



***1. Sastanak JavaSvet Zajednice***  
***14. Avgust 2004.***

# Aspektno programiranje u Javi

autor: Nemanja Kostić

Ovo predavanje ima za cilj da vas uputi u osnove AOP metodologije i trenutno najboljeg jezika koji implementira tu metodologiju – AspectJ.

Ja sam nedavno počeo da tražim spas u AOP-u jer na svim projektima na kojima sam do sada radio, bilo je mnogo koda koji je bio raštrkan na sve strane. Radio se *copy&paste* gde god se stiglo, ali ne zato što je nekog mrzelo da piše dobro strukturiran kod, već zato što je to jednostavno bio jedini način da se neke stvari urade. Održavanje takvog koda vrlo brzo postaje noćna mora.

Današnje aplikacije su postale glomazne. Distribuirane aplikacije se rade minimalno u troslojnoj arhitekturi. Sve je češći slučaj da takve aplikacije koriste petoslojnu arhitekturu zbog bolje separacije slojeva. Čak i ne-distribuirane aplikacije (ili tkz. *collocated* aplikacije) gde se kompletna aplikacija sa bazom nalazi na jednoj mašini ili na više mašina koje su fizički vidljive međusobno, se rade po istom modelu.

Ovakve aplikacije više ne mogu jednostavno da se modeluju objektno orijentisanom paradigmom. Upravo zbog činjenice da u OO paradigmi svaka klasa mora da bude svesna svoje okoline, dolazi do tih *copy&paste* delova koda u aplikacijama, ili kako se to zove *code-scattering*. Krajnji efekat je ono što svi mi najmanje želimo, a to je nemodularnost.

A šta su posledice nemodularnosti? To su: slaba mogućnost praćenja toka izvršavanja, samim tim i smanjena produktivnost, smanjeno ponovno iskorišćenje koda (*code reuse*), slab krajnji kvalitet celog sistema i na kraju teško održavanje aplikacije.

Zato je na scenu stupila nova paradigma, pod nazivom aspektno orijentisano programiranje. Svi ovi problemi se u velikoj meri rešavaju AO paradigmom. Povećava se individualnost modula, sistem se znatno lakše održava, što je za arhitekta sistema najvažnije odluke mogu kasnije da se donose a ne u početnoj fazi projektovanja sistema, povećava se *code reuse* i što je menadžerima najvažnije smanjuje se vreme isporuke gotovog proizvoda.

Pre nego krenemo dublje u AOP i AspectJ, postoji nekoliko pitanja koja verovatno svima padaju na pamet kada je AOP u pitanju. Prvo da se ta pitanja raščiste.

- da li je AOP zamena za OOP? - Ne. Aspektno orijentisana paradigma se nadograđuje na OO paradigmu. Programi ne mogu da se pišu samo u AOP-u. Osnovna, bazna logika i dalje se piše u OO paradigmi i verovatno će tako ostati i ubuduće, s obzirom da se takvom paradigmom sasvim lepo modeluju osnovne osobine sistema. Kao što OOP dodaje nove koncepte na proceduralne jezike, tako i AOP dodaje nove koncepte na OOP. Jednostavno, kako kompleksnost sistema postaje veća tako se i evolucija programskih jezika nastavlja.
- da li AOP rešava neke nove probleme? – Ne. Sve što AOP-om može da se uradi može da se uradi i OOP-om, pa i bilo kojim nižim programskim jezikom. Stvar je u tome da se AOP-om ti problemi rešavaju na mnogo jednostavniji, bolji i brži način. A to je upravo ono što se od programera traži. Da nije tako i dalje bismo programirali u assembleru.

- da li AOP može da bude opasan? – Da. AOP može da naruši enkapsulaciju klasa. Sa AOP-om možete da pristupite privatnim metodama i poljima klase. Sa AOP-om takođe možete da presretnete pozive metoda, pročitate parametre koji im se prosleđuju, da ih izmenite i potom pozovete taj metod. Takođe, sa AOP-om možete da u postojeće klase unesete nove metode i polja koje uopšte nisu postojale u prvobitnoj verziji. Otac Java James Gosling je nedavno izjavio: “AOP je motorna testera u rukama deteta”. To je tačno, ali svaka tehnologija ako se ne koristi pravilno može da bude opasna i da prouzrokuje probleme.

## ***Istorijat:***

Tokom godina, važno je pravilo da ako želimo da napravimo lako održiv sistem, moramo da indentifikujemo i razdvojimo sistemske funkcionalnosti. Ovaj postupak je nazvan razdvajanje funkcionalnosti (separation of concerns). 1972. godine je David Parnas u svom radu objasnio da se funkcionalnosti najbolje mogu razdvojiti kroz modularizaciju. U godinama koje su sledile, proučavani su načini da se funkcionalnosti razdvoje kroz modularizaciju. Metodologija koja se pokazala najuspešnijom je OOP metodologija. OOP je obezbedio moćan način da se razdvoje sistemske funkcionalnosti. Međutim, ostala je još jedna grupa funkcionalnosti koju OOP nije uspeo kvalitetno da reši. To su takozvane “rasprostranjene funkcionalnosti” (crosscutting concerns).

