiedzi są rozpoznawane jako poprawne.

W tym przykładzie obserwowaliśmy przekazywanie konkretnych danych od użytkownika (podającego na wejście wartość zmiennej `$response`) do obiektu strategii oceny; przekazanie odbywało się za pośrednictwem metody `Question::mark()`. W pewnych sytuacjach nie zawsze znana z góry jest ilość informacji wymaganych przez obiekt, na rzecz którego wywoływana jest operacja. Decyzję co do ilości i rodzaju pozyskiwanych danych można więc oddelegować, przekazując do obiektu strategii egzemplarz obiektu reprezentującego użytkownika. Wtedy obiekt strategii może wywoływać na rzecz obiektu użytkownika metody zwracające wymagane dane.

Wzorzec Observer

Znamy już pojęcie ortogonalności jako jednej z cnót projektu. Jednym z naszych (programistów) celów powinno być konstruowanie komponentów, które można swobodnie zmieniać albo przenosić, a których modyfikacje nie przenoszą się na pozostałe komponenty. Jeśli każda zmiana jednego z komponentów systemu prowokuje szereg zmian w innej części systemu, programowanie zmienia się w wyszukiwanie i poprawianie błędów wprowadzanych w coraz większej liczbie.

Rzecz jasna nie zawsze da się osiągnąć pożądaną ortogonalność projektu. Elementy systemu muszą przecież dysponować referencjami do pozostałych części systemu. Można jednak minimalizować zawiązywane w ten sposób zależności. Obserwowaliśmy już choćby różne przykłady zastosowań polimorfizmu, dzięki któremu użytkownik musi jedynie poznać i korzysta wyłącznie z interfejsu komponentu, zaś jego właściwa implementacja pozostaje poza zakresem jego zainteresowań.

W pewnych okolicznościach komponenty można oddalić od siebie jeszcze bardziej. Weźmy jako przykład klasę odpowiedzialną za pośredniczenie w dostępie użytkownika do systemu:

```
class Login {
    const LOGIN_USER_UNKNOWN = 1;
    const LOGIN_WRONG_PASS = 2;
    const LOGIN_ACCESS = 1;
    private $status = array();

    function handleLogin($user, $pass, $ip) {
        switch(rand(1, 3)) {
            case 1:
                $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
                $ret = true; break;
            case 2:
                $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
                $ret = false; break;
            case 3:
                $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
                $ret = false; break;
        }
        return $ret;
    }

    private function setStatus($status, $user, $ip) {
        $this->status = array($status, $user, $ip);
    }

    function getStatus() {
        return $this->status;
    }
}
```

Klasa ta imituje proces logowania się użytkownika w systemie — wynik logowania określany jest losowo, na podstawie wartości wywołania funkcji `rand()`. Znacznik statusu użytkownika może w wyniku „logowania" przyjąć wartość `LOGIN_ACCESS` (przyznany dostęp do systemu), `LOGIN_WRONG_PASS` (niepoprawne hasło) bądź `LOGIN_USER_UNKNOWN` (niepoprawne konto).

Ponieważ klasa `Login` to strażnik systemowych skarbów, będzie cieszyć się w czasie implementacji projektu (i zapewne również później) szczególną uwagą. Może się okazać, że w przyszłości kierownictwo działu marketingu zażąda utrzymywania w rejestrze logowania nazw domenowych użytkowników. Łatwo będzie wprowadzić żądane uzupełnienie:

```
function handleLogin($user, $pass, $ip) {
    switch(rand(1, 3)) {
        case 1:
            $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
            $ret = true; break;
        case 2:
            $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
            $ret = false; break;
        case 3:
```

```
            $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
            $ret = false; break;
        }
        Logger::logIP($user, $ip, $this->getStatus());
        return $ret;
    }
```

Nieustający w trosce o bezpieczeństwo administratorzy mogą z kolei zażądać powiadamiania o nieudanych próbach logowania. Trzeba będzie ponownie wrócić do implementacji metody `handleLogin()` i umieścić w jej ciele dodatkowe wywołanie:

```
if (!$ret) {
    Notifier::mailWarning($user, $ip, $this->getStatus());
}
```

Nie można wykluczyć, że w nieokreślonej przyszłości sekcja rozwoju ogłosi strategiczne połączenie działalności z pewnym dostawcą usług internetowych i zażąda ustawiania dla pewnej grupy użytkowników wyróżniających ich ciasteczek. I tak dalej, i tak dalej.

Wszystkie te żądania z osobna są proste do spełnienia, ale zawsze ich realizacja odbywa się kosztem projektu. Klasa `Login` niechybnie stanie się w ich wyniku klasą głęboko osadzoną w konkretnym systemie. Nie da się jej potem łatwo wyciągnąć z projektu i zastosować w kolejnym — trzeba będzie „obrać" jej kod z wszystkich naleciałości charakterystycznych dla systemu, w którym była osadzona. Jeśli nawet okaże się to nieskomplikowane, powrócimy do programowania opartego nie na projekcie, a na umiejętnym na wycinaniu i wklejaniu kodu. W efekcie otrzymamy zaś w dwóch różnych systemach dwie niepodobne już do siebie klasy `Login`, a ulepszenia jednej z nich będziemy próbować niezależnie wprowadzić w drugiej, aż synchronizacja taka stanie się niemożliwa z powodu zbyt wielkiej ich odmienności.

Cóż możemy zrobić, aby zachować klasę `Login`? Możemy wdrożyć wzorzec Observer.

Implementacja

Sedno wzorca Observer (obserwator) polega na rozdzieleniu elementów użytkujących (obserwatorów) od klasy centralnej (podmiotu obserwacji). Obserwatory muszą być informowane o zdarzeniach zachodzących w podmiocie obserwacji. Równocześnie nie chcemy wprowadzać trwałych i sztywnych zależności pomiędzy podmiotem obserwacji a klasami obserwatorów.

