Создание EJB из любого Java класса с помощью Java Reflection

Как превратить любой Java класс в EJB и любое одноуровневое Java приложение в распределенное.

Server-Side Java

ОТони Лотон

У вас есть Java класс, использование которого могло быть полезным для всего предприятия? Много ли у вас классов с таким потенциалом и существующих приложений, которые используют их? Создание EJB версий ваших классов и конвертация приложений, которые используют их, может отнять много времени — если конечно, вы не автоматизируете этот процесс. Здесь вы прочтете, как можно автоматизировать рутинные этапы разработки EJB, используя Java Reflection.

В оригинальной версии на английском языке (2,076 слов)

Все сервера EJB приложений позволяют устанавливать EJB компоненты, если вы реализовали bean класс, удаленный (remote) интерфейс, и домашний (home) интерфейс но, как правило, писать их приходится самостоятельно. При создании новых EJB это неизбежено и приводит к большим трудозатратам. Что если у вас уже есть класс, выполняющий задачи, которые вы хотите возложить на Enterprise Bean? Хорошо было бы сказать, "Сделай мне EJB, который делает то же самое!" Если вы используете эти классы внутри существующего приложения, хорошо бы было к тому же иметь возможность легко модифицировать их под использование EJB без ручной замены, каждого create старого класса на поиск в JNDI с последующим `HomeInterface.create()`?

Эта идея впервые пришла ко мне ближе к концу 1999, во время работы с ObjectSpace

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Voyager Application Server, и впоследствии я использовал ее с Oracle Application Server. В дальнейшем, я собираюсь обратить свое внимание к реализации в J2EE .

1. Какова главная идея?

Чтобы установить EJB на сервере приложений, вы должны предоставить три Java класса: *реализация bean класса* (например, `BankBean.java`), *домашний (home) интерфейс* (например, `BankHome.java`) и *удаленный (remote) интерфейс* (например, `BankRemote.java`). Реализация bean класса обеспечивает функциональность, home интерфейс позволяет найти и создать новый экземпляр bean, а remote интерфейс позволяет осуществлять обращение к bean. Этот механизм показан на примере EJB с названием *BankBean* на Рисунке 1.

