Aspektler

Aspects, Concerns, and Java

James Cooper - Aspects, Concerns, and Java

file:///F:\Belgelerim\Yayınlar\Java\Makaleler\Dizayn\Aspect\Aspects%20Concerns%20and%20Java.doc

Program ne kadar iyi tasarlanmış olursa olsun, nesne odaklı programlamanın mantıksal sınırlarından dolayı, birçok metot tekrar tekrar aynı şekilde çağrılmak zorundadır.

Mesela çizim programı. Çizgi, kare, daire nesnelerin var. Bunların herbirinin çiz() metodu var. Bu çiz metodunun içinde kendilerini ekrana çiziyorlar ve sonunda da ekranın update() metodunu çağırıyorlar. update() metodu bütün sınıflarda aynı şekilde çağrılıyor.

Bu çağırma şeklini ayrı bir modülde tutabilseydik iyi olurdu.

AspectJ terminolojisinde, modül halinde ayırmak istediğin noktaların her birine bir **mesele (concern)** denir. Bu modüllerin ayrılması işlemine de **çapraz kesme (crosscutting)** denir. Birleştirebildiğin belirli noktalara, **birleştirme noktaları (join point)** denir. Bunlar şu tipten bir şey olabilir:

method calls or constructor calls (they happen before the object is called); method or constructor execution; field get or set; exception handler execution; class initialization (when static initializers are called); and object initialization (when dynamic initializers are called)

Birleştirme noktalarını, **kesik (pointcut)** cümleleriyle tanımlarsın. Mesela aşağıda, Line sınıfının setP1 ve setP2 metotlarının çağrıldığı her noktayı birleştirmek istiyoruz.

```
pointcut move();  //names the spot
   call (void Line.setP1(Point))
      //where this call occurs
||  //OR
   call (void Line.setP2(Point));
      //where this call occurs
```

Bu kesiğe move ismini verdik.

AspectJ ile **tavsiyeler (advice)** de belirtebilirsin. Bu, bir koddan önce, sonra, beraber, dönmenin ardından, hata fırlatmanın ardından çalıştırılacak kod anlamına gelir. Tavsiyeler özel bir sınıf türü olan aspektlerde tanımlanır. Bir aspekt, başka sınıfları kesebilen sınıfa verilen addır.

Buradaki tavsiyede, bir move kesiği çağrıldığında ne yapılacağını belirtiyoruz:

```
aspect MoveTracking {

pointcut move() :
   call (void Line.setP1(Point)) ||
   call (void Line.setP2(Point)) ;

//code runs after each call
after() returning move() {
   screen.update();
}
}
```

Bir aspekt birçok sınıfı kesebilir.

```
aspect DisplayUpdating {

   pointcut move():
      call(void FigureElement.
         moveBy(int, int)) ||
      call(void Line.setP1(
         Point))              ||
      call(void Line.setP2(
         Point))              ||
      call(void Point.setX(
         int))                ||
```

```
      call(void Point.setY(int));

   after() returning: move() {
      Display.update();
   }
}
```

Burada move kesiği, birçok farklı sınıfın metotlarının çağrılmasıdır.

```
aspect SimpleErrorLogging {

   Log log = new Log();
      //some logger

   pointcut publicEntries():
      receptions(public *
      com.yourcom.printers.*.*
      (..));

   after() throwing (Error e):
      publicEntries() {
      log.write("stuff");
}
}
```

Bu kodun yaptığı şu: com.yourcom.printers paketindeki herhangi bir sınıfın herhangi bir public metodu eğer hata fırlatılırsa, bununla ilgili log burada alınır.

Aspect-Oriented Programming with Sun ONE Studio

Aspect-Oriented Programming with Sun ONE Studio - Vaughn Spurlin
file:///F:\Belgelerim\Yayınlar\Java\Makaleler\Dizayn\Aspect\Aspect-Oriented%20Programming%20with%20Sun%20ONE%20Studio.doc

Birleştirme noktası (join point):
- call - Bir metodun çağrıldığı nokta
- execution - Çağrılmış bir metodun başladığı nokta
- set - Private olmayan bir fielde değer atandığı nokta

Kesik (pointcut)
- birleştirme noktalarını tanımlayan bir yüklem

Tavsiye beyanı (advice declaration)
- Bir tavsiye tipi, arkasından kesik ve arkasından çalıştırılabilir bir tavsiye gövdesi. Eğer java programı kesiğe gelirse, bu tavsiye gövdesi çalışır. Tavsiye tipi, gövdenin ne zaman çalışacağını belirtir. Eğer after tipindeyse, kesik bittikten sonra gövde çalışır.

