Enterprise JavaBeans

Third Edition

Richard Monson-Haefel

O'REILLY®

Enterprise JavaBeans

Третье издание

Ричард Монсон-Хейфел

Санкт-Петербург
2002

Ричард Монсон-Хейфел

Enterprise JavaBeans, 3-е издание

Перевод И. Васильева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*В. Шальнев*
Редактор	*В. Овчинников*
Корректура	*С. Беляева*
Верстка	*А. Дорошенко*

Монсон-Хейфел Р.
Enterprise JavaBeans, 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 672 с., ил.
ISBN 5-93286-041-3

Третье издание «Enterprise JavaBeans» описывает технологию построения сложных ответственных систем, основанную на объединении компонентов, моделирующих прикладные объекты и процессы. EJB берет на себя заботу об объектном постоянстве, безопасности и управлении транзакциями. Еще недавно казалось невероятным, что компоненты EJB смогут не только выполняться в любой операционной системе без какой-либо модификации, но и работать на любом корпоративном сервере EJB. В EJB 2.0 введены компоненты, управляемые сообщениями, способные с помощью службы сообщений JMS взаимодействовать с системами промежуточного уровня; предложена более развитая модель управляемого контейнером постоянства и поддерживаются сложные отношения между объектными компонентами.

В книге рассматриваются: EJB 2.0 и 1.1, сеансовые и объектные компоненты (включая новую модель CMP и язык запросов EJB QL), компоненты, управляемые сообщениями, и служба JMS, XML-дескрипторы развертывания, управление транзакциями и безопасность, взаимосвязь EJB и Java 2, Enterprise Edition. Опытные разработчики корпоративных программных продуктов знают, как сильно технология EJB изменила эту область. Данное третье издание поможет им освоить последние достижения. А для тех, кто не имеет опыта работы с EJB, автор подготовил стартовую площадку, с которой они могут начать изучение этой захватывающей технологии построения прикладных систем.

ISBN 5-93286-041-3
ISBN 0-596-00226-2 (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 15.07.2002. Формат 70×100 1/16 . Печать офсетная.
Объем 42 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Заметки автора

Зимой 1997 года я консультировал один проект по электронной коммерции, использовавший Java RMI. Не удивительно, что проект провалился, потому что Java RMI не ориентирован на производительность, масштабируемость, надежность, безопасность и транзакции – на все то, что жизненно необходимо производственному окружению. Хотя исход этого проекта не является уникальным для Java RMI, я вижу, что то же самое происходит и с CORBA – время, в которое проводилась реализации этого проекта, было особенно интересным. Примерно в это же время Sun Microsystems впервые представила Enterprise JavaBeans™, и если бы Enterprise JavaBeans стал доступным немного раньше, этот проект, возможно, завершился бы успешно.

Во время работы над тем злополучным проектом я также вел колонку в «JavaReport Online» под названием «The Cutting Edge» (Срезанный угол). Колонка охватывала новые на тот момент технологии Java, такие как Java Naming and Directory Interface™ (JNDI) и JavaMail™ API. Я искал новую тему для третьего выпуска «Срезанного угла», когда обнаружил первый опубликованный черновой проект Enterprise JavaBeans версии 0.8. Первый раз я услышал об этой технологии в 1996 году, но открытая документация стала доступной только в начале 1997. Работая с CORBA, Java RMI и другими технологиями распределенных объектов, я понял, что это хорошая вещь, и сразу же начал свою статью об этой новой технологии.

Кажется, что прошла уже вечность. С тех пор как я опубликовал статью в марте 1998 года, о EJB были написаны буквально сотни статей и уже несколько книг пришло и ушло. Эта книга (уже третье издание) идет в ногу с тремя версиями спецификации EJB много лет. Сейчас, когда выходит очередная новая версия спецификации и появляется множество новых книг на эту тему, я не могу удержаться, чтобы не вспомнить те дни, когда слова «Enterprise JavaBeans» почти у всех вызывали непонимающие взгляды. И я рад, что эти времена прошли навсегда.

Что такое Enterprise JavaBeans?

Летом 1995 года, после первого представления Java, почти вся индустрия информационных технологий обратила внимание на возможности графического пользовательского интерфейса и конкурентоспособные преимущества, предлагаемые этой технологией с точки зрения переносимости и платформенной независимости. Это было интересное время. Лишь немногие из нас пробовали применять апплеты на стороне сервера. В действительности мы тратили половину нашего времени на программирование, а другую – на то, чтобы убедить руководство, что Java – это не прихоть моды.

Сегодня фокус значительно расширился: язык Java получил признание как отличная платформа для создания корпоративных решений, особенно при разработке распределенных серверных приложений. Такое смещение акцентов произошло благодаря становлению Java как универсального языка для создания независимых от реализации абстракций для всех распространенных корпоративных технологий. Программный интерфейс JDBC – первый и наиболее известный пример. JDBC (Java Database Connectivity) предлагает независимый от производителя Java-интерфейс для доступа к базам данных. Эта абстракция была такой успешной, что сейчас трудно найти производителя реляционных баз данных, который не поддерживал бы JDBC. Абстракции Java значительно расширились и включают JNDI для абстрактных служб каталогов, JTA (Java Transaction API – программный интерфейс транзакций) для абстрактного доступа к менеджерам транзакций, JMS (Java Message Service – служба сообщений) для абстрактного доступа к различным продуктам, ориентированным на сообщения.

Технология Enterprise JavaBeans впервые была представлена в виде рабочей спецификации в конце 1997 года и заявила о себе как о наиболее значительной из всех корпоративных технологий Java, предлагаемых компанией Sun Microsystems. EJB обеспечивает абстракцию для мониторов компонентных транзакций (CTM), представляющих собой сплав двух технологий: традиционных мониторов обработки транзакций (TP), таких как CLCS, TUXEDO и Encina, и служб распределенных объектов, таких как CORBA, DCOM и «родной» для Java RMI. Соединив в себе все самое лучшее из обеих технологий, мониторы компонентных транзакций предлагают мощное объектно-ориентированное окружение, которое упрощает распределенную разработку и при этом автоматически управляет наиболее сложными аспектами корпоративных вычислений, такими как взаимодействие между объектами, управление транзакциями, безопасность, постоянство и параллелизм.

Enterprise JavaBeans определяет модель серверных компонентов, позволяющую разрабатывать прикладные объекты и перемещать их между контейнерами EJB, созданными разными производителями. Ком-

понент (enterprise bean) представляет собой простую программную модель, которая позволяет разработчику сконцентрироваться на стоящей перед ним конкретной проблеме. А сервер EJB отвечает за то, чтобы сделать этот компонент распределенным объектом, и за управляющие службы, такие как транзакции, постоянство, параллелизм и безопасность. Кроме создания бизнес-логики компонента разработчик задает атрибуты поведения компонента во время выполнения, используя подход, похожий на выбор свойств отображения визуальных элементов. Поведение компонента по отношению к транзакциям, постоянству и безопасности может быть задано путем выбора из списка свойств. Конечный результат состоит в том, что EJB значительно облегчает создание систем распределенных компонентов, управляемых мощными транзакционными средами. Для разработчиков и корпоративных информационно-технических отделов, которые борются с проблемами развертывания надежных, производительных распределенных систем на базе CORBA, DCOM и Java RMI, EJB предоставляет в качестве основы для программирования более простую и более производительную платформу.

В 1998 году была завершена спецификация Enterprise JavaBeans 1.0, которая фактически сразу же стала промышленным стандартом. Многие производители объявили о поддержке спецификации еще до ее завершения. С того времени EJB дважды улучшалась. Первый раз спецификация была обновлена в 1999 году до версии 1.1, которая была рассмотрена во втором издании этой книги. Самый последний вариант спецификации – версия 2.0 – описывается в этом, третьем издании «Enterprise JavaBeans», охватывающем также версию EJB 1.1, которая, по большей части, является подмножеством функциональных возможностей, предлагаемых EJB 2.0.

Продукты, соответствующие стандартам EJB, приходят из всех секторов информационной индустрии, в числе которых мониторы обработки транзакций (TPM), CORBA ORB, серверы приложений, реляционные и объектные базы данных, веб-серверы. Некоторые из этих продуктов базируются на собственных моделях, адаптированных под EJB.

Вкратце, обе версии – Enterprise JavaBeans 2.0 и 1.1 – представляют стандартную модель распределенных компонентов, которая значительно упрощает процесс разработки и позволяет компонентам, созданным и развернутым на сервере EJB одного производителя, с легкостью быть развернутыми на сервере другого производителя. Эта книга дает основу, необходимую для создания независимых от производителя EJB-проектов.

Для кого предназначена эта книга?

Эта книга объясняет и демонстрирует основные принципы архитектур Enterprise JavaBeans 2.0 и 1.1. Хотя EJB сильно упрощает распреде-

ленные вычисления, она достаточно сложна и требует много времени для овладения ею на уровне мастера. Книга понятно и последовательно разъясняет лежащую в основе технологию, классы и интерфейсы Java, компонентную модель и поведение компонентов во время выполнения программы. Она содержит материал, обратно совместимый с EJB 1.1, и включает отдельные замечания и главы, охватывающие наиболее значимые различия между версиями 1.1 и 2.0.

