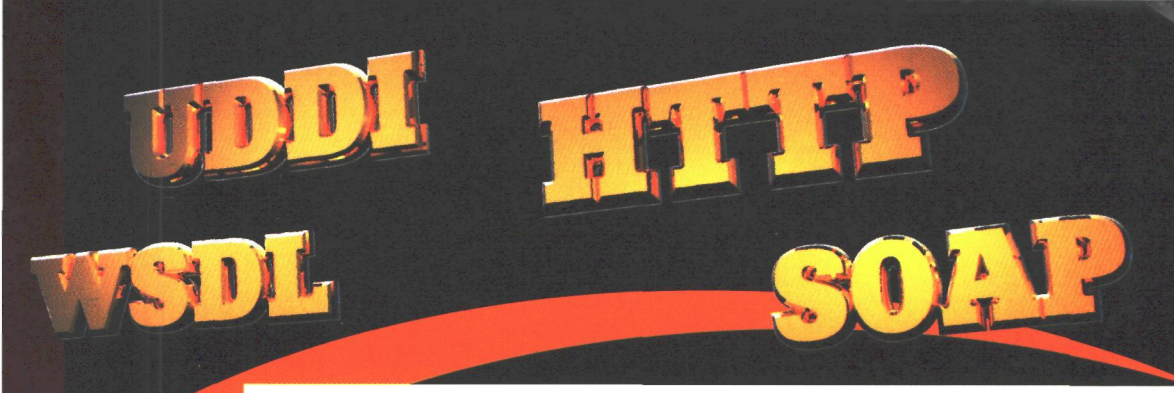


Ильдар Хабибуллин

Разработка Web-служб средствами Java

- Архитектура Web-служб
- Работа Web-служб в J2EE
- Безопасность предоставления услуг

МАСТЕР_ПРОГРАММ_

Ильдар Хабибуллин

Разработка WEB-СЛУЖБ средствами Java

Санкт-Петербург
«БХВ-Петербург»
2003

УДК 681.3.068
ББК 32.973.202
Х12

Хабибуллин И. Ш.

Х12 Разработка Web-служб средствами Java. — СПб.: БХВ-Петербург, 2003. — 400 с: ил.

ISBN 5-94157-227-1

Книга посвящена описанию методов разработки Web-служб — приложений, доступных в Internet с помощью языка XML и обычных средств WWW. В книге последовательно излагаются все сведения, необходимые для создания Web-служб, рассматриваются средства разработки на языке Java, разъясняются приемы их использования. Приводятся необходимые сведения о языке XML, протоколы SOAP, WSDL, UDDI и другие протоколы и спецификации, применяемые при создании Web-служб. Подробно изучаются пакеты классов Java и прочие инструментальные средства, облегчающие создание приложений. Особое внимание уделено русификации Web-служб.

Пользуясь материалом книги, читатель сможет создавать самые разнообразные Web-службы любой сложности. Книга адресована студентам старших курсов, ИТ-специалистам и профессиональным разработчикам Web-приложений.

Для программистов

УДК 681.3.068
ББК 32.973.202

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анатолий Хрипов*
Компьютерная верстка	*Татьяны Валерьяновой*
Корректор	*Евгений Камский*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 21.02.03.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 32,25.
Тираж 3000 экз. Заказ № 748
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар No 77.99.02.953.Д.001537.03.02 от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-227-1

Содержание

Введение

Уже несколько лет в Интернете и на страницах компьютерных журналов мелькает словосочетание "Web Services" ("Web-службы"). Это понятие бурно обсуждается специалистами, создаются рабочие группы и целые фирмы, занимающиеся Web Services, выходят программные продукты, публикуются статьи и книги. Буквальный перевод этих слов ничего не проясняет — вся "Всемирная паутина" только тем и занимается, что предоставляет сервисное обслуживание своим клиентам. Что же именно включают в себя Web Services?

Объяснение придется начать издалека.

Передача информации по компьютерным сетям должна удовлетворять двум противоречивым требованиям. Это надежность и скорость. Особенно важно обеспечить надежность связи и скорость передачи информации для компонентов распределенного приложения [10], работающих на разных машинах и часто на значительном удалении друг от друга. Распределенное приложение должно выполняться как единое целое, без сбоев и задержек, как будто все оно целиком работает на одном компьютере.

Надежность обеспечивается избыточностью информации: проверками поступающей информации, подтверждением ее получения, повторной передачей. Все эти приемы сильно загружают сеть и замедляют скорость пересылки информации. Для увеличения скорости передачи информация сжимается и уплотняется. Придуманы изощренные способы кодирования, экономно использующие каждый бит. Сетевые протоколы задают правила формирования сетевых пакетов, похожие на инструкции шифровальным отделам секретных служб. Сетевые приложения жестко привязываются к определенному протоколу, теряют переносимость, сфера их применения резко сужается. Создаются различные технологии сетевого взаимодействия распределенных компонентов: RPC (Remote Procedure Call), DCOM, RMI (Remote Method Invocation), CORBA.

Компьютерное сообщество разделилось, как человечество после Вавилонского столпотворения. Технологии RPC, DCOM, RMI плохо стыкуются друг с другом. Распределенное приложение приходится строить на основе только какой-то одной технологии. Это сильно ограничивает область его применения и затрудняет включение в приложение новых компонентов.

Технология Web Services создана фирмами IBM, Microsoft и UserLand для того, чтобы объединить компьютерный мир. Эта технология отказалась от упаковки и сжатия информации и предлагает прямо противоположный подход — по сети посылается простой "плоский" текст, записанный в байтовой кодировке ASCII, Latin1, CP866, CP1251, KOI-8 или в кодировках Unicode: UTF-8, UTF-16. Этим сразу решается проблема переносимости сетевой информации — любой сервер легко прочитает обычный текст. Для выявления структуры пересылаемых документов их записывают на языке XML (extensible Markup Language, расширяемый язык разметки). Это не сужает область распространения документов, поскольку любой сервер в состоянии понять язык XML. Надо сделать только одно — стандартизировать структуру документа XML. Для этого в технологии Web Services разработан специальный протокол SOAP (Simple Object Access Protocol). Сообщение, записанное на языке XML по правилам протокола SOAP, может принять и обработать любая Web-служба, на каком бы языке программирования она ни была написана, и в какой бы операционной системе она ни работала.

Второе предложение технологии Web Services — использовать для передачи сообщений, написанных по правилам протокола SOAP, только самый распространенный в Интернете, общедоступный и простой протокол HTTP (HyperText Transfer Protocol). Сервер, работающий по протоколу HTTP, можно легко развернуть на любой машине. Информация, предназначенная для HTTP-сервера, пропускается практически всеми брандмауэрами, что тоже удобно для распределенных приложений.

Это предложение выполняется не столь строго. Web-службы могут взаимодействовать и по почтовому протоколу SMTP (Simple Mail Transfer Protocol), и непосредственно по транспортному протоколу TCP (Transmission Control Protocol). Не запрещены и другие протоколы.

Технология Web Services не остановилась на создании единого протокола пересылки сообщений. Она создала и средства описания Web-службы. Сервер, на котором установлена Web-служба, предоставляет всем желающим ее описание. Описание выполняется на специально разработанном языке WSDL (Web Services Description Language, язык описания Web-служб). Это еще одна реализация языка XML. Клиент, обращающийся к Web-службе, читает ее описание и формирует свой запрос. Интересно, что разработаны программы, которые по готовой Web-службе создают ее описание на WSDL. Еще интереснее то, что есть программы, создающие, наоборот, Web-службу по ее описанию, сделанному на языке WSDL.

Для того чтобы клиент не обшаривал весь Интернет в поисках нужной Web-службы, создаются реестры, хранящие информацию о местоположении Web-служб вместе с описанием предоставляемых ими услуг. Правила регистрации Web-службы в реестре, а также правила хранения информации в реестре определяются еще одной реализацией XML — спецификацией

UDDI (Universal Discovery, Description and Integration). Многие известные фирмы, в числе которых IBM, Microsoft, Hewlett Packard, SAP AG, содержат свои реестры UDDI. Клиенту достаточно обратиться к реестру, найти в нем подходящую Web-службу, выбрать из реестра контактную информацию Web-службы и запросить у нее нужные услуги.

Итак, технология Web Services стоит на "трех китах" — протоколах и спецификациях SOAP, WSDL и UDDI. "Черепаха", на которой они стоят — это протокол HTTP. Имея такое надежное основание, Web Services уверенно плывут по безбрежному морю Интернета. Кроме этих основополагающих протоколов, есть еще несколько спецификаций, решающих вопросы, возникающие при практическом построении Web-служб, например, вопросы безопасности информации. Они вводятся в книге по мере возникновения решаемых ими вопросов.

Для быстрого развития Web Services нужны реализации спецификаций в виде библиотек и пакетов классов. Этим занимаются многие фирмы-производители программного обеспечения. Корпорация Microsoft поставляет программное обеспечение Web Services в рамках технологии Microsoft .NET. Описанию этой технологии посвящены многие публикации. На русском языке уже вышла очень живо и доступно написанная книга Игоря Шапошникова [12]. Фирмы IBM, Sun Microsystems и многие другие компании развивают Web Services как часть технологии Java. Поговорим об этом подробнее.

Web Services и Java

Создавая технологию Java, фирма Sun Microsystems, девиз которой: "Сеть — это компьютер", стремилась обеспечить создание межплатформенных приложений. Такое приложение пишется на любом языке, для которого есть компилятор в байт-коды Java. Приложение, откомпилированное в байт-коды, распространяется по Интернету и может быть выполнено на любом компьютере, имеющем JVM (Java Virtual Machine, виртуальная машина Java), без всяких дополнительных компиляций и инсталляций. При этом не надо компоновать приложение с библиотеками классов — библиотеки стандартизированы, они прилагаются к JVM и динамически компонуются во время выполнения приложения.

Прекрасно! Не надо компилировать приложение под разные платформы, мучительно добиваясь совместимости, не надо подбирать под каждую платформу подходящие библиотеки классов, не надо публиковать в Интернете список версий приложения, разработанных под разные платформы, вместе со списком версий подобранных библиотек. Пользователю не надо выкачивать из Интернета "толстый" дистрибутив и необходимые версии библиотек и инсталлировать их на своем компьютере.

Почему же технология Java не вытеснила все другие технологии Интернета? Почему мы до сих пор используем CGI, ASP, PHP, JavaScript, Perl, Python? Все дело в последнем условии — на каждом компьютере должна работать виртуальная машина Java. Не говоря уже о том, что она есть не на всяком компьютере, у JVM, как у всякого приложения, есть разные производители и разные версии, не полностью совместимые между собой. Кроме того, JVM должна быть полностью совместима с операционной системой и оборудованием того компьютера, на котором она работает.

Выясняется, что технология Java только частично решает проблему совместимости. Она переносит эту проблему с самих приложений на виртуальную машину Java. Конечно, решить проблему совместимости различных моделей JVM гораздо проще, но до сих пор это не сделано полностью.

С появлением Web Services эта проблема отпадает сама собой. Web-служба, построенная по технологии Java, может работать под управлением любой JVM. Ее легко встроить в любой J2EE-сервер (Java 2 Enterprise Edition) или Web-контейнер.

- Более того, Java облегчает создание и работу Web Services. В технологии Java уже есть все средства для их быстрой разработки. В стандартную поставку JDK (Java Developer Kit, пакет разработчика на Java) входят средства генерации документов XML вместе со средствами разбора готовых документов XML. Их удобно применять для создания и чтения сообщений Web-службы.

- На языке Java написаны самые популярные анализаторы XML-документов, программы для их проверки и преобразования и другие готовые средства работы с XML-документами .

- Сервлеты и страницы JSP (Java Server Page) позволяют организовать Web Services с минимальными затратами усилий. Уже одних сервлетов вполне достаточно для создания полноценных Web-служб.

- Компоненты EJB (Enterprise JavaBeans) дают возможность создать повторно применяемые модули Web-служб, а также связать их с базами данных и другими источниками информации.

- Новые библиотеки классов, специально разработанные для создания Web Services, предоставляют готовые компоненты Web-служб. Очень часто можно просто собрать Web-службу из готовых компонентов.

- Во все средства быстрой разработки (RAD) и интегрированные среды разработки (IDE, Integrated Development Environment) на языке Java уже встроены "мастера", собирающие Web-службу несколькими щелчками кнопки мыши.

Таким образом, технология Java прекрасно сочетается с технологией Web Services. Поэтому можно прийти к выводу, что дальнейшее развитие Web

Services будет тесно связано с Java. Эта книга знакомит вас с Java Web Services — самой мощной ветвью технологии Web Services.

Структура книги

Основной язык, постоянно используемый при создании Web Services — это язык XML. Поэтому в *главе 1* подробно изложены все необходимые конструкции XML и его реализаций. Особенно подробно изложен язык XSD (XML Schema Definition Language, язык определения схем XML), описывающий элементы документа. Это первое описание языка XSD, сделанное на русском языке.

Кроме того, в *главе 1* подробно рассматриваются интерфейсы и классы Java, предназначенные для чтения, анализа и преобразования документов XML, а также готовые инструментальные средства, реализующие эти интерфейсы, и использующие эти классы.

Глава 2 посвящена описанию работы Web Services на конкретных примерах. Прочитав эту главу, вы получите полное представление о структуре Web Services, механизмах их работы, поймете, как создаются Web-службы и как клиенты используют их услуги. Здесь приведено строгое определение Web Services и его реализаций различными фирмами-производителями средств создания Web-служб. Дан обзор наиболее распространенных пакетов классов, на которых основаны Web Services, и разобраны наиболее популярные инструментальные средства. Здесь же описывается средство описания Web-служб — язык WSDL — и средство регистрации Web-служб в Интернете — язык UDDI.

Прочитав *главу 2,* вы получите полное представление о Web Services и сможете свободно ориентироваться в публикациях, посвященных Web-службам.

Для создания Web-служб нужны более глубокие сведения. В *главах 3, 4* и 5 подробно рассматриваются новейшие версии "трех китов", на которых стоят Web Services — протоколов SOAP, WSDL и UDDI. Здесь же приведены программные средства, реализующие эти протоколы, показаны приемы их применения, приведены многочисленные примеры.

К этим главам примыкает *глава 6,* в которой разобраны внутренние механизмы Java Web Services. Прочитав эту главу, вы поймете, как технология Java реализует протоколы и спецификации Web Services. Освоив эти механизмы, вы сможете изменять исходные тексты используемых вами программных продуктов, добиваясь полного решения ваших задач, или самостоятельно создавать реализации Java Web Services.

Большинство Web-служб, созданных с применением Java, работает в составе серверов приложений. Серверы приложений, как правило, реализуют ин-

терфейсы, входящие в состав J2EE. Правила встраивания Web-служб в J2EE и особенности их работы в составе J2EE-сервера изложены в *главе* 7. Прочитав эту главу, вы поймете, какое место занимают Web-службы среди других компонентов J2EE-сервера приложений, и как Web-службы взаимодействуют с другими компонентами сервера. Кроме того, здесь рассказано, как установить готовую Web-службу в J2EE-сервер приложений.

В Web-технологиях важное место занимают вопросы безопасности. Поначалу Web Services не рассматривали их, предпочитая переложить эти вопросы на серверы, содержащие Web-службы, и на нижележащие протоколы. Очень скоро оказалось, что от проблемы безопасности не уйти, и в технологию Web Services были введены спецификации, обеспечивающие безопасность предоставления Web-услуг. Они обсуждаются в *главе 8.* На момент написания книги выходили только черновые версии этих спецификаций. Поэтому *глава 8* носит обзорный характер.

Наконец, *глава 9* содержит обзор новейших веяний в технологии Web Services и позволяет проследить возможные пути ее дальнейшего развития.

ГЛАВА 1

Обработка документов XML

В развитии Web-технологии огромную роль сыграл язык HTML (HyperText Markup Language, язык разметки гипертекста) . Любой человек, совсем не знакомый с программированием, мог бы за полчаса понять принцип разметки текста и за пару дней изучить теги HTML. Пользуясь простейшим текстовым редактором он мог бы написать свою страничку HTML, тут же посмотреть ее в своем браузере, испытать чувство глубокого удовлетворения, и гордо выставить в Интернете свой шедевр.

Замечательно! Не надо месяцами изучать запутанные языки программирования, явно предназначенные только для яйцеголовых "ботаников", осваивать сложные алгоритмы, возиться с компиляторами и отладчиками, размножать свое творение на дисках. Очень скоро появились текстовые редакторы, размечающие обычный "плоский" текст тегами HTML. Разработчику осталось только поправлять готовую страницу HTML, созданную таким редактором.

Простота языка HTML привела к взрывному росту числа сайтов, пользователей Интернета и авторов многочисленных Web-страничек. Обычные пользователи компьютеров ощутили себя творцами, получили возможность заявить о себе, высказать свои мысли и чувства, найти в Интернете своих единомышленников.

Ограниченные возможности языка HTML быстро перестали удовлетворять поднаторевших разработчиков, почувствовавших себя "профи". Набор тегов языка HTML строго определен и должен одинаково пониматься всеми браузерами. Нельзя ввести дополнительные теги или указать браузеру, как следует отобразить на экране содержимое того или иного тега. Введение таблиц стилей CSS (Cascading Style Sheet) и включений на стороне сервера SSI (Server Side Include) лишь ненадолго уменьшило недовольство разработчи-

ков. Профессионалу всегда не хватает средств разработки, он постоянно испытывает потребность добавить к ним какое-то свое средство, позволяющее воплотить все его фантазии.

Такая возможность есть. Еще в 1986 году стал стандартом язык создания языков разметки SGML (Standard Generalized Markup Language), с помощью которого и был создан язык HTML. Основная особенность языка SGML заключается в том, что он позволяет создать новый язык разметок, определив набор тегов создаваемого языка. Каждый конкретный набор тегов, созданный по правилам SGML, снабжается описанием DTD (Document Type Definition) — *определением типа документа,* разъясняющим связь тегов между собой и правила их применения. Специальная программа — драйвер принтера или SGML-браузер — руководствуется этим описанием для печати или отображения документа на экране дисплея.

В это же время выявилась еще одна, самая важная область применения языков разметки — поиск и выборка информации. В настоящее время подавляющее большинство информации хранится в реляционных базах данных. Они удобны для хранения и поиска информации, представимой в виде таблиц: анкет, ведомостей, списков и тому подобного, но неудобны для хранения различных документов, планов, отчетов, статей, книг, не представимых в виде таблицы. Тегами языка разметки можно задать структурную, а не визуальную разметку документа, разбить документ на главы, параграфы и абзацы или на какие-то другие элементы, выделить важные для поиска участки документа. Легко написать программу, анализирующую размеченный такими тегами документ и извлекающую из него нужную информацию.

Язык SGML оказался слишком сложным, требующим тщательного и объемистого описания элементов. Он применяется только в крупных проектах, например, для создания единой системы документооборота крупной фирмы. Скажем, man-страницы Solaris Operational Environment написаны на специально сделанной реализации языка SGML.

Золотой серединой между языками SGML и HTML стал язык разметок XML. Это подмножество языка SGML, избавленное от излишней сложности, но позволяющее разработчику Web-страниц создавать свои собственные теги. Язык XML достаточно широк, чтобы можно было создать все нужные теги, и достаточно прост, чтобы можно было быстро их описать.

Создавая описание документа на языке XML надо, прежде всего, продумать структуру документа. Приведем пример. Пусть мы решили, наконец, упорядочить свою записную книжку с адресами и телефонами. В ней записаны фамилии, имена и отчества родственников, сослуживцев и знакомых, дни их рождения, их адреса, состоящие из почтового индекса, города, улицы, дома и квартиры, и телефоны, если они есть: рабочие и домашние. Мы

придумываем теги для выделения каждого из этих элементов, продумываем вложенность тегов и получаем структуру, показанную в листинге 1.1.

Листинг 1.1. Пример XML-документа

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE notebook SYSTEM "ntb.dtd">

<notebook>

  <person>

    <name>
      <first-name>Иван</first-name>
      <second-name>Петрович</second-name>
      <surname>Сидоров</surname>
    </name>

    <birthday>25.03.1977</birthday>

    <address>
      <street>Садовая, 23-15</street>
      <city>Урюпинск</city>
      <zip>123456</zip>
    </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
    </phone-list>

  </person>

  <person>

    <name>
      <first-name>Мария</first-name>
      <second-name>Петровна</second-name>
```

```
      <surname>Сидорова</surname>
    </name>

    <birthday>17.05.1969</birthday>

    <address>
      <street>Ягодная, 17</street>
      <city>Жмеринка</city>
      <zip>23456K/zip>
    </address>

    <phone-list>
      <home-phone>2334455</home-phone>
    </phone-list>

  </person>

</notebook>
```

Документ XML начинается с необязательного пролога, состоящего из двух частей.

В первой части пролога — *объявлении XML* (XML declaration), — записанной в первой строке листинга 1.1, указывается версия языка XML, необязательная кодировка документа и отмечается, зависит ли этот документ от других документов XML (`standalone="yes"/"no"`). По умолчанию принимается кодировка UTF-8.

Все элементы документа XML обязательно должны содержаться в *корневом элементе* (root element), в листинге 1.1 это — элемент <notebook>. Имя корневого элемента считается именем всего документа и указывается во второй части пролога, называемой *объявлением типа документа* (Document Type Declaration). (Не путайте с определением типа документа DTD!) Имя документа записывается после слова DOCTYPE. Объявление типа документа записано во второй строке листинга 1.1. В этой части пролога после слова DOCTYPE и имени документа в квадратных скобках идет описание DTD:

```
<!DOCYPE notebook [ Сюда заносится описание DTD ]>
```

Очень часто описание DTD составляется сразу для нескольких документов XML. В таком случае его удобно записать отдельно от документа. Если описание DTD отделено от документа, то во второй части пролога вместо квадратных скобок записывается одно из слов SYSTEM ИЛИ PUBLIC. За словом SYSTEM идет URI (Universal Resource Identifier, универсальный идентифика-

тор ресурсов) файла с описанием DTD, а за словом PUBLIC, кроме того, можно записать дополнительную информацию.

XML-документ состоит из *элементов.* Элемент начинается *открывающим тегом,* потом идет необязательное *тело элемента* и в заключении — *закрывающий тег:*

```
<Открывающий тег>Тело элемента</Закрывающий тег>
```

Закрывающий тег содержит наклонную черту, после которой повторяется имя открывающего тега.

Язык XML, в отличие от языка HTML, требует обязательно записывать закрывающие теги. Если у элемента нет тела и закрывающего тега (empty — *пустой элемент),* то его открывающий тег должен заканчиваться символами "/>", например:

```
<br />
```

Сразу надо сказать, что язык XML, в отличие от **HTML,** различает регистры букв.

Из листинга 1.1 видно, что элементы документа XML могут быть вложены друг в друга. Надо следить за тем, чтобы элементы не пересекались, а полностью вкладывались друг в друга. Как уже говорилось выше, все элементы, составляющие документ, вложены в корневой элемент этого документа. Тем самым документ наделяется структурой дерева вложенных элементов. На рис. 1.1 показана структура адресной книжки, описанной в листинге 1.1.

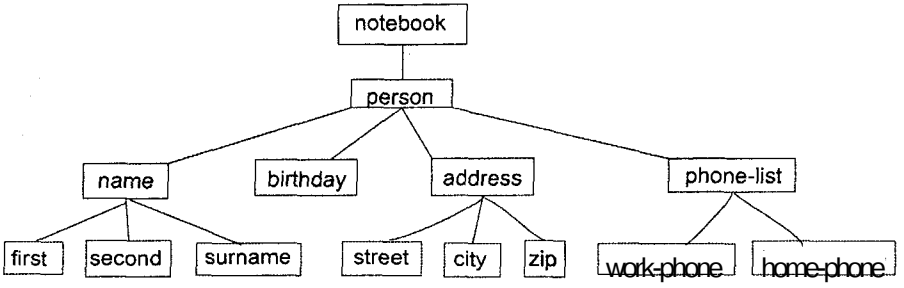


Рис. 1.1. Дерево элементов документа XML

У открывающих тегов XML могут быть *атрибуты.* Например, имя, отчество и фамилию можно записать как атрибуты first, second и surname тега <name>;

```
<name first="Иван" second="Петрович" surname="Сидоров" />
```

В отличие от языка HTML в языке XML значения атрибутов обязательно надо заключать в кавычки или апострофы.

Атрибуты удобны для описания простых значений. У каждого гражданина России, уважающего паспортный режим, обязательно есть одно имя, одно отчество и одна фамилия. Их удобно записывать атрибутами. Но у гражданина России может быть несколько телефонов, поэтому их номера удобнее оформить как элементы <work-phone> и <home-phone>, вложенные в элемент <phone-list>, а не атрибуты открывающего тега <phone-list>. Заметьте, что элемент <name> с атрибутами пустой, у него нет тела, следовательно, не нужен закрывающий тег. Поэтому тег <name> с атрибутами завершается символами "/>". В листинге 1.2 приведена измененная адресная книжка.

Листинг 1.2. Пример XML-документа с атрибутами в открывающем теге

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE notebook SYSTEM "ntb.dtd">

<notebook>

  <person>

    <name first="Иван" second="Петрович" surname="Сидоров" />

    <birthday>25.03.1977</birthday>

    <address>
      <street>Садовая, 23-15</street>
      <city>Урюпинск</city>
      <zip>123456</zip>
    </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
    </phone-list>

  </person>

  <person>
```

```
    <name first="Мария" second="Петровна" surname="Сидорова" />

    <birthday>17.05.1969</birthday>

    <address>
      <street>Ягодная, 17</street>
      <city>Жмеринка</city>
      <zip>23456K/zip>
    </address>

    <phone-list>
      <home-phone>2334455</home-phone>
    </phone-list>

  </person>

</notebook>
```

Атрибуты открывающего тега удобны и для указания типа элемента. Например, мы не уточняем, в городе живет наш родственник, в поселке или деревне. Можно ввести в тег <city> атрибут type, принимающий одно из значений: город, поселок, деревня. Например:

```
<city type="город">Москва</city>
```

Для описания адресной книжки нам понадобились открывающие теги <notebook>, <person>, <name>, <address>, <street>, <city>, <zip>, <phone-list>, <work-phone>, <home-phone> и соответствующие им закрывающие теги, помеченные наклонной чертой. Теперь необходимо дать их описание. В описании указываются только самые общие признаки логической взаимосвязи элементов и их тип.

- Элемент <notebook> может содержать в себе только нуль или больше элементов <person>.
- Элемент <person> содержит ровно один элемент <name>, нуль или несколько элементов <address> и нуль или один элемент <phone-list>.
- Элемент <name> пустой.
- В открывающем теге <name> три атрибута first, second, surname, значения которых — строки символов.
- Элемент <address> содержит по одному элементу <street>, <city> и <zip>.

- Элементы <street> и <city> содержат по одной текстовой строке.
- Элемент <zip> содержит одно целое число.
- У открывающего тега <city> есть один необязательный атрибут type, принимающий одно из трех значений город, поселок или деревня. Значение по умолчанию город.

❑ Необязательный элемент <phone-list> содержит нуль или более элементов <work-phone> И <home-phone>.

- Элементы <work-phone> и <home-phone> содержат по одной строке, состоящей только из цифр.

Это словесное описание, называемое *схемой* документа XML, формализуется несколькими способами. Наиболее распространены два способа: можно сделать описание DTD, пришедшее в XML из SGML, или описать схему на языке XSD.

Описание DTD

Описание DTD нашей адресной книжки записано в листинге 1.3.

Листинг 1.3. Описание DTD документа XML

```
<!ELEMENT notebook (person)*>
<!ELEMENT person (name, birthday?, address*, phone-list?)>
<!ELEMENT name EMPTY>
<!ATTLIST name
   first   CDATA #IMPLIED
   second  CDATA #IMPLIED
   surname CDATA #REQUIRED>
<!ELEMENT birthday (#PCDATA)>
<!ELEMENT address (street, city, zip)?>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ATTLIST city
   type (город | поселок I деревня) "город">
<!ELEMENT zip (#PCDATA)>
<!ELEMENT phone-list (work-phone*, home-phone*)>
<!ELEMENT work-phone (#PCDATA)>
<!ELEMENT home-phone (#PCDATA)>
```

Как видите, описание DTD почти очевидно. Оно повторяет приведенное выше словесное описание. Первое слово ELEMENT означает, что элемент может содержать тело с вложенными элементами. Вложенные элементы перечисляются в круглых скобках. Порядок перечисления вложенных элементов в скобках должен соответствовать порядку их появления в документе. Слово EMPTY в третьей строке листинга 1.3 означает пустой элемент.

Слово ATTLIST начинает описание списка атрибутов элемента. Для каждого атрибута указывается имя, тип и обязательность указания атрибута. Типов атрибута всего девять, но чаще всего употребляется тип CDATA (Character DATA), означающий произвольную строку символов Unicode, или перечисляются значения типа. Так сделано в описании атрибута type тега <city>, принимающего одно из трех значений город, поселок или деревня. В кавычках показано значение по умолчанию город.

Обязательность указания атрибута отмечается одним из трех слов:

- #REQUIRED — атрибут обязателен,
- #IMPLIED — атрибут необязателен,
- #FIXED — значение атрибута фиксировано, оно задается в DTD.

Первым словом могут быть, кроме слов ELEMENT ИЛИ ATTLIST, слова ANY, MIXED или ENTITY. Слова ANY и MIXED, означают, что элемент может содержать и простые данные и/или вложенные элементы. Слово ENTITY означает обозначение или адрес данных, записанный в описании DTD, так называемую *сущность*.

После имени элемента в скобках записываются вложенные элементы или тип данных, содержащихся в теле элемента. Тип PCDATA (Parsed Character DATA) означает строку символов Unicode, которую надо интерпретировать.

Звездочка, записанная после имени элемента, означает "нуль или более вхождений" данного элемента, а плюс — "одно или более вхождений". Вопросительный знак означает "нуль или один раз". Если эти символы относятся ко всем вложенным элементам, то их можно указать после круглой скобки, закрывающей список вложенных элементов.

Описание DTD можно занести в отдельный файл, например, ntb.dtd, указав его имя во второй части пролога, как показано во второй строке листингов 1.1 и 1.2. Можно включить описание во вторую часть пролога XML-файла, заключив его в квадратные скобки:

```
<!DOCTYPE notebook [ Описание DTD ]>
```

После того как создано описание DTD нашей реализации XML и написан документ, размеченный тегами этой реализации, следует проверить пра-

вильность их написания. Для этого есть специальные программы — *проверяющие анализаторы* (validating parsers). Все фирмы, разрабатывающие средства для работы с XML, выпускают бесплатные или коммерческие проверяющие анализаторы. Например, фирма IBM выпускает анализатор xml4j, входящий в состав сервера приложений WebSphere. Его можно использовать и отдельно, свободно загрузив с адреса **http://www.alphaworks.ibm.com/** архивы xml4j.jar, xerces.jar и xercesSamples.jar.

Проверяющий анализатор фирмы Sun Microsystems содержится в пакете классов JAXP (Java API for XML Processing, интерфейс программирования приложений на Java для обработки XML), входящем в состав и J2SDK Standard Edition, и J2SDK Enterprise Edition [10]. Кроме того, этот пакет можно загрузить отдельно или в составе пакета Java XML Pack с адреса **http://java.sun.com/xml/**.

Корпорация Microsoft поставляет проверяющий анализатор MSXML (Microsoft XML Parser), доступный по адресу **http://msdn.microsoft.com/xml/.**

Есть еще множество проверяющих анализаторов, но лидером среди них является, пожалуй, Apache Xerces 2, входящий во многие средства обработки документов XML, выпускаемые другими фирмами. Он свободно доступен по адресу **http://xml.apache.org/xerces2-j/**.

Ограниченные средства DTD не позволяют полностью описать структуру документа XML. В частности, описание DTD не указывает точное количество повторений вложенных элементов, оно не задает точный тип тела элемента. Например, в листинге 1.3 из описания DTD не видно, что в элементе `<birthday>` содержится дата рождения. Эти недостатки DTD привели к появлению других схем описания документов XML. Наиболее развитое описание дает язык XSD. Мы будем называть описание на этом языке просто *схемой XML* (XML Schema).

Посмотрим, как создаются схемы XML, но сначала познакомимся еще с одним понятием XML — пространством имен.

Пространство имен тегов

Поскольку в разных языках разметок — реализациях XML — могут встретиться одни и те же имена тегов и их атрибутов, имеющие совершенно разный смысл, а в документе XML их часто приходится смешивать, анализатору надо дать возможность их как-то различать. Для этого имена тегов и атрибутов снабжают префиксом, который отделяется от имени двоеточием. Префикс имени связывается с идентификатором, определяющим *пространство имен* (namespace). Все имена тегов и атрибутов, префиксы которых связаны с одним и тем же идентификатором, образуют одно пространство

имен. Префикс и идентификатор пространства имен определяются атрибутом xmlns следующим образом:

```
<ntb:notebook xmlns:ntb = "http://some.firm.com/2003/ntbml">
```

Как видите, префикс ntb только что определен, но его уже можно использовать в имени ntb:notebook. В дальнейшем имена тегов и атрибутов, которые мы хотим отнести к пространству имен http://some.firm.com/2003/ntbml, снабжаются префиксом ntb, например:

```
<ntb:city ntb:type="поселок">Горелово</ntb:city>
```

Имя вместе с префиксом, например, ntb:city, называется *расширенным* или *уточненным именем* QName (Qualified Name).

Идентификатор пространства имен должен иметь форму URI. Адрес URI, такой как http://some.firm.com/2003/ntbml, не имеет никакого значения и может не соответствовать никакому действительному адресу. Анализатор документа XML и другие программы, использующие документ, не будут обращаться по этому адресу. Там даже нет никакой Web-странички. Просто идентификатор пространства имен должен быть уникальным во всем Интернете, и разработчики рекомендации по применению пространства имен, которую можно посмотреть по адресу http://www.w3.org/TR/1999/REC-xml-**names-19990114/**, справедливо решили, что будет удобно использовать для него DNS-имя сайта, на котором размещено определение пространства имен. Смотрите на URI просто как на строку символов, идентифицирующую пространство имен. Обычно указывается URL (Universal Resource Locator, адрес универсальных ресурсов) фирмы, создавшей данную реализацию XML, или имя файла с описанием схемы XML.

По правилам SGML и XML двоеточие может применяться в именах как обычный символ, поэтому имя с префиксом — это просто фокус, анализатор рассматривает его как обычное имя. Отсюда следует, что в описании DTD нельзя опускать префиксы имен. Некоторым анализаторам надо специально указать необходимость учета пространства имен. Например, при работе с анализатором Xerces надо применить метод `setNamespaceAware(true)`.

Атрибут xmins может появиться в любом элементе XML, а не только в корневом элементе. Определенный им префикс можно применять в том элементе, в котором записан атрибут xmins, и во всех вложенных в него элементах. Больше того, в одном элементе можно определить несколько пространств имен:

```
<ntb:notebook
    xmlns:ntb = "http://some.firm.com/2003/ntbml"
    xmlns:bk  = "http://some.firm.com/2003/bookral">
```

Появление имени тега без префикса в документе, использующем пространство имен, означает, что имя принадлежит *пространству имен по умолчанию* (default namespace). Например, язык XHTML допускает применение тегов HTML и тегов XML в одном документе. Допустим, мы определили тег с именем `title`. Чтобы анализатор не принял его за один из тегов HTML, поступаем следующим образом:

```
<html xmlns = "http://www.w3.org/1999/xhtml"
    xmlns:ntb = "http://same.firm.com/2002/ntbml">

   <head>
      <title>Моя библиотека</title>
   </head>

   <body>
      <ntb:book>
         <ntb:title>Создание Java Web Services</ntb:title>
      </ntb:book>
   </body>

</html>
```

В этом примере пространством имен по умолчанию становится пространство имен XHTML, имеющее общеизвестный идентификатор http://www.w3.org/1999/xhtml, и теги, относящиеся к этому пространству имен, записываются без префикса.

Атрибуты никогда не входят в пространство имен по умолчанию. Если имя атрибута записано без префикса, то это означает, что атрибут не относится ни к одному пространству имен.

Теперь, после того как мы ввели понятие пространства имен, можно обратиться к схеме XML.

Схема XML

В мае 2001 года консорциум W3C (WWW Consortium) рекомендовал описывать структуру документов XML на языке описания схем XSD. На этом языке записываются *схемы XML* (XML Schema), описывающие элементы документов XML.

Схема XML сама записывается как документ XML. Его элементы называют *компонентами* (components), чтобы отличить их от элементов описываемого

документа XML. Корневой компонент схемы носит имя <schema>. Компоненты схемы описывают элементы XML и определяют различные типы элементов. Рекомендация схемы XML, которую можно посмотреть по адресу **http://www.w3.org/xml/schema/,** перечисляет 13 типов компонентов, но наиболее важны компоненты, определяющие простые и сложные типы элементов, сами элементы и их атрибуты.

Язык XSD различает простые и сложные элементы XML. *Простыми* (simple) элементами описываемого документа XML считаются элементы, не содержащие атрибутов и вложенных элементов. Соответственно, *сложные* (complex) элементы содержат атрибуты и/или вложенные элементы. Схема XML описывает *простые типы* — типы простых элементов, и *сложные типы* — типы сложных элементов.

Язык описания схем содержит много встроенных простых типов. Они перечислены в следующем разделе.

Встроенные простые типы XSD

Встроенные типы языка описания схем XSD позволяют записывать двоичные и десятичные целые числа, вещественные числа, дату и время, строки символов, логические значения, адреса URI. Рассмотрим их по порядку.

Вещественные числа

Вещественные числа в языке XSD разделены на три типа: decimal, float и double.

Тип decimal составляют вещественные числа, записанные с фиксированной точкой: 123.45, —0.1234567689345 и так далее. Фактически хранятся два целых числа: мантисса и порядок. Спецификация языка XSD не ограничивает количество цифр в мантиссе, но требует, чтобы можно было записать не менее 18 цифр. Этот тип легко реализуется классом `java.math.BigDecimal`, входящем в стандарт Java API (Application Programming Interface, интерфейс программирования приложений) [9].

Типы float и double соответствуют стандарту IEEE754–85 и одноименным типам Java [9]. Они записываются с фиксированной или с плавающей десятичной точкой.

Целые числа

Основной целый тип integer понимается как подтип типа decimal, содержащий числа с нулевым порядком. Это целые числа с любым количеством десятичных цифр: -34567, 123456789012345 и так далее. Этот тип легко реализуется классом `java.math.BigInteger` [9].

Типы long, int, short и byte полностью соответствуют одноименным типам Java. Они понимаются как подтипы типа integer, типы более коротких чисел считаются подтипами более длинных чисел: тип byte — это подтип типа short, оба они подтипы типа int, и так далее.

Типы nonPositiveInteger и negativeInteger — подтипы типа integer — составлены из неположительных и отрицательных чисел соответственно с любым количеством цифр.

Типы nonNegativeInteger и positiveInteger — подтипы типа integer — составлены из неотрицательных и положительных чисел соответственно с любым количеством цифр.

У типа nonNegativeInteger есть подтипы беззнаковых целых чисел: unsignedLong, unsignedInt, unsignedShort и unsignedByte.

Строки символов

Основной символьный тип string описывает произвольную строку символов Unicode. Его можно реализовать классом `java.lang.String`.

Тип normalizedString — подтип типа string — это строки, не содержащие символы перевода строки '\n', возврата каретки '\r' и символы горизонтальной табуляции '\t'.

В строках типа token — подтипа типа normaiizedstring — нет, кроме того, начальных и завершающих пробелов и нет нескольких подряд идущих пробелов.

В типе token выделены три подтипа. Подтип language определен для записи названия языка согласно RFC 1766, например, ru, en, de, fr. Подтип NMTOKEN используется только в атрибутах для записи их перечисляемых значений. Подтип name составляют *имена XML* — последовательности букв, цифр, дефисов, точек, двоеточий, знаков подчеркивания, начинающиеся с буквы (кроме зарезервированной последовательности букв X, x, M, m, L, 1 в любом сочетании регистров) или знака подчеркивания. Двоеточие в значениях типа name используется для выделения префикса пространства имен.

Из типа name выделен подтип NCName (Non-Colonized Name) имен, не содержащих двоеточия, в котором, в свою очередь, определены три подтипа ID, ENTITY, IDREF, описывающие идентификаторы XML, сущности и перекрестные ссылки.

Дата и время

Тип duration описывает промежуток времени, например, запись P1Y2M3DT10H30M45S означает один год (1Y), два месяца (2M), три дня (3D),

десять часов (10H), тридцать минут (зом) и сорок пять секунд (4 5s). Запись может быть сокращенной, например, P120M означает 120 месяцев, а T120M — 120 минут.

ТИП dateTime содержит дату И время В формате CCYY-MM-DDThh:mm:ss, Например, 2003-04-25T09:30:05. Остальные типы выделяют какую-либо часть даты или времени.

Тип time содержит время В обычном формате hh:mm:ss.

Тип date содержит дату в формате CCYY-MM-DD.

Тип gYearMonth выделяет год и месяц в формате CCYY-MM.

Тип gMonthDay содержит месяц и день месяца в формате -MM-DD.

Тип gYear означает год в формате CCYY, ТИП gMonth — месяц в формате -мм-, тип gDay — день месяца в формате -DD.

Двоичные типы

Двоичные целые числа записываются либо в 16-ричной форме без всяких ДОПОЛНИТеЛЬНЫХ СИМВОЛОВ: 0B2F, 356C0A И так далее, ЭТО — ТИП hexBinary, либо В кодировке Base64 [10], ЭТО — ТИП base64Binary.

Прочие встроенные простые типы

Еще три встроенных простых типа описывают значения, часто используемые в документах XML.

Адреса URI относятся к типу anyURI.

Расширенное имя тега или атрибута (qualified name, то есть, имя вместе с префиксом, отделенным от имени двоеточием, — это тип QName.

Элемент NOTATION описания DTD выделен как отдельный простой тип схемы XML. Его используют для записи математических, химических и других символов, нот, азбуки Бройля и прочих обозначений.

Определение простых типов

В схемах XML с помощью встроенных типов можно тремя способами определить новые типы простых элементов. Они вводятся как *сужение* (restriction) встроенного или ранее определенного простого типа, *список* (list) или *объединение* (union) простых типов.

Простой тип определяется компонентом схемы <simpleType>, имеющим вид

```
<xsd:simpleType name="имя типа">Определение типа</xsd:simpleType>
```

Сужение

Сужение простого типа определяется компонентом <restriction>, в котором атрибут base указывает сужаемый простой тип, а в теле задаются ограничения, выделяющие определяемый простой тип. Например, почтовый индекс zip можно определить как шесть арабских цифр следующим образом:

```
<xsd:simpleType name="zip">

  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{6}" />
  </xsd:restriction>

</xsd:simpleType>
```

Можно дать другое определение простого типа zip как целого положительного числа, находящегося в диапазоне от 100000 до 999999:

```
<xsd:simpleType name="zip">

  <xsd:restriction base="xsd:positiveInteger">
    <xsd:minInclusive value="100000" />
    <xsd:maxInclusive value="999999" />
  </xsd:restriction>

</xsd:simpleType>
```

Теги <pattern>, <maxInclusive> **и другие теги, задающие ограничения, называются** ***фасетками*** (facets). **Вот их список:**

- <maxExclusive> — **наибольшее значение, оно уже не входит в определяемый тип;**
- <maxInclusive> — **наибольшее значение определяемого типа;**
- **<minExclusive> — наименьшее значение, уже не входящее в определяемый тип;**
- <minInclusive> — **наименьшее значение определяемого типа;**
- <totalDigits> — **общее количество цифр в определяемом числовом типе — сужении типа** decimal;
- <fractionDigits> — **количество цифр в дробной части числа;**
- <length> —**длина значений определяемого типа;**

- <maxLength> — наибольшая длина значений определяемого типа;
- <minLength> — наименьшая длина значений определяемого типа;
- <enumeration> — одно из перечисляемых значений;
- <pattern> — регулярное выражение [8];
- <whitespace> — применяется при сужении типа string и определяет способ преобразования пробельных символов '\n', '\r', '\t' Атрибут value этого тега принимает одно из трех значений:
  - preserve — не убирать пробельные символы,
  - replace — заменить пробельные символы пробелами,
  - collapse — после замены пробельных символов пробелами убрать начальные и конечные пробелы, а из нескольких подряд идущих пробелов оставить только один.

В тегах-фасетках можно записывать следующие атрибуты, называемые *базисными фасетками* (fundamental facets):

- ordered — задает упорядоченность определяемого типа, принимает одно из трех значений:
  - false — тип неупорядочен,
  - partial — тип частично упорядочен,
  - total — тип полностью упорядочен;
- bounded — задает ограниченность или неограниченность типа значениями true ИЛИ false;
- cardinality — задает конечность или бесконечность типа значениями finite ИЛИ countably infinite;
- numeric — показывает, числовой этот тип или нет, значениями true или false.

Как видно из приведенных выше и ниже примеров, в одном сужении может быть несколько ограничений-фасеток. При этом фасетки <pattern> и <enumeration> задают независимые друг от друга ограничения, их можно мысленно объединить союзом "или". Остальные фасетки задают общие, совместно накладываемые ограничения, их можно мысленно объединить союзом "и".

Список

Простой тип-список — это тип элементов, в теле которых записывается, через пробел, несколько значений одного и того же простого типа. Напри-

мер, в документе XML может встретиться такой элемент, содержащий список целых чисел:

```
<days>21 34 55 4 6</days>
```

Список определяется компонентом <list>, в котором атрибутом itemType указывается тип элементов определяемого списка. Тип элементов списка можно определить и в теле элемента <list>. Например, показанный выше элемент документа XML <days> можно определить в схеме так:

```
<xsd:element name="days" type="listOfInteger" />
```

а использованный при его определении тип listOfinteger задать как список не более чем из пяти целых чисел следующим образом:

```
<xsd:simpleType name="listOfInteger">

  <xsd:restriction>

    <xsd:simpleType>
      <xsd:list itemType="xsd:integer" />
    </xsd:simpleType>

    <xsd:maxLength value="5" />

  </xsd:restriction>

</xsd:simpleType>
```

При определении списка можно применять фасетки <length>, <minLength>, <maxLength>, <enumeration>, <pattern>. В приведенном выше примере список — тело элемента <days> — не может содержать более пяти чисел.

Объединение

Простой тип-объединение определяется компонентом <union>, в котором атрибутом memberTypes можно указать имена объединяемых типов. Например:

```
<xsd:union memberTypes="xsd:string xsd:integer listOfInteger" />
```

Другой способ — записать в теле компонента <union> определения простых типов, входящих в объединение. Например:

```
<xsd:attribute name="size">

  <xsd:s impleType>
```

```
    <xsd:union>

      <xsd:simpleType>

        <xsd:restriction base="xsd:positiveInteger">
          <xsd:minInclusive value="8"/>
          <xsd:maxInclusive value="72"/>
        </xsd:restriction>

      </xsd:simpleType>

      <xsd:simpleType>

        <xsd:restriction base="xsd:NMTOKEN">
          <xsd:enumeration value="small"/>
          <xsd:enumeration value="medium"/>
          <xsd:enumeration value="large"/>
        </xsd:restriction>

      </xsd:simpleType>

    </xsd:union>
  </xsd:simpleType>

</xsd:attribute>
```

После этого атрибут size можно использовать, например, так:

```
<font size='large'>Глава K/font>
<font size='12'>Простой TeKCT</font>
```

Описание элементов и их атрибутов

Элементы, которые будут применяться в документе XML, описываются в схеме компонентом <element>:

```
<xsd:element name="имя элемента" type="тип элемента"
  minOccurs="наименьшее число появлений элемента в документе"
  maxOccurs="наибольшее число появлений" />
```

Значение по умолчанию необязательных атрибутов minoccurs и maxOccurs равно 1. Это означает, что если эти атрибуты отсутствуют, то элемент дол-

жен появиться в документе XML ровно один раз. Определение типа элемента можно вынести в тело элемента <element>:

```
<xsd:element пате="имя элемента" >
   Определение типа элемента
</xsd:element>
```

Описание атрибута элемента тоже несложно:

```
<xsd:attribute name=" имя атрибута" type="тип атрибута"
  use="обязательность атрибута" default="значение по умолчанию" />
```

Необязательный атрибут use принимает три значения:

- optional — описываемый атрибут необязателен (это значение по умолчанию);
- required — описываемый атрибут обязателен;
- prohibited — описываемый атрибут неприменим. Это значение полезно при определении подтипа, чтобы отменить некоторые атрибуты базового типа.

Если описываемый атрибут необязателен, то атрибутом default можно задать его значение по умолчанию.

Определение типа атрибута, — а это должен быть простой тип, — можно вынести в тело элемента <attribute>:

```
<xsd:attribute пате="имя атрибута">
   Тип атрибута
</xsd:attribute>
```

Определение сложных типов

Напомним, что тип элемента называется сложным, если в элемент вложены другие элементы и/или в открывающем теге элемента есть атрибуты.

Сложный тип определяется компонентом <complexType>, имеющим вид:

```
<xsd:complexType пате="имя типа" >Определение типа</xsd:complexType>
```

Необязательный атрибут name задает имя типа, а в теле компонента <complexType> описываются элементы, входящие в сложный тип, и/или атрибуты открывающего тега.

Определение сложного типа можно разделить на определение типа пустого элемента, элемента с простым телом, и элемента, содержащего вложенные элементы. Рассмотрим эти определения подробнее.

Определение типа пустого элемента

Проще всего определяется тип пустого элемента — элемента, не содержащего тела, а содержащего только атрибуты в открывающем теге. Таков, например, элемент <name> листинга 1.2. Каждый атрибут описывается одним компонентом <attribute>, как в предыдущем разделе, например:

```
<xsd:complexType name="imageType">
  <xsd:attribute name="href" type="xsd:anyURI" />'
</xsd:complexType>
```

После этого определения можно в схеме описать элемент <image> типа imageType:

```
<xsd:element name="image" type="imageType" />
```

а в документе XML использовать это описание:

```
<image href="http://some.com/images/myface.gif" />
```

Определение типа элемента с простым телом

Немного сложнее описание элемента, содержащего тело простого типа и атрибуты в открывающем теге. Этот тип отличается от простого типа только наличием атрибутов и определяется компонентом <simpleContent>. В теле этого компонента должен быть либо компонент <restriction>, либо компонент <extension>с атрибутом base, задающим тип (простой) тела описываемого элемента.

В компоненте <extension> описываются атрибуты открывающего тега описываемого элемента. Все вместе выглядит так, как в следующем примере:

```
<xsd:complexType name="calcResultType">

   <xsd:simpleContent>

      <xsd:extension base="xsd:decimal">
         <xsd:attribute name="unit" type="xsd:string" />
         <xsd:attribute name="precision"
             type="xsd:nonNegativeInteger" />
      </xsd:extension>

   </xsd:simpleContent>

</xsd:complexType>
```

Эту конструкцию можно описать словами так: "Определяется тип calcResultType элемента, тело которого содержит значения встроенного

простого типа xsd:decimal. Простой тип расширяется тем, что к нему добавляются атрибуты unit И precision".

Если в схеме описать элемент <result> этого типа следующим образом:

```
<xsd:element name="result" type="calcResultType" />
```

то в документе XML можно написать

```
<result unit="cM" precision="2">123.25</result>
```

В компоненте <restriction>, кроме атрибутов, описывается простой тип тела элемента и/или фасетки, ограничивающие тип, заданный атрибутом base. Например:

```
<xsd:complexType name="calcResultType">

   <xsd:simpleContent>

      <xsd:restriction base="xsd:decimal">
         <xsd:totalDigits value="8" />
         <xsd:attribute name="unit" type="xsd:string" />
         <xsd:attribute name="precision"
             type="xsd:nonNegativeInteger" />
      </xsd:restriction>

   </xsd:simpleContent>

</xsd:complexType>
```

Определение типа вложенных элементов

Если значениями определяемого сложного типа будут элементы, содержащие вложенные элементы, как, например, элементы <address>, <phone-list> листинга 1.2, то перед тем, как перечислять описания вложенных элементов, надо выбрать *модель группы* (model group) вложенных элементов. Дело в том, что вложенные элементы, составляющие определяемый тип, могут появляться или в определенном порядке, или в произвольном порядке, кроме того, можно выбирать только один из перечисленных элементов. Эта возможность и называется *моделью группы* элементов. Она определяется одним из трех компонентов <sequence>, <all> ИЛИ <choice>.

Компонент <sequence> применяется в том случае, когда перечисляемые элементы должны записываться в документе в определенном порядке. Пусть, например, мы описываем книгу. Сначала определяем тип:

```
<xsd:complexType name="bookType">

  <xsd:sequence maxOccurs="unbounded">

    <xsd:element name="author" type="xsd:normalizedString"
      minOccurs="0" />

    <xsd:element name="title" type="xsd:normalizedstring" />

    <xsd:element name="pages" type="xsd:positiveInteger"
      minOccurs="0" />

    <xsd:element name="publisher" type="xsd:normalizedString"
      minOccurs="0" />

  </xsd:sequence>

</xsd:complexType>
```

Потом описываем элемент:

```
<xsd:element name="book" type="bookType" />
```

Элементы <author>, <title>, <pages> И <publisher> ДОЛЖНЫ входить в элемент <book> именно в таком порядке. В документе XML надо писать:

```
<book>
  <author>И. Ильф, Е. Петров</author>
  <title>Золотой теленок</title>
  <publisher>Детская литература</publisher>
</book>
```

Если же вместо компонента <xsd:sequence> записать компонент <xsd:all>, то элементы <author>, <title>, <pages> и <publisher> можно перечислять в любом порядке.

Компонент <choice> применяется в том случае, когда надо выбрать один из нескольких элементов. Например, при описании журнала вместо издательства, описываемого элементом <publisher>, надо записать название журнала. Это можно определить так:

```
<xsd:complexType name="bookType">

  <xsd:sequence maxOccurs="unbounded">

    <xsd:element name="author" type="xsd:normalizedString"
      minOccurs="0" />
```

```
    <xsd:element name="title" type="xsd:normalizedString" />

    <xsd:element name="pages" type="xsd:positiveInteger"
      minOccurs="0" />

    <xsd:choice>

      <xsd:element name="publisher" type="xsd:normalizedString"
          minOccurs="0" />

      <xsd:element name="magazine" type="xsd:normalizedString"
          minOccurs="0" />

    </xsd:choice>

  </xsd:sequence>

</xsd:complexType>
```

Как видно из этого примера, компонент <choice> можно вложить в компонент <sequence>. Можно, наоборот, вложить компонент <sequence> в компонент <choice>. Такие вложения можно проделать сколько угодно раз. Кроме того, каждая группа в этих моделях может появиться сколько угодно раз, то есть, в компоненте <choice> тоже можно записать атрибут `maxOccurs="unbounded"`.

Модель группы <all> отличается в этом от моделей <sequence> и <choice>. В компоненте `<all>` не допускается применение компонентов <sequence> и <choice>. Обратно, в компонентах <sequence> и <choice> нельзя применять компонент `<all>`. Каждый элемент, входящий в группу модели <all>, может появиться не более одного раза, то есть, атрибут maxoccurs этого элемента может равняться только единице.

Определение типа со сложным телом

При определении сложного типа можно воспользоваться уже определенным, *базовым,* сложным типом, расширив его дополнительными элементами, или, наоборот, удалив из него некоторые элементы. Для этого надо применить компонент `<complexContent>`. В этом компоненте, так же как и в компоненте `<simpleContent>`, записывается либо компонент <extension>, если надо расширить базовый тип, либо компонент <restriction>, если надо сузить базовый тип. Базовый тип указывается атрибутом base, так же как и при записи компонента `<simpleContent>`, но теперь это должен быть сложный, а не простой тип!

Расширим, например, определенный выше тип Ьооктуре, добавив год издания — элемент <year>:

```
<xsd:complexType name="newBookType">

  <xsd:complexContent>

    <xsd:extension base="bookType">

      <xsd:sequence>
        <xsd:element name="year"  type="xsd:gYear">
      </xsd:sequence>

    </xsd:extension>

  </xsd:complexContent>

</xsd:complexType>
```

При сужении базового типа компонентом <restriction> надо перечислить те элементы, которые останутся после сужения. Например, оставим в типе newbookType только автора и название книги из типа bookType:

```
<xsd:complexType name="newBookType">

  <xsd:complexContent>

    <xsd:restriction base="bookType">

      <xsd:sequence>
         <xsd:element name="author" type="xsd:normalizedString"
             minOccurs="0" />
         <xsd:element name="title" type="xsd:normalizedString" />
      </xsd:sequence>

    </xsd:restriction>

  </xsd:complexContent>

</xsd:complexType>
```

Это описание выглядит странно. Почему надо заново описывать все элементы, остающиеся после сужения? Не проще ли определить новый тип?

Дело в том, что в язык XSD внесены элементы объектно-ориентированного программирования, которых мы не будем касаться. Расширенный и суженный типы связаны со своим базовым типом отношением наследования, и к ним можно применить операцию подстановки. У всех типов языка XSD есть общий предок — базовый тип anyType. От него наследуются все сложные типы. Это подобно тому, как у всех классов Java есть общий предок — класс `Object`, а все массивы наследуются от него. От базового типа anyType наследуется и тип `anySimpleType` — общий предок всех простых типов.

Таким образом, сложные типы определяются как сужение типа anyType. Если строго подходить к определению сложного типа, то определение типа `bookType`, сделанное в начале предыдущего раздела, надо записать так:

```
<xsd:complexType name="bookType">
  <xsd:complexContent>
    <xsd:restriction base="xsd:anyType">
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="author" type="xsd:normalizedString"
          minOccurs="0" />
        <xsd:element name="title" type="xsd:normalizedString" />
        <xsd:element name="pages" type="xsd:positiveInteger"
          minOccurs="0" />
        <xsd:element name="publisher" type="xsd:normalizedString"
          minOccurs="0" />
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Рекомендация языка XSD позволяет сократить эту запись, что мы и сделали в предыдущем разделе. Это подобно тому, как в Java мы опускаем слова "extends object" в заголовке описания класса.

Закончим на этом описание языка XSD и перейдем к примерам.

Пример: схема адресной книги

В листинге 1.4 записана схема документа, приведенного в листинге 1.2.

Листинг 1.4. Схема документа XML

```
<?xml version="1.0"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="notebook" type="notebookType" />

<xsd:complexType name="notebookType">
    <xsd:element name="person" type="personType"
       minOccurs="0" maxOccurs="unbounded" />

</xsd:complexType>

<xsd:complexType name="personType">
<xsd:sequence>

  <xsd:element name="name">

    <xsd:complexType>
      <xsd:attribute name="first"   type="xsd:string"
         use="optional" />
      <xsd:attribute name="second"  type="xsd:string"
         use="optional" />
      <xsd:attribute name="surname" type="xsd:string"
         use="required" />
    </xsd:complexType>

  </xsd:element>
```

```
  <xsd:element name="birthday" type="ruDate" minOccurs="0" />

  <xsd:element name="address" type="addressType"
     minOccurs="0" maxOccurs="unbounded" />

  <xsd:element name="phone-list" type="phone-listType"
     minOccurs="0" />

</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="addressType" >
<xsd:sequence>

  <xsd:element name="street" type="xsd:string" />
  <xsd:element name="city"  type="cityType" />
  <xsd:element name="zip"   type="xsd:positiveInteger" />

</xsd:sequence>
</xsd:complexType>

<xsd:complexType name='cityType'>

  <xsd:simpleContent>
     <xsd:extension base='xsd:string' >
       <xsd:attribute name='type' type='placeType'
           default='город' />
     </xsd:extension>
  </xsd:simpleContent>

</xsd:complexType>

<xsd:simpleType name="placeType">

  <xsd:restriction base = "xsd:string">
    <xsd:enumeration value="город" />
    <xsd:enumeration value="поселок" />
    <xsd:enumeration value="деревня" />
```

```
  </xsd:restriction>

</xsd:simpleType>

<xsd:complexType name="phone-listType">

    <xsd:element name="work-phone" type="xsd:string"
       minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="home-phone" type="xsd:string"
       minOccurs="0" maxOccurs="unbounded" />

</xsd:complexType>

<xsd:simpleType name="ruDate">

  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{2}.[0-9]{2}.[0-9]{4}" />
  </xsd:restriction>

</xsd:simpleType>

</xsd:schema>
```

Листинг 1.4, как обычный документ XML, начинается с пролога, показывающего версию XML и определяющего стандартное пространство имен схемы XML с идентификатором http://www.w3.org/2001/XMLSchema. Этому идентификатору дан префикс xsd." Конечно, префикс может быть другим, часто пишут префикс xs.

Все описание схемы нашей адресной книжки заключено в одной третьей строке, в которой указано, что адресная книга состоит из одного элемента с именем notebook, имеющего сложный тип notebookType. Этот элемент должен появиться в документе ровно один раз. Остаток листинга 1.4 посвящен описанию типа этого элемента и других типов.

Описание сложного типа notebookType несложно (простите за каламбур). Оно занимает три строки листинга, не считая открывающего и закрывающего тега, и просто говорит о том, что данный тип составляют несколько элементов `person` типа `personType`.

Описание типа personType немногим сложнее. Оно говорит, что этот тип составляют четыре элемента name, birthday, address И phone-list. Для элемента name сразу же определены необязательные атрибуты first и

second простого типа string, определенного в пространстве имен xsd. Тип обязательного атрибута surname — тоже string.

Далее в листинге 1.4 определяются оставшиеся типы addressType, phoneListType и ruDate. Необходимость определения простого типа ruDate возникает потому, что встроенный в схему XML тип date предписывает записывать дату в виде 2003-02-22, а в России принят формат 22.02.2003. Тип ruDate определяется как сужение (restriction) типа string с помощью шаблона. Шаблон (pattern) для записи даты в виде ДД.ММ.ГГГГ задается регулярным выражением [8].

Безымянные типы

Все описанные в листинге 1.4 типы используются только один раз. Поэтому необязательно давать типу имя. Схема XML, как говорилось выше, позволяет определять безымянные типы. Такое определение дается внутри описания элемента. Именно так в листинге 1.4 описаны атрибуты элемента name. В листинге 1.5 показано упрощенное описание схемы адресной книги.

Листинг 1.5. Схема документа XML с безымянными типами

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

<xsd:element name='notebook'>
<xsd:complexType>
<xsd:sequence>

  <xsd:element name='person' maxOccurs='unbounded'>
  <xsd:complexType>
  <xsd:sequence>

    <xsd:element name='name'>
    <xsd:complexType>

      <xsd:attribute name='first'
          type='xsd:string' use='optional' />

      <xsd:attribute name='second'
          type='xsd:string' use='optional' />
```

```
    <xsd:attribute name='surname'
        type='xsd:string' use='required' />

</xsd:complexType>
</xsd:element>

<xsd:element name='birthday'>
<xsd:simpleType>

    <xsd:restriction base='xsd:string'>
       <xsd:pattern value='[0-9]{2}.[0-9]{2}.[0-9]{4}' />
    </xsd:restriction>

</xsd:simpleType>
</xsd:element>

<xsd:element name='address' maxOccurs='unbounded'>
<xsd:complexType>
<xsd:sequence>

    <xsd:element name='street' type='xsd:string' />
    <xsd:element name='city'>
    <xsd:complexType>
    <xsd:simpleContent>

       <xsd:extension base='xsd:string'>
           <xsd:attribute name='type' type='xsd:string'
               use='optional' default='gorod' />
       </xsd:extension>

    </xsd:simpleContent>
    </xsd:complexType>
    </xsd:element>

    <xsd:element name='zip' type='xsd:positiveInteger' />

</xsd:sequence>
</xsd:complexType>
```

```
    </xsd:element>

    <xsd:element name='phone-list'>
    <xsd:complexType>
    <xsd:sequence>

       <xsd:element name='work-phone' type='xsd:string'
            minOccurs='0' maxOccurs='unbounded' />

       <xsd:element name='home-phone' type='xsd:string'
            minOccurs='0' maxOccurs='unbounded' />

    </xsd:sequence>
    </xsd:complexType>
    </xsd:element>

    </xsd:sequence>
  </xsd:complexType>
  </xsd:element>

  </xsd:sequence>
  </xsd:complexType>

</xsd:element>

</xsd:schema>
```

Еще одно упрощение можно сделать, используя пространство имен по умолчанию. Посмотрим, какие пространства имен применяются в схемах XML.

