VII Seminarium PLOUG
Warszawa
Marzec 2003

Budowa komponentów Enterprise JavaBeans

Maciej Zakrzewicz, Marek Wojciechowski

{mzakrz, marek}@cs.put.poznan.pl

Politechnika Poznańska
Instytut Informatyki

Streszczenie

Technologia Enterprise JavaBeans (EJB) pozwala programistom Java na budowę rozproszonych komponentów aplikacyjnych, najczęściej stosowanych do implementacji tzw. logiki biznesowej. W artykule omówiono architekturę i techniki konstrukcji komponentów EJB oraz metody komunikacji z nimi z poziomu programów tworzonych w języku Java.

1. Wprowadzenie

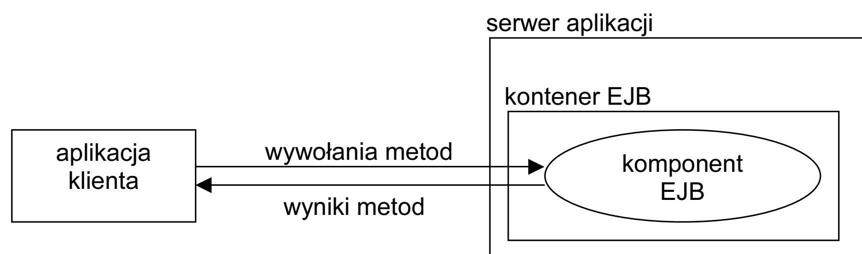
W ciągu ostatnich kilku lat obserwujemy istotny wzrost znaczenia i popularności aplikacji wielowarstwowych. Charakterystyczne dla nich rozdzielenie warstwy prezentacji od warstwy przetwarzania danych, nazywanej również warstwą *logiki biznesowej*, umożliwia redukcję obciążenia obliczeniowego stacji końcowych oraz poważnie upraszcza administrowanie aplikacją. Dostępne w przemyśle technologie budowy aplikacji wielowarstwowych dzielone są zwykle na dwie grupy: *internetowe* i *intranetowe*. W obszarze technologii internetowych, wykorzystujących przeglądarkę WWW i protokół komunikacyjny HTTP, wyróżniają się m.in. technologie serwletów Java, JavaServer Pages, Active Server Pages, PHP i CGI. Druga grupa, czyli technologie o charakterze intranetowym, wykorzystujące takie sesyjne protokoły komunikacyjne jak IIOP, obejmuje m.in. rozwiązania CORBA, DCOM i Enterprise JavaBeans.

Technologia *Enterprise JavaBeans* (EJB) jest naturalnym wyborem dla programisty Java, który planuje budowę modułowych, wielowarstowych aplikacji dla zastosowań intranetowych. Formalnie rzecz biorąc, specyfikacja Enterprise JavaBeans stanowi element standardu Java 2 Enterprise Edition (J2EE). Enterprise JavaBeans definiuje architekturę rozproszonych komponentów i klientów dla niezależnych od platformy sprzętowo-programowej aplikacji Java oraz obejmuje biblioteki programisty umożliwiające ich implementację. Specyfikacja ta została dotychczas opublikowana w trzech kolejnych wersjach: EJB 1.0, EJB 1.1 i EJB 2.0. Wersja EJB 1.0 przedstawiła założenia architektury dwóch podstawowych typów komponentów: sesyjnych i encyjnych. W wersji EJB 1.1 zmieniono format zapisu tekstowego pliku deskryptora instalacji (*deployment descriptor*) na postać XML. Wersja EJB 2.0 wprowadziła nowy, trzeci typ komponentów - komponenty komunikatowe, integrowane z usługą Java Messaging Service (JMS). Ponadto, zdefiniowano mechanizmy lokalnego dostępu do komponentu, rozszerzono mechanizmy obsługi trwałości komponentów encyjnych, wprowadzono obsługę związków pomiędzy komponentami encyjnymi oraz zdefiniowano składnię języka zapytań dla metod wyszukiwawczych komponetów encyjnych (EJB QL).

