Adam Lejman, Altkom Akademia, Warszawa 2001

Wzorce w tworzeniu aplikacji

Tworzenie systemów informatycznych wyróżniających się poprawną architekturą wewnętrzną wymaga od wykonawców dużego doświadczenia, które można nabyć uczestnicząc w projektach. Budowa każdego kolejnego rozwiązania opiera się na dobrych pomysłach i praktykach wykorzystanych poprzednio. Unika się konstrukcji, które okazywały się w poprzednich projektach niezręczne, mało wydajne lub trudne w rozwijaniu.

Cóż jednak mają zrobić ci, którzy tego doświadczenia mają mało i nie chcą popełniać podstawowych błędów? Od jakiegoś czasu w dziedzinie budowania systemów informatycznych można zauważyć swoistą wręcz modę na stosowanie i publikowanie wzorców (patterns), dobrych praktyk i konstrukcji obiektowych. Również ostatnio pojawiają się publikację najczęściej popełnianych błędów, czyli antywzorce (antipatterns). Dzięki temu znacznie łatwiej jest rozpoznać i zaradzić niefortunnych konstrukcjom.

Popularność stosowania wzorców zaczęła się od kultowej pozycji Gangu Czterech (Gang of Four) [1]. Autorzy wyłuskali i skatalogowali najczęściej pojawiające się konstrukcje obiektowe w różnego rodzaju aplikacjach, uogólnili je i przedstawili jako gotowe pomysły do wykorzystania w realizacji systemów obiektowych. Od tego czasu o wzorcach zaczęto mówić coraz powszechniej, zaczęto tworzyć strony internetowe przedstawiające kolejne odkrywane wzorce [2]. Projektanci zauważyli, że korzystanie z wzorców rzeczywiście znacznie ułatwia budowanie nowych rozwiązań, polepsza wymianę pomysłów w zespole projektowym, podczas której wystarczy posługiwać się nazwami wzorców zamiast przekazywać swoje myśli w sposób graficzny.

Istota modelowania podczas tworzenia aplikacji

Nie ulega wątpliwości, że budowanie aplikacji powinno być poprzedzone etapem analizy i projektowania obiektowego. O ile analizę można porównać do śledztwa, w którym zbierane są materiały dowodowe, czyli jak aplikacja ma się zachowywać i wyglądać, to kolejny etap – projektowanie, jest budowaniem rozwiązania dostosowanego do wyników analizy. Po zakończeniu projektowania, z gotowym modelem obiektowym, pozostaje wypełnienie wszystkiego kodem i przetestowanie.

Oczywiście, w rzeczywistości kolejne etapy budowania aplikacji nie są tak kategorycznie odseparowane. Nie dość, że zachodzą na siebie, to normalną sprawą są nawroty do poprzednich kroków, głównie ze względu na zmieniające się wymagania użytkownika lub przeoczenia na wcześniejszym etapie.

Utrzymywanie przejrzystego i aktualnego modelu aplikacji jest, zatem sprawą niezwykle ułatwiającą budowanie nietrywialnych aplikacji. Stosowanie wzorców zwiększa szansę uniknięcia błędów projektowych, które wcześniej czy później zostałyby dostrzeżone. Często jednak zbyt późno, kiedy trud ich usunięcia jest zbyt duży.

Wzorce w analizie

Wzorce stosowane są już na etapie analizy. Tutaj zazwyczaj poza zebraniem i wyartykułowaniem w sposób formalny wymagań użytkownika powstaje też model obiektowy problemu obrazujący logikę systemu.

Model analityczny abstrahuje od języka programowania i wszelakich aspektów technicznych. Taki model przedstawia relacje pomiędzy obiektami, spotykanymi w modelowanej dziedzinie. Często mówi się o takich obiektach – obiekty biznesowe.

