Michał Stochmiałek <misto@e-informatyka.pl>

Wprowadzenie do programowania aspektowego

Politechnika Wrocławska, Wrocław, maj 2004

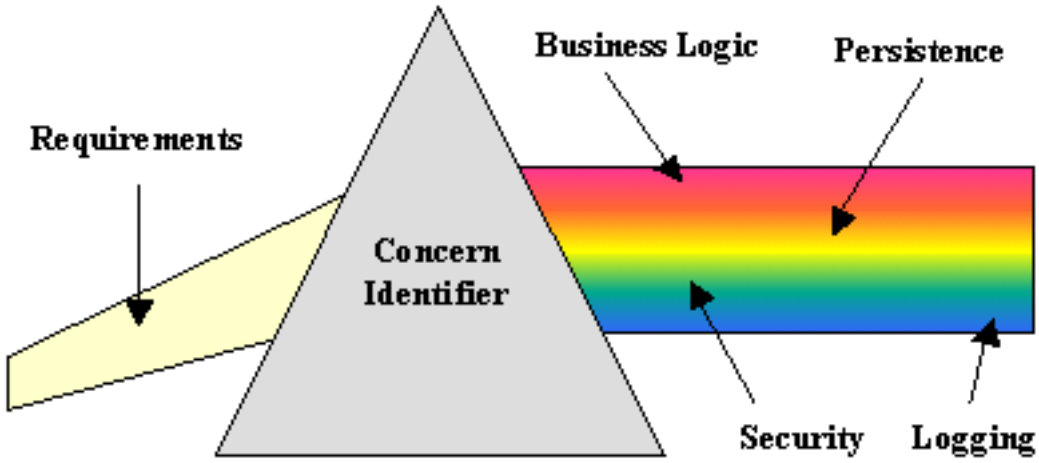
Streszczenie

Artykuł wprowadza czytelnika w teoretyczne podstawy programowania aspektowego (ang. *Aspect Oriented Programming - AOP*). Ten nowy paradygmat programistyczny za kilka lat będzie kursem obowiązkowym każdego studenta informatyki. Do tego czasu jednak warto zapoznać się z tym innowacyjnym podejściem do programowania, jak również, w ogólności, do projektowania systemów informatycznych.

1. Problem przecinających się zagadnień

Prawdopodobnie każdy programista spotkał się z poniższym paradoksem. Rozwijana przez niego aplikacja składa się nie tylko z implementacji jej głównego celu, ale również innych pobocznych zagadnień (ang. *concern*) (można powiedzieć: ortogonalnych względem głównego celu). Na przykład źródła aplikacji biznesowej nie zawierają tylko logiki biznesowej (na przykład obliczenie wartości koszyka z zakupami w sklepie internetowym), ale dodatkowo implementację logowania, spójności transakcyjnej, autoryzacji, bezpieczeństwa, synchronizacji wielowątkowej i wiele innych. Jest to oczywiście zjawisko normalne, będące implikacją złożoności wymagań klienta dotyczących projektowanego systemu. Odbiorca na pewno nie zaakceptowałby aplikacji, która nie implementuje wymaganych ortogonalnych zagadnień.

Dobrą metaforą takiego stanu rzeczy jest pryzmat, co ilustruje rysunek 1. Puszczając na niego promień wymagań klienta, uzyskujemy rozczepione poszczególne aspekty. Odpowiedzialnym za ten proces w większości metodyk jest analityk. Jego zadaniem jest przeanalizowanie wymagań klienta oraz wyodrębnienie poszczególnych ich aspektów.



Rys. 1. Metafora pryzmatu (źródło: [Laddad2002])

Zjawisko to, jak mówiłem, jest naturalne. Problem jednak leży w innej kwestii: w sposobie przełożenia szeregu ortogonalnych aspektów wymagań klienta na konkretną implementację (czy we wcześniejszych fazach wytwarzania oprogramowania na model projektowy).