Tipler arası üye tanımı (Inter-Type Member Declaration)
- Bir sınıfın kaynak kodunda olmayan yeni metotlar ve alanlar ekleyebiliriz.

Aspekt
- Kesikleri, tavsiye beyanlarını, tipler arası üye tanımlarını içeren geçerli java sınıflarıdır.

---

aspekt-deney-2 içinde bu projeyi bulabilirsiniz.

NetBeans entegrasyonu için: http://aspectj4netbean.sourceforge.net/index.html adresinden ek modülü indirin.

Açıklama:

file:///?Kod%20Parcalari|NetBeans-AspectJ%20entegrasyonu|974|0

```
aspect Logger {
    pointcut log():
        execution(* *.getGreeting(..)) || execution(* *.setGreeting(..));

    before() : log() {
        System.out.println("  Logger: " +
            thisJoinPoint.getSignature());
```

```
    }
}
```

Yukarıdaki kodun anlamı:

Bütün getGreeting ve setGreeting metotları çalışmalarından önce log() tavsiyesini ekle.

```
aspect Counter {
    private int HelloBean.countGetGreeting = 0;
    private int GoodbyeBean.countGetGreeting = 0;
    declare parents : GoodbyeBean extends HelloBean;

    pointcut count(HelloBean h): call(* *.getGreeting(..)) && target(h);

    after(HelloBean h) : count(h) {
        h.countGetGreeting++;
        System.out.println("  Counter: " +
            h.countGetGreeting + " calls to " +
            thisJoinPoint.getSignature());
    }
}
```

Burada countGetGreeting adlı bir alan, hem HelloBean hem GoodbyeBean sınıfları içine eklendi.

Ayrıca GoodbyeBean, HelloBeanden türetildi. declare parents : ifadesiyle...

Dikkat ederseniz burada çapraz kesen bir mesele ele alındı. Sayma (count) işlemi, farklı sınıflarda yapılması gerekiyor. Ancak saymakla hiçbir ilgisi olmayan bir sınıfta, sayma işlemini yürütmek yerine bu şekilde çok daha iyi bir modüle konulmuş oluyor.

Kesik tanımlaması:

```
    pointcut count(HelloBean h): call(* *.getGreeting(..)) && target(h);
```

Burada count metodu HelloBean h parametresini aldı. Bu bu kesiğin sadece HelloBean sınıfı ve türevlerini ilgilendirdiği söyleniyor.

Kullanım Tavsiyeleri:

Aspektlerin temel kullanımı oldukça basit. Uzmanlaştıkça çok daha kompleks imkanlar sunuyor. Belirli bir yerden sonra, daha modülerleştirmek için ne yapabilirim sorusuna yardımcı olmak için Aspect Mining Toolu kullanabilirsiniz: http://www.cs.ubc.ca/%7Ejan/amt/

Introduction to AOP

Introduction to AOP
file:///F:\Belgelerim\Yayınlar\Java\Makaleler\Dizayn\Aspect\Introduction%20to%20AOP.pdf

Meselelerin ayrıştırılmasını sağlayan bir yöntem...

Meselelerin ayrıştırılmasına yönelik prizma modeli:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\55.gif

Buna yönelik AOP dışı çözümler:
- Dinamik proxyler
- Dekoratör kalıbı
- Servlet filtreleri

Genel olarak bu üçünde de müdahele (interception) kalıbı uygulanıyor:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\56.gif

OOP nesne soyutlamaları (object abstraction) için iyi bir yaklaşım:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\57.gif

Ama çapraz kesen (cross-cutting) mesele soyutlamaları (concern abstraction) için iyi bir yaklaşım değil:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\58.gif

Niçin bu önemli?

Çünkü çapraz kesen meselelerin nesne hiyerarşilerinde modülerleştirilmesi pek kolay olmuyor. Bu da kodun karmaşıklaşmasına sebep oluyor. Bir sınıfın sorumlulukları giderek artıyor.Ve kodun dağınıklaşmasına sebep oluyor. Aynı mesele birçok yere dağılıyor.