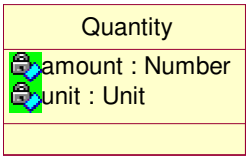
Wzorce wykorzystywane na tym etapie są zazwyczaj przypisane do konkretnych sytuacji biznesowych i ich zakres zastosowań jest może nieco mniejszy niż przy wzorcach z etapu projektowania. Ale i tutaj można doszukać się kilku znanych, uniwersalnych konstrukcji.

Najbardziej klasycznym przykładem wzorca analitycznego jest, być może zbyt szumnie nazwany jako wzorzec, bo dotyczy tylko jednej klasy, wzorzec modelowania wartości z jednostkami [3].

Krótkowzrocznym podejściem jest przedstawianie takich wartości jako danych liczbowych, do których panuje umowa, co do jednostek. I tak na przykład budując dane o sportowcach, przechowuje się ich wagę w kilogramach. Jest to tak oczywiste, że nie zadajemy sobie trudu zapisania jednostek w modelu analitycznym.

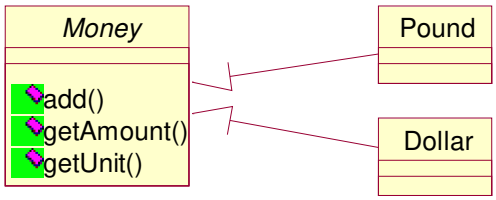
W przyszłości system może zostać wystawiony w internecie i użytkownik systemu zza oceanu w sposób naturalny posługuje się wartościami wagi wyrażonymi w funtach. I powstaje problem. Należy zacząć uwzględniać nowe jednostki, dokonywać przeliczeń, co czasem może się wiązać z utratą precyzji. O ile prościej byłoby gdyby nasz model analityczny był uniezależniony od jednostek.

Z pomocą przychodzi wzorzec zwany **Quantity**. W tym wzorcu wszelakie dane liczbowe przechowywane są jako obiekt, który przechowuje osobno wielkość i jednostkę. Taki obiekt może mieć zaszyte algorytmy konwersji jednej jednostki na drugą (o ile taka konwersja ma sens) i jej algorytm jest znany programiście.



Wzorzec Quantity możemy zastosować do liczenia pieniędzy w różnych walutach, przetwarzania wartości fizycznych, w każdej sytuacji gdzie z wartością skojarzona jest jednostka, w jakiej ta wartość została wyrażona.

Jako przykład można zbudować w oparciu o omawiany wzorzec klasy reprezentujące pieniądze, klasę bazową Money i dwie przykładowe klasy specjalizowane Dollar i Pound.



Metoda *add()* pozwala dokonywać operacji dodawania obiektów typu Money. W czasie implementacji konkretnych klas należy zdefiniować jak taką konwersję należy przeprowadzić. Czasem okazuję się, że zamiast natychmiastowego przeliczania wartości lepiej jest budować historię tego, co ma być wykonane i dopiero w momencie skorzystania z aktualnej wartości, jaką obiekt przechowuje, operacja dodawania wartości powinna być przeprowadzona.

Na przykład przy opracowywaniu cennika produktów importowanych na rynek krajowy można przyjąć, że cena pudełka zapałek składa się z sumy cen samych zapałek wyrażonej w dolarach 0.05 oraz ceny samego pudełka wyrażonej w funtach 0.02. Cena w złotówkach jest oczywiście zależna od kursu obu walut i powinna w sobie kryć zapis w postaci sumy cen jednostkowych, która jest zamieniana na złotówki wówczas, gdy istnieje potrzeba poznania dzisiejszej ceny pudełka zapałek.

Można zarzucić, że wzorzec jest trywialny. To prawda, ale uświadomienie sobie konsekwencji jego niezastosowania jest chyba bardziej istotne od samego wzorca. Wzorce to dobre praktyki zaczerpnięte z wielu projektów tak, więc dla doświadczonych projektantów są rzeczą oczywistą.