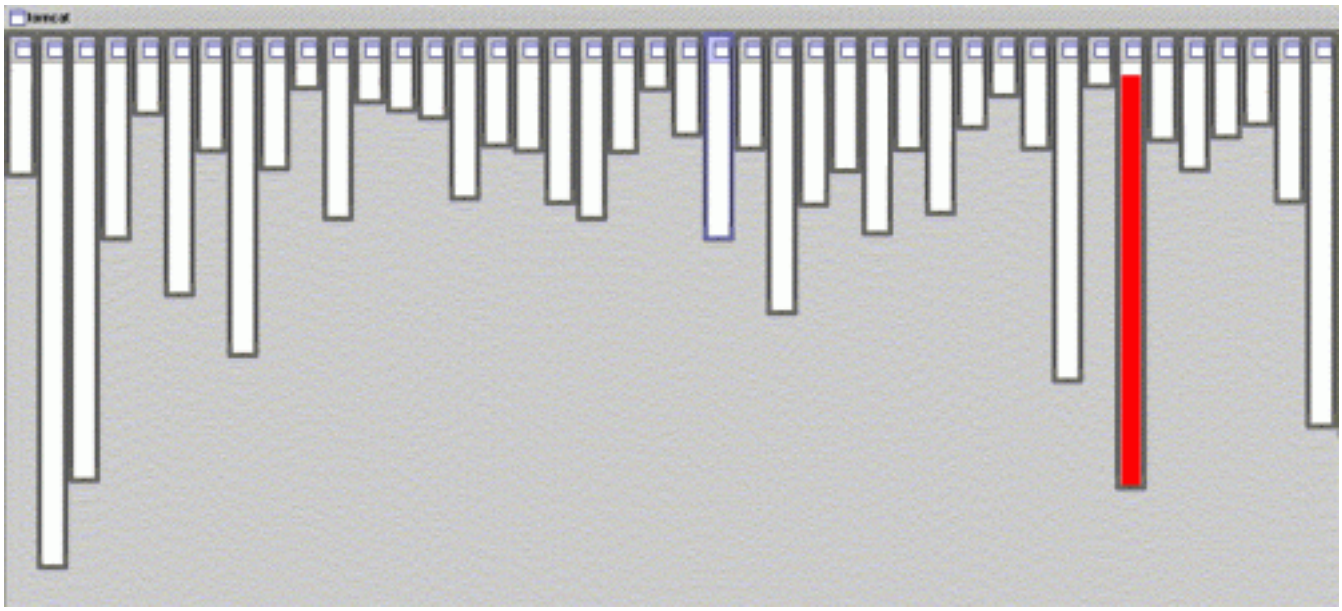
2. Paradygmat programowania obiektowego

Na przestrzeni ostatnich kilkudziesięciu lat pojawiło się wiele różnych paradygmatów programistycznych. Ich rozwój jest podyktowany coraz większą złożonością systemów informatycznych, ale czasami wydaje się być niewystarczający.

Aktualnie powszechne są języki bazujące na paradygmacie programowania obiektowego (np. C#, Java, C++). Dostarczając mechanizmy takie jak klasy, enkapsulacja czy dziedziczenie, pozwalają na modelowanie systemów informatycznych w kategoriach świata rzeczywistego. Człowiek z natury klasyfikuje elementy otoczenia zewnętrznego, więc jest to dla nas naturalne. Pojawiają się jednak sytuacje, gdy języki obiektowe są niewystarczające.

