Programowanie aspektowe – AspectJ

Redaktorzy styczniowo-lutowego wydania MIT Technology Review [1] z 2001 r. wybrali dziesiec nowo powstalych obszarów technologii, które wkrótce – wedlug autorów – beda mialy znaczacy wplyw na ekonomie oraz sposób, w jaki ludzie zyja i pracuja. Za jeden z takich obszarów zostalo uznane programowanie aspektowe (*Aspect-Oriented Programming (AOP)*). Niniejszy artykul przybliza tematyke zwiazana z programowaniem aspektowym oraz jezyk AspectJ, bedacy aspektowym rozszerzeniem Javy.

Ewolucja technik programistycznych

Od dawna mozna obserwowac tendencje do osiagania przez programy komputerowe coraz wiekszych rozmiarów. Ze wzrostem tym wiaze sie tez przyrost skomplikowania i zlozonosci powstajacych systemów. Dotyczy to wszystkich aspektów ich funkcjonalnosci, miedzy innymi bezpieczenstwa, zarzadzania zasobami czy obslugi bledów. Duza zlozonosc systemów informatycznych, rozbicie kodu spelniajacego okreslona funkcjonalnosc po róznych modulach, przeplatanie go z kodem odpowiedzialnym za inne aspekty programu oraz wielokrotne wystepowanie tych samych fragmentów kodu w róznych miejscach utrudniaja rozwój i konserwacje oprogramowania. Wszystkie potencjalne zmiany w programie wiaza sie z ogromnym ryzykiem wygenerowania nowych bledów.
Znaczna wiekszosc powstajacego obecnie oprogramowania implementowana jest za pomoca jezyków obiektowych. Podejscie obiektowe pomaga lepiej zrozumiec rozpatrywany problem (obiekty odzwierciedlaja elementy swiata rzeczywistego), pozwala na hermetyzacje (ukrywanie informacji) i dziedziczenie. Ulatwia tez powtórne wykorzystywanie stworzonych elementów i ich przystosowywanie do nowych potrzeb. Niestety, takie podejscie ma równiez wady. W przypadku duzych aplikacji powstaly kod zwykle nie jest zbyt przejrzysty i do konca zrozumialy.
Na bazie programowania proceduralnego i obiektowego, zapewniajacego juz duza modularnosc projektów, powstala i jest caly czas rozwijana metodologia programowania aspektowego. Ma ona na celu udoskonalenie tzw. rozdzielenia zagadnien (ang. *separation of concerns*), czyli rozdzielenia pewnych aspektów funkcjonalnosci programu (np. synchronizacji dostepu do zasobów, sledzenia przebiegu programu) na osobne, niezalezne moduly. Pozadane jest przy tym, aby modularnosc systemu bardziej odzwierciedlala sposób, w jaki myslimy o problemie, niz metode narzucana przez jezyk lub inne narzedzia.
W przypadku zlozonych systemów programowanie obiektowe nie pozwala na modularyzacje wszystkich zagadnien systemu. Pewne problemy zawsze beda przecinaly granice (ang. *crosscutting*) innych modulów. Idea programowania aspektowego jest dostarczenie mechanizmów pozwalajacych na pelniejsza modularyzacje systemu. Ma ono uproscic kod, zwiekszyc predkosc jego powstawania, uczynic go latwiejszym do zrozumienia, rozwoju, konserwacji i ponownego uzycia.
Zmodularyzowane zagadnienia przecinajace (ang. *crosscutting concerns*) nazywane sa aspektami (zagadnienia te zostana wyjasnione w dalszej czesci artykulu). Jednym z rozwiazan pozwalajacych na wykorzystanie w praktyce idei programowania aspektowego jest AspectJ.