Możemy więc umożliwić obserwatorom rejestrowanie się w klasie podmiotu. W tym celu powinniśmy uzupełnić klasę `Login` o trzy nowe metody: rejestracji (`attach()`), rezygnacji (`detach()`) i powiadomienia (`notify()`), przystosowując klasę do wymogów wyróżniających podmioty obserwacji interfejsu (tutaj ma on nazwę `Observable`):

```
interface Observable {
    function attach(Observer $observer);
    function detach(Observer $observer);
    function notify();
}
```

```
// Klasa Login...
    private $observers;
    // ...
    function attach(Observer $observer) {
        $this->observers[] = $observer;
    }

    function detach(Observer $observer) {
        $this->observers = array_diff($this->observers, array($observer));
    }

    function notify() {
        foreach($this->observers as $obs) {
            $obs->update($this);
        }
    }
// ...
```

Mamy więc klasę podmiotu utrzymującą listę obiektów-obserwatorów. Obiekty te są dodawane do listy z zewnątrz poprzez wywołanie metody `attach()`. Rezygnacja z obserwacji i usunięcie z listy następuje w wyniku wywołania metody `detach()`. Z kolei wywołanie metody `notify()` służy jako powiadomienie obiektów obserwatorów o potencjalnie interesujących ich zdarzeniach. Implementacja tej metody sprowadza się do przejrzenia tablicy obiektów obserwatorów i wywołania na rzecz każdego z nich metody `update()`.

Wywołanie metody rozsyłającej powiadomienia następuje we wnętrzu klasy `Login`, w ciele metody `handleLogin()`:

```
function handleLogin($user, $pass, $ip) {
    switch(rand(1, 3)) {
        case 1:
            $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
            $ret = true; break;
        case 2:
            $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
            $ret = false; break;
        case 3:
            $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
            $ret = false; break;
    }
    $this->notify();
    return $ret;
}
```

Zdefiniujmy interfejs klas-obserwatorów:

```
interface Observer {
    function update(Observable $observable);
}
```

Do listy obserwatorów można dodawać (za pośrednictwem metody `attach()` klasy podmiotu obserwacji) dowolne obiekty, które implementują interfejs `Observable`. Utwórzmy kilka takich obiektów:

```
class SecurityMonitor extends Observer {
    function update(Observable $observable) {
        $status = $observable->getStatus();
```

```
        if ($status[0] == Login::LOGIN_WRONG_PASS) {
            // wyślij wiadomość do administratora...
            print __CLASS__."\twysyłam wiadomości e-mail do administratora\n";
        }
    }
}

class GeneralLogger extends Observer {
    function update(Observable $observable) {
        $status = $observable->getStatus();
        // dodaj dane sesji logowania do rejestru...
        print __CLASS__."\tdodaję wpis do rejestru logowania\n";
    }
}

class PartnershipTool extends Observer {
    function update(Observable $observable) {
        $status = $observable->getStatus();
        // sprawdź adres IP...
        // jeśli adres rozpoznany, ustaw plik cookie...
        print __CLASS__."\tustawiam plik cookie dla rozpoznanego adresu IP\n";
    }
}
```

Zauważmy, że obiekty-obserwatory mogą korzystać z przekazanego egzemplarza `Observable` celem pozyskania dodatkowych informacji o zdarzeniu. Klasa podmiotu obserwacji powinna więc przewidywać stosowne metody udostępniające dane interesujące obserwatorów. U nas rolę tego interfejsu pełni metoda `getStatus()`, za pośrednictwem której powiadamiane obiekty-obserwatory otrzymują migawkę bieżącego stanu logowania.

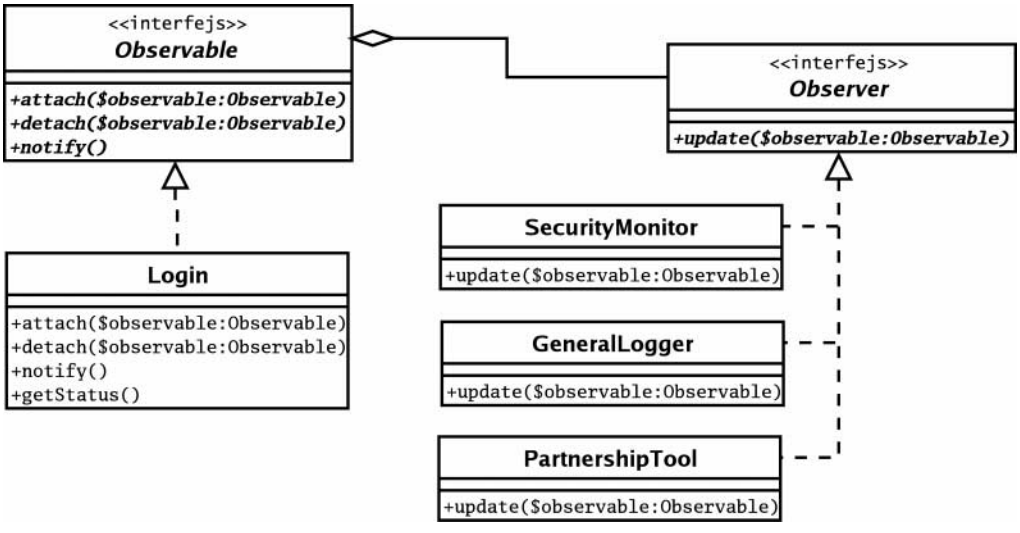
W obiekcie klasy `Login` można rejestrować dowolne obiekty, byle tylko implementowały interfejs `Observer`:

```
$login = new Login();
$login->attach(new SecurityMonitor());
$login->attach(new GeneralLogger());
$login->attach(new PartnershipTool());
```

Otrzymaliśmy więc elastyczne powiązanie pomiędzy klasą-podmiotem obserwacji a klasami-obserwatorami. Diagram klas odpowiedni dla omawianego przykładu prezentowany jest na rysunku 11.6.