Wzorce w projektowaniu

Wzorce związane z etapem projektowania można podzielić na wzorce związane z tworzeniem obiektów, wzorce związane ze strukturą zależności pomiędzy obiektami i wzorce modelujące zachowanie się obiektów. W każdej kategorii można odnaleźć od kilku do kilkunastu najczęściej powtarzających się mikro-modeli.

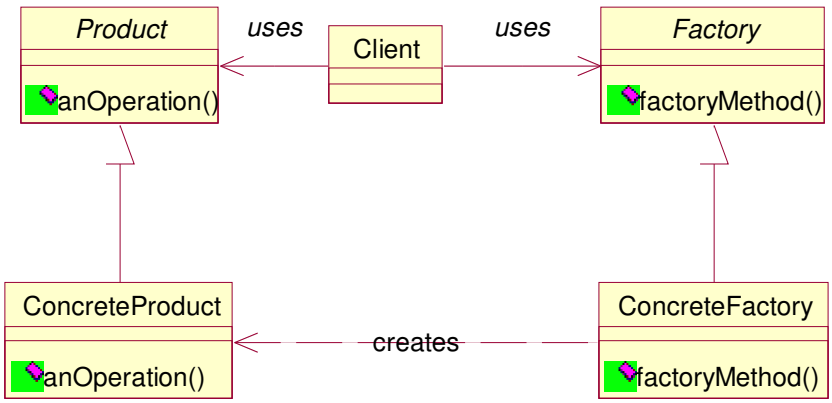
Wzorce tworzenia obiektów

Wzorce związane z tworzeniem obiektów mają na celu abstrahowanie samego procesu tworzenia. Nie zawsze miejsce, w którym dokonuje się aktu kreacji obiektu, jest miejscem, w którym dysponuje się pełną informacją, jak taki obiekt stworzyć, czy w ogóle go tworzyć, czy też może skorzystać z już istniejącej instancji. Poza tym, sam akt kreacji pewnych obiektów może być kosztowny i może warto odłożyć go na później do czasu jego pierwszego użycia.

Najbardziej popularne wzorce tej kategorii to Abstract Factory, Builder, Factory Method i Singleton. Każdy znajduje zastosowanie w konkretnych sytuacjach. Rozważmy bliżej wzorzec zwany **Factory Method**. Jego zastosowanie można znaleźć na przykład w pakiecie *java.net* lub *java.rmi*.

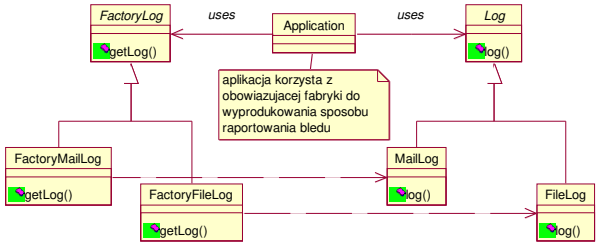
Istota tego wzorca polega na zrzuceniu aktu kreacji obiektu-produktu innemu obiektowi-producentowi. Obiekt korzystający ze wzorca (klient) posługuje się najczęściej interfejsem lub klasą abstrakcyjną opisującą produkt i jego producenta. W trakcie działania aplikacji klient oczywiście otrzymuje konkretną implementację producenta, która to implementacja *wie*, jaką

konkretną implementację produktu wykonać. Obiekt-klient korzystając z metod ogólnego typu produktu korzysta w sposób nieświadomy z konkretnej implementacji.



W następującym przykładzie opisany wzorzec znajduje swoje zastosowanie. *Aplikacja w razie wystąpienia określonej sytuacji krytycznej ma raportować zaistniałe zdarzenie administratorowi systemu poprzez aktualnie obowiązujący mechanizm powiadamiania. Aplikacja ma być niezależna od mechanizmu powiadamiania. Wprowadzenie nowych sposobów powiadamiania nie powinno wywoływać zmian w samej aplikacji.*

Produktem w tym przypadku jest mechanizm powiadamiania, zaś producent przekazywany do obiektu aplikacji będzie znał obowiązujący sposób raportowania. Przykład osadzony na tym wzorcu mógłby wyglądać następująco.