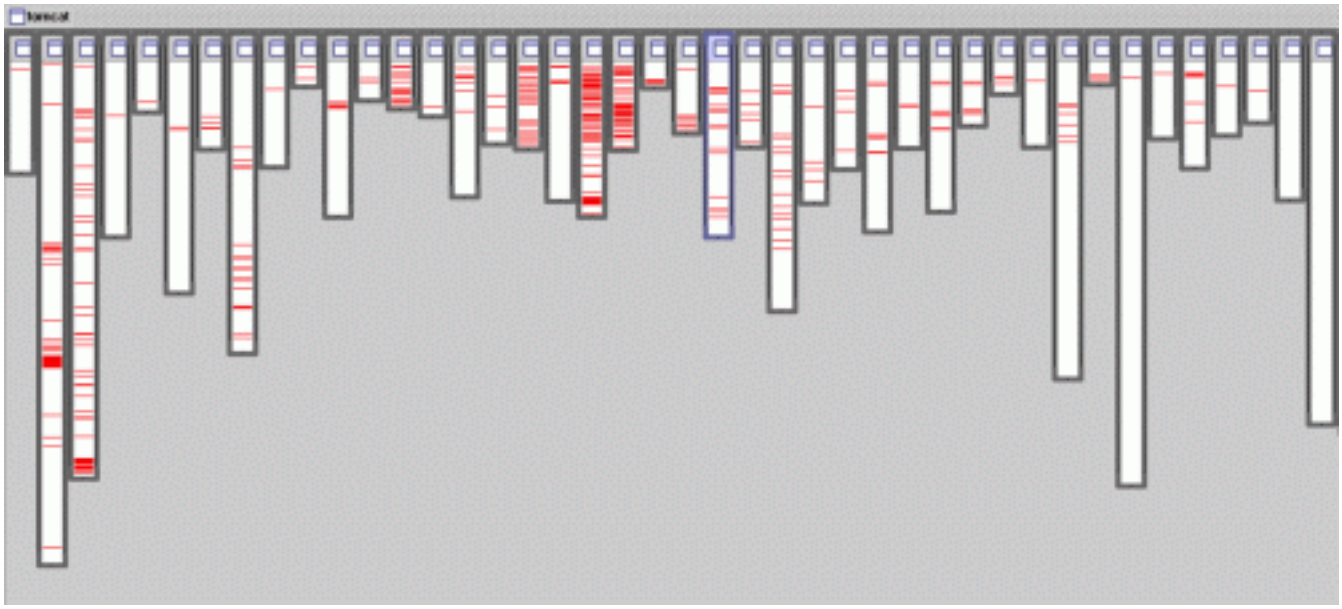
Zilustruje to przykładem dużej aplikacji *OpenSource* zaprojektowanej według paradygmatu obiektowego. Na rysunku 2 został przedstawiony wykres podzbioru klas serwera *Apache*

Tomcat. Każdy słupek obrazuje pojedynczą klasę. Natomiast na czerwono został zaznaczony kod źródłowy odpowiedzialny za przetwarzanie plików formatu XML. Jak widać, kod ten jest znakomicie enkapsulowany w jednej klasie.



Rys. 2. Dobra enkapsulacja na przykładzie serwera Apache Tomcat

Niestety istnieją także takie wymagania klienta, których enkapsulacja w pojedynczej klasie nie jest już taka prosta. Ich realizacja rozprasza się po całym modelu obiektowym (można powiedzieć: przecina go) i nie sposób zamknąć jej w jednym obiekcie. Zilustruje to przykładem przedstawionym na rysunku 3. Tym razem czerwonym kolorem została zaznaczona implementacja logowania (również w serwerze Apache Tomcat). Jak widać, jest rozproszona po większości przedstawionych klas.



Rys. 3. Rozproszenie implementacji logowania w serwerze Apache Tomcat

3. Problem refaktoryzacyjny?

Zastanówmy się, dlaczego niektóre wymagania klienta znakomicie modeluje się przy użyciu paradygmatu obiektowego, a niektóre wręcz przeciwnie. Wydaje się, że rozwiązanie tego problemu można znaleźć w sercu systemów informatycznych, czyli w kodzie źródłowym.

Zilustruję powyższe podejście przykładem. Rozważmy szkielet metody realizującej typową akcję biznesową. Jej odpowiedzialnością jest wykonanie operacji zakupu koszulek w pewnym sklepie. Zamawiający dodatkowo wymaga śledzenia wszystkich operacji (obsługa logowania do dziennika systemowego) oraz integralności operacji (obsługa transakcji).

```
public void sellTShirts(Client client, int num)
{
    super.getLogger().info("Klient " + client.getFullName()
        + " rozpoczął transakcje zakupu koszulek.");
    UserTransaction ut = getUserTransaction();
    try {
        ut.begin();

        // *** wykonaj operację zakupu koszulek ***

        ut.commit();
        super.getLogger().info("Klient " + client.getFullName()
            + " pomyślnie zakończył transakcje zakupu koszulek.");
    } catch(Exception e) {
        ut.rollback();
        super.getLogger().info("Przerwana transakcja zakupu koszulek"
            + " (klient: " + client.getFullName() + ").");
    }
}
```