Пространства имен языка XSD

Имена элементов и атрибутов, используемые при записи схем, определены в пространстве имен с идентификатором http://www.w3.org/2001/XMLSchema. Префикс имен, относящихся к этому пространству, часто называют `xs` или xsd, как в листингах 1.4 и 1.5. Каждый анализатор "знает" это пространство имен и "понимает" имена из этого пространства.

Можно сделать это пространство имен пространством по умолчанию, но тогда надо определить пространство имен для определяемых в схеме типов

и элементов. Для удобства такого определения введено понятие *целевого пространства имен* (target namespace). Идентификатор целевого пространства имен определяется атрибутом targetNamespace, например

```
<xsd:schema targetNamespace="http://some.firm.com/2003/ntbNames">
```

После такого определения имена, определяемые в этой схеме, будут относиться к новому пространству имен с идентификатором http://some.firm.com/2003/ntbNames. В листинге 1.6 вводится целевое пространство имен, а для упрощения записи листинга 1.5 стандартное пространство имен схемы XML с идентификатором http://www.w3.org/2001/XMLSchema сделано пространством имен по умолчанию. Имена, относящиеся к целевому пространству, снабжены префиксом ntb, чтобы они не попали в пространство имен по умолчанию.

Листинг 1.6. Схема документа XML с целевым пространством имен

```
<?xml version='1.0'?>
<schema xmlns='http://www.w3.org/2001/XMLSchema'
   targetNamespace='http://some.firm.com/2003/ntbNames'
   xmlns:ntb='http://some.firm.com/2003/ntbNames'>

<element name='ntb:notebook'>

  <complexType>
    <sequence>

      <element name='person' maxOccurs='unbounded'>
        <complexType>
          <sequence>

            <element name='name'>
              <complexType>

                <attribute name='first'
                   type='string' use='optional' />

                <attribute name='second'
                   type='string' use='optional' />

                <attribute name='surname'
                   type='string' use='required' />
```

```
  </complexType>
</element>

<element name='birthday'>
  <simpleType>

    <restriction base='string'>
      <pattern value='[0-9]{2}.[0-9]{2}.[0-9]{4}' />
    </restriction>

  </simpleType>
</element>

<element name='address ' maxOccurs='unbounded'>
  <complexType>
    <sequence>

      <element name='street' type='string' />
      <element name='city' type='string' />
      <element name='zip' type='positiveInteger' />

    </sequence>
  </complexType>
</element>

<element name='phone-list'>
  <complexType>
    <sequence>

      <element name='work-phone' type='string'
        minOccurs='0' raaxOccurs='unbounded'/>

      <element name='home-phone' type='string'
        minOccurs='0' maxOccurs='unbounded'/>

    </sequence>
  </coraplexType>
</element>
```

```
        </sequence>
      </complexType>
    </element>

  </sequence>
</complexType>

</element>

</schema>
```

Поскольку в листинге 1.6 пространством имен по умолчанию сделано пространство http://www.w3.org/2001/XMLSchema, префикс xsd не нужен.

Следует заметить, что в целевое пространство имен попадают только *глобальные имена,* чьи описания непосредственно вложены в элемент <schema>. Это естественно, потому что только глобальными именами можно воспользоваться далее в этой или другой схеме. В листинге 1.6 только одно глобальное имя <notebook>. Вложенные имена name, address и другие только *ассоциированы* с глобальными именами.

В схемах и документах XML часто применяется еще одно стандартное пространство имен. Рекомендация языка XSD определяет несколько атрибутов: type, nil, schemaLocation, noNamespaceSchemaLocation, которые применяются не только в схемах, а и непосредственно в описываемых этими схемами документах XML, называемых *экземплярами схем* (XML schema instance). Имена этих атрибутов относятся к пространству имен http://www.w3.org/2001/XMLSchema-instance. Этому пространству имен чаще всего приписывают префикс xsi, например:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Включение файлов схемы в другую схему

В создаваемую схему можно включить файлы, содержащие другие схемы. Для этого есть два элемента схемы: <include> и <import>. Например:

```
<xsd:include xsi:schemaLocation="names.xsd" />
```

Включаемый файл задается атрибутом xsi: schemaLocation. В примере он использован для того, чтобы включить в создаваемую схему содержимое файла names.xsd. Файл должен содержать схему с описаниями и определениями из того же пространства имен, что и в создаваемой схеме, или без пространства имен, то есть в нем не использован атрибут targetNamespace.

Это удобно, если мы хотим добавить к создаваемой схеме определения схемы names.xsd или просто хотим разбить большую схему на два файла. Можно представить себе результат включения так, как будто содержимое файла names.xsd просто записано на месте элемента <include>.

Перед включением файла можно изменить некоторые определения, приведенные в нем. Для этого используется элемент <redefine>, например:

```
<xsd:redefine schemaLocation="names.xsd">

  <xsd:simpleType name="nameType">

    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="40"/>
    </xsd:restriction>

  </xsd:simpleType>

</xsd:redefine>
```

Если же включаемый файл содержит имена из другого пространства имен, то надо воспользоваться элементом схемы <import>. Например, пусть файл A.xsd начинается со следующих определений:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://some.firm.com/someNames">,
```

а файл B.xsd начинается с определений

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://some.firm.com/anotherNames">
```

Мы решили включить эти файлы в новый файл C.xsd. Это делается так:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://some.firm.com/yetAnotherNames"
    xmlns:pr1="http://some.firm.com/someNames"
    xmlns:pr2="http://some.firm.com/anotherNames">

<xsd:import namespace="http://some.firm.com/someNames"
    xsi:schemaLocation="A.xsd" />
<xsd:import namespace="http://some.firm.com/anotherNames"
    xsi:schemaLocation="B.xsd" />
```

После этого в файле C.xsd можно использовать имена, определенные в файлах A.xsd и B.xsd, снабжая их префиксами `pr1` и `pr2` соответственно.

Элементы `<include>` и `<import>` следует располагать перед всеми определениями схемы.

Значение атрибута `xsi:schemaLocation` — строка URI, поэтому файл с включаемой схемой может располагаться в любом месте Интернета.

Связь документа XML со своей схемой

Программе-анализатору, проверяющей соответствие документа XML его схеме, надо как-то указать файлы (один или несколько), содержащие схему документа. Это можно сделать разными способами. Во-первых, можно подать эти файлы на вход анализатора. Так делает, например, проверяющий анализатор XSV (XML Schema Validator) (**ftp://ftp.cogsci.ed.ac.uk/pub/XSV/**):

```
$ xsv ntb.xml ntb1.xsd ntb2.xsd
```

Во-вторых, можно задать файлы со схемой как свойство анализатора, устанавливаемое методом `setProperty()`, или значение переменной окружения анализатора. Так делает, например, проверяющий анализатор Xerces.

Эти способы удобны в тех случаях, когда документ в разных случаях нужно связать с разными схемами. Если же схема документа фиксирована, то ее удобнее указать прямо в документе XML. Это делается одним из двух способов.

1. Если элементы документа не принадлежат никакому пространству имен и записаны без префикса, то в корневом элементе документа записывается атрибут `noNamespaceSchemaLocation`, указывающий расположение файла со схемой в форме URI:

```
<notebook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ntb.xsd">
```

В этом случае в схеме не должно быть целевого пространства имен, то есть, не следует использовать атрибут `targetNamespace`.

2. Если же элементы документа относятся к некоторому пространству имен, то применяется атрибут schemaLocation, в котором через пробел перечисляются пространства имен и расположение файла со схемой, описывающей это пространство имен. Продолжая пример предыдущего раздела, можно написать:

```
<notebook xmlns="http://some.firm.com/2003/ntbNames"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://some.firm.com/someNamesA.xsd
```

```
        http://some.firm.com/anotherNames B.xsd"
    xmlns:pr1="http://some.firm.com/someNames"
    xmlns:pr2="http://some.firm.com/anotherNames">
```

После этого в документе можно использовать имена, определенные в схемах A.xsd и B.xsd, снабжая их префиксами `pr1` и `pr2` соответственно.

Другие языки описания схем

Даже из приведенного выше краткого описания языка XSD видно, что он получился весьма сложным и запутанным. Есть уже несколько книг, полностью посвященных этому языку. Их объем ничуть не меньше объема этой книги.

Есть и другие, более простые языки описания схемы документа XML. Наибольшее распространение получили следующие языки:

- Schematron — **http://www.ascc.net/xml/resource/schematron/**,
- RELAX NG (Regular Language Description for XML, New Generation, регулярный язык описания XML нового поколения) — **http://www.oasis-open.org/committees/relax-ng/;** этот язык возник как слияние языков Relax и TREX,
- Relax — **http://www.xml.gr.jp/relax/**,
- TREX (Tree Regular Expressions for XML, древовидные регулярные выражения для XML) — **http://www.thaiopensource.com/trex/**,
- DDML (Document Definition Markup Language, язык разметки определения документа), известный еще как XSchema — **http://purl.oclc.org/NET/ddml/.**

Менее распространены языки DCD (Document Content Description, описание содержания документа), SOX (One's Schema for Object-Oriented XML, схема для объектно-ориентированного XML), XDR (XML-Data Reduced, редуцированные XML-данные).

Все эти языки позволяют более или менее полно описывать схему документа. Возможно, они вытеснят язык XSD, возможно, будут существовать совместно.

Инструкции по обработке

Упомянем еще одну конструкцию языка XML — *инструкции по обработке* (processing instructions). Инструкции по обработке позволяют передать анализатору или другой программе-обработчику документа дополнительные сведения для обработки. Инструкция по обработке выглядит так:

```
<? сведения для анализатора ?>
```

Первая часть пролога документа XML — первая строка XML-файла — это как раз инструкция по обработке. Она передает анализатору документа версию языка XML и кодировку символов, которыми записан документ.

Первая часть работы закончена. Документы XML и их схемы написаны. Теперь надо подумать о том, каким образом они будут отображаться на экране дисплея, на листе бумаги, на экране сотового телефона, то есть, надо подумать о визуализации документа XML.

Прежде всего, документ надо разобрать, проанализировать (parse) его структуру.

Анализ документа XML

На первом этапе разбора проводится *лексический анализ* (lexical parsing) документа XML. Документ разбивается на отдельные неделимые элементы (tokens), которыми являются теги, служебные слова, разделители, текстовые константы. Проводится проверка полученных элементов и их связей между собой. Лексический анализ выполняют специальные программы — *сканеры* (scanners). Простейшие сканеры — это классы `java.util.StringTokenizer` и java.io.streamTokenizer из стандартной поставки Java 2 SDK Standard Edition.

Затем выполняется *грамматический анализ* (grammar parsing). При этом анализируется логическая структура документа, составляются выражения, выражения объединяются в блоки, блоки — в модули, которыми могут являться абзацы, параграфы, пункты, главы. Грамматический анализ проводят программы-анализаторы, так называемые *парсеры* (parsers).

Создание сканеров и парсеров — любимое развлечение программистов. За недолгую историю XML написаны десятки, если не сотни XML-парсеров. Многие из них написаны на языке Java. Все парсеры можно разделить на две группы.

В первую группу входят парсеры, проводящие анализ, основываясь на структуре дерева, отражающего вложенность элементов документа (tree-based parsing). Такие парсеры проще реализовать, но создание дерева требует большого объема оперативной памяти, ведь размер документов XML не ограничен. Необходимость частого просмотра дерева замедляет работу парсера.

Во вторую группу входят парсеры, проводящие анализ, основываясь на событиях (event-based parsing). Событием считается появление какого-либо элемента XML: открывающего или закрывающего тега, текста, содержащегося в теле элемента. При возникновении события вызывается какой-нибудь метод его обработки: `startElement()`, `endElement()`, `characters()`

Такие парсеры сложнее в реализации, зато они не строят дерево в оперативной памяти и могут анализировать не весь документ, а его отдельные элементы вместе с вложенными в них элементами. Фактическим стандартом здесь стал свободно распространяемый набор классов и интерфейсов SAX (Simple **API** for XML Parsing, простой **API** для анализа XML), созданный Давидом Меггинсоном (David Megginson). Основной сайт этого проекта **http://www.saxproject.org/.** Сейчас применяется второе поколение этого набора, называемое SAX2. Набор SAX2 входит во многие парсеры, например, Xerces2.

В стандартную поставку Java 2 Standard Edition и Enterprise Edition входит JAXP — набор интерфейсов и классов для создания парсеров и преобразования документов XML. С помощью одной из частей этого набора, называемой DOM API (Document Object Model API, **API** объектной модели документов), можно создавать парсеры первого типа, создающие дерево объектов, а с помощью второй части набора JAXP, называемой SAX API, можно создавать SAX-парсеры. Интерфейсы и классы SAX2 собраны **В пакеты** `org.xml.sax`, `org.xml.sax.ext`, `org.xml.sax.helpers`, `javax.xml.parsers`. **Рассмотрим их подробнее.**

Анализ документов XML с помощью SAX2API

Основу SAX2 составляет интерфейс `org.xml.sax.ContentHandler`, **описывающий** методы обработки событий: начало документа, появление открывающего тега, появление тела элемента, появление закрывающего тега, окончание документа. При возникновении такого события SAX2 обращается к методу-обработчику события, передавая ему аргументы, содержащие информацию о событии. Дело разработчика — реализовать эти методы, обеспечив правильный анализ документа.

В начале обработки документа вызывается метод

```
public void startDocument();
```

В нем можно задать начальные действия по обработке документа.

При появлении символа '<', начинающего открывающий тег, вызывается метод

```
public void startElement(String uri, String name,
    String qname, Attributes attrs);
```

В метод передается три имени, два из которых связаны с пространством имен: идентификатор пространства имен `uri`, имя тега без префикса `name` и расширенное имя с префиксом `qname`, а также атрибуты открывающего тега элемента `attrs`, если они есть. Если пространство имен не определено, то

значения первого и второго аргумента null. Если нет атрибутов, то передается ссылка на пустой объект attrs.

При появлении символов "</", начинающих закрывающий тег, вызывается метод

```
public void endElement(String uri, String name, String qname);
```

При появлении строки символов вызывается метод

```
public void characters(char[] ch, int start, int length);
```

В него передается массив символов ch, индекс начала строки символов start в этом массиве и количество символов length.

При появлении в тексте документа инструкции по обработке вызывается метод

```
public void processingInstruction(String target, String data);
```

В метод передается имя программы-обработчика target и дополнительные сведения data.

При появлении пробельных символов, которые должны быть пропущены, вызывается метод

```
public void ignorableWhitespace(char[] ch, int start, int length);
```

В него передается массив ch идущих подряд пробельных символов, индекс начала символов в массиве start и количество символов length.

Интерфейс `org.xml.sax.ContentHandler` реализован классом `org.xml.sax.helpers.DefaultHandler`. В нем сделана пустая реализация всех методов. Разработчику остается реализовать только те методы, которые ему нужны.

Применим методы SAX2 для обработки нашей адресной книжки. Запись документа на языке XML удобна для выявления структуры документа, но неудобна для работы с документом в объектно-ориентированной среде. Поэтому чаще всего содержимое документа XML представляется в виде одного или нескольких объектов, называемых *объектами данных* JDO (Java Data Objects). Эта операция называется *связыванием данных* (data binding) с объектами JDO.

Свяжем содержимое нашей адресной книжки с объектами Java. Для этого сначала опишем классы Java, которые представят содержимое адресной книги.

Листинг 1.7. Класс, описывающий адрес

```
public class Address)

  private String street, city, zip, type = "город";
```

```
  public Address() {}

  public String getStreet(){ return street; }
  public void setStreet(String street){ this.street = street; }

  public String getCity(){ return city; }
  public void setCity(String city){ this.city = city; }

  public String getZip(){ return zip; }
  public void setZip(String zip){ this.zip = zip; }

  public String getType(){ return type; }
  public void setType(String type){ this.type = type; }

  public String toString(){
    return "Address: " + street + " " + city + " " + zip;
  }
}
```

Листинг 1.8. **Класс**, описывающий запись адресной книжки

```
public class Person{

  private String firstName, secondName, surname, birthday;
  private Vector address;
  private Vector workPhone;
  private Vector homePhone;

  public Person(){}

  public Person(String firstName, String secondName,
                String surname){

    this.firstName = firstName;
    this.secondName = secondName;
    this.surname = surname;
  }

  public String getFirstName(){ return firstName; }
  public void setFirstName(String firstName){
```

```
    this.firstName = firstName;
  }

  public String getSecondName(){ return secondName; }
  public void setSecondName(String secondName){
    this.secondName = secondName;
  }

  public String getSurname(){ return surname; }
  public void setSurname(String surname){
     this.surname = surname;
  }

  public String getBirthday(){ return birthday; }
  public void setBirthday(String birthday))
     this.birthday = birthday;
  }

  public void addAddress(Address addr){

    if (address == null) address = new Vector();
    address.add(addr);
  }
  public Vector getAddress(){ return address; }

  public void removeAddress(Address addr){
    if (address != null) address.remove(addr);
  }

  public void addWorkPhone(String phone){

    if (workPhone == null) workPhone = new Vector();
    workPhone.add(new Integer(phone));
  }
  public Vector getWorkPhone(){ return workPhone; }

  public void removeWorkPhone(String phone)(
    if (workPhone != null)
         workPhone.remove(new Integer(phone));
```

```
  }

  public void addHomePhone(String phone){

    if (homePhone = null) homePhone = new Vector();
    homePhone.add(new Integer(phone));
  }
  public Vector getHomePhone(){ return homePhone; }

  public void removeHomePhone(String phone){

    if (homePhone != null)
          homePhone.remove(new Integer(phone));
  }

  public String toString(){
    return "Person: " + surname;
  }
}
```

После определения классов Java, в экземпляры которых будет занесено содержимое адресной книжки, напишем программу, читающую адресную книжку и связывающую ее с объектами Java.

В листинге 1.9 приведен пример класса-обработчика NotebookHandler для адресной книжки, описанной в листинге 1.2. Методы класса NotebookHandler анализируют содержимое адресной книжки и помещают его в вектор, составленный из объектов класса Person, описанного в листинге 1.8.

Листинг 1.9. Класс-обработчик документа XML

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;
import java.util.*;
import java.io.*;

public class NotebookHandler extends DefaultHandler{
   static final String JAXP_SCHEMA_LANGUAGE =
     "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

   static final String W3C_XML_SCHEMA =
```

```
    "http://www.w3.org/2001/XMLSchema";

  Person person;
  Address address;
  static Vector pers = new Vector();
  boolean inBirthday, inStreet, inCity, inZip,
      inWorkPhone, inHomePhone;

 public void startElement(String uri, String name,
       String qname, Attributes attrs)

                        throws SAXException{

   if (qname.equals("name"))

      person = new Person(attrs.getValue("first") ,

         attrs.getValue("second"), attrs.getValue("surname"));

   else if (qname.equals("birthday"))

      inBirthday = true;

   else if (qname.equals("address"))

      address = new Address();

   else if (qname.equals("street"))

      inStreet = true;

   else if (qname.equals("city")){

      inCity = true;
      if (attrs != null) address.setType(attrs.getValue("type"));
   }else if (qname.equals("zip"))

      inZip = true;

   else if (qname.equals("work"))
```

```
        inWorkPhone = true;

    else if (qname.equals("home"))

        inHomePhone = true;
}

public void characters(char[] buf, int offset, int len)
        throws SAXException{

    String s = new String(buf, offset, len);

    if (inBirthday){

        person.setBirthday(s);
        inBirthday = false;

    }else if (inStreet){

        address.setStreet(s);
        inStreet = false;

    }else if (inCity){

        address.setCity(s);
        inCity = false;

    }else if (inZip){

        address.setZip(s);
        inZip = false;
    }else if (inWorkPhone){

        person.addWorkPhone(s);
        inWorkPhone = false;

    }else if (inHomePhone){

        person.addHomePhone(s);
```

```
        inHomePhone = false;
    }
}

public void endElement(String uri, String name,
        String qname) throws SAXException{

    if (qname.equals("address")){

        person.addAddress(address);
        address = null;

    (else if (qname.equals("person")){

        pers.add(person);
        person = null;
    }
}

public static void main(String[] args){

    if (args.length < 1){

        System.err.println("Usage: java Notebook ntb.xml");
        System.exit(1);
    }

    try{
      NotebookHandler handler = new NotebookHandler();

      SAXParserFactory fact = SAXParserFactory.newInstance ();
      fact.setNamespaceAware(true);
      fact.setValidating(true);

      SAXParser saxParser = fact.newSAXParser();

      saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);

      File f = new File(args[0]);
```

```
      saxParser.parse(f, handler);

      for (int к = 0; к < pers.size(); k++)
        System.out.println(((Person)pers.get(k)).getSurname());

      }catch(SAXNotRecognizedException x){

        System.err.println("Неизвестное свойство: " +
              JAXP_SCHEMA_LANGUAGE);
        System.exit(1);

    }catch(Exception ex){

      System.err.println(ex);
    }
  }

  public void warning(SAXParseException ex){

    System.err.println("Warning: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
         "  col = " + ex.getColumnNumber());
  }

  public void error(SAXParseException ex){

    System.err.println("Error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
         "  col = " + ex.getColumnNumber());
  }

  public void fatalError(SAXParseException ex){

    System.err.println("Fatal error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
         "  col = " + ex.getColumnNumber());
  }
}
```

После того как класс-обработчик написан, проанализировать документ очень легко. Стандартные действия приведены в методе `main` о программы листинга 1.9.

Поскольку реализация парсера сильно зависит от его программного окружения, SAX-парсер — объект класса SAXParser — создается не конструктором, а фабричным методом `newSAXParser()` .

Объект-фабрика, в свою очередь, создается методом `newInstance()`. После создания объекта-фабрики можно методом void setFeature(String name, boolean value) установить свойства парсеров, создаваемых этой фабрикой. Например, после

```
fact.setFeature("http://xml.org/sax/features/namespace-prefixes", true);
```

парсеры, создаваемые фабрикой fact, будут учитывать префиксы имен тегов и атрибутов.

Список таких свойств можно посмотреть на сайте проекта SAX **http://www.saxproject.org/**. Следует учитывать, что не все парсеры выполняют эти свойства.

Если к объекту-фабрике применить метод void setValidating(true), как это сделано в листинге 1.9, то она будет производить парсеры, проверяющие структуру документа. Если применить метод void setNamespaceAware(true), то объект-фабрика будет производить парсеры, учитывающие пространства имен.

После того как объект-парсер создан, остается только применить метод parse о, передав ему имя анализируемого файла и экземпляр класса-обработчика событий.

В классе `javax.xml.parsers`. SAXParser есть десяток методов `parse()`. Кроме метода `parse(File, DefaultHandler)`, использованного в листинге 1.9, есть еще методы, позволяющие извлечь документ из входного потока класса `InputStream`, объекта класса `InputSource`, адреса URI или из специально созданного источника класса inputsource.

Методом setProperty о можно задать различные свойства парсера. В листинге 1.9 этот метод использован для того, чтобы парсер проверял правильность документа с помощью схемы XSD. Эта возможность включена в JAXP, начиная с версии JAXP 1.2.

Если парсер выполняет проверки, то есть, применен метод `setvalidating(true)`, то имеет смысл сделать развернутые сообщения об ошибках. Это предусмотрено интерфейсом `ErrorHandler`, Он различает предупреждения, ошибки и фатальные ошибки и описывает три метода, которые автоматически выполняются при появлении ошибки соответствующего вида:

```
public void warning(SAXParserException ex);
public void error(SAXParserException ex);
public void fatalError(SAXParserException ex);
```

Класс `DefaultHandler` делает пустую реализацию этого интерфейса. При расширении этого класса можно сделать реализацию одного или всех методов интерфейса `ErrorHandler`. Пример такой реализации приведен в листинге 1.9. Класс SAXParserException хранит номер строки и столбца проверяемого документа, в котором замечена ошибка. Их можно получить методами `getLineNumber()` И `getColumnNumber()`, как сделано В листинге 1.9.

Связывание данных XML с объектами Java

В приведенном примере МЫ сами создали классы Address И Person, представляющие документ XML. Поскольку структура документа XML четко определена, можно разработать стандартные правила связывания данных XML с объектами Java и создать программные средства для их реализации.

Фирма Sun Microsystems разрабатывает пакет интерфейсов и классов JAXB, облегчающих связывание данных. На время написания книги проект JAXB еще не вышел из стадии разработки, но уже выставлен по адресу **http://java.sun.com/xml/jaxb/**. Для работы с пакетом JAXB анализируемый документ XML обязательно должен быть снабжен описанием DTD. В то время, когда я знакомился с JAXB, он не мог работать со схемой XSD. Видимо, это одна из причин задержки выпуска пакета.

В состав пакета JAXB входит компилятор xjc (XML-Java Compiler). Он просматривает описание DTD и строит по нему объекты Java в оперативной памяти, а также создает исходные файлы объектов Java. Например, после выполнения команды

```
$ xjc -roots notebook ntb.dtd -d sources
```

в которой ntb.dtd — файл листинга 1.3 — в каталоге sources (по умолчанию в текущем каталоге) будут созданы файлы Address.java, Name.java, Notebook.java, Person.java, PhoneList.java с описаниями объектов Java.

Флаг `-roots` показывает один или несколько корневых элементов, разделенных запятыми.

Кроме описания DTD компилятору xjc можно дать дополнительное описание схемы документа на специальном языке (binding language) — реализации XML. Это язык, похожий на язык XSD, но с меньшими возможностями. По-видимому, в окончательном выпуске пакета JAXB он будет расширен или заменен языком XSD. Если дополнительное описание записано в

файл ntb.xjs (XML-Java binding schema), то вызов компилятора будет выглядеть следующим образом:

```
$ xjc ntb.dtd ntb.xjs -d sources
```

Созданные компилятором xjc исходные файлы обычным образом с помощью компилятора javac компилируются в классы Java.

Получив объекты данных, можно перенести в них содержимое документа XML методом `unmarshal()`, который создает дерево объектов, или, наоборот, записать объекты Java в документы XML методом marshal о. Эти методы уже записаны в созданный компилятором xjc класс корневого элемента, В примере ЭТО — класс Notebook.

Объекты данных JDO

Задачу связывания данных естественно обобщить — связывать объекты Java не только с документами XML, но и с текстовыми файлами, реляционными или объектными базами данных, другими хранилищами данных.

Фирма Sun Microsystems опубликовала спецификацию JDO и разработала интерфейсы для работы с JDO. Их можно посмотреть по адресу **http://www.jcp.org/jsr/detail/12.jsp** или **http://accessl.sun.com/jdo/.**

Спецификация JDO рассматривает более широкую задачу связывания данных, полученных не только из документа XML, но и из любого источника данных, называемого *информационной системой предприятия* EIS (Enterprise Information System). Спецификация описывает два набора классов и интерфейсов:

- JDO SPI (JDO Service Provider Interface) — вспомогательные классы и интерфейсы, которые следует реализовать в сервере приложений для обращения к источнику данных, создания объектов, обеспечения их сохранности, выполнения транзакций, проверки прав доступа к объектам; эти классы и интерфейсы составляют пакет `javax.jdo.spi`;
- JDO API (JDO Application Programming Interface) — интерфейсы, предоставляемые пользователю для доступа к объектам, управления транзакциями, создания и удаления объектов; эти интерфейсы собраны в пакет `javax.jdo`.

Несмотря на то, что спецификация JDO уже опубликована, на время выпуска книги фирма Sun еще не реализовала весь проект. Это сделали другие фирмы.

Фирма Prism Technologies, **http://www.prismtechnologies.com/**, выпускает продукт OpenFusion JDO.

Фирма SolarMetric, **http://www.solarmetric.com/,** выпускает свою реализацию спецификации JDO под названием Kodo JDO. Ее можно встроить в серверы приложений WebLogic, WebSphere, JBoss.

Есть и другие разработки, их обзор можно посмотреть на сайте разработчиков JDO **http://www.jdocentral.com/.**

Некоторые фирмы, не дожидаясь выхода спецификации JDO, разработали свои реализации JDO, более или менее соответствующие спецификации. Наиболее известна свободно распространяемая разработка фирмы Exolab, названная Castor. Ее можно посмотреть по адресу **http://castor.exolab.org/**.

С помощью Castor можно предельно упростить связывание данных. Например, создание объекта Java из простого документа XML, если отвлечься от проверок и обработки исключительных ситуаций, выполняется одним действием:

```
Person person = (Person)Unmarshaller.unmarshal(
        Person.class, new FileReader("person.xml"));
```

Обратно, сохранение объекта Java в виде документа XML в самом простом случае выглядит так:

```
Marshaller.marshall(person, new FileWriter("person.xml"));
```

В более сложных случаях надо написать файл XML, аналогичный схеме XSD, с указаниями по связыванию данных (mapping file).

Анализ документов XML с помощью DOM API

Как видно из предыдущих разделов, SAX-парсер читает документ только один раз, отмечая появляющиеся по ходу чтения открывающие теги, содержимое элементов и закрывающие теги. Этого достаточно для связывания данных, но неудобно для редактирования документа.

Консорциум W3C разработал спецификации и набор интерфейсов DOM, которые можно посмотреть на сайте этого проекта **http://www.w3.org/DOM/.** Методами этих интерфейсов документ XML можно загрузить в оперативную память в виде дерева объектов. Это позволяет не только анализировать документ анализаторами, основанными на структуре дерева, но и менять дерево, добавляя или удаляя объекты из дерева. Кроме того, можно обращаться непосредственно к каждому объекту в дереве и не только читать, но и изменять информацию, хранящуюся в нем. Однако, все это требует большого объема оперативной памяти для загрузки большого дерева.

Фирма Sun Microsystems реализовала интерфейсы DOM в пакетах `javax.xml.parsers` и `org.w3c.dom`, входящих в состав пакета JAXP. Воспользоваться этой реализацией очень легко:

```
DocumentBuilderFactory fact =
        DocumentBuilderFactory.newInstance();
```

```
DocumentBuilder builder = fact.newDocumentBuilder();

Document doc = builder.parse("ntb.xml");
```

Метод parse () строит дерево объектов и возвращает ссылку на него в виде объекта типа Document. В классе DocumentBuilder есть несколько методов `parse`n, позволяющих загрузить файл с адреса URL, из входного потока, как объект класса File или из источника класса `InputSource`.

Интерфейс Document, расширяющий интерфейс Node, описывает дерево объектов документа в целом. Интерфейс Node — основной интерфейс в описании структуры дерева — описывает узел дерева. У него есть еще один наследник — интерфейс Element, описывающий лист дерева, соответствующий элементу документа XML. Как видно из структуры наследования этих интерфейсов, и само дерево, и его лист считаются узлами дерева. Каждый атрибут элемента дерева описывается интерфейсом Attr. Еще несколько интерфейсов — CDATASection, Comment, Text, Entity, `EntityReference`, `ProcessingInstruction`, Notation — описывают разные типы элементов XML.

На рис. 1.2 показано начало дерева объектов, построенного по документу, приведенному в листинге 1.2. Обратите внимание на то, что текст, находящийся в теле элемента, хранится в отдельном узле дерева — потомке узла элемента. Для каждого атрибута открывающего тега тоже создается отдельный узел.

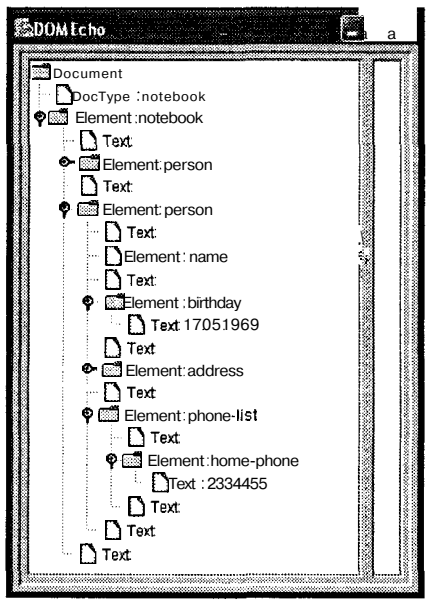


Рис. 1.2. Дерево объектов документа XML

Интерфейс *Node*

Интерфейс Node описывает тип узла одной из следующих констант:

- ATTRIBUTE_NODE — узел типа Attr, содержит атрибут элемента;
- CDATA_SECTION_NODE — узел типа CDADASection, содержит данные типа CDATA;
- COMMENT_NODE — узел типа comment, содержит комментарий;
- DOCUMENT_FRAGMENT_NODE — В узле типа DocumentFragment находится фрагмент документа;
- DOCUMENT_NODE — корневой узел типа Document;
- DOCUMENT_TYPE_NODE — узел типа Document;
- ELEMENT_NODE — узел является листом дерева типа Element;
- ENTITY_NODE — в узле типа Entity хранится сущность ENTITY;
- ENTITY_REFERENCE_NODE — В узле ТИПа EntityReference хранится ССЫЛКа на сущность;
- NOTATION_NODE — В узле хранится нотация ТИПа Notation;
- PROCESSING_INSTRUCTION_NODE — узел ТИПа ProcessingInstruction, СО-держит инструкцию по обработке;
- TEXT_NODE — в узле типа Text хранится текст.

Методы интерфейса Node описывают действия с узлом дерева. Узнать тип узла можно методом:

```
public short getNodeType();
```

Имя узла возвращает метод:

```
public String getNodeName();
```

Значение, хранящееся в узле, можно получить методом:

```
public String getNodeValue();
```

Проверить, есть ли атрибуты у элемента XML, хранящегося в узле в виде объекта типа NamedNodeMap, если это узел типа Element, можно методом:

```
public boolean hasAttributes();
```

Атрибуты возвращает метод:

```
public NamedNodeMap getAttributes();
```

Метод возвращает null, если у элемента нет атрибутов.

Следующий метод проверяет, есть ли у данного узла узлы-потомки:

```
public boolean hasChildNodes();
```

Если они есть, то можно получить их список в виде объекта типа `NodeList` методом:

```
public NodeList getChildNodes();
```

Первый и последний узлы в этом списке можно получить методами:

```
public Node getFirstChild();
public Node getLastChild();
```

Родительский узел можно получить методом:

```
public Node getParentNode();
```

а соседние узлы с тем же предком, что и данный узел — методами:

```
public Node getPreviousSibling();
public Node getNextSibling();
```

Можно получить и ссылку на весь документ методом:

```
public Document getOwnerDocument();
```

Следующие методы позволяют изменить дерево объектов.

Добавить новый узел-потомок `newChild` можно методом:

```
public Node appendChild(Node newChild);
```

Вставить новый узел-потомок `newChild` перед существующим потомком `refchild` можно методом:

```
public Node insertBefore(Node newChild, Node refChild);
```

Заменить один узел-потомок `oldChild` новым узлом `newchild` можно методом:

```
public Node replaceChild(Node newChild, Node oldChild);
```

Наконец, удалить узел-потомок можно методом:

```
public Node removeChild(Node child);
```

Интерфейс *Document*

Интерфейс Document добавляет к методам своего предка Node методы работы с документом в целом. Метод

```
public DocumentType getDocType();
```

возвращает общие сведения о документе в виде объекта типа DocumentType. Методы `getName()`, `getEntitied()`, `getNotations()` и другие методы интерфейса DocumentType возвращают конкретные сведения о документе.

Метод

```
public Element getDocumentElement();
```

возвращает корневой элемент дерева объектов, а методы

```
public NodeList getElementsByTagName(String name);
public NodeList getElementsByTagNameNS(String uri, String qname);
public Element getElementById(String id);
```

возвращают все элементы с указанным именем tag без префикса или с префиксом, а также элемент, определяемый значением атрибута с именем ID.

Несколько методов позволяют изменить структуру и содержимое дерева объектов.

Создать новый пустой элемент по его имени или по имени с префиксом можно методами

```
public Element createElement(String name);
public Element createElementNS(String uri, String name);
```

Создать узел типа CDATA_SECTION_NODE можно методом

```
public CDATASection createCDATASection(String name);
```

Создать узел типа ENTITY_REFERENCE_NODE можно методом

```
public EntityReference createEntityReference(String name);
```

Создать узел типа PROCESSING_INSTRUCTION_NODE можно методом

```
public ProcessingInstruction createProcessingInstruction(String name);
```

Создать узел типа TEXT_NODE можно методом

```
public TextNode createTextNode(String name);
```

Создать узел-атрибут с именем name можно методом

```
public Attr createAttribute(String name);
public Attr createAttributeNS(String uri, String name);
```

Узел-комментарий создается методом

```
public Comment createComment(String comment);
```

Наконец, можно создать пустой документ — фрагмент данного документа с целью его дальнейшего заполнения

```
public DocumentFragment createDocumentFragment();
```

Вставить созданный узел, а значит, и все его поддерево, в дерево документа, можно методом

```
public Node importNode(Node importedNode, boolean deep);
```

Этим методом можно соединить два дерева объектов. Если второй аргумент равен true, то рекурсивно вставляется все поддерево.

Интерфейс *Element*

Интерфейс Element добавляет к методам своего предка Node методы работы с атрибутами открывающего тега элемента XML и методы, позволяющие обратиться к вложенным элементам. Только один метод

```
public String getTagName();
```

дает сведения о самом элементе, а именно, имя элемента.

Прежде чем получить значение атрибута с именем name надо проверить его наличие методами

```
public boolean hasAttribute(String name);
public boolean hasAttributeNS(String uri, String name);
```

Второй из этих методов учитывает пространство имен с именем uri, записанным в виде строки URI; имя name должно быть полным, с префиксом.

Получить атрибут в виде объекта типа Attr или его значение в виде строки по имени name с учетом префикса или без него можно методами

```
public Attr getAttributeNode(String name);
public Attr getAttributeNodeNS(String uri, String name);
public String getAttribute(String name);
public String getAttributeNS(String uri, String name);
```

Удалить атрибут можно методами

```
public Attr removeAttributeNode(Attr name);
public void removeAttribute(String name);
public void removeAttributeNS(String uri, String name);
```

Установить значение атрибута можно методами

```
public void setAttribute(String name, String value);
public void setAttributeNS(String uri, String name, String value);
```

Добавить атрибут в качестве потомка можно методами

```
public Attr setAttributeNode(String name);
public Attr setAttributeNodeNS(Attr name);
```

Два метода позволяют получить список узлов-потомков:

```
public NodeList getElrmentsByTagName(String name);
public NodeList getElrmentsByTagNameNS(String uri, String name);
```

Итак, методы перечисленных интерфейсов позволяют перемещаться по дереву, менять структуру дерева, просматривать информацию, хранящуюся в узлах и листьях дерева и изменять ее.

Приведем пример работы с деревом объектов, построенным по документу XML. Добавим в адресную книжку листинга 1.2 новый рабочий или домашний телефон Сидоровой. Это действие записано в листинге 1.10.

Листинг 1.10. Анализ адресной книжки с помощью DOM API

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.xml.sax.*;

class ErrHand implements ErrorHandler{

  public void warning(SAXParseException ex){

    System.err.println("Warning: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
         "  col = " + ex.getColumnNumber());
  }

  public void error(SAXParseException ex){

    System.err.println("Error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
         "  col = " + ex.getColumnNumber());
  }

  public void fatalError(SAXParseException ex){

    System.err.println("Fatal error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
         "  col = " + ex.getColumnNumber());
  }
}

public class TreeProcessDOM{

  static final String JAXP_SCHEMA_LANGUAGE =
      "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

  static final String W3C_XML_SCHEMA =
```

```
    "http://www.w3.org/2001/XMLSchema";

public static void main(String[] args) throws Exception}

  if (args.length != 3){

    System.err.println("Usage: java TreeProcessDOM " +
          "<file-name>.xml {worklhome} <phone>");
    System.exit(-1);
  }

  DocumentBuilderFactory fact =

     DocumentBuilderFactory.newInstance() ;

  fact.setNamespaceAware(true);

  fact.setValidating(true);

  try{
     fact.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
  }catch(IllegalArgumentException x){

     System.err.println("Неизвестное свойство: " +
           JAXP_SCHEMA_LANGUAGE);
     System.exit(-1);
  }

  DocumentBuilder builder = fact.newDocumentBuilder();

  builder.setErrorHandler(new ErrHand());

  Document doc = builder.parse(args[0]);

  NodeList list = doc.getElementsByTagName("notebook");

  int n = list.getLength();

  if (n == 0) {
    System.err.println("Документ пуст.");
```

```
    System.exit(-1);
  }

 Node thisNode = null;

 for (int к = 0; к < n; k++){

   thisNode = list.item(k);
   String elemName = null;

   if (thisNode.getFirstChild() instanceof Element){
      elemName = (thisNode.getFirstChild()).getNodeName();

   if (elemName.equals("name")){

      if (!thisNode.hasAttributes()){

        System.err.println("Атрибуты отсутствуют " + elemName);
        System.exit(1);
      }

      NamedNodeMap attrs = thisNode.getAttributes();

      Node attr = attrs.getNamedItem("surname");

      if (attr instanceof Attr)
         if (((Attr)attr).getValue().equals("Сидорова")) break;
   }
  }
}

NodeList topics = ((Element)thisNode)
                                                    ;
          .getElementsByTagName("phone-list")

Node newNode;

if (args[1].equals("work"))
```

```
            newNode = doc.createElement("work-phone");
        else newNode = doc.createElement("home-phone");

        Text textNode = doc.createTextNode(args[2]);

        newNode.appendChild(textNode);

        thisNode.appendChild(newNode);
      }
}
```

Дерево объектов можно вывести на экран дисплея, например, как дерево JTree — компонент графической библиотеки Swing [10]. Именно так сделано на рис. 1.2. Для вывода применена программа DomEcho из электронного учебника "Web Services Tutorial". Исходный текст программы слишком велик, чтобы приводить его здесь, но его можно посмотреть по адресу **http://java.sun.com/webservices/tutorial.html**. В состав парсера Xerces в качестве примера анализа документа в раздел `samples/ui/` входит программа `TreeView`, которая тоже показывает дерево объектов в виде дерева JTree библиотеки Swing.

Другие DOM-парсеры

Модель дерева объектов D0M была первоначально разработана группой OMG (Object Management Grou) в рамках языка IDL (Interface Definition Language, язык определения интерфейса) без учета особенностей Java. Только потом она была переведена на Java консорциумом W3C в виде интерфейсов и классов, составивших пакет `org.w3c.dom`. Этим объясняется, в частности, широкое применение в DOM API интерфейсов и фабричных методов вместо классов и конструкторов.

Это неудобство привело к появлению других разработок.

Участники общественного проекта JDOM не стали реализовать модель D0M, а разработали свою модель дерева объектов, получившую название JDOM [6]. Они выпускают одноименный свободно распространяемый программный продукт, с которым можно ознакомиться на сайте проекта **http://www.jdom.org/**. Этот продукт никак не может выйти из стадии разработки, но уже широко используется для обработки документов XML средствами Java.

Участники другого общественного проекта dom4j приняли модель W3C D0M, но упростили и упорядочили DOM API. С их одноименным продуктом dom4j можно ознакомиться на сайте **http://www.dom4j.org/.**

Фирма TME (The Mind Electric), **http://www.themindelectric.com/**, выпускающая известные программные продукты GLUE и GAIA, поставляет в составе GLUE или отдельно набор инструментальных средств EXML (Electric XML) и его расширение EXML+ (Electric XML+). Этот продукт полностью реализует W3C DOM, весьма компактен, экономичен и быстр.

Преобразование дерева объектов в XML

Итак, дерево объектов D0M построено надлежащим образом. Теперь надо его преобразовать в документ XML, страничку HTML, документ PDF или объект другого типа. Средства для выполнения такого преобразования составляют третью часть набора JAXP — пакеты `javax.xml.transform`, `javax.xml.transform.dom`, `javax.xml.transform.sax`, `javax.xml.transform.stream`, которые представляют собой реализацию языка описания таблиц стилей для преобразований XSLT (XML Stylesheet Language for Transformations) средствами Java.

Язык XSLT разработан консорциумом W3 как одна из трех частей, составляющих язык записи таблиц стилей XSL (XML Stylesheet Language). Все материалы по XSL можно посмотреть на сайте проекта **http://www.w3.org/Style/XSL/.**

Интерфейсы и классы, входящие в пакеты `javax.xml.transform.*`, управляют процессором XSLT, в качестве которого выбран процессор Xalan, разработанный в рамках проекта Apache Software Foundation, **http://xml.apache.org/xalan-j/**.

Исходный объект преобразования должен иметь тип `Source`. Интерфейс Source определяет всего два метода доступа к идентификатору объекта:

```
public String getSystemId();
public void setSystemId(String id);
```

У интерфейса source есть три реализации. Класс DOMSource подготавливает к преобразованию дерево объектов D0M, а классы SAXsource и streamSource подготавливают SAX-объект и простой поток данных. В конструкторы этих классов заносится ссылка на исходный объект — для конструктора класса DOMSource это узел дерева, для конструктора класса SAXSource — имя файла, для конструктора класса `StreamSource` — входной поток. Методы этих классов позволяют задать дополнительные свойства исходных объектов преобразования.

Результат преобразования описывается интерфейсом Result. Он тоже определяет точно такие же методы доступа к идентификатору объекта-результата, как и интерфейс `Source`. У него тоже есть три реализации — классы DOMResult, SAXResult И `StreamResult`. В конструкторы этих классов заносит-

ся ссылка на выходной объект. В первом случае это узел дерева, во втором — объект типа ContentHandler, в третьем — файл, в который будет занесен результат преобразования, или выходной поток.

Само преобразование выполняется объектом класса Transformer. Вот стандартная схема преобразования дерева объектов DOM в документ XML, записываемый в файл.

```
TransformerFactory transFactory =
        TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
DOMSource source = new DOMSource(document);
File newXMLFile = new File("ntbl.xml");
FileOutputStream fos = new FileOutputStream(newXMLFile);
StreamResult result = new StreamResult(fos);
transformer.transform(source, result);
```

Вначале методом newlnstance o создается экземпляр transFactory фабрики объектов-преобразователей. Методом

```
public void setAttrbute(String name, String value);
```

класса TransformerFactory можно установить некоторые атрибуты этого экземпляра. Имена и значения атрибутов зависят от реализации фабрики.

С помощью фабрики преобразователей создается объект-преобразователь класса Transformer. При создании этого объекта в него можно занести объект, содержащий правила преобразования, например, таблицу стилей XSL.

В созданный объект класса Transformer методом

```
public void setParameter(String name, String value);
```

можно занести параметры преобразования, а методами

```
public void setOutputProperties(Properties out);
public void setOutputProperty(String name, String value);
```

можно определить свойства преобразованного объекта. Имена свойств name задаются константами, которые собраны в специально определенный класс outputKeys, содержащий только эти константы. Вот их список:

- CDATA_SECTION_ELEMENTS — список имен секций CDATA через пробел.
- DOCTYPE_PUBLIC — открытый идентификатор PUBLIC преобразованного документа.

- DOCTYPE_SYSTEM — системный идентификатор SYSTEM преобразованного документа.
- ENCODING — кодировка символов преобразованного документа, значение атрибута encoding объявления XML.
- INDENT — делать ли отступы в тексте преобразованного документа. Значения этого свойства "yes" или "no".
- MEDIA_TYPE — MIME-тип содержимого преобразованного документа.
- METHOD — метод вывода, одно из значений "xml", "html" или "text".
- OMIT_XML_DECLARATION — не включать объявление XML. Значения "yes" или "no".
- STANDALONE — отдельный или вложенный документ, значение атрибута standalone объявления XML. Значения "yes" или "no".
- VERSION — номер версии XML для атрибута version объявления XML.

Например, можно задать кодировку символов преобразованного документа следующим методом:

```
transformer.setOutputProperty(OutputKeys.ENCODING, "Windows-1251");
```

Затем в приведенном примере по дереву объектов document типа Node создается объект класса DOMSource — упаковка дерева объектов для последующего преобразования. По типу аргумента конструктора видно, что можно преобразовать не все дерево, а какое-либо его поддерево, записав в конструкторе класса DOMSource корневой узел поддерева.

Наконец, определяется результирующий объект result, связанный с файлом newCourses.xml и выполняется преобразование методом transform().

Более сложные преобразования выполняются с помощью таблицы стилей XSL.

Таблицы стилей XSL

В документах HTML часто применяются таблицы стилей CSS, задающие общие правила оформления документов HTML: цвет, шрифт, заголовки. Выполнение этих правил придает документам единый стиль оформления.

Для документов XML, в которых вообще не определяются правила визуализации, идея применить таблицы стилей оказалась весьма плодотворной. Таблицы стилей для документов XML записываются на специально сделанной реализации языка XML, названной XSL (XML Stylesheet Language). Все теги документов XSL относятся к пространству имен **http://www.w3.org/1999/XSL/Transform**. Обычно они записываются с пре-

фиксом xsl. Если принят этот префикс, то корневой элемент таблицы стилей XSL будет называться <xsl:stylesheet>.

Простейшая таблица стилей выглядит так, как записано в листинге 1.11.

Листинг 1.11. Простейшая таблица стилей XSL

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet  version="1.0"

      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

   <xsl:output method="text" encoding="CP866"/>

</xsl:stylesheet>
```

Здесь только определяется префикс пространства имен xsl и правила вывода, а именно, выводится плоский текст, на что показывает значение text (другие значения — html и xml), в кодировке CP866. Такая кодировка выбрана для вывода кириллицы на консоль MS Windows. Объект класса Transformer, руководствуясь таблицей стилей листинга 1.11, просто выводит тела элементов так, как они записаны в документе XML, то есть просто удаляет теги вместе с атрибутами, оставляя их содержимое.

Эту таблицу стилей записываем в файл, например, simple.xsl. Ссылку на таблицу стилей можно поместить в документ XML как одну из инструкций по обработке:

```
<?xml version="1.0" encoding="Windows-1251"?>
<?xml-stylesheet type="text/xsl" href="simple.xsl"?>
<notebook>
<!— Продолжение адресной книжки —>
```

После этого XML-парсер, если он, кроме того, является XSLT-процессором, выполнит преобразование, заданное в файле simple.xsl.

Другой путь — использовать таблицу стилей при создании объекта-преобразователя, например, так, как записано в листинге 1.12.

Листинг 1.12. Консольная программа преобразования документа XML

```
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

public class SimpleTransform{
```

```
  public static void main(String[] args)
                    throws TransformerException{

    if (args.length != 2){

       System.out.println("Usage: " +
           "java SimpleTransform xmlFileName xsltFileName");
       System.exit(1);
    }

    File xmlFile = new File(args[0]);
    File xsltFile = new File(args[1]);

    Source xmlSource = new StreamSource(xmlFile) ;
    Source xsltSource = new StreamSource(xsltFile);
    Result result = new StreamResult(System.out);

    TransformerFactory transFact =
              TransformerFactory.newInstance();

    Transformer trans =
              transFact.newTransformer(xsltSource);

    trans.transform(xmlSource, result);
  }
}
```

После компиляции набираем в командной строке

```
java SimpleTransform ntb.xml simple.xsl
```

и получаем на экране дисплея содержимое тегов документа ntb.xml.

В более сложных случаях просто изменяем таблицу стилей. Например, если мы хотим получить содержимое документа на консоли, то таблицу стилей для объекта-преобразователя класса Transformer надо записать так, как показано в листинге 1.13.

Листинг 1.13. Таблица стилей для показа адресной книжки

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet  version="1.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="CP866" />

  <xsl:template match="person">

      <xsl:apply-templates />

  </xsl:template>

  <xsl:template match="name">

     <xsl:value-of select="@first" />   <xsl:text> </xsl:text>
     <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
     <xsl:value-of select="@surname" />

  </xsl:template>

  <xsl:template match="address">

     <xsl:value-of select="street" /> <xsl:text> </xsl:text>
     <xsl:value-of select="city" />   <xsl:text> </xsl:text>
     <xsl:value-of select="zip" />

  </xsl:template>

  <xsl:template match="phone-list">

     <xsl:value-of select="work-phone" /> <xsl:text>&#xA;</xsl:text>
     <xsl:value-of select="home-phone" /> <xsl:text>&#xA;</xsl:text>

  </xsl:template>

</xsl:stylesheet>
```

Мы не будем в этой книге заниматься языком XSL — одно описание языка будет толще всей этой книги. На русский язык переведена "библия" XSLT [5]. Ее автор Майкл Кэй (Michael H. Kay) создал

и свободно распространяет популярный XSLT-процессор Saxon, **http://users.iclway.co.uk/mhkay/saxon/index.html.** Хорошее изложение XSLT сделал Алексей Валиков [2], его книгой вы можете воспользоваться для дальнейшего изучения XSLT. В этой книге вы найдете описание XSLT-процессоров и их сравнение.

Надо заметить, что язык XSL, в котором выделяются правила преобразования XSLT (XSL Transform), правила форматирования XSL FO (XSL Formatting Objects) и язык описания путей к различным частям документов XPath (XML Path Language), бурно развивается, и сделать полное его описание сейчас просто невозможно.

Преобразование документа XML в HTML

С помощью языка XSL и методов класса Transformer можно легко преобразовать документ XML в документ HTML. Достаточно написать соответствующую таблицу стилей. В листинге 1.14 показано, как это можно сделать для адресной книжки листинга 1.2.

Листинг 1.14. Таблица стилей для преобразования XML в HTML

```
<?xml version="1.0" encoding="Windows-1251"?>

<xsl:stylesheet  version="1.0"

    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" encoding="Windows-1251"/>

  <xsl:template match="/">
    <html><head><title>Адресная книжка</title></head>
    <body><h2>Фамилии, адреса и телефоны</h2>
       <xsl:apply-templates />
    </bodyx/html>

  </xsl:template>

  <xsl:template match="name">

    <p />
```

```
    <xsl:value-of select="@first" /> <xsl:text> </xsl:text>
    <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
    <xsl:value-of select="@surname" /> <br />

  </xsl:template>

  <xsl:template match="address">

    <br />
    <xsl:value-of select="street" /> <xsl:text> </xsl:text>
    <xsl:value-of select="city" /> <xsl:text> </xsl:text>
    <xsl:value-of select="zip" /> <br />

  </xsl:template>

  <xsl:template match="phone-list">

    Рабочий: <xsl:value-of select="work-phone" /> <br />
    Домашний: <xsl:value-of select="home-phone" /> <br />

  </xsl:template>

</xsl:stylesheet>
```

Эту таблицу стилей можно записать в файл, например, ntb.xsl, и сослаться на него в документе XML, описывающем адресную книжку:

```
<?xml version="1.0" encoding="Windows-1251"?>
<?xml-stylesheet type="text/xsl" href="ntb.xsl"?>
<notebook>
<!- Содержание адресной книжки ->
```

После этого любой браузер, "понимающий" XML и XSLT, например, Mozilla или Internet Explorer, покажет содержимое адресной книжки как предписано листингом 1.14.

Если набрать в командной строке

```
$ java SimpleTransform ntb.xml ntb.xsl > ntb.html
```

то в текущем каталоге будет записан преобразованный HTML-файл.

ГЛАВА 2

Архитектура Web Services

Широкое распространение Интернета началось после того, как была создана "Всемирная паутина" WWW (World Wide Web, "Всемирный словарь Вебстера", если считать слово Web сокращением слова Webster). Она сделала получение информации из Интернета легким и приятным занятием. На каждой машине есть стандартный браузер: Mozilla, Opera, Internet Explorer, Netscape Communicator — выбирай, что нравится. Человек запрашивает Web-страницу с любого сервера, включенного в WWW, нимало не интересуясь, на какой платформе работает Web-сервер, какой операционной системой он управляется, в каком порядке идут байты в его машинных словах. Да и название и версия самого Web-сервера вовсе не интересуют клиента. Ему достаточно набрать адрес URL, что-нибудь вроде **http://www.some.com/coolpage.html**, или просто щелкнуть кнопкой мыши на гиперссылке. Браузер сам оформляет запрос по правилам протокола HTTP, устанавливает связь с сервером и передает ему запрос по сети. На машине-сервере запрос принимает Web-сервер, обычно прослушивающий порт номер 80. Сервер выполняет запрос и передает ответ в стандартном виде, по правилам протокола HTTP, не зависящим от платформы клиента и от браузера, которым пользуется клиент.

Успех WWW основан на едином языке HTML, понятном любому браузеру, и на простом протоколе HTTP, легко реализуемом любым сервером. Успех языка HTML, в свою очередь, основан на том, что запись на нем ведется обычным байтовым ASCII-текстом, который одинаково понимается всеми машинами. Его легко передать по сети без искажений и интерпретировать стандартным образом в браузерах.

Серверы, включенные в систему WWW, готовы предоставить не только статичную информацию, записываемую, в конце концов, HTML-файлами,

звуковыми файлами, файлами с изображениями и прочими файлами. Они могут предоставить свои программные средства — отдельные процедуры и целые объекты — для выполнения их удаленными клиентами, находящимися, может быть, по другую сторону Земли.

Процедуры и объекты, которые сервер предоставляет всем нуждающимся в них клиентам, называются *распределенными* (distributed) процедурами и объектами. С точки зрения клиента это *удаленные* (remote) процедуры и объекты. Технологии обращения к удаленным процедурам и объектам существуют давно. Более того, есть несколько различных технологий. В голову сразу приходит множество аббревиатур: RPC, RMI, DCOM, COM+, CORBA, .NET. Однако их реализация ограничивается выбранной технологией: сокетами BSD, технологией Java или Microsoft Windows. Только технология CORBA претендует на всеобщность, но это делает ее чрезвычайно громоздкой и запутанной, потому что CORBA стремится использовать возможности всех или хотя бы наиболее распространенных платформ.

Использование распределенных процедур началось на рубеже 80-х годов с создания в фирме Xerox механизма вызова удаленных процедур RPC (Remote Procedure Call). Суть RPC заключается в том, что на машину клиента вместо вызываемой процедуры пересылается небольшой программный код, называемый *заглушкой* (stub). Заглушка внешне выглядит как вызываемая процедура, но ее код не выполняет процедуру, а преобразует (marshall) ее аргументы к виду, удобному для пересылки. Такое преобразование называется *сборкой* (marshalling). После сборки заглушка устанавливает связь с сервером по понятному для него протоколу и пересылает собранные аргументы на сервер. Клиент, вызвавший удаленную процедуру, взаимодействует не с ней, а с заглушкой, как с обычной локальной процедурой, выполняющейся на его машине. Сервер, получив собранные аргументы процедуры, разбирает (unmarshall) их, вызывает процедуру, передает ей параметры, дожидается результата, собирает (marshall) его и пересылает заглушке. Заглушка снова разбирает (unmarshall) результат и передает его клиенту как обычная локальная процедура.

Эффективно реализовать механизм RPC совсем не просто. Значительная часть толстой книги [10] посвящена методам вызова распределенных процедур и методов распределенных объектов. Различные реализации RPC и других способов вызова удаленных процедур и объектов стремились ускорить работу удаленных процедур и придумывали изощренные и наиболее быстрые алгоритмы и форматы сборки аргументов. В результате эти форматы могли применяться только в данной реализации RPC.

Тем временем компьютерные сети развивались и улучшались, на предприятиях стали широко применяться распределенные приложения, и понадобилось эффективное взаимодействие распределенных объектов.

Напомним [10], что *распределенным* называется приложение, отдельные компоненты которого работают на разных компьютерах и используют разные сетевые средства, но взаимодействуют так, что приложение выглядит как единое целое, как будто все его компоненты расположены на одной машине.

Простейшая архитектура распределенного приложения, называемая *архитектурой клиент-сервер,* предполагает, что приложение состоит из двух частей: серверной части, оказывающей услуги, и клиентской части, пользующейся услугой. В Web-приложении, построенном по архитектуре клиент-сервер, услуги оказывает Web-сервер, а клиентом служит браузер, или текстовый редактор, подключенный к Интернету, или другое клиентское приложение, связанное с Web-сервером как показано на рис. 2.1.

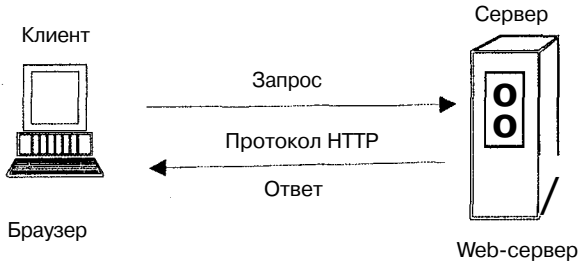


Рис. 2.1. Архитектура клиент-сервер в WWW

В архитектуре клиент-сервер очень важно правильно разделить работу между клиентом и сервером. Можно сделать клиента "тонким", только отображающим результаты запроса. Это удобно для организации клиента. Его можно разместить на простейшем дешевом компьютере. Ему не нужно сложного программного обеспечения, в большинстве случаев достаточно обыкновенного браузера. Но тогда сервер становится "толстым", на него ложится большая нагрузка, ему приходится выполнять все запросы к источнику данных и всю обработку этих данных. Сервер не справляется с нагрузкой, его трудно модернизировать, обновлять, неудобно наращивать его мощность.

Можно, наоборот, сделать клиента "толстым", выполняющим всю обработку результатов запроса, а сервер "тонким", только рассылающим необработанные данные клиентам. Так организован классический обмен информацией по WWW между "толстыми" браузерами и "тонкими" Web-серверами. В этом случае для клиента требуется мощный дорогостоящий компьютер. В случае смены алгоритма обработки данных или обнаружения ошибок придется обновлять программное обеспечение на всех клиентских машинах.

Для того чтобы избавиться от этих недостатков архитектуры клиент-сервер, программы, которые обрабатывают данные, выделяют в отдельный, *кроме-*

жуточный (middleware) слой программного обеспечения. Этот слой может работать на той же машине, что и серверный слой, работать на другой машине или даже на нескольких машинах. Распределенное приложение становится трехслойным. В технологии Java промежуточный слой обычно реализован *сервером приложений* (application server). Выпускается много промышленных серверов приложений: BEA WebLogic, JBoss, IBM WebSphere, Sun ONE Application Server, Oracle9i Application Server, Sybase EAServer, IONA Orbix E2A, Borland Enterprise Server и другие. В технологии Microsoft Corporation это сервер IIS (Internet Information Services).

Очень часто промежуточный слой делится на две части. Одна часть взаимодействует с клиентом: принимает запросы, анализирует их, формирует ответ на запрос и отправляет его клиенту. В технологии Java это *Web-слой* — сервлеты и страницы JSP. Другая часть промежуточного слоя получает данные от серверного слоя и обрабатывает их, руководствуясь указаниями, полученными от Web-слоя. В технологии Java это *EJB-слой* — компоненты EJB. Такая схема показана на рис. 2.2.

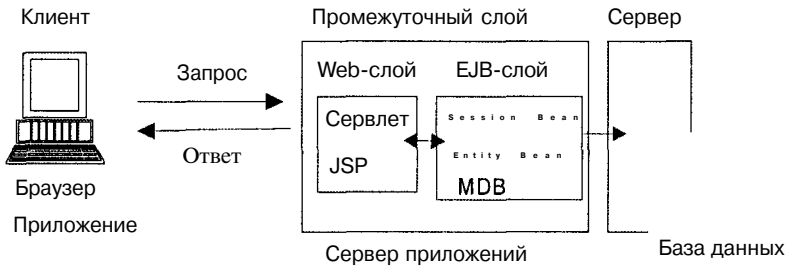


Рис. 2.2. Многослойная архитектура

Количество слоев можно увеличивать, но важнее установить надежную и быструю связь между всеми компонентами распределенного приложения. Казалось бы, надо воспользоваться старыми проверенными средствами: RPC, RMI, DCOM, CORBA, но тогда компоненты оказываются намертво привязанными к выбранной технологии, и мы получаем *тесно связанное* (tightly coupled) распределенное приложение. Это хорошо, если все компоненты созданы для одной платформы, но такая ситуация редко встречается в Интернете. Гораздо чаще компоненты распределенного приложения работают на разных платформах: клиентская часть разработана для MS Windows, Linux или Apple Macintosh, серверная часть — для Solaris, Linux, Free BSD, AIX, HP UX, для других UNIX или для мейнфреймов. Более того, сейчас наблюдается явная тенденция создавать приложения, независимые от какой бы то ни было платформы. Вся технология Java создана в русле этой тенденции.

Распределенное приложение, компоненты которого могут работать на разных платформах и заменять друг друга, называется *слабо связанным* (loosely

coupled) приложением. Например, браузер и Web-сервер слабо связаны друг с другом. Они не только могут работать на разных платформах, но и обмениваются самой разной информацией с разными MIME-типами. Кроме того, связь между ними очень коротка, она состоит только из запроса и ответа. Фактически, общее у браузера и Web-сервера только то, что они работают по одному протоколу HTTP и то, что они должны быть на связи одновременно. Для слабо связанных приложений необязательно даже последнее условие. Они могут работать асинхронно, их компоненты могут выходить на связь в удобное для них время. Так работает электронная почта, которую можно считать классическим примером слабо связанного приложения.

Создатели Web Services решили, что Web-услуги будут предоставляться в рамках слабо связанных приложений. Они решили опереться только на общие средства, имеющиеся на всех платформах. Таким "наименьшим общим знаменателем" различных вычислительных платформ, использующих WWW, оказался язык HTML и протокол HTTP. Языка HTML явно недостаточно для описания вызовов распределенных процедур и методов распределенных объектов и тут на выручку приходит язык XML. Действительно, он не зависит от платформы. Средства обработки документов XML, как мы убедились в предыдущей главе, имеются на всех платформах. Документы XML легко передаются по сети любым прикладным протоколом, поскольку это просто байтовый ASCII-текст.

Таким образом, Web Services — это услуги, предоставляемые по WWW с использованием той или иной реализации языка XML, протокола HTTP или других Web-протоколов. Есть множество определений этого понятия. Каждая фирма, разрабатывающая Web Services, дает свое определение Web-услуг. Мне показалось наиболее точным определение, которое дал консорциум W3C в документе "Требования к архитектуре Web Services", http://www.w3.org/TR/wsa-reqs. Его стоит привести в оригинале:

"A Web service is a software application identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via internet-based protocols".

Вот дословный перевод:

"Web Service — это приложение, идентифицируемое строкой URI, интерфейсы и связи которого могут определяться, описываться и отыскиваться документами XML. Оно взаимодействует напрямую с другими приложениями по межсетевым протоколам с помощью сообщений, записанных на языке XML".

По этому определению Web Service — это не всякая услуга, оказываемая посредством Web-технологии. Пересылка файлов, даже XML-документов, к ней не относится. Обычными Web-услугами пользуются независимые приложения: браузеры, связанные с Web-сервером только на время оказания

услуги. В отличие от них услуги, названные "Web Services", предоставляются, как правило, в рамках распределенного приложения. Можно сказать, что обычные Web-услуги предоставляются клиенту-человеку, а Web Services — это услуги, оказываемые клиенту-программе. Как правило, Web Service применяется как компонент распределенной информационной системы, разбросанной по компьютерам с разной архитектурой и разными средствами сетевой связи.

При описании всякой новой технологии возникает вопрос о переводе английских терминов. Поскольку русский перевод термина "Web Service" еще не устоялся, будем в этой книге записывать его и без перевода ("Web Service"), и словом "Web-служба", а каждую отдельную услугу, предоставляемую данной Web-службой, будем называть "Web-услугой".

Итак, для удобства предоставления Web-услуг необходим эффективный протокол пересылки информации, основанный на XML. Все началось с очень простого протокола XML-RPC, оформленного как реализация языка XML. Рассмотрим его подробнее.

Протокол XML-RPC

Самое простое применение языка XML для сборки аргументов вызываемой удаленной процедуры, и пересылки результатов ее работы было сделано в 1999 году Дейвом Винером (Dave Winer), который вместе со своими друзьями создал протокол XML-RPC — реализацию XML и написал спецификацию XML-RPC. Она доступна по адресу **http://www.xmlrpc.com/spec.** В том же году был написан сервер XML-RPC, названный Frontier (**http://www.userland.com/**). С тех пор написано множество клиентов и серверов XML-RPC на разных языках: Java, Perl, Python, C/C++, Ruby.

Спецификация XML-RPC очень проста, она вводит не более двадцати элементов XML.

Пусть мы хотим получить прогноз погоды на завтра и обращаемся к процедуре