Aplikacja korzysta z ogólnej klasy (w tym przypadku dodatkowo abstrakcyjnej) *FactoryLog*, poprzez którą pobiera obiekt, któremu będzie raportować zdarzenie. Aplikację absolutnie nie interesuje czy dostanie nową instancję obiektu, poprzez który będzie raportować, czy z wykorzystaniem innego wzorca (Singleton) otrzyma referencję do już istniejącego obiektu. Ważny jest fakt, że mechanizm pobierania obiektu raportującego i korzystanie z niego jest stały.

```
//...factoryLog jest referencją typu FactoryLog
try {
      // sekcja krytyczna
```

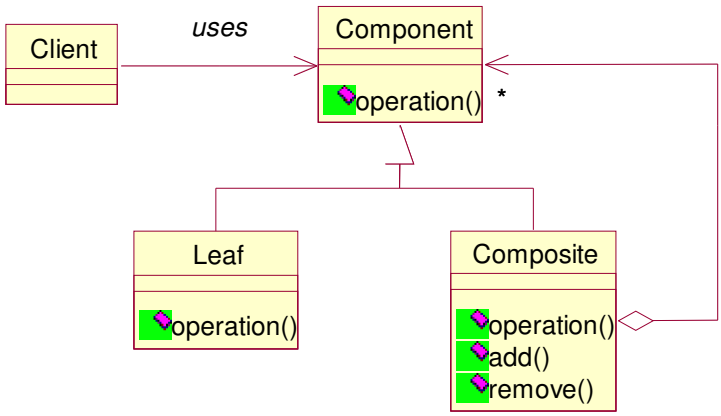
```
      // ...
} catch (Exception e) {
      Log log = factoryLog.getLog();
      log.log(e.getMessage);
}
```

Dołożenie nowego sposobu raportowania, np. raportowanie do bazy danych, wymaga tylko dopisania w hierarchii obiektów nowej klasy produktu i nowej klasy producenta. Kod samej aplikacji jest już przygotowany na raportowanie w nowy sposób!

Wzorce struktury obiektów

Wzorce struktury pokazują jak organizować kooperacje pomiędzy obiektami, aby taki zestaw obiektów tworzył rodzaj mikro-architektury, wnoszącej coś ponad sam zestaw kilku obiektów. Trudno w tym przypadku jasno określić cel wzorców struktury, jak to miało miejsce w przypadku wzorców tworzenia obiektów. Tutaj raczej zostały zgromadzone najczęściej pojawiające się w projektach konstrukcje obiektowe. Do najbardziej znanych wzorców tej kategorii należą Adapter, Composite, Proxy i Facade.

Jednym z najczęściej ze spotykanych wzorców, z jakim mają do czynienia programiści graficznego interfejsu użytkownika jest wzorzec **Composite**. Znane z biblioteki AWT relacja pomiędzy klasą Component i Container jest klasycznym przeniesieniem tegoż wzorca.



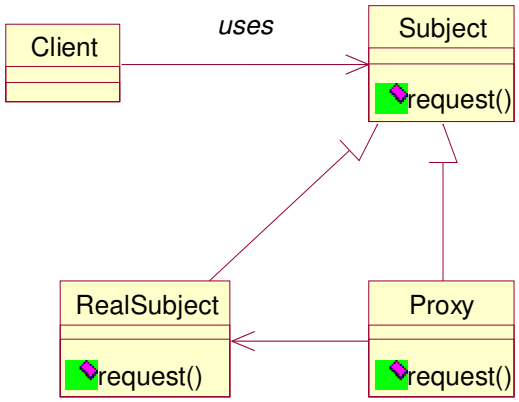
Celem powstania takiego układu obiektów jest jednakowe traktowanie pojedynczego obiektu i grupy obiektów określonego typu. Dla obiektu-klienta bez znaczenia jest czy ma do czynienia z jednym obiektem czy z całą grupą obiektów określonego typu, ponieważ grupa przejawia dokładnie takie samo zachowanie jak pojedynczy obiekt.

