Programowanie aspektowe sposobem na ulepszona modularyzacje systemu

Przygotowane przez Pawla Slowikowskiego (ps@agh.edu.pl)

Niezaprzeczalnym faktem jest to, ze rozmiar prawie kazdego oprogramowania komputerowego z czasem rosnie, co objawia sie wzrostem poziomu jego skomplikowania i zlozonosci. Wynikiem przybywajacego kodu zródlowego jest oprogramowanie coraz bardziej niejasne i nieprzejrzyste, nawet mimo umieszczania licznych komentarzy. Dotyczy to wszystkich modulów oprogramowania, których kod zródlowy z czasem zaczyna tracic spójnosc, a funkcjonalnosc przecinac inne moduly. Nie jest to powodem do zadowolenia ani dla szefów projektów, ani dla programistów, poniewaz zarzadzanie zlozonymi projektami jest niezmiernie trudne i, niestety, czesto prowadzi do produktów o obnizonej jakosci.

Ewolucja technik programistycznych

W ostatniej dekadzie podejscie obiektowe do tworzenia systemów informatycznych prawie calkowicie wyparlo podejscie proceduralne. O programowaniu obiektowym zaczynalo sie mówic juz z koncem lat szescdziesiatych i trwalo paredziesiat nastepnych lat, aby jego idea zyskala tak szerokie poparcie, jakie ma w tej chwili. Dzisiaj juz chyba nikt nie wyobraza sobie tworzenia duzego systemu bez zastosowania technik obiektowych. Czy mozna pisac oprogramowanie jeszcze szybciej, lepiej i latwiej niz obecnie? Przy uzyciu klasycznych metod obiektowych – raczej nie. Powstalo jednak kilka pomyslów, takich jak programowanie aspektowe, systemy komponentowe czy wzorce, starajacych sie rozwiazac ten problem na rózne sposoby.
Referat ten poswiecony jest pierwszemu z wymienionych rozwiazan. Jezykom programowania aspektowego przyswieca idea modularyzacji przecinajacych sie zagadnien, czyli takiego rozdzielenia przecinajacych sie aspektów funkcjonalnosci programu (np. synchronizacji dostepu do zasobów czy zarzadzania bezpieczenstwem), aby odpowiedzialny za nie kod tworzyl osobne moduly. Jezyki aspektowe, dostarczajac nowa metodyke, pozwalaja na nieporównywalnie wieksza modularyzacje niz jezyki proceduralne i obiektowe. Czesto mozna sie spotkac z opiniami, ze programowanie aspektowe w porównaniu z obiektowym wprowadza podobna zmiane jakosciowa, jak programowanie obiektowe w porównaniu z proceduralnym. Nalezy jednak zaznaczyc, ze programowanie aspektowe nie jest bezposrednio zwiazane ani z programowaniem obiektowym, ani z zadnym konkretnym jezykiem programowania. Istnieja jezyki aspektowe oparte zarówno na C++, jak i na C. Przykladem najbardziej zaawansowanego jezyka aspektowego, szerzej opisanym ponizej, jest oparty na bazie Javy, a powstaly w laboratoriach Xerox PARC, AspectJ.