W artykule przedstawiono podstawowe założenia i funkcjonalność komponentów Enterprise JavaBeans 2.0. Przedstawiono szereg przykładów implementacji prostych operacji rozproszonego przetwarzania danych przy wykorzystaniu sesyjnych i encyjnych komponentów EJB. Struktura tego artykułu jest następująca. W rozdziale drugim przedstawiono podstawowe założenia architektury Enterprise JavaBeans. Rozdział trzeci omawia metodykę budowy sesyjnych komponentów EJB. Rozdział czwarty poświęcony jest metodyce budowy encyjnych komponentów EJB. Rozdział piąty zawiera podsumowanie.

2. Elementy architektury Enterprise JavaBeans

Program Java realizowany w technologii Enterprise JavaBeans składa się z dwóch rodzajów elementów składowych: lokalnej *aplikacji klienta* i zdalnych *komponentów przetwarzania danych*. Aplikacja klienta komunikuje się poprzez sieć komputerową z komponentami przetwarzania danych, przesyłając do nich żądania zdalnego wykonania metod. Wyniki działania tych metod są następnie tą samą drogą odsyłane do aplikacji klienta. Aplikacja klienta jest implementowana jako tradycyjna samodzielna aplikacja Java, jako serwlet Java lub jako inny komponent EJB. Zdalny komponent EJB jest implementowany w formie odrębnej klasy Java, której obiekty są obsługiwane przez specjalne środowisko uruchomieniowe, nazywane kontenerem EJB, osadzonym na platformie serwera aplikacji. Na rysunku 1 przedstawiono ogólną architekturę aplikacji zbudowanej zgodnie ze specyfikacją Enterprise JavaBeans.



Rys. 1. Ogólna architektura Enterprise JavaBeans

Istotną cechą architektury Enterprise JavaBeans jest przezroczystość lokalizacji i implementacji komponentów EJB z punktu widzenia aplikacji klienta. Referencja do każdego komponentu jest rejestrowana w serwisie nazw, którego przeszukiwanie aplikacja klienta realizuje przy użyciu interfejsu JNDI.

Bezpośrednim środowiskiem uruchomieniowym dla każdego komponentu Enterprise JavaBeans jest *kontener EJB*. Kontener EJB całkowicie pośredniczy w komunikacji pomiędzy komponentem EJB a światem zewnętrznym. Ponadto, kontener EJB oferuje komponentowi szereg usług o charakterze systemowym: ochrona dostępu, obsługa transakcji, itp. Komponent EJB może uzyskać dostęp do usług kontenera EJB korzystając z mechanizmu typu *callback* - w chwili powoływania do życia obiektu komponentu EJB, kontener EJB przekazuje komponentowi EJB pewien rodzaj uchwytu zwrotnego. Za pomocą tego uchwytu obiekt komponentu EJB może wykonywać wywołania metod kontenera EJB.

Specyfikacja Enterprise JavaBeans definiuje trzy typy komponentów EJB: *sesyjne* (Session Bean), *encyjne* (Entity Bean) i *komunikatowe* (Message-Driven Bean). Sesyjny komponent EJB to krótkotrwały obiekt wykorzystywany przez pojedynczą aplikację klienta i nie współdzielony z innymi aplikacjami, stanowiący logiczne rozszerzenie kodu aplikacji klienta umieszczone po stronie serwera aplikacji. Encyjny komponent EJB reprezentuje dane, które są trwale przechowywane w systemie bazy danych. Cechuje się długim czasem życia, pozwala na atomowe, transakcyjne modyfikacje bazy danych, jest współdzielony przez wielu klientów i dostępny dla wielu sesji. Ostatni typ komponentu EJB - komponent komunikatowy - jest asynchronicznym konsumentem komunikatów JMS pochodzących ze środowisk kolejkowych. Jest uruchamiany wtedy, kiedy nadchodzi komunikat od klienta, jest wykonywany asynchronicznie, może modyfikować zawartość bazy danych, jest bezstanowy.

