Степанов Е.О., к.ф.-м.н.
Ярцев Б.М.

Учебно-методическое пособие по дисциплине «Архитектуры и технологии разработки распределенного программного обеспечения»

Оглавление

Тема 1. Основные платформы и технологии

1.1. Введение

В настоящее время аппаратные и программные платформы и технологии стремительно развиваются, непрерывно появляются все новые и новые возможности. В то же время, налицо тенденция к интероперабельности (*interoperability*) платформ и совместимости технологий. В этом разделе рассматриваются основные тенденции этого развития.

Изучив учебный материал данного раздела, Вы:

- узнаете или пополните свои знания о том, каковы существующие на сегодняшний день основные программные и аппаратные платформы;
- узнаете или пополните свои знания об основных кроссплатформенных технологиях.

В рамках темы рассматриваются следующие учебные вопросы:

- основные аппаратные и программные платформы;
- кросс-платформенные технологии;
- web-технологии.

1.2. Основные аппаратные и программные платформы

Рост компьютерных и информационных технологий за сравнительно недолгое время, прошедшее с момента появления первых компьютеров (конец 1940х гг.) был невероятно стремительным и пока не проявляет никакой тенденции к замедлению. Считается, что каждые 10 лет происходит полная смена технологий в этих областях. В результате невероятно большое число аппаратных и программных технологий и платформ, которые, казалось бы, еще недавно были самыми передовыми и повсеместно используемыми, в настоящее время осталось лишь в памяти тех, кому с ними пришлось работать. Новые поколения разработчиков программного обеспечения, как правило, не знают даже техники и технологий десятилетней давности (а если и знают, то только из специальных ВУЗовских курсов), поскольку состояние дел в области компьютерных и информационных технологий успело полностью поменяться несколько раз за эти годы. Такие стремительные изменения, кстати, делают весьма неустойчивым компьютерный бизнес: на наших

глазах многие фирмы-производители оборудования или программного обеспечения, имевшие, казалось бы, сверхустойчивое положение на рынке, в считанные годы проигрывали конкуренцию и иногда полностью исчезали, а на их месте появлялись новые «звезды». Так, к примеру, всего несколько лет назад произошло с одной из крупнейших в компьютерном мире фирмой *DEC*, долгие годы в значительной мере определявшей пути развития вычислительной техники и программного обеспечения, и сумевшей построить вполне самобытную «цивилизацию» компьютерных и программных решений – фирмы уже больше не существует, а про ее супер-брэнды *PDP*, *VAX* и соответствующее программное обеспечение помнят весьма немногие.

Учитывая все сказанное, представляется практически нецелесообразным давать сколько-нибудь подробный обзор аппаратных и программных архитектур, имеющихся в настоящее время – их срок жизни весьма мал. Ограничимся поэтому лишь весьма схематическим изложением основных платформ, с которыми приходится иметь дело современному разработчику. Весьма условно можно классифицировать основные встречающиеся в наше время аппаратные платформы следующим образом.

- Платформы на базе процессоров *Intel* и их аналогов (*AMD*).
- Высокопроизводительные сервера и рабочие станции *SUN* (на базе процессоров *SunSparc*).
- Высокопроизводительные сервера *HP* (на базе *RISC*-процессоров).
- Платформы *Apple*.

Архитектура процессора: *RISC* или *CISC*?

В 80-х годах прошлого века была предложена архитектура процессора с сокращенным набором машинных команд (*RISC – Reduced Instruction Set Computer*). Дейв Паттерсон и Карло Секуин сформулировали четыре основных принципа архитектуры *RISC*:

1. Любая операция должна выполняться за один такт, вне зависимости от ее типа.

2. Система команд должна содержать минимальное количество наиболее часто используемых простейших инструкций одинаковой длины.

3. Операции обработки данных реализуются только в формате «регистр–регистр» (операнды выбираются из оперативных регистров процессора, и результат операции записывается также в регистр; а обмен между оперативными регистрами и памятью выполняется только с помощью команд чтения/записи).

4. Состав системы команд должен быть «удобен» для компиляции операторов языков высокого уровня

Создатели *RISC*-процессоров взяли набор из очень простых наиболее часто используемых команд, которые выполняются быстро, и объединили его с такими

технологиями, как конвейерная обработка. В результате получился процессор, который имеет лучшую производительность для большинства приложений и теоретически стоит меньше, поскольку сам он небольшой и его производство обходится дешевле.

По аналогии процессоры традиционной архитектуры стали называть *CISC – Complex Instruction Set Computer*.

В список основных поставщиков *RISC*-систем входят компании *Hewlett-Packard* (*PA-RISC*), *Sun Microsystems Computers* (*SPARC*), *Digital Equipment* (*Alpha*), *Silicon Graphics* – модуль *MIPS* (*R210000*) и союз *IBM* и *Motorola* (*PowerPC*).

С другой стороны, семейство *Pentium* компании *Intel* продолжает реализацию более традиционной вычислительной архитектуры с полным набором машинных команд (*CISC*). *CISC*-процессоры содержат в сотни раз больше команд, чем *RISC*-процессоры, и используют от 8 до 12 способов адресации памяти по сравнению с 2-3 способами в *RISC*. Однако технические различия между *RISC* и *CISC* в последние годы становятся все менее четкими, особенно в том, что касается общей производительности систем. Одна архитектура заимствует хорошие идеи у другой. Раньше *RISC*-процессоры определялись как микропроцессоры с количеством команд меньше 128, сейчас же они имеют 200 команд – сравните с набором из 300 и более команд в *CISC*. Сегодня *CISC*-процессоры используют конвейеризацию и другие современные технологии. Оба лагеря применяют большую кэш-память для повышения производительности.

Основные программные платформы можно классифицировать условно следующим образом:

- Платформы *Microsoft* (*Windows NT*/*XP*/...)
- Платформы на базе *Unix*. В последнее время среди версий *Unix* наиболее популярен *Linux* разных модификаций. К версиям *Unix* относятся и системы *Solaris* (для платформ *Sun*), а также весьма своеобразный «гибрид» *Unix* и *Windows* для платформ *Apple — Mac OS*.

Следует отметить, что операционные системы *Unix* и созданный вокруг них универсум программных продуктов, идей и технологий, являются одними из едва ли не самых «долгоживущих» в мире программного обеспечения. Первая система *Unix*, практически ничем принципиально не отличающаяся от ее современных «клонов», была создана более 30 лет назад!

1.3. Состояние корпоративной программной среды типичной организации

В настоящее время наблюдается тенденция к унификации программных и аппаратных платформ, используемых в типовых конфигурациях.

- Основная масса компьютеров базируется на платформе *Intel* или *AMD*, возможно имеются несколько компьютеров *Compaq*, *Sun* и т.п.
- Используемые операционные системы *MS Windows*, *Linux*, другие *UNIX*-подобные ОС. Основная операционная система, установленная на рабочих местах пользователей – *MS Windows*. Большинство серверов работает также под *Windows*. Часть серверов работает под *Linux* или другими *UNIX*-подобными ОС.
- Основное офисное программное обеспечение – *MS Office*.
- Основная почтовая программа *MS Outlook*, *MS Outlook Express* или специальный почтовый клиент (популярность набирает *Mozilla Thunderbird)*.
- Основной интернет-браузер *MS Internet Explorer* (сейчас набирает популярность браузер *Mozilla Firefox*, иногда используется браузер *Opera*).
- Основная система документооборота работает на основе *Windows* и *MS Exchange*.
- Основной *Web-сервер Apache* или *IIS* на платформе *UNIX* или *Windows*.
- Основная корпоративная *СУБД MS SQL Server* или *Oracle*, также дополнительно используется *MS Access* или другие *СУБД*, но исключительно как локальные. В малых компаниях бывают популярны MySQL и PostgreSQL.

1.4. Кросс-платформенные технологии

Кросс-платформенные технологии обеспечивают совместную эксплуатацию различных аппаратных и программных платформ в интересах организаций-потребителей.

1.4.1. Основные архитектуры программного обеспечения

Автономные (standalone) приложения

Такими могут быть, как правило, сервисные программы, системные утилиты, текстовые и графические редакторы, компиляторы, достаточно простые корпоративные программы. Развитая корпоративная информационная система, как правило, не может состоять из отдельных, не связанных между собой компонентов.

Двухзвенная архитектура «клиент-сервер»

Эта архитектура получила распространение с начала 1990-х годов на фоне роста рынка персональных компьютеров и снижения спроса на мэйнфреймы.

В архитектуре «клиент-сервер» программное обеспечение разделено на две части – клиентскую часть и серверную часть. Задача клиентской-части (программы-клиента) состоит во взаимодействии с пользователем, передаче пользовательского запроса серверу, получение запроса от серверной части (программы-сервера) и представление его в удобном для пользователя виде. Программа-сервер же обрабатывает запросы клиента и выдает ответы. Классические примеры: *Web*-технологии (клиент-браузер, сервер-*Web*-сервер), работа с распределенными СУБД (клиент – специальная программа, сервер – сервер базы данных).

Развитие архитектуры «клиент-сервер», а особенно появление современных графических интерфейсов, привело сначала к появлению разновидности архитектуры клиент-сервер, называемой «*архитектура с толстым клиентом»*. Здесь логика представления данных и бизнес-логика размещаются на клиенте, который (скажем, в случае, когда сервером является СУБД) общается с логикой хранения и накопления данных на сервере, используя язык структурированных запросов *SQL*. Однако необходимость установки «толстых клиентов», требующих значительного количества специальных библиотек и специальной настройки окружения, на большое число пользовательских компьютеров с различными операционными средами, как правило вызывает массу проблем. Как альтернатива поэтому возникла также двухзвенная архитектура «*с тонким клиентом»*. При этом в идеале программа-клиент реализует лишь графический интерфейс пользователя (*GUI*) и передает/принимает запросы, а вся бизнес-логика выполняется сервером. В идеале клиентом является просто интернет-браузер, который имеется в стандартной операционной среде любого пользовательского компьютера и не требует специальной настройки, установки специализированного ПО и т.п. К сожалению, такая схема тоже не свободна от недостатков, хотя бы уже потому, что серверу приходится брать на себя иногда не свойственные для него функции реализации бизнес-логики приложения (например, серверу СУБД приходится выполнять расчеты!)

Многозвенная (multitiered) архитектура

Начало процессу развития корпоративного программного обеспечения в многозвенной архитектуре было положено еще в рамках технологии «клиент/сервер». В них наряду с клиентской частью

приложения и сервером баз данных появились серверы приложений (*Application Servers*). В идеале:

- программа-клиент реализует *GUI*, передает запросы серверу приложений и принимает от него ответ,
- сервер приложений реализует бизнес-логику и обращается с запросами к серверу «третьего уровня» (например, серверу базы данных за данными),
- сервер третьего уровня обслуживает запросы сервера приложений.

Программа-клиент, таким образом, может быть «тонкой». Преимущества такой архитектуры очевидны:

- изменения на каждом из звеньев можно осуществлять независимо;
- снижаются нагрузки на сеть, поскольку звенья не обмениваются между собой большими объемами информации;
- обеспечивается масштабирование и простая модернизация оборудования и программного обеспечения, поддерживающего каждое из звеньев, в том числе обновление серверного парка и терминального оборудования, СУБД и т.д.;
- Приложения могут создаваться на стандартных языках третьего или четвертого поколения (*Java*, *C*/*C++*).

Следующий логический шаг — дальнейшее увеличение числа звеньев, причем возрастет не только за счет разбиения, когда «утоньшается» каждое из известных технических звеньев, но вся бизнес-модель строится как многозвенная. Современные корпоративные программные системы представляют собой, как правило, сложные системы взаимодействующих между собой на разных уровнях компонентов, каждые из которых могут являться клиентами для одних компонентов и серверами для других.

Основной проблемой систем, основанных на двухзвенной архитектуре «клиент-сервер», или тем более на многозвенной архитектуре, является то, что от них требуется мобильность в как можно более широком классе аппаратно-программных сред. Даже если ограничиться *UNIX*-ориентированными локальными сетями, в разных сетях применяется разная аппаратура и протоколы связи. Попытки создания систем, поддерживающих все возможные протоколы, приводит к их перегрузке сетевыми деталями в ущерб функциональности. Еще

более сложный аспект этой проблемы связан с возможностью использования разных представлений данных в разных узлах неоднородной локальной сети. В разных компьютерах может существовать различная адресация, представление чисел, кодировка символов и т.д. Это особенно существенно для серверов высокого уровня: телекоммуникационных, вычислительных, баз данных.

Общим решением проблемы мобильности такого рода систем является использование технологий, реализующие протоколы удаленного вызова процедур (*RPC – Remote Procedure Call*) стандартизованным и платформо-независимым способом. При использовании таких технологий обращение к сервису в удаленном узле выглядит как обычный вызов процедуры (методов удаленных объектов). Средства *RPC*, в которых, естественно, содержится вся информация о специфике аппаратуры локальной сети и сетевых протоколов, переводит вызов в последовательность сетевых взаимодействий. Тем самым, специфика сетевой среды и протоколов скрыта от прикладного программиста.

При вызове удаленной процедуры, программы *RPC* производят преобразование форматов данных клиента в промежуточные машинно-независимые форматы, и затем преобразование в форматы данных сервера. При передаче ответных параметров производятся обратные преобразования. Таким образом, если система реализована на основе стандартного пакета *RPC*, она может быть легко перенесена в любую открытую среду.

1.4.2. Технология *CORBA*

CORBA (*Common Object Request Broker Architecture*) – это набор открытых спецификаций интерфейсов, определяющий архитектуру технологии межпроцессного и платформо-независимого манипулирования объектами. Разработчиками данных интерфейсов являются *OMG* и *X/Open*.

Object Management Group, *Inc.* (*OMG*) – это интернациональная организация, основана в 1989 г., состоящая более чем из 800 членов: поставщиков информационных систем, разработчиков программного обеспечения и пользователей. *OMG* продвигает теорию и практику объектно-ориентированной технологии в область практической разработки программного обеспечения. Этот процесс включает в себя разработку промышленных стандартов и спецификаций управления объектами с целью создания общей базы для разработки программного обеспечения. Первоочередными задачами являются: повторное использование, переносимость и интероперабельность объектно-ориентированного программного обеспечения в распределенных, гетерогенных средах. Поддержка данных стандартов создает

возможность разрабатывать гетерогенные приложения, работающие на всех основных платформах и операционных системах.

X/Open — независимая всемирная открытая организация, поддерживаемая большинством крупнейших поставщиков информационных систем, пользовательских организаций и компаний-производителей программного обеспечения. *X/Open* разрабатывает на основе существующих и создающихся стандартов всеобъемлющее и интегрированное системное окружение – *Common Applications Environment* (*CAE*). Компоненты *CAE* определены в стандартах *X/Open CAE*. Основная цель *CAE* – создание пакетов программных интерфейсов (*API*) которые могут применяться на практике с сохранением максимальной переносимости на уровне исходных кодов программ. *API* также повышают уровень взаимодействия приложений при помощи предоставления определений и ссылок на протоколы и их профили. Вышеназванные спецификации тщательно тестируются, выдержавшим тестирование присваивается *X/Open trademark* (*XPG brand*), лицензированная *X/Open*.

Концептуальной инфраструктурой, на которой базируются все спецификации *OMG*, является *Object Management Architecture* (*OMA*). В состав *OMA* входят разнообразные стандартизованные или в настоящий момент стандартизируемые *OMG* службы, сервисы, программные образцы и шаблоны (*CORBAservices*, *horizontal and vertical CORBAfacilities*), язык определения интерфейсов распределенных объектов *IDL* (*Interface Definition Language*), стандартизованные или стандартизируемые отображения *IDL* на языки программирования и, наконец, объектная модель *CORBA*.

Реализовать технологию в соответствии со спецификациями может кто угодно. Созданные программные продукты, естественно, уже не являются открытыми, а становятся коммерческими продуктами.

1.4.2.1. Архитектура *CORBA*

CORBA определяет, каким образом программные компоненты, распределенные по сети, могут взаимодействовать друг с другом вне зависимости от окружающих их операционных систем и языков реализации. Центральным элементом архитектуры *CORBA* является *ORB* (*Object Request Broker*) – программное обеспечение, обеспечивающее связь между объектами, в том числе позволяющее

- найти удаленный объект по Объектной Ссылке (*IOR – Interoperable Object Reference*),
- вызвать метод удаленного объекта, передав ему входные параметры (*marshaling parameters*),
- получить возвращаемое значение и выходящие параметры (*unmarshaling parameters*).

Тем самым *ORB* является связующим звеном между распределенными частями основанной на технологии *CORBA* системы, позволяя одной части системы не заботиться о физическом расположении других частей (объектов) системы. На рынке представлены *ORB* разных производителей (например, *VisiBroker*, *WebLogic*), но все они соответствуют единой спецификации *CORBA*. Поэтому в принципе *CORBA* позволяет строить распределенные системы, одновременно используя *ORB* разных производителей, и строя систему одновременно на различных платформах и различных сетевых протоколах (это в терминологии *CORBA* называется интероперабельностью – *interoperability*).

В архитектуре *CORBA* каждый объект, методы которого доступны другим объектам (обычно его называют *CORBA*-объектом) имеет уникальную по всей доступной сети Объектную Ссылку (*IOR – Interoperable Object Reference*), по которой к нему можно обратиться. Искать *CORBA*-объекты можно как по *IOR*, так и по символическим именам, если они зарегистрированы (обычно при создании) в специальном сервисе имен (*NameService*). Для обращения к методам *CORBA*-объекта последний имеет открытый для всех остальных *CORBA*-объектов интерфейс. Интерфейсы *CORBA*-объектов принято описывать на специальном, определенном спецификацией *CORBA* языке *IDL* (*Interface Definition Language*). Производители *ORB* поставляют вместе с *ORB* также и утилиты, преобразующие описания интерфейсов *CORBA*-объектов в конструкции соответствующих языков программирования.

Основой интероперабельности является протокол *GIOP – General inter-ORB Protocol*, предназначенный для связи между объектами и *ORB* в сети. Стандартизация коммуникационного протокола позволяет разработчикам различных частей корпоративной системы совершенно не заботиться об используемых *ORB*ах в других частях (*ORB* доменах) системы. Почти все современные *ORB*ы строятся на основе *IIOP — Internet inter-ORB Protocol* (это версия общего протокола *GIOP*, предусматривающая использование в качестве транспортного протокола *TCP/IP*).

Спецификация *CORBA* предусматривает также ряд стандартизованных сервисов (*CORBA Services*) и горизонтальных и вертикальных Общих Средств (*Common Facilities*). Сервисы собой обычные *CORBA*-объекты со стандартизованными (и написанными на *IDL*) интерфейсами. К таким сервисам относится, например, уже упомянутый сервис имен *NameService*, сервис сообщений, позволяющий *CORBA*-объектам обмениваться сообщениями, сервис транзакций, позволяющий *CORBA*-объектам организовывать транзакции. В реальной системе не обязательно должны присутствовать все сервисы, их набор зависит от требуемой функциональности. На сегодня разработано всего 14 объектных сервисов.

Между объектными сервисами и общими средствами *CORBA* нет четкой границы. Последние тоже представляют собой *CORBA*-объекты со стандартизованными интерфейсами. *Common Facilities* делятся на горизонтальные (общие для всех прикладных областей) и вертикальные (для конкретной прикладной области). Например, разработаны *Common Facilities* для медицинских организаций, для ряда производств и т.п.

1.4.3. Технология *SOAP*

Основное содержание *SOAP* (*Simple Object Access Protocol*) состоит в обмене сообщениями между удаленными объектами по протоколу *HTTP* с использованием *XML* в качестве транспорта. Спецификация *SOAP* поддерживается и развивается консорциумом *W3C* (см. http://www.w3.org/TR/SOAP/).

По функциональным возможностям технология *SOAP* весьма сходна с первыми версиями *CORBA*. Однако у нее есть одно несомненное достоинство: простота. На уровне передачи данных в глобальных сетях, между предприятиями, где большой сложности взаимодействие не предвидится – это оптимальное решение по соотношению время разработки/функциональность. Существуют многочисленные мосты (*CORBA*/*SOAP*, *C++*/*SOAP*, *Java*/*SOAP*).

1.4.4. Технологии *COM*/*DCOM* и *.NET*

COM (*Component Object Model*) – это стандарт *Microsoft*, определяющий структуру и взаимодействие компонентов программного обеспечения в современных операционных системах *MS Windows*. Архитектура современных *Windows*-приложений основана на *COM*: мир этих приложений – это мир *COM*-компонент. Компоненты *COM* обладают уникальностью и предоставляют другим компонентам *COM* стандартным образом описанные интерфейсы, позволяющие получить доступ к методам этих компонентов. *COM* определяет механизм связи только между локальными (т.е. находящимися на том же компьютере) компонентами.

DCOM (*Distributed Component Object Model*) – это распределенная версия *COM*, обеспечивающая механизм связи между удаленным *COM*-компонентами (т.е. находящимися на разных компьютерах, но в среде *MS Windows*). Фактически *DCOM* это *COM* с добавленным к последнему механизмом *RPC* (*remote procedure call*). Сходную функциональность взаимодействия удаленных *Windows*-приложений можно получить с использованием активно развиваемой в последнее время фирмой *Microsoft* технологии *.NET*.

Важно подчеркнуть, что упомянутые в данном разделе технологии относятся исключительно к операционным системам *Microsoft*.

1.4.5. Технология *Enterprise Java Beans*

Архитектура *EJB* – это компонентная архитектура, предназначенная для разработки и развертывания распределенных бизнес-приложений, основанных на компонентах. Приложения, созданные с помощью архитектуры *EJB*, являются масштабируемыми, ориентированными на транзакции и безопасными при работе в многопользовательском режиме. Эти приложения, однажды написанные, могут затем быть развернуты на любой серверной платформе, поддерживающей спецификацию *EJB*. Это определение можно немного упростить при помощи описанных ранее понятий. *Enterprise Java Beans* – это стандартная модель серверных компонентов для мониторов компонентных транзакций.

Enterprise Bean-компоненты являются *Java* (*J2EE*) объектами, реализующими технологию *Enterprise Java Beans* (*EJB*). Каждый такой компонент выполняется под управлением сервера приложений, который должен соответствовать так называемой спецификации *EJB*-контейнера, т.е. поддерживать соответствующий *API — EJB Container API* (обычно сервер приложений в таком случае называют *EJB*-контейнером). *EJB*-контейнер предоставляет компонентам (*Enterprise Beans*) сервисы системного уровня (например, многопоточность, механизм транзакций), оставаясь при этом прозрачным для разработчика приложений. Эти системные сервисы позволяют разработчику быстро создавать и разворачивать *Enterprise Bean*-компоненты: контейнер как бы «закрывает» от разработчика *EJB* все сложности системного характера (например, уже упомянутые многопоточность или механизм транзакций), позволяя ему сосредоточиться исключительно на бизнес-логике приложения.

Enterprise Bean-компонент — это объект требуемого класса, описанного на языке программирования *Java*, расположенный на стороне сервера приложений и выполняющий часть бизнес-логики приложения (этим занимается собственно код компонента, осуществляющий задачи приложения). Например, в приложении контроля инвентаря, *Enterprise Bean*-компоненты могут реализовывать бизнес-логику приложения в методах *checkInventoryLevel()* и *orderProduct()*. Вызывая эти методы, удаленные клиенты могут получать доступ к инвентарным сервисам приложения.

Существует несколько причин, по которым использование *Enterprise Bean*-компонентов упрощает разработку больших распределенных корпоративных приложений.

- Первой причиной является тот факт, что *EJB*-контейнер предоставляет все необходимые сервисы системного уровня, позволяя разработчику *Bean*-компонента сконцентрироваться на решении бизнес-задач. Именно *EJB*-

контейнер (а не *Bean*-разработчик) является ответственным за обеспечение работоспособности таких механизмов, как транзакции и авторизация доступа.

- Вторым преимуществом *EJB*-компонентов является их расположение на сервере. Вследствие этого, разработчикам клиентов не приходится включать бизнес-логику в состав клиентского приложения – в таком коде не должно быть функциональности, реализующей бизнес-правила или доступ к базам данных. В результате, клиентское приложение получается гораздо меньшего размера, что очень важно для выполнения на устройствах с ограниченными ресурсами.
- Третьим преимуществом *Enterprise Bean*-компонент является их переносимость, сборщик приложений может собирать новые приложения из уже существующих *Bean*-компонент. Такие приложения могут быть запущены на любом *J2EE*-совместимом сервере.

Следует задуматься об использовании *Enterprise Bean*-компонент, если ваше приложение отвечает хотя бы каким-то требованиям из перечисленных ниже.

- Приложение должно быть масштабируемым. Чтобы подстроится к растущему количеству пользователей, разработчикам, возможно, придется распределить компоненты приложения между несколькими серверами. Вне зависимости от компоновки компонент на серверах, их расположение остается прозрачным для клиентов.
- Требуется механизм транзакция для обеспечения целостности данных. *Enterprise Bean*-компоненты поддерживают транзакции – механизм, управляющий одновременным доступом к разделяемым объектам.
- У приложения будет множество клиентов. Требуется всего несколько строк кода в клиентских приложениях для нахождения *Enterprise Bean*-компонент. Клиентские приложения могут быть небольшими, многочисленными и различными.

На сегодняшний день корпорацией *Sun Microsystems* было выпущено пять спецификации *EJB – EJB 1.0, EJB 1.1, EJB 2.0, EJB 2.1 и EJB 3.0.* В спецификации *EJB 1.0* были впервые описаны сеансовые (*session bean*) и объектные (*entity bean*) компоненты. Спецификация *EJB 1.1* расширяет спецификацию *EJB 1.0*. В *EJB 2.0* были добавлены

компоненты, управляемые асинхронными сообщениями *JMS* (*Java Messaging Service*), а также *EJB Query Language* (*EQL*) – язык запросов. В *EJB 2.1* был модифицирован и улучшен *EQL*, добавлена возможность вызова объектных компонент через *HTTP*/*SOAP*. Также компоненты, управляемые сообщениями, смогли принимать сообщения не только по протоколу *JMS*, но и по другим протоколам. Последняя на данный момент версия *EJB – EJB 3.0*. В ней модифицированы механизмы описания компонент (вместо *XML*-файла – метаданные), а сам процесс разработки переведен на *JAVA 5.0*.

1.4.6. Технология *JINI*

Jini представляет собой технологию создания распределенных систем, ориентированную исключительно на использование *Java*. В настоящий момент *Jini* является торговой маркой *Sun Microsystems*.

Технология *Jini* состоит из трех основных компонентов:

- **Инфраструктура.** Включает в себя распределенную систему защиты, которая интегрирована в *RMI* (*Remote Method Invocation*), представляющий собой механизм для нахождения, активации и захвата объектов *сервисов*. Инфраструктура состоит из объектов, использующих протоколы для передачи информации во время транзакций. На уровне транзакций происходят запросы и передача информации. Для поиска объектов и передачи информации между ними используется менеджер транзакций (*transaction manager*). Обязанности менеджера транзакций этим не ограничиваются. Помимо этого, он обязан координировать работу системы во время выполнения запросов и передавать найденую по этим запросам информацию;
- **Модель программирования** использует язык программирования *Java* и компоненты *JavaBeans* для организации интерфейсов транзакций и написания приложений, использующих модель распределенных вычислений;
- **Сервисы** имеют определенный унифицированный интерфейс и набор методов, посредством которых возможно общение с ними. Реализация сервисов не требует использования программной модели *Jini*, однако эта модель необходима при взаимодействии сервисов между собой. Причем сервисы в этом случае чем-то подобны процессам в *Unix*. Каждый сервис может использовать другие сервисы для выполнения своих задач, а также порождать новые сервисы, специализирующиеся на решении определенных вопросов.

В отличие от *EJB*, технология *JINI* не требует наличия специальных серверов приложений. Кроме того, если модель использования *EJB* принципиально двух- или трехзвенна (существует клиент, запрашивающий методы *EJB*, работающий под управлением контейнера, и, как правило, сервер, например, СУБД, к которому обращается в процессе работы *EJB*, причем иерархия запросов в этой схеме строго задана), то в модели *JINI* все сервисы абсолютно равноправны между собой (каждый из них может быть как сервером, так и клиентом к любому). Такая «равноправная» архитектура взаимосвязей называется *одноранговой* (*peer-to-peer*). В модели *JINI* сервисы представляют, таким образом, своего рода «интеллектуальные устройства» (можно представить себе в качестве примера сервис печати), общающиеся между собой по стандартизованным правилам, имеющие стандартизованные имена, общую модель безопасности и т.п. Такой универсум сервисов-«интеллектуальных устройств» *JINI* принято называть *JINI Federation*. «Интеллектуальные устройства» могут сами добавлять себя в этот универсум (например, сервис печати при включении принтера) или, наоборот, выходить из него (сервис печати при выключении принтера), без необходимости какого-либо «внешнего» воздействия (диспетчера, оператора и т.п.)

Поиск сервиса, который может выполнить определенную задачу, происходит приблизительно по такому сценарию.

- Объект клиента посредством провайдера сервисов (*Service Provider*) — объекта, «специализирующегося» на поиске объектов-сервисов, — находит необходимый сервис.
- При нахождении необходимого объекта, клиент исследует его, исходя из параметров соответствующего запроса. Это исследование найденого сервиса происходит посредством проверки его свойств.
- После этого подходящий сервис копируется *на локальный диск компьютера клиента*. Последующие действия с ним происходят, как с локальным объектом, с помощью вызова его методов, что в некоторой мере разгружает трафик.

Технология *Jini* разрабатывалась с целью создания системы, которая бы требовала к себе мало внимания при обслуживании, успевала за постоянным изменением и наращиванием системы и обеспечивала постоянную доступность сервисов посредством Интернета. Безопасность системы и конфиденциальность информации передаваемой в сети достигается за счет распределенной системы безопасности. Наращиваемость систем возможна за счет добавления новых, наследования и изменения старых сервисов, доступных

посредством интернет. Постоянная доступность становится возможной за счет рассредоточенности системы, в которой можно изменять приложения беспрерывно, так, что сервис будет доступен из Интернета постоянно.

1.5. *Web*-технологии

Web-технологии чрезвычайно сильно используются в современном корпоративном программном обеспечении. Перечислим основные используемые технологии *Web*-программирования.

1.5.1. *CGI*-скрипты

CGI-скрипт – это программа, выполняемая на стороне сервера и следующая правилам интерфейса *CGI* (*Common gateway interface*). Исторически это первая технология «динамического» программирования для *Web*. *CGI*-скрипты могут быть как обычными исполнмаемыми модулями (написанными на любом языке программирования, например, на C++), так и сценариями («скриптами»), написанными на интерпретируемых языках (например, на *Perl*, *Unix shell*, *Tcl*).

Последовательность действий при работе с *CGI*-скриптом следующая:

- Клиент посылает *HTTP*-запрос *Web*-серверу.
- *Web*-сервер инициирует выполнение *CGI*-скрипта (т.е. просто запускает программу, если она представляет собой исполняемый модуль, либо запуская соответствующий интерпретатор с подачей ему на вход текста сценария, если это сценарий) и передает ему необходимые данные.
- *CGI*-скрипт выполняется, и по окончании работы передает результаты (ответ на исходный запрос) вызвавшему его серверу. При этом *CGI*-скрипт может производить сколь угодно сложные действия, например, обращаться к другим удаленным программам и т.п.
- *Web*-сервер отдает полученные от *CGI*-скрипта данные клиенту.

Из современных *Web*-технологий это, пожалуй, самая простая, но и самая немаштабируемая, а также немобильная (платформо-зависимая) и не вполне устойчивая технология.

1.5.2. Специальные интерпретируемые языки скриптов

Ряд *Web*-серверов предусматривают встроенные интерпретаторы специальных языков для динамического *Web*-программирования. Примерами являются *ASP* для *Web*-сервера *Internet Information Server* (*IIS*) и *PHP* (например, для *Web*-сервера *Apache*).

ASP (или, соответственно, *PHP*) страница представляет собой обычный *HTML* файл, который кроме текста и тэгов *HTML* содержит еще и конструкции соответствующего языка (*ASP* или *PHP*). При запросе этого документа клиентом *Web*-сервер сначала просматривает документ, интерпретируя директивы соответствующего языка (*ASP* или *PHP*) и преобразуя их в обычный статический *HTML*, который и отдается клиенту.

Важно отметить, что как *ASP*, так и *PHP*, являются полноценными языками программирования, что позволяет создать с использованием этих технологий сложнейшие *Web*-системы (вплоть до полномоаштабномго управления производством или, например, торговлей). Эти технологии кроме того, весьма просты, и поэтому популярны, например, для создания электронных магазинов. Недостатком является принципиальная интерпретируемость этих языков, а также существенная привязка к конкретному *Web*-серверу (например, *ASP* работает только для *IIS*).

1.5.3. Апплеты

Апплеты — это программы на *Java*, работающие под управлением другой программы (как правило, интернет-браузера). Апплеты загружаются с *Web* сайта вместе со статическим *HTML* кодом, а затем выполняются браузером на компьютере пользователя (естественно, для этого браузер использует виртуальную *Java*-машину). Они могут использоваться для создания богатых графикой и интерактивными возможностями пользовательских интерфейсов, которые не способны выразить средствами обычного языка разметки *HTML*. Важно однако понимать, что апплет – это интеллектуальная программа, а не просто мультипликация (как, например, *Flash* анимация). Другими словами, апплет способен обрабатывать действия пользователя и динамически менять свое поведение.

При работе с программами, полученными из сети, пользователь может столкнуться с неприятными последствиями их работы. Существует множество вирусов, «троянских коней» или просто некачественных программ. Апплет автоматически запускается при загрузке *web*-страницы, поэтому апплеты требуют повышенного режима безопасности. Для обеспечения защиты, создателями *Java* был разработан механизм, получивший название «песочницы» (*sandbox*), ограничивает доступ «ненадежных» апплетов к компьютеру

пользователя. Если разработчику апплета понадобилось расширить возможности апплета – ему необходимо поставить цифровую подпись, тогда апплет воспринимается броузером как «надежный», и вы сами решаете, доверять апплету или нет. Хотя цифровая подпись не обеспечивает вашей безопасности, вы можете установить происхождение апплета, при возникновении проблем. «Песочница» включает в себя три основных механизма защиты:

- проверки на уровне *JVM*;
- защита на уровне языка;
- интерфейс *JavaSecurity*.

Апплеты могли бы быть почти идеальным со всех точек зрения решением для создателей динамических *Web*-сайтов и корпоративных *Web*-систем: они не требуют затрат на установку, соответствуют лозунгу сторонников чистого *HTML* («написано однажды – работает везде») и имеют собственный богатый графический пользовательский интерфейс. Но до сих пор эти надежды не сбылись. Апплеты, в общем, используются сравнительно редко. Возможно, потому, что некоторые разработчики неверно оценили накладные расходы при интерпретации байт-кода в виртуальной машине *Java*. У других множество нареканий вызывает защита, основанная на принципе «песочницы» (sandbox), который не позволяет *Java* использовать в полной мере локальные и удаленные службы. Третьи отмечают различия между виртуальными машинами основных браузеров, имеющихся на рынке. Так или иначе до сих апплеты не оправдали возложенных на них ожиданий, и *Web*-приложения на базе HTML не были вытеснены *Web*-приложениями с равным уровнем переносимости и мобильности, но функционально более мощным графическим пользовательским интерфейсом.

Тем не менее, при помощи апплетов можно сделать немало полезного. Вот несколько ярких примеров.

- *AnywareOffice* компании *VistaSource* (http://www.anywareoffice.com/). *VistaSource* использует апплет *Java* для реализации *Applixware*, своего популярного офисного пакета, в браузерах, ориентированных на *Java*. Когда провайдер услуг доступа к приложениям использует *AnywareOffice*, приложения (такие, как текстовый процессор) работают на сервере, но отображаются в апплете.
- *QuestAgent* компании *JObjects* (http://www.jobjects.com/). Этот апплет представляет собой кроссплатформенный механизм поиска, часто включаемый в состав компакт-диска с публикациями на базе *HTML*. Браузер может отображать информационное наполнение таких публикаций, но не

может выполнять поиск в своем индексе. *QuestAgent* предлагает мобильный поиск и позволяет отказаться от необходимости создавать и отображать оригинальный механизм поиска.

- *MindTerm* компании *Mindbright Technologies* (http://www.mindbright.com/). Предположим, что пользователь оказался вне офиса, и при нем нет мобильного компьютера, а ему необходимо передать файл на домашний сервер. *MindTerm* – реализация защищенной версии интерпретатора команд *Secure Shell* (*SSH*) на базе *Java* позволяет преобразовать любой ориентированный на *Java* браузер в клиент *SSH*, который можно применять для шифрования сеансов передачи файла.

1.5.4. Сервлеты

Сервлеты – это программы на *Java*, которые работают *на серверном компьютере*. Их выполнение инициируется *Web*-сервером или сервером приложений (*Application Server*) по запросу клиента. Последовательность выполнения сервлета следующая.

- Клиент посылает запрос *Web*-серверу или серверу приложений.
- *Web*-сервер или сервер приложений инициирует выполнение сервлета, передавая ему необходимые данные.
- Сервлет выполняется (как правило, на виртуальной *Java*-машине сервера), и по окончании работы передает результаты (ответ на исходный запрос) вызвавшему его серверу. При этом сервлет может производить сколь угодно сложные действия, например, обращаться к другим сервлетам или удаленным программам и т.п. Обмен данными между сервлетом сервлетом и сервером происходит при помощи специального *Java-API* (его главные составляющие это классы *HttpServletRequest* для передачи запроса и *HttpServletResponse* для ответа).
- Сервер отдает полученные от сервлета данные клиенту.

На самом деле схема, как правило, чуть более сложная. В связке с сервером работает базовый сервлет. Именно ему сервер отправляет данные и от него же получает ответ, отправляемый клиенту. Фактически, базовый сервлет является «мозгом» сервера. Основная функция этого сервлета — прочитать запрос клиента, расшифровать его

и, в соответствии с расшифровкой, передать работу сервлету, отвечающему за конкретный тип запрашиваемой информации. Зачастую, для достижения скорости, роль базового сервлета играет сам сервер. Именно по такой схеме работает, скажем, *Web*-сервер *Jakarta Tomcat*.

По сути дела, в этой схеме нет ничего принципиально нового по сравнению с *CGI*-скриптами, кроме, конечно, большей унифицированности и существенно меньшей зависимости от платформ (благодаря использованию *Java*). Действительно, сервлеты и были разработаны, чтобы заменить *CGI*-скрипты.

Среда исполнения сервлетов, кроме того, обеспечивает некоторые полезные и экономящие время возможности, включая преобразование *HTTP*-запросов из сети в удобный для использования *HttpServletRequest* объект, обеспечивая выходной поток для программиста, чтобы использовать его для ответа, и преобразование удобного в работе объекта *HttpServletResponse* в *HTTP*-ответ, который может быть послан обратно по сети. Она также обеспечивает удобные возможности управления сессиями, в том числе хранения состояния сессии, что позволяет, например, назначать ресурсы (такие как подключения к базе данных), которые могут использоваться для многократных запросов.

По сравнению с апплетами сервлеты имеют преимущества с архитектурной точки зрения. Если апплет, посланный по сети, окажется в несовместимой с ним виртуальной машине *Java*, то он, скорее всего, корректно работать не будет. Сервлет развертывается в более управляемой среде. Так как параметры *JVM* известны, проблем совместимости не возникает. Более того, среда, которая окружает данную виртуальную машину, может увеличивать производительность сервлета. Некоторые серверы *Java*-приложений могут компилировать сервлеты в «родной» для себя код и тем самым значительно увеличивать скорость выполнения. Другие серверы запускают параллельно несколько *JVM*, иногда в различных процессах хостовой ОС. Эти стратегии увеличивают масштабируемость и отказоустойчивость службы.

Вариантом сервлета является *JSP*-страница (*Java Server Pages*). *JSP*-страница, подобно *ASP* или *PHP* скрипту представляет собой обычный *HTML* файл с записанным внутри него при помощи специального синтаксиса исходным *Java*-кодом сервлета. Каждая *JSP*-страница автоматически преобразуется в сервлет *Web*-сервером при запросе на эту страницу со стороны клиента. Затем сервлет выполняется по описанной схеме.

Подытоживая, можно сказать, что сервлеты имеют преимущество максимальной переносимости: они могут работать на большем количестве *Web*-серверов или серверов приложений и на большем количестве платформ, чем любая другая технология динамических *Web*-приложений, доступная сегодня. Следует также отметить, что *API*

сервлета намного проще в изучении и в использовании, чем технология *EJB*, поэтому и применяется пока что чаще (хотя *EJB* также уже стал фактически промышленной технологией).

1.6. Заключение

В современном развитии программных и аппаратных платформ прослеживаются две отчетливые тенденции:

- программные и аппаратные платформы становятся все более и более совместимыми друг с другом, границы между ними становятся легко преодолимыми;
- все время появляются новые технологии, которые предлагают разные способы решения одних и тех же задач.

Таким образом, одна и та же задача по разработке программного продукта может быть решена множеством разных способов, и менеджер проекта по разработке программного продукта должен уметь выбирать платформы и технологии, исходя из особенностей задачи и конкретных условий.

Тема 2. Технология *CORBA*

2.1. Введение

CORBA (*Common Object Request Broker Architecture*) – объектно-ориентированная технология создания распределенных приложений. Технология основана на использовании *брокера объектных запросов* (*Object Request Broker*, *ORB*) для прозрачной отправки и получения объектами запросов в распределенном окружении. Технология позволяет строить приложения из распределенных объектов, реализованных на различных языках программирования. Стандарт *CORBA* разработан *Object Management Group* (*OMG*).

Изучив учебный материал данного раздела, Вы:

- узнаете или пополните свои знания об основах *CORBA*-технологии;
- приобретете умения в области разработки *CORBA*-систем.

В рамках темы рассматриваются следующие учебные вопросы:

- основы *CORBA* технологии и ее архитектура;
- основы языка *IDL*
- основы создания *CORBA*-систем

2.2. Основы технологии *CORBA*

2.2.1. Архитектура *CORBA*

В данном разделе приводится краткий обзор *CORBA* в том виде, как ее описание дается в спецификации *OMG* версии 3.0. Требования этого документа могут в различной степени удовлетворяться фактическими реализациями брокеров объектных запросов.

На рис. 1 изображен запрос, посылаемый клиентом реализации объекта. *Клиент* – это сущность, которая хочет выполнить операцию с объектом, а *Реализация* – это совокупность кода и данных, которые в действительности реализуют объект.

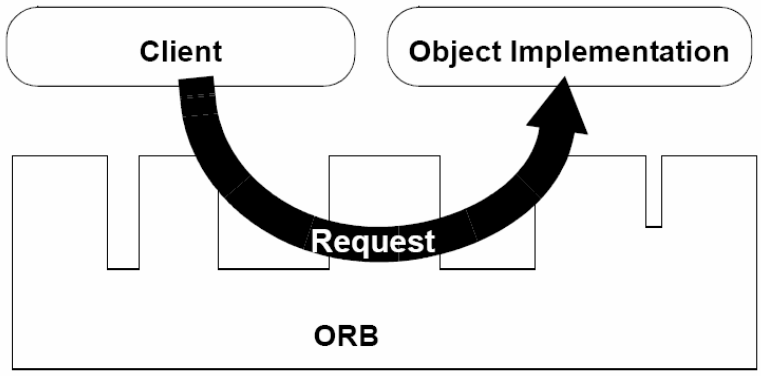


Рис. 1 Клиент посылает запрос реализации объекта

ORB отвечает за все механизмы, необходимые для поиска подходящей для запроса реализации объекта, подготовки реализации к получению запроса и передачи данных в процессе выполнения запроса. Интерфейс, видимый клиенту, совершенно независим от расположения реализации объекта, языка программирования, на котором она написана и любых других аспектов, не отраженных в спецификации интерфейса.

На рис. 2 изображена структура *брокера объектных запросов*. Его интерфейсы показаны на рисунке штрихованными прямоугольниками, стрелки обозначают, вызывается ли брокер или сам выполняет вызов.

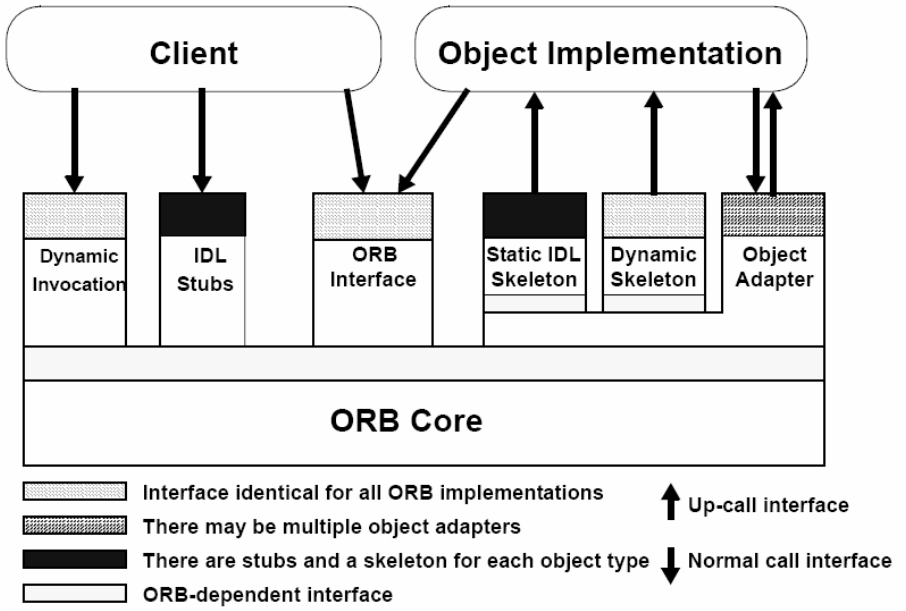


Рис. 2 Интерфейсы брокера объектных запросов

Чтобы сделать запрос, Клиент может использовать *Динамический интерфейс вызова* (*Dynamic Invocation Interface*), один и тот же, вне зависимости от интерфейса целевого объекта, или *IDL заглушку* (*stub*), специфичную для интерфейса целевого объекта. Клиент также может напрямую взаимодействовать с брокером для получения некоторых функций.

Реализация объекта получает запрос как вызов либо через автоматически сгенерированный *IDL скелетон*, либо через *динамический скелетон*. Реализация объекта может вызывать объектный адаптер или *ORB* во время выполнения запроса или в другой время.

Интерфейсы объектов могут быть описаны двумя способами. Во-первых, статически, на языке описания интерфейсов *IDL*. Этот язык позволяет описывать типы объектов через предоставляемые ими операции и их параметры. Во-вторых, интерфейсы могут быть добавлены в *Репозиторий Интерфейсов*. Это специальный сервис, представляющий компоненты интерфейсов как объекты и предоставляющий доступ к этим компонентам во время выполнения.

Для выполнения запроса клиент должен иметь доступ к *объектной ссылке* (*IOR – Interoperable Object Reference*), знать тип объекта и ту операцию, которую он хочет выполнить. Клиент инициирует запрос, вызывая подпрограммы заглушки, специфичные для конкретного объекта, или создавая запрос динамически (рис. 3).

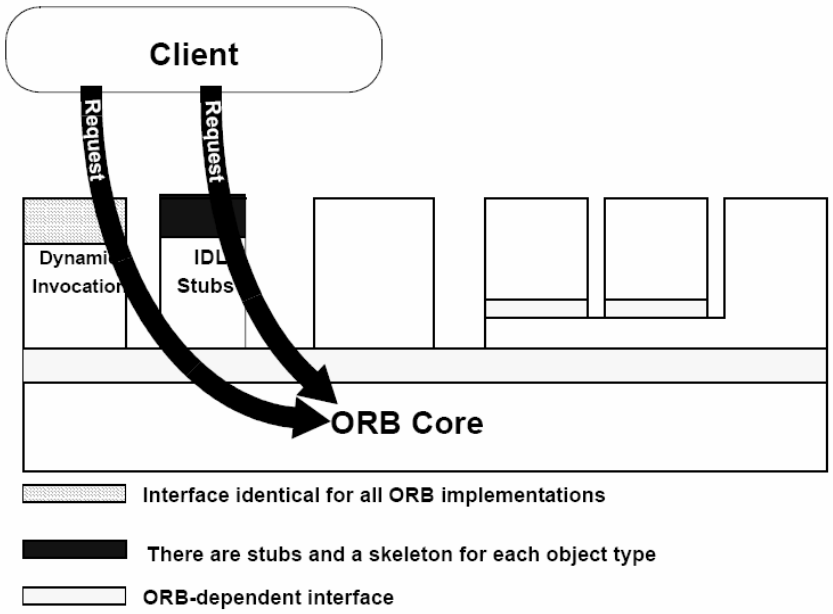


Рис. 3 Клиент выполняет запрос (динамически или через заглушку)

Динамический интерфейс вызова и интерфейс заглушки имеет одинаковую семантику, так что получатель сообщения не может определить, как был послан запрос. *ORB* находит подходящий код реализации объекта, пересылает ему параметры и отдает управление через *IDL* скелетон или динамический скелетон (рис. 4). Скелетоны специфичны для конкретного интерфейса и объектного адаптера. Во время выполнения запроса реализация может пользоваться некоторыми сервисами *ORB* через объектный адаптер. Когда запрос выполнен, управление и значения результата возвращаются клиенту.

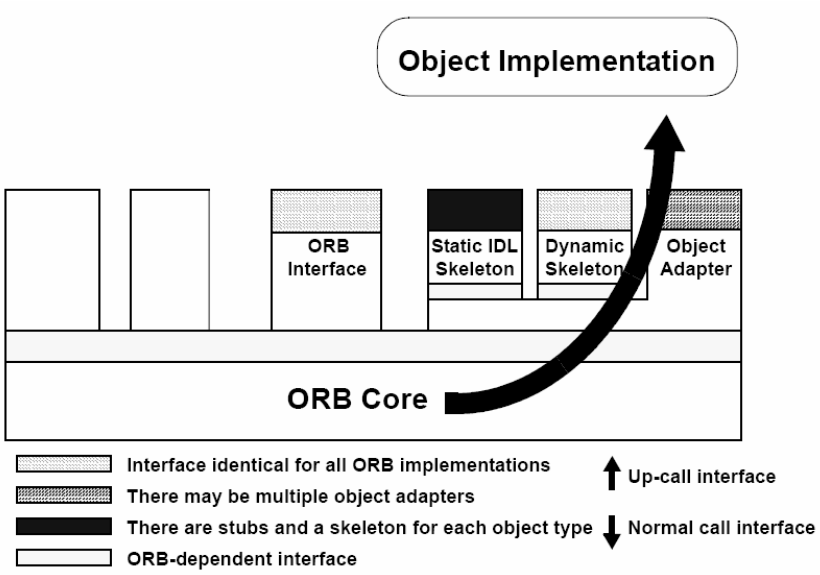


Рис. 4 Реализация объекта получает запрос

Реализация объекта может выбрать, какой объектный адаптер использовать, в зависимости от того, в каких сервисах она нуждается. На рис. 5 показано, как информация об интерфейсе и реализации становится доступной клиентам и реализациям объектов. Интерфейсы описываются на *IDL* или с помощью репозитория интерфейсов. Их описания используются для генерации клиентских заглушек и скелетонов для реализации.

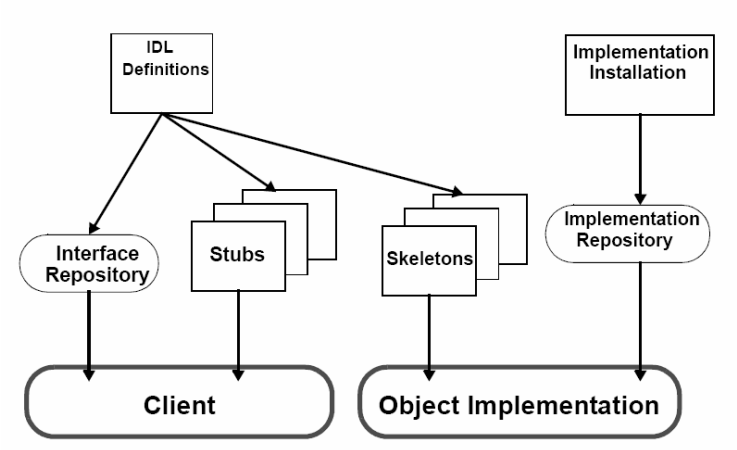


Рис. 5 Репозитории интерфейсов и реализаций

Информация о реализации объекта предоставляется во время инсталляции и хранится в репозитории реализации, а затем используется в процессе доставки запроса.

Брокер объектных запросов (ORB)

Вместе с *IDL*-компилятором, репозиторями и различными объектными адаптерами, *ORB* предоставляет полный набор сервисов самым разным клиентам и объектным реализациям. Ядро *ORB* – это

часть брокера, обеспечивающая базовое представление объектов и передачу запросов. Технология *CORBA* может поддерживать различные объектные механизмы за счет компонентов, надстроенных над ядром и предоставляющих интерфейсы, которые позволяют скрыть различия между разными ядрами.

Клиенты

Клиент объекта имеет доступ к объектной ссылке и вызывает операции объекта. Клиент знает только логическую структуру объекта в соответствии с его интерфейсом и может наблюдать за поведением объекта через вызовы методов. Несмотря на то, что клиентом обычно считается программа или процесс, важно помнить, что понятие клиент может применяться только относительно определенного объекта. Например, реализация одного объекта может быть клиентом другого.

Клиенты обычно видят брокер через призму связывания с языком программирования. Клиенты являются максимально переносимыми и могут работать без изменения исходного кода с любым брокером, поддерживающим связывания с данным языком программирования и любым экземпляром объекта, реализующим данный интерфейс. Клиенты ничего не знают о реализации объекта, используемом ей объектном адаптере, а также о брокере, который осуществляет доступ к реализации.

Реализации объектов (Object implementation)

Реализация предоставляет семантику объекта, определяя данные, хранимые в экземпляре объекта, и код его методов. Реализация может использовать другие объекты или программное обеспечения для выполнения функций объекта. В некоторых случаях, главное предназначение объекта – иметь побочные эффекты на что-то, не являющееся объектом. *ORB* может поддерживать множество видов реализаций объектов, включая разделенные серверы, библиотеки, «программа на метод», «инкапсулированное приложение», объектно-ориентированные базы данных и т.д. Используя дополнительные объектные адаптеры можно осуществлять поддержу абсолютно любого стиля реализации объектов.

Вообще, реализации объектов не зависят от *ORB* и от того, как клиент вызывает объект. Реализации могут выбирать интерфейс брокера, меняя объектный адаптер.

Объектные ссылки (IOR)

Объектная ссылка – это информация, необходимая для определения конкретного объекта внутри *ORB*. Как для клиента, так и для реализации объектная ссылка представляется так, как диктует

связывание соответствующего языка программирования, таким образом, они изолированы от конкретного представления ссылки.

Объектная ссылка, переданная клиенту, действительна только на время жизни клиента.

Различные брокеры должны предоставлять одно и то же представление объектной ссылки для данного языка программирования (это позволяет одной и той же программе получать доступ к объекту по ссылке независимо от используемого брокера). Кроме того, для удобства брокер может предоставлять другие способы доступа к объекту.

Существует особенная объектная ссылка, не указывающая ни на один объект.

Язык описания интерфейсов (*IDL*)

Язык *IDL* определяет типы объектов путем спецификации их интерфейсов. Интерфейс состоит из списка операций и их параметров. Несмотря на то, что *IDL* предоставляет каркас для описания объектов, которыми манипулирует *ORB*, нет необходимости в том, что брокер имел доступ к исходному коду на *IDL*. Брокер может работать с эквивалентной информацией в виде заглушек подпрограмм и репозитория интерфейсов.

IDL является средством, с помощью которого реализация объекта сообщает своим потенциальным клиентам, какие операции доступны и как они могут быть вызваны. Из *IDL*-описания *CORBA*-объект можно перевести на определенный язык программирования или в другую объектную систему.

Связывание языков программирования с *IDL*

Различные объектно-ориентированные и не объектно-ориентированные языки программирования могут получать доступ к *CORBA*-объектам по-разному. Для объектно-ориентированных языков, скорее всего, предпочтительно видеть *CORBA*-объекты как объекты языка программирования. И даже для не объектно-ориентированных языков скрытие фактического представления объектных ссылок и методов внутри брокера представляется удобным. Связывание того или иного языка программирования с *IDL* должно быть одинаковым для всех реализаций *ORB*. Связывание языка включает определение специфичных для языка типов данных и интерфейсов процедур для доступа к объекту через *ORB*. Оно включает структуру интерфейса клиентской заглушки (для объектно-ориентированных языков не обязательно), интерфейс динамического вызова, скелетон реализации, объектные адаптеры и интерфейс для обращения напрямую к брокеру.

Связывание также определяет взаимодействие между вызовами объекта и потоками выполнения в клиенте и реализации. Самые

распространенные связывания предоставляют синхронные вызовы, когда управление возвращается клиенту после завершения операции. Дополнительные связывания могут возвращать управления программе сразу после инициации вызова. В этом случае должны предоставляться дополнительные подпрограммы, зависящие от языка, осуществляющие синхронизацию потоков программы и вызова объекта.

Клиентские заглушки (client stubs)

Обычно клиентские заглушки предоставляют доступ к операциям объекта, описанным на *IDL*, способом, ожидаемым для программиста, знакомого с *IDL* и связыванием конкретного языка программирования. Заглушки вызывают функции остальной части *ORB*, используя закрытые интерфейсы, которые могут быть оптимизированы для использования с конкретной реализацией ядра брокера.

Динамический интерфейс вызова (Dynamic invocation)

Также доступен интерфейс, позволяющий создавать вызовы объекта динамически, то есть, вместо того, чтобы вызывать подпрограмму заглушки, специфичную для конкретного объекта, клиент может определить объект, который требуется вызвать, операцию, которую требуется выполнить, и набор параметров путем вызова (или последовательности вызовов) универсальной функции. Клиентский код должен предоставить информацию об операции, которую требуется выполнить, включая типы передаваемых параметров (их можно получить из репозитория интерфейсов или другого источника времени выполнения). Природа динамического интерфейса вызова может значительно различаться в зависимости от связывания.

Скелетон реализации (Server skeleton)

Для каждого конкретного связывания языка программирования и, возможно, в зависимости от конкретного объектного адаптера, будет создан определенный интерфейс к методам, реализующим некоторый тип объектов. При этом реализация объекта предоставляет подпрограммы, удовлетворяющие интерфейсу, а *ORB* вызывает эти подпрограммы через скелетон.

Из существования скелетона не следует существование соответствующей клиентской заглушки: клиент может делать запросы и через динамический интерфейс вызова. Для некоторых объектных адаптеров скелетоны могут быть не нужны: например, в таких языках как *Smalltalk* есть возможность создавать реализации динамически.

Динамический интерфейс скелетона

Также доступен интерфейс, позволяющий управлять вызовами объектов динамически. Вместо того, чтобы обращаться к реализации объекта через скелетон, специфичный для определенной операции, можно обратиться к реализации через интерфейс, предоставляющий доступ к имени операции и ее параметрам так же, как клиентский динамический интерфейс вызова. Для определения параметров может быть использована как чисто статическая, так и динамическая (например, предоставленная репозиторием интерфейсов) информация.

Реализация должна предоставить брокеру информацию обо всех параметрах операции, брокер, в свою очередь, предоставляет значения входных параметров операции. По завершении операции, код реализации предоставляет брокеру значения всех выходных параметров или исключения.

Динамические скелетоны могут быть вызваны как клиентскими заглушками, так и динамическим интерфейсом вызова на стороне клиента, причем результат должен быть одинаковым.

Объектные адаптеры

Объектный адаптер предоставляет основной способ доступа к сервисам брокера со стороны реализации объекта. Предполагается, что будут существовать несколько общедоступных объектных адаптеров, интерфейсы которых подходят для определенных видов объектов.

Сервисы, предоставляемые брокером через объектный адаптер, включают генерацию и интерпретацию объектных ссылок, вызов методов, безопасность взаимодействий, *активацию* и *деактивацию* объектов и их реализаций, сопоставление объектных ссылок реализациям и регистрацию реализаций.

Широкий диапазон уровней модульности, времен жизни, политик, стилей реализации и других свойств объектов делает невозможным предоставление ядром брокера единого интерфейса, удобного и эффективного для всех объектов. С помощью объектных адаптеров брокер может выделять группы реализаций объектов, имеющие схожие требования, и предоставлять интерфейсы, предназначенные для этих групп.

Интерфейс *ORB*

Это интерфейс, позволяющий обращаться напрямую к брокеру объектных запросов, он одинаков для всех брокеров и не зависит ни от интерфейса объекта, ни от объектного адаптера. Поскольку основная функциональность брокера предоставляются через объектный адаптер, заглушки, скелетоны или динамический вызов, только несколько

операций могут запрашиваться напрямую. Эти операции полезны как клиентам, так и реализациям объектов.

Репозиторий интерфейсов

Репозиторий интерфейсов – это сервис, предоставляющий устойчивые объекты, отражающие *IDL*-информацию в форме, доступной во время выполнения. Информация из репозитория интерфейсов может быть использована брокером для осуществления запросов. Более того, используя информацию из репозитория, программа может найти объект, интерфейс которого был неизвестен во время компиляции программы, и, тем не менее, определить, какие операции могут выполняться объектом, и вызвать эти операции.

В дополнение к этой роли, репозиторий интерфейсов используется также для хранения дополнительной информации, связанной с интерфейсами объектов брокера. Например, отладочной информации, библиотек заглушек и скелетонов, и т.д.

Репозиторий реализаций

Репозиторий реализаций содержит информацию, которая позволяет брокеру находить и активировать реализации объектов. Большая часть информации в репозитории специфична для конкретного *ORB* и рабочей среды. Обычно, инсталляция реализаций и управление политиками, связанными с активацией и исполнением реализаций выполняется через операции с репозиторием реализаций.

Репозиторий реализаций используется также для хранения дополнительной информации, связанной с реализациями объектов (отладочная информация, административный контроль, выделение ресурсов, безопасность и т.д.).

2.2.2. Язык *IDL*

Появление в программировании того или иного языка обычно связано с возникновением и развитием некоей новой концепции. Так, одновременно с идеей объектно-ориентированной разработки (в ее современном варианте) был создан язык *Smalltalk*. Дальнейшее применение объектов и компонентов вкупе с внедрением последних в распределенные системы вызвало необходимость создания такого языка программирования, который бы позволил описать любой объект или компонент. И, что не менее важно, это описание должно быть одинаковым для любой платформы. Этим требованиям удовлетворяет язык описания интерфейсов *IDL* (*Interface Definition Language*).

Следует заметить, что *IDL* не только язык, но и инструмент, с помощью которого можно сохранять метаинформацию об объектах, т. е. данные о том, как устроен объект. Известно довольно много случаев,

когда язык *IDL* используется для описания контрактов — технических параметров, позволяющих нескольким независимым группам работать одновременно над различными частями проекта. Хотя в основном исходные тексты на *IDL* служат своеобразным «сырьем», из которого специальные компиляторы *IDL* генерируют исходные тексты на одном из языков программирования высокого уровня. Такой процесс будем называть *трансляцией*. Типичный процесс создания распределенных объектных приложений состоит из описания объектов на *IDL*, их трансляции на какой-либо язык программирования, реализации объектов на данном языке и компиляции полученных исходных текстов в готовые для запуска модули.

Приведем список терминов, используемых далее при описании языка *IDL*:

- *модуль* – блок с заданным именем, объединяющий логически связанные конструкции языка *IDL*; в целом модуль можно воспринимать как пакет (package) в языке *Java* или пространство имен (namespace) в языке *C++*;
- *интерфейс* – набор атрибутов и операций объекта, с помощью которого потребитель может обращаться к объекту;
- *операция* – сущность, которую вызывают для выполнения действий, связанных с функциональным назначением объекта; операцию можно сравнить с методом класса.
- *атрибут* – сущность, описывающая какое-либо свойство объекта; атрибут эквивалентен паре операций, предназначенных для чтения и записи свойства класса

2.2.3. Синтаксис *IDL*

Приступим к описанию элементов и конструкций языка.

Комментарии

Комментарии *IDL* — точная копия комментариев языка *C++*.

Идентификаторы

К идентификаторам *IDL* относятся имена интерфейсов, модулей, атрибутов, операций, констант и т. д. Идентификатор может содержать латинские буквы, цифры, а также знак подчеркивания ‘_’. Все идентификаторы в *IDL*-файлах должны начинаться с буквенного символа. Регистр букв в идентификаторах не различается (это сделано для того, чтобы не возникало лишних проблем при трансляции *IDL*

описания на язык программирования, нечувствительный к регистру). Символ подчеркивания ‘_’ не может быть первым в имени идентификатора (однако очень часто при трансляции *IDL* компилятор сам добавляет перед получаемыми идентификаторами символы подчеркивания, чтобы избежать конфликтов имен).

Ключевые слова

К регистру букв в ключевых словах, в отличие от идентификаторов, компиляторы *IDL* чувствительны. Ниже приведена таблица ключевых слов *IDL*.

Таблица 1. Ключевые слова IDL

any	double	interface	readonly	unsigned
attribute	enum	long	sequence	union
boolean	exception	module	short	void
case	FALSE	Object	string	wchar
char	fixed	octet	struct	wstring
const	float	oneway	switch	
context	In	out	TRUE	
default	inout	raises	typedef	

Литералы

Литералы языка *IDL* подразделяются на булевы, «узкие» и «широкие» символьные, целочисленные, с плавающей точкой, с фиксированной точкой и строковые. Все эти виды литералов, кроме чисел с фиксированной точкой, аналогичны литералам *C/C++*, поэтому описывать их подробно нет необходимости.

Таблица 2 Литералы IDL

Вид литерала	**Примеры значений**
Булев	TRUE, FALSE
Символьный	‘p’, ‘\t’, ‘010’, ‘x1A’
Целочисленный	99, 013, 0xFFFF, 0XFFFF
С плавающей точкой	0.15, 1234E+13, 0.987e-150
С фиксированной точкой	d0.15, D36.28
Строковый	“hello, world”

Когда одного байта для хранения символа не хватает, применяются так называемые «широкие» символы, содержащие более 8 бит. Таблицы этих символов могут быть разными на разных платформах. Поэтому, задавая литералы с «широкими» символами, следует не выходить за рамки таблицы *ISO Latin-1* (8859-1).

Для финансовых вычислений в *IDL* предусмотрены литералы с фиксированной точкой, состоящие из целой и дробной частей, разделенных десятичной точкой и отмеченных буквой *d* или *D*. Такого рода литералы будут в дальнейшем применяться вместе с типом *fixed*. Однако, несмотря на то, что эти элементы языка описаны в спецификации *CORBA 2.2*, найти их реализацию в компиляторах *IDL* не удалось.

Строковые литералы должны состоять из символов, допустимых в качестве символьных литералов за исключением символа *'\0'*. И хотя компиляторы корректно обрабатывают эти символы, тем не менее, ясно, что при трансляции *IDL* на *C* и *C++* наличие нулевого символа в строке может послужить источником ошибки. Строковые литералы, как и символьные, могут быть основаны на простых и «широких» символах.

Препроцессинг

Для организации условной компиляции и задания некоторых опций компиляторы *IDL* используют препроцессинг, основанный на директивах, описанных в стандарте *ANSI C++*. Однако, в препроцессинге *IDL* присутствуют специальные разновидности директивы *#pragma.*

Область видимости имен

Любое имя *IDL* видно в том блоке, где оно описано. В качестве подобного блока могут выступать описания модулей, интерфейсов, составных типов. Для явного указания блока, в котором содержится используемое имя, применяется пара символов :: (оператор доступа к области видимости из языка *C++*).

Простые типы

Простые типы служат для задания атрибутов объектов, параметров их операций, констант и в описаниях конструируемых типов.

Булев тип *boolean* отвечает за хранение логических значений *TRUE* и *FALSE*.

Символьные типы могут описывать обычные 8-битовые символьные данные (тип *char*) или «широкие» символьные данные (тип *wchar*), размер которых более 8 бит. В процессе передачи символьных данных по сети они могут быть переконвертированы так, что изменится их представление, но значение символа будет сохранено. Такая ситуация может возникнуть при передаче данных между компьютерами с отличающимися кодировками.

Целочисленные типы — это *short*, *long*, *long long*, а также их беззнаковые разновидности с префиксом *unsigned*. Обратите внимание, что тип *int* в *IDL* отсутствует.

Таблица 3. Диапазоны допустимых хранимых значений для целочисленных типов

short	от -2^{15} до $2^{15} - 1$
long	от 2^{31} до $2^{31} - 1$
long long	от -2^{63} до $2^{63} - 1$
unsigned short	от 0 до $2^{16} - 1$
unsigned long	от 0 до $2^{32} - 1$
unsigned long long	от 0 до $2^{64} - 1$

К типам чисел с плавающей точкой относятся *float*, *double* и *long double*. Тип *float* соответствует *IEEE*-числу с плавающей точкой одинарной точности. Тип *double* — *IEEE*-числу с плавающей точкой двойной точности. И последний тип, *long double*, используется для описания *IEEE*-числа с плавающей точкой двойной расширенной точности. Более полные данные о числах с плавающей точкой *IEEE* можно узнать из документа «*IEEE Standard for Binary Floating-Point Arithmetic*», *ANSI/IEEE Standard 754-1985*.

В *IDL* есть два специальных простых типа: *octet* и *any*. Первый служит для передачи по коммуникационным системам 8-битовых чисел так, чтобы они в процессе пересылки не подверглись изменениям. Тип *octet* — хорошая альтернатива типу *char* в тех случаях, когда передается маленькое число.

В объектах типа *any* могут храниться значения любого типа, позволенного *IDL*, *any* можно сравнить с универсальным типом *void** в языке *C* или с классом *java.lang.Object* в языке *Java*.

Константы

Описание констант начинается с ключевого слова *const*, за которым указываются: тип константы, ее имя, символ присвоения = и константное выражение, состоящее из литералов и математических операторов. Константа может иметь любой из простых типов. Математические операторы *IDL* соответствуют аналогичным операторам *C/C++*.

Конструируемые типы

Для начала отметим, что ключевое слово *typedef* используется в *IDL* так же, как и в *C/C++*: для присвоения синонима имени типа.

К конструируемым типам *IDL* относятся перечислимые типы, дискриминируемые объединения и структуры.

Перечислимые типы

Перечислимые типы знакомы большинству программистов. Общая схема их описания:

```
enum < Имя энумератора > {< Список элементов >};
```

В качестве списка элементов выступают идентификаторы, разделенные запятыми:

```
enum Semaphore {red, yellow, green};
```

Дискриминируемые объединения

В *IDL* объединения обладают дискриминатором – элементом, определяющим, какой член объединения использовать в том или ином случае. Таким образом, объект, полученный в результате компиляции дискриминируемого объединения, способен в разное время хранить значения разных типов. Главное, чтобы во время описания объединения с помощью *IDL* были перечислены все возможные варианты хранимых типов. Типичное описание дискриминируемого объединения выглядит следующим образом:

```
union < Имя объединения > switch
 (< Тип дискриминатора >)
{
   < Список элементов выбора >
};
```

Здесь <Тип дискриминатора> может быть любым интегральным типом (символьным, целочисленным, булевым или перечислимым).

Со списком элементов выбора дело обстоит несколько сложнее. Каждый элемент состоит из ключевого слова *case* и следующего за ним константного выражения, после которого ставится двоеточие и производится собственно описание хранимого типа. Константное выражение должно возвращать тип, совпадающий с типом дискриминатора. При записи в объединение некоторого значения объединение принимает тип, совпадающий с типом сохраняемого значения. Заодно запоминается дискриминатор. В дальнейшем, если будет произведена попытка считать значение под типом, отличающимся от того, под которым это значение было сохранено, произойдет генерация исключения *org.omg.CORBA.BAD_OPERATION*. Однако для программиста все-таки предусмотрено некоторое облегчение: объединение обладает значением по умолчанию, которое на языке *IDL* описывается ключевым словом *default*. Объединение может содержать только один такой элемент.

Рассмотрим короткий пример описания дискриминируемого объединения:

```
union MyType switch (short)
{
  case 13: short alpha;
```

```
    case 0x0C << 3: long beta;
    default: arr alphabet;
};
```

В этом случае тип *MyType* имеет дискриминатор типа *short*. Числа, стоящие после *case*, представляют собой значения дискриминатора. Именно от них зависит, какой член объединения будет использован. Если значение дискриминанта равняется *13*, то под именем *alpha* будет храниться значение *short*. При дискриминанте *0X0C << 3* (конечное значение этого константного выражения – *96*) хранимое значение будет иметь тип *long* и носить имя *beta*. И наконец, по умолчанию значение, хранящееся в объединении, будет иметь тип *arr* и к нему можно обращаться по имени *alphabet*.

Структуры

Структуры служат для определения сложных типов, призванных хранить наборы разнородных данных. Типичная структура описывается следующим образом (аналогично структурам *C/C++*):

```
struct < Имя структуры >
{
   < Список членов >
};
```

Список членов – это разделенный точкой с запятой набор элементов, являющихся комбинацией из имени типа и идентификатора. Имя типа может быть любым типом *IDL* (допускается использование массивов с обязательным указанием числа элементов).

Шаблонные типы

К шаблонным типам можно отнести строки («узкие» и «широкие»), последовательности и числа с фиксированной точкой.

Строки.

Строки в *IDL* – это 8-битовые «узкие» *string* и «широкие» *wstring*. Первые могут содержать любые символы типа *char* за исключением *'\0'*. Второй тип строки состоит из символов, подпадающих под базовый тип *wchar*, и заканчивается «широким» нулевым символом. Длина строки может быть как ограниченной, так и неограниченной. Ограниченные (*bounded*) строки можно сравнить с символьным массивом заданной длины.

Типичное описание подобного типа: `typedef wstring <28> boundedString;` В угловых скобках задается размер строки в символах. Неограниченные же (*unbounded*) строки содержат в себе символов столько, сколько потребуется:

```
typedef string unboundedString;
```

Описание строчных типов должно предваряться ключевым словом *typedef*.

Последовательности.

Последовательности во многом схожи со строками. Различие между ними состоит в том, что последовательности помимо символьных данных могут хранить данные и других типов. Точно так же, как и строки, последовательности могут быть ограниченными по размеру либо неограниченными. Правда, в описании последовательности всегда присутствуют угловые скобки, в которых пишется тип хранимых данных (для неограниченных последовательностей) или тип хранимых данных с размером последовательности, разделенные запятой. Первый тип последовательности может выглядеть так:

```
typedef sequence unboundedSequence;
```

А вот пример описания типа ограниченной последовательности:

```
typedef sequence boundedSequence;
```

Последовательности могут быть рекурсивно вложены друг в друга:

```
typedef sequence< sequence > recursedSequence;
```

Числа с фиксированной точкой.

Тип данных *fixed* представляет собой десятичное число с фиксированной точкой, имеющее до 31 значащей цифры. К сожалению, на данный момент, тип *fixed* только описан в спецификации *CORBA*, но не реализован.

Прочие типы

К оставшимся типам относятся массивы и *native*-типы. Массивы описываются ключевым словом *typedef*, за которым следуют тип, идентификатор и размерность массива. Допускаются многомерные массивы:

```
typedef double someArray[100][100];
```

Тип *native* служит для введения новых типов данных, которые реализованы на языке программирования, отличном от *IDL*. Следующая строка исходного текста на IDL: `native NonIDLType;` говорит компилятору, что где-то имеется тип *NonIDLType*, реализованный непонятным для него образом, но, тем не менее, к нему нужно сделать определенный интерфейс.

Исключения

Описание типов-исключений практически полностью совпадает с описанием структур. Минимальное различие состоит в замене ключевого слова *struct* на *exception*:

```
exception < Имя исключения >
{
< Список членов >
};
```

Интерфейсы

Интерфейс представляет собой набор операций и атрибутов для обращения к объекту. Интерфейсы могут наследоваться от других интерфейсов, причем наследование может быть множественным. В дополнение к операциям и атрибутам в интерфейсах *IDL* могут быть описаны константы и исключения.

Следует различать описания (или опережающие описания) интерфейсов и их определения (или полные описания). Допускаются множественные описания, однако множественные определения являются ошибкой.

Описания требуются в том случае, если необходимо обратиться к интерфейсу еще до его определения. Чтобы получить подобное описание, достаточно написать ключевое слово *interface* и его имя:

```
interface < Имя интерфейса >;
```

При определении после имени добавляются двоеточие и имена интерфейсов-предков:

```
interface < Имя интерфейса > [ :< Имя интерфейса-предка
1 > ... [ ,< Имя интерфейса-предка n >]...]
{
...
     <Описания типов, констант, исключений, атрибутов и
     операций >
...
};
```

Внутри описаний интерфейсов описываются прочие элементы *IDL*, которые допускаются внутри интерфейсов. Новые типы, константы и исключения нам уже известны. Осталось познакомиться с описаниями операций и атрибутов.

Операции

Операции – это единственное средства манипулирования внутренним состоянием объекта. Описываются операции по схеме, показанной ниже:

```
{ oneway void | < Возвращаемый тип > }
< Имя операции > ({ in | out | inout } <Параметр> ...
     [, { in | out | inout } <Параметр>]... )
     [ raises ( < Имя исключения > ...
[, < Имя исключения > ] ... ) ] ;
```

Если определяется операция, возвращающая некоторое значение, то слева от ее имени необходимо написать тип возвращаемого значения. Исключением являются операции, определенные как *oneway*, они должны возвращать тип *void*, т. е. не возвращать никакого значения. Ключевое слово *oneway* говорит, что при вызове операции программа не ждет, пока эта операция завершится, а продолжает выполнение. Параметры операции могут быть входными (*in*), выходными (*out*) или комбинированными (*inout*). Если операция может возбудить исключения, то они должны быть перечислены в скобках через запятую после ключевого слова *raises*.

Атрибуты

Атрибуты можно воспринимать как аналог переменных (полей) класса, однако, это не совсем корректно, так как интерфейс не может иметь состояния. Поэтому более правильно считать атрибут сокращением для пары операций считывания и модификации определенного свойства класса.

В *IDL* атрибуты описываются следующим образом:

```
[readonly] attribute < Тип атрибута > < Имя атрибута >;
```

Необязательный модификатор *readonly* говорит, что значение описываемого атрибута не может быть изменено, только считано.

Модуль

Модуль – самая «старшая» единица языка *IDL*. Он служит для группировки типов, интерфейсов и т. д., логически связанных друг с другом. У модуля есть имя и тело:

```
module < Имя модуля >
{
...
   < Тело модуля >
...
};
```

Можно считать, что модули являются прямым отображением пакетов языка *Java* и пространств имен в *C++*.

2.2.4. Связывание с *IDL*

Спецификация *CORBA* регламентирует, во что должен превратиться каждый элемент языка *IDL* в процессе его трансляции в исходные тексты на языке программирования высокого уровня. Версия 2.2 спецификации *CORBA* расписывает подобные соответствия для *C*, *C++*, *Smalltalk*, *Cobol*, *Ada* и *Java*. Для примера рассмотрим подробнее трансляцию в *Java*.

Комментарии

Комментарии *IDL* никак не отражаются на сгенерированном *Java*-коде.

Имена

Результаты работы различных компиляторов могут различаться. В основном это касается добавления символов подчеркивания перед сгенерированными именами. Поскольку трансляция зависит от того, что за элемент языка *IDL* подвергается обработке, различается и количество файлов, получаемых в процессе генерации. Для пользовательских типов, например, будут созданы специальные файлы, имена которых заканчиваются суффиксами *Helper* и *Holder*. По спецификации, компилятор резервирует за собой следующие имена:

`<тип>Helper`, где `<тип>` — имя пользовательского типа;

`<тип>Holder`, где `<тип>` — имя пользовательского типа;

`<базовыйТипJava>Holder`, где `<базовыйТипJava>` — один из примитивных типов языка *Java*;

`<интерфейс>Package`, где `<интерфейс>` — имя интерфейса *IDL*.

Вспомогательные классы

Если программист описывает собственные типы, в результате их трансляции появляются два вспомогательных класса, имена которых состоят из имени типа с добавлением суффиксов `Helper` и `Holder`. Они

необходимы для корректной работы с объектами. `Helper` содержит набор статических методов, выполняющих одни и те же рутинные действия. Класс с суффиксом `Holder` работает «оболочкой» для пользовательского типа, когда его нужно передать в качестве параметров операции объекта. Его генерация происходит не во всех случаях.

Класс `Helper` всегда имеет статические методы для чтения и записи данных в поток `read()` и `write()`, упаковки данных в тип `Any` и распаковки (методы `insert()` и `extract()`), а также методы определения типа `type()` и его идентификатора в репозитарии `id()`.

Класс `Holder` должен не только уметь записывать данные в поток методом _write(), читать их оттуда методом `_read()` и возвращать код типа методом `_typecode()`. В нем должна быть предусмотрена открытая переменная `value`, хранящая значение, и два конструктора: один — по умолчанию, т.е. без параметров, и второй — с параметром, инициализирующим переменную `value`.

Модули

Во время трансляции описания модулей превращаются в пакеты с теми же самыми именами, что и сами модули. Соответственно все описания типов внутри модулей после трансляции в классы и интерфейсы *Java* приобретают область видимости внутри сгенерированных пакетов. Если описания типов находятся за пределами модулей, то они транслируются в глобальный пакет *Java*, т. е. не включаются ни в один пакет.

Для примера опишем следующий модуль:

```
module UserModule
{
  typedef string UserType;
};
```

После его трансляции в выходном каталоге появится подкаталог с именем модуля, и в нем будут сохранены файлы, появившиеся в результате генерации исходных текстов для типа `UserType`. А в самих этих текстах появится строка принадлежности к пакету `UserModule`:

```
package UserModule;
```

Интерфейсы

В первую очередь создаются описания общедоступных интерфейсов *Java*, наследуемые от базового *CORBA*-интерфейса `org.omg.CORBA.Object`. Возьмем следующее описание на *IDL*:

```
interface UserInterface
{
};
```

После компиляции создается следующий интерфейс на языке *Java*:

```
public interface UserInterface
     extends org.omg.CORBA.Object
{
}
```

Внутри сгенерированного интерфейса описываются операции, которые компилятор обнаружит в *IDL*-файле. Для каждого из атрибутов интерфейса них создаются описания методов чтения и записи. Если атрибут объявлен как *readonly*, для него генерируется лишь метод чтения. Исходный текст на *IDL*:

```
attribute float UserAttribute;
```

будет транслирован в следующие описания методов:

```
float UserAttribute();
void UserAttribute(float arg);
```

Если *IDL*-интерфейс наследуется от другого интерфейса, то в его описании на *Java* также будет присутствовать наследование.

Любой параметр операции с модификатором `in` транслируется в аргумент метода, имеющий соответствующий тип на языке *Java*. То же самое и с возвращаемым операцией значением. Параметры `inout` и `out` не могут транслироваться непосредственно в параметры методов на *Java*. Поэтому приходится пользоваться `Holder` классами. Программа-клиент подставляет в качестве параметра экземпляр подобного класса, в котором, как в контейнере, находится передаваемое значение. После передачи параметра по значению хранимые данные изменяются на серверной стороне и возвращаются клиенту, который «вскрывает контейнер» и извлекает новое значение аргумента. Например, показанная операция имеет параметр, объявленный как `inout`:

```
void userOperation(inout double param);
```

Компилятор *IDL* сделает из этого следующий метод на языке *Java*:

```
void userOperation(org.omg.CORBA.DoubleHolder param);
```

Такой подход, конечно, создает дополнительные сложности программиста: придется создать экземпляр `Holder` класса вручную.

Для интерфейса также создаются класс `Helper` и класс `Holder`. В первом из них дополнительно к методам, описанным в разделе «Вспомогательные классы», генерируется метод `narrow()`, с помощью которого делается приведение к оригинальному типу интерфейса. Дело в том, что программе при запросе ссылки на объект возвращается ссылка типа `org.omg.CORBA.Object`, которую необходимо привести к запрошенному типу перед использованием, что и делает `narrow()`. При невозможности произвести эту операцию происходит исключение `CORBA::BAD_ PARAM`.

Некоторые компиляторы (в том числе, *idl2java* из *Visibroker*) генерируют еще один метод. Он называется `bind()` и служит для получения ссылки на запрашиваемый объект. Этот метод не является частью спецификации *CORBA*.

Часто у программистов возникают сложности с пониманием того, как транслируются вложенные в описания интерфейсов конструкции. Кажется, что достаточно сгенерировать новый пакет с его именем и поместить в него внутреннее содержимое интерфейса. Однако, по спецификации *CORBA*, во избежание конфликтов имен нельзя создавать пакеты, имена которых совпадают с уже имеющимися именами. Поэтому решено задавать имена пакетов, добавляя к имени интерфейсов суффикс `Package`.

Опишем интерфейс, внутри которого объявляется пользовательский тип:

```
interface UserInterface
{
typedef any UserType;
};
```

В результате трансляции получится пакет `UserInterfacePackage`, в который и будут помещены все сгенерированные для пользовательского типа файлы, и все они будут начинаться с директивы:

```
package UserInterfacePackage;
```

Простые типы

Трансляция простых типов *IDL* приводит к появлению соответствующих идентификаторов, но уже имеющих *Java*-типы. В табл. 4 показано соответствие между ними: если данный тип может привести к возникновению исключительной ситуации, то она отмечена в графе «Исключения».

Программист должен быть осторожен, когда работает с целочисленными типами *IDL*, объявленными как `unsigned`. Как известно, в *Java* нет беззнаковых типов, и это может стать причиной

ошибок. Следовательно, требуется позаботиться о соблюдении знаковости транслированного числа.

`Holder` классы для простых типов *IDL* определены в библиотеках, отвечающих за поддержку *CORBA* в пакете `org.omg.CORBA`. Имена этих классов начинаются с имени *IDL*-типа, написанного с заглавной буквы, и заканчиваются суффиксом `Holder`.

Таблица 4. Соответствие простых типов IDL типам Java

Тип *IDL*	Тип *Java*	Исключения
boolean	Boolean	
Char	Char	CORBA::DATA_CONVERSION
Wchar	Char	
Octet	Byte	
string	java.lang.String	CORBA::MARSHAL, CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL
Short	Short	
unsigned short	Short	
Long	Int	
unsigned long	Int	
Long long	Long	
unsigned long long	Long	
Float	Float	
Double	Double	
Long double	double (?)	
Fixed	java.math.BigDecimal	CORBA::DATA_CONVERSION
Any	org.omg.CORBA.Any	CORBA::BAD_OPERATION

Константы

Константы внутри интерфейса.

Константы, декларируемые внутри интерфейса, транслируются в поля, описанные как `public final static`, т. е. константы *Java*. Например, строки:

```
interface UserInterface
{
const string constIntoInterface = ”Hello!”;
};
```

будут превращены компилятором *java2idl* в следующий исходный текст на языке *Java*:

```
public interface UserInterface extends
com.inprise.vbroker.CORBA.Object {
final public static java.lang.String
     constIntoInterface = (java.lang.String) ”Hello!”;
}
```

Константы вне интерфейса

Константы, не включенные ни в один интерфейс, превращаются в интерфейс с тем же самым именем, что и константа. Внутри этого интерфейса помещается `public static final` поле с именем `value`. Например, следующий исходный текст:

```
module UserModule {
const string constIntoInterface = ”Hello!”;
};
```

будет транслирован следующим образом:

```
package UserModule;
public interface constIntoInterface {
final public static java.lang.String value =
(java.lang.String)”Hello!”;
}
```

Конструируемые типы

Перечислимые типы

В результате трансляции перечисления получается класс с модификаторами `public final` и именем, соответствующим имени, описанному в *IDL*-файле. Для каждого элемента выбора внутри класса создаются два статических члена. Первый является уникальной целочисленной константой, а второй — ссылкой на экземпляр перечисления, инициализированный константным значением для данного выбора. Заодно генерируется закрытый конструктор, инициализируемый целочисленным значением. Еще один метод, `value()`, возвращает целое число, которым инициализировано перечисление, а в дополнение к нему имеется метод получения элемента пересиелния по заданному числу `from_int()`. При недопустимом значении параметра этого метода возникает исключение `CORBA::BAD_PARAM`.

Например, исходный текст:

```
enum UserEnum { labelOne, labelTwo, labelThree };
```

будет транслирован следующим образом:

```
public final class UserEnum {
public static final int _labelOne = 0,_labelTwo = 1,
_labelThree = 2;
public static final UserEnum labelOne = new
UserEnum(_labelOne);
public static final UserEnum labelTwo = new
UserEnum(_labelTwo);
public static final UserEnum labelThree = new
UserEnum(_labelThree);
public int value() { return _value; }
public static final UserEnum from_int(int i)  throws
     org.omg.CORBA.BAD_PARAM {
          switch (i) {
               case _labelOne:
                    return labelOne;

               case _labelTwo:
                    return labelTwo;

               case _labelThree:
                    return labelThree;
               default:
                    throw new org.omg.CORBA.BAD_PARAM();
          }
     }

private UserEnum(int _value) {
     this._value = _value;
}

private int _value;
}
```

Дискриминируемые объединения

Объединение, описанное на языке *IDL*, транслируется в *Java*-класс тем же самым именем и с модификаторами `public final`. Внутри можно также найти:

- конструктор по умолчанию (без параметров);
- метод чтения дискриминатора `discriminator()`;
- метод чтения для каждого варианта с именем, заимствованным из декларируемого варианта;
- методы модификации значения для каждого декларируемого варианта;
- методы модификации значения для каждого декларируемого варианта, который объявляется для нескольких меток `case`;

- метод `default()`, если в нем есть необходимость.

В качестве примера рассмотрим следующее дискриминируемое объединение:

```
enum UserEnum {Single, Double, Any};

union UserUnion switch (UserEnum) {
case Single:
case Double: wchar anySymbol;
default: any other;
};
```

и полученный в результате трансляции *Java*-код:

```
final public class UserUnion {
private java.lang.Object _object;
private UserModule.UserEnum _disc;
private UserModule.UserEnum _defdisc = UserModule.UserEnum.Any;
public UserUnion() { }
public UserModule.UserEnum discriminator()
     {  return _disc;  }
public char anySymbol()  {...}
public org.omg.CORBA.Any other() {...}
public void anySymbol(char value) {...}
public void anySymbol(UserModule.UserEnum disc, char value) {...}
public void other(org.omg.CORBA.Any value) {...}
}
```

Все методы чтения генерируют исключительную ситуацию `CORBA::BAD_OPERATION`, если читаемое значение не установлено. Поэтому желательно сначала вызывать метод `descriminator()`, чтобы ознакомиться с текущим типом хранимого значения.

Если не указать в объединении метку `default`, компилятор сверит все имеющиеся метки со всеми возможными значениями дискриминанта. Если таких значений больше, чем ветвей `case`, будет сгенерирован еще один метод `default()` (или `_default()` в случае конфликта имен), в котором хранимое значение будет установлено так, чтобы оно было за пределами дискриминанта. Если в предыдущем примере удалить строку с меткой default, то сгенерируется следующий метод:

```
public void _default() {
_disc = _defdisc;
_object = null;
}
```

Структуры

Тип `struct` во время компиляции транслируется в класс *Java* с модификаторами `final` и `public`. Имя полученного класса совпадает с именем структуры. В классе объявляются переменные-члены для каждого объявленного в *IDL* поля структуры. Тип переменных-членов уже относится к языку *Java* и выясняется в процессе трансляции полей структуры. Так же внутри полученного класса декларируются два конструктора: один — по умолчанию, т. е. без параметров, и другой — конструктор инициализации с параметрами для инициализации переменных-членов. Некоторые компиляторы создают также метод `toString()`, возвращающий строку, в текстовой форме отражающую содержимое полей класса.

Например, объявлена следующая структура:

```
struct UserStructure
{
any descriptor;
Object reference;
};
```

После трансляции полученный класс `UserStructure` имеет следующий вид:

```
public final class UserStructure {
public org.omg.CORBA.Any descriptor;
public org.omg.CORBA.Object reference;
public UserStructure() { }
public   UserStructure(org.omg.CORBA.Any   __descriptor,
     org.omg.CORBA.Object __reference) {
     descriptor = __descriptor;
     reference = __reference;
}
}
```

Последовательности и массивы

Последовательности не создают в процессе трансляции исходного текста какого-либо исходного текста, но значительно усложняют класс `Helper` и класс `Holder`. Класс `Holder` теперь содержит массив с именем value для хранения элементов последовательности, а в методах класса `Holder` идет проверка диапазона передаваемого массива на предмет выхода за границы. Например, при трансляции:

```
typedef sequence UserSequence;
```

вызывает генерацию следующего массива в `Holder` классе:

```
public byte[] value;
```

и несколько мест, где происходит проверка границ массива:

```
abstract public class UserSequenceHelper {
...
public static byte[] read(org.omg.
     CORBA.portable.InputStream_input) {

     if(_length3 > 128) {
          throw       new       org.omg.CORBA.BAD_PARAM(
               ”Sequence exceeded bound”);
     }

}

public static void
write(org.omg.CORBA.portable.OutputStream _output,
     byte[] value) {
     if(value.length > 128) {
          throw new org.omg.CORBA.BAD_PARAM(
               ”Sequence exceeded bound”);
     }

}

}
```

Трансляция массивов во многом похожа на трансляцию ограниченных последовательностей. Разница лишь в проверке границ. Если ограниченные последовательности, как показано выше, предполагают проверку на выход за границы размера, то массив проверяется на четкое соответствие размеру:

```
public static void
write(org.omg.CORBA.portable.OutputStream _output,
byte[] value) {
     if(value.length != 128) {
          throw new org.omg.CORBA.BAD_PARAM(
               ”Invalid array length”);
     }

_output.write_octet_array(value, 0, value.length);
}
```

Исключения

Поскольку исключения имеют схожее со структурой строение, любое исключение, описанное пользователем на *IDL*, транслируется в

класс с модификатором `final public`, ведущим свою родословную от `org.omg.CORBA.UserException`. Системные исключения *CORBA*, наоборот, наследуются от исключения `java.lang.RuntimeException`, которое, как правило, не перехватывается.

Сгенерированный класс содержит по переменной для каждого описанного в *IDL* поля и два конструктора: по умолчанию (без параметров) и инициализации. Следующий пример:

```
exception UserException {
string why;
octet errorCode;
};
```

показывает, как происходит трансляция исключения:

```
public final class UserException extends
org.omg.CORBA.UserException {
public String why;
public byte errorCode;
public UserException() { super(); }

public UserException(String __why, byte __errorCode) {
     super();
     why = __why;
     errorCode = __errorCode;
}
}
```

В *CORBA* имеется ряд предопределенных системных исключений, каждое из которых косвенно наследует класс `java.lang.RuntimeException` через другой класс `org.omg.CORBA.SystemException`.

Псевдонимы типов (`typedef`)

Поскольку оператор `typedef` создает псевдонимы для уже имеющихся типов, то во время трансляции любое упоминание пользовательского типа, полученного с помощью `typedef` (за исключением последовательностей и массивов), приведет к подстановке оригинального типа.

2.2.5. Создание *CORBA*-систем

Для того чтобы создать *CORBA*-систему, сначала необходимо установить и настроить соответствующий инструментарий. В этом разделе будет вкратце описаны основные действия на каждом этапе разработки, а затем в разделе с примерами эти действия будут рассмотрены на конкретном примере.

Инструменты и их конфигурирование

Среди наиболее популярных и доступных инструментов для создания *CORBA*-систем брокер для *Java* от *Sun Microsystems*, входящий в стандартную поставку *Java*, *VisiBroker* от *Inprise/Borland*, *WebLogic*.

Порядок действия при создании *CORBA*-системы

Создавая *CORBA*-приложения, нужно помнить, что их модель отличается от модели традиционных монолитных программ и даже клиент-серверных систем, хотя с последними есть и нечто общее. Связку объектов *CORBA* и клиентов трудно назвать приложением как таковым. Подобные системы похожи на паутину, где все переплетено: клиент может в любую минуту стать сервером, и пользователь вряд ли узнает, с каким сервером объектов он работает в данный отрезок времени, а если проект выполнен грамотно, может даже и не заметить сбоя. Типичная тактика действий программы, использующей технологию *CORBA*, такова: соединиться с нужным объектом, использовать его функции и отсоединиться от него. И таких атомарных циклов могут быть сотни.

Добиться хороших результатов в создании программ на основе *CORBA* можно, придерживаясь определенного порядка действий:

- объектно-ориентированный анализ и моделирование;
- описание и трансляция объектов;
- создание сервера;
- создание клиента;
- отладка объектов.

Объектно-ориентированный анализ и моделирование

CORBA – объектно-ориентированная технология, потому в первую очередь необходимо осуществить объектную декомпозицию и представить систему в виде взаимодействующих между собой классов. Чтобы модель была понятна и разработчикам, нужно задокументировать ее. Построить *IDL*-описания по *UML*-модели поможет пакет *Rational Rose*.

Разработайте порядок действий, в соответствии с которым будете создавать реализации объектов. Выделите в готовой модели атомарные объекты, не зависящие от других, они и станут кандидатами на первоочередное создание и отладку. Неплохо подумать и о размещении объектов в сети, согласуясь с топологией последней. В итоге образуется четкая последовательность инсталляции готового кода, определятся виртуальные домены.

Описание и трансляция интерфейсов

Готовая модель системы содержит классы, которые должны быть описаны с помощью языка *IDL*. Далее это описание можно транслировать с помощью *IDL*-компилятора в базовые исходные тексты на конкретном языке программирования (заглушки и скелетоны) или добавить *IDL*-описания в репозиторий интерфейсов.

2.3. Пример «Служба мгновенных сообщений»

Функциональность

Разработаем систему обмена мгновенными сообщениями с выделенным сервером. Каждый подключенный к серверу клиент будет получать информацию о других зарегистрированных на сервере клиентах и статусе их подключения, передавать сообщения и получать отправленные в его адрес сообщения сразу после их отправки (асинхронно). Сервер будет обеспечивать авторизацию отправителя и получателя сообщения, предотвращая фальсификацию и утечку информации.

Рис. 6 Окно регистрации и аутентификации

В окне регистрации/аутентификации (рис. 6) пользователь вводит свое имя и пароль, а потом нажимает одну из кнопок *Регистрация* (*Register*) или *Аутентификация* (*Login*). При регистрации сервер запоминает имя и пароль пользователя (если такое имя еще не зарегистрировано) и выполняет вход в систему с этими параметрами. При аутентификации сервер проверяет наличие комбинации имя/пароль среди ранее зарегистрированных пользователей и в случае совпадения выполняет вход в систему.

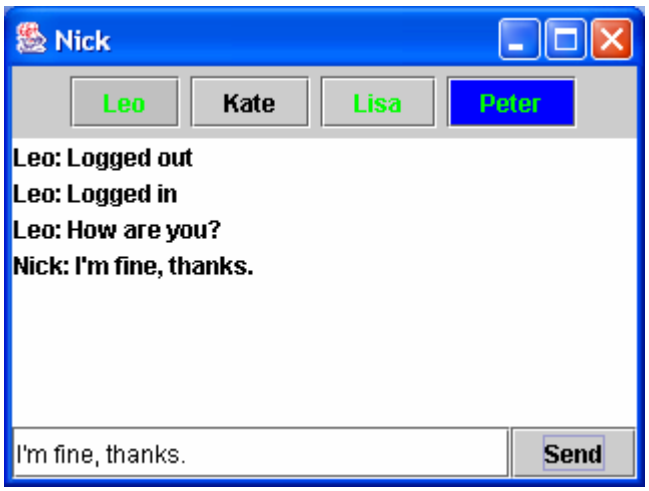


Рис. 7 Окно сообщений

У Ника открыто окно обмена сообщениями с Лео (рис. 7). Сообщения Лео появляются на экране сразу после отправки. Кэйт зарегистрирована на сервере, но не подключена к системе в данный момент. Питер написал Нику сообщение, о чем свидетельствует синий цвет кнопки. Ник прочитает это сообщение после того, как переключится в режим общения с Питером, нажав на кнопку с его именем.

Инструменты

Разработка будет вестись на языках *Java* и *C++* с помощью средств *Java 2 SE 1.4.2* и *Microsoft Visual Studio 8* (2005). В качестве брокера объектных запросов (*ORB*) для обоих языков будем использовать *Borland VisiBroker 7*. На настройке брокера остановимся подробнее.

На момент написания этого текста седьмая версия *VisiBroker* – единственная официально бесплатно распространяемая для ознакомительного использования (trial, 60 дней). Стабильную версию *Inprise* обещает выпустить в следующем году. Описываемые здесь способы настройки могут не потребоваться при использовании окончательной версии. Временную (на 60 дней) лицензию можно получить на официальном сайте. После окончания ее действия штатная замена лицензии не помогает, требуется удалить старую лицензию как файл, и после этого установить новую.

В качестве *C++* компилятора *VisiBroker 7*, в отличие от предыдущих версий, поддерживает только компилятор из *Visual Studio 8* (компилятор от *Borland* не поддерживается). *Java* поставляется в комплекте, причем две версии – 1.4.2 и 1.5. Несоответствие версий стало одной из причин неспособности данного брокера работать с *Java* без модификации. Для компиляции и выполнения написанных на *Java CORBA*-объектов используются входящие в *VisiBroker* программы vbj и vbjc вместо java и javac соответственно. При этом для компиляции

используется версия *JDK 1.5*, а для выполнения – 1.4.2, что приводит к неработоспособности. Можно было бы добавить опцию компиляции с целью 1.4.2, но проще выбрать одну версию, в нашем случае 1.4.2. Для этого требуется заменить в файле <VBDir>\bin\toolsjdk.config строку
“`javahome $var(installRoot)/jdk/jdk1.5.0`” на
“`javahome $var(installRoot)/jdk/jdk1.4.2`”.

Кроме *vbj*, *vbjc* из программ от *VisiBroker* нам потребуются *SmartAgent* и *Naming Service*. Первый должен быть запущен для выполнения всех примеров (кроме первого). Второй проще запускать указанным в соответствующих примерах образом. Несмотря на то, что согласно документации запуск *Naming Service* с соответствующими параметрами является альтернативой использованию *SmartAgent*, для работы текущей версии сервиса имен требуется и запуск *SmartAgent*, и явное указание параметров.

Интерфейс

Реализацию службы мгновенных сообщений с использованием *CORBA* мы начнем с описания интерфейсов *CORBA*-объектов на языке *IDL*. Создадим *IDL*-файл с описанием модуля `Message`, в котором содержатся два интерфейса – `MessageReceiver` и `MessagingService`. Методы этих интерфейсов будут доступны для вызова клиентам *CORBA*-объектов. Первый интерфейс реализуется на стороне клиента службы мгновенных сообщений, а второй – на стороне сервера.

```
module Message {
interface MessageReceiver {
     void newMessage(in unsigned long from, in string
     text);
     void userStatusNotification(
          in unsigned long uid, in string name, in
          boolean online);
};
interface MessagingService {
     unsigned long registerUser(
           string userName, in string password);
     unsigned long login(
          in string userName, in string password,
          in MessageReceiver receiver);
     boolean sendMessage(
          in unsigned long from, in string password,
          in unsigned long to, in string msg);
     boolean logout(in unsigned long uid, in string
     password);
};
};
```

Интерфейс `MessageReceiver` предназначен для уведомления клиента о событиях в системе. Метод `newMessage` уведомляет клиента о новом предназначенном ему сообщении, а метод `userStatusNotification` – об изменении статуса других пользователей службы.

Интерфейс сервера `MessagingService` предоставляет методы для регистрации пользователя (`registerUser`), аутентификации (`login`), отправки сообщения конкретному пользователю (`sendMessage`) и выхода из системы (`logout`). В целях гарантии аутентичности каждый метод сервера требует передачи пароля в качестве одного из параметров.

Далее *IDL*-файл необходимо скомпилировать, чтобы получить классы заглушек, скелетонов и другие вспомогательные классы на целевом языке программирования. Для компиляции *IDL*-описания в *Java* будем использовать команду:

```
idl2java [params] Message.idl
```

Здесь в параметрах указывается тип используемого объектного адаптера. От него зависит, какие вспомогательные классы будут созданы. Тип объектного адаптера по умолчанию – *POA*. Для компиляции в *C++* используется команда:

```
idl2cpp [params] Message.idl
```

IDL-файл необязательно компилировать отдельно. Имена создаваемых автоматически файлов вполне предсказуемы, поэтому их можно использовать в коде реализации и до компиляции. На практике удобно создать единый makefile, с помощью которого будет компилироваться сначала *IDL*-описание, а затем классы реализации на целевом языке.

Прямое задание *IOR*

В этом примере и сервер и клиент службы мгновенных сообщений будут реализованы на языке *Java*. Для вызова методов *CORBA*-объекта необходимо сначала получить ссылку на этот объект. В данном примере ссылка на сервер задается явно в виде *Interoperable Object Reference* (*IOR*).

Начнем с реализации сервера службы мгновенных сообщений. Он выполняет следующие действия:

1. Инициализация *ORB*.

```
ORB orb = ORB.init(args,null);
```

2. Создание серванта (объекта реализации).

```
Message.MessagingService messagingService = new
MessagingServiceImpl();
```

3. Получение *IOR* серванта и сохранение его в файл.

```
String ior = orb.object_to_string(messagingService);
...
FileWriter fw = new FileWriter("MS.ior"); fw.write(ior);
```

4. Ожидание подключения клиентов.

Ниже приведен полный листинг кода сервера.

```
import org.omg.CORBA.*;
import java.io.*;

public class Server {
public static void main(String[] args) {
     ORB orb = ORB.init(args,null);
     Message.MessagingService messagingService =
          new MessagingServiceImpl();
     String ior =
orb.object_to_string(messagingService);
     System.out.println(messagingService +
     " is ready.\n" + ior);
     try {
          FileWriter fw = new FileWriter("MS.ior");
          fw.write(ior);
          fw.close();
          System.out.println("IOR written to file");
     } catch(IOException e) {
          System.out.println(
          "Failed to write IOR to file. Exception: ");
          e.printStackTrace();
     }

     try {
          Thread.currentThread().join();
     } catch(InterruptedException ex) {}
}
}
```

Далее опишем сервант, реализующий интерфейс `MessagingService`. Как видно из листинга, приведенного ниже, его необходимо унаследовать от класса-скелетона `Message._MessagingServiceImplBase`, сгенерированного автоматически компилятором *IDL*.

```
import org.omg.CORBA.*;
import org.omg.Messaging.*;
import java.util.ArrayList;
import java.util.LinkedList;
```

```
import java.util.HashMap;

public      class      MessagingServiceImpl      extends
Message._MessagingServiceImplBase {
private HashMap nameToId;
private ArrayList users;
public MessagingServiceImpl() {
     System.out.println("Constructing
     MessagingServiceImpl");
     nameToId = new HashMap();
     users = new ArrayList();
}
public   int   registerUser   (String   userName,   String
password) {
     if (nameToId.containsKey(userName)) {
          System.out.println("User " + userName +
          " already registered");
          return 0;
     } else {
          int uid = nameToId.size();
          nameToId.put(userName, new Integer(uid));
          users.add(new User(userName, password));
          notifyAllUsers(++uid, false);
          System.out.println("User " + userName +
               " registered successfully, user ID is " +
               (new Integer(uid)).toString());
          return uid;
     }
}
...
private void notifyAllUsers(int uid, boolean online)
{
     String userName = ((User)users.get(uid - 1)).name;
     for (int i = 0; i < users.size(); ++i) {
          if (i != uid - 1) {
               User buddy = (User)users.get(i);
               if (buddy.online) {

               buddy.receiver.userStatusNotification(
                    uid, userName, online);
               }
          }
     }
}
class User {...}
class Msg  {...}
}
```

Теперь создадим клиент службы мгновенных сообщений. Он выполняет следующие действия:

1. Инициализация *ORB*.

```
ORB orb = ORB.init(args, null);
```

2. Получение *IOR* сервера из файла.

```
String ior = br.readLine();
```

2. Преобразование *IOR* сервера в CORBA-объект и приведение его к соответствующему типу.

```
org.omg.CORBA.Object obj = orb.string_to_object(ior);
Message.MessagingService service =
Message.MessagingServiceHelper.narrow(obj);
```

4. Создание окон пользовательского интерфейса и серванта.

Полный листинг класса клиента приведен ниже. Листинги классов пользовательского интерфейса можно найти в файлах с примерами.

```
import org.omg.CORBA.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.ArrayList;
import java.util.Vector;
import java.io.*;

public class Client {
public static void main(String[] args) {
     try {
          ORB orb = ORB.init(args, null);
          FileReader fr = new FileReader("MS.ior");
          BufferedReader br = new BufferedReader(fr);
          String ior = br.readLine();
          org.omg.CORBA.Object           obj           =
          orb.string_to_object(ior);
          System.out.println(ior);

          Message.MessagingService service =
          Message.MessagingServiceHelper.narrow(obj);

          MessagingFrame        mf        =        new
          MessagingFrame(service);
          Message.MessageReceiver receiver =
               new MessageReceiverImpl(mf);
          LoginFrame loginFrame =
               new LoginFrame(service, receiver, mf);
          loginFrame.show();
     } catch (Throwable t) {
          t.printStackTrace();
     }
}
}
```

Наконец, создадим сервант, реализующий интерфейс `MessageReceiver`. По аналогии с сервантом сервера, он наследуется от скелетона `Message._MessageReceiverImplBase`.

```
import javax.swing.*;
import org.omg.CORBA.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.ArrayList;
import java.util.Vector;

public class MessageReceiverImpl extends
Message._MessageReceiverImplBase {
public MessageReceiverImpl(MessagingFrame mf) {
     mesFrm = mf;
}

public void newMessage(int from, String text) {
     mesFrm.newMessage(from, text);
     System.out.println("New message from " +
          (new Integer(from)).toString() + ": " + text);
}

public void userStatusNotification
     (int uid, String name, boolean online) {
     mesFrm.buddyStateChanged(uid, name, online);
     System.out.println(name + " is now " +
          (online ? "online" : "offline"));
}

private MessagingFrame mesFrm;
}
```

2.4. Приложение: словарь терминов *CORBA*

BOA (*Basic Object Adapter*) – стандарт *объектного адаптера* до *CORBA* 2.2 (недостаточно полно специфицированный).

CORBA (*Common Object Request Broker Architecture*) – технология создания распределенных приложений; стандарт, разработанный *Object Management Group* (*OMG*); независимая от языка реализации модель взаимодействия распределенных объектов. Позволяет создавать запросы между объектами на разных языках программирования.

***CORBA*-объект** – виртуальное понятие: нечто, посылающее запросы к другим *CORBA*-объектам – *серверным объектам* и получающее запросы от других *CORBA*-объектов – *клиентов*.

IDL (*Interface Definition Language*) – язык описания интерфейсов в формате, который не зависит от языка программирования.

IIOP (*Internet Inter-ORB Protocol*) – протокол передачи объектных запросов по *TCP/IP*.

IOR (*Interoperable Object Reference*) – ссылка на объект, уникальная в пределах сервера (как правило, содержит *идентификатор объекта* как составную часть).

Java IDL – не вполне корректное название реализации *CORBA* для *Java* (содержит не только компилятор *IDL*).

Java-IDL **компилятор** – компилятор описаний *IDL* в классы-заглушки и вспомогательные классы *Java*.

ORB (*Object Request Broker*) – программа-транслятор межобъектного взаимодействия; работая на клиенте и на сервере, передает объектные запросы между ними.

POA (*Portable Object Adapter*) – стандарт *объектного адаптера* начиная с *CORBA* 2.2 (достаточно полно специфицирован, является платформенно-независимым).

Smart Agent – административная утилита, осуществляет поиск объектов в домене и балансировку нагрузки.

Активация *CORBA*-объекта – запуск существующего *CORBA*-объекта для обработки клиентских запросов (в зависимости от политик объектного адаптера предполагает создание *сервантов*, занесение в *карту активных объектов*, и т.д.).

Виртуальный домен – один или несколько компьютеров, логически объединенных для выполнения некоторой задачи.

Временный (*transient*) *CORBA*-объект – объект, который уничтожается с завершением активировавшего его потока.

Деактивация *CORBA*-объекта – останов *CORBA*-объекта (разрыв связки между объектом и сервантом, в общем случае без разрушения объекта).

Демон активации объектов (*Object Activation Daemon, OAD*) – демон, отслеживающий входящие запросы и активизирующий нужные объекты-серверы.

Идентификатор объекта (*Object ID*) – уникальное имя объекта внутри его объектного адаптера.

Инкарнация серванта – связывание серванта с *CORBA*-объектом для обработки клиентского запроса.

Карта активных объектов (*Active Object Map*) – таблица *объектного адаптера*, в которой он ведет реестр активных *CORBA*-объектов и связанных с ними *сервантов* (первые представлены в карте своими идентификаторами).

Менеджер сервантов – элемент технологии *CORBA*, один из способов управлять связками объект-*сервант*, предоставляет подходящий *сервант* для объекта.

Объектный адаптер – элемент технологии *CORBA*, отображающий понятие программно-реализованных *сервантов* на концепцию *CORBA*-объектов; в его обязанности входит: создание *CORBA*-объектов и их объектных ссылок; демультиплексирование запросов на каждый серверный *CORBA*-объект; перенаправление запросы к соответствующему серванту, который обеспечивает реализацию серверного CORBA-объекта; *активация* и *деактивация CORBA*-объектов (соответственно, *инкарнация* и *эфемеризация* соответствующие серванты).

Связывание языка программирования – правила трансляции *IDL*-описаний в код на данном языке; эти правила определены *OMG*.

Сервант – физическая реализация *CORBA*-объекта; серверная программа, написанная на каком-либо из языков программирования и выполняющая *CORBA*-объект.

Сервис именования (*Naming Service*) – *CORBA*-объект, который позволяет обнаружить другие объекты по имени. Может быть *устойчивым* (запоминать ссылки и имена после остановки) и *временным* (не запоминать).

Скелетон – заготовка для серванта, генерируемая *IDL*-компилятором.

Устойчивый (*persistent*) *CORBA*-объект – объект, который может существовать дольше, чем активировавший его поток.

Эфемеризация серванта – разрушение связки *CORBA*-объект – сервант.

Тема 3. Технология *Enterprise Java Beans*

3.1. Введение

Enterprise JavaBeans (также часто употребляется в виде аббревиатуры *EJB*) — спецификация технологии написания и поддержки серверных компонент, содержащих бизнес-логику. Является частью *J2EE*.

Изучив учебный материал данного раздела, Вы:

- узнаете или пополните свои знания об основах *EJB*-технологии;
- приобретете умения в области разработки систем, использующих *EJB*-технологию.

В рамках темы рассматриваются следующие учебные вопросы:

- основы *EJB* технологии и ее архитектура;
- создание систем на основе *EJB*-технологии

3.2. История и необходимость появления *EJB*

В данном разделе будут рассмотрены сеансовые (*Session*), объектные (*Entity*) и управляемые сообщениями (*Message-driven*) компоненты (*Beans*) *EJB*. Все примеры разработаны в среде *Windows XP SP2*. Несмотря на это, все используемые компоненты являются кросс-платформенными, так что примеры должны работать так же в *Linux*, *Solaris* или *Mac OS X*.

Для демонстрации примеров необходимо следующее ПО:

- *Java 5.0* (http://java.sun.com/);
- Сервер приложений (контейнер *EJB*) – открытый сервер приложений *JBoss* (http://www.jboss.org/);
- СУБД – *Oracle Database 9i*.
- Среда разработки – *Eclipse-WTP* (*Eclipse Web Tools Platform*). Это модифицированная версия среды разработки Eclipse, в которую добавлена широкая поддержка веб-приложений и *EJB*. Скачать ее можно также на сайте *Eclipse* – http://www.eclipse.org/.

- Для *Eclipse-WTP* следует скачать еще плагин *XDoclet* (http://xdoclet.sourceforge.net/). Он используется при развертывании объектных компонентов (*Entity Beans*) *EJB* в базе данных.
- Для примеров web-приложений, взаимодействующих через *EJB* с базой данных будут использоваться веб-сервер *Apache* (http://www.apache.org/), и контейнер *JSP Tomcat* (http://tomcat.apache.org/).

3.3. Общее описание архитектуры *EJB*

3.3.1. Компонент *Enterprise Bean*

Серверные компоненты *Enterprise Java Beans* бывают трех принципиально разных типов:

- объектные компоненты (*Entity Beans;* точный перевод с английского – компоненты-сущности);
- сеансовые компоненты (*Session Beans*) и
- компоненты, управляемые сообщениями (*Message-driven Beans*).

Сеансовые и объектные компоненты являются серверными компонентами, основанными на *RMI*, к которым можно получить доступ при помощи распределенных объектных протоколов. Компоненты, управляемые сообщениями, появившиеся в *EJB 2.0* – это асинхронные серверные компоненты, которые отвечают на асинхронные сообщения *JMS* и других протоколов.

Хорошее эмпирическое правило при конструировании компонентов *EJB* состоит в том, что объектные компоненты моделируют прикладные понятия, которые могут быть выражены существительными. Например, объектный компонент может представлять покупателя, часть оборудования, элемент инвентаря или даже место. Другими словами, объектные компоненты моделируют объекты реального мира. Эти объекты являются обычными постоянными записями в какой-либо базе данных.

Сеансовые компоненты являются расширениями клиентского приложения и отвечают за управление процессами и задачами. Для примера рассмотрим задачу о проектировании системы управления отелем. В качестве объектных компонентов в отеле могут выступать “Комната”, “Постоялец”, “Член персонала”. Сеансовые компоненты отвечают за взаимодействие этих элементов и предназначены для управления различными видами деятельности, такими как заказ мест в гостинице. Сеансовый компонент “Бронирование места” может

использовать компоненты “Постоялец” и “Комната”, чтобы сделать заказ.

Точно так же компоненты, управляемые сообщениями, отвечают за координацию задач, включающих другие сеансовые и объектные компоненты. Главное отличие между компонентами, управляемыми сообщениями, и сеансовыми компонентами состоит в том, как к ним осуществляется доступ. В то время как сеансовый компонент предоставляет удаленный интерфейс, определяющий, какие методы могут быть вызваны, компонент, управляемый сообщениями, этого не делает. Вместо этого он подписывается или прослушивает определенные асинхронные сообщения, обрабатывает их и управляет действиями, которые выполняют другие компоненты в ответ на эти сообщения. Например, компонент “Обработчик заказов”, управляемый сообщениями, мог бы принимать асинхронные сообщения, возможно из существующей системы заказов, с помощью которых он бы мог координировать взаимодействие компонентов “Постоялец” и “Комната”, для того, чтобы забронировать место в отеле.

Действия, которые принимают сеансовые компоненты и компоненты, управляемые сообщениями, по своей сути являются кратковременными: вы начали делать заказ, вы сделали часть работы, и на этом их действие заканчивается. Сеансовые компоненты и компоненты, управляемые сообщениями не соответствуют никаким объектам в базе данных. Понятно, что в процессе своей работы они оказывают влияние на базу данных, но через объектные компоненты. По крайней мере, такого подхода придерживаются при разработке больших распределенных приложений. В принципе, ничто не мешает из сеансового компонента сделать подключение к базе данных при помощи *JDBC*, и произвести необходимые операции с базой данных. Такой способ работы с базой данных через *EJB* будет рассмотрен далее в примерах.

Главное различие компонентов в том, что объектные компоненты имеют устойчивое состояние, а сеансовые компоненты и компоненты, управляемые сообщениями, моделируют взаимодействия и устойчивого состояния не имеют.

3.3.2. Классы и интерфейсы

Для реализации объектного или сеансового компонента необходимо определить:

- интерфейсы компонента,
- класс компонента и
- первичный ключ (только для объектных компонентов).

В данной главе будут сначала рассмотрены примеры сеансовых компонентов, разработанные в соответствии со спецификацией *EJB 2.1*. Компонент может иметь следующие интерфейсы:

- *Удаленный интерфейс* (*Remote Interface*) определяет прикладные методы компонента, доступные из приложений, внешних по отношению к контейнеру *EJB*, то есть прикладные методы, которые компонент представляет внешним приложениям. Удаленный интерфейс расширяет (*extends*) интерфейс *javax.ejb.EJBObject*. Внешние по отношению к *EJB* приложения вызывают методы удаленного интерфейса, а получают эти методы они при помощи домашнего интерфейса (*Home Interface*).
- *Домашний интерфейс* (*Home Interface*) определяет методы жизненного цикла компонента, доступные из приложений, внешних по отношению к контейнеру *EJB*, то есть методы создания новых компонентов, удаления и поиска существующих компонентов. Домашний интерфейс (*Home Interface*) расширяет (*extends*) интерфейс *javax.ejb.EJBHome*. Для того чтобы создать новый *EJB* или обратиться к существующему *EJB*, внешнее приложение должно сначала вызвать соответствующий метод `create()` его домашнего интерфейса (*Home Interface*).
- *Локальный интерфейс* (*Local Interface*) определяет прикладные методы *EJB*, которые могут использоваться другими компонентами, размещенными в том же контейнере *EJB*, но не внешними по отношению к контейнеру *EJB* приложениями. Он позволяет компонентам, находящимся в одном контейнере *EJB*, взаимодействовать без дополнительных затрат, связанных с применением протокола распределенных объектов, что повышает их производительность. Локальный интерфейс расширяет (*extends*) интерфейс *javax.ejb.EJBLocalObject*. Внешние по отношению к *EJB* компоненты, находящиеся в том же самом контейнере, работают с методами локального интерфейса, а получают эти методы они при помощи локального домашнего интерфейса (*Local Home Interface*)
- *Локальный домашний интерфейс* (*Local Home Interface*) определяет методы жизненного цикла компонента, доступные из компонентов, находящихся в том же контейнер *EJB*, то есть методы создания новых компонентов, удаления и поиска существующих компонентов. Это позволяет компонентам взаимодействовать без дополнительных затрат, связанных с

использованием протокола распределенных объектов, что улучшает их производительность. Домашний интерфейс (*Home Interface*) расширяет (*extends*) интерфейс *javax.ejb.EJBLocalHome*. Для того чтобы создать новый *EJB* или обратиться к существующему *EJB*,; компонент, находящийся в том же контейнере *EJB*, должен сначала вызвать соответствующий метод его домашнего интерфейса (*Home Interface*).

Важным отличием между удаленным и локальным интерфейсом является то, что вызовы методов локального интерфейса не используют *RMI* и не генерируют *RemoteException*. Для корректной работы не обязательно предоставлять классу компонента все четыре интерфейса. К примеру, если компонент не используется никаким клиентским приложением (а нужен только другим компонентам, работающим под управлением того же *EJB* контейнера), то нет необходимости предоставлять удаленный интерфейс. То есть в зависимости от тех задач, которые решает данный компонент, ему может быть предоставлен либо локальные, либо удаленные, либо и те и другие интерфейсы.

- Для реализации компонентов, помимо интерфейсов, необходимо реализовать также
- *Класс компонента.* Классы сеансовых и объектных компонентов фактически реализуют прикладные методы и методы жизненного цикла компонента. Заметим, что класс компонента для сеансовых и объектных компонентов не реализует непосредственно (*implements* в терминах *Java*) ни один интерфейс компонента, однако он должен содержать методы, названия и списки параметров которых совпадают с названиями и списками параметров методов, определенных в соответствующих интерфейсах. Также этот класс должен содержать метод (или методы) `ejbCreate()`, соответствующие некоторым методам в обоих, домашнем (*Home Interface*) и локальном домашнем (*Local Home Interface*), интерфейсах. Объектные компоненты должны реализовывать интерфейс *javax.ejb.EntityBean*, а сеансовые компоненты – интерфейс *javax.ejb.SessionBean*. Компоненты, управляемые сообщениями, не используют ни один из интерфейсов компонентов, потому что к ним никогда не осуществляется доступ посредством вызовов методов из других приложений или компонентов. Вместо этого компоненты, управляемые сообщениями, содержат один метод `onMessage()`. Этот метод вызывается при прибытии нового сообщения.
- *Первичный ключ.* Это простой класс, предоставляющий указатель внутри базы данных. Первичный ключ необходим

только объектным компонентам. Он должен реализовывать интерфейс *java.io.Serializable*.

У компонентов, управляемых сообщениями нет никаких компонентных интерфейсов, но, несмотря на это, они могут быть клиентами любых сеансовых или объектных компонентов и взаимодействовать с этими компонентами через их интерфейсы. Объектные и сеансовые компоненты, с которыми взаимодействует компонент, управляемый сообщениями, могут быть совмещенными, в этом случае он работает с их локальными компонентными интерфейсами, или они могут располагаться в разных адресных пространствах или контейнерах *EJB* – в этом случае используются удаленные интерфейсы.

Спецификации *EJB 1.1* и старше определяют удаленные (*Remote*) интерфейсы компонента в терминах протокола *Java RMI-IIOP*, который обеспечивает совместимость с *CORBA*. Для совместимости с типами *Java RMI-IIOP* поставщики *EJB* должны ограничить определение интерфейсов и параметров типами, соответствующими *CORBA IIOP 1.2* (в соответствии со спецификацией *CORBA 2.3.1*). Локальные интерфейсы в *EJB 2.0*, *EJB 2.1* и *EJB 3.0* не являются интерфейсами *Java RMI* и не обязаны поддерживать *IIOP 1.2* и использовать типы совместимые с протоколом *Java RMI-IIOP*.

В *Java RMI-IIOP* определены три базовых типа объектов:

- `byte`, `boolean`, `char`, `int`, `long`, `short`, `double`, `float`;
- объекты, реализующие интерфейс `java.io.Serializable`;
- удаленные типы *Java RMI* – классы реализующие интерфейс `java.rmi.Remote`.

При попытке передать через удаленный метод объект, не относящийся к одному из трех типов, будет сгенерировано исключение `java.rmi.RemoteException`.

Каждый метод в удаленном интерфейсе (*Remote Interface*) должен генерировать исключение `java.rmi.Remote`. Оно генерируется при проблемах с передачей данных в сети или невозможности найти требуемый сервер. Помимо этого разработчики могут использовать любые другие исключения при разработке компонент *EJB*.

Протокол *Java RMI IIOP* также накладывает ограничения на удаленные интерфейсы. Этих ограничений много, но реально используется только два из этих ограничений.

- Сериализуемые объекты (*Serializable*) не должны реализовывать (*implement*) интерфейс *java.rmi.Remote*.

- Удаленный интерфейс не может реализовывать два и более интерфейса, у которых есть методы с одинаковыми именами. Удаленный интерфейс может перегружать (*overload*) свои собственные методы и расширять (*extends*) удаленный интерфейс с перегруженными (*overloaded*) методами.

Существует много инструментальных средств, которые позволяют сильно упростить работу с *EJB*. В частности одним из средств, которое также предоставляет платформа *Eclipse-WTP*, является средство создания связей между объектными компонентами и базой данных. В *Eclipse-WTP* это делается при помощи пакета *XDoclet*. Каким образом это делается, будет рассмотрено далее.

3.4. Сеансовые компоненты

Сеансовые компоненты могут быть с состоянием (*Stateful*) и без состояния (*Stateless*). Компоненты с состоянием при использовании клиентом поддерживают состояния диалога (*Conversational state*). Состояние диалога не заносится в базу данных, оно сохраняется в памяти все время, пока клиент использует сессию. Состояние сеансового компонента может измениться при вызове любого его метода, и это изменение может привести к последующим вызовам методов. Состояние диалога поддерживается только до тех пор, пока клиентское приложение активно использует компонент. Если клиент отключается или освобождает компонент, его состояние теряется навсегда. Сеансовый компонент с состоянием не может использоваться несколькими клиентами. Он выделяется для одного клиента на все время своей жизни.

Сеансовые компоненты без состояния (*Stateless Session Bean*) ни поддерживают никакого состояния диалога. Каждый метод полностью независим и пользуется только данными, переданными через его параметры. Сеансовые компоненты без состояния (*Stateless Session Bean*) имеют более высокую производительность, с точки зрения пропускной способности и потребления ресурсов, чем объектные компоненты и сеансовые компоненты с состоянием (*Stateful Session Bean*), поскольку для обслуживания большого количества клиентов требуется всего несколько экземпляров сеансового компонента без состояния.

При создании сеансового компонента без состояния (*Stateless Session Bean*) контейнер *EJB* выполняет следующую последовательность действии:

- При помощи метода `Class.newInstance()` создается экземпляр класса компонента.

- Вызывается метод `SessionBean.setSessionContext( SessionContext context)`. При помощи этого метода экземпляру компонента передается ссылка на объект класса `EJBContext`, предоставляющий интерфейс для взаимодействия с контейнером *EJB (*в частности, в этом объекте есть метод `lookup()`, позволяющий искать другие объекты по путям *JNDI)*.
- Вызывается метод `ejbCreate()`. У сеансовых компонентов без состояния (*Stateless Session Bean*) этот метод не имеет параметров. В этом методе компонент может создать подключения к базам данных и к другим ресурсам.
- Контейнер помещает созданный компонент в пул готовых компонентов.

При запуске контейнер обычно создает несколько экземпляров сеансовых компонентов без состояния (*Stateless Session Bean*). При необходимости контейнер может либо создавать новые экземпляры, либо удалять ненужные. При удалении компонента из пула готовых компонентов вызывается метод `ejbRemove()`. В этом методе следует закрыть все открытые подключения к различным ресурсам.

То, каким образом и когда контейнер *EJB* должен создавать и удалять экземпляры сеансовых компонентов без состояния, не описывается спецификацией *EJB*. В каждом из существующих контейнеров *EJB* этот механизм реализован по-разному.

Сеансовый компоненты с состоянием создаются тогда, когда клиент запрашивает метод `create()` у домашнего интерфейса (*Home Interface*). После этого созданный компонент переходит в состояние готовых методов. В этом состоянии он готов выполнять свои методы, вызываемые клиентом через удаленную ссылку. Компонент в это время может хранить открытые ресурсы (базы данных, *TCP/IP* сессии и др.), а также хранить посредством своих переменных определенное состояние, которое может изменяться при вызове клиентом различных его методов. Последовательность методов, вызываемых контейнером при создании сеансового компонента с состоянием аналогична последовательности методов при создании сеансового компонента без состояния. Только компонент в конце переходит в состояние готовых методов, а не в пул готовых компонентов.

При нахождении компонента в состоянии готовых методов, он может либо оставаться в этом состоянии, либо переведен в пассивное состояние (при этом будет вызван метод `ejbPassivate()` – см. далее), либо быть удаленным. То, при каких условиях компонент будет переведен в пассивное состояние, в каждом из существующих

контейнеров определяется по-разному. При активизации компонента (вызывается метод `ejbActivate()` – см. далее), компонент переводится из пассивного состояние в состояние готовых методом. При этом значение всех внутренних переменных и состояние диалога сохраняется.

Компонент может быть удален при вызове соответствующего метода `remove()` домашнего интерфейса, либо контейнером после определенного промежутка времени, в течение которого компонент «бездействовал». Компонент не может быть удален при выполнении транзакции. При удалении у компонента будет вызван метод `ejbRemove()`.

3.5. Пример “Конвертор валют”

Компонент

В данном примере мы покажем разработку сеансового компонента без состояния (*Stateless Session Bean*) для конвертации денежных сумм из одной валюте в другую. В качестве валют будут использоваться доллары, евро и йены. Компонент будет переводить сумму в долларах в сумму в евро или йенах.

Удаленный интерфейс

В удаленном интерфейсе (*Remote Interface*) определим два метода – `dollarToEuro(double)` и `dollarToYena(double)`. Эти методы, получив в качестве параметра сумму в долларах, переведут эту сумму по некоторому постоянному курсу и вернут сумму в евро или йенах соответственно.

```
public interface CurrencyRemote extends EJBObject {
public double dollarToEuro(double dollars) throws Remot
Exception;
public double dollarToYena(double yena) throws
RemoteException;
}
```

Домашний интерфейс

Домашний интерфейс (*Home Interface*) состоит из одного метода.

```
public interface CurrencyHome extends EJBHome {
public CurrencyRemote create() throws RemoteException,
CreateException;
}
```

Класс компонента

В классе компонента заданы для простоты два постоянных курса перевода валют (задание этих констант, а также основных используемых извне методов выделены жирным шрифтом).

```
public class CurrencyBean implements SessionBean {
private static final long serialVersionUID =
8897567469393327031L;

private double dollarToYenaRate = 100.25;
private double dollarToEuroRate = 0.83;

public void ejbCreate()  {

}
public double dollarToEuro(double dollars) {
     return dollars * dollarToEuroRate;
}
public double dollarToYena(double dollars) {
     return dollars * dollarToYenaRate;
}

public void ejbActivate() throws EJBException,
RemoteException {
}

public void ejbPassivate() throws EJBException,
RemoteException {
}

public void ejbRemove() throws EJBException,
RemoteException {
}

public void setSessionContext(SessionContext arg0)
throws EJBException,
     RemoteException {
}
}
```

Дескриптор развертывания

Дескриптор развертывания для этого компонента ничем принципиально не отличается от дескриптора развертывания компонента из предыдущего примера.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar id="ejb-jar_ID" version="2.1"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
<description> Second session bean </description>
<display-name>CurrencyConvertorBean</display-name>

<enterprise-beans>
     <session>
          <ejb-name>CurrencyBean</ejb-name>
```

```
<home>currencyConvertorBean.CurrencyHome</home>

<remote>currencyConvertorBean.CurrencyRemote</remote>

<ejbclass>currencyConvertorBean.CurrencyBean</ejb-class>
          <session-type>Stateless</session-type>
          <transaction-type>Container</transaction-type>
     </session>
</enterprise-beans>
</ejb-jar>
```

Создание проекта для компонента

В этом примере не будет рассматриваться создание простого *Java*-проекта для компонента, ограничимся *EJB*-проектом. Создаем *EJB*-проект *CurrencyConvertorBean*. Также в каталоге ejbModule создаем интерфейсы и классы, вставляем дескриптор развертывания. После этого добавляем проект к контейнеру *JBoss*.

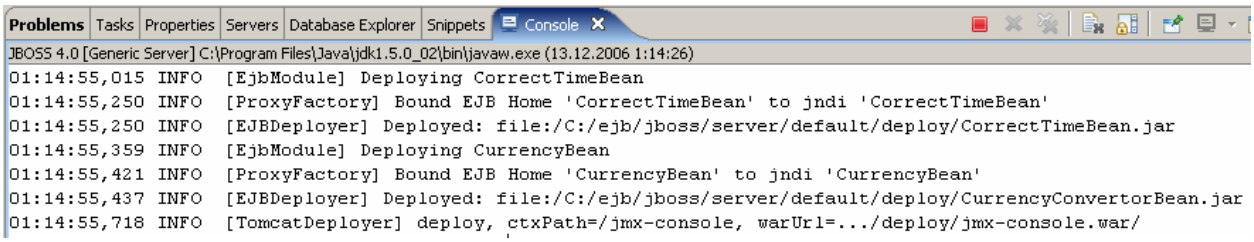


Рис. 8. Теперь их два

Если все было сделано правильно – то при запуске *JBoss* появится сообщение об успешном развертывании двух компонентов (компонента из предыдущего примера и текущего компонента) (рис. 8).

Создание проекта для клиентского приложения

Теперь напишем небольшое приложение, которое будет работать с компонентом *CurrencyConvertorBean*. Это будет графическое приложение, в котором пользователь сможет ввести наименования валют, и сумму денег в одной валюте. После этого приложение выведет сумму денег в другой валюте.

Вначале создадим *Java*-проект и назовем его *CurrencyConvertorBeanClient*. Аналогичным образом, как было описано в предыдущем примере, подключим необходимые библиотеки и выберем в качестве зависимого проекта *CurrencyConvertorBean*. Далее создадим пакет, а в нем – класс клиента.

```
public class CurrencyConvertorClient extends JFrame
implements ActionListener {

private static final long serialVersionUID =
3889223123915102442L;

private Context jndiContext;
```

```
     private JLabel cur1Label;
     private JTextField cur1;
     private JLabel cur2Label;
     private JTextField cur2;
     private JLabel amountLabel;
     private JTextField amount;

     private JButton getRate;
     private JLabel rate;

     public CurrencyConvertorClient() {
          super("CurrencyConvertor");
          layoutPane();

          try {
               jndiContext = createJBossContext();
          } catch (NamingException e) {
               e.printStackTrace();
          }
     }

     // Получение удаленного интерфейса
     private   CurrencyRemote   getRemoteInterface()   throws
NamingException,
          CreateException, RemoteException {
          Object ref = jndiContext.lookup("CurrencyBean");
          CurrencyHome home = (CurrencyHome)
               PortableRemoteObject.narrow(ref,
               CurrencyHome.class);

          CurrencyRemote remote = home.create();
          return remote;
     }

     private void layoutPane() {
          GridBagConstraints gc = new GridBagConstraints();
          gc.insets = new Insets(5, 5, 5, 5);

          Container content = getContentPane();
          content.setLayout(new GridBagLayout());

          cur1Label = new JLabel("Currency 1: ");
          gc.fill = GridBagConstraints.BOTH;
          gc.weightx = 1.0;
          gc.gridx = 0;
          gc.gridy = 0;
          gc.gridwidth = 2;
          gc.gridheight = 1;
          content.add(cur1Label, gc);

          cur2Label = new JLabel("Currency 2: ");
          gc.gridy = 1;
```

```
        content.add(cur2Label, gc);

        amountLabel = new JLabel("Amount: ");
        gc.gridy = 2;
        content.add(amountLabel, gc);

        getRate = new JButton("Get value: ");
        gc.gridy = 3;
        content.add(getRate, gc);
        getRate.addActionListener(this);

        cur1 = new JTextField();
        gc.gridx = 2;
        gc.gridy = 0;
        content.add(cur1, gc);

        cur2 = new JTextField();
        gc.gridy = 1;

        content.add(cur2, gc);

        amount = new JTextField();
        gc.gridy = 2;
        content.add(amount, gc);

        rate = new JLabel("Not loaded");
        gc.gridy = 3;
        content.add(rate, gc);
    }

    public void actionPerformed(ActionEvent ae) {
        loadExchangeRate();
    }

    private void loadExchangeRate() {
        String currency1 = cur1.getText();
        String currency2 = cur2.getText();

        double am;

        try {
            am = Double.parseDouble(amount.getText());
        } catch (NumberFormatException e) {
            rate.setText("Invalid number");
            return;
        }

        try {
            CurrencyRemote remote = null;
            remote = getRemoteInterface();
            boolean loaded = true;
```

```
                // Пока поддерживается только перевод из
                долларов в евро
                // и долларов в йены
                loaded &= currency1.equals("USD");
                loaded &= (currency2.equals("EUR") ||
                     currency2.equals("YEN"));

                if (loaded) {
                     double res = 0;
                     if (currency2.equals("EUR")) {
                          res = remote.dollarToEuro(am);
                     } else if (currency2.equals("YEN")) {
                          res = remote.dollarToYena(am);
                     }

                     NumberFormat             nf             =
                     NumberFormat.getInstance();
                     nf.setMaximumFractionDigits(2);
                     rate.setText(nf.format(res));
                } else {
                     rate.setText("Rate not found");
                }
           } catch (RemoteException e) {
                e.printStackTrace();
                rate.setText("Error...");
           } catch (NamingException e) {
                e.printStackTrace();
                rate.setText("Error...");
           } catch (CreateException e) {
                e.printStackTrace();
                rate.setText("Error...");
           }
      }

      public static void main(String args[]) {
           CurrencyConvertorClient      client      =      new
           CurrencyConvertorClient();

      client.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
           client.setLocation(400, 400);
           client.setSize(300, 200);
           client.setVisible(true);
      }

      private  static  Context  createJBossContext()  throws
NamingException {
           Properties p = new Properties();
           p.put("java.naming.factory.initial",
                "org.jnp.interfaces.NamingContextFactory");
           p.put("java.naming.provider.url",
           "jnp://127.0.0.1:1099");
           p.put("java.naming.factory.url.pkgs",
                "org.jboss.naming:org.jnp.interfaces");
```

```
        Context jndiContext = new InitialContext(p);
        return jndiContext;
    }
    }
```

Рис. 9. Перевод долларов в евро

Введем в качестве первой валюты доллар, в качестве второй – евро, а в качестве суммы – 100 долларов. Щелкнем кнопку Get value**:**, и появится сумма уже в евро – 83 евро.

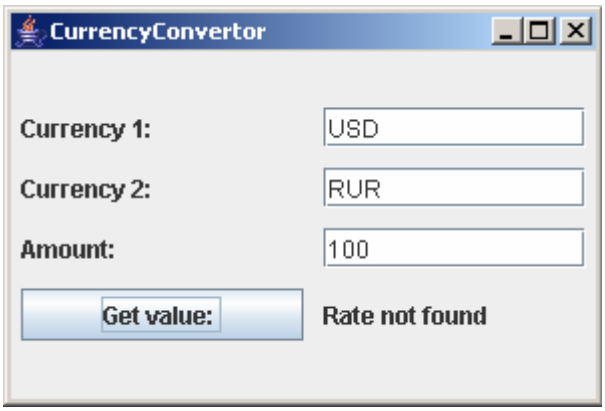


Рис. 10. Курс перевода не найден

Если же в качестве валют для перевода указывается что-либо отличное от пар USD -> EUR или USD -> YEN, то тогда выводится сообщение Rate not found (рис. 10).

3.6. Объектные компоненты

Как уже говорилось ранее, объектные компоненты моделируют прикладные понятия, которые могут быть описаны существительными. Это не закон установленный раз и навсегда, скорее эмпирическое правило, которое позволяет определять, когда прикладное понятие следует реализовать как объектный компонент, а когда – как сеансовый или вообще управляемый состоянием компонент. Объектные компоненты описывают и состояние, и поведение объектов из жизни.

Разработчики могут добавлять к объектным компонентам данные и правила, характеризующие компоненты.

Объектные компоненты представляют записи в базе данных. При изменении объектных компонентов происходят изменения в базе данных. В предыдущих примерах уже рассматривалась возможность внесения изменения в базу данных непосредственно через сеансовые компоненты, но это было не очень удобно – приходилось, к примеру, для создания заказа в базе данных выполнять несколько запросов *SQL*. При использовании объектных компонентов можно было бы просто использовать метод `Order.createOrder(List commoditiesList)`. Компонент *Order* представляет приложению полный доступ к информации о заказах, а также возможность управлять ими – создавать, модифицировать и удалять. При помощи объектных компонентов разработку можно упростить и сделать более рентабельной.

Когда объектный компонент создается, в базу данных должна быть вставлена новая запись и создан экземпляр компонента, связанный с этими данными. По мере того, как компонент изменяется, его состояние меняется, и изменения должны быть синхронизированы с данными в базе данных. За процесс координации данных, представляющий экземпляр компонента, с базой данных, отвечает служба *Persistence*.

Существует два основных типа объектных компонентов, различающихся реализацией механизма постоянства: компоненты с постоянством, управляемым контейнером (*container-managed persistence* или *CMP*), и компоненты поддерживающие постоянство самостоятельно (*bean-managed persistence* или *BMP*).

В случае с *CMP*, контейнер знает, как постоянство экземпляра и поля отношений с другими компонентами отображаются в базе данных, и автоматически вставляет, модифицирует и удаляет данные. В случае *BMP*, вся работа выполняется разработчиком самостоятельно. Ему необходимо написать код манипуляции базой данных. Контейнер *EJB* сообщает экземпляру компонента, когда безопасно вставлять, модифицировать и удалять его данные из базы данных. Экземпляр выполняет всю работу по обеспечению своего постоянства самостоятельно.

Начиная со спецификации *EJB 2.0*, *CMP* подверглось очень сильным изменениям. В результате *CMP* теперь больше не является обратно совместимым с предыдущими спецификациями. Поэтому довольно часто используют сокращение *CMP 2.0*.

В *CMP 2.0* контейнер объектных компонентов автоматически управляет их состоянием. Контейнер заботится о регистрации компонентов в транзакциях и поддерживает их состояние в базе данных. Разработчик компонентов описывает атрибуты и связи компонентов при помощи виртуальных (*virtual*) полей постоянства и отношений.

Разработчик не определяет их явно. Вместо этого в классе создаются абстрактные методы доступа (`set` и `get`).

Несмотря на то, что *BMP* требует от разработчиков большего количества усилий, чем *CMP*, использование его в некоторых случаях является оправданным. *CMP* не предоставляет возможностей работы с разными источниками данных, поэтому если компонент должен работать сразу с несколькими базами данных или же получать информацию из других источников, необходимо использование *BMP*.

При создании объектного компонента (*BMP* или *CMP*), контейнер *EJB* выполняет следующую последовательность действий:

- Контейнер вызывает метод `Class.newInstance()` у класса объектного компонента. До этого момента объектный компонент существовал только как набор файлов (класса компонента, интерфейсов и дескриптора развертывания). У компонента вызывается метод `setEntityContext(EntityContext)`. После этого созданный компонент помещается в пул объектных компонентов. Обычно контейнер хранит несколько экземпляров компонента в пуле. Они помещаются туда при запуске контейнера. В дальнейшем, в зависимости от оьстоятельств, контейнер может увеличивать или уменьшать количество экземпляров компонентов в пуле. Компонент, находящийся в пуле, способен выполнять запросы `ejbFind<суффикс>` и `ejbSelect<суффикс>` (рассмотрим далее).

- При вызове методов `ejbCreate()` а затем метода `ejbPostCreate()` у домашнего интерфейса, или же когда контейнер активирует компонент посредством метода `ejbActivate()`, компонент переходит в состояние готовности. После завершения метода `ejbCreate()` создается первичный ключ – идентификатор созданного объекта, а затем уже вызывается метод `ejbPostCreate()`. В состоянии готовности компонент способен обслуживать запросы от клиента. Когда клиент вызывает метод удаленного интерфейса `getName()`, то этот метод вызывается посредством *RMI* у компонента на сервере. В состоянии готовности могут также вызываться методы `ejbLoad()` и `ejbStore()`. Они вызываются в любом порядке – порядок определяется производителем компонента. Некоторые производители вызывают эти методы перед каждым выполнением прикладного метода.

- Компонент может переходить обратно из состояния готовности в пул в случае пассивизации (посредством

метода `ejbPassivate()`). Эта операция может выполняться для более эффективного управления ресурсами. Также компонент может переходить обратно при вызове метода `ejbRemove()` – в этом случае все данные о компоненте удаляются из базы данных.

- Когда контейнер решит уменьшить количество экземпляров в пуле, он удаляет некоторые из них. Теперь сборщик мусора может их окончательно удалить из памяти. После удаления из пула, у компонента вызывается метод `unsetEntityContext()`.

3.7. Компоненты, управляемые сообщениями

Компоненты, управляемые сообщениями (*Message-Driven Beans*) – это серверные компоненты без состояния, способные обрабатывать асинхронные сообщения протокола *JMS* (*Java Messaging Service*). Помимо этого компоненты поддерживают механизм транзакций (о нем дальше). Компонент отвечает за обработку сообщений, а его контейнер заботится об автоматическом управлении всем окружением компонента, включающим в себя транзакции, безопасность, ресурсы, совместный доступ и подтверждения получения сообщений.

Одним из важных свойств компонентов, управляемых сообщениями, является то, что они могут получать и обрабатывать сообщения параллельно. Эта возможность дает им значительное преимущество перед обычными компонентами *JMS*, где разработчику приходится самому реализовывать поддержку безопасности и транзакций.

3.8. Механизм транзакций в EJB

3.8.1. Что такое транзакция?

В бизнесе под транзакцией обычно понимают обмен чем-либо между двумя сторонами. Скажем, покупка книги в магазине является транзакцией, потому что происходит обмен денег на товар. Чтобы стороны остались довольны результатом, каждый участник должен выполнить некоторый набор взаимосвязанных действий. В нашем примере, отдав кассиру сумму, превышающую цену книги, покупатель законно ожидает сдачу, кассовый чек и свою книгу.

В программном обеспечении понятие транзакции имеет сходное значение. Здесь транзакцией называют операцию, связанную с доступом к одному или нескольким ресурсам, обычно базам данных. Такая

операция представляет собой последовательность связанных между собой элементарных действий, которые требуется выполнить вместе.

Транзакции обладают следующими свойствами, называемыми по-английски *ACID*:

- *Атомарность* (*Atomicity*). Транзакция должна быть выполнена целиком или не выполнена совсем. Для успешного выполнения транзакции все составляющие ее действия должны завершиться благополучно. В таком случае произведенные изменения фиксируются (*commit*). Если же в одном из действий возникает ошибка, транзакция отменяется и все возвращается в исходное состояние (*rollback*), как будто транзакция и не начиналась. Например, покупатель не согласится уйти без своей покупки после того, как заплатил за нее деньги: либо он получит свою покупку, либо добьется возврата денег. Остановка в промежуточном состоянии невозможна (если человек в здравом уме).
- *Согласованность* (*Consistency*). Речь идет о согласованности (непротиворечивости) данных в хранилище, над которым производится транзакция. Транзакция должна переводить данные из одного согласованного состояния в другое согласованное. Например, после того, как покупатель ушел с покупкой, информация об этом должна попасть в систему складского учета. Если же товар все еще будет числиться на складе, то возникнет несогласованность.
- *Изолированность* (*Isolation*). Транзакция должна выполняться изолированно от других транзакций или процессов. Во время выполнения транзакции данные не могут изменяться извне, другой частью системы. Например, в чек, который выбивает кассир, не должны попасть товары покупателя, которого в тот же момент обслуживает соседний кассир.
- *Устойчивость* (*Durability*). Устойчивость подразумевает, что по завершении транзакции все изменения данных записываются в хранилище. Таким образом, даже сбой в системе не приведет к потере данных, и работа системы может быть возобновлена из того состояния, когда успешно завершилась последняя транзакция.

3.8.2. Транзакции в *EJB*

Одним из главных достоинств *EJB* является поддержка декларативного управления транзакциями. Без такой возможности

пришлось бы смешивать бизнес-логику с громоздкой обработкой всех возможных ошибок и с поддержкой отката данных к исходному состоянию (а что если при откате произойдут ошибки?), а это негативно сказалось бы на читаемости и поддерживаемости кода.

Декларативное управление транзакциями позволяет обеспечить выполнение *ACID*-свойств путем маркирования бизнес-методов аннотацией `@javax.ejb.TransactionAttribute` или редактирования *XML*-дескриптора развертывания, то есть без изменений бизнес-логики. Всю сложную работу по отмене транзакции при возникновении ошибки возьмет на себя *EJB*-контейнер.

Обычно каждый бизнес-метод *EJB* является транзакцией. Это означает, что все действия, выполняемые в рамках этого метода, включая вызовы других *EJB*, являются одной транзакцией и обладают *ACID*-свойствами. В случае возникновения исключения (потомка `RuntimeException` или произвольного исключения, помеченного как `@javax.ejb.ApplicationException(rollback=true)`) транзакция отменяется.

Более тонкое управление транзакциями, вплоть до запрета транзакций, осуществляется путем задания параметров аннотации `@javax.ejb.TransactionAttribute`:

- `NOT_SUPPORTED`. При вызове метода с таким атрибутом текущая транзакция приостанавливается. Метод исполняется вне транзакции. При выходе из метода транзакция возобновляется.
- `SUPPORTS`. Такой атрибут означает, что метод может вызываться как в контексте транзакции, так и вне его. Если вызов метода произошел в рамках транзакции, то метод исполняется как ее часть. При вызове метода вне транзакции новая транзакция не инициируется.
- `REQUIRED`. Атрибут говорит о том, что метод может исполняться только в рамках транзакции. Это может быть либо уже существующая транзакция, из которой был вызван этот метод, либо новая транзакция, автоматически инициируемая для этого метода при вызове его извне транзакции. Все методы *EJB* по умолчанию имеют именно этот атрибут.
- `REQUIRES_NEW`. Данный атрибут означает, что при вызове метода всегда инициируется новая транзакция, вне зависимости от того, вызван он из уже существующей транзакции или нет. Если метод был вызван в контексте транзакции, на время работы метода та транзакция приостанавливается.

- `MANDATORY`. Атрибут показывает, что метод может вызываться только в контекте существующей транзакции. В противном случае будет выброшено исключение `javax.ejb.EJBTransactionRequiredException`.
- `NEVER`. Такой атрибут говорит о том, что метод не может быть вызван в рамках транзакции. Вызов метода из транзакции приведет к исключению `javax.ejb.EJBException`. Успешный вызов метода возможен только извне транзакции. Еще более гибкое управление транзакциями возможно при помощи объекта `UserTransaction`, однако этот способ увеличивает вероятность «сделать что-нибудь не так» и сводит на нет достоинства *EJB* по декларативному управлению транзакциями.

3.8.3. Пример: перевод денег с одного счета на другой

Классическим примером транзакции является перевод денег с одного банковского счета на другой. Эта операция состоит из двух действий: снятие денег с первого счета и добавление денег на второй счет. Нетрудно заметить, что ошибка в одном из этих действий приведет к нарушению согласованности системы, то есть к потере денег или к появлению денег из ниоткуда. Различные неожиданности возможны и в случае, когда один счет одновременно участвует в двух переводах. Решение проблемы известно: необходимо использовать транзакции с их *ACID*-свойствами.

В данном примере мы разработаем сеансовый компонент без состояния (*Stateless session bean*), который осуществляет перевод денег с одного банковского счета на другой и делает запись об этом в протокол. Вот удаленный интерфейс этого *EJB*:

```
package ru.ifmo.javaee.bank;

import java.util.List;

import javax.ejb.Remote;

@Remote
public interface Bank {

public int createAccount();

public void removeAccount(int id);

public List<BankAccount> listAccounts();

public void deposit(int id, int sum);
```

```
public void withdraw(int id, int sum);

public void transfer(int srcId, int dstId, int sum);

}
```

Демонстрировать транзакции мы будем на примере метода `transfer()`. Остальное играет вспомогательную роль.

Рассмотрим реализацию *EJB*:

```
package ru.ifmo.javaee.bank;

import java.util.List;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

@Stateless
public class BankBean implements Bank {

@PersistenceContext
private EntityManager em;

public int createAccount() {
     BankAccount account = new BankAccount();
     em.persist(account);
     return account.getId();
}

public void removeAccount(int id) {
     BankAccount  account  =  em.find(BankAccount.class,
     id);
     em.remove(account);
}

public List<BankAccount> listAccounts() {
     Query query = em.createQuery("FROM BankAccount");
     return query.getResultList();
}

public void deposit(int id, int sum) {
     BankAccount  account  =  em.find(BankAccount.class,
     id);
     account.deposit(sum);
}

public void withdraw(int id, int sum) {
```

```
        BankAccount account = em.find(BankAccount.class,
        id);
        account.withdraw(sum);
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRES
    _NEW)
    public void transfer(int srcId, int dstId, int sum) {
        withdraw(srcId, sum);
        deposit(dstId, sum);
        em.persist(new TransferRecord(srcId, dstId, sum));
        if (Math.random() > 0.5) {
            throw new RuntimeException("fake exception");
        }
    }

    }
```

Атрибут `REQUIRES_NEW` обеспечивает исполнение каждого вызова метода `transfer()` в контексте отдельной транзакции.

Транзакция состоит из снятия денег с первого счета, добавления на второй счет и создания записи о произведенном переводе. Ошибка в любом из этих действий приведет к генерации исключения и откату транзакции.

Для простоты ошибка генерируется искусственно с вероятностью 50% (см. последние три строки метода `transfer()`). В среднем половина транзакций должна завершаться удачно, оставшиеся должны отменяться.

Приведем исходные коды используемых здесь вспомогательных классов.

```
package ru.ifmo.javaee.bank;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="account")
public class BankAccount implements Serializable {

private static final long serialVersionUID =
8754634761874208747L;

private int id;
private int balance;

public BankAccount() { }
```

```
public BankAccount(int balance) {
     this.balance = balance;
}

@Id
@GeneratedValue
public int getId() {
     return id;
}

public void setId(int id) {
     this.id = id;
}

public int getBalance() {
     return balance;
}

public void setBalance(int balance) {
     if (balance >= 0) {
          this.balance = balance;
     } else {
          throw  new  IllegalArgumentException("Negative
          balance is not allowed");
     }
}

public void deposit(int sum) {
     if (sum < 0) {
          throw  new  IllegalArgumentException("Negative
          sum is not allowed");
     }
     balance += sum;
}

public void withdraw(int sum) {
     if (sum < 0) {
          throw  new  IllegalArgumentException("Negative
          sum is not allowed");
     }
     if (sum > balance) {
          throw  new  IllegalArgumentException("Withdrawn
          sum greater than balance");
     }
     balance -= sum;
}

public String toString() {
     return  "Account  #"  +  id  +  "  with  balance  $"  +
     balance;
}
```

```
}

package ru.ifmo.javaee.bank;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="transfer_log")
public class TransferRecord {

private int id;
private int srcId;
private int dstId;
private int sum;

public TransferRecord(int srcId, int dstId, int sum) {
     this.srcId = srcId;
     this.dstId = dstId;
     this.sum = sum;
}

@Id
@GeneratedValue
public int getId() {
     return id;
}

public void setId(int id) {
     this.id = id;
}

public int getSrcId() {
     return srcId;
}

public void setSrcId(int srcId) {
     this.srcId = srcId;
}

public int getDstId() {
     return dstId;
}

public void setDstId(int dstId) {
     this.dstId = dstId;
}

@Column(name="transfer_sum")
public long getSum() {
```

```
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }

    }
```

Дескриптор развертывания для *JBoss*:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
<enterprise-beans>
        <session>
            <ejb-name>BankBean</ejb-name>
            <jndi-name>Bank/Remote</jndi-name>
        </session>
</enterprise-beans>
</jboss>
```

Все готово к тестированию. Тестировать работу транзакций будем при помощи простой программы-клиента.

```
package ru.ifmo.javaee.bank;

import java.util.Hashtable;
import java.util.List;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class BankClient {

public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage:          BankClient
            <command> [arguments]");
            return;
        }

        Hashtable<String,     String>     env     =     new
        Hashtable<String, String>();

        env.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.jnp.interfaces.NamingContextFactory");

        env.put(Context.PROVIDER_URL, "localhost:1099");

        try {
            InitialContext ctx = new InitialContext(env);
```

```
            Bank bank = (Bank) ctx.lookup("Bank/Remote");
            ctx.close();

            if (args[0].equals("createAccount")) {
                int balance = Integer.parseInt(args[1]);
                int id = bank.createAccount();
                bank.deposit(id, balance);
                System.out.println("Created account #" +
                id + " with $" + balance);
            } else if (args[0].equals("removeAccount")) {
                int id = Integer.parseInt(args[1]);
                bank.removeAccount(id);
                System.out.println("Removed account #" +
                id);
            } else if (args[0].equals("listAccounts")) {
                System.out.println("Listing all
                accounts...");
                List<BankAccount> accounts =
                bank.listAccounts();
                for (BankAccount account : accounts) {
                    System.out.println(account);
                }
            } else if (args[0].equals("transfer")) {
                int srcId = Integer.parseInt(args[1]);
                int dstId = Integer.parseInt(args[2]);
                int amount = Integer.parseInt(args[3]);
                System.out.println("Transferring $" +
                amount + " from #" + srcId +
                " to #" + dstId + "...");
                try {
                    bank.transfer(srcId, dstId, amount);
                } catch (RuntimeException e) {
                    System.out.println("Caught
                    exception: " + e.getMessage());
                    System.out.println("Transaction
                    rolled back");
                }
            } else {
                System.out.println("Expecting one of:
                createAccount, removeAccount,
                listAccounts, transfer");
            }

        } catch (NamingException e) {
            e.printStackTrace();
        }
    }

    }
```

В зависимости от переданных в командной строке параметров программа выполняет разные действия:

- `createAccount <сумма>` – создание счета с заданной суммой, будет выведен номер нового счета;
- `removeAccount <номер>` – удаление счета с данным номером;
- `listAccounts` – вывод списка счетов с номерами и суммами;
- `transfer <номер1> <номер2> <сумма>` – перевод заданной суммы с первого счета на второй.

