Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Н. Н. Цаценко

ТЕХНОЛОГИИ РАБОТЫ С ДАННЫМИ В ИНТЕРОПЕРАБЕЛЬНЫХ СРЕДАХ CORBA, RMI, EJB

Препринт
93

Новосибирск 2002

Работа представляет собой обзор двух объектно-ориентированных технологий создания современных распределенных систем — CORBA и RMI.

Архитектура RMI (Remote Method Invocation, т.е. вызов удаленного метода), которая интегрирована с JDK, является продуктом компании JavaSoft и реализует распределенную модель вычислений. RMI позволяет клиентским и серверным приложениям через сеть вызывать методы клиентов/серверов, выполняющихся в Java Virtual Machine, и также предоставляет возможность пересылать сами объекты от машине к машине.

Технология CORBA (Common Object Request Broker Architecture), разрабатываемая OMG (Object Managment Group) с 1990-го года, позволяет вызывать методы у объектов, находящихся в сети где угодно, так, как если бы все они были локальными объектами. Одним из основных принципов CORBA является интероперабельность, возможность использования в одной системе различных приложений, реализованных на разных языках и выполняющихся на разных платформах.

Целью данной работы является показать масштабность и преимущества технологии CORBA по сравнению с RMI.

Кроме того, так как еще нет реализации компонентной модели CORBA, в работе рассматривается технология создания серверных объектов EJB (Enterprise JavaBeans), предложенная фирмой Sun. В компонентной модели EJB используются сильные стороны как CORBA, так и RMI.

В приложении приводятся с подробными комментариями два примера применения CORBA и EJB.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

N. N. Tsatsenko

TECHNOLOGIES OF WORK WITH DATA IN INTEROPERABLE SYSTEMS CORBA, RMI, EJB

Preprint
93

Novosibirsk 2002

Work represents the review of two object-oriented technologies of creation of the modern distributed systems — CORBA and RMI.

Architecture RMI (Remote Method Invocation) which is integrated with JDK, is a product of company JavaSoft and realizes the distributed model of calculations. RMI allows client's and server's applications through a network to cause methods of the clients / servers which are carried out in Java Virtual Machine, and also gives an opportunity to send objects from the machine to the machine.

Technology CORBA (Common Object Request Broker Architecture), developed OMG (Object Managment Group) since 1990 year, allows to cause methods in the objects which are taking place in a network anywhere how if all of them were local objects. One of main principles CORBA is interoperability, it means the opportunity of use in one system of the various applications realized in different languages and carried out on different platforms.

The purpose of the given work is to show scale and advantages of technology CORBA in comparison with RMI. Besides as still there is no realization of componental model CORBA, in work the technology of creation of server objects EJB (Enterpise JavaBeans) considered and supported by Sun. In componental model EJB strengths both CORBA, and RMI are used.

In the appendix two examples of application CORBA and EJB are resulted with detailed comments.

ВВЕДЕНИЕ

В данной работе содержится обзор объектно-ориентированных технологий создания современных распределенных систем. Под распределенными системами понимают программные комплексы, составные части которых функционируют на разных компьютерах в сети. Эти части взаимодействуют друг с другом, используя ту или иную технологию различного уровня — от непосредственного использования сокетов TCP/IP до технологий с высоким уровнем абстракции. Будут рассмотрены следующие задачи, решаемые с помощью распределенных вычислений.

1. Выполнение методов объектов, находящихся на удаленных машинах так, как будто эти объекты созданы и находятся в локальном доступе. Одно из решений этой задачи было создано и поддерживается до сих пор фирмой JavaSoft: механизм удаленного вызова методов Java (RMI, Remote Method Invocation).
2. Использование различных приложений, реализованных на разных языках и выполняющихся на других платформах. В том числе для решения этой задачи консорциумом OMG (Object Management Group) разрабатывается архитектура брокера объектных запросов (CORBA, Common Object Request Broker Architecture).
3. Изолирование кода бизнес-логики от кода соединения и поддержки связи с базами данных, сюда включаются управление транзакциями и безопасность. Решение было предложено фирмой Sun в виде технологии создания серверных объектов EJB (Enterprise JavaBeans). Кроме того, именно в компонентной модели EJB используются сильные стороны и CORBA, и RMI.

CORBA

Технология CORBA создается консорциумом OMG, как универсальная технология создания распределенных систем в интероперабельных средах.

Применительно к CORBA, универсальным является решение, которое не зависит от

- языка программирования,
- аппаратной платформы,
- операционной системы,
- сетевого протокола передачи данных,
- двоичных стандартов и структур.

В технологиях и спецификациях CORBA OMG реализует свое видение концепций объектно-ориентированного программирования. Стандарт CORBA 2.0 образует структуру объектной системы, так называемую ссылочную модель (Reference Model), и содержит следующие компоненты.

Object Request Broker (Брокер Запросов Объекта) — позволяет объектам строить и отправлять запросы и ответы в распределенной системе. Является основанием для приложений с распределенными объектами и для интероперабельности между приложениями в одно- и разнородной средах. Архитектура и спецификации ORB [1, 2] описываются в CORBA**.**

Object Services (Сервисы Объекта) — набор сервисов (интерфейсов и объектов), которые поддерживают основные функции использования и применения объектов. Сервисы являются необходимой частью при конструировании распределенных приложений и всегда являются независимыми от приложений. Например, Life Cycle Service (сервис жизненного цикла) определяет конструкции для создания, удаления, копирования и перемещения объектов. Но не определяется, как именно объекты реализуются. Информация по сервисам объектов рассматривается в спецификациях CORBAservices.

Common Facilities (Общие Свойства) — набор сервисов, которые многие приложения могут использовать совместно, но которые не являются такими фундаментальными, как Object Services**.** Например, сервис электронной почты может рассматриваться как common facility. Информация по общим свойствам рассматривается в спецификациях CORBAfacilities**.**

Application Objects (Объекты приложений) — являются продуктами индивидуального разработчика, реализующие собственные интерфейсы. Application Objects соответствуют обычным приложениям, поэтому не стандартизуются OMG. Они являются самым верхним уровнем ссылочной модели.

Спецификации CORBA создаются на специальном языке — OMG IDL (Interface Definition Language). Универсальность технологии главным образом обеспечивается базированием исключительно на IDL и использованием утвержденных OMG стандартов отображения IDL-объявлений на конкретные языки программирования. В связи с отсутствием пока реализации собственной компонентной модели CORBA — Corba Beans, разработка которой ведется [3], очевидное преимущество имеют реализации технологий CORBA на Java, в силу универсальности Java и переносимости его кода на

другие платформы. Ярким примером является компонентная модель EJB, рассматриваемая в разд. 3.

Язык определения интерфейсов, IDL (Interface Definition Language)

Использование архитектуры CORBA реализуется с помощью описаний методов, которые доступны в объекте, на языке определения интерфейсов IDL. IDL является языком описания, то есть он предназначен для объявления типов данных, переменных, констант, исключений, структур и т.д., но не для манипулирования ими. Дальнейшее использование IDL-деклараций происходит следующим образом: специальный компилятор по этим IDL-декларациям генерирует код на определенном языке программирования, разделенный на серверную и клиентскую части. Код, выполняемый на сервере, называется скелетом (skeleton) и определяет структуру серверных объектов. Скелеты являются основой, при использовании которой пишется код реализации всех методов, описанных в IDL-декларациях. На стороне клиента образуется стаб (stub). Стаб содержит полностью готовый к выполнению код, который превращает вызовы клиентом локальных методов стаба в вызовы соответствующих им удаленных методов реальных серверных объектов.

В настоящий момент стандартизовано отображение языка IDL на шесть языков программирования — Ada, C, C++, Cobol, Java и Smalltalk.

За основу IDL взят язык C++. Следующая таблица показывает соответствие основных концепций языков JAVA и C++ с языком IDL.

IDL	***Java***	***C++***
Module	Package	Namespace
Interface	Public Interface	Pure abstract class
Method	Method	Member function

В модуле IDL описываются предоставляемые объектам услуги, реализации которых производятся потом на любом языке, поддерживаемом CORBA.

По таблице видно, что модуль, также как и package в Java и namespace и C++, определяет собственное пространство имен.

Приведем пример модуля:

```
module My {

   //тело модуля

};
```

В глобальном пространстве имен и в теле любого модуля могут находиться следующие конструкции:

- модули,
- объявления типов (синонимы типов (typedef), перечисления, структуры, объединения, native),
- исключения,
- константы,
- интерфейсы,
- типы-значения (valuetype).

Дальнейшее изложение будет посвящено последним двум конструкциям, интерфейсам и типам-значениям. Полное описание всех конструкций языка IDL можно найти в спецификации [4] и в руководстве [5, с. 81–132].

Взаимодействие с удаленным объектом в CORBA возможно двумя способами.

Первый способ — получение объектной ссылки на него. Объектную ссылку проще всего трактовать, как обобщение понятия указателя в C++ или ссылки в Java. Для того чтобы определить объект, достаточно определить интерфейс. Создавая таким образом CORBA-объект, одновременно создается и объектная ссылка на него. При вызове удаленного метода объекта передается объектная ссылка, и говорят по аналогии с C++, что «объект передается по ссылке».

Другой способ взаимодействия с удаленным объектом использует «передачу по значению». Суть заключается в следующем: при вызове удаленных методов в качестве аргумента передается не объектная ссылка (т.е. указатель на объект), а копия состояния этого объекта. В общем случае подразумевается, что клиент знает все о методах объекта, состояние которого он получил. Это достигается использованием доступной для клиента, базы данных интерфейсов вместе со всеми их характеристиками.

Эти базы данных называются репозитариями интерфейсов (Interface Repository) [6]. Созданные таким способом объекты, могут продолжать существовать после завершения создавших их процессов. При реализации этого способа на Java возможна передача не только состояния объекта, но и при необходимости байт-кода его класса (подробно о сериализации см. [5]), что и было эффективно проделано в технологии Enterprise JavaBeans (см. разд. 3). Типы-значения (Valuetype) служат для представления вышеизложенного способа взаимодействия.

Интерфейсы

Интерфейсы определяют новое пространство имен, не могут быть вложенными, но могут быть связаны друг с другом отношениями наследования (как простого, так и множественного). Также нет возможности на уровне IDL задать интерфейс с состоянием, т.е. создать поля для интерфейсов, хотя CORBA-объект как правило имеет состояние. Вот пример интерфейса, реализующего большинство его возможностей:

```
interface MyInterface  {

    //Определение синонимов типов
    typedef string StringArray[10];
    typedef long ResultType;

    //Определение исключительных ситуаций
    exception Wrong {};

    //Определение констант
    const string  Header = "This is Header";

    //Определение атрибутов
    attribute ResultType myQuery;

    //Определение методов
    string myMethod (in StringArray args) raises (Wrong);
    void getTotal (out double i);
};
```

Опишем кратко синтаксис определений атрибутов и методов.

При определении методов для каждого аргумента в списке необходимо указывать его признак — является ли параметр входным (in), выходным (out) или и тем, и другим (inout). Это служит для явного задания того, кто — клиент или сервер — отвечает за выделение памяти под объекты. Входной параметр (in) передается от клиента серверу, за выделение и освобождение памяти для объекта отвечает клиент. Соответственно, выходной параметр (out) передается от сервера клиенту. В случае возникновения исключительной ситуации значение out и inout параметров не определено.

Использование атрибутов — просто краткий способ определения методов для данного интерфейса. Данное выше определение атрибута логически эквивалентно двум объявлениям методов:

```
ResultType set_myQuery ();
void get_myQuery (in ResultType x);
```

Недостатком является невозможность задания пользовательских исключений, системные же исключения возможны.

Типы-значения

Концепция тип-значение (valuetype) является заимствованной из технологии RMI. Объект, определенный с помощью такой IDL-декларации, является на самом деле не CORBA-объектом, а обычным объектом некоторого языка программирования, генерируемым соответствующим компилятором с IDL. Использование CORBA-оболочки для такого объекта служит для того, чтобы ORB помог универсальным образом отправить информацию об объекте и его состояние в другое адресное пространство. При получении на стороне получателя создается сначала пустая копия данного объекта, а затем устанавливается состояние этого нового объекта при использовании информации, переданной через ORB. После выполнения такой операции исходный объект и его копия становятся полностью независимыми друг от друга объектами. Этим и обуславливается термин «передача по значению». Обычно тип-значение используется для передачи копий больших и сложных структур данных: деревьев, хэш-таблиц, XML-документов [7].

Рассмотрим синтаксис задания типов-значений на примере.

```
valuetype MyValueType {
        //Поле состояния, доступное только для методов этого типа
        private long prField;
```

```
        //Открытое поле состояния
        public string pbField;

        //Инициализаторы (фабрики)
        factory myInit (in long arg);
        factory myInit1 (in long arg, in string s);

        //Локальные методы, которые и вызывает пользователь
        long getValue();
};
```

Инициализаторы (initializers) являются аналогами конструкторов C++ или Java, в них они и отображаются компилятором. Все их аргументы должны иметь входной тип (in).

CORBA не обеспечивает в отличие от C++ конструктора по умолчанию. Это означает, что при отсутствии явных инициализаторов (конструкторов, фабрик) невозможно создать экземпляр такого типа-значения при выполнении приложения.

Типы-значения такого вида, т.е. те, которые имеют состояние, называются объектами с состоянием (stateful). Существуют также абстрактные типы-значения (abstract), которые не содержат ни полей состояния, ни инициализаторов.

```
abstract valuetype MyAbstructValueType {
        long getValue(in long x);
};
```

Их единственная задача — служить базой для построения реальных типов-значений путем наследования.

Брокер объектных запросов, ORB (Object Request Broker)

Посредник запросов к объектам (ORB) — это шина связи, посредством которой объекты могут запрашивать услуги других объектов, вне зависимости от их физического расположения. ORB является многофункциональным компонентом, т.е. выполняет следующие операции:

- отвечает за передачу данных между объектами;

- выполняет распространение контекстов вызовов, транзакций и безопасности;
- обеспечивает стандартный доступ ко многим сервисам (к примеру, resolve_initial_references).

Существует несколько реализаций ORB от разных производителей (JavaIDL, VisiBroker, OrbixWeb и др.). Стандарт CORBA определил следующие основные компоненты технологии, которые позволяют обеспечить совместное использование различных ORB в одной распределенной системе:

- пересылка информации между объектами с использованием низкоуровневого протокола взаимодействия ORB (General Inter-ORB Protocol**,** GIOP) [8];
- интернет-протокол взаимодействия — реализация GIOP на базе протокола TCP/IP (Internet Inter-ORB Protocol, IIOP);
- специфические протоколы взаимодействия с CORBA (точнее с IIOP) в различных средах (Environment-Specific Inter-ORB Protocols, ESIOP);
- общий формат представления данных всех IDL-типов;
- универсальный формат представления объектных ссылок (Interoperable Object Reference, IOR).

Рассмотрим основные методы работы с ORB.

Для установки взаимодействия объектов первое, что необходимо сделать, — это выполнить инициализацию ORB. Для этого существует метод ORB_init():

```
module CORBA {
    typedef string ORBid;
    typedef sequence <string> arg_list;
    ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
}
```

Первый аргумент служит для передачи параметров из командной строки в виде опций, таких как IP-адрес хоста, номер порта и т.д.

Второй аргумент позволяет идентифицировать конкретный ORB в случае, когда используются несколько ORB.

На Java проинициализировать ORB можно следующим образом:

```
orb.omg.CORBA.ORb orb = org.omg.CORBA.ORB.init (args, null);
```

Здесь args — аргумент функции main().

Объектная ссылка на уровне ORB представляет собой совокупность некоторой информации, которая позволяет однозначно идентифицировать конкретный CORBA-объект. Работать с объектными ссылками, приведенными ORB к универсальному формату IOR, можно преобразовывая их из и в строки:

```
string object_to_string (in Object obj);
Object string_to_object (in string str);
```

Стандартный механизм доступа к базовым CORBA-сервисам (см. следующий раздел) предоставляет метод

```
Object resolve_initial_references (in ObjectId identifier) raises (InvalidName);
```

На Java разрешение ссылки на сервер службы идентификации выглядит следующим образом:

```
orb.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
```

Более подробную информацию по ORB можно получить в спецификациях [1, 2].

Сервисы CORBA

В предыдущем разделе были рассмотрены основные концепции ORB, предлагающие решение задачи взаимодействия клиента с сервером. Теперь рассмотрим другие задачи и их решения, возложенные на элементы архитектуры CORBA, а именно: задачи поиска, отношений между объектами, сохранения их состояний, управления транзакциями и безопасностью и другие. Этими элементами являются Сервисы CORBA (Object Services). Спецификация сервисов состоит из множества интерфейсов, описывающих поведение сервисов на языке IDL.

Ниже приведено очень краткое описание 15 стандартизованных на данный момент сервисов CORBA.

Naming Service (Сервис Именования) — предоставляет возможность связывания имени с объектом относительно именного контекста. Именной контекст является объектом, содержащим множество уникальных именованных связей. Доступ к объекту производиться механизмом разрешения имени — это определение объекта по ассоциированному с ним имени в данном именном контексте.

Event Service (Сервис Событий) — позволяет универсальным образом уведомлять объекты распределенной системы о происходящих событиях. Служит для организации callback-вызовов. В настоящее время этот сервис является лишь базой для более развитого и обобщенного сервиса уведомлений (Notification Service).

Persistent Service (Сервис Долговременного Хранения) — обеспечивает универсальный механизм сохранения состояния CORBA-объектов как в реляционных, так и в объектных базах данных.

Life Cycle Service (Сервис Цикла Жизни) — имеет существенные отличия от всех других сервисов, так как отвечает за операции создания, копирования, перемещения, удаления объектов. Является набором интерфейсов, многие методы которых необходимо реализовывать самостоятельно. Находиться не изолированно от других, а в тесной связи с сервисом отношений (Relationship Service).

Relationship Service (Сервис Отношений) — позволяет динамически устанавливать связи между объектами. Совокупности отношений образуют графы и рассматриваются как CORBA-объекты. Используется при копировании, перемещении или удалении группы связанных друг с другом объектов.

Property Service (Сервис Свойств) — позволяет сопоставлять с объектами те или иные свойства в виде пары «имя—значение».

Security Service (Сервис Безопасности) — решает многие стандартные проблемы безопасности: идентификация пользователя, определение прав доступа, защита информации при передаче и т.д. Распространение контекста безопасности осуществляется ORB.

Trading Service (Трейдер-Сервис) — сервис поиска, с помощью которого клиент получает нужную объектную ссылку. Поиск осуществляется не по имени, а по совокупности свойств объекта.

Externalization Service (Сервис Внешнего Представления) — предназначен для построения образа объекта, взаимодействующего со стандартными потоками ввода-вывода (эквивалентный механизм — сериализация в JAVA).

Transaction Service (Сервис Транзакций) — взаимодействует на уровне реализации с ORB и служит для управления транзакциями. Поддерживает двухфазное подтверждение транзакций и вложенные транзакции.

Concurrency Control Service (Сервис Совместного доступа) — предназначен для универсального управления разделяемыми ресурсами. Осуществляет координацию параллельных транзакций.

Query Service (Сервис Запросов) — предназначен для выполнения поиска объектов, соответствующих определенным критериям. Языком выполнения является либо Object Query Language (OQL), либо расширенный SQL. Результатом является совокупность объектов, для манипуляция с которыми используется сервис контейнеров.

Collections Service (Сервис Контейнеров) — позволяет определять группы объектов и управлять ими единообразным способом. Используются стандартные контейнеры — множества (set), наборы (baq), последовательности (sequence) и др. Для каждого контейнера определена своя фабрика (интерфейс для создания контейнера) и итератор.

Licensing Service (Сервис Лицензирования) — служит для отслеживания использования CORBA-объекта и для ограничения его применения по разным критериям.

Time Service (Сервис Времени) — служит для синхронизации часов на различных компьютерах. Основой является использование UTC(Univeral Time Coordinated)-времени.

Многие сервисы очень близки к ORB и обычно в приложениях используются вместе, например, Сервис Именования (Naming Service), Сервис Транзакций (Transaction Service), Сервис Цикла Жизни (Life Cycle Service). Далее рассматривается один из самых важных сервисов — Сервис Именования (Naming Service).

Сервис Именования (Naming Service)

Сервис именования [9] — один из основных сервисов, служит для сопоставления с объектными ссылками различных имен и организации структуры, которая позволяет группировать эти имена. Этот сервис присваивает строковое имя объектной ссылке, с помощью которой осуществляется доступ к CORBA-объекту. Имя в Naming Service представляет собой последовательность IDL-структур, содержащих два строковых поля:

```
module CosNaming {
        typedef string Istring;

        struct NameComponent {
                Istring id;
                Istring kind;
        };
};
typedef sequence <NameComponent> Name;
```

Задание синонима для типа string (Istring) предназначено для возможности в дальнейшем изменить представление типа полей id и kind в структуре. Наличие второго поля (kind) позволяет задавать различную дополнительную информацию собственно имени (id). Ассоциация имени с объектом называется связыванием объекта (naming binding), и под связанным именем понимается связка «имя_сервиса_именования—объектная_ссылка_CORBA». Связывание объекта всегда определено относительно именного контекста. Именной контекст является объектом, которой содержит множество уникальных именованных связок (примерно так, как каталог файловой системы может содержать имена файлов и имена других каталогов). Таким образом, структура имен Naming Service может быть представлена в виде произвольного направленного графа.

Так как контекст является сам по себе объектом, то он также может быть связан с именем в именном контексте. Связи контекстов в другом общем контексте создают именной граф (naming graph) — это ориентированный граф, узлы которого являются контекстами. Именованный граф позволяет давать более сложные имена при связывании объекта. Последовательность имен (compound name) определяет путь в именном графе для навигации разрешающего процесса. Как правило, контекст, которому не сопоставлено никакого имени, используется как «корень» при перемещении по

графу имен и также является вспомогательной основой, с помощью которой строится граф имен (подробнее об этом ниже).

Разрешением имени называется определение объекта по ассоциированному с ним имени в данном контексте. Так как имя в общем случае является составным, то процесс разрешения имени происходит по шагам аналогично тому, как при работе с Windows получается доступ к файлу при использовании его полного имени (путь).

Основной интерфейс Naming Service, NamingContext, используется для создания иерархии контекстов и разрешения имен. Рассмотрим несколько главных операций, опуская код возбуждаемых исключений.

```
module CosNaming
   {
    interface NamingContext {
   ...
   //Операция добавления в контекст уже связанного
   //имени, составными частями которого являются
   //имя n и объектная ссылка obj.
   void bind (in Name n, in Object obj) raises(...);

   //Операция аналогичная bind(), плюс - если
   //контекст содержит уже такое связанное имя,
   //происходит замена старого связанного имени на
   //новое.
   void rebind (in Name n, in Object obj) raises(...);

   //Операция добавления контекста имен.
   void bind_context (in Name n, in NamingContext nc) raises(...);

   //Операция добавления контекста имен с
   //перезаписью.
   void rebind_context (in Name n, in NamingContext nc)
        raises(...);

   //Операция разрешения имени. Возвращает
   //объектную ссылку, сопоставленную с указанным
   //именем.
   Object resolve (in Name n) raises(...);
```

```
        //Удаление из контекста связанного имени.
        void unbind (in Name n) raises(...);

        //Создание нового безымянного контекста,
        //который будет включен в граф имен.
        NamingContext new_context();

      //Создание нового контекста по параметру n и
      //добавление в контекст, для которого был вызван
      //этот метод.
      NamingContext bind_new_context(in Name n) raises(...);

      //Уничтожение контекста, не содержащего
      //связанных имен.
      void destroy () raises (...);

      //Операция получения списка связанных имен и
      //итератора для контекста.
      void list (in unsigned long how_many, out BindingList bl,
            out BindingIterator bi);
   };
};
```

Фабрикой (инициализатором) контекстов имен являются только ранее созданные контексты, и использование методов new_context() и bind_new_context() возможно только после того, как получена ссылка на ранее созданный контекст. Для создания начального контекста имен служит специальная утилита, входящая в компонент реализации. Для VisiBroker это утилита nameserv, которую необходимо запускать из командной строки до использования приложением Naming Service.

В JavaIDL сервис именования реализован в виде Java-приложения (tnameserv), выполняется виртуальной машиной JVM и по умолчанию прослушивает сетевой порт 900.

Для удаления контекста необходимо сначала удалить все связанные имена из контекста (unbind), затем вызвать метод destroy() для него и, наконец, вызвать unbind() для всех контекстов, которые содержат связанные имена с объектными ссылками на только что уничтоженный контекст.

После того как запущен Naming Service, серверные и клиентские приложения могут использовать этот сервис для получения объектных ссылок на объекты. Для доступа к начальному контексту используется метод:

```
orb.omg.CORBA.Object r_obj =
obj.resolve_initial_references("NameService");
```

Когда сервер и клиент находятся на разных машинах, ORB может разрешить их строковые ссылки с помощью компонента, известного под названием репозитарий реализаций (Implementation Repository, ImplR) [5, с. 425–440]. Этот необязательный компонент CORBA представляет базу данных, используемую во время работы приложений.

Пример, иллюстрирующий некоторые основные возможности CORBA, можно найти в разделе приложений (пример 1).

УДАЛЕННЫЙ ВЫЗОВ МЕТОДОВ, RMI (Remote Method Invocation)

Технология RMI, требующая использования языка программирования Java, также оказала сильное влияние на технологию CORBA. Главная задача, которую призвана решать RMI, — обеспечение передачи сообщений от объекта, существующего в контексте одной виртуальной машины Java (JVM), к объекту, существующему в контексте другой. RMI использует два стандартных протокола обмена: собственный не универсальный протокол обмена RMP и стандартный для CORBA протокол IIOP. Поддержка IIOP позволяет RMI-приложениям получить доступ ко всем сервисам CORBA и возможность доступа к CORBA-серверам, написанным на любом языке, поддерживающим CORBA. В свою очередь, интеграция с RMI позволяет CORBA компенсировать отсутствие в настоящий момент собственной компонентной модели CORBA и, соответственно, универсального монитора транзакций.

В основу RMI заложены средства сериализации Java. Сериализация — это техника обработки объектов для последующей передачи всего объекта в потоках данных, которая позволяет записывать и считывать объекты целиком.

Последовательность создания приложений с использованием RMI следующая:

1. Определение удаленных интерфейсов достигается путем наследования интерфейсом стандартного интерфейса java.rmi.Remote.
2. Создание класса, реализующего данный интерфейс. Этот класс должен наследовать класс java.rmi.UnicastRemoteObject.

3. Генерация специального кода-«заглушки», называемого еще стабом, как для клиента, так и для сервера с помощью компилятора rmic.
4. Запуск службы имен на стороне сервера (rmiregistry).
5. Создание серверного приложения, которое использует классы стаба. Это приложение служит для создания серверных объектов и регистрации их в службе имен.
6. Создание клиентского приложения с реализованным в нем механизмом вызова удаленных методов с передачей им аргументов посредством сериализации объектов в Java.

Библиотека RMI большей частью использует интерфейсы, и с помощью них предусматривается наличие подсистем поиска объектов по именам (Naming Service), управления безопасностью, маршалинга. При создании удаленного объекта передается ссылка не на сам этот объект, а на интерфейс, по которому создается серверный стаб, который и проводит пересылку по сети.

Первый шаг в написании RMI-приложения — это создание интерфейса, каждый метод которого можно будет вызывать с удаленного компьютера клиента. Естественно интерфейс описывается на Java:

```
public interface myRemote extends java.rmi.Remote {
String talking (string message) throws java.rmi.RemoteException;
}
```

Класс, который реализует этот интерфейс, должен помимо реализаций всех методов, описанных в интерфейсе, создать и установить менеджер безопасности сервисов RMI

```
System.setSecurityManager(new RMISecurityManager());
```

и зарегистрировать по крайне мере один из созданных удаленных объектов в реестре удаленных объектов RMI

```
//создание удаленного объекта
myRemoteServer server = new myRemoteServer();
//регистрация
Naming.bind("myServer"), server);
```

После проведенной регистрации объекта «server» приложения на стороне клиента могут использовать имя «myServer» для получения доступа к

этому удаленному объекту. Объект существует пока запущен сервер реестра (rmiregistry). Клиентское приложение должно отыскать (возможно по URL) и получить от сервера удаленный интерфейс этого удаленного объекта

```
myRemote I = (myRemote)Naming.lookup("myServer");
//вызов удаленного метода
I.talking();
```

Введение в технологию RMI можно найти в [10, с. 281–312].

Технология, в которой нашли применение как RMI, так и CORBA, образовав множество, называемое RMI/IDL, является технологией от фирмы Sun, Enterprise JavaBeans (EJB).

ENTERPRISE JAVABEANS, EJB

Спецификация EJB составляет значительную часть платформы Java 2 Enterprise Edition (J2EE) и описывает модель компонентов серверной стороны, ориентированную на создание масштабируемых, устойчивых и надежных серверных приложений. EJB служит для использования ресурсов серверов, управления правами доступа, безопасностью, транзакциями и взаимодействием с базами данных различного типа. При программировании EJB-компонента программист получает возможность избавиться от написания низкоуровневого кода и полностью сфокусироваться на реализации бизнес-логики.

Составной частью EJB может быть только то, что соответствует стандартам как RMI, так и CORBA. Такое подмножество называется RMI/IDL.

Технология создания серверных объектов EJB базируется на Java и использует технологию RMI для организации удаленных вызовов между виртуальными машинами JVM. Протоколом взаимодействия в EJB является стандартный протокол CORBA IIOP. Схема управления транзакциями JTS (Java Transaction Service) — это реализованный на Java сервис транзакций CORBA (OTS, Object Transaction Service). Существует стандарт отображения EJB на CORBA, касающийся управления транзакциями, безопасностью и службы имен (Naming Service).

EJB используются интерфейсы, определяемые стандартной семантикой RMI. На базе этих Java-объявлений с помощью компилятора, разработанного OMG (rmi2idl), можно сгенерировать IDL-файл, который может быть использован CORBA-программистами для создания на любом поддержи-

вающем CORBA языке клиентских приложений, взаимодействующих с EJB как с серверными объектами CORBA. Т.е. любой компонент EJB является одновременно полноценным CORBA объектом.

Спецификация EJB (версия 1.1) определяет несколько ключевых понятий технологии.

Сервер EJB

Сервер EJB — это среда выполнения, в которой функционируют компоненты EJB. Задачей сервера является обеспечение доступа к системным сервисам, необходимым для работы компонентов. Важнейшим из сервисов является сервис распределенных транзакций JTS. Сервер не взаимодействует непосредственно с компонентами, а реализует взаимодействие через некий логический уровень управления — контейнер EJB.

Контейнер EJB

Контейнер взаимодействует с сервером, когда компонентам, находящимся под управлением контейнера, необходим доступ к системным ресурсам.

Контейнер обеспечивает

- управление циклом жизни компонента: его созданием, инициализацией, сохранением его состояния в базе данных, если это необходимо;
- поиск компонента;
- гарантию того, что вызов методов происходит в контексте нужной транзакции;
- базовый уровень обеспечения безопасности.

В EJB используется спецификация управления транзакциями CORBA, реализованная на Java (JTS). Управлять транзакциями — начинать транзакцию, завершать или откатывать транзакцию — может как компонент, так и контейнер.

Обычно сервер и контейнер поставляются в готовом виде, например, Inprise Application Server, WebLogic Application Server.

Компонент EJB

Компонент EJB — это класс Java, который и реализует всю необходимую функциональность. Компонент находится под управлением контейнера и состоит из нескольких частей — класс компонента, реализация неко-

торых интерфейсов и информационный файл. Все это пакуется вместе в специальный jar файл. Все объекты в реализациях компонента должны быть действительными в RMI/IDL.

Класс компонента — реализует интерфейс Enterprise Bean и обеспечивает реализацию бизнес-методов, которые выполняет компонент. Класс не реализует никакую авторизацию, многопоточность или код транзакции, реализации этих ролей берет на себя поставщик EJB.

При реализации EJB класса необходимо следовать нескольким требованиям (полный список требований содержится в спецификации Enterprise JavaBeans [11]), основными из которых являются следующие.

- Класс должен быть публичным (public).
- Класс должен реализовывать EJB интерфейс (либо javax.ejb.SessionBean, либо javax.ejb.EntityBean).
- Класс должен определять методы, которые напрямую связываются с методами внешнего интерфейса. Но класс не реализует удаленный интерфейс, он лишь отражает методы внешнего интерфейса и не обрабатывает java.rmi.RemoteException.
- В классе должен быть определен один или несколько методов ejbCreate() для инициализации компонента EJB.

Все запросы к компоненту EJB передаются контейнеру компонентов посредством внутреннего и удаленного интерфейсов.

Удаленный интерфейс. Клиент может обращаться к компонентам через специальный интерфейсный объект-посредник (EJBObject), определяемый контейнером EJB. EJBObject реализует удаленный интерфейс (Remote-интерфейс). Удаленный интерфейс играет ту же роль, что и IDL интерфейс в CORBA. При создании удаленного интерфейса необходимо следовать следующим принципам.

- Удаленный интерфейс должен быть публичным (public).
- Удаленный интерфейс должен расширять интерфейс javax.ejb.EJBObject.
- Каждый метод удаленного интерфейса должен декларировать java.rmi.RemoteException в предложении throws помимо всех исключений, специфичных для приложения.

Пример удаленного интерфейса:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface myInterface extends EJBObject {
    public long getStatus() throws RemoteException;
}
```

Внутренний интерфейс — используется для создания компонента и реализуется объектом HomeObject. Также клиент использует внутренний интерфейс для нахождения экземпляра данного EJB и создания нового экземпляра данного EJB. При создании внутреннего интерфейса необходимо следовать следующим принципам.

- Внутренний интерфейс должен быть публичным (public).
- Внутренний интерфейс должен расширять интерфейс javax.ejb.EJBHome.
- Каждый метод create внутреннего интерфейса должен декларировать java.rmi.RemoteException в преложении throws наряду с javax.ejb.CreateException.
- Возвращаемое значение метода create должно быть удаленным интерфейсом.

Метод finder используется только для компонентов с данными (см. ниже) и должен иметь в качестве возвращаемого значения удаленный интерфейс или java.util.Enumeration, или java.util.Collection. Стандартное соглашение об именах внутренних интерфейсов состоит в прибавлении слова «Home» в конец имени удаленного интерфейса.

Вот пример внутреннего интерфейса для компонента CustomerBean:

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
public interface myInterfaceHome extends EJBHome {
    public Customer create()
            throws CreateException, RemoteException;
    public Customer findByKey(Integer customerNumber)
            throws RemoteException, FinderException;
}
```

Описатель развертывания (deployment descriptor) — заключает в себе всю информацию о компоненте EJB и является XML файлом. Использование XML позволяет программисту легко менять атрибуты данного EJB. Конфигурационные атрибуты, определенные в описателе развертывания, включают

- имена внутреннего и внешнего интерфейса,
- имя для публикации в JNDI для внутреннего интерфейса,
- транзакционные атрибуты для каждого метода EJB,
- контрольный Список Доступа для авторизации,
- инструменты разработки компонентов EJB, которые могут автоматически генерировать описатель развертывания.

EJB-jar файл — это обычный java jar файл, который содержит класс EJB, внутренний и удаленный интерфейсы и описатель развертывания.

Далее рассмотрим типы компонент, которые обладают различными характеристиками и свойствами. Существует два типа компонентов — Сессионный компонент (Session Bean) и компонент с данными (Entity Bean). В каждом из них имеются свои разновидности.

Сессионный компонент (Session Bean) — используется для представления событий в потоке работы с клиентом, его создавшим. Такие компоненты представляют операции с постоянными данными, но не сами постоянные данные. Сессионный компонент может иметь состояние (Stateful Session Bean) и может не иметь (Stateless Session Bean). Все Сессионные компоненты должны реализовывать интерфейс javax**.**ejb**.**SessionBean. EJB Контейнер управляет жизнью Сессионного компонента.

Сессионный компонент без состояния (Stateless Session Bean) — не содержит никаких значимых состояний для клиента между вызовами методов, а вся информация о состоянии компонента должна храниться за пределами самого компонента. Пример реализации такого типа EJB подробно рассматривается в [12, с. 783–787].

Сессионный компонент с состоянием (Stateful Session Bean) — сохраняет состояние между вызовами. Каждый компонент взаимнооднозначно соответствует определенному клиенту и может содержать состояние в себе. EJB контейнер может управлять такими компонентами, используя активизацию (activation) и деактивизацию (passivation). При деактивизации контейнер временно удаляет компонент из памяти, помещая его состояние в объектную или реляционную базу данных. Обратный процесс — это активизация, размещение объекта в памяти с восстановлением

его предыдущего состояния. Естественно, компонент без состояния может быть просто уничтожен контейнером, а потом заново создан в нужный момент.

Если работа EJB контейнера нарушается, данные всех EJB Сессионных компонентов с состоянием могут быть потеряны.

Компоненты с данными (Entity Bean) — являются компонентами, представляющими постоянные данные и их поведение, и управляются EJB контейнером. Они могут быть разделены между многими клиентами, точно так же, как могут разделяться данные в базе данных. Существование компонента с данными определяется EJB контейнером, так что если работа EJB контейнера нарушается, то данные не теряются, но сам компонент будет доступен только тогда, когда будет доступен EJB контейнер. Есть два типа компонент с данными: существующие с управлением контейнером (Container-managed persistence beans) и существующие с управлением компонентами (Bean-Managed persistence beans).

Container Managed Persistence (CMP) — через указанные в описании развертывания спецификации EJB контейнер связывает атрибуты компонента с некоторым постоянным хранилищем, обычно это база данных. CMP снижает время разработки для EJB, так же как и значительно снижает число требуемого кода. Пример CMP Bean приводится в приложении.

Bean Managed Persistence (BMP) — BMP компоненты реализуются Поставщиком Enterprise Bean. Поставщик Enterprise Bean отвечает за реализацию логики, требуемой для создания новых EJB, изменения некоторых атрибутов EJB, удаление EJB и нахождение EJB в постоянном хранилище. Обычно для этого требуется написание JDBC кода для взаимодействия с базой данных или другим постоянным хранилищем. С помощью BMP разработчик полностью контролирует управление существования компонента. BMP также дает гибкость в тех местах, где реализация CMP не может быть использована. Например, если необходимо создать EJB, который включает в себя код, реализующий некие сервисы CORBA.

Более подробно о спецификации EJB можно посмотреть на официальной странице Enterprise JavaBeans [11] и в книге [12, с. 776–787].

СПИСОК ЛИТЕРАТУРЫ

1. **Спецификация** Interface ORB
   ftp://ftp.omg.org/pub/docs/formal/99-07-08.pdf.
2. **Спецификация** ORB Interoperability Architecture
   ftp://ftp.omg.org/pub/docs/formal/99-07-17.pdf.
3. **Спецификация** Corba Beans
   ftp://ftp.omg.org/pub/docs/formal/99-07-05.pdf.
4. **Спецификация** IDL Syntax and Semantics
   ftp://ftp.omg.org/pub/docs/formal/99-07-07.pdf.
5. **Цимбал А.** Технология CORBA для профессионалов. — СПб: Питер, 2001. — 624 с.
6. **Спецификация** Interface Repository
   ftp://ftp.omg.org/pub/docs/formal/99-05-18.pdf.
7. http://www.w3.org.
8. **Спецификация** GIOP
   ftp://ftp.omg.org/pub/docs/formal/99-10-11.pdf.
9. **Спецификация** Naming Service
   ftp://ftp.omg.org/pub/docs/formal/97-12-10.pdf.
10. Orfali R., Harkey D. Client/Server Programming with Java and CORBA. 2-nd Edition. — John Wiley & Sons, 1998.
11. **Enterprise** JavaBeans http://java.sun.com/products/ejb/
12. **Эккель Б.** Философия Java. — СПб: Питер, 2001. — 880 с. — (Библиотека программиста).
13. **Вебер Дж.** Технология Java в подлиннике. — Пер. с англ. — СПб.: BHV, 2000. — 968 с.

ПРИЛОЖЕНИЕ

Пример 1. CORBA

Приводимый пример состоит из серверной и клиентской части. Для простоты предполагается, что сервер и клиент находятся на одной машине. Обе программы реализованы на Java, но можно использовать два разных языка (что чаще всего случается в реальных ситуациях).

Первый шаг состоит в написании IDL-описания разрабатываемого сервиса сервера myServer. IDL файл передается программисту клиентской стороны и становится мостом между языками.

```
module my {
    interface myInterface {
       string getInfo();
    };
};
```

Это декларация интерфейса myInterface внутри пространства имен my. Интерфейс состоит из единственного метода, который возвращает информацию в формате string.

Второй шаг состоит в компиляции IDL для создания кода стабов и скелетов Java, который будет использоваться для реализации клиента и сервера. Инструмент от Sun, поставляемый с JavaIDL, называется idltojava:

```
idltojava my.idl
```

Idltojava создаст директорию my с несколькими Java файлами. _myInterfaceImplBase.java — это скелет, который мы будем использовать для реализации объекта сервера, а _myInterfaceStub.java будет использован для клиента. Существует Java представление IDL интерфейса в my.java и набор других файлов поддержки, например, для облегчения доступа к операции сервиса указания имен. Классы клиента и сервера будут находиться в пакете «my».

Реализация серверного объекта выполнена в классе myServer, который создает объект сервера, регистрирует его с помощью ORB, дает имя ссылке на объект, а затем ожидает клиентского запроса.

```
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Реализация серверного объекта
class myServer extends _myInterfaceImplBase {
    public String getInfo() {
            return DateFormat.getTimeInstance(DateFormat.FULL).
                        format(new Date(System.currentTimeMillis()));
    }
}
```

Сервер реализует единственный метод интерфейса, getInfo(). Он возвращает точное время на сервере.

```
// Реализация удаленного приложения
public class myRemoteServer {
public static void main(String[] args) throws Exception {
   // Создание и реализация ORB
   ORB orb = ORB.init(args, null);
   // Создание серверного объекта и регистрция
   myServer ServerObjRef =  new myServer();
   orb.connect(ServerObjRef);
   // Получение корневого контекста имен
   org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
   NamingContext ncRef = NamingContextHelper.narrow(objRef);
   // Присвоение строкового имени ссылки на объект
   NameComponent nc = new NameComponent("myInterface", "");
   NameComponent[] path = { nc };
   ncRef.rebind(path, ServerObjRef);
   // Ожидание запроса клиента:
   java.lang.Object sync =  new java.lang.Object();
   synchronized(sync) { sync.wait(); }
}
}
```

После запуска сервера процесс входит в состояние ожидания. Из-за того что это временное обслуживание, время жизни ограничено серверным процессом.

Код клиента:

```
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class RemoteClient {
    public static void main(String[] args) throws Exception {
      // Создание и инициализация ORB:
      ORB orb = ORB.init(args, null);
```

```
        // Получение контекста наименования:
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);
        // Разрешение ссылки на строковый объект для
        // сервера времени:
        NameComponent nc = new NameCompo-
nent("myInterface","");
        NameComponent[] path = { nc };
        ExactTime timeObjRef = ExactTimeHelper.narrow(
                                    ncRef.resolve(path));
        // Выполнение запроса к серверу:
        String exactTime = timeObjRef.getInfo();
        System.out.println(exactTime);
    }
}
```

Первые несколько строк делают то же, что они делали в серверном процессе: инициализируют ORB и разрешают указатель на сервис указания имен. Далее необходима ссылка на объект для обслуживающего объекта, поэтому передается ссылка на строковый объект в метод resolve(), и приводиться результат к ссылке на интерфейс myInterface методом narrow(). В конце выполняется запрос к серверу и вызывается метод удаленного объекта getInfo().

Этот простой пример предназначен для работы без сети, но ORB обычно конфигурируется в независимости от местоположения. Когда сервер и клиент находятся на разных машинах, ORB может разрешать удаленные строковые ссылки, используя Implementation Repository. Хотя Implementation Repository является частью CORBA, для него нет спецификации, так что он различен у разных производителей.

Пример 2. EJB (CMP Entity Bean)

Компонент с данными CMP является наиболее простым в реализации для разработчиков и наиболее сложным при поддержке для EJB сервера. Вся логика по синхронизации компонента с базой данных поддерживается контейнером автоматически. Это значит, что разработчику нет необходимости писать код доступа к данным. Уровень поддержки различных баз данных зависит от поставщика EJB сервера.

Как и компоненты других типов, CMP компонент должен определяться как минимум двумя интерфейсами и классом, который реализует этот компонент. Класс компонента должен реализовывать методы создания (create-методы), которые определяются во внутреннем интерфейсе, и бизнес-методы, которые определяются в удаленном интерфейсе.

Рассмотрим класс:

```
public class CustomerBean implements javax.ejb.EntityBean {

    // Состояние компонента
    int        customerID;
    Address myAddress;
    Name    myName;

    // Методы создания
    public Customer ejbCreate (Integer id) {
      customerID = id.intValue();
      return null;
    }
    public void ejbPostCreate (Integer id) {}
    public Customer ejbCreate (Integer id, Name name) {
      myName = name;
      return ejbCreate(id);
    }
    public void ejbPostCreate (Integer id, Name name) { }



// Бизнесс методы
 public Name getName() {
    return myName;
 }
 public void setName(Name name) {
       myName = name;
 }
 public Address getAddress() {
       return myAddress;
 }
```

```
    public void setAddress(Address address) {
        myAddress = address;
    }

    // Callback методы
    public void setEntityContext(EntityContext cntx) { }
    public void unsetEntityContext() { }
    public void ejbLoad() { }
    public void ejbStore() { }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void ejbRemove() { }
}
```

Все поля этого класса (customerID, myAddress, myName) EJB контейнер сопоставляет с соответствующими полями в базе данных. В данном случае состояние компонента определяется данными примитивного типа int и объектами типа Name и Address. Реализация классов Name и Address может выглядеть следующим образом:

```
public class Name implements Serializable {
    public String lastName, firstName, middleName;
    public Name(String lastName, String firstName,
                String middleName) {
        this.lastName   = lastName;
        this.firstName  = firstName;
        this.middleName = middleName;
    }
    public Name() {}
}

public class Address implements Serializable {
    public String street, city, state, country;
    public Address(String street, String city, String state,
                String country) {
        this.street = street;
        this.city   = city;
        this.state  = state;
```

```
        this.country    = country;
    }
    public Address() {}
}
```

Таким образом, эти поля могут иметь либо примитивный тип, либо быть сериализованы (о сериализации Java [12], [13]). Следует отметить, что не все поля в классе отображаются в базе данных, возможно наличие полей для внутреннего использования компонентом. Индицирование полей, синхронизируемых с данными в базе данных, производится в описателе развертывания.

Поля состояния должны иметь типы соответствующие типам в базе данных. Данный CustomerBean возможно, к примеру, сопоставить с таблицей (CUSTOMER) в базе данных, которая может быть определена следующим образом:

```
CREATE TABLE CUSTOMER
{
    id              INTEGER PRIMARY KEY,
    last_name       CHAR(30),
    first_name      CHAR(20),
    middle_name     CHAR(20),
    street          CHAR(50),
    city            CHAR(20),
    country         CHAR(20)
}
```

После того как синхронизация состояния компонента и полей в базе данных установлена, контейнер будет управлять созданием записей, загрузкой записей, редактированием записей и удалением записей в CUSTOMER-таблице, используя методы, определяемые в удаленном и внутреннем интерфейсах. Множество полей состояния идентифицируется по уникальному ключу (customerID), которому в базе данных соответствует индекс (id) на уникальную запись, которая и устанавливает состояние компонента.

Внутренний интерфейс служит для создания нового экземпляра CMP entity bean и, следовательно, для вставки данных в базу данных. Для этого определяется метод create(). Для компонента Customer внутренний интерфейс определяется CustomerHome интерфейсом:

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

public interface CustomerHome extends EJBHome {
    public Customer create (Integer number)
      throws RemoteException,CreateException;
    public Customer create (Integer number, Name name)
      throws RemoteException, CreateException;
    public Customer findByPrimaryKey (Integer number)
      throws RemoteException, FinderException;
}
```

Приложение клиента может использовать метод create() для создания нового экземпляра Customer следующим образом:

```
CustomerHome home = // Ссылка на удаленный объект
Name name = new Name ("John", "W", "Smith");
Customer customer = home.create (new Integer(33), name);
```

Каждому декларированному методу create() во внутреннем интерфейсе должен соответствовать метод ejbCreate() и ejbPostCreate() в классе компонента. При обращении к методу create() в интерфейсе контейнер запускает соответствующий метод ejbCreate().

В нашем случае методы ejbCreate() инициализируют поля customerID и Name, и затем контейнер вставляет новую запись в таблицу CUSTOMER, индексированную уникальным ключом (customerID соответствует полю CUSOTMER.ID).

Для каждого ejbCreate() метода необходим ejbPostCreate() метод, который позволяет компоненту производить какие-либо действия с уже созданными данными до начала обслуживания запросов клиентов.

Методы, начинающиеся с "find", используются запросами EJB сервера для определения компонента по имени или по передающимся аргументам. Существуют два типа методов поиска: простой поиск и сложный. Каждый entity bean должен определять метод простого поиска findByPrimaryKey(), который возвращает ссылку на удаленный интерфейс определенного компонента, вычисляющего запрос поиска.

Методы сложного поиска возвращают коллекции (Enumeration или Collection) компонентов, которые вычисляют поисковый запрос.

Удаленный интерфейс определяет бизнес-методы компонента. Для компонента Customer определяется следующий удаленный интерфейс:

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

public interface Customer extends EJBObject {
    public Name getName () throws RemoteException;
    public void setName (Name name) throws RemoteException;
    public Address getAddress () throws RemoteException;
    public void setAddress (Address address)
       throws RemoteException;
}
```

Клиент может использовать эти методы следующим образом:

```
Customer customer = //ссылка на удаленный интерфейс компонента
// получить адресс
Address addr = customer.getAddress();
// изменить адресс
customer.setAddress(addr);
```

Бизнес методы служат не только для чтения-записи данных, но могут также использоваться для более сложных задач.

Callback Методы — в классе, реализующем компонент, также находится множество callback-методов, которые вызываются контейнером и используются для управления состоянием компонента. Callback-методы определены в интерфейсе javax.ejb.EntityBean.

```
public interface javax.ejb.EntityBean {
public void setEntityContext() { }
public void unsetEntityContext() { }
public void ejbLoad() { }
public void ejbStore() { }
```

```
public void ejbActivate() { }
public void ejbPassivate() { }
public void ejbRemove() { }
}
```

Метод setEntityContext() обеспечивает компонент контейнером EntityContext, который содержит информацию о контексте компонента такую, как текущая транзакция и информация безопасности.

Обычно код этих методов генерируется автоматически. В данном примере они не используются.

Enterprise JavaBeans клиентом — может быть одиночное простое приложение, сервлет, апплет, другой enterprise компонент или даже распределенный объект CORBA или RMI.

В первую очередь клиент должен найти EJB компонент внутри JNDI и получить ссылку на его домашний интерфейс. Домашний интерфейс используется для создания экземпляра EJB.

```
public class PerfectTimeClient {
    public static void main(String[] args) throws Exception {
    // Получение контекста JNDI с помощью JNDI
    // службы Указания Имен:
    javax.naming.Context context =
                    new javax.naming.InitialContext();
    // Поиск Домашнего интерфейса в службе JNDI
    // Naming:
    Object ref = context.lookup("Customer");
    // Приведение удаленного объекта к домашнему
    // итерфейсу:
CustomerHome home =
     (PerfectTimeHome)javax.rmi.PortableRemoteObject.
                        .narrow(ref, CustomerHome.class);
// Создание удаленного объекта из домашнего
// интерфейса:
Customer customer = home.create(customerID);
 // Вызов бизнес-метода
customer.setName (someName);
}
```

Последовательность выполняемых действий поясняется комментариями.

Н. Н. Цаценко

ТЕХНОЛОГИИ РАБОТЫ С ДАННЫМИ В ИНТЕРОПЕРАБЕЛЬНЫХ СРЕДАХ CORBA, RMI, EJB

Препринт
93

Рукопись поступила в редакцию 28.12.2001
Рецензент Ф. А. Мурзин
Редактор З. В. Скок

Подписано в печать 18.04.02
Формат бумаги 60 × 84 1/16 Объем 2.1 уч.-изд.л., 2.3 п.л.
Тираж 50 экз.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6