Proces konstrukcji aplikacji Enterprise JavaBeans przez programistę jest złożony i wieloetapowy. W pierwszym kroku, programista przygotowuje kod źródłowy klasy Java, reprezentującej komponent EJB. W kroku drugim, tworzy dwa interfejsy Java, reprezentujące punkty kontaktu świata zewnętrznego z komponentem EJB: interfejs Home, służący do zarządzania cyklem życia komponentu EJB, oraz interfejs Remote/Local, służący do wywoływania metod logiki biznesowej komponentu EJB. W trzecim kroku, programista przygotowuje XML-owy plik konfiguracyjny nazywany *deskryptorem instalacji* (Deployment Descriptor). Deskryptor instalacji opisuje m.in. wymagania dotyczące sposobu obsługi komponentu przez kontener EJB, sposób implementacji trwałości komponentu encyjnego, deklaratywne reguły obsługi transakcji oraz kontrolę dostępu do operacji realizowanych przez komponent EJB. Po tym etapie dokonywana jest kompilacja całego przygotowanego kodu Java i tworzony z niego jest plik JAR/EAR o specjalnej wewnętrznej strukturze katalogów. W kolejnym kroku, programista posługuje się narzędziami serwera aplikacji służącymi do instalacji komponentów EJB i umieszcza przygotowane pliki w systemie plików serwera aplikacji. Narzędzia te dokonują zwykle również rejestracji komponentu EJB w lokalnym serwisie nazw, dostępnym przez JNDI. Co istotne, w wyniku uruchomienia narzędzi instalacyjnych automatycznie generowany jest kod Java pomocniczych klas do wykorzystania podczas budowy aplikacji

klienta. Wreszcie, w kroku ostatnim, programista przygotowuje kod aplikacji klienta oraz dokonuje jej kompilacji i uruchomienia.

3. Tworzenie sesyjnych komponentów EJB

Jak już wspomniano, sesyjne komponenty EJB mogą być traktowane jako logiczna kontynuacja kodu aplikacji klienta, fizycznie zlokalizowana jednak po stronie serwera aplikacji. Komponenty takie żyją jedynie przez czas trwania sesji aplikacji klienta. Ich zastosowanie przypomina klasyczne rozwiązania mechanizmów zdalnego wołania procedury RPC. Specyfikacja Enterprise JavaBeans opisuje dwa rodzaje sesyjnych komponentów EJB: stanowe i bezstanowe. *Stanowe* (stateful) komponenty EJB są przez cały czas życia skojarzone z jedną i tą samą aplikacją klienta i pamiętają swój stan obiektu. *Bezstanowe* (stateless) sesyjne komponenty EJB nie gwarantują pamiętania swojego stanu obiektu i mogą być przy każdym wywołaniu wykorzystywane przez inną aplikację klienta.

Sesyjne komponenty EJB mogą być wykorzystywane przez dwa rodzaje aplikacji klienta: *aplikacje klienta zdalnego* lub *aplikacje klienta lokalnego*. Aplikacja klienta zdalnego to taka, która pracuje na innej maszynie wirtualnej niż kontener EJB. Uzyskuje ona dostęp do komponentu poprzez interfejsy Remote i Remote Home (przez RMI). Umożliwia to osiągnięcie niezależność fizycznej lokalizacji komponentu EJB od lokalizacji klienta. Drugi rodzaj aplikacji klienta to aplikacja klienta lokalnego - pracującego na tej samej maszynie wirtualnej co kontener EJB. Uzyskuje on dostęp do komponentu EJB poprzez interfejsy Local i Local Home (bezpośrednie wywołanie metody zamiast RMI).

W celu zilustrowania zasad konstrukcji sesyjnych komponentów EJB, zrealizujemy przykładową aplikację, która umożliwi klientowi zdalną realizację operacji mnożenia dwóch liczb. Klient przekaże komponentowi EJB dwie liczby zmiennoprzecinkowe, a odbierze ich iloczyn. Mnożenie zostanie fizycznie zrealizowane przez serwer aplikacji.