Często spotykane zastosowanie tego wzorca dotyczy organizowania pracy obiektów opisujących składniki graficzne, które mogą być skupiane w większe obiekty, a te z kolei w jeszcze większe. Ważną ich cechą jest to, że tak samo obsługuje się najdrobniejszy element grafiki jak i całe gotowe struktury graficzne.

Innym przykładem zastosowania jest tworzenie grup użytkowników. Obiekt opisujący użytkownika jest tutaj obiektem typu Component, a grupa stanowi Composite. Jeśli opis użytkownika zawiera na przykład uprawnienia do korzystania z zasobów lub określonych funkcji systemu, to w łatwy sposób możemy te uprawnienia przenosić na grupy, która z racji

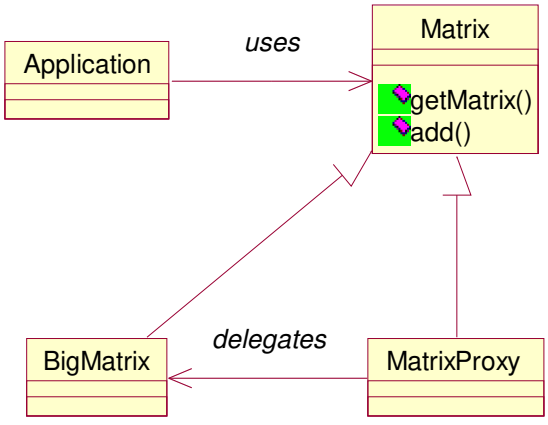
relacji dziedziczenia może być traktowana jak obiekt użytkownika, a same grupy mogą być skupiane w grupy grup itd.

Dosyć powszechną konstrukcją wzorców struktury jest wzorzec zwany **Proxy**. Jest stosowany wszędzie tam, gdzie koszt utworzenia rzeczywistego obiektu jest zbyt duży. Wówczas w akcie kreacji obiektu korzysta się z obiektu pośrednika, a dopiero w momencie pierwszego skorzystania z funkcji obiektu, pośrednik tworzy rzeczywisty obiekt i deleguje wykonanie operacji na jego rzecz.



Przykład tego wzorca występuje przy korzystaniu z obiektów Image z pakietu *java.awt*. Pobranie obrazka z sieci metodą *getImage()* klasy java.applet.Applet, które wiązałoby się z załadowaniem grafiki przez sieć jest operacją kosztowną. Wobec czego metoda zwraca nam referencję na pośrednika, jakim jest obiekt typu Image i dopiero pierwsze jego użycie (np. narysowanie na ekranie) pociąga za sobą stworzenie rzeczywistego obiektu ImagePeer, do którego delegowane są zawołania obiektu Image. W ten sposób zbudowanych jest większość komponentów AWT.

Spróbujmy zastosować wzorzec pośrednika w obliczeniach wykorzystującym duże macierze. Obiekty korzystające z macierzy nie muszą być świadome, w jaki sposób jest organizowana implementacja macierzy w pamięci. Czasem, w przypadku bardzo dużych macierzy ta implementacja jest dosyć kosztowna i korzystnie jest dostęp do nich oprzeć o Proxy. Mikro-architektura mogłaby wyglądać następująco:




```
//Application pobiera dużą macierz
Matrix matrix = Matrix.getMatrix(1000, 1000);
```