3.9. Безопасность в EJB

Безопасность является серьезной задачей в enterprise приложениях, поэтому технология EJB предоставляет средства для ее решения. Обеспечение безопасности приложения включает в себя три подзадачи.

- *Аутентификация* (*Authentication*). Аутентификация – это процесс подтверждения личности пользователя, пытающегося получить доступ в систему. Чаще всего подтверждение личности представляет собой ввод пароля, хотя возможны и более экзотические варианты, например, сканирование отпечатков пальцев или сетчатки глаза.
- *Авторизация* (*Authorization*). Авторизация заключается в определении того, какие действия может и не может совершать каждый пользователь, вошедший в систему. Например, одним пользователям может быть дан доступ на запись в базу данных, другие же могут только читать из нее данные.
- *Конфиденциальность* (*Confidentiality*). Требуется защитить данные, передаваемые по каналам связи между системой и пользователями, от перехвата и фальсификации злоумышленниками. Обычно для этого используется шифрование, например, *SSL*.

Авторы спецификации *EJB* сосредоточились на решении задачи авторизации, оставив остальные задачи разработчикам *EJB*-контейнеров.

Обычно аутентификация происходит на уровне *JNDI*, при создании контекста. Когда пользователь подключается к *EJB* системе, с ним ассоциируется определенный идентификатор безопасности (security identity). Этот идентификатор неявно передается при каждом вызове методов *EJB*. На основе идентификатора безопасности контейнер *EJB* решает, имеет ли пользователь право на совершение запрошенной операции.

В *EJB* реализована авторизация на основе ролей. Каждому подключенному к системе пользователю присваивается роль, например, «служащий», «менеджер», «администратор». В то же время, каждому методу *EJB* приписывается набор ролей пользователей, имеющих право на вызов этого метода. В момент вызова метода пользователем производится проверка того, что роль пользователя является одной из допустимых ролей вызываемого метода.

Допустимые роли определяются аннотацией `@javax.annotation.security.RolesAllowed`, принимающей в качестве параметра массив строк – названий ролей. Аннотация `@javax.annotation.security.PermitAll` позволяет открыть доступ к методу пользователям, играющим любые роли.

Аннотация `@javax.annotation.security.RunAs` задает роль, от имени которой метод EJB будет вызывать другие методы. По умолчанию метод работает в роли вызвавшего его пользователя.

3.10. Пример: ограничение доступа к методам EJB

В этом примере мы добавим защиту от несанкционированного доступа к написанному нами *EJB*, реализующему операции с банковскими счетами.

Дадим доступ к методам `BankBean` работникам банка, запретив его всем остальным. Соответствующую роль назовем `BankEmployee`. Желаемого эффекта можно добиться, добавив следующую аннотацию к объявлению класса `BankBean`:

```
...
@Stateless
@RolesAllowed("BankEmployee")
public class BankBean implements Bank {
...
```

Также необходимы дополнительные настройки контейнера *EJB*. Мы рассмотрим настройки контейнера *JBoss*.

Во-первых, необходимо обновить дескриптор развертывания приложения:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
<enterprise-beans>
     <session>
          <ejb-name>BankBean</ejb-name>
          <jndi-name>SecureBank/Remote</jndi-name>
     </session>
</enterprise-beans>
<security-domain>SecureBank</security-domain>
</jboss>
```

Security domain – это база данных пользователей. Приложения могут пользоваться общей базой, а могут иметь свои собственные базы пользователей. Мы используем security domain, который мы назвали `SecureBank`.

Во-вторых, требуется описать используемый security domain в конфигурационных файлах *JBoss*. Для это добавим следующие строки в файл `${jboss_home}/server/default/conf/login-config.xml`:

```
<application-policy name="SecureBank">
<authentication>
     <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                flag="required">
          <module-option name="usersProperties">
               props/bank-users.properties
          </module-option>
          <module-option name="rolesProperties">
               props/bank-roles.properties
          </module-option>
     </login-module>
</authentication>
</application-policy>
```

Этот способ настройки пользователей использует два текстовых файла. Первый из них, `props/bank-users.properties`, задает пользователей и их пароли:

```
vladykin=asdfgh
```

Второй, `props/bank-roles.properties`, задает роли пользователей:

```
vladykin=BankEmployee
```

Потребует небольших изменений и наша тестовая программа. Информация о пользователе и его пароле передается при подключении к *EJB*-контейнеру средствами *JNDI*:

```
Hashtable<String, String> env = new
Hashtable<String, String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.security.jndi.JndiLoginInitialContextFac
tory");
env.put(Context.PROVIDER_URL, "localhost:1099");
env.put(Context.SECURITY_PRINCIPAL, args[3]);
env.put(Context.SECURITY_CREDENTIALS, args[4]);
```

За это отвечают параметры `SECURITY_PRINCIPAL` и `SECURITY_CREDENTIALS`. Их значения извлекаются из последних параметров, переданных программе в командной строке.

Тема 4. JNDI

4.1. Введение

Java Naming and Directory Interface (*JNDI*) – это *API* для доступа к службам имен и каталогов.

Изучив учебный материал данного раздела, Вы:

- узнаете или пополните свои знания о *JNDI API* и протоколе *LDAP*;
- приобретете умения в области разработки программ, использующих *JNDI API* и протокол *LDAP*.

В рамках темы рассматриваются следующие учебные вопросы:

- основы *JNDI API*;
- основы протокола *LDAP*.

4.2. Общие сведение о *JNDI*

Прежде чем погружаться в *JNDI*, поясним, к каким службам это *API* дает доступ.

Службой имен, в самом широком смысле, называют систему, управляющую отображением множества имен во множество объектами. Зная имя объекта в системе, клиент может получить доступ к этому объекту или ассоциировать с этим именем другой объект. Примером является *DNS*, служба доменных имен. В ее ведении находится соответствие между понятными человеку доменными именами (например, www.ifmo.ru) и понятными компьютеру сетевыми *IP*-адресами (например, 184.35.65.10). Посылая *DNS* доменное имя, клиент получает соответствующий ему *IP*-адрес.

В *службе каталогов* поименованные объекты сгруппированы в древовидную структуру. Кроме того объекты каталога имеют атрибуты. Наиболее близким и понятным примером такой службы является файловая система. Объекты файловой системы – файлы – собраны в каталоги и идентифицируются путями, например, C:\windows\notepad.exe. У файлов есть атрибуты: скрытый, архивный, только для чтения и другие. Передавая файловой системе путь, можно получить содержимое соответствующего файла, записать в него какие-то данные, изменить его атрибуты.

JNDI предназначен для единообразного доступа к разнообразным службам имен и каталогов, включая упомянутые выше *DNS* и файловую систему, а также *LDAP*, о котором еще пойдет речь. Разные службы каталогов интегрируются с *JNDI* через интерфейс поставщика услуг (*Service Provider Interface*, *SPI*).

Первая редакция спецификация *JNDI* была выпущена корпорацией *Sun Microsystems* 10 марта 1997 г. В 2006 г. вышла спецификация *JNDI* версии 1.2.

JNDI состоит из следующих пяти пакетов:

- *javax.naming* – содержит основные классы и интерфейсы, необходимые для взаимодействия со службами имен. В частности, интерфейс `Context` для поиска объектов, привязки объекта к имени, создания и удаления контекстов.
- *javax.naming.directory* – расширяет пакет `javax.naming` средствами взаимодействия со службами каталогов. Определяет интерфейс `DirContext`, позволяющий работать с атрибутами объектов каталога.
- *javax.naming.event* – определяет классы и интерфейсы событий, происходящих в каталоге, а также средства перехвата этих событий.
- *javax.naming.ldap* – предоставляет средства для работы со специфическими возможностями *LDAP v3*, не покрываемыми более общим пакетом `javax.naming.directory`. Однако эти возможности редко используются, и в большинстве случаев достаточно использовать пакет `javax.naming.directory`.
- *javax.naming.spi* определяет способ интеграции новых систем имен или каталогов с *JNDI*, чтобы клиенты могли пользоваться этими службами из Java-программ средствами *JNDI*.

В этой главе мы рассмотрим использование *JNDI* для извлечения объектов из службы имен, а в следующей главе пойдет речь о доступе к *LDAP*.

4.3. Пример: использование *JNDI* для доступа к *DataSource*

Часто использование *JNDI* в программе ограничивается всего несколькими строками и играет вспомогательную, второстепенную роль. Типичный пример – подключение к источнику данных (data source). Следующий код, включенный в *EJB*, позволяет подключиться к

источнику данных, привязанному к имени "java:/DefaultDS". Этот источник данных существует в *JBoss* по умолчанию. Разумеется, можно настроить собственный источник данных и извлечь его из службы имен совершенно аналогично.

```
InitialContext ctx = new InitialContext();
DataSource ds =
(DataSource)ctx.lookup("java:/DefaultDS");
```

Работа с *JNDI* всегда начинается с создания объекта `InitialContext` (или `InitialDirContext` в случае работы со службой каталогов). Конструктор этого объекта может принимать параметры, определяющие, к какой службе и каким образом подключаться. В данном случае параметры в конструктор не передаются, поэтому происходит подключение к службе имен сервера JBoss, в котором запущен данный EJB. Затем вызов метода `lookup()` извлекает из службы имен объект, соответствующий имени "java:/DefaultDS".

Спецификация *EJB 3.0* еще более упростила задачу получения объектов из службы имен. Теперь для получения того же источника данных достаточно завести член класса *EJB* со специальной аннотацией, а об извлечении объекта из службы имен и присвоении его члену класса позаботится *EJB*-контейнер.

```
@Resource(mappedName="java:/DefaultDS")
private DataSource ds;
```

В следующем разделе будет приведен более содержательный пример использования *JNDI* для доступа к *LDAP* каталогу, хранящему информацию о пользователях.

4.4. Общие сведения о *LDAP*

Lightweight Directory Access Protocol (*LDAP*) – это сетевой протокол для доступа к каталогам (впрочем, иногда *LDAP* называют не только протокол, но и сам каталог). *LDAP* был разработан как замена более старому и тяжеловесному протоколу *DAP*, определенному в стандарте *X.500* и построенному на стеке протоколов *Open Systems Interconnection* (*OSI*).

В глобальной службе каталогов *X.500* может храниться информация о подразделениях компаний, людях, компьютерах, документах и т.д. Зная имя объекта, можно запросить в каталоге информацию о нем (*сервис белых страниц*, *white-pages service*), а можно просто просматривать каталог в поисках чего-то интересного (*сервис желтых страниц*, *yellow-pages service*).

Информация, содержащаяся в каталоге, составляет базу данных, называемую *directory information base* (*DIB*). Элементы *DIB* образуют дерево, называемое *directory information tree* (*DIT*). Каждый элемент имеет имя и набор типизированных атрибутов с их значениями. Схема каталога определяет обязательные и опциональные атрибуты для каждого класса объектов каталога.

Пространство имен каталога *X.500* является иерархическим – как файловая система. Каждый объект каталога однозначно идентифицируется уникальным именем, называемым *distinguished name* (*DN*). *DN* – это конкатенация значений некоторого атрибута всех объектов, находящихся на пути от корня дерева к искомому объекту (аналогия в файловой системе – путь к файлу). Указанный атрибут называется *relative distinguished name* (*RDN*).

Отличие *X.500* каталога от файловой системы в том, что имена объектов, *DN*, записываются справа налево, например, `cn=Alexey Vladykin,dc=ifmo,dc=ru`. Корень дерева здесь расположен справа, а лист – слева. В файловой системе наоборот: корень – слева, лист – справа, например, c:\windows\notepad.exe.

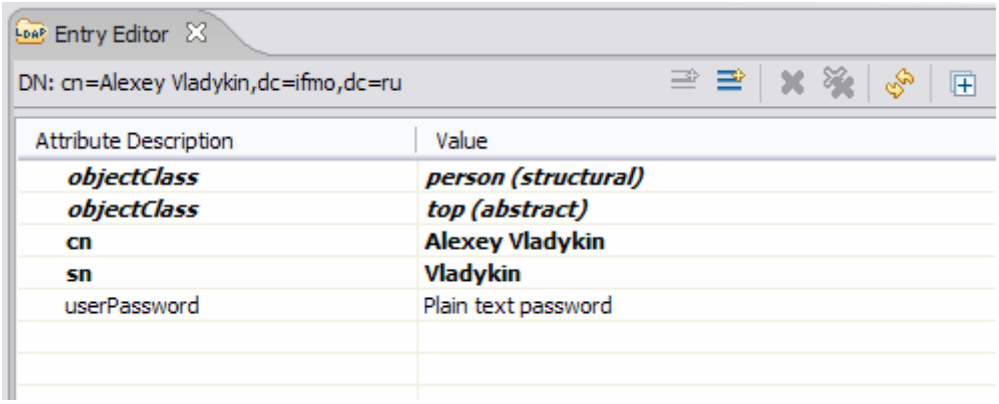
Пользователи каталога *X.500* могут, с учетом прав доступа, читать и изменять объекты и их атрибуты в *DIB*.

Недостатком стандарта *X.500* и определяемого в нем протокола доступа к каталогам *DAP* является то, что они основаны на стеке протоколов *OSI*, в то время как стандартом де-факто для Интернета стал более простой стек *TCP/IP*. В связи с этим потребовалась разработка службы каталогов и протокола доступа к ней, работающих на стеке протоколов *TCP/IP*. Решением проблемы стал *Lightweight DAP*, или просто *LDAP*, разработанный в 1992 г в Мичиганском университете. *LDAP* сохранил основные концепции *X.500*, такие как *DIT* и *DN*. Текущей версией протокола является *LDAP v3* (*RFC 2251*).

Одним из самых известных и широко распространенных *LDAP* серверов является *OpenLDAP* (http://www.openldap.org/), доступный бесплатно вместе с исходными кодами. Во время работы над этой главой автор использовал сборку *OpenLDAP* под *Windows*, взятую по адресу http://lucas.bergmans.us/hacks/openldap/.

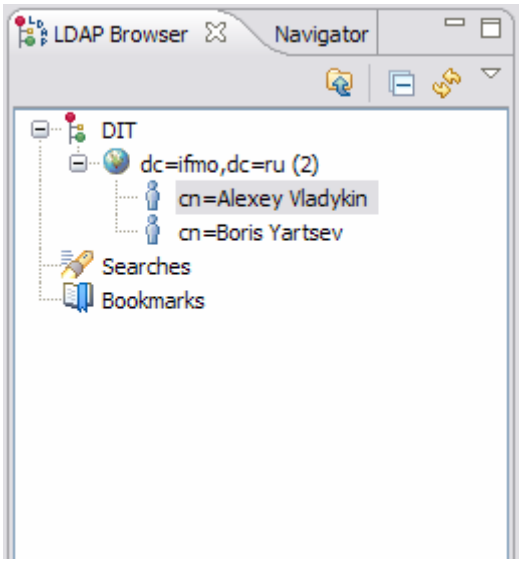
4.5. Пример: система авторизации пользователей на основе *LDAP*

LDAP часто используется для хранения информации о людях, в частности, о пользователях корпоративных сетей. Информация о человеке может храниться в *LDAP*-объекте класса `person` или расширяющих его `organizationalPerson` и `inetOrgPerson`. Например, автор данного текста может быть описан в *LDAP*-каталоге следующим образом:



В этом примере мы разработаем систему аутентификации/авторизации пользователей на основе *LDAP*. При этом информация о пользователях и их паролях будет храниться в *LDAP* каталоге. Предположим, что, как в примере, все пользователи в каталоге являются потомками узлом `dc=ifmo, dc=ru`, а их пароли хранятся в атрибуте `userPassword`.

Пример соответствующего *DIT* с двумя пользователями представлен на следующем рисунке.



Центром системы станет сеансовый компонент без состояния (*stateless session bean*), инкапсулирующий всю работу с каталогом и имеющий следующий удаленный интерфейс с единственным методом `authenticate()`:

```
package ru.ifmo.javaee.ldapauth;

import javax.ejb.Remote;

@Remote
```

```
public interface LdapAuth {

public boolean authenticate(String name, String
password);

}
```

Различные приложения могут обращаться к этому *EJB* для проверки того, что введенные пользователем имя и пароль являются верными.

Рассмотрим реализацию этого *EJB*.

```
package ru.ifmo.javaee.ldapauth;

import java.util.Hashtable;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.naming.Context;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchControls;

@Stateless
public class LdapAuthBean implements LdapAuth {

@Resource(name="providerUrl")
private String providerUrl;

@Resource(name="securityAuthentication")
private String securityAuthentication;

@Resource(name="securityPrincipal")
private String securityPrincipal;

@Resource(name="securityCredentials")
private String securityCredentials;

@Resource(name="baseContext")
private String baseContext;

private InitialDirContext ctx;

@PostConstruct
public void init() {
     Hashtable<String,     String>     args     =     new
     Hashtable<String, String>();
     args.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.sun.jndi.ldap.LdapCtxFactory");
     args.put(Context.PROVIDER_URL, providerUrl);
```

```
        args.put(Context.SECURITY_AUTHENTICATION,
        securityAuthentication);
        args.put(Context.SECURITY_PRINCIPAL,
        securityPrincipal);
        args.put(Context.SECURITY_CREDENTIALS,
        securityCredentials);
        try {
             ctx = new InitialDirContext(args);
        } catch (NamingException e) {
             e.printStackTrace();
        }
    }

    public boolean authenticate(String name, String
    password) {
        SearchControls controls = new SearchControls();

    controls.setSearchScope(SearchControls.SUBTREE_SCOPE);
        try {
             NamingEnumeration results = ctx.search(
                       baseContext,
                       "(&(cn={0})(userPassword={1}))",
                       new String[]{ name, password },
                       controls);
             return results.hasMore();
        } catch (NamingException e) {
             e.printStackTrace();
        }
        return false;
    }

    }
```

Интерес для нас представляют методы `init()` и `authenticate()`.

В методе `init()`, автоматически вызываемом после создания экземпляра компонента, осуществляется соединение с *LDAP*-каталогом. Для этого хеш-таблица `args` заполняется параметрами (так мы сообщаем *JNDI*, к какой службе подключаться), а затем создается начальный контекст `ctx`, представляющий собой корень каталога.

Метод `authenticate()` производит в каталоге поиск пользователя с заданным именем и паролем, используя возможности *JNDI* по поиску. В случае успеха возвращает `true`, иначе – `false`.

Перед развертыванием *EJB* требуется создать еще несколько файлов. Во-первых, ejb-jar.xml, задающий все параметры соединения с *LDAP*-каталогом, используемые `LdapAuthBean`:

```
<ejb-jar version="3.0"
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"
>

<enterprise-beans>
     <session>
          <ejb-name>LdapAuthBean</ejb-name>
          <env-entry>
               <env-entry-name>providerUrl</env-entry-
               name>
               <env-entry-type>java.lang.String</env-
               entry-type>
               <env-entry-
               value>ldap://localhost:389</env-entry-
               value>
          </env-entry>
          <env-entry>
               <env-entryname>
               securityAuthentication</env-entry-name>
               <env-entry-type>java.lang.String</env-
               entry-type>
               <env-entry-value>simple</env-entry-value>
          </env-entry>
          <env-entry>
               <env-entry-name>securityPrincipal</env-
               entry-name>
               <env-entry-type>java.lang.String</env-
               entry-type>
               <env-entry-value>cn=Administrator,
               dc=ifmo,dc=ru</env-entry-value>
          </env-entry>
          <env-entry>
               <env-entry-name>securityCredentials</env-
               entry-name>
               <env-entry-type>java.lang.String</env-
               entry-type>
               <env-entry-value>secret</env-entry-value>
          </env-entry>
          <env-entry>
               <env-entry-name>baseContext</env-entry-
               name>
               <env-entry-type>java.lang.String</env-
               entry-type>
               <env-entry-value>dc=ifmo,dc=ru</env-
               entry-value>
          </env-entry>
     </session>
</enterprise-beans>

</ejb-jar>
```

Во-вторых, jboss.xml, описывающий развертывание *EJB* для контейнера *JBoss*:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
<enterprise-beans>
     <session>
          <ejb-name>LdapAuthBean</ejb-name>
          <jndi-name>LdapAuth/Remote</jndi-name>
     </session>
</enterprise-beans>
</jboss>
```

Теперь *EJB* готов к развертыванию и тестированию. Для его тестирования можно использовать такую программу:

```
package ru.ifmo.javaee.ldapauth;

import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class LdapAuthClient {

public static void main(String[] args) {
     Hashtable<String,     String>     env     =     new
     Hashtable<String, String>();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
     "org.jnp.interfaces.NamingContextFactory");
     env.put(Context.PROVIDER_URL, "localhost:1099");
     try {
          InitialContext ctx = new InitialContext(env);
          LdapAuth        auth        =        (LdapAuth)
          ctx.lookup("LdapAuth/Remote");
          System.out.println(auth.authenticate(args[0],
          args[1]));
          ctx.close();
     } catch (NamingException e) {
          e.printStackTrace();
     }
}

}
```

Программа подключается к *EJB* (обратите внимание, для этого используется *JNDI*) и вызывает его метод `authenticate()`, передавая полученные с командной строки имя пользователя и пароль. Результат – `true` или `false` – выводится на экран.