Interfejs Remote

Programista definiuje interfejs Java, dziedziczony z EJBObject, nazywany *interfejsem Remote*. Interfejs ten opisuje wszystkie metody logiki biznesowej komponentu EJB, które będą udostępniane zdalnemu klientowi. W naszym przypadku będzie to jedna metoda, nazwana *multiply()*. Opisywane w interfejsie metody muszą propagować wyjątek *RemoteException*. Poniżej przedstawiono kompletny kod źródłowy interfejsu Remote.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Multiplicator1Remote extends EJBObject
{
  float multiply(float factor1, float factor2) throws RemoteException;
}
```

Interfejs Home

Programista definiuje interfejs Java, dziedziczony z EJBHome, nazywany *interfejsem Home*. Interfejs ten opisuje wszystkie metody zarządzania cyklem życia komponentu EJB (tworzenie, usuwanie). W naszym przykładzie interfejs ten pełni charakter czysto formalny - zawiera metodę *create()*, aktywowaną w chwili tworzenia obiektu komponentu EJB. Metoda *create()* musi propagować wyjątki *RemoteException* i *CreateException*. Poniżej przedstawiono kompletny kod źródłowy interfejsu Home.

```
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
```

```
public interface Multiplicator1Home extends EJBHome
{
  Multiplicator1Remote create() throws RemoteException, CreateException;
}
```

Klasa komponentu EJB

Programista implementuje klasę Java, dziedziczoną z SessionBean, nazywaną *klasą komponentu EJB*. Klasa ta implementuje metody logiki biznesowej zadeklarowane wcześniej w interfejsie Remote. Ponadto, programista ma obowiązek zadeklarowania metod, które nie są istotne z punktu widzenia naszego przykładu, lecz mogłyby służyć do zaawansowanej inicjalizacji komponentu, obsługi jego wywłaszczania oraz wykorzystywania usług kontenera EJB: *ejbCreate()*, *ejbActivate()*, *ejbPassivate()*, *ejbRemove()*, *setSessionContext()*. Poniżej przedstawiono kompletny kod źródłowy klasy komponentu EJB.

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class Multiplicator1Bean implements SessionBean
{
  public void ejbCreate()  {}
  public void ejbActivate()  {}
  public void ejbPassivate()  {}
  public void ejbRemove()  {}
  public void setSessionContext(SessionContext ctx)  {}
  public float multiply(float factor1, float factor2) {
    return factor1 * factor2;
  }
}
```

Plik deskryptora instalacji

Programista przygotowuje opisowy plik XML zawierający odnośniki do wcześniej utworzonych klas i interfejsów (<home>, <remote> i <ejb-class>) oraz nazwę, pod jaką komponent EJB zostanie zarejestrowany w serwisie nazw dostępnym poprzez JNDI. Znacznik <session-type> wskazuje, czy opisywany komponent będzie pracować jako stanowy czy bezstanowy. Plik deskryptora instalacji formalnie wiąże ze sobą tworzone dotychczas: interfejs Home, interfejs Remote i klasę komponentu EJB. Poniżej przedstawiono kompletny kod pliku deskryptora instalacji.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise Java-
Beans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
   <enterprise-beans>
      <session>
         <description>Session Bean ( Stateless )</description>
         <display-name>Multiplicator1</display-name>
         <ejb-name>Multiplicator1</ejb-name>
         <home>Multiplicator1Home</home>
         <remote>Multiplicator1Remote</remote>
         <ejb-class>Multiplicator1Bean</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
      </session>
```