W metodzie *getMatrix()*angażowana jest imitacja MatrixProxy, która udaje rzeczywistą macierz wszędzie tam, gdzie nie są potrzebne odwołania do macierzy BigMatrix. W szczególności imitacja może być używana przy budowaniu struktur danych lub jako argument przekazywany między metodami. Natomiast w momencie pierwszego skorzystania z operacji, których nie da wykonać się na imitacji, pośrednik tworzy rzeczywisty obiekt i przekazuje dane zawołanie bezpośrednio do niego.

```
// metoda add klasy MatrixProxy może wyglądać następująco
public void add(int m, int m, double value) {
      // sprawdzamy czy już istnieje właściwa macierz
      if (realMatrix == null) {
            realMatrix = new BigMatrix (sizeX, sizeY);
      }
      // delegujemy operacje
      realMatrix.add(m, n, value);
}
```

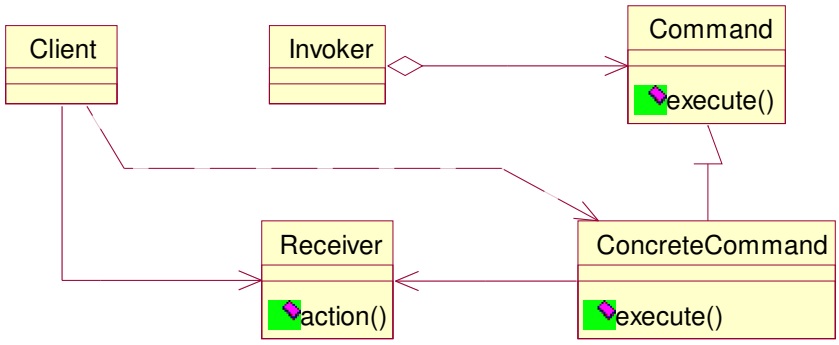
Wzorce zachowania

Wzorce zachowania są niejako dalszym rozwinięciem wzorców struktury, w których koncentrujemy się na zachowaniu się obiektów w mikro-architekturze. Służą do sprawnego organizowania sterowania działaniem aplikacji, w których wykraczamy ponad kolejność wykonywanych kroków programu na rzecz łączenia obiektów w sposób taki, aby same przekazywały kontrolę programu między sobą.

Są to wzorce, które z znacznej mierze decydują o łatwości rozszerzania (rozbudowie) danej aplikacji. Sprawne ich stosowanie pozwala rozwiązać wiele wydawałoby się złożonych problemów kontroli działania aplikacji. Najbardziej znanymi wzorcami tej grupy są: Command, State, Visitor, Observer, Iterator czy Chain of Responsibility.

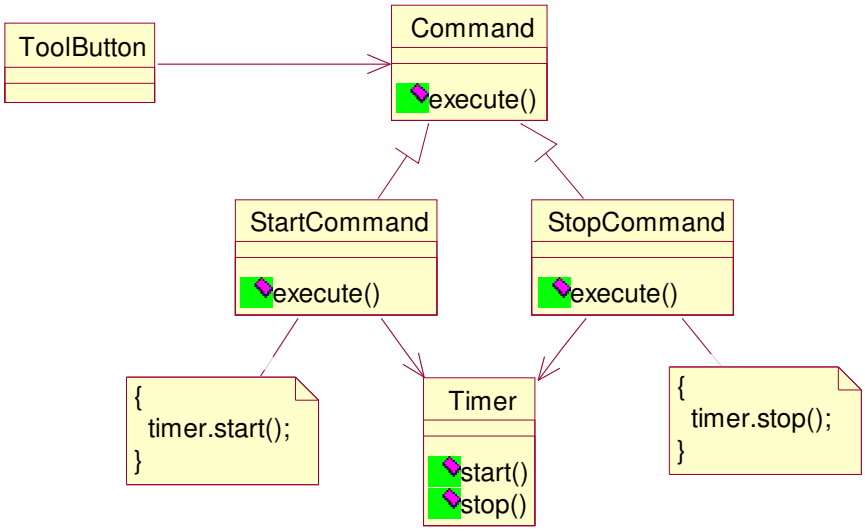
Wzorcem, który warto stosować w aplikacjach korzystających z interfejsu graficznego jest wzorzec organizujący proces wydawania poleceń - **Command**. Częstym problemem jest przekazanie polecenia wydanego poprzez menu lub naciśnięcie przycisku do odpowiedniego miejsca w kodzie, ponieważ wiąże się to ze zbyt dużym obarczeniem informacją klas realizujących obsługę zdarzeń. Każda klasa powinna mieć jasno określoną odpowiedzialność realizowaną w systemie i obiekt odbierający zdarzenia ma *wiedzieć*, kogo ma o tym powiadomić, a nie koniecznie już dostarczać parametry do tegoż powiadamiania.