Poznaliśmy już najważniejsze elementy kodu implementującego wzorzec Observer. Dla większej przejrzystości możemy zebrać je na wspólnym listingu:

```
interface Observable {
    function attach(Observer $observer);
    function detach(Observer $observer);
    function notify();
}

class Login implements Observable {
    private $observers = array();
    const LOGIN_USER_UNKNOWN = 1;
```



Rysunek 11.6. *Klasy wzorca Observer*

```
const LOGIN_WRONG_PASS = 2;
const LOGIN_ACCESS = 3;
private $status = array();

function attach(Observer $observer) {
    $this->observers[] = $observer;
}

function detach(Observer $observer) {
    $this->observers = array_diff($this->observers, array($observer));
}

function notify() {
    foreach($this->observers as $obs) {
        $obs->update($this);
    }
}

function handleLogin($user, $pass, $ip) {
    switch(rand(1, 3)) {
        case 1:
            $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
            $ret = true; break;
        case 2:
            $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
            $ret = false; break;
        case 3:
            $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
            $ret = false; break;
    }
    $this->notify();
    return $ret;
}
```

```
    private function setStatus($status, $user, $ip) {
        $this->status = array($status, $user, $ip);
    }

    function getStatus() {
        return $this->status;
    }
}

interface Observer {
    function update(Observable $observable);
}

class SecurityMonitor extends Observer {
    function update(Observable $observable) {
        $status = $observable->getStatus();
        if ($status[0] == Login::LOGIN_WRONG_PASS) {
            // wyślij wiadomość do administratora...
            print __CLASS__."\twysyłam wiadomości e-mail do administratora\n";
        }
    }
}

class GeneralLogger extends Observer {
    function update(Observable $observable) {
        $status = $observable->getStatus();
        // dodaj dane sesji logowania do rejestru...
        print __CLASS__."\dodaję wpis do rejestru logowania\n";
    }
}

class PartnershipTool extends Observer {
    function update(Observable $observable) {
        $status = $observable->getStatus();
        // sprawdź adres IP...
        // jeśli adres rozpoznany, ustaw specjalne cookie...
        print __CLASS__."\ustawiam plik cookie dla rozpoznanego adresu IP\n";
    }
}
```

Wzorzec ten doczekał się szeregu wariacji, nie jest też wolny od wad. Po pierwsze, metoda zwracająca stan podmiotu obserwacji (`getStatus()`) nie jest opisywana w interfejsie `Observable`. Daje to typowi `Observable` nadzwyczajną elastyczność, ale równocześnie redukuje bezpieczeństwo typowania. Co się stanie, jeśli któryś z obiektów obserwatorów odwołujących się do metody `getStatus()` zostanie zarejestrowany w obiekcie klasy implementującej interfejs `Observable`, ale nie implementującej metody `getStatus()`? Cóż, wiadomo — dojdzie do błędu krytycznego.

Jak zwykle mamy tu problem zrównoważenia elastyczności i ryzyka. Pominięcie w specyfikacji interfejsu `Observable` daje mu elastyczność, ale naraża użytkowników na ryzyko chybionego wywołania, jeśli obiekt obserwatora zarejestrowany zostanie w złym obiekcie `Observable`. Ryzyko to można wyeliminować przez uzupełnienie interfejsu `Observable` o metodę `getStatus()`, ale kosztem elastyczności. Niektóre z klas podmiotów obserwacji mogą bowiem na przykład udostępniać więcej niż jedną metodę zwracającą status.

Można by rozwiązać problem, przekazując w wywołaniu metody `update()` obiektów-obserwatorów nie egzemplarz podmiotu obserwacji, ale komplet informacji o jego stanie. Osobiście często stosuję tę metodę, jeśli mam na szybko skonstruować działające rozwiązanie. W naszym przykładzie metoda `update()` powinna więc oczekiwać przekazania nie egzemplarza klasy `Login`, ale znacznika statusu logowania, identyfikatora użytkownika i adresu IP maszyny, z której zainicjowano próbę logowania, najlepiej w postaci tablicy. Pozwala to na wyeliminowanie jednej metody z klasy `Login`. Z drugiej strony, jeśli stan podmiotu obserwacji miałby być opisywany zbyt wielką liczbą danych, znacznie bardziej elastycznym rozwiązaniem byłoby jednak przekazywanie w wywołaniu `update()` egzemplarza `Login`.

Można też zablokować typ w ogóle, odmawiając w klasie `Login` współpracy z obiektami klas innych niż wyróżniona (np. `LoginObserver`). W takim układzie należałoby jeszcze pomyśleć o jakichś realizowanych w czasie wykonania testach obiektów przekazywanych w wywołaniu metody `attach()`; alternatywą byłaby zmiana (uszczegółowienie) interfejsu `Observable`.

Mamy tu ponowne zastosowanie kompozycji w czasie wykonania celem skonstruowania elastycznego i rozszerzalnego modelu. Klasa `Login` może zostać teraz łatwo wyodrębniona z kontekstu i przerzucona do zupełnie innego projektu, gdzie może współpracować z zupełnie odmiennym zestawem obserwatorów.

Wzorzec Visitor

Jak widzieliśmy, wiele wzorców, podążając za zasadą wyższości kompozycji nad dziedziczeniem, zakłada konstruowanie struktur w czasie wykonania programu. Znakomitym przykładem takiego wzorca jest powszechnie stosowany wzorzec kompozycji — Composite. Tam, gdzie trzeba operować na zbiorach obiektów, niektóre z operacji mogą odwoływać się do zbiorów jako takich, ale inne mogą wymagać operowania na poszczególnych komponentach zbioru. Takie operacje można wbudować w same komponenty — w końcu to one znajdują się na najlepszej możliwej pozycji do ich realizacji.

Podejście to nie jest jednak pozbawione wad. Nie zawsze na przykład mamy dostateczną ilość informacji o operacjach, które będą wykonywane na strukturze. Jeśli klasy są uzupełniane operacjami od przypadku do przypadku, ich interfejsy mogą się nadmiernie rozrosnąć. Wtedy można uciec się do wzorca Visitor (wizytator).