Nekoliko metodologija su se pojavile kao rešenje modularizacije rasprostranjenih funkcionalnosti, a to su: **generativno programiranje, meta programiranje, reflektivno programiranje, kompozicionalno filtriranje, adaptivno programiranje, subject-oriented programiranje i aspektno orijentisano programiranje**. Od svih ovih, jedino je AOP preživeo i postao opšteprihvaćeno rešenje za modularizaciju rasprostranjenih funkcionalnosti.

Najveći deo rada na AOP-u je rađen tokom godina na univerzitetima širom sveta, a među pionirima AOP spominju se *Kristina Lopez* i *Gregor Kiczales* iz Paolo Alto istraživačkog centra, koji je bio deo **Xerox** korporacije. 1996. godine Gregor je krstio ovu metodologiju sa nazivom aspektno orijentisano programiranje. Krajem devedesetih Gregor je sa timom ljudi napravio prvu implementaciju AOP-a, a to je AspectJ. Nedavno je **Xerox** prebacio AspectJ na Eclipse open source zajednicu koja će nastaviti da unapređuje projekat. Jedan od glavnih programera sada je Adrian Colyer.

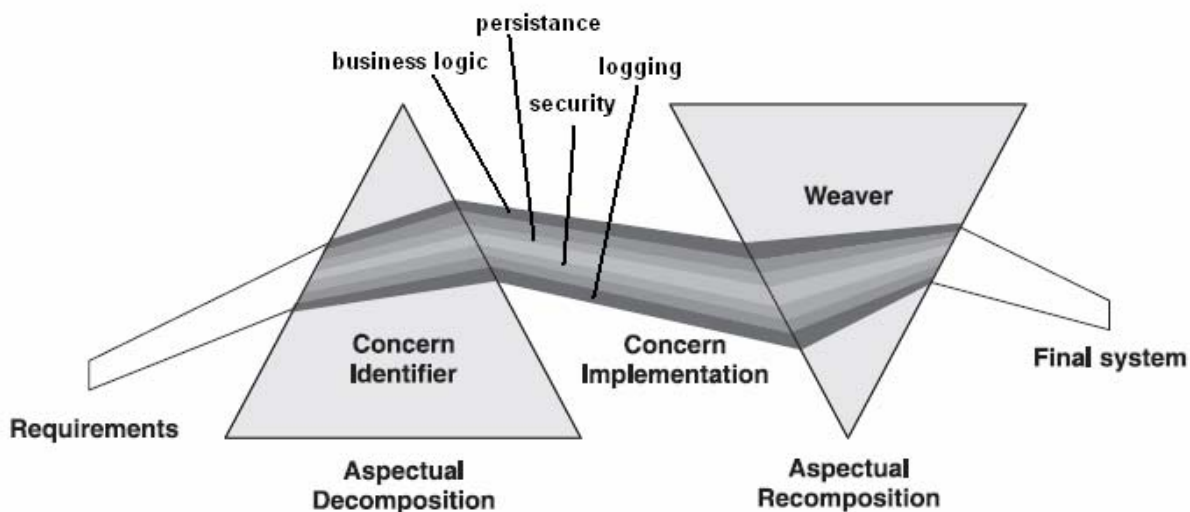
Pored AspectJ-a postoji još nekoliko implementacija AOP metodologije, a to su: JBoss AOP, AspectWerkz, Nanning, HyperJ, kao i slične implementacije za druge jezike poput AspectC++, AspectC# i još nekoliko drugih. AspectJ se svojim kvalitetom izdigao daleko iznad konkurencije.

## ***Sistemske funkcionalnosti:***

Svaki softverski sistem je sačinjen od skupa funkcionalnosti. U bankarskom sistemu, na primer, funkcionalnosti su menadžment klijenata, menadžment računa, računanje kamatnih stopa, međubankarske transakcije itd. Ovo su bazne funkcionalnosti koje implementiraju biznis logiku.

Pored njih postoje i rasprostranjene funkcionalnosti. U tipičnoj enterprise aplikaciji (kao što je bankarski sistem), rasprostranjene funkcionalnosti su: autentikacija, logovanje poruka, resource pooling, perzistencija podataka, sigurnost, multithreading, provera grešaka, keširanje i tako dalje. Sve ove rasprostranjene funkcionalnosti se protežu na nekoliko modula i čine da nam ti moduli međusobno postanu zavisni i samim tim ne mogu da budu trivijalno iskorišćeni ponovo.

Postoji jedna jako lepa analogija AOP-a i svetlosne prizme koja razlaže ulazni zrak na spektralne komponente. Upravo je u tome i suština AOP-a.



