

Herança de EJBs com XDoclet

Marcelo Mrack

Este artigo mostra como criar herança entre EJBs através do uso do Xdoclet. Para uma introdução à herança de EJBs é recomendado a leitura de [1].

Introdução

Muitas vezes é necessário compartilhar uma característica comum entre um conjunto de classes correlatas. Na abordagem tradicional, usando classes, isso é facilmente implementado usando o conceito de herança. Outras vezes, quando a herança não é possível (classes final não podem ser estendidas) ou não é desejada (já existe uma hierarquia muito grande de classes no sistema) também é possível usar adaptadores [2], tendo de objetos como de classes. Essas abordagens são simples e eficazes.

A tecnologia Enterprise Java Beans, ou simplesmente EJB [3] também suporta essa característica. Entretanto, muitos desenvolvedores encontram problemas ao implementar essa característica nesses objetos. Isso ocorre devido às características implícitas nesses componentes, tais como necessidade de implementar certas interfaces e serem construídos e gerenciados pelo container da aplicação. Assim, para minimizar essas inconveniências é interessante o uso de ferramentas que automatizem o processo de criação dos famosos EJBs.

O XDoclet [4] é uma dessas tecnologias e funciona muito bem, obrigado. Como é demonstrado nesse artigo, é muito fácil usar o XDoclet para fazer o trabalho "grosso" quando queremos criar nossos EJBs, mais precisamente quando queremos criar EJBs usando herança entre eles.

As sessões seguintes mostram um estudo de caso, que mostra como usar o XDoclet para automatizar a geração de EJBs com herança entre eles.

Nota 1: Presume-se que o leitor já tenha experiência com a criação de EJBs simples e tenha conhecimento das tecnologias comentadas.

Nota 2: Os exemplos abaixo foram implementados na ferramenta Eclipse 3, Ant 1.6, XDoclet 1.2 e JBoss 3.2.3. O uso de plugins J2EE como o LomboZ ou o JBossIDE é recomendado, mas não obrigatório.

O problema

Como exemplo para o estudo de caso vamos imaginar um ambiente onde o desenvolvedor deseja criar um EJB para representar o banco de dados da aplicação. Esse EJB tem todos os métodos básicos de operação sobre dados, tais como *inserts*, *updates*, *deletes* e *selects*. Em outras palavras esse EJB implementa as operações CRUD. Ainda, nesse exemplo, vamos imaginar que o banco de dados da aplicação é acessado via um framework de mapeamento OO-ER, tal como o framework Hibernate [5]. Depois, queremos criar outros EJBs (derivados do EJB genérico) que inicializem a camada de dados e disponibilizem-a para o cliente. Finalmente, queremos que o código cliente tenha uma interface única que possa acessar *qualquer* um dos EJBs.

O EJB genérico

O primeiro passo é criar o EJB genérico. O código abaixo mostra parte desse EJB:

```
1) /**
2)  * @author mrack
3)  * @version 1.0
4)  * @since Sep 2, 2004 9:41:39 PM
5)  * @ejb.bean name="BaseHibernate"
6)  * jndi-name="ejb/BaseHibernate"
7)  * type="Stateless"
8)  * @ejb.interface
9)  * local-class="br.xpto.commom.interfaces.PersistenciaLocal"
10) * remote-class="br.xpto.commom.interfaces.PersistenciaRemota"
11) */
12) public class BaseHibernateBean implements javax.ejb.SessionBean {
```

Nesse código, o mais importante são as linhas 8, 9 e 10. Elas indicam para o XDoclet para gerar as interfaces de acesso para este EJB no pacote "br.xpto.commom.interfaces". Essas

interfaces tem os nomes "PersistenciaLocal" e "PersistenciaRemota". Isso indica que o EJB de nome "BaseHibernate" vai ser acessado pelas interfaces de nome "PersistenciaX". Por quê? Um dos nossos requisitos do nosso estudo de caso é que o código cliente não precise ser modificado caso alguma alteração ocorra na estrutura ou nome dos EJBs. Também deseja-se que, caso um novo EJB seja criado, o cliente possa acessar ele da mesma forma, sem alteração de código. É por esse motivo que essas interfaces são usadas. Elas são a forma com que o cliente usa os EJBs. Se você não entendeu muito bem, não se preocupe, logo abaixo você terá mais informações.