Problem

Wróćmy do prezentowanego w poprzednim rozdziale przykładu zastosowania wzorca kompozycji. Na potrzeby pewnej gry stworzyliśmy tam armię komponentów o ciekawej cesze zastępowalności komponentu zbiorem komponentów. Operacje były tam wbudowywane w same komponenty — właściwe operacje realizowane były przez obiekty składowe, do których odwoływał się obiekt-kompozyt.

```
class Army extends CompositeUnit {

    function bombardStrength() {
        $ret = 0;
        foreach($this->units() as $unit) {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}

class LaserCannonUnit extends Unit {
    function bombardStrength() {
        return 44;
    }
}
```

Nie są tu problemem te operacje, które stanowią podstawowe zadania klasy kompozytu. Gorzej z operacjami pobocznymi.

Weźmy choćby operację, w ramach której trzeba wykonać zrzut (w postaci tekstowej) informacji o obiektach składowych kompozytu. Operację taką można by włączyć do klasy `Unit`:

```
// klasa Unit
function textDump($num = 0) {
    $ret = "";
    $pad = 4 * $num;
    $ret .= sprintf("%{$pad}s", "");
    $ret .= get_class($this).": ";
    $ret .= "siła rażenia: ".$this->bombardStrength()."\n";
    return $ret;
}
```

Metodę tę można następnie przesłonić w klasie `CompositeUnit`:

```
// klasa CompositeUnit
function textDump($num = 0) {
    $ret = "";
    $pad = 4 * $num;
    $ret .= sprintf("%{$pad}s", "");
    $ret .= get_class($this).": ";
    $ret .= "siła rażenia: ".$this->bombardStrength()."\n";
    foreach($this->units as $unit) {
        $ret .= $unit->textDump($num + 1);
    }
    return $ret;
}
```

Moglibyśmy, idąc tym tropem, utworzyć metody zliczające liczbę jednostek w kompozycie, zapisywania danych o składowych kompozytu w bazie danych czy też obliczania liczby jednostek aprowizacji konsumowanych codziennie przez armię.

Ale czy koniecznie powinniśmy włączać tego rodzaju operacje do interfejsu kompozytu? Po co rozdymać interfejs o funkcje niekoniecznie związane z podstawowym zadaniem klasy? Odpowiedź jest prosta: postanowiliśmy zdefiniować te funkcje tu, bo z tego miejsca łatwo o dostęp do składowych kompozytu.

Choć co prawda łatwość przeglądania zbioru jest jedną z podstawowych cech kompozytu, nie oznacza to, że dosłownie każda operacja wymagająca przejrzenia składowych kompozytu powinna być implementowana w jego klasie i zajmować miejsce w jego interfejsie.

Mamy więc kolejny cel: wykorzystać w dowolnych operacjach łatwość odwołań do komponentów kompozytu bez niepotrzebnego rozdymania jego interfejsu.

Implementacja

Zacznijmy od zdefiniowania w abstrakcyjnej klasie `Unit` metody `accept()`.

```
function accept(ArmyVisitor $visitor) {
    $method = "visit".getClass($this);
    $visitor->$method($this);
}
```

Jak widać, metoda `accept()` oczekuje przekazania w wywołaniu obiektu klasy `ArmyVisitor`. W języku PHP mamy możliwość dynamicznego konstruowania nazw metod wykorzystywaną tu do realizacji dynamicznego wyboru metody do wywołania na rzecz przekazanego obiektu. Dzięki temu nie musimy implementować metody `accept()` osobno dla każdego węzła końcowego naszej hierarchii klas. Trzeba jedynie zdefiniować tę samą metodę w abstrakcyjnej klasie kompozytu.

```
function accept(ArmyVisitor $visitor) {
    $method = "visit".getClass($this);
    $visitor->$method($this);
    foreach($this->units as $thisunit) {
        $thisunit->accept($visitor);
    }
}
```

Metoda ta realizuje to samo zadanie co `Unit::accept()`, z jednym tylko dodatkiem. Na podstawie nazwy bieżącej klasy konstruuje nazwę metody do wywołania i wywołuje ją na rzecz przekazanego obiektu klasy `ArmyVisitor`. Jeśli więc bieżącą klasą będzie `Army`, nastąpi wywołanie klasy `ArmyVisitor::visitArmy()`; jeśli bieżącą klasą będzie `TroopCarrier`, metoda zrealizuje wywołanie `ArmyVisitor::visitTroopCarrier()` i tak dalej. Po powrocie z wywołania rozpocznie się zaś pętla przeglądająca komponenty kompozytu i wywołująca na ich rzecz ich implementację metody `accept()`. A ponieważ `accept()` powtarza tu operacje definiowane w klasach nadrzędnych, możemy w prosty sposób wyeliminować duplikację kodu:

```
function accept(ArmyVisitor $visitor) {
    parent::accept($visitor);
    foreach($this->units as $thisunit) {
        $thisunit->accept($visitor);
    }
}
```

Takie ulepszenie jest bardzo eleganckie, ale choć tutaj dało oszczędność jednego zaledwie wiersza, odbyło się z pewną szkodą dla czytelności i przejrzystości kodu. Tak czy inaczej, metoda `accept()` pozwala nam na dwie rzeczy:

- wywoływanie metody wizytacji właściwej dla bieżącego komponentu;
- przekazywanie obiektu wizytatora do wszystkich komponentów bieżącego kompozytu przez wywołanie ich metod `accept()`.

Trzeba nam jeszcze zdefiniować interfejs klasy `ArmyVisitor`. Pewne pojęcie o jego składnikach daje już metoda `accept()`. Otóż klasa wizytatora powinna definiować wersje metody `accept()` dla wszystkich konkretnych klas w hierarchii. Dzięki temu na kompozytach różnych obiektów będzie można wykonywać różne operacje. W mojej wersji tej klasy zdefiniowałem domyślne wersje metody `visit()` wywoływane automatycznie, jeśli klasa implementująca nie określi własnej wersji tej operacji dla danej klasy hierarchii `Unit`.

```
abstract class ArmyVisitor {
    abstract function visit(Unit $node);

    function visitArcher(Archer $node) {
        $this->visit($node);
    }

    function visitCavalry(Cavalry $node) {
        $this->visit($node);
    }

    function visitLaserCannonUnit(LaserCannonUnit $node) {
        $this->visit($node);
    }

    function visitTroopCarrierUnit(TroopCarrierUnit $node) {
        $this->visit($node);
    }

    function visitArmy(Army $node) {
        $this->visit($node);
    }
}
```