```
   </enterprise-beans>
</ejb-jar>
```

W tym momencie programista jest gotowy do przeprowadzenia instalacji komponentu EJB na serwerze aplikacji. Procedura instalacyjna jest zależna od platformy, lecz w ogólności sprowadza się do spakowania skompilowanego kodu Java do pliku JAR/EAR, a następnie użyciu zewnętrznego programu instalacyjnego.

Kod aplikacji klienta

Podczas instalowania komponentu EJB na serwerze aplikacji, narzędzie instalacyjne automatycznie generuje dla programisty implementacje klas pomocniczych do użycia w budowie aplikacji klienta. Z pomocą tych klas, programista przygotowuje kod aplikacji klienta, która, w pierwszej kolejności nawiązuje połączenie z serwisem nazw, pozyskuje tzw. *obiekt Home* dla komponentu EJB (automatycznie wygenerowany w oparciu o interfejs Home), następnie powołuje do życia obiekt komponentu EJB w kontenerze EJB, wreszcie wywołuje jego metody. W poniższym przykładzie fragmentu aplikacji, kontener EJB pracuje na komputerze *miner*, serwis nazw jest dostępny na porcie o numerze 1811, a zainstalowany komponent *multiplicator1* jest umieszczony wewnątrz aplikacji J2EE o nazwie *ejbapp1*. Klient wywołuje metodę *multiply()* obiektu komponentu EJB i wyświetla wynik mnożenia dwóch liczb.

```
Multiplicator1Home multiplicator1Home;
Multiplicator1Remote multiplicator1Remote;

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.evermind.server.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.PROVIDER_URL,
       "ormi://miner:1811/ejbapp1");
Context ctx = new InitialContext(env);

multiplicator1Home = (Multiplicator1Home)ctx.lookup("multiplicator1");
multiplicator1Remote = multiplicator1Home.create();
System.out.println(multiplicator1Remote.multiply(2,2));
```

Zwróćmy uwagę na to, że w aplikacji klienta, wywołanie metod logiki biznesowej zdalnego komponentu EJB odbywa się tak, jakby komponent ten był w istocie lokalnym obiektem w programie Java.

4. Tworzenie encyjnych komponentów EJB

Encyjne komponenty EJB służą programistom do uzyskania efektu trwałości obiektów programowych Java poprzez ich automatyczne fizyczne składowanie w bazie danych. Komponenty takie żyją aż do momentu jawnego ich usunięcia. Mogą być wykorzystywane do budowy warstwy komunikacji logiki biznesowej z bazą danych lub też do implementacji takiej logiki biznesowej, która intensywnie eksploatuje bazę danych. Istotne jest również to, że pojedynczy encyjny komponent EJB może być współdzielony przez wiele aplikacji klienta.

W zależności od mechanizmów implementacji trwałości, encyjne komponenty EJB dzielą się na dwa typy: *z trwałością obsługiwaną przez komponent* (BMP - Bean-Managed Persistency) i *z trwałością obsługiwaną przez kontener EJB* (CMP - Container-Managed Persistency). Komponenty BMP muszą same umieszczać swój stan w bazie danych, korzystając np. z biblioteki JDBC lub SQLJ. Komponenty CMP polegają na funkcjonalności kontenera EJB, którego zadaniem jest w

tym przypadku automatyczne zapisywanie i odczytywanie stanu komponentu do/z bazy danych. Podobnie jak komponenty sesyjne, encyjne komponenty EJB również mogą być wykorzystywane przez dwa rodzaje aplikacji klienta: lokalne i zdalne.

W celu zilustrowania zasad konstrukcji sesyjnych komponentów EJB z trwałością obsługiwaną przez kontener EJB, zrealizujemy przykładową aplikację, która umożliwi klientowi zdalną realizację przetwarzania danych o pracownikach. Trwałość będzie obsługiwana przez kontener EJB, a obiekty będą zapisywane w postaci rekordów tabeli *Emp*. Struktura tabeli *Emp* została przedstawiona poniżej.