Kako zapravo radi AOP možemo da pogledamo najbolje na jednom primeru.  
U mnogim delovima sistema zateći ćemo klase poput ove:

```
public class NekaKlasa extends NekaDrugaKlasa {  
    ... otvaranje logger-a  
  
    public void NekaMetoda(...) {  
        ... autorizacija  
  
        ... lock-ovanje objekta da bi se obezbedio thread safety  
  
        ... azuriranje cache-a  
  
        ... logovanje starta operacije  
  
        ... sama operacija  
  
        ... logovanje zavrsetka operacije  
  
        ... unlock objekta  
    }  
}
```

Ako je sama operacija koju izvršava NekaMetoda nešto na primer vezano za proveru validnosti kreditne kartice, ta operacija nema nikakve veza sa logovanjem, zaključavanjem objekata i keširanjem. Te funkcionalnosti ne treba da budu tu, jer to narušava modularnost. Zamislimo samo da želimo da promenimo način logovanja, umesto Log4J da koristimo Java Logging API. Tada bi morali u svim klasama u kojima se vrši logovanje da izvršimo promene.

U krajnjem slučaju, developeri zaduženi za implementaciju algoritma za validaciju kreditne karitce, uopšte ne treba da vode računa da li se i kako nešto loguje, kešira, zaključava i slično. Sve su to rasprostranjene funkcionalnosti koje treba da se modeluju aspektima aspektno-orijentisane paradigme.

Sa AOP-om, gornja klasa postaje:

```
public class NekaKlasa extends NekaDrugaKlasa {  
  
    public void NekaMetoda(...) {  
  
        ... sama operacija  
    }  
}
```

Kada se na ovu klasu primeni aspekt za logovanje, nakon kompajliranja klasa dobija izgled:

```
public class NekaKlasa extends NekaDrugaKlasa {  
  
    Logger _logger = ...  
  
    public void NekaMetoda(...) {  
  
        _logger.log ("Zapocinjemo validaciju kreditne kartice");  
  
        ... sama operacija  
  
        _logger.log ("Validacija kreditne kartice je završena");  
    }  
}
```

AspectJ kompajler u postupku koji se naziva WEAVING (spajanje) spaja klase i aspekte koji se primenjuju na te klase i opet je rezultat klasa gore prikazanog oblika.

Weaver radi na nivou bajt koda, što je vrlo značajno jer se aspekti mogu primeniti na gotov sistem za koji nemamo sors kod, vec samo JAR fajl na primer.

Pre nego pređemo na konkretne primere korišćenja AspectJ-a, treba objasniti nekoliko izraza koji se koriste u aspektno orijentisanoj paradigmi.

A to su:

- joinpoint (tačka spajanja) - bilo koja tačka u programu koja može nedvosmisleno da se definiše je joinpoint. Recimo, neke od joinpoint-ova su: izvršavanje metode, promena vrednosti nekom parametru, inicijalizacija promenljivih, trenutak bacanja izuzetaka itd. Joinpoint je samo naziv za tačke u kojima mogu da deluju aspekti, nije nikakva ključna reč.
- pointcut – je skup jointpoint-ova. Pointcut je za razliku od joinpoint-a ključna reč, i kazuje AspectJ kompajleru na koje joinpoint-e da primeni odgovarajuću radnju.
- advice – je ta odgovarajuća radnja koja se primenjuje na pointcut-ove. To je fizičko parče koda koje treba da se izvrši kada se dođe do određenog pointcut-a. To je biznis logika samog aspekta. Postoje 3 vrste advice-a: before, after i around advice. Before advice se izvršava pre izvršenja pointcut-a, after nakon, a around i pre i posle.

- introduction – je instrukcija kojoj se dinamički uvodi novo ponašanje neke klase. Recimo, može da se dinamički dodeli nekoj klasi da implementira neki interfejs ili da se doda novi parametar ili cela metoda u neku klasu.
- compile-time declaration – instrukcija kojom se izdaju greške ili upozorenja u toku samog kompajliranja.
- aspect – aspekt je centralna jedinica obrade u AspectJ-u, kao što je klasa centralna jedinica u Javi. Unutar aspekta se definišu sve gore navedene instrukcije.

Kako to sve izgleda zajedno videćemo na jednom Hello World primeru.

```
public class MessageWriter {

    public static void writeOut ( String message ) {
        System.out.println ( message );
    }

    public static void writeOut ( String person, String message ) {
        System.out.println ( person + “ , ” + message );
    }

    public static void main ( String args[] ) {
        MessageWriter.writeOut( “AspectJ Hello World!” );
        MessageWriter.writeOut( “Kosticu”, “how are you?” );
    }
}
```

Svaki Java program može da se kompajlira i AspectJ kompajlerom (koji se naziva **ajc**), s obzirom da je **ajc** u stvari **javac** sa dodatnim funkcijama.

Izlaz bi ovako izgledao:

- ajc MessageWriter.java
- java MessageWriter

```
AspectJ Hello World!
Kosticu, how are you?
```

Bez menjanja i jedne linije koda u našoj MessageWriter klasi, proširićemo njenu funkcionalnost korišćenjem AOP-a.