Karmaşıklaşma:
file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\60.gif

Mesela, aynı metodun içinde hem güvenlik, hem logging, hem senkronizasyon ele alınıyor.

Dağınıklaşma:
file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\59.gif

AOP ne yapıyor?
Prizma gibi. Önce meseleleri ayrıştırıyoruz. Compile timeda aspectj compiler her şeyi birbirine dokuyor (weaving).

AspectJ ne?
- Xerox Parc laboratuvarlarında başladı.
- Grigor Kriczales yönetiminde
- Şimdi eclipse çatısında geliştiriliyor. Ama her türlü ideyle uyumlu çalışabilir (ant aracılığıyla)

Join point (birleştirme noktası) nedir?
- metot çağrısı (call) (çağırıcı tarafı)
- metot işleme (execution) (çağrılan taraf)
- değişkene değer atamak veya almak
- constructor

Pointcut (kesme noktası) nedir?
Birleştirme noktalarından oluşan bir küme. Mesela, şu metotların çağrılması, bir kesme noktasıdır, diye tanımlıyoruz.

Advice (tavsiye) nedir?
Pointcut (kesme noktalarıyla) bağlanan kod parçaları.
Pointcuta erişildiği vakit çalıştırılır.
- before: Kesme noktasından önce
- after: Kesme noktasından sonra
- around: Kesme noktasının etrafında

Toparlayalım:
Kesme noktasına vardığında çalıştırmak istediğin kodun (advice) çalıştığı yere birleşme noktası denir.

Kodla:

```
public class Foo {
   private int count;
   public Foo() {
   }
   public void sayHello() {
      System.out.println("Hello, AOP!");
      count++;
   }
   public int getCount() {
      return count;
   }
}
```

Aspekt:

```
public aspect FooAspect {
   before() : call( * sayHello(..) )
   {
      System.out.println("Before the greeting!");
   }
}
```

sayHello() metotlarının (argümanları neler olursa olsun) çağrıldığı tüm yerlere, System.out.println("Before the greeting!"); kodu dokunur.

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\61.gif

```
public static void main( String [] args ) {
   Foo f = new Foo();
   f.sayHello();
}
```

Çalıştırmak:

```
ajc -classpath .:/usr/local/aspectj1.1/lib/aspectjrt.jar *.java
java -cp .:/usr/local/aspectj1.1/lib/aspectjrt.jar Foo
Before the greeting!
Hello, AOP!
```

ajc, aspectj compilerıdır. aspekt dosyalarını alır ve diğer sınıfların içine dokur (bytecode olarak).

Yeni bir pointcut daha ekleyelim:

```
before() : execution( * sayHello(..) )
{
   System.out.println("Also before the greeting!");
}
```

```
ajc -classpath .:/usr/local/aspectj1.1/lib/aspectjrt.jar *.java
java -cp .:/usr/local/aspectj1.1/lib/aspectjrt.jar Foo
Before the greeting!
Also before the greeting.
Hello, AOP!
```

Ancak bir öncekinden bir farkı var. Buradaki birleşme noktası, execution. Yani çağrılan metodun tarafı:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\62.gif

Around advice:

Around, çağrılan metodun etrafında çalışır:

```
int around() : execution( * getCount() )
{
   System.out.println( "Before getCount()" );
   int retValue = proceed();
   System.out.println("After getCount()");
   if( retValue == 0 ) {
      System.out.println("That was the first call!" );
   }
   return retValue;
}
```

proceed() ile, bağlantı noktası çalışır.

Dikkat ederseniz, around adviceının döndürdüğü değerin tipi, int. Çünkü normal getCount() metodu yerine bu advice çalışıyor.