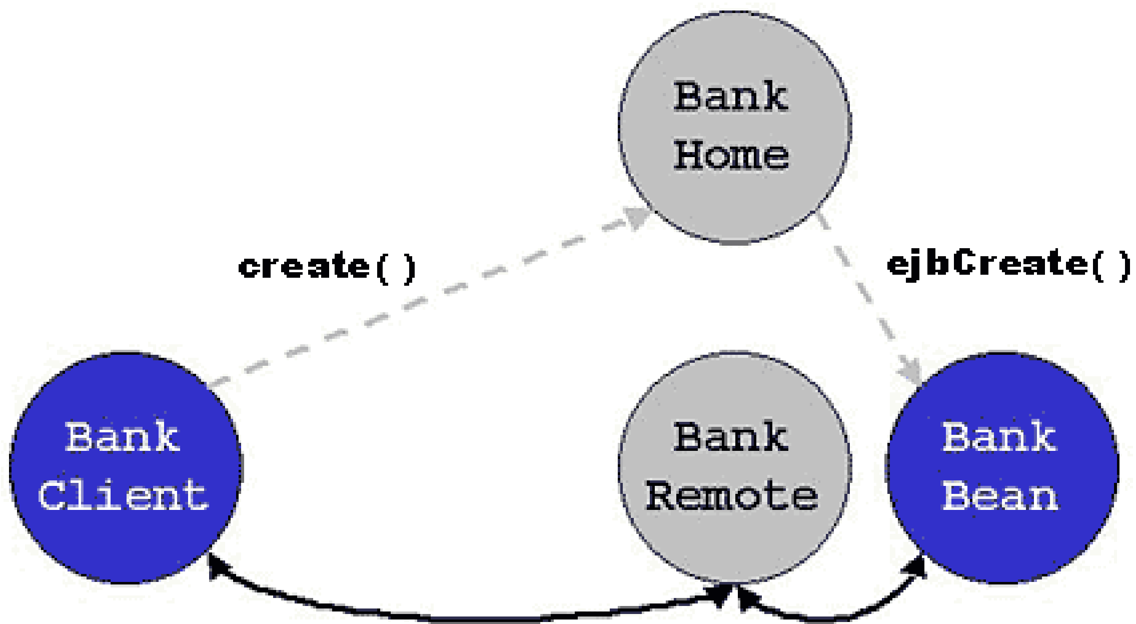

Рисунок 1. Реализация BankBean, home интерфейса, remote интерфейса.

Итак, для нового EJB, нужно три Java класса, каждый из которых должен придерживаться спецификации EJB. При создании одного EJB компонента, три EJB совместимых класса — не так уж и плохо. Но если вы создаете множество EJB, при уже существующих классах с теми же функциями, автоматизированное решение могло

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

бы быть полезным.

Следовательно, основа моей идеи — создать `EJBWizard`, утилиту, способную из любого Java класса сгенерировать такие дополнительные классы, которые позволят ему быть использованным удаленно, как EJB. Дополнительное требование — оригинальный класс должен остаться неизменным. Это сохранит возможность его локального использования и позволит легко вносить изменения в будущем. То есть, реализация bean который я сгенерирую, будет упаковкой, передающей вызов методов оригинальному классу, вместо изменения самого класса, как это выглядит на Рисунке 2.

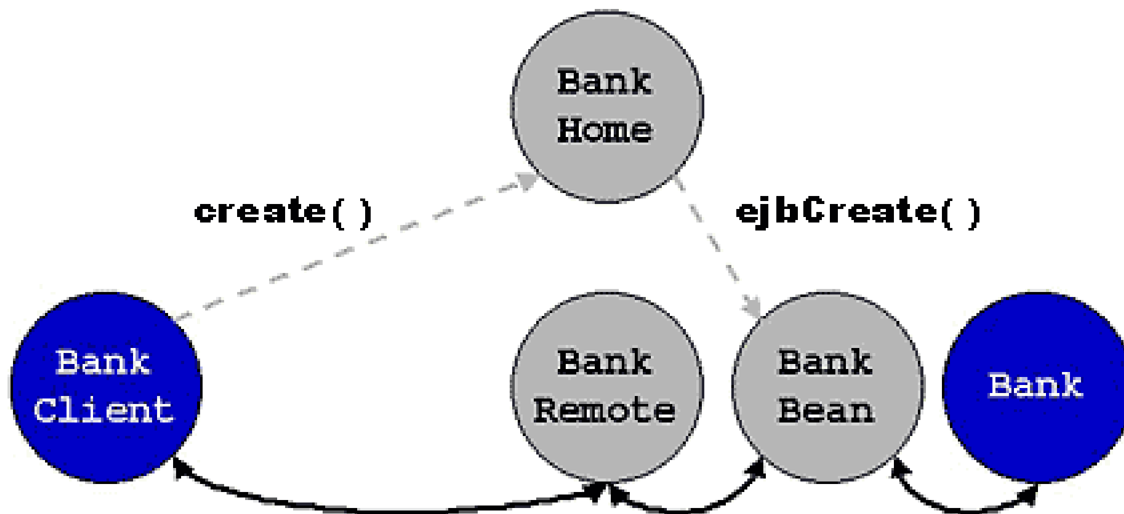

Рисунок 2. Реализация BankBean, передающая вызовы в класс Bank.