Napravićemo aspekt koji će ispred svake linije koda koja se ispisuje, da ispiše još i “output is:”. Nazvaćemo taj aspekt OutputAspect, i on izgleda ovako:

```
public aspect OutputAspect {  
  
    pointcut writeOutMessage() : call (* MessageWriter.writeOut(...));  
  
    before () : writeOutMessage() {  
        System.out.println(“output is:”);  
    }  
}
```

Sada bi izlaz ovako izgledao:

- ajc MessageWriter.java OutputAspect.java
- java MessageWriter

```
output is: AspectJ Hello World!  
output is: Kosticu, how are you?
```

Dodaćemo još jedan aspekt da bi videli kako se može sam kontekst menjati. Recimo, dodaćemo jedno Mr. ispred prezimena. Nazvaćemo taj aspekt kao MannersAspect.

```
public aspect MannersAspect {  
  
    pointcut maleTitle( String person ) :  
        call ( * MessageWriter.writeOut(String, String) )  
        && args ( person, String );  
  
    void around (String person) : maleTitle (person) {  
        proceed ( “Mr.” + person );  
    }  
}
```

Kratko objašnjenje: args(person, String) govori da se samo prvi argument hvata iz konteksta i prosleđuje se kao parametar pointcut-u maleTitle, pod nazivom person. Drugi parametar nije bitan, ali mora biti tipa String.

Kada menjamo kontekst uvek moramo da koristimo around advice jer jedino on u sebi ima mogućnost da pozove *proceed* instrukciju koja poziva originalnu metodu sa promenjenim parametrom.

Sada će izlaz ovako izgledati:

- ajc MessageWriter.java OutputAspect.java MannersAspect.java
- java MessageWriter

output is: AspectJ Hello World!

output is: Mr. Kosticu, how are you?

Dakle, kao što vidimo aspekti su vrlo slični klasama. Mogu da imaju svoje parametre i metode, mogu da imaju opseg vidljivosti (`public`, `private`, `protected`, `packaged`), mogu biti apstraktni, mogu da nasleđuju druge aspekte, aspekti takođe mogu i da nasleđuju i implementiraju Javine klase ali to nije uobičajeno i aspekti mogu biti embedovani unutar klasa kao unutrašnji aspekti.

Ono po čemu se aspekti razlikuju od klasa je u sledećem: aspekti ne mogu direktno biti instancirani (nema ključne reči *new*) već se oni instanciraju na nivou virtuelne mašine. Po defaultu, samo jedan aspekt po VM se instancira. Postoje načini i da se aspekti instanciraju po svakom objektu na koje se ti aspekti odnose. Iako aspekti mogu da nasleđuju apstraktne aspekte, ne mogu da nasleđuju konkretne aspekte (ovo je uvedeno zbog smanjenja kompleksnosti). Aspekti mogu biti markirani kao privilegovani, i to im daje pristup privatnim članovima klasa.

Toliko o teoriji. Sada krećemo na prave primere. Do sada je pronađeno dosta funkcionalnosti na koje AOP može da se primeni. Neke od njih ćemo sada pomenuti.

Ono što je bitno napomenuti je da sve što je do sada rečeno je manje više sve što ima da se kaže o aspektno orijentisanoj paradigmi. Dalje preostaje samo da se pronađu njene konkretne primene. AOP je relativno nova metodologija i prema tome niko nije ekspert u ovoj oblasti. AspectJ sintaksa nije trivijalna i zahteva malo učenja pre nego postane potpuno razumljiva.

Primene AOP će svakim danom biti sve izraženije i svako od nas će vremenom u radu na projektima da primećuje delove koda koji se ponavljaju. Tada možete početi da primenjujete AOP da biste poboljšali projekat na kome radite.

## ***Primeri primene AOP-a korišćenjem AspectJ jezika:***

Primene AOP-a su mnogobrojne i još uvek se istražuje gde bi još mogao da se primeni. Mi ćemo pomenuti samo neke od niza postojećih:

- 1. Logging and debugging**
- 2. Policy enforcement**
- 3. Resource pooling**
- 4. Caching**
- 5. Wormhole**
- 6. Exception introduction**
- 7. Thread safety**
- 8. Autentikacija i autorizacija**
- 9. Transaction management**
- 10. Implementacija biznis pravila – integracija sa rule engine-ima**

### **Logging and debugging**

Može se logovati ulazak/izlazak iz metoda, klasa, izvršavanje odgovarajućeg koda itd. Upotreba je jednostavna i nećemo se zadržavati na ovome, a već je delimično i pomenuto u gornjem primeru.

### **Policy enforcement**

Služi za efikasnu kontrolu ispravnosti koda. Na primer, može se evidentirati u toku kompajliranja da li se koristi System.out ili System.error umesto nekog log mehanizma.

```
aspect DetectSystemOutErrorUsage {  
  
    declare warning : get( * System.out ) || get( * System.err )  
        : “Consider using Logger.log() instead System.out or System.err.”  
}
```

Ukoliko se u kodu nađe na System.out ili System.err javiće se upozorenje prilikom kompajliranja.