Depois de criar esse código você pode usar seu plugin preferido de EJB no Eclipse para gerar as interfaces e classes acessórias. Tanto o JBossIDE [6] como o LomboZ [7] servem. No meu caso eu criei um script de build por achar mais prático e funcional.

Uma vez que as interfaces do EJB genérico tenham sido geradas com sucesso, o próximo passo é criar os EJBs derivados.

Nota 3 : Como esse EJB não aponta para nenhuma base de dados não tem lógica o cliente usar esse EJB. Ele atua simplesmente como um template para os outros EJBs que herdam dele.

Um EJB que herda do EJB genérico

Como comentado anteriormente, o EJB genérico tem todos os métodos CRUD para acesso e manutenção de dados na camada de persistência. Mas tem um detalhe: ele não aponta para nenhuma base de dados! Assim, é tarefa de um EJB que herda do EJB genérico inicializar a camada de persistência. É exatamente isso que vou mostrar agora.

```
import br.xpto.common.ejb.BaseHibernateBean;

/**
 * @author mrack
 * @version 1.0
 * @since Aug 30, 2004 10:56:35 PM
 * @ejb.bean name="GapeHibernate"
 *          display-name="Name for GapeHibernate"
 *          description="Description for GapeHibernate"
 *          jndi-name="ejb/GapeHibernate"
 *          type="Stateless"
 *          view-type="both"
 *
 * @ejb.home
 *          local-extends="javax.ejb.EJBLocalHome"
 *          extends="javax.ejb.EJBHome"
 *
 * @ejb.interface
 *          local-class="br.xpto.common.interfaces.PersistenciaLocal"
 *          remote-class="br.xpto.common.interfaces.PersistenciaRemota"
 */
public class GapeHibernateBean extends BaseHibernateBean implements javax.ejb.SessionBean {

    /**
     * Inicializa a camada de persistência do sistema.<br>
     */
    public GapeHibernateBean() {
        setJNDI("java:/br/xpto/gape/GapeHibernate");
    }
}
```

Nesse trecho de código percebe-se que o nosso EJB está herdando do EJB genérico, nada de mais. Também notamos a existência das interfaces "PersistenciaX". Como queremos que o código cliente use sempre a mesma interface para acessar qualquer EJB dessa família, devemos indicar ao XDoclet que ele deve gerar o acesso à este EJB também através dessas interfaces. Porém, notamos a presença de uma tag "@ejb.home". Ela está presente para indicar ao XDoclet que, ao gerar as interfaces desse EJB, elas extendam as interfaces normais dos EJBs. Esse tópico é coberto em detalhes em [2] e por isso não falo sobre eles. A dica é sempre use @ejb.home quando quiser herança de EJBs. Finalmente, notamos que o construtor faz a inicialização da camada de persistência.

Pronto. Agora basta usar nossos plugins ou nosso script de build e gerar as classes para esse EJB!

O código cliente

Vamos recapitular o que temos até aqui. Temos um EJB base que possui todos os métodos CRUD. Esse EJB é acessado por duas interfaces padrões, chamadas “PersistenciaLocal” e “PersistenciaRemota” mas não aponta para nenhuma base de dados e por isso não deve ser usado pelo cliente. Temos um EJB que herdou do EJB genérico e também pode ser acessado pelas mesmas interfaces que o EJB pai. Esse EJB inicializa a camada de dados no seu construtor e pode ser usado pelo cliente. Agora vamos criar esse cliente!

```
/**
 * @author mrack
 * @version 1.0
 * @since Sep 14, 2004 10:42:26 PM
 * @ejb.bean name="Cadastros"
 *          display-name="Name for Cadastros"
 *          description="Description for Cadastros"
 *          jndi-name="ejb/Cadastros"
 *          type="Stateless"
 *          transaction-type = "Container"
 *          view-type="local"
 */
public class CadastrosBean implements SessionBean {

    Log log = Log.getInstance();

    PersistenciaLocal persistencia = null;

    protected SessionContext ctx;

    /**
     * ...
     * <p><b>Autor...:</b> mrack</p>
     * <p><b>Criação...:</b> Sep 14, 2004 10:42:05 PM</p>
     */
    public CadastrosBean() {
        super();
        //Inicializa a camada local de persistência
        persistencia =
PersistenciaFactory.getInstance(ServiceLocator.SERVIDOR_LOCAL).getPersistenciaLocal(GapeHibernateLocalHome.class);
        log.log.info("Camada de persistencia local inicializada com sucesso.");
    }
}
```