Teraz więc problem sprowadza się do implementacji klas pochodnych `ArmyVisitor`. Oto przykład w postaci kodu generującego zestawienie informacji o obiektach-kompozytach przeniesiony już do klasy `ArmyVisitor`:

```
class TextDumpArmyVisitor extends ArmyVisitor {
    private $text = "";

    function visit(Unit $node) {
        $ret = "";
        $pad = 4 * $node->getDepth();
        $ret .= sprintf("%{$pad}s", "");
        $ret .= get_class($node);
        $ret .= "siła rażenia: ".$node->bombardStrength()."\n";
        $this->text .= ret;
    }
    function getText() {
        return $this->text;
    }
}
```

Spójrzmy, jak stosować taki kod:

```
$main_army = new Army();
$main_army->addUnit(new Archer());
$main_army->addUnit(new LaserCannonUnit());
$main_army->addUnit(new Cavalry());

$texdump = new TextDumpArmyVisitor();
$main_army->accept($textdump);
print $textdump->getText();
```

Powyższy kod powinien dać następujący rezultat:

```
Army: siła rażenia: 50
    Archer: siła rażenia: 4
    LaserCannonUnit: siła rażenia: 44
    Cavalry: siła rażenia: 2
```

Utworzyliśmy obiekt klasy `Army`. Ponieważ jest to kompozyt, włączyliśmy do niego (za pomocą metody `addUnit()`) pewną liczbę utworzonych specjalnie w tym celu obiektów klasy `Unit`. Następnie utworzyliśmy obiekt klasy `TextDumpArmyVisitor` i przekazaliśmy go w wywołaniu metody `Army::accept()`. Metoda ta skonstruowała na podstawie nazwy klasy przekazanego obiektu nazwę metody do wywołania i wywołała metodę `TextDumpArmyVisitor::visitArmy()`. Ponieważ nie przewidzieliśmy specjalnej obsługi wywołania `visit()` dla obiektów klasy `Army`, wywołanie to zostanie zrealizowane domyślną wersją metody `visit()` dla obiektów tego typu. W wywołaniu `visit()` przekazywana jest referencja obiektu klasy `Army`, a w samej metodzie następują wywołania metod tegoż obiektu (w tym nowej metody `getDepth()`, informującej o bieżącym zagłębieniu w drzewie kompozytu), generując za ich pośrednictwem zestawienie opisujące kompozyt. Aby zestawienie było kompletne, metoda `Army::accept()` wywołuje następnie metody `accept()` komponentów, przekazując w wywołaniu ten sam obiekt wizytatora, który sama otrzymała. W ten sposób klasa `ArmyVisitor` „odwiedza" wszystkie obiekty wchodzące w skład drzewa kompozytu.

Uzupełniając istniejący szkielet klas o kilka zaledwie metod, utworzyliśmy mechanizm, za pośrednictwem którego można dołączać do klasy kompozytu nowe funkcje, nie ingerując równocześnie w interfejs kompozytu i unikając powielania kodu przeglądania komponentów.

Załóżmy teraz, że na niektórych polach planszy jednostki muszą uiszczać myto. Poborca odwiedza wtedy poszczególne jednostki armii, przy czym różne jednostki są różnie opodatkowane. Na tym przykładzie dobrze będzie widać zalety specjalizowania metod klasy wizytatora:

```
class TaxCollectionVisitor extends ArmyVisitor {
    private $due = 0;
    private $report = "";

    function visit(Unit $node) {
        $this->levy($node, 1);
    }

    function visitArcher(Archer $node) {
        $this->levy($node, 2);
    }
```

```
    function visitCavalry(Cavalry $node) {
        $this->levy($node, 3);
    }

    function visitTroopCarrierUnit(TroopCarrierUnit $node) {
        $this->levy($node, 5);
    }

    private function levy(Unit $unit, $amount) {
        $this->report .= "Myto należne za ".getClass($unit);
        $this->report .= ": $amount\n";
        $this->due += $amount;
    }

    function getReport() {
        return $this->report;
    }

    function getTax() {
        return $this->due;
    }
}
```

W tym prostym przykładzie nie skorzystamy wprost z obiektu klasy `Unit` przekazywanego do różnych metod wizytacji. Korzystamy jedynie ze specjalizacji (podziału na klasy obiektów odwiedzanych), ustalając dla różnych klas jednostek różne stawki myta.

Oto jak wygląda pobieranie myta z punktu widzenia użytkownika hierarchii:

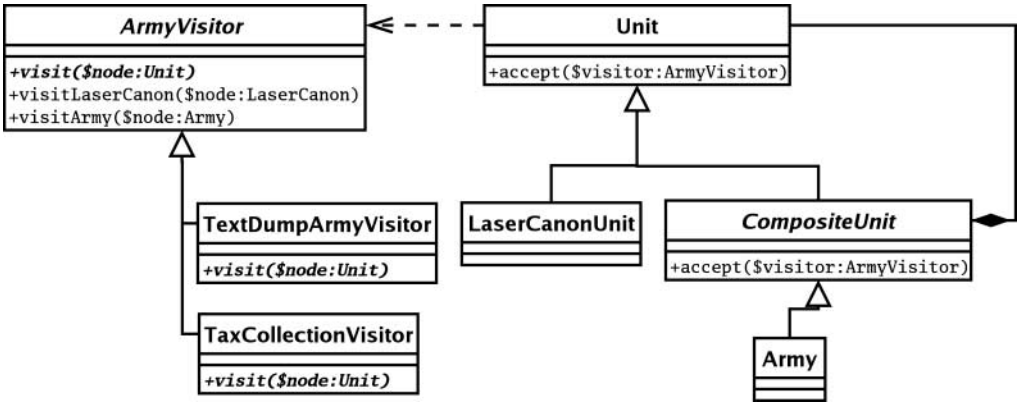
```
$main_army = new Army();
$main_army->addUnit(new Archer());
$main_army->addUnit(new LaserCannonUnit());
$main_army->addUnit(new Cavalry());

$taxcollector = new TaxCollectorVisitor();
$main_army->accept($taxcollector);
print "ŁĄCZNIE: ";
print $taxcollector->getTax()."\n";
```

Tak jak poprzednio, do metody `accept()` wywołanej na rzecz obiektu klasy `Army` przekazany został obiekt wizytatora — tutaj `TaxCollectorVisitor`. Ponownie też obiekt klasy `Army` przekazuje referencję do samego siebie do metody `visitArmy()`, tuż przed oddelegowaniem wywołania do metod `accept()` swoich komponentów. Komponenty są nieświadome operacji przeprowadzonych w ramach wizytacji. Ograniczają się do współpracy z publicznym interfejsem wizytatora, przekazując się po kolei do metody wizytacji właściwej dla swojego typu, w ramach której następuje obliczenie należnego myta.