Na ovaj način može da se kontroliše i pristup nekim klasama. Recimo da imamo klasu `ShoppingCart` u kojoj se nalaze metode za dodavanje i izbacivanje sadržaja u korpu. Takođe, recimo da imamo klasu `ShoppingCartOperator` kroz koju treba da se pozivaju metode za ubacivanje i izbacivanje sadržaja iz korpe, jer se jedino kroz nju recimo to vrši pravilno uz ažuriranje još nekih parametara.

```
public aspect ShoppingCartAccessAspect {  
  
    declare error :  
        (call (* ShoppingCart.add*(...))  
        || call (* ShoppingCart.remove*(...)))  
        && !within(ShoppingCartOperator) :  
        “Illegal manipulation to ShoppingCart! Only ShoppingCartOperator  
        may perform such operations”;  
  
}
```

Sa ovakvim aspektom, ubacivanje/izbacivanje iz korpe van `ShoppingCartOperator` klase neće ni biti moguće jer se kompajliranje neće izvršiti do kraja.

Na ovaj način može se slediti i dobar programerski model – ne dozvoliti postojanje `public` polja u klasama.

```
aspect DetectPublicAccessMembers {  
  
    declare warning :  
        get(public !final * *) || set (public * *) :  
        “Consider using nonpublic access to member variables”;  
  
}
```

## Resource pooling

Danas bi svaki softverski sistem trebalo da ima pooling onih resursa čije kreiranje je jako skupo. To su na primer resursi poput JDBC konekcije ka bazi i recimo kreiranje novog Thread-a.

Ukoliko u sistemu nije implementiran pooling ovih resursa, taj nedostatak se vrlo lako može nadoknaditi AOP-om. Zamislimo da imamo sistem koji radi sa bazom preko JDBC drajvera i za pristup bazi koristi Connection objekat. Prikazaćemo kako izgleda aspekt koji na postojeći sistem koji ne koristi connection pooling, nadograđuje takvu funkcionalnost.

```
import java.sql.*;

public aspect DBConnectionPoolingAspect {
    // Kreiranje resource pool-a
    DBConnectionPool _connPool = new SimpleDBConnectionPool();

    // Pointcut koji definiše trenutak kreiranja nove konekcije
    pointcut connectionCreation(String url, String username, String password)
        : call (public static Connection DriverManager.getConnection(String, String, String))
          && args(url, username, password);

    // Pointcut koji definiše trenutak oslobađanja stare konekcije
    pointcut connectionRelease(Connection connection)
        : call (public void Connection close())
          && target(connection);

    Connection around(String url, String username, String password) throws SQLException
        : connectionCreation(url, userName, password) {

        Connection connection = connPool.getConnection(url, userName, password);
        if(connection == null){
            connection.proceed(url, userName, password);
            _connPool.registerConnection(connection, url, userName, password);
        }

        return connection;
    }

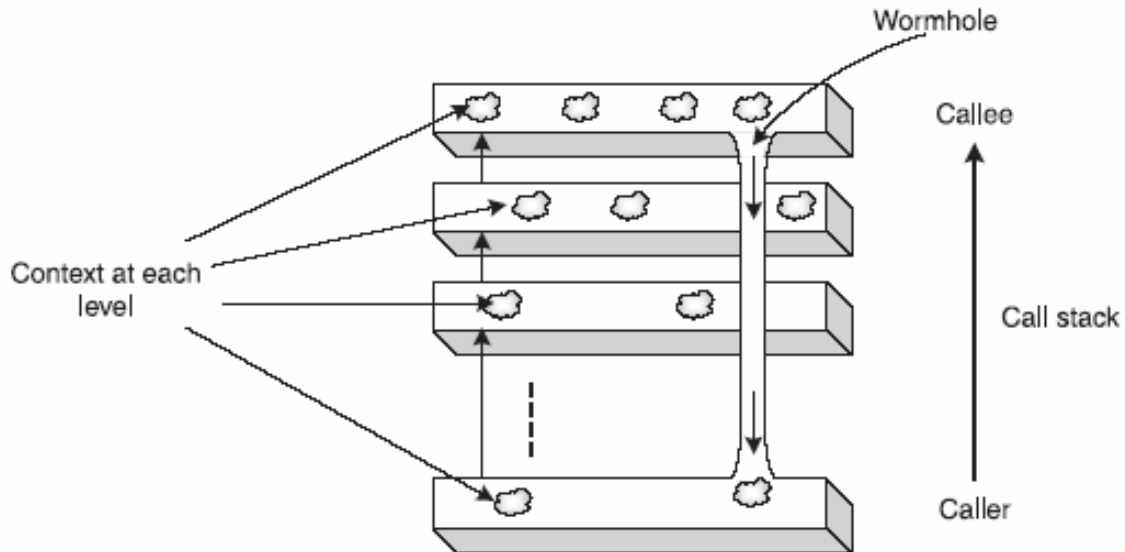
    void around(Connection connection) : connectionRelease(connection) {
        if(!_connPool.putConnection(connection)) {
            proceed(connection);
        }
    }
}
```

Na sličan način može da se uradi i pooling Thread-ova na primer.