```
EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
----- ---------- --------- ---------- --------- ---------- ---------- ----------
 7369 SMITH      CLERK           7902 17-DEC-80        800                    20
 7499 ALLEN      SALESMAN        7698 20-FEB-81       1600        300         30
 7521 WARD       SALESMAN        7698 22-FEB-81       1250        500         30
 ...
```

Interfejs Remote

Programista definiuje interfejs Java, dziedziczony z EJBObject, nazywany *interfejsem Remote*. Interfejs ten opisuje wszystkie metody zdalnego dostępu do danych reprezentowanych przez encyjny komponent EJB W naszym przypadku będą to metody dostępowe do pól rekordu opisującego pracownika. Opisywane w interfejsie metody muszą propagować wyjątek *RemoteException*. Poniżej przedstawiono kompletny kod źródłowy interfejsu Remote.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Employee2Remote extends EJBObject
{
  long getEmpno() throws RemoteException;
  void setEmpno(long newEmpno) throws RemoteException;
  String getEname() throws RemoteException;
  void setEname(String newEname) throws RemoteException;
  String getJob() throws RemoteException;
  void setJob(String newJob) throws RemoteException;
  long getMgr() throws RemoteException;
  void setMgr(long newMgr) throws RemoteException;
  String getHiredate() throws RemoteException;
  void setHiredate(String newHiredate) throws RemoteException;
  long getSal() throws RemoteException;
  void setSal(long newSal) throws RemoteException;
  long getComm() throws RemoteException;
  void setComm(long newComm) throws RemoteException;
  long getDeptno() throws RemoteException;
  void setDeptno(long newDeptno) throws RemoteException;
}
```

Interfejs Home

Programista definiuje interfejs Java, dziedziczony z EJBHome, nazywany *interfejsem Home*. Interfejs ten opisuje wszystkie metody zarządzania cyklem życia komponentu EJB (tworzenie, usuwanie, wyszukiwanie istniejącego). Kod źródłowy tych metod zostanie automatycznie wygenerowany przez narzędzie instalacyjne. Metody muszą propagować wyjątki *RemoteException* i *CreateException/FinderException*. Poniżej przedstawiono kompletny kod źródłowy interfejsu Home.

```
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
```

```
import javax.ejb.FinderException;
import java.util.Collection;

public interface Employee2Home extends EJBHome
{
  Employee2Remote create()
                  throws RemoteException, CreateException;
  Employee2Remote create(long empno)
                  throws RemoteException, CreateException;
  Employee2Remote findByPrimaryKey(Employee2RemotePK primaryKey)
                  throws RemoteException, FinderException;
  Collection findAll() throws RemoteException, FinderException;
  Employee2Remote findByName(String val)
                  throws RemoteException, FinderException;
}
```

Użyta w przykładzie klasa Employe2RemotePK służy do reprezentacji klucza podstawowego z tabeli *Emp*. Definicja tej klasy została przedstawiona poniżej.

```
import java.io.Serializable;

public class Employee2RemotePK implements Serializable
{
  public long empno;

  public Employee2RemotePK() {}
  public Employee2RemotePK(long empno){this.empno = empno;}
  public boolean equals(Object other){return super.equals(other);}
  public int hashCode() {return super.hashCode();}
}
```

Klasa komponentu EJB
Programista implementuje klasę Java, dziedziczoną z EntityBean, nazywaną *klasą komponentu EJB*. Klasa ta implementuje metody dostępu do danych reprezentowanych przez encyjny komponent EJB, zadeklarowane wcześniej w interfejsie Remote. Ponadto, programista ma obowiązek zadeklarowania metod, które nie są istotne z punktu widzenia naszego przykładu, lecz mogłyby służyć do zaawansowanej inicjalizacji komponentu, obsługi jego trwałości i wywłaszczania oraz wykorzystywania usług kontenera EJB: *ejbCreate()*, *ejbPostCreate(), ejbActivate()*, *ejbPassivate()*, *ejbRemove()*, *ejbLoad(), ejbStore(), setEntityContext(), unsetEntityContext()*. Poniżej przedstawiono kompletny kod źródłowy klasy komponentu EJB.

```
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public class Employee2Bean implements EntityBean
{
  public long empno;
  public String ename;
  public String job;
  public long mgr;
  public String hiredate;
  public long sal;
  public long comm;
  public long deptno;

  public Employee2RemotePK ejbCreate() {return null;}
```

```
  public void ejbPostCreate() {}
  public Employee2RemotePK ejbCreate(long empno)
  {
    this.empno = empno;
    return new Employee2RemotePK(empno);
  }
  public void ejbPostCreate(long empno) {}
  public void ejbActivate() {}
  public void ejbLoad() {}
  public void ejbPassivate() {}
  public void ejbRemove() {}
  public void ejbStore() {}
  public void setEntityContext(EntityContext ctx) {}
  public void unsetEntityContext() {}
  public long getEmpno() {return empno;}
  public void setEmpno(long newEmpno) {empno = newEmpno;}
  public String getEname() {return ename;}
  public void setEname(String newEname) {ename = newEname;}
  public String getJob() {return job;}
  public void setJob(String newJob) {job = newJob;}
  public long getMgr() {return mgr;}
  public void setMgr(long newMgr) {mgr = newMgr;}
  public String getHiredate() {return hiredate;}
  public void setHiredate(String newHiredate) {hiredate = newHiredate;}
  public long getSal() {return sal;}
  public void setSal(long newSal) {sal = newSal;}
  public long getComm() {return comm;}
  public void setComm(long newComm) {comm = newComm;}
  public long getDeptno() {return deptno;}
  public void setDeptno(long newDeptno) {deptno = newDeptno;}
}
```