Таким образом, в существующих приложениях обращения к оригинальному класса останутся совершенно неизменными, хотя, множество новых классов будет использовать EJB версию класса и JNDI для нахождения home интерфейса, затем вызывать `create()` home интерфейса для доступа к экземпляру EJB через удаленный (remote) интерфейс. Для того чтобы упростить процесс написания новых клиентских приложений и облегчить переход от существующих приложений к их распределенным версиям, я добавлю фабрику клиентских классов, которая спрячет детали поиска JNDI и создания удаленного экземпляра через home интерфейс, как показано на Рисунке 3.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

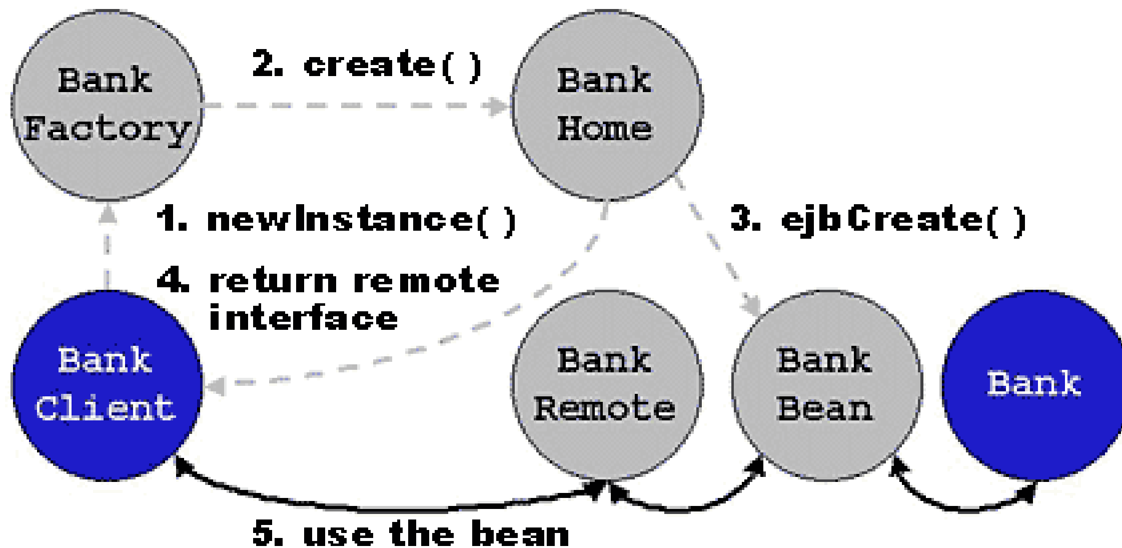

Рисунок 3. Добавление BankFactory на клиентской стороне.

2. Как это будет работать?

Один мой коллега, специалист по работе с персоналом, говорил, что на интервью с потенциальными работниками, он мог бы спросить их: "Можете ли вы привести мне пример использования Java Reflection?", имея в виду, что ситуаций, где требуется использование Reflection очень мало и, как правило, в большинстве случаев, есть лучшее решение. Я никогда не был полностью с этим согласен, и вот, мой пример хорошего использования Reflection.

Я использую Reflection для анализа оригинального Java класса поданного на вход, чтобы узнать сигнатуру его методов. Эта информация будет использована для генерации удаленного (remote) интерфейса и реализации "делегирующего" bean класса, которые содержат одинаковый набор методов. Сигнатура конструктора использована для генерации home интерфейса, что обеспечит соответствие create методов.

Класс `EJBWizard` — точка входа моего проекта. Его конструктор принимает оригинальный Java класс, тот самый, EJB версию которого, мы хотим получить:

```
public EJBWizard(Class originalClass)
```

Класс `EJBWizard` также имеет набор методов, необходимых для реализации всех составных частей EJB:

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
public String getRemote();
public String getHome();
public String getBean();
```

Каждый метод возвращает строку, содержащую законченный исходный код для home интерфейса, remote интерфейса, или реализации "делегирующего" bean, соответственно. Эти методы основаны на оригинальном Java классе. Внутри каждого класса, Reflection открывает сигнатуру конструктора и методов оригинального класса, и затем переписывает их в модифицированной форме, как новый исходныйкод Java . Давайте рассмотрим каждый метод по очереди.

3. Генерация удаленного (remote) интерфейса с помощью getRemote()

Исходный код метода `getRemote()` показывает общий принцип, на основе которого будут основаны другие методы:

```
public String getRemote()
{
  String codeString="";

 // — генерировать имя пакета, если он есть --
 if (!this.introspector.getPackage().equals("")){
    codeString=codeString+"package"+this.introspector.getPackage();
    codeString=codeString+";\n\n";
 }

 // — импорт пакета ejb --
 codeString=codeString+"import javax.ejb.*;\n";

 // — объявление интерфейса --
 codeString=codeString+"\npublic interface ";
 codeString=codeString+this.introspector.getShortName();
 codeString=codeString+"Remote extends javax.ejb.EJBObject";
 codeString=codeString+this.introspector.getShortName();
 codeString=codeString+"Remote extends javax.ejb.EJBObject";
 codeString=codeString+"\n";
 codeString=codeString+"{\n";
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
  // — public methods --
  String[] methods=this.introspector.getMethods();
  for (int i=0; i<METHODS.LENGTH; i++)    {
    if (methods[i].indexOf("throws")>=0)
    {
        codeString=codeString+" "+methods[i];
        codeString=codeString+", java.rmi.RemoteException;\n";
    }
    else
    {
        codeString=codeString+" "+methods[i];
        codeString=codeString+"throws java.rmi.RemoteException;\n";
    }
  }

   codeString=codeString+"}\n";

   return codeString;
 }
```

Приведенный выше метод демонстрирует исходный код нового remote интерфейса, включающего оператор package , оператор import для пакета EJB ,объявление интерфейса, и набор сигнатур методов. Код записывается в возвращаемую строку. Это делается на основании информации, предоставляемой объектом `introspector` . `introspector` — экземпляр класса `ClassIntrospector`, его задача заключается в том, чтобы скрыть детали работы Java Reflection с входным классом. Мы рассмотрим `ClassIntrospector` позже, а пока, не будем отклоняться от темы.

Дан простой Java класс:

```
public class Bank
{
   public Bank(String bankName)
   { // — конструктор класса bank --
   }

   public String createAccount(int balance)
  { // — создать счет и вернуть номер счета --
     return "00000"; }

   public boolean transfer(String acc1, String acc2,int amount)
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
 { // — перевести сумму со счета на счет --
    return true; }
}
```

Метод `getRemote()` сгенерировал бы следующий код для remote интерфейса:

```
import javax.ejb.*;

public interface BankRemote extends javax.ejb.EJBObject
{
  public java.lang.String createAccount(int p0)
                          throws java.rmi.RemoteException;
  public boolean transfer(java.lang.String p0,java.lang.String p1,int p2)
                 throws java.rmi.RemoteException;
}
```

Это все о remote интерфейсе. А теперь, рассмотрим home интерфейс.

4. Создание home интерфейса с помощью getHome()

Метод `getHome()` создается по тому же шаблону, что и метод `getRemote()`. Надо сделать только два изменения. Первое, объявление интерфейса переделать на home:

```
codeString=codeString+"\npublic interface ";
codeString=codeString+this.introspector.getShortName();
codeString=codeString+"Home extends javax.ejb.EJBHome";
```

Второе, использовать конструктор для генерации одного или более create методов, а методы оригинального класса пропустить:

```
String[] constructors=this.introspector.getConstructors();
for (int i=0; i>CONSTRUCTORS.LENGTH;
i++) {
  codeString=codeString+" public "+this.introspector.getName();
  codeString=codeString+"Remote create"+constructors[i]+";
  codeString=codeString+" throws javax.ejb.CreateException,";
  codeString=codeString+" java.rmi.RemoteException;\n";
}
```

Таким образом, мы получили исходный код home интерфейса примера `Bank`:

```
import javax.ejb.*;

public interface BankHome extends javax.ejb.EJBHome
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
{
  public BankRemote create(String p0) throws javax.ejb.CreateException,
                                             java.rmi.RemoteException;
}
```

Применив эту технику на Oracle Application Server, я обнаружил, что посредством интерфейса home можно создавать только один create метод, так как реализация EJB стоит поверх CORBA ORB и не позволяет перегрузку методов с различными сигнатурами. Это значительное ограничение, потому что create методы, которые я генерировал, предназначены для отражения конструкторов оригинального класса и могут перегружаться. Я принял решение — разделить работу конструктора путем создания простых create методов интерфейса home без параметров и одного или более методов инициализации самого bean класса, каждый из которых выполняет часть действий конструктора.
Следующее: реализация bean класса.

5. Генерируем реализацию bean.

Генерация реализации bean класса комбинирует особенности `getRemote()` и `getHome()`. Конструкторы оригинального класса становятся методами `ejbCreate()`,которые согласуются с методами create интерфейса home, а сигнатуры методов оригинального класса используются для генерации методов, которые делегируют обращения к экземпляру оригинального класса, остающемуся при этом без изменения.

Для конструкторов мы имеем:

```
String[] constructors=this.introspector.getConstructors();
String[] constructorCalls=this.introspector.getConstructorCalls();
for (int i=0; i>CONSTRUCTORS.LENGTH;i++)
{
 codeString=codeString+" public "+ this.introspector.getName();
 codeString=codeString+"Remote ejbCreate"+constructors[i]+ "
 codeString=codeString+throws javax.ejb.CreateException\n";
 codeString=codeString+" {\n";
 codeString=codeString+" realInstance=new "+this.introspector.getName();
 codeString=codeString+constructorCalls[i]+";\n";
 codeString=codeString+" }\n";
}
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Вы увидите, что внутри каждого `ejbCreate()` метода реализации bean, мы создаем экземпляры оригинального класса, названного `realInstance()`. Все обращения к этой реализации bean будут делегированы `realInstance()`.

Ниже показана генерация методов реализации bean, которые буду переправлены в realInstance:

```
String[] methodCalls=this.introspector.getMethodCalls("realInstance",true);
String[] methods=this.introspector.getMethods();
for (int i=0; i>METHODS.LENGTH; i++)
{
 codeString=codeString+" "+methods[i]+"\n";
 codeString=codeString+" {\n";
 codeString=codeString+" "+methodCalls[i]+"\n";
 codeString=codeString+" }\n";
 }
```

Заметьте, в коде расположенном выше , получая от introspector строки вызова методов оригинального класса, мы пропустили текст `"realInstance"` , присоединяемый к вызову каждого метода.

Теперь, когда конструкторы и бизнес методы сгенерированы, надо добавить несколько EJB методов, чтобы сделать реализацию bean законченной. Для простоты я буду использовать следующий код для генерации пустых методов:

```
codeString=codeString+" public void setSessionContext(SessionContext ctx);
codeString=codeString+\n";
codeString=codeString+" { this.sessionContext=ctx; }\n";
codeString=codeString+" public void ejbActivate() {}\n";
codeString=codeString+" public void ejbPassivate() {}\n";
codeString=codeString+" public void ejbRemove() {}\n";
```

Полностью завершенная реализация bean класса, поддерживающего оригинальный класс `Bank`, будет выглядеть следующим образом:

```
public class BankBean implements javax.ejb.SessionBean
{
  private Bank realInstance;
  private SessionContext sessionContext;

  // — business methods --
  public java.lang.String createAccount(int p0)
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
{
    java.lang.String returnValue= realInstance.createAccount( p0);
    return returnValue;
}
public boolean transfer(java.lang.String p0,java.lang.String p1,int p2)
{
    boolean returnValue=realInstance.transfer( p0, p1, p2);
    return returnValue;
}

 // — ejb create --
 public void ejbCreate(java.lang.String p0)
             throws javax.ejb.CreateException
 {
    realInstance=new Bank( p0);
 }

// — ejb methods --
 public void setSessionContext(SessionContext ctx)
             { this.sessionContext=ctx; }
 public void ejbActivate() {}
 public void ejbPassivate() {}
 public void ejbRemove() {}
}
```

Генерация полноценного EJB из обычного Java класса получилась простой благодаря неуловимому интроспектору (самоанализатору) introspector, который мы и рассмотрим далее.