I want my AOP

I want my AOP

örnekler: aspect-deney-8 file:///?Kod%20Parcalari|aspect-deney-8|0|0

```
public class SomeBusinessClass extends OtherBusinessClass {
    // Core data members

    // Other data members: Log stream, data-consistency flag

    // Override methods in the base class

    public void performSomeOperation(OperationInformation info) {
        // Ensure authentication
```

```
        // Ensure info satisfies contracts

        // Lock the object to ensure data-consistency in case other
        // threads access it

        // Ensure the cache is up to date

        // Log the start of operation

        // ==== Perform the core operation ====

        // Log the completion of operation

        // Unlock the object
    }

    // More operations similar to above

    public void save(PersitanceStorage ps) {
    }

    public void load(PersitanceStorage ps) {
    }
}
```

Bağlantı noktaları (join pointler):

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\63.gif

Pointcutlar:

metot bağlantı noktalarını belirtmek için:
call
execution

alan erişim bağlantı noktalarını belirtmek için:
set - alanlar için
get

istisna yakalama bağlantı noktalarını belirtmek için:
handler(*RemoteException*) - exception yakalamak

lexikal bağlantı noktalarını belirtmek için:
Bunlar, bir sınıfın veya metodun içindeki bütün bağlantı noktalarını yakalar:
within(*MyClass*) - MyClass sınıfının içindeki bütün bağlantı noktaları
withincode(* *MyClass.myMethod(..)*) - myMethod içindeki bütün bağlantı noktaları

kontrol akışı (control flow):
cflow(*call(* MyClass.myMethod(..))* - Bu farklı bir şey. myMethod metodundan b, c metotları çağrılabilir. myMethodun kontrol akışı içindeki bütün bağlantı noktalarını bu şekilde belirtmiş oluyoruz.

within´den farkı ne? within, scopea göre tanımlanıyor. cflow, kontrole göre tanımlanıyor.

bağlama bağlı pointcutlar:
this(JComponent+) - JComponent türünden bütün nesnelere ait bağlantı noktaları
target(MyClass) - MyClass türünden nesnelerin metotlarının çağrıldığı bütün bağlantı noktaları
args(String,..,int) - İlk argümanın String, son argümanın int tipinden olduğu bütün metot çağırma bağlantı noktaları
args(RemoteException) - Bütün RemoteException yakalayan try-catchler.

koşullu pointcutlar:
if(EventQueue.isDispatchThread()) - EventQueue.isDispatchThread() true olduğu bütün bağlantı noktaları.

Pointcutlar ya isimlendirilmiştir, ya da anonimdir. İsimlendirilmişse, farklı advicelardan bunlara başvurulabilir. İsimsizse, anonim sınıflar gibi sadece kullanıldıkları yerde başvurulabilir.

Farklı pointcutları birleştirmek için, &&, ||, ! işaretleri kullanılabilir.

Bağlama ulaşmak:

thisJoinPoint adlı nesneyle refleksiyonu kullanarak bağlama ilişkin bilgi toplanır. Mesela logging için bu kullanılır.

```
pointcut publicOperationCardAmountArgs(CreditCard card, Money amount):
        execution(public * CreditCardProcessor.*(..)) && args(card, amount);
```

Burada args(card, amount) deyimiyle, bağlantı noktalarındaki metotların CreditCard ve Money argümanlarını tanımlamış olduk. Daha sonra adviceların içinden, bu değişkenlere başvurarak, bağlama erişebiliriz.

Örnek:

```
    pointcut callSayMessageToPerson(String person)
            : call(* HelloWorld.sayToPerson(String, String))
              && args(*, person);

    void around(String person)
            : callSayMessageToPerson(person) {
        proceed(person + "-san");
    }
```

person değişkeni, sayToPerson(String, String) metodunun, ikinci argümanıdır. Biz bu değişkene adviceın içinden erişiyoruz. Ve buna -san takısını ekleyerek, gerçek sayToPerson(String, String) metodunu çağırıyoruz.

Advice:

```
void around(Connection conn) : call(Connection.close()) && target(conn) {
    if (enablePooling) {
        connectionPool.put(conn);
    } else {
        proceed();
    }
}
```

Yukarıdaki örnekte bütün Connection.close() çağrıları yakalanıyor. Eğer bağlantılar havuzdan gerçekleştiriliyorsa, bu bağlantı havuza konuluyor. Yok değilse, o zaman direk Connection.close() yapılıyor.