Plik deskryptora instalacji

Programista przygotowuje opisowy plik XML zawierający odnośniki do wcześniej utworzonych klas i interfejsów (<home>, <remote> i <ejb-class>) oraz nazwę, pod jaką komponent EJB zostanie zarejestrowany w serwisie nazw dostępnym poprzez JNDI. Znacznik <persistence-type> wskazuje, czy trwałość opisywanego komponentu EJB będzie obsługiwana przez kontener czy przez sam komponent. Wszystkie trwałe elementy komponentu EJB są zadeklarowane przy użyciu znaczników <cmp-field> - będą one automatycznie zapisywane w bazie danych przez kontener EJB. Plik deskryptora instalacji formalnie wiąże ze sobą tworzone dotychczas: interfejs Home, interfejs Remote i klasę komponentu EJB. Poniżej przedstawiono kompletny kod pliku deskryptora instalacji.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise Java-
Beans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
<enterprise-beans>
 <entity>
  <description>Encyjny komponent EJB - CMP</description>
  <display-name>Employee</display-name>
  <ejb-name>Employee</ejb-name>
  <home>Employee2Home</home>
  <remote>Employee2Remote</remote>
  <ejb-class>Employee2Bean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>Employee2RemotePK</prim-key-class>
  <reentrant>False</reentrant>
```

```
  <cmp-field><field-name>empno</field-name></cmp-field>
  <cmp-field><field-name>ename</field-name></cmp-field>
  <cmp-field><field-name>job</field-name></cmp-field>
  <cmp-field><field-name>mgr</field-name></cmp-field>
  <cmp-field><field-name>hiredate</field-name></cmp-field>
  <cmp-field><field-name>sal</field-name></cmp-field>
  <cmp-field><field-name>comm</field-name></cmp-field>
  <cmp-field><field-name>deptno</field-name></cmp-field>
 </entity>
</enterprise-beans>
</ejb-jar>
```

W przypadku instalowania encyjnych komponentów EJB, plik deskryptora instalacji jest uzupełniany przez dodatkowy plik opisujący odwzorowanie atrybutów komponentu EJB w struktury obiektowo-relacyjnej bazy danych. Poniżej przedstawiono przykład takiego pliku dla serwera Orion.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE orion-ejb-jar PUBLIC "-//Evermind//DTD Enterprise JavaBeans 1.1
runtime//EN" "http://xmlns.oracle.com/ias/dtds/orion-ejb-jar.dtd">
<orion-ejb-jar>
 <enterprise-beans>
   <entity-deployment name="Employee"
                      data-source="jdbc/Connection1DS" table="EMP">
    <primkey-mapping>
     <cmp-field-mapping>
      <fields>
       <cmp-field-mapping name="empno" persistence-name="EMPNO"
                          persistence-type="NUMBER(4)"/>
      </fields>
     </cmp-field-mapping>
    </primkey-mapping>
    <cmp-field-mapping name="empno" persistence-name="EMPNO"
                       persistence-type="NUMBER(4)"/>
    <cmp-field-mapping name="ename" persistence-name="ENAME"
                       persistence-type="VARCHAR2(10)"/>
    <cmp-field-mapping name="job" persistence-name="JOB"
                       persistence-type="VARCHAR2(9)"/>
    <cmp-field-mapping name="mgr" persistence-name="MGR"
                       persistence-type="NUMBER(4)"/>
    <cmp-field-mapping name="hiredate" persistence-name="HIREDATE"
                       persistence-type="DATE"/>
    <cmp-field-mapping name="sal" persistence-name="SAL"
                       persistence-type="NUMBER(7,2)"/>
    <cmp-field-mapping name="comm" persistence-name="COMM"
                       persistence-type="NUMBER(7,2)"/>
    <cmp-field-mapping name="deptno" persistence-name="DEPTNO"
                       persistence-type="NUMBER(2)"/>
    <finder-method partial="True" query="ename=$1">
     <method>
      <ejb-name>Employee</ejb-name>
      <method-name>findByName</method-name>
      <method-params>
       <method-param>java.lang.String</method-param>
      </method-params>
     </method>
    </finder-method>
   </entity-deployment>
```

```
 </enterprise-beans>
</orion-ejb-jar>
```