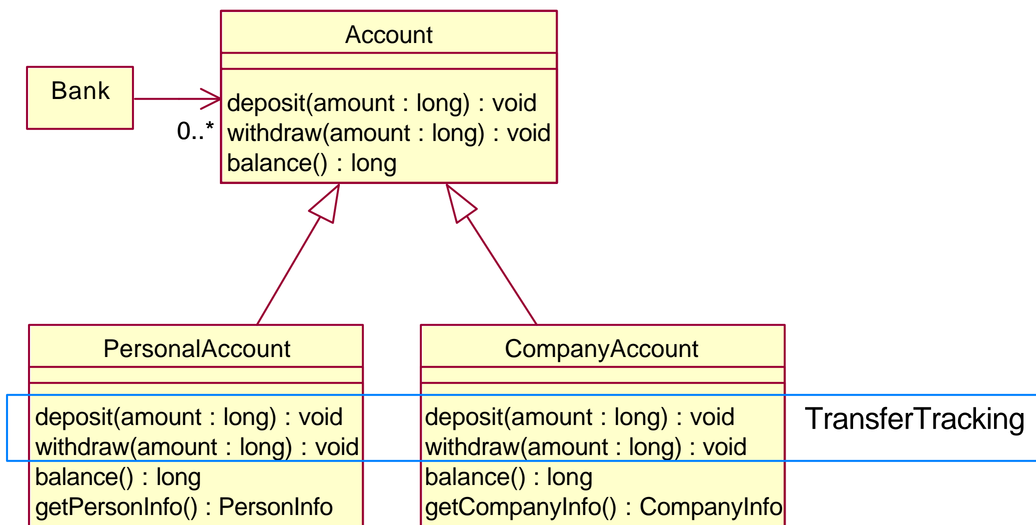
AspectJ – aspektowe rozszerzenie Javy

Pierwsze koncepcje zwiazane z programowaniem aspektowym pojawily sie juz w latach siedemdziesiatych, jednak prawdziwy ich rozwój to historia ostatnich kilkunastu lat. AspectJ jest wynikiem ponad dziesiecioletnich prac prof. Gregora Kiczalesa oraz kierowanej przez niego grupy badawczej, zlozonej z naukowców University of British Columbia oraz pracowników firmy Xerox. W chwili obecnej dostepna jest wersja AspectJ 1.0candidate2 i do tej wersji odnosi sie referat.
Podstawowa i naturalna jednostka modularnosci w jezykach obiektowych jest klasa, które posiada okreslona funkcjonalnosc oraz zdefiniowany interfejs. W AspectJ jednostka taka jest aspekt – zagadnienie przecinajace wiecej niz jedna klase. Programy w AspectJ skladaja sie z klas reprezentujacych tradycyjna strukture modularna oraz aspektów przecinajacych strukture klas. Realizacja przecinania zagadnien oparta jest przede wszystkim na dwóch mechanizmach: punktach zlaczen (ang. *join points*) oraz radach (ang. *advices*). Punkty zlaczen sa okreslonymi punktami wykonania programu, dostepnymi z poziomu jezyka za pomoca deklaracji punktów ciec (ang. *pointcuts*). Punkty wykonania programu identyfikuja wywolania i wykonania metod, konstruktorów i kodu obslugujacego wyjatki oraz odczytywanie i zapisywanie pól. Z punktami zlaczen powiazane sa rady – konstrukcje programowe podobne do metod, zawierajace kod wykonywany w czasie osiagniecia punktu zlaczenia. Rady moga byc wykonywane przed, po lub zamiast punkt zlaczenia.

Aspekt jest konstrukcja programowa skladajaca sie punktów ciec oraz z rad, moze równiez, podobnie jak klasa, zawierac normalne metody, byc abstrakcyjny oraz dziedziczony z innych aspektów. AspectJ udostepnia takze mechanizmy zwane wprowadzeniami (ang. *introduction*) pozwalajace modyfikowac statyczne struktury programu takie jak skladniki klas oraz zaleznosci pomiedzy klasami.

Zastosowanie AspectJ w systemie kont bankowych

W celu zaprezentowania AspectJ na konkretnych przykladach, wykorzystany zostanie fragment prostego systemu bankowego, którego diagram przedstawiony jest na rysunku 1. Klasa `Account` reprezentuje konto bankowe. Klasy `PersonalAccount` i `CompanyAccount` odpowiadaja odpowiednio kontu prywatnemu i kontu firmy. Klasy posiadaja metody pozwalajace na sprawdzenie i zmiane salda, oraz na uzyskanie dodatkowej informacji o wlascicielu konta.



Rysunek 1 Diagram klas czesci prostego systemu kont bankowych. Prostokat z etykieta „TransferTracking" pokazuje aspekt przecinajacy klasy PersonalAccount i CompanyAccount.

Punkty zlaczen

Punkty zlaczen identyfikuja pewne miejsca programu. AspectJ okresla kilka rodzajów punktów zlaczen (patrz Tabela 1).