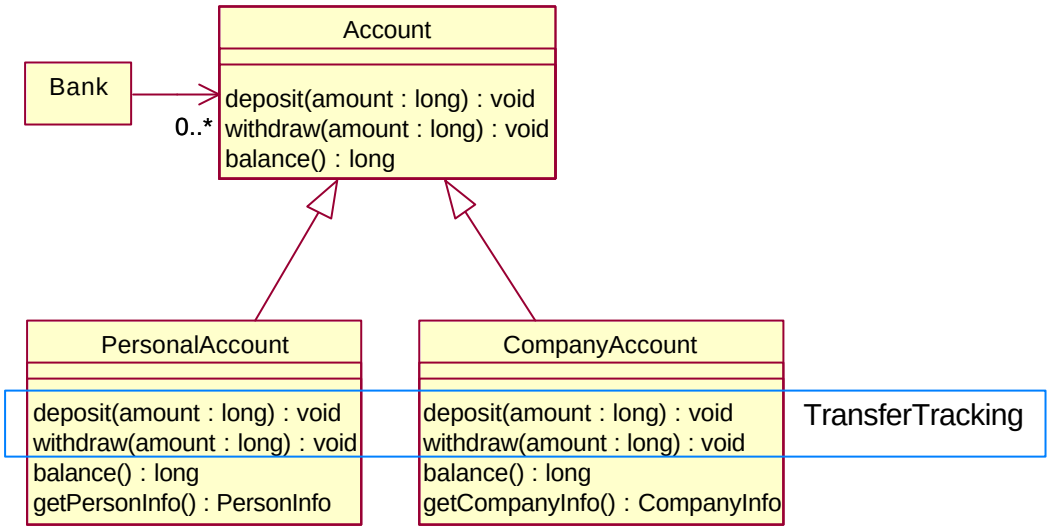
AspectJ metoda na problemy

Pierwsze koncepcje zwiazane z programowaniem aspektowym pojawily sie w latach siedemdziesiatych. Jednym z pionierów tej dziedziny informatyki byl Gregor Kiczales, profesor University of British Columbia i pracownik Xerox Palo Alto Research Center. W wyniku ponad dziesiecioletnich prac jego zespolu pojawil sie AspectJ – uniwersalne aspektowe rozszerzenie Javy, wzbogacajace Jave mozliwosciami programowania aspektowego. W chwili powstawania tego

artykulu dostepna jest wersja AspectJ 0.8 (do tej wersji odnosi sie artykul) i trwaja intensywne prace nad wersja AspectJ 1.0.

Przecinanie implementacji

Naturalna jednostka modularnosci w Javie i w innych jezykach obiektowych jest klasa. W AspectJ jednostka taka jest aspekt – czyli zagadnienie przecinajace wiecej niz jedna klase. Programy w AspectJ skladaja sie z klas reprezentujacych tradycyjna strukture modularna, oraz z aspektów przecinajacych strukture klas. Rysunek 1 przedstawia przykladowy aspekt, reprezentujacy sledzenie operacji bankowych zwiazanych z przeplywem pieniadza.



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

AspectJ zawiera dwa mechanizmy przecinania implementacji. Pierwszy z nich umozliwia definiowanie kodu który wykonywany jest w pewnych okreslonych sytuacjach w czasie dzialania programu. Mechanizm ten nazywany jest dynamicznym przecinaniem (ang. *dynamic crosscutting*). Drugi ze wspomnianych mechanizmów umozliwia definiowanie nowych operacji w odniesieniu do istniejacych typów i nosi nazwe statycznego przecinania (ang. *static crosscutting*). W dalszej czesci zostanie opisany dokladniej tylko mechanizm dynamicznego przecinania.

Dynamiczne przecinanie

Modularyzacja w AspectJ realizowana jest za pomoca punktów zlaczen (*join points*) i rad (*advices*). Punkty zlaczen sa okreslonymi miejscami w programie, natomiast rady sa kodem wykonywanym w czasie osiagniecia punktów zlaczen.

Punkty zlaczen

Struktura zagadnien przecinajacych okreslona jest bezposrednio poprzez punkty zlaczen. AspectJ dostarcza kilkunastu rodzajów punktów zlaczen (patrz: Tabela 1).