## Caching

Na sličan način kao i Resource pooling

## Wormhole



**Figure 8.1** The wormhole pattern. Each horizontal bar shows a level in the call. The wormhole makes the object in the caller plane available to the methods in the called plane without passing the object through the call stack.

Wormhole patern je izuzetno koristan pri transferu parametara između dva objekta. Naime, čest je slučaj da se u toku izvršavanja parametri moraju prosleđivati od jednog objekta (pozivaoca) do nekog drugog (pozvanog) kroz nekoliko slojeva – primer komunikacija između threadova. Da bi se izbeglo ponavljanje parametara, koristi se wormhole AOP patern gde se direktno kontekst prosleđuje sa jednog na drugi objekat.

Ostale primene AOP su jos zanimljivije i korisnije, ali bi nam trebalo daleko više vremena za detaljno objašnjavanje svakog.

## ***Pogled u budućnost:***

Adrian Colyer je najavio uskoro podršku za metapodatke u AspectJ jeziku. To će biti verzija za Java Tiger 5.0, koja bi se za par meseci već mogla pojaviti. Metapodaci u saradnji sa AOP-om će doneti tek revoluciju u programiranju. Trenutno samo JBossAOP i AspectWerkz podržavaju metapodatke.

Ono što je svakako zanimljivo jeste da će metapodaci moći da se specificiraju kao joinpoint-ovi. Tada ćemo moći da napišemo našu klasu na sledeći način:

```
.....  
  
@ dugotrajna_operacija  
public void NekeMetoda() {  
  
    .... neka duga obrada ...  
  
}
```

Na ovaj način smo metapodatkom `@ dugotrajna_operacija` obeležili one operacije za čije izvršavanje je potrebno vreme.

Uz pomoć sledećeg aspekta, kad god se naiđe na ovaj metapodatak kreiraće se jedan novi Thread za tu operaciju.

```
public aspect LongOperationAspect {  
  
    public pointcut longOperation() : call (@dugotrajna_operacija);  
  
    void around() : longOperation() {  
  
        Runnable worker = new Runnable() {  
            public void run() {  
                proceed();  
            }  
        }  
  
        Thread workerThread = new Thread(worker);  
        workerThread.start();  
    }  
  
}
```

Zamislimo samo kakve sve ovo može da ima primene. Potpuno razdvajanje programerskih uloga. Programer koji piše biznis logiku ne mora više da brine o načinu upisa u bazu. Ne interesuje ga koja je baza u pitanju, kako se otvara konekcija, ne interesuje ga uopšte SQL. Samo na mestima na kojima nešto želi da upiše u bazu treba da navede metapodatak:

```
@Upis_U_Bazu("Account", "name", "Nemanja" )
```

Kako će to biti upisano u bazu, njega ne interesuje. AspectJ će da pokupi metapodatak i njegove parametre i da odradi posao.

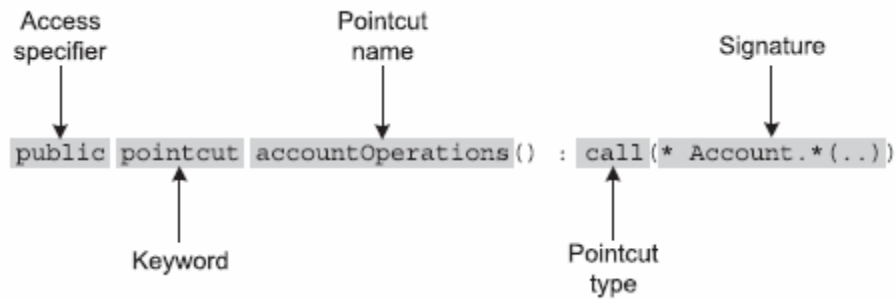
## *Reference:*

1. [www.aspectprogrammer.org](http://www.aspectprogrammer.org) – sajt Adrian Colyera
2. “AspectJ in Action” – Ramnivas Laddad, Manning
3. [www.aosd.net](http://www.aosd.net) - zvaničan sajt AOP-a
4. [www.eclipse.org/aspectj](http://www.eclipse.org/aspectj) - AspectJ
5. [www.eclipse.org/ajdt](http://www.eclipse.org/ajdt) - AspectJ IDE plugin za Eclipse

**KRAJ**

## Appendix: AspectJ sintaksa

- pointcut sintaksa:



Three wildcard notations are available in AspectJ:

- \* denotes any number of characters except the period.
- .. denotes any number of characters including any number of periods.
- + denotes any subclass or subinterface of a given type.

**Table 3.1 Examples of type signatures**

Signature Pattern	Matched Types
<code>Account</code>	Type of name <code>Account</code> .
<code>*Account</code>	Types with a name ending with <code>Account</code> such as <code>SavingsAccount</code> and <code>CheckingAccount</code> .
<code>java.*.Date</code>	Type <code>Date</code> in any of the direct subpackages of the <code>java</code> package, such as <code>java.util.Date</code> and <code>java.sql.Date</code> .
<code>java..*</code>	Any type inside the <code>java</code> package or all of its direct subpackages, such as <code>java.awt</code> and <code>java.util</code> , as well as indirect subpackages, such as <code>java.awt.event</code> and <code>java.util.logging</code> .
<code>javax..*Model+</code>	All the types in the <code>javax</code> package or its direct and indirect subpackages that have a name ending in <code>Model</code> and their subtypes. This signature would match <code>TableModel</code> , <code>TreeModel</code> , and so forth, and all their subtypes.