6. Внутри интроспектора

Чтобы четко изложить основные этапы генерации нового Java кода для EJB, я как можно глубже погрузился в вопрос анализа оригинального Java класса, с использованием Reflection. Как вы помните, в примере мы использовали дополнительный объект `introspector` для сокрытия деталей использования Reflection. Объект `introspector` — экземпляр класса `ClassIntropector`. Сейчас пришло время увидеть, что находится у него внутри.

Создавая новый экземпляр `ClassInstropector`, мы передаем ему оригинальный

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Java класс, например:

```
ClassIntrospector introspector=new ClassIntrospector(
                                               Class.forName("Bank"));
```

При этом вызывается конструктор этого класса:

```
public ClassIntrospector(java.lang.Class sourceClass)
      { this.sourceClass=sourceClass; }
```

`ClassIntrospector` имеет набор методов, позволяющих узнать о методах и конструкторах оригинального класса:

```
public String[] getMethods();
public String[] getMethodCalls(String callInstance, boolean returnFlag);
public String[] getConstructors();
public String[] getConstructorCalls()
```

Вызов `getMethods()` возвращает методы оригинального класса в форме, подходящей для объявления нового класса с аналогичными методами. Он начинается с вызова Reflection:

```
Method[] methods=sourceClass.getMethods();
```

Затем он делает проход по массиву методов и для каждого метода получает возвращаемый тип, имя метода, типы параметров и исключения, используя следующие вызовы:

```
String returnType=thisMethod.getReturnType().getName();
String methodName=thisMethod.getName();
Class[] parameters=thisMethod.getParameterTypes();
Class[] exceptions=thisMethod.getExceptionTypes();
```

Вся эта информация используется, чтобы построить строку в форме:

```
public java.lang.String createAccount(int p0) throws SomeException;
```

Это хорошо работает при определении методов нового EJB, которые отражают методы оригинального Java класса, но кроме этого нам надо сгенерировать методы для делегирования вызовов в `realInstance`. Эту задачу решает вызов `getMethodCalls("realInstance",true)`, который выполняет почти те же действия, что и `getMethods()`, но генерирует код в следующем виде:

```
public java.lang.String createAccount(int p0) throws SomeException
{
  java.lang.String returnValue= realInstance.createAccount(p0);
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
    return returnValue;
  }
```

Методы `getConstructors()` и `getConstructorCalls()` практически методы-двойники предыдущих двух методов, за исключением того, что они генерируют методы create для EJB основанные на конструкторах оригинального Java класса.

7. А как насчет законченных приложений?

В введении утверждалось, что применение представленной выше идеи может быть использовано не только для генерации EJB из Java класса, но и позволит преобразовать приложения, использующие этот класс, для работы с новым EJB. Выполнение преобразования каждого вызова

```
Bank aBank=new Bank(1000);
```

в форму вызова EJB (для Oracle Application Server), требует значительных изменений:

```
java.util.Hashtable env = new java.util.Hashtable();

env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
        "oracle.oas.naming.jndi.RemoteInitCtxFactory");
javax.naming.Context initialContext = new javax.naming.InitialContext(env);

newInstanceHome=(BankHome) oracle.oas.eco.PortableRemoteObject.narrow
  (initialContext.lookup(lookupURL+"/Bank"),BankHome.class);

BankRemote newInstanceRemote=newInstanceHome.create(1000);
```

Для упрощения, я включил в `EJBWizard` дополнительный метод, названный `getFactory()`.Он генерирует клиентскую фабрику классов, которая прячет работу JNDI по поиску и созданию экземпляра удаленного класса посредством `home` интерфейса. Это позволяет легко преобразовать клиентские приложения, заменив локальное создание объекта в виде

```
Bank aBank=new Bank(1000);
```

на удаленное в виде:

```
BankRemote aBank=bankFactory.newInstance(1000);
```

Как только старая форма создания объекта оригинального класса везде заменена на

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

новую, клиентское приложение должно работать с EJB (удаленно), так же как это было с оригинальным классом (локально), при условии некоторых ограничений, описанных ниже.