Tabela 1 Punkty zlaczen zdefiniowane w AspectJ

Rodzaj punktu zlaczenia	Umiejscowienie w programie:
wywolanie metody	miejsce, z którego wywolywana jest metoda
wykonanie metody	miejsce, w którym zaczyna byc wykonywana metoda
wywolanie konstruktora	miejsce, gdy obiekt jest juz stworzony i nastepuje wywolanie konstruktora, wylaczajac wywolania konstruktora poprzez *this* i *super*
wykonanie inicjatora	miejsce, w którym wykonywany jest niestatyczny inicjator klasy
wykonanie konstruktora	miejsce, w którym wykonywany jest konstruktor, ale po wywolaniu konstruktorów nadklas
wykonanie statycznego inicjatora	miejsce, w którym wykonywany jest statyczny inicjator klasy
inicjalizacja obiektu	miejsce, w którym wykonywany jest kod inicjalizujacy obiekt; obejmuje czas pomiedzy powrotem z wywolania konstruktora nadklasy a powrotem z pierwszego

	wywolania wlasnego konstruktora; obejmuje wszystkie dynamiczne inicjatory i konstruktory uzyte do tworzenia obiektu
odczytanie pola	miejsce, w którym nastepuje odczytanie pola nie zadeklarowanego jako *final*
ustawienie pola	miejsce, w którym ustawiane jest pole
wykonanie kodu obslugi wyjatków	miejsce, w którym wywolywany jest kod obslugi wyjatku

Oznaczenia punktów ciec

Oznaczenia punktów ciec okreslaja punkty zlaczen poprzez definicje pewnych zdarzen pojawiajacych sie w trakcie wykonania programu, takich jak np. wywolanie metody czy obsluzenie wyjatku. Oznaczenia punktów ciec moga skladac sie z pojedynczej deklaracji, jak ponizej:

```
call( void PersonalAccount.deposit( long ) )
```

lub tez moga byc laczone w bardziej skomplikowane struktury przy pomocy logicznych operatorów: *and*, *or* i *not* („&&", „||" i „!") oraz nawiasów. Prosty przyklad zlozonego oznaczenia punktu ciecia znajduje sie ponizej:

```
call( void PersonalAccount.deposit( long ) ) ||
call( void PersonalAccount.withdraw( long ) )
```

Pierwszy z przykladów oznaczników definiuje wszystkie wywolania metody `deposit()` obiektów klasy `PersonalAccount`, drugi natomiast wprowadza dodatkowe rozszerzenie o wywolania metody `withdraw()`. Ponizsza deklaracja równowazna jest deklaracji z przykladu drugiego:

```
target( PersonalAccount ) &&
( call( void deposit( long ) ) || call( void withdraw( long ) ) )
```

Konstrukcja `target( PersonalAccount )` oznacza wszystkie punkty wykonania programu, w których obiekt docelowy jest klasy `PersonalAccount`, natomiast `call( void deposit( long ) )` i `call( void withdraw( long ) )` okreslaja wywolania metod `deposit( long )` oraz `withdraw( long )`, bez wzgledu na klase w której sa zdefiniowane.
Punktom ciec mozna nadawac nazwy, a nastepnie wykorzystywac przy definiowaniu innych punktów ciec lub rad. Ponizej przedstawiona jest deklaracja nazwanego punktu ciecia `moneyTransfer()`:

```
pointcut moneyTransfer(): target( Account ) &&
      ( call( void deposit( long ) ) || call( void withdraw( long ) ) )
```

Przy definiowaniu oznaczen punktów ciec mozemy poslugiwac sie znakami ogólnymi. Przykladowa deklaracja:

```
call( * d*(..) )
```

oznacza wywolania metod które spelniaja ponizsze warunki:

- nazwa metody pasuje do wzorca „`d*`",
- zwracaja rezultat dowolnego typu (pierwsza „`*`"),
- parametry metody sa dowolne („`(..)`") .

Powyzsze warunki spelnia np. metoda `deposit()` klasy `PersonalAccount`.

AspectJ udostepnia równiez mechanizmy pozwalajace na definicje oznaczen punktów ciec w oparciu o dynamiczny kontekst innych punktów ciec. Ponizszy oznacznik:

```
cflow( moneyTransfer() )
```

identyfikuje wszystkie punkty zlaczen pojawiajace sie pomiedzy otrzymaniem sterowania przez metode nalezaca do punktu ciecia `moneyTransfer()`, a powrotem z jej wywolania.