**Table 3.2 Examples of a combined type signature using unary and binary operators**

Signature Pattern	Matched Types
<code>!Vector</code>	All types other than <code>Vector</code> .
<code>Vector    Hashtable</code>	<code>Vector</code> or <code>Hashtable</code> type.
<code>javax..*Model    javax.swing.text.Document</code>	All types in the <code>javax</code> package or its direct and indirect subpackages that have a name ending with <code>Model</code> or <code>javax.swing.text.Document</code> .
<code>java.util.RandomAccess+ &amp;&amp; java.util.List+</code>	All types that implement both the specified interfaces. This signature, for example, will match <code>java.util.ArrayList</code> since it implements both the interfaces.

**Table 3.3 Examples of method signatures**

Signature Pattern	Matched Methods
<code>public void Collection.clear()</code>	The method <code>clear()</code> in the <code>Collection</code> class that has public access, returns <code>void</code> , and takes no arguments.
<code>public void Account.debit(float) throws InsufficientBalanceException</code>	The public method <code>debit()</code> in the <code>Account</code> class that returns <code>void</code> , takes a single <code>float</code> argument, and declares that it can throw <code>InsufficientBalanceException</code> .

**Table 3.3** Examples of method signatures (continued)

Signature Pattern	Matched Methods
<code>public void Account.set*(*)</code>	All public methods in the <code>Account</code> class with a name starting with <code>set</code> and taking a single argument of any type.
<code>public void Account.*()</code>	All public methods in the <code>Account</code> class that return <code>void</code> and take no arguments.
<code>public * Account.*()</code>	All public methods in the <code>Account</code> class that take no arguments and return any type.
<code>public * Account.*(..)</code>	All public methods in the <code>Account</code> class taking any number and type of arguments.
<code>* Account.*(..)</code>	All methods in the <code>Account</code> class. This will even match methods with <code>private</code> access.
<code>!public * Account.*(..)</code>	All methods with nonpublic access in the <code>Account</code> class. This will match the methods with <code>private</code> , <code>default</code> , and <code>protected</code> access.
<code>public static void Test.main(String[] args)</code>	The static <code>main()</code> method of a <code>Test</code> class with public access.
<code>* Account+.*(..)</code>	All methods in the <code>Account</code> class or its subclasses. This will match any new method introduced in <code>Account</code> 's subclasses.
<code>* java.io.Reader.read(..)</code>	Any <code>read()</code> method in the <code>Reader</code> class irrespective of type and number of arguments to the method. In this case, it will match <code>read()</code> , <code>read(char[])</code> , and <code>read(char[], int, int)</code> .
<code>* java.io.Reader.read(char[], ..)</code>	Any <code>read()</code> method in the <code>Reader</code> class irrespective of type and number of arguments to the method as long as the first argument type is <code>char[]</code> . In this case, it will match <code>read(char[])</code> and <code>read(char[], int, int)</code> , but not <code>read()</code> .
<code>* javax.*.add*Listener(EventListener+)</code>	Any method whose name starts with <code>add</code> and ends in <code>Listener</code> in the <code>javax</code> package or any of the direct and indirect subpackages that take one argument of type <code>EventListener</code> or its subtype. For example, it will match <code>TableModel.addTableModelListener(TableModelListener)</code> .
<code>* *.*(..) throws RemoteException</code>	Any method that declares it can throw <code>RemoteException</code> .

**Table 3.4 Examples of constructor signatures**

Signature Pattern	Matched Constructors
<code>public Account.new()</code>	A public constructor of the <code>Account</code> class taking no arguments.
<code>public Account.new(int)</code>	A public constructor of the <code>Account</code> class taking a single integer argument.
<code>public Account.new(...)</code>	All public constructors of the <code>Account</code> class taking any number and type of arguments.
<code>public Account+.new(...)</code>	Any public constructor of the <code>Account</code> class or its subclasses.
<code>public *Account.new(...)</code>	Any public constructor of classes with names ending with <code>Account</code> . This will match all the public constructors of the <code>SavingsAccount</code> and <code>CheckingAccount</code> classes.
<code>public Account.new(...) throws InvalidAccountNumberException</code>	Any public constructors of the <code>Account</code> class that declare they can throw <code>InvalidAccountNumberException</code> .