Poza metodami zdefiniowanymi w klasie bazowej hierarchii wizytatorów klasa `TaxCollectorVisitator` definiuje dwie metody dodatkowe: `getReport()` i `getTax()`. Ich wywołania zwracają dane zebrane podczas wizytacji:

```
Myto należne za Army: 1
Myto należne za Archer: 2
Myto należne za LaserCannonUnit: 1
Myto należne za Cavalry: 3
ŁĄCZNIE: 7
```

Uczestników operacji prezentowanych w przykładzie ilustruje diagram z rysunku 11.7.



Rysunek 11.7. *Wzorzec Visitor*

Wady wzorca Visitor

Wzorzec Visitor to kolejny wzorzec łączący prostotę z efektywnością. Zalety nie mogą jednak przesłonić całkowicie wad wzorca, gdyż i te istnieją.

Po pierwsze, choć najlepiej dopasowany do wzorca kompozycji, wizytator może być stosowany odnośnie dowolnych kolekcji obiektów. Da się więc na przykład zaimplementować wizytację list obiektów, w której każdy z obiektów przechowuje referencję swoich sąsiadów.

Wyodrębniając operacje na kolekcji poza nią samą, sprzeciwiamy się jednak hermetyzacji. Otóż może się okazać, że aby wizytator mógł w jakikolwiek użyteczny sposób przetworzyć obiekty kolekcji, będą one musiały udostępniać na zewnątrz swoje aspekty wewnętrzne (prywatne). Widać to było już choćby w pierwszym przykładzie z tego podrozdziału, kiedy to na potrzeby klasy wizytatora `TextDumpArmyVisitor` trzeba było uzupełnić interfejs `Unit` o dodatkową metodę.

Ponieważ wzorzec ten zakłada również oddzielenie iteracji od operacji wykonywanych na komponentach kolekcji, trzeba zrzec się pewnej części kontroli — nie da się na przykład łatwo utworzyć metody wizytacji `visit()`, która wykonuje pewne operacje tak przed, jak i po odwiedzeniu komponentów zagnieżdżonych w kolekcji. Można by tę niedogodność wyeliminować, przenosząc odpowiedzialność za iterację do samych obiektów wizytatorów. Tyle że wtedy w obiektach tych dojdzie niechybnie do powielenia kodu iteracji.

Osobiście preferuję więc obsługę iteracji w ramach klas wizytowanych, choć nie przeczę, że jej wysunięcie poza te klasy dałoby pewną zasadniczą zaletę: można wtedy zmieniać w poszczególnych wizytatorach sposób wizytacji.

Wzorzec Command

Ostatnimi laty rzadko kiedy udawało mi się zakończyć projekt aplikacji WWW bez wdrażania w nim wzorca Command, czyli wzorca polecenia. Obiekty poleceń, choć pierwotnie stosowane w kontekście projektu graficznego interfejsu użytkownika, sprawdzają się również w projektach aplikacji korporacyjnych, wymuszając separację pomiędzy warstwą kontroli żądań (kodem obsługi i rozprowadzania żądań) a warstwą logiczną aplikacji.

Problem

Wszystkie systemy muszą podejmować decyzje o sposobie reagowania na żądania użytkowników. W PHP proces podejmowania decyzji jest często rozproszony pomiędzy wieloma formularzami-stronami tworzącymi interfejs aplikacji. Wybór funkcji i interfejsu odbywa się tutaj przez wybór jednej ze stron witryny WWW, np. *feedback.php*. Ostatnio programiści PHP optują jednak coraz silniej za podejściem, w którym wyróżnione jest tylko jedno miejsce styku (patrz też następny rozdział). Tak czy inaczej, odbiorca żądania musi je oddelegować do warstwy bliższej samej logice aplikacji. Owa delegacja jest szczególnie istotna, kiedy użytkownik może inicjować żądania za pośrednictwem różnych stron WWW. Bez delegacji projekt zostałby w nieunikniony sposób obciążony powieleniem kodu obsługi żądania.

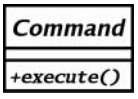
Wyobraźmy sobie więc projekt, w ramach którego powinniśmy realizować pewną liczbę zadań. W szczególności system nasz powinien pozwalać wybranym użytkownikom na zalogowanie się, a innym na przesłanie formularza zwrotnego. Do obsługi tych zadań moglibyśmy wyznaczyć strony *login.php* i *feedback.php*, konkretyzując w nich specjalizowane klasy realizujące żądania. Niestety, perspektywy systemu dla różnych użytkowników rzadko pokrywają się dokładnie z zadaniami, które system ma realizować. Może się więc okazać, że na każdej stronie potrzebujemy zarówno możliwości logowania, jak i przesłania informacji zwrotnej. Jeśli zaś strony mają obsługiwać różne zadania, to może powinniśmy oprzeć hermetyzację właśnie na zadaniach. W ten sposób ułatwimy sobie uzupełnianie funkcjonalności systemu o nowe zadania i stworzymy wyraźną granicę pomiędzy warstwami systemu. W ten sposób dojdziemy do wdrożenia wzorca Command.

Implementacja

Interfejs obiektu polecenia jest tak prosty jak to możliwe — składa się w najprostszym wydaniu z jednej tylko metody — `execute()`.