Хотя эта книга сфокусирована на основах, она не предназначена для начинающих. Enterprise JavaBeans – крайне сложная и амбициозная корпоративная технология. Применение EJB может быть достаточно простым, но для ее полного освоения и правильного понимания требуется очень много времени. Перед тем как читать эту книгу вы должны свободно овладеть языком Java и получить практический опыт создания бизнес-проектов. Опыт работы с системами распределенных объектов не требуется, но для того чтобы разобраться с примерами, приведенными в книге, необходимо определенное знание (или хотя бы понимание основ) технологии JDBC. Если вы не знакомы с языком Java, я рекомендую книгу «Learning Java» (Изучаем Java) Патрика Нимайера (Patrick Niemeyer) и Джонатана Кнудсена (Jonathan Knudsen), которая первоначально называлась «Exploring Java». Тем же, кто не знаком с JDBC, рекомендую книгу «Database Programming with JDBC and Java» (Программирование баз данных с помощью JDBC и Java) Джорджа Риза (George Reese), а тем, кому нужны прочные знания распределенных вычислений – «Java Distributed Computing» (Распределенные вычисления на Java) Джима Фарли (Jim Farley).

Структура книги

Первые три главы содержат общий ознакомительный материал, рассматривающий Enterprise JavaBeans 2.0 и 1.1 в контексте других родственных технологий и объясняющий на максимально отвлеченном уровне, как работает технология EJB и из чего состоят компоненты EJB. В главах с 4 по 13 подробно объяснено, как разрабатываются компоненты различных типов. Главы 14 и 15 могли бы считаться материалом для дополнительного изучения, если бы не транзакции (глава 14), которые существенны для всего, что происходит в корпоративных вычислениях, и стратегии дизайна (глава 15), помогающие справляться с реальными проблемами, влияющими на разработку компонентов. В главе 16 подробно описан дескриптор развертывания на базе XML, используемый в EJB 2.0 и 1.1. Глава 17 дает общий обзор пакета Java 2, Enterprise Edition (J2EE), включающего сервлеты, Java Server Pages (JSP) и EJB. Наконец, три приложения предоставляют справочную информацию, которая может быть вам полезна.

Глава 1. «Введение»

Здесь вводится определение мониторов компонентных транзакций и объясняется, каким образом они формируют технологию, лежащую в основе модели компонентов Enterprise JavaBeans.

Глава 2. «Обзор архитектуры»

Вводится понятие архитектуры компонентной модели Enterprise JavaBeans и рассматриваются различия между тремя основными типами компонентов: объектными компонентами, сеансовыми компонентами и компонентами, управляемыми сообщениями.

Глава 3. «Управление ресурсами и основные службы»

Объясняется, как EJB-совместимый сервер управляет компонентами во время выполнения.

Глава 4. «Создание вашего первого компонента»

В этой главе описываются все этапы разработки простого компонента.

Глава 5. «Клиентское представление»

Содержит подробный рассказ о том, как удаленное клиентское приложение получает доступ и использует компоненты.

Глава 6. «EJB 2.0 CMP: Основы постоянства»

Здесь объясняется, как в EJB 2.0 создать простой объектный компонент, управляемый контейнером.

Глава 7. «EJB 2.0 CMP: Отношения между объектами»

В этой главе продолжается тема, начатая в главе 6, расширяя границы вашего понимания от постоянства, управляемого контейнером, до сложных взаимоотношений между компонентами.

Глава 8. «EJB 2.0 CMP: EJB QL»

Эта глава обращается к языку запросов Enterprise JavaBeans (EJB Query Language, EJB QL), который применяется для обращения к компонентам и для поиска определенных компонентов при использовании постоянства, поддерживаемого контейнером EJB.

Глава 9. «EJB 1.1 CMP»

Описывается постоянство, управляемое контейнером в EJB 1.1, которое для обратной совместимости поддерживается и в EJB 2.0. Прочитайте эту главу, если только вам нужно поддерживать существующие EJB-приложения.

Глава 10. «Постоянство, реализуемое компонентом»

Здесь рассказывается о создании компонентов, самостоятельно реализующих постоянство (bean-managed persistence), в том числе о том, когда сохранять, загружать и удалять данные из базы данных.

Глава 11. «Соглашения между объектом и контейнером»

Эта глава описывает общий протокол взаимодействия между объектным компонентом и его контейнером во время выполнения, применяемый для реализации постоянства, поддерживаемого контейнером как в версии 2.0, так и в 1.1.

Глава 12. «Сеансовые компоненты»

В этой главе показано, как создавать сеансовые компоненты с состоянием и без состояния.

Глава 13. «Компоненты, управляемые сообщениями»

Показано, как в EJB 2.0 создавать компоненты, управляемые сообщениями.

Глава 14. «Транзакции»

Эта глава дает глубокое объяснение транзакций и описывает модель транзакций, введенную в Enterprise JavaBeans.

Глава 15. «Стратегии дизайна»

Даются некоторые основные стратегии дизайна, уменьшающие усилия, требуемые для разработки компонентов и делающие вашу систему EJB более эффективной.

Глава 16. «XML-дескрипторы развертывания»

В этой главе представлено подробное описание XML-дескрипторов развертывания, используемых в EJB 1.1 и 2.0.

Глава 17. «Java 2, Enterprise Edition»

Приводится обзор J2EE 1.3 и объясняется, как EJB 2.0 встраивается в эту новую платформу.

Приложение A. «Программный интерфейс Enterprise JavaBeans»

Данное приложение представляет собой краткий справочник по классам и интерфейсам, определенным в пакетах EJB.

Приложение B. «Диаграммы состояний и последовательностей»

Представлены диаграммы, поясняющие жизненный цикл компонентов во время выполнения.

Приложение C. «Производители EJB»

В этом приложении приводится информация о производителях серверов EJB.

Программы и версии

Эта книга описывает Enterprise JavaBeans версий 2.0 и 1.1, включая все дополнительные возможности. В ней используются особенности языка Java из платформы Java 1.2 и JDBC. Ввиду того что данная книга нацелена на создание компонентов Enterprise JavaBeans и проек-

тов, не зависящих от конкретного производителя, я старался избегать нестандартных расширений и диалектов, специфичных для определенного производителя. Для чтения этой книги достаточно любого совместимого с EJB сервера, но работа с примерами требует знания процедур установки, развертывания и управления сервером во время выполнения.

EJB 2.0 и 1.1 имеют много общего, но там, где они отличаются, главы или разделы глав, специфичные для каждой версии, выделены особо. Можете пропускать те разделы, которые вас не касаются. Если не указано обратное, исходные коды из этой книги будут работать как для EJB 2.0, так и для EJB 1.1.

Примеры, приведенные в этой книге, доступны на сайте *ftp://ftp.oreilly.com/pub/examples/java/ejb*. Примеры сгруппированы по главам.

Рабочие книги с примерами

Хотя EJB-приложения сами по себе переносимы, способы установки и запуска EJB-продуктов варьируются от производителя к производителю. По этой причине не представлялось возможным охватить все доступные EJB-продукты и был выбран радикальный, но в то же время эффективный способ, позволяющий рассмотреть эти различия, – рабочие упражнения (workbooks).

Для того чтобы помочь читателям установить примеры из книги на различные EJB-продукты, я опубликовал несколько бесплатных рабочих упражнений, которые можно использовать вместе с этой книгой для запуска примеров на конкретных коммерческих и некоммерческих серверах. Рабочие книги для конкретного продукта будут относиться к самой последней версии сервера. Так, если производитель поддерживает EJB 2.0, пример из рабочей книги будет использовать особенности EJB 2.0. С другой стороны, если производитель поддерживает только EJB 1.1, примеры будут специфичными для версии 1.1.

Я планирую опубликовать рабочие книги для нескольких различных серверов EJB, и по крайней мере две из них будут доступны в самое ближайшее время. Эти книги будут бесплатно доступны (в формате PDF) на *http://www.oreily.com/catalog/entjbeans3/* и *http://www.titanbooks.com*.

Соглашения

В книге приняты следующие типографские соглашения:

Курсив

Предназначен для имен файлов и путей, имен хостов, доменов, URL и адресов электронной почты. Курсивом также выделяются вновь вводимые термины.

`Моноширинный`

Используется в примерах программ и их фрагментов, для элементов XML, тегов, команд SQL, имен таблиц и колонок. Моноширинный шрифт применяется также для имен классов, переменных методов и для ключевых слов Java, используемых в тексте.

`Моноширинный полужирный`

Служит для выделения в некоторых примерах программ.

`Моноширинный курсив`

Указывает на то, что текст должен быть заменен. Например, в слове *`BeanName`*`PK` следует заменить *`BeanName`* конкретным именем.

Указывает на подсказки, советы, общие замечания.

Означает предупреждение или предостережение.

Компонент Enterprise JavaBeans состоит из нескольких частей. Это не один объект, а набор объектов и интерфейсов. Для ссылки на компонент как на единое целое мы будем использовать его прикладное имя, набранное основным шрифтом. Например, будем ссылаться на компонент Customer тогда, когда нам нужно обозначить компонент вообще. Если мы набираем имя моноширинным шрифтом, значит, мы явно ссылаемся на удаленный интерфейс. Таким образом, `CustomerRemote` представляет собой удаленный интерфейс, определяющий прикладные методы компонента Customer.

Комментарии и вопросы

Пожалуйста, направляйте комментарии и вопросы, касающиеся этой книги, в издательство:

O'Reilly & Associaties, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (для США и Канады)
(707) 829-0515 (международный и местный)
(707) 829-0104 (факс)

У этой книги есть веб-страница, содержащая найденные опечатки, примеры и некоторую дополнительную информацию. Ее можно найти по адресу:

http://www.orielly.com/catalog/entjbeans3/

Комментарии и технические вопросы направляйте по адресу:

bookquestions@orielly.com

За дополнительной информацией о книгах, конференциях, программных продуктах, центрах ресурсов и сети O'Reilly обращайтесь на вебсайт O'Reilly:

http://www.orielly.com

Автор поддерживает сайт для дискуссий по EJB и связанным с ней распределенным компьютерным технологиям на *http://www.jmiddleware.com*. *jMiddleware.com* предлагает последние новости по этой книге, а также подсказки по программированию, статьи и большой список ссылок на ресурсы EJB.