Wzorzec Command rozwiązuje problem rozdzielenia odpowiedzialności. Uniezależnia obiekty odbierające zdarzenia od samych zadań, które wiążą się z danym zdarzeniem. W szczególności z danym zdarzeniem mogą być związane zestawy komend do wykonania. Tutaj jak najbardziej można zastosować wzorzec Composite w miejscu samej klasy Command.



W omawianym wzorcu Invoker ma przypisany do siebie obiekt typu Command, który ma zdefiniowaną metodę *execute()*, wołaną w momencie wystąpienia zdarzenia. Aplikacja definiuje, jaka konkretna komenda jest dowiązana do poszczególnych obiektów typu Invoker. Konkretna implementacja typu Command zawiera w sobie pełną logikę wykonania komendy. Co więcej, poszczególne komendy mogą zawierać logikę odwołania komendy i stąd już tylko mały krok do wprowadzenia możliwości cofania komend w budowanej aplikacji.

Rozważmy przykład budowania aplikacji posiadającej pasek przycisków narzędziowych. W momencie ustalania zawartości paska, określane są dowiązania przycisków z ich funkcjami. Same przyciski nie są świadome, jakie czynności wykonują. Cała logika wykonywanych poleceń zaszyta jest w implementacji konkretnych klas typu Command.

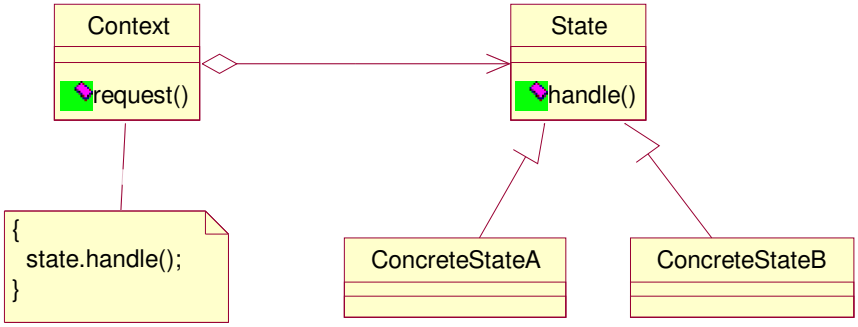


W przypadku korzystania z biblioteki AWT, ToolButton zapewne jest obiektem pochodzącym z Command, w którego konstruktorze przekazywana jest rzeczywista instancja komendy, np.:

```
ToolButton tb = new ToolButton ("Stop", new StopCommand(timer));
```

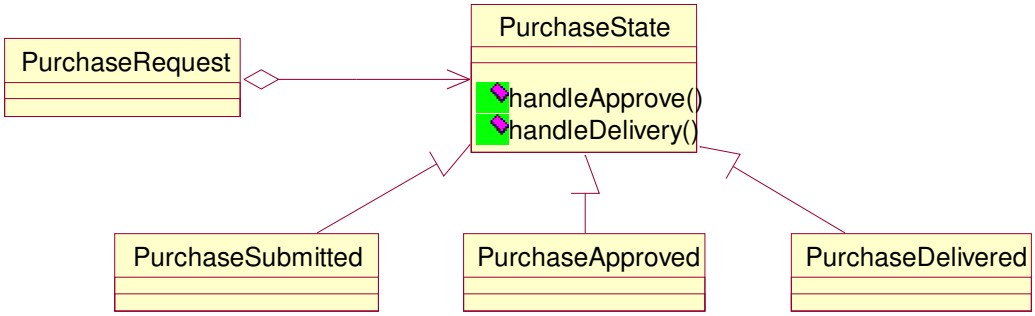
Sam obiekt typu ToolButton wywołuje tylko skojarzoną z nim komendę, natomiast do komendy należy wykonanie powierzonego zadania. Jeśli jedną z implementacji typu Command będzie klasa zaczerpnięta ze wzorca Composite, wówczas automatycznie zyskuje możliwość organizowania makrokomend poprzez fakt, że zawołanie *execute()* na rzecz takiej klasy grupującej poszczególne komendy wiąże łańcuchowe wołanie komend obiektów w niej zgrupowanych.