Metot Alan Eklemek:

```
aspect IntroduceMethodIllustration {
        private void MyClass.foo() {
                System.out.println("This is foo");
        }

        private static int MyClass.instanceCount = 0;
}
```

Kalıtım hiyerarşisini değiştirmek:

```
aspect MakeMyClassSerializable {
        declare parents : MyClass implements Serializable;
}
```

Exceptionı yumuşatmak:

Checked exceptionı unchecked haline getirmek (bunun içine sarmak için)

```
aspect SoftenDateFetcherRemoteException {
        declare soft :
            RemoteException : (call(public * DateFetcher.*(..))
                               || call(public DateFetcherImpl.new(..)))
                              && within(Test);
}
```

Başka neler yapılabilir:

1. Pointcutlar arasında öncelik sıralaması kurulabilir (çakışmalar için)

2. Varsayılan olarak, AspectJ bütün aspektler için aynı durum bilgisini kullanır. Her bir dokunan aspekt için ayrı bir durum tutulabilir. per... deyimleriyle.

3. Privileged aspektler, sınıfların private üyelerini kullanabilir.

4. Compile time hata ve uyarı kontrolu

Kısaltmalar

```
* herhangi bir karakter dizisi
+ kendisi ve alt sınıfları
```

Örnek 1 - Swing zincirlerine dışarıdan erişim

```
Örnek 1 - Swing zincirlerine dışarıdan erişim

public aspect SwingThreadSafetyCheckAspect {
    private static final String WARNING_MESSAGE
        = "BEWARE: Swing update called from non-AWT thread\n"
        + "Change code to use EvenetQueue.invokeLater() or invokeAndWait()";

    // Define a pointcut to capture calls to TableModel and its subclass's
    // method that would update the UI. Note, this list is incomplete.
    pointcut swingTableUpdateCalls() :
           call(* javax..*TableModel+.set*(..))
        || call(* javax..*TableModel+.add*(..))
        || call(* javax..*TableModel+.remove*(..));

    // Define a pointcut to capture calls to ListModel and its subclass's
    // method that would update the UI. Note, this list is incomplete.
    pointcut swingListUpdateCalls() :
           call(* javax..*ListModel+.set*(..))
        || call(* javax..*ListModel+.insert*(..));

    // Pointcuts for tree, text, and so on models
    // Implementation left out...

    // Pointcut for updates to any Swing model (use || to add pointcuts
    // to be defined above for tree, text, and so on).
    pointcut swingUpdateCalls()
        : swingTableUpdateCalls() || swingListUpdateCalls();

    /**
     * Advice to print a warning and stack trace that led to a call
     * to Swing update methods from nonevent dispatch thread.
     */
    before() : swingUpdateCalls() && !if(EventQueue.isDispatchThread()) {

        System.err.println(WARNING_MESSAGE);
        System.err.println("Joinpoint captured: " + thisJoinPoint);
        Thread.dumpStack();
        System.err.println();
    }
}
```

Örnek 2 - Yetkilendirme

```
public abstract aspect AbstractAuthenticationAspect {
    public abstract pointcut operationsNeeddingAuthentication();

    before() : operationsNeeddingAuthentication() {
        // Perform authentication. If it could not be authenticated,
        // let the exception thrown by authenticate propagate.
        Authenticator.authenticate();
    }
}

public aspect DatabaseAuthenticationAspect
    extends AbstractAuthenticationAspect {

    public pointcut operationsNeeddingAuthentication():
        call(* DatabaseServer.connect());
```

```
}
```

Örnek 3 - JapaneseMannersAspect

```
// JapaneseMannersAspect.java
public aspect JapaneseMannersAspect {
    pointcut callSayMessageToPerson(String person)
        : call(* HelloWorld.sayToPerson(String, String))
               && args(*, person);

    void around(String person)
        : callSayMessageToPerson(person) {
        proceed(person + "-san");
    }
}
```

Örnek 4 - Kaynak Havuzu

Resource-pool management modularization

Kodu, aspekt-deney-8 projesinde...

Örnek 5 - Politika uygulama

MVC çeşitli kuralların uygulanmasını gerektiriyor.

Mesela bir modele bir sunum nesnesini birden fazla kere dinleyici olarak atayabiliriz. Bunun önlenmesi gerekir.

İkincisi, bir sunumu yok etmeden önce, dinleyicilerden kaldırmayı unutabilirsin. Bu durumda çöp toplayıcısı asla bu sunum nesnesini kaldıramaz. Hafıza üzerinde yük oluşur.