Tabela 1. Lista dynamicznych punktów zlaczen

Rodzaj punktu zlaczenia	**Umiejscowienie w programie**
wywolanie metody, wywolanie konstruktora	miejsce, z którego wywolywana jest metoda (lub konstruktor klasy).

otrzymanie wywolania metody, otrzymanie wywolania konstruktora	miejsce, w którym obiekt otrzymuje wywolanie metody lub konstruktora.
wykonanie metody, wykonanie konstruktora	miejsce, w którym zaczyna byc wykonywana metoda lub konstruktor.
odczytanie pola	miejsce, w którym odczytywane jest pole obiektu, klasy lub interfejsu
ustawienie pola	miejsce, w którym ustawiane jest pole obiektu lub klasy
wykonanie programu obslugi wyjatków	miejsce, w którym wywolany jest program obslugi wyjatków
inicjalizacja klasy	miejsce, w którym wywolywane sa statyczne inicjatory klasy
inicjalizacja obiektu	miejsce, w którym w czasie tworzenia obiektu wywolywane sa dynamiczne inicjatory

Jednym z wymienionych typów punktów zlaczen jest wywolanie metody. Obejmuje on wszystkie akcje obiektu otrzymujacego wywolanie metody, zaczynajace sie po wyznaczenia wartosci argumentów metody, a konczace na powrocie z jej obslugi – zarówno normalnym jak i naglym (spowodowanym np. wyslaniem wyjatku).

Oznaczenia punktów ciec

Kolekcje punktów zlaczen, mogace nalezec do róznych klas, tworza oznaczniki punktów ciec (ang. *pointcut designators*). Na przyklad oznacznik punktu ciecia:

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

okresla wszystkie wywolania metody `deposit()` zdefiniowanej na obiektach klasy `PersonalAccount`. Oznaczenia punktów ciec moga byc tworzone zgodnie z semantyka algebry zbiorów, za pomoca operatorów *and*, *or* i *not* („&&”, „||” i „!”). W zwiazku z tym zapis:

```
pointcut transfer():
      calls( void PersonalAccount.deposit( long ) ) ||
      calls( void PersonalAccount.withdraw( long ) );
```

bedzie identyfikowal wszystkie wywolania metod `deposit()` lub `withdraw()` obiektów klasy `PersonalAccount`.
Ponizszy punkt ciecia (otrzymanie wywolanej metody) `getsInfo` oznacza wszystkie metody z przykladu dotyczacego systemu kont bankowych, zwracajace informacje o stanie konta lub jego o wlascicielu:

```
pointcut getsInfo():
      receptions( long Account.balance() )||
      receptions( PersonInfo PersonAccount.getPersonInfo() ) ||
      receptions( CompanyInfo CompanyAccount.getCompanyInfo() );
```

Powyzsze oznaczniki punktów ciec oparte byly na bezposrednim wyliczeniu sygnatur wszystkich metod (przecinanie oparte na nazwach – ang. *name-based*). W AspectJ dostepne jest tez przecinanie oparte na wlasnosciach (*property-based*), pozwalajace miedzy innymi na uzywanie znaków ogólnych przy okreslaniu sygnatur metod. Na przyklad:

```
receptions ( public * Account.*( .. ) )
```

oznacza wszystkie publiczne metody klasy `Account`.
Jednym z bardzo mocnych elementarnych oznaczen jest `cflow`, o skladni:

```
cflow( pointcut_designator )
```

Identyfikuje on wszystkie punkty zlaczen pojawiajace sie w dynamicznym obrebie (czyli wewnatrz przebiegu sterowania) punku zlaczenia okreslonego przez `pointcut_designator`. Pociaga to za soba koniecznosc sprawdzenia w czasie dzialania programu: czy aktualnie wykonywana metoda identyfikowana jest przez `pointcut_designator`?. Za pomoca `cflow` mozna przykladowo wylaczyc z tworzonego oznacznika okreslone metody rekurencyjnie wywolywane z wnetrza metody najwyzszego poziomu:

```
pointcut topLevelGetsInfo():
      getsInfo() && !cflow( getsInfo() );
```

Powyzsza definicja obejmuje wszystkie punkty laczne (powinno byc zlaczen) okreslone przez `getsInfo()`, lecz nie wywolywane z przebiegu sterowania `getsInfo()`. Do tak zdefiniowanego punktu ciecia nie nalezaloby wywolanie metody `balance()` z wnetrza metody `getPersonInfo()`.