**Table 3.6 Mapping of exposed join points to pointcut designators**

Join Point Category	Pointcut Syntax
Method execution	<code>execution(MethodSignature)</code>
Method call	<code>call(MethodSignature)</code>
Constructor execution	<code>execution(ConstructorSignature)</code>
Constructor call	<code>call(ConstructorSignature)</code>
Class initialization	<code>staticinitialization(TypeSignature)</code>
Field read access	<code>get(FieldSignature)</code>
Field write access	<code>set(FieldSignature)</code>
Exception handler execution	<code>handler(TypeSignature)</code>
Object initialization	<code>initialization(ConstructorSignature)</code>
Object pre-initialization	<code>preinitialization(ConstructorSignature)</code>
Advice execution	<code>adviceexecution()</code>

**Table 3.7 Examples of control-flow based pointcuts**

Pointcut	Description
<code>cflow(call(* Account.debit(..))</code>	All the join points in the control flow of any <code>debit()</code> method in <code>Account</code> that is called, including the call to the <code>debit()</code> method itself
<code>cflowbelow(call(* Account.debit(..))</code>	All the join points in the control flow of any <code>debit()</code> method in <code>Account</code> that is called, but excluding the call to the <code>debit()</code> method itself
<code>cflow(transactedOperations())</code>	All the join points in the control flow of the join points captured by the <code>transactedOperations()</code> pointcut
<code>cflowbelow(execution(Account.new(..))</code>	All the join points in the control flow of any of the <code>Account</code> 's constructor execution, excluding the constructor execution itself
<code>cflow(staticinitializer(BankingDatabase))</code>	All the join points in the control flow occurring during the class initialization of the <code>BankingDatabase</code> class

**Table 3.8 Examples of lexical-structure based pointcuts**

Pointcut	Natural Language Description
<code>within(Account)</code>	Any join point inside the <code>Account</code> class's lexical scope
<code>within(Account+)</code>	Any join point inside the lexical scope of the <code>Account</code> class and its subclasses
<code>withincode(* Account.debit(..))</code>	Any join point inside the lexical scope of any <code>debit()</code> method of the <code>Account</code> class
<code>withincode(* *Account.getBalance(..))</code>	Any join point inside the lexical scope of the <code>getBalance()</code> method in classes whose name ends in <code>Account</code>

One common usage of the `within()` pointcut is to exclude the join points in the aspect itself. For example, the following pointcut excludes the join points corresponding to the calls to all `print` methods in the `java.io.PrintStream` class that occur inside the `TraceAspect` itself:

```
call(* java.io.PrintStream.print*(..)) && !within(TraceAspect)
```

**Table 3.9 Examples of execution object pointcuts**

Pointcut	Natural Language Description
<code>this(Account)</code>	All join points where <code>this</code> is <code>instanceof Account</code> . This will match all join points like methods calls and field assignments where the current execution object is <code>Account</code> , or its subclass, for example, <code>SavingsAccount</code> .
<code>target(Account)</code>	All the join points where the object on which the method called is <code>instanceof Account</code> . This will match all join points where the target object is <code>Account</code> , or its subclass, for example, <code>SavingsAccount</code> .

**Table 3.10 Examples of argument pointcuts**

Pointcut	Natural Language Description
<code>args(String, ..., int)</code>	All the join points in all methods where the first argument is of type <code>String</code> and the last argument is of type <code>int</code> .
<code>args(RemoteException)</code>	All the join points with a single argument of type <code>RemoteException</code> . It would match a method taking a single <code>RemoteException</code> argument, a field write access setting a value of type <code>RemoteException</code> , or an exception handler of type <code>RemoteException</code> .

**Table 3.11 Examples of conditional check pointcuts**

Pointcut	Natural Language Description
<code>if(System.currentTimeMillis() &gt; triggerTime)</code>	All the join points occurring after the current time has crossed the <code>triggerTime</code> value.
<code>if(circle.getRadius() &lt; 5)</code>	All the join points where the <code>circle</code> 's radius is smaller than 5. The <code>circle</code> object must be a context collected by the other parts of the pointcut. See section 3.2.6 for details about the context-collection mechanism.

- advice sintaksa:

```

pointcut> creditOperation(Account account, float amount ) :
    call (void Account.credit(float))
    && target ( account )
    && args ( amount );

before (Account account, float amount ):
    creditOperation(account, amount) {
        System.out.println("Crediting " + amount
            + " to " + account );
    }

```

**Figure 3.5** Passing an executing object and an argument captured by a named pointcut. This code snippet is functionally equivalent to figure 3.4, but achieves it using a named pointcut. For the advice to access the join point's context, the pointcut itself must collect the context, as opposed to the advice collecting the context when using anonymous pointcuts.

```

after() returning(Connection conn ) :
    call(Connection DriverManager.getConnection(..)) {
        System.out.println("Obtained database connection: "
            + conn );
    }

```

Passing return object

**Figure 3.6** Passing a return object context to an advice body. The return object is captured in returning() by specifying the type and object ID.

```

after() throwing (RemoteException ex )
: call(* *.*(..) throws RemoteException) {
    System.out.println("Exception " + ex
        + " while executing "
        + thisJoinPoint );
}

```

Accessing special variable

Passing exception object

**Figure 3.7** Passing a thrown exception to an advice body. The exception object is captured in throwing() by specifying the type and object ID. The special variables such as thisJoinPoint are accessed in a similar manner to this inside an instance method.