Благодарности

Хотя на обложку этой книги вынесено только одно имя, к ее написанию и распространению приложили усилия многие люди. Успех каждого издания этой книги зависел в основном от Майкла Лукидеса (Michael Loukides). Без его опыта, ловкости и руководства этой книги просто не было бы. Другие люди из O'Reilly & Associates, включая Рэчел Уилер (Rachel Wheeler), Роба Романо (Rob Romano), Кайла Харта (Kyle Hart) и многих других, также много сделали для успеха этой книги.

Многие технические редакторы помогли обеспечить то, что материал о сущности Enterprise JavaBeans был технически точным и правильным. Отдельно нужно упомянуть Грега Найберга (Greg Nyberg) из Object Partners, Хеманта Ханделвала (Hemant Khandelwal) из Pramati, Кайла Брауна (Kyle Brown) из IBM, Роберта Кастанеда (Robert Castaneda) из Custom Ware, Джоя Фиалли (Joy Fially) из Sun Microsystems, Анила Шарму (Anila Sharma) и Прайенка Растджи (Priyank Rastogi) из Pramanti, Сева Уйта (Seth White) из BEA, Эвана Ирланда (Evan Ireland) и Майера Тануана (Meyer Tanuan) из Subase, Дэвида Чаппела (David Chappell) из David Chappell & Associates, Джима Фарли (Jim Farley) – автора «Java Distributed Computing» (O'Reilly) и Прасада Муппиралу (Prasad Muppirala) из BORN Information Services. Они во многом способствовали обеспечению технической точности книги и вложили свой профессиональный и жизненный опыт, помогая сделать ее лучшей книгой по Enterprise JavaBeans из выпускаемых сегодня.

Я хотел бы поблагодарить всех тех, кто присылал ценные замечания по книге во время ее написания по адресу *http://orielly.techrev.org*, включая (в алфавитном порядке):

Диона Алмаера (Dion Almaer), Джима Арчера (Jim Archer), Стефана Дейвиса (Stephen Davis), Джона Де Ла Круза (John De La Cruz), Тома Гулло (Tom Gullo), Марти Харви (Marty Harvey), Хая Хонга (Hai Hoang), Мирея Куннумпурата (Meeraj Kunnumpurath), Тома Ларсона (Tom Larson), Бьерна Расмуссена (Bjarne Rassmussen), rgparker (имя неизвестно), Лари Сельцера (Larry Seltzer) и Курта Смита (Curt Smith).

Отдельные благодарности Шрираму Шринивасону (Sriram Srinivason) из BEA, Анне Томас (Anne Tomas) из Sun Microsystems, Яну Маккалиону (Ian McCallion) из IBM Hursley, Тиму Рохайли (Tim Rohaly) из jGuru.com, Джеймсу Френтрессу (James D Frentress) из корпорации ITM, Анджее Джан Тарамину (Andrzej Jan Taramina) из Accredo Systems, Марку Лою (Marc Loy), соавтору книги «Java Swing», Дону Уэйсу (Don Weiss) из Step 1, Майку Слину (Mike Slinn) из корпорации The Dialog и Кэвину Дику (Kevin Dick) из Kevin Dick & Associates. Помощь этих технических экспертов была очень важна для технической и принципиальной точности первых изданий этой книги. Кроме этого, я хотел бы поблагодарить Мегги Мескита (Maggie Mezquita), Грэга Хартсела (Greg Hartzel), Джона Клуга (John Klug) и Яна Джамсу (Jon Jamsa) из BORN Information, которые просмотрели первый вариант первого издания и составили ценные замечания.

Также хочу поблагодарить Влада Матену (Vlad Matena) и Марка Хапнера (Mark Hapner) из Sun Microsystem, главных архитекторов Enterprise JavaBeans, Линду Де Мишель (Linda DeMichiel), руководителя спецификации EJB 2.0, и Бони Кельта (Bonnie Kellett), менеджера программы J2EE, за их желание ответить на мои самые сложные вопросы. Благодарю всех участников списка рассылки EJB-INTEREST, поддерживаемого компанией Sun Microsystems за их интересные и иногда противоречивые, но всегда информативные послания за последние четыре года.

И наконец, выражаю самую искреннюю благодарность моей жене Холли за помощь и поддержку в течение трех лет мучительных исследований и написания всего того, что потребовалось для создания трех изданий этой книги. Без ее неизменной поддержки и любви эта книга не была бы завершена.

1

Введение

Эта книга об Enterprise JavaBeans 2.0 и 1.1, представляющих собой третью и вторую версии спецификации Enterprise JavaBeans. Точно так же, как платформа Java кардинально изменила наши представления о разработке программного обеспечения, Enterprise JavaBeans изменила представления о разработке надежных корпоративных программных систем. Она сочетает серверные компоненты с технологиями распределенных объектов и системами асинхронных сообщений, что значительно упрощает задачу разработки приложений. Кроме того, Enterprise JavaBeans автоматически учитывает многие требования прикладных систем, включая безопасность, поддержание пула ресурсов (resource pooling), постоянство, параллелизм и целостность транзакций.

В книге показано, как использовать Enterprise JavaBeans для создания масштабируемых, переносимых прикладных систем. Но перед тем как начать разговор о самой EJB, необходимо вкратце рассмотреть технологии, связанные с EJB, такие как модели компонентов, распределенные объекты, мониторы компонентных транзакций (Component Transaction Monitor, CTM) и системы асинхронных сообщений. Особенно важно иметь общее представление о мониторах компонентных транзакций, технологии, лежащей в основе EJB. В главах 2 и 3 мы начнем рассмотрение самой технологии EJB и увидим, как ее компоненты работают вместе. Остаток этой книги посвящен разработке компонентов для вымышленного предприятия и обсуждению сложных вопросов.

Предполагается, что читатель уже знаком с Java, а если нет, то книга «Learning Java» (Изучаем Java) Патрика Нимайера и Джона Пека будет превосходным введением в тему. Предполагается также, что вы знакомы с интерфейсом JDBC или, по крайней мере, с SQL. Если же нет – взгляните на «Database programming with JDBC and Java» (Программирование баз данных с помощью JDBC и Java) Джорджа Риза (George Reese).

Одна из наиболее важных особенностей Java – ее платформенная независимость. Со времени своего первого выпуска Java присутствует на рынке под лозунгом «написать однажды, использовать везде». Хотя эта реклама временами кажется немного неуклюжей, код, написанный на языке Java, на удивление независим от платформы. Enterprise JavaBeans не просто не зависит от платформы, она также не зависит и от реализации. Тот, кто работал с JDBC, должен немного представлять, что это значит. Интерфейс JDBC может не только работать на машинах с Windows или UNIX, он также может обращаться к реляционным базам данных от множества различных производителей (DB2, Oracle, Sybase, SQLServer и т. д.), используя различные драйверы JDBC. Нет необходимости программировать конкретную реализацию базы данных, просто замените драйвер JDBC, и будет заменена база данных. То же самое и с EJB. В идеале компонент EJB (enterprise bean) может выполняться на любом сервере приложений, реализующем спецификацию EJB.[1] Это значит, что можно разработать и развернуть прикладную систему на одном сервере, таком как Orion или Weblogic от BEA, а позже переместить ее на другой сервер EJB, такой как Pramati, Sybase EAServer, WebSphere от IBM или на проект с открытым кодом, например OpenEJB, JonAS или Jboss. Независимость от реализации означает, что прикладные компоненты не зависят от марки используемого сервера, что дает свободу выбора до, во время и после разработки и развертывания.

Подготовительный этап

Прежде чем дать более точное определение Enterprise JavaBeans, обсудим некоторые важные концепции: распределенные объекты, прикладные объекты, мониторы компонентных транзакций и системы асинхронных сообщений.

Распределенные объекты

Благодаря распределенным вычислениям прикладные системы становятся более доступными. Распределенные системы позволяют своим

[1] При условии, что и компоненты и серверы EJB соответствуют этой спецификации и при разработке не используются никакие нестандартные расширения.

отдельным частям размещаться на разных компьютерах, возможно, расположенных в разных местах, там, где это необходимо. Другими словами, распределенные вычисления обеспечивают доступность прикладной логики и данных из удаленных мест. Клиенты, деловые партнеры и другие удаленные группы пользователей могут работать с прикладной системой в любое время и почти отовсюду. Самая последняя разработка в области распределенных вычислений – *распределенные объекты* (*distributed objects*). Благодаря технологиям распределенных объектов, таким как Java RMI, CORBA и .NET от Microsoft, объекты, работающие на одной машине, могут использоваться клиентскими приложениями, выполняющимися на других компьютерах.

Распределенные объекты возникли на основе достаточно старой модели трехуровневой архитектуры, применяемой в системах с мониторами обработки транзакций (TP), таких как CICS от IBM и TUXEDO от BEA. Эти системы разбиваются на три отдельных уровня, или слоя. В прошлом эти системы состояли из «зеленых экранов», или «тупых» (dumb) терминалов, на уровне представления (первый уровень), приложений на языках COBOL или PL/1 в качестве промежуточного уровня (второй уровень) и какой-нибудь базы данных, например DB2, в качестве источников данных (третий уровень). Появление в последние годы распределенных объектов привело к появлению новой формы трехуровневой архитектуры. Технологии распределенных объектов позволили заменить процедурные приложения второго уровня, использующие COBOL и PL/1, новыми прикладными объектами. Трехуровневая архитектура распределенных прикладных объектов может иметь изощренный графический или веб-интерфейс в качестве первого уровня, прикладные объекты на втором уровне и реляционные либо другие базы данных в качестве хранилища данных. Более сложные архитектуры часто имеют больше уровней: различные объекты размещаются на разных серверах и взаимодействуют между собой, выполняя общую задачу. Создание таких многоуровневых архитектур с помощью Enterprise JavaBeans выполняется сравнительно легко.