```
public String getWeatherForecast(String location);
```

выполняющейся на сервере метеослужбы. В процедуру заносится название местности `location`, она возвращает строку с прогнозом. Код процедуры нас сейчас не интересует.

Собранные по протоколу XML-RPC аргументы вызова этой удаленной процедуры, готовые к пересылке на сервер, выглядят так:

```
<?xml version="1.0"?>

<methodCall>
```

```
    <methodName>getWeatherForecast</methodName>

    <params>

      <param>
          <value>
             <string>Гадюкино</string>
          </value>
      </param>

  </params>

</methodCall>
```

Корневой элемент вызова удаленной процедуры называется <methodCall>. В него вложен элемент <methodName>, в теле которого записывается имя процедуры, и необязательный элемент <params>, тело которого содержит список аргументов процедуры.

Тело элемента <methodName> содержит строку символов, состоящую из букв A—Z, a—z, арабских цифр 0—9, знаков подчеркивания, точек, двоеточий и слешей.

В элемент <params> вложены нуль или несколько элементов <param>, описывающих аргументы процедуры.

Каждый элемент <param> описывает один аргумент процедуры вложенным элементом <value>. В него вкладывается элемент, описывающий тип аргумента. В табл. 2.1 приведен список этих элементов.

Таблица 2.1. *Типы аргументов XML-RPC*

Элемент	Тип
<i4> или <int>	Целое 4-байтовое число
<boolean>	0 (false) или 1 (true)
<string>	Строка символов ASCII
<double>	8-байтовое вещественное число
<dateTime.iso8601>	Дата и время в формате CCYYMMDDTHH:MM:SS
<base64>	Строка кода Base 64
<array>	Массив
<struct>	Структура

По умолчанию аргумент относится к типу <string>.

Как видно из таблицы 2.1, аргумент процедуры может быть массивом или структурой.

Аргумент-массив выглядит так:

```
<array>

   <data>
      <value><int>12</int></value>
      <value>Гадюкино</value>
      <value><boolean>0</boolean></value>
      <value><i4>-3K/i4x/value>
   </data>

</array>
```

В элемент <array> вкладывается один элемент <data>, в котором элементами <value> перечисляются элементы массива. Как видно из примера, элементы массива могут быть разных типов, в том числе снова массивы или структуры. Элементы массива различаются своими порядковыми номерами, поэтому порядок их записи очень важен.

Аргумент-структура оформляется в следующем виде:

```
<struct>

   <member>
      <name>day</name>
      <valuexi4>18</i4x/value>
   </member>

   <member>
      <name>month</name>
      <valuexi4>ll</i4x/value>
   </member>

</struct>
```

Структура подобна структуре языка С или записи языка Pascal. Она состоит из нескольких членов, описываемых элементами <member>. У каждого члена есть имя <name> — строка символов, и значение <value> любого типа, которым может быть и массив и другая структура. Члены структуры различаются по именам, поэтому порядок их записи не имеет значения.

Собранные аргументы процедуры в виде документа XML с корневым элементом <methodCall> пересылаются серверу по протоколу HTTP методом POST с типом содержимого

```
Content-Type: text/xml
```

Сервер оформляет ответ, содержащий результат выполнения процедуры, как документ XML вида

```
<?xml version="1.0"?>

<methodResponse>

   <params>

      <param>
         <value>Будут дожди</value>
      </param>
   </params>

</methodResponse>
```

В случае удачного выполнения процедуры в корневой элемент <methodResponse> ответа вкладывается один элемент <params>, содержащий один элемент <param> с единственным элементом <value>.

Если при выполнении процедуры возникли ошибки, то в элемент <methodResponse> вместо элемента <params> вкладывается элемент <fault>. В этом случае ответ выглядит так:

```
<?xml version="1.0"?>

<methodResponse>
  <fault>
    <value>

      <struct>

        <member>
           <name>faultCode</name>
           <value><int>4</int></value>
        </member>

        <member>
```

```
            <name>faultString</name>
            <value>Too many parameters.</value>
          </member>

        </struct>

      </value>
    </fault>
</methodResponse>
```

В элемент `<fault>` вкладывается элемент `<value>`, содержащий структуру с двумя членами, имеющими фиксированные имена `faultCode` и `faultString`. Эти члены содержат целочисленный код ошибки и строковое сообщение об ошибке. Спецификация не определяет никакие коды ошибок — это остается на долю реализации протокола.

Вот и все описание протокола XML-RPC. Как видите, он предельно прост, но описан неточно. Его спецификация не дает даже описания DTD определенных в ней элементов XML. При таком расплывчатом описании клиентскую и серверную части, реализующие этот протокол, должна делать одна команда программистов, одинаково понимающих все недоговоренности спецификации.

Кроме того, средств протокола XML-RPC явно недостаточно для вызова процедур со специфичными типами аргументов, с аргументами по умолчанию. Еще менее он приспособлен для вызова методов распределенных объектов. Поэтому теми же авторами в то же время был создан протокол **SOAP.**

Протокол SOAP

Протокол **SOAP** возник в 1998 году в фирме UserLand и корпорации Microsoft, но затем его разработка была передана в консорциум W3C, который и готовит сейчас рекомендации по его применению. Их можно посмотреть на странице проекта **http://www.w3.org/TR/SOAP/.**

Протокол **SOAP** не различает вызов процедуры и ответ на него, а просто определяет формат *послания* (message) в виде документа XML. Послание может содержать вызов процедуры, ответ на него, запрос на выполнение каких-то других действий или просто текст. Спецификацию SOAP не интересует содержимое послания, она задает только его оформление.

Корневой элемент посылаемого документа XML `<Envelope>` содержит необязательный заголовок <Header> и обязательное тело <Body>. Схема SOAP-послания такова:

```
<?xml version='1.0' ?>

<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope">

   <env:Header>
       <!— Блоки заголовка —>
   </env:Header>

   <env:Body>
      <!— Содержимое послания —>
   </env:Body>

</env:Envelope>
```

В заголовке содержится один или несколько блоков, оформление и содержание которых никак не регламентируются. Точно так же ничего не говорится о содержании тела послания. Тем не менее, различают *процедурный стиль* послания SOAP, предназначенный для вызова удаленных процедур, и *документный стиль,* предназначенный для обмена документами XML. Процедурный стиль часто называют *RPC-стилем*, а документный стиль — *XML-стилем.*

Процедурный стиль послания SOAP

При вызове удаленных процедур по протоколу SOAP в заголовке послания устанавливают параметры вызова, например, номер транзакции, в которой выполняется процедура. В теле послания перечисляются аргументы вызываемой процедуры.

Например, обращение к процедуре получения прогноза погоды, приведенной в предыдущем разделе, может выглядеть так:

```
<?xml version='1.0' ?>

<env:Envelope

    xmlns:env="http://www.w3.org/2002/06/soap-envelope">

  <env:Body>

    <met:getWeatherForecast env:encodingStyle=
           "http://www.w3.org/2002/06/soap-encoding"
```

```
        xmlns:met="http://www.meteoservice.com/">

      <met:location>Гадюкино</met:location>

    </met:getWeatherForecast>

  </env:Body>

</env:Envelope>
```

Аргументы вызываемой процедуры записываются как члены структуры, именем которой служит имя процедуры. Оно же служит именем элемента XML, представляющего структуру. Имя аргумента становится именем элемента XML, в теле которого записывается значение аргумента. Типы аргументов определены в пространстве имен, указанном атрибутом `encodingStyle`. Приведенное в примере значение этого атрибута **http://www.w3.org/2002/06/soap-encoding** определяет стандартное пространство имен SOAP, описанное в спецификации. В этом пространстве имен определены стандартные типы SOAP.

Записанное таким образом SOAP-послание передается серверу по сети каким-нибудь прикладным протоколом. Чаще всего это протокол **HTTP** или **SMTP.** При пересылке по протоколу HTTP редко используется метод GET. Обычно это метод **POST** с типом содержимого

Content-Type: `application/soap+xml;`

Ответ сервера выглядит так:

```
<?xml version='1.0' ?>

<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope">

  <env:Body>

    <met:getWeatherForecastResponse
       env:encodingStyle="http://www.w3.org/2002/06/soap-encoding"
       xmlns:rpc="http://www.w3.org/2002/06/soap-rpc"
       xmlns:met="http://www.meteoservice.com/">

       <rpc:result>met:forecast</rpc:result>

       <met:forecast>Будут дожди</met:forecast>
```

```
    </met:getWeatherForecastResponse>

  </env:Body>

</env:Envelope>
```

Имя элемента, вложенного в тело, не имеет значения, но часто повторяется имя процедуры, дополненное словом "Response", как сделано в примере.

Возвращаемое значение представляется структурой или массивом. Имя структуры или массива определяется элементом `<result>`. Этот элемент определен в пространстве имен с идентификатором **http://www.w3.org/2002/06/ soap-rpc.** Такое двухступенчатое описание возвращаемого значения — сначала описывается элемент met: forecast, а уж потом возвращаемое значение — удобно тем, что можно точно описать тип возвращаемого значения в схеме документа XML. В приведенном примере это можно сделать при определении элемента <met:forecast> в пространстве имен с идентификатором **http://www.meteoservice.com/.**

Впрочем, можно вернуть результат и прямо в теле элемента `<result>`.

Типы данных SOAP, определенные в пространстве имен с идентификатором **http://www.w3.org/2002/06/soap-encoding,** во многом совпадают с типами данных языка XSD *(*см. *главу 1).* Их можно указать в схеме документа или прямо в документе, например:

```
<?xml version="1.0"?>

<env:Envelope
   env:encodingStyle="http://www.w3.org/2002/06/soap-encoding"
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <env:Body>

      <met:getWeatherForecast

          xmlns:met="http://www.meteoservice.com/">

         <met:location xsi:type="xsd:string">

             Гадюкино

         </met:location>
```

```
        </met:getWeatherForecast>

    </env:Body>

</env:Envelope>
```

Как видно из приведенных примеров, кроме типов XSD, в SOAP-посланиях широко применяются массивы и структуры.

Массив состоит из безымянных элементов, они различаются своим порядковым номером, например:

```
<person enc:arrayType="xsd:string[3]" xsi:type="enc:Array">
    <name>Иван</name>
    <name>Петрович</name>
    <пате>Сидоров</пате>
</person>
```

Элементы массива надо записывать в определенном порядке.

В отличие от массива у каждого элемента структуры есть свое имя, например:

```
<person>
    <firstname>Иван</firstname>
    <secondname>Петрович</secondname>
    <surname>Сидоров</surname>
</person>
```

Элементы структуры можно записывать в любом порядке.

Документный стиль послания SOAP

Послание SOAP может содержать не аргументы вызова какой-то процедуры, а просто документ XML или структуру данных. Важно только сохранить схему послания — у него может быть необязательный заголовок и должно быть обязательное тело. Например:

```
<?xml version='1.0' ?>

<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope">

    <env:Header>
        <!— Блоки заголовка -->
    </env:Header>
```

```
    <env:Body>

        <ms:passport xmlns:ms="http://burou.org/">

            <ms:series>13-XM</ms:series>
            <ms:number>123456</ms:number>

        </ms:passport>

    </env:Body>

</env:Envelope>
```

Такие послания характерны для асинхронного обмена сообщениями в духе электронной почты. Очень часто они пересылаются не по протоколу HTTP, а по какому-нибудь почтовому протоколу, например, SMTP .

Мы ознакомимся подробно с протоколом SOAP в следующей главе, а пока заметим лишь, что консорциум W3C решил разработать стандартный протокол пересылки документов XML, взяв за основу XML-RPC, SOAP и другие известные протоколы. Новый протокол сначала был назван XP (XML Protocol). Это сокращение совпало с широко распространенным обозначением экстремального программирования (eXtreme Programming). Кроме того, в моду вошли четырехбуквенные аббревиатуры, и новый протокол назвали XMLP (XML Protocol). Его разработка затягивается. На время написания книги она была на начальной стадии, и не было ни одной производственной реализации протокола XMLP. Более того, появились сведения, что консорциум оставит за ним название SOAP, изменив только номер версии протокола.

Средства разработки SOAP

Итак, протокол предоставления Web-услуг есть, остается его реализовать. Написано уже много средств создания SOAP-серверов и SOAP-клиентов. Их длинный список можно посмотреть на сайте **http://www.soapware.org/**. Ведущие фирмы, выпускающие продукты для предоставления Web-услуг — IBM, Microsoft, Sun, Oracle — выпустили свои SOAP-продукты.

Фирма IBM вначале выпустила продукт разработки SOAP-серверов и клиентов SOAP4J, но затем передала его разработку в сообщество Apache Software Foundation, выпустившее на его основе набор классов и интерфейсов Apache SOAP **(http://xml.apache.org/SOAP/).** Этот продукт входил в состав сервера приложений IBM WebSphere и до сих пор входит в состав многих серверов приложений.

Сообщество Apache, с появлением сообщений о протоколе XP, а затем о протоколе XMLP, кроме сервера Apache SOAP, стало выпускать средство реализации протокола XMLP под названием Axis (**http://xml.apache.org/axis/**). В названии звучит ирония по поводу переименований исходного протокола XMLP, но оно расшифровывается как Apache extensible Interaction System. Его иногда называют Apache SOAP 3, следующей версией Apache SOAP. К моменту выпуска Axis протокол XMLP еще не был окончательно разработан, и Axis надо считать самостоятельным продуктом, реализующим некоторое расширение протокола SOAP. Axis входит в состав сервера приложений IBM WebSphere Application Server (**http://www-3.ibm.com/software/webservers/**) и многих других серверов приложений.

Для создания SOAP-серверов и клиентов фирма IBM выпускает отдельный набор инструментальных средств разработчика Web-служб под названием WSTK (Web Services Toolkit) (**http://www.alphaworks.ibm.com/tech/webservicestoolkit**), основанный на Axis. Конечно, для разработки Web-служб можно воспользоваться и основным средством разработчика IBM WebSphere Studio (**http://www-4.ibm.com/software/webservers/studio/**).

Фирма Sun Microsystems выпускает пакет интерфейсов и классов `javax.xml.soap`, названный SAAJ (SOAP with Attachments API for Java) и описанный спецификацией SAAJ. Интерфейсы и классы, входящие в этот пакет, помогают сформировать SOAP-послания и реализовать синхронный обмен ими. Асинхронный обмен посланиями осуществляется с помощью дополнительных интерфейсов и классов, входящих в пакет `javax.xml.messaging`, названный JAXM (Java API for XML Messaging). Этот пакет использует для пересылки SOAP-посланий систему обмена сообщениями JMS (Java Message Service).

Классы, входящие В пакеты javax.xmi. soap И `javax.xml.messaging`, действуют в документном стиле SOAP. Они просто записывают и отправляют послание с любым заголовком и телом. Процедурный стиль посланий SOAP реализован фирмой Sun в пакете `javax.xml.rpc` и его подпакетах. Этот набор пакетов назван JAX-RPC (Java API for XML-RPC). В нем используется механизм обращения к методам удаленных объектов RMI, а SOAP применяется как транспортный протокол вместо "родного" для RMI протокола JRMP (Java Remote Method Protocol) или протокола CORBA ПОР (Internet Inter-ORB Protocol).

Все эти пакеты фирмы Sun входят в набор инструментальных средств J2EE (**http://java.sun.com/j2ee/**) и в сервер приложений Sun ONE Application Server (**http://wwws.sun.com/software/products/appsrvr/**). Кроме того, выпускается отдельный набор средств создания Web-служб WSDP (Web Services Developer Pack) (**http://java.sun.com/webservices/webservicepack.html**). Для разработки Web-служб можно воспользоваться и универсальным средством разработки Sun ONE Studio (**http://wwws.sun.com/software/sundev/jde/**).

Корпорация Microsoft — пионер создания SOAP — выпускает собственный набор средств реализации протокола SOAP под названием Microsoft SOAP Toolkit (**http://msdn.microsoft.com/webservices/downloads/**) и включает его в свой Web-сервер IIS как ISAPI-сервер или как ASP-сервер. Конечно, для разработки Web-служб можно пользоваться и Microsoft Visual Studio .NET (**http://msdn.microsoft.com/vstudio/**).

Корпорация Oracle (**http://www.oracle.com/**) использует Apache SOAP, а также собственную разработку, входящую в состав OC4J (Oracle9iAS Containers for J2EE), и включает их в сервер приложений Oracle9i Application Server. Для разработки Web-служб Oracle предлагает свой продукт JDeveloper.

Фирма Borland Software Corporation использует Axis и Apache SOAP в составе своего сервера приложений BES (Borland Enterprise Server) (**http://www.borland.com/besappserver/**). Для разработки Web-служб фирма Borland предлагает использовать JBuilder (**http://www.borland.com/jbuilder/**) с добавлением пакета Borland Web Services Kit.

Фирма TME (The Mind Electric) выпускает средство разработчика Web-служб GLUE (**http://www.themindelectric.com/glue/**). Его можно встраивать в серверы приложений или применять как отдельный продукт.

Есть еще масса средств реализации протокола SOAP, которые невозможно перечислить хотя бы потому, что каждый день появляются все новые и новые продукты.

Создание простейшей Java Web-службы

Продолжим рассмотрение примера метеослужбы и опишем ее классом Java. Назовем этот класс `MeteoService`. Наша метеослужба предоставляет только одну Web-услугу — прогноз погоды на завтра. Пусть эта услуга предоставляется методом getweatherForecast о. Не будем изощряться в сложных алгоритмах прогнозирования погоды, а просто дадим наиболее вероятный для России прогноз.

В листинге 2.1 приведено описание нашей Web-службы.

Листинг 2.1. Класс, описывающий простейшую Web-службу

```
public class MeteoService{

   public String getWeatherForecast(String location);

      return location + ": пойдет дождь или снег";
   }
}
```

Воспользуемся Axis для реализации нашей Web-службы. Для этого запишем исходный текст класса `MeteoService` в файл MeteoService.jws (расширение имени файла ".jws" означает "Java Web Service"), а файл занесем в каталог axis нашего сервера приложений, использующего Axis. Каталог axis должен быть вложен в каталог `webapps` сервера приложений.

Все, Web-служба создана и готова предоставить свои услуги любому клиенту. Не нужна ни компиляция, ни установка класса `MeteoService` в Web-контейнер. Получив SOAP-послание, обращающееся к Web-службе прямо через файл MeteoService.jws, сервер приложений, увидев расширение имени файла ".jws", обратится к Axis, точнее к сервлету `AxisServlet`. С этого сервлета Axis начинает работу на сервере.

В листинге 2.2 приведен клиент метеослужбы, написанный просто как Java-приложение, использующее Axis.

Листинг 2.2. Клиент метеослужбы, написанный с помощью Axis

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;
import java.net.*;

public class MeteoServiceClient{

   public static void main(String[] args) throws Exception;

      if (args.length != 1){

         System.err.println("Usage: " +
            "java MeteoServiceClient <location>");

         System.exit(1);
      }

      Service service = new Service();

      Call call = (Call)service.createCall();

      String endpoint =
         "http://www.meteo.com:8080/axis/MeteoService.jws";
```

```
        call.setTargetEndpointAddress(new URL(endpoint));
        call.setOperationName(new QName("getWeatherForecast"));

        String result =
          (String)call.invoke(new Object[]{new String(args[0])});

        System.out.println("Погода на завтра: " + result);
    }
}
```

Откомпилируем программу листинга 2.2 на машине клиента:

```
$ javac MeteoServiceClient.java
```

и запустим программу-клиент:

```
$ java MeteoServiceClient Гадюкино
```

На консоли увидим прогноз погоды на завтра.

Как видите, для получения услуги надо только знать адрес URL Web-службы, размещенной в файле MeteoService.jws, и передать ей имя населенного пункта. Все остальное Axis берет на себя.

Посмотрим подробнее на то, что делает Axis в листинге 2.2. Вначале создается объект класса service. Этот объект установит связь с Web-службой. Он предоставляет экземпляр класса `Call`, в котором формируется HTTP-запрос, содержащий SOAP-послание. Для создания запроса в объект класса `Call` заносится адрес Web-службы и имя Web-услуги "getWeatherForecast". После того как запрос сформирован, Axis обращается к Web-услуге методом invoke (). Аргумент этого метода — массив объектов — содержит аргументы предоставляемой Web-услуги.

Запрос направляется по сети серверу приложений, работающему по указанному в запросе адресу URL. Сервер приложений запускает сервлет `AxisServlet`, играющий роль диспетчера и контейнера, в котором работают Web-службы. Он разбирает SOAP-послание, пришедшее в запросе, отыскивает указанный в послании файл MeteoService.jws и компилирует его содержимое. Затем Axis загружает `MeteoService.class` и обращается к методу getWeatherForecast(), передав ему аргумент. После выполнения этого метода Axis формирует SOAP-ответ и отправляет его клиенту.

Клиентская часть Axis разбирает полученный ответ и передает его как результат выполнения метода invoke ().

Программу-клиент Web-службы, показанную в листинге 2.2, можно встроить в клиентское приложение или использовать отдельно, снабдив ее графическим интерфейсом.

Запрос, сформированный Axis и посылаемый клиентом нашей метеослужбы на сервер, без HTTP-заголовка выглядит буквально так, как показано в листинге 2.3.

Листинг 2.3. SOAP-послание клиента метеослужбы

```
<?xml version="1.0" encoding="UTF-8"?>

<SOAP-ENV:Envelope
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

   <SOAP-ENV:Body>

     <getWeatherForecast>
        <arg0 xsi:type="xsd:string">Гадюкино</arg0>
     </getWeatherForecast>

   </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Как видите, нигде в запросе нет упоминания о платформе, на которой работает клиент, и тем более не сказано, что клиентом должно быть Java-приложение. Поэтому клиентом Axis может быть любая программа, способная составить такой запрос и разобрать ответ сервера. Ее можно написать на любом языке. Поскольку книга посвящена созданию Web Services с помощью Java, приведем в листинге 2.4 клиент, написанный только стандартными сетевыми средствами Java, без использования Axis.

Листинг 2.4. Клиент метеослужбы

```
import java.net.*;
import java.io.*;

public class MeteoServiceClient2{

  public static void main(String[] args){
```

```
    if (args.length != 1){

       System.err.println("Usage: " +
          "java MeteoServiceClient2 <location>");

       System.exit(1);
    }

String message =
  "<?xml version='1.0' encoding='UTF-8'?>" +
"<SOAP-ENV:Envelope" +
  "SOAP-ENV:encodingStyle=" +
    "'http://schemas.xmlsoap.org/soap/encoding/'" +
"xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'" +
"xmlns:xsd='http://www.w3.org/2001/XMLSchema'" +
"xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'" +
"xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'>" +
"<SOAP-ENV:Body>" +
   "<getWeatherForecast>" +
      "<arg0 xsi:type='xsd:string'>" + args[0] + "</arg0>" +
   "</getWeatherForecast>" +
  "</SOAP-ENV:Body>" +
"</SOAP-ENV:Envelope>";

try{
   byte[] data = message.getBytes("KOI8-R");

   URL url =
     new URL("http://www.meteo.com:8080/axis/MeteoService.jws");

   URLConnection uc = url.openConnection();

   uc.setDoOutput(true);
   uc.setDoInput(true);
   uc.setUseCaches(false);

   uc.setRequestProperty("Content-Type",
         "text/xml;charset=\"utf-8\"");
```

```
        uc.setRequestProperty("Content-Length",
             ""+message.length());

        uc.connect();

        DataOutputStream dos =
             new DataOutputStream(uc.getOutputStream());

        dos.write(data, 0, data.length);
        dos.close();

        BufferedReader br = new BufferedReader(
           new InputStreamReader(
                  uc.getInputStream(), "KOI8-R"));

        String res = null;

        while ((res = br.readLine()) != null)
          System.out.println(res);

        br.close();

      }catch(Exception e){

        System.err.println("From client: " + e);
        e.printStackTrace(System.out);
      }
   }
}
```

Ответ, полученный клиентом в листинге 2.4, приведен в листинге 2.5.

Листинг 2.5. SOAP-ответ метеослужбы

```
<?xml version="1.0" encoding="UTF-8"?>

<SOAP-ENV:Envelope
   xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
   xmlns:xsd='http://www.w3.org/2001/XMLSchema'
   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
```

```
   <SOAP-ENV:Body>

      <getWeatherForecastResponse
         SOAP-ENV:encodingStyle=
            'http://schemas.xmlsoap.org/soap/encoding/'>

         <getWeatherForecastResult xsi:type='xsd:string'>
             Гадюкино: пойдет дождь или снег
         </getWeatherForecastResult>

      </getWeatherForecastResponse>

   </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Конечно, клиент, показанный в листинге 2.4, должен еще разобрать ответ, присланный сервером в виде SOAP-послания листинга 2.5, выделив из него результат — прогноз погоды. Для этого придется применить SAX-парсер, один из описанных в *главе 1.* Не будем сейчас заниматься этим.

Внимательный читатель заметил, что та часть Axis на сервере, которая принимает SOAP-послания, оформлена в виде сервлета `AxisServlet`. Действительно, в технологии Java Web-службы реализуются сервлетами и страницами JSP, которые, возможно, обращаются к компонентам EJB типа SLSB (Stateless Session Beam) или типа MDB (MessageDriven Bean). Нашу простую метеослужбу можно реализовать без Axis одним сервлетом, например, таким, какой показан в листинге 2.6.

Листинг 2.6. Сервлет, реализующий метеослужбу

```
import java.io.*;
import java.util.*;
import javax.servlet.http.*;

public class MeteoServiceServlet extends HttpServlet{

   public String getWeatherForecast(String location))
      return location + ": пойдет дождь или снег";
   }

  public void doPost(HttpServletRequest req,
```

```
                    HttpServletResponse resp){
    try{
       req.setCharacterEncoding("KOI8-R");

       BufferedReader br = req.getReader();

       String message = "", buf = "", location = "";

       while ((buf = br.readLine()) != null)
          message += buf;

       // Анализ SOAP-послания message SAX-парсером и
       // выделение аргумента location.

      resp.setContentType("text/xml; charset=\"utf-8\"");

      String response =
         "<?xml version='1.0' encoding='UTF-8'?>" +
"<SOAP-ENV:Envelope" +
"xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'" +
"xmlns:xsd='http://www.w3.org/2001/XMLSchema'" +
"xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>" +
"<SOAP-ENV:Body>" +
   "<getWeatherForecastResponse" +
     "SOAP-ENV:encodingStyle=" +
           'http://schemas.xmlsoap.org/soap/encoding/'>" +
     "<getWeatherForecastResult xsi:type='xsd:string'>" +

          getWeatherForecast(location) +

     "</getWeatherForecastResult>" +
  "</getWeatherForecastResponse>" +
"</SOAP-ENV:Body>" +
"</SOAP-ENV:Envelope>";

       PrintWriter pw = resp.getWriter();

       pw.println(response);
```

```
            pw.flush();
            pw.close();

        }catch(Exception e){
            System.err.println(e);
        }
    }
}
```

Вся наша простейшая метеослужба уместилась в одном методе `getWeatherForecast()`. В реальной ситуации ее лучше вынести в один или несколько компонентов EJB.

Итак, построить любую Java Web-службу можно стандартными свободно доступными средствами Java, имеющимися в пакетах инструментальных средств J2SDK SE, J2SDK EE и WSDP. Их можно свободно загрузить с сайта **http://java.sun.com/.**

Описание Web-службы

После создания Web-службы и размещения ее на сервере в виде сервлета, страницы JSP, JWS-файла или другого объекта, следует подумать, как об этой службе узнают клиенты. На каждом сервере необходимо создать список имеющихся на нем Web-служб и услуг, предоставляемых ими. Но для этого надо точно знать, какие именно услуги оказывает каждая Web-служба, и описать их.

Средства для такого описания есть во многих языках — это интерфейсы Java, язык IDL, на котором описываются объекты CORBA, заголовочные header-файлы языка C. Для описания Web-служб нужен язык, не зависящий от платформы, операционной системы, системы программирования. Поэтому для точного описания Web-услуг консорциумом W3C создан специальный язык WSDL. Этот язык — еще одна реализация XML. Его спецификация опубликована на странице **http://www.w3.org/TR/wsdl.**

Корневым элементом документа XML — описания WSDL — служит элемент `<definitionsx` В этом элементе атрибутом name можно дать имя описанию. Кроме того, это удобное место для введения используемых в описании пространств имен. Описания WSDL активно используют различные пространства имен. Кроме собственных имен WSDL часто использует язык описания схем XSD (см. *главу 7)* и язык протокола SOAP. Пространство имен WSDL часто описывается как пространство имен по умолчанию, тогда целевое пространство имен получает префикс, обычно tns (Target NameSpace).

В корневой элемент вкладываются пять основных элементов.

- <types> — определяет сложные типы XSD, используемые Web-службой. Этот элемент не нужен, если служба применяет только простые типы XSD.
- <message> — описывает каждое SOAP-послание: запрос, ответ, пересылку документов. В этот элемент вкладываются элементы <part>, описывающие имя и тип каждого аргумента запроса или возвращаемого значения.
- <portType> — описывает Web-службу, называемую здесь *пунктом назначения* (endpoint) или *портом* (port) прибытия послания. Она описывается как набор услуг, называемых здесь *операциями*. Операции описываются вложенными элементами <operation>, описывающими каждую услугу: отправку запроса — получение ответа или, наоборот, получение запроса — отправку ответа. Получение и отправка, в свою очередь, описываются вложенными элементами <input> и <output>, а сообщение об ошибке — элементом <fault>.
- <binding> — описывает конкретный формат пересылки: протоколы SOAP, HTTP, тип содержимого HTML, XML или другой MIME-тип послания. У каждой Web-службы может быть несколько таких элементов, по одному для каждого способа пересылки.
- <service> — указывает местоположение Web-службы как один или несколько *портов*. Каждый порт описывается вложенным элементом <port>, содержащим адрес Web-службы, заданный по правилам выбранного в элементе <binding> способа пересылки.

Кроме этих пяти основных элементов есть еще два дополнительных элемента.

- <import> — включает файл с XSD-схемой описания WSDL или другой WSDL-файл.
- <documentation> — комментарий. Его можно включить в любой элемент описания WSDL.

Можно сказать, что элементы <types>, <message> и <portType> показывают, ЧТО есть в описываемой Web-службе, какие услуги она предоставляет, как организованы услуги, какие типы данных у этих услуг.

Элементы <binding> объясняют, КАК реализована Web-служба, каков протокол передачи посланий: HTTP, SMTP или что-то еще, задает технические характеристики передачи данных.

Наконец, элементы <service> показывают, ГДЕ находится Web-служба, связывая описание <binding> с конкретными адресами Web-службы.

В листинге 2.6 показано, как будет выглядеть WSDL-описание нашей метеослужбы.

Листинг 2.6. Описание WSDL метеослужбы

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions name="MeteoService"
   targetNamespace="http://www.meteo.com/wsdl/MeteoService.wsdl"
   xmlns="http://www.w3.org/2002/07/wsdl"
   xmlns:soap="http://www.w3.org/2002/07/wsdl/soap12"
   xmlns:tns="http://www.mi.com/wsdl/MeteoService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="ForecastRequest">
      <part name="location" type="xsd:string" />
   </message>

   <message name="ForecastResponse">
      <part name="forecast" type="xsd:string" />
   </message>

   <portType name="Meteo_PortType">

      <operation name="getWeatherForecast">
         <input  message="tns:ForecastRequest" />
         <output message="tns:ForecastResponse" />
      </operation>

   </portType>

   <binding name="Meteo_Binding" type="tns:Meteo_PortType">

      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http" />

      <operation name="getWeatherForecast">
         <soap:operation soapAction="getWeatherForecast" />
         <input>
            <soap:body encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/"
```

```
            namespace="urn:MeteoService"
            use="encoded" />
        </input>
        <output>
           <soap:body encodingStyle=
                 "http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:MeteoService"
            use="encoded" />
        </output>
     </operation>

  </binding>

  <service name="Meteo_Service">

     <documentation>WSDL File for MeteoService</documentation>

     <port binding="tns:Meteo_Binding" name="Meteo_Port">

        <soap:address
         location="http://www.meteo.com:8080/servlet/MeteoServlet" />

     </port>

  </service>

</definitions>
```

В листинге 2.6 мы дали в элементе <definitions> описанию имя "MeteoService" и определили префиксы всех нужных нам пространств имен. Далее мы описали запрос и ответ в двух элементах <message>. Мы дали им имена "ForecastRequest" и "ForecastResponse". В запросе один аргумент типа xsd: string. Этот тип определен в языке XSD. Мы дали аргументу имя location, совпадающее с именем аргумента метода getweatherForecast (). Значению, возвращаемому методом, мы дали имя forecast, его тип тоже xsd: string.

Имена "ForecastRequest" и "ForecastResponse" использованы в следующем элементе <portType> для указания входных и выходных параметров Web-услуги. В нем один элемент <operation>. Это означает, что Web-служба "Meteo_PortType" предоставляет одну услугу, имя которой

"getWeatherForecast" совпадает с именем метода, выполняющего эту услугу. В элементе <operation> указаны входной <input> и выходной <output> параметры услуги.

Затем, элементом <binding> мы указываем один способ пересылки посланий, а именно, как SOAP-посланий в процедурном стиле, пересылаемых по протоколу HTTP, на что указывает элемент

```
<soap:binding style="rcp"
    transport="http://schemas.xmlsoap.org/soap/http" />
```

Документный стиль SOAP отмечается значением "document" атрибута style.

Далее повторяется описание операции, но уже в терминах SOAP.

Наконец, элементом <service> указываем адрес, по которому расположена Web-служба.

Мы рассмотрим подробно спецификацию WSDL в *главе 4,* а пока дадим обзор инструментальных средств для создания WSDL-описаний.

Инструменты создания описаний WSDL

Описание WSDL написано в листинге 2.6 вручную. Однако строгость описания позволяет автоматизировать этот процесс. Многие инструментальные средства создания Web-служб содержат утилиты, которые автоматически создают WSDL-файлы, описывающие готовые Web-службы. Например, уже упоминавшееся средство создания Web-служб Apache Axis содержит в своем составе класс Java2WSDL, создающий WSDL-файл по классу или интерфейсу Java. Пакет IBM WSTK, в состав которого входит Axis, содержит утилиту java2wsdl, создающую объект этого класса и работающую из командной строки. Достаточно набрать в командной строке

```
$ java2wsdl MeteoService
```

и будет создан файл MeteoService.wsdl, содержащий по одному элементу <portType> для каждого открытого метода класса, указанного в командной строке.

Интересно, что Axis может выполнить и обратное действие: по имеющемуся WSDL-файлу класс WSDL2java создаст все классы Java, необходимые для работы Web-службы. В пакете IBM WSTK класс WSDL2java можно вызвать из командной строки утилитой wsdl2java. Например:

```
$ wsdl2java MeteoService.wsdl
```

Такие же утилиты есть в составе уже упоминавшегося продукта GLUE фирмы The Mind Electric.

Пакет Microsoft SOAP Toolkit содержит графическую утилиту `wsdlgen3.exe`, вызываемую из стартового меню WSDL Generator, и утилиту командной строки `wsdlstb3.exe`, которые создают WSDL-файлы.

Фирма Sun Microsystems готовит к выпуску пакет интерфейсов JWSDL (Java API for WSDL), преобразующих описание WSDL в классы Java и обратно. Это позволяет программно создавать, изменять, читать описания WSDL. Фирма IBM уже реализовала этот пакет в своем продукте WSTK, назвав набор интерфейсов и реализующих их классов WSDL4J (WSDL for Java). Этот набор можно использовать в составе WSTK или отдельно, загрузив его с сайта **http://www-124.ibm.com/developerworks/projects/wsdl4j/**.

Самое ценное в описаниях WSDL то, что клиент Web-службы может обратиться не к ней самой, а к ее WSDL-описанию. В состав GLUE входит утилита командной строки invoke, обращающаяся к Web-службе по ее WSDL-описанию. Например, достаточно набрать в командной строке:

```
$ invoke http://www.meteo.com:8080/services/MeteoService.wsdl  Гадюкино
```

— и на консоли появится прогноз погоды.

Фирма IBM выпускает пакет классов WSIF (Web Services Invocation Framework), работающий в Web-контейнере Tomcat под управлением Apache SOAP. С помощью этого пакета можно, в частности, сделать ту же работу:

```
$ java clients.DynamicInvoker \
   http://www.meteo.com:8080/services/MeteoService.wsdl \
   getWeatherForecast  Гадюкино
```

Напомним, что обратная наклонная черта здесь означает продолжение командной строки на следующую строку текста.

После этого на консоли появляются сообщения WSIF и прогноз погоды.

Регистрация Web-службы

Во всех приведенных выше примерах Web-служба вызывалась по ее адресу, записанному строкой URI. Это приемлемо для вызова Web-услуг из командной строки или графического приложения, но неудобно для работы распределенного приложения, поскольку адрес Web-службы может поменяться и придется сделать замены во всех клиентских приложениях. Для стабильной работы распределенных приложений нужно средство автоматического поиска и связи с Web-службами, подобное реестру RMI, службам именования JNDI или CORBA Naming Service, сетевой информационной службе NIS, применяемой в UNIX. Это особенно важно для слабо связанных приложений: ведь основное их достоинство — быстрый поиск "на лету" необходимых для работы компонентов.

Создано уже несколько систем поиска Web-служб. Наиболее распространены две системы: универсальная система описания, обнаружения и интеграции UDDI и электронный бизнес-реестр ebXML Registry (electronic business XML Registry). Обе системы громоздки и сложны в употреблении. Поэтому фирмы IBM и Microsoft недавно разработали облегченную систему обнаружения Web-служб, названную WS-Inspection (Web Services Inspection Language).

Схема взаимодействия клиента Web-службы с ее поставщиком через реестр показана на рис. 2.3.

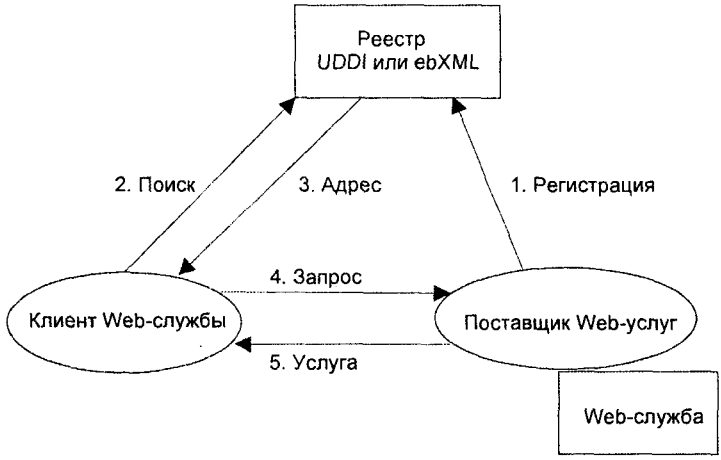


Рис. 2.3. Взаимодействие клиента Web-службы с ее поставщиком

Рассмотрим последовательно эти три наиболее употребительные службы поиска Web-служб.

Система описания и обнаружения UDDI

Система описания, обнаружения и интеграции UDDI создана фирмами IBM (**http://www-3.ibm.com/services/uddi/**) и Microsoft (**http://uddi.microsoft.com/**). Сейчас она развивается группой крупных компаний. На официальном сайте сообщества UDDI **http://www.uddi.org/** приведен список около трехсот компаний-участников проекта. Сообщество выпустило спецификацию UDDI, которую можно получить на том же сайте. Спецификация уже реализована множеством продуктов разработки реестра UDDI.

Реестр UDDI (UDDI Business Registry) состоит из множества *узлов* (nodes), размещенных в Интернете. Они хранят информацию о Web-службах, доступную на всех узлах, образующих распределенный UDDI-реестр. Клиент "видит" UDDI-реестр как единое целое, совершенно не ощущая того, что

он размещен на нескольких машинах. Конечно, сами узлы организованы как Web-службы, а реестр UDDI — как слабо связанное распределенное приложение.

Многие крупные компании организовали и содержат свои UDDI-реестры, например, реестр фирмы IBM расположен по адресу **https://www-3.ibm.com/services/uddi/v2beta/protect/registry.html**, адрес реестра компании Hewlett Packard **https://uddi.hp.com/uddi/index.jsp**, реестр корпорации Microsoft находится на сайте **https://uddi.rte.microsoft.com/register.aspx**. Эти UDDI-реестры связаны между собой и постоянно обмениваются информацией. Кроме того, это открытые (public) реестры. Любой желающий может зарегистрировать в них свою Web-службу или отыскать нужную Web-услугу. Фирмы могут организовать и закрытые частные (private) реестры, доступные только зарегистрированным участникам. Список UDDI-реестров можно посмотреть на сайте проекта UDDI.

Реестр UDDI разбивает хранящуюся в нем информацию на несколько групп.

Четыре основные группы состоят из следующих элементов.

- Бизнес-информация — XML-элемент <businessEntity> — описание фирмы-поставщика Web-услуг: ее ключ UUID (Unique Universal Identifier), уникальный в пределах реестра и описанный атрибутом businessKey, название фирмы — вложенный элемент <name>, краткое описание сферы ее деятельности, типы предоставляемых услуг, контактная информация, ссылки URL. Эта информация предназначена для всех, кто хочет воспользоваться услугами фирмы.

- Бизнес-услуги — элемент <businessServices>, вложенный в элемент <businessEntity> - список услуг, оказываемых фирмой. Каждая услуга описывается вложенным элементом <businessService>. В описание входит ключ UUID каждой услуги, описанный атрибутом serviceKey, имя услуги — вложенный элемент <name>, ее краткое описание и ссылки на подробную информацию. Услуги могут быть любыми, не обязательно Web-услугами.

- Указатели на услуги — элемент <bindingTemplates>, вложенный в элемент <businessservice> - способы получения каждой услуги. Они могут быть прямыми, например, URL-адрес Web-службы, или косвенными, например, описание WSDL или IDL. Каждый способ описывается элементом <bindingTemplate>. Его атрибут bindingKey определяет уникальный ключ UUID указателя. Элемент <bindingTemplate> содержит ссылку на соответствующий элемент <tModel>.

- Модель услуги — элемент <tModel> (technical Model) — подробное формальное описание каждой услуги. Оно используется программным обеспечением узла. Обычно это отдельный документ XML.

В реестре есть еще несколько дополнительных элементов.

- Утверждение — элемент <publisherAssertion> — описание установленных ранее отношений между фирмами (утверждение peer-peer) или фирмой и ее подразделениями (утверждение parent-child). Фирма *утверждает,* что она тесно связана с перечисляемыми фирмами или что это — ее подразделения. Третий вид утверждения — identity — отношение между одинаковыми фирмами — это фактически псевдоним. Отношение входит в силу, когда его утвердят оба участника. Это отдельный документ, использующий элементы <businessEntity> обеих фирм.
- Подписка — элемент <subscription> — список фирм и сведений, которые надо послать перечисленным фирмам при каких-либо изменениях в деятельности фирмы.

Итак, схема одного из документов XML, хранящихся в UDDI-реестре, такова:

```
<businessEntity businessKey="ключ UUID">

   <businessServices>

      <businessService serviceKey="ключ UUID">

         <bindingTemplates>

           <bindingTemplate bindingKey="ключ UUID">
               <!— Описание указателя -->
           </bindingTemplate>

           <!— Описания других указателей —>

         </bindingTemplates>

      </businessService>

      <!— Описания других услуг —>

   </businessServices>

</businessEntity>
```

Клиент обращается к UDDI-реестру для того, чтобы зарегистрировать свою Web-службу, изменить ее, или для того, чтобы отыскать нужную Web-услугу. Реестр предоставляет необходимый для этого интерфейс. Регистра-

ция и последующие изменения выполняются методами `save xxx()`, поиск Web-службы и получение услуги — методами `find_xxx()` и `get_xxx{}`. Все эти методы описаны в спецификации UDDI.

Есть уже много реализаций интерфейса UDDI на различных языках. В технологии Java наиболее популярен пакет классов UDDI4J (UDDI for Java), разработанный фирмой IBM **(http://www.uddi4j.org/).** Классы, входящие в этот пакет, обеспечивают выполнение всех действий клиента в UDDI-реестре.

UDDI-интерфейс реализован в пакете UDDI4J классом `UDDIProxy`. Точнее говоря, этот класс служит посредником (proxy), обращающимся к соответствующим методам интерфейса UDDI. Методы этого класса возвращают объекты классов, предоставляющих дополнительную информацию. Мы рассмотрим подробно пакет UDDI4J в *главе 5.*

В листинге 2.7 показано, как можно зарегистрировать нашу метеослужбу в том или ином UDDI-реестре, а потом найти ее.

Листинг 2.7. Регистрация метеослужбы в UDDI-реестре

```
import com.ibm.uddi.*;
import com.ibm.uddi.datatype.business.*;
import com.ibm.uddi.response.*;
import com.ibm.uddi.client.*;
import org.w3c.dom.Element;
import java.util.Vector;
import java.util.Properties;

public class SaveMeteoService{

   public static void main (String[] args){

      SaveMeteoService app = new SaveMeteoService();

      UDDIProxy proxy = new UDDIProxy();

      try{
         // Выбираем UDDI-реестр:

         // Тестовый реестр IBM
       proxy.setInquiryURL("http://www-3.ibm.com/services/" +
```

```
            "uddi/testregistry/inquiryapi");

    proxy.setPublishURL("https://www-3.ibm.com/services/" +
             "uddi/testregistry/protect/publishapi");

      // Официальный реестр IBM
// proxy.setInquiryURL("http://www-3.ibm.com/services/" +
//         "uddi/inquiryapi");
// proxy.setPublishURL("https://www-3.ibm.com/services/" +
//         "uddi/protect/publishapi");

      // Тестовый реестр Microsoft
// proxy.setInquiryURL(
//         "http://test.uddi.microsoft.com/inquire");
// proxy.setPublishURL(
//         "https://test.uddi.microsoft.com/publish");

      // Официальный реестр Microsoft
// proxy.setInquiryURL("http://uddi.microsoft.com/inquire");
// proxy.setPublishURL("https://uddi.microsoft.com/publish");

      // Реестр Hewlett Packard
// proxy.setInquiryURL("http://uddi.hp.com/inquire");
// proxy.setPublishURL("https://uddi.hp.com/publish");

      // Отладочный локальный реестр из пакета WSTK
// proxy.setInquiryURL(
//      "http://localhost:8080/services/uddi/inquiryapi");
// proxy.setPublishURL(
//       "http://localhost:8080/services/uddi/publishapi");

     // Заходим на сайт UDDI-реестра.
    AuthToken token = proxy.get_authToken(
                           "userid", "password");

    System.out.println(
         "Регистрационный код:  " + token.getAuthInfoString());

    System.out.println("Регистрируем Meteo Service");
```

```
    Vector entities = new Vector();

      // Создаем элемент <businessEntity>.
      // Первый аргумент — UUID — пока неизвестен.
    BusinessEntity be = new BusinessEntity("", "MeteoService");

    entities.addElement(be);

      // Регистрируем Web-службу
    BusinessDetail bd =
      proxy.save_business(token.getAuthInfoString(), entities);

      // Для проверки получаем UUID.
    Vector businessEntities = bd.getBusinessEntityVector();

    BusinessEntity returnedBusinessEntity =
        (BusinessEntity)(businessEntities.elementAt(0));

    System.out.println("Получили UUID: " +
        returnedBusinessEntity.getBusinessKey());

    System.out.println("Список Web-служб:");

      // Получаем список зарегистрированных
      // Web-служб, чьи имена начинаются с буквы "M".
    BusinessList Ы = proxy.find_business("M", null, 0);

    Vector businessInfoVector =

          bl.getBusinessInfos().getBusinessInfoVector();

    for (int i = 0; i < businessInfoVector.size(); i++){

       BusinessInfo businessInfo =
          (BusinessInfo)businessInfoVector.elementAt(i);
       System.out.println(businessInfo.getNameString());
    }

}catch(UDDIException e){
```

```
        DispositionReport dr = e.getDispositionReport();

        if (dr != null){

            System.out.println(
              "UDDIException faultCode:" + e.getFaultCode() +
              "\n operator:" + dr.getOperator() +
              "\n generic:"  + dr.getGeneric() +
              "\n errno:"    + dr.getErrno() +
              "\n errCode:"  + dr.getErrCode() +
              "\n errInfoText:" + dr.getErrInfoText());
        }
        e.printStackTrace();

      }catch(Exception e){
        System.err.println("From proxy: " + e);
      }
    }
}
```

Система взаимодействия фирм ebXML

Электронный бизнес-реестр ebXML первоначально был создан двумя организациями: центром международной торговли и электронного бизнеса ООН UN/CEFACT (United Nations Centre for Trade Facilitation and Electronic Business) (**http://www.uncefact.org/**) и общественной организацией по стандартизации структурной информации OASIS (Organization for the Advancement of Structured Information Standards) (**http://www.oasis-open.org/**). Затем в разработку проекта ebXML включились другие фирмы, сейчас их уже несколько сотен, в их числе **IBM** и Sun. Всю информацию о сообществе ebXML можно получить на сайте проекта **http://www.ebxml.org/**.

Цель проекта ebXML широка — обеспечить взаимодействие между деловыми партнерами по Интернету, обычно обозначаемое сокращением B2B (Business to Business). Взаимодействие может быть любым, Web-службы — это только частный случай, причем не самый важный. Поэтому во множестве спецификаций проекта ebXML (**http://www.ebxml/org/specs/**) описывается только каркас построения реестра. Спецификация ebXML делает это очень расплывчато, в самых общих терминах.

С точки зрения ebXML есть две фирмы, называемые *сторонами* (parties), осуществляющие деловое сотрудничество (Business Collaboration). В реестре

ebXML хранится информация о каждой стороне и их сотрудничестве, которая разделена на две части: заявление о сотрудничестве CPP (Collaboration Protocol Profile) и соглашение о сотрудничестве CPA (Collaboration Protocol Agreement).

Заявление о сотрудничестве CPP

Заявление о сотрудничестве CPP — это документ **XML,** содержащий информацию о фирме-стороне, предоставляемых ею услугах, возможных соглашениях с другими сторонами, способах обмена услугами и получения информации, в том числе о транспортных протоколах. Заявление **CPP** открыто для всех, кто желает сотрудничать с фирмой. Каждая сторона может зарегистрировать в реестре одно заявление CPP, несколько заявлений, или не регистрировать их вообще.

Структура документа XML, содержащего заявление CPP, такова:

```
<CollaborationProtocolProfile
   xmlns="http://www.ebxml.org/namespaces/tradePartner"
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   version="1.1">

   <PartyInfo>
       <!— Сведения о фирме-стороне —>
   </PartyInfo>

   <Packaging id="ID">
      <!— Описание XML-структуры сообщений —>
   </Packaging>

   <ds:Signature>
      <!— Цифровая подпись (необязательна) —>
   </ds:Signature>

    <Comment>Комментарий (необязателен) </Comment>

</CollaborationProtocolProfile>
```

У каждого элемента этого документа есть вложенные элементы, детально описывающие заявление CPP. Мы не будем сейчас углубляться в них.

Соглашение о сотрудничестве CPA

Соглашение о сотрудничестве CPA — это документ XML, содержащий технические сведения о соглашениях двух сторон. Эти сведения собираются из заявлений CPP двух фирм-сторон, желающих сотрудничать, или составляются фирмой, желающей получить услуги.

Схема документа XML, содержащего соглашение CPA, такова:

```
<CollaborationProtocolAgreement
   xmlns="http://www.ebxml.org/namespaces/tradePartner"
   xmlns:bpm="http://www.ebxml.org/namespaces/businessProcess"
   xmlns:ds = "http://www.w3.org/2000/09/xmldsig#"
   xmlns:xlink = "http://www.w3.org/1999/xlink"
   cpaid="Идентификатор документа"
   version="1.2">

   <!— Состояние документа: proposed, agreed, signed —>
   <Status value = "proposed"/>

   <Start>Дата и время вступления в силу соглашения</Start>

   <End>Дата и время окончания соглашения</End>

   <!— Договоренность о продолжительности переговоров —>
   <ConversationConstraints invocationLimit = "100"
       concurrentConversations = "4"/>

   <PartyInfo>
      <!— Сведения об одной стороне —>
   </PartyInfo>

   <PartyInfo>
       <!— Сведения о другой стороне —>
   </PartyInfo>

   <Packaging id="идентификатор">
      <!— Структура сообщений —>
   </Packaging>

   <ds:Signature>
```

```
        <!-- Цифровая подпись (необязательно) —>
    </ds:Signature>

    <Comment>Комментарий (необязателен)</Comment>

</CollaborationProtocolAgreement>
```

У элементов этого документа есть вложенные элементы весьма сложной структуры, которые мы не будем сейчас рассматривать.

Вся информация, включая CPP и CPA, заносится в *хранилище* (repository) вместе с любой дополнительной информацией.

Во время переговоров и после них стороны обмениваются *сообщениями* (messages).

Соглашение достигается так:

1. Сторона А, оказывающая услуги, составляет заявления CPP и регистрирует CPP в реестре ebXML.
2. Сторона В, желающая получить услуги, отыскивает в реестре подходящее заявление CPP стороны А.
3. Сторона В создает свое заявление CPP, затем на основании обоих заявлений составляет проект соглашения CPA и посылает его стороне А.
4. Стороны достигают соглашения, обмениваясь сообщениями.
5. После достижения соглашения стороны хранят копии CPA на своих серверах и/или в хранилище ebXML.
6. В процессе сотрудничества серверы обеих сторон обмениваются информацией согласно достигнутому соглашению CPA.

Реализация ebXML

В то время, когда пишутся эти строки, еще нет промышленных Java-реализаций реестра ebXML. Фирма Sun готовит пакет сервлетов и компонентов EJB, предварительно названный Sun ebXML Registry/Repository Implementation. Этот пакет устанавливается в сервер приложений iPlanet Application Server. В качестве хранилища пакет использует СУБД Oracle.

Познакомиться с этой разработкой фирмы Sun и получить исходные тексты ее компонентов можно по адресу **http://wwws.sun.com/software/xml/developers/regrep/**.

Другая реализация реестра ebXML готовится в рамках общественного проекта EBXMIRR (OASIS ebXML Registry Reference Implementation Project).

Текущее состояние проекта можно посмотреть на сайте **http://ebxmirr.sourceforge.net/**.

Язык WS-Inspection для поиска Web-служб

Системы UDDI и ebXML решают не только задачу поиска Web-службы, но и задачи их описания, изменения, интеграции. Это усложняет ведение реестра и поиск в нем нужной Web-службы. Если же нужно только отыскать данную Web-службу на конкретном сайте, то лучше воспользоваться другими средствами поиска.

Одно такое средство поиска — WS-Inspection — создано фирмами IBM и Microsoft в 2001 году. Одной из целей создания нового средства было предельное упрощение поиска Web-службы на Web-сайте. При помощи WS-Inspection эта задача решается так: составляется описание Web-службы на специально разработанном языке WSIL (WS-Inspection Language) — еще одной реализации XML. Оно записывается в файл с расширением wsil и заносится в определенный каталог сервера приложений. Сервер, получив запрос, просматривает все файлы с расширением wsil и находит запрошенную Web-службу.

Описание Web-службы на языке WSIL выполняется всего одним документом XML с корневым элементом <inspection>, содержащим несколько вложенных элементов.

В корневой элемент <inspection> вкладывается необязательное краткое описание Web-службы <abstract>, служащее комментарием, и одно или несколько описаний Web-услуг <service> и/или одна или несколько ссылок на описания <link>. В документе обязательно должен присутствовать хотя бы один элемент <service> или <link>.

В элементе <service> тоже может быть краткое описание Web-услуги <abstract>, одно или несколько имен <name> и обязательно одно или несколько описаний <description>, в которых необязательным атрибутом location задаются URI-адреса документов XML, содержащих описание Web-службы. Описание обычно выполняется на языке WSDL, но может быть сделано и на другом языке. Пространство имен описания задается вторым атрибутом элемента <description> — атрибутом referencedNamespace.

В элемент <description> можно вложить описание <abstract>, служащее комментарием, и один произвольный элемент с дополнительным описанием. В этом описании может быть адрес Web-службы — в таком случае оно заменит значение атрибута location.

Элемент <link> внешне отличается от элемента <service> тем, что в нем нет описания <description>, а атрибуты referencedNamespace и location

относятся непосредственно к элементу <link>. Эти атрибуты описывают пространство имен и адрес другого WSIL-описания или хранилища UDDI. Вместо атрибута location можно записать вложенный элемент, в котором указать ссылку на другое описание.

Итак, WSIL-описание строится по следующей схеме:

```
<inspection

   xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">

   <abstract>Краткое описание Web-службы</abstract>

   <service>

      <abstract>Краткое описание Web-услуги</abstract>

      <пате>Произвольное имя услуги</пате>
      <description referencedNamespace="адрес URI"
                   location="адрес URI">

         <abstract>Краткое описание</abstract>

         <!— дополнительное описание —>

      </description>

   </service>

   <link referencedNamespace="адрес URI"
         location="адрес URI">

      <abstract>КраТКое описание</abstract>

      <!— дополнительное описание —>

   </link>

</inspection>
```

В листинге 2.8 показано, как просто выглядит WSIL-описание нашей метеослужбы.

Листинг 2.8. Описание WSIL метеослужбы

```
<?xml version="1.0"?>

<inspection

   xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">

   <service>

      <name>MeteoService</name>
      <description
         referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
         location="http://www.meteo.com/wsdl/MeteoService.wsdl" />

   </service>

</inspection>
```

После того как WSDL-описание сделано, его надо записать в файл с именем, например, `MeteoService.wsil`, а файл поместить в определенный каталог сервера приложений. Если обратиться к серверу с браузера со стороны клиента, например, так: http://www.meteo.com/common/wsil/inspection.wsil, то браузер покажет список всех Web-служб, имеющихся на сервере. Спецификация WS-Inspection предписывает, чтобы в корневом каталоге сайта, реализующего WS-Inspection, располагался или создавался динамически файл inspection.wsil. Вы можете набрать в браузере, например, адрес фирмы **IBM http://www.ibm.com/inspection.wsil** или адрес **http://www.xmethods.com/inspection.wsil** и получить список всех Web-служб, зарегистрированных на этом сайте. При этом файла inspection.wsil на сервере нет, он создается динамически по запросу клиента.

Разумеется, сервер приложений должен "понимать" WS-Inspection. Для этого на нем должна быть сделана реализация спецификации WS-Inspection, например, установлен пакет интерфейсов и классов WSIL4J, входящий в состав **IBM** WSTK.

В пакете WSIL4J разработаны Java-интерфейсы WS-Inspection и сделана их реализация классами Java. Основную работу на сервере выполняет сервлет `WSILServlet`. Он просматривает все файлы с расширением имени wsil и отправляет их содержимое клиенту.

Если у Web-службы нет WSIL-описания, то сервлет создает его динамически.

У клиента работает класс `WSILProxy`. Он запрашивает `WSILServlet`, читает полученные от него WSIL-документы и выбирает ссылку на описание WSDL или другое, какое там есть, описание нужной Web-службы. После этого можно обратиться к найденной Web-службе.

В листинге 2.9 приведен пример поиска нашей метеослужбы посредством WS-Inspection и пакета WSIL4J. Программа просто выводит полученные от сервера описания метеослужбы.

Листинг 2.9. Поиск метеослужбы с помощью WS-Inspection

```
import com.ibm.wsil.*;
import com.ibm.wsil.client.*;
import com.ibm.wstk.wsdl.*;
import java.util.*;

public class MeteoWSInspection{

   public static void main(String[] args){

      if (args.length < 2){
        System.err.println(
          "Usage: MeteoWSInspection <URL> <serviceName>");
        System.exit(1);
      }

      try{
        WSILProxy proxy = new WSILProxy(args[0]);

        WSDLDocument[] wsdlDocs =
            proxy.getWSDLDocumentByServiceName(args[1]);

        for (int k = 0; k < wsdlDocs.length; k++)
            System.out.println(wsdlDocs[k].serializeToXML());

    }catch(Exception e){
       System.err.println(e);
    }
  }
}
```

Пакет JAXR

Итак, уже разработано несколько различных систем регистрации и поиска Web-служб: UDDI, ebXML, WS-Inspection. Есть и другие системы, не указанные в этой книге. Все эти системы требуют разных методов доступа к реестру и работы с ним. Для каждой системы приходится создавать своего клиента, работающего с реестром по правилам данной системы поиска Web-служб.

Фирма Sun решила разработать единую методику доступа к реестрам разных типов и создала набор интерфейсов JAXR (Java API for XML Registries). Этот набор реализован в пакете Sun WSDP. Там же приведен пример клиента JAXR. Это графическая утилита просмотра реестров Registry Browser. Она вызывается из командной строки просто набором имени командного файла

```
$ jaxr-browser
```

В составе WSDP есть и простой тренировочный реестр, реализованный в виде сервлета `RegistryServerServlet`. Реестр работает в Web-контейнере Tomcat и хранит информацию в небольшой базе данных Apache Xindice (произносится, как говорят авторы, "Зин-ди-чи", с итальянским акцентом), тоже входящей в состав WSDP. Для запуска реестра надо запустить Tomcat и стартовать базу данных:

```
$ cd $WSDP_HOME/bin
$ startup
$ xindice-start
```

Для проверки работы реестра и для посылки ему сообщений применяется специальная утилита, работающая из командной строки. Для входа в реестр надо набрать следующую командную строку:

```
$ registry-server-test run-cli-request -Dxml/GetAuthToken.xml
```

Для работы с реестром прямо через базу данных Xindice в составе WSDP есть графическая утилита Indri. Она запускается из командной строки:

```
$ registry-server-test run-indri
```

Мы рассмотрим подробнее работу с реестром в *главе* 5, а пока займемся набором интерфейсов JAXR.

Поскольку набор JAXR рассчитан на работу с реестрами самых разных типов, он содержит только интерфейсы, которые должен реализовать *поставщик услуг* (service provider) конкретного реестра. Интерфейсы будут поразному реализованы поставщиком услуг UDDI и поставщиком услуг ebXML, но клиент этого не замечает, он просто связывается с поставщиком

услуг и пользуется интерфейсами пакета JAXR. Схема обращения клиента к реестру через поставщика услуг показана на рис. 2.4.

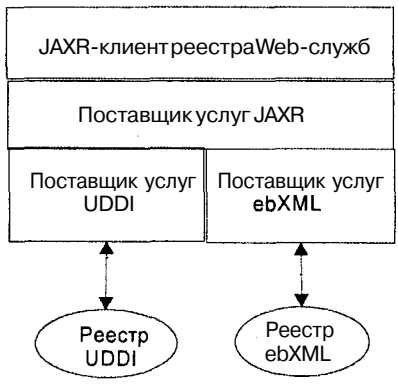


Рис. 2.4. Обращение к реестру Web-служб

Набор JAXR состоит из двух пакетов `javax.xml.registry` и `javax.xml.registry.infomodel`. Интерфейсы первого пакета применяются клиентом для работы с реестром, интерфейсы второго пакета — поставщиком услуг для преобразования информации к виду, пригодному для хранения в базе данных реестра.

Для связи с поставщиком услуг клиент пользуется интерфейсом `Connection`. Экземпляр этого класса клиент получает с помощью класса-фабрики ConnectionFactory. Сначала статическим методом `newInstance()` создается объект этого класса, затем методом setProperties o в него заносятся характеристики реестра, после этого методом `createConnection()` создается объект типа connection.

После того как связь с поставщиком услуг установлена, методом `getRegistryService()` интерфейса Connection получаем объект типа `RegistryService`, в котором сосредоточены методы работы с реестром. Например, методом `getBusinessQueryManager()` можно получить объект типа `BusinessQueryManager`, содержащий методы `findXxx()` получения информации из реестра.

В листинге 2.10 показано, как можно связаться с реестром и получить из него различную информацию с помощью JAXR.

Листинг 2.10. Получение информации из реестра с помощью JAXR

```
import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;
import java.net.*;
```

```
import java.util.*;

public class JAXRClient{

  public static void main(String[] args){

        // Несколько адресов реестров на выбор.
    String queryURL =
      "http://www-3.ibm.com/services/uddi/inquiryapi";
    //"http://uddi.rte.microsoft.com/inquire";
    //"http://localhost:8080/registry-server/RegistryServerServlet";

    if (args.length < 1){
       System.out.println("Usage: java JAXRClient <name>");
       System.exit(1);
    }

    JAXRClient jq = new JAXRClient();

    String httpProxyHost = "localhost";
    String httpProxyPort = "8080";

    Properties props = new Properties();
    props.setProperty("javax.xml.registry.queryManagerURL",
         queryURL);
    props.setProperty("com.sun.xml.registry.http.proxyHost",
         httpProxyHost);
    props.setProperty("com.sun.xml.registry.http.proxyPort",
         httpProxyPort);

    Connection connection = null;

    try{
       ConnectionFactory factory =
          ConnectionFactory.newInstance();

       factory.setProperties(props);
```

```
connection = factory.createConnection();

System.out.println("Связь с реестром установлена");

RegistryService rs = connection.getRegistryService();
BusinessQueryManager bqm = rs.getBusinessQueryManager();

Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);

Collection namePatterns = new ArrayList();
namePatterns.add("%"+args[0]+" % " ) ;

     // Поиск фирмы по имени.
BulkResponse response =
    bqm.findOrganizations(findQualifiers,
         namePatterns, null, null, null, null);

Collection orgs = response.getCollection();

     // Сведения о фирме.
Iterator orgIter = orgs.iterator();
while (orgIter.hasNext()){

   Organization org = (Organization) orgIter.next();

   System.out.println("Название: " +
         org.getName().getValue());

   System.out.println("Описание: " +
         org.getDescription().getValue());

   System.out.println("Идентификатор: " +
         org.getKey().getId());

     // Контактная информация
   User pc = org.getPrimaryContact();
```

```
if (pc != null)(

   PersonName pcName = pc.getPersonName();

   System.out.println("Название: " +
         pcName.getFullName());

   Collection phNums =

      pc.getTelephoneNumbers(null);

   Iterator phIter = phNums.iterator();

   while (phIter.hasNext()){

      TelephoneNumber num =
            (TelephoneNumber)phIter.next();
      System.out.println("Номер телефона: " +
            num.getNumber());
   }
   Collection eAddrs = pc.getEmailAddresses();
   Iterator eaIter = eAddrs.iterator();

   while (eaIter.hasNext()){

      EmailAddress eAd =
            (EmailAddress) eaIter.next();

      System.out.println("E-mail: " +
            eAd.getAddress());
   }
}
      // Услуги и доступ к ним
Collection services = org.getServices();

Iterator svcIter = services.iterator();
```

```
            while (svcIter.hasNext()){

                Service svc = (Service) svcIter.next();

                System.out.println("Название услуги: " +
                      svc.getName().getValue());

                System.out.println("Описание услуги: " +
                     svc.getDescription().getValue());

                Collection serviceBindings =

                      svc.getServiceBindings();

                Iterator sbIter = serviceBindings.iterator();

                while (sbIter.hasNext()){

                    ServiceBinding sb =
                       (ServiceBinding)sbIter.next();

                    System.out.println("Адрес URI: " +
                        sb.getAccessURI());

                }
            }
            System.out.println("------");
        }
    }catch(Exception e){
        e.printStackTrace();
    }finally{

        if (connection != null)
          try{
              connection.close();
          }catch(JAXRException je){}
    }
  }
}
```

Стек протоколов Web Services

Итак, мы описали архитектуру Web-служб и сделали краткий обзор составляющих ее компонентов. Мы увидели, что архитектуру Web-служб составляет масса протоколов и спецификаций. Их можно разбить на четыре части, образующие стек протоколов, в котором каждый верхний уровень опирается на нижний уровень. Основные протоколы этого стека показаны на рис. 2.5.

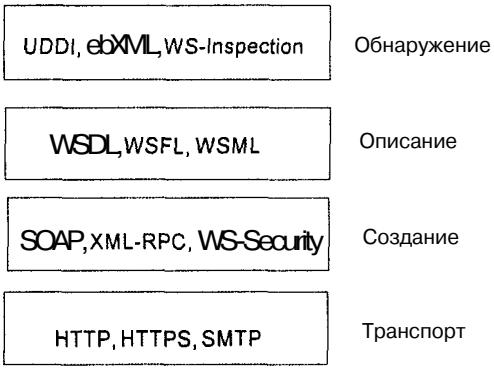


Рис. 2.5. Стек протоколов Web Services

ГЛАВА 3

Протокол SOAP и Web Services

Как уже говорилось в предыдущей главе, Web-службы обмениваются информацией с клиентами и между собой, посылая сообщения на языке XML. Теги этой реализации XML, правила оформления документа XML и порядок обмена документами определены протоколом SOAP. Протокол SOAP создан в 1998 году командой разработчиков под руководством Дейва Винера (Dave Winer), работавшей в корпорации Microsoft и фирме Userland. Название протокола — "Простой протокол доступа к объектам" — отражает его первоначальное назначение — обращаться к методам удаленных объектов. Назначение протокола изменилось, сейчас это протокол всякого взаимодействия Web-служб и компонентов слабо связанных распределенных приложений. Он уже не совсем прост, да и об объектах он ничего не говорит. Многие разработчики предлагают назвать его "Service Oriented Architecture Protocol", оставив прежнее сокращение. Чтобы прекратить эти попытки, в спецификации SOAP 1.2 указано, что слово "SOAP" отныне никак не будет расшифровываться.

В конце 1999 года разработка протокола была передана в консорциум W3C **(http://www.w3.org/)**.

В мае 2000 года консорциум выпустил свою версию SOAP 1.1. Послание, написанное по протоколу SOAP, оформляется документом XML, активно использующим пространства имен. Имена элементов XML версии SOAP 1.1 относятся к пространству имен с идентификатором http://schemas.xmlsoap.org/soap/envelope/.

Черновик второй версии SOAP 1.2 был выпущен в 2001 году, его пространство имен называлось в то время http://www.w3.org/2001/06/soap-envelope.

Заметьте, что именно идентификатор пространства имен, а не номер 1.1 или 1.2 определяет версию SOAP. Сервер не станет рассматривать SOAP-послание и вернет сообщение об ошибке "VersionMismatch", если заметит несоответствие пространства имен.

В то время как я пишу эти строки, версия SOAP 1.1 остается рабочей. Версия 1.2 никак не может выйти из подготовительной стадии, но уже используется, например, в SOAP::Lite, Apache SOAP 2.3, Apache Axis. Поэтому в этой главе я буду излагать версию 1.2, отмечая ее отличия от версии 1.1.

Спецификация рабочей версии SOAP всегда хранится по адресу **http://www.w3.org/TR/SOAP/**. Документы, лежащие по этому адресу, заменяются новыми при замене рабочей версии.

Черновая версия SOAP постоянно обновляется, при этом меняется идентификатор пространства имен. Новейший вариант черновой версии на время написания книги хранился по адресу **http://www.w3.org/TR/soap12-part1/**, а пространство используемых ею имен называлось http://www.w3.org/2002/06/soap-envelope. Заметьте, что спецификация SOAP 1.2 состоит из двух частей: part1 и part2. Вторая часть спецификации — приложение — содержит правила записи сложных типов данных. У спецификации есть еще одна часть partO — примеры посланий, составленных по правилам SOAP 1.2.

Структура SOAP-послания

Спецификация определяет SOAP-послание как документ XML, не содержащий объявление типа документа и инструкций по обработке. Корневой элемент этого документа XML называется `<Envelope>`. У элемента `<Envelope>` могут быть атрибуты `xmlns`, определяющие пространства имен, и другие атрибуты, снабженные префиксами. В корневой элемент вкладывается один необязательный элемент `<Header>`, содержащий заголовок послания, и один обязательный элемент <Body>, в который записывается содержимое послания. Версия 1.1 позволяла после тела <Body> записать произвольные элементы, их имена обязательно следовало снабжать префиксами. Версия 1.2 запрещает писать что-либо после элемента <Body>. Короче говоря, общая структура SOAP-послания такова:

```
<?xml version="1.0" ?>

<env:Envelope
   xmlns:env="http://www.w3.org/2002/06/soap-envelope">

   <env:Header>
      <!— Блоки заголовка —>
   </env:Header>

   <env:Body>
```

```
    <!— Содержимое послания —>
  </env:Body>

</env:Envelope>
```

Элемент <Header>, если он есть в послании, записывается первым в теле элемента `<Envelope>`. Кроме атрибутов `xmlns`, в нем может быть атрибут `actor`, указывающий адресом URI конкретный SOAP-сервер, которому предназначено послание.

Дело в том, что SOAP-послание может пройти через несколько SOAP-серверов или через несколько приложений на одном сервере. Эти приложения выполняют предварительную обработку блоков заголовка послания и передают его друг другу. Все эти серверы и/или приложения называются *SOAP-узлами* (SOAP nodes). Спецификация SOAP не определяет правила прохождения послания по цепочке серверов. Для этого разрабатываются другие протоколы, например, Microsoft WS-Routing.

Атрибут `actor` задает целевой SOAP-узел — тот, который расположен в конце цепочки и будет обрабатывать заголовок полностью. Значение http://schemas.xmlsoap.org/soap/actor/next атрибута `actor` показывает, что обрабатывать заголовок будет первый же сервер, получивший его. Атрибут `actor` может встречаться в отдельных блоках заголовка, указывая узел-обработчик этого блока. После обработки блок удаляется из SOAP-послания.

В версии 1.2 атрибут `actor` заменен атрибутом `role`, потому что в этой версии SOAP каждый узел играет одну или несколько ролей. Спецификация пока определяет три роли SOAP-узла.

- Роль http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver играет конечный, целевой узел, который будет обрабатывать заголовок.
- Роль http://www.w3.org/2002/06/soap-envelope/role/next играет промежуточный или целевой узел. Такой узел может играть и другие, дополнительные роли.
- Роль http://www.w3.org/2002/06/soap-envelope/role/none не должен играть ни один SOAP-узел.

Распределенные приложения, исходя из своих нужд, могут добавить к этим ролям другие роли, например, ввести промежуточный сервер, проверяющий цифровую подпись и определить для него эту роль какой-нибудь строкой URI.

Значением атрибута `role` может быть любая строка URI, показывающая роль узла, которому предназначен данный блок заголовка. Значением по умолчанию для этого атрибута служит пустое значение, то есть, просто пара кавычек, или строка URI http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver.

Значение атрибута role показывает, что блок должен быть обработан узлом, играющим роль, определенную такой же строкой.

Еще один атрибут элемента <Header>, называемый mustUnderstand, принимает значения о или 1. Его значение по умолчанию равно о. Если атрибут mustunderstand равен 1, то SOAP-узел при обработке элемента обязательно должен учитывать его синтаксис, определенный в схеме документа, или совсем не обрабатывать послание. Это повышает точность обработки послания.

В версии SOAP 1.2 вместо цифры о надо писать слово false, а вместо цифры 1 писать слово true.

В тело заголовка <Header> можно вложить произвольные элементы, ранее называвшиеся *статьями* (entries) заголовка. В версии 1.2 они называются *блоками* (blocks) заголовка. Их имена обязательно помечаются префиксами. В блоках заголовка могут встретиться атрибуты role или actor и mustunderstand. Их действие будет относиться только к данному блоку. Это позволяет обрабатывать отдельные блоки заголовка промежуточными SOAP-узлами, теми, чья роль совпадает с ролью, указанной атрибутом role. В листинге 3.1 приведен пример такого блока.

Листинг 3.1. Заголовок с одним блоком

```
<env:Header>

   <t:Transaction
      xmlns:t="http://some.com/transaction"
      env:role=
   "http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver"
      env:mustUnderstand="1">
          5
   </t:Transaction>

</env:Header>
```

Элементы, вложенные в блоки заголовка, уже не называются блоками. Они не могут содержать атрибуты role, actor И mustunderstand.

Элемент <Body> обязательно записывается сразу за элементом <Header>, если он есть в послании, или первым в SOAP-послании, если заголовок отсутствует. В элемент <Body> можно вложить произвольные элементы, спецификация никак не определяет их структуру. Впрочем, определен один элемент <Fault>, содержащий сообщение об ошибке.

Сообщение об ошибке <Fault>

Если SOAP-сервер, обрабатывая поступившее к нему SOAP-послание, заметит ошибку, то он прекратит обработку и отправит клиенту SOAP-послание, в тело которого запишет один элемент `<Fault>` с сообщением об ошибке.

В сообщении, записанном в теле элемента `<Fault>` версии SOAP 1.1 выделяются четыре части, описанные следующими вложенными элементами.

- Код ошибки `<faultcode>` — сообщение, показывающее тип ошибки. Оно предназначено для программы, обрабатывающей ошибки.
- Описание ошибки <faultstring> — словесное описание типа ошибки, предназначенное для человека.
- Место обнаружения ошибки `<faultactor>` — адрес URI сервера, заметившего ошибку. Полезно, когда послание проходит цепочку SOAP-узлов, для уточнения природы ошибки. Промежуточные SOAP-узлы обязательно записывают этот элемент, целевой SOAP-сервер не обязан это делать.
- Детали ошибки `<detail>` — описывают ошибки, встреченные в теле <Body> послания, но не в его заголовке. Если при обработке тела ошибки не обнаружены, то этот элемент отсутствует.

Например:

```
<env:Envelope

   xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">

   <env:Body>

      <env:Fault>

         <faultcode>env:MustUnderstand</faultcode>

         <faultstring>SOAP Must Understand Error</faultstring>

      </env:Fault>

   </env:Body>

</env:Envelope>
```

Версия SOAP 1.2 изменила содержание элемента <Fault>. Как описано в пространстве имен http://www.w3.org/2002/06/soap-envelope, в него входят два обязательных элемента и три необязательных элемента.

Обязательные элементы.

- Код ошибки <code>. Он содержит обязательный вложенный элемент <Value> с кодом ошибки и необязательный вложенный элемент <subcode>, содержащий, опять-таки, элемент <Value> с уточняющим кодом ошибки и элемент <subcode>, и далее все повторяется рекурсивно.
- Причина ошибки <Reason>. Содержит необязательный атрибут xml:lang, указывающий язык сообщения (см. *главу Г),* и произвольное число вложенных элементов с описанием ошибки.

Необязательные элементы.

- <Node> — адрес URI промежуточного SOAP-узла, заметившего ошибку.
- <Role> — роль SOAP-узла, заметившего ошибку.
- <Detail> — описание ошибки, замеченной при обработке тела <Body> послания, но не его заголовка.

Листинг 3.2 показывает сообщение об ошибке, возникшей при попытке выполнения процедуры. Ошибка заключается в том, что имена аргументов процедуры неправильно записаны в SOAP-послании и процедура не может их понять.

Листинг 3.2. Сообщение об ошибке

```
<?xml version='1.0' ?>

<env:Envelope
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:rpc='http://www.w3.org/2002/06/soap-rpc'>

   <env:Body>

      <env:Fault>

         <env:Code>

            <env:Value>env:Sender</env:Value>

            <env:Subcode>
```

```
            <env:Value>rpc:BadArguments</env:Value>
         </env:Subcode>

      </env:Code>

      <env:Reason>Processing Error</env:Reason>

      <env:Detail>

         <e:myfaultdetails

            xmlns:e="http://www.example.org/faults">

            <message>Name does not match</message>

            <errorcode>999</errorcode>

         </e:myfaultdetails>

      </env:Detail>

   </env:Fault>

</env:Body>

</env:Envelope>
```

Типы ошибок

Список кодов ошибок постоянно меняется и расширяется. Версия 1.1 определяет четыре типа ошибок.

- `VersionMismatch` — пространство имен неопознано. Может быть, оно устарело или его имя написано неправильно.
- `MustUnderstand` — блок заголовка, помеченный атрибутом `mustUnderstand` со значением 1, не отвечает своему синтаксису, определенному в схеме документа.
- `client` — документ XML, содержащий послание, неправильно сформирован и по этой причине сервер не может его обработать. Клиенту следует изменить послание.

- `Server` — сервер не может обработать правильно записанное послание по своим внутренним причинам.

Версия 1.2 определяет пять типов ошибок.

- `VersionMismatch` — пространство имен неопознано. Может быть, оно устарело или его название написано неправильно, или в послании встретилось имя элемента XML, не определенное в этом пространстве имен. В заголовок ответа сервер записывает элемент <upgrade>, перечисляющий вложенными элементами `<envelope>` правильные названия пространств имен, понимаемые сервером. Ответ сервера показан в листинге 3.3.

- `MustUnderstand` — блок заголовка, помеченный атрибутом `mustUnderstand` со значением `true`, не отвечает своему синтаксису, определенному в схеме документа. Сервер записывает в заголовок ответа элементы <Misunderstood>, атрибут qname которых содержит имя неправильного блока. Листинг 3.4 содержит пример ответа, который будет сделан сервером, если заголовок листинга 3.1 окажется неправильно записанным.

- DataEncodingUnknown — в послании встретились непонятные данные, может быть, они записаны в неизвестной кодировке.

- `Sender` — документ XML, содержащий послание, неправильно сформирован и по этой причине сервер не может его обработать. Клиенту следует изменить послание.

- Receiver — сервер не может обработать правильно записанное послание по своим внутренним причинам, например, отсутствует нужный XML-парсер.

Сервер может добавить к этим типам ошибок какие-то свои типы. Обычно они детализируют стандартные типы, и сообщения о них появляются в элементах <subcode>, как было показано выше в листинге 3.2.

Листинг 3.3. Ответ сервера с сообщением об ошибке типа VersionMismatch

```
<?xml version="1.0" ?>

<env:Envelope

   xmlns:env="http://www.w3.org/2002/06/soap-envelope">

   <env:Header>

      <upg:Upgrade
```

```
        xmlns:upg="http://www.w3.org/2002/06/soap-upgrade">

      <envelope qname="ns1:Envelope"
        xmlns:ns1="http://www.w3.org/2002/06/soap-envelope"/>

      <envelope qname="ns2:Envelope"

        xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"/>

      </upg:Upgrade>

  </env:Header>

  <env:Body>

   • <env:Fault>

      <env:Code>
        <env:Value>env:VersionMismatch</env:Value>
      </env:Code>

      <env:Reason>Version Mismatch</env:Reason>

      </env:Fault>

   </env:Body>

</env:Envelope>
```

Листинг 3.4. Ответ сервера с сообщением об ошибке типа MustUnderstand

```
<?xml version="1.0" ?>

<env:Envelope xmlns:env='http://www.w3.org/2002/06/soap-envelope'
              xmlns:flt='http://www.w3.org/2002/06/soap-faults' >

   <env:Header>

      <flt:Misunderstood qname='t:Transaction'
                  xmlns:t='http://some.com/transaction' />
```

```
    </env:Header>

    <env:Body>

        <env:Fault>

            <env:Code>
                <env:Value>env:MustUnderstand</env:Value>
            </env:Code>

            <env:Reason>
                One or more mandatory headers not understood
            </env:Reason>

        </env:Fault>

    </env:Body>

</env:Envelope>
```

Типы данных SOAP

В SOAP-посланиях передаются данные самых разных типов: числа, даты, строки символов, массивы, структуры. Определение типов этих данных выполняется, как обычно, в схемах XML. Схема может быть записана любым способом, но чаще всего применяется язык XSD, который мы разобрали в *главе 1*. Типы, определенные в схеме, заносятся в пространство имен, идентификатор которого служит значением атрибута `encodingStyle`. Атрибут encodingstyle может появиться в любом элементе SOAP-послания, но версия SOAP 1.2 запрещает его появление в корневом элементе `<Envelope>`. Указанное атрибутом `encodingStyle` пространство имен будет известно в том элементе, в котором записан атрибут, и во всех вложенных в него элементах. Конечно, какие-то из вложенных элементов могут изменить пространство имен своим атрибутом encodingstyle.

Стандартное пространство имен, в котором расположены имена типов данных SOAP 1.1, называется http://schemas.xmlsoap.org/soap/encoding/. У текущей черновой версии SOAP 1.2 идентификатор пространства имен — http://www.w3.org/2002/06/soap-encoding. Идентификатор того или иного пространства имен, в котором определены типы данных, обычно получает

префикс enc или SOAP-ENC. Вот обычное описание наиболее часто применяемых пространств имен:

```
<someelem
   xmlns:enc="http://www.w3.org/2002/06/soap-encoding"
   env:encodingStyle="http://www.w3.org/2002/06/soap-encoding"
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Ниже мы будем считать, что эти описания уже сделаны. Для краткости будем называть пространство имен типов данных SOAP просто enc.

В основном типы данных, определенные в пространстве имен enc, совпадают с типами данных языка XSD, описанными в *главе 1.* Это не случайно. Создатели протокола SOAP намеревались просто взять имена типов языка XSD, долго ждали окончания работы над этим языком, не дождались — и определили свое пространство имен, включив в него черновую версию языка XSD. В частности, полностью совпадают простые типы: числа, даты, строки, логические значения.

Интересно, что в пространстве имен enc описаны специальные элементы XML, типы которых — простые типы XSD. Имена этих элементов совпадают с именами соответствующих простых типов XSD. Например, тип элемента <enc:int> — это тип целых чисел int, определенный в языке XSD. Элемент можно использовать так:

```
<enc:int>37</enc:int>
```

У этого элемента фактически нет имени, он только показывает, что в его теле следует записывать целое число. Такие элементы удобно использовать в массивах. Для того чтобы на такой элемент можно было сослаться, определен атрибут id:

```
<enc:int  id="age">37</enc:int>
```

Атрибут id, определенный вместе с элементом <enc:int>, задает имя элемента, по которому можно сослаться на элемент атрибутом ref следующим образом:

```
<personAge  ref="#age"  />
```

Знак номера # означает ссылку на имя, определенное в другом элементе.

Атрибут ref можно использовать, как в языке HTML, для ссылок на другие документы. Например:

```
<person>

   <name>Иванов</name>
```

```
    <address ref="http://some.com/addr.xml#Ivanov" />

</person>
```

В версии SOAP 1.1 атрибут ref назывался href.

Разумеется, для указания типа элемента можно использовать обычную для документов XML конструкцию `xsi:type` (см. *главу 1*), например, определить элемент

```
<age xsi:type="xsd:int">37</age>
```

или передать комплексное число `2.25+3.46i` так:

```
<arg0 xsi:type="complex">

  <re xsi:type="xsd:double">2.25</re>
  <im xsi:type="xsd:double">3.46</im>

</arg0>
```

Элементы XML, соответствующие простым типам XSD, удобно применять для описания элементов массивов.

Массивы

В пространстве имен http://schemas.xmlsoap.org/soap/encoding/ определен тип массива Array. Этот тип не определяет ни количество элементов массива, ни их тип. Все это надо уточнить позднее. В схеме, описывающей SOAP-послание, можно описать элемент типа массив, например, так:

```
<element name="arr" type="enc:Array" />
```

После этого в SOAP-послании можно определить конкретный массив:

```
<arr enc:arrayType="xsd:string[3]">

    <item>Иванов</item>
    <item>Петров</item>
    <item>Сидоров</item>

</arr>
```

Можно сделать все определение целиком прямо в SOAP-послании:

```
<arr xsi:type="enc:Array" enc:arrayType="xsd:string[3]">

    <item>Иванов</item>
    <item>Петров</item>
```

```
    <item>Сидоров</item>

</arr>
```

Как видно из этих примеров, при определении массива атрибутом `arrayType` надо задать тип массива и количество его элементов. В одном массиве можно записать элементы разных типов. Типы элементов массива должны быть подтипами типа массива или совпадать с ним. Тип массива может быть любым, в том числе снова массивом, что отмечается квадратными скобками, например, тип "массив из пяти целых чисел" записывается как `int[5]`, тип "неограниченный двумерный массив целых чисел" — `int[,]` и так далее. Вот определение массива из десяти элементов, каждый элемент которого — двумерный массив целых чисел, число строк и столбцов которого не определено:

```
<tabl xsi:type="enc:Array" enc:arrayType="xsd:int[,] [10]">
```

А вот массив, число элементов которого неизвестно, но каждый элемент — массив из двух строк:

```
<list xsi:type="enc:Array" enc:arrayType="xsd:string[2][]">
```

Если все размерности массива уже известны, то их можно записать в определении массива:

```
<tabl xsi:type="enc:Array" enc:arrayType="xsd:int[15,10]">
```

Элементы многомерных массивов записываются построчно: сначала перечисляются элементы первой строки, потом второй и так далее.

Имена элементов массива (в примере, приведенном выше, повторяется имя item), не имеют никакого значения. Поэтому удобно применять элементы XML, соответствующие простым типам XSD, тем более, что элементы одного и того же массива могут быть разных типов. Например:

```
<addr enc:arrayType="xsd:anySimpleType[2]">

    <enc:string>Яблочная, 23</enc:string>
    <enc:int>234765</enc:int>

</addr>
```

При разборе массива его элементы отыскиваются по индексу. Начальный индекс равен нулю. Впрочем, в версии SOAP 1.1 допускаются массивы, индексы которых начинаются с положительного числа. Такие массивы определяются с атрибутом `offset` следующим образом:

```
<arr xsi:type="env:Array"
       enc:arrayType="xsd:string[5]"  enc:offset="[2]">
```

```
    <item>Элемент с индексом 3</item>
    <item>Элемент с индексом 4</item>

</arr>
```

В версии SOAP 1.1 допускаются даже массивы, не все элементы которых определены ("массивы с дырками"). Скажем, мы можем определить массив из 10 элементов, но задать в нем только элементы с индексами 2 и 4. Это выполняется атрибутом position, например, определяем двумерный массив из 100 элементов, но записываем только два элемента:

```
<arr xsi:type="enc:Array"  enc:arrayType="xsd:string[10,10]">

<item enc:position="[2,5]">Элемент с индексами [2,5]</item>
<item enc:position="[5,3]">Элемент с индексами [5,3]</item>

</arr>
```

Версия SOAP 1.2 запрещает эти вольности, поскольку они сильно затрудняют разбор массива сервером. В этой версии индексы массива всегда начинаются с нуля и все элементы массива должны быть записаны явно.

Структуры

Структуры определяются гораздо проще массивов. Поля структуры записываются как элементы XML, вложенные в структуру. Они различаются по именам своих элементов XML. Все имена, разумеется, должны быть различны. Например:

```
<person>

    <name>Иван</name>
    <secondname>Петрович</secondname>
    <lastname>Сидоров</lastname>

</person>
```

Типы полей структуры могут быть какими угодно, в частности, снова структурами или массивами. Например, следующий элемент <Book> будет структурой, поле <authors> которой — массив:

```
<Book>

    <title>Золотой теленок</title>
```

```
    <authors>

        <пате>Илья Ильф</name>
        <name>Евгений Петров</name>

    </authors>

</Book>
```

но следующий элемент <aBook>:

```
<aBook>

    <title>Золотой теленок</title>
    <author>Илья Ильф</author>
    <author>Евгений Петров</author>

</aBook>
```

не будет структурой, поскольку у вложенных в него элементов совпадают имена. Это просто элемент XML какого-то сложного типа.

Введение новых типов

Итак, в пространстве имен епс уже определено множество простых типов, совпадающих с простыми типами языка XSD, и два сложных типа — массивы и структуры. Этих типов обычно хватает для работы с большинством Web-служб. При необходимости ввести новые типы, следует определить их в какой-либо схеме, например, на языке XSD. Их имена надо записать в новое пространство имен, а идентификатор этого пространства имен сделать значением атрибута `encodingStyle`.

В запросах клиента и ответах сервера новые типы обычно выражаются классами Java. Преобразование классов Java в типы, определенные схемой XML, называется в протоколе SOAP *сериализацией* (serialization), обратное преобразование называется *десериализацией* (deserialization). После определения новых типов и классов Java, надо разработать методы сериализации и десериализации этих типов и классы, содержащие их. Затем клиенту и серверу SOAP надо указать соответствие классов Java определенным типам XML. Это выполняется разными способами, чаще всего пишутся конфигурационные файлы или описание на языке WSDL.

В результате получится специальный программный код, который надо переносить на все Web-службы, использующие новые типы, а также встраи-

вать во все клиентские приложения. Это противоречит основному принципу Web Services — повсеместной доступности Web-услуг. Поэтому не стоит увлекаться введением новых типов данных в свои Web-службы.

Процедурный стиль SOAP

Послание, предназначенное для вызова удаленной процедуры или обращения к методу удаленного объекта, оформляется очень просто.

В тело <Body> послания помещается одна структура, имя которой совпадает с именем вызываемой процедуры или метода. В полях структуры перечисляются аргументы процедуры или метода. Имена полей совпадают с именами аргументов, порядок следования полей в структуре совпадает с порядком следования аргументов в заголовке процедуры или метода.

Версия SOAP 1.2 позволяет вместо структуры записать массив, имя которого совпадает с именем вызываемой процедуры или метода, а элементы содержат значения аргументов процедуры или метода, причем элементы массива должны строго соблюдать порядок следования аргументов в процедуре или методе.

Пусть, например, SOAP-послание хочет обратиться к методу

```
public String getAddress(String name, int number);
```

следующим образом:

```
String address = getId("Иванов", 2345678);
```

Послание будет выглядеть так:

Листинг 3.5. SOAP-запрос процедуры

```
<env:Envelope
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <env:Body>

      <getAddress xmlns="http://www.gts.com/inquiries"
         env:encodingStyle="http://www.w3.org/2002/06/soap-encoding">

         <name xsi:type="xsd:string">Иванов</name>
```

```
        <number xsi:type="xsd:int">2345678</number>
      </getAddress>

   </env:Body>

</env:Envelope>
```

В версии SOAP 1.1 ответ сервера записывается в теле <Body> одной структурой. Имя структуры не имеет значения, но по общему соглашению обычно повторяется имя процедуры с добавлением слова "Response". Для каждого выходного аргумента процедуры записывается поле структуры, содержащее значение этого аргумента. Имя поля совпадает с именем аргумента. Возвращаемое значение процедуры или метода заносится в первое поле этой структуры. Имя этого первого поля не имеет значения.

Листинг 3.6 показывает ответ сервера, содержащий результат обращения к методу getAddress () В версии SOAP 1.1.

Листинг 3.6. Ответ сервера с результатом работы процедуры. Версия SOAP 1.1

```
<env:Envelope
   xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <env:Body>

      <getAddressResponse env:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/">

         <result xsi:type="xsd:string">Садовая, 25-4</result>

      </getAddressResponse>

   </env:Body>

</env:Envelope>
```

В версии SOAP 1.2 возвращаемое значение процедуры или метода записывается по-другому. Сначала определяется имя элемента, в тело которого будет помещено возвращаемое значение. Имя элемента определяется в теле элемента <rpc: result>, который определен в пространстве имен с идентификатором

фикатором http://www.w3.org/2002/06/soap-rpc. Этому идентификатору обычно дается префикс rpc. Например:

```
<rpc:result>address</rpc:result>
<address>Садовая, 25-4</address>
```

Смысл такого двухступенчатого определения в том, чтобы дать возможность проверки типа возвращаемого значения.

Результат работы процедуры или метода в версии SOAP 1.2 можно записать не только структурой, но и массивом. При этом возвращаемое значение должно быть первым элементом массива, следующие элементы должны содержать значения выходных аргументов процедуры в порядке их расположения в заголовке процедуры.

Листинг 3.7 показывает ответ сервера с результатом работы процедуры getAddress O В версии SOAP 1.2.

Листинг 3.7. Ответ сервера с результатом работы процедуры. Версия SOAP 1.2

```
<env:Envelope
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <env:Body>

      <getAddressResponse
          xmlns:rpc="http://www.w3.org/2002/06/soap-rpc"
          env:encodingStyle="http://www.w3.org/2002/06/soap-encoding">

         <rpc:result>address</rpc:result>

         <address> Садовая, 25-4</address>

      </getAddressResponse>

   </env:Body>

</env:Envelope>
```

Сложные аргументы и результаты

Аргументы процедуры или метода и их возвращаемые значения, имеющие сложные типы, следует записывать при пересылке массивами или

структурами. Пусть, например, в Web-службе есть класс, описывающий человека:

```
public class Person)

   String name; // Имя.
   int age; // Возраст.

   public Person(){}

   public Person(String name, int age){
      this.name = name;
      this.age = age;
   }

   public String getName(){ return name; }
   public void setName(String name){ this.name = name; }

   public int getAge(){ return age; }
   public void setAge(int age){ this.age = age; }
}
```

Этот класс используется в описании сотрудника:

```
public class Employee{

   Person pers; // Имя и возраст сотрудника.
   int numb; // Табельный номер сотрудника.

   public Employee(){}

   public Person getPerson(){ return pers; }
   public void setPerson(Person pers){ this.pers = pers; }

   public int getNumb(){ return numb; }
   public void setNumb(int numb)) this.numb = numb; }
}
```

Допустим, что в этой Web-службе, в классе `Info`, есть метод, возвращающий список табельных номеров сотрудников, составленный по их имени и возрасту:

```
public int[] getClockNumbers(Person person);
```

Обращение клиента к методу getClockNumbers о будет выглядеть так:

```
Person ivanov = new Person("Иванов", "27");
Info inf = new Info();
int[] numbs = inf.getClockNumbers(ivanov);
```

Сложный аргумент ivanov можно передать структурой, как показано в листинге 3.9.

Листинг 3.9. Передача объекта структурой

```
<?xml version='1.0' ?>

<env:Envelope
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <env:Body>

      <getClockNumbers env:encodingStyle=
            "http://www.w3.org/2002/06/soap-encoding">

       <person
          xmlns:enc="http://www.w3.org/2002/06/soap-encoding">

            <name xsi:type="xsd:string">Иванов</name>
            <age xsi:type="xsd:int">27</age>

       </person>

      </getClockNumbers>

   </env:Body>

</env:Envelope>
```

Можно передать аргумент ivanov и массивом, как показано в листинге 3.10.

Листинг 3.10. Передача объекта массивом

```
<?xml version='1.0' ?>

<env:Envelope
```

```
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <env:Body>

      <getClockNumbers
         env:encodingStyle=
               "http://www.w3.org/2002/06/soap-encoding">

       <params
         xmlns:enc="http://www.w3.org/2002/06/soap-encoding"
         xsi:type="enc:Array"
         enc:arrayType="xsd:anyType[2]">

            <item xsi:type="xsd:string">Иванов</item>
            <item xsi:type="xsd:int">27</item>

       </params >

      </getClockNumbers>

   </env:Body>

</env:Envelope>
```

Ответ сервера с результатом выполнения метода `getClockNumbers()`, оформленным в виде массива, показан в листинге 3.11.

Листинг 3.11. Результат

```
<?xml version='1.0' ?>

<env:Envelope
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <env:Body>

      <getClockNumbersResponse
```

```
        env:encodingStyle=
             "http://www.w3.org/2002/06/soap-encoding">

    <return
      xmlns:enc="http://www.w3.org/2002/06/soap-encoding/"
      xsi:type="enc:Array" enc:arrayType="xsd:int[4]">

        <item xsi:type="xsd:int">678456</item>
        <item xsi:type="xsd:int">256830</item>
        <item xsi:type="xsd:int">132587</item>
        <item xsi:type="xsd:int">563904</item>

    </return>

    </getClockNumbersResponse>

  </env:Body>

</env:Envelope>
```

Пересылка послания по протоколу HTTP

Итак, SOAP-послание написано и готово к пересылке. Поскольку это простой текст, его можно пересылать по любому протоколу прикладного уровня: HTTP, SMTP, FTP, лишь бы он не исказил послание. Поэтому спецификация SOAP не указывает какой-то определенный протокол и не ограничивает их список. В следующих двух разделах этой главе мы приведем в качестве примера модель оформления SOAP-послания в виде почтового сообщения с одной и с несколькими частями.

Тем не менее, SOAP-послания применяются в Web-технологии и поэтому наиболее часто передаются по наиболее распространенному в WWW протоколу HTTP. При этом для посылки запроса применяется метод POST или GET. Спецификация протокола SOAP все-таки задает некоторые правила оформления POST-запроса.

В версии SOAP 1.2 в POST-запросе в поле Content-Type заголовка записывается MIME-тип содержимого application и специально созданный и зарегистрированный МШЕ-подтип `soap+xml`. Короче говоря, в версии SOAP 1.2 HTTP-запрос выглядит так:

```
POST /services/InfoServlet HTTP/1.1
Host: some.com
```

```
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>

<env:Envelope

   xmlns:env="http://www.w3.org/2002/06/soap-envelope">

   <!—SOAP-послание —>

</env:Envelope>
```

Конечно, в HTTP-заголовке могут быть и другие поля, определенные протоколом HTTP.

В версии SOAP 1.1 сохранялся старый тип содержимого `text/xml` и присутствовало нестандартное для HTTP поле soAPAction заголовка, в котором строкой URI указывался адресат SOAP-послания, например:

```
POST /services/InfoServlet HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://some.org/abc#MyMessage"

<?xml version='1.0' ?>

<env:Envelope
   xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">

   <!—SOAP-послание —>

</env:Envelope>
```

Строка URI, записанная в поле `SOAPAction` заголовка, предназначалась промежуточным серверам или прокси-серверам, не просматривающим тело HTTP-запроса. Пустое значение заголовка, то есть пара кавычек, означало, что адресатом будет строка URI, указанная в заголовке HTTP-запроса после слова POST. Отсутствие поля `SOAPAction` означало, что адресат не указан.

Версия SOAP 1.2 отказалась от поля `SOAPAction`, но, на всякий случай, добавила необязательный атрибут action к полю HTTP-заголовка content-Type, имеющий тот же смысл, что и поле SOAPAction. Например:

```
Content-Type: application/soap+xml; \
   charset="utf-8"; action="http://some.org/abc#MyMessage"
```

Использование метода GET

Версия протокола SOAP 1.2 позволяет сделать запрос к SOAP-серверу по HTTP-методу GET. GET-запрос состоит только из заголовка, у него нет тела, следовательно, он не может передать SOAP-сообщение. Единственное, что он может сделать — это запросить у SOAP-сервера какое-то SOAP-послание, которое можно определить параметрами, переданными в GET-запросе. Например, можно передать аргументы name и age метода getClockNumbers(), приведенного выше в качестве примера, в заголовке GET-запроса:

```
GET /services/InfoServlet?name=Иванов&age=27 HTTP/1.1
Host: some.com
Accept: text/html, application/soap+xml
```

В GET-запросе, в поле заголовка Accept, важно указать тип application/soap+xml ожидаемого SOAP-послания.

Более того, спецификация SOAP 1.2 рекомендует простые аргументы SOAP-запроса дублировать в заголовке HTTP-запроса даже при пересылке SOAP-запроса методом POST.

Листинг 3.12 показывает обращение к процедуре getciockNumbers о с учетом этой рекомендации.

Листинг 3.12. Полный SOAP-запрос по методу POST

```
POST /services/InfoServlet?name=Иванов&age=27 HTTP/1.1
Host: some.com
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: 553

<?xml version='1.0' ?>

<env:Envelope
   xmlns:env="http://www.w3.org/2002/06/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <env:Body>

      <getClockNumbers
         env:encodingStyle=
```

```
            "http://www.w3.org/2002/06/soap-encoding">

        <person
           xmlns:enc="http://www.w3.org/2002/06/soap-encoding">

            <name xsi:type="xsd:string">Иванов</name>
            <age xsi:type="xsd:int">27</age>

        </person>

      </getClockNumbers>

   </env:Body>

</env:Envelope>
```

В любом случае, ответ сервера обычен:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>

<env:Envelope

   xmlns:env="http://www.w3.org/2002/06/soap-envelope" >

   <!- SOAP-ответ сервера ->

</env:Envelope>
```

В случае ошибки при обработке SOAP-послания Web-сервер должен дать КОД ответа не 200 ОК, а 400 Bad Request ИДИ 500 Internal Server Error. Код 400 записывается при возникновении SOAP-ошибки "sender", а код 500 – при появлении ошибок "VersionMismatch", "MustUnderstand", "Receiver".

Пересылка послания по протоколу SMTP

Правила пересылки SOAP-послания версии 1.2 по почтовому протоколу, описываются спецификацией "SOAP 1.2 по Email" ("SOAP Version 1.2 Email

Binding") **(http://www.w3.org/TR/soapl2-email/).** Впрочем, эти правила не добавляют ничего нового ни к правилам оформления SOAP-послания, ни к правилам оформления почтового сообщения, а только фиксируют их. Просто SOAP-послание служит телом обычного почтового сообщения. Вот пример почтового сообщения, содержащего SOAP-послание:

```
From: ivan@some.com
To: petr@another.com
Subject: Some Message
Date: Mon, 18 Nov 2002 13:20:00 GMT
Message-Id: <EE4 92E16A09009027 6D208424 960C0C@some.com>

<?xml version='1.0' ?>
<env:Envelope
xmlns:env="http://www.w3.org/2002/06/soap-envelope">

... Здесь SOAP-послание самого обычного вида...
</env:Envelope>
```

SOAP-послание с дополнениями

Компоненты распределенного приложения часто должны обмениваться не только текстовыми SOAP-посланиями, но и графическими документами: изображениями, схемами, чертежами, рукописями. Такие документы обычно оформляются в бинарных форматах GIF, JPEG, PDF. Метод POST протокола HTTP может передавать не только текстовую, но и самую разнообразную информацию, определяемую MIME-типами text, image, audio, video, application, multipart, message. Остается только совместить передачу SOAP-посланий и бинарной информации. Для этого удобен MIME-тип multipart, введенный рекомендацией RFC 2046.

MIME-тип multipart/related

Тип multipart объединяет несколько сообщений разных типов, каким-то образом связанных между собой, в одно целое сообщение. Например, можно переслать фильм и звук к нему в одном HTTP-сообщении. Сообщение типа multipart начинается с заголовка, в котором, в частности, определяется *разделитель* отдельных частей сообщения. После заголовка и пустой строки на отдельной строке записывается разделитель, после него первая часть сообщения, состоящая из своего заголовка, пустой строки и тела первой части. Потом на отдельной строке записывается разделитель, после него

вторая часть сообщения, опять состоящая из заголовка, пустой строки и тела и так далее. В конце всего сообщения опять ставится разделитель.

Пустая строка создается двумя ASCII-символами CRLF, на языках C/C++, Java они записываются как "\r\n".

Разделитель определяется обязательным параметром boundary поля Content-Type HTTP-заголовка и представляет собой любой набор символов, не совпадающий с начальными символами ни одной строки сообщения. Например:

Content-Type: multipart/related; boundary=vT45fke0nS34x

В строке, отделяющей одну часть от другой, перед разделителем записывается два дефиса:

--vT45fke0nS34x

В конце разделителя, завершающего все сообщение, ставятся еще два дефиса:

--vT45fke0nS34x--

Заголовки отдельных частей необязательны, каждая часть может начинаться с пустой строки. В заголовках важно только поле Content-Type, отсутствие которого означает

Content-Type: text/plain; charset="US-ASCII"

В зависимости от степени связанности сообщений в типе multipart различают несколько подтипов: alternative, parallel, related. Для технологии Web Services наиболее удобен подтип related.

Тип multipart/related описан в рекомендации RFC 2387. Это тип сообщения, состоящего из тесно связанных частей. В сообщении выделяется одна *начальная* (start) часть, с нее начинается обработка полученного сообщения. На начальную часть указывает необязательный параметр start поля content-Type заголовка MIME-сообщения. Если параметр start отсутствует, то начальной считается первая часть сообщения. Тип содержимого начальной части записывается в обязательном параметре type поля content-Type. Например:

```
Content-Type: multipart/related;
    boundary=delimeter;
    start="<ivan1206@some.com>";
    type=text/xml;
    start-info="-o -df type"
```

Напомним, что параметры поля заголовка не должны разделяться символами перевода строки, здесь это сделано только для наглядности.

В технологии Web Services в начальной части записывается SOAP-послание, остальные части содержат дополнительную информацию.

Заголовок каждой части сообщения типа multipart/related снабжается полем content-ID, содержащим идентификатор этой части, уникальный в пределах всего Интернета. Он описан рекомендациями RFC 1521, RFC 1873 и RFC 2045 как уникальный адрес электронной почты, заключенный в угловые скобки. Например:

```
Content-ID: <ivan1206@some.com>
```

Идентификатор начальной части сообщения используется в параметре start для ссылки на нее. В начальной части записываются ссылки на идентификаторы других частей сообщения.

Еще один необязательный параметр start-info поля Content-Type содержит дополнительные сведения о начальной части сообщения, предназначенные программе, которая будет обрабатывать начальную часть. Например, это могут быть аргументы командной строки программы-обработчика.

В заголовке каждой части сообщения может быть поле Content-Disposition, описанное рекомендацией RFC 1806. Оно указывает, встроенная это часть, inline, или присоединенная, attachment. Любая часть сообщения может не находиться непосредственно в сообщении, а быть записанной в файл. В таком случае параметр поля filename содержит имя файла, содержащего данную часть сообщения. Например:

```
Content-Disposition: attachment; filename=myface.gif
```

Вот и все описание MIME-типа multipart/related. Приведем пример сообщения этого типа, состоящего из двух частей: HTML-текста и изображения в формате GIF.

```
MIME-Version: 1.0
Content-Type: multipart/related; boundary=delimeter; type=text/html

--delimeter
Content-Type: text/html; charset="windows-1251"

<html><body><p>
Какой-то текст, затем показана картинка,
<img src="cid:myface2409@some.net">
расположенная в другой части сообщения.
</body></html>
--delimeter
Content-ID: myface2409@some.net
Content-Type: image/gif
Content-Transfer-Encoding: base64
```

```
R0lGODlhGAGgAPEAAP/////ZRaCgoAAAACH+PUNvcHlyaWdodCAoQykgMTk5
NSBJRVRGLiBVbmF1dGhvcml6ZWQgZHVwbGljYXRpb24gcHJvaGliaXRlZC4A
 ... дальнейший код ...

--delimeter--
```

В этом примере, в HTML-теге <img>, для ссылки на вторую часть сообщения применена разновидность URL со схемой cid: (Content-ID), специально разработанной для ссылок на другие части сообщения в рекомендации RFC 2111. После символов cid: просто записывается значение поля `Content-ID` без угловых скобок.

Рекомендация RFC 2557 добавила необязательное поле `Content-Location`, которое можно поместить в заголовок каждой части сообщения. Его значение — строка URL, идентифицирующая данную часть сообщения так же, как И ПОЧТОВЫЙ адрес В Поле заголовка Content-ID. Поле Content-Location можно использовать вместе или вместо поля `Content-ID`. Его отличие от поля `Content-ID` заключается в том, что его значение — URL, а не адрес электронной почты. Следовательно, при ссылке на него не надо применять схему cid. Вот ТОТ же пример, С добавлением ПОЛЯ Content-Location И ссылки на него.

```
MIME-Version: 1.0
Content-Type: multipart/related; boundary=delimeter; type=text/html

--delimeter
Content-Type: text/html; charset="windows-1251"

<html><body><p>
Какой-то текст, затем показана картинка,
<img src="cid:myface2409@some.net">
<img src="images/myface2409.gif">
расположенная в другой части сообщения.
</body></html>
--delimeter
Content-ID: myface2409@some.net
Content-Location: images/myface2409.gif
Content-Type: image/gif
Content-Transfer-Encoding: base64

R0lGODlhGAGgAPEAAP/////ZRaCgoAAAACH+PUNvcHlyaWdodCAoQykgMTk5
```

```
NSBJRVRGLiBVbmF1dGhvcml6ZWQgZHVwbGljYXRpb24gcHJvaGliaXRlZC4A
... дальнейший код ...

--delimeter--
```

Оформление SOAP-послания с дополнениями

В 2000 году консорциумом W3C выпущена спецификация "SOAP-послания с дополнениями" (**http://www.w3.org/TR/SOAP-attachments/**), опирающаяся на версию SOAP 1.1. В спецификации описаны правила включения SOAP-послания в MIME-сообщение типа multipart/related и правила пересылки его по протоколу HTTP. Правила эти просты.

1. SOAP-послание помещается в начальную часть MIME-сообщения типа multipart/related.
2. Тип MIME-сообщения устанавливается как text/xmi.
3. Заголовок начальной части содержит поле content-ID, а **HTTP**-заголовок, также как и заголовок MIME-сообщения, содержит поле Content-Type с параметром start, ссылающимся на значение поля content-ID начальной части. Остальные части могут содержать поле Content-ID **и/или Поле** Content-Location.
4. Ссылки на другие части MIME-сообщения выполняются в SOAP-послании с помощью атрибута href.

В листинге 3.13 приведен пример SOAP-послания, ссылающегося на два изображения в форматах GIF и JPEG.

Листинг 3.13. POST-запрос с SOAP-посланием и двумя изображениями

```
POST /services/SomeServlet HTTP/1.1
Host: www.some.com
Content-Type: multipart/related; boundary=delimeter; type=text/xml;
        start="<info2709.xml@some.com>"
Content-Length: XXXX
SOAPAction: http://some.com/services/InfoServlet

--delimeter
Content-Type: text/xmi; charset=utf-8
Content-Transfer-Encoding: 8bit
Content-ID: <info2709.xml@some.com>
```

```
<?xml version='1.0' ?>
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Body>
<ns:todayinfo
xmlns:ns="http://some.com/infonames">
<ns:map href="cid:info2709.gif@some.com" />
<ns:map href="cid:info2709.jpeg@some.com" />
</ns:todayinfo>
</env:Body>
</env:Envelope>

--delimeter
Content-Type: image/gif
Content-Transfer-Encoding: base64
Content-ID: <info2709.gif@some.com>

...Здесь GIF-изображение в коде base64...
--delimeter
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <info2709.jpeg@some.com>

...Здесь изображение в формате JPEG...
--delimeter--
```

Остается заметить, что MIME-сообщение, подготовленное по этим правилам и содержащее SOAP-послание, самым естественным образом передается по почтовому протоколу SMTP.

Формат сообщения DIME

В мае 2001 года появились первые сообщения о том, что фирмы IBM и Microsoft предложили новый формат DIME (Direct Internet Message Encapsulation) для записи сообщений с дополнениями, не использующий MIME-тип multipart. В июне 2002 года появился черновой вариант спецификации **DIME.** Его можно посмотреть по адресу **http://www.ibm.com/developerworks/ws-dime/.**

Сообщение **DIME** состоит из нескольких частей, называемых *записями* (record). В начале каждой записи идут несколько полей, отмечающих харак-

теристики записи. Каждая запись определяется тремя основными характеристиками:

- длиной в байтах своего содержимого (поле DATA_LENGTH);
- типом, записываемым строкой URI, MIME-типом или как-то по-другому (поле TYPE);
- необязательным идентификатором — строкой URI (поле ID).

Длина записи ограничена 2^{32} - 1 байтами. Допустимы записи нулевой длины.

Тип записи может быть любым. Длина поля TYPE заносится в поле TYPE_LENGTH, а формат поля TYPE, — строка это URI, MIME-тип или что-то другое — определяется полем TYPET. ЕСЛИ ТИП записи определяется строкой URI, то значение поля TYPE_T равно 0x01, если тип записи определяется MIME-типом — то 0x02. Значение Охоз записывается, если тип записи неизвестен, при этом значение поля TYPE_LENGTH равно нулю. Поле TYPET равно 0x04, если содержимое записи отсутствует. В таком случае поля TYPE_LENGTH И DATA_LENGTH раВНЫ НуЛЮ.

Идентификатор записи ID может быть любой строкой в виде URI. Его длина, записываемая в поле ID_LENGTH, всегда выравнивается до величины, кратной 4.

Кроме этих трех характеристик, запись может быть снабжена номером своей версии и несколькими дополнительными полями. Номер версии — это версия спецификации DIME, сейчас это 1. Спецификация не определяет назначение дополнительных полей, но предписывает, чтобы у них были определены тип и длина.

Тип всего сообщения DIME определен как MIME-тип application/dime. Сообщение DIME может входить с этим типом как одна из записей в другое сообщение DIME.

Первая запись DIME-сообщения помечается значением 1 флага MB (Message Begin), последняя запись — значением 1 флага ME (Message End). У сообщения, состоящего всего из одной записи, оба флага равны 1.

Несколько записей могут содержать связанную информацию. Все такие записи должны быть одного и того же типа, у них должен совпадать номер версии. Такие записи помечаются флагом CF (Chunk Flag), у первой и промежуточных записей этот флаг равен 1, у последней записи флаг CF равен о.

Итак, сообщение DIME может содержать какие угодно записи, относящиеся к Web-службам или совершенно другой природы. Правила применения DIME-сообщений к пересылке SOAP-посланий с дополнениями изложены в спецификации "WS-Attachments". Ее можно посмотреть по адресу **http://www.ibm.com/developerworks/ws-attach.html**.

Сообщение DIME, содержащее SOAP-послание, состоит из первичного SOAP-послания и нескольких вторичных частей произвольной природы. Это могут быть изображения различных форматов, записи звука в различных кодировках, программные коды, не обязательно определенные какими-нибудь типами MIME.

Первой записью сообщения DIME должно быть первичное SOAP-послание. Значение поля TYPE первой записи для версии SOAP 1.1 должно совпадать с идентификатором пространства имен http://schemas.xmlsoap.org/soap/envelope/, для версии SOAP 1.2 тип первой записи — `application/soap+xml`. У первой записи может быть необязательный идентификатор, у следующих записей идентификатор должен быть обязательно. Он используется для ссылок из одних записей сообщения на другие записи с помощью атрибута href.

Спецификация DIME описывает только формат упаковки нескольких записей в одно сообщение. Она ничего не говорит о способах пересылки полученного сообщения. Единственное условие — при пересылке сообщения POST-методом протокола HTTP в поле заголовка `Content-Type` следует указывать тип application/dime.

Из этого краткого описания видно, что сообщение DIME определено гораздо точнее, чем MIME-тип multipart. Поля, записанные в начале каждой записи сообщения DIME, определяют тип и длину каждой записи. Это дает возможность программе, разбирающей сообщение, сделать необходимые проверки, не просматривая все сообщение целиком. Это экономит время и ресурсы компьютера.

С другой стороны, такое точное описание сообщения DIME можно сделать только каким-нибудь инструментальным средством, автоматически создающим сообщение. Не стоит и пытаться написать сообщение DIME вручную.

Средства создания SOAP-посланий

Несмотря на то, что технология Web Services еще очень молода, есть уже много средств создания SOAP-серверов и SOAP-клиентов. Эти средства можно разбить на четыре группы.

1. "Мастера", встроенные в IBM WebSphere Studio, Sun ONE Studio, Eclipse, IntelliJ IDEA, Microsoft Visual Studio .NET, Borland JBuilder и в другие инструментальные средства разработчика. Они позволяют несколькими щелчками по кнопке мыши создать Web-службу. Это хорошо, но созданные таким образом Web-службы редко удовлетворяют профессионала.
2. Наборы крупных классов и реализованных интерфейсов, автоматизирующих почти всю работу и требующих минимальной доводки: Apache SOAP, Apache Axis, TME GLUE, Microsoft SOAP Toolkit. Часто они уже

встроены или легко встраиваются в серверы приложений. Эти наборы удобны, когда надо быстро создать несложную Web-службу.

3. Наборы более мелких классов и нереализованных интерфейсов, требующих более тщательной сборки и значительных усилий от программиста: Sun SAAJ, JAXM, JAX-RPC, JAXR, IBM WSTK. Они дают больше свободы и позволяют создавать самые разнообразные Web-службы.

4. Языки программирования: Java, Perl, C/C++, C#, VB.NET и другие. С их помощью можно создать все, что угодно, в том числе и Web-службы. Сейчас в их стандартные библиотеки включаются классы, облегчающие создание Web-служб.

Средства первой группы не требуют какого-либо дополнительного описания, кроме встроенной в них интерактивной справки. Средства четвертой группы описаны в других книгах, например, [9, 10]. В этой главе я опишу средства второй группы, в *главах 6—7*— средства третьей группы.

Среди средств создания Web-служб второй группы в настоящее время наиболее активно развивается и широко распространяется Apache Axis. На момент написания книги он только что вышел из чернового состояния, но уже встроен или легко встраивается во многие серверы приложений: IBM WebSphere, BEA WebLogic, Sun ONE Application Server. Его можно включить даже в состав Web-контейнеров, таких как Tomcat, Resin, JBoss. Кроме того, Axis свободно распространяется, доступны его исходные тексты. Поэтому мы рассмотрим подробнее работу с Axis.

Работа с Axis

Программный продукт Axis (Apache extensible Interaction System) (**http://xml.apache.org/axis/**), разрабатываемый консорциумом W3C, представляет собой набор классов, большинство из которых реализуют интерфейсы пакетов Sun SAAJ, JAXM, JAX-RPC. Эти интерфейсы мы рассмотрим подробно в *главе 6.* Кроме того, в состав Axis входит небольшой отладочный SOAP-сервер — класс `SimpleAxisServer` — и классы для преобразования описаний WSDL в объекты Java и обратно.

Установка Axis

Установить Axis в ваш сервер приложений очень легко. Достаточно скопировать каталог axis из каталога `$AXIS_HOME/webapps/` в каталог `webapps` вашего сервера приложений или в другой каталог, в который устанавливаются сервлеты. После этого надо скопировать в каталог `axis/WEB-INF/lib/` архив вашего любимого XML-парсера, например, xerces.jar, и

можно работать. Axis установлен как одно из Web-приложений, работающих на вашем сервере. Запустите сервер приложений, наберите в браузере, запущенном на сервере, адрес **http://localhost:8080/axis/** — и вы увидите начальную страничку Axis.

Для дальнейшей работы с Axis необходимо, чтобы сервер приложений знал, где расположены файлы axis.jar, commons-logging.jar, jaxrpc.jar, log4j-1.2.4.jar, tt-bytecode.jar и xerces.jar. Для этого надо занести полные пути к этим файлам (к файлам, а не к каталогам, в которых они лежат) в переменную CLASSPATH или положить эти файлы в каталоги, известные серверу приложений. Обычно это один из каталогов lib, ext, common или endorsed сервера приложений.

Запросы клиентов при работе с Axis принимает сервлет `AxisServlet`. Убедиться в его правильной работе можно, набрав в браузере, работающем на сервере, адрес **http://localhost:8080/axis/servlet/AxisServlet.** В окне браузера должна появиться строка "And now Some Services" и список некоторых Web-служб, установленных на сервере.

В стандартную поставку Axis входит много примеров Web-служб. Посмотрите каталог axis/WEB-INF/classes/samples/. Вы увидите десяток подкаталогов, в которых лежат файлы с классами примерных Web-служб. Их исходные тексты лежат в каталоге $AXIS_HOME/samples/.

Создание Web-службы для Axis

Создать Web-службу, которая будет работать под управлением Axis, очень просто. Достаточно написать классы Java, реализующие Web-службу, дать файлу, содержащему исходные тексты этих классов, расширение ".jws", и положить этот файл в каталог axis сервера приложений.

Создадим Web-службу `EchoService`, полезную для отладки и проверки правильности передачи кириллицы. Она предоставляет одну Web-услугу getEcho (), просто возвращающую клиенту содержимое SOAP-запроса. Класс EchoService, содержащий метод getEcho о, приведен в листинге 3.14.

Листинг 3.14. Web-служба EchoService

```
public class EchoService{

   public String getEcho(String req){
      return req;
   }
}
```

Назовем файл с содержимым листинга 3.14 EchoService.jws и положим его в каталог axis сервера приложений. Все, Web-служба готова. Не нужно компилировать класс EchoService, Axis сделает это сам при первом запросе к Web-службе.

Клиент Axis

Клиентское приложение, обращающееся к Web-службе, выполняющейся под управлением Axis, написать нетрудно. В листинге 3.15 показан клиент Web-службы EchoService листинга 3.14.

Листинг 3.15. Клиент Web-службы EchoService

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import java.net.*;

public class EchoServiceClient{

   public static void main(String[] args) throws Exception)

      if (args.length != 1)(

        System.err.println(
           "Usage: java EchoServiceClient <request>");

        System.exit(1);
      }

      Service service = new Service();
      Call call = (Call)service.createCall();

      String endpoint =
          "http://localhost:8080/axis/EchoService.jws";

      call.setTargetEndpointAddress(new URL(endpoint));
      call.setOperationName("getEcho");

      String request = args[0];

      String response =
```

```
            (String)call.invoke(new Object[]{request});

        System.out.println("Послано: " + request);
        System.out.println("Получено: " + response);
    }
}
```

В листинге 3.15 использованы интерфейсы service и Call пакета JAX-RPC. Они реализованны классами, входящими в Axis. Мы познакомимся с этими интерфейсами в *главе 6.* Как видно из листинга 3.15, основную роль в получении Web-услуги играет объект типа call. Он содержит адрес Web-службы endpoint и название Web-услуги getEcho. В него заносятся аргументы Web-услуги в виде массива объектов. В нашем простом примере массив аргументов состоит только из одного элемента — строки request. Наконец, объект call методом invoke о возвращает объект, содержащий результат выполнения Web-услуги.

Использование конфигурационного файла

Возможность создать Web-службу, просто написав JWS-файл — большое достоинство Axis, но часто возникают ситуации, в которых этот способ не годится. Например, надо создать Web-службу из готовых классов Java, исходные тексты которых отсутствуют, или задать какое-то особенное отображение сложных типов данных для передачи их по протоколу SOAP, или создать Web-службу, независимую от Axis. В таких случаях Axis предлагает написать конфигурационный файл — дескриптор установки (Deployment Descriptor), описывающий Web-службу во всех подробностях. Мы будем называть его *DD-файлом.* Потом, с помощью DD-файла надо выполнить установку Web-службы в Web-контейнер.