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\66.gif

Her iki aspekt de sunumların, modellere dinleyici olarak eklendiği pointcutlarla ilgili olduğu için, bu pointcutı üst aspektte tanımlarız: EventListenerManagement

Burada: addListenerCall() pointcutı, listener ekleme bağlantı noktalarını yakalar.

```
public abstract aspect EventListenerManagement {
    pointcut addListenerCall(Object model, EventListener listener)
        : call(void *.add*Listener(EventListener+))
        && target(model) && args(listener) && modelAndListenerTypeMatch();

    abstract pointcut modelAndListenerTypeMatch();
}
```

Teklik meselesinin gerçekleştirilmesi: Bu mesele, temelinde bir dinleyici eklemeden önce, bu dinleyicinin mevcut olup olmadığını kontrol ediyor.

addListenerCall kesitine bir tavsiye(advice) yapıyoruz. Bu eklenen dinleyicinin daha önce bulunup bulunmadığını kontrol ediyor. Eğer yoksa, normal işleme devam ediyor (proceed ile).

```
public abstract aspect EventListenerUniqueness
    extends EventListenerManagement {
    void around(Object model, EventListener listener)
        : addListenerCall(model, listener) {

        EventListener[] listeners = getCurrentListeners(model);
        if (!Utils.isInArray(listeners, listener)) {
            System.out.println("Accepting " + listener);
            proceed(model, listener);
        } else {
            System.out.println("Already listening " + listener);
        }
    }

    public abstract EventListener[] getCurrentListeners(Object model);
```

```
}
```

Şimdi bu aspekti somutlaştırmalıyız.

TableModelListenerUniqueness somut aspekti, bu aspekti sadece TableModellere sınırlıyor. Bunu yapmak için modelAndListenerTypeMatch() pointcutının sadece AbstractTableModel TableModelListenera sınırlayan bir implementasyonunu sunuyor.

```
aspect TableModelListenerUniqueness extends EventListenerUniqueness {
    pointcut modelAndListenerTypeMatch()
        : target(AbstractTableModel) && args(TableModelListener);

    public EventListener[] getCurrentListeners(Object model) {
        return ((AbstractTableModel)model)
            .getListeners(TableModelListener.class);
    }
}
```

Simplify your logging with AspectJ

Simplify your logging with AspectJ

file:///F:\Belgelerim\Yayınlar\Java\Makaleler\Dizayn\Aspect\Simplify%20your%20logging%20with%20AspectJ.doc

Katkı (Introduction):

Introduction ile bir sınıfa yeni bir özellik katıyoruz.

Mesela şurada yeni bir interface implemantasyonunu katıyoruz:

```
       declare parents: Account implements BankingEntity;
```

Compile-time declaration:

Compile zamanında kullanım kurallarını kontrol eder.

Mesela, AWT sınıflarını model paketindeki sınıflardan çağırmak yasak olsun. Böyle bir çağrı bulduğunda, hemen uyarı verir.

```
declare warning : call(void Persistence.save(Object))
                : "Consider using Persistence.saveOptimized()";
```

Yukarıdaki örnekte Persistence.save metodunun çağrılması uyarı oluşturuyor.

Kodlar:

```
public class Test {
    public static void main(String[] args) {
        Inventory inventory = new Inventory();
        Item item1 = new Item("1", 30);
        Item item2 = new Item("2", 31);
        Item item3 = new Item("3", 32);
        inventory.addItem(item1);
        inventory.addItem(item2);
        inventory.addItem(item3);
        ShoppingCart sc = new ShoppingCart();
        ShoppingCartOperator.addShoppingCartItem(sc, inventory, item1);
        ShoppingCartOperator.addShoppingCartItem(sc, inventory, item2);
    }
}


public class ShoppingCartOperator {
    public static void addShoppingCartItem(ShoppingCart sc,
                                           Inventory inventory,
                                           Item item) {
        inventory.removeItem(item);
        sc.addItem(item);
```

```
    }

    public static void removeShoppingCartItem(ShoppingCart sc,
                                              Inventory inventory,
                                              Item item) {
        sc.removeItem(item);
        inventory.addItem(item);
    }
}

public class Inventory {
    private List _items = new Vector();

    public void addItem(Item item) {
        _items.add(item);
    }

    public void removeItem(Item item) {
        _items.remove(item);
    }
}

public class Item {
    private String _id;
    private float _price;

    public Item(String id, float price) {
        _id = id;
        _price = price;
    }

    public String getID() {
        return _id;
    }

    public float getPrice() {
        return _price;
    }

    public String toString() {
        return "Item: " + _id;
    }
}

public class ShoppingCart {
    private List _items = new Vector();

    public void addItem(Item item) {
        _items.add(item);
    }

    public void removeItem(Item item) {
        _items.remove(item);
    }

    public void empty() {
        _items.clear();
    }

    public float totalValue() {
        // unimplemented... free!
        return 0;
    }
}
```

Loglama için logp() metodu kullanılıyor, log() yerine. log()´un hem performansla ilgili hem de doğru kaydetmekle ilgili bazı problemleri varmış.

Bunun için bütün sınıflara şu şekilde logp() metotlarını ekliyoruz:

```
public class Inventory {
    private List _items = new Vector();
    static Logger _logger = Logger.getLogger("trace");
```

```
    public void addItem(Item item) {
        _logger.logp(Level.INFO, "Inventory", "addItem", "Entering");
        _items.add(item);
    }

    public void removeItem(Item item) {
        _logger.logp(Level.INFO, "Inventory", "removeItem", "Entering");
        _items.remove(item);
    }
}
```

Bütün sınıflar ve metotlar için bunu yapacağız...

Aynı şeyi aspektlerle yapmak:

Sadece bir aspekt tanımlayarak:

```
public aspect TraceAspect {
    private Logger _logger = Logger.getLogger("trace");

    pointcut traceMethods()
            : execution(* versiyon3.*.*(..)) && !within(TraceAspect);

    before() : traceMethods() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        _logger.logp(Level.INFO, sig.getDeclaringType().getName(),
                sig.getName(), "Entering");
    }
}
```

Geleneksel yaklaşımın hatası:

Loglama normal iş sınıflarının sorumluluğunda olmadığı halde, bunların içine karışmış durumda. -» Kodun okunaklığını güçleştiriyor.

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\64.gif

Ayrıca yarın bir gün loglama yöntemimizi veya loglama modülümüzü değiştirmek istesek, her yerde bunu değiştirmemiz gerekecek.

Ayrıca bütün her yerde tutarlı bir şekilde log aldığımızı nasıl temin edeceğiz. Biri farklı bir formatta log alabilir. Unutmuş olabilir.

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\65.gif

---

Takip (tracing)

Tracing, debug etmenin çözüm olmadığı yerlerde elzemdir.

Tracing, loglamanın özel bir türüdür. Her metoda giriş ve çıkışın loglanması anlamına gelir. Geliştirme aşamasında çok faydalı.

```
public aspect TraceAspectV1 {
    pointcut traceMethods()
            : (execution(* versiyon3.*.*(..))
           || execution(versiyon3.*.new(..))) && !within(TraceAspectV1);

    before() : traceMethods() {
        Signature sig = thisJoinPointStaticPart.getSignature();     //2
        System.out.println("Entering ["                             //2
                + sig.getDeclaringType().getName() + "."  //2
                + sig.getName() + "]");                   //2
    }
}
```

Burada System.out´u kullandık. Bunun yerine Logger paketini kullanacak olsak:

```
import java.util.logging.*;
```

```
import org.aspectj.lang.*;

public aspect TraceAspectV2 {
    private Logger _logger = Logger.getLogger("trace");

    TraceAspectV2() {
        _logger.setLevel(Level.ALL);
    }

    pointcut traceMethods()
        : (execution(* *.*(..))
          || execution(*.new(..))) && !within(TraceAspectV2);

    before() : traceMethods() {
        if (_logger.isLoggable(Level.INFO)) {
            Signature sig = thisJoinPointStaticPart.getSignature();
            _logger.logp(Level.INFO,
                         sig.getDeclaringType().getName(),
                         sig.getName(),
                         "Entering");
        }
    }
}
```

Yapacağımız değişiklik oldukça kolay.

Not:

Another reason to use logp() is that calling the log() method would cause the logging kit to deduce that the caller is the advice instead of the advised method. This happens because the log() method uses the call stack to infer the caller.