Серверные компоненты

Объектно-ориентированные языки, такие как Java, C++ и Smalltalk, применяются для написания программ, являющихся гибкими, расширяемыми и пригодными для повторного использования, – это три аксиомы объектно-ориентированного подхода. В прикладных системах объектно-ориентированные языки служат для ускорения разработки пользовательского интерфейса, упрощения доступа к данным и для инкапсуляции прикладной логики. Инкапсуляция прикладной логики в *прикладные объекты* – достаточно новое направление в области информационных технологий. Бизнес очень мобилен, а это значит, что бизнес-продукты, процессы и цели находятся в постоянном развитии. Если программа, моделирующая бизнес-процесс, может

быть инкапсулирована или встроена в прикладной объект, она становится гибкой, расширяемой и пригодной для повторного использования и, таким образом, может развиваться вместе с бизнесом.

Модель серверных компонентов способна задать архитектуру для разработки *распределенных прикладных объектов* (*distributed business objects*), которые сочетают в себе доступность систем распределенных объектов с изменчивостью, воплощенной в их прикладной логике. Модель серверных компонентов используется в серверах приложений промежуточного уровня, которые управляют компонентами во время выполнения и делают их доступными для удаленных клиентов. Они предоставляют основу функциональности, облегчающую разработку распределенных прикладных объектов и сборку их в прикладной проект.

Серверные компоненты также могут применяться и для моделирования других аспектов прикладных систем, таких как представление и маршрутизация. Сервлеты Java, например, являются серверными компонентами, предназначенными для создания данных в форматах HTML и XML, используемых на уровне представления трехуровневой архитектуры. Компоненты EJB 2.0, управляемые сообщениями, обсуждаемые далее, являются серверными компонентами, используемыми для приема и обработки асинхронных сообщений.

Серверные компоненты, как и другие компоненты, могут продаваться и покупаться в качестве независимых частей исполняемых программных продуктов. Они соответствуют стандартной модели компонентов и могут без непосредственной модификации работать на сервере, поддерживающем эту компонентную модель. Модели серверных компонентов часто поддерживают программирование, основанное на атрибутах (attribute-based programming), позволяющее изменять поведение компонента во время выполнения при его развертывании, без необходимости изменения его программного кода. В зависимости от используемой компонентной модели администратор сервера может задать поведение объекта, относящееся к транзакциям, безопасности и даже постоянству, задав для этих атрибутов определенные значения.

По мере развития корпоративных служб, продуктов и алгоритмов работы серверные компоненты могут перекомпоновываться, изменяться и расширяться так, чтобы отражать изменения, происходящие в бизнес-системе. Представим себе прикладную систему в виде набора серверных компонентов, моделирующих такие понятия, как клиенты, продукты, заказы и товарные склады. Каждый компонент похож на блок из набора Lego, который может комбинироваться с другими компонентами для построения прикладных проектов. Продукты могут храниться на складах или отправляться клиентам, клиенты могут делать заказы и покупать продукты. Компоненты можно собирать вместе, раскладывать их по отдельности, использовать в различных комби-

нациях и изменять их определения. Прикладная система, основанная на серверных компонентах, обладает гибкостью благодаря разделению ее на объекты и доступностью благодаря тому, что компоненты могут быть рассредоточены.

Мониторы компонентных транзакций

Не так давно возникло новое поколение программных продуктов, называемых *серверами приложений* (*application server*), направленных на решение проблем, связанных с созданием прикладных систем в современном мире Интернета. Как правило, прикладной сервер состоит из некоторого набора нескольких различных технологий, таких как веб-серверы, посредники объектных запросов (Object Request Broker, ORB), программное обеспечение, ориентированное на сообщение (Message-Oriented Middleware, MOM), базы данных и пр. Сервер приложений может быть также ориентирован только на одну технологию, такую как технология распределенных объектов. Возможности серверов приложений, базирующиеся на распределенных объектах, сильно варьируются. Самыми простыми являются посредники ORB, обеспечивающие связь между клиентскими приложениями и распределенными объектами. ORB позволяют клиентским приложениям без труда находить и использовать распределенные объекты. Однако ORB часто оказываются непригодными в массивных транзакционных окружениях. Они предоставляют коммуникационный каркас для распределенных объектов, но не могут обеспечить сколько-нибудь мощную инфраструктуру, способную справиться с большим количеством пользователей и гарантировать надежную работу. Кроме этого, ORB предлагает недостаточно проработанную модель серверных компонентов, которая перекладывает обязанности по обработке транзакций, параллелизма, постоянства и другие задачи системного уровня на плечи разработчика приложения. Все эти службы не поддерживаются автоматически в ORB. Разработчики приложений должны напрямую обращаться к ним (если они доступны) или, в некоторых случаях, создавать их с нуля.

В начале 1999 года Анна Мейнс (Anne Manes)[1] ввела в обращение термин *монитор компонентных транзакций* (*Component Transaction Monitor, CTM*) для того, чтобы описать наиболее развитый прикладной сервер распределенных объектов. CTM возникли как сплав традиционных TP-мониторов и технологии ORB. Они реализуют мощную модель серверных компонентов, которая помогает разработчикам создавать, использовать и развертывать прикладные системы. CTM предлагают инфраструктуру, которая автоматически управляет транзак-

[1] Когда Мейнс ввела этот термин, она работала в Patricia Seybold Group под именем Анны Томас. Сейчас Мейнс является директором Business Strategy for Sun Microsystem – подразделения Sun Software.

циями, распределением объектов, параллелизмом, безопасностью, постоянством и контролем за ресурсами. Они способны управлять огромным числом пользователей и обеспечивать надежную работу системы и, кроме этого, представляют ценность для малых систем из-за легкости в обращении. СТМ – это законченные серверы приложений. Другие термины, применяемые к этой технологии, включают в себя: монитор объектных транзакций (Object Transaction Monitor, OTM), сервер компонентных транзакций, сервер распределенных компонентов и COM-ware. В данной книге используется термин «монитор компонентных транзакций», – из-за того, что он охватывает три ключевые характеристики этой технологии: применение компонентной модели, направленность на управление транзакциями и управление ресурсами и сервисами, которое обычно связывают с мониторами.

Определение Enterprise JavaBeans

Компания Sun Microsystems определяет Enterprise JavaBeans следующим образом:

> Архитектура Enterprise JavaBeans – это компонентная архитектура, предназначенная для разработки и развертывания основанных на компонентах распределенных бизнес-приложений. Приложения, созданные с помощью архитектуры Enterprise JavaBeans, 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 Enterprise JavaBeans.[1]

Весьма многословно, но достаточно типично для определений, которые Sun дает многим Java-технологиям. Вы когда-нибудь читали определение для самого языка Java? Оно примерно в два раза длиннее. Здесь предлагается сокращенное определение EJB:

> Enterprise JavaBeans – это стандартная модель серверных компонентов для мониторов компонентных транзакций.

Мы подготовили почву для этого, определив кратко термины «распределенные объекты», «серверные компоненты» и «мониторы компонентных транзакций». Построим прочный фундамент для изучения Enterprise JavaBeans, для чего далее в этой главе и раскроем все эти определения.

Те, кто уже ясно представляет себе, что такое распределенные объекты, мониторы транзакций, СТМ и системы асинхронных сообщений, могут пропустить остаток этой главы и перейти к главе 2.

---

[1] Спецификация Enterprise JavaBeans версия 2.0, Copyright 2001, Sun Microsystems.

Архитектуры распределенных объектов

EJB – это компонентная модель для мониторов компонентных транзакций, основанных на технологиях распределенных объектов. Следовательно, для того чтобы понять EJB, необходимо представлять, как работают распределенные объекты. Системы распределенных объектов являются фундаментом для современных трехуровневых архитектур. Как показано на рис. 1.1, логика представления расположена на стороне клиента (первый уровень), прикладная логика размещена на промежуточном уровне (второй уровень), а другие ресурсы, такие как базы данных, – на стороне источников данных (третий уровень).

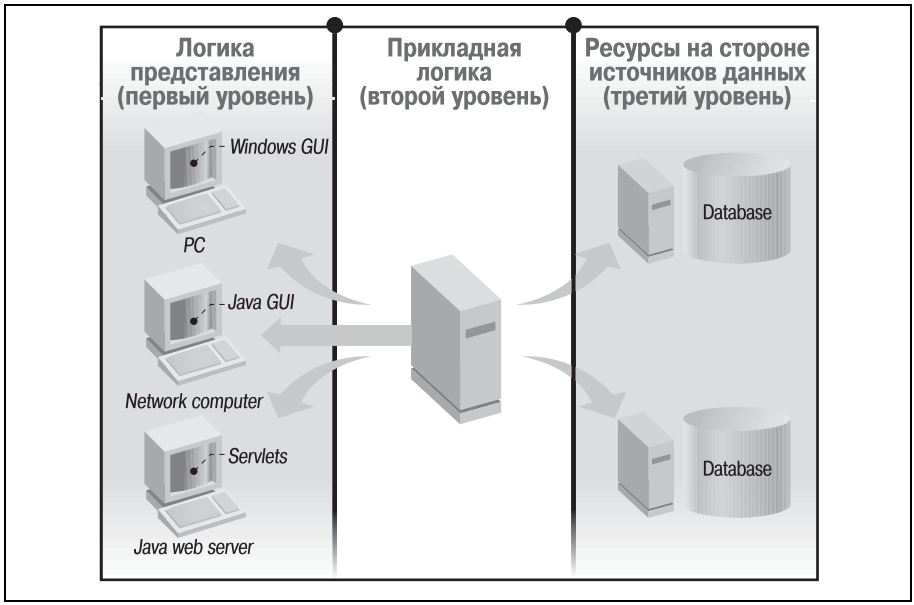

Рис. 1.1. Трехуровневая архитектура

Все протоколы распределенных объектов построены на этой же базовой архитектуре, спроектированной так, чтобы объект, расположенный на одном компьютере, казался расположенным на другом. Архитектура распределенных объектов опирается на уровень сетевых коммуникаций, который в действительности крайне прост. По существу эта архитектура состоит из трех частей: прикладного объекта, каркаса и заглушки.

Прикладной объект (*business object*) располагается на промежуточном уровне. Это экземпляр объекта, моделирующий состояние и логику некоторого реально существующего понятия, такого как человек, заказ или расчетный счет. У каждого класса прикладных объектов есть соответствующие ему классы заглушки и каркаса, специально созданные для этого типа прикладного объекта. Например, прикладной объ-

ект с именем `Person` будет иметь соответствующие классы `Person_Stub` и `Person_Skeleton`. Как показано на рис. 1.2, прикладной объект и каркас расположены на втором уровне, а заглушка – на клиентском.

Заглушка (*stub*) и *каркас* (*skeleton*) отвечают за то, чтобы прикладной объект, находящийся на промежуточном уровне, выглядел так, как будто бы он выполняется локально на клиентской машине. Это достигается за счет применения некоторого протокола, называемого *удаленным вызовом методов* (*Remote Method Invocation, RMI*). Протокол RMI служит для передачи вызовов методов через сеть. CORBA, Java RMI и Microsoft .NET пользуются своими собственными протоколами.[1] Каждый экземпляр прикладного объекта, находящийся на втором уровне, обернут (wrapped) экземпляром соответствующего ему класса каркаса. Каркас настраивается на определенный порт и IP-адрес и ожидает запросы от заглушки, находящейся на клиентской машине и соединяющейся с каркасом через сеть. Заглушка действует в качестве заместителя прикладного объекта на клиентской машине и отвечает за передачу запросов от клиента прикладному объекту при помощи каркаса. Рис. 1.2. иллюстрирует передачу вызова метода от клиента серверному объекту и обратно. Заглушка и каркас скрывают детали передачи, специфичные для RMI-протокола, от клиента и класса реализации соответственно.

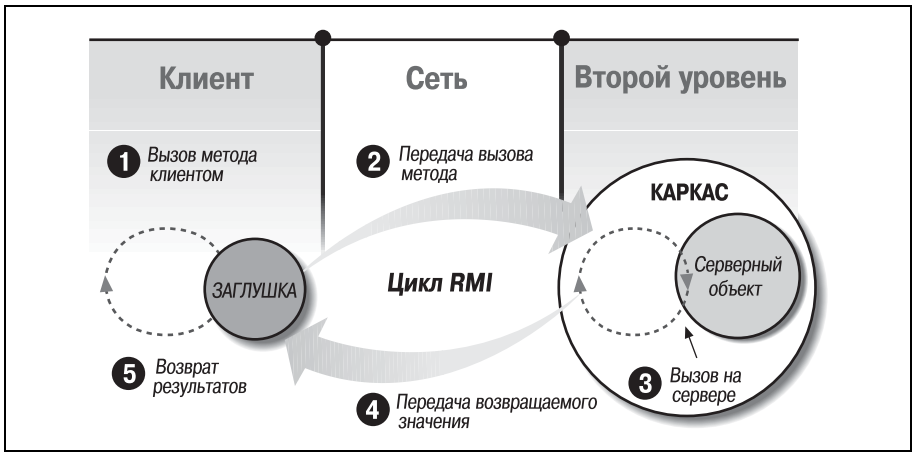

Рис. 1.2. Цикл RMI

Прикладной объект реализует открытый интерфейс, объявляющий его прикладные методы. Заглушка реализует тот же самый интерфейс, что и прикладной объект, но методы заглушки не содержат

[1] Аббревиатура RMI не относится только к Java RMI. В этом разделе термин RMI используется для описания протоколов распределенных объектов вообще. Java RMI представляет собой версию протокола распределенных объектов языка программирования Java.

прикладной логики. Вместо этого прикладные методы заглушки реализуют некоторые сетевые операции, требуемые для направления запроса прикладному объекту и принятия от него результата. Когда клиент вызывает прикладной метод заглушки, запрос передается через сеть посредством отправки каркасу имени вызванного метода и значений входных параметров. Когда каркас принимает входной поток, он анализирует его и определяет запрашиваемый метод, а затем вызывает соответствующий метод прикладного объекта. Все значения, возвращаемые вызванным методом прикладного объекта, передаются обратно заглушке через каркас. Заглушка затем возвращает эти значения клиентскому приложению, как будто они были обработаны локальной прикладной логикой.

Обкатка своего собственного объекта

Наилучший способ проиллюстрировать работу объекта состоит в том, чтобы показать, как вы можете сами реализовать распределенный объект с помощью собственного протокола распределенных объектов. Это дает вам некоторое представление о том, как работает настоящий протокол распределенных объектов, например CORBA. Однако существующие системы распределенных объектов, такие как DCOM, CORBA и Java RMI, гораздо сложнее и мощнее, чем этот простой пример, который мы сейчас разработаем. Система распределенных объектов, которую мы создадим в этой главе, является исключительно иллюстративной. Она не представляет собой ни настоящую технологию, ни часть Enterprise JavaBeans. Ее цель – дать читателю некоторое понимание механизма работы более изощренных систем распределенных объектов.

Здесь приведен очень простой прикладной объект с именем `PersonServer`, который реализует интерфейс `Person`. Интерфейс `Person` охватывает концепцию прикладного объекта, соответствующего определенному человеку. У него есть два метода: `getAge()` и `getName()`. В реальном приложении мы, возможно, определили бы более сложное поведение для объекта Person, но для этого примера вполне достаточно двух методов:

```
public interface Person {
    public int getAge() throws Throwable;
    public String getName() throws Throwable;
}
```

Реализация этого интерфейса `PersonServer` не содержит ничего неожиданного. Она определяет прикладную логику и состояние объекта Person:

```
public class PersonServer implements Person {
    int age;
    String name;
```

```
    public PersonServer(String name, int age){
        this.age = age;
        this.name = name;
    }
    public int getAge(){
        return age;
    }
    public String getName(){
        return name;
    }
}
```

Сейчас нам нужен некоторый способ сделать объект PersonServer доступным для удаленного клиента. Это является обязанностью объектов Person_Skeleton и Person_Stub. Интерфейс Person описывает независящее от реализации понятие человека. Оба объекта – и PersonServer, и Person_Stub – реализуют интерфейс Person, поскольку они оба предназначены для поддержки понятия «человек». Объект PersonServer реализует интерфейс для того, чтобы обеспечить существующую прикладную логику и состояние. Person_Stub реализует данный интерфейс для того, чтобы походить на прикладной объект Person на стороне клиента и передавать запросы обратно каркасу, который, в свою очередь, должен передать их самому объекту. Здесь показано, как выглядит эта заглушка.

```
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;

public class Person_Stub implements Person {
    Socket socket;

    public Person_Stub() throws Throwable {
        /* Устанавливаем сетевое соединение с каркасом.
            Используем имя "localhost" или IP-адрес каркаса,
            если он находится на другой машине */
        socket = new Socket("localhost",9000);
    }
    public int getAge() throws Throwable {
        // Во время вызова этого метода имя метода передается каркасу
        ObjectOutputStream outStream =
            new ObjectOutputStream(socket.getOutputStream());
        outStream.writeObject("age");
        outStream.flush();
        ObjectInputStream inStream =
            new ObjectInputStream(socket.getInputStream());
        return inStream.readInt();
    }
    public String getName() throws Throwable {
        // Во время вызова этого метода имя метода передается каркасу
```

```
            ObjectOutputStream outStream =
                new ObjectOutputStream(socket.getOutputStream());
            outStream.writeObject("name");
            outStream.flush();
            ObjectInputStream inStream =
                new ObjectInputStream(socket.getInputStream());
            return (String)inStream.readObject();
        }
    }
```

Когда вызывается метод объекта `Person_Stub`, создается и направляется каркасу маркер (token), представляющий собой объект `String`. В этой посылке указывается вызываемый метод заглушки. Каркас анализирует посылку, определяет метод, вызывает соответствующий метод прикладного объекта и отправляет результат обратно. Когда заглушка считывает ответ от каркаса, она анализирует его значение и возвращает клиенту. С точки зрения клиента, заглушка обрабатывает запрос локально. Посмотрим на каркас:

```
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;
import java.net.ServerSocket;

public class Person_Skeleton extends Thread {
    PersonServer myServer;

    public Person_Skeleton(PersonServer server){
        // Получаем ссылку на прикладной объект, представленный каркасом
        this.myServer = server;
    }
    public void run(){
        try {
            // Создаем серверный сокет на порту 9000.
            ServerSocket serverSocket = new ServerSocket(9000);
            // Ждем и получаем сетевое соединение с заглушкой
            Socket socket = serverSocket.accept();
            while (socket != null){
                // Создаем входной поток для приема запросов от заглушки
                ObjectInputStream inStream =
                    new ObjectInputStream(socket.getInputStream());
                // Читаем очередной запрос метода от заглушки
                // Блокируем поток на время приема
                String method = (String)inStream.readObject();
                // Определяем тип запрошенного метода
                if (method.equals("age")){
                    // Вызываем прикладной метод серверного объекта
                    int age = myServer.getAge();
                    // Создаем выходной поток для отправки результата
                    // обратно заглушке
                    ObjectOutputStream outStream =
```

```
                    new ObjectOutputStream(socket.getOutputStream());
                // Отправляем результат заглушке
                outStream.writeInt(age);
                outStream.flush();
            } else if(method.equals("name")){
                // Вызываем прикладной метод серверного объекта
                String name = myServer.getName();
                // Создаем выходной поток для отправки результата
                // обратно заглушке
                ObjectOutputStream outStream =
                    new ObjectOutputStream(socket.getOutputStream());
                // Отправляем результат заглушке
                outStream.writeObject(name);
                outStream.flush();
            }
        }
    } catch(Throwable t) {t.printStackTrace();System.exit(0); }
  }
  public static void main(String args [] ){
    // Получаем уникальный экземпляр Person
    PersonServer person = new PersonServer("Richard", 36);
    Person_Skeleton skel = new Person_Skeleton(person);
    skel.start();
  }
}
```

Объект `Person_Skeleton` перенаправляет запросы, получаемые от заглушки, прикладному объекту `PersonServer`. Естественно, что `Person_Skeleton` тратит все свое время на ожидание запроса от заглушки. Полученный запрос анализируется и направляется соответствующему методу объекта `PersonServer`. Значение, возвращаемое прикладным объектом, направляется затем обратно заглушке, которая возвращает его клиенту, как будто оно было получено локально.

Итак, создав весь механизм, давайте посмотрим на простую клиентскую программу, использующую объект `Person`:

```
public class PersonClient {
  public static void main(String [] args){
    try {
      Person person = new Person_Stub();
      int age = person.getAge();
      String name = person.getName();
      System.out.println(name+" is "+age+" years old");
    } catch(Throwable t) {t.printStackTrace();}
  }
}
```

Это клиентское приложение показывает, как заглушка используется клиентом. Если не считать создания экземпляра `Person_Stub` в самом

начале, клиент не знает, что прикладной объект Person в действительности – сетевой заместитель реального прикладного объекта, расположенного на промежуточном уровне. На рис. 1.3. диаграмма цикла RMI изменена, для того чтобы можно было видеть, как процесс RMI применяется к нашему коду.

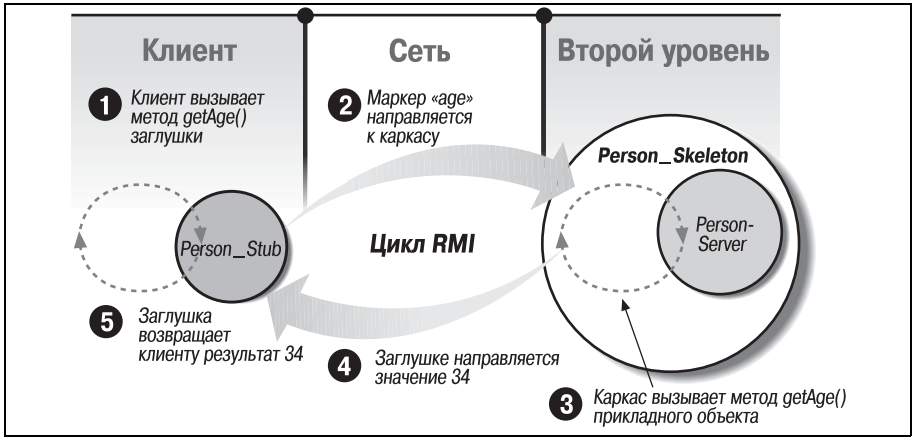

Рис. 1.3. Цикл RMI для прикладного объекта Person

Рис. 1.3. иллюстрирует реализацию цикла RMI при помощи распределенного объекта Person. RMI является основой систем распределенных объектов и отвечает за *прозрачность расположения* (*location transparency*) распределенных объектов. Прозрачность расположения означает, что действительное расположение серверного объекта – обычно в промежуточном слое – неизвестно использующему его клиенту и не имеет для него значения. В нашем примере клиент может находиться на той же машине либо на другой машине очень далеко отсюда, но взаимодействие клиента с прикладным объектом остается таким же. Прозрачность расположения является одним из самых больших преимуществ систем распределенных объектов. Хотя прозрачность и является преимуществом, во время проектирования вы не должны относиться к распределенным объектам как к локальным, из-за разницы в их производительности. В этой книге представлены удачные стратегии дизайна распределенных объектов, которые благодаря преимуществам прозрачности максимизируют производительность распределенной системы.

В данной книге заглушка на стороне клиента часто называется *удаленной ссылкой* (*remote reference*) на прикладной объект. Это позволяет нам прямо говорить о прикладном объекте и его представлении на стороне клиента.

Протоколы распределенных объектов, такие как CORBA, DCOM и Java RMI, обеспечивают гораздо более мощную инфраструктуру для распределенных объектов, чем в примере с Person. Большинство реализа-

ций протоколов распределенных объектов предоставляют средства автоматического создания соответствующих заглушек и каркасов для прикладных объектов. Это устраняет необходимость ручного создания этих конструкций и позволяет включать в заглушки и каркасы гораздо больше функциональных возможностей.

Даже при автоматической генерации заглушек и каркасов пример с `Person` лишь касается поверхности изощренного протокола распределенных объектов. Реальные протоколы, такие как Java RMI и CORBA IIOP, предоставляют обработку ошибок и исключений, передачу параметров и другие службы, например передачу контекстов транзакций и безопасности. Кроме этого, протоколы распределенных объектов поддерживают гораздо более изощренные механизмы связи между заглушками и каркасами. Прямое соединение заглушки с каркасом, используемое в примере с `Person`, – достаточно примитивно.

Реальные протоколы распределенных объектов, такие как CORBA, предоставляют также посредников объектных запросов (Object Request Broker, ORB), позволяющих клиентам находить и соединяться с распределенными объектами через сеть. ORB – это каркас (или коммутатор) для распределенных объектов. Кроме управления соединениями, ORB обычно используют систему имен для поиска объектов и массу других возможностей, например передачу ссылок, распределенный сбор мусора и управление ресурсами. Однако ORB ограничиваются соединениями между клиентами и распределенными прикладными объектами. Хотя они и могут поддерживать службы, такие как управление транзакциями и безопасность, службы эти не используются автоматически. При работе с ORB большая часть обязанностей по обеспечению функциональных возможностей системного уровня и взаимодействию различных служб ложится на плечи разработчика приложения.

Модели компонентов

Термин «модели компонентов» имеет много различных толкований. Enterprise JavaBeans определяет модель *серверных* (*server-side*) *компонентов*. Посредством набора классов и интерфейсов из пакета `java.ejb` разработчики могут создавать, собирать и развертывать компоненты, соответствующие спецификации EJB.

Исходная (первоначальная) технология JavaBeans также является компонентной моделью, но не моделью серверных компонентов, как EJB. Фактически, кроме общего имени «JavaBeans», эти две компонентные модели абсолютно ничем не связаны. В прошлом во многих публикациях EJB называли расширением исходной JavaBeans, но это неверное представление. Эти два программных интерфейса служат совершенно разным целям, и EJB не расширяет и не использует исходную компонентную модель JavaBeans.

Технология JavaBeans предназначена для *внутри*процессного (*intra*process) использования, тогда как EJB разрабатывалась для *меж*процессных (*inter*process) компонентов.

JavaBeans может быть использована для решения различных проблем, но в основном она применяется для построения клиентских приложений путем соединения визуальных (GUI) и невизуальных компонентов. Это превосходная компонентная модель, возможно, лучшая из всех когда-либо созданных для внутрипроцессной разработки, но она не является моделью серверных компонентов. EJB, с другой стороны, специально разрабатывалась для решения проблем, связанных с управлением распределенными прикладными объектами в трехуровневой архитектуре.

Но если технологии JavaBeans и Enterprise JavaBeans – совершенно разные, почему тогда они обе были названы компонентными моделями? В этом контексте компонентная модель определяет набор правил, определяющих взаимодействие между разработчиком компонентов и системой, содержащей эти компоненты. Эти правила определяют, как следует разрабатывать и упаковывать компоненты. Когда компонент определен, он становится независимым программным модулем, который может распространяться отдельно и использоваться в других приложениях. Компоненты разрабатываются для определенных целей, а не для определенного приложения. В исходной технологии JavaBeans компоненты могли быть кнопками или электронными таблицами (spreadsheet) и использоваться в любом графическом приложении, в соответствии с правилами, заданными компонентной моделью JavaBeans. В модели EJB компонент может быть прикладным объектом, представляющим клиент, развертываемым на любом сервере EJB и используемым для создания любого приложения, нуждающегося в таком объекте. К другим типам компонентных моделей Java относятся сервлеты, страницы JSP и апплеты.

Мониторы компонентных транзакций (CTM)

Технология CTM выросла из технологий ORB и TP-мониторов. CTM – это гибрид двух технологий, представляющий собой мощную платформу для распределенных объектов. Для того чтобы лучше понять, что такое CTM, рассмотрим сильные и слабые стороны TP-мониторов и ORB.

Мониторы обработки транзакций

Мониторы обработки транзакций (Transaction Processing, TP) появились около 30 лет назад (Customer Information Control System, CICS, была представлена в 1968 году) и сделались мощными, быстрыми серверными платформами для надежных приложений. Некоторые TP-продукты, такие как CICS и TUXEDO, должны быть вам знакомы.

Мониторы TP являются операционными системами для приложений, написанных на языках типа COBOL. «Операционная система» может показаться странным названием для монитора TP, но оно вполне оправданно, поскольку монитор управляет всей средой приложения. Системы с мониторами TP автоматически управляют всем окружением, в котором выполняется приложение, включая транзакции, управление ресурсами и восстановление после сбоев. Приложения, выполняющиеся внутри мониторов TP, пишутся на процедурных языках (т. е. COBOL и C) и обычно доступны через *систему сетевых сообщений* (*network messaging*) и *удаленные вызовы процедур* (*remote procedure calls, RPC*). Система сообщений позволяет клиенту посылать сообщения монитору TP, требуя запустить некоторое приложение с заданными параметрами. Это похоже на принцип работы модели событий Java. Сообщения могут быть синхронными или асинхронными. Это означает, что источник сообщений не должен ждать ответа. RPC, которая является предком RMI, – это механизм распределения, позволяющий клиентам вызывать процедуры приложения, работающего внутри монитора TP, как если бы процедура выполнялась локально. Основное различие между RPC и RMI состоит в том, что RPC используется в *процедурных* приложениях, а RMI – в системах распределенных *объектов*. С помощью RMI могут вызываться методы заданных экземпляров объектов, т. е. заданных прикладных сущностей. С помощью RPC клиент может вызвать процедуру определенного типа приложения, где нет понятия экземпляра объекта. Таким образом, RMI представляет собой объектно-ориентированный механизм, а RPC – процедурный.

Мониторы TP существуют уже достаточно долгое время, поэтому технология, лежащая в их основе, стала прочной, как камень. Вот почему они сегодня применяются во многих прикладных системах. Но мониторы TP не являются объектно-ориентированными. Они работают с процедурным кодом, способным выполнять сложные задачи, но не имеющим представления об объектах. Доступ к мониторам TP через RPC похож на вызов статических методов, здесь не существует понятия уникального объекта. Кроме этого, по той причине, что мониторы TP основаны на процедурных приложениях, а не на объектах, прикладная логика в мониторах TP не является гибкой, расширяемой или многократно используемой, как у прикладных объектов в системах с распределенными объектами.

Посредники объектных запросов

Благодаря системам распределенных объектов доступ к уникальным объектам, имеющим состояние и идентификатор, возможен из всех точек сети. Технологии распределенных объектов, такие как CORBA и Java RMI, отличаются от RPC одной существенной чертой: мы вызываем метод, принадлежащий экземпляру компонента, а не прикладной

ную процедуру. Распределенные объекты разворачиваются обычно на каком-либо ORB, отвечающем за предоставление клиентским приложениям возможности легкого поиска этих объектов.

Однако ORB не являются «операционными системами» для распределенных объектов. Они представляют собой просто коммуникационный каркас, используемый для доступа и взаимодействия с уникальными удаленными объектами. При разработке приложения на основе распределенных объектов, использующего ORB, вся ответственность за реализацию параллелизма, транзакций, механизмов управления ресурсами и защиты от сбоев ложится на плечи программиста. Эти службы могут поддерживаться ORB, но разработчик приложения отвечает за объединение их с прикладными объектами. В ORB отсутствует понятие операционной системы, в которой задачи системного уровня обрабатываются автоматически. Недостаток неявной инфраструктуры системного уровня создает многочисленные проблемы для разработчика приложения. Создание такой инфраструктуры, требующей обработки параллелизма, транзакций, безопасности, постоянства и кроме этого нуждающейся в поддержке большого количества пользователей, является задачей, достойной Геркулеса, которую способны решить очень немногие корпоративные коллективы разработчиков.

CTM: гибрид ORB и TP-мониторов

Когда преимущества распределенных объектов стали очевидными, количество внедряемых систем, применяющих ORB, стало быстро увеличиваться. ORB поддерживают распределенные объекты, используя некую «сырую» модель серверных компонентов, позволяющую распределенным объектам подключаться к коммуникационному каркасу, но в то же время они не обеспечивают полную поддержку транзакций, безопасности, постоянства и управления ресурсами. Эти службы должны быть явно доступны распределенным объектам через программные интерфейсы, что приводит к большей сложности и зачастую к более серьезным проблемам при развертывании. Кроме этого, стратегии управления ресурсами, такие как подкачка экземпляров, пул ресурсов и активация не поддерживаются вовсе. Такие виды стратегии делают возможным масштабирование систем распределенных объектов, улучшают производительность и пропускную способность и являются более понятными. Без автоматической поддержки управления ресурсами разработчик приложения вынужден применять «доморощенные» решения по управлению ресурсами, требующие отличного знания систем распределенных объектов. ORB проигрывают в отношении сложности управления компонентами в крупномасштабных прикладных средах, в тех областях, где мониторы TP всегда считались непревзойденными.

После тридцатилетнего опыта работы с мониторами TP компании, такие как IBM и BEA, начали разработку гибрида систем ORB и TP-мо-

ниторов, названного мониторами компонентных транзакций (CTM). Этот тип серверов приложений соединяет в себе гибкость и доступность систем распределенных объектов, основанных на ORB с мощностью «операционных систем» мониторов TP. CTM предоставляют комплексную среду для серверных компонентов, обеспечивая автоматическое управление параллелизмом, транзакциями, распределением объектов, выравниванием загрузки, безопасностью и ресурсами. Хотя разработчики приложений по-прежнему должны быть знакомы с этими возможностями, в случае применения CTM им нет необходимости явно реализовывать их.

Основными особенностями CTM являются распределенные объекты, инфраструктура, включающая управление транзакциями и другие службы, и модель серверных компонентов. CTM поддерживают эти особенности в разной степени. При выборе CTM не так важно найти самый мощный и многофункциональный, как тот, который лучшим образом удовлетворяет вашим требованиям. Очень большие и мощные CTM могут стоить чрезвычайно дорого, а их применение для небольших проектов избыточно. CTM используется в нескольких областях, включая реляционные базы данных, серверы приложений, веб-серверы, ORB для CORBA и мониторы TP. Каждый производитель предлагает продукты, отражающие его область компетенции. Однако для начинающего выбор CTM с поддержкой компонентной модели Enterprise JavaBeans может быть важнее любого другого набора функциональных возможностей. По той причине, что Enterprise JavaBeans является независимой от реализации, выбор EJB CTM даст прикладной системе в случае необходимости возможность масштабирования на более крупный CTM. Мы обсудим важность EJB как стандартной компонентной модели для CTM далее в этой главе.

Сравнение с реляционными базами данных

В этой главе обсуждению CTM уделяется так много времени потому, что они важны для определения EJB. Обсуждение CTM еще не закончено, но для того чтобы дальнейшее изложение стало понятным, мы сравним CTM с реляционной базой данных.

Реляционные базы данных предоставляют разработчикам приложений простую среду разработки в сочетании с мощной инфраструктурой данных. Разработчик приложения, использующего базу данных, может задавать расположение таблиц, решать, какие столбцы следует сделать первичными ключами, определить индексы и хранимые процедуры, но он не должен разрабатывать алгоритмы индексации, анализатор SQL и системы управления курсорами. Реализация этих типов функциональности системного уровня возложена на производителей баз данных, а разработчик просто выбирает тот продукт, который наилучшим образом удовлетворяет его требованиям. Разработчиков приложений интересует только организация их прикладных данных,

а не то, как работает процессор базы данных. Для разработчика приложений написание реляционной базы данных с нуля стало бы напрасной тратой времени, ведь Microsoft и Oracle уже предоставляют их.

Для эффективной работы распределенные объекты требуют такого же управления системным уровнем со стороны СТМ, как и прикладные данные от реляционных баз данных. Функциональные возможности системного уровня, а именно параллелизм, управление транзакциями и управление ресурсами, необходимы тогда, когда прикладную систему планируется применять для работы с большим количеством пользователей. Нереально и напрасно ждать от разработчика приложения, что он еще раз изобретет всю эту функциональность системного уровня, тогда как уже существуют готовые коммерческие решения.

СТМ является для прикладных объектов тем, чем реляционные базы данных являются для данных. СТМ отвечает за всю работу системного уровня, позволяя разработчикам приложений сосредоточиться на стоящих перед ними задачах. Применяя СТМ, разработчик приложения может сфокусироваться на разработке и реализации прикладных объектов без необходимости тратить тысячи часов на разработку инфраструктуры, в которой эти объекты будут выполняться.

СТМ и модели серверных компонентов

СТМ требуют, чтобы прикладные объекты придерживались модели серверных компонентов, реализованной производителем. Удачная модель компонентов представляет собой решающий фактор успеха разрабатываемого проекта, т. к. она определяет, насколько легко разработчик приложения может написать прикладной объект под данный СТМ. Модель компонентов является соглашением, определяющим обязанности СТМ и прикладных объектов. Имея дело с удачной компонентной моделью, разработчик знает, что можно ожидать от СТМ, а СТМ «понимает», как следует управлять прикладными объектами. Модели серверных компонентов хорошо подходят для описания обязанностей разработчика приложения и производителя СТМ.

Модели серверных компонентов опираются на определенную спецификацию. Если компонент придерживается этой спецификации, он может быть использован СТМ. Отношения между серверным компонентом и СТМ похожи на отношения между CD-диском и CD-проигрывателем. Если компонент (CD-диск) придерживается спецификации проигрывателя, он может быть воспроизведен на нем.

Отношения СТМ с его компонентной моделью похожи также на отношения между железнодорожной системой и поездом. Железнодорожная система управляет окружением поезда, предоставляя дополнительные пути с целью равномерного распределения нагрузки, множественные пути для обеспечения параллельной работы и систему управления трафиком для управления ресурсами. Железная дорога

предоставляет инфраструктуру, на которой работает поезд. Точно так же CTM предоставляет серверным компонентам всю инфраструктуру, необходимую для поддержки параллелизма, транзакций, распределения загрузки и т. д.