Конфигурационный DD-файл — это документ XML с корневым элементом <deployment>. Используемые в DD-файле имена элементов описаны в пространстве имен с идентификатором http://xml.apache.org/axis/wsdd/. В DD-файлах часто записываются имена классов Axis. Эти имена описаны в пространстве имен с идентификатором http://xml.apache.org/axis/wsdd/providers/java. Обычно корневой элемент с описанием пространств имен выглядит так:

```
<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
```

В начале файла необязательным элементом <globalConfiguration> можно установить начальные параметры работы Web-службы.

Основную роль в описании играют элементы <handler>, атрибут type которых задает класс-обработчик, устанавливаемый в Axis, или ссылку на другой, уже определенный элемент <handler>. Классы-обработчики **SOAP**-посланий создаются в полном соответствии с правилами **JAX-RPC,** описанными в *главе 6.* Для ссылки на элемент <handler> из другого элемента предназначен необязательный атрибут name. Режим работы класса-обработчика и другие параметры можно задать элементами

```
<option name="имя" value="значение" />,
```

вложенными в элемент <handler>. Например:

```
<option name="namespace"
        value="http://tempuri.org/wsdl/2002/12/" />
```

Другой часто встречающийся элемент —

```
<parameter name="имя" value="значение" />
```

описывает самые разнообразные параметры Web-служб, например, элемент

```
<parameter name="adminPassword" value="v2xYUcm" />
```

содержит пароль администратора. Элементы <parameter> чаще всего вкладываются в элемент <service>, описывающий Web-службу. Например, следующий элемент:

```
<service name="AdminService" provider="java:MSG">

  <parameter name="methodName" value="AdminService" />
  <parameter name="enableRemoteAdmin" value="false" />
  <parameter name="className" value="org.apache.axis.utils.Admin" />
  <parameter name="allowedMethods" value="*"/>

</service>
```

описывает Web-службу AdminService, входящую в состав Axis. Она реализована классом Admin, у которого описан метод AdminService (да-да, имя метода почему-то начинается с заглавной буквы). Второй параметр, с именем `enableRemoteAdmin`, запрещает или разрешает удаленное администрирование Web-службы. Это зависит от его значения value. Четвертый параметр, с именем `allowedMethods`, описывает открытые (public) методы класса. Их имена — значение атрибута value — надо перечислять через пробел или запятую. Звездочка означает, что все методы класса открыты.

Атрибут provider элемента <service> определяет класс-обработчик SOAP-посланий. Этот класс должен реализовать интерфейс Handler из пакета **JAX-RPC** или расширить какой-нибудь из абстрактных классов `GenericHandler` или BasicHandler. Для облегчения этой работы в составе

Axis есть расширение BasicProvider класса BasicHandler. Класс BasicProvider, В СВОЮ очередь, расширен классом JavaProvider, а у того есть расширения MsgProvider И RPCProvider.

Атрибут style элемента <service> или некоторые значения атрибута provider определяют стиль Web-службы. Процедурный стиль, а он принимается ПО умолчанию, записывается как provider="java:RPC" или style="RPC". Послания процедурного стиля будет обрабатывать класс RPCProvider. Документный стиль записывается как style="document". Axis вводит разновидность документного стиля, записываемую как style="wrapped". В этом случае при создании метода обработки XML-элемента Axis записывает вложенные в него элементы как аргументы метода. Четвертый стиль Web-службы, записываемый как provider="java:MSG" или style="message", означает, что Axis будет обрабатывать послание непосредственно в виде XML, не переводя его в объекты Java. Это выполняется С ПОМОЩЬЮ класса MsgProvider.

Web-услуги описываются элементами <operation>. Атрибут name задает имя метода, предоставляющего Web-услугу, атрибут returnType — тип возвращаемого значения. Атрибутом returnQName можно дать имя возвращаемому значению для ссылок внутри DD-файла. Аргументы метода описываются вложенными элементами <parameter>. Например:

```
<operation name="getEcho"
        returnQName="echo" returnType="xsd:string">
  <parameter name="req" type="xsd:string" />
</operation>
```

В элемент <service>, описывающий Web-службу, обрабатывающую запросы, часто вкладываются элементы <requestFlow> и <responseFlow>. Вложенные в них элементы <handler> описывают классы-обработчики, начинающие и заканчивающие обработку запроса. Например:

```
<service name="test">
  <parameter name="className" value="test.Implementation"/>
  <parameter name="allowedMethods" value="*"/>
  <namespace>http://testservice/</namespace>
  <requestFlow>
    <handler type="java:MyHandlerClass"/>
```

```
    <handler    type="somethingIDefinedPreviously"/>

  </requestFlow>
```

</service>

Элемент <transport> описывает протокол пересылки SOAP-послания. Название протокола указывается атрибутом name. У элемента <transport>, описывающего клиента Axis, есть еще один атрибут pivot, задающий класс-отправитель SOAP-послания, например, класс HTTPSender, входящий в состав Axis. Протокол отправки послания может быть описан так:

```
<transport name="SimpleHTTP" pivot="HTTPSender">

  <requestFlow>

    <handler name="HTTPActionHandler"
     type="java:org.apache.axis.handlers.http.HTTPActionHandler"/>

  </requestFlow>
```

</transport>

Два элемента описывают правила сериализации и десериализации сложных типов данных. Элемент

```
<typeMapping qname="xml-имя" classname="имя класса"
   serializer="имя класса" deserializer="имя класса" />
```

описывает атрибутом classname сложный тип данных — класс Java. Атрибуты serializer и deserializer задают имена классов, содержащих методы сериализации и десериализации. Например:

```
<typeMapping
  xmlns:ns="http://soapinterop.org/"
  qname="ns:ArrayOf_apachesoap_Map"
  type="java:java.util.HashMap[]"
  serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
  deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
```

Элемент

```
<beanMapping qname="xml-имя" languageSpecificType="имя класса" />
```

описывает тип данных, описываемый классом Java, оформленным как JavaBean. У такого класса есть методы getxxx() и setxxx(), используемые для сериализации сложных типов Java в XML стандартными классами Axis, реализующими интерфейсы Serializer И Deserializer, а именно, BeanSerializer и BeanDeserializer. **Компоненту JavaBean не нужны специальные классы для сериализации. Например:**

```
<beanMapping qname="reg: Service"
   languageSpecificType="java:samples.bidbuy.Service"/>
```

В листинге 3.16 приведено полное описание Web-службы администратора AdminService, входящей в состав Axis.

Листинг 3.16. Конфигурационный файл Web-службы администратора

```
<deployment
   xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <globalConfiguration>

    <parameter name="adminPassword" value="admin" />
    <parameter name="sendXsiTypes" value="true" />
    <parameter name="sendMultiRefs" value="true" />
    <parameter name="sendXMLDeclaration" value="true" />

    <requestFlow>
      <handler type="java:org.apache.axis.handlers.JWSHandler" />
    </requestFlow>

  </globalConfiguration>

  <handler name="Authenticate"
type="java:org.apache.axis.handlers.SimpleAuthenticationHandler"/>

  <handler name="LocalResponder"
type="java:org.apache.axis.transport.local.LocalResponder"/>

  <handler name="URLMapper"
type="java:org.apache.axis.handlers.http.URLMapper"/>

  <handler name="MsgDispatcher"
```

```
type="java:org.apache.axis.providers.java.MsgProvider"/>

  <handler name="RPCDispatcher"
type="java:org.apache.axis.providers.java.RPCProvider"/>

  <service name="AdminService" provider="java:MSG">

    <parameter name="methodName" value="AdminService" />
    <parameter name="enableRemoteAdmin" value="false" />
    <parameter name="className" value="org.apache.axis.utils.Admin" />

  </service>

  <service name="JWSProcessor" provider="Handler">

    <parameter name="handlerClass"

       value="org.apache.axis.handlers.JWSProcessor" />

  </service>

  <transport name="http">

    <requestFlow>
      <handler type="URLMapper" />
      <handler

         type="java:org.apache.axis.handlers.http.HTTPAuthHandler" />

    </requestFlow>

  </transport>

  <transport name="local">

    <responseFlow>

      <handler
         type="java:org.apache.axis.transport.local.LocalResponder" />

    </responseFlow>

  </transport>

</deployment>
```

Конфигурационный DD-файл запоминается обычно под именем deploy.wsdd. После его создания надо перейти в каталог с этим файлом и выполнить команду

```
$ java org.apache.axis.client.http.AdminClient deploy.wsdd
```

Класс `AdminClient` установит описанную Web-службу в Axis. На консоли появятся сообщения

```
Processing file deploy.wsdd
<Admin>Done processing</Admin>
```

После этого клиент может обратиться к Web-службе, воспользовавшись ее адресом URL, например, **http://localhost:8080/axis/services/EchoService**.

С помощью утилиты AdminClient можно просмотреть описания всех Web-служб, установленных на данной машине. Для этого надо набрать команду

```
$ java org.apache.axis.client.http.AdminClient list
```

Для удаления Web-службы из Web-контейнера создается файл undeploy.wsdd. Он похож на DD-файл, но его корневым элементом служит элемент `<undeployment>`.

Кроме установки Web-службы в Web-контейнер, утилита AdminClient создает все необходимые для ее работы файлы, в том числе клиентские заглушки. Пример клиента, использующего их, приведен в листинге 3.18.

Использование описаний WSDL

Конфигурационный DD-файл записывается в формате, специально разработанном для Axis. Такой файл не переносим в другие системы обеспечения Web-служб. Но в Axis можно использовать и другое средство описания Web-служб — язык WSDL.

В своей работе Axis широко использует описания **WSDL** (см. *главы 2, 4),* даже если вы их не сделали. Наберите в браузере Internet Explorer строку **http://localhost:8080/axis/EchoService.jws?WSDL**, и вы увидите в окне браузера описание Web-службы `EchoService`, сделанное на языке WSDL. Оно сгенерировано автоматически методами класса `Java2WSDL`.

Создание WSDL-файла по классу Java

По имеющемуся откомпилированному классу или интерфейсу Axis может создать его описание на языке WSDL. Для этого в составе Axis есть класс `Java2WSDL`. Он работает из командной строки примерно так:

```
$ java org.apache.axis.wsdl.Java2WSDL -o echo.wsdl \
-l"http://localhost:8080/axis/services/echo" \
-n "urn:echo" -p"echoservice" urn:echo EchoService
```

В этой командной строке за флагом -o записывается имя создаваемого WSDL-файла echo.wsdl.

За флагом -l идет URI, последнее имя которого, echo, будет именем порта в WSDL-файле.

За флагом -n стоит идентификатор пространства имен WSDL-файла `urn:echo`.

За флагом -p расположены два параметра через пробел: параметр echoservice — это имя пакета, а параметр `urn:echo` — идентификатор пространства имен.

Наконец, последнее имя `EchoService` в командной строке — это имя класса, для которого создается описание WSDL.

В результате выполнения этой команды в текущем каталоге создается файл echo.wsdl с описанием WSDL класса Echoservice.

Создание Web-службы по WSDL-файлу

По имеющемуся описанию Web-службы на языке WSDL можно построить Web-службу с помощью класса `WSDL2Java`. Для этого надо набрать в командной строке следующее:

```
$ java org.apache.axis.wsdl.WSDL2Java -o . -d Session -s \
-p echo.ws echo.wsdl
```

В этой командной строке флаг -o указывает каталог, в который надо поместить создаваемые файлы. В данном случае это текущий каталог. На это указывает точка, следующая за флагом -o.

Флаг -d определяет продолжительность работы Web-службы одним из трех слов:

- Application — один экземпляр Web-службы обслуживает все запросы;
- Request — для обслуживания каждого запроса создается свой экземпляр Web-службы (по умолчанию);
- `Session` — экземпляр Web-службы образует сеанс связи с клиентом.

Флаг -s указывает на то, что следует создать все файлы, необходимые для работы сервера Web-службы.

За флагом -p идет имя пакета echo.ws, в котором будут лежать создаваемые классы.

Наконец, последнее слово в командной строке — это имя WSDL-файла echo.wsdl.

Команда создаст в текущем каталоге echo подкаталог ws, а в нем множество файлов, образующих RMI-сервер, заглушки, скелетоны и вспомогательные файлы, необходимые для работы удаленного объекта системы RMI. В их числе есть и конфигурационные DD-файлы deploy.wsdd. и undeploy.wsdd.

Среди этих файлов находится файл EchoServiceSoapBindingImpl.java, в котором надо реализовать методы Web-службы, как это обычно делается в системе RMI [10].

Удаленный интерфейс системы RMI заносится в файл EchoServicePortType.java, а его клиентская заглушка — в файл EchoServiceSoapBindingStub.java.

Клиент создает заглушку методом getEchoServicePort о класса-фабрики, записанного в файл EchoService.java.

После того как это сделано, надо откомпилировать все файлы в каталоге echo/ws:

```
$ javac echo/ws/*.java
```

Потом перейти в каталог ws и установить созданную Web-службу в сервер приложений:

```
$ java org.apache.axis.client.AdminClient deploy.wsdd
```

Сеанс связи с Axis

Еще одна интересная особенность Axis — он может установить сеанс связи с клиентом. Сеанс связи описан интерфейсом session, реализованным двумя классами — классом `SimpleSession`, работающим через заголовок SOAP-послания, и классом AxisHttpSession, создающим сеанс связи через сервлет при помощи cookie [10].

В первом случае в заголовок SOAP-послания записывается блок `<SessionID>`, содержащий уникальный идентификатор сеанса. Все действия по его записи и проверке выполняются методами класса `SimpleSessionHandler`. И в клиенте и в сервере должно находиться по экземпляру этого класса.

Во втором случае обычным образом записываются и проверяются поля `setCookie` И Cookie [10] HTTP-заголовка.

Методы сеанса

Интерфейс `Session` описывает метод

```
public void set(String key, Object value);
```

задающий идентификатор сеанса key со значением value.

Все идентификаторы сеанса можно получить методом

```
public Enumeration getKeys();
```

а значение отдельного идентификатора key — методом

```
public Object get(String key);
```

Идентификатор key удаляется методом

```
public void remove(String key);
```

Сеанс завершается, когда в течение времени timeout, определенного методом

```
public void setTimeout(int timeout);
```

не было выполнено ни одного действия. Время отсчитывается в секундах.

Узнать это время можно методом

```
public int getTimeout();
```

Если надо продлить сеанс, то можно воспользоваться еще одним методом

```
public void touch();
```

интерфейса `Session`. Этот метод "помечает" сеанс связи, сообщая, что он только что использовался.

Класс simplesession реализует все методы интерфейса session, добавляя к ним конструктор по умолчанию, начинающий сеанс связи, и метод

```
public long getLastAccessTime() ;
```

возвращающий время последнего действия в сеансе.

Класс AxisHttpSession работает в сервлетах в рамках сеанса класса `HttpSession`, экземпляр которого передается или записывается прямо в конструкторе

```
public AxisHttpSession(HttpSession session);
```

или косвенно, через запрос req, в конструкторе

```
public AxisHttpSession(HttpServletRequest req);
```

Поэтому, кроме методов интерфейса session, в нем есть методы доступа к сеансу сервлета

```
public void setRep(HttpSession session);
public HttpSession getRep();
```

Создание сеанса средствами Axis

Хотя разработчик может "вручную" создать сеанс связи с Web-службой методами классов `SimpleSession` И AxisHttpSession, В Axis есть средства, ав-

томатизирующие этот процесс. Рассмотрим простейший пример Web-службы, которая при первом обращении к ней запоминает имя клиента и использует его в дальнейшей работе с тем же клиентом. Эта Web-служба приведена в листинге 3.17.

Листинг 3.17. Web-служба, использующая сеанс связи

```
public class HelloSession{

   private String lastName = "";

   public String sayHello(String name){

      String resp = "";

      if (name.equals(lastName))
            resp = "Рады снова видеть Вас, ";
      else  resp = "Здравствуйте, ";

      lastName = name;

      return resp + name + "\n";
   }
}
```

Для того чтобы Axis организовал сеанс связи с Web-службой `HelloSession`, не создавая при каждом обращении к ней новый экземпляр, в ее конфигурационном файле надо записать элемент <parameter>, дающий значение "Session" параметру "scope":

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

   <service name="HelloSession" provider="java:RPC">

      <parameter name="className" value="HelloSession" />
      <parameter name="methodName" value="sayHello" />
      <parameter name="scope" value="Session"/>

   </service>

</deployment>
```

Если Web-служба создается по ее описанию WSDL, то достаточно при вызове утилиты WSDL2Java записать параметр -d session, как это сделано в примере использования этой утилиты, приведенном выше.

Это еще не все. Сервер создаст сеанс связи только по прямому указанию клиента. Указание дается обращением к методу

```
public void setMaintainSession(boolean session);
```

класса-заглушки stub, реализующего в Axis одноименный интерфейс системы JAX-RPC. Axis, точнее, утилиты AdminClient или WSDL2Java, автоматически создают расширение класса stub, в нашем примере имя этого расширения — HelloSessionSoapBindingStub. Все вместе выглядит так, как показано в листинге 3.18.

Клиент, записанный в листинге 3.18, пользуется Web-услугой sayHelloO два раза. Первый раз Web-услуга говорит ему: "Здравствуйте, ", второй раз — "Рады снова видеть Вас, ".

Листинг 3.18. Клиент Web-службы, создающий сеанс связи

```
public class HelloSessionClient{

   public static void main(String[] args){

      String name = "";

      if (args[0] != null) name = args[0];

      // Класс-фабрика HelloSession создан утилитой WSDL2Java.
      HelloSession fact = new HelloSession();

      HelloSessionPortType service =
                fact.getHelloSessionPort();

      HeiioSessionSoapBindingStub stub =

            (HeiioSessionSoapBindingStub)service;

      stub.setMaintainSession(true);

      try{
```

```
            System.out.println(service.sayHello(name));
            System.out.println(service.sayHello(name));

        }catch(Exception e){
            System.out.println(e);
        }
    }
}
```

ГЛАВА 4

Описание Web Services на языке WSDL

В *главе 2* мы говорили о том, что после создания Web-службы на сервере в виде сервлета, страницы JSP, JWS-файла, компонента EJB или другого объекта, следует описать состав и возможности Web-службы на языке, не зависящем от платформы, операционной системы, системы программирования, использованной при создании Web-службы. Это описание регистрируется в общедоступном месте Интернета, например, реестре UDDI или ebXML, или хранится на сервере Web-службы. Описание должно содержать полную и точную информацию обо всех услугах, предоставляемых Web-службой, способы получения услуг, содержимое запроса на получение услуги, формат предоставляемой информации.

Одно из средств точного и единообразного описания Web-услуг — язык WSDL, созданный консорциумом W3C. Этот язык — еще одна реализация XML. Его последняя рекомендованная спецификация всегда публикуется на странице **http://www.w3.org/TR/wsdl.** Во время написания книги на черновой стадии была версия WSDL 1.2, которую мы и опишем в этой главе.

Состав документа WSDL

Корневым элементом документа XML — описания WSDL — служит элемент <definitions>. В этом элементе необязательным атрибутом name можно дать имя описанию. Кроме того, это удобное место для введения используемых в описании пространств имен.

Описания WSDL активно используют различные пространства имен. Кроме собственных имен, язык WSDL часто использует имена типов и эле-

ментов языка описания схем XSD *(см. главу 1)* и имена языка протокола SOAP. Пространство имен языка WSDL часто описывается как пространство имен по умолчанию. Идентификатор пространства имен последней на время написания этих строк версии WSDL 1.2 был равен **http://www.w3.org/2002/07/wsdl.** Целевое пространство имен, идентификатор которого определяется атрибутом `targetNamespace`, обычно получает префикс tns (target namespace).

В корневой элемент <definitions> вкладываются элементы шести основных и двух дополнительных типов. Все элементы необязательны, их может быть любое количество, за исключением элемента <types>, который может встретиться в документе только один раз. У каждого элемента есть имя, определяемое обязательным атрибутом name. Элементы ссылаются друг на друга с помощью этих имен. Вот элементы, вкладываемые в корневой элемент `<definitions>`.

- <types> — определяет сложные типы, используемые Web-службой, с помощью языка XSD или другого языка описания типов. Этот элемент не нужен, если Web-служба применяет только простые типы, описанные в языке XSD.
- <message> — описывает каждое SOAP-послание: запрос, ответ, пересылку документов. В этот элемент вкладываются элементы <part>, описывающие неделимые с точки зрения WSDL части послания. Для посланий процедурного типа каждый элемент <part> может описывать имя и тип одного аргумента запроса или тип возвращаемого значения. Для посланий документного типа элементы <part> могут описывать каждую часть послания MIME-типа "multipart/related". Это абстрактное описание затем конкретизируется элементами <binding>.
- <portType> — описывает интерфейс Web-службы, называемый в языке WSDL *пунктом назначения* (endpoint) или *портом* (port) прибытия послания. Он описывается как набор Web-услуг, называемых в языке WSDL *операциями.* Переводя это описание на язык программирования можно заметить, что порт хорошо соотносится с интерфейсом Java, а каждая операция — с методом этого интерфейса. Операции описываются вложенными элементами <operation>, описывающими каждую отдельную услугу. Услуга описывается действиями, которые разбиты на четыре вида. Это два простых действия: "получение послания", "отправка ответа", и два комбинированных действия: "отправка послания — получение ответа" или, наоборот, "получение послания — отправка ответа". Получение и отправка, в свою очередь, описываются вложенными элементами <input> и `<output>`, а сообщение об ошибке — элементом `<fault>`. Получаемые и отправляемые послания уже должны быть описаны элементами <message>, элементы `<input>`, <output> И <fault> ссылаются на них своим атрибутом message.

- <serviceType> — перечисляет вложенными элементами <portType> набор портов, связанных с одной Web-службой. Один и тот же порт может быть связан с несколькими службами.
- <binding> — описывает конкретный формат пересылки послания: протоколы, такие как SOAP или HTTP, способы упаковки послания, тип его содержимого: HTML, XML или другой MIME-тип послания. Каждый элемент <message> может быть связан с несколькими такими элементами, по одному для каждого способа пересылки. В этот элемент вкладываются элементы, определенные в схеме выбранного протокола.
- <service> — указывает местоположение Web-службы как один или несколько *портов*. Каждый порт описывается вложенным элементом <port>, содержащим адрес интерфейса Web-службы, заданный по правилам выбранного в элементе <binding> способа пересылки.

Кроме этих шести основных элементов есть еще два вспомогательных элемента.

- <import> — включает файл с XSD-схемой описания WSDL или другой WSDL-файл.
- <documentation> — комментарий. Его можно включить в любой элемент описания WSDL.

Можно сказать, что элементы <types>, <message> и <portType> показывают, ЧТО есть в описываемой Web-службе, какие услуги она предоставляет, как организованы услуги, какие типы данных у этих услуг.

Элементы <binding> объясняют, КАК реализована Web-служба, каков протокол передачи посланий: HTTP, SMTP или какой-то другой, а также задает технические характеристики передачи данных.

Наконец, элементы <service> показывают, ГДЕ находится Web-служба, связывая описание <binding> с конкретными адресами Web-службы.

Структура документа WSDL показана в листинге 4.1. Символы в квадратных скобках не содержатся в документе. Они показывают повторяемость элемента или атрибута в описании Web-службы:

- символ [?] означает, что элемент или атрибут может появиться в документе нуль или один раз;
- символ [*] означает, что элемент может появиться нуль или несколько раз;
- символ [+] означает, что элемент может появиться один или несколько раз;
- отсутствие символа в квадратных скобках означает, что атрибут должен появиться ровно один раз.

Листинг 4.1. Схема WSDL-документа

```
<?xml version="1.0" ?>

<wsdl:definitions name="имя" [?]
    targetNamespace="идентификатор пространства имен"
    xmlns:префикс="идентификатор пространства имен" [*] >

  <import namespace="идентификатор пространства имен"
          location="URI-адрес" /> [*]

  <wsdl:documentation> [?]
       Произвольный комментарий
  </wsdl:documentation>

  <wsdl:types> [?]

     <wsdl:documentation> [?]
         Описания сложных и нестандартных типов.
     </wsdl:documentation>

     <xsd:schema> [*]
         <!—Здесь записывается описание сложных типов —>
     </xsd:schema>

  </wsdl:types>

  <wsdl:message пате="имя"> [*]

     <wsdl:documentation> [?]
         Абстрактное описание SOAP-послания как
         набора составляющих его частей.
     </wsdl:documentation>

      <part пате="имя части (аргумента метода)"
            element="XSD-имя элемента" [?]
            type="XSD-тип аргумента" [?] /> [*]

  </wsdl:message>
```

```
<wsdl:portType name="имя">  [*]

    <wsdl:documentation>  [?]
        Абстрактное описание Web-службы как
        набора операций (услуг).
    </wsdl:documentation>

    <wsdl:operation пате="имя услуги">  [*]

        <wsdl:documentation>  [?]
            Описание услуги как получения (input) и
            отправки (output, fault) посланий.
        </wsdl:documentation>

        <wsdl:input message="имя соотв. элемента <message>">  [?]
            <wsdl:documentation>  [?]
                Получаемое послание.
            </wsdl:documentation>
        </wsdl:input>

        <wsdl:output message="имя соотв. элемента <message>">  [?]
            <wsdl:documentation>  [?]
                Отправляемое послание.
            </wsdl:documentation>
        </wsdl:output>

        <wsdl:fault пате="имя"
                    message="имя соотв. элемента <message>">  [*]
            <wsdl:documentation>  [?]
                Отправляемое сообщение об ошибке.
            </wsdl:documentation>
        </wsdl:fault>

    </wsdl:operation

</wsdl:portType>

<wsdl:serviceType пате="имя">  [*]
```

```
    <wsdl:portType name="имя соотв. элемента <portType>"/> [+]
</wsdl:serviceType>

<wsdl:binding name="имя"
              type="имя соотв. элемента <portType>"> [*]

    <wsdl:documentation . . ./>[?]

        <!--
        Сюда записываются элементы, описывающие детали
        конкретного протокола. Они определяются в схеме
        этого протокола.
        -->

    <wsdl:operation пате="имя"> [*]

        <wsdl:documentation . . ./>[?]

            <!--
            Сюда записываются элементы, описывающие детали
            конкретной операции.
            -->

        <wsdl:input> [?]

            <wsdl:documentation . . ./>[?]

                <!--
                Сюда записываются элементы, описывающие
                детали конкретного получаемого послания.
                -->

        </wsdl:input>

        <wsdl:output> [?]

            <wsdl:documentation . . ./>[?]

                <!--
```

```
            Сюда записываются элементы, описывающие
            детали конкретного отправляемого послания.
            -->

        </wsdl:output>

        <wsdl:fault name="имя"> [*]

            <wsdl:documentation . . . /> [?]

                <! -
                Сюда записываются элементы, описывающие
                детали конкретного сообщения об ошибке.
                -->

        </wsdl:fault>

    </wsdl:operation>

</wsdl:binding>

<wsdl:service name="имя"
        serviceType="имя соотв. элемента <serviceType>"> [*]

    <wsdl:documentation> [?]
        Описание интерфейса Web-службы как набора портов.
    </wsdl:documentation>

    <wsdl:port пате="имя"
            binding="имя соотв. элемента <binding>"> [*]

        <wsdl:documentation . . ./>[?]

            <!--
            Сюда записывается обязательный и единственный
            адрес интерфейса Web-службы, записанный по правилам
            протокола, указанного в элементе <binding>.
             -->
```

```
    </wsdl:port>

  </wsdl:service>

</wsdl:definitions>
```

Каждый конкретный протокол пересылки посланий — SOAP, HTTP, FTP, SMTP — добавляет к шести основным и двум вспомогательным элементам языка WSDL свои дополнительные элементы, описывающие особенности данного протокола.

Приведем простой пример. В листинге 3.14 мы записали в виде класса Java простейшую Web-службу, возвращающую без всякой обработки присланный запрос:

```
public class EchoService{
   public String getEcho(String req){
      return req;
   }
}
```

В листинге 4.2 приведено описание этой Web-службы на языке WSDL, использующее протокол SOAP.

Листинг 4.2. Описание Web-службы EchoService

```
<?xml version="1.0" encoding="UTF-8" ?>

<definitions
   targetNamespace="http://echoservice.com/echoservice.wsdl"
   xmlns="http://www.w3.org/2002/07/wsdl"
   xmlns:tns="http://echoservice.com/echoservice.wsdl"
   xmlns:soap="http://www.w3.org/2002/07/wsdl/soap12"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="getEchoRequest">
     <part name="req" type="xsd:string" />
  </message>

  <message name="getEchoResponse">
     <part name="return" type="xsd:string" />
  </message>
```

```
<portType name="EchoServicePort">

   <operation name="getEcho">
      <input  message="tns:getEchoRequest"  name="getEchoRequest" />
      <output message="tns:getEchoResponse" name="getEchoResponse" />
   </operation>

</portType>

<binding name="EchoServiceSoapBinding" type="tns:EchoServicePort">

  <soap:binding style="rpc"

      transport="http://schemas.xmlsoap.org/soap/http" />

   <operation name="getEcho">

     <soap:operation soapAction="" />

     <input name="getEchoRequest">
        <soap:body encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://echoservice.com/echoservice.wsdl"
          use="encoded" />

     </input>

     <output name="getEchoResponse">

        <soap:body encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://echoservice.com/echoservice.wsdl"
          use="encoded" />

     </output>

  </operation>
```

```
  </binding>

  <service name="EchoServService">

     <port binding="tns:EchoServiceSoapBinding" name="EchoService">

        <soap:address location=

              "http://localhost:8080/axis/EchoService.jws" />

     </port>

  </service>

</definitions>
```

В листинге 4.2 мы в элементе <definitions> определили префиксы всех нужных нам пространств имен. Далее мы описали запрос и ответ в двух элементах <message>. Мы дали им имена "getEchoRequest" и "getEchoResponse". В запросе один аргумент типа xsd:string. Этот тип определен в языке XSD. Мы дали аргументу имя req, совпадающее с именем аргумента метода getEcho(). Значению, возвращаемому методом, мы дали имя return, его тип тоже xsd: string.

Имена "getEchoRequest" и "getEchoResponse" использованы в следующем элементе <portType> для указания входных и выходных параметров Web-услуги. В него вложен один элемент <operation>. Это означает, что Web-служба предоставляет одну услугу, имя которой "getEcho" совпадает с именем метода, выполняющего эту услугу. В элементе <operation> указаны входной <input> и выходной <output> параметры услуги.

Затем, элементом <binding> мы указали один способ пересылки посланий — SOAP-послания в процедурном стиле, пересылаемые по протоколу HTTP, на что указывает элемент

```
<soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http" />
```

Если применяется документный стиль SOAP, то в атрибуте style записывается значение "document".

Далее в элементе <binding> повторяется описание операции, но уже в терминах выбранного протокола SOAP.

Наконец, в элементе <service> вложенным элементом <port> связываем элемент <binding> с элементом <address>, указывающим адрес, по которому расположена Web-служба.

В листинге 4.2 имена с префиксом soap конкретизировали описание послания и способы его пересылки. Посмотрим, какие конкретные протоколы предлагает спецификация WSDL 1.2.

Конкретизация описания WSDL

Спецификация WSDL 1.2 задает правила описания адреса Web-службы и предоставляемых ею услуг для протокола SOAP, методов GET и POST протокола HTTP и — для пересылки документов с различными МШЕ-типами. Эти правила записываются дополнительными элементами XML, специфичными для каждого протокола.

Дополнительные элементы протокола SOAP

При использовании протокола SOAP в элемент <binding> и в элемент <service>, точнее, во вложенный в него элемент <port>, документа WSDL вкладывается несколько дополнительных элементов. Для версии WSDL 1.2 они определены в пространстве имен с идентификатором http://www.w3.org/2002/07/wsdl/soap12. Будем считать, что префиксом имен из этого пространства выбрано слово soap.

Непосредственно в элемент <binding> обязательно вкладывается элемент <soap:binding>, определяющий необязательным атрибутом style документный ("document") или процедурный ("rpc") стиль SOAP. По умолчанию принимается документный стиль. Второй, обязательный атрибут transport определяет строкой URI транспортный протокол пересылки SOAP-посланий. Протокол HTTP определяется строкой "http://schemas.xmlsoap.org/soap/http", для других протоколов в спецификации нет никаких обозначений.

В элемент <operation> — тот, что вложен в элемент <binding> — вкладывается необязательный элемент <soap:operation>. Он выбирает документный или процедурный стиль операции необязательным атрибутом style, принимающим значения "rpc" или "document", а также определяет поле заголовка soAPAction необязательным атрибутом soapAction.

В элементы <input> и <output>, вложенные в элемент <operation>, вкладываются элементы <soap:header> И <soap:body>, а В элемент <fault> вкладывается элемент <soap:fault>. В элемент <soap:header> можно вложить один или несколько элементов <soap:headerfault>. У каждого из этих элементов есть обязательный атрибут use, принимающий одно из двух значений — "literal" или "encoded". Он указывает способ сериализации заголовка или содержимого послания.

Значение "encoded" атрибута use говорит о том, что послание использует способы сериализации, описание которых надо искать через атрибуты type, namespace И encodingStyle.

Значение "literal" атрибута use указывает, что способы сериализации определены в схеме документа.

В элемент <port>, вложенный в элемент <service>, вкладывается ровно один элемент <soap:address>, в атрибуте location которого записывается адрес интерфейса Web-службы в виде строки URI.

Примеры всех этих элементов уже приведены в листинге 4.2. В следующем листинге 4.3 показана схема расположения элементов протокола SOAP (они помечены префиксом soap) внутри элементов WSDL. Символы в квадратных скобках имеют то же значение, что и в листинге 4.1.

Листинг 4.3. Схема вложенности элементов протокола SOAP

```
<definitions .... >

  <binding .... >

    <soap:binding style="rpc | document" [?]
                  transport="строка URI" />

    <operation .... >

    <soap:operation soapAction="строка URI" [?]

                    style="rpc | document" [?] /> [?]

      <input>
                     parts="имя" [?]
        <soap:body   use="literal| encoded"
                     encodingStyle="строки URI" [?]
                     namespace="идентификатор" [?] />

        <soap:header message="имя" part="имя"
                  use="literal | encoded"
                  encodingStyle="uri-list"  [?]
                  namespace="идентификатор" [?] > [*]
            <soap:headerfault
```

```
                message="имя"
                part="имя"
                use="literal | encoded"
                encodingStyle="строки URI" [?]
                namespace="идентификатор" [?] /> [*]
        </soap:header>

    </input>

    <output>

        <soap:body parts="имя" [?]
                use="literal | encoded" [?]
                encodingStyle="строки URI" [?]
                namespace="идентификатор" [?] />

        <soap:header message="имя" part="имя"
                use="literal | encoded"
                encodingStyle="строки URI" [?]
                namespace="идентификатор" [?] > [*]
            <soap:headerfault
                message="имя"
                part="имя"
                use="literal | encoded"
                encodingStyle="строки URI" [?]
                namespace="идентификатор" [?] /> [*]
        </soap:header>

    </output>

    <fault>

        <soap:fault name="имя"
                use="literal | encoded"
                encodingStyle="строки URI" [?]
                namespace="идентификатор" [?] />
    </fault>

</operation>
```

```
  </binding>

  <service .... >

     <port .... >
         <soap:address location="адрес в виде строки URI" />
     </port>

  </service>

</definitions>
```

Дополнительные элементы протокола HTTP

Дополнительные элементы протокола HTTP определены в пространстве имен с идентификатором http://www.w3.org/2002/07/wsdl/http. Будем считать, что этим именам дан префикс http.

Для описания пересылки посланий по протоколу HTTP вводятся три дополнительных элемента.

Элемент <http:binding>, вкладываемый непосредственно в элемент <binding>, указывает, что это именно протокол HTTP. У него один атрибут verb, содержащий указание на применяемый HTTP-метод — строку "GET" или строку "POST".

Элемент <http:address>, вкладываемый в элемент <port> элемента <service>, своим единственным атрибутом location определяет абсолютный адрес интерфейса Web-службы в форме строки URI.

Третий элемент <http:operation>, вкладываемый в элемент <operation> элемента <binding>, своим единственным атрибутом location задает относительный адрес URI операции. Вместе с элементом <http:address> этот адрес образует абсолютный адрес Web-услуги.

Кроме этих трех элементов, вводятся еще два элемента-пометки <http:urlEncoded/> и <http:urlReplacement/>. Эти элементы можно поместить в элементы <input> и/или <output> для указания способа кодировки строки запроса GET или POST.

Элемент <http:urlEncoded/> показывает, что запрос послан в MIME-типе "application/x-www-form-urlencoded", то есть, в строке запроса стоят пары "имя=значение", разделенные амперсендами, пробелы заменены плюсами, а специальные символы записаны за знаком процента в 16-ричной форме.

Листинг 4.4. показывает вложенность дополнительных элементов HTTP.

Листинг 4.4. Вложенность дополнительных элементов протокола HTTP

```
<definitions .... >

    <binding .... >

       <http:binding verb="GET | POST" />

       <operation .... >

          <http:operation location="относительный адрес URI" />

           <input .... >
              <http:urlEncoded/> [?]
              <http:urlReplacement/> [?]
           </input>

           <output .... >
              <http:urlEncoded/> [?]
              <http:urlReplacement/> [?]
           </output>

       </operation>

    </binding>

  <service>

    <port .... >
       <http:address location="адрес Web-службы — строка URI" />
    </port>

  </service>

</definitions>
```

Приведем пример. Запрос посылается HTTP-методом GET или POST. Например:

GET `/services/EmpService?empid=123456` HTTP/1.1

Он содержит табельный номер empid сотрудника некоторой фирмы. Web-служба EmpService возвращает клиенту информацию о сотруднике в виде документа XML, содержащего сведения вида

```
<emp xmlns="http://some.com/emp" >

   <пате>Иванов П. С.</пате>
   <age>27</age>
   <position>Инженер</position>

</етр>
```

Описание этих элементов и всей Web-службы приведено в документе WSDL, записанном в листинге 4.5.

Листинг 4.5. Описание WSDL с дополнительными элементами HTTP

```
<definitions
   xmlns="http://www.w3.org/2002/07/wsdl"
   xmlns:http="http://www.w3.org/2002/07/wsdl/http"
   xmlns:tns="http://some.com/emp"
   targetNamespace="http://some.com/emp"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:mime="http://www.w3.org/2002/07/wsdl/mime">

  <types>

    <xsd:schema attributeFormDefault="qualified"
        elementFormDefault="qualified"
        targetNamespace="http://some.com/emp">

      <xsd:element name="emp"

          type="tns:EmpInfoType" />

      <xsd:complexType name="EmpInfoType">

        <xsd:sequence>
          <xsd:element name="name" type="xsd:string" />
          <xsd:element name="age" type="xsd:int" />
```

```
        <xsd:element name="position" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
</types>

<message name="EmpIn">
    <part name="empid" type="xsd:string" />
</message>

<message name="EmpOut">
    <part name="EmpInfo" element="tns:emp" />
</message>

<portType name="EmpPortType">
  <operation name="EmployeeInfo">
      <input message="tns:EmpIn" />
      <output message="tns:EmpOut" />
  </operation>
</portType>

<binding name="EmpGET" type="tns:EmpPortType">
    <http:binding verb="GET" />
    <operation name="Employee">
        <http:operation location="/EmpInfo" />
        <input>
```

```
            <http:urlEncoded/>
         </input>

         <output>
            <mime:mimeXml part="EmpInfo" />
         </output>

      </operation>

   </binding>

   <binding name="EmpPOST" type="tns:EmpPortType">

      <http:binding verb="POST" />

      <operation name="Employee">

         <http:operation location="/EmpInfo" />

            <input>
               <mime:content
                  type="application/x-www-form-urlencoded" />
            </input>

            <output>
               <mime:mimeXml part="EmpInfo" />
            </output>

      </operation>

   </binding>

   <service name="Employee">

     <port name="EmpGETPort" binding="tns:EmpGET">

        <http:address
           location="http://some.com:8080/services/Empservice" />
```

```
    </port>

    <port name="EmpPOSTPort" binding="tns:EmpPOST">

       <http:address

          location="http://some.com:8080/services/EmpService" />

    </port>

  </service>

</definitions>
```

Дополнительные элементы MIME-типов

В описании WSDL, приведенном в листинге 4.5 использованы дополнительные элементы, описывающие MIME-типы. Они определены в пространстве имен с идентификатором http://www.w3.org/2002/07/wsdl/mime. Будем считать, что этим именам дан префикс mime.

Существует всего три дополнительных элемента описания MIME-типа.

Элемент <mime:content> содержит только два атрибута. Необязательный атрибут name задает имя этой части послания, а необязательный атрибут type определяет MIME-тип содержимого, например:

```
<mime:content type="text/xml" />
<mime:content type="image/jpeg" />
```

Отсутствие атрибута type означает "любой тип", то есть, эквивалентно

```
<mime:content type="*/*" />
```

Элемент <mime:multipartRelated> содержит несколько вложенных элементов <mime:part>, которые описывают каждую часть MIME-типа "multipart/related" необязательным атрибутом name. Например:

```
<mime:multipartRelated>

  <mime:part>
     <soap:body parts="contract" use="literal" />
  </mime:part>

  <mime:part>
      <mime:content part="signature" type="image/jpeg"/>
```

```
</mime:part>

</mime:multipartRelated>
```

Третий элемент <mime:mimeXml> описывает MIME-тип, являющийся произвольным документом XML. Необязательный атрибут part ссылается на часть послания, содержащую схему, в которой описан корневой элемент этого документа.

Дополнительные элементы с MIME-типами вкладываются в элементы <input> и/или <output> при описании операции элементом <operation>, вложенным в элемент <binding>. Структура вложенности показана в листинге 4.6.

Листинг 4.6. Вложенность дополнительных MIME-элементов

```
<definitions .... >

  <binding .... >

    <operation .... >

      <input .... >
        <!— Сюда вкладываются дополнительные элементы —>
      </input>

      <output .... >
        <!— Сюда вкладываются дополнительные элементы —>
      </output>

    </operation>

  </binding>

</definitions>
```

Инструменты создания описаний WSDL

Описания WSDL в листингах 4.2 и 4.5 сделаны вручную. Однако строгая формализация языка WSDL позволяет автоматизировать этот процесс. Многие инструментальные средства создания Web-служб содержат утилиты, ко-

торые автоматически создают WSDL-файлы, описывающие готовые Web-службы. Например, уже упоминавшееся в *главе 3* средство создания Web-служб Apache Axis содержит в своем составе класс Java2wsDL, создающий WSDL-файл по классу или интерфейсу Java, описывающему Web-службу. Пакет IBM WSTK, в состав которого входит Axis, содержит утилиту `java2wsdl`, создающую и запускающую объект этого класса. Она работает из командной строки. Достаточно набрать в командной строке

```
$ java2wsdl EchoService
```

и будет создан файл EchoService.wsdl, содержащий по одному элементу <portType> для каждого открытого метода класса, указанного в командной строке. В листинге 4.6 показан этот файл.

Листинг 4.6. Описание Web-службы EchoService, сделанное Axis

```
<?xml version="1.0" encoding="UTF-8" ?>

<wsdl:definitions
   targetNamespace=
"http://localhost:8080/axis/EchoService.jws/axis/EchoService.jws"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:apachesoap="http://xml.apache.org/xml-soap"
   xmlns:impl=
"http://localhost:8080/axis/EchoService.jws/axis/EchoService.jws-impl"
   xmlns:intf=
"http://localhost:8080/axis/EchoService.jws/axis/EchoService.jws"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:types>

    <schema
       targetNamespace="http://www.w3.org/2001/XMLSchema"
       xmlns="http://www.w3.org/2001/XMLSchema">

  <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />

      <element name="any" nillable="true" type="xsd:any" />
```

```
    </schema>

  </wsdl:types>

  <wsdl:message name="getEchoRequest">
     <wsdl:part name="req" type="xsd:any" />
  </wsdl:message>

  <wsdl:message name="getEchoResponse">
     <wsdl:part name="return" type="xsd:any" />
  </wsdl:message>

  <wsdl:portType name="EchoService">

    <wsdl:operation name="getEcho" parameterOrder="req">

       <wsdl:input message="intf:getEchoRequest"
                   name="getEchoRequest" />

      <wsdl:output message="intf:getEchoResponse"
                   name="getEchoResponse" />

  </wsdl:operation>

  </wsdl:portType>

  <wsdl:binding name="EchoServiceSoapBinding"
                type="intf:EchoService">

    <wsdlsoap:binding style="rpc"

         transport="http://schemas.xmlsoap.org/soap/http" />

    <wsdl:operation name="getEcho">

      <wsdlsoap:operation soapAction="" />

      <wsdl:input name="getEchoRequest">

      <wsdlsoap:body encodingStyle=
```

```
            "http://schemas.xmlsoap.org/soap/encoding/"
          namespace=
"http://localhost:8080/axis/EchoService.jws/axis/EchoService.jws"
          use="encoded" />

    </wsdl:input>

    <wsdl:output name="getEchoResponse">

       <wsdlsoap:body encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/"
          namespace=
"http://localhost:8080/axis/EchoService.jws/axis/EchoService.jws"
          use="encoded" />

    </wsdl:output>

   </wsdl:operation>

 </wsdl:binding>

 <wsdl:service name="EchoServiceService">

   <wsdl:port binding="intf:EchoServiceSoapBinding"
              name="EchoService">

     <wsdlsoap:address location=

            "http://localhost:8080/axis/EchoService.jws" />

   </wsdl:port>

 </wsdl:service>

</wsdl:definitions>
```

Интересно, что Axis может выполнить и обратное действие: по имеющемуся WSDL-файлу класс wsDL2java создаст все классы Java, необходимые для работы Web-службы. В пакете IBM WSTK класс `WSDL2java` можно вызвать из командной строки утилитой `wsdl2java`. Например:

```
$ wsdl2java EchoService.wsdl
```

Такие же утилиты есть в составе уже упоминавшегося продукта GLUE фирмы The Mind Electric.

Пакет Microsoft SOAP Toolkit содержит графическую утилиту `wsdlgen3.exe`, вызываемую из стартового меню **WSDL Generator,** и утилиту командной строки `wsdlstb3.exe`, которые создают WSDL-файлы.

Фирма Sun Microsystems готовит к выпуску пакет интерфейсов JWSDL (Java API for WSDL), преобразующих описание WSDL в классы Java и обратно. Это позволяет программно создавать, изменять, читать описания WSDL. Фирма IBM уже реализовала этот пакет в своем продукте WSTK, назвав набор интерфейсов и реализующих их классов WSDL4J (WSDL for Java). Этот набор можно использовать в составе WSTK или отдельно, загрузив его с сайта **http://www-124.ibm.com/developerworks/projects/wsdl4j/.** Мы рассмотрим его подробнее в следующем разделе.

Самое ценное в описаниях WSDL то, что клиент Web-службы может обратиться не к ней самой, а к ее WSDL-описанию. В состав GLUE входит утилита командной строки invoke, обращающаяся к Web-службе по ее WSDL-описанию. Например, достаточно набрать в командной строке:

```
$ invoke http://some.com:8080/services/EchoService.wsdl  Эхо
```

и на консоли появится ответ Web-службы.

Фирма IBM выпускает пакет классов WSIF (Web Services Invocation Framework), работающий в Web-контейнере Tomcat под управлением Apache SOAP. С помощью этого пакета можно, в частности, сделать ту же работу:

```
$ java clients.DynamicInvoker \
   http://some.com:8080/services/EchoService.wsdl \
   getEcho  Эхо
```

Напомним, что обратная наклонная черта здесь означает продолжение командной строки на следующую строку текста.

После окончания работы класса DynamicInvoker **на консоли появляются** сообщения WSIF и ответ сервера.

Пакет JWSDL и его реализация WSDL4J

Пакет JWSDL состоит из интерфейсов и классов, собранных в пакет `javax.wsdl` и несколько его подпакетов. Они реализованы в пакете com. ibm. `wsdl` и его подпакетах, входящих в пакет IBM WSDL4J.

В пакете javax.wsdi содержатся интерфейсы, описывающие различные **XML-элементы WSDL-ДОКумента:** Definition, Types, Message, PortType, Binding, Service, **И вложенные В НИХ** элементы: Operation, Part, Input,

Output, Port, BindingOperation, `BindingInput`, `BindingOutput`, BindingFault. Методы `getXxx()` этих интерфейсов позволяют читать соответствующие элементы, а методы setxxx () — записывать их.

В интерфейсе Definition, кроме того, есть методы createXxxO, создающие объекты этих интерфейсов, например,

```
Message msg = def.createMessage();
```

Методы `addXxx()` добавляют созданные объекты к документу **WSDL,** например:

```
def.addMessage(msg);
```

Сам же объект типа Definition создается фабричным методом newDefinition() класса `WSDLFactory` ИЗ пакета javax.wsdl. factory:

```
Definition def = fact.newDefinition();
```

Экземпляр класса WSDLFactory, в свою очередь, создается статическим методом `newInstance` о , следующим образом:

```
WSDLFactory fact = WSDLFactory.newInstance();
```

Этот метод отыскивает класс-фабрику, сначала просматривая системное СВОЙСТВО `javax.wsdl.factory.WSDLFactory`, ПОТОМ файл jre/lib/wsdl.properties, потом загружает фабрику по умолчанию. Можно загрузить другую фабрику классов, воспользовавшись вторым методом:

```
public static WSDLFactory newInstance(String factory);
```

Кроме объекта типа Definition, класс-фабрика WSDLFactory создает объекты типа `WSDLReader`, читающие описание **WSDL,** и объекты типа wsDLWriter, записывающие заранее подготовленный документ **WSDL.** Интерфейсы WSDLReader И WSDLWriter содержатся В пакете `javax.wsdl.xml`. Создание объектов выполняется методами newWSDLReader () и newwsDLWriter () следующим образом:

```
WSDLReader reader = fact.newWSDLReader();
WSDLWriter writer = fact.newWSDLWriter();
```

После создания объекта reader можно определить его свойства методом

```
public void setFeature(String name, boolean value);
```

Спецификация **WSDL** определяет два свойства объекта типа `WSDLReader`:

- если значение свойства `"javax.wsdl.verbose"` равно true, то объект выводит свои сообщения в стандартный вывод `System.out`;
- если свойство `"javax.wsdl.importDocuments"` равно true, то все импортированные документы будут обработаны, если false — то они игнорируются.

По умолчанию оба свойства равны true.

После того как объект reader получен, с помощью его методов readWSDL() можно получить объект типа Definition, заполненный содержимым прочитанного документа WSDL, например:

```
Definition def = rdr.readWSDL("sample.wsdl");
```

В листинге 4.7 приведен пример программы, читающей и разбирающей средствами пакета JWSDL файл sample.wsdl, содержащий документ WSDL, и выводящий встреченные в нем сообщения и имена в стандартный вывод.

Листинг 4.7. Чтение документа WSDL

```
import javax.wsdl.*;
import javax.wsdl.factory.*;
import javax.wsdl.xml.*;

public class ReadWSDL{

  public static void main(String[] args){

    try{
      WSDLFactory fact = WSDLFactory.newInstance();

      WSDLReader reader = fact.newWSDLReader();
      reader.setFeature("javax.wsdl.verbose", false);
      reader.setFeature("javax.wsdl.importDocuments", true);

      Definition def = reader.readWSDL(null, "sample.wsdl");

      Service service = def.getService(
                new QName("http://some.com/wsdl",
                          "SomeService"));

      Port port = service.getPort("SomePort");

      Binding binding = port.getBinding();

      PortType portType = binding.getPortType();

      Map messages = def.getMessages();
```

```
        Iterator msgIt = messages.values O .iterator();

        while (msgIt.hasNext()){

           Message msg = (Message)msgIt.next();

           if (!msg.isUndefined())
              System.out.println(msg.getQName());
        }

        List operations = portType.getOperations();
        Iterator opIt = operations.iterator();

        while (opIt.hasNext()){

           Operation operation = (Operation)opIt.next();

           if (!operation.isUndefined())
              System.out.println(operation.getName());
        }
     }catch(WSDLException e){
          System.err.println(e);
     }
  }
}
```

Листинг 4.8 показывает процесс создания описания WSDL средствами пакета **JWSDL.**

'. Листинг 4.8. Создание документа WSDL

```
import javax.wsdl.*;
import javax.wsdl.factory.*;

import javax.wsdl.xml.*;

public class WriteWSDL{

                                        {
  public static void main(String[] args)

    try{
```

```
WSDLFactory fact = WSDLFactory.newInstance();
WSDLWriter writer = fact.newWSDLWriter();
writer.writeWSDL(def, System.out);

Definition def = fact.newDefinition() ;

Part part1 = def.createPart();
Part part2 = def.createPart();

Message msg1 = def.createMessage();
Message msg2 = def.createMessage();

Input input = def.createInput();
Output output = def.createOutput();

Operation operation = def.createOperation();
PortType portType = def.createPortType();

String tns = "http://some.com/wsdl";
def.setQName(new QName(tns, "SomeService"));
def.setTargetNamespace(tns);
def.addNamespace("tns", tns);

String xsd = "http://www.w3.org/2001/XMLSchema";
def.addNamespace("xsd", xsd);

part1.setName("symbol");
part1.setTypeName(new QName(xsd, "string"));

msg1.setQName(new QName(tns, "getQuoteInput"));
msg1.addPart(part1);
msg1.setUndefined(false);

def.addMessage(msg1);

part2.setName("quote");
part2.setTypeName(new QName(xsd, "float"));

msg2.setQName(new QName(tns, "getQuoteOutput"));
```

```
        msg2.addPart(part2);
        msg2.setUndefined(false);

        def.addMessage(msg2) ;

        input.setMessage(msg1);
        output.setMessage(msg2) ;

        operation.setName("getQuote") ;
        operation.setInput(input);
        operation.setOutput(output);
        operation.setUndefined(false);

        portType.setQName(new QName(tns, "GetQuote"));
        portType.addOperation(operation) ;
        portType.setUndefined(false) ;

        def.addPortType(portType);

    }catch(WSDLException e){
          System.err.println(e);
    }
  }
}
```

ГЛАВА 5

Регистрация Web Services в реестре UDDI

В *главе 2* мы вкратце рассмотрели порядок регистрации и поиска Web-служб в различных системах поиска и обнаружения информации. Наиболее распространены две системы: универсальная система описания, обнаружения и интеграции UDDI и электронный бизнес-реестр ebXML Registry (electronic business XML Registry). Обе системы, особенно ebXML, предназначены для предоставления информации не только о Web-службах, но и о любом другом бизнесе. Поэтому они хранят информацию не столько об услугах, сколько о фирме, предоставляющей товары и услуги: ее название, логотип, устав, контактные телефоны, образцы договоров, прейскуранты и прочую деловую информацию.

Схема взаимодействия клиента Web-службы с ее поставщиком через реестр показана на рис 2.3.

Система описания, обнаружения и интеграции UDDI создана фирмами **IBM (http://www-3.ibm.com/services/uddi/)** и Microsoft **(http://uddi.microsoft.com/).** Сейчас она развивается группой крупных компаний, среди которых активное участие принимают, кроме IBM и Microsot, компании Oracle, Hewlett-Packard и SAP AG. На официальном сайте сообщества UDDI **http://www.uddi.org/** приведен список около трехсот компаний-участников проекта. Сообщество выпустило несколько спецификаций, описывающих требования к UDDI, которые можно получить на том же сайте. Спецификации уже реализованы множеством продуктов разработки реестра UDDI.

Реестр UDDI (UDDI Business Registry) состоит из множества *узлов* (nodes), размещенных в Интернете. Они хранят информацию о Web-службах, доступную на всех узлах, образующих распределенный UDDI-реестр. Клиент "видит" UDDI-реестр как единое целое, совершенно не ощущая того, что он размещен на нескольких машинах. Конечно, сами узлы можно организовать

как Web-службы, а реестр UDDI — как слабо связанное распределенное приложение.

Многие крупные компании организовали и содержат свои UDDI-реестры. Наибольшей известностью пользуются следующие реестры:

- реестр фирмы IBM, расположенный по адресу **https://www-3.ibm.com/services/uddi/v2beta/protect/registry.html**;
- реестр компании Hewlett Packard, его адрес **https://uddi.hp.com/uddi/index.jsp**;
- реестр корпорации Microsoft находится на сайте **https://uddi.rte.microsoft.com/register.aspx**;
- реестр фирмы SAP AG расположен по адресу **http://udditest.sap.com/**.

Эти UDDI-реестры связаны между собой и постоянно обмениваются информацией. Кроме того, это *открытые* (public) реестры. Любой желающий может зарегистрировать в них свою Web-службу или отыскать нужную Web-услугу. Фирмы могут организовать и закрытые *частные* (private) реестры, доступные только зарегистрированным участникам. Список UDDI-реестров можно посмотреть на сайте проекта **UDDI http://www.uddi.org/solutions.html#registrars**.

Состав реестра UDDI

Реестр UDDI разбивает хранящуюся в нем информацию на несколько групп.

Четыре основные группы состоят из следующих отдельных документов или частей одного документа.

- Бизнес-информация — документ с корневым элементом <businessEntity> — описание фирмы-поставщика Web-услуг: ее ключ UUID (Unique Universal Identifier), уникальный в пределах реестра и описанный атрибутом businessKey, название фирмы — вложенный элемент <name>, краткое описание сферы ее деятельности, типы предоставляемых услуг, контактная информация, ссылки URL. Эта информация предназначена для всех, кто хочет воспользоваться услугами фирмы.
- Бизнес-услуги — элемент <businessServices>, вложенный в элемент <businessEntity> - список услуг, оказываемых фирмой. Каждая услуга описывается вложенным элементом <businessservice>. В описание входит ключ UUID каждой услуги, описанный атрибутом serviceKey, имя услуги — вложенный элемент <name>, ее краткое описание и ссылки на

на подробную информацию. Услуги могут быть любыми, не обязательно Web-услугами.

- Указатели на услуги — элемент <bindingTemplates>, вложенный в элемент <businessservice> — способы получения каждой услуги. Они могут быть прямыми, например, URL-адрес Web-службы, или косвенными, например, описание WSDL или IDL. Каждый способ получения услуги описывается одним вложенным элементом <bindingTemplate>. Его атрибут bindingKey определяет уникальный ключ UUID указателя. Элемент <bindingTemplate> содержит ссылку на соответствующий элемент <tModel>.
- Модель услуги — элемент <tModel> (technical Model) — подробное формальное описание каждой услуги. Оно используется программным обеспечением узла. Обычно это отдельный документ XML.

В реестре есть еще несколько дополнительных элементов.

- Утверждение — элемент <publisherAssertion> — описание установленных ранее отношений между фирмами (утверждение "peer-peer") или фирмой и ее подразделениями (утверждение "parent-child"). Фирма *утверждает,* что она тесно связана с перечисляемыми фирмами или что это — ее подразделения. Третий вид утверждения — "identity" — отношение между одинаковыми фирмами — это фактически псевдоним. Описание утверждения выполняется вложенными элементами <fromKey> и <toKey>, а вид отношения описывается вложенным элементом <keyedReference>. Отношение входит в силу, когда его утвердят оба участника. Это отдельный документ, использующий элементы <businessEntity> обеих фирм.
- Информация — элемент <operationalInfo> — дата создания и последней модификации записи в реестре, идентификатор узла реестра, идентификатор владельца информации.

❑ Подписка — элемент <subscription> — список фирм и сведений, которые надо послать перечисленным фирмам при каких-либо изменениях в деятельности фирмы.

Эти элементы определены в пространстве имен UDDI. Идентификатор пространства имен UDDI версии 3.0 равен "urn:uddi-org:api_v3".

Уникальный ключ UUID, встречающийся в этих элементах, имеет примерно такой вид: "uddi:example.com: 1" или "uddi:example.com:sales-division:53". Устаревшая форма записи UUID выглядит примерно так: "4CD7E4BC-648B-426D-9936-443EAAC8AE23".

В листинге 5.1 приведена схема документа XML, хранящегося в UDDI-реестре и описывающего бизнес-информацию. В этой схеме показаны не все, а только основные элементы первого уровня вложенности. В листинге 5.1 использованы те же пометки, что и в *главе 4:*

- символ [?] означает, что элемент или атрибут может появиться в документе нуль или один раз;
- символ [*] означает, что элемент может появиться нуль или несколько раз;
- символ [+] означает, что элемент может появиться один или несколько раз;
- отсутствие символа в квадратных скобках означает, что атрибут должен появиться ровно один раз.

Листинг 5.1. Основные элементы UDDI-описания бизнес-информации

```
<businessEntity businessKey="ключ UUID" [?] >

   <name lang="язык" [?] >имя</name> [+]

   <buslnessServices>  [?]

      <businessService serviceKey="ключ UUID" [?] > [+]

         <bindingTemplates> [?]

           <bindingTemplate bindingKey="ключ UUID" [?] > [+]
               <!— Описание указателя —>
           </bindingTemplate>

         </bindingTemplates>

      </businessService>

   </businessServices>

</businessEntity>
```

Рассмотрим подробнее каждый элемент. У элементов <businessServices> и <bindingTemplates> нет атрибутов, они содержат только один или несколько вложенных элементов <businessService> ИЛИ <bindingTemplate> соответственно.

Структура остальных элементов сложнее. Мы рассмотрим их в следующих разделах.

Элемент <businessEntity>

У элемента <businessEntity> есть один необязательный атрибут businessKey, уже показанный в листинге 5.1. Этот атрибут определяет уникальный ключ UUID бизнес-информации. Если он отсутствует, то ключ UUID генерируется реестром UDDI.

В элемент <businessEntity> вкладывается восемь элементов, из них обязателен только один элемент <name>. Ниже перечислены эти элементы.

- Элемент <discoveryURLs> содержит один или несколько вложенных элементов <discoveryURL>, содержащих ссылки на файлы, в которых расположено описание бизнес-информации.
- Элементы <name>, которые могут встретиться несколько раз, содержат названия фирмы в полном и сокращенном виде на разных языках.
- Элемент <description>, содержащий произвольное описание бизнес-информации, тоже может встретиться несколько раз, например, на разных языках.
- Элемент <contacts> содержит один или несколько вложенных элементов <contact>, содержащих, в свою очередь, несколько вложенных элементов, описывающих контактную информацию:
  - нуль или несколько элементов <description> с произвольным описанием контактной информации на разных языках;
  - один или несколько элементов <personName> с именами или должностями партнеров;
  - нуль или несколько элементов <phone> с номерами контактных телефонов;
  - нуль или несколько элементов <email> с адресами электронной почты;
  - нуль или несколько элементов <address>, содержащих почтовые адреса фирмы.
- Уже описанный элемент <businessService>.
- Элемент <identifierBag> содержит идентификаторы бизнес-информации в различных системах идентификации или классификации. Они записываются в атрибутах одного или нескольких вложенных элементов <keyedReference>. У этих элементов два обязательных атрибута: tModelKey, содержащий ключ UUDI элемента <tModel>, описывающего систему идентификации, и keyValue со значением идентификатора. Необязательный атрибут keyName содержит произвольное краткое имя идентификатора. Это имя должно быть уникальным в пределах данной

системы идентификации. Таким образом, значение атрибута `tModelKey` работает аналогично идентификатору пространства имен, обеспечивая уникальность имени.

- Элемент <categoryBag> тоже содержит один или несколько элементов `<keyedReference>`, описывающих некоторые категории бизнеса, например, его разделение по географическим районам. Он может содержать НУЛЬ ИЛИ НеСКОЛЬКО элементов `<keyedReferenceGroup>` С атрибутом tModeiKey И ВЛОЖеННЫМИ элементами `<keyedReference>`. В ОДНу Группу собирается неделимая информация, например, долгота и широта географического пункта. В отличие от элемента <identifierBag>, элемент <categoryBag> относит имена `keyName` не к определенной системе идентификации, а к некоторой категории, например, виду продукции, определяемому атрибутом tModeiKey.

- Бизнес-информация может быть подписана цифровыми подписями, расположенными В элементах <Signature>.

Как видите, элемент <businessEntity> содержит обширную и всестороннюю информацию о зарегистрированной в реестре UDDI фирме. В листинге 5.2 приведен пример такой информации.

Листинг 5.2. Пример бизнес-информации

```
<businessEntity

  businessKey="677cfa1a-2717-4620-be39-6631bb74b6e1">

 <discoveryURLs>

   <discoveryURL useType="businessEntity">

     http://uddi.hp.com/discovery?