Rady

Punkty ciec wykorzystywane sa przy definiowaniu rad. Rada jest kodem programu wykonywanym przy osiagnieciu punktu zlaczenia okreslonego poprzez punkt ciecia w definicji rady. Kod ten moze byc wykonany przed, po lub zamiast akcji identyfikowanej przez punkt zlaczenia. Przykladem rady jest deklaracja:

```
after(): moneyTransfer() {
      System.out.println( ”Money transfer” );
}
```

Skutkiem jej dzialania bedzie wypisanie na ekranie tekstu ”Money transfer” po kazdym powrocie z funkcji `deposit()` lub `withdraw()` (zgodnie z deklaracja punktu ciecia `moneyTransfer()` wprowadzona w jednym z poprzednich przykladów).

Wprowadzenia

Wprowadzenie (ang. *introduction*) jest mechanizmem pozwalajacym na modyfikacje klas oraz ich hierarchii. Wprowadzenie pozwala na dodawanie nowych skladników do istniejacych klas, jak i na zmiane relacji dziedziczenia, bez koniecznosci bezposredniej ingerencji w istniejacy kod klasy. Wprowadzanie zmian odbywa sie statycznie, w czasie kompilacji. Poprzez wprowadzenie nowych pól i metod zyskujemy mozliwosc dodania niezbednej funkcjonalnosci, wymaganej przez projektowane aspekty.

Aspekty

Podstawowa jednostka modularnosci w AspectJ jest aspekt. Deklaracja aspektu zblizona jest do deklaracji klasy: moze posiadac metody, pola oraz inicjatory. Do definiowania przecinajacych sie zagadnien sluza deklaracje punktów ciec, rad i wprowadzen. Przyklad deklaracji prostego aspektu sledzacego transfer pieniadza w systemie bankowym znajduje sie ponizej:

```
aspect TransferTracking {
      pointcut transfers( Account account, long amount ):
            target( account ) && args( amount ) && (
            call( void withdraw( long ) )||
            call( void deposit( long ) ) );

      after( Account account, long amount ): transfers( account, amount ) {
            System.out.println("TransferTracking: " +
            thisJoinPoint.getSignature().getName() +
            "( " + amount + " ). Account: " + account );
      }
}
```

W ciele rady mozemy odwolywac sie do punktu zlaczenia, który rade wywolal, m.in. poprzez specjalny obiekt `thisJoinPoint` implementujacy interfejs `JoinPoint`. Interfejs `JoinPoint` pozwala na uzyskanie róznych informacji o punkcie zlaczenia w kontekscie wywolanej rady, takich jak sygnatura czy argumenty punktu zlaczenia. W prezentowanej deklaracji konstrukcja:

```
thisJoinPoint.getSignature().getName()
```

zwraca nazwe wywolywanej metody, która w tym przypadku moze byc „withdraw” lub „deposit”. Wprowadzony punkt ciecia `args( amount )` okresla te punkty zlaczen, których argument jest tego samego typu co `amount` (typu `long`).

Modularyzacja systemu

Programowanie aspektowe pozwala na lepsza modularyzacje systemu. W przypadku omawianego systemu kont bankowych mozemy wyróznic kilka zagadnien, których przy uzyciu programowania obiektowego nie da sie dobrze zmodularyzowac. Zagadnieniami tego typu sa np. sledzenie wywolywania operacji bankowych, kontrola dostepu kont, obsluga transakcji i obsluga bledów. Przy próbach implementacji wymienionych mechanizmów uzyskamy kod rozproszony po róznych modulach, trudny do zrozumienia i zarzadzania, a przede wszystkim ciezki do rozwoju i pielegnacji. Aspekty pozwalaja na stworzenie takich modulów, których funkcjonalnosc bedzie z góry okreslona, a kod odizolowany od innych czesci oprogramowania.
Nalezy wspomniec, ze programowanie aspektowe mozemy stosowac na dwa sposoby. Jednym z nich jest wykorzystanie programowania do wzbogacania funkcjonalnosci systemu. Drugi sposób to uzycie aspektów tylko w fazie tworzenia systemu np. do sledzenia przebiegu wywolan funkcji, badania charakterystyki programu czy debugowania. Takie aspekty mozna w latwy sposób usunac z projektu przed faza ostatecznego wydania oprogramowania, bez koniecznosci zmian w reszcie kodu.