Logger yerine log4J´i kullanmak istesek:

```
public aspect TraceAspectV3 {
    Logger _logger = Logger.getLogger("trace");

    TraceAspectV3() {
        _logger.setLevel(Level.ALL);
    }

    pointcut traceMethods()
        : (execution(* *.*(..))
          || execution(*.new(..))) && !within(TraceAspectV3);

    before() : traceMethods() {
        if (_logger.isEnabledFor(Level.INFO)) {
            Signature sig = thisJoinPointStaticPart.getSignature();
            _logger.log(Level.INFO,
                        "Entering ["
                        + sig.getDeclaringType().getName() + "."
                        + sig.getName() + "]");
        }
    }
}
```

Sentaks

- * 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. Örnek: javax.swing.JComponent+ JComponent ve alt sınıfları
- ! : Tekil operatör. Örnek: !within(JoinPointTraceAspect) Bu pointcutın içerdiği dışındaki bağlantı noktaları
- || && : veya ve. İkili operatörler

İmza Kalıp Örnekleri:

```
Account               Type of name Account.
*Account              Account ile biten tipler, mesela: SavingsAccount
java.*.Date    Type Date in any of the direct subpackages of the java package, such as java.util.Date
```

and java.sql.Date.
java..* Any type inside the java package or all of its direct subpackages, such as java.awt and java.util, as well as indirect subpackages, such as java.awt.event and java.util.logging.
javax..*Model+ javax paketinde ismi Model ile biten bütün tipler ve alt tipleri. This signature would match TableModel, TreeModel, and so forth, and all their subtypes.

Bağlam Bilgisi:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\69.gif

Aynı şeyi isimlendirilmiş pointcutlarla da yapabiliriz:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\70.gif

Dönen nesneyi yakalamak:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\71.gif

Exception nesnesine ulaşmak

Örnek 1 - Failure Handling

Bir exception oluşuyor. Onu aspektten yakalıyoruz ve onu çözmeye yönelik bir algoritmayı buradan uyguluyoruz. Tekrar deneyerek çözmeye çalışıyoruz:

```
public class RemoteService {
    public static int getReply() throws RemoteException {
        if(Math.random() > 0.5) {
            throw new RemoteException("Simulated failure occurred");
        }
        System.out.println("Replying");
        return 5;
    }
}

public class RemoteClient {
    public static void main(String[] args) throws Exception {
        int retVal = RemoteService.getReply();
        System.out.println("Reply is " + retVal);
    }
}

public aspect FailureHandlingAspect {
    final int MAX_RETRIES = 3;

    Object around() throws RemoteException
        : call(* RemoteService.get*(..) throws RemoteException) {
            int retry = 0;
            while(true){
                try{
                    return proceed();
                } catch(RemoteException ex){
                    System.out.println("Encountered " + ex);
                    if (++retry > MAX_RETRIES) {
                        throw ex;
                    }
                    System.out.println("\tRetrying...");
                }
            }
        }
}
```

Örnek 2 - Factorial

Bu örnekte, bağlam bilgisine nasıl ulaşılacağı çok güzel görülüyor:

```
public aspect OptimizeFactorialAspect {
    pointcut factorialOperation(int n) : call(long *.factorial(int)) && args(n);

    pointcut topLevelFactorialOperation(int n) : factorialOperation(n)
        && !cflowbelow(factorialOperation(int));
```

```
    private Map _factorialCache = new HashMap();

    before(int n) : topLevelFactorialOperation(n) {
        System.out.println("Seeking factorial for " + n);
    }

    long around(int n) : factorialOperation(n) {
        Object cachedValue = _factorialCache.get(new Integer(n));
        if (cachedValue != null) {
            System.out.println("Found cached value for " + n
                    + ": " + cachedValue);
            return ((Long)cachedValue).longValue();
        }
        return proceed(n);
    }

    after(int n) returning(long result) : topLevelFactorialOperation(n) {
        _factorialCache.put(new Integer(n), new Long(result));
    }
}

public class TestFactorial {
    public static void main(String[] args) {
        System.out.println("Result: " + factorial(5) + "\n");
        System.out.println("Result: " + factorial(10) + "\n");
        System.out.println("Result: " + factorial(15) + "\n");
        System.out.println("Result: " + factorial(15) + "\n");
    }
    public static long factorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n-1);
        }
    }
}
```

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\73.gif

Advice çalışma şeması

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\84.gif