Технологии программирования. Компонентный подход

В. В. Кулямин

Лекция 14. Разработка различных уровней Web-приложений в J2EE и .NET

Аннотация

Рассматриваются используемые в рамках Java Enterprise Edition и .NET техники разработки компонентов Web-приложений, связывающих приложение с базой данных и представляющих собой элементы пользовательского интерфейса.

Ключевые слова

Компонент EJB, компонент данных, сеансовый компонент, компонент, управляемый сообщениями, протокол HTTP, сервлет Java, серверная страница Java (JSP), Web-форма .NET.

Текст лекции

Общая архитектура Web-приложений

В данной лекции мы рассмотрим техники разработки компонентов Web-приложений на основе платформ J2EE и .NET. Общая архитектура такого приложения может быть представлена схемой, изображенной на Рис. 74. Обе платформы предоставляют специальную поддержку для разработки компонентов на двух уровнях: уровне интерфейса пользователя (WebUI) и уровне связи с данными.

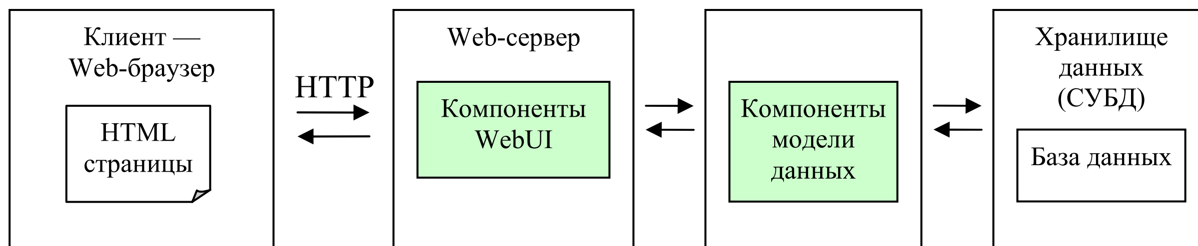

Рисунок 74. Общая схема архитектуры Web-приложений J2EE и .NET.

Пользовательский интерфейс Web-приложений основан на генерации динамических страниц HTML, содержащих данные, которые запрашивает пользователь. Уровень модели данных предоставляет приложению возможность работать с данными, обычно хранящимися в виде набора таблиц и связей между ними, как с набором связанных объектов.

Основные отличия между техниками разработки компонентов этих двух уровней, используемыми в рамках J2EE и .NET, можно сформулировать следующим образом.

- В J2EE компоненты EJB предназначены не только для представления данных приложения в виде объектов, но и для реализации его бизнес-логики, т.е. объектов предметной области и основных способов работы с ними.
  В .NET нет специально выделенного вида компонентов, предназначенного для реализации бизнес-логики — она может реализовываться с помощью обычных классов, что часто удобнее. Это положение должно измениться с выходом EJB 3.0.
- EJB компоненты являются согласованным с объектно-ориентированным подходом представлением данных приложения. Работа с ними организуется так же, как с объектами обычных классов (с точностью до некоторых деталей).
  В .NET-приложениях все предлагаемые способы представления данных являются объектными обертками вокруг реляционного представления — в любом случае приходится

работать с данными как с набором таблиц. В .NET нет автоматической поддержки их преобразования в систему взаимосвязанных объектов и обратно.

Уровень бизнес-логики и модели данных в J2EE

В рамках приложений, построенных по технологии J2EE связь с базой данных и бизнес-логику, скрытую от пользователя, принято реализовывать с помощью компонентов Enterprise JavaBeans. На момент написания этой лекции последней версией технологии EJB является версия 2.1, в первой половине 2006 года должны появиться инструменты для работы с EJB 3.0 (в рамках J2EE 5.0).

Возможны и другие способы реализации этих функций. Например, бизнес-логика может быть реализована непосредственно в методах объектов пользовательского интерфейса, а обмен данными с базой данных — через интерфейс JDBC. При этом, однако, теряется возможность переиспользования функций бизнес-логики в разных приложениях на основе единой базы данных, а также становится невозможным использование автоматических транзакций при работе с данными. Транзакции в этом случае нужно организовывать с помощью явных обращений к JTA (см. предыдущую лекцию).

Компонент Enterprise JavaBeans (EJB) является компонентом, представляющим в J2EE-приложении элемент данных или внутренней, невидимой для пользователя логики приложения. Для компонентов EJB определен жизненный цикл в рамках рабочего процесса приложения — набор состояний, через которые проходит один экземпляр такого компонента. Компоненты EJB работают внутри EJB-контейнера, являющегося для них компонентной средой. Функции EJB-контейнера следующие.

- Управление набором имеющихся EJB-компонентов, например, поддержкой пула компонентов для обеспечения большей производительности, а также жизненным циклом каждого отдельного компонента, в частности, его инициализацией и уничтожением.
- Передача вызовов между EJB-компонентами, а также их удаленных вызовов. Несколько EJB-контейнеров, работающих на разных машинах, обеспечивают взаимодействие наборов компонентов, управляемых каждым из них.
- Поддержка параллельной обработки запросов.
- Поддержка связи между EJB-компонентами и базой данных приложения и синхронизация их данных.
- Поддержка целостности данных приложения с помощью механизма транзакций.
- Защита приложения с помощью механизма ролей: передача прав ролей при вызовах между компонентами и проверка допустимости обращения в рамках роли к методам компонентов.

Для разработки набора компонентов EJB нужно, во-первых, для каждого компонента создать один или несколько классов и интерфейсов Java, обеспечивающих реализацию самой функциональности компонента и определение интерфейсов для удаленных обращений к нему, и, во-вторых, написать дескриптор развертывания — XML-файл, описывающий следующее.

- Набор EJB-компонентов приложения.
- Совокупность элементов кода на Java, образующих один компонент.
- Связь свойств компонента с полями таблиц БД и связями между таблицами.
- Набор ролей, правила доступа различных ролей к методам компонентов, правила передачи ролей при вызовах одними компонентами других.
- Политику компонентов и их методов по отношению к транзакциям.
- Набор ресурсов, которыми компоненты могут пользоваться в своей работе.

Правила создания EJB компонента зависят от его вида. Различают три таких вида EJB-компонентов.

- ***Компоненты данных (сущностные, entity beans)***.
  Представляют данные приложения и основные методы работы с ними.
- ***Сеансовые компоненты (session beans)***.
  Представляют независимую от пользовательского интерфейса и конкретных типов данных логику работы приложения, называемую иногда бизнес-логикой.
- ***Компоненты, управляемые сообщениями (message driven beans)***.
  Тоже предназначены тоже для реализации бизнес-логики. Но, если сеансовые компоненты предоставляют интерфейс для синхронных вызовов, компоненты, управляемые сообщениями, предоставляют асинхронный интерфейс. Клиент, вызывающий метод в сеансовом компоненте, ждет, пока вызванный компонент не завершит свою работу. Компоненту же, управляемому сообщениями, можно отослать сообщение и продолжать работу сразу после окончания его передачи, не дожидаясь окончания его обработки.

Далее описываются основные правила построения EJB компонентов разных видов. Более детальное описание этих правил можно найти в [1,2].

Компоненты данных и сеансовые компоненты

Компонент данных или сеансовый компонент могут состоять из следующих элементов: пара интерфейсов для работы с самим компонентом — *удаленный интерфейс* и *локальный интерфейс*; пара интерфейсов для поиска и создания компонентов — *удаленный исходный интерфейс* и *локальный исходный интерфейс*; *класс компонента*, реализующий методы работы с ним; и, для компонентов данных, — *класс первичного ключа*. Обязательно должен быть декларирован класс компонента и один из интерфейсов — удаленный или локальный. Для компонентов данных обязательно должен быть определен класс первичного ключа.