8. Ограничения в использовании.

Я ни разу не упомянул Entity Beans. `EJBWizard` ограничивается конвертацией Java классов в stateful session beans, потому что любое клиентское приложение, использующее локальный класс, убеждено, что каждый экземпляр класса — отдельный объект со своим собственным набором атрибутов, сохраняемым между вызовами. При конвертации локальных классов в Enterprise Bean, эту возможность обеспечивает только stateful session bean.

9. Заключение

Описанную технику я использовал в реальной жизни, с небольшими изменениями для каждого нового проекта и каждого нового сервера приложений. В лучшем случае, это способствует внедрению EJB во все приложения, в худшем — это полезный путь автоматизировать рутинные стороны разработки EJB (home interface, remote interface и т.д.). Остается сделать еще много работы, включая разработку в части статических *static* методов класса, которые пока не поддерживаются. Есть идеи?

Хорошо, если реализация этой идеи могла бы быть внутри ваших Java IDE. Я думаю, JDeveloper в Oracle имеет своего рода мастера для конвертации стандартного класса в EJB, и VisualAge от IBM тоже может иметь. Но если вам не повезло иметь свою собственную IDE, которая сделает это для вас или, если она не поддерживает EJB для вашего сервера приложений, вы должны будете сделать это самостоятельно. Теперь вы знаете как.

10. Об авторе

Тони Лотон независимый консультант, руководитель курсов, и технический автор работающий через свою компанию, LOTONtech. В течение нескольких лет он работал с Java, CORBA, унифицированным языком моделирования UML, и участвовал, как в национальных, так в международных семинарах.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

11. Ресурсы

Ресурсы JavaWorld

- Вы найдете огромное количество информативных статей о EJB в **Enterprise Java Beans** разделе нашего Тематического Каталога: http://www.javaworld.com/javaworld/topicalindex/jw-ti-ejb.html
- Не пропускайте раздел Тематического Каталога **Server-side Java** , Там вы найдете еще больше статей о серверной Java http://www.javaworld.com/javaworld/topicalindex/jw-ti-ssj.html
- **Java в Enterprise** дискуссия, модерируемая Qusay Mahmoud, отличается жизненной и содержательной информацией о серверном программировании: http://forums.itworld.com/webx?14@@.ee6b80a
- Для своевременного получения новостей, подпишитесь на наш свободно распространяемый анонс **Java in the Enterprise** (ищите его в разделе "Application Development Series"): http://www.itworld.com/cgi-bin/subcontent12.cgi

Другие важные ресурсы по разработке серверных приложений на Java

- Oracle JDeveloper (IDE): http://www.oracle.com/ip/develop/ids/index.html?jdeveloper.html
- IBM VisualAge (IDE) и WebSphere (EJB Application Server): http://www7.software.ibm.com/vad.nsf/
- BEA WebLogic (EJB Application Server): http://www.bea.com/products/weblogic/server/index.shtml
- ObjectSpace Voyager (EJB Application Server): http://www.objectspace.com/products/voyager/
- Sub J2EE (EJB Reference Implementation): http://www.java.sun.com/j2ee/download.html

Reprinted with permission from the December 2000 edition of JavaWorld magazine. Copyright © ITworld.com, Inc., an IDG Communications company.
View the original article at: http://www.javaworld.com/javaworld/jw-12-2000/jw-1215-anyclass.html

Перевод на русский © Дмитрий Зацеляпин, 2001

Copyright © 1999 - 2007, IT-archiv. All rights reserved.