   ↳businessKey=677cfa1a-2717-4620-be39-6631bb74b6e1

   </discoveryURL>

 </discoveryURLs>

 <name xml:lang="en">Weather Service</name>

 <description xml:lang="RU-ru">

     Интерактивная метеослужба предоставляет прогноз погоды
```

```
    на завтра в указанном населенном пункте.
  </description>
  <contacts>
    <contact useType="Technical support">
      <description xml:lang="RU-ru">
          Контактная информация метеослужбы.
      </description>
      <personName>оператор И. П. Сидоров</personName>
      <phone>234-45-67</phone>
      <phone>234-45-38</phone>
      <email>sidor@some.com</email>
      <address>318123 Жилки, Ягодная, 23-6</address>
    </contact>
  </contacts>
  <businessServices>
      <!— Сюда вкладываются описания бизнес-услуг. —>
  </businessServices>
  <identifierBag>
    <keyedReference
      tModelKey="uddi:ubr.uddi.org:identifier:dnb.com:D-U-N-S"
      keyName="HP"
      keyValue="31-626-8655" />
  </identifierBag>
  <categoryBag>
```

```
    <keyedReference
       tModelKey="uddi:c0b9fe13-179f-413d-8a5b-5004db8e5bb2"
       keyValue="61" />

    <keyedReference
       tModelKey="uuid:c0b9fe13-179f-413d-8a5b-5004db8e5bb2"
       keyValue="51419" />

</categoryBag>

</businessEntity>
```

Элемент <businessService>

У элемента <businessService>, описывающего отдельную бизнес-услугу, два необязательных атрибута serviceKey И businessKey.

Атрибут serviceKey содержит уникальный ключ UUID описываемой услуги. Если он отсутствует, то ключ будет сгенерирован реестром UDDI.

Атрибут businessKey содержит ключ UUID соответствующего элемента <businessEntity>. Его можно не писать, если элемент <businessService> явно вложен в элемент <businessEntity>.

В элемент <businessService> можно вложить пять необязательных элементов. Это имя, записанное в элементе <name>, описание, сделанное в элементе <description>, уже описанный элемент <bindingTemplates> И уже знакомые нам элементы <categoryBag> И <Signature>.

В листинге 5.3 приведен пример описания бизнес-услуги.

Листинг 5.3. Пример описания бизнес-услуги

```
<businessService
   serviceKey="d8091de4-0a4a-4061-9979-5dl9131aece5"
   businessKey="677cfala-2717-4620-be39-6631bb74b6el">

  <name xml:lang="en">Meteo Service</name>

  <description xml:lang="RU-ru">

     Прогноз погоды на завтра.

  </description>
```

```
  <bindingTemplates>

    <!— Сюда вкладываются описания указателей. —>

  </bindingTemplates>

</businessService>
```

Элемент <bindingTemplate>

В открывающем теге элемента <bindingTemplate>, описывающего сетевой адрес и другие способы получения услуги, два необязательных атрибута bindingKey И serviceKey.

Атрибут bindingKey содержит уникальный ключ UUID элемента. Если этот атрибут отсутствует, то его значение генерируется реестром UDDI.

Атрибут serviceKey содержит ключ UUID элемента <businessService>, В котором он содержится.

В элемент <bindingTemplate> вкладывается нуль или несколько описаний <description> на разных языках и ровно один обязательный элемент <accessPoint>, содержащий адрес описываемой Web-службы, обычно в виде строки URL или электронной почты. В прежних версиях UDDI вместо него использовался элемент <hostingRedirector>, который оставлен в версии UDDI 3.0 для совместимости со старыми версиями.

Кроме того, в элемент <bindingTemplate> можно вложить один из трех необязательных элементов. Два из них — уже знакомые нам элементы <categoryBag> И <Signature>.

Третий элемент <tModelInstanceDetails> содержит один или несколько вложенных элементов <tModelInstanceInfo>, ссылающихся на соответствующее описание <tModel> обязательным атрибутом tModelKey. Кроме того, В элемент <tModelInstanceInfo> МОЖНО ВЛОЖИТЬ описания <description> И дополнительные сведения <instanceDetails>. Элемент <instanceDetails>, в свою очередь, содержит или элементы <overviewDoc>, содержащие ссылки на дополнительные описания в виде URL, или элемент <instanceParms>, содержащий дополнительные параметры.

В листинге 5.4 приведен пример описания способов получения услуги.

Листинг 5.4. Пример описания UDDI способов получения услуги

```
<bindingTemplate
    bindingKey="942595d7-0311-48b7-9c65-995748a3a8af"
```

```
    serviceKey="d8091de4-0a4a-4061-9979-5d19131aece5">
  <accessPoint URLType="http">
     http://some.com/services/MeteoService
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
        tModelKey="uddi:42fab02f-300a-4315-aa4a-f97242ff6953">
      <instanceDetails>
        <overviewDoc>
          <overviewURL>
            http://some.com/services/MeteoService?WSDL
          </overviewURL>
        </overviewDoc>
      </instanceDetails>
    </tModelInstanceInfo>
  </tModelInstanceDetails>
</bindingTemplate>
```

Элемент <*tModel*>

Открывающий тег элемента `<tModel>`, описывающего технические детали представления бизнес-услуги, может содержать два необязательных атрибута: `tModelKey`, содержащий уникальный ключ UUID, и deleted, принимающий значения "true" (по умолчанию), или "false", говорящее о том,

что описание логически удалено. Отсутствие атрибута tModelKey означает, что ключ UUID будет сгенерирован реестром UDDI.

В элемент <tModel> вкладывается один обязательный элемент <name> — имя описания в виде строки URI — и пять необязательных элементов: <description>, <overviewDoc>, <identifierBag>, <categoryBag> И <signature>. Их содержание и смысл описаны выше.

В листинге 5.5 приведен пример описания технических деталей.

Листинг 5.5. Описание UDDI технических деталей

```
<tModel
   tModelKey="uddi:42fab02f-300a-4315-aa4a-f97242ff6953">
  <name>Meteo Service Interface</name>
  <description xml:lang="RU-ru">
    Интерфейс метеослужбы.
  </description>
  <overviewDoc>
    <description xml:lang="RU-ru">
       Описание WSDL метеослужбы.
    </description>
    <overviewURL>
       http://some.com/services/ifaces/meteoservice.wsdl
  </overviewURL>
  </overviewDoc>
  <categoryBag>
     <keyedReference
```

```
        tModelKey="uddi:c1acf26d-9672-4404-9d70-39b756e62ab4"
        keyValue="wsdlSpec" />

    < keyedReference
        tModelKey="uddi:c0b9fe13-179f-413d-8a5b-5004db8e5bb2"
        keyValue="514191" />

  </categoryBag>

</tModel>
```

Элемент <*publisherAssertion*>

Элемент <publisherAssertion>, описывающий связи между фирмами, устроен проще всего. В него ровно один раз вкладываются три элемента <fromKey>, <toKey> и <keyedReference> и может присутствовать одна или несколько цифровых подписей, занесенных в элементы <signature>.

Элемент <keyedReference> уже встречался выше. Его атрибут tModelKey содержит ссылку на полное описание связи между фирмами.

Элементы <fromKey> и <toKey> просто содержат ключи UUID связанных между собой фирм, как показано в листинге 5.6.

Листинг 5.6. Описания связи между фирмами

```
<publisherAssertion>

   <fromKey>677cfa1a-2717-4620-be39-6631bb74b6e1</fromKey>

   <toKey>677cfa1a-2717-4620-be27-6631bb45b34</toKey>

   <keyedReference
      tModelKey='uddi:807a2c6a-ee22-470d-adc7-e0424a337c03'
      keyValue='peer-peer'/>

</publisherAssertion>
```

Программный интерфейс UDDI

Клиент обращается к UDDI-реестру для того, чтобы зарегистрировать свою Web-службу, изменить ее, или для того, чтобы отыскать нужную Web-

услугу. Реестр предоставляет необходимый для этого интерфейс. Функции, входящие в UDDI API, можно разделить на четыре группы:

- регистрация Web-службы и ее последующие изменения выполняются функциями вида save_xxx;
- описание Web-службы можно получить функциями вида `get_xxxDetails`;
- для поиска Web-службы применяются функции вида `find_xxx`;
- услуга удаляется функциями вида `delete_xxx`.

Здесь под символами "xxx" понимается тип объекта, например, "business", "service", "binding" или "tModel".

Хотя спецификация **UDDI** и называет объекты, входящие в состав **API,** "функциями", на самом деле это элементы XML. То, что спецификация называет "аргументами функций" — это атрибуты открывающих тегов или вложенные элементы XML. В дальнейшем мода на XML может пройти, и UDDI API можно будет реализовать функциями языка C или методами языка Java, или чем-нибудь более модным.

Клиент реестра UDDI оформляет запрос в виде элемента XML, например, поиск Web-услуги по имени может выглядеть так:

```
<find_service businessKey="" generic="3.0" xmlns="urn:uddi-org:api_v3">

   <name>Meteo Service</name>

</find_service>
```

Этот элемент помещается в тело SOAP-послания и отправляется Web-службе, обслуживающей реестр UDDI. Она отыскивает все Web-службы с именем "Meteo Service" и посылает клиенту их ключи UUID в SOAP-послании, содержащем список найденных Web-служб в элементе <serviceList>, например:

```
<serviceList generic="3.0" operator="HP"
             truncated="false"
             xmlns="urn:uddi-org:api_v3">

   <serviceInfos>

      <serviceInfo  serviceKey="7cl9ab87-lbc5-4fb6-841b-a4149a802a71"
                    businessKey="">

         <name xml:lang="en">Meteo Service</name>
```

```
    </serviceInfo>

  </serviceInfos>

</serviceList>
```

Функции API делятся на две группы:

- Inquiry API — функции, запрашивающие информацию в реестре;
- Publish API — функции, регистрирующие Web-службу, удаляющие ее из реестра или изменяющие информацию о ней.

Рассмотрим подробнее состав каждого API.

Функции запроса информации

В состав Inquiry API входят 9 функций — элементов XML, отыскивающих информацию в реестре. У них может быть три необязательных атрибута:

- атрибут serviceKey содержит ключ UUID того элемента реестра, в котором следует производить поиск;
- атрибут `maxRows` ограничивает число строк, возвращаемых функцией;
- атрибут listHead задает номер пункта, с которого надо начать возвращаемую информацию.

У многих функций есть аргумент (вложенный элемент) `<findQualifiers>`. Он задает условия поиска, например, можно запросить сортировку результата поиска вложенными элементами <binarySort>, <caseSensitiveSort>, <caseInsensitiveSort>, `<sortByNameAsc>`, `<sortByNameDesc>`, <sortByDateAsc>, <sortByDateDesc>.

Перечислим функции Inquiry API.

- Функция find_business отыскивает элементы <businessEntity> С ПОМОЩЬЮ элементов (аргументов) `<authInfo>`, <categoryBag>, <discoveryURLs>, `<identifierBag>`, <name>, `<findQualifiers>`, `<find_tModel>`, `<find_relatedBusinesses>`, <tModelBag>. Результат поиска — найденные элементы `<businessEntity>` — отправляется клиенту в элементе (SOAP-структуре) <businessList>.
- Функция find_relatedBusinesses ищет бизнес-информацию О фирмах, связанных с фирмой, указанной в запросе, аргументами `<authInfo>`, <findQualifiers>, <keyedreference>, ОДНИМ ИЗ трех элементов <fromKey>, <toKey>, `<businessKey>`. Результат поиска возвращается клиенту В элементе <relatedBusinessesList>.

- **Функция** findservice **отыскивает элементы** <businessService> **по признакам** <authInfo>, <categoryBag>, <name>, <findQualifiers>, <find_tModel>, <tModelBag>. **Результат поиска возвращается в элементе** <serviceList>.
- **Функция** find_binding **осуществляет поиск элементов** <bindingTemplate> **по признакам, указываемым вложенными элементами (аргументами функции)** <authInfo>, <categoryBag>, <findQualifiers>, <find_tModel>, <tModelBag>. **Результат поиска, то есть найденные элементы** <bindingTemplate>, **возвращается в элементе (SOAP-структуре)** <bindingDetail>.
- **Функция** find_tModel **ищет элементы** <tModel> **по признакам, указанным элементами** <authInfo>, <categoryBag>, <identifierBag>, <name>, <findQualifiers>. **Результат возвращается в элементе** <tModelList>.
- **Функция** get_businessDetail **отыскивает элементы** <businessEntity> **по аргументу** businessKey, **содержащему ключи UUID отыскиваемых элементов. Она отсылает клиенту структуру** <businessDetail>.
- **Функция** get_serviceDetail **ищет элементы** <businessservice> **ПО аргументу** serviceKey, **содержащему ключи UUID отыскиваемых элементов. Она отсылает** клиенту структуру <serviceDetail>.
- **Функция** get_bindingDetail ОТЫСКИВаеТ **элементы** <bindingTemplate> **по аргументу** bindingKey, **содержащему ключи UUID отыскиваемых элементов. Она отсылает клиенту структуру** <bindingDetail>.
- **Функция** get_tModelDetail **осуществляет поиск элементов** <tModel> **по аргументу** tModelKey, **содержащему ключи UUID отыскиваемых элементов. Она отсылает клиенту структуру** <tModelDetail>.
- Функция get_operationalInfo ищет **элементы** <operationalInfo> ПО **аргументу** entityKey, **содержащему ключи UUID отыскиваемых элементов. Она отсылает клиенту структуру** <operationalInfos>.

Функции регистрации и модификации Web-службы

Функции Publish API регистрируют, добавляют, изменяют и удаляют информацию в реестре UDDI, а также выдают служебную информацию о зарегистрированных Web-службах. Перечислим эти функции.

- **Функция** save_business **регистрирует** НОВЫЕ **элементы** <businessEntity> — **аргументы функции, или изменяет существующие элементы.**
- **Функция** delete_business **удаляет элементы** <businessEntity>, **заданные аргументом** businessKey, **содержащим ключи UUID удаляемых элементов.**

- Функция save_service регистрирует новые элементы <businessService> — аргументы функции, или изменяет существующие элементы.
- Функция delete_service удаляет элементы <businessService>, заданные аргументом serviceKey, содержащим ключи UUID удаляемых элементов.
- Функция save_binding регистрирует новые элементы <bindingTemplate> — аргументы функции, или изменяет существующие элементы.
- Функция delete_binding удаляет элементы <bindingTemplate>, заданные аргументом bindingKey, содержащим ключи UUID удаляемых элементов.
- Функция save_tModel регистрирует новые элементы <tModel> — аргументы функции, или изменяет существующие элементы.
- Функция delete_tModel удаляет элементы <tModel>, заданные аргументом tModelKey, содержащим ключи UUID удаляемых элементов.
- Функция add_publisherAssertions добавляет элементы <publisherAssertion> — аргументы функции.
- Функция set_publisherAssertions обновляет все элементы <publisherAssertion>.
- Функция delete_publisherAssertions удаляет один или несколько элементов <publisherAssertion>.
- Функция get_publisherAssertions возвращает клиенту элемент <publisherAssertions>, содержащий список фирм, связанных с данной фирмой.
- Функция get_assertionStatusReport возвращает клиенту информацию о связях между фирмами в виде элемента-структуры <assertionStatusReport>.
- Функция get_registeredInfo выдает сокращенную информацию о зарегистрированных данным клиентом Web-службах.

Реализации UDDI API

Функции, входящие в UDDI API, предоставляют все средства для работы клиента с реестром UDDI, но записывать элементы XML, реализующие эти функции — тяжелая и кропотливая работа. Для того чтобы облегчить эту работу, создаются библиотеки классов, автоматически создающих SOAP-послания с запросами к реестру UDDI.

Пакет IBM UDDI4J

Есть уже много библиотек классов и отдельных функций, написанных на различных языках. В технологии Java чаще всего используется пакет классов UDDI4J (UDDI for Java), разработанный фирмой IBM **(http://www.uddi4j.org/).** Пакет UDDI4J входит в состав IBM WSTK. Классы, входящие в пакет UDDI4J, обеспечивают работу клиента в UDDI-реестре.

Все обращения к реестру UDDI идут через класс-посредник `UDDIProxy`, реализованный в пакете UDDI4J. Класс UDDiProxy преобразует эти обращения в функции UDDI API. Методы этого класса называются так же, как и функции UDDI API: save_business (), `find_service()` И так далее. Они возвращают объекты классов, представляющих SOAP-структуры: `ServiceList`, `BusinessList`, `BindingDetail`, `TModelList` И другие классы, собранные В пакет `com. ibm.uddi. response`.

Аргументами методов класса `UDDIProxy`, дублирующих функции UDDI API, служат объекты классов, представляющих аргументы функций UDDI API: `AuthInfo`, `FindQualifiers`, `FindQualifier`, `IdentifierBag`, `CategoryBag`, `TModelBag`, `BindingKey`, `BusinessKey`, `ServiceKey` И другие классы, собранные В пакет `com.ibm.uddi.util`.

С помощью перечисленных методов и классов можно полностью смоделировать работу UDDI API.

Другой способ использования UDDI API, предлагаемый пакетом UDDI4J, — сначала смоделировать функцию UDDI API объектом одного ИЗ классов SaveBusiness, SaveService, SaveTModel, FindBusiness, FindService, DeleteBusiness, GetServiceDetail ИЛИ другого класса ИЗ пакета `com.ibm.uddi.request`. Все эти классы расширяют абстрактный класс `UDDIElement`.

После того как объект-запрос создан, он посылается реестру методом

```
public org.w3c.dom.Element send(UDDIElement el, boolean inquiry);
```

Метод возвращает результат запроса в виде дерева элементов XML, которое затем надо разобрать каким-нибудь DOM-парсером.

При создании объекта-запроса полезны классы, моделирующие элементы реестра UDDI: BusinessEntity, BusinessServices, BusinessService, BindingTemplates, `BindingTemplate`, TModel И множество Других. Они Собраны в пакет `com.ibm.uddi.datatype` и его подпакеты.

В листинге 5.7 показано, как можно средствами UDDI4J зарегистрировать Web-службу в том или ином UDDI-реестре.

Листинг 5.7. Регистрация Web-службы в UDDI-реестре

```
import com.ibm.uddi.*;
import com.ibm.uddi.datatype.business.*;
import com.ibm.uddi.response.*;
import com.ibm.uddi.client.*;
import org.w3c.dom.Element;
import java.util.Vector;

import java.util.Properties;

public class SaveSomeService{

   public static void main (String[] args){

      SaveSomeService app = new SaveSomeService();

      UDDIProxy proxy = new UDDIProxy();

      try{
         // Выбираем UDDI-реестр:
         // Тестовый реестр IBM
       proxy.setInquiryURL("http://www-3.ibm.com/services/" +
               "uddi/testregistry/inquiryapi");

       proxy.setPublishURL("https://www-3.ibm.com/services/" +
                "uddi/testregistry/protect/publishapi");

         // Официальный реестр IBM
    // proxy.setInquiryURL("http://www-3.ibm.com/services/" +
    //         "uddi/inquiryapi");
    // proxy.setPublishURL("https://www-3.ibm.com/services/" +
    //         "uddi/protect/publishapi");

         // Тестовый реестр Microsoft
    // proxy.setInquiryURL(
    //         "http://test.uddi.microsoft.com/inquire");
    // proxy.setPublishURL(
```

```
//          "https://test.uddi.microsoft.com/publish");

      // Официальный реестр Microsoft
// proxy.setInquiryURL("http://uddi.microsoft.com/inquire");
// proxy.setPublishURL("https://uddi.microsoft.com/publish");

      // Реестр Hewlett Packard
// proxy.setInquiryURL("http://uddi.hp.com/inquire");
// proxy.setPublishURL("https://uddi.hp.com/publish");

      // Отладочный локальный реестр из пакета WSTK
// proxy.setInquiryURL(
//       "http://localhost:8080/services/uddi/inquiryapi");
// proxy.setPublishURL(
//        "http://localhost:8080/services/uddi/publishapi");

     // Заходим на сайт UDDI-реестра.
    AuthToken token = proxy.get_authToken(
                           "userid", "password");

    System.out.println(
         "Регистрационный код:  " + token.getAuthInfoString());

    System.out.println("Регистрируем Web-службу");

    Vector entities = new Vector();

      // Создаем элемент <businessEntity>.
      // Первый аргумент — UUID — пока неизвестен.
    BusinessEntity be = new BusinessEntity("", "SomeService");

    entities.addElement(be);

      // Регистрируем Web-службу
    BusinessDetail bd =
      proxy.save_business(token.getAuthInfoString(), entities);

      // Для проверки получаем UUID.
```

```
        Vector businessEntities = bd.getBusinessEntityVector();

        BusinessEntity returnedBusinessEntity =
             (BusinessEntity)(businessEntities.elementAt(0));

        System.out.println("Получили UUID: " +

             returnedBusinessEntity.getBusinessKey());

      }catch(UDDIException e){

        DispositionReport dr = e.getDispositionReport();

        if (dr != null){
           System.out.println(
             "UDDIException faultCode:" + e.getFaultCode() +
             "\n operator:" + dr.getOperator() +
             "\n generic:"  + dr.getGeneric() +
             "\n errno:"    + dr.getErrno() +
             "\n errCode:"  + dr.getErrCode() +
             "\n errInfoText:" + dr.getErrInfoText());
        }
        e.printStackTrace();

      }catch(Exception e){
        System.err.println("From proxy: " + e);
      }
    }
}
```

Программа листинга 5.8 отыскивает Web-службу в том или ином реестре UDDI и получает информацию о зарегистрированных в нем Web-службах.

Листинг 5.8. Поиск информации в реестре UDDI

```
import com.ibm.uddi.*;
import com.ibm.uddi.datatype.business.*;
import com.ibm.uddi.response.*;
import com.ibm.uddi.client.*;
import org.w3c.dom.Element;
```

```
import java.util.Vector;
import java.util.Properties;

public class SaveSomeService{

   public static void main (String[] args){

      SaveSomeService app = new SaveSomeService();

      UDDIProxy proxy = new UDDIProxyO;

      try{
         // Выбираем UDDI-реестр:

         // Тестовый реестр IBM
       proxy.setInquiryURL("http://www-3.ibm.com/services/" +
               "uddi/testregistry/inquiryapi");

       proxy.setPublishURL("https://www-3.ibm.com/services/" +
                "uddi/testregistry/protect/publishapi");

         // Официальный реестр IBM
      // proxy.setInquiryURL("http://www-3.ibm.com/services/" +
      //         "uddi/inquiryapi");
      // proxy.setPublishURL("https://www-3.ibm.com/services/" +
      //         "uddi/protect/publishapi");

         // Тестовый реестр Microsoft
      // proxy.setInquiryURL(
      //         "http://test.uddi.microsoft.com/inquire");
      // proxy.setPublishURL(
      //         "https://test.uddi.microsoft.com/publish");

         // Официальный реестр Microsoft
      // proxy.setInquiryURL("http://uddi.microsoft.com/inquire");
      //proxy.setPublishURL("https://uddi.microsoft.com/publish");

         // Реестр Hewlett Packard
```

```
IIproxy.setInquiryURL("http://uddi.hp.com/inquire");
//proxy.setPublishURL("https://uddi.hp.com/publish");

    // Отладочный локальный реестр из пакета WSTK
// proxy.setInquiryURL(
//      "http://localhost:8080/services/uddi/inquiryapi");
// proxy.setPublishURL(
//       "http://localhost:8080/services/uddi/publishapi");

   // Заходим на сайт UDDI-реестра.
   AuthToken token = proxy.get_authToken(
                          "userid", "password");

   System.out.println(
        "Регистрационный код:  " + token.getAuthInfoString());

   System.out.println("СписокWeb-служб:");

     // Получаем список зарегистрированных
     // Web-служб, чьи имена начинаются с буквы "S".
   BusinessList bl = proxy.find_business("S", null, 0);

   Vector businessInfoVector =

          bl.getBusinessInfos().getBusinessInfoVector();

   for (int i = 0; i < businessInfoVector.size(); i++){

      BusinessInfo businessInfo =
          (BusinessInfo)businessInfoVector.elementAt(i);
      System.out.println(businessInfo.getNameString());
   }

}catch(UDDIException e){

  DispositionReport dr = e.getDispositionReport();

  if (dr != null){
```

```
            System.out.println(
              "UDDIException faultCode:" + e.getFaultCode() +
              "\n operator:" + dr.getOperator() +
              "\n generic:"  + dr.getGeneric() +
              "\n errno:"    + dr.getErrno() +
              "\n errCode:"  + dr.getErrCode() +
              "\n errInfoText:" + dr.getErrInfoText());
          }
          e.printStackTrace();

        }catch(Exception e){
          System.err.println("From proxy: " + e);
        }
      }
    }
```

Пакет JAXR

В *главе 2* и в этой главе мы говорили о нескольких различных системах регистрации и поиска Web-служб: UDDI, ebXML, WS-Inspection. Есть и другие системы, не рассматриваемые в этой книге. Все эти системы требуют разных методов доступа к реестру и работы с ним. Для каждой системы приходится создавать своего клиента, работающего с реестром по правилам данной системы поиска Web-служб.

Как уже говорилось в *главе 2,* фирма Sun Microsystems решила разработать единую методику доступа к реестрам разных типов и создала набор интерфейсов JAXR. Этот набор реализован в пакете Sun WSDP и будет включен в состав J2EE, предположительно начиная с версии 1.4. В пакет Sun WSDP включен пример клиента реестра, созданного средствами JAXR. Это графическая утилита просмотра реестров Registry Browser. Она вызывается из командной строки просто набором имени командного файла.

```
$ jaxr-browser
```

Как мы уже говорили в *главе 2,* в составе WSDP есть и простой тренировочный реестр, реализованный в виде сервлета RegistryServerServlet. Реестр работает в Web-контейнере Tomcat и хранит информацию в небольшой базе данных Xindice (произносится, как говорят авторы, "Зин-ди-чи" — с итальянским акцентом), созданной сообществом Apache Software Foundation и тоже входящей в состав WSDP. Для запуска реестра надо запустить Tomcat и стартовать базу данных:

```
$ cd $WSDP_HOME/bin
$ startup
$ xindice-start
```

Для проверки работы реестра и для посылки ему сообщений применяется специальная утилита, работающая из командной строки. Для входа в реестр надо набрать следующую командную строку:

```
$ registry-server-test run-cli-request -Dxml/GetAuthToken.xml
```

Для работы с реестром прямо через базу данных Xindice в составе WSDP есть графическая утилита Indri. Она запускается из командной строки:

```
$ registry-server-test run-indri
```

Поскольку набор JAXR рассчитан на работу с реестрами самых разных типов, он содержит только интерфейсы, которые должен реализовать *поставщик услуг* (service provider) конкретного реестра. Интерфейсы будут поразному реализованы поставщиком услуг реестра UDDI и поставщиком услуг реестра ebXML, но клиент этого не замечает, он просто связывается с поставщиком услуг и пользуется интерфейсами пакета JAXR. Схема обращения клиента к реестру через поставщика услуг показана на рис 2.4.

Состав пакета JAXR

Набор JAXR состоит из двух Java-пакетов `javax.xml.registry` и `javax.xml.registry.infomodel`. Интерфейсы первого пакета применяются клиентом для работы с реестром, интерфейсы второго пакета — поставщиком услуг для преобразования информации к виду, пригодному для хранения в базе данных реестра.

Для связи с поставщиком услуг клиент пользуется интерфейсом connection. Экземпляр этого класса клиент получает с помощью класса-фабрики ConnectionFactory. Сначала статическим методом `newInstance()` создается объект этого класса, затем методом `setProperties()` в него заносятся характеристики реестра, после этого методом `createConnection()` создается объект типа `Connection`. Вот примерная схема соединения:

```
Properties props = new Properties();
props.setProperty("j avax.xml.registry.queryManagerURL",
    queryURL);

ConnectionFactory fact = ConnectionFactory.newInstance();

fact.setProperties(props);

Connection conn = factory.createConnection();
```

После того как связь с поставщиком услуг установлена, методом getRegistryService() интерфейса Connection получаем объект типа RegistryService:

```
RegistryService rs = conn.getRegistryService();
```

Интерфейс RegistryService — это центральный интерфейс в пакете JAXR. Своими методами getXxx() он предоставляет основные объекты, содержащие методы работы с реестром.

Так, методом getBusinessQueryManager() получаем объект типа BusinessQueryManager:

```
BusinessQueryManager bqm = rs.getBusinessQueryManager();
```

Поиск в реестре

В интерфейсе BusinessQueryManager собран добрый десяток методов findXxx() получения информации из реестра. Конечно, их сигнатуры не совпадают с сигнатурами функций UDDI API, поскольку пакет JAXR рассчитан на разные реестры, не только UDDI. Вот заголовок одного из этих методов:

```
public BulkResponse findServices(Key orgKey,
    Collection findQualifiers,    Collection namePatterns,
    Collection classifications,   Collection specifications);
```

Первый аргумент этого метода orgKey содержит идентификатор отыскиваемой Web-службы, которым может быть ключ UDDI.

Во втором аргументе findQualifiers, аналогичном аргументу <findQualifiers> функций UDDI, в метод передается набор условий поиска. Условия поиска записаны строковыми константами интерфейса FindQualifier. Например, условие сортировки результата поиска по имени записывается константой

```
FindQualifier.SORT_BY_NAME_ASC.
```

Третий аргумент namePatterns содержит набор шаблонов имен для поиска. Каждый шаблон — строка символов, содержащая метасимволы в духе SQL.

Четвертый аргумент classifications передает в метод набор объектов типа classification, классифицирующих искомую Web-службу. Этот аргумент аналогичен аргументу <categoryBag> UDDI API.

Наконец, пятый аргумент, specifications, содержит набор технических деталей, аналогично аргументу <tModelBag> UDDI API. Тип каждого элемента этой коллекции RegistryObject. Интерфейс RegistryObject — это суперинтерфейс большинства объектов реестра. Среди них service, ServiceBinding, User, Organization.

Как видите, аргументы метода findServices() весьма сложны, что отражает сложность поиска в реестре.

Метод findservices o возвращает объект типа BulkResponse, содержащий сведения о найденных Web-службах в виде коллекции типа collection. Прежде чем воспользоваться результатом, следует проверить успешность запроса. Проверка выполняется методом getstatus (), возвращающим одну из четырех констант STATUS_SUCCESS, STATUS_WARNING, STATUS_FAILURE, STATUS_UNAVAILABLE.

Итак, искать Web-службу надо примерно так:

```
BulkResponse resp = bqm. findService (key, null, null, null, null);

if (resp.getStatus() = BulkResponse.STATUS_SUCCESS){
   Collection coll = resp.getCollection();
}
```

Полученная коллекция должна содержать объекты типа Service, аналогичные элементу <businessService> реестра UDDI. Выделив из коллекции один такой объект:

```
Service serv = (Service)coll.iterator().next();
```

извлекаем из него информацию многочисленными методами интерфейса Service И его Предка — интерфейса RegistryObject: getKey(), getDescription(), getName(), getSubmittingOrganization() И другими.

Остальные методы findXxx() интерфейса BusinessQueryManager ОТЫСКИвают сведения о фирме, фирмы, связанные с фирмой, указанной в запросе, способы классификации, принятые в реестре, и другую информацию о реестре. Наиболее полезен из них метод findServiceBindings o с теми же аргументами и возвращаемым значением, что и метод findServices(). По информации, возвращаемой методом findServiceBindings () можно узнать способы доступа к реестру. Метод findOrganizations() возвращает самую обширную информацию о всей фирме.

В листинге 5.9 показано, как можно связаться с реестром и получить из него различную информацию о фирме с помощью JAXR.

Листинг 5.9. Получение информации из реестра с помощью JAXR

```
import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;
import java.net.*;
import java.util.*;
```

```
public class JAXRClient{

  public static void main(String[] args){

        // Несколько адресов реестров на выбор.
    String queryURL =
      "http://www-3.ibm.com/services/uddi/inquiryapi";
    //"http://uddi.rte.microsoft.com/inquire";
    //"http://localhost:8080/registry-server/RegistryServerServlet";

    if (args.length < 1){
       System.out.println("Usage: java JAXRClient <name>");
       System.exit(1);
    }

    JAXRClient jq = new JAXRClient();

    String httpProxyHost = "localhost";
    String httpProxyPort = "8080";

    Properties props = new Properties();
    props.setProperty("javax.xml.registry.queryManagerURL",
         queryURL);
    props.setProperty("com.sun.xml.registry.http.proxyHost",
         httpProxyHost);
    props.setProperty("com.sun.xml.registry.http.proxyPort",
         httpProxyPort);

    Connection conn = null;

    try{
       ConnectionFactory fact =
          ConnectionFactory.newInstance();

       fact.setProperties(props);

       conn = fact.createConnection();

       System.out.println("Связь с реестром установлена");
```

```
RegistryService rs = conn.getRegistryService();
BusinessQueryManager bqm = rs.getBusinessQueryManager();

Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);

Collection namePatterns = new ArrayList();
namePatterns.add("%" + args[0] + "%");

    // Поиск фирмы по имени.
BulkResponse response =
    bqm.findOrganizations(findQualifiers,
        namePatterns, null, null, null, null);

Collection orgs = response.getCollection();

    // Сведения о фирме.
Iterator orgIter = orgs.iterator();
while (orgIter.hasNext()){

   Organization org = (Organization)orgIter.next();

   System.out.println("Название: " +
         org.getName().getValue());

   System.out.println("Описание: " +
         org.getDescription().getValue());

   System.out.println("Идентификатор: " +
         org.getKey().getId());

     // Контактная информация
   User pc = org.getPrimaryContact();

   if (pc != null){

      PersonName pcName = pc.getPersonName();

      System.out.println("Название: " +
```

```
            pcName.getFullName());

        Collection phNums =

           pc.getTelephoneNumbers(null);

        Iterator phIter = phNums.iterator();

        while (phIter.hasNext()){

           TelephoneNumber num =
                (TelephoneNumber)phIter.next();
           System.out.println("Номер телефона: " +
                num.getNumber());
        }
        Collection eAddrs = pc.getEmailAddresses();
        Iterator eaIter = eAddrs.iterator();

        while (eaIter.hasNext()){

           EmailAddress eAd =
                (EmailAddress) eaIter.next();

           System.out.println("E-mail: " +
                eAd.getAddress());
        }
    }
        // Услуги и доступ к ним
    Collection services = org.getServices();

    Iterator svcIter = services.iterator();

    while (svcIter.hasNext()){

        Service svc = (Service) svcIter.next();

        System.out.println("Название услуги: " +
              svc.getName().getValue());
```

```
            System.out.println("Описание услуги: " +
                 svc.getDescription().getValue());

            Collection serviceBindings =

                  svc.getServiceBindings();

            Iterator sbIter = serviceBindings.iterator();

            while (sbIter.hasNext()) {

                ServiceBlnding sb =
                    (ServiceBinding)sbIter.next();
                System.out.println("Адрес URI: " +
                    sb.getAccessURI());
            }
        }
        System.out.println("——————");
      }
    }catch(Exception e){
        e.printStackTrace();
    }finally{

      if (connection != null)
        try{
            connection.close();
        }catch(JAXRException je){}
    }
  }
```

Изменением информации в реестре занимаются методы интерфейса `BusinessLifeCycleManager`.

Изменение записей реестра

В интерфейсе BusinessLifeCycleManager собраны методы вида `deleteXxx()` и `saveXxx()`, удаляющие или изменяющие информацию, хранящуюся в реестре. Кроме того, поскольку этот интерфейс расширяет интерфейс `LifeCycleManager`, можно воспользоваться многочисленными методами `createXxx()` последнего интерфейса, создающими записи в реестре.

Объект интерфейса BusinessLifeCycleManager получаем с помощью объекта типа RegistryService следующим методом:

```
BusinessLifeCycieManager blcm = rs.getBusinessLifeCycleManager();
```

Около сорока унаследованных методов полученного объекта создают различные записи в реестре: createClassification(), createEmailAddress(), createKey(), createPostalAddress(), createService(), createUser() и другие.

Шесть методов удаляют записи разных типов, например, метод deleteServices(). Их аргументами служат коллекции ключей удаляемых записей.

Шесть методов заносят новые сведения, например, метод saveservices(). Аргументами этих методов служат коллекции объектов соответствующего типа.

В листинге 5.10 показано создание записи о фирме в реестре.

Листинг 5.10. Создание записи в реестре

```
import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;
import java.net.*;
import java.security.*;
import java.util.*;

public class JAXRPublish{

  Connection connection = null;

  public JAXRPublish(){}

  public static void main(String[] args){

    String queryURL =
       "http://www-3.ibm.com/services/uddi/v2beta/inquiryapi";
     //"http://uddi.rte.microsoft.com/inquire";
     //"http://localhost:8080/registry-server/RegistryServerServlet";

    String publishURL =
       "https://www-3.ibm.com/services/uddi/v2beta/protect/publishapi";
```

```
 //"https://uddi.rte.microsoft.com/publish";
//"http://localhost:8080/registry-server/RegistryServerServlet";

String username = "testuser";
String password = "testuser";

String httpProxyHost = "localhost";
String httpProxyPort = "8080";

Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
     queryURL);
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
     publishURL);
props.setProperty("com.sun.xml.registry.http.proxyHost",
     httpProxyHost);
props.setProperty("com.sun.xml.registry.http.proxyPort",
     httpProxyPort);
props.setProperty("com.sun.xml.registry.https.proxyHost",
     httpsProxyHost);
props.setProperty("com.sun.xml.registry.https.proxyPort",
     httpsProxyPort);

try{
   ConnectionFactory fact =
        ConnectionFactory.newInstance();
   fact.setProperties(props);
   conn = factory.createConnection();

   System.out.println("Соединение установлено.");

   RegistryService rs = connection.getRegistryService();

   BusinessLifeCycleManager blcm =

       rs.getBusinessLifeCycleManager();

   BusinessQueryManager bqm = rs.getBusinessQueryManager();

   PasswordAuthentication passwdAuth =
```

```
        new PasswordAuthentication(username,
                password.toCharArray());

    Set creds = new HashSet();
    creds.add(passwdAuth);
    conn.setCredentials(creds);

         // Создание фирмы.
    Organization org =
         blcm.createOrganization("Рога и копыта");
    InternationalString s =
       blcm.createInternationalString(
            "Прием круглосуточно.");
    org.setDescription(s);

         // Контактная информация.
    User primaryContact = blcm.createUser();
    PersonName pName = blcm.createPersonName("О. И. Бендер");
    primaryContact.setPersonName(pName);

    TelephoneNumber tNum = blcm.createTelephoneNumber();
    tNum.setNumber("333-44-55");
    Collection phoneNums = new ArrayList();
    phoneNums.add(tNum);
    primaryContact.setTelephoneNumbers(phoneNums);

    EmailAddress emailAddress =
       blcm.createEmailAddress("bender@hornhoof.com");
    Collection emailAddresses = new ArrayList();
    emailAddresses.add(emailAddress);
    primaryContact.setEmailAddresses(emailAddresses);

    org.setPrimaryContact(primaryContact);

    ClassificationScheme cScheme =
        bqm.findClassificationSchemeByName(
                null, "ntis-gov:naics");

    Classification classification =  (Classification)
```

```
    blcm.createClassification(cScheme,
        "Snack and Nonalcoholic Beverage Bars", "722213");
Collection classifications = new ArrayList();
classifications.add(classification);
org.addClassifications(classifications);

Collection services = new ArrayList();
Service service =
    blcm.createService("Прием рогов");
InternationalString is =
    blcm.createInternationalString("Высшее качество");
service.setDescription(is);

Collection serviceBindings = new ArrayList();
ServiceBinding binding = blcm.createServiceBinding();
is = blcm.createInternationalString("My Service Binding " +
        "Description");
binding.setDescription(is);
    // allow us to publish a bogus URL without an error
binding.setValidateURI(false);
binding.setAccessURI("http://TheCoffeeBreak.com:8080/sb/");
    serviceBindings.add(binding);

    // Add service bindings to service
service.addServiceBindings(serviceBindings);

    // Add service to services, then add services to organization
services.add(service);
org.addServices(services);

    // Add organization and submit to registry
    // Retrieve key if successful
Collection orgs = new ArrayList();
orgs.add(org);
BulkResponse response = blcm.saveOrganizations(orgs);
Collection exceptions = response.getExceptions();
if (exceptions == null){
    System.out.println("Organization saved");
```

```
        Collection keys = response.getCollection();
        Iterator keyIter = keys.iterator();
        if (keyIter.hasNext()) {
           javax.xml.registry.infomodel.Key orgKey =
              (javax.xml.registry.infomodel.Key) keyIter.next();
           String id = orgKey.getId();
           System.out.println("Organization key is " + id);
        }
      }else{
        Iterator exciter = exceptions.iterator();
        Exception exception = null;
        while (excIter.hasNext()){
            exception = (Exception) excIter.next();
            System.err.println("Exception on save: " +
                       exception.toString());
        }
      }
    }catch(Exception e){
        e.printStackTrace();
    }finally{

       if (connection != null){
          try{
              connection.close();
          }catch(JAXRException je){}
       }
    }
  }
}
```

ГЛАВА 6

Детали создания Web Services

В *главе 3* мы разделили все средства создания Web-служб на четыре группы, от самых простых, быстро создающих типичную Web-службу, до самых сложных, позволяющих детально прописать нужные свойства создаваемой Web-службы. В этой главе мы рассмотрим средства третьей группы — наборы интерфейсов и реализующих их классов, детально описывающие создаваемые Web-службы. В настоящее время общеприняты интерфейсы фирмы Sun Microsystems, образующие пакеты SAAJ, JAXM, JAXR, JAX-RPC. Мы уже обращались к этим пакетам на протяжении всей книги. Они распространяются отдельно или в составе набора инструментальных средств Sun WSDP (Web Services Developer Pack). В набор WSDP входит, кроме этих пакетов, Web-контейнер Tomcat, выполняющий сервлеты и страницы JSP, и небольшой UDDI-реестр Registry Server. В нем есть еще пакет JAXP со средствами обработки документов XML и классы для создания сервлетов и страниц JSP.

С таким оснащением набор WSDP может работать самостоятельно, но его мощности недостаточно для обеспечения работы созданных с его помощью производственных Web-служб. Для создания мощных боевых Web-служб набор WSDP встраивается в систему уровня предприятия J2EE, а следовательно и во все серверы приложений, основанные на J2EE-технологии: BEA WebLogic, IBM WebSphere, Sun ONE Application Server, JBoss, Sybase EAServer, Oracle Application Server, IONA Orbix E2A, Borland Enterprise Server. Начиная с версии 1.4 системы J2EE, набор WSDP будет сразу входить в ее состав.

Каждый из перечисленных выше пакетов, создающих Web-службы, составляет один или несколько пакетов Java:

- пакет SAAJ, предназначенный для создания отправляемого SOAP-послания или разбора полученного SOAP-послания, а также для синхронного обмена SOAP-посланиями в документном стиле, расположен в одном Java-пакете `javax.xml.soap`;

- пакет JAXM, содержащий средства асинхронной отправки и получения SOAP-посланий в документном стиле, находится в Java-пакете `javax.xml.messaging`;
- пакет JAX-RPC, реализующий процедурный стиль SOAP-посланий, образован ИЗ семи Java-пакетов: `javax.xml.rpc`, `javax.xml.rpc.handler.soap`, `javax.xml.rpc.encoding`, `javax.xml.rpc.soap`, `javax.xml.rpc.handler`, `javax.xml.rpc.holders`, `javax.xml.rpc.` server.

В этой главе мы рассмотрим последовательно все эти пакеты и способы работы с ними. Начнем с пакета SAAJ.

Создание SOAP-послания

Интерфейсы Java, входящие в пакет `javax.xml.soap`, описывают SOAP-послание и его элементы. Для представления элементов XML, составляющих SOAP-послание, применяется модель дерева DOM (см. *главу* 7). Поэтому во главе всех интерфейсов, описывающих послание, стоит интерфейс Node.

Узел дерева элементов *Node*

Методы интерфейса Node традиционны для работы с узлом дерева:

- public string `getValue()` — возвращает ссылку на значение первого узла-потомка, если оно представимо в виде строки; в противном случае возвращается null;
- public `SOAPElement getParentElement()` — возвращает ССЫЛКУ на узел-предок данного узла или null, если предка нет:
- public void `setParentElement(SOAPElement` parent) — устанавливает новый узел parent в качестве предка данного узла;
- public void `detachNode()` — удаляет узел из дерева.

Элемент послания *SOAPElement*

Интерфейс Node расширен интерфейсом SOAPElement, определяющим в общих чертах элементы XML, составляющие SOAP-послание. Его методы addxxx () наполняют создаваемый элемент содержимым:

- `public SOAPElement addAttribute(Name name, String value)` — добавляет атрибут name со значением value в открывающий тег данного элемента; возвращает ссылку на обновленный элемент с добавленным атрибутом;

- public SOAPElement addChildElement(Name name) — создает новый элемент с именем name, вложенный в данный элемент, и возвращает ссылку на него;
- public SOAPElement addChildElement(SOAPElement element) — добавляет элемент element, вложенный в данный элемент, и возвращает ссылку на него же;
- public SOAPElement addChildElement(String localName) — создает новый элемент с именем localName без префикса, вложенный в данный элемент, и возвращает ссылку на него;
- public SOAPElement addChildElement(String localName, String prefix) — создает новый элемент с расширенным именем prefix:localName, вложенный в данный элемент, и возвращает ссылку на него;
- public SOAPElement addChildElement(String localName, String prefix, string uri) — создает новый элемент с расширенным именем prefix:localName и идентификатором пространства имен uri, вложенный в данный элемент, и возвращает ссылку на него;
- public SOAPElement addNamespaceDeclaration(String prefix, String uri) — добавляет атрибут xmlns со значением uri к открывающему тегу данного элемента и возвращает ссылку на измененный элемент;
- public SOAPElement addTextNode(String text) — добавляет тело элемента в виде строки text и возвращает ссылку на измененный элемент;
- public void setEncodingStyle(String encStyle) — добавляет атрибут encodingstyle со значением encStyle к открывающему тегу данного элемента.

Остальные методы вида getXxx(), removeXxx() интерфейса SOAPElement возвращают различные сведения об элементе и удаляют различные части элемента.

Открывающий тег элемента XML

Интерфейс Name, используемый в перечисленных выше методах, определяет имя элемента XML с префиксом или без префикса и с идентификатором Пространства имен. Его методы getLocalName(), getPrefix(), getQualifiedName(), getURI() возвращают составные части имени XML в виде строки типа string.

Основные элементы SOAP-послания

Интерфейс SOAPElement расширен интерфейсами, описывающими корневой элемент SOAP-послания <Envelope> и вложенные в него элементы

<Header> И <Body>. Это интерфейсы SOAPEnvelope, SOAPHeader, SOAPBody. **Блоки заголовка описывает интерфейс** SOAPHeaderElement, **а содержимое послания** — интерфейс SOAPBodyElement.

Интерфейс SOAPBodyElement **расширен интерфейсом** SOAPFault, **описывающим элемент <Fault> — единственный элемент, вложенный в элемент** <Body> **и определенный в спецификации SOAP. Элементы, вложенные в <Fault>, описаны интерфейсом** SOAPFaultElement, **а элемент** <detail> **специально описан интерфейсом** Detail. **Элементы, вложенные в <detail>, описаны интерфейсом** DetailEntry.

Вся эта иерархия интерфейсов показана на рис. 6.1.

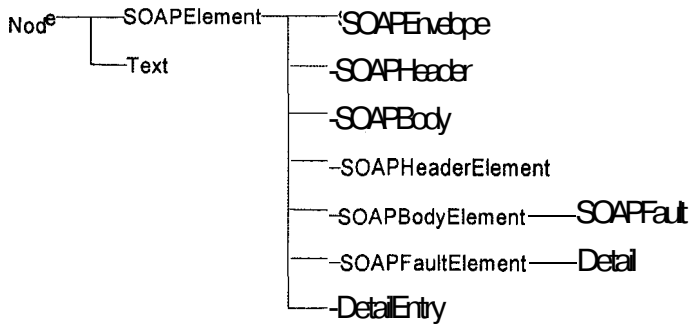


Рис. 6.1. Иерархия интерфейсов SAAJ

Корневой элемент *SOAPEnvelope*

Интерфейс SOAPEnvelope добавляет к методам интерфейса SOAPElement **следующие методы, создающие элементы SOAP-послания:**

- public SOAPHeader addHeader() — **добавляет к SOAP-посланию пустой заголовок — элемент** <Header> — **и возвращает ссылку на него;**
- public SOAPHeader getHeader() — **формирует элемент** <Header> **и возвращает ссылку на него;**
- public SOAPBody addBody() — **добавляет к SOAP-посланию пустой элемент** <Body> **и возвращает ссылку на него;**
- public SOAPBody getBody() — **возвращает ссылку на созданный этим же методом пустой элемент** <Body>;
- public Name createName (String localName) — **создает элемент XML с именем** localName **без префикса;**
- public Name createName(String localName, String prefix, String uri) — **создает элемент XML с расширенным именем** prefix:localName **и идентификатором пространства имен** uri.

Заголовок послания SOAPHeader

Интерфейс SOAPHeader, в свою очередь, описывает метод

public SOAPHeaderElement addHeaderElement(Name name),

создающий и добавляющий блок заголовка с именем name. Кроме этого метода в интерфейс входит еще метод

public Iterator examineHeaderElements(String actor),

возвращающий итератор для перебора всех блоков заголовка, предназначенных для роли actor, и метод

public Iterator extractHeaderElements(String actor),

удаляющий из заголовка блоки, предназначенные для роли actor, и возвращающий итератор для их перебора.

Содержимое послания SOAPBody

Интерфейс SOAPBody содержит метод

public SOAPBodyElement addBodyElement(Name name),

добавляющий к содержимому SOAP-послания произвольный элемент, и методы

public SOAPFault addFault(),

public SOAPFault getFault(),

public boolean hasFault(),

работающие с элементом <Fault>, вкладываемым в элемент <Body> для отправки сообщения об ошибке.

Как видите, интерфейсы, входящие в пакет SAAJ, детально описывают структуру SOAP-послания и содержат все методы для создания и изменения его элементов.

Послание SOAPMessage

Пакет SAAJ позволяет создавать SOAP-послания с дополнениями, следуя спецификации "SOAP с дополнениями", с которой мы познакомились в *главе 3*. Для этого в его состав введен абстрактный класс SOAPMessage, методы которого создают, добавляют или удаляют дополнительные части послания. Начальная часть SOAP-послания с дополнениями, содержащая собственно SOAP-послание, описана абстрактным классом soAPPart, а сами дополнения определены в абстрактном классе AttachmentPart. Заголовок каждой части добавляется и удаляется методами класса MimeHeaders, а каждое поле заголовка — методами класса MimeHeader.

Таким образом, пакет SAAJ составлен из интерфейсов и абстрактных классов, требующих реализации. В наборе Sun WSDP такая реализация сделана классами, упакованными в архив saaj-ri.jar. Реализация сильно зависит от операционной среды, в которой будет работать система SAAJ. Поэтому у классов, реализующих пакет SAAJ, почти нет конструкторов, их экземпляры создаются через фабричные классы и их методы. Фабричные классы MessageFactory, SOAPFactory и SOAPConnectionFactory входят в состав SAAJ.

Процесс создания SOAP-послания

Формирование SOAP-послания удобно начать с создания экземпляра класса SOAPMessage. Для этого применяем метод `createMessage()` класса MessageFactory, предварительно создав объект класса MessageFactory его же статическим методом `newInstance()`:

```
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage smsg = mf.createMessage();
```

Далее из объекта smsg, представляющего SOAP-послание с дополнениями, выделяем начальную часть soapPart, содержащую пока еще пустое SOAP-послание:

```
SOAPPart soapPart = smsg.getSOAPPart();
```

Потом создаем корневой элемент SOAP-послания:

```
SOAPEnvelope env = soapPart.getEnvelope();
```

После этого уже сформирован элемент <Envelope>:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
</SOAP-ENV:Envelope>
```

В то время, когда были написаны эти строки, пакет SAAJ 1.1 формировал SOAP-послания версии SOAP 1.1, поэтому он записал старый идентификатор пространства имен http://schemas.xmlsoap.org/soap/envelope/.

Затем формируем заголовок <Header>:

```
SOAPHeader header = env.getHeader();
```

В результате появляется элемент

```
<SOAP-ENV:Header>
</SOAP-ENV:Header>
```

Он пуст, но уже вложен в элемент <Envelope>.

Аналогично формируем элемент <Body>:

```
SOAPBody body = env.getBody();
```

Получаем пустой элемент

```
<SOAP-ENV:Body>
</SOAP-ENV:Body>
```

вложенный в элемент <Envelope>.

Если заголовок <Header> нам не нужен, то его можно удалить:

```
header.detachNode();
```

После этого можно заполнять SOAP-послание нужной информацией.

Например, мы хотим обратиться к методу

```
public String getEcho(String request)
```

следующим образом:

```
String response = echoService.getEcho("Привет!").
```

Для этого сначала создаем элемент

```
<m: getEchoxmlns :m="http://some.com/names"></m:getEcho>
```

методом `createName()`:

```
Name element = env.createName("getEcho", "m",
    "http://some.com/names");
```

и помещаем элемент <getEcho> в элемент <Body>:

```
SOAPBodyElement ge = body.addBodyElement(element);
```

Затем создаем элемент

```
<request>Привет!</request>
```

и вкладываем его в элемент <getEcho>. Это выполняется в следующих трех строчках:

```
Name name = env.createName("request");
SOAPElement req = ge.addChildElement(name);
req.addTextNode("Привет!");
```

Все, SOAP-послание готово и выглядит так:

```
<SOAP-ENV:Envelope

   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

   <SOAP-ENV:Body>

      <m:getEcho xmlns:m="http://some.com/names">
         <request>Привет
                        !</request>
```

```
        </m:getEcho>

    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Отправка SOAP-послания и получение ответа

Теперь надо отправить SOAP-послание `smsg`. Для этого надо установить соединение с SOAP-сервером. Сначала подготавливаем объект класса `SOAPConnection`:

```
SOAPConnectionFactory scf =
    SOAPConnectionFactory.newInstance();

SOAPConnection con = scf.createConnection();
```

Потом определяем адрес Web-службы в виде объекта класса URL ИЛИ объекта класса Endpoint, например:

```
URL endpoint =
    new URL("http://some.com/services/EchoService");
```

Все готово. Теперь одним махом отправляем SOAP-послание и получаем ответ:

```
SOAPMessage response = con.call(smsg, endpoint);
```

После получения ответа можно закрыть соединение:

```
con.close();
```

или посылать следующее сообщение.

Как видно из приведенного выше описания, для работы метода `callo` необходимо, чтобы SOAP-сервер находился на связи, иначе клиентская программа зависнет в ожидании ответа. Как говорят, пакет SAAJ обеспечивает *синхронную* работу клиента с Web-службой. Такой способ сетевого взаимодействия часто называют "point-to-point", сокращенно "P2P".

После получения от SOAP-сервера ответа response, надо его разобрать и выделить полезную информацию. Для этого последовательно выделяем отдельные части ответного SOAP-послания. Сначала выделяем начальную часть послания:

```
SOAPPart sp = response.getSOAPPart();
```

Затем из начальной части выделяем SOAP-послание:

```
SOAPEnvelope respEnv = sp.getEnvelope();
```

Потом выщеляем содержимое послания:

```
SOAPBody respBody = respEnv.getBody();
```

Из элемента <Body>, представленного объектом respBody, выбираем все вложенные элементы. Сначала получаем их итератор:

```
Iterator it = respBody.getChildElements();
```

Нас интересует только первый элемент, содержащий ответ сервера. Получим его:

```
SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
```

Выщеляем содержимое ответа:

```
String echo = bodyElement.getValue();
```

и выводим в стандартный вывод клиентской машины:

```
System.out.println("Ответ сервера: " + echo);
```

Все сделано, мы воспользовались услугами Web-службы. В листинге 6.1 приведен полный текст синхронного клиента Web-службы.

Листинг 6.1. Клиент Web-службы, созданный средствами SAAJ

```
import javax.xml.soap.*;
import java.util.*;
import java.net.URL;

public class ClientSAAJ{

  public static void main(String[] args){

    try{
      MessageFactory mf =
          MessageFactory.newInstance();

      SOAPMessage smsg = mf.createMessage();

      SOAPPart soapPart = smsg.getSOAPPart();
      SOAPEnvelope env = soapPart.getEnvelope();
      SOAPHeader header = env.getHeader();
      SOAPBody body = env.getBody();
      header.detachNode();

      Name element = env.createName(
```

```
        "getEcho", "m", "http://some.com/names");

      SOAPBodyElement ge =
        body.addBodyElement(element);

      Name name = env.createName("request");
      SOAPElement req = ge.addChildElement(name);
      req.addTextNode("Привет!");

      SOAPConnectionFactory scf =

          SOAPConnectionFactory.newInstance();

      SOAPConnection con = scf.createConnection();

      URL endpoint =

          new URL("http://some.com/services/EchoService");

      SOAPMessage response = con.call(smsg, endpoint);

      con.close();
      SOAPPart sp = response.getSOAPPart();
      SOAPEnvelope respEnv = sp.getEnvelope();
      SOAPBody respBody = respEnv.getBody();
      Iterator it = respBody.getChildElements();

      SOAPBodyElement bodyElement =

              (SOAPBodyElement)it.next();

      String echo = bodyElement.getValue();

      System.out.print("Ответ сервера: " + echo);
    }catch(Exception e){
      System.err.println(e);
    }
  }
}
```

Создание SOAP-послания с дополнениями

Как сказано выше, объект класса SOAPMessage содержит SOAP-послание с дополнениями. Пока мы использовали только начальную часть послания — объект класса soAPPart. В этом разделе добавим к посланию дополнительные части.

Каждое дополнение формируется методами абстрактного класса AttachmentPart. Создать экземпляр этого класса можно одним из трех методов класса SOAPMessage.

Первый метод

```
AttachmentPart ap = smsg.createAttachmentPart();
```

создает пустую дополнительную часть ap. После создания, экземпляр ap надо заполнить содержимым, применяя методы класса AttachmentPart.

После полного оформления дополнения ap, о чем речь пойдет чуть ниже, его надо добавить к посланию следующим методом:

```
smsg.addAttachmentPart(ap);
```

Второй метод создания экземпляра класса AttachmentPart:

```
public AttachmentPart createAttachmentPart(
    Object content, String contentType);
```

формирует дополнительную часть с уже готовым, предварительно определенным, содержимым content, MIME-тип которого, записываемый в поле Content-Type заголовка создаваемой дополнительной части, равен contentType. Например:

```
AttachmentPart ap = smsg.createAttachmentPart(
    "Это простой плоский ASCII-текст дополнения",
    "text/plain; charset=windows-1251");

smsg.addAttachmentPart(ap);
```

Третий метод создания дополнительной части

```
public AttachmentPart createAttachmentPart(
   javax.activation.DataHandler handler);
```

использует дескриптор handler данных, полученных из файла или с какого-то адреса URL с помощью механизма Java Activation Framework [10]. Например:

```
URL url = new URL("http://some.com/images/myface.jpg");

AttachmentPart ap =
```

```
    smsg.createAttachmentPart(new DataHandler(url));

smsg.addAttachmentPart(ap);
```

Итак, если содержимое дополнительной части уже готово и не нужны дополнительные заголовки, то дополнение создается одним действием. Посмотрим, какими методами обладает класс `AttachmentPart` для того чтобы создать нестандартное дополнение к SOAP-посланию.

Класс *AttachmentPart*

Напомним, что каждая дополнительная часть послания, оформленного по спецификации "SOAP с дополнениями" (см. *главу 3),* записывается после строки-разделителя. Дополнительная часть состоит из необязательного MIME-заголовка, состоящего из нескольких полей, пустой строки и произвольного содержимого. Среди полей заголовка наиболее важно поле content-Type, определяющее MIME-тип содержимого.

Создание дополнения

Несколько методов класса AttachmentPart формируют MIME-заголовок дополнительной части SOAP-послания.

Поле name MIME-заголовка со значением value добавляется к заголовку дополнительной части методом

```
public void addMimeHeader(String name, String value);
```

Ввиду особой важности поля Content-Type, для его создания есть специальный метод

```
public void setContentType(String value);
```

Еще два поля — `Content-Id` и `Content-Location` — добавляются к заголовку собственными методами

```
public void setContentId(String value);
public void setContentLocation(String value);
```

Если поля заголовка уже определены, то эти три метода `setXxx()` меняют их значения. Заменить старое значение любого поля name заголовка новым значением value можно методом

```
public void setMimeHeader(String name, String value);
```

Этот же метод определяет новое поле заголовка, если оно еще не было создано.

Содержимое content дополнения включается в него методом

```
public void setContent(Object content, String contentType);
```

Кроме того, этот метод записывает значение contentType поля Content-Type.

Разбор дополнения

Следующие методы облегчают SOAP-серверу разбор полученного от клиента послания.

Выделить все дополнительные части из полученного послания класса SOAPMessage можно методом

```
public Iterator getAttachments();
```

класса SOAPMessage, возвращающим итератор, с помощью которого можно перебрать все дополнительные части послания.

Еще один метод того же класса

```
public Iterator getAttachments(MimeHeaders headers);
```

возвращает итератор всех дополнительных частей с полями заголовков, занесенными в объект headers.

Получив итератор одним из этих двух методов, обычным образом выбираем дополнение:

```
Iterator it = smsg.getAttachments();

while (it.hasNext()){
   AttachmentPart ap = (AttachmentPart)it.next();
   // Обработка дополнительной части ap послания
}
```

Выбрав нужное дополнение, разбираем его и выделяем полезную информацию. Для этого в классе AttachmentPart есть несколько методов доступа getxxx (). Перечислим их.

Содержимое дополнительной части можно получить методом

```
public Object getContent();
```

Размер содержимого в байтах возвращает метод

```
public int getSize();
```

Перебрать значения всех полей заголовка можно, получив их итератор методом

```
public Iterator getAllMimeHeaders();
```

Значения всех полей с именем name в виде массива строк возвращает метод

```
public String[] getMimeHeader(String name);
```

Метод

```
public Iterator getMatchingMimeHeaders(String[] names);
```

возвращает итератор всех полей заголовка, чьи имена занесены в массив names.

Метод

```
public Iterator getNonMatchingMimeHeaders(String[] names);
```

решает двойственную задачу — возвращает итератор всех полей заголовка, чьи имена не совпадают с именами, перечисленными в массиве names.

Наконец, значения трех полей заголовка: content-Type, Content-Id и Content-Location можно получить специальными методами

```
public String getContentType();
public String getContentId();
public String getContentLocation();
```

Удаление частей дополнения

Удалить содержимое дополнения можно методом

```
public void clearContent();
```

Все поля заголовка дополнения удаляет метод

```
public void removeAllMimeHeaders();
```

Следующий метод

```
public void removeMimeHeader(String name);
```

удаляет все поля заголовка с именем name.

В листинге 6.2 приведен пример формирования и отправки SOAP-послания с дополнением — страничкой HTML.

Листинг 6.2. Послание с дополнениями

```
import javax.xml.soap.*;
import javax.xml.messaging.*;
import javax.activation.*;
import java.util.*;
import java.net.URL;

public class ClientSAAJAttach{

  public static void main(String[] args){
```

```
try{
  MessageFactory mf = MessageFactory.newInstance();

  SOAPMessage smsg = mf.createMessage();

  SOAPPart soapPart = smsg.getSOAPPart();
  SOAPEnvelope env = soapPart.getEnvelope();
  SOAPHeader header = env.getHeader();
  SOAPBody body = env.getBody();
  header.detachNode();

  Name element = env.createName(
     "getEcho", "m", "http://some.com/names");

  SOAPBodyElement ge =
    body.addBodyElement(element);

  Name name = env.createName("request");
  SOAPElement req = ge.addChildElement(name);
  req.addTextNode("Привет!");

  URL url = new URL("http://localhost:8080/data/info.html");

  AttachmentPart ap =

      smsg.createAttachmentPart(new DataHandler(url));

  ap.setContentType("text/html");

  smsg.addAttachmentPart(ap);

  SOAPConnectionFactory scf =

      SOAPConnectionFactory.newInstance();

  SOAPConnection con = scf.createConnection();

  URLEndpoint endpoint =
      new URLEndpoint("http://some.com/services/EchoService");
```

```
        SOAPMessage response = con.call(smsg, endpoint);

        con.close();

        SOAPPart sp = response.getSOAPPart();
        SOAPEnvelope respEnv = sp.getEnvelope();
        SOAPBody respBody = respEnv.getBody();

        Iterator it = respBody.getChildElements();

        SOAPBodyElement bodyElement =

                (SOAPBodyElement)it.next();

        String echo = bodyElement.getValue();

        System.out.print("Ответ сервера: " + echo);
      }catch(Exception e){
        System.err.println(e);
      }
    }
}
```

Сообщение об ошибке

Как уже говорилось в *главе 3,* заметив ошибку в полученном послании, SOAP-сервер отправляет сообщение, содержащее в теле ответного SOAP-послания всего один элемент <Fault>, вложенный в элемент <Body>. В пакете SAAJ есть интерфейс soAPFauit, описывающий элемент <Fault>.

Интерфейс *SOAPFault*

Напомним, что в элемент <Fault> версии SOAP 1.1 можно вложить элементы <faultcode>, <faultstring>, <actor> И <detail>. Они создаются следующими методами, описанными в интерфейсе soAPFauit:

```
public void setFaultCode(String faultCode);
public void setFaultString(String faultString);
public void setFaultActor(String faultActor);
public Detail addDetail();
```

Остальные методы интерфейса soAPFauit предназначены для разбора клиентом полученного от сервера сообщения об ошибке:

```
public String getFaultCode();
public String getFaultString();
public String getFaultActor();
public Detail getDetail();
```

Как видно из этого описания, метод `addDetail()` создает пустой элемент <detail>. Его надо наполнить содержимым. Это выполняется методом

```
public DetailEntry addDetailEntry(Name name);
```

описанным в интерфейсе Detail.

Посмотреть при разборе содержимое элемента <detail> можно при помощи итератора, полученного вторым методом интерфейса Detail:

```
public Iterator getDetailEntries();
```

Все эти действия приведены в листинге 6.3.

Листинг 6.3. Пример создания и разбора сообщения об ошибке

```
import javax.xml.soap.*;
import java.util.*;

public class SOAPFaultTest{

  public static void main(String[] args)(

    try{

      MessageFactory mf = MessageFactory.newInstance();

     SOAPMessage smsg = mf.createMessage();

     SOAPEnvelope env =

         smsg.getSOAPPart().getEnvelope();

     SOAPBody body = env.getBody();

     SOAPFault fault = body.addFault();

     fault.setFaultCode("Client");
```

```
fault.setFaultString(

    "Послание неправильно оформлено");

fault.setFaultActor("http://some.com/SomeService");

Detail detail = fault.addDetail();

Name entryName1 = env.createName(
   "comment", "ns1", "http://some.com/comments");
DetailEntry entry = detail.addDetailEntry(entryName1);
entry.addTextNode("Не задано значение параметра");

Name entryName2 = env.createName(
   "confirmation", "ns2", "http://some.com/confirm");

DetailEntry entry2 = detail.addDetailEntry(entryName2);
entry2.addTextNode("Неверный код отправителя");

smsg.saveChanges();

// Проверим созданное послание.

if (body.hasFault()){

  fault = body.getFault();
  String code = fault.getFaultCode();
  String s = fault.getFaultString();
  String actor = fault.getFaultActor();

  System.out.println("Код ошибки: " + code);
  System.out.println("Разъяснение: " + s);
  if (actor != null)

    System.out.println("Ошибка замечена " + actor);

  detail = fault.getDetail();

  if (detail != null){
```

```
            System.out.println("Детали: ");

            Iterator it = detail.getDetailEntries();
            while (it.hasNext()){

               entry = (DetailEntry)it.next();
               String value = entry.getValue();
               System.out.println(value);
            }
          }
        }
      }catch(Exception e){
        System.out.println(e);
      }
    }
}
```

Асинхронный обмен сообщениями

Пакет SAAJ обеспечивает создание SOAP-посланий и обмен ими в синхронном режиме P2P (point-to-point). Часто такой способ обмена сообщениями оказывается невозможным из-за того, что участники обмена не всегда одновременно находятся на связи. В таких случаях приходится обмениваться сообщениями *асинхронно,* не дожидаясь ответа на посланное сообщение. Так работает электронная почта. В технологии Java асинхронный обмен SOAP-посланиями обеспечивается интерфейсами и классами пакета JAXM, находящимися в Java-пакете `javax.xml.messaging`.

Реализация асинхронного обмена посланиями основана на том, что послание принимает не адресат, которому оно направлено, а так называемый поставщик сообщений. *Поставщик сообщений* (messaging provider) — это программа, которая должна постоянно находиться на связи, принимать послания, временно хранить их и передавать адресату, как только тот выйдет на связь. Поставщик сообщений реализуется как распределенное приложение, отдельные компоненты которого находятся и у отправителя сообщений и у их получателя. В технологии Java система обмена сообщениями реализована механизмом JMS (Java Message Service). С механизмом JMS можно ознакомиться, например, по книге [10], но это необязательно. Присутствие поставщика сообщений совершенно не ощущается SOAP-клиентом и SOAP-сервером, пакет JAXM берет на себя всю работу с поставщиком.

Поставщик сообщений может посылать отправителю подтверждение (acknowledgement) в получении послания или не посылать его. В любом случае отправитель не дожидается ответа, а переключается на решение других задач. Ответ может прийти позднее, даже через несколько дней, или вообще не прийти.

Некоторые системы обмена сообщениями задают специфичные элементы SOAP-посланий. Например, спецификация "ebXML Message Service Specification" определяет обязательные для ebXML-послания блоки заголовка <MessageHeader>, <Service>, <SyncReply> и другие. Совокупность таких элементов образует *профиль* (profile) SOAP-послания. Профили посланий различаются своими именами. Имя профиля может быть простой строкой, например, "ebxml", "soaprp", или строкой URI. У каждого поставщика сообщений есть некоторый набор известных ему профилей. Стандартная реализация JAXM "понимает" профиль ebXML и профиль протокола Microsoft WS-Routing, называемый в документации "SOAP-RP". Послания с этими профилями создаются методами специально разработанных классов `EbXMLMessageImpl` И `SOAPRPMessageImpl` соответственно, ИЗ пакетов `com.sun.xml.messaging.jaxm.ebxml` И `com. sun.xml.messaging.jaxm.soaprp`.

Создавая SOAP-послание средствами JAXM, нужно указывать его профиль. Поэтому мы сначала рассмотрим подробнее протокол WS-Routing, описывающий профиль SOAP-RP.

Протокол WS-Routing и его реализация

Протокол WS-Routing (Web Services Routing Protocol), разработанный корпорацией Microsoft в 2001 году **(http://msdn.microsoft.com/library/en-us/dnsrvspec/html/ws-routing.asp)**, предназначен для создания и отправки SOAP-посланий, не требующих ответа и подтверждения получения. Послание, отправленное по протоколу WS-Routing, может на своем пути пройти несколько промежуточных серверов (actors), которые могут сделать предварительную обработку послания (forward message path). Это делает и обычное SOAP-послание, но протокол WS-Routing, в отличие от протокола SOAP, определяет точный порядок прохождения промежуточных серверов. Это позволяет спланировать последовательность обработки послания промежуточными серверами. Хотя протокол не предназначен для двусторонней связи, он позволяет отследить обратный путь (reverse message path) для посылки ответного послания.

Послание, созданное по правилам протокола WS-Routing, можно отправить не только по протоколам HTTP или SMTP, но и прямо по протоколу TCP или UDP, причем промежуточные серверы могут по своему усмотрению менять транспортный протокол.

Итак, главная цель протокола WS-Routing — обеспечить точный маршрут прохождения SOAP-послания от отправителя до получателя. Дополнительно можно задать обратный маршрут. Для достижения этих целей в заголовок SOAP-послания <Header> вкладывается элемент <path>, в котором вложенным элементом <from> указывается адрес URI отправителя, элементом <to> — адрес URI получателя, а элементами <via> — адреса URI промежуточных серверов. Элементы <via> записываются внутри элемента <fwd> и показывают маршрут от отправителя к получателю. Порядок прохождения промежуточных пунктов соответствует порядку записи элементов <via>. Обратный маршрут, если он нужен, указывается элементами <via>, вложенными в элемент <rev>. Все эти элементы определены в пространстве имен с идентификатором http://schemas.xmlsoap.org/rp/.

Простейшее послание не содержит промежуточных адресов и адреса отправителя. Оно выглядит примерно так:

```
<env:Envelope

   xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">

   <env:Header>

      <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">

         <m:action>http://some.com/update</m:action>
         <m:to>soap://D.com/endpoint;up=udp</m:to>

         <m:id>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</m:id>

      </m:path>

   </env:Header>

   <env:Body>

      <!- Содержимое послания... ->
   </env:Body>
</env:Envelope>
```

В этом примере показаны еще два обязательных элемента SOAP-послания, составленного по протоколу WS-Routing.

Обязательный элемент <action> указывает строкой URI программу-обработчик послания.

Обязательный элемент <id> содержит строку URI, однозначно идентифицирующую данное послание. Это полезно для связи друг с другом подряд идущих посланий или для ссылки ответного послания на послание, вызвавшее ответ. Ссылка производится элементом <relatesTo>, например:

```
<relatesTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</relatesTo>
```

Заметьте, что адрес получателя в элементе <to> указан строкой URI со схемой soap:. Эта схема URI ко времени написания книги еще не была утверждена комитетом IETF, хотя ее введение давно витает в воздухе. Схема soap: аналогична схеме http:, но добавляет параметр up (underlying protocol), указывающий транспортный протокол. По умолчанию принимается протокол TCP.

Если надо указать промежуточные серверы, например, B.com и C.com, то к посланию добавляется элемент <fwd> и вложенные в него элементы <via>:

```
<env:Envelope

   xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">

   <env:Header>

      <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
         <m:action>http://some.com/update</m:action>
         <m:to>soap://D.com/endpoint</m:to>
         <m:fwd>
            <m:via>soap://B.com</m:via>
            <m:via>soap://C.com</m:via>
         </m:fwd>
         <m:id>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</m:id>

      </m:path>

   </env:Header>

   <env:Body>
      <!— Содержимое послания... —>
   </env:Body>

</env:Envelope>
```

Наконец, если нужно указать обратный путь, то добавляются элементы <from> И <rev>:

```
<env:Envelope

   xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">

   <env:Header>

      <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
         <m:action>http://www.notification.org/update</m:action>
         <m:to>soap://D.com/endpoint</m:to>

         <m:fwd>
            <m:via>soap://B.com</m:via>
            <m:via>soap://C.com</m:via>
         </m:fwd>

         <m:rev>
            <m:via/>
         </m:rev>

         <m:from>mailto:ivanov@some.com</m:from>
         <m:id>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</m:id>

      </m:path>

   </env:Header>

   <env:Body>
      <!—Содержимое послания... —>
   </env:Body>

</env:Envelope>
```

В этом примере пустой элемент <m:via/> указывает, что выбор обратного пути возлагается на транспортный протокол.

В стандартную поставку JAXM ВХОДИТ пакет com.sun.xml.messaging.jaxm.soaprp, содержащий всего два класса — SOAPRPMessageFactorylmpl И

SOAPRPMessageImpl — **реализующих протокол WS-Routing и создающих SOAP-послания с профилем, названным "SOAP-RP".**

Класс soAPRPMessageimpi **расширяет класс** soAPMessage, **поэтому его экземпляр можно создать приведением экземпляра класса** soAPMessage, **полученного методом createMessage O , К нужному типу:**

```
SOAPRPMessageImpl soaprpMsg =
    (SOAPRPMessageImpl)mf.createMessage();
```

Затем, методами класса soAPRPMessageimpi **и методами, унаследованными от класса** soAPMessage, **создаем SOAP-послание профиля SOAP-RP. При этом полезны следующие методы класса** soAPRPMessageimpi:

- public void newMessageId() — **создает элемент <id> и его содержимое — уникальный идентификатор послания;**
- public void setAction(Endpoint actor) — создает элемент <action> С адресом actor;
- public void setFrom (Endpoint from) — создает элемент <from> с адресом отправителя from;
- public void setTo (Endpoint to) — создает элемент <to> с адресом получателя to;
- public void setRelatesTo (String id) — создает ССЫЛКУ <relatesTo> на послание с идентификатором id;
- public void updateFwdMessagePath(Endpoint actor, int position) — добавляет элемент <via> с адресом actor в позицию position элемента <fwd>;
- public void updateRevMessagePath(Endpoint actor) — добавляет элемент <via> с адресом actor в начало элемента <rev>.

При разборе послания полезны аналогичные методы:

- public string getSOAPRPMessageId () — возвращает содержимое элемента <id> — уникальный идентификатор послания;
- public Endpoint getSOAPRPAction() — возвращает содержимое элемента <action>;
- public Endpoint getFrom() — возвращает содержимое элемента <from> с адресом отправителя;
- public Ebdpoint getTo() — возвращает содержимое элемента <to> с адресом получателя;
- public string getRelatesTo () — возвращает содержимое ссылки <relatesTo>;

- public Vector getSOAPRPFwdMessagePath() — возвращает содержимое элемента <fwd>;
- public Vector getSOAPRPRevMessagePath() — возвращает содержимое элемента <rev>.

После создания и/или изменения послания следует обратиться к методу public void saveChanges();

Пример использования этих методов приведен в листинге 6.4.

Связь с поставщиком сообщений

Связь с поставщиком сообщений устанавливается одним из двух способов.

Первый способ — связаться с заранее определенным поставщиком, называемым *поставщиком по умолчанию*. Это осуществляется последовательным применением двух методов класса ProviderConnectionFactory:

```
ProviderConnectionFactory pcf = ProviderConnectionFactory.newInstance();
ProviderConnection pc = pcf.createConnection();
```

Второй способ — найти поставщика сообщений через систему именования JNDI (Java Naming and Directory Interface) (см., например, [10]) по его JNDI-имени:

```
Context ctx = new InitialContext();

ProviderConnectionFactory pcf =
    (ProviderConnectionFactory)ctx.lookup("SomeProvider");

ProviderConnection pc = pcf.createConnection();
```

Конечно, поставщик сообщений должен быть уже зарегистрирован в системе JNDI под определенным именем. В примере это имя "SomeProvider".

И В ТОМ, И В Другом случае получаем объект pc типа ProviderConnection. Интерфейс ProviderConnection описывает четыре метода работы с поставщиком услуг.

Установив связь с поставщиком, надо получить сведения о нем первым методом интерфейса ProviderConnection:

```
public ProviderMetaData getMetaData ();
```

Этот метод заносит сведения о поставщике в объект типа ProviderMetaData. Объект содержит имя поставщика и номер версии поставщика, состоящий из двух чисел "major.minor", записанных через точку, вроде "1.2". Их можно получить методами

```
public String getName();
public int getMajorVersion();
public int getMinorVersion();
```

Кроме этого, объект типа ProviderMetaData содержит список имен профилей, различаемых поставщиком. Этот список можно получить в виде массива строк методом

```
public String[] getSupportedProfiles();
```

Получив сведения о поставщике и выбрав нужный профиль послания можно начать создание послания.

Создание SOAP-послания и его отправка

Создание SOAP-послания осуществляется методами класса MessageFactory из пакета SAAJ, так же, как это делалось в предыдущем разделе этой главы, но сначала надо получить экземпляр класса MessageFactory вторым методом интерфейса ProviderConnection

```
public MessageFactory createMessageFactory(String profile);
```

Затем, используя полученный экземпляр, создаем SOAP-послание с дополнениями, если они нужны, и отправляем его получателю. Создание послания сильно зависит от выбранного профиля. В листинге 6.4 создается SOAP-послание профиля SOAP-RP.

Отправка SOAP-послания выполняется третьим методом интерфейса ProviderConnection:

```
public void send(SOAPMessage message);
```

Как видите, метод send () не возвращает никакого значения и не требует, чтобы клиент, пославший сообщение, ждал ответа. Тем не менее, после работы надо закрыть связь с поставщиком четвертым методом интерфейса ProviderConnection;

```
public void close O;
```

Листинг 6.4 показывает пример клиента системы JAXM, создающего SOAP-послание профиля SOAP-RP с дополнением.

Листинг 6.4. Создание и асинхронная отправка SOAP-послания

```
import java.net.*;
import javax.xml.messaging.*;
import javax.xml.soap.*;
import javax.activation.*;
```

```
import com.sun.xml.messaging.jaxm.soaprp.*;

public class ClientJAXM{

   public static void main(String[] args){

      String from ="http://some.com/soaprp/sender";
      String to = "http://another.com/soaprp/receiver";

      try{
        ProviderConnectionFactory pcf =
             ProviderConnectionFactory.newInstance();
        ProviderConnection pc = pcf.createConnection();

        ProviderMetaData metaData = pc.getMetaData();
        String[] supportedProfiles =
           metaData.getSupportedProfiles();
        String profile = null;

        for (int i=0; i < supportedProfiles.length; i++){

           if (supportedProfiles[i].equals("soaprp")){
              profile = supportedProfiles[i];
              break;
           }
        }
        MessageFactory mf = pc.createMessageFactory(profile);

        SOAPRPMessageImpl soaprpMsg =
            (SOAPRPMessageImpl)mf.createMessage();

        soaprpMsg.setFrom(new Endpoint(from));
        soaprpMsg.setTo(new Endpoint(to));

        URL url =
           new URL("http://127.0.0.1:8080/jaxm-soaprp/data.xml");

        AttachmentPart ap =
           soaprpMsg.createAttachmentPart(new DataHandler(url));
```

```
        ap.setContentType("text/xml");

        soaprpMsg.addAttachmentPart(ap);

        pc.send(soaprpMsg);

        pc.close();

      }catch(Exception e){
        System.err.println(e);
      }
    }
}
```

Итак, мы научились писать клиентские приложения Web-служб на основе пакетов SAAJ и JAXM. Посмотрим, как можно создавать SOAP-серверы с помощью этих пакетов.

В технологии Java SOAP-серверы реализуются классами специального вида, называемыми сервлетами. Сервлеты подробно описаны в книгах [10, 11]. Нам не обязательно знать все тонкости их создания и употребления, приведем здесь только необходимые сведения о них.

Сервлеты

Сервлетом (servlet) формально называется всякий класс, реализующий интерфейс `Servlet` из пакета `javax.servlet`. Основу сервлета составляют метод `init()`, выполняющий начальные действия сервлета, метод `destroy()`, завершающий работу сервлета, и метод `service()`, в котором заключена вся работа сервлета. Основная особенность сервлета в том, что он работает не сам по себе, а в составе Web-сервера.

Когда от клиента приходит запрос к сервлету, Web-сервер, а точнее один из его модулей, называемый *Web-контейнером,* загружает сервлет и обращается сначала к его методу `init()`, а затем к методу `service()`. Метод `init()` выполняется только один раз, при загрузке сервлета, при следующих запросах к сервлету сразу выполняется метод `service()`. Это очень похоже на работу апплета в браузере, откуда и произошло слово "сервлет".

У метода `service()` два аргумента, его заголовок выглядит так:

```
public void service(ServletRequest req, ServletResponse resp)
    throws ServletException, IOException;
```

Первый аргумент req — это ссылка на объект типа ServletRequest, созданный Web-контейнером при получении запроса от клиента, еще до обращения к методу service(). Объект содержит в себе запрос клиента, различные параметры и атрибуты которого сервлет может получить несколькими методами вида getParameter(String name), getAttribute(String name). Для получения содержимого запроса интерфейс ServletRequest предоставляет два входных потока: байтовый поток класса ServletInputStream и обычный символьный поток класса BufferedReader из пакета java.io.

Второй аргумент resp — это ссылка на пустой объект типа servletResponse, тоже созданный Web-контейнером. Метод service о разбирает запрос, заключенный в объект req, методами интерфейса ServietRequest, обрабатывает его, и результат обработки заносит в объект resp методами интерфейса ServletResponse. Для этого интерфейс ServletResponse предоставляет байтовый выходной поток класса ServletOutputStream И СИМВОЛЬНЫЙ ВЫХОДНОЙ ПОТОК класса PrintWriter.

После заполнения объекта resp сервлет заканчивает работу. Web-контейнер анализирует содержимое объекта resp, формирует ответ, и пересылает его клиенту через Web-сервер, в котором работает Web-контейнер.

В пакете javax.servlet есть абстрактный класс GenericServlet, реализующий все методы интерфейса Servlet за исключением метода service о. Как правило, сервлеты, обрабатывающие произвольные Web-запросы, расширяют класс GenericServiet, реализуя метод service () .

В листинге 6.5 дан пример простого сервлета, отправляющего обратно содержимое полученного запроса.

Листинг 6.5. Простой сервлет

```
import javax.servlet.*;
import java.io.*;

public class EchoServlet extends GenericServlet{

   public void service(ServletRequest req,
                       ServletResponse resp)
               throws ServletException, IOException{

      Servietinputstream sis = req.getInputStream();
      ServletOutputStream sos = resp.getOutputStream();

      int b = 0;
      while ((b = sis.read()) != -1) sos.write(b);
```

```
        sis.close();
        sos.close();
    }
}
```

Класс `EchoServlet` компилируется обычным образом и устанавливается в Web-контейнер. *Установка* (deployment) сервлета в контейнер — это процедура, в результате которой Web-контейнер "узнает" о существовании сервлета, может найти его и загрузить. В процессе установки можно задать начальные параметры сервлета. При создании экземпляра сервлета эти начальные параметры заносятся в объект типа `ServletConfig`, из которого их можно получить методами `getInitParameter(string)`. Еще один метод интерфейса `ServletConfig` — метод `getServletContext()` — позволяет получить контекст, в котором выполняются все сервлеты Web-приложения, в виде объекта типа `ServletContext`.

Процедура установки зависит от конкретного Web-контейнера. Обычно надо поместить сервлет в какой-нибудь каталог Web-контейнера и записать путь к нему в определенный конфигурационный файл. Многие Web-контейнеры предлагают утилиты установки, например, у J2EE-сервера, входящего в состав J2EE, есть утилита `deploytool`.

Интерфейс `Servlet` и класс `GenericServlet` ничего не знают о протоколе, по которому прислан запрос. Они не учитывают особенности протокола. Ввиду особой важности протокола HTTP для Web-приложений, для обработки HTTP-запросов создан абстрактный класс `HttpServlet` — расширение класса GenericServiet. В класс HttpServiet введены специализированные методы обработки различных HTTP-методов передачи данных:

```
protected void doDelete(HttpServletRequest req,
                        HttpservletResponse resp);
protected void doGet(HttpServletRequest req,
                     HttpservletResponse resp);
protected void doHead(HttpServletRequest req,
                      HttpservletResponse resp);
protected void doOptions(HttpServletRequest req,
                         HttpservletResponse resp);
protected void doPost(HttpServletRequest req,
                      HttpservletResponse resp);
protected void doPut(HttpServletRequest req,
                     HttpservletResponse resp);
protected void doTrace(HttpServletRequest req,
                       HttpservletResponse resp);
```

Как видно из сигнатур этих методов, тип их аргументов тоже изменен. В интерфейс HttpServletRequest, расширяющий интерфейс ServletRequest, введены методы getxxx() получения различных частей адреса URL, полей HTTP-заголовка и cookie. В интерфейс `HttpServletResponse`, расширяющий интерфейс `ServletResponse`, введены аналогичные методы `setXxx()`, формирующие заголовок HTTP-ответа и cookie.

После загрузки сервлета класса `HttpServlet`, Web-контейнер, как обычно, обращается последовательно к его методам `init()` и service о. Метод service о реализован в классе HttpServiet как диспетчер. Он только определяет HTTP-метод передачи данных и обращается к соответствующему методу `doXxx()` обработки данного HTTP-метода. Разработчику не надо переопределять метод service (), достаточно определить методы `doXxx ()`, чаще всего определяются два метода: doGet () и doPost ().

В листинге 6.6 приведен пример сервлета, принимающего HTTP-запрос вида

```
GET /servlet/HttpEchoServlet?name=Иванов&age=27 HTTP/1.1
```

пришедший по методу GET или POST, и отправляющий его обратно в виде страницы HTML.

Листинг 6.6. Сервлет, работающий по протоколу HTTP

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class EchoHttpServlet extends HttpServlet{

   public void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
               throws ServletException, IOException{

      doPost(req, resp);
      }

   public void doPost(HttpServletRequest req,
                      HttpservietResponse resp)
               throws ServletException, IOException{

      req.setCharacterEncoding("windows-1251");

      String name = req.getParameter("name");
```

```
        String age = req.getParameter("age");

        resp.setContentType("text/html;charset=windows-1251");

        PrintWriter pw = resp.getWriter();

        pw.println("<html><head><title>Echo</title></head>");
        pw.println("<body><h2>Результаты запроса:</h2>");
        pw.println("Имя: " + name + "<br/>");
        pw.println("Возраст: " + age + "<br/>");
        pw.println("</body></html>");

        pw.flush();
        pw.close();
    }
}
```

Класс HttpServiet вместе С интерфейсами HttpServletRequest, HttpServietResponse и сопутствующими интерфейсами и классами, составляет пакет `javax.servlet.http`.

В технологии Java естественно оформлять Web-службы в виде сервлетов. Если Web-служба будет работать только по протоколу HTTP, то удобно создавать сервлеты, расширяющие класс HttpServiet. Если Web-служба должна работать по нескольким протоколам, то лучше расширить класс GenericServlet ИЛИ прямо реализовать интерфейс Servlet.

И в том, и в другом случае сервлету приходится самому оформлять ответ в виде SOAP-послания, явно выписывая все его элементы XML. Даже теги странички HTML, как видно из предыдущего примера, надо записывать в методах `println()`.

Для того чтобы облегчить создание сервлетов, принимающих и отправляющих SOAP-послания, в пакете `javax.xml.messaging` приведено одно расширение класса HttpServiet. Это абстрактный класс JAXMServiet. Он предназначен для получения, обработки и отправки SOAP-посланий, оформленных средствами JAXM.

Сервлеты класса *JAXMServiet*

Класс JAXMServiet добавляет к классу HttpServiet три метода.

Первый метод

```
protected static MimeHeaders getHeaders(HttpServletRequest req);
```

заносит все поля HTTP-заголовка запроса req в объект класса MimeHeaders.

Второй метод

```
protected void putHeaders(MimeHeaders headers,
    HttpServletResponse resp);
```

формирует HTTP-заголовок ответа resp из заранее подготовленного объекта headers.

Третий метод

```
public void static setMessageFactory(MessageFactory factory);
```

заносит В сервлет экземпляр factory класса MessageFactory.

В сервлете класса JAXMServlet Всегда есть объект класса MessageFactory. Это защищенное (protected) поле сервлета. Оно называется msgFactory. Объект, на который ссылается msgFactory, по умолчанию создается в методе init () обычным для системы JAXM методом:

```
msgFactory = MessageFactory.newInstance();
```

Кроме введения трех новых методов и переопределения метода init (), создающего объект msgFactory, класс JAXMServlet реализовал один из методов doxxx(), а именно метод doPost(). Эта реализация вначале читает HTTP-заголовок и содержимое запроса:

```
MimeHeaders headers = getHeaders(req);
InputStream is = req.getInputStream();
```

и воссоздает исходное SOAP-послание в виде объекта класса soAPMessage, ИСПОЛЬзуя объект msgFactory:

```
SOAPMessage msg = msgFactory.createMessage(headers, is);
```

Затем SOAP-послание msg передается на обработку методу onMessage(). Об этом методе поговорим чуть ниже.

Если сервлет обслуживает асинхронный прием посланий, не требующих ответа, то на этом работа метода doPost () и сервлета заканчивается, а в объект resp заносится код ответа сервера "204 No Content".

Если же сервлет работает по принципу P2P, осуществляя синхронную передачу посланий, то в объект resp заносится результат работы метода onMessage (). Он-то и будет ответом SOAP-сервера на запрос клиента.

На этом метод doPost о завершает свою работу.

Таким образом, в сервлете класса JAXMServlet обработка SOAP-послания и формирование ответа на него выполняется не методом doPost (), а методом onMessage (), не ВХОДЯЩИМ В класс JAXMServlet. Метод onMessage() должен быть определен во всяком классе, расширяющем класс JAXMServlet, но не переопределяющем его метод doPost о .

Кроме того, сервлет, расширяющий класс JAXMServlet, реализует собой либо Web-службу, работающую с клиентом синхронно, либо Web-службу, работающую асинхронно. Реализация осуществляется так. Если Web-служба будет работать асинхронно, то сервлет должен реализовать интерфейс OnewayListener, если синхронно — интерфейс ReqRespListener. В каждом из этих интерфейсов описано всего по одному методу. Это и есть тот метод onMessage () , о котором мы говорили.

В интерфейсе OnewayListener метод onMessage о не возвращает результат, его заголовок таков:

```
public void onMessage(SOAPMessage msg);
```

В интерфейсе RegRespListener результатом метода будет новое **SOAP**-послание:

```
public SOAPMessage onMessage(SOAPMessage msg);
```

Если метод возвратил null, то ответ не будет передан клиенту.

Итак, если мы хотим создать синхронную Web-службу, основанную на пакете **SAAJ,** то расширяем класс JAXMServiet и реализуем интерфейс ReqRespListener, определяя метод onMessage о . Схема сервлета такова:

```
import javax.xml.messaging.*;

public class SyncSAAJServlet extends JAXMServiet
        implements ReqRespListener{

   public SOAPMessage onMessage(SOAPMessage msg){
      // Разбираем послание msg, формируем ответное послание...
   }
}
```

Если же мы решили создать асинхронную Web-службу **на** основе пакета **JAXM,** то в сервлете, расширяющем класс JAXMServiet, следует реализовать интерфейс OnewayListener. Вот схема такого сервлета:

```
import javax.xml.messaging.*;

public class AsyncJAXMServlet extends JAXMServiet
        implements OnewayListener{

   public void onMessage(SOAPMessage msg){
      // Разбираем послание msg...
   }
}
```

В листинге 6.7 показана реализация синхронной Web-службы, выводящей полученное послание в свой стандартный вывод и посылающей клиенту подтверждение в получении запроса. Реализация сделана в системе SAAJ сервлетом, который мы назвали AckJAXMServlet.

Листинг 6.7. Сервлет, отправляющий клиенту подтверждение

```
import javax.xml.messaging.*;
import javax.xml.soap.*;

public class AckJAXMServlet extends JAXMServlet

                        implements ReqRespListener{

   public SOAPMessage onMessage(SOAPMessage msg){

     try{
        System.out.println("Содержимое полученного запроса:\n");
        msg.writeTo(System.out);
        SOAPMessage reply = msgFactory.createMessage();
        SOAPEnvelope env = reply.getSOAPPart().getEnvelope();

        env.getBody().addChildElement(
             env.createName("Response"))
                  .addTextNode("Запрос получен.");

        return reply;

     }catch(Exception e){
        System.err.println(e);
        return null;
     }
   }
}
```

Послания процедурного стиля

Процедурный стиль SOAP-послания, предполагающий вызов удаленной процедуры, расположенной на SOAP-сервере, и получение ее результата, реализован в технологии Java пакетом JAX-RPC. Впрочем, средства пакета JAX-RPC позволяют реализовать и документный стиль SOAP-посланий.

Реализация пакета JAX-RPC основана на механизме обращения к методам удаленных объектов RMI (Remote Method Invocation), с которым можно познакомиться, например, по книге [10]. Механизм RMI так глубоко спрятан в пакет JAX-RPC, что вы можете ничего про него и не знать. Он проявляет себя только в том, что удаленные объекты должны реализовать интерфейс, расширяющий интерфейс Remote из пакета `java.rmi`. Такой интерфейс называется *удаленным* (remote) или *распределенным* (distributed) интерфейсом. Про интерфейс Remote тоже можно ничего не знать, потому что он пуст, он не содержит констант и заголовков методов. Пометка "extends Remote" в заголовке удаленного интерфейса означает только, что объекты этого типа будут распределенными объектами, доступными удаленным клиентам.

Итак, при создании Web-службы средствами JAX-RPC сначала записывается удаленный интерфейс, а Web-служба строится как реализация этого удаленного интерфейса.

Суть механизма RMI заключается в том, что после того как клиент обратился к методу удаленного объекта, на его машину пересылается программный код, называемый *заглушкой* (stub). Заглушка оформлена как класс Java, реализующий удаленный интерфейс, но реализация содержит не Web-услуги, а средства сетевого соединения, создания SOAP-посланий, включая их сериализацию, и средства пересылки SOAP-посланий.

На сервере тоже создается дополнительный профаммный код, называемый *серверной заглушкой* или *связкой* (tie). Связка получает сообщение от клиентской заглушки, десериализует содержащееся в нем SOAP-послание, извлекает из него аргументы Web-услуги и передает их Web-службе. Результат выполнения Web-услуги проходит обратный путь. Он сериализуется связкой и пересылается клиентской заглушке. Та десериализует полученное сообщение, извлекает из него результат Web-услуги и передает клиенту как результат обращения к удаленному методу. Эта схема показана на рис. 6.2.

Таким образом, с точки зрения клиента создается обычный локальный объект Java, и обычным образом выполняются его методы, как будто они работают прямо на клиентской машине.

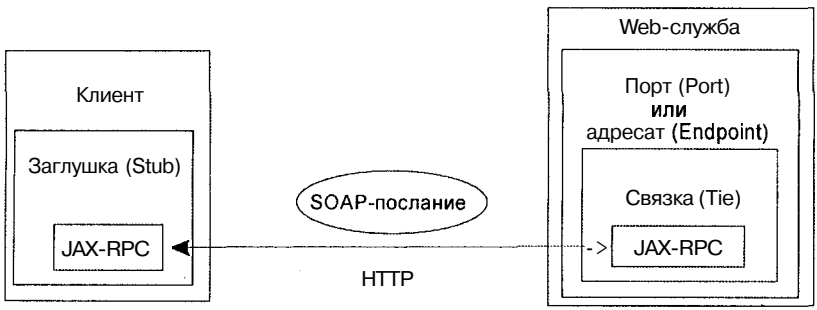


Рис. 6.2. Механизм JAX-RPC

Создание Web-службы средствами JAX-RPC

Из предыдущего описания видно, что для создания Web-службы надо, прежде всего, написать удаленный интерфейс, называемый в спецификации JAX-RPC "Service Endpoint Interface". Мы будем называть его *адресатом* или *SEI-интерфейсом* Web-службы. В WSDL-описании Web-службы адресату соответствует элемент <portType>. Поэтому SEI-интерфейс вместе с его реализацией часто называется *портом* (port) Web-службы. Описанные в нем методы как раз и составляют Web-услуги создаваемой Web-службы. На этот интерфейс накладываются следующие условия:

- он должен быть открытым (public);
- он должен прямо или косвенно расширять интерфейс Remote;
- он не должен содержать константы;
- его методы должны выбрасывать исключение класса RemoteException;
- другие исключения, выбрасываемые методами, могут быть только проверяемого (checked) типа;
- аргументы методов и их возвращаемые значения могут быть только допустимых типов и не могут прямо или косвенно реализовывать интерфейс Remote.

SEI-интерфейс Web-службы может выглядеть примерно так, как показано в листинге 6.8.

Листинг 6.8. Удаленный интерфейс Web-службы (порт)

```
package bookinfo;

import java.rmi.*;

public interface BookInfoPort extends Remote{

   Book getInfo(String isbn) throws RemoteException;

   void setInfo(Book book) throws RemoteException;
}
```

К допустимым типам аргументов и возвращаемых значений относятся:

- примитивные типы Java, кроме типа char, то есть типы boolean, byte, short, int, long, float, double;
- соответствующие классы-оболочки Boolean, Byte, Short, Integer, Long, Float, Double;

- **классы** String, Date, Calendar, BigInteger, BigDecimal, Object;
- массивы, элементы которых относятся к перечисленным выше типам;
- классы-коллекции типа Collection, Map, set;
- классы Java, называемые в спецификации JAX-RPC "value types".

Классы, названные "value types" — это обычные классы Java, удовлетворяющие трем требованиям. Они не реализуют прямо или косвенно интерфейс Remote, имеют открытый (public) конструктор по умолчанию, все их открытые (public) поля могут быть только допустимых для JAX-RPC типов. Например, класс Book, использованный в листинге 6.8, может выглядеть так:

```
package bookinfo;

public class Book{

   private String isbn;
   private String[] authors;
   private String title;

   public Book(){}

   public String getIsbn(){ return isbn; }
   public void setIsbn(String isbn){ this.isbn = isbn; }

   public String[] getAuthors (){ return authors; }
   public void setAuthors(String[] authors){ this.authors = authors; }

   public String getTitle(){ return title; }
   public void setTitle(String title){ this.title = title; }
}
```

Ограничения типов данных вызваны тем, что аргументы и возвращаемые значения сериализуются значениями типов языка XSD (см. *главу 1*). Например, типы int **и** Integer **сериализуются типом** xsd:int, **тип** BigInteger **сериализуется типом** xsd:integer, **типы** Date **и** calendar **сериализуются** типом xsd:dateTime, массивы типов byte[] и Byte[] — типом xsd:base64Binary. Типы классов-коллекций требуют наличия специальных классов, преобразующих их в структуры и массивы языка XSD.

После создания SEI-интерфейса надо его реализовать классом, который и будет представлять Web-службу. Этот класс в спецификации JAX-RPC называется "Service Endpoint Class". Часто его называют *служителем* (servant).

Очень часто имя класса-служителя образуется из имени SEI-интерфейса добавлением окончания "Impl".

Класс-служитель должен удовлетворять следующим требованиям:

- класс должен быть открытым (public), не абстрактным (abstract) и не окончательным (final);
- у него должен быть открытый конструктор по умолчанию;
- методы, реализующие интерфейс, не должны быть окончательными(final) или статическими (static);
- класс не должен реализовать метод finalize {).

В листинге 6.9 показано, как можно реализовать Web-службу BookInfo средствами JAX-RPC.

Листинг 6.9. Реализация Web-службы средствами JAX-RPC

```
package bookinfo;

import javax.xml.rpc.server.*;
import javax.servlet.ServletContext;
import java.sql.*;
import java.util.*;

public class BookInfoImpl
                implements BookInfoPort, ServiceLifecycle{

   private ArrayList books;
   private Connection con;

   public void init(Object context){

      ServletContext sc =

         ((ServletEndpointContext)context).getServletContext();

      String dbUrl =sc.getInitParameter("dbUrl");

     try{
        con = DriverManager.getConnection(dbUrl);
     }catch(Exception e){}
```

```
    }

    public void destroy(){

        setToDB();
        con = null;
    }

    public BookInfoImpl(){

      try{
         Statement st = con.createStatement();
         ResultSet rs = st.executeQuery("SELECT * FROM book");

         insertFromDB(rs);

         rs.close();
         st.close();

      }catch(Exception e){}
}

    public Book getInfo(String isbn){

        Book t = null;
        for (int k = 0; k < books.size O; k++){

            t = (Book)books.get(k);
            if (isbn.equals(t.getIsbn())) break;
        }
        return t;
}

    public void setInfo(Book book){

        books.add(book);
    }

    private void insertFromDB(ResultSet rs){
```

```
        // Заполнение списка книг books выборкой из базы
    }

    private void setToDB(){

        // Сохранение списка книг books в базе
    }
}
```

Обратите внимание на то, что класс-служитель — это не сервлет. Когда Web-служба будет устанавливаться в Web-контейнер, утилита установки автоматически создаст сервлет, который будет обращаться к методам класса-служителя. Обычно этот сервлет называется `JAXRPCServlet`, он входит в каждую реализацию пакета **JAX-RPC.**

Жизненный цикл Web-службы

В листинге 6.9 реализуется еще один интерфейс `ServiceLifecycle`, описывающий жизненный цикл Web-службы. Описание производится двумя методами:

```
public void init(Object context);
public void destroy();
```

выполняющимися автоматически Web-контейнером под управлением исполняющей системы **JAX-RPC** по одному разу при создании экземпляра Web-службы и при завершении его работы соответственно.

Исполняющая система создает и передает в метод `init()` объект `context`, содержащий сведения о Web-контейнере, в котором работает Web-служба. Тип этого объекта указан просто как `Object`, но им чаще всего бывает контекст сервлета типа `ServletContext` или специально разработанный тип, описанный интерфейсом ServletEndpointContext.

Контекст Web-службы

Интерфейс ServletEndpointContext описывает четыре метода, предоставляющие сведения об окружении, в котором работает Web-служба.

- public HttpSession `getHttpSession()` — передает сведения о **HTTP**-сеансе сервлета, под управлением которого работает Web-служба или null, если сервлет не участвует в сеансе;
- public `MessageContext getMessageContext()` — передает сведения О полученном сообщении;

- public ServletContext getServletContext() — передает контекст сервлетов Web-приложения, в котором работает Web-служба;
- public Principal getUserPrincipal() — передает имя пользователя, обратившегося к Web-службе.

Среди этих сведений особенно интересен интерфейс MessageContext, даже не он, а его расширение SOAPMessageContext.

Методы интерфейса MessageContext позволяют установить, получить, удалить свойства сообщения и узнать, есть ли свойство с заданным именем:

```
public void setProperty(String name, Object value);
public Iterator getPropertyNames();
public Object getProperty(String name);
public void removeProperty(String name);
public boolean containsProperty(String name);
```

Интерфейс SOAPMessageContext добавляет к этим методам еще три метода, которые позволяют получить и изменить присланное SOAP-послание:

```
public SOAPMessage getMessage();
public void setMessage(SOAPMessage message);
public String [] getRoles();
```

Применяя эти методы в классах-обработчиках, можно сделать предварительную обработку SOAP-послания и направить его другой Web-службе. Рассмотрим подробнее эти возможности.

Предварительная обработка послания

Средствами JAX-RPC можно сделать предварительную обработку **SOAP**-послания: раскодировать или закодировать сообщение, проверить подлинность сообщения, подтвердить права клиента. Очень часто предварительно обрабатываются блоки заголовка SOAP-послания, предназначенные для обработки промежуточными серверами, но не прошедшие ее.

Основу обработки задает интерфейс Handler. Он описывает методы, позволяющие обработать полученное SOAP-послание, отправляемое **SOAP**-послание и сообщение об ошибке:

```
public boolean handleRequest(MessageContext context);
public boolean handleResponse(MessageContext context);
public boolean handleFault(MessageContext context);
```

Хотя типом аргумента этих методов назван интерфейс MessageContext, на самом деле это тип SOAPMessageContext. Это дает возможность получить для обработки само SOAP-послание.

Кроме того, в интерфейсе Handler есть метод

```
public QName[] getHeaders();
```

позволяющий получить блоки заголовка SOAP-послания.

Еще два метода

```
public void init(HandlerInfo config);
public void destroy();
```

выполняются при создании объекта типа Handle и его удалении соответственно. В метод `init()` передается ссылка на объект класса `HandlerInfo`, содержащий сведения, необходимые контейнеру для инициализации класса-обработчика, а также сведения о цепочке классов-обработчиков SOAP-послания.

Дело в том, что для обработки SOAP-послания сразу создается целая цепочка классов-обработчиков, даже если в нее входит только один обработчик. Цепочка обработчиков — это объект типа `HandlerChain`. Интерфейс `HandlerChain` расширяет интерфейс List, значит, описывает упорядоченный список объектов типа Handler. К многочисленным методам интерфейса List и его предка — интерфейса `Collection`, позволяющим работать со списком — добавлять, удалять и перебирать элементы списка, — интерфейс добавляет следующие методы:

```
public boolean handleRequest(MessageContext context);
public boolean handleResponse(MessageContext context);
public boolean handleFault(MessageContext context);
public void init(Map config);
public void destroy();
public String[] getRoles();
public void setRoles(string[] actorNames);
```

Как видите, первые пять методов похожи на методы интерфейса Handler. Они включены в интерфейс HandierChain для того чтобы его объекты можно было включить в другую цепочку, получив тем самым вложенную цепочку классов-обработчиков. Последние два метода позволяют цепочке играть роль "next" одного или нескольких промежуточных серверов (actors), обрабатывающих блоки заголовка SOAP-послания (см. *главу 3).*

Методы `handleXxx()` сделаны логическими именно потому, что они участвуют в цепочке — если метод handleRequest () вернул логическое значение `true`, то обработка пришедшего SOAP-послания передается следующему в цепочке методу handleRequest o, или адресату Web-службы, если обработчик был последним в цепочке. Если метод handleRequest () вернул значение false, то следующие обработчики не вызываются, а послание передает-

ся методу handleResponse() того же экземпляра класса-обработчика и дальше идет по цепочке методов handieResponse(). Если метод handleRequest() выбросил исключение класса SOAPFaultException, то управление передается методу handleFault() того же экземпляра класса-обработчика. Впрочем, разработчик может изменить эту стандартную последовательность действий по своему усмотрению.

Для удобства реализации интерфейса Handler, в пакете JAX-RPC имеется его частичная реализация — абстрактный класс GenericHandler. Реализация тривиальна — все логические методы просто возвращают значение true, прочие методы пусты, и только метод getHeaders о оставлен абстрактным.

Итак, класс-обработчик может выглядеть следующим образом:

```
public class MyHandler extends GenericHandler{

   public MyHandler(){}
   public boolean handleRequest(MessageContext context){
      try{
       SOAPHeader sh = ((SOAPMessageContext)context)
          .getMessage().getSOAPPart().getEnvelope().getHeader();

        // Обрабатываем блоки заголовка sh

       return true;

      }catch(Exception ex){ }
   }
}
```

Цепочку классов-обработчиков SOAP-послания следует зарегистрировать у адресата или у клиента Web-службы. Для этого предназначен интерфейс HandlerRegistry, описывающий два метода доступа к цепочке:

public List getHandlerChain(QName portName);
public void setHandlerChain(QName portName, List chain);

Объект типа HandierRegistry можно получить методом

public HandierRegistry getHandlerRegistry();

интерфейса Service, о котором речь пойдет ниже.

Продолжим наш пример. Получаем цепочку и включаем в нее класс MyHandler:

```
QName serviceName = new QName("myservice");
```

```
ServiceFactory sf = ServiceFactory.newInstance();
Service serv = sf.createService(serviceName);
HandlerRegistry reg = serv.getHandlerRegistry();

List chain = reg.getHandlerChain(serv.getServiceName());
chain.add(new HandlerInfo(MyHandler.class, null, null));
```

Кроме регистрации классов-обработчиков в цепочке, их следует указать в конфигурационном файле. Как это сделать — рассказано ниже в этой главе.

Компиляция файлов Web-службы

Интерфейс `BookInfoPort` и классы `BookInfoImpl`, Book, а также другие нужные классы компилируются обычным образом

```
javac *.java
```

Кроме интерфейса-порта, реализующего его класса и сопутствующих классов, для работы Web-службы надо получить еще несколько классов, обеспечивающих реализацию механизма RMI. Это, в частности, клиентская заглушка и серверная связка, классы, сериализующие и десериализующие сложные типы. Кроме того, надо сделать описание Web-службы на языке WSDL.

Создать вручную эти классы очень сложно, они используют специфику механизма RMI, а это весьма запутанная система. Пакет Sun WSDP предлагает три утилиты для создания классов JAX-RPC. Это уже успевший устареть компилятор `xrpcc`, создающий и заглушки и связки, а также новые утилиты: компилятор `wsdeploy`, создающий серверные связки и описания WSDL, и компилятор `wscompile`, создающий клиентские заглушки.

Компилятор xrpcc работает из командной строки. Для создания серверных файлов он запускается примерно так:

```
$ xrpcc -server config.xml
```

Для создания клиентских файлов указывается флаг `-client`. Если надо сразу создать все файлы, и серверные, и клиентские, то указывается флаг `-both`.

Для работы компилятора xrpcc нужны уже откомпилированные SEI-интерфейс и его реализация классом-служителем, а также специальный конфигурационный XML-файл, описывающий Web-службу. Этот файл обычно называется config.xml. Его структура описана в документации, приложенной к компилятору xrpcc, и может в дальнейшем измениться вместе с ним.

Компилятор wsdeploy тоже запускается из командной строки следующим образом:

```
$ wsdeploy -tmpdir temp -o deployable.war portable.war
```

Он получает информацию из архива deployable.war (имя файла может быть другим), в котором лежат откомпилированные классы, специальный конфигурационный файл с именем jaxrpc-ri.xml и конфигурационный DD-файл (Deployment Descriptor) Web-приложения web.xml [10]. Компилятор сразу же упаковывает все файлы созданной Web-службы в архив, в нашем примере это файл portable.war. Имя архива произвольно, обычно оно отражает название Web-службы.

Флаг -tmpdir указывает каталог для временных файлов, в примере это подкаталог temp текущего каталога. Если дополнительно указать флаг -keep, то в этом каталоге останутся исходные файлы всех классов.

После этого остается установить Web-службу, то есть архивный файл (в примере — это файл portable.war), в Web-контейнер как обычное Web-приложение. Утилита установки, в частности, создает сервлет, который будет представлять класс-служитель (tie) в Web-контейнере.

Конфигурационный файл компилятора jaxrpc-ri.xml

Конфигурационный файл jaxrpc-ri.xml компилятора wsdeploy — это XML-файл с корневым элементом <webServices>, в который вкладывается всего два элемента — <endpoint> и <endpointMapping>. Каждый элемент может появиться сколько угодно раз.

У корневого элемента <webServices>, кроме атрибута xlmns есть один обязательный атрибут version и три необязательных атрибута. Атрибуты targetNamespaceBase И typeNamespaceBase задают идентификаторы пространств имен описания WSDL И типов данных, а атрибут urlPatternBase показывает начальный контекст Web-службы.

Элемент <endpoint> описывает адресата Web-службы следующими атрибутами:

- обязательный атрибут name определяет название Web-службы, оно будет служить именем файла WSDL с описанием Web-службы;
- необязательный атрибут displayName определяет название Web-службы, используемое утилитами JAX-RPC и Web-контейнером в заголовках и пунктах меню;
- необязательный атрибут description содержит произвольное описание Web-службы;
- необязательный атрибут interface содержит имя SEI-интерфейса;

- необязательный атрибут implementation содержит имя класса-служителя;
- необязательный атрибут model дает ссылку в виде строки URI на дополнительное описание Web-службы.

В элемент <endpoint> может быть вложен элемент <handlerChains>, описывающий цепочку классов-обработчиков. Каждый класс-обработчик описывается Элементом <chain>.

Элемент <endpointMapping> содержит путь к файлу с описанием **WSDL. Он** не содержит вложенных элементов, у него есть только два обязательных атрибута:

- атрибут endpointName содержит имя файла с описанием WSDL;
- атрибут urlPattern определяет путь к нему.

Вот схема файла jaxrpc-ri.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<webServices
   xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
   version=" xsd:string"
   targetNamespaceBase=" xsd:string"
   typeNamespaceBase=" xsd:string"
   urlPatternBase=" xsd:string">

   <endpoint
      name="xsd:string"
      displayName="xsd:string"
      description="xsd:string"
      interface=" xsd:string"
      implementation=" xsd:string"
      model="xsd:anyURI"> [*]

      <handlerChains> [?]
         <!— Цепочка классов-обработчиков —>
      </handlerChains>

   </endpoint>

   <endpointMapping
```

```
        endpointName=" xsd:string"
        urlPattern=" xsd:string" /> [*]

</webServices>
```

В листинге 6.10 приведен конфигурационный файл jaxrpc-ri.xml компилятора wsdeploy нашей примерной Web-службы BookInfo.

Листинг 6.10. Конфигурационный файл компилятора jaxrpc-ri.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<webServices
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
    version="1.0"
    targetNamespaceBase="http://bookinfo.com/wsdl"
    typeNamespaceBase="http://bookinfo.com/types"
    urlPatternBase="/ws">

    <endpoint
        name="BookInfo"
        displayName="BookInfo Service"
        description="Сведения о книгах"
        interface="bookinfo.BookInfoPort"
        implementation="bookinfo.BookInfoImpl" />

    <endpointMapping
        endpointName="BookInfo"
        urlPattern="/bookinfo" />

</webServices>
```

После того как компилятор wsdeploy закончит работу, устанавливаем созданный им архив portable.war в Web-контейнер. Для Web-контейнера Tomcat в пакете WSDP есть графическая утилита установки deploytool. Она запускается из командной строки при работающем Tomcat просто набором ее имени

```
$ depioytool
```

После ее запуска на экране последовательно появляется ряд окон с подробными объяснениями процесса установки.

После установки Web-службы в Web-контейнер, можно проверить ее наличие, открыв на сервере браузер и набрав в нем адрес **http://localhost:8080/services/bookinfo.** В окне браузера появится список установленных Web-служб.

Слово "services" в этом адресе — это контекст сервлета, созданный в процессе установки. Он может получить другое название. Слово "/bookinfo" вместе с начальной наклонной чертой — это значение атрибута `urlPattern` элемента `<endpointName>`, записанного в файле jaxrpc-ri.xml.

Итак, Web-служба создана, установлена в Web-контейнер, контейнер запущен и ждет, когда пойдут запросы. Теперь можно перейти к созданию клиента.

Создание клиента JAX-RPC

Система JAX-RPC позволяет создавать клиентские приложения трех типов. Проще всего использовать в приложении заранее созданные заглушки (generated stubs). Их создает старый компилятор `xrpcc` с флагом `-client`, например, так:

```
$ xrpcc -client -d bookclient config.xml
```

или новый компилятор `wscompile`, работающий из командной строки примерно следующим образом:

```
$ wscompile -gen:client -d bookclient config.xml
```

Оба компилятора читают предварительно написанный конфигурационный файл config.xml и записывают создаваемые файлы в каталог, указанный параметром -d, в примере это каталог с именем bookclient. Кроме того, компилятор xrpcc использует SEI-интерфейс Web-службы, а компилятор wscompile — описание WSDL, причем компилятор смотрит это описание через Web-службу по адресу, например, **http://localhost:8080/services/bookinfo?WSDL**. Поэтому Web-служба уже должна быть установлена в Web-контейнер, а контейнер должен работать.

Итак, перед созданием заглушек надо написать конфигурационный файл config.xml. Рассмотрим его структуру.

Конфигурационный файл компилятора config.xml

Корневой элемент конфигурационного файла `<configuration>` и вложенные в него элементы определены в пространстве имен с идентификатором http://java.sun.com/xml/ns/jax-rpc/ri/config. В корневой элемент вкладывается один из трех элементов:

- элемент `<service>`, если Web-служба строится по SEI-интерфейсу;

- элемент <wsdl>, если Web-служба строится по WSDL-описанию;
- элемент <modelfile>, если описание находится в другом файле.

У элемента <service> четыре обязательных атрибута.

- Атрибут name задает название создаваемой Web-службы.
- Атрибут packageName определяет имя пакета, в котором будут расположены создаваемые классы.
- Атрибут targetNamespace содержит идентификатор пространства имен создаваемого описания WSDL.
- Атрибут typeNamespace содержит идентификатор пространства имен типов, определяемых в создаваемом описании WSDL.

В элемент <service> вкладываются нуль или несколько элементов <interface>, описывающих SEI-интерфейсы Web-службы. Полное имя SEI-интерфейса записывается обязательным атрибутом name, полное имя реализующего его класса-служителя — атрибутом servantName. Необязательные атрибуты soAPAction и soAPActionBase задают одноименные поля MIME-заголовка.

У элемента <wsdl> два обязательных атрибута. Атрибут location содержит строку URI с адресом описания WSDL, атрибут packageName — имя пакета с классами Web-службы.

Остальные элементы конфигурационного файла необязательны. Они показаны на следующей схеме, показывающей структуру конфигурационного файла.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration
     xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">

  <service name="xsd:string"
       packageName="xsd:xtring"
       targetNamespace="xsd:anyURI"
       typeNamespace="xsd:anyURI">

    <interface name="xsd:string"
       servantName="xsd:string"

<!— Оставшиеся атрибуты и элементы необязательны —>

       soapAction="xsd:string"
```

```
        soapActionBase="xsd:string">

      <handlerChains>
          <!— Цепочка классов-обработчиков —>
      </handlerChains>

    </interface>

    <typeMappingRegistry>
        <!— Правила сериализации нестандартных типов —>
    </typeMappingRegistry>

    <handlerChains>
        <!— Цепочка классов-обработчиков —>
    </handlerChains>

    <namespaceMappingRegistry>
       <!— Соответствие пространств имен и пакетов Java —>
    </namespaceMappingRegistry>

  </service>

</configuration>
```

В листинге 6.11 приведен конфигурационный файл нашей примерной Web-службы Bookinfo для создания Web-службы по ее SEI-интерфейсу.

Листинг 6.11. Конфигурационный файл компилятора SEI-интерфейса

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration
     xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">

  <service name="BookInfo"
           packageName="book"
           targetNamespace="http://book.org"
           typeNamespace="http://book.org/types">

    <interface name="BookInfoPort"
```

```
                servantName="BookInfoImpl">
    </interface>

  </service>

</configuration>
```

Кроме заглушки BookInfoPort_Stub.class, имя которой составлено из имени SEI-интерфейса и сопутствующих классов, компилятор `xrpcc` создает WSDL-описание Web-службы, в нашем примере это файл BookInfo.wsdl, и файл свойств Web-службы, в примере это файл BookInfo_Config.properties. Имена этих файлов содержат в себе название Web-службы, заданное атрибутом name элемента <service> конфигурационного файла config.xml.

Конфигурационный файл config.xml для компилятора `wscompile`, создающего Web-службу `BookInfo` по ее описанию WSDL, написан в листинге 6.12.

Листинг 6.12. Конфигурационный файл компилятора WSDL-описания

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration
   xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">

  <wsdl location="http://localhost:8080/services/bookinfo?WSDL"
        packageName="bookinfo"/>

</configuration>
```

Клиент, использующий заранее созданные заглушки

Методы использования заглушки описаны в интерфейсе stub пакета `javax.xml.rpc`. Они задают и читают свойства заглушки. Это методы

```
public void _setProperty(String name, Object value);
public Object _getProperty(String name);
public Iterator _getPropertyNames();
```

В интерфейсе описаны четыре стандартных свойства заглушки:

- `USERNAME_PROPERTY` — строка типа string с именем пользователя Web-службы;
- `PASSWORD_PROPERTY` — строка типа string с паролем пользователя Web-службы;

- ENDPOINT_ADDRESS_PROPERTY — **строка типа** string **с URI-адресом Web-службы;**
- SESSION_MAINTAIN_PROPERTY — **объект типа** Boolean, **указывающий, создавать** сеанс связи с Web-службой или нет. По умолчанию сеанс не создается.

Поскольку заглушка — в нашем примере это класс BookInfo_Stub — уже создана, можно создать ее экземпляр как объект обычного класса и обращаться к методам этого объекта. Например:

```
Stub stub = new BookInfo_Stub();
BookInfoPort info = (BookInfoPort)stub;
Book book = info.getInfo("5-94157-041-4");
```

Однако удобнее использовать вспомогательный класс BookInfo_Impl, созданный вместе с заглушкой. Метод этого класса getxxxPort(), где Xxx — название Web-службы, взятое из атрибута name элемента <service> конфигурационного файла config.xml, выполняет необходимые проверки и обработку исключений и возвращает заглушку. С использованием этого класса клиентскую программу можно написать так, как это сделано в листинге 6.13.

Листинг 6.13. Клиент, использующий существующую заглушку

```
import javax.xml.rpc.Stub;
import bookinfo.*;

public class BookInfoStubClient{

   public static void main(String[] args){

      try{
        Stub stub =
            (Stub) (new BookInfo_Impl().getBookInfoPort());

        stub._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,

                "http://localhost:8080/services/bookinfo");

        BooklnfoPort bip = (BookInfoPort)stub;

        Bookbook=bip.getInfo("5-94157-041-4");

        System.out.println(book.getTitle());
```

```
        }catch(Exception e){
          System.err.println(e);
        }
    }
}
```

Клиент, использующий описание WSDL

Клиенты второго типа обращаются к Web-службе по ее описанию, сделанному на языке WSDL. При этом обращении создается заглушка и пересылается на клиентскую машину. Методы, помогающие клиенту системы JAX-RPC обратиться к WSDL-описанию Web-службы, описаны в интерфейсе Service пакета `javax.xml.rpc`. Интерфейс service предназначен для связи с SOAP-сервером и получения информации о вызываемой Web-службе.

Объект типа service создается уже знакомым путем с помощью класса-фабрики ServiceFactory:

```
ServiceFactory sf = ServiceFactory.newInstance();
Service serv = sf.createService(QName serviceName);
```

При создании объекта serv сразу указывается название Web-службы serviceName. Второй метод создания объекта типа service

```
public Service createService(URL wsdl, QName serviceName);
```

требует указать адрес wsdl WSDL-описания Web-службы.

После того как объект типа `Service` создан, можно получить заглушку одним из двух методов `getPort()`.

Если объект serv был создан первым способом и адрес wsdl WSDL-описания ему еще неизвестен или описание содержит несколько SEI-интерфейсов, то применяем метод

```
public Remote getPort(QName wsdlName, Class sei);
```

в котором аргумент wsdlName содержит имя SEI-интерфейса.

Если же эти сведения уже указаны, то можно применить второй метод

```
public Remote getPort(Class sei);
```

В листинге 6.14 показан пример клиента нашей Web-службы `BookInfo`, обращающегося к ней через описание WSDL.

Листинг 6.14. Клиент, использующий описание WSDL

```
import java.net.URL;
import javax.xml.rpc.*;
```

```
import javax.xml.namespace.QName;
import bookinfo.*;

public class BookInfoProxyClient{

   public static void main(String[] args){
     try{
        String ns = "http://bookinfo.com/wsdl/";

        URL wsdl = new URL(

              "http://localhost:8080/services/bookinfo?WSDL");

        ServiceFactory sf = ServiceFactory.newInstance();

        Service serv = sf.createService(wsdl,
                          new QName(ns, "Bookinfo"));
        BookInfoPort bip = (BookInfoPort)serv.getPort(
                new QName(ns, "BooklnfoPort"),
                book.BookInfoPort.class);

        Book book = bip.getInfo("5-94157-041-4");

        System.out.println(book.getTitle());

      }catch(Exception ex)(
          ex.printStackTrace();
      }
   }
}
```

Клиент, динамически получающий Web-услуги

В тех случаях, когда нет заранее созданных заглушек и недоступно WSDL-описание Web-службы, клиент может динамически получить Web-услуги методами, описанными в интерфейсе `Call`. Методы интерфейса `call` задают аргументы вызываемой Web-услуги, вызывают ее и получают результат. Для их использования не нужен даже SEI-интерфейс и сопутствующие ему классы. Достаточно знать название Web-службы, имя **SEI-интерфейса** и сигнатуру метода вызываемой Web-услуги.

Объект типа call создается одним из методов createCall() интерфейса Service. Получив объект serv типа Service, применяем один из следующих его методов:

```
public Call createCall();
public Call createCall(QName portName);
public Call createCall(QName portName, QName operationName);
public Call createCall(QName portName, String operationName);
```

Аргумент portName — это имя SEI-интерфейса Web-службы, представленное в виде объекта класса QName. Аргумент operationName — это имя метода, предоставляющего Web-услугу.

После создания объекта типа call, его нужно дополнить недостающими сведениями. Для этого в интерфейсе Call описаны методы

```
public void setOperationName(QName operationName);
public void setPortTypeName(QName portType);
public void setReturnType(QName xmltype);
public void setTargetEndpointAddress(String address);
```

Кроме того, надо описать типы аргументов вызываемого метода. Это выполняется методами

```
public void addParameter(String name, QName xmlType,
                         ParameterMode mode);
public void addParameter(String name, QName xmlType,
                         Class javaType, ParameterMode mode);
```

Первые аргументы задают произвольное имя для аргумента name, его XSD-тип xmlType, соответствующий тип Java javaType, если для этого типа нет стандартного соответствия типов XML и Java.

Последний аргумент mode — это одна из трех констант класса ParameterMode:

- ParameterMode.IN — указывает, что описываемый аргумент является входным аргументом метода;
- ParameterMode.OUT — указывает, что описываемый аргумент является выходным аргументом метода;
- ParameterMode.INOUT — указывает, что описываемый аргумент может быть и входным и выходным аргументом метода.

Наконец, определяются свойства объекта типа Call, например, имя и пароль пользователя. Свойства устанавливаются методом

```
public void setProperty(String name, Object value);
```

В интерфейсе Call определены имена некоторых свойств.

- USERNAME_PROPERTY — строка типа string с именем пользователя Web-службы;
- PASSWORD_PROPERTY — строка типа string с паролем пользователя Web-службы;
- ENCODINGSTYLE_URI_PROPERTY — строка типа String с идентификатором пространства имен типов;
- SESSION_MAINTAIN_PROPERTY — объект типа Boolean, указывающий, создавать сеанс связи с Web-службой или нет. По умолчанию сеанс не создается.
- OPERATION_STYLE_PROPERTY — строка "rpc" или "document", определяющая процедурный или документный стиль SOAP-послания;
- SOAPACTION_USE_PROPERTY — объект типа Boolean, указывающий, использовать или нет поле SOAPAction заголовка SOAP-послания; по умолчанию это поле не используется;
- SOAPACTION_URI_PROPERTY — строка типа String с содержимым поля SOAPAction заголовка SOAP-послания.

После того как объект типа Call сформирован, можно обратиться к Web-службе и сразу получить Web-услугу одним из методов

```
public Remote invoke(Object[] args);
public Remote invoke(QName operationName, Object[] args);
public void invokeOneWay(Object[] args);
```

В листинге 6.15 приведен пример динамического получения Web-услуги от нашей Web-службы BookInfo.

Листинг 6.15. Клиент, использующий динамический вызов

```
import javax.xml.rpc.*;
import javax.xml.namespace.QName;

public class BookInfoDynamicClient{

   private static String BODY_NAMESPACE_VALUE =
           "http://bookinfo.com/wsdl/";

   private static String ENCODING_STYLE_PROPERTY =
          "javax.xml.rpc.encodingstyle.namespace.uri";

   private static String NS_XSD =
           "http://www.w3.org/2001/XMLSchema";
```

```
    private static String URI_ENCODING =
            "http://schemas.xmlsoap.org/soap/encoding/";

    public static void main(String[] args){

        try{
          ServiceFactory sf =
               ServiceFactory.newInstance() ;
          Service serv =
               sf.createService(newQName("BookInfo")) ;

          QName port = new QName("BookInfoPort");

          Call call = serv.createCall(port);

          call.setTargetEndpointAddress(
                "http://localhost:8080/services/bookinfo");

          call.setProperty(Call.SOAPACTION_USE_PROPERTY,
                           new Boolean(true));
          call.setProperty(Call.SOAPACTION_URI_PROPERTY," ") ;
          call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);

          QName xsdString = new QName(NS_XSD, "string");

          call.setReturnType(xsdString);

          call.setOperationName(new QName(BODY_NAMESPACE_VALUE,
                 "getInfo"));

          call.addParameter("String1", xsdString,
                 ParameterMode.IN);

          String[] params = { "5-94157-041-4" };

          String title = (String)call.invoke(params),•

          System.out.println(title);

        }catch(Exception e){
          System.err.println(e);
        }
    }
}
```

ГЛАВА 7

Web Services как часть J2EE

В предыдущих главах мы видели, что Web-службы легко и естественно создаются сервлетами и страницами JSP. Сервлеты и страницы JSP, носящие общее название *Web-компоненты,* работают под управлением специального программного модуля, называемого *Web-контейнером.* В предыдущих главах мы использовали самый популярный на сегодняшний день Web-контейнер Tomcat, созданный сообществом Apache Software Foundation. Он встраивается в Web-сервер как один из его модулей.

Чаще всего Web-сервер со своими дополнительными модулями является частью *сервера приложений* (application server), который, кроме того, ведет электронную почту, службу сообщений, распределенные каталоги, создает и передает распределенные объекты, соединяется с базами данных и предоставляет другие услуги. Сейчас наиболее известны серверы приложений BEA WebLogic, IBM WebSphere, JBoss, Sun ONE Application Server, Oracle9i Application Server, Borland Enterprise Server.

Сервер приложений, в свою очередь, расширяется компонентами EJB, работающими под управлением еще одного программного модуля — *EJB-контейнера.* Вся эта конструкция чаще всего управляется интерфейсами и классами, входящими в состав J2EE (Java 2 Enterprise Edition), и называется *J2EE-сервером.* Набор J2EE свободно распространяется фирмой Sun со своего сайта **http://java.sun.com/j2ee/.** Каждый производитель сервера приложений стремится сделать его совместимым с набором J2EE, чтобы Web-компоненты, компоненты EJB, Web-службы и другие серверные приложения выполнялись на нем без всяких переделок.

У Web-службы, развернутой на J2EE-сервере, появляется возможность использовать не только сервлеты и страницы JSP, но и компоненты EJB. Они описаны подробно в книгах [7, 10]. Напомним их структуру.

Компоненты EJB

Компоненты EJB, как следует из их названия, должны обладать той же особенностью, что и компоненты JavaBeans, а именно — легко и без всяких изменений встраиваться в создаваемые приложения. В случае компонентов EJB это условие означает — встраиваться в любой J2EE-сервер и работать под управлением любого EJB-контейнера. Для этого компоненты EJB должны строго соблюдать интерфейс с J2EE-сервером. Этот интерфейс — интерфейс EnterpriseBean — на удивление прост. Он не содержит ни констант, ни заголовков методов, а только расширяет интерфейс `Serializable`. Это означает, что каждый компонент EJB должен быть сериализуемым. На самом деле интерфейс EnterpriseBean — это только суперинтерфейс для интерфейсов, описывающих конкретные типы компонентов EJB. На сегодняшний день существует три типа компонентов EJB, описанных тремя расширениями интерфейса EnterpriseBean — интерфейсами SessionBean, EntityBean и MessageDrivenBean. Каждый компонент EJB должен реализовать один из этих интерфейсов.

Компоненты типа SessionBean предназначены для работы с клиентом EJB-приложения, которым может быть сервлет, страница JSP, отдельное приложение, другой компонент EJB. Мы будем называть их *session-компонентами*. Клиенты session-компонента могут быть *удаленными* (remote) или *локальными* (local). Локальные клиенты работают на той же самой виртуальной машине Java, что и компонент. Удаленные клиенты работают на другой виртуальной машине Java.

Есть две разновидности session-компонентов: компоненты, не сохраняющие свое текущее состояние для последующих вызовов клиента (Stateless Session Beans), и компоненты, сохраняющие свое состояние в оперативной памяти, но не на постоянном носителе информации (Stateful Session Beans). Первые удобны для создания Web-служб, поскольку Web-службы не создают сеанс связи с клиентом. Последние удобны для создания сеанса связи с клиентом и в Web-службах не применяются.

Компоненты типа EntityBean предназначены для хранения данных, извлеченных из постоянных хранилищ, в виде объектов Java. Мы будем называть их *entity-компонентами*. Клиенты entity-компонента могут изменить его содержимое. Entity-компоненты поддерживают связь с источником данных и сохраняют свое содержимое в постоянном хранилище при всяком изменении своего состояния. Они не применяются для создания Web-служб, а используются ими когда Web-службам надо получить информацию из базы данных или сохранить информацию в базе данных.

Компоненты типа MessageDrivenBean выполняют те же функции, что и session-компоненты без сохранения состояния, но действуют асинхронно.

Мы будем называть их *MDB-компонентами*. Они удобны для создания асинхронных Web-служб документного стиля.

Итак, для создания Web-служб процедурного стиля удобны session-компоненты без сохранения состояния, а для создания Web-служб документного стиля — MDB-компоненты. Рассмотрим подробнее устройство этих компонентов EJB.

Session-компоненты

Как ясно из сказанного выше, session-компонент — это класс или несколько классов, реализующих интерфейс SessionBean. Интерфейс SessionBean описывает всего четыре метода. Три метода предназначены для EJB-контейнера.

Метод

```
public void ejbActivate()
    throws EJBException, RemoteException
```

автоматически выполняется EJB-контейнером при каждом обращении клиента к компоненту, в него можно записать действия, активизирующие компонент, например, восстанавливающие его предыдущее состояние.

Метод

```
public void ejbPassivate()
    throws EJBException, RemoteException
```

автоматически выполняется EJB-контейнером, когда он решает, что пора сохранить состояние session-компонента.

Как видно из этого краткого описания, методы `ejbActivate()` и `ejbPassivate` о полезны только для session-компонентов с сохранением состояния. Для Web-служб делается пустая реализация этих методов.

Третий метод

```
public void ejbRemove()
    throws EJBException, RemoteException
```

выполняется EJB-контейнером при завершении работы с контейнером. В него можно записать завершающие действия компонента.

Четвертый метод

```
public void setSessionContext(SessionContext ctx)
    throws EJBException, RemoteException
```

позволяет session-компоненту получить контекст, в котором он работает, в виде объекта типа SessionContext.

Интересно то, что никакой клиент не может обратиться к session-компоненту напрямую. Вместо этого клиент обращается к EJB-контейнеру, который активизирует session-компонент и передает ему запрос клиента. Результаты выполнения методов компонента опять-таки передаются клиенту через EJB-контейнер. Контейнер взаимодействует с клиентом через специальные remote- и home-интерфейсы.

Remote-интерфейс session-компонента

Для взаимодействия с удаленным клиентом EJB-контейнер предоставляет ему интерфейс типа `EJBObject`. Разработчик Web-службы должен расширить этот интерфейс, описав в расширении Web-услуги реализуемой компонентом Web-службы. Разработчик не должен реализовать интерфейс каким-либо классом, реализацию делает EJB-контейнер. Соответствующий интерфейс локального клиента называется `EJBLocalObject`.

Интерфейс EJBObject расширяет интерфейс Remote, а значит, описывает распределенный объект. Будем называть интерфейс EJBObject и всякое его расширение *remote-интерфейсом* session-компонента. Web-услуги, описанные в remote-интерфейсе, должны следовать правилам распределенных методов — выбрасывать исключение класса RemoteException и не иметь аргументов и возвращаемых значений, которые сами являются распределенными объектами.

Интерфейс EJBObject описывает пять методов, которые реализуются самим EJB-контейнером. Разработчику Web-службы можно не беспокоиться о реализации этих методов, но он может воспользоваться ими для получения сведений о компоненте.

Метод

```
public EJBHome getEJBHome() throws RemoteException
```

возвращает объект типа EJBHome — home-интерфейс, о котором речь пойдет чуть ниже.

Метод

```
public Handle getHandle() throws RemoteException
```

возвращает идентификатор компонента типа Handle. Идентификатор хранит ссылку на объект типа EJBObject, которую можно получить методом `getEJBObject()`. Необходимость в этом методе возникает тогда, когда компоненту надо сослаться на себя, поскольку использовать ссылку this в компонентах EJB нельзя.

Метод

```
public boolean isIdentical(EJBObject obj)
    throws RemoteException
```

заменяет обычный метод сравнения объектов equals о, тоже не используемый в распределенных объектах, из-за того, что они выполняются на разных виртуальных машинах Java.

Метод

```
public void remove 0 throws RemoteException,
    RemoveException
```

удаляет компонент.

Последний, пятый, метод интерфейса `EJBObject`

```
public java.lang.Object getPrimaryKey()
    throws RemoteException
```

используется только в entity-компонентах для получения первичного ключа.

В remote-интерфейсе локального клиента `EJBLocalObject` четыре метода: отсутствует метод `getHandle ()`, а метод

```
public EJBLocalHome getEJBLocalHome() throws EJBException
```

возвращает ссылку на home-интерфейс локального клиента.

Home-интерфейс session-компонента

Еще один интерфейс — интерфейс типа EJBHome, тоже расширяющий интерфейс Remote — предназначен для создания экземпляров session-компонента по требованию клиента. Мы будем называть всякое расширение интерфейса EJBHome *home-интерфейсом.* Интерфейс EJBHome описывает четыре метода, реализуемые EJB-контейнером. Разработчик Web-службы не должен беспокоиться об этих методах — они реализуются EJB-контейнером — но может воспользоваться ими для получения сведений о компоненте.

Метод

```
public EJBMetaData getEJBMetaData()
    throws RemoteException
```

возвращает объект типа EJBMetaData, из которого клиент может извлечь некоторые сведения о компоненте.

Метод

```
public HomeHandle getHomeHandle()
    throws RemoteException
```

возвращает объект типа HomeHandle, содержащий всего один метод getEJBHome о . Этот метод используется вместо ссылки `this`.

Два метода

```
public void remove(Handle handle) throws RemoteException,
    RemoveException
```

```
public void remove(Object pk) throws RemoteException,
    RemoveException
```

удаляют компонент, заданный идентификатором или первичным ключом.

В home-интерфейсе локального клиента `EJBLocalHome` всего один метод

```
public void remove(java.lang.Object primaryKey)
    throws RemoveException, EJBException
```

Разработчик Web-службы должен расширить интерфейс EJBHome, и/или интерфейс EJBLocalHome, если у компонента будут локальные клиенты, записав в расширении метод с именем `create` о без аргументов.

Метод `create` () home-интерфейса удаленного клиента, как всякий метод распределенного объекта, должен выбрасывать исключение класса RemoteException. Кроме того, он должен выбрасывать исключение класса CreateException. Метод `create`о должен возвращать объект, тип которого совпадает с типом remote-интерфейса, поэтому он не описан в интерфейсах `EJBHome` и EJBLocalHome. Именно этим методом клиент дает указание EJB-контейнеру создать экземпляр session-компонента. Выполнив это указание, EJB-контейнер возвращает клиенту ссылку на remote-интерфейс созданного компонента. Клиент пользуется полученной ссылкой для обращения к Web-услугам, описанным в remote-интерфейсе.

Интерфейс адресата Web-службы

У session-компонента без сохранения состояния могут быть не только удаленные и локальные клиенты, но и клиенты Web-службы. Для них session-компонент предоставляет не remote- и home-интерфейсы, а *интерфейс адресата Web-службы* (Web Service Endpoint Interface). Будем называть его *WSEI-интерфейсом* session-компонента.

В отличие от remote- и home-интерфейсов, WSEI-интерфейс расширяет непосредственно интерфейс Remote, то есть является распределенным объектом. Он включается в систему JAX-RPC, описанную в *главе 6,* и должен следовать правилам SEI-интерфейсов этой системы. Это означает, что типы аргументов и возвращаемых значений его методов должны быть типами JAX-RPC, а сами методы должны выбрасывать исключение класса RemoteException. В частности, нельзя использовать объекты типов EJBObject и `EJBLocalObject`. Разработчику компонента не надо реализовать WSEI-интерфейс, это сделает EJB-контейнер.

Для каждого метода WSEI-интерфейса в классе session-компонента должен присутствовать соответствующий метод с тем же заголовком.

При установке компонента в EJB-контейнер, WSEI-интерфейс описывается в конфигурационном файле ear-jar.xml элементом <service-endpoint>, вложенном в элемент <session>, а также в WSDL-файле.

Приведем самый простой пример session-компонента, который только можно придумать. Клиент посылает session-компоненту свое имя name, используя для этого метод `sayHello()` session-компонента. Компонент отвечает: "Привет, <name>!", добавляя имя клиента. Назовем этот компонент `HelloEJB`. Пусть у него будут только удаленные клиенты и клиенты Web-службы, но не будет локальных клиентов.

Сначала напишем remote-интерфейс, описывающий метод `sayHello()`. Он приведен в листинге 7.1.

Листинг 7.1. Remote-интерфейс session-компонента

```
package hello;

import java.rmi.*;
import javax.ejb.*;

public interface Hello extends EJBObject{

   String sayHello(String name) throws RemoteException;
}
```

Потом записываем home-интерфейс, описывающий метод `create()`, это сделано в листинге 7.2.

Листинг 7.2. Home-интерфейс session-компонента

```
package hello;

import java.rmi.*;
import javax.ejb.*;

public interface HelloHome extends EJBHome{

   Hello create() throws CreateException, RemoteException;
}
```

Напишем и WSEI-интерфейс session-компонента. Он приведен в листинге 7.3.

Листинг 7.3. WSEI-интерфейс session-компонента

```
package hello;

import java.rmi.*;
```

```
public interface HelloPort extends Remote{

   String sayHello(String name) throws RemoteException;
}
```

После этого пишем сам класс session-компонента. Это сделано в листинге 7.4.

Листинг 7.4. Класс session-компонента

```
package hello;

import java.rmi.*;
import javax.ejb.*;

public class HelloBean implements SessionBean{

   private SessionContext ctx;

   public HelloBean(){}

   public String sayHello(String name) throws RemoteException{
      return "Привет, " + name + "!";
   }

   public void setSessionContext(SessionContext ctx){
      this.ctx = ctx;
   }

    public void ejbCreate(){}

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

В листинге 7.4 сделана пустая реализация трех методов интерфейса SessionBean и реализован метод `setSessionContext()`. Обратите внимание на то, что в классе компонента EJB обязательно должен быть открытый (public) конструктор по умолчанию, хотя бы пустой. Еще один обязательный метод ejbCreate() соответствует методу create() home-интерфейса, не

являясь его реализацией. В отличие от метода create о он не возвращает никакого значения и не выбрасывает исключений. В нем записываются начальные действия компонента, которые будут выполнены, когда клиент обращается К методу create () .

Обратите внимание на соглашение об именах, принятое для компонентов EJB. Создавая компонент HelloEJB, мы называем remote-интерфейс компонента именем Hello, home-интерфейс называем HelloHome, а класс компонента — HelloBean.

Компонент создан, теперь его надо установить (deploy) в EJB-контейнер. При установке или до нее создается конфигурационный XML-файл с именем ear-jar.xml. Установка выполняется какой-либо утилитой, входящей в состав EJB-контейнера. В набор J2EE входит графическая утилита deploytool, запускаемая из командной строки. Работа с этой утилитой подробно описана в книге [10]. В процессе установки компонент регистрируется в системе именования JNDI [10], обычно в контексте java:comp/env/ejb/. Для клиентов Web-служб рекомендуется создавать контекст java:comp/env/service/.

Пусть наш session-компонент HelioEJB зарегистрирован под JNDI-именем hello. Тогда удаленный клиент может обратиться к нему так, как показано в листинге 7.5.

Листинг 7.5. Удаленный клиент session-компонента

```
package hello;

import javax.naming.*;
import javax.rmi.*;

public class HelloClient{

   public static void main(String[] args){
     try{
       InitialContext ctx = new InitialContext();

       HelloHome home = (HelloHome)PortableRemoteObject.narrow(
         ctx.lookup("java:comp/env/ejb/hello"), HeiioHome.class);

       Hello h = home.create();
       System.out.println(h.sayHello("Иван"));
```

```
    }catch(Exception e){
      System.err.println(e);
    }
  }
}
```

Клиент Web-службы обращается к компоненту по обычным правилам системы JAX-RPC, изложенным в предыдущей *главе 6*.

Создавая session-компонент, EJB-контейнер сначала создает экземпляр класса компонента методом `Class.newInstance()` с помощью конструктора по умолчанию, который обязательно должен присутствовать в классе компонента. Затем EJB-контейнер создает объект типа sessionContext и устанавливает его в компонент методом `setSessionContext()`. После этого **EJB**-контейнер выполняет метод ejbCreate ().

Посмотрим, что может получить класс компонента от объекта `SessionContext`.

Контекст session-компонента

Интерфейс `SessionContext` предоставляет ссылки на remote-интерфейсы типа EJBObject и `EJBLocalObject` методами

```
public EJBObject getEJBObject();
public EJBLocalObject getEJBLocalObject();
```

Версия **EJB** 2.1 добавила еще один метод, доступный только для адресатов Web-служб:

```
public MessageContext getMessageContext();
```

Этот метод предоставляет объект типа MessageContext, содержащий свойства сообщения. Он описан в *главе 6*.

Кроме того, интерфейс SessionContext расширяет интерфейс EJBContext И, следовательно, обладает его методами

```
public EJBHome getEJBHome();
public EJBLocalHome getEJBLocalHome();
public UserTrasaction getUserTransaction();
public boolean getRollbackOnly();
public Principal getCallerPrincipal();
```

предоставляющими ссылки на home-интерфейсы, сведения о транзакции, в которой участвует компонент, и сведения о клиенте компонента.

Все эти сведения можно использовать в классе компонента.

MDB-компоненты

Компоненты типа MDB (Message Driven Beans) работают асинхронно под управлением EJB-контейнера. Контейнер получает сообщение от клиента, точнее говоря, от службы сообщений, через которую действует клиент, активизирует MDB-компонент и обращается к его методам. Клиент никак не связан с MDB-компонентом, более того, клиент не подозревает о его существовании. Клиент обращается только к службе сообщений. Поэтому для MDB-компонента не нужны ни remote- ни home-интерфейсы, он состоит только из одного или нескольких классов. Класс MDB-компонента должен реализовать интерфейс MessageDrivenBean.

Интерфейс MessageDrivenBean описывает всего два метода

```
public void setMessageDrivenContext(MessageDrivenContext ctx)
    throws EJBException
public void ejbRemove() throws EJBException
```

реализуемые EJB-контейнером.

Интерфейс MessageDrivenContext, расширяющий интерфейс EJBContext, ничего не добавляет к методам интерфейса EJBContext. Поэтому класс MDB-компонента может получить только сведения о транзакции, если он участвует в какой-либо транзакции. Более того, он не может воспользоваться методами интерфейса EJBContext, предоставляющими сведения о клиенте и ссылки на remote- и home-интерфейсы.

EJB-контейнер создает экземпляр MDB-компонента так же, как экземпляр session-компонента, поэтому в нем должен быть открытый конструктор по умолчанию и метод `ejbCreate()` без аргументов. В этом методе можно записать начальные действия компонента.

Главное, что должен сделать класс MDB-компонента — это реализовать интерфейс-слушатель службы сообщений. Если MDB-компонент связан со службой сообщений JMS (Java Message Service), то он должен реализовать интерфейс MessageListener. Если MDB-компонент связан с пакетом JAXM, то он реализует интерфейс OnewayListener или интерфейс `RegRespListener`. Об этих интерфейсах говорилось в предыдущей *главе 6*. Каждый из этих интерфейсов описывает по одному методу `onMessage()`, в котором и заключена основная работа MDB-компонента.

Правила реализации интерфейса MessageListener описаны в книгах [7, 10]. Заголовок его единственного метода имеет следующий вид:

```
public void onMessage(Message msg);
```

Аргумент msg, передаваемый этому методу контейнером, содержит полученное сообщение в виде объекта типа Message. Интерфейс Message содержит

множество методов доступа к различным заголовкам и частям сообщения системы JMS. Он подробно описан в книге [10].

Приведем в листинге 7.6 простейший пример MDB-компонента, отправляющего сведения о полученных сообщениях в стандартный вывод.

Листинг 7.6. Простейший MDB-компонент

```
import  javax.ejb.*;
import javax.jms.*;
import  java.util.*;

public class DummyMDB
    implements MessageDrivenBean, MessageListener{

   public void onMessage(Message msg){
      try{
         System.out.println("\nПолучено сообщение " +
            msg.getJMSMessageID() +
          " для адресата " + msg.getJMSDestination());

         System.out.println("\nСрок хранения  " +
            (msg.getJMSExpiration() > 0 ?
                 new Date(msg.getJMSExpiration()) :
                "не ограничен."));

         System.out.println("\nОтвет послать: " +
             msg.getJMSReplyTo());

         System.out.println("\nТип сообщения " +
             msg.getJMSType());

        if  (msg  instanceof  TextMessage)
           System.out.println("\nТело сообщения:\n" +
               ((TextMessage)msg).getText());

      }catch(Exception e){
         System.err.println(e);
      }
   }
```

```
    public void ejbCreate(){}
    public void ejbRemove(){}
    public void setMessageDrivenContext(
        MessageDrivenContext mdc){}
}
```

Конфигурационный файл EJB-приложения

В процессе установки компонентов EJB в контейнер сведения о компонентах заносятся в конфигурационный файл с именем ear-jar.xml, хранящийся в каталоге META-INF архива EJB-приложения. Для каждого EJB-приложения создается только один конфигурационный файл. Именно этот файл связывает класс session-компонента с относящимися к нему remote-, home- и WSEI-интерфейсами. В этом файле MDB-компонент привязывается к определенной службе сообщений. Здесь указываются ссылки на другие компоненты EJB и определяются переменные окружения.

Структура файла ear-jar.xml понятна без всяких объяснений. В листинге 7.7 показан конфигурационный файл, содержащий сведения о компонентах HelloEJB и DummyMDB, созданных в предыдущих листингах.

Листинг 7.7. Конфигурационный файл ear-jar.xml EJB-приложения

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC
    '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
    'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>

  <display-name>Simple EJB</display-name>
  <enterprise-beans>

    <session>

      <display-name>HelloEJB</display-name>
      <ejb-name>HelloEJB</ejb-name>
      <home>HelloHome</home>
      <remote>Hello</remote>
      <service-endpoint>HelloPort</service-endpoint>
```

```
      <ejb-class>HelloBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <security-identity>
        <descriptionx/description>
        <use-caller-identity></use-caller-identity>
      </security-identity>

    </session>

    <message-driven>

      <display-name>DummyMDB</display-name>
      <ejb-name>DummyMDB</ejb-name>
      <ejb-class>DummyMDB</ejb-class>
      <transaction-type>Bean</transaction-type>
      <message-selector></message-selector>
      <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
        <subscription-durability>Durable</subscription-durability>
      </message-driven-destination>
      <security-identity>
        <description></description>
        <run-as>
          <description></description>
          <role-name></role-name>
        </run-as>
      </security-identity>

    </message-driven>

  </enterprise-beans>

  <assembly-descriptor>
    <method-permission>
      <unchecked />
```

```
<method>
  <ejb-name>HelloEJB</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>remove</method-name>
  <method-params>
    <method-param>java.lang.Object</method-param>
  </method-params>
</method>

<method>
  <ejb-name>HelloEJB</ejb-name>
  <method-intf>Remote</method-intf>
  <method-name>getHandle</method-name>
  <method-params />
</method>

<method>
  <ejb-name>HelloEJB</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>remove</method-name>
  <method-params>
    <method-param>javax.ejb.Handle</method-param>
  </method-params>
</method>

<method>
  <ejb-name>HelloEJB</ejb-name>
  <method-intf>Remote</method-intf>
  <method-name>sayHello</method-name>
  <method-params>
    <method-param>java.lang.String</method-param>
  </method-params>
</method>

<method>
  <ejb-name>HelloEJB</ejb-name>
  <method-intf>Home</method-intf>
```

```
    <method-name>getHomeHandle</method-name>
    <method-params />
  </method>

  <method>
    <ejb-name>HelloEJB</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getPrimaryKey</method-name>
    <method-params />
  </method>

  <method>
    <ejb-name>HelloEJB</ejb-name>
    <method-intf>Home</method-intf>
    <method-name>getEJBMetaData</method-name>
    <method-params />
  </method>

  <method>
    <ejb-name>HelloEJB</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>remove</method-name>
    <method-params />
  </method>

  <method>
    <ejb-name>HelloEJB</ejb-name>
    <method-intf>Home</method-intf>
    <method-name>create</method-name>
    <method-params />
  </method>

  <method>
    <ejb-name>HelloEJB</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>isIdentical</method-name>
    <method-params>
```

```
        <method-param>javax.ejb.EJBObject</method-param>
      </method-params>
    </method>

    <method>
      <ejb-name>HelloEJB</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getEJBHome</method-name>
      <method-params />
    </method>

  </method-permission>
 </assembly-descriptor>
</ejb-jar>
```

Размещение Web-служб на J2EE-сервере

У Web-службы, работающей в составе J2EE-сервера, есть возможность использовать компоненты EJB. Реализовать эту возможность можно самыми разными способами. Можно отвести компонентам EJB роль только вспомогательных классов, разгружающих сервлеты и страницы JSP. Схема такой реализации показана на рис. 7.1.

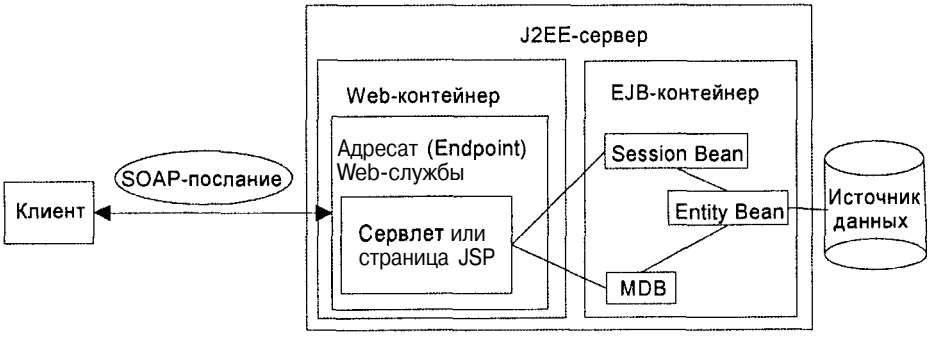


Рис 7.1. Компоненты EJB в составе Web-службы

Другой способ использования компонентов EJB — реализовать ими саму Web-службу, отказавшись от сервлетов, как показано на рис. 7.2.

Между этими двумя крайними схемами лежит множество промежуточных решений, использующих компоненты EJB в качестве адресата Web-службы наряду с сервлетами и страницами JSP.

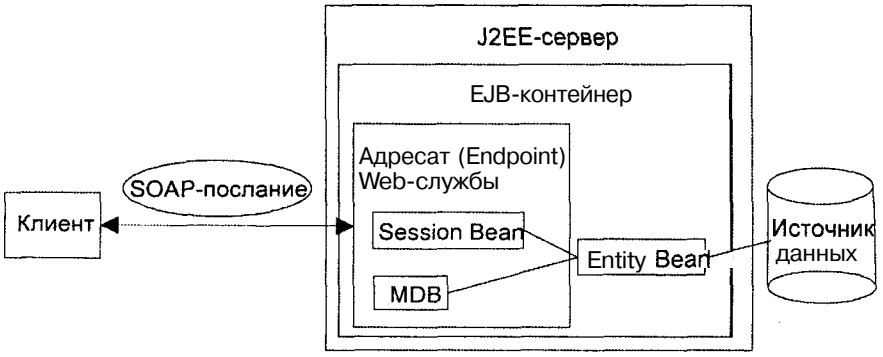


Рис 7.2. Реализация Web-службы компонентами EJB

Реализуя Web-службу на J2EE-сервере по той или иной схеме, следует помнить о возможности ее переноса на другой сервер. Ценность созданной вами Web-службы резко возрастет, если она, будучи упакована в EAR-архив, установится на любом J2EE-сервере без всяких изменений. Кроме того, следует обеспечить одинаковый доступ клиентам, независимый от способа реализации Web-службы на J2EE-сервере.

Для того чтобы обеспечить выполнение этих двух условий, группой ведущих компаний под патронажем фирмы IBM разработана спецификация "Web Services for J2EE". Она доступна по адресу **http://www-106.ibm.com/developerworks/webservices/**. Мы будем называть ее сокращенно спецификацией "WS4EE".

Спецификация WS4EE предлагает свою обобщенную схему реализации Web-службы на J2EE-сервере, основанную на средствах пакета JAX-RPC. Эта схема показана на рис. 7.3. По этой схеме Web-служба помещается в контейнер, которым может служить Web- или EJB-контейнер. Контейнер создает и регистрирует в системе именования JNDI объект, реализующий интерфейс `Service` из пакета JAX-RPC. Этот объект содержит адрес Web-службы и служит фабрикой классов, реализующих SEI-интерфейс Web-службы. Клиент находит его средствами JNDI, создает с его помощью объект SEI-интерфейса и через него получает Web-услуги.

Такая схема дает одинаковый доступ к Web-службам, построенным самыми разными сочетаниями сервлетов, страниц JSP и компонентов EJB.

Саму Web-службу, называемую в спецификации WS4EE *портом* (port), предлагается реализовать либо классом-служителем (tie) системы JAX-RPC, следовательно, с помощью сервлета `JAXRPCServlet`, либо session-компонентом без сохранения состояния. В первом случае Web-служба будет работать под управлением Web-контейнера, во втором — под управлением EJB-контейнера.

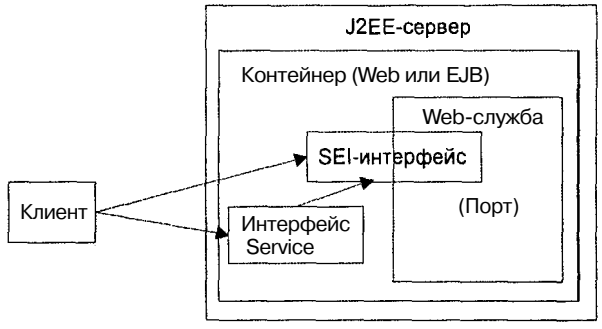


Рис. 7.3. Схема, предлагаемая спецификацией "WS for J2EE"

Название "порт" идет от описания WSDL (см. *главу 4).* Порт включает в себя и SEI-интерфейс, соответствующий WSDL-элементу `<wsdl:portType>`, и класс-служитель, чьи методы описаны элементом `<wsdl:binding>`, и вспомогательные классы.

У каждого порта есть свой адрес URI, но два порта с разными адресами могут разделять одни и те же классы Web-службы. Адрес порта хранится в объекте-фабрике типа service, который прямо соответствует элементу <wsdl:service>. Такое соответствие не случайно. Спецификация WS4EE исходит из описания WSDL и строит Web-службу как воплощение такого описания.

Реализация порта

Поскольку описание WSDL ничего не говорит о способах реализации порта, спецификация WS4EE предлагает для этого две модели.

- Модель системы JAX-RPC, описанная в *главе 6.* Она основана на SEI-интерфейсах, классах-служителях и сервлетах и работает в Web-контейнере.
- Модель компонентов EJB, основанная на session-компонентах без сохранения состояния, предоставляющих WSEI-интерфейс. Она работает в EJB-контейнере.

Спецификация WS4EE ничего не говорит о других моделях, например, модели, основанной на службе сообщений, системе JAXM и MDB-компонентах. Однако она не ограничивает реализацию Web-службы указанными в ней двумя моделями. Разработчик может выбрать и другую модель.

Обе модели, описанные в спецификации WS4EE, предполагают наличие в составе порта двух обязательных компонентов.

- Первый компонент — это интерфейс, предоставляемый клиенту, который спецификация называет *SEI-интерфейсом* так же как в системе JAX-

RPC. Этот интерфейс реализует WSDL-описания порта, сделанные элементами <wsdl:portType> И <wsdl:binding>.

- Второй компонент — это класс, воплощающий методы SEI-интерфейса. Его адрес указан элементом <wsdl:port> WSDL-описания. Он может быть классом-служителем системы JAX-RPC, session-компонентом или каким-то другим классом. Спецификация WS4EE называет этот класс "Service Implementation Bean". Мы будем называть его *порт-компонентом*. Этот класс обязан реализовать все методы SEI-интерфейса, хотя не обязан реализовать сам SEI-интерфейс.

У каждого порта есть свой адрес URI, указанный в элементе <wsdl:port>. Несколько портов с разными адресами могут разделять один порт-компонент, но они считаются разными портами Web-службы.

Поскольку WSDL-файла недостаточно для точного описания реализации порта, а конфигурационные файлы Web-приложения web.xml и EJB-приложения ear-jar.xml описывают только отдельные сервлеты и отдельные компоненты EJB, необходим дополнительный конфигурационный файл порта. Спецификация WS4EE предлагает написать целых три конфигурационных файла.

- Первый файл с именем webservices.xml описывает Web-службу.
- Второй файл с именем webservicesclient.xml содержит ссылки на другие Web-службы. Он предназначен клиенту Web-службы.
- Третий файл, имя которого не фиксировано, описывает соответствие типов Java и WSDL по правилам системы JAX-RPC. Будем называть его условно *JAX-RPC-файлом*.

Рассмотрим подробнее каждый из трех конфигурационных файлов.

Конфигурационный файл Web-службы

Для каждого Web- или EJB-контейнера записывается один конфигурационный файл, описывающий все Web-службы, установленные в этом контейнере. Это XML-файл webservices.xml с корневым элементом <webservices>. В Web-контейнере этот файл хранится вместе с файлом web.xml в каталоге WEB-INF, в EJB-контейнере — вместе с файлом ear-jar.xml в каталоге META-INF.

В корневой элемент <webservices> вкладывается один или несколько элементов <webservice-description>, описывающих каждую Web-службу. При описании в элементе <webservice-description-name> обязательно задается имя Web-службы, применяемое внутри описания, в элементе <wsdl-file> — записывается полный путь к WSDL-файлу, начинающийся от корневого

каталога Web- или EJB-модуля. Элементом <jaxrpc-mapping-file> указывается полный путь к JAX-RPC-файлу.

Один или несколько элементов <port-component>, вложенных в элемент <webservice-description>, описывают каждый порт-компонент Web-службы. При описании порт-компоненту дается имя в элементе <port-component-name>. Оно не связано с именем порта в WSDL-файле, и должно быть уникально в пределах модуля. В элементе <wsdl-port> двумя вложенными элементами <namespaceURI> и <localpart> записывается префикс и имя (QName) XML-элемента, описывающего порт в WSDL-файле. Элемент <service-endpoint-interface> показывает полное имя SEI-интерфейса, а элемент <service-impl-bean> — полное имя порт-компонента.

Имя порт-компонента указывается в одном из двух элементов, вложенных в элемент <service-impl-bean>. Имя сервлета записывается во вложенном элементе <servlet-link>. Оно должно соответствовать имени, записанном в элементе <servlet-class> файла web.xml. Имя session-компонента записывается во вложенном элементе <ejb-link> и должно соответствовать логическому имени компонента, записанному в элементе <ejb-name> файла ear-jar.xml.

Кроме этого, в необязательных элементах <handler> следует описать классы-обработчики системы JAX-RPC, если они есть в составе Web-службы, указав при этом их логическое имя, полное имя класса, начальные параметры и блоки заголовка SOAP-послания, которые они обрабатывают.

В листинге 7.8 приведена полная схема файла webservices.xml. Значки в квадратных скобках не входят в файл, они, как и в предыдущих главах, отмечают обязательность и повторяемость элемента. Атрибут id во всех элементах необязателен, он применяется для ссылки на элемент из другого файла.

Листинг 7.8. Структура конфигурационного файла Web-службы

```
<webservices id="">

   <description id=""> [?]
   <display-name id=""> [?]
   <small-icon id=""> [?]
   <large-icon id=""> [?]

   <webservlce-description> [+]

       <description id=""> [?]
```

```
<display-name id=""> [?]
<small-icon id=""> [?]
<large-icon id=""> [?]

<webservice-description-name id="">
<wsdl-file id="">
<jaxrpc-mapping-file id="">

<port-component id=""> [+]

    <description id=""> [?]
    <display-name id=""> [?]
    <small-icon id=""> [?]
    <large-icon id=""> [?]

    <port-component-name id="">

    <wsdl-port id="">
            <namespaceURI id="">
            <localpart id="">
    </wsdl-port>

    <service-endpoint-interface id="">

    <service-impl-bean id="">
        <ejb-link id="">  |  <servlet-link id="">
    </service-impl-bean>

    <handler id="">*
        <description id="">  [?]
        <display-name id="">  [?]
        <small-icon id="">  [?]
        <large-icon id="">  [?]

        <handler-name id="">
        <handler-class id="">

        <init-param id="">  [*]
```

```
                <param-name id="">
                <param-value id="">
                <description id=""> [?]
            </init-param>

            <soap-header id=""> [*]
                <namespaceURI id="">
                <localpart id="">
            </soap-header>

            <soap-role id=""> [*]

        </handler>

    </port-component>

</webservice-description>

</webservices>
```

В листинге 7.9 приведен конфигурационный файл Web-службы "HelloService", построенной в листингах 7.3, 7.4, 7.7.

Листинг 7.9. Конфигурационный файл Web-службы "HelloService"

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE webservices PUBLIC
  "-//IBM Corporation, Inc.//DTD J2EE Web services 1.0//EN"
  "http://www.ibm.com/webservices/dtd/j2ee_web_services_1_0.dtd">

<webservices>
  <webservice-description>

    <webservice-description-name>
        HelloService
    </webservice-description-name>

    <wsdl-file>META-INF/hello.wsdl</wsdl-file>
```

```
    <jaxrpc-mapping-file>
        META-INF/HelloMapping.xml
    </jaxrpc-mapping-file>

    <port-component>

        <port-component-name>
            HelloPortComponent
        </port-component-name>

        <wsdl-port>
            <namespaceURI></namespaceURI>
            <localpart>HelloPortComponent</localpart>
        </wsdl-port>

        <service-endpoint-interface>
            hello.HelloPort
        </service-endpoint-interface>

        <service-impl-bean>
            <ejb-link>HelloEJB</ejb-link>
        </service-impl-bean>

    </port-component>

  </webservice-description>
</webservices>
```

Конфигурационный файл клиента

Второй конфигурационный файл с именем webservicesclient.xml — это XML-файл с корневым элементом <webservicesclient>. В него заносятся ссылки на другие Web-службы, чьими услугами пользуется данная Web-служба. В этой ситуации данная Web-служба является клиентом тех Web-служб, на которые делается ссылка. Отсюда происходит название файла и корневого элемента.

Каждая ссылка описывается внутри элемента <service-ref>. Если данная ссылка должна находиться в области действия компонента, описанного в файле ear-jar.xml или в файле web.xml, то элемент <service-ref> помещает-

ся внутрь элемента <component-scoped-refs>, в котором элементом <component-name> отмечается имя компонента.

Все остальные элементы вложены в элемент <service-ref>.

В элементе <service-ref-name> дается имя ссылке. Это имя используется клиентом при поиске Web-службы методом lookup () в системе именования JNDI. Рекомендуется начинать имя со строки "service/".

В элементе <service-interface> указывается полное имя фабрики классов, обычно это интерфейс javax.xml.rpc.Service или его расширение.

Остальные элементы, вложенные в элемент <service-ref>, необязательны.

Элементы <wsdl-file> и <jaxrpc-mapping-file> указывают расположение WSDL-файла и JAX-RPC-файла в модуле.

Элемент <service-qname> содержит расширенное имя (QName) XML-элемента WSDL-файла, описывающего Web-службу. Он записывается только в том случае, когда в файле есть элемент <wsdl-file>, а соответствующий WSDL-файл содержит несколько элементов <wsdl:service>.

Элемент <port-component-ref> связывает полное имя SEI-интерфейса с именем порт-компонента, определенным в элементе <port-component-name> файла webservices.xml. Эта связка используется затем контейнером при обращении клиента к методу getPort о для получения заглушки.

Наконец, элементы <handler> описывают классы-обработчики SOAP-послания. Порт, к которому приписан обработчик, указывается вложенным элементом <port-name>. Если порт не указан, то обработчик приписывается ко всем портам.

В листинге 7.10 приведена полная схема конфигурационного файла webservicesclient.xml.

Листинг 7.10. Структура конфигурационного файла клиента

```
<webservicesclient id="">

   <component-scoped-refs id=""> [+] | <service-ref id=""> [+]

      <component-name id="">

      <service-ref id=""> [+]

           <description id=""> [?]
           <display-name id=""> [?]
```

```
<small-icon id=""> [?]
<large-icon id=""> [?]

<service-ref-name id="">
<service-interface id="">
<wsdl-file id="">?
<jaxrpc-mapping-file id=""> [?]

<service-qname id=""> [?]
     <namespaceURI id="">
     <localpart id="">
</service-qname>

<port-component-ref id=""> [*]
    <service-endpoint-interface id="">
    <port-component-link id=""> [?]
</port-component-ref>

<handler id=""> [*]

    <description id=""> [?]
    <display-name id=""> [?]
    <small-icon id=""> [?]
    <large-icon id=""> [?]

    <handler-name id="">
    <handler-class id="">

    <init-param id=""> [*]
        <param-name id="">
        <param-value id="">
        <description id=""> [?]
    </init-param>

    <soap-header id=""> [*]
        <namespaceURI id="">
        <localpart id="">
    </soap-header>
```

```
            <soap-role id=""> [*]
            <port-name id=""> [*]

        </handler>

    </service-ref>

  </component-scoped-refs>

</webservicesclient>
```

В листинге 7.11 приведен пример простого конфигурационного файла клиента Web-службы "HelloService", описанной в конфигурационном файле листинга 7.9.

Листинг 7.11. Конфигурационный файл клиента Web-службы "HelloService"

```
<?xmlversion="1.0"encoding="UTF-8"?>

<!DOCTYPE webservicesclient PUBLIC
  "-//IBM Corporation, Inc.//DTD J2EE Web services client 1.0//EN"
  "http://www.ibm.com/webservices/dtd/j2ee_web_services_client_1_0.dtd">

<webservicesclient>

  <service-ref>

    <service-ref-name>service/HelloService</service-ref-name>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-file>META-INF/hello.wsdl</wsdl-file>

    <jaxrpc-mapping-file>
          META-INF/HelloMapping.xml
    </jaxrpc-mapping-file>

    <port-component-ref>

        <service-endpoint-interface>
            hello.HelloPort
        </service-endpoint-interface>
```

```
        <port-component-link>
            HelloPortComponent
        </port-component-link>

    </port-component-ref>

  </service-ref>

</webservicesclient>
```

Конфигурационный файл JAX-RPC

У третьего конфигурационного файла, который мы назвали JAX-RPC-файлом, нет определенного имени, обычно его имя складывается из имени порта и слова "Mapping". Этот файл хранится в одном каталоге с WSDL-файлом, то есть, в каталоге WEB-INF или в каталоге META-INF.

В самом простом случае, когда WSDL-файл описывает только один элемент <wsdl:service> с одним вложенным элементом <wsdl:port>, в каждый элемент <wsdl:operation> вложен только один элемент <wsdl:input> и не больше одного элемента <wsdl:output>, используются только типы WSDL и значения по умолчанию и так далее, в этом случае в корневом элементе <java-wsdl-mapping> обязательны ТОЛЬКО элементы <package-mapping>.

Элемент <package-mapping> связывает пакет Java, указанный во вложенном элементе <package-type>, с идентификатором пространства имен, определенным в WSDL-файле. Идентификатор указывается во вложенном элементе <namespaceURI>.

В остальных случаях в корневой элемент <java-wsdl-mapping> надо обязательно вложить по одному элементу <java-xml-type-mapping> для каждого типа, введенного в WSDL-файле, по одному элементу <exception-mapping> для каждого WSDL-элемента <wsdl:fault> и по одному элементу <service-interface-mapping> — для каждого нестандартного WSDL-элемента <wsdl:service>. Кроме ТОГО, надо ВЛОЖИТЬ ПО ОДНОМУ Элементу <service-endpoint-interface-mapping> для каждой комбинации WSDL-элементов <wsdl:portType> И <wsdl:binding>.

В листинге 7.12 приведена полная схема конфигурационного JAX-RPC-файла.

Листинг 7.12. Структура конфигурационного файла Java-WSDL-Mapping

```
<java-wsdl-mapping id="">

  <package-mapping id=""> [+]
```

```
    <package-type id="">
    <namespaceURI id="">
</package-mapping>

<java-xml-type-mapping id="">  [*]
    <class-type id="">
    <root-type-qname id="">
        <namespaceURI id="">
        <localpart id="">
    </root-type-qname>
    <qname-scope id="">
    <variable-mapping id="">  [*]
        <java-variable-name id="">
        <data-member id="">  [?]
       <xml-element-name id="">
    </variable-mapping>
</java-xml-type-mapping>

<exception-mapping id="">  [*]
    <exception-type id="">
    <wsdl-message id="">
        <namespaceURI id="">
        <localpart>
    </wsdl-message>
    <constructor-parameter-order>  [?]
        <element-name>  [+]
    </constructor-parameter-order>
</exception-mapping>

<service-interface-mapping id="">  [?]
     <service-interface id="">
     <wsdl-service-name id="">
     <port-mapping id="">  [*]
         <port-name id="">
         <java-port-name id="">
     </port-mapping>
</service-interface-mapping>

<service-endpoint-interface-mapping id="">  [+]
    <service-endpoint-interface id="">
```

```
    <wsdl-port-type id="">
        <namespaceURI id="">
        <localpart id="">
    </wsdl-port-type>

    <wsdl-binding id="">
        <namespaceURI id="">
        <localpart>
    </wsdl-binding>

    <service-endpoint-method-mapping id="">  [*]
        <java-method-name id="">
        <wsdl-operation id="">
        <method-param-parts-mapping id="">  [*]
           <param-position id="">
           <param-type id="">
           <wsdl-message-mapping id="">
               <wsdl-message id="">
               <wsdl-message-part-name id="">
               <parameter-mode id="">
               <soap-header id="">  [?]
           </wsdl-message-mapping>
        </method-param-parts-mapping>
        <wsdl-return-value-mapping id="">
             <method-return-value id="">
             <wsdl-message id="">
             <wsdl-message-part-name id="">
        </wsdl-return-value-mapping>
    </service-endpoint-method-mapping>
  </service-endpoint-interface-mapping>

</java-wsdl-mapping>
```

В листинге 7.13 приведен JAX-RPC-файл Web-службы "HelloService". Его имя HelloMapping.xml.

Листинг 7.13. Конфигурационный JAX-RPC-файл Web-службы "helloService"

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE java-wsdl-mapping PUBLIC
```

```
  "-//IBM Corporation, Inc.//DTD J2EE JAX-RPC mapping 1.0//EN"
  "http://www.ibm.com/webservices/dtd/j2ee_jaxrpc_mapping_1_0.dtd">

<java-wsdl-mapping>

   <package-mapping>

      <package-type>hello</package-type>
      <namespaceURI>myservices.hello</namespaceURI>

   </package-mapping>

</java-wsdl-mapping>
```

Установка Web-службы в контейнер

Поскольку по спецификации "Web Services for J2EE" Web-служба реализуется с помощью сервлетов или session-компонентов, ее надо установить (deploy) в Web- или EJB-контейнер как всякий компонент J2EE-приложения. В процессе установки создаются клиентские заглушки и серверные связки системы RMI, каталоги для их размещения, пути к ним и переменные окружения, составляющие контекст приложения. Правила установки описаны, например, в книге [10]. Производители J2EE-серверов предоставляют утилиты установки, облегчающие этот процесс. В стандартную поставку пакета J2EE SDK входит графическая утилита `deploytool`, которая открывает последовательно несколько диалоговых окон, собирая сведения об устанавливаемом компоненте, и затем устанавливает компонент в контейнер.

Утилита установки Web-службы, реализованной по правилам спецификации WS4EE, должна учитывать, что Web-служба работает под управлением механизма JAX-RPC, а не механизма RMI. Кроме того, в процессе установки надо создать или использовать несколько конфигурационных файлов: WSDL-файл, конфигурационный файл компонента web.xml или ear-jar.xml, конфигурационный файл Web-службы webservices.xml, конфигурационный файл клиента webservicesclient.xml и JAX-RPC-файл. Поэтому для Web-службы надо использовать специализированные средства установки.

Фирма IBM поставляет эталонную реализацию (RI — Reference Implementation) спецификации "Web Services for J2EE" со своего сайта **http://www-106.ibm.com/developerworks/webservices/. В** нее входят классы, полностью реализующие интерфейсы спецификации, вспомогательные классы и утилиты. Классы эталонной реализации расширяют пакет классов

J2EE SDK, образующих J2EE-сервер и используются Web-службами, установленными на этом сервере.

Для построения Web-службы эталонная реализация предлагает утилиту svcgen (Web Service Generator), работающую из командной строки. Ее можно использовать двумя способами: или для создания Web-службы по уже откомпилированным интерфейсам и классам, или для создания Web-службы по ее описанию WSDL.

В первом случае командная строка выглядит так:

```
$ svcgen [ -classpath <path> ] -sei <classname> \
    -map <namespace> <pkg> -wsdl <WSDLfile>
```

В этой строке необязательный параметр -classpath указывает путь к классам Web-службы. Параметр -sei содержит краткое имя SEI-интерфейса без расширения .class. Параметр -map связывает идентификатор целевого пространства имен создаваемого утилитой WSDL-файла с пакетом классов Java. Параметр -wsdl содержит имя создаваемого утилитой WSDL-файла.

Во втором случае командная строка выглядит иначе:

```
$ svcgen -wsdl <WSDLfile> -map <namespace> <pkg> [ -dir <path> ]
```

Здесь параметр -wsdl содержит имя уже существующего WSDL-файла. Параметр -map определяет имя создаваемого пакета Java. Необязательный параметр -dir указывает имя каталога, в который будут помещены создаваемые утилитой файлы Java и JAX-RPC-файл, имя которого повторяет имя WSDL-файла с окончанием ".deployment.xml". Если этот параметр опущен, то файлы будут помещены в каталог $WS4EE_HOME/temp/svcgen/. Созданные файлы надо просмотреть, добавить в них фактический код Web-службы и откомпилировать.

После создания Web-службы ее надо упаковать в архив application.ear обычными средствами J2EE-сервера, например, утилитой deploytool. При упаковке надо включить в архив файл webservices.xml, а для клиента файл webservicesclient.xml. Это можно сделать впоследствии утилитой wsdd.

Установка созданной и упакованной Web-службы выполняется утилитой wsdeploy. Ее командная строка выглядит так:

```
$ wsdepioy <input.ear> [ <output.jar> ]
```

Первый параметр <input.ear> —исходный архив — должен находиться в текущем каталоге. Утилита создает файл с именем Enabled<input.ear>, полностью готовый для утилиты deploytooi. Если указан второй параметр <output.jar>, то утилита создаст архив, содержащий все необходимое для установки клиента Web-службы. Затем утилита wsdepioy вызывает утилиту deploytooi, которая заканчивает процесс установки.

ГЛАВА 8

Безопасность предоставления услуг

Сфера применения Web-служб стремительно расширяется. В настоящее время в нее вовлекаются банковские транзакции, финансовые операции, медицинские консультации — те Web-услуги, которые требуют секретности, точности, достоверности. Это заставляет обращать особенное внимание на безопасность предоставления услуг.

Вопросы безопасности обсуждаются с тех самых пор, как люди стали предоставлять и получать услуги. При заключении всякой сделки важно убедиться в том, что она будет совершена точно, в срок, с соблюдением всех оговоренных условий. Применительно к Web-услугам можно выделить пять основных проблем безопасности.

- *Конфиденциальность* (confidentiality) данных. Сообщения, пересылаемые между Web-службой и ее клиентами, не должны читаться никем, кроме отправителя и получателя. Конфиденциальность достигается шифрованием сообщений.
- *Целостность* (integrity) данных. Клиент и Web-служба должны быть уверены, что данные во время пересылки не изменились посторонними лицами или аппаратными средствами. Уверенность в целостности сообщения обеспечивается цифровой подписью или добавлением дайджеста сообщения MD (Message Digest).
- *Аутентификация* (authentication) партнера или сообщения. И клиент, и Web-служба должны быть уверены, что имеют дело с тем партнером, за кого он себя выдает. Для этого сообщение подписывается цифровой подписью, известной другой стороне, или к сообщению прилагается цифровой сертификат.

- *Авторизация* (authorization) клиента. Клиент может получить только те услуги, которые он имеет право получать. Это достигается разными способами: назначением ролей, составлением списков доступа ACL (Access Control Lists), выбором политики доступа.
- *Неаннулируемость* (nonrepudiation) услуг. И клиент, и Web-служба должны быть уверены, что другая сторона строго выполнит взятые на себя обязательства. Этого можно достигнуть при помощи цифровой подписи или NR-маркера.

Как видно из этого перечня, за многие века обмена услугами выработаны и методы решения проблем безопасности. Давно существует и бурно развивается целая наука — криптография. Приведем краткое описание ее основных понятий.

Криптография

Криптография занимается разработкой методов обработки данных с целью сокрытия их смысла. Для этого одним из множества алгоритмов шифрования исходное сообщение преобразуется в бессмысленный набор битов, который можно быстро преобразовать обратно в первоначальный текст, только зная *ключ* (key) шифрования. Секретность ключа обеспечивает конфиденциальность сообщения.

Различают два вида ключей: *симметричные* (закрытые) ключи, и *асимметричные* (открытый и закрытый) ключи.

Симметричные ключи

Шифрование и расшифровка сообщения производятся с помощью одного и того же ключа, поэтому он и называется симметричным ключом. Алгоритмы шифрования с симметричным ключом работают быстро и надежно. Их надежность напрямую зависит от длины ключа — чем длиннее ключ, тем больше времени требует дешифровка сообщения. *(Дешифровкой* называется попытка чтения зашифрованного сообщения без знания ключа.) Наиболее распространены следующие алгоритмы, применяющие симметричные ключи.

- Широко распространен алгоритм *DES* (Data Encryption Standard) с 56-разрядным ключом. Это стандарт для защиты от несанкционированного доступа к служебной документации в государственных учреждениях США.
- В России для тех же целей применяется *ГОСТ28147–89.* Он использует 256-разрядный ключ.

- Алгоритм DES часто применяется к сообщению трижды с разными ключами. Эта схема шифрования называется *Triple-DES или 3DES.*
- Международный алгоритм шифрования данных *IDEA* (International Data Encryption Algorithm) использует 128-разрядный ключ.
- Алгоритмы Брюса Шнайера (Bruce Schneier) *Blowfish,* с 56-разрядным ключом, и *Twofish,* в котором длина ключа достигает 256 разрядов, не запатентованы и свободно доступны в Интернете по адресу **http://www.counterpane.com/labs.html**.
- Широко распространена целая серия алгоритмов Рональда Ривеста (Ronald Rivest) *RC2, RC4, RC5, RC6.* Они различаются скоростью шифрования и длиной секретного ключа.

Вся проблема заключается в передаче секретного ключа. Обеспечить ее конфиденциальность не менее важно и не менее трудно, чем обеспечить конфиденциальность самого сообщения. Очень часто эту проблему решают применением асимметричных ключей, при помощи которых передается секретный симметричный ключ.

Асимметричные ключи

В этом случае у каждого участника обмена сообщениями — Web-службы и ее клиента — есть по два ключа — *открытый* (public key), известный всем, и *закрытый* (private key), известный только самому участнику. Открытый ключ предназначен для шифрования сообщения, закрытый — для его расшифровки. Клиент шифрует свое сообщение открытым ключом Web-службы и посылает шифровку вместе со своим открытым ключом. Зашифрованное сообщение может прочитать только Web-служба, которой оно предназначено. Web-служба расшифровывает его своим закрытым ключом, читает, составляет ответ, и шифрует ответное сообщение открытым ключом клиента. Зашифрованный ответ может прочитать только клиент, расшифровав его своим закрытым ключом.

Итак, при использовании асимметричных ключей вместе с зашифрованным сообщением или до него посылается открытый ключ отправителя.

Разработано множество алгоритмов шифрования с асимметричными ключами.

- Алгоритм *DSA* (Digital Signature Algorithm) позволяет использовать ключи любого размера. Алгоритм стандартизирован, стандарт *DSS* (Digital Signature Standard) рекомендует ключи длиной от 512 до 1024 двоичных разрядов. Алгоритм применяется, главным образом, для создания цифровой подписи.
- Алгоритм *RSA,* названный по первым буквам фамилий авторов (Rivest, Shamir, Adleman), использует ключи длиной до 1024 битов.

- Алгоритм Диффи-Хелмана *DH* (Diffie-Hellman) работает с ключами длиной до 4096 битов.
- Алгоритм *AlGamal* Тегера Эль-Гамала (Taher ElGamal) не запатентован, он используется свободно.

В реальных криптографических системах симметричные и асимметричные ключи часто комбинируются. Например, известно, что шифрование открытым ключом занимает больше времени, чем шифрование секретным ключом. Для ускорения работы поступают так: сообщение шифруют симметричным ключом, затем этот симметричный ключ шифруют открытым ключом получателя и отправляют вместе с зашифрованным сообщением. Получатель сначала расшифровывает симметричный ключ своим закрытым ключом, а затем расшифровывает сообщение уже известным ему симметричным ключом.

Дайджест сообщения

Иногда в результате шифрования получают короткий *дайджест сообщения* MD (Message Digest), который посылают вместе с незашифрованным сообщением. Это делается не с целью достижения конфиденциальности, а для сохранения целостности послания. Получатель повторно шифрует сообщение тем же алгоритмом и сравнивает полученный дайджест с тем, который был прислан вместе с сообщением. Совпадение дайджестов гарантирует неизменность сообщения по пути следования. Своеобразие дайджеста заключается в том, что его не надо расшифровывать. Это позволяет применять для получения дайджеста мощные односторонние алгоритмы.

- В настоящее время наибольшее распространение получил алгоритм Рональда Ривеста *MD5,* описанный в рекомендации RFC 1321. Он создает 128-разрядный дайджест.
- Алгоритм *SHA-1* (Secure Hash Algorithm) использует хеш-функцию для получения 160-разрядного дайджеста. У алгоритма есть расширения *SHA-256, SHA-384, SHA-512* для получения более длинных дайджестов.
- В алгоритмах под общим названием *MAC* (Message Authentication Code) кроме самого алгоритма используется и секретный ключ. Алгоритм используется главным образом для создания цифровой подписи.

Цифровая подпись

Для решения задач аутентификации и неаннулируемости, сообщение часто снабжается цифровой подписью. В самом простом случае для получения цифровой подписи применяются асимметричные ключи обратным спосо-

бом — автор сообщения шифрует его своим закрытым, а не открытым ключом. После этого всякий желающий может расшифровать сообщение открытым ключом автора и прочитать его. Поскольку закрытый ключ знает только автор, зашифровать сообщение мог только он. Этим подтверждается авторство сообщения.

Зашифровать сообщение можно и симметричным ключом, но потом его придется открыть для проверки авторства. Такой ключ будет одноразовым, для следующего сообщения придется генерировать другой ключ.

В других случаях в качестве цифровой подписи используется дайджест сообщения, зашифрованный с помощью закрытого ключа автора. Адресат может получить тот же дайджест, используя открытый ключ автора сообщения, сравнить его с дайджестом самого сообщения и убедиться в авторстве сообщения.

Цифровой сертификат

Широкое распространение открытых ключей поставило задачу целостности и аутентификации самих открытых ключей. Отправитель, шифрующий сообщение открытым ключом адресата, должен быть уверен, что ключ действительно принадлежит адресату, а не подменен злоумышленником, выступающим от имени адресата. Для этого открытые ключи заверяются цифровой подписью, но подписью не адресата, а доверенного лица — центра сертификации CA (Certificate Authority).

Кроме открытого ключа и цифровой подписи, сертификат содержит сведения о центре сертификации и пользователе ключа, время действия ключа, назначение ключа: шифрование, цифровая подпись, дайджест. Структура сертификата и правила его использования определены международным стандартом ISO X.509.

Центры сертификации имеются во многих странах. Наиболее авторитетна организация VeriSign, **http://www.verisign.com/**, у которой есть сеть филиалов на нескольких сайтах Интернета.

Реализация криптографии в Java

В стандартную поставку Java 2 SDK, Standard Edition входят средства Security API, реализующие многие криптографические алгоритмы. В Security API входит пакет JCA (Java Cryptography Architecture) и дополнение к нему — пакет JCE (Java Cryptography Extension). Эти пакеты составлены из пакетов Java `java.security`, `javax.crypto` и их подпакетов, а также пакета `com.sun.security` и его подпакетов, содержащих средства генерации ключей, шифрования, создания дайджестов и цифровых подписей. Кроме того,

в Security API входит пакет `javax.security.auth` и его подпакеты, содержащие средства аутентификации и авторизации и составляющие, вместе с другими пакетами Java, пакет JAAS (Java Authentication and Authorization Service).

Классы, составляющие средства Security API, реализуют криптографические алгоритмы DES, IDEA, DSA, MD5, SHA-1, RSA и другие, создают сертификаты по протоколу X.509, связывают приложения Java с системой безопасности Kerberos.

Мы не будем здесь рассматривать Security API. Средства этого пакета подробно описаны в книге [13] и множестве других книг, посвященных безопасности Java. Текущая версия Security API полностью описана в документации к вашему набору J2SE SDK, в разделе $JAVA_HOME/docs/guide/security/.

Безопасность на транспортном уровне

Сообщения Web-службы передаются по компьютерным сетям. Поэтому, казалось бы, для безопасности их пересылки достаточно применить средства сетевой безопасности. Одно из таких средств — *протокол защищенных соединений* SSL (Secure Sockets Layer), запатентованный в США фирмой Netscape Communication. Протокол работает поверх стека TCP/IP и основан на шифровании содержимого сетевых IP-пакетов с помощью симметричных или асимметричных ключей. Он описывает правила установления соединения, открытия защищенных сокетов, обмена ключами, шифрования и передачи данных. С ним тесно связан основанный на версии SSL 3.0 открытый протокол TLS (Transport Layer Secure), первая версия которого определена в рекомендации RFC 2246. Протокол TLS очень похож на SSL и призван заменить его, но эти протоколы несовместимы.

Протоколы SSL и TLS могут применять разные алгоритмы шифрования, в том числе DES, RSA, SHA-1, MD5, RC2, RC4. Это позволяет найти общий алгоритм для клиента и сервера и практически всегда установить защищенное соединение. При этом алгоритмы шифрования с асимметричными ключами применяются при установлении связи и передаче сгенерированного секретного ключа, а алгоритмы с симметричными ключами — для пересылки сообщений, зашифрованных полученным секретным ключом.

Технология Java реализовала протоколы SSL 3.0 и TLS в пакете JSSE (Java Secure Socket Extension). Соответствующие пакеты Java `javax.net.ssl`, `javax.security.cert` входят в стандартную поставку J2SE SDK. Они подробно описаны в книге [13] и в документации J2SE SDK.

Средства SSL и TLS удобны и хорошо обеспечивают конфиