Лекция 15

Технология RMI

Как правило, приложения RMI состоят из двух различных программ: сервера и клиента.

Программа-сервер создаёт удалённые объекты, делает ссылки на эти объекты доступными и ожидает, пока клиенты не начнут вызывать методы этих объектов.

Программа-клиент получает ссылки на один или несколько удалённых объектов на сервере и вызывает их методы.

Технология RMI предоставляет механизм, с помощью которого сервер и клиент общаются и передают друг другу информацию.

Такие приложения называют распределёнными объектными приложениями.

Распределённые объектные приложения должны осуществлять следующее:

- **Нахождение удалённых объектов.** Приложения могут использовать различные механизмы получения ссылок на удалённые объекты.

  Например, приложение может зарегистрировать свои удалённые объекты с помощью аппарата простых имён — реестра RMI (RMI registry).

  В качестве альтернативы, приложение может передавать и возвращать ссылки на удалённые объекты внутри других удалённых вызовов.
- **Общение с удалёнными объектами.** Детали взаимосвязи между удалёнными объектами управляется RMI.

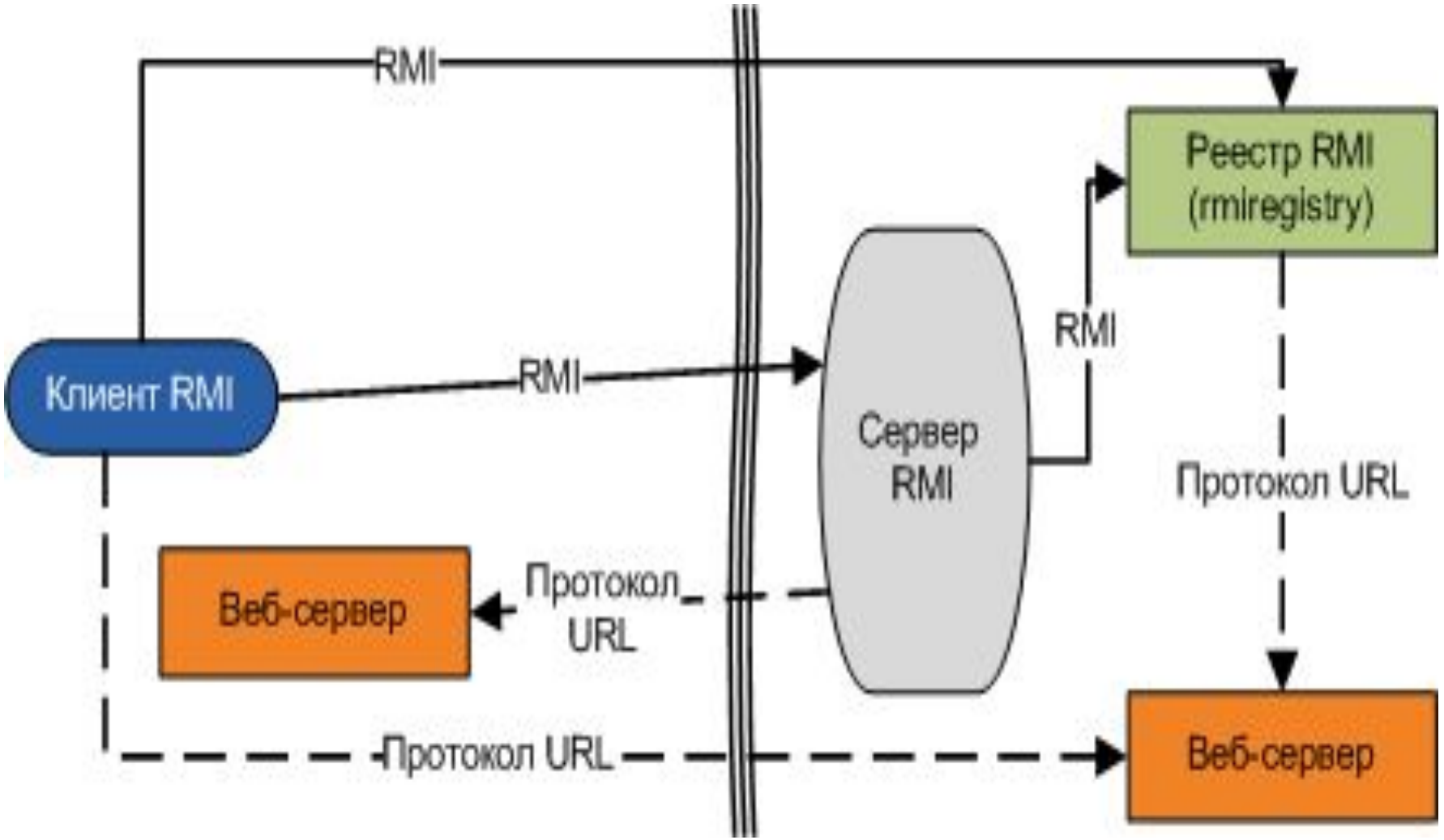
  Для программиста удалённое общение выглядит почти так же, как и вызов обычных методов в Java.
- **Загрузка определений классов, объекты которых передаются.**

  Так как RMI позволяет передавать объекты, он

  также предоставляет механизмы загрузки определений классов

  и передачи информации объектов.

На рисунке показана структура RMI приложения



Сервер вызывает реестр для того, чтобы связать имя с удалённым объектом.

Клиент ищет удалённый объект по его имени в реестре сервера и затем вызывает его метод.

Реализация RMI приложения включает следующие шаги:

1. **Определение удалённых интерфейсов.**

   Удалённый интерфейс определяет методы, которые могут быть удалённо вызваны клиентом.

   Клиент программируется под удалённый интерфейс, а не под классы-реализации этих интерфейсов.

   Проектирование таких интерфейсов включает определение типов объектов, которые будут использоваться в качестве параметров и возвращаемых значений для этих методов.

   Если какие-либо из этих интерфейсов ещё не существуют, то их тоже нужно определить

2. Реализация удалённых объектов.

Удалённые объекты должны реализовать один или несколько удалённых интерфейсов.

Класс удалённого объекта может включать реализации других интерфейсов и методов, которые будут доступны только локально.

Если какой-либо локальный класс используется в качестве параметра или возвращаемого значения хотя бы одного из этих методов, то они должны быть тоже реализованы.

3. Реализация клиентов.

Клиенты, использующие удалённые объекты, могут быть реализованы в любой момент после того, как определены удалённые интерфейсы

Рассмотрим пример RMI приложения

1. Определение удалённых интерфейсов

```
import java.rmi.*;
public interface AddServerIntf extends Remote{
    double add (double d1, double d2) throws
                                        RemoteException;
}
```

2. Реализация удалённых объектов.

```
import java.rmi.*;
import java.rmi.server.*;
/* все удалённые объекты должны расширять
   UnicastRemoteObject, который обеспечивает
   функциональные возможности на удалённых машинах */
public class AddServerImpl extends
        UnicastRemoteObject implements AddServerIntf{
  public AddServerImpl() throws RemoteException{
      // конструктор серверной реализации
   }
  public double add (double d1, double d2) throws
                                          RemoteException{
      return d1+d2; // реализация метода
  }
}
```

Регистрация RMI объекта в реестре

```
import java.rmi.*;
import java.net.*;
public class AddServer{
 public static void main (String[] args){
   try{
     // создаём экземпляр «обработчика»
     AddServerImpl ASI = new AddServerImpl();
     // метод rebind обновляет RMI-реестр и
     // связывает имя(“Plus”) с объектной ссылкой(ASI)
    Naming.rebind(“rmi://localhost:1099/Plus”, ASI);
   }catch(Exception e){
     e.printStackTrace ();
   }
  }
}
```

3. Реализация клиентов.

```
import java.rmi.*;
public class AddClient{
  public static void main (String[] args){
  try{
     // получаем ссылку на удалённую реализацию
    AddServerIntf plus = (AddServerIntf)Naming.lookup(
                                                “rmi://localhost/Plus”);
    // Непосредственно обращение к удалённому
        методу:
    System.out.println(“The sum is: ” + plus.add(8,9));
     }catch(Exception e){
          e.printStackTrace ();
     }
  }
}
```

Рассмотрим использование rmi.

1. Компилируем все файлы

>javac AddServerIntf.java

>javac AddServerImpl.java

>javac AddServer.java

>javac AddClient.java

2.Генерируем заглушки и скелеты, для этого нужно использовать инструмент, называемый компилятором RMI:

>rmic AddServerImp

3. Запускаем rmiregistry(сервер)

>rmiregistry

4. На сервере запускаем

>java AddServer

4. Запуск клиента

>java AddClient

На экране получаем

The sum is: 17

Enterprise JavaBeans

Архитектура технологии EJB

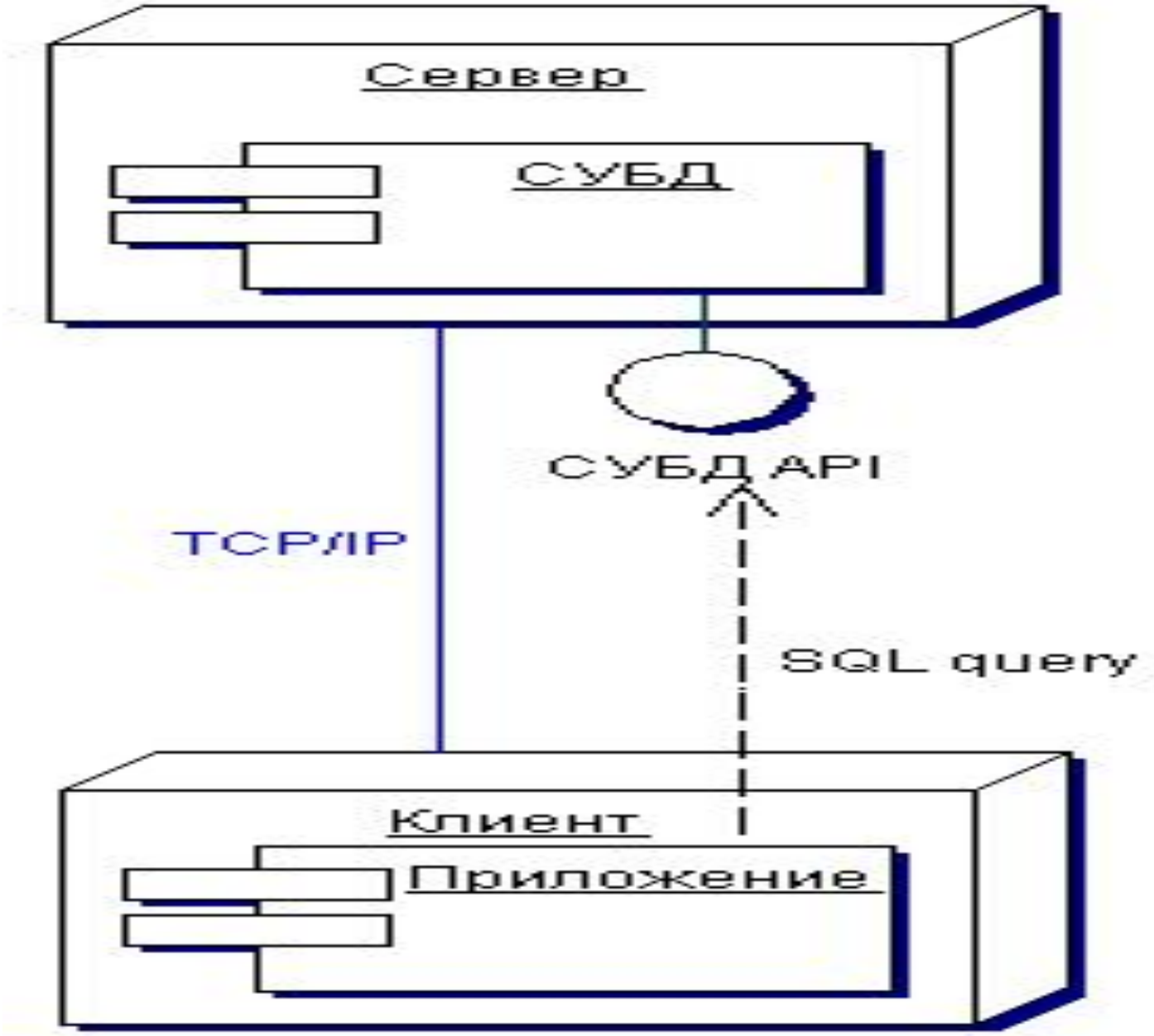
Чаще всего системы строятся следующим образом.

Есть клиентское приложение, которое соединяется с сервером БД и посредством SQL запросов манипулирует данными, отображаемыми в клиентском GUI интерфейсе.

Клиентская часть таких систем обычно сложная и на сервер баз данных возлагается, в основном задача, хранения и поддержки целостности данных.

Иногда базы данных поддерживают хранимые процедуры, что позволяет снизить сетевой трафик между сервером и клиентом.

Такая система имеет вид:


Сервер
СУБД
СУБД API
TCP/IP
SQL query
Клиент
Приложение

Такой подход имеет свои плюсы и минусы.

В плюс идет относительно простая архитектура системы и относительно высокая скорость работы при небольшом количестве клиентских обращений к серверу.

В минус идет то, что такую систему сложно модернизировать, так как изменение в БД влекут за собой изменения в клиентской части и наоборот.

В случае нехватки ресурсов сервера, приходится либо наращивать его вычислительную мощность либо использовать распределенную БД, которая не всегда сможет решить возникшую проблему.

Существует другой подход построения информационных систем.

Система разделяется на три уровня.

Каждый уровень имеет свои обязанности и функциональные возможности.

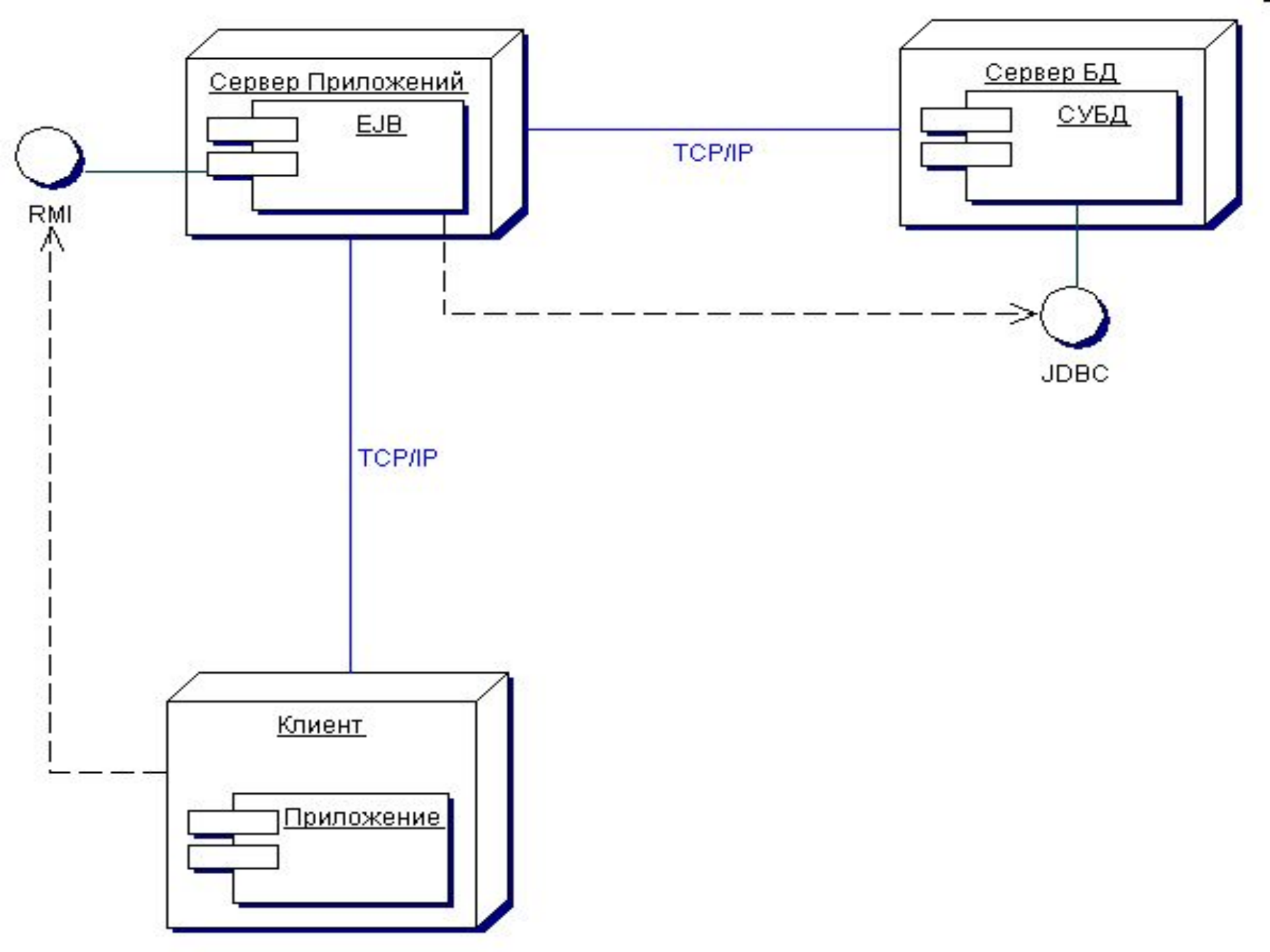
На первом уровне находится клиентское приложение, которое обычно "легкое" и занимается в основном презентационным слоем системы.

Второй уровень отвечает за бизнес логику системы и взаимодействует с презентационным слоем, отвечая на его запросы.

Вторым уровнем называют сервер приложения.

На третьем уровне находится база данных, которая, отвечает за хранения данных и за их целостность.

Такая система имеет вид:


Сервер Приложений
EJB
RMI
TCP/IP
Сервер БД
СУБД
JDBC
TCP/IP
Клиент
Приложение

Такой подход тоже имеет свои плюсы и минусы.

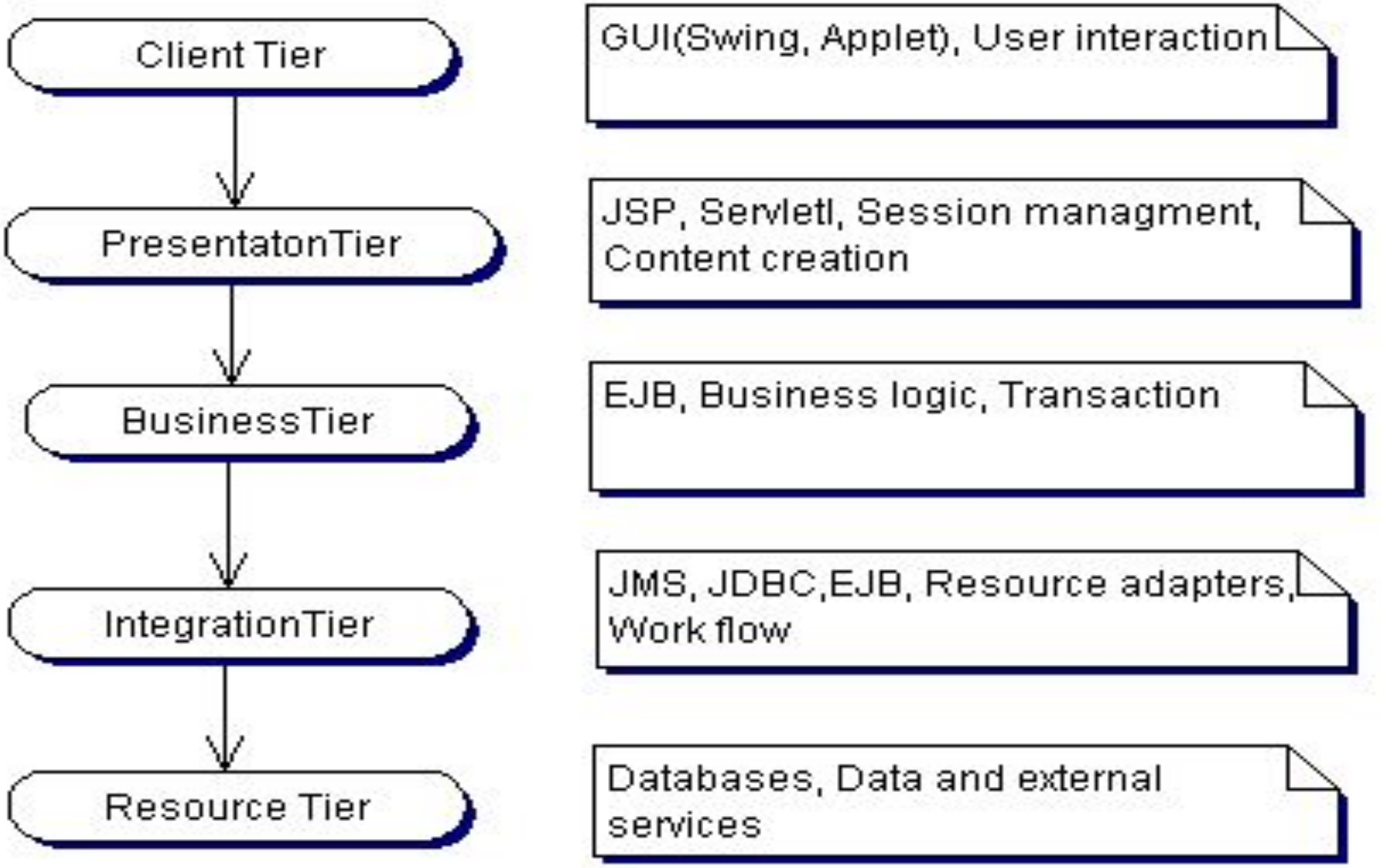
В плюс идет:

1. разделение системы на уровни, позволяющее относительно легко модернизировать систему.
2. возможность групповой работы над системой, в которой каждый из уровней разрабатывается независимо.
3. компонентная технология EJB ориентирована на возможность распределения второго уровня, т.е. если сервер приложений не справляется с нагрузкой, то есть возможность без единого изменения кода сервера приложений, разнести его на несколько вычислительных машин.

   Компоненты, из которых состоит второй уровень, не будут чувствовать разницы между работой на одной вычислительной машине и на нескольких машинах.

Минусом таких систем является их направленность на крупные корпоративные решения.

Продвигается 5 уровневая архитектура на основе EJB:



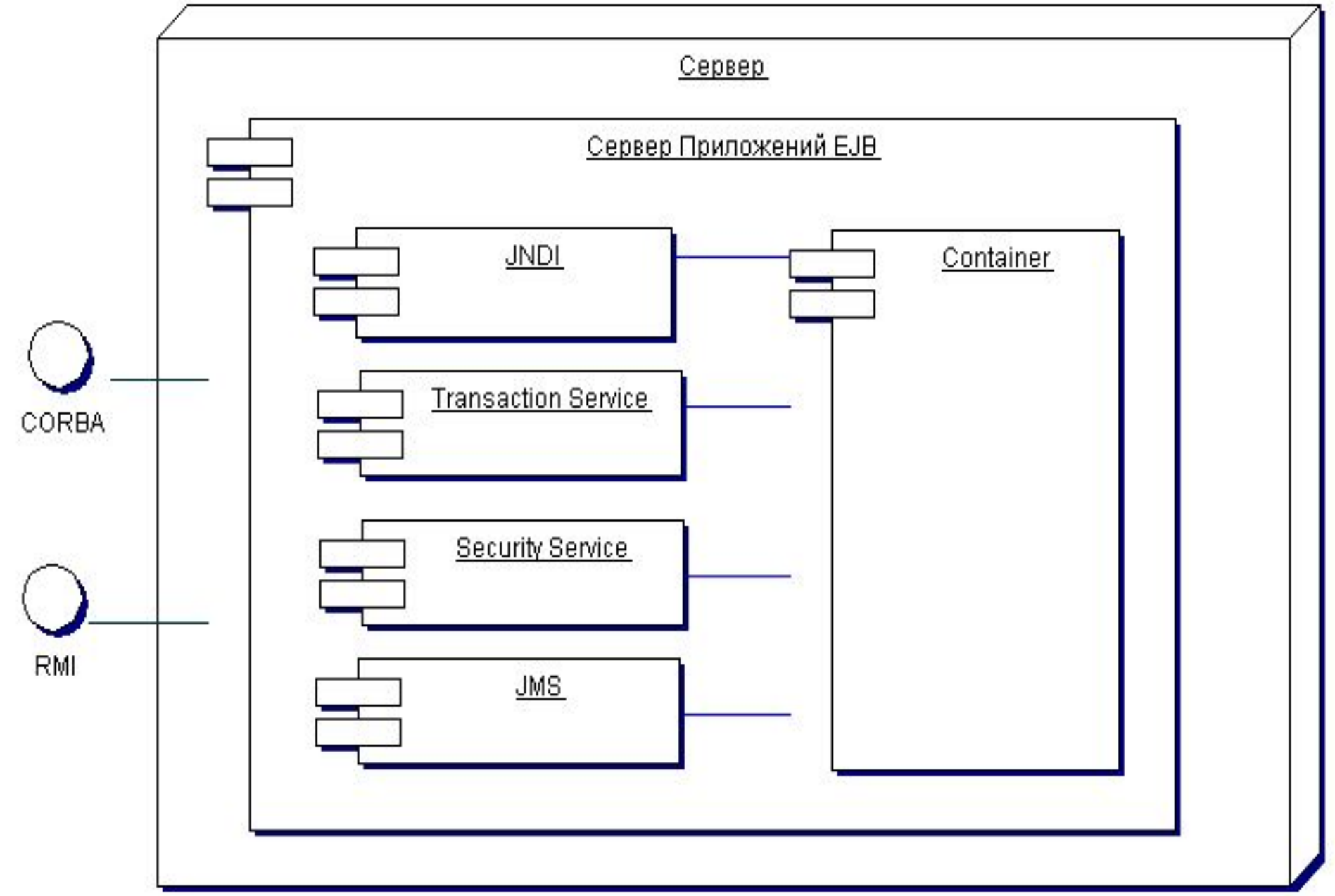
Основное в EJB это сервер приложений.

Клиентские приложения будут общаться с ним через RMI или CORBA.

Обычно сервер приложений предоставляет EJB компонентам соответствующую среду:

1. хранит права доступа к компонентам (а точнее логины с паролями по доступу к серверу приложений)
2. поддерживает RMI и CORBA взаимодействие с ними
3. предоставляет JNDI сервис (сервис именования EJB компонентов),
4. является координатором транзакций
5. предоставляет контейнер, в котором будут храниться EJB компоненты,
6. предоставляет (не всегда) сервис асинхронных сообщений JMS.

Сервер приложений имеет вид:



JNDI (Java Naming Directory Interface) - эта служба позволяет клиентскими приложениям находить на сервере приложений EJB компоненты по их имени.

Другими словами, можно взять множество компьютеров, объединить их в сеть, установить на них сервера приложений.

Но сервис JNDI включить только на одном из них.

И получится, что все компоненты EJB будут доступны на одном дереве имен, а работать на разных серверах приложений.

Transaction Service - сервис транзакций.

Этот сервис предоставляет услуги транзакций, как в обычных реляционных базах данных.

Однако, вместо SQL запросов вызываются методы изменяющие состояния компонентов и как только началась транзакция, то все объекты, с которыми осуществляется работа будут в нее вовлечены.

Security Service - сервис безопасности.

Так как сервер приложений предоставляет удаленный доступ к EJB компонентам, то этот доступ можно ограничить.

Для этого этот сервис создан.

JMS (Java Message Service) - сервис асинхронных сообщений.

Есть возможность послать сообщение и не ждать подтверждение о получении или ответа.

JMS берет всю ответственность за доставку и хранения очередей сообщений, что значительно разгружает клиентские приложения.

Контейнер

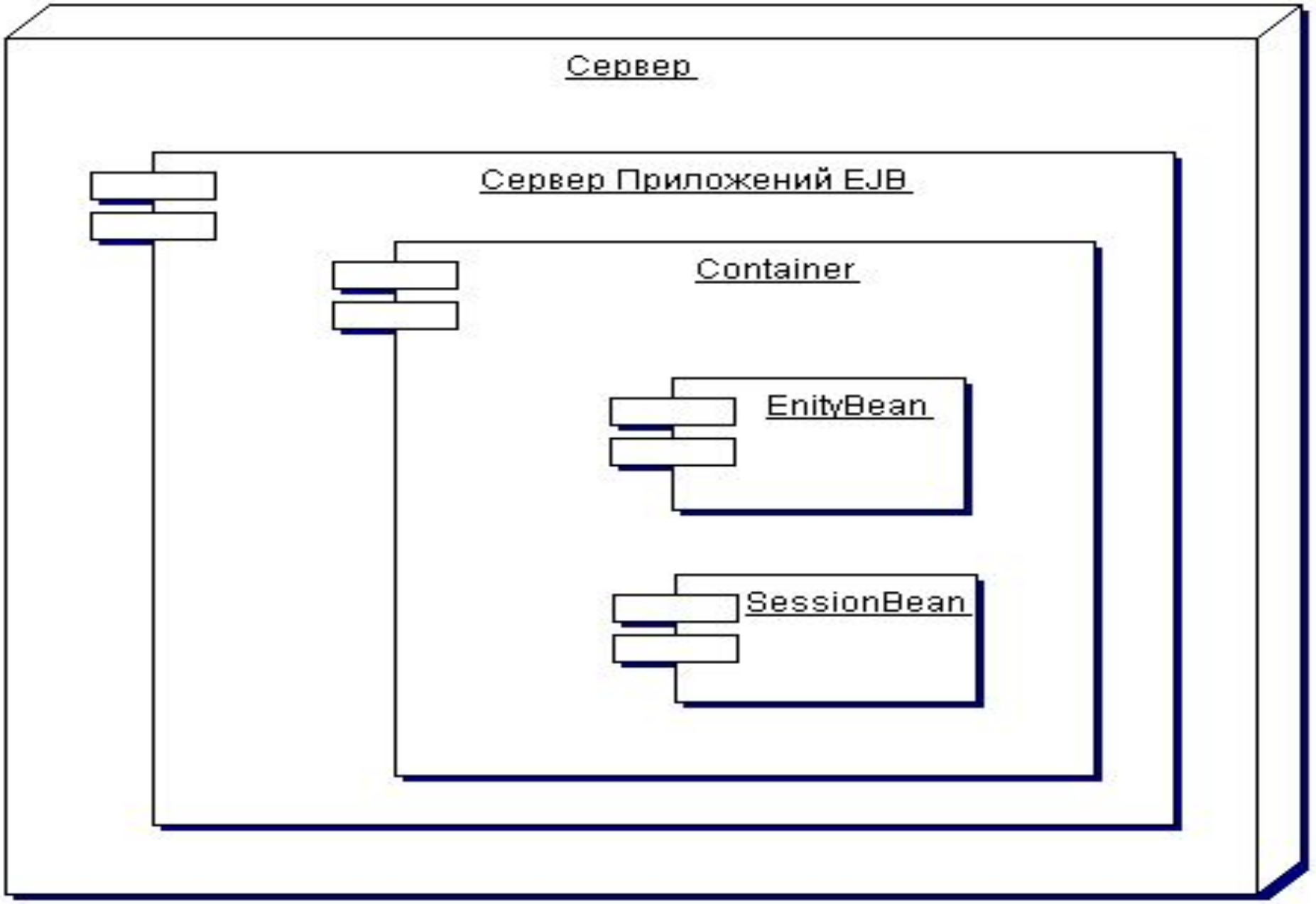
Контейнер предоставляет среду, в которой могут функционировать компоненты EJB.

Функции контейнера:

- Разбор XML-описания компонента EJB (deployment descriptor) и поддержка конфигурации, описанной в этом XML-файле.
- Управление жизненным циклом компонента EJB, т.е. для предоставления услуг компонента прописанных в его интерфейсе и зарегистрированном на JNDI, необходимо создавать, уничтожать и кэшировать реализации (объекты), которые будут отвечать на запросы клиентов.
- Разбалансировка нагрузки между реализациями (объектами) обслуживающими компонент EJB и управление пулом таких объектов.
- Управление транзакциями в компонентах EJB.

  В случае с компонентами, которые работают с СУБД, управление транзакциями сильно связано с механизмом синхронизации состояния компонентов с состоянием СУБД.
- Управление безопасностью доступа к компонентам. Опционально эта функция может быть отключена и проверку прав доступа к методам компонента придется реализовывать своими руками в самом компоненте.

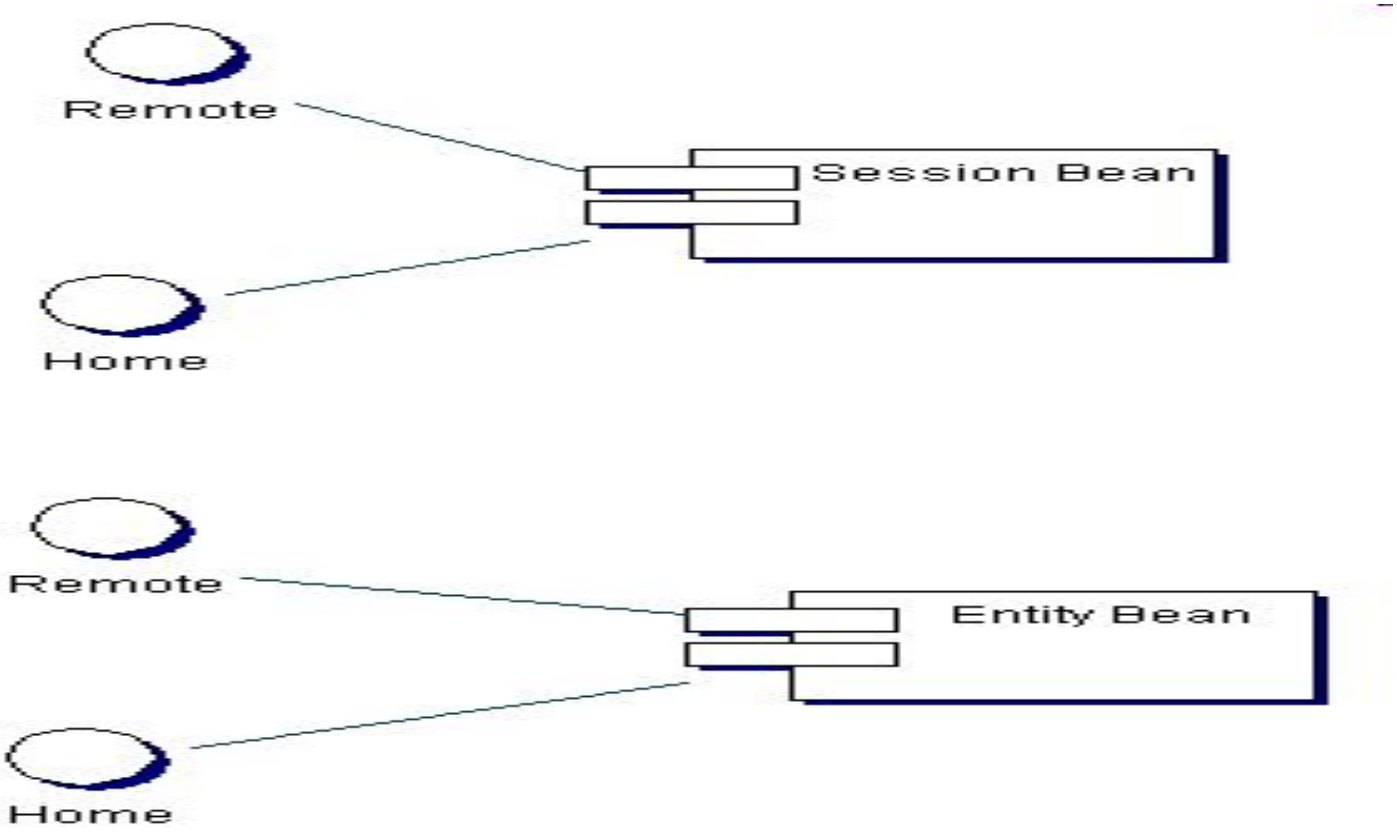
Контейнер имеет вид:



Компонентная модель

Компоненты EJB имеют два внешних описания (интерфейса).

Через них, собственно, клиент и взаимодействует с компонентом.



Home-интерфейс является точкой входа в компонент.

Другими словами любое начало взаимодействия с компонентами происходит через Home-интерфейсы.

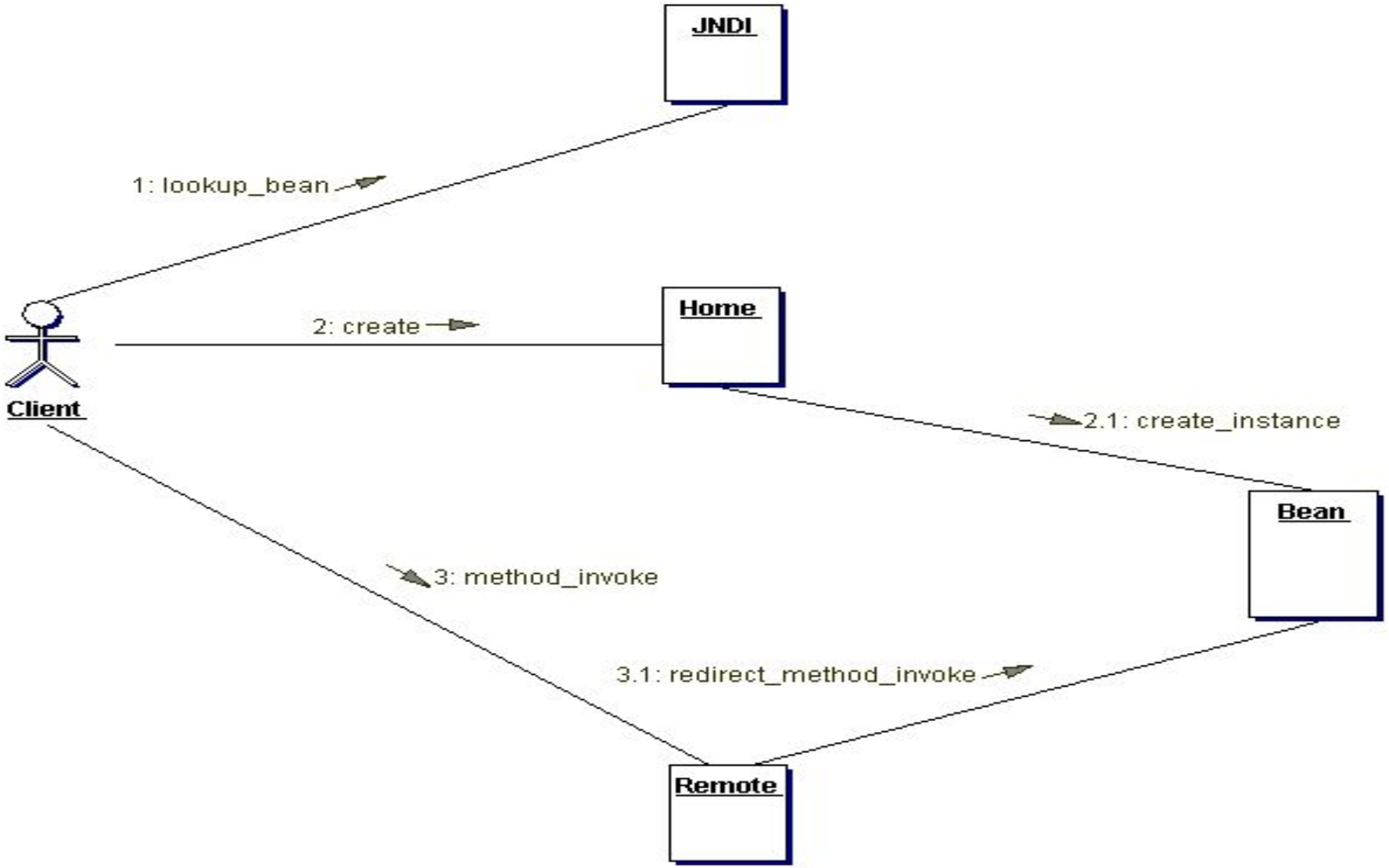
Клиент обращается к интерфейсу и создает через него экземпляры (объекты), которые обслуживают данный компонент.

А в конце своей работы он их уничтожает.

Remote-интерфейс позволяет взаимодействовать с экземплярами (объектами), которые были созданы через фабрику (Home-интерфейс).

Через Remote-интерфейс пользователь вызывает бизнес-методы компонента, которые естественно придется реализовывать, описывая логику приложения.

Рассмотрим стандартный сценарий взаимодействия клиента с компонентами EJB.



1. Клиент ищет Home-интерфейс нужного ему компонента по его имени через сервис имен JNDI (клиенту возвращается в результате поиска Home-интерфейс этого найденного компонента).
2. Клиент, через найденный Home-интерфейс, вызывает функцию создания экземпляра компонента на стороне сервера (клиенту возвращается Remote-интерфейс созданного экземпляра компонента).

2.1. Сервер создает этот экземпляр.

3. Клиент вызывает бизнес-метод на созданном компоненте через его Remote-интерфейс этого компонента.

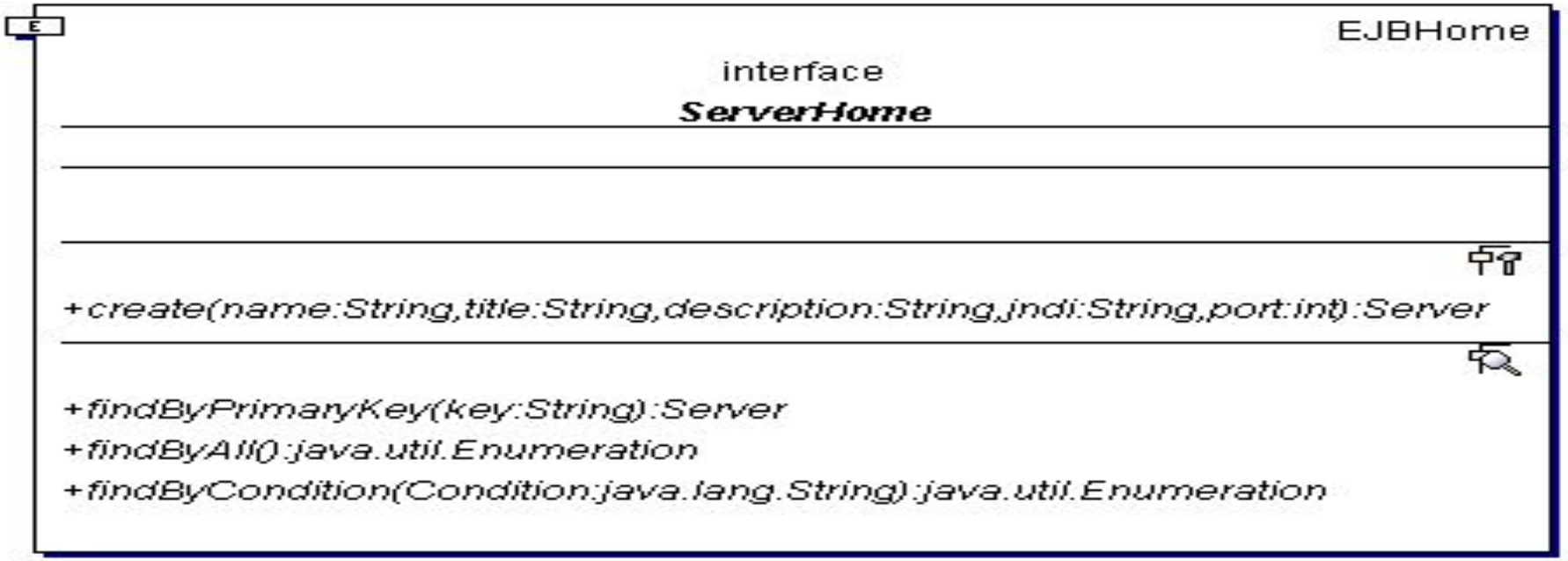
3.1. Сервер вызывает бизнес-метод на конкретном экземпляре компонента.

Home-интерфейс

Таким образом вся работа с компонентами начинается с обращения к Home-интерфейсу.

Каждый тип компонент должен его иметь.

Пример Home-интерфейса:



В этом интерфейсе необходимо определить методы двух типов.

Это фабричные методы create и поисковые find.

Фабричные методы позволяют создавать для себя экземпляры компонентов на стороне сервера.

При вызове этого метода можно передать параметры инициализации компонента.

Можно иметь несколько фабричных методов с разным числом параметров.

При вызове фабричного метода возвращается ссылка созданного компонента на стороне сервера.

Получив эту ссылку, можно начать общение с созданным компонентом, т.е. вызывать его бизнес методы.

Поисковые методы позволяют найти уже созданные компоненты на стороне сервера.

Поисковые методы применимы только к компонентам, которые называются EntityBean или сущностные бины.

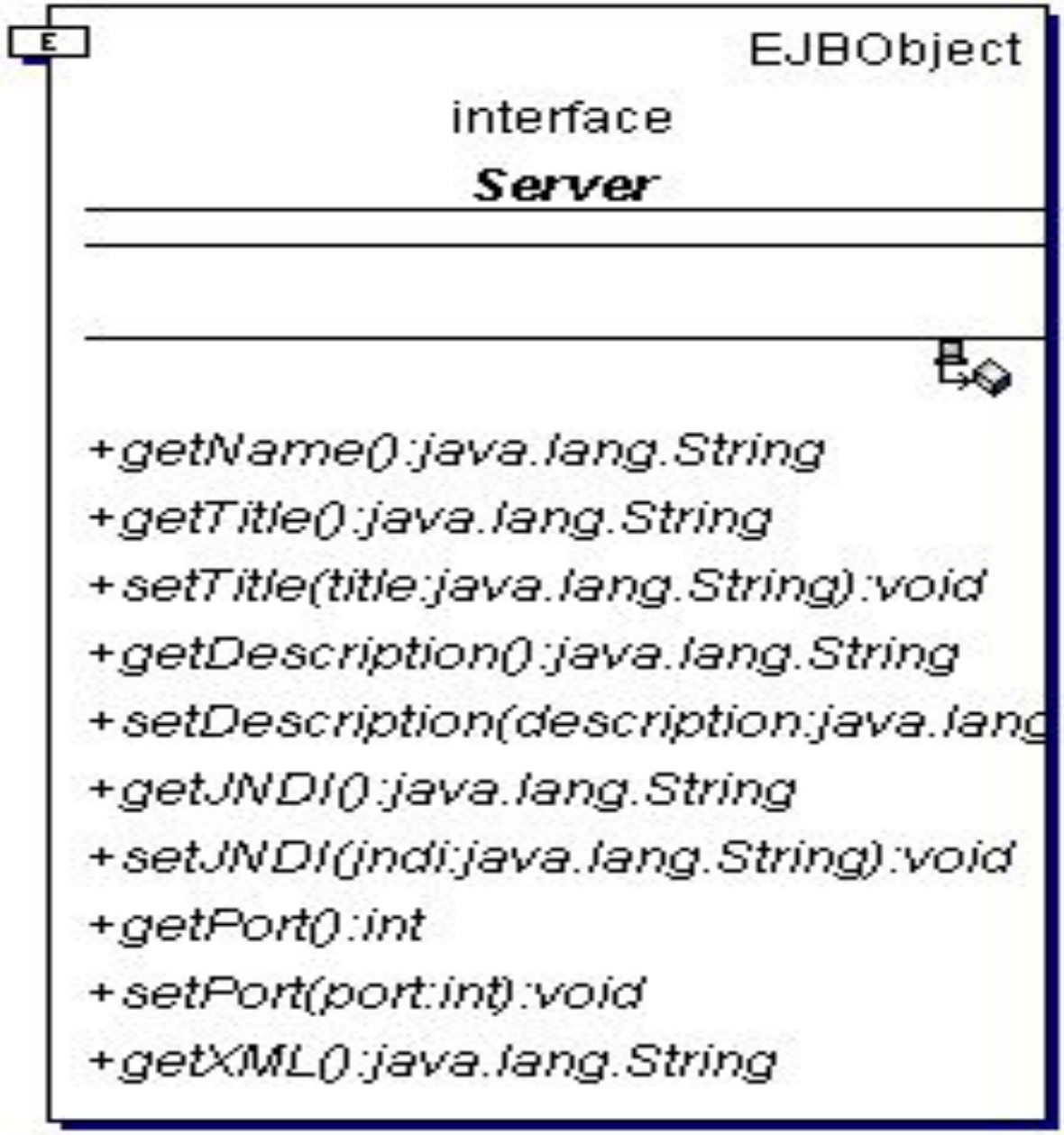
Другими словами, время жизни таких компонентов превышает время работы сервера приложений и, чаще всего, состояние таких компонентов отображается в реляционные базы данных.

Remote-интерфейс

После того, как компонент был создан или найден через его Home-интерфейс и получена ссылка на его Remote-интерфейс, можно приступить к взаимодействию с этим EJB-компонентом.

Все способы взаимодействия с компонентом строго определены в полученном Remote-интерфейсе.

Пример Remote-интерфейса:


EJBObject
interface
Server
+getName():java.lang.String
+getTitle():java.lang.String
+setTitle(title:java.lang.String):void
+getDescription():java.lang.String
+setDescription(description:java.lang
+getJNDI():java.lang.String
+setJNDI(jndi:java.lang.String):void
+getPort():int
+setPort(port:int):void
+getXML():java.lang.String

Стандартом, конечно, являются get/set-методы, считывающие и устанавливающие состояния параметров EJB-компонентов. Можно определить любые методы в Remote-интерфейсе.

Реализация компонента

После того как были определены Home и Remote интерфейсы своего компонента, необходимо написать реализации методов определенных в них.

К некоторым методам в реализации добавляется приставка ejb.

Пример реализации имеет вид:


EntityBean
ServerBean
-ctx:EntityContext
-ds:DataSource
-name:String
-title:String
-description:String
-jndi:String
-port:int
-serverHome:ServerHome
+setEntityContext(ctx:EntityContext):void
+unsetEntityContext():void
+ejbActivate():void
+ejbPassivate():void
+ejbRemove():void
+ejbStore():void
+ejbLoad():void
-getConnection():Connection
+ejbCreate(name:String,title:String,description:String,jndi:String,port:int):String
+ejbPostCreate(name:String,title:String,description:String,jndi:String,port:int):void
+ejbFindByPrimaryKey(key:String):String
+ejbFindByCondition(Condition:String):Enumeration
+ejbFindByAll():java.util.Enumeration
+getTitle():String
+setTitle(title:String):void
+setDescription(description:String):void
+getDescription():String
+setJNDI(jndi:String):void
+getJNDI():String
+setPort(port:int):void
+getPort():int
+getName():String
+getXML():String
+toString():String

- **ctx** - ссылка на объект, которая позволяет компоненту получать служебную информацию о пользовательских транзакциях и данные о том какой пользователь работает с компонентом.
- **ds** - ссылка на пул соединений с базой данных.
- **name, title, description, jndi, port** - параметры компонента доступные через методы Remote-интерфейса
- **serverHome** - ссылка на Home-интерфейс компонента Server .
- **setEntityContext/unsetEntityContext** - методы, в которых устанавливается ctx. Вызываются только контейнером.
- **ejbActivate/ejbPassivate** - методы управляющие жизненным циклом компонента. Вызываются только контейнером.
- **ejbRemove** - метод который вызывается перед уничтожением компонента на стороне сервера. Для сущностного бина, например, реализует запрос в базу данных на удаление этого компонента из базы.
- **getConnection** - метод который вызывают для взятия соединения из пула соединений (см. ds). Его определяют больше для удобства и он к спецификации EJB не имеет ни какого отношения.

- **ejbCreate** - методы которые реализует create методы из Home-интерфейса.

  Например, для сущностных бинов в нем реализуют запрос к базе данных для создания компонента и в нем устанавливают параметры компонента.
- **ejbPostCreate** - методы вызываются после ejbCreate.
- **ejbFind** - методы реализуют find методы определенные в Home-интерфейсе и производят поиск компонентов в базе данных.
- **get/set** - методы реализуют get/set методы определенные в Remote-интерфейсе.
- **toString** - определен для совместимости с инфраструктурой JAVA.

  К спецификации EJB не имеет ни какого отношения.

Дерево имен JNDI

Каждому компоненту EJB сопоставляется имя, которое публикуется на дереве имен JNDI.

И клиентское приложение обращается к этой службе зная имя под которым зарегистрирован EJB компонент.

Обратившись к службе имен клиентское приложение получает по имени объектную ссылку.

Так как изначально полагается что компоненты работают в разных адресных пространствах с клиентским приложением, т.е. в разных виртуальных машинах и все их взаимодействие является сетевым, то тогда понятие объектной ссылки стоит обобщить.

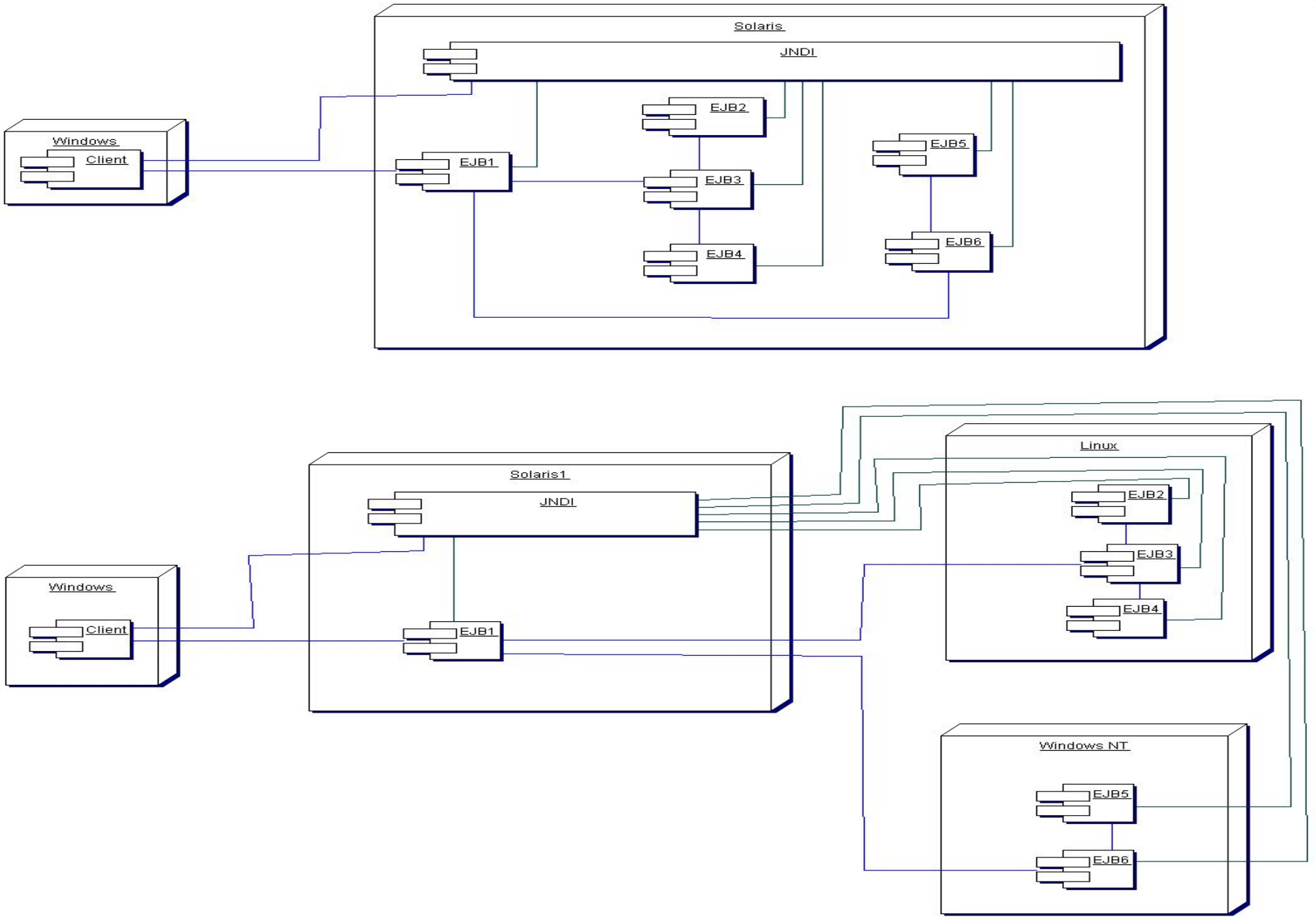
Классически объектная ссылка это адрес в памяти, но в случае удаленного взаимодействия это адрес хоста, номер порта, плюс служебная информация и плюс еще к тому что эта объектная ссылка уникальная и ее значение не возможно предсказать.

Самым первый способ получения объектной ссылки выглядит так: запускается компонент на стороне сервера и полученная объектная ссылка записывается в файл и передается на сторону клиента.

Сервис именования JNDI позволяет по имени получить объектную ссылку компонента.

Сценарий запуска сервера и взаимодействия клиента уже выглядит следующим образом: запускается компонент на стороне сервера, который себя регистрирует на дереве имен JNDI под заранее оговоренным с клиентом именем, а потом клиентское приложение через сервис JNDI по имени получает объектную ссылку на этот компонент.

Схематически это выглядит следующим образом:



Сессионные бины

Сессионный бин по функциональности очень похож на обычный класс, от которого можно порождать объекты и использовать.

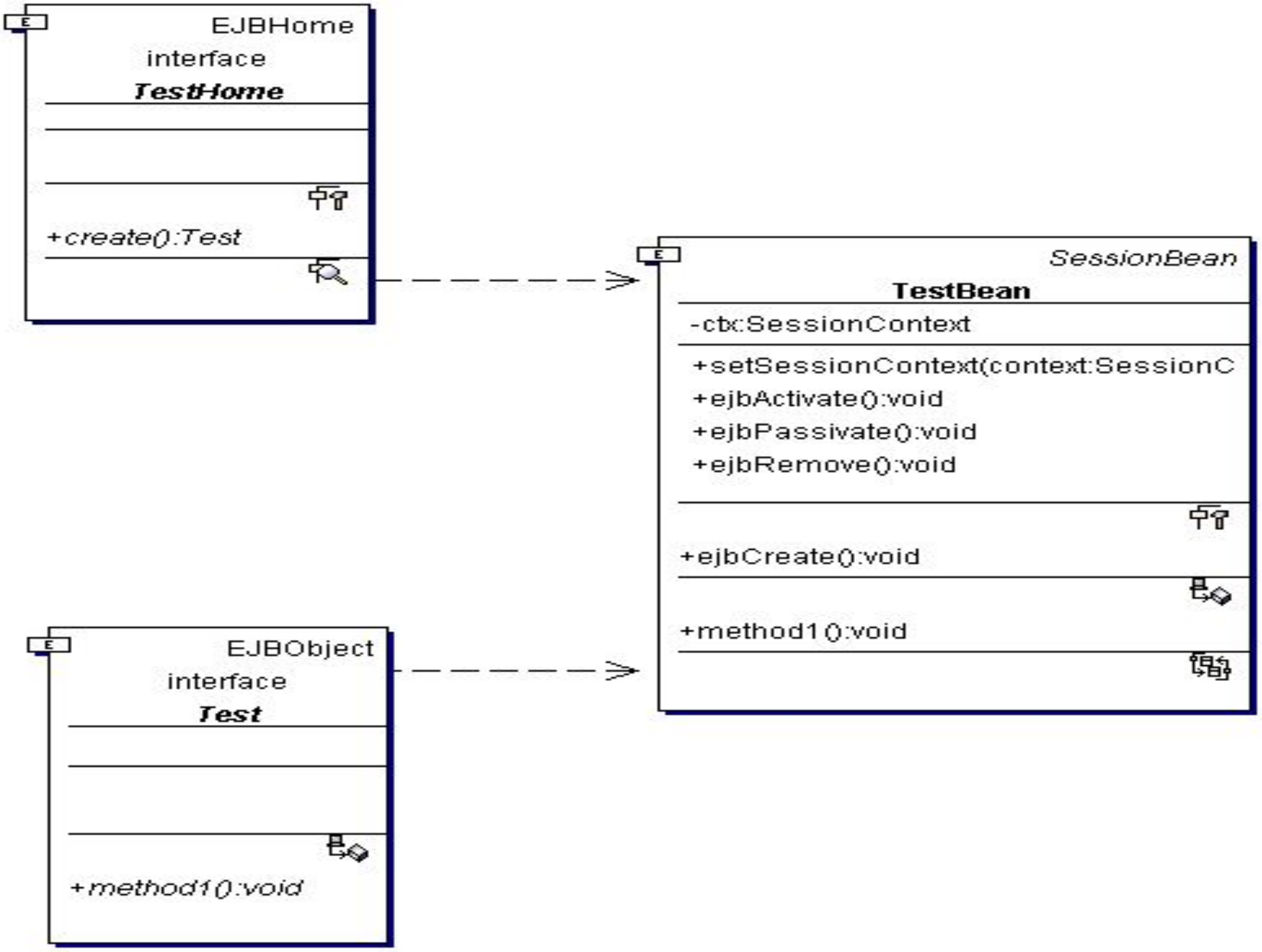
Отличительной чертой является способ создания объекта.

Как уже говорилось выше существуют Home-интерфейс, который является точкой входа в бин.

В нем определяется метод create, через который можно создавать сессионные бины на стороне сервера.

Вообще сессионные бины предназначены быть представителем клиента на стороне сервера или быть его функциональным расширением, другими словами они нужны в течении сессии работы клиента, а потом их уничтожают.

Сессионный бин имеет вид:


EJBHome
interface
TestHome
+create():Test
SessionBean
TestBean
-ctx:SessionContext
+setSessionContext(context:SessionC
+ejbActivate():void
+ejbPassivate():void
+ejbRemove():void
+ejbCreate():void
+method1():void
EJBObject
interface
Test
+method1():void

Сессионные бины бывают двух видов: Stateless and Stateful.

Другими словами бины могут не помнить свое состояние и помнить.

Сессионный бин не помнящий свое состояние

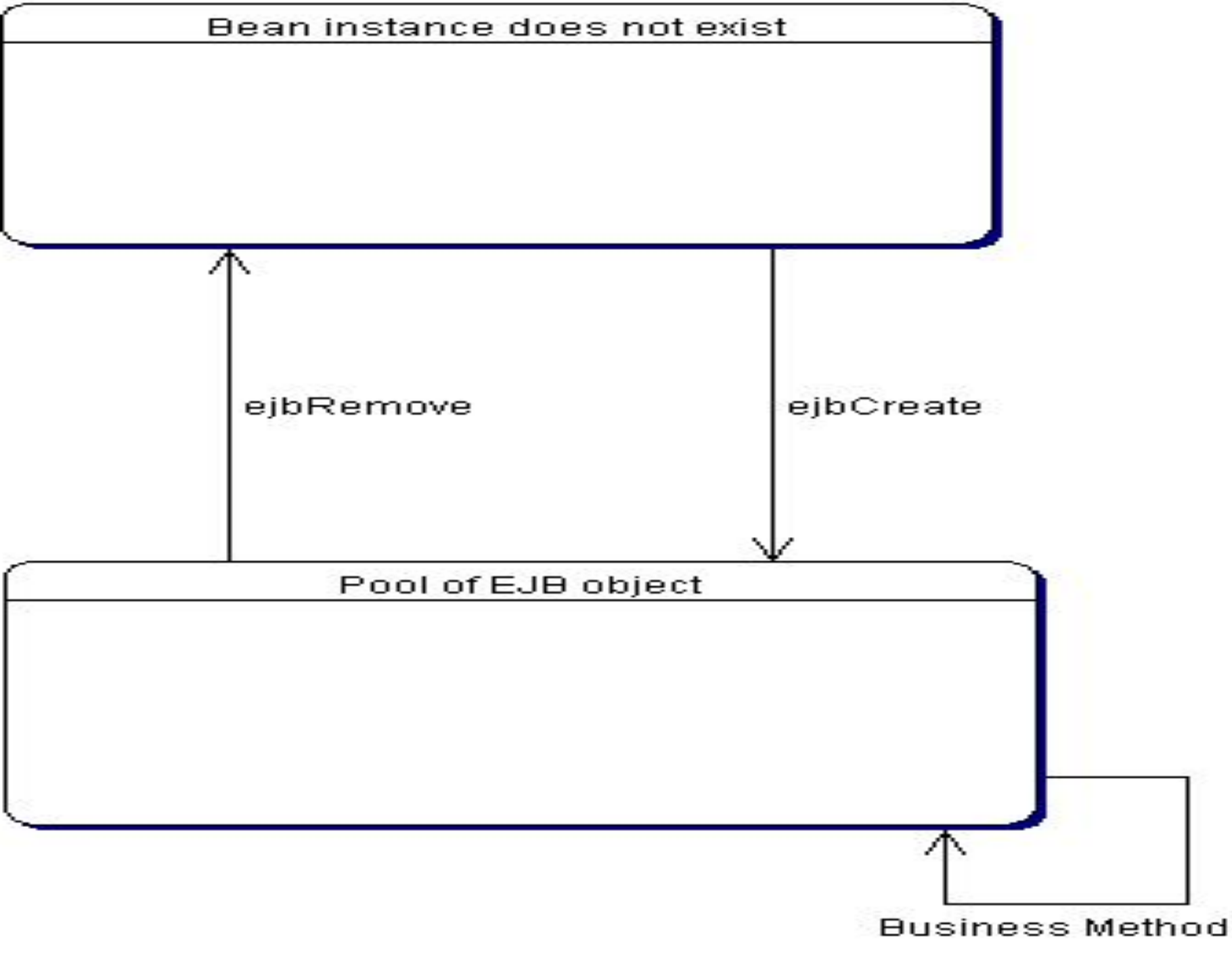
Бывает ситуации, когда не нужно, что бы бин помнил свое состояния между двумя вызовами методов, т.е. информация в бине не сохраняется с помощью методов set и в последующем не запрашивается методами get.

Сессионный бин помнящий свое состояние

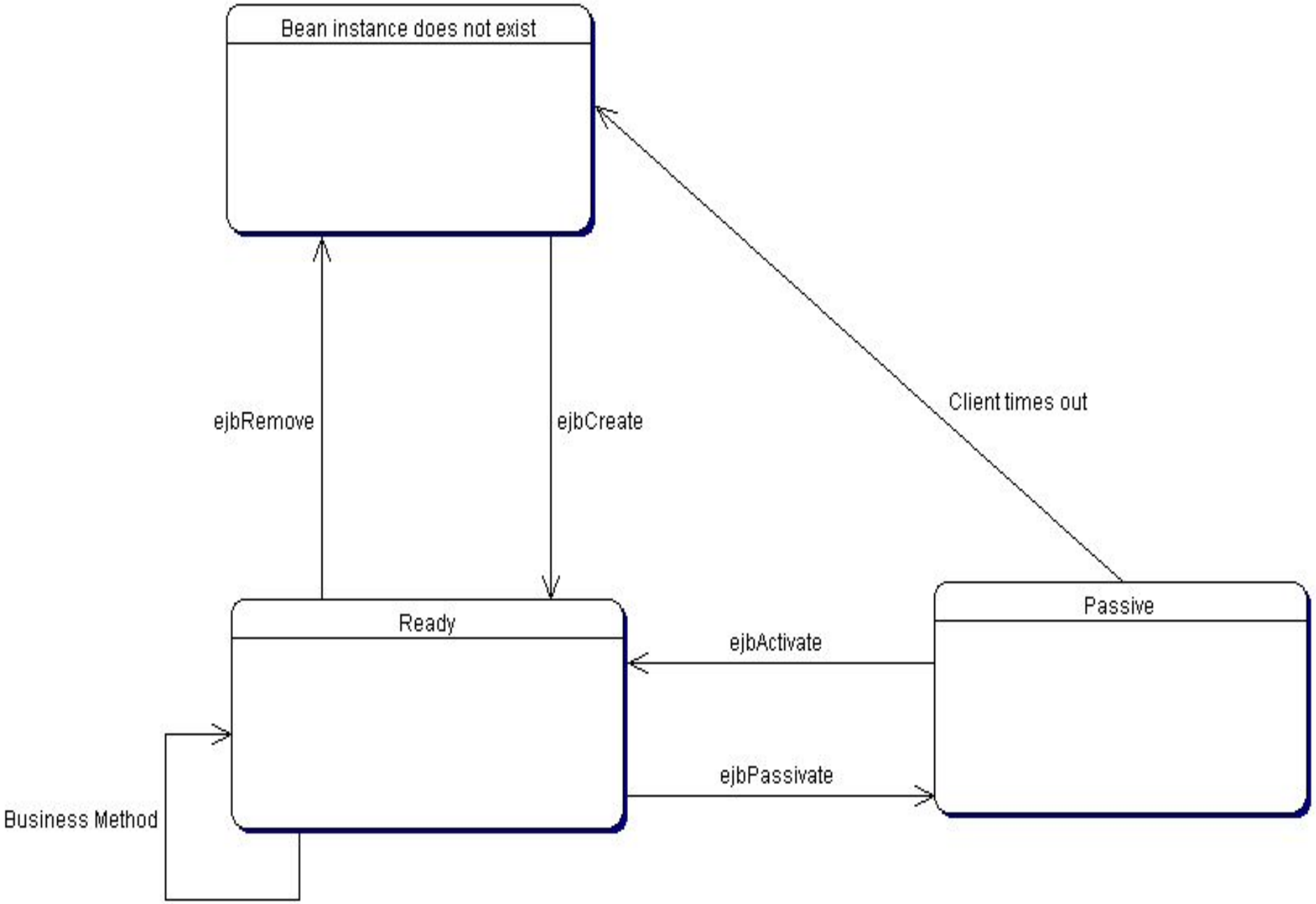
Бывает ситуации, когда необходимо, что бы бин помнил свое состояния между двумя вызовами методов, т.е. информация о состоянии запоминается в бине с помощью методов set и в последующем восстанавливается методами get.

Жизненный цикл сессионных бинов.

Рассмотрим жизненный цикл Stateless сессионного бина.


Bean instance does not exist
ejbRemove
ejbCreate
Pool of EJB object
Business Method

Жизненный цикл в Stateful сессионного бина имеет вид:



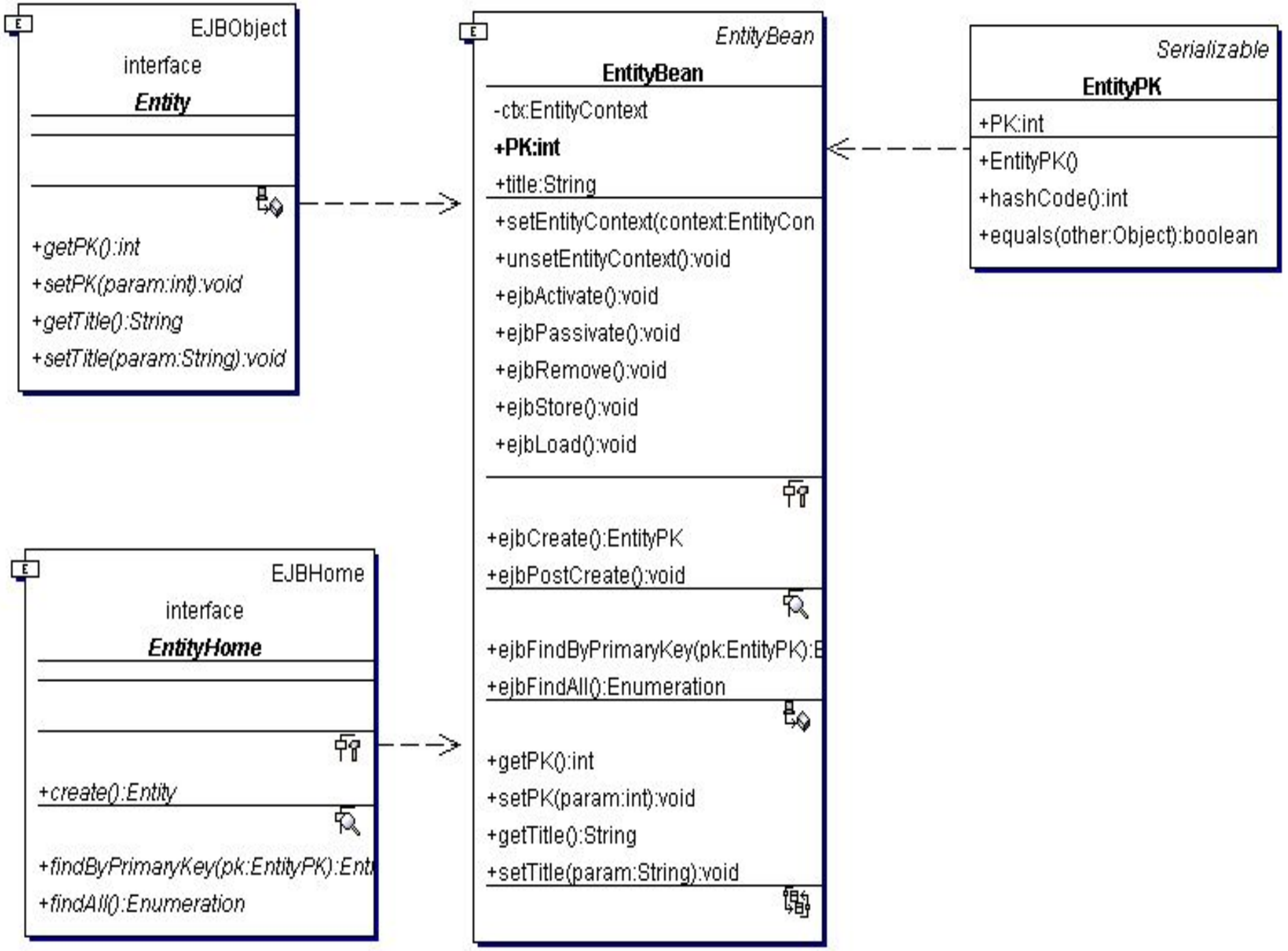
Идея заложенная в сущностные бины следующая.

Необходимо хранить информацию в реляционных таблицах и обеспечить к ним гибкий доступ используя ООП.

Есть реляционная таблица на нее нужно создать сущностный бин.

Реализация сущностного бина соответствует строчки в базе данных.

Сущностный бин имеет вид:


EJBObject
interface
Entity
+getPK():int
+setPK(param:int):void
+getTitle():String
+setTitle(param:String):void
EntityBean
EntityBean
-ctx:EntityContext
+PK:int
+title:String
+setEntityContext(context:EntityCon
+unsetEntityContext():void
+ejbActivate():void
+ejbPassivate():void
+ejbRemove():void
+ejbStore():void
+ejbLoad():void
+ejbCreate():EntityPK
+ejbPostCreate():void
+ejbFindByPrimaryKey(pk:EntityPK):E
+ejbFindAll():Enumeration
+getPK():int
+setPK(param:int):void
+getTitle():String
+setTitle(param:String):void
Serializable
EntityPK
+PK:int
+EntityPK()
+hashCode():int
+equals(other:Object):boolean
EJBHome
interface
EntityHome
+create():Entity
+findByPrimaryKey(pk:EntityPK):Ent
+findAll():Enumeration

Здесь показан пример бина, который обслуживает таблицу всего с двумя столбцами PK и title.

Также присутствуют поисковые методы ejbFind, которые позволяют выдергивать идентификаторы объектов из базы данных, но не отвечают за загрузку их состояний в память.

Процесс перехода из строчки таблицы в объект называется материализацией, **а сохранения объекта в строчку таблицы - дематериализацией.**

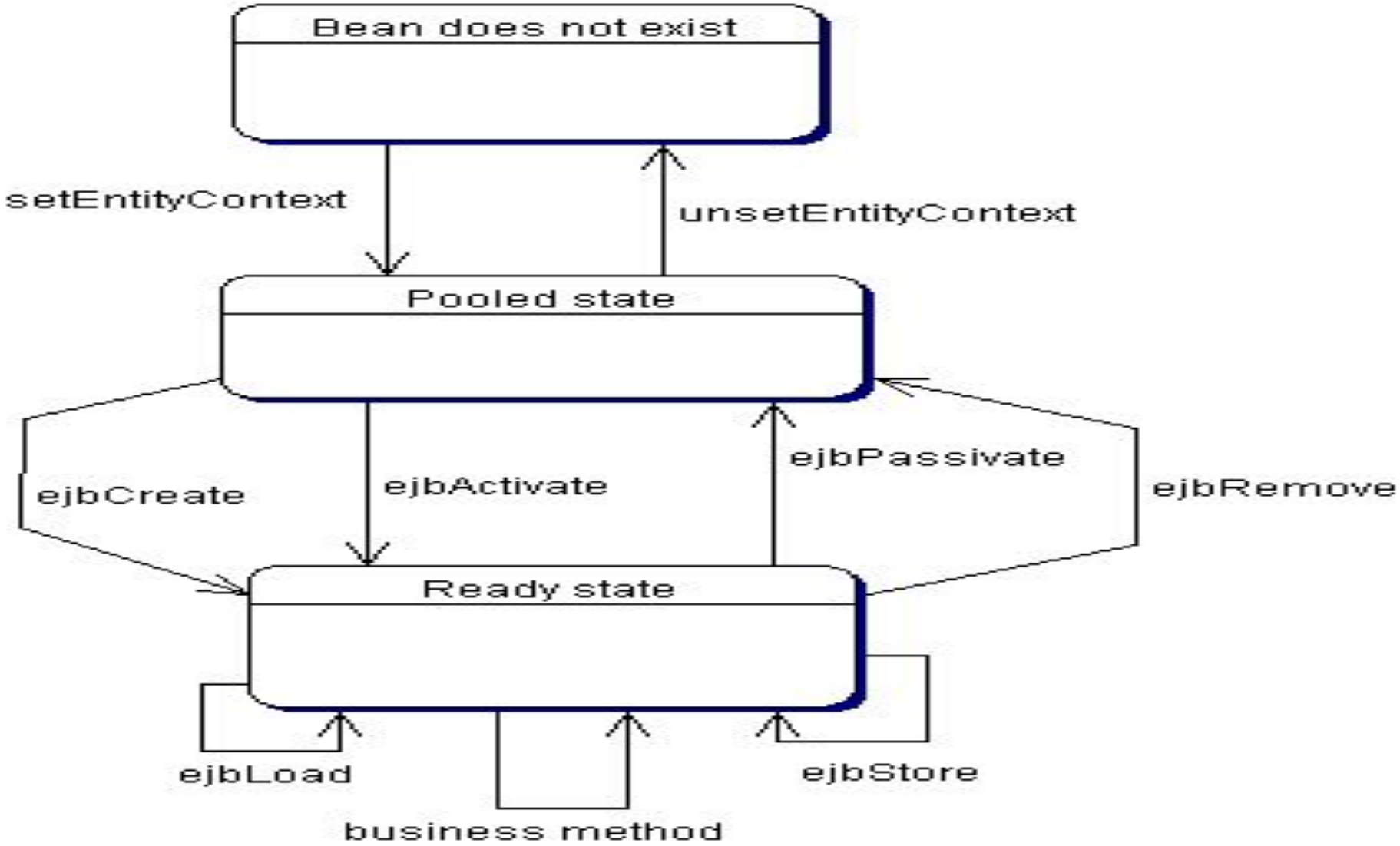
Механизм взаимодействия с сущностными бинами такой же как с сессионными, за исключением того, что на HOME-интерфейсе появляются Find-методы, которые не создают объекты в базе данных, а находят их в БД.

Архитектура сущностного бина отличается понятием основной ключ Primary Key, который представлен в нашем примере как класс EntityPK.

Этот класс обворачивает основной ключ таблицы, которую обслуживает бин.

Жизненный цикл сущностных бинов

Жизненный цикл (ЖЦ) сущностных бинов похож на сессионный, но несколько сложнее:



Рассмотрим пример.

Разработаем сеансовый компонент без состояния (Stateless Session Bean), который в ответ на запрос будет возвращать некоторую строку.

Удаленный интерфейс

Сначала определим удаленный интерфейс (Remote Interface) разрабатываемого компонента в файле greetRemote.java.

Все виды удаленных интерфейсов расширяют (extends) интерфейс javax.ejb.EJBObject.

```
import javax.ejb.*;
import java.rmi.*;
public interface greetRemote extends EJBObject {
  public String greetme(String s) throws RemoteException;
}
```

Здесь присутствует единственный метод greetme() - он возвращает текущее время сервера.

Домашний интерфейс

Теперь определим домашний интерфейс (Home Interface) в файле greetHome.**java**.

```
import javax.ejb.*;
import java.rmi.*;
public interface greetHome extends EJBHome {
    public greetRemote create() throws
                        RemoteException,CreateException;
}
```

Метод create() отвечает за инициализацию экземпляра разрабатываемого компонента.

При необходимости метод create() может быть перегружен, то есть, определены методы create с возможно другим списком формальных параметров.

Класс компонента

```
import javax.ejb.*;
import java.rmi.*;
import javax.naming.*;
public class greetBean implements SessionBean {

   public String greetme(String s) throws RemoteException  {
       return "How are you?......."+s;
    }

 public void ejbCreate(){}
 public void ejbRemove(){}
 public void ejbActivate(){}
 public void ejbPassivate(){}
 public void setSessionContext (SessionContext sc ) {}

}
```

Клиентское приложение

```
import java.net.*;
import javax.ejb.*;
import javax.rmi.*;
import java.rmi.*;
import javax.naming.*;
import java.util.*;
import java.io.*;
import org.jnp.interfaces.NamingContextFactory;

public class Main{
   public static void main(String[] args){
    try{
       Properties props=new Properties();
```

```
 props.put(Context.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");
 props.put(Context.PROVIDER_URL, "localhost:1099");
 props.put(Context.URL_PKG_PREFIXES,
                        "org.jboss.naming:org.jnp.interfaces");
 Context ctx=new InitialContext(props);
 greetHome home = (greetHome)ctx.lookup("greetJndi");
 greetRemote remote=home.create();
 String a="Heloooo";
 String s = remote.greetme(a);
 System.out.println(s);
} catch(Exception e) {
    System.out.println(""+e);
}
}
```

Компиляция и развертывание

Развернем данное приложение с использованием сервера приложений JBoss 3.2.

Прежде всего необходимо создать дескрипторы развертывания ejb-jar.xml и jboss.xml.

Дескриптор jboss.xml необходим для регистрации ejb приложения в службе jndi. Данный файл имеет вид:

```
<?xml version="1.0"?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS 3.2//EN“
                        "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<jboss>
    <enterprise-beans>
        <entity>
            <ejb-name>greetBean</ejb-name>
            <jndi-name>greetJndi</jndi-name>
        </entity>
    </enterprise-beans>
</jboss>
```

Дескриптор ejb-jar.xml описывает само приложение ejb и имеет вид:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
  Enterprise JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-
                                                    jar_2_0.dtd">
<ejb-jar>
      <enterprise-beans>
        <session>
            <ejb-name>greetBean</ejb-name>
            <home>greetHome</home>
            <remote>greetRemote</remote>
            <ejb-class>greetBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container </transaction-type>
        </session>
      </enterprise-beans>
</ejb-jar>
```

В этом дескрипторе развертывания имеются следующие элементы:

<ejb-jar> - корневой элемент.

В нем обязательно должен быть один элемент <enterprise-beans>, а также дополнительные элементы.

<description> - словесное описание.

<display-name> - этот элемент представляет собой идентификатор, который могут использовать некоторые программы, работающие с дескриптором развертывания (Eclipse, JBuilder, NetBeans).

<enterprise-beans> - содержит объявления всех компонентов, описываемых этим дескриптором развертывания.

<session> - в этом элементе описывается сеансовый компонент (Session Bean).

<ejb-name> - имя компонента.

<home> - полное имя класса домашнего интерфейса.

<remote> - полное имя класса удаленного интерфейса.

<ejb-class> - полное имя класса компонента. Этот класс реализует прикладные методы компонента.

<session-type> - тип сеансового компонента. Stateless - без состояния, Stateful - с состоянием.

<transaction-type> - определяет, управляет ли сеансовый компонент сам своими транзакциями, или за него это делает контейнер EJB.

Значение Container - управляет контейнер, значение Bean - управляет компонент.

Теперь рассмотрим непосредственно установку и компиляцию:

1. Скачать архив JBoss 3.2 и развернуть (установка осуществляется простым копированием), например, в директорию

   C:\boss-3.2.8.SP1

2. Создать директорию, например, mybean на диске C
3. В директории mybean создать поддиректорию META-INF, в нее скопировать файлы ejb-jar.xml и jboss.xml

   Файлы greetHome.java, greetBean.java и greetRemote.java скопировать в каталог mybean.

4. Скомпилировать файлы greetRemote.java и greetHome.java, greetBean.java:

>javac –cp .;c:\jboss-3.2.8.SP1\client\jboss-j2ee.jar *.java

5. Создать jar архив:

>jar cvf greet.jar *.class META-INF*.xml

6. Скопировать greet.jar в каталог c:\jboss-3.2.8.SP1\server\default\deploy\

7. Запустить JBoss : запустить файл run.bat

в каталоге c:\jboss-3.2.8.SP1\bin\

8. Скомпилировать клиент

>javac –cp .;c:\jboss-3.2.8.SP1\client\jboss-j2ee.jar; c:\jboss-3.2.8.SP1\client\jbossall-client.jar Main.java

9. Запустить клиент

>java –cp .;c:\jboss-3.2.8.SP1\client\jbossall-client.jar Main

В случае использование JBoss 6 и ejb 3, код бина поменяется.

Интерфейс бина имеет вид Greet.java

```
public interface Greet {
  public String greetme(String s);
}
```

Удаленный интерфейс будет иметь вид greetRemote.java:

```
import javax.ejb.Remote;
@Remote
public interface greetRemote extends Greet { }
```

Локальный интерфейс greetHome.java будет иметь вид:

```
import javax.ejb.Local;
@Local
public interface greetHome extends Greet { }
```

Код бина имеет вид:

```
import javax.ejb.*;
@Stateless
public class greetBean implements greetHome,
                                        greetRemote {
   public String greetme(String s)  {
       return "How are you?......."+s;
    }
}
```

Файл ejb-jar.xml имеет вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                  http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"

   version="3.0">
  <description>JBoss Stateless Session Bean Tutorial</description>
  <display-name>JBoss Stateless Session Bean Tutorial</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>Greet</ejb-name>
      <business-local>greetHome</business-local>
      <business-remote>greetRemote</business-remote>
      <ejb-class>greetBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Файл jboss.xml:

```
<?xml version="1.0"?>
<jboss xmlns:xs="http://www.jboss.org/j2ee/schema"
  xs:schemaLocation="http://www.jboss.org/j2ee/schema
                                    jboss_5_0.xsd" version="5.0">
   <enterprise-beans>
      <session>
         <ejb-name>Greet</ejb-name>
         <jndi-name>greetRemote</jndi-name>
         <local-jndi-name>greetHome</local-jndi-name>
      </session>
   </enterprise-beans>
</jboss>
```

Клиент будет иметь вид:

```
import javax.ejb.*;
import javax.naming.*;
import java.util.*;
import java.io.*;

public class Main{
    public static void main(String[] args){
     try{
        Properties props=new Properties();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
                          "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.PROVIDER_URL, "localhost:1099");
        props.put(Context.URL_PKG_PREFIXES,
                                "org.jboss.naming:org.jnp.interfaces");
```

```
  InitialContext ctx=new InitialContext(props);
  Class<? extends Greet> clazz =
                                        greetRemote.class;
  Greet home =
            clazz.cast(ctx.lookup("greetRemote"));
  String a="Heloooo";
  String s = home.greetme(a);
  System.out.println(s);
 } catch(Exception e){
  System.out.println(""+e);
 }
}
```

JMS

JMS является еще одной технологией создания распределенных приложений, основанных на модели обмена сообщениями.

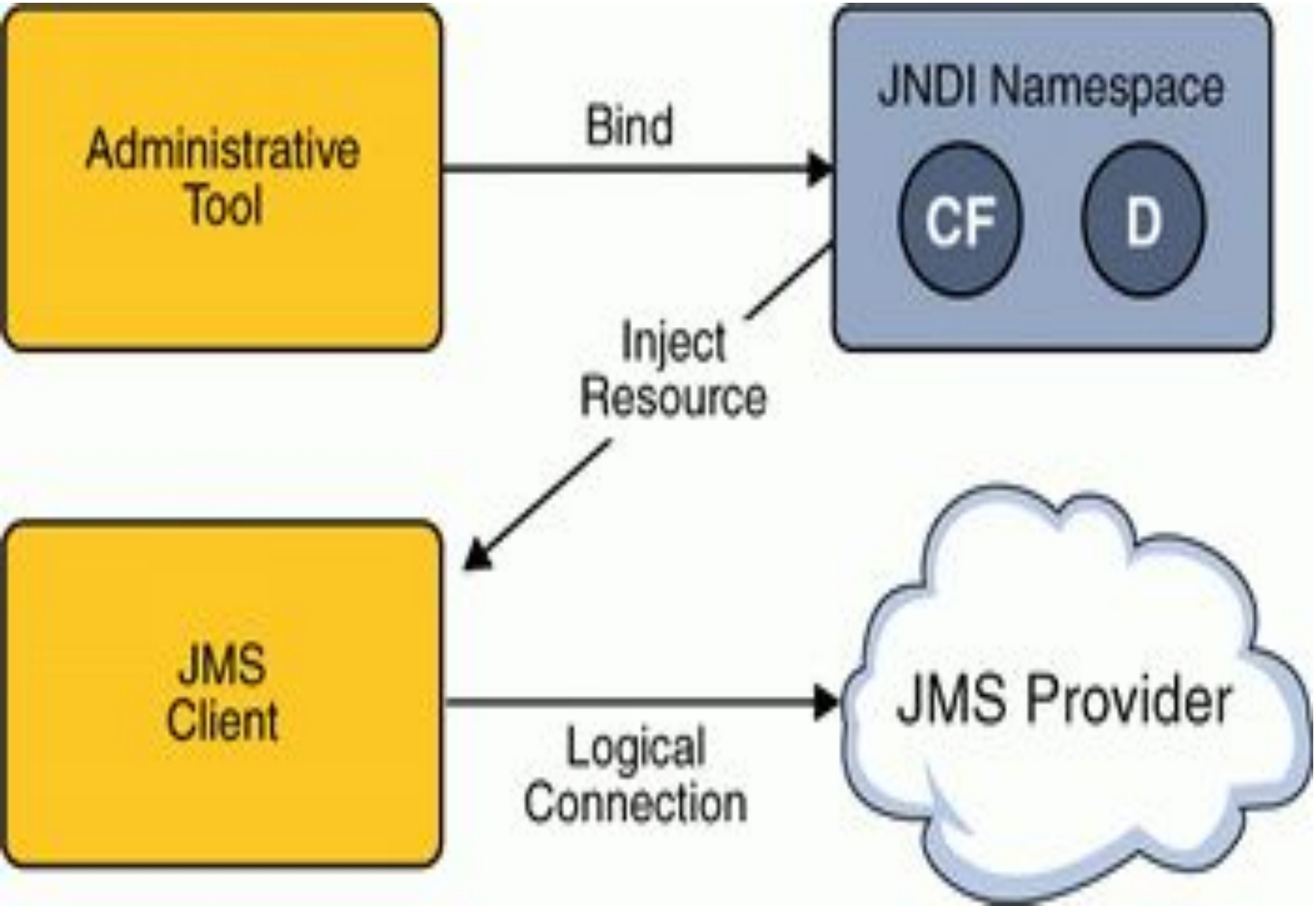
JMS (Java Messaging System) представляет собой интерфейс к внешним системам, ориентированный на работу через сообщения.

При разработке JMS в качестве основной задачи рассматривалось создание обобщенного Java API для приложений, ориентированных на работу с сообщениями (message-oriented application programming), и обеспечение независимости от конкретных реализаций соответствующих служб обработки сообщений.

Модель обмена сообщениями (и JMS) удобно использовать в том случае, если распределенное приложение обладает следующими характеристиками:

- взаимодействие между компонентами является асинхронным;
- информация (сообщение) должна передаваться нескольким или даже всем компонентам системы (семантика передачи от одного ко многим);
- передаваемая информация используется многими внешними системами, часть из которых неизвестна на момент проектирования системы или интерфейсы которых подвержены частым изменениям (концепция ESB - Enterprise Service Bus);
- обменивающиеся информацией (сообщениями) компоненты выполняются в разное время, что требует наличия посредника для промежуточного хранения переданной информации.

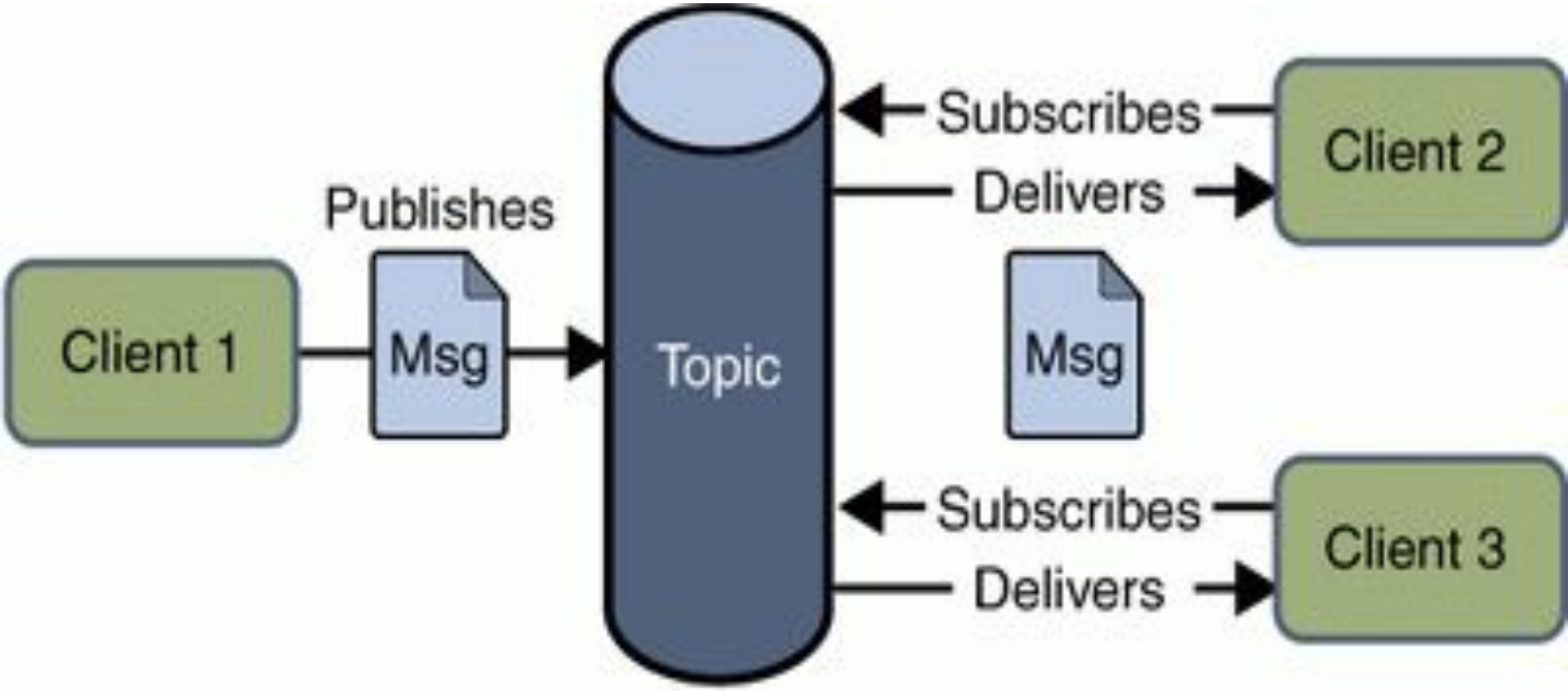
Архитектура JMS выглядит следующим образом



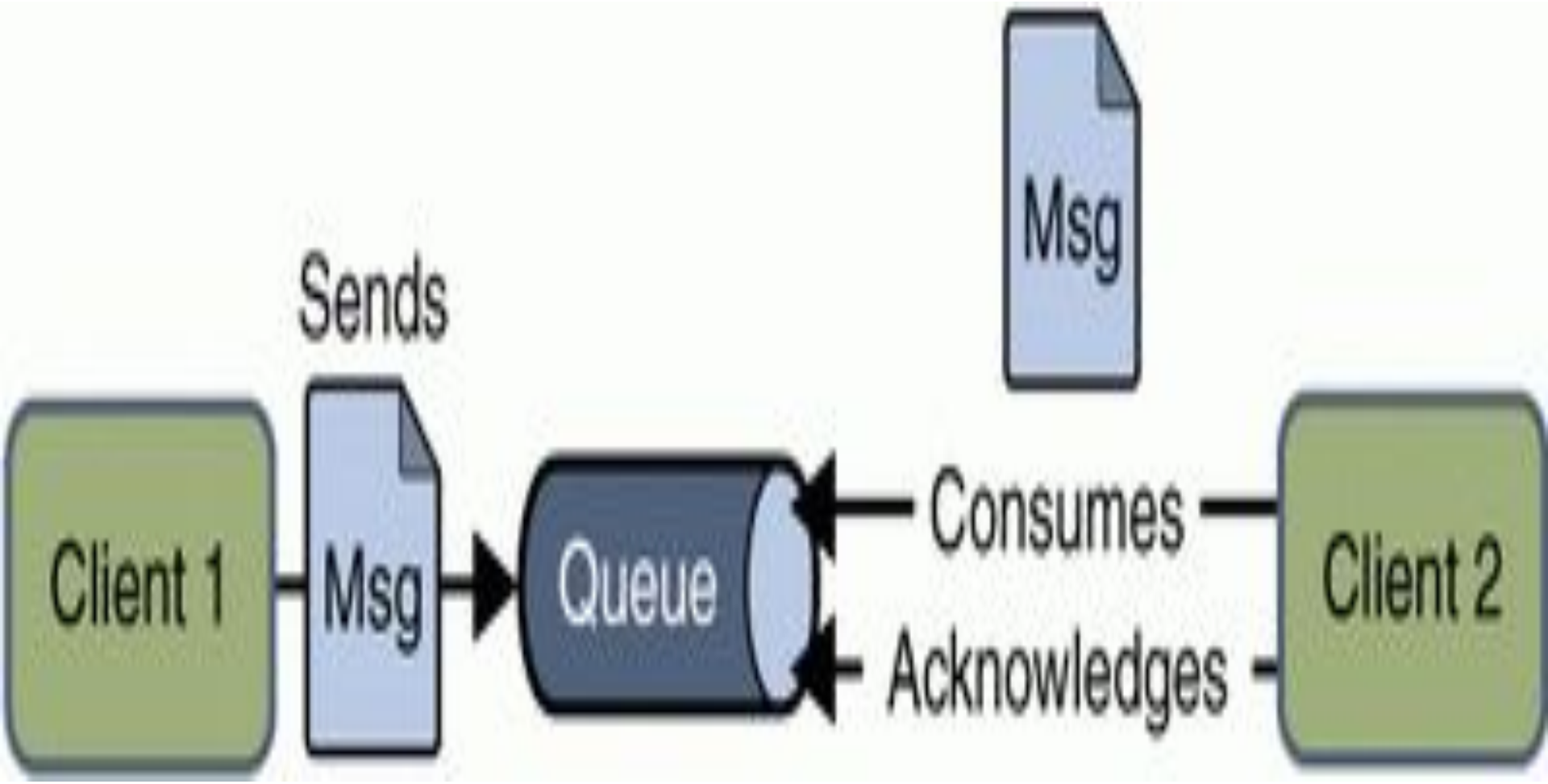
- прикладные программы Java, использующие JMS, называются клиентами JMS (JMS client);
- система обработки сообщений, управляющая маршрутизацией и доставкой сообщений, называется JMS-провайдером (JMS provider);
- приложение JMS (JMS application) - это прикладная система, состоящая из нескольких JMS клиентов, и, как правило, одного JMS -провайдера. JMS -клиент, посылающий сообщение, называется поставщиком (producer). JMS -клиент, принимающий сообщение, называется потребителем (consumer). Один и тот же JMS - клиент может быть одновременно и поставщиком, и потребителем в разных актах взаимодействия;
- сообщения (Messages) - это объекты, передающиеся и принимающиеся компонентами (клиентами JMS);
- средства администрирования (Administrative tools) - средства управления ресурсами, использующимися клиентами.

JMS предоставляет два подхода к передаче сообщений.

Первый называется "издание-подписка" (publish an subscribe) Этот подход используется в том случае, если сообщение, отправленное одним клиентом, должно быть получено несколькими.



Второй подход называется "точка-точка" (point to point) и служит для реализации обмена сообщениями между двумя компонентами.



Модель передачи сообщений "точка-точка" предоставляет возможность клиентам JMS посылать и принимать сообщения (как синхронно, так и асинхронно) через виртуальные каналы, называемые очередями (queues).

Модель основывается на методе опроса, при котором сообщения явно запрашиваются (считываются) клиентом из очереди. Несмотря на то, что чтение из очереди могут осуществлять несколько клиентов, **каждое сообщение будет прочитано только единожды - провайдер JMS это гарантирует**.

Модель взаимодействия "издание-подписка"

При использовании модели взаимодействия "издание-подписка" один клиент (поставщик) может посылать сообщения многим клиентам (потребителям) через виртуальный канал, называемый темой (topic).

Потребители могут выбрать подписку (subscribe) на любую тему.

Все сообщения, направляемые в тему, передаются всем потребителям данной темы.

Каждый потребитель принимает копию каждого сообщения.

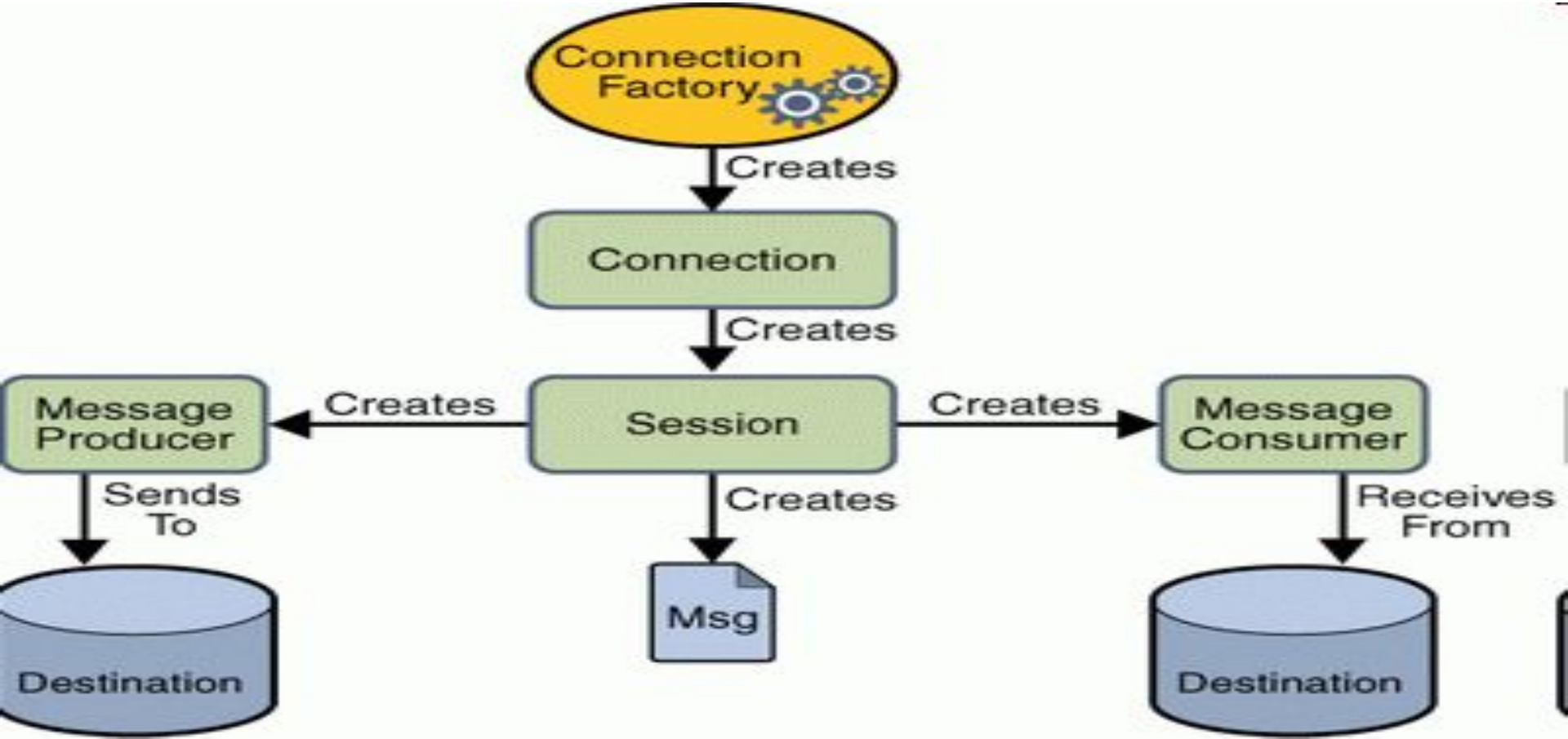
Модель передачи сообщений "издание-подписка", по существу, представляет собой модель сервера, инициирующего соединение и "проталкивающего" информацию на клиента.

В JMS эта концепция реализуется с помощью специальных "слушателей", регистрируемых в системе.

При возникновении нового события слушатель, закрепленный за данной темой, возбуждается.

Использование JMS технологии.

Любой компонент, использующий JMS, прежде всего должен создать соединение с JMS -провайдером - собственно системой, обеспечивающей всю функциональности управления сообщениями. Последовательность действий изображена на схеме



Рассмотрим модель взаимодействия “точка-точка” на базе JMS провайдера ActiveMQ.

```
import javax.jms.*;
import org.apache.activemq.ActiveMQConnection;
import
   org.apache.activemq.ActiveMQConnectionFactory;
public class Producer {
//Задаем URL JMS сервера, в данном случае
   подразумевается, что сервер висит на
   tcp://localhost:61616
    private static String url =
        ActiveMQConnection.DEFAULT_BROKER_URL;
//имя очереди сообщений
   private static String subject = "TESTQUEUE";
```

```
public static void main(String[] args) throws
                                        JMSException {
//создаем JMS соединение с сервером
ConnectionFactory connectionFactory =
                    new ActiveMQConnectionFactory(url);
Connection connection =
                    connectionFactory.createConnection();
connection.start();
//Для посылки и приема JMS сообщений необходимо
   создать сессию. True следует использовать в
   случае транзакций.
Session session = connection.createSession(false,
                              Session.AUTO_ACKNOWLEDGE);
```

```
//задаем имя очереди
Destination destination =
                    session.createQueue(subject);
//Для отправки сообщения используется
   MessageProducer
MessageProducer producer =
            session.createProducer(destination);
// Создаем обертку для сообщения
TextMessage message =
            session.createTextMessage(“Hello”);
```

```
//отправляем сообщение
producer.send(message);
//закрываем соединение
connection.close();
 }
}
```

Рассмотрим процедуру получения сообщения

```
public class Consumer {
ConnectionFactory connectionFactory = new
                    ActiveMQConnectionFactory(url);
Connection connection =
            connectionFactory.createConnection();
connection.start();
```

```
session session =
  connection.createSession(false,
                  Session.AUTO_ACKNOWLEDGE);
Destination destination =
                        session.createQueue(subject);
//Создаем объект MessageConsumer для
   приема сообщений
MessageConsumer consumer =
            session.createConsumer(destination);
//Прием сообщения
Message message = consumer.receive();
```

```
if (message instanceof TextMessage) {
   TextMessage textMessage = (TextMessage)
                                        message;
System.out.println("Received message "+
                              textMessage.getText() );
   }
connection.close();
} }
```

JMS API определяет несколько типов сообщений:

- **BytesMessage** предназначен для передачи потока байт, который система никак не интерпретирует;
- **MapMessage** предназначен для передачи множества элементов типа "имя-значение", где имена являются объектами строкового типа, а значения - объектами примитивных типов данных Java ;
- **ObjectMessage** предназначен для передачи сериализуемых объектов;
- **StreamMessage** предназначен для передачи множества элементов примитивных типов данных Java (они могут быть последовательно записаны, а затем прочитаны из тела сообщения этого типа);
- **TextMessage** предназначен для передачи текстовой информации.

Рассмотрим другой пример использования интерфейса MessageListener.

```
public class Myclass implements
                                    MessageListener{
 public static void main(String[] args) throws
                                    JMSException{
ConnectionFactory connectionFactory=
              new ActiveMQConnectionFactory(
                          "tcp://localhost:61616");
Connection connection =
        connectionFactory.createConnection();
connection.start();
```

```
Session session =
                    connection.createSession(false,
                  Session.AUTO_ACKNOWLEDGE);
Destination destination =
      session.createQueue(“Queue1”);
MessageConsumer consumer =
            session.createConsumer(destination);
//Привязываем слушатель к получателю
Myclass m=new Myclass();
consumer.setMessageListener(m);
}
```

```
//Данный метод вызывается, когда в очереди
   Queue1 появляется сообщение
public void onMessage(Message message){
        String messagerecv;
        try{
            if (message instanceof TextMessage) {
            TextMessage textMessage =
                                (TextMessage) message;
             messagerecv = textMessage.getText();
             System.out.println(messagerecv);
       } catch(JMSException e) {}
}
}
```

Рассмотрим модель взаимодействия "издание-подписка“.
Прежде всего рассмотрим класс издателя:

```
public class Publisher {


public static void main(String[] args) {
      try{
//Данные свойства можно также вынести в отдельный файл jndi.properties
       Properties props = new Properties();
       props.setProperty(
                              Context.INITIAL_CONTEXT_FACTORY,
  "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
       props.setProperty(Context.PROVIDER_URL,
                                                  "tcp://localhost:61616");
      Context ctx = new InitialContext(props);
```

```
//Создаем соединение с JMS сервером
TopicConnectionFactory factory =
   (TopicConnectionFactory)ctx.lookup(
                                        "ConnectionFactory");
TopicConnection conn =
                        factory.createTopicConnection();
conn.start();
//Создаем сессию и тему для подписки
TopicSession session =
   conn.createTopicSession(false,
                    TopicSession.AUTO_ACKNOWLEDGE);
Topic mytopic=session.createTopic("MyTopic");
```

```
//Создаем издателя
TopicPublisher topicPublisher =
                    session.createPublisher(mytopic);
//константа NON_PERSISTENT означает,
   что сообщения не должны логироваться
topicPublisher.setDeliveryMode(
                    DeliveryMode.NON_PERSISTENT);
//Создаем объект TextMessage
TextMessage message =
                        session.createTextMessage();
message.setText("Hello World");
```

```
//Записываем сообщение в тему
topicPublisher.publish(message);
session.close();
conn.close();
} catch(NamingException e){
            e.printStackTrace();
} catch(JMSException e){
           e.printStackTrace();
 } }
```

Рассмотрим класс подписчика

```
import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class Subscriber implements
            MessageListener,ExceptionListener {
public static void main(String[] args) throws
                JMSException,NamingException{
```

//свойства можно вынести в файл jndi.properties, путь к файлу следует указать в переменной окружения CLASSPATH

```
Properties props = new Properties();
props.setProperty(
        Context.INITIAL_CONTEXT_FACTORY,
        "org.apache.activemq.jndi.
                ActiveMQInitialContextFactory");
props.setProperty(Context.PROVIDER_URL,
                        "tcp://localhost:61616");
props.setProperty("topic.MyTopic","MyTopic");
Context ctx = new InitialContext(props);
```

```
//Создаем объект - слушатель
Subscriber asyncSubscriber =
                                    new Subscriber();
//Создаем соединение
TopicConnectionFactory factory =
  (TopicConnectionFactory)ctx.lookup(
                                    "ConnectionFactory");
TopicConnection conn =
                    factory.createTopicConnection();
//Добавляем обработчика исключений
conn.setExceptionListener(asyncSubscriber);
conn.start();
```

```
//Ищем тему “MyTopic”
Topic mytopic =  (Topic)ctx.lookup("MyTopic");
//Создаем сессию
TopicSession session =
conn.createTopicSession(false,
                    TopicSession.AUTO_ACKNOWLEDGE);
//Создаем подписчика и добавляем слушателя
TopicSubscriber topicSubscriber =
                          session.createSubscriber(mytopic);
topicSubscriber.setMessageListener(
                                                asyncSubscriber);
}
```

```
//Метод вызывается при появлении сообщения в теме.
public void onMessage(Message message){
    TextMessage msg = (TextMessage)message;
    try {
        System.out.println("received: " +
                                        msg.getText());
    } catch (JMSException ex) {
        ex.printStackTrace();
    }
}
```

```
// Данный метод вызывается при появлении
   исключения
public void onException(
                              JMSException exception){
 System.err.println("something bad
                             happended: " + exception);
}
}
```