Powyższy przykład jest realizacją tych wymagań na platformie technologicznej J2EE. Jak widać, aby je zrealizować programista musi poprzeplatać implementacje odpowiedzialne za poszczególne aspekty wymagań klienta. Najpierw loguje rozpoczęcie operacji do dziennika systemowego (kolor czerwony) i rozpoczyna transakcję (kolor niebieski). Dopiero przystępuje do implementacji właściwej odpowiedzialności metody, czyli zakupu koszulek. Ostatecznie kończy transakcję i zapisuje do dziennika systemowego powodzenie operacji.

To przeplatanie implementacji odpowiedzialnej za różne aspekty wymagań klienta jest jedną z najważniejszych wad paradygmatu obiektowego. Jej implikacje są następujące:

- programista nie skupia się na właściwej odpowiedzialności danej klasy czy metody
- opisane wyżej przeplatanie zmniejsza zdolność do ponownego użycia danego kodu
- kod związany z ortogonalnymi aspektami wymagań klienta jest rozproszony w wielu modułach
- pojawia się trudna do refaktoryzacji duplikacja kodu źródłowego

Wróćmy jednak do przykładu. Załóżmy, że analogicznych metod w systemie jest więcej i potraktujmy je jak problem refaktoryzacyjny. Jego rozwiązanie jest kluczem tych rozważań. Dobry programista na pewno znajdzie rozwiązanie tego problemu (na przykład używając szablonów), nie będzie to jednak rozwiązanie przejrzyste i eleganckie. Satysfakcjonujące rozwiązanie powinno tak refaktoryzować poprzedni przykład, aby metoda zawierała tylko i wyłącznie jej główną odpowiedzialność. Powinno doprowadzić ją do poniższej postaci.

```
public void sellTShirts(Client client, int num)
{
    // *** wykonaj operację zakupu koszulek ***
}
```

4. Programowanie aspektowe

Programowanie aspektowe (ang. *Aspect-Oriented Programming*) jest odpowiedzią na opisany powyżej problem. Rozszerza paradygmat obiektowy dodając idee aspektów. W rezultacie pozwala na **przejrzyste** odwzorowanie ortogonalnych aspektów wymagań klienta na ortogonalne aspekty implementacji. Umożliwia unikniecie przedstawionego przed chwilą przeplatania kodu (*warkocza*).

Przykładowo, jeżeli wymagane jest śledzenie operacji zakupów, to programista tworzy nowy **aspekt** (tak jak jednostką modularności w językach obiektowych jest klasa, tak w językach aspektowych jest to aspekt), który odpowiada za obsługę logowania. Podobnie można enkapsulować wszystkie poboczne aspekty wymagań klienta. W wyniku czego klasy implementujące główny cel aplikacji (np. logikę biznesową) zawierają tylko i wyłącznie to, za co odpowiadają.