Rady

Rady sa mechanizmami nieco podobnymi do metod. Deklaruja one kod, który ma byc wykonany po osiagnieciu punktów zlaczen w ramach oznacznika punktu ciecia. Dostepnych jest kilka typów rad. Wywolanie rady *before* ma miejsce po osiagnieciu punktu zlaczenia, ale przed dalsza czescia programu. Rada *after* wykonywana jest z kolei po zakonczeniu wykonywania metody, lecz przed zwróceniem sterowania do punktu wywolania. Rada *around* jest wywolywana po osiagnieciu punktu zlaczenia i otrzymuje bezposrednia kontrole nad tym, czy dozwolone jest wykonywanie metody punktu zlaczenia. Oprócz powyzszych rad istnieja dwa specjalne przypadki rady *after*: *after returning* i *after throwing*, których wywolanie zalezy od sposobu powrotu sterowania z punktu zlaczenia. Odwolujac sie ponownie do naszego przykladu, rada:

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

bedzie wywolana po kazdej zmianie stanu konta.
Oznaczniki punktów ciec moga miec dostep do kontekstu wykonania. Wartosci przekazane przez oznacznik mozna nastepnie wykorzystac w obrebie rady:

```
pointcut withdrawing ( Account account, long amount ):
      calls( void account.withdraw( amount ) );

after( Account account, long amount): withdrawing( account, amount ) {
      System.out.println( amount + “ withdrawn from “ + account );
}
```

Aspekty

Aspekty zdefiniowano jako modularne jednostki przecinajace implementacje (powinno byc implementacje). Deklaracja aspektu podobna jest do deklaracji klasy. Moze on zawierac deklaracje punktów ciec, rad, jak i wszystkie dozwolone dla klasy deklaracje.
Ponizsza deklaracja definiuje aspekt pozwalajacy sledzic czy od ostatniego sprawdzenia zostala wykonana operacja powodujaca zmiane stanu konta.

```
aspect TransferTracking{

      static boolean flag = false;
```

```
	static boolean testAndClear(){
		boolean result = flag;
		flag = false;
		return result;
	}

	pointcut transfer():
		receptions( void Account.deposit( long ) ) ||
		receptions( void Account.withdraw( long ) );

	after(): transfer(){
		flag = true;
	}
}
```

AspectJ wspomaga tworzenie bibliotek aspektów poprzez dostarczenie mechanizmów dziedziczenia i przedefiniowywania aspektów.

Implementacja

Glównym celem implementacji jezyka aspektowego jest doprowadzenie do współdzialania w skoordynowany sposób kodu aspektowego i nieaspektowego. Proces laczenia obu kodów nazywamy tkaniem aspektów (ang. *aspect weaving*). Moze on przebiegac na wiele sposobów: za pomoca preprocesora, w czasie kompilacji, przez procesor postkompilacyjny, w czasie ladowania kodu, jako czesc maszyny wirtualnej, poprzez instrukcje czasu wykonania lub poprzez kombinacje wymienionych podejsc. Implementowanie jezyka AspectJ oparte jest na kompilatorze i dokonuje sie prawie w calosci podczas kompilacji, a w przypadku niektórych rad dodatkowo w czasie wykonania programu.
Kompilacja programu zródlowego odbywa sie w nastepujacych etapach:

- deklaracje rad kompilowane sa do standardowych metod,
- czesci programu odwolujace sie do rad przeksztalcane sa na statyczne punkty odpowiadajace dynamicznym punktom zlaczen,
- do statycznych punktów wstawiany jest dodatkowy kod wymagajacy dzialania w czasie wykonania programu.

Narzedzia

AspectJ udostepniany jest w pakiecie, w sklad którego wchodza kompilator (`ajc`), debugger (`ajdb`), generator dokumentacji (`ajdoc`) oraz rozszerzenia do srodowisk programistycznych Emacs, JBuilder i Forte. W planach sa rozszerzenia do innych srodowisk programistycznych. W kolejnych wersjach AspectJ planowane jest zwiekszenie predkosci dzialania kompilatora poprzez wprowadzenie kompilacji przyrostowej.

Wykorzystanie AspectJ w projektach

Programowanie aspektowe ma wiele zalet. Przed wykorzystaniem AspectJ w swoich projektach nalezy jednak przeanalizowac ryzyko takiej strategii, zwiazane z korzystaniem z narzedzia bedacego caly czas w fazie rozwoju i badan. AspectJ moze ulatwic pisanie kodu, jednak moze tez stac sie elementem blokujacym rozwój systemu. Korzystanie z nowych, nie sprawdzonych na duza skale technologii, bedacych caly czas w trakcie definiowania, oraz niedokladne zrozumienie zasad programowania i statusu AspectJ zawsze moze wiazac sie z niepowodzeniem przedsiewziecia. Wybrane cechy AspectJ przedstawione zostaly w wielkim skrócie. Wiecej informacji mozna znalezc na stronach AspectJ [2].
AspectJ jest oprogramowaniem typu open-source, dostepnym na zasadach Mozzila Public License.

Rozwój programowania aspektowego

Przelozenie podejscia aspektowego na jezyki programowania objawia sie w róznych projektach i wystepuje pod róznymi nazwami. Zwiazane z tematyka sa m.in. prace poswiecone: programowaniu adaptacyjnemu (ang. *adaptive programming*), filtrom kompozycyjnym (ang. *composition filters*), metaprogramowaniu (ang. *logic meta-programming*), programowaniu tematowemu (*subject-oriented programming*) i refleksji (*reflection*). Poziom zaawansowania prowadzonych prac jest rózny, poczawszy od bardzo wstepnych, a skonczywszy na bardzo zaawansowanych. Do tej drugiej grupy zaliczyc mozna badania w laboratoriach IBM nad wielowymiarowym rozdzieleniem zagadnien [3], prowadzone w laboratoriach IBM.

Musi uplynac jeszcze sporo czasu, zanim podejscie aspektowe stanie sie bardziej popularne i zyska uznanie produkujacych oprogramowanie firm. Dzisiaj implementacje jezyków aspektowych z trudnoscia opuszcza granice laboratoriów i tylko niewiele firm ryzykuje wprowadzenie ich do powszechnego uzytku. Dzieje sie tak glównie dlatego, ze metodologia programowania nie jest ani ogólnie znana, ani dopracowana. Nie ma tez pewnosci co do poprawnosci dzialania kompilatorów i stabilnosci narzedzi programistycznych, nie jest mozliwa integracji z wieloma istniejacymi srodowiskami programistycznymi. Bedziemy tez musieli poczekac na „dojrzale” implementacje dla wiekszosci jezyków, w tym dla C++. Pomimo to wydaje sie, ze programowanie aspektowe ma w sobie olbrzymi potencjal i jego spopularyzowanie jest tylko kwestia czasu.

Odnosniki

1. *MIT Technology Review*: *http://www.techreview.com/*
2. *AspectJ Website: http://aspectj.org/*
3. *MDSOC Project: http://www.research.ibm.com/hyperspace/*

O autorze

Pawel Slowikowski jest doktorantem na wydziale EAIE Akademii Górniczo-Hutniczej w Krakowie. Pracuje w grupie zajmujacej sie systemami rozproszonymi, pod kierownictwem profesora Krzysztofa Zielinskiego. Dziedzina jego zainteresowan sa systemy rozproszone, systemy obiektowe, EJB oraz zwiazane z nimi zagadnienia bezpieczenstwa.

Kontakt z autorem: ps@agh.edu.pl