Na rysunku 11.8 `Command` jest klasą abstrakcyjną. Przy tym poziomie uproszczenia mógłby zostać równie dobrze zdefiniowany jako interfejs. Osobiście skłaniam się do stosowania abstrakcji w miejsce interfejsów dlatego, że niejednokrotnie okazuje się, że w abstrakcyjnej klasie bazowej można upchnąć parę funkcji wspólnych dla wszystkich obiektów pochodnych.

Rysunek 11.8.
Klasa Command



We wzorcu Command mamy jeszcze przynajmniej trzech innych uczestników: klienta, który konkretyzuje obiekt polecenia, inicjatora (ang. *invoker*), który wdraża obiekt w systemie, oraz odbiorcę, do którego polecenie się odnosi.

Odbiorca może zostać wskazany poleceniu przez klienta w ramach konstrukcji obiektu polecenia albo pozyskany z pewnego rodzaju wytwórni. Osobiście preferuję to drugie podejście, bo pozwala na ujednolicenie sposobu konkretyzacji obiektów wszystkich poleceń.

Spróbujmy skonstruować konkretną klasę polecenia dziedziczącą po `Command`:

```
abstract class Command {
    abstract function execute();
}

class LoginCommand extends Command {

    function execute(CommandContext $context) {
        $manager = ReceiverFactory::getAccessManager();
        $user = $context->get('username');
        $pass = $context->get('password');
        $user = $manager->login($user, $pass);
        if (!$user) {
            $this->context->setError($manager->getError());
            return false;
        }
        $context->addParam("user", $user);
        return true;
    }
}
```

Klasa `LoginCommand` jest przewidziana do współpracy z obiektem klasy `AccessManager`. Ten jest na razie wyimaginowaną klasą, której zadaniem jest obsługa szczegółów związanych z procesem rejestrowania użytkowników w systemie. Zauważ, że nasza metoda `Command::execute()` żąda przekazania w wywołaniu obiektu klasy `CommandContext` (w książce *Core J2EE Patterns* występuje ona jako `RequestHelper`). Za jego pośrednictwem obiekt polecenia może odwoływać się do danych związanych z żądaniem i za jego pośrednictwem może przekazywać odpowiedzi do warstwy prezentacji. Zastosowanie w tej roli obiektu jest o tyle wygodne, że pozwala na ujednolicenie interfejsu obiektu polecenia, który przecież w zależności od realizowanego zadania musiałby przyjmować odmienne zestawy argumentów. `CommandContext` jest tu zasadniczo kopertą obiektową ujmującą zmienną typu tablicy asocjacyjnej, a niekiedy uzupełnioną o parę dodatkowych funkcji. Oto prosta implementacja tej klasy:

```
class CommandContext {
    private $params = array();
    private $error = "";

    function __construct() {
        $this->params = $_REQUEST;
    }
```

```
    function addParam($key, $val) {
        $this->params[$key] = $val;
    }

    function get($key) {
        return $this->params[$key];
    }

    function setError($error) {
        $this->error = $error;
    }

    function getError() {
        return $this->error;
    }
}
```

Obiekt polecenia, uzbrojony w obiekt kontekstu, może odwoływać się do danych związanych z żądaniem inicjującym polecenie, tutaj do przekazanych w żądaniu — nazwy konta użytkownika i jego hasła. Obiekt `AccessManager`, służący do realizacji właściwego logowania, pozyskiwany jest za pośrednictwem prostej klasy `ReceiverFactory` implementującej za pośrednictwem swoich statycznych składowych wytwórnię. Jeśli `AccessManager` w ramach operacji, którą realizuje, zgłosi błąd, metoda `execute()` obiektu polecenia wstawi (na potrzeby warstwy prezentacji) do kontekstu stosowny komunikat i zwróci po prostu wartość `false`. Jeśli zaś wszystko pójdzie dobrze, obiekt klasy `LoginCommand` zwróci wartość `true`. Zauważmy, że obiekty hierarchii `Command` same w sobie nie implementują żadnej logiki związanej z wykonaniem właściwego zadania. Sprawdzają jedynie parametry wejściowe, kontrolują sytuacje wyjątkowe i buforują dane wyjściowe, a w pozostałych zadaniach zdają się całkowicie na inne obiekty.

Brakuje nam już tylko klienta (klasy, która generowałaby obiekty poleceń) oraz inicjatora (ang. *invoker*). Najprostszym sposobem wyboru polecenia do konkretyzacji w aplikacji WWW jest uwzględnienie w żądaniu stosownego parametru. Oto uproszczona implementacja klienta:

```
class CommandNotFoundException extends Exception {}

class CommandFactory {
    private static $dir = 'commands';

    function getCommand($action='default') {
        $class = ucfirst(strtolower($action))."Command";
        $file = self::$dir."/$class.php";
        if (!file_exists($file)) {
            throw new CommandNotFoundException("nie można znaleźć pliku '$file'");
        }
        require_once($file);
        if (!class_exists($class)) {
            throw new CommandNotFoundException("nie można znaleźć klasy '$class'");
        }
        $cmd = new $class();
        return $cmd;
    }
}
```

Klasa `CommandFactory` przeszukuje katalog o nazwie `commands`, szukając w nim pliku konkretnej klasy. Nazwa pliku konstruowana jest na bazie wyodrębnianego z obiektu `CommandContext` parametru `$action`, który z kolei powinien zostać przekazany wraz z żądaniem. Jeśli plik klasy uda się odnaleźć, a w pliku zdefiniowana jest szukana klasa, wtedy obiekt tej klasy jest zwracany wywołującemu. Moglibyśmy ten fragment kodu uzupełnić odpowiednimi operacjami kontroli błędów, upewniając się choćby, czy znaleziona klasa należy aby do hierarchii `Command`, czy przekazany w żądaniu ciąg określający klasę nie odnosi się do nazwy katalogu (a nie pliku) albo czy konstruktor klasy faktycznie nie wymaga przekazania żadnych argumentów — dla celów przykładu tak okrojona implementacja jest jednak zupełnie wystarczająca. Siłą tego rozwiązania jest to, że system można uzupełniać o nowe klasy poleceń w dowolnym momencie, uzupełniając po prostu katalog `commands` — po umieszczeniu w nim nowej klasy system od razu może obsługiwać nowe polecenie.