Kod aplikacji klienta

Podczas instalowania komponentu EJB na serwerze aplikacji, narzędzie instalacyjne automatycznie generuje dla programisty implementacje klas pomocniczych do użycia w budowie aplikacji klienta. Z pomocą tych klas, programista przygotowuje kod aplikacji klienta, która, w pierwszej kolejności nawiązuje połączenie z serwisem nazw, pozyskuje tzw. *obiekt Home* dla komponentu EJB (automatycznie wygenerowany w oparciu o interfejs Home), następnie pobiera obiekt komponentu EJB, wreszcie wywołuje metodę wyszukiwania pracownika według nazwiska. W poniższym przykładzie fragmentu aplikacji, kontener EJB pracuje na komputerze *miner*, serwis nazw jest dostępny na porcie o numerze 1811, a zainstalowany komponent *Employee* jest umieszczony wewnątrz aplikacji J2EE o nazwie *ejbapp2*. Klient wyświetla nazwę stanowiska i pensję pracownik o nazwisku King, a następnie zmienia wartość jego pensji na 550.

```
Employee2Home employee2Home;
Employee2Remote e;

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.evermind.server.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.PROVIDER_URL,
        "ormi://miner:1811/ejbapp2");
Context ctx = new InitialContext(env);

employee2Home = (Employee2Home)ctx.lookup("Employee");
e = (Employee2Remote)employee2Home.findByName("KING");

System.out.println(e.getJob());
System.out.println(e.getSal());
e.setSal(550);
```

5. Podsumowanie

Technologia Enterprise JavaBeans umożliwia budowę rozproszonych programów Java, składających się z aplikacji klienta i jednego lub wielu zdalnych komponentów EJB. Powszechnym zastosowaniem tej technologii jest konstrukcja intranetowych aplikacji wielowarstwowych, w których aplikacja klienta odpowiada za obsługę graficznego interfejsu użytkownika, jedna lub dwie warstwy komponentów EJB realizują logikę biznesową i dostęp do bazy danych, a serwer bazy danych pełni swoje tradycyjne funkcje. Praca wejścia programisty rozpoczynającego korzystanie z Enterprise JavaBeans jest znaczna, lecz korzyści wynikające z możliwości balansowania obciążenia obliczeniowego konstruowanych aplikacji wystarczająco pokrywają ten koszt.

6. Literatura

[1] Nirva Morisseau-Leroy, "Oracle9iAS Building J2EE Applications", Oracle Press, 2002
[2] Oracle, "Oracle9iAS Containers for J2EE", dokumentacja techniczna, 2002
[3] java.sun.com, "Enterprise JavaBeans 2.1 Specification", 2002