- *Удаленный интерфейс (remote interface)*.
  Этот интерфейс декларирует методы компонента, к которым можно обращаться удаленно, т.е. из компонентов, работающих в рамках другого процесса или на другой машине.
  Удаленный интерфейс должен наследовать интерфейс `javax.ejb.EJBObject` (в свою очередь, наследующий `java.rmi.Remote`).
  Для компонента данных он определяет набор свойств (в смысле JavaBeans, т.е. пар методов `Type getName()`/`void setName(Type)`), служащих для работы с отдельными полями данных или компонентами, связанными с этим компонентом по ссылкам. Это могут быть и часто используемые дополнительные операции, как-то выражающиеся через операции с отдельными полями данных, в том числе и вычислимые свойства. Например, для книги в базе данных приложения хранится набор ссылок на данные об ее авторах, а число авторов может быть часто используемым свойством книги, вычисляемым по этому набору ссылок.
  Для сеансового компонента методы удаленного интерфейса служат для реализации некоторых операций бизнес-логики или предметной области, вовлекающих несколько компонентов данных.
- *Локальный интерфейс (local interface)*.
  По назначению этот интерфейс полностью аналогичен удаленному, но декларирует методы компонента, которые можно вызывать только в рамках того же процесса. Этот интерфейс служит для увеличения производительности приложений, в которых взаимодействия между компонентами происходят в основном в рамках одного процесса. При этом они могут использовать локальные интерфейсы друг друга, не привлекая сложный механизм реализации удаленных вызовов методов. Однако, при использовании локального интерфейс компонента нужно обеспечить развертывание этого компонента в рамках того же EJB-контейнера, что и вызывающий его компонент.
  Локальный интерфейс должен наследовать интерфейсу `javax.ejb.EJBLocalObject`.
- *Удаленный исходный интерфейс (remote home interface)*.
  Исходные интерфейсы служат для поиска и создания компонентов. Такой интерфейс может

декларировать метод поиска компонента данных по значению его первичного ключа `findByPrimaryKey(…)` и метод создания такого компонента с указанным значением первичного ключа `create(…)`. Могут быть также определены методы, создающие компонент по набору его данных или возвращающие коллекцию компонентов, данные которых соответствуют аргументам такого метода. Например, метод, создающий компонент, который представляет книгу с данным названием, `createByTitle(String title)`, или метод, находящий все книги с данным набором авторов `Collection findByAuthors(Collection authors)`.
Удаленный исходный интерфейс предназначен для создания и поиска компонентов извне того процесса, в котором они работают. Его методы возвращают ссылку на удаленный интерфейс компонента или коллекцию таких ссылок.
Такой интерфейс должен наследовать интерфейсу `javax.ejb.EJBHome` (являющемуся наследником `java.rmi.Remote`).

- *Локальный исходный интерфейс (local home interface).*
  Имеет то же общее назначение, что и удаленный исходный интерфейс, но служит для работы с компонентами в рамках одного процесса. Соответственно, его методы поиска или создания возвращают ссылку на локальный интерфейс компонента или коллекцию таких ссылок.
  Должен наследовать интерфейсу `javax.ejb.EJBLocalHome`.
- *Класс компонента (bean class).*
  Этот класс реализует методы удаленного и локального интерфейсов (но не должен реализовывать сами эти интерфейсы!). Он определяет основную функциональность компонента.
  Для компонентов данных такой класс должен быть абстрактным классом, реализующим интерфейс `javax.ejb.EntityBean`. Свойства, соответствующие полям хранимых данных или ссылкам на другие компоненты данных, должны быть определены в виде абстрактных пар методов `getName()/setName()`. В этом случае EJB-контейнер может взять на себя управление синхронизацией их значений с базой данных. Вычислимые свойства, значения которых не хранятся в базе данных, реализуются в виде пар неабстрактных методов.
  Для сеансовых компонентов класс компонента должен быть неабстрактным классом, реализующим интерфейс `javax.ejb.SessionBean` и все методы удаленного и локального интерфейсов.
  Кроме того, класс компонента может (а иногда и должен) реализовывать некоторые методы, которые вызываются EJB-контейнером при переходе между различными этапами жизненного цикла компонента.

  Например, при инициализации экземпляра компонента всегда вызывается метод `ejbCreate()`. Для компонентов данных он принимает на вход и возвращает значение типа первичного ключа компонента. Если первичный ключ — составной, он должен принимать на вход значения отдельных его элементов. Такой метод для компонента данных должен возвращать **`null`** и всегда должен быть реализован в классе компонента. Для сеансовых компонентов он имеет тип результата `void`, а на вход принимает какие-то параметры, служащие для инициализации экземпляра компонента. Для каждого метода исходных интерфейсов с именем `createSomeSuffix(…)` в классе компонента должен быть реализован метод `ejbCreateSomeSuffix(…)` с теми же типами параметров. Для компонентов данных все такие методы возвращают значение типа первичного ключа, для сеансовых — **`void`**.

  Другие методы жизненного цикла компонента, которые можно перегружать в классе компонента, декларированы в базовых интерфейсах компонентов соответствующего вида (`javax.ejb.EntityBean` или `javax.ejb.SessionBean`). Это, например, `ejbActivate()` и `ejbPassivate()`, вызываемые при активизации и деактивизации экземпляра компонента; `ejbRemove()`, вызываемый перед удалением экземпляра компонента из памяти; для компонентов данных — `ejbStore()` и `ejbLoad()`, вызываемые при сохранении данных

экземпляра в базу приложения или при их загрузке оттуда.
Схема жизненного цикла компонента данных показана на Рис. 75.

Сеансовые компоненты могут поддерживать состояние сеанса, обеспечивая пользователю возможность получения результатов очередного запроса с учетом предшествовавших запросов в рамках данного сеанса, или не поддерживать. Во втором случае компонент реализует обработку запросов в виде чистых функций.
Жизненный цикл сеансового компонента различается в зависимости от того, поддерживает ли компонент состояние сеанса или нет.

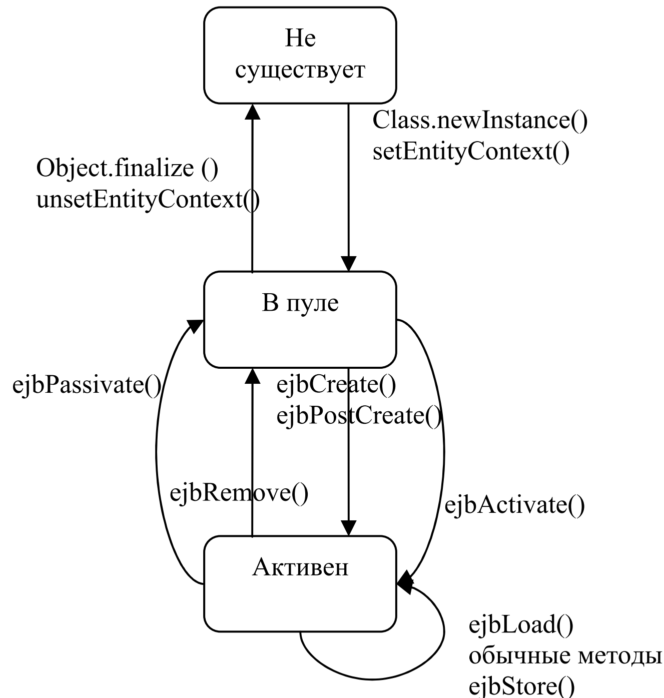

Рисунок 75. Жизненный цикл EJB компонента данных.

Схема жизненного цикла сеансового компонента с состоянием показана на Рис. 76.
Отличие от жизненного цикла компонента данных единственное — метод `ejbCreate()` сразу переводит компонент в активное состояние.

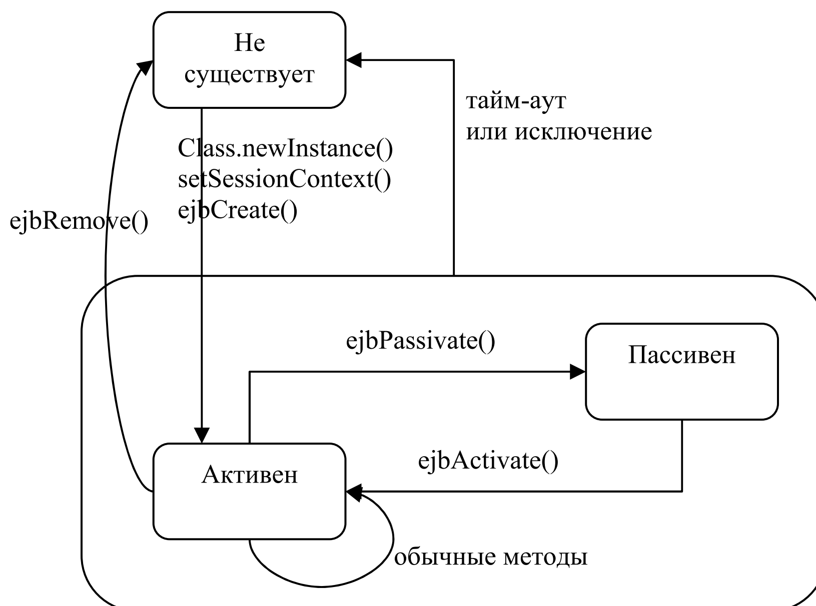

Рисунок 76. Жизненный цикл сеансового компонента с состоянием.

Жизненный цикл сеансового компонента без состояния гораздо проще. Его схема представлена на Рис. 77.

- *Класс первичного ключа (primary key class).*
  Декларируется только для компонентов данных, если в этом качестве нельзя использовать подходящий библиотечный класс.
  Определяет набор данных, которые образуют первичный ключ записи базы данных, соответствующей одному экземпляру компонента.
  Чаще всего это библиотечный класс, например, `String` или `Integer`. Пользовательский класс необходим, если первичный ключ составной, т.е. состоит из нескольких значений простых типов данных. В таком классе должен быть определен конструктор без параметров и правильно перегружены методы `equals()` и `hashCode()`, чтобы EJB-контейнер мог корректно управлять коллекциями экземпляров компонентов с такими первичными ключами. Такой класс также должен реализовывать интерфейс `java.io.Serializable`.

