Eigene „Aspekte“ in EJBs

Markus Völter, voelter@acm.org, www.voelter.de

EJB und Aspektorientierung

Das primäre Ziel serverseitiger Komponentenarchitekturen wie EJB ist die Herausfaktorisierung technischer Belange aus Anwendungscode [VSW02]. Der Container kümmert sich um die technischen Belange, der Entwickler schreibt Komponenten in deren Implementierung er sich nicht um die technischen Aspekte kümmern muß. Er muß den Container lediglich *steuern* (mittels des Deployment Descriptors). Wenn man Komponenteninfrastrukturen unter diesem Licht betrachtet sind sie eine Form der Aspektorientierung. AOP hat zum Ziel querliegende Belange (crosscutting concerns) aus einem System herauszuziehen und in einem Aspekt zu lokalisieren. Dies erlaubt ein leichteres Verständnis des Programms und erlaubt des weiteren, bestimmte Aspekte „ein- oder auszuschalten“. Außerdem fördert diese Art von Vorgehen die Wiederverwendbarkeit: Kernlogik und Aspekte können (bei guten Design) separat wiederverwendet werden. Der offensichtliche Nachteil von EJB wenn man es als AOP Framework betrachtet ist, daß die Anzahl und Art der Concerns auf die von der EJB Spezifikation beschriebenen (Persistenz, Transaktionen, Security, ...) beschränkt ist.

Der vorliegende Artikel erläutert, wie man sich ein einfaches Framework bauen kann, welches es erlaubt, bei Beans beliebige Aspekte an bestimmten Stellen (Joinpoints) einzuhängen.

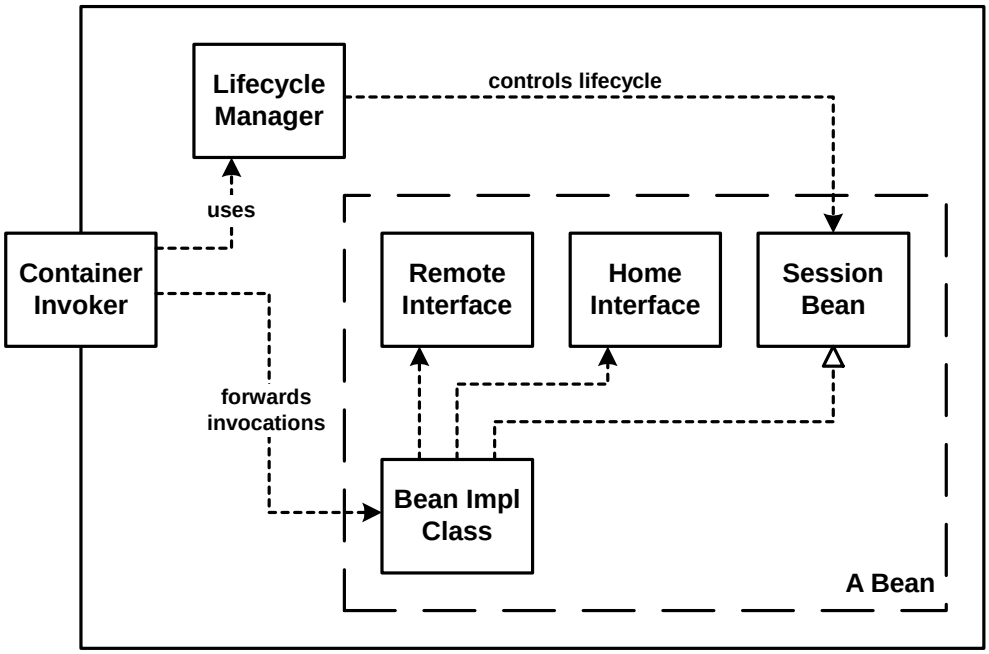
Join Points und Point Cuts

Eine der wichtigsten Aspekte bei der Betrachtung der Mächtigkeit von AOP Frameworks sind das Joinpoint Modell und die Pointcut Sprache. Das Joinpoint Modell definiert, an welchen Stellen im Ablauf eine Programmes man mittels Aspekten prinzipiell Verhalten einfügen („advisen“) kann. Die Pointcut Sprache definiert, mittels welcher Selektionsmittel man definiert, an welchen der möglichen Joinpoints ein bestimmter Aspekt nun tatsächlich greifen soll.

Spracherweiterungen wie AspectJ bieten hier eine Fülle von Möglichkeiten [AspectJ]. Die in diesem Artikel vorgestellte Technik erlaubt nur eine sehr geringe Anzahl von Joinpoints. Wie EJB selbst, können auch hier Aufrufe auf Methoden die im Local- oder Remote Interface deklariert sind per Aspekt „erweitert“ werden. Auch die Pointcuts sind prinzipiell nur auf dieser Ebene definierbar. Dies ist in der Praxis für die später vorzustellenden Anwendungsfälle aber keine Limitierung.

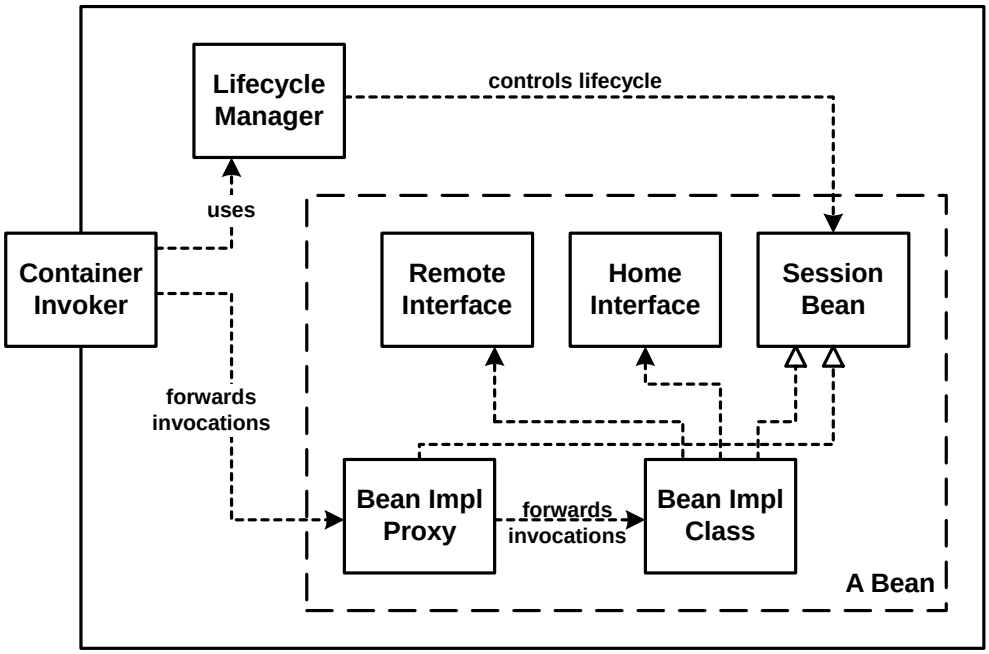
Der Proxy

Das folgende Bild zeigt die prinzipielle Struktur des Containers und einer darin lebenden Bean.



Der *ContainerInvoker* bekommt von der Netzwerkschicht die Aufrufe die Clients auf der Bean (bzw. auf deren Client-seitigen Proxy) ausführen. Er verwendet dann den *LifecycleManager* um eine Bean Instanz zu erhalten. Wie dies genau passiert hängt a) von der Art der Bean ab (Entity, Stateless/Stateful Session, MDB) und b) von der Implementierung des Containers selbst. Wenn der *Invoker* eine Referenz auf eine Bean Instanz hat, ruft er (üblicherweise per Reflection) die vom Client angegebene Methode auf dem entsprechenden Bean Implementierungsobjekt auf.

Um jetzt beliebige Aspekte integrieren zu können, ist es nötig, sich zwischen den Invoker und die Bean Implementierungsklasse einzuhängen. Unter Verwendung von Interceptoren ist dies prinzipiell sehr einfach möglich; das folgende Bild zeigt, wie.



Die Idee ist, aus der Bean Implementierungsklasse mittels eines einfachen Code Generators einen Proxy zu generieren. Dieser implementiert jede der öffentlichen Methoden der Bean mit folgender Logik (hier in Pseudocode):

```
public class SomeBeanAOPProxy implements SessionOrEntityBean {

  private SomeBeanImplClass delegate = new SomeBeanImplClass();
  private SessionOrEntityContext context = …;

  public RetType someOp( T1 param1, T2 param2, … ) {
    Object[] params = {param1, param2, …};
    Class[] types = {T1.class, T2.class, …};
    Invocation inv = new Invocation( “SomeBean”, “someOp”,
                                     params, types);
    AOPHandler.handlePreInvoke(context, inv);
    return delegate.someOp( param1, param2, … );
  }

  // the same for the other methods…

}
```

Ziel dieser Implementierung ist es, die Informationen über den Aufruf in ein Hilfsobjekt zu verpacken (*Invocation*) und dann dieses Objekt and den *AOPHandler* zu übergeben. Dieser weiß, welche Aspekte für die entsprechende Methoden definiert sind und führt diese aus. Dazu werden zunächste in der Methode zwei Arrays angelegt. Eines enthält die Werte der

Parameter der Operation, das zweite die Typen. Nun wird ein *Invocation* Objekt erstellt welches zusätzlich zu den zwei Arrays den Name der Bean und den Name der Operation enthält. Dieses *Invocation* Objekt wird nun zusammen mit dem aktuellen Kontext der Bean an den *AOPHandler* weitergegeben. Was dieser dann genau tut werden wir in Kürze sehen.

Nach Ausführung des *AOPHandlers* wird der eigentliche Methodenaufruf auf dem *delegate*, dem eigentlichen Beanimplementierungsobjekt durchgeführt und der Rückgabewert an den Aufrufer zurückgegeben.

Die Lifecycle-Operationen der Bean (*setContext()*, *ejbLoad()*, *ejbActivate()*, ...) werden üblicherweise ohne den Handler implementiert weil sie meist nicht von Aspekten beeinflusst werden sollen – sie delegieren also schlicht an das Implementierungsobjekt weiter.

Und das geht so?

Im Prinzip geht das so; auf das eine oder andere Detail ist natürlich zu achten:

- Der Deployment Descriptor muß als Implementierungsklasse der Bean natürlich den Proxy enthalten, nicht die eigentliche Implementierungsklasse.
- Primitive Typen (*int*, *long*, ...) müssen mit ihren Box-Typen (*Integer*, *Long*, ...) gewrappt werden.
- Die Signatur der Proxyoperationen muß die Exceptions des Originals enthalten.
- Möglicherweise will man nicht nur *vor* dem Aufruf der Operation eingreifen können sondern auch *danach*. Das folgende Stückchen Code zeigt, wie dies geschehen kann. Auch hier muß auf ggfs. auftretende Exceptions noch passend reagiert werden.

```
public RetType someOp( T1 param1, T2 param2, … ) {
  // as before…
  RetType retval = delegate.someOp( param1, param2, … );
  return AOPHandler.handlePostInvoke(context, inv, retval);
}
```

- Möglicherweise will man dem *AOPHandler* die Möglichkeit geben, die Ausführung der Operation zu stoppen (und bspw. eine *SecurityException* zu werfen). Dies ist mit obigem Vorgehen leicht zu implementieren.

Generierung & Build

Der vorgeschlagene Ansatz verwendet Code Generierung; dies ist aber dennoch kein Grund für die Verwendung einer großen Generatorinfrastruktur. Ein einfacher *System.out.println*-Generator reicht völlig aus. Als Eingabe in den Generator dient das Class-File der Implementierungsklasse: Mittels Reflection auf der Bean-Implementierungsklasse sind alle Informationen zu erhalten, die man benötigt. Die

Implementierung der Methode im Proxy ist außerdem sehr einfach und immer gleich, der Generator also straight forward.

Auch die Integration in den Build-Prozess ist sehr einfach. Der selbst geschriebene Generator kann mittels eines selbstgeschriebenen Ant Tasks sehr leicht in den Buildprozess eingegliedert werden.

Man wird sich vielleicht fragen, warum wir nicht dynamische Proxies statt eines Codegenerators verwenden. Dies können ja auch Methodenaufrufe „verpacken" und an einen *InvocationHandler* weitergeben. Das Problem dabei ist, dass man diesen dynamischen Proxy ja für das Bean-Implementierungsobjekt anlegen müsste. Da die Instantiierung dieses Objektes vom ApplicationServer erledigt wird, hat man keinen weg, dort einzugreifen.

Aspekte bzw. Interceptoren

Bisher haben wir nur beschrieben, wie Aspektcode mittels des *AOPHandlers* in die Bean-Infrastruktur eingehängt werden kann. Offen blieb noch die Frage, wie der Aspekt, also das zusätzliche Verhalten implementiert wird. Das folgende Stückchen Code zeigt die abstrakte Oberklasse der dazu verwendeten Interceptoren:

```
public abstract class AOPInterceptor {
  public void handlePreInvoke(SessionOrEntityContext ctx,
                              Invocation invocation);
  public Object handlePostInvoke(SessionOrEntityContext ctx,
                         Invocation invocation, Object result);
}
```

Wie immer bei AOP Tools verwenden auch wir einen einfachen Logger als „Hello-World" der Aspektorientierung:

```
public class LoggingInterceptor extends AOPInterceptor {
  public void handlePreInvoke(SessionOrEntityContext ctx,
                              Invocation invocation) {
    // logger is a log4j logger or something similar
    logger.log( "Invoking "+invocation.getOperation()+" on "+
                invocation.getBeanName() );
  }
  public Object handlePostInvoke(SessionOrEntityContext ctx,
                         Invocation invocation, Object result) {
    // do nothing here.
  }
}
```

Es kann für jeden Joinpoint potentiell viele Interceptoren geben. Der *AOPHandler* muß also für alle konfigurierten Interceptoren die *handlePreInvoke()* bzw. *handlePostInvoker()* nacheinander aufrufen.

Konfiguration

Wir haben nun (in AOP-Speak) Aspekte, Joinpoints, und ein Framework welches sich um die Ausführung kümmert. Was fehlt ist die Spezifikation der Pointcuts: also welcher Aspekt wann ausgeführt werden soll. Es ist im Interesse der Wiederverwendbarkeit des Frameworks ratsam, diese Konfiguration flexibel zu halten. Dazu bietet sich ein XML Konfigurationsfile an, welches vom *AOPHandler* einmal geladen und geparst wird. Das Format kann bspw. folgendermaßen aussehen:

```
<aopconfig>
  <aspect class="de.voelter.ejbaop.LoggingInterceptor">
    <applyTo
       beanName="de.voelter.ejbaop.example.SomeBean"
       allOps="true"/>
    <applyTo
       beanName="de.voelter.ejbaop.example.SomeOtherBean"
       allOps="false">
      <operation name="someOp"/>
      <operation name="AnotherOp"/>
    </applyTo>
   </aspect>
</aopconfig>
```

Auch hier kann man sich beliebige Erweiterungen vorstellen, die je nach Anwendungsfall auch nötig sind. Bspw. kann bei der Spezifikation einer Operation (im *operation*-Tag) die Signatur mit angegeben werden, wenn es überladene Operationen in der Bean gibt.

Anwendungen

Die folgenden Aspekte lassen sich beispielsweise mittels dieses Frameworks schön separat von der Beanimplementierung handeln:

- Logging: wie im Beispiel oben
- Dynamische Security: Security-Überprüfungen abhängig von den Werten der Operationsparameter sowie dem Principal im Kontext
- Sessions: Abhängig von Principal im Kontext kann der Implementierungsklasse ein Session-Objekt für Stateless Session Beans zur Verfügung gestellt werden. Dies entspräche ungefähr der HTTPSession in Servlets.
- Lizenzierung/Overload: Die Menge an gleichzeitig eintreffenden Requests kann verfolgt werden. Wenn diese größer sind als eine bestimmte Maximalzahl kann eine Exception zum Client zurückgeworfen werden.

Praxiserfahrungen

Eine Implementierung des hier vorgestellten Frameworks habe ich in verschiedenen Projekten erfolgreich eingesetzt. Dabei kamen alle oben genannten Anwendungsfälle zum

Einsatz. Zu diesem Zweck habe ich einen entsprechenden Codegenerator geschrieben. Insgesamt ist der Aufwand mit ca. 2-3 Tagen zu beziffern, damit ist das für viele Projekte durchaus akzeptabel.

Andere Ansätze und Zusammenfassung

Der Wert der Möglichkeit, beliebige Aspekte im Rahmen von EJB herausfaktorisieren zu können wird derzeit von verschiedenen Stelle erkannt. JBoss [JBoss] bietet ab Version 4 den sog. Generalized Aspect Container, wo man selbst implementierte Aspekte einhängen kann. Die Standard-EJB Aspekte werden genauso behandelt wie die vom Entwickler selbst geschriebenen, liegen aber eben im Rahmen einer Bibliothek bereits vor. Implementiert wird das ganze hier durch Modifikation der Beanklasse beim Laden unter Zuhilfenahme von *javassist* [SC03]. Auch BEA hat dies erkannt und bietet mit dem Weblogic Aspect Framework eine auf AspectJ basierende Lösung an.

Einige dieser Ansätze sind zwar flexibler, dafür ist der hier vorgestellte Ansatz portabel über alle Applikationsserver und kommt ohne zusätzliche Tools aus (wenn man von dem einfachen Codegenerator absieht). Und man kann damit schon recht sinnvolle Dinge tun... Viel Spaß beim Ausprobieren!

Referenzen

JBoss www.jboss.org

SC03 http://www.csg.is.titech.ac.jp/~chiba/javassist/

VSW02 Völter, Schmid, Wolff; *Server Component Patterns*; Wiley & Sons, 2002