Kod inicjatora jest teraz równie prosty:

```
class Controller {
    private $context;
    function __construct() {
        $this->context = new CommandContext();
    }

    function getContext() {
        return $this->context;
    }

    function process() {
        $cmd = CommandFactory::getCommand($this->context->get('action'));
        if (!$cmd->execute($this->context)) {
            // obsługa błędu...
        } else {
            print "polecenie wykonane";
            // sukces
        }
    }
}

$controller = new Controller();
// imitacja obsługi żądania użytkownika
$context = $controller->getContext();
$context->addParam('action', 'login');
$context->addParam('username', 'bob');
$context->addParam('pass', 'hop125');
$controller->process();
```

Przed wywołaniem `Controller::process()` tworzymy fikcyjne żądanie WWW, ustawiając odpowiednio parametry obiektu kontekstu konkretyzowanego w konstruktorze kontrolera. Metoda `process()` deleguje konkretyzację obiektu polecenia do wytwórni `CommandFactory`, a następnie na rzecz tak otrzymanego obiektu wywołuje metodę `execute()`. Zauważmy, że kontroler nie wie wiele o cechach wewnętrznych polecenia — właśnie ta niezależność od szczegółów wykonania polecenia umożliwia nam dodawanie do systemu kolejnych klas poleceń przy minimalnym wpływie na zastany szkielet aplikacji.

Utwórzmy jeszcze jedną klasę hierarchii `Command`:

```
class FeedbackCommand extends Command {

    function execute(CommandContext $context) {
        $msgSystem = ReceiverFactory::getMessageSystem();
        $email = $context->get('email');
        $msg = $context->get('msg');
        $topic = $context->get('topic');
        $result = $msgSystem->despatch($email, $msg, $topic);
        if (!$user) {
            $this->context->setError($msgSystem->getError());
            return false;
        }
        $context->addParam("user", $user);
        return true;
    }
}
```

Do wzorca Command wrócimy jeszcze w rozdziale 12., przy okazji omawiania pełniejszej implementacji klasy wytwórni poleceń. Zaprezentowany tu szkielet wykonywania poleceń jest jedynie uproszczoną wersją innego wzorca, z którym się niebawem zetkniemy — wzorca Front Controller.

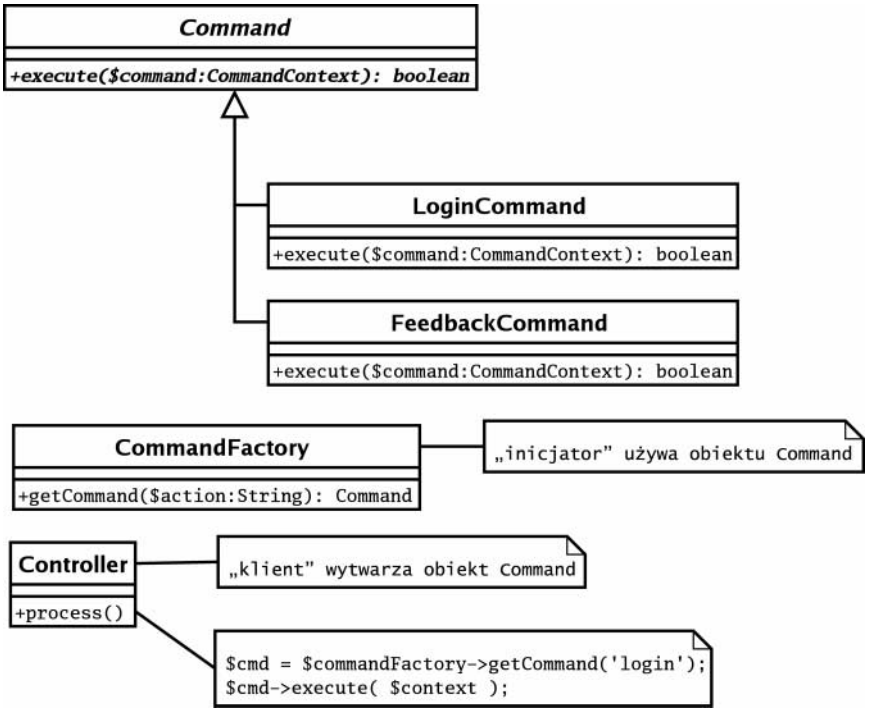
Jeśli prezentowana wyżej klasa będzie definiowana w pliku *FeedbackCommand.php*, a plik umieszczony w katalogu *commands*, będzie można korzystać z pośrednictwa jej obiektów w obsłudze żądania obsługi formularza zwrotnego; ewentualne zmiany w sposobie tej obsługi nie będą wymagać żadnych czynności dostosowawczych w kodzie kontrolera ani w kodzie klas wytwórni poleceń.

Uczestników wzorca Command prezentuje rysunek 11.9.

Podsumowanie

Niniejszym rozdziałem zakończyliśmy przegląd wzorców z katalogu Bandy Czworga. Udało się przy tym zaprojektować miniaturowy język programowania i skonstruować na bazie wzorca Interpreter mechanizm jego interpretacji. We wzorcu Strategy rozpoznaliśmy kolejny sposób korzystania z kompozycji na rzecz zwiększania elastyczności i redukowania potrzeby wyprowadzania dublujących się po części pochodnych. Wzorzec Observer rozwiązał problem powiadamiania oddzielonych i różnych od siebie komponentów o zdarzeniach zachodzących w systemie. Wróciliśmy też na chwilę do przykładu z omówienia wzorca Composite, pokazując zastosowanie wzorca Visitor do wykonywania rozmaitych operacji na składnikach obiektu-kompozytu. Na koniec mogliśmy docenić ułatwienie konstruowania rozszerzalnego systemu warstwowego w postaci wzorca Command.

W następnym rozdziale porzucimy już katalog Bandy Czworga, zwracając się ku wzorcom powstałym specjalnie z myślą o programowaniu aplikacji korporacyjnych.



Rysunek 11.9. *Uczestnicy wzorca Command*