Narzedzia

Sposób, w jaki aspekty wplywaja na strukture programu moze byc nieslychanie zlozony i na pierwszy rzut oka nielatwy do zrozumienia. Twórcy AspectJ widza narzedzia IDE jako integralna czesc pakietu. Dla programistów przygotowali rozszerzenia do srodowisk: JBuilder (wersja 4 i 5), Forte for Java (wersja 2 i 3), Netbeans 3.2, Emacs (wersja 20.3), XEmacs (wersja 21.1 dla Unix'a i 21.4 dla Windows), a w planie sa rozszerzenia do innych popularnych srodowisk programistycznych. W pakiecie dostarczane sa równiez kompilator (*ajc*), debugger (*ajdb*) oraz generator dokumentacji (*ajdoc*). Dostepna jest równiez dokumentacja do jezyka oraz przyklady.
Caly pakiet dostepny jest jako Open Source na zasadach Mozzila Public Licence.

Zalety i wady AspectJ

Glówna zaleta AspectJ jest mozliwosc ulepszenia modularyzacji systemu, zwlaszcza takich aspektów oprogramowania, które wystepuja prawie w kazdym systemie, a których modularyzacja za pomoca metod obiektowych jest trudna do osiagniecia. Przykladami takich zagadnien sa: kontrola bledów, synchronizacja dostepu do zasobów, rozproszenia obiektów. Za lepsza modularyzacja kryje sie kod latwiejszy do zrozumienia, pielegnacji, modyfikacji i wielokrotnego uzycia.
Glówna wada AspectJ jest brak wypracowanej metodyki programowania aspektowego. Poniewaz AspectJ daje potencjalnie bardzo duze mozliwosci wplywu na strukture programu, nieuwazne i nieprzemyslane uzycie mechanizmów jezyka moze prowadzic do zlego kodu, którego struktura z czasem stanie sie elementem blokujacym rozwój systemu. Niebezpieczenstwem zwiazanym z AspectJ jest takze uzywanie technologii bedacej caly czas w fazie badan i rozwoju. Wiaze sie z tym pewne ryzyko, które z drugiej strony równowazone jest publicznym dostepem do zródel i dosyc dobra pomoca techniczna ze strony twórców.
Zawsze jednak, przez zdecydowaniem sie na uzycie AspectJ we wlasnych projektach, nalezy zastanowic sie nad skala jego wykorzystania w projekcie oraz przeanalizowac potencjalne korzysci i niebezpieczenstwa wiazace sie z jego zastosowaniem.

Rozwój programowania aspektowego

Programowanie aspektowe jest czestym tematem na konferencjach dotyczacych programowania obiektowego. Wiele grup badawczych zajmuje sie tworzeniem jezyków pozwalajacych na lepsza modularyzacje systemów niz osiagalne jest to za pomoca obecnie stosowanych metod. Powstaja jezyki aspektowe oparte zarówno na istniejacych jezykach obiektowych (takich jak Java, C++,

Smalltalk), jak i na jezykach proceduralnych (np. na C). Pojawiaja sie równiez eksperymentalne jezyki aspektowe stworzone calkowicie od podstaw. Niezmiernie popularna jest takze tematyka blisko zwiazana z programowaniem aspektowym: programowanie adaptacyjne, programowanie tematowe, meta-programowanie i refleksja.
Wydaje sie, ze w najblizszych latach programowanie aspektowe bedzie zyskiwalo coraz wieksza popularnosc i kwestia czasu jest tylko to, kiedy z laboratoriów trafi „pod szczechy". Ja osobiscie zachecam do blizszego zaznajomienia sie z programowanie aspektowym, a szczególnie z AspectJ, który jest juz na tyle dojrzalym produktem, ze nadaje sie do prób wykorzystywac w procesie tworzenia oprogramowania.
Pod adresem http://aspectj.org/ mozna znalezc wszystkie potrzebne skladniki do rozpoczecia pracy z AspectJ. Na tymze serwerze WWW znajduja sie takze odnosniki do prac poswieconych programowaniu aspektowemu oraz pokrewnej tematyce badawczej.