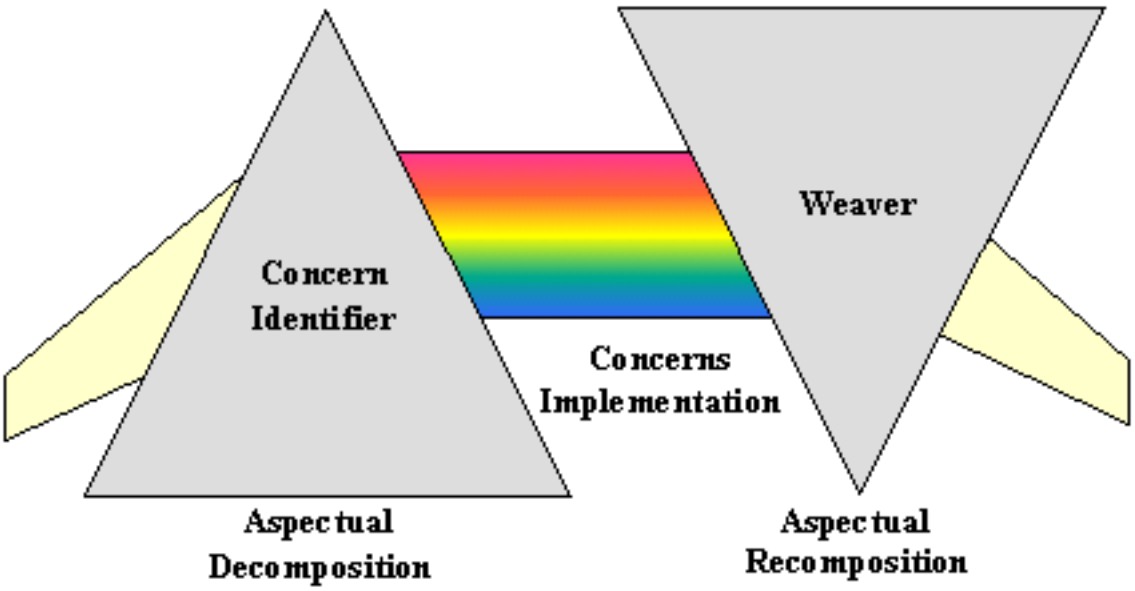
Aspekty w połączeniu z klasami umożliwiają o wiele lepszą dekompozycję systemu informatycznego niż zastosowanie samych klas. Często mówi się, że jest to dekompozycja wielowymiarowa. Klasy realizujące główny cel są na jednej płaszczyźnie, aspekty na ortogonalnej (patrz rysunek 4). Płaszczyzny te przecinają się, gdzie punktami przecięć są wywołania aspektów. Wracając do przykładu, klasy logiki biznesowej odpowiadające za operację zakupu towarów leżałyby na jednej płaszczyźnie, na ortogonalnej do niej - aspekty odpowiadające za obsługę dziennika systemu itd.

Rys. 4. Wielowymiarowość programowania aspektowego

Tak naprawdę, przeplatanie kodu źródłowego jest nie do uniknięcia. Ale programując aspektowo przesuwamy ten proces w czasie do przodu. Jest on wykonywany automatycznie podczas kompilacji lub wykonania programu. Proces ten nazywany jest **tkaniem** (ang. *weaving*), a odpowiedzialny za niego jest **tkacz** (ang. *weaver*).

Na czym ten proces polega? Tkanie dodaje w punktach złączeń (ang. *join points*) do klasy docelowej kod zawarty w aspektach. Przykładowymi punktami złączeń są *"przed wywołaniem metody"* lub *"po rzuceniu wyjątkiem"*.

Przeanalizujmy rolę tkacza w procesie wytwarzania oprogramowania opierając się na metaforze pryzmatu (rysunek 5). Jeżeli rzucimy promień wymagań na pierwszy pryzmat (metafora



Rys. 5. Tkanie w metaforze przymatu (źródło: [Laddad2002])

analityka), uzyskamy wyodrębnione aspekty wymagań klienta. Każdy z nich, łącznie z głównym celem systemu, jest implementowany i enkapsulowany w jednostkach modularności (w klasach lub aspektach). Następnie padają one na drugi pryzmat (metafora tkacza) i uzyskujemy kod wynikowy zawierający implementację głównego celu oraz aspektów ortogonalnych.

5. Anatomia programu aspektowego na przykładzie

Abstrakcyjne teorie najłatwiej wyjaśnia się na konkretnym przykładzie. Poniżej przedstawiam klasę `HelloWorld` realizującą klasyczny przykład *HelloWorld* (źródło: [Laddad2002]).

```
public class HelloWorld {
  public static void say(String message) {
    System.out.println(message);
  }

  public static void sayToPerson(String message, String name) {
    System.out.println(name + ", " + message);
  }
}
```

Dodajmy do niej obsługę zwrotów grzecznościowych, tak aby przed wypisaniem zadanego komunikatu klasa przywitała się (*"Good day!"*), a po – podziękowała (*"Thank you!"*).

```
public aspect MannersAspect {
  pointcut callSayMessage()
              : call(public static void HelloWorld.say*(..));

  before() : callSayMessage() {
    System.out.println("Good day!");
  }
```

```
  after() : callSayMessage() {
    System.out.println("Thank you!");
  }
}
```

Przyjrzyjmy się dokładniej przykładowemu aspektowi. Zawiera on definicję jednej linii cięć (konstrukcja `pointcut`), która określa zbiór miejsc uruchomienia kodu aspektowego. W przykładzie linia cieć `callSayMessage` składa się z wywołań publicznych, a zarazem statycznych metod klasy `HelloWorld`. Dodatkowo ich nazwa musi zaczynać się od prefiksu *"say"*.