Innym często przydatnym wzorcem zachowania obiektów jest wzorzec **State**. Przynosi nam korzyści w przypadku obiektów, które mogą występować w różnych stanach, od których zależy implementacja poszczególnych operacji. Tak, więc zamiast złożonych konstrukcji typu *if* lub *switch* można wykorzystać osobny obiekt odpowiedzialny za zachowanie się obiektu w określonym stanie.



Stosując wzorzec State obiekt typu Contex przekazuje zawołanie *request()* do obecnie związanego z min stanu. To właśnie konkretna implementacja klasy State wie, co ma w danej chwili zrobić.

Wzorzec daje najlepsze efekty w przypadku obiektów realizujących inne zadania w różnych stanach swego życia. Dobrym przykładem jest tutaj przetwarzanie dokumentów związanych z dokonywaniem zakupów. Obiekt reprezentujący zamówienie na nabycie jakiegoś dobra będzie znajdował się w stanach typu: zamówienie przedłożone, zaakceptowane, zrealizowane. W każdym z tych stanów z obiektem mogą być związane inne działania.



Wykonywanie poszczególnych zadań obiektu powierzymy tutaj obiektom odpowiedzialnym za stan. Kod obiektu PurchaseRequest mógłby wyglądać następująco.

```
public class PurchaseRequest {
      PurchaseState state;
      public PurchaseRequest(/* argumenty */) {
            // dokonanie niezbędnej inicjalizacji
            state = new PurchaseSubmited(this);
      }

      // akceptacja zakupu
      public void approve() {
            state = state.handleApprove();
      }

      //przekazanie do doręczenia
      public void delivery() {
```

```
            state = state.handleDelivery();
        }
        // inne metody
        // ...
}
```

Za wykonywanie akcji w poszczególnych stanach odpowiedzialne są specjalizowane obiekty, które również kontrolują czy dana akcja może być wykonana z bieżącego stanu i jaki jest stan następujący po wykonaniu akcji.

Co dalej?

Oczywiście ten krótki artykuł absolutnie nie wyczerpuje tematu wzorców w projektowaniu. Nie dość, że zostało przedstawionych zaledwie kilka wzorców z wybranych grup, to również same wzorce pokazane zostały w ich klasycznych zastosowaniach. Bardzo często wzorzec jest jedynie inspiracją do własnego rozwiązania i nigdzie nie jest powiedziane, że wzorca nie można modyfikować. Dlatego też wzorce określonego typu często można spotkać pod różnymi nazwami.

Na wzorce należy spojrzeć też szerzej, jako zbiór doświadczeń nie tylko dotyczący samego projektowania aplikacji, ale też kodowania, testowania czy nawet wdrażania systemów. Możemy odnaleźć wzorce dotyczące budowania GUI, tworzenia komponentów związanych z logiką aplikacji czy wręcz tworzenia serwerów sieciowych. Mam nadzieję, że tematyka jest interesująca i będzie inspiracją w studiowaniu źródeł katalogujących wzorce.

Literatura

1. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley, 1994
2. http://hillside.net/patterns/
3. M. Fowler, *Analysis Patterns*, Addison-Wesley, 1997