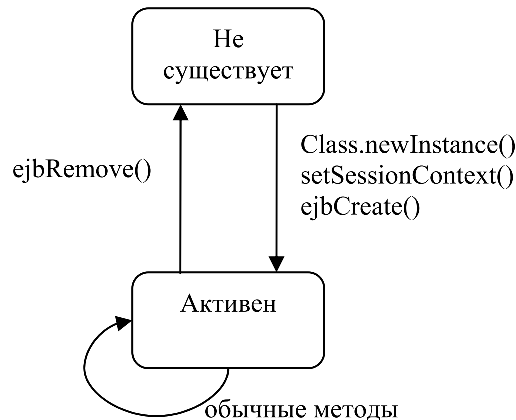

Рисунок 77. Жизненный цикл сеансового компонента без состояния.

Ниже приведены примеры декларации класса компонента и интерфейсов для компонентов данных, соответствующих простой схеме из двух таблиц, которая изображена на Рис. 78.

В рамках этой схемы, состоящей из таблиц, где хранятся данные книг и организаций-издателей, каждая книга связана с одним и только одним издателем, а каждый издатель может иметь ссылки на некоторое множество книг (возможно, пустое). Каждая таблица имеет поле `ID`, являющееся первичным ключом. Таблица `Book` имеет поле `PublisherID`, содержащее значение ключа записи об издателе данной книги.

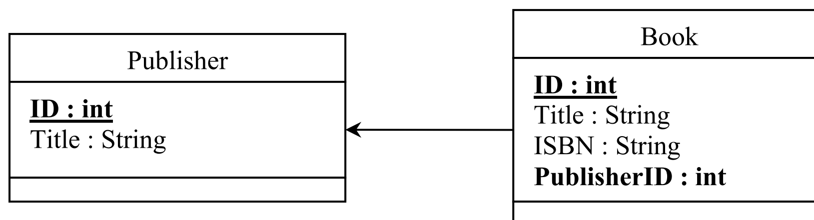

Рисунок 78. Пример схемы БД.

Примеры кода удаленных интерфейсов для компонентов, представляющих данные о книгах и издателях в рамках EJB-приложения.

```
package ru.msu.cmc.prtech.examples;

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

```
public interface PublisherRemote extends EJBObject
{
  public String getTitle ()             throws RemoteException;
  public void   setTitle (String title) throws RemoteException;

  public Collection getBooks ()throws RemoteException;
  public void       setBooks (Collection books)
    throws RemoteException;

  public void addBook    (String title, String isbn)
    throws RemoteException;
  public void removeBook (String title, String isbn)
    throws RemoteException;
}

package ru.msu.cmc.prtech.examples;

import java.rmi.RemoteException;

import javax.ejb.EJBObject;

public interface BookRemote extends EJBObject
{
  public String getTitle ()             throws RemoteException;
  public void   setTitle (String title) throws RemoteException;

  public String getISBN ()            throws RemoteException;
  public void   setISBN (String isbn) throws RemoteException;

  public PublisherRemote getPublisher () throws RemoteException;
  public void            setPublisher (PublisherRemote publisher)
    throws RemoteException;
}
```

Примеры кода удаленных исходных интерфейсов.

```
package ru.msu.cmc.prtech.examples;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface PublisherHomeRemote extends EJBHome
{
  public PublisherRemote create (Integer id)
    throws CreateException, RemoteException;

  public PublisherRemote findByPK (Integer id)
    throws FinderException, RemoteException;
}

package ru.msu.cmc.prtech.examples;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface BookHomeRemote extends EJBHome
{
  public BookRemote create (Integer id)
    throws CreateException, RemoteException;
```

```
  public BookRemote createBook (String title, String isbn)
    throws CreateException, RemoteException;

  public BookRemote findByPK (Integer id)
    throws FinderException, RemoteException;
}
```

Примеры кода классов компонентов. Показано, как реализовывать дополнительные, не поддерживаемые контейнером автоматически, методы работы со связями между данными об издателях и книгах и дополнительные методы создания компонентов.

```
package ru.msu.cmc.prtech.examples;

import java.util.Collection;
import java.util.Iterator;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public abstract class PublisherBean implements EntityBean
{
  public Integer ejbCreate (Integer pk)
  {
    setId(pk);
    return null;
  }

  public void ejbPostCreate (Integer pk) { }

  public abstract Integer getId ();
  public abstract void    setId (Integer pk);

  public abstract String getTitle ();
  public abstract void   setTitle (String title);

  public abstract Collection getBooks ();
  public abstract void       setBooks (Collection books);

  public void addBook (String title, String isbn)
  {
    try
    {
      InitialContext context = new InitialContext();
      BookHomeRemote bookHome =
         (BookHomeRemote)context.lookup("BookHomeRemote");
      BookRemote book = bookHome.createBook(title, isbn);

      Collection books = getBooks();
      books.add(book);
    }
    catch (NamingException e) { e.printStackTrace(); }
    catch (CreateException e) { e.printStackTrace(); }
    catch (RemoteException e) { e.printStackTrace(); }
  }

  public void removeBook (String title, String isbn)
  {
    Collection books = getBooks();
    Iterator it = books.iterator();
```

```
    try
    {
      while(it.hasNext())
      {
        BookRemote book = (BookRemote)it.next();
        if(   book.getTitle().equals(title)
           && book.getISBN().equals(isbn)
          )
        {
          it.remove();
          break;
        }
      }
    }
    catch (RemoteException e) { e.printStackTrace(); }
  }
}

package ru.msu.cmc.prtech.examples;

import javax.ejb.EntityBean;

public abstract class BookBean implements EntityBean
{
  public Integer ejbCreate (Integer pk)
  {
    setId(pk);
    return null;
  }

  public void ejbPostCreate (Integer pk) { }

  public Integer ejbCreateBook (String title, String isbn)
  {
    setTitle(title);
    setISBN(isbn);
    return null;
  }

  public void ejbPostCreateBook (String title, String isbn) { }

  public abstract Integer getId ();
  public abstract void    setId (Integer pk);

  public abstract String getTitle ();
  public abstract void   setTitle (String title);

  public abstract String getISBN ();
  public abstract void   setISBN (String isbn);

  public abstract PublisherRemote getOrganization ();
  public abstract void            setOrganization (PublisherRemote pr);
}
```

Компоненты, управляемые сообщениями

Компоненты, управляемые сообщениями, не доступны для удаленных вызовов, и поэтому не имеют удаленных и исходных интерфейсов. Для создания такого компонента нужно определить только его класс. Обращения к компоненту организуются в виде посылки сообщений к объекту этого класса как к реализующему интерфейс `javax.jms.MessageListener`. Вместе с этим интерфейсом класс компонента EJB, управляемого сообщениями, обязан реализовывать интерфейс `javax.ejb.MessageDrivenBean`.

Первый интерфейс требует определения метода **void** `onMessage(javax.jms.Message)`, который разбирает содержимое пришедшего сообщения и определяет способ его обработки. Кроме того, нужно определить методы **void** `ejbCreate()` для создания компонента и **void** `ejbRemove()` для освобождения ресурсов при его удалении.

Жизненный цикл компонента, управляемого сообщениями выглядит в целом так же, как и жизненный цикл сеансового компонента без состояния (Рис. 77). Вся необходимая информация передается такому компоненту в виде данных обрабатываемого им сообщения.

Пример реализации класса компонента, управляемого сообщениями, приведен ниже. Данный компонент получает идентификатор издателя, название и ISBN книги и добавляет такую книгу к книгам, изданным данным издателем.

```
package ru.msu.cmc.prtech.examples;

import javax.ejb.EJBException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TransferProcessorBean
  implements MessageDrivenBean, MessageListener
{
  Context context;

  public void setMessageDrivenContext (MessageDrivenContext mdc)
      throws EJBException
  {
    try { context = new InitialContext(); }
    catch (NamingException e) { throw new EJBException(e); }
  }

  public void ejbCreate() { }

  public void onMessage (Message msg)
  {
    MapMessage message = (MapMessage)msg;
    try
    {
      Integer publisherPK = (Integer)message.getObject("Publisher");
      String  title       = (String)message.getObject("Title");
      String  isbn        = (String)message.getObject("ISBN");

      PublisherHomeRemote publisherHome =
        (PublisherHomeRemote)context.lookup("PublisherHomeRemote ");

      PublisherRemote publisher = publisherHome.findByPK(publisherPK);

      publisher.addBook(title, isbn);
    }
    catch (Exception e) { throw new EJBException(e); }
  }

  public void ejbRemove () throws EJBException
  {
    try { context.close(); }
    catch (NamingException e) { }
  }
}
```

Дескрипторы развертывания компонентов EJB

Помимо декларации интерфейсов и классов компонентов, для построения EJB компонента необходимо написать дескриптор развертывания — XML файл в специальном формате, определяющий набор компонентов приложения и их основные свойства.

Чаще всего дескрипторы развертывания не пишут вручную, их готовят с помощью специализированных инструментов для развертывания J2EE приложений или сред разработки (например, такими возможностями обладает среда NetBeans 4.0 [3]). Здесь мы опишем только часть содержимого дескрипторов развертывания. Полное описание используемых в них тегов и их назначения см. в [2].

Дескриптор развертывания упаковывается вместе с байт-кодом классов компонентов приложения в JAR-архив. При развертывании такой архив помещают в выделенную директорию, в которой сервером J2EE ищет развертываемые приложения. После этого сервер сам осуществляет запуск приложения и поиск кода компонентов, необходимых для обработки поступающих запросов, на основании информации, предоставленной дескриптором.

Заголовок дескриптора развертывания для набора EJB компонентов версии 2.1 выглядит следующим образом.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" http://java.sun.com/j2ee/dtds/ejb-jar_2_1.dtd">
```

Дескриптор развертывания содержит следующие теги.

- `<ejb-jar>`
  Обязательный элемент.
  Это корневой тег дескриптора развертывания набора EJB компонентов, содержащий все остальные теги.
- `<enterprise-beans>`
  Обязательный элемент, должен появиться внутри `<ejb-jar>` ровно один раз.
  Содержит набор описаний отдельных EJB компонентов в виде элементов `<entity>`, `<session>`, `<message-driven>`.
- `<entity>` и `<session>`
  Эти теги вложены в тег `<enterprise-beans>` и служат для описания, соответственно, компонентов данных и сеансовых компонентов. Они могут содержать следующие вложенные теги.
  - `<ejb-name>`
    Требуется ровно один.
    Задает имя компонента.
  - `<home>`
    Необязателен, начиная с EJB 2.0. В EJB 1.1 требуется ровно один.
    Указывает полное имя удаленного исходного интерфейса.
  - `<remote>`
    Необязателен, начиная с EJB 2.0. В EJB 1.1 требуется ровно один.
    Указывает полное имя удаленного интерфейса.
  - `<local-home>`
    Необязателен.
    Указывает полное имя локального исходного интерфейса.
  - `<local>`
    Необязателен.
    Указывает полное имя локального интерфейса.

- `<ejb-class>`
  Требуется ровно один.
  Указывает полное имя класса компонента.
- `<primkey-field>`
  Необязателен, используется в описании компонентов данных.
  Указывает имя поля, являющегося первичным ключом (если он состоит только из одного поля и синхронизацией компонента с базой данных управляет контейнер).
- `<prim-key-class>`
  Требуется ровно один, используется в описании компонентов данных.
  Указывает имя класса первичного ключа. Можно отложить точное определение класса первичного ключа до развертывания, тогда в этом поле указывается `java.lang.Object`.
- `<persistence-type>`
  Требуется ровно один, используется в описании компонентов данных.
  Имеет значения `Bean` или `Container`, в зависимости от того, управляется ли синхронизация компонента с базой данных самим компонентом или контейнером.
- `<cmp-version>`
  Необязателен.
  Указывает версию спецификаций EJB, в соответствии с которой разработан компонент, что определяет способ управления этим компонентом. Может иметь значения 2.x и 1.x.
- `<abstract-schema-name>`
  Необязателен.
  Задает уникальный идентификатор компонента для использования в запросах на языке EJB QL, который используется для описания запросов к схеме данных при реализации компонента данных, самостоятельно управляющего связью с СУБД.
- `<cmp-field>`
  Один или более, используется в описании компонентов данных.
  Каждый такой элемент описывает одно поле данных, синхронизация которого с СУБД управляется EJB контейнером. Он может содержать тег `<description>` с описанием поля и должен содержать тег `<field-name>` с именем поля. В EJB 2.0 это имя совпадает с именем абстрактного свойства (для которого в классе компонента декларированы методы `getName()` и `setName()`), а в EJB 1.1 — с именем одного из полей класса компонента.
- `<security-role-ref>`
  Один или более, необязателен.
  Указывает роли, используемые данным компонентом. Они при работе приложения служат для авторизации доступа — сопоставляются с ролями, которым разрешен доступ к тому или иному методу.
  Может содержать тег `<description>` (необязателен) и теги `<role-name>` (обязателен), `<role-link>` (необязателен, служит для сопоставления указанного имени роли с логической ролью, описанной в `<security-role>` раздела `<assembly-descriptor>`).
- `<security-identity>`
  Необязателен.
  Определяет, какую логическую роль будет играть данный компонент при обращениях к другим компонентам. Для этого может быть использован вложенный тег `<run-as><role-name>…</role-name></run-as>` для указания имени конкретной логической роли, или `<use-caller-identity/>` для указания того, что нужно использовать роль вызывающего клиента.
- `<session-type>`
  Требуется ровно один, используется в описании сеансовых компонентов.
  Имеет значения `Stateful` или `Stateless`, в зависимости от того, использует ли данный компонент состояние сеанса.

- <transaction-type>
  Требуется ровно один, используется в описании сеансовых компонентов.
  Имеет значения Container или Bean, в зависимости от того, управляет ли транзакциями данного компонента контейнер или он сам. В первом случае соответствующие транзакции должны быть описаны в разделе <assembly-descriptor>, см. далее.
- <query>
  Один или более, необязателен.
  Используется для описания запросов, с чьей помощью реализуются некоторые методы компонентов данных, которые сами управляют связью с базой данных. Запросы описываются на языке EJB QL [X] и привязываются к методам компонента с помощью тегов <query-method>. Сам код запроса описывается внутри элемента CDATA во вложенном теге <ejb-ql>.
  Например

```
<query>
  <query-method>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <!CDATA[
      SELECT OBJECT(c) FROM Client c WHERE c.name = ?1
    ]]
  </ejb-ql>
</query>
```

- <relationships>
  Необязателен. Вложен в <ejb-jar>.
  Описывает набор отношений между компонентами, которые соответствуют связям в схеме базы данных и автоматически поддерживаются контейнером. Каждое отношение описывается с помощью вложенного тега <ejb-relation>.
  - <ejb-relation>
    Описывает одно отношение и может иметь следующие элементы.
    - <ejb-relation-name>
      Необязателен, только один. Задает имя отношения.
    - <ejb-relationship-role>
      Обязательно два. Описывает одну роль в рамах отношения. В описании роли должны присутствовать следующие данные.
      - Имя роли — во вложенном теге <ejb-relationship-role-name>.
      - Множественность — сколько экземпляров компонента могут играть такую роль в раках данного отношения с одним экземпляром в другой роли. Описывается в теге <multiplicity> и может иметь значения One или Many.
      - Имя компонента, экземпляры которого играют данную роль в этом отношении. Определяется в теге <relationship-role-source> внутри тега <ejb-name> в виде имени, которое присвоено компоненту в рамках данного дескриптора.
      - Имя поля, которое хранит ссылку или коллекцию ссылок, поддерживающие это отношение в рамках экземпляра компонента. Определяется в теге <cmr-field>, во вложенном теге <cmr-field-name>, и для него в классе компонента должно быть определено абстрактное свойство с тем же именем.
- <assembly-descriptor>
  Этот обязательный тег внутри <ejb-jar> содержит дополнительные указания для сборки компонентов, в частности следующие.

- <container-transaction>
  Один или более, необязателен.
  Содержит необязательный элемент <description>, а также приведенные ниже.
  Для компонента данных должно быть по одному такому элементу на каждый метод удаленного интерфейсов. Сеансовые компоненты, транзакциями которых управляет EJB-контейнер, также должны подчиняться этому правилу.
  - <method>
    Один или более.
    Содержит тег <ejb-name>, указывающий имя компонента, и <method-name>, указывающий имя метода или знак *, который обозначает применение указанного атрибута ко всем методам.
    Может также включать элементы <description>, <method-params> и <method-intf>, который может иметь значения Remote, Home, Local, Local-Home, в зависимости от того, в каком интерфейсе этот метод декларирован — для поддержки возможности декларировать методы с одним именем и набором параметром в разных интерфейсах.
  - <trans-attribute>
    Ровно один.
    Определяет атрибут транзакции, управляющий политикой включения в транзакции или создания новых транзакций.
    Для компонентов данных атрибуты транзакции должны быть определены для всех методов удаленного интерфейса и методов, декларированных в исходном интерфейсе, для сеансовых компонентов — для всех методов удаленного интерфейса.
    Может иметь значения NotSupported, Supports, Required, RequiresNew, Mandatory, Never. Об их смысле рассказывалось в предыдущей лекции.
- <security-role>
  Один или более, необязателен.
  Определяет роли, служащие для контроля доступа к методам компонентов. В таком элементе должен содержаться тег <role-name>, задающий имя роли.
- <method-permission>
  Один или более, необязателен.
  Указывает правила доступа ролей, определенных в тегах <security-role>, к методам компонентов.
  Содержит необязательный тег <description>, один или несколько тегов <role-name> и один или несколько тегов <method> (см. выше), кроме того, в нем может присутствовать тег <unchecked/>, который обозначает отсутствие проверки прав доступа во время работы, даже если они описаны.
  Каждый тег <method> содержит тег <ejb-name>, указывающий имя компонента, и <method-name>, указывающий имя метода или знак *, который обозначает применение указанного атрибута ко всем методам.
- <exclude-list>
  Необязателен.
  Содержит один или несколько тегов <method> (см. выше), определяющих методы, которые не должны вызываться при работе приложения. Каждый вызов такого метода создает исключительную ситуацию.

Уровень модели данных в .NET

В среде .NET нет средств, полностью аналогичных тем, которые предоставляются в J2EE для разработки компонентов EJB. Уровень бизнес-логики в .NET-приложениях предлагается реализовывать с помощью обычных классов, что значительно проще, чем реализовывать

специальные наборы классов и интерфейсов для компонентов EJB. С другой стороны, связь с базой данных в .NET не реализуется в виде аналогичного EJB объектного интерфейса, если, конечно, не разрабатывать его целиком самостоятельно (или с использованием библиотек компонентов от третьих партий). Вместо этого предлагается для связи с базой данных использовать набор компонентов ADO.NET [4], представляющих собой объектные обертки реляционной структуры данных.

Классы ADO.NET располагаются в сборке System.Data (дополнительные классы можно найти в System.Data.OracleClient и System.Data.SqlXml) и пространстве имен `System.Data`, вместе с вложенными в него пространствами.

Основным классом, с помощью которого представляются данные из базы данных, является `System.Data.DataSet`. Он представляет набор таблиц, связанных между собой некоторыми связями и выполняющими определенные ограничения. Каждая таблица представляется объектом класса `System.Data.DataTable`, каждая связь — объектом класса `System.Data.Relation`, каждое ограничение — объектом класса `System.Data.Constraint`. Структура таблиц описывается с помощью их полей (представляемых объектами `System.Data.DataColumn`). Содержимое одной таблицы представлено как набор объектов-записей, имеющих тип `System.Data.DataRow`.

Из перечисленных классов только `DataSet` и `DataTable` являются сериализуемыми, т.е. только их объекты могут быть переданы в другой процесс или на другую машину.

Объект класса `DataSet` может представлять собой и набор данных документа XML. Получить такой объект можно с помощью класса `System.Xml.XmlDataDocument`.

Само взаимодействие с источником данных происходит с помощью объектов классов `DataAdapter`, `DataReader`, `DbConnection`, `DbTransaction`, `DbCommand` и производных от них, специфичных для того или иного вида источников данных (в рамках поставляемых в составе среды Visual Studio .NET библиотек имеются специфичные классы для работы с источниками ODBC, OleDB, MS SQL Server, Oracle). Все перечисленные классы находятся в пространстве имен `System.Data.Common`, а их производные для данного вида источников данных — в соответствующем этому виду источников подпространстве `System.Data`.

Объекты классов `DataAdapter` и `DataReader` служат для чтения и записи данных в виде объектов `DataSet`. Остальные классы используются для определения соединений, организации транзакций, определения и выполнения SQL-команд по чтению или изменению данных.

Ниже приводится простой пример работы с данными с помощью библиотек ADO.NET.

```
DbConnection connection = new SqlConnection("Data Source=localhost;" +
    "Integrated Security=SSPI;Initial Catalog=DBCatalog");

DbCommand command = new SqlCommand("SELECT ID, Title, ISBN FROM Book",
    connection);

DataAdapter adapter = new SqlDataAdapter();
Adapter.SelectCommand = command;

connection.Open();

DataSet dataset = new DataSet();
adapter.Fill(dataset, "Book");

connection.Close();
```

Протокол HTTP

Прежде, чем перейти к построению интерфейса пользователя в Web-приложениях на основе J2EE и .NET, стоит рассмотреть основные элементы протокола HTTP, используемого для передачи данных между серверами и клиентами в таких приложениях. Поскольку основная функциональность компонентов интерфейса Web-приложений связана с обработкой и созданием

сообщений HTTP, знание элементов этого протокола необходимо для понимания технологий разработки приложений такого рода.

HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста) представляет собой протокол прикладного уровня, использующий для пересылки данных протокол транспортного уровня. Достаточно подробное описание его можно найти в [5,6].

Сообщения HTTP бывают двух видов: ***запросы клиента*** и ***ответы сервера***.

Запрос HTTP состоит из идентификации метода запроса, универсального идентификатора запрашиваемого ресурса (Universal Resource Identifier, URI), указания версии протокола и, возможно, набора заголовков с дополнительной информацией, а также поля данных общего вида.

```
<Request> ::= GET <URI> CrLf
            | <Method> <URI> <HTTP-Version> CrLf
              <Req-Header>* ( <Data> )?
<Req-Header>  ::= <Field> : <Value> CrLf
CrLf      ::= '\r''\n'
```

Основные методы протокола HTTP следующие.

- `GET`
  Служит для получения любой информации по URI запроса, обычно — документа, хранящегося по указанному адресу или генерируемого по запросу с таким URI. Может иметь заголовок `If-Modified-Since`, который предписывает не посылать тело запрашиваемого ресурса, если он не изменялся с указанной даты.
- `POST`
  Служит для создания нового ресурса, связанного с указанным по URI. Чаще всего используется для аннотации ресурсов, добавления сообщений в группы новостей, дистанционной работы с базами данных. Реальная обработка такого запроса зависит от содержащегося в нем URI.

Остальные методы — `HEAD`, `PUT`, `DELETE`, `LINK`, `UNLINK` — используются гораздо реже.

Заголовки запроса служат для передачи дополнительной информации об этом запросе или о клиенте. Заголовок состоит из идентификатора поля и его значения, разделенных двоеточием, и бывает одного из следующих типов.

- `From`
  Содержит e-mail адрес пользователя, под чьим именем работает клиент.
  Пример: `From: webmaster@yandex.ru`
- `Accept`, `Accept-Encoding`, `Accept-Charset` и `Accept-Language`
  В таком заголовке через запятую перечисляются возможные форматы (соответственно, кодировки, используемые таблицы символов и языки) ответов на данный запрос.
  Пример: `Accept: text/plain, text/html, text/x-dvi; q=.8; mxb=100000; mxt=5.0`
- `User-Agent`
  Содержит название используемой клиентской программы.
- `Referer`
  Используется для указания адреса ресурса, с которого был получен данный запрос.
- `If-Modified-Since`
  Используется для отмены ответной пересылки документов, модифицированных не позднее указанной даты, с целью снижения нагрузки на сеть.
- `Authorization`
  Содержит авторизационную информацию, специфичную для используемых сервером протоколов авторизации.
- `ChargeTo`
  Содержит информацию о том, куда выставить счет за обработку запроса.

- `Pragma`
  Содержит дополнительные директивы для промежуточных серверов, например, прокси-серверов.

Пример запроса.

```
GET /locate?keywords=HTTP+description HTTP/1.1
Date: Mon, 15 Dec 2004 12:18:15 GMT
Accept: image/gif, image/jpg, */*
Accept-Charset: iso-8859-1, *, utf-8
Accept-Language: en
Connection: keep-Alive
User-Agent: Mozilla/4.7 [en] (Win98; u)
```

Группа кодов	Код	Фраза-объяснение, следующая за кодом	Значение кода
1xx			Информационное сообщение
2xx			Успешная обработка
	200	OK	Все нормально
	201	Created	Документ создан
3xx			Перенаправление запроса
	301	Moved Permanently	Ресурс перемещен
	302	Moved Temporarily	Ресурс перемещен временно
4xx			Ошибка клиента
	400	Bad Request	Некорректно составленный запрос
	401	Unauthorized	Нужна аутентификация клиента
	403	Forbidden	Доступ к ресурсу запрещен
	404	Not Found	Запрашиваемый ресурс отсутствует
5xx			Ошибка сервера
	500	Internal Server Error	Внутренняя ошибка сервера

Таблица 12. Некоторые коды статуса ответа HTTP и их объяснение.

Ответ сервера на HTTP-запрос состоит либо только из запрашиваемого клиентом документа, либо в дополнение к нему содержит код статуса ответа и, возможно, несколько заголовков ответа.

```
<Response>    ::= ( <Content> )?
                | <HTTP-Version> <Code> <Explanation> CrLf
                  <Resp-Header>* ( <Content> )?
<Resp-Header> ::= <Field> : <Value> CrLf
CrLf       ::= '\r''\n'
```

Некоторые коды статуса ответа поясняются в Таблице 12.

Возможны следующие заголовки ответа.

- `Allowed`
  Перечисляются через пробел доступные для пользователя методы запросов по данному URI.

- `Public`
  Перечисляет доступные всем методы запросов.

- `Content-Length`, `Content-Type`, `Content-Encoding` и `Content-Language`
  Задают размер содержимого в байтах (подразумевается, что содержимое имеет бинарный формат и не должно разбиваться на строки при чтении), его формат или MIME-тип, кодировку и язык.

- `Date`
  Дата создания содержащегося документа или объекта.

- `Last-Modified`
  Дата последнего изменения содержащегося объекта.

- `Expires`
  Дата, послед которой содержащийся объект считается устаревшим.
- `URI`
  URI содержащегося объекта.
- `Title`
  Заголовок содержащегося документа.
- `Server`
  Описывает серверную программу.
- `Retry-After`
  Определяет промежуток времени на обработку запроса, до прохождения которого не надо направлять запрос повторно, если ответа на него еще нет.

Пример HTTP-ответа.

```
HTTP/1.0 200 OK
Content-Length: 2109
Content-Type: text/html
Expires: 0
Last-Modified: Thu, 08 Feb 2001 09:23:17 GMT
Server: Apache/1.3.12
<HTML> <HEAD> <TITLE> … </TITLE> </HEAD>
<BODY> …
</BODY>
</HTML>
```

Уровень пользовательского интерфейса в J2EE

Компоненты пользовательского интерфейса в Web-приложениях, построенных как по технологии J2EE, так и по .NET, реализуют обработку HTTP-запросов, приходящих от браузера, и выдают в качестве результатов HTTP-ответы, содержащие сгенерированные HTML-документы с запрашиваемыми данными. Сами запросы автоматически строятся браузером на основе действий пользователя — в основном, переходов по ссылкам и действий с элементами управления в HTML-формах.

Если стандартных элементов управления HTML не хватает для реализации функций приложения или они становятся неудобными, используются специальные библиотеки элементов управления WebUI, предоставляющие более широкие возможности для пользователя и более удобные с точки зрения интеграции с остальными компонентами приложения.

В рамках J2EE версии 1.4 два основных вида компонентов WebUI — ***сервлеты (servlets)*** и ***серверные страницы Java (Java Server Pages, JSP)*** — отвечают, соответственно, за обработку действий пользователя и представление данных в ответе на его запросы. В следующей версии J2EE 5.0 будут также использоваться ***компоненты серверного интерфейса Java (Java Server Faces, JSF)*** — библиотека элементов управления WebUI.

Сервлеты представляют собой классы Java, реализующие обработку запросов HTTP и генерацию ответных сообщений в формате этого протокола. Страницы JSP являются упрощенным представлением сервлетов, основанным на описании генерируемого в качестве ответа HTML-документа при помощи смеси из его постоянных элементов и кода на Java, генерирующего его изменяемые части. При развертывании Web-приложения содержащиеся в нем страницы JSP транслируются в сервлеты и далее работают в таком виде. Описание генерируемых документов на смеси из HTML и Java делает страницы JSP более удобными для разработки и значительно менее объемными, чем получаемый из них и эквивалентный по функциональности класс-сервлет.

Сервлеты

Интерфейс Java-сервлетов определяется набором классов и интерфейсов, входящих в состав пакетов `javax.servlet и javax.servlet.http`, являющихся частью J2EE SDK. Первый пакет

содержит классы, описывающие независимые от протокола сервлеты, второй — сервлеты, работающие с помощью протокола HTTP.

Основные классы и интерфейсы пакета `javax.servlet.http` следующие.

- `HttpServlet`
  Предназначен для реализации сервлетов, работающих с HTTP-сообщениями. Содержит защищенные методы, обрабатывающие отдельные методы HTTP-запросов, из которых наиболее важны `void doGet(HttpServletRequest, HttpServletResponse)`, определяющий обработку `GET`-запросов, и `void doPost(HttpServletRequest, HttpServletResponse)`, обрабатывающий `POST`-запросы. В обоих методах первый параметр содержит всю информацию о запросе, а второй — о генерируемом ответе.
- `HttpServletRequest` и `HttpServletResponse` — интерфейсы, содержащие методы для получения и установки (второй) заголовков и других атрибутов HTTP-запросов и ответов. Второй интерфейс также содержит метод, возвращающий поток вывода для построения содержимого ответа.
- `Cookie`
  Класс, представляющий закладки сервера, которые хранятся на клиентской машине для запоминания информации о данном пользователе.
- `HTTPSession`
  Интерфейс, предоставляющий методы для управления сеансом обмена HTTP-сообщениями. Информация о сеансе используется в том случае, если она должна быть доступна нескольким сервлетам.

При развертывании J2EE приложения, помимо самих классов сервлетов, надо создать их ***дескриптор развертывания***, который оформляется в виде XML-файла web.xml.

Web-приложение поставляется в виде архива .war, содержащего все его файлы. На самом деле это zip-архив, расширение .war нужно для того, чтобы Web-контейнер узнавал архивы развертываемых на нем Web-приложений. Содержащаяся в этом архиве структура директорий Web-приложения должна включать директорию WEB-INF, вложенную непосредственно в корневую директорию приложения. Директория WEB-INF содержит две поддиректории — classes для .class-файлов сервлетов, классов и интерфейсов EJB-компонентов и других Java-классов, и lib для .jar и .zip файлов, содержащих используемые библиотеки. Файл web.xml также должен находится непосредственно в директории WEB_INF.

Заголовок дескриптора развертывания сервлета выглядит так.

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc...BEB Web Application 2.2.. EN"
http://java.sun.com/j2ee/dtds/web-app_2_2.dtd>
```

Содержимое дескриптора развертывания помещается внутри тега `<web-app>`. В нем указывается список сервлетов, входящих в приложение и отображение сервлетов в URL, запросы к которым они обрабатывают. Один сервлет описывается в следующем виде.

```
<servlet>
  <servlet-name>ServletName</servlet-name>
  <servlet-class>com.company.deprtment.app.ServletClassName</servlet-class>
  <description>…</description>
  <init-param>
    <param-name>ParameterName</param-name>
    <param-value>ParameterValue</param-value>
    <description>…</description>
  </init-param>
</servlet>
```

Значения параметров инициализации сервлета можно получить с помощью методов `String getInitParameter(String)` и `Enumeration getInitParametersNames()` связанного с сервлетом объекта класса `ServletContext`.

Отображение сервлета на URL описывается так.

```
<servlet-mapping>
  <servlet-name>ServletName</servlet-name>
  <url-pattern>URL</url-pattern>
</servlet-mapping>
```

Серверные страницы Java

Серверные страницы Java [7,8] представляют собой компоненты, разрабатываемые на смеси из HTML и Java и предназначенные для динамического создания HTML-документов, содержащих результаты обработки запросов пользователя. Таким образом, JSP обычно играют роль представления в образце «данные-представление-обработчик», принятом за основу архитектуры приложений J2EE. Результатом работы JSP является HTML-страничка, а вставки Java кода служат для построения некоторых ее элементов на основе результатов работы приложения.

При работе Web-приложения JSP компилируются в сервлеты специального вида. При этом основное содержание страницы JSP превращается в метод `doGet()`, в котором HTML-элементы записываются в поток содержимого ответа в неизменном виде, а элементы Java-кода преобразуются в код, записывающий некоторые данные в тот же поток на основании параметров запроса или данных приложения.

Для развертывания JSP-страниц необходимо их описание в дескрипторе развертывания приложения web.xml, которое устроено так же, как описание сервлетов. Сами JSP-страницы помещаются, вместе с HTML-файлами и другими файлами, используемыми приложением в корневую директорию этого приложения или ее поддиректории.

Основные интерфейсы и базовые классы JSP-страниц и их отдельных элементов находятся во входящих в J2EE SDK пакетах `javax.servlet.jsp`, `javax.servlet.jsp.el`, `javax.servlet.jsp.tagext`.

Элементами JSP-страниц могут быть обычные теги HTML, а также специальные элементы JSP — *директивы*, *теги* или *действия (tags, actions)* и *скриптовые элементы*.

JSP-директивы описывают свойства страницы в целом и служат для передачи информации механизму управления JSP-страницами.

Директивы имеют следующий общий синтаксис.

```
<%@ directive attribute1="value1" ... attributeN="valueN" %>.
```

Основные директивы JSP следующие.

- Директива **`page`** предоставляют общую информацию о данной странице и статически включаемых в нее файлах. Такая директива на странице может быть только одна. Она может иметь следующие атрибуты.
  - `import = "`имена включаемых классов и пакетов через запятую`"`
    Порождает соответствующую Java-директиву **`import`** в сгенерированном коде сервлета.
  - `contentType = "`MIME-тип[;`charset=`таблица символов]`"`
    Задает тип MIME для генерируемого документа. По умолчанию используется `text/html`. Эквивалентен скриплету
    `<% response.setContentType(`MIME-тип`); %>` (см. далее).
  - `isThreadSafe = "true|false"`
    Значение `true` позволяет использовать один экземпляр сервлета, полученного из странички, для обработки множественных запросов. При этом необходимо синхронизовать доступ к данным этого сервлета.
  - `session = "true|false"`
    Значение `true` предписывает привязать сервлет к имеющейся HTTP-сессии, значение `false` говорит, что сессии использоваться не будут и обращение к переменной `session` приведет к ошибке.

- `autoFlush = "true|false"`
  Определяет необходимость сбрасывать буфер вывода при заполнении.
- `buffer = "`размер в KB|`none"`
  Задает размер буфера для выходного потока сервлета.
- `extends = "`наследуемый класс`"`
  Определяет класс, наследуемый сгенерированным из данной JSP сервлетом.
- `errorPage = "`url странички с информацией об ошибках`"`
  Определяет страницу, которая используется для обработки исключений, не обрабатываемых в рамках данной.
- `isErrorPage = "true|false"`
  Допускает или запрещает использование данной страницы в качестве страницы обработки ошибок.
- `language = "java"`
  Определяет язык программирования, применяемый в скриптовых элементах данной страницы. Пока есть возможность использовать только Java. Впоследствии предполагается (аналогично .NET) разрешить использование других языков, код которых будет также транслироваться в байт-код, интерпретируемый JVM.

- Директива **`include`** обеспечивает статическое (в ходе трансляции JSP в сервлет) включение в страничку внешнего документа. Она имеет атрибут `file`, значением которого должна быть строка, задающая URL включаемого файла.
- Директива **`taglib`** указывает используемую в данной странице библиотеку пользовательских тегов. Она имеет два атрибута — `uri`, значением которого является URI библиотеки, и `prefix`, определяющий префикс тегов из данной библиотеки. Префикс употребляется в дальнейшем с тегами только данной библиотеки. Он не может быть пустым и не должен совпадать с одним из зарезервированных префиксов `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun`, `sunw`.

Теги или *действия* определяют основные действия, выполняемые при обработке данных и построении результирующего документа.

Теги могут быть *стандартными*, использование которых возможно в любой странице без дополнительных объявлений, или *пользовательскими*, которые могут употребляться, только если предварительно с помощью директивы `taglib` была подключена содержащая их библиотека. Любой тег имеет следующий синтаксис.

```
<tagprefix:tag attribute1="value1" … attributeN="valueN" />
```

Теги могут содержать вложенные теги, такие как `jsp:param`, `jsp:attribute`. В этом случае они выглядят следующим образом.

```
<tagprefix:tag attribute1="value1" … attributeN="valueN">
  … (вложенные теги)
</tagprefix:tag>
```

Стандартные теги имеют префикс `jsp`, а префикс пользовательских тегов определяется в директиве `taglib`, подключающей содержащую их библиотеку.

Имеется довольно много стандартных тегов. Основные из них следующие.

- `jsp:include`
  Определяет динамическое включение некоторой страницы или файла в данную страницу при обработке запроса. С помощью вложенных тегов `jsp:param` может указывать один или несколько пар параметр-значение в качестве параметров включаемой страницы.
  Имеет атрибуты `page`, определяющий URL включаемой страницы, и `flush`, имеющий значения `true` или `false` в зависимости от того, нужно ли сразу после включения сбросить буфер выходного потока в генерируемом ответе или нет.

- `jsp:useBean`
  Определяет используемый объект или компонент. Фактически такой тег эквивалентен декларации переменной определенного типа, инициализируемой определенным объектом и доступной в рамках некоторого контекста.
  Имеет следующие атрибуты.
  - `id = "`имя объекта`"`
    Задает имя объекта, которое будет использоваться в коде JSP. Должно быть уникально в пределах страницы.
  - `class = "`класс объекта`"`
    Задает класс этого объекта.
  - `scope = "page|request|session|application"`
    Задает область видимости декларируемого объекта.
  - `type = "`тип используемой ссылки на объект`"`
    Указанный тип должен быть предком класса объекта. Это тип декларируемой переменной, а класс объекта определяет истинный тип объекта, хранящегося в ней.
- `jsp:setProperty`, `jsp:getProperty`
  Устанавливает или получает значение свойства объекта.
  Атрибут `name` определяет имя объекта, чье свойство используется, а атрибут property — имя свойства.
  Тег `jsp:getProperty` записывает полученное значение свойства в результирующий документ в виде строки (результата вызова `toString()` для этого значения).
  Тег `jsp:setProperty` имеет также дополнительный атрибут — либо `value`, значение которого присваивается свойству, либо `param`, который указывает имя параметра запроса, значение которого записывается в свойство. Если в теге `jsp:setProperty` вместо имени свойства в атрибуте `property` указан символ `*`, то всем свойствам указанного объекта с именами, совпадающими с именами параметров запроса, будут присвоены значения соответствующих параметров.
- `jsp:forward`
  Этот тег употребляется для перенаправления запроса на обработку другой странице. URL этой страницы указывается в качестве значения атрибута `page`. В качестве этого URL может использоваться JSP-выражение (см. далее), вычисляемое на основе параметров запроса.
  С помощью вложенных тегов `jsp:param` можно передать одно или несколько значений параметров странице, на которую переключается управление.
- `jsp:plugin`
  Этот тег вставляет аплет или компонент JavaBean на страницу. Параметры инициализации компонента могут быть заданы при помощи вложенного тега `jsp:params`. Кроме того, `jsp:plugin` имеет следующие атрибуты.
  - `type = "bean|applet"`
    Задает вид вставляемого компонента.
  - `code = "`имя файла класса компонента (включая расширение .class)`"`
  - `codebase = "`имя директории, в которой находится файл класса компонента`"`
    Если этот атрибут отсутствует, используется директория, содержащая данную JSP-страницу.
  - `name = "`имя используемого экземпляра компонента`"`
  - `archive = "`список разделенных запятыми путей архивных файлов, которые будут загружены перед загрузкой компонента`"`
    Эти архивы содержат дополнительные классы и библиотеки, необходимые для работы компонента.

- `align = "bottom|top|middle|left|right"`
  Задает положение экземпляра компонента относительно базовой строки текста содержащего его HTML-документа.
- `height = "`высота изображения объекта в точках`"`
- `width = "`ширина изображения объекта в точках`"`
- `hspace = "`ширина пустой рамки вокруг объекта в точках`"`
- `vspace = "`высота пустой рамки вокруг объекта в точках`"`
- `jreversion = "`версия JRE, необходимая для работы компонента`"`
  По умолчанию используется версия 1.1.

Пользовательские теги могут быть определены для выполнения самых разнообразных действий. Одна из наиболее широко используемых библиотек тегов core (подключаемая с помощью директивы `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c" %>`), содержит теги, представляющие все конструкции языка Java.

Например, с помощью тега `<c:set var="variable" value="value" />` можно присвоить значение `value` в переменную `variable`, с помощью тега `<c:if test="expression">…</c:if>` можно выполнить код JSP, расположенный внутри этого тега только при том условии, если `expression` имеет значение **`true`**, с помощью тега `<c:forEach var="variable" items="collection">…</c:forEach>` можно выполнить содержащийся в нем код для всех элементов коллекции `collection`, используя для обращения к текущему элементу переменную `variable`.

Таким образом, весь код, который может быть записан в виде скриптовых элементов (см. далее), можно записать и в виде тегов. Применение тегов при этом вносит некоторые неудобства для Java-программиста. Но оно же делает код JSP-страниц более однородным и позволяет обрабатывать его с помощью инструментов для разработки Web-сайтов, гораздо лучше приспособленных к работе с размеченным с помощью тегов текстом.

Скриптовые элементы могут иметь один из следующих трех видов.

- *JSP-объявления*, служащие для определения вспомогательных переменных и методов. Эти переменные становятся впоследствии полями сгенерированного по JSP сервлета, а методы — его методами. Синтаксис JSP-объявления следующий.
  `<%!` код на Java `%>`
- *Скриплеты*, которые служат для вставки произвольного кода, обрабатывающего данные запроса или генерирующего элементы ответа, в произвольное место. Они имеют следующий синтаксис.
  `<%` код на Java `%>`
- *JSP-выражения*, используемые для вставки в какое-то место в результирующем документе вычисляемых значений (они также могут использоваться в качестве значений атрибутов тегов). Их синтаксис может иметь три вида.
  `<% =` выражение на Java `%>`
  `${`выражение на Java`}`
  `#{`выражение на Java`}`

Различий между первым и вторым способом представления выражений практически нет. Выражение третьего типа вычисляется отложено — вычисление его значения происходит только тогда, когда это значение действительно понадобится.

Комментарии в коде JSP оформляются в виде содержимого тега `<%-- … --%>`. Встречающийся в них код не обрабатывается во время трансляции и не участвует в работе полученного сервлета. Элементы кода внутри HTML-комментариев `<!-- … -->` обрабатываются так же, как и в других местах — они генерируют содержимое комментариев в результирующем HTML-документе.

Помимо объявленных в объявлениях и тегах `jsp:useBean` переменных в скриптовых элементах могут использоваться неявно доступные объекты, связанные с результирующим сервлетом, обрабатываемым им запросом и генерируемым ответом, например, следующие.

- `request` — запрос клиента (тип `ServletRequest`).
- `param` — параметры запроса (тип `Map`).
- `response` — ответ сервера (тип `ServletResponse`).
- `out` — выходной поток сервлета (тип `PrintWriter`).
- `session` — сеанс (тип `HttpSession`).
- `application` —приложение (`ServletContext`).
- `config` — конфигурация сервлета (`ServletConfig`).
- `pageContext` — контекст страницы (`javax.servlet.jsp.PageContext`).
- `exception` — произошедшее исключение.

Ниже приведен пример JSP страницы, генерирующей таблицу балансов клиентов некоторой организации в долларовом и рублевом выражениях.

```
<%@ page import="java.util.Date, java.util.Iterator,
    com.company.Client" %>
<jsp:useBean id="clients" class="com.company.ClientList"
    scope="page" />
<jsp:useBean id="convertor" class="com.company.ExchangeRate"
    scope="page" />
<html>
<head>
<title>Table of clients</title>
</head>
<body>
<h3 align="center">Table of clients</h3>
Created on <%= new Date() %> <br><br>

<table width="98%"  border="1" cellspacing="1" cellpadding="1">
  <tr>
  <%!
    private double dollarsToRubles(double m)
    {
      return m*convertor.getDollarToRubleRate(new Date());
    }
  %>
    <th width="50%" scope="col">Client</th>
    <th width="25%" scope="col">Balance, dollars</th>
    <th width="25%" scope="col">Balance, rubles</th>
  </tr>
  <%
     Iterator it = clients.getNumberOfClients().iterator();
     while(it.hasNext())
     {
       Client client = (Client)it.next();
       %>
  <tr>
    <td> ${client.getFullName()} </td>
    <td> ${client.getBalance()} </td>
    <td> ${dollarsToRubles(client.getBalance())} </td>
  </tr>
  <%
     }
  %>
</table> <br><br>

<jsp: include page="footer.txt" flush= "true" />
```

```
</body>
</html>
```

Уровень пользовательского интерфейса в .NET

Разработка компонентов пользовательского интерфейса Web-приложений в рамках .NET выделена в виде отдельной технологии ASP.NET [9,10] и в целом очень похожа на разработку тех же компонентов в J2EE. В .NET имеются те же виды компонентов: элементы управления, представленные *серверными элементами управления HTML (HTML server controls)* и просто *серверными элементами управления (Web Server Controls)*, обработчики HTTP запросов (аналог сервлетов в Java), представленные интерфейсами `IHttpHandler` и `IHttpAsyncHandler`, и так называемые ***Web-формы (Web forms)***, аналог серверных страниц Java.

Элементы управления WebUI в .NET могут быть размещены на HTML-страницах, но выполняются на сервере. Библиотеки таких компонентов находятся в сборке System.Web и в пространстве имен `System.Web.UI`, вместе с его подпространствами. Их употребление в рамках HTML-документа оформляется в виде специальных тегов с атрибутом `runat`, имеющим значение `server`. Ниже приведен пример использования компонента `System.Web.UI.WebControls.Button` в коде Web-формы.

```
<%@ Page Language="C#" AutoEventWireup="True" %>

<html>
<head>
   <script language="C#" runat=server>
      void OnButtonClick(object sender, EventArgs e)
      {
         Message.Text="Hello World!!";
      }
   </script>
</head>
<body>
   <form runat="server">
      <h3>Button Example</h3>
      Click on the submit button.<br><br>

      <asp:Button id="MyButton"
           Text="Submit"
           OnClick="OnButtonClick"
           runat="server"/>

      <p>
      <asp:label id="Message" runat="server"/>
   </form>
</body>
</html>
```

Аналогом сервлетов в .NET являются объекты, реализующие интерфейсы `System.Web.IHttpHandler` и `System.Web.IHttpAsyncHandler`. Оба они являются частью *программного интерфейса Web-сервера* Microsoft *(Internet Server Application Program Interface, ISAPI)*. Первый интерфейс предназначен для синхронной обработки запросов, с блокированием на время обработки вызвавшего ее потока Web-сервера. Второй интерфейс позволяет реализовывать такую обработку в отдельном потоке.

Единственный метод первого интерфейса — `void ProcessRequest (System.Web.HttpContext context)`. Все данные, связанные с запросом, ответом на него, приложением и контекстом, в котором работает данный обработчик, можно получить, используя различные свойства параметра этого метода.

Интерфейс `IHttpAsyncHandler` имеет два метода — `IAsyncResult BeginProcessRequest( HttpContext context, AsyncCallback cb, object extraData)` и `void EndProcessRequest(`

IAsyncResult result). Первый вызывается при передаче запроса данному обработчику, второй — для прекращения обработки.

Web-формы .NET являются аналогом серверных страниц Java. Они так же оформляются в виде документов, содержащих конструкции как HTML, так и одного из языков программирования, используемых в рамках .NET, и специальные конструкции, аналогичные директивам, тегам и скриптовым элементам JSP.

Специальные конструкции Web-форм включают *директивы*, имеющие тот же самый смысл, что и для серверных страниц Java, *объявления*, аналогичные JSP-объявлениям, конструкции *встроенного кода (code render)* и конструкции *привязки к данным (data binding expressions)*.

Директивы Web-форм имеют в целом точно такой же синтаксис, как и директивы JSP: <%@ directive attribute1="value1" ... attributeN="valueN" %>. Список директив шире, чем в JSP: имеется директива **Page**, аналог **page** в JSP, но с несколько отличающимся списком атрибутов, директива **Import**, аналог **include**, директива **Control** для описания пользовательских элементов управления, директива **Register** для определения синонимов (алиасов), и пр.

Объявления полей данных и методов в Web-формах обрамляются в тег <script> … </script>. Такой тег должен иметь атрибут runat со значением server и атрибут language, который определяет язык кода, написанного внутри тега. Он может также иметь атрибут src для указания URL файла, код из которого должен быть вставлен перед содержимым тега.

Конструкции встроенного кода обрамляются в тег <% … %>. Как и в JSP, могут использоваться выражения в виде <%=… %>. Комментарии тоже оформляются, как и в JSP, в виде тегов <%-- … --%>.

Конструкции привязки к данным имеют синтаксис <%# expression %> и работают примерно так же, как и выражения встроенного кода. Они могут использоваться и в значениях атрибутов элементов управления.

Конфигурационные файлы компонентов .NET, являющиеся аналогами дескрипторов развертывания в J2EE, оформляются в виде XML-документов специального вида и размещаются в различных директориях Web-приложения. В качестве корневого тега таких документов всегда выступает тег configuration. Он может содержать теги location, которые определяют конфигурацию для ресурсов, путь к которым указывается в атрибуте path таких тегов. Теги location для компонентов ASP.NET содержат тег system.web, который, в свою очередь, может содержать следующие теги (перечислены не все возможные, более полную информацию см. в [11]).

- authentication
  Определяет используемый вид аутентификации — атрибут mode задает используемый механизм (Windows, Forms, Passport или None), вложенные теги forms описывают свойства отдельных форм, используемых для аутентификации.
- authorization
  Определяет права доступа для пользователей, ролей и отдельных методов HTTP-запросов. Разрешения на доступ указываются в атрибутах вложенного теги allow, запреты — в атрибутах вложенного тега deny.
- compilation
  Определяет параметры компиляции компонента ASP.NET.
- customErrors
  Определяет специфические для данного приложения ошибки и URL, на которые переходит управление при возникновении этих ошибок.
- globalization
  Определяет кодировки и локализацию запросов и ответов.

- `httpHandlers`
  Определяет отображение адресов и методов запросов на обрабатывающие их объекты типа `IHttpHandelr` или `IHttpHandlerFactory`.
- `pages`
  Определяет настройки для отдельных страниц.
- `sessionState`
  Описывает настройки для поддержки состояния сеансов работы с данным приложением.

Литература к Лекции 14

[1] Р. Монсон-Хейфел. Enterprise JavaBeans. СПб.: Символ-Плюс, 2002.
[2] Enterprise JavaBeans Specification, version 2.1.
Доступны по ссылке http://java.sun.com/products/ejb/docs.html.
[3] Сайт проекта NetBeans http://www.netbeans.org/.
[4] Документация MSDN по ADO.NET
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconaccessingdatawithadonet.asp.
[5] Hypertext Transfer Protocol — HTTP/1.1. RFC 2616.
Доступно по ссылке http://www.w3.org/Protocols/rfc2616/rfc2616.html.
[6] http://www.opennet.ru/docs/RUS/http/index.html
[7] Документация по JSP http://java.sun.com/products/jsp/docs.html.
[8] Б. У. Перри. Java сервлеты и JSP: сборник рецептов. М.: Кудиц-Образ, 2005.
[9] Документация MSDN по ASP.NET
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconintroductiontoasp.asp.
[10] Р. Андерсон, Б. Френсис, А. Хомер, Р. Хоуорд, Д. Сассмэн, К. Уотсон. ASP.NET 1.0 для профессионалов. М.: Лори, 2004.
[11] Схема конфигурационных файлов ASP.NET
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/gngrfASPNETConfigurationSectionSchema.asp.
[12] П. Аллен, Дж. Бамбара, М. Ашнаульт, Зияд Дин, Т. Гарбен, Ш. Смит. J2EE. Разработка бизнес-приложений. СПб.: ДиаСофт, 2002.
[13] Д. Просиз. Программирование для Microsoft.NET. М.: Русская редакция, 2003.