Po określeniu miejsc uruchomienia kodu aspektowego, możemy go teraz napisać. Zawierają go zdefiniowane w przykładzie dwie rady. Pierwsza wywoływana przed cięciem `callSayMessage` wypisuje komunikat przywitania. Natomiast druga wywoływana po – wypisuje komunikat podziękowania.

Tak napisany kod jest przetwarzany przez tkacza, który automatycznie przeplata kod obiektowy z aspektowym. W rezultacie uzyskujemy analogiczny wynik, jaki wygenerowałaby nam poniższa klasa. Przy czym modularność obu tych rozwiązań jest nieporównywalna.

```
public class HelloWorld {
  public static void say(String message) {
    System.out.println("Good day!");
    System.out.println(message);
    System.out.println("Thank you!");
  }

  public static void sayToPerson(String message, String name) {
    System.out.println("Good day!");
    System.out.println(name + ", " + message);
    System.out.println("Thank you!");
  }
}
```

6. Implementacje AOP

Macierzystą platformą technologiczną, na której wyrosło programowanie aspektowe jest Java. Dlatego dziwne nie jest, że cieszy się najbardziej dojrzałymi implementacjami paradygmatu AOP. Przede wszystkim trzeba wspomnieć o projekcie AspectJ [AspectJ], którego prekursorem jest Gregor Kiczales - pionier podstaw teoretycznych programowania aspektowego. Ta sama osoba jest głównym założycielem towarzystwa AOSA (*Aspect-Oriented Software Association*, [AOSA]), której głównym celem jest organizacja konferencji tyczących programowania aspektowego.

Oczywiście, programowanie aspektowe nie jest ograniczone tylko do języka Java. Tak jak dla języka C++[AspectC], tak samo dla C# powstają rozszerzenia zorientowane aspektowo. Są to jednak prototypy, eksperymenty naukowe. Brak niestety stabilnej implementacji, którą można byłoby użyć bez obaw w projektach komercyjnych.

Jedynym projektem opartym na platformie .NET, który został wybrany przez AOSA jako *"narzędzie dla praktyków"* jest biblioteka RAPIER-LOOM.NET[LoomNet][Codeguru2004]. Reszta jest określona jako *"projekty badawcze"*.

7. Podsumowanie

Programowanie aspektowe na razie się rozwija. Potrzebne jest kilka lat, abyśmy otrzymali stabilne narzędzia czy dobre implementacje tkaczy. To jednak nie wszystko. Oprócz narzędzi, potrzebne jest wypracowanie dobrych praktyk (*best practices*) oraz wzorców projektowych opartych na programowaniu aspektowych. Brakuje również ustalonej notacji w językach modelowania, takich jak UML. Jednak najbardziej dotkliwy jest brak programistów piszących aspektowo. Nie wystarczy stworzyć narzędzie, trzeba również nauczyć się go używać.

Literatura

[Laddad2002] Laddad Ramnivas, *I want my AOP!, Part 1-3, Separate software concerns with aspect-oriented programming*, JavaWorld, 2002

[Codeguru2004] Stochmiałek Michał, *Wprowadzenie do programowania aspektowego*, CodeGuru.pl, 2004

[AOSA] Strona towarzystwa *Aspect-Oriented Software Association*, `www.aosd.net`

[AspectJ] Strona projektu AspectJ, `www.eclipse.org/aspectj/`

[AspectC] Strona projektu AspectC, `www.aspectc.org`

[LoomNet] Strona projektu RAPIER-LOOM.NET, `www.aspectc.org`