Поезда на железной дороге похожи на серверные компоненты: все они выполняют различные задачи, но выполняют их, используя один базовый проект. Поезда ориентированы на выполнение задач, таких как перевозка автомобилей, а не на управление их окружением. Для машиниста – лица, управляющего поездом, интерфейс управления достаточно прост: тормоза и дроссель. Для разработчика приложения интерфейсы серверного компонента ограничены похожим образом.

Различные CTM могут реализовывать различные компонентные модели точно так же, как различные железные дороги могут иметь различные типы поездов. А различия между моделями компонентов варьируются так же, как железнодорожные системы имеют пути разной ширины и различные системы управления, но базовые операции CTM остаются одними и теми же. Все они гарантируют, что управление прикладными объектами обеспечит поддержку большого числа пользователей в критичных ситуациях. Это означает, что ресурсы, параллельность, транзакции, безопасность, постоянство, баланс загрузки и распределение объектов могут быть обработаны автоматически, ограничив разработчика приложения простым интерфейсом. Это позволяет разработчику приложения концентрироваться на прикладной логике, а не на корпоративной инфраструктуре.

Среда .NET от Microsoft

Корпорация Microsoft была первым производителем, начавшим распространять CTM. Первоначально названный сервером транзакций (Microsoft Transaction Server, MTS), он позднее был переименован в COM+. Microsoft COM+ основан на модели компонентных объектов (Component Object Model, COM), первоначально разработанной для использования в настольных компьютерах, но со временем переродившейся в сервис в виде модели серверных компонентов. Для распределенного доступа клиента COM+ используют распределенную модель компонентных объектов (Distributed Component Object Model, DCOM).

В 1996 году, когда MTS был представлен впервые, он казался весьма впечатляющим, поскольку предоставлял универсальное окружение для прикладных объектов. При помощи MTS разработчики приложений могли писать компоненты COM, не заботясь о вопросах системного уровня. Если прикладной объект был разработан в соответствии с моделью COM, то MTS (а сейчас COM+) заботился обо всем остальном, включая управление транзакциями, параллелизм и управление ресурсами.

Недавно COM+ стал частью новой среды – .NET (.NET Framework) от Microsoft. Базовая функциональность, предоставляемая сервисами COM+, фактически сохранилась в .NET, но перед разработчиком она теперь предстает в существенно изменившейся «ипостаси». Разработчик вместо того, чтобы писать компоненты в виде объектов COM, в среде .NET строит приложения в виде *управляемых объектов* (*managed object*). Все управляемые объекты и фактически весь код, написанный под .NET, зависит от общеязыковой среды времени выполнения (Common Language Runtime, CLR). Для разработчиков на Java среда CLR во многом напоминает виртуальную машину (VM) Java, а управляемые объекты представляют собой аналоги экземпляров классов Java, т. е. объектов Java.

Хотя среда .NET и предоставляет много интересных особенностей, в качестве открытого стандарта они использоваться не могут. Службы COM+ в среде .NET являются собственным СТМ фирмы Microsoft. Это может быть не так уж плохо, поскольку .NET обещает работать хорошо, а платформа Microsoft широко распространена. Кроме того, поддержка средой .NET простого протокола доступа к объектам (Simple Object Access Protocol, SOAP) позволяет прикладным объектам из среды .NET связываться с объектами любой другой платформы, написанными на любом языке. Это потенциально может сделать прикладные объекты из среды .NET универсально доступными – возможность, от которой не так легко отказаться.

Однако если какая-то компания намеревается развертывать серверные компоненты не на платформе Microsoft, .NET не будет жизнеспособным решением. Кроме того, службы COM+ в среде .NET ориентированы на компоненты без состояния, в них нет встроенной поддержки для постоянных транзакционных объектов (persistent transaction objects). Хотя объекты без состояния могут предложить более высокую производительность, прикладным системам требуется такой тип гибкости, предлагаемый СТМ, который включает компоненты с состоянием и постоянные компоненты.

EJB и CORBA СТМ

До осени 1997 года мониторы СТМ, можно сказать никто, кроме Microsoft, не выпускал. Многообещающие продукты от IBM, BEA и Hitachi были еще «на чертежной доске», а MTS уже присутствовал на рынке. Все остальные проекты, оставаясь лишь проектами, имели одну общую черту: в качестве службы распределенных объектов они использовали технологию CORBA.

Большинство СТМ «не от Microsoft» были направлены на то, чтобы стать своевременными и более открытыми стандартами CORBA, благодаря этому они должны были работать на несовместимых с Microsoft платформах и поддерживать несовместимые с Microsoft клиенты. Благодаря тому что CORBA не зависит как от языка, так и от платформы,

производители СТМ могли дать своим клиентам больше альтернативных вариантов при реализации.[1] Проблемой разработок CORBA СТМ было то, что они поддерживали различные модели серверных компонентов. Другими словами, если компонент разрабатывался под СТМ одного производителя, то уже нельзя было повернуть назад и использовать этот же самый компонент на СТМ от другого производителя. Компонентные модели были слишком разные.

Для конкуренции с MTS от Microsoft, удерживавшим лидерство вплоть до 1997 года (к тому времени он уже существовал около года), СТМ, ориентированные на CORBA, нуждались в конкурентоспособной идее. Одной из проблем, с которой столкнулись СТМ, была раздробленность рынка CORBA, в котором продукт каждого производителя отличался от своего соседа. Раздробленный рынок не был нужен никому, поэтому производителям CORBA СТМ был необходим стандарт, вокруг которого можно было бы сплотиться. Кроме протокола CORBA, наиболее очевидным требуемым стандартом была компонентная модель, которая позволила бы клиентам и третьим производителям разрабатывать собственные прикладные объекты для одной спецификации, способные работать на любой CORBA СТМ. Microsoft, разумеется, проталкивала свою компонентную модель в качестве стандарта, который был привлекателен, поскольку MTS представлял собой реально работающий продукт, но он не поддерживал CORBA. Object Management Group (OMG) – те же самые люди, которые создали стандарт CORBA, определили альтернативную модель серверных компонентов. Модель была многообещающей, поскольку наверняка прекрасно ужилась бы с CORBA, но OMG разрабатывала стандарт медленно, слишком медленно для развивающегося рынка СТМ.[2]

В 1997 году компания Sun Microsystems разработала самый многообещающий стандарт для серверных компонентов: Enterprise JavaBeans. Sun предложила несколько важных преимуществ. Во-первых, Sun была уважаемой компанией, получившей известность по работе с производителями с целью определения ориентированных на Java программных интерфейсов для общераспространенных сервисов. Во-вторых,

---

[1] Недавнее появление SOAP поставило под вопрос будущее протокола CORBA IIOP (Internet Inter-Operability Protocol). Очевидно, что эти два протокола борются между собой за право стать стандартным, независимым от языка протоколом для распределенных вычислений. IIOP существует уже около семи лет и, следовательно, более проработан, но SOAP может быстро его нагнать, используя опыт, накопленный при его создании.

[2] Компонентная модель CORBA СТМ, CORBA Component Model (CCM), в конце концов была завершена. В целом она получила довольно слабое признание и была вынуждена включить Enterprise JavaBeans как часть своей компонентной модели просто для того, чтобы быть нужной и привлекательной.

Sun имела привычку перенимать лучшие идеи и затем выпускать реализацию Java открытым стандартом – и, как правило, удачно. Программный интерфейс взаимодействия с базами данных, названный JDBC, тому прекрасный пример. В-третьих, опираясь в основном на ODBC от Microsoft, JDBC предложил производителям более гибкую модель для включения в нее их собственных драйверов доступа к базам данных. Ну и, кроме того, разработчики нашли интерфейс JDBC более легким в работе, чем ODBC. Sun проделала эти же вещи и со своими более новыми технологиями, такими как программный интерфейс JavaMail и Java Naming and Directory Interface (JNDI). Эти технологии все еще находились в разработке, но сотрудничество между производителями обнадеживало, а открытость программных интерфейсов была очень привлекательна.

Хотя CORBA предложила открытый стандарт, она пыталась стандартизировать низкоуровневые механизмы, а именно безопасность и транзакции. Производители не могли позволить себе переписать заново существующие продукты вроде TUXEDO или CICS в соответствии со стандартами CORBA. EJB обошла эту проблему, объявив, что способ реализации низкоуровневых служб не имеет значения. Имеет значение только то, что все эти механизмы, применяемые к компонентам, соответствуют спецификации – гораздо более удобное решение для существующих и будущих производителей СТМ. Кроме того, язык Java предлагает несколько довольно соблазнительных преимуществ, не все из которых чисто технические. Во-первых, Java была очень популярной технологией, и, просто сделав свой продукт совместимым с Java, можно было обеспечить ему прорыв на рынок. Java является более или менее независимой от платформы, а компонентные модели, определенные в языке Java, обладают четко выраженными коммерческими и техническими преимуществами.

Анонсировав Enterprise JavaBeans, корпорация Sun не стала почивать на лаврах. Ее инженеры продолжают работать с несколькими ведущими производителями над созданием гибкого и открытого стандарта, под который производители могли бы легко адаптировать свои существующие продукты. Это было трудной задачей из-за того, что у производителей имелись различные виды серверов, включающих веб-серверы, серверы реляционных баз данных, серверы приложений и ранние версии СТМ. Похоже на то, что никто не хотел жертвовать своей архитектурой ради общего блага, но постепенно производители согласились с моделью, которая была достаточно гибкой для согласования различных реализаций и в то же время достаточно надежной для поддержки реальных прикладных разработок. В декабре 1997 году Sun Microsystems выпустила первую рабочую спецификацию Enterprise JavaBeans – EJB 1.0, и производители сплотились вокруг этой модели серверных компонентов как никогда раньше.