Esse código é muito simples. Percebe-se logo que nosso cliente também é um EJB. Nada de mais para um ambiente J2EE. Esse cliente declara um objeto do tipo “PersistenciaLocal”. Esse objeto pode ser inicializado com qualquer um dos EJBs, tanto o genérico quanto o “dedicado”. Obviamente, não vamos inicializá-lo com o EJB genérico, pois ele não aponta para nenhuma base de dados. Logo, no construtor do nosso cliente inicializamos nossa persistência local. Isso é feito por uma classe factory. Essa classe factory é simplesmente uma conveniência para nós, pois poderíamos inicializar a persistência local usando o localizador de serviços normalmente. Digamos que é uma boa prática. No método “getPersistenciaLocal” notamos que o parâmetro é a classe “GapeHibernateLocalHome”. O que é isso?

Essa técnica não é padrão em um localizador de serviços, tal como descrito em [8], mas é justamente o que precisamos para o nosso caso. Se olharmos dentro da nossa factory vamos ver que ela usa recursos de reflexão [9] para obter o nome JNDI que deve ser usado para instanciar o EJB e ligá-lo no cliente. Nada impediria que eu tivesse feito de outra forma, como por exemplo, passando para a factory o nome JNDI a ser usado. Isso apenas é uma conveniência.

Usando o cliente

Uma vez instanciado o EJB no cliente, basta usá-lo:

```
public boolean deleteAreaPreferencial(int id) {
    if (persistencia == null) {
        log.log.error("Camada de persistencia local nao inicializada");
        return false;
    }
    String hql = "from AreaPreferencial a where a.id = " + id;
    persistencia.delete(hql);
    //Se houve algum erro na deleção dos objetos (ex.: violação de FK), esse método deve
retornar false.
    if (ctx.getRollbackOnly() == true) {
        log.log.info("Erro na delecao dos registros. Retornando false.");
        return false;
    }
    return false;
}
```

Nesse trecho de código temos um método no EJB cliente que apaga um registro com base no seu ID. O primeiro pedaço de código verifica se realmente o EJB que representa a camada de persistência foi inicializado. Em caso positivo, invoca o método “delete”. Lembrem-se que esse método foi definido no nosso EJB genérico e por isso está presente no EJB instanciado (o GapeHibernateBean) no cliente. Se a execução do método gerou algum problema, o contexto da transação é inválido e por isso o método retorna false (estou usando transações gerenciadas por container nesse exemplo).

Criando outros EJBs que herdam do EJB genérico

Se quisermos criar outros EJBs a partir do EJB genérico, basta seguirmos os passos de criação do EJB GapeHibernateBean, observando as tags “@ejb.home” e “@ejb.interface”. Depois, no cliente simplesmente apontamos para a classe “localhome” do EJB que acabamos de criar. Nada mais foi mudado no cliente.

Conclusões

Esse artigo mostrou como criar EJBs que usam herança através da poderosa tecnologia XDoclet. Criei esse artigo pois foi difícil achar na web artigos práticos sobre o assunto. Em [1] é comentado muito bem o problema, porém não é mostrado como automatizar essa tarefa. E, obviamente é esse o ponto realmente importante no trabalho dos desenvolvedores. Agora basta usar essas técnicas para outros contextos. Espero que esse texto tenha sido útil :-)

Arquivos

<http://www.3layer.com.br/public/ejbInheritance.zip>

Referências

- 1 - EJB Inheritance, <http://www.onjava.com/pub/a/onjava/2002/09/04/ejbinherit.html>
- 2 - Pattern Adapter, <http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/adapter.htm>
- 3 - EJB, <http://java.sun.com/products/ejb>
- 4 - XDoclet, <http://xdoclet.sourceforge.net>
- 5 - Hibernate, <http://www.hibernate.org>
- 6 - JBossIDE, www.jboss.org
- 7 - LomboZ, www.lomboz.org
- 8 - Pattern ServiceLocator, <http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html>
- 9 - Java Reflection, <http://java.sun.com/developer/technicalArticles/ALT/Reflection>

Marcelo Mrack (mrac@unisc.br) é desenvolvedor e trabalha na Universidade de Santa Cruz do Sul usando Eclipse, Jboss e Hibernate para sistemas Web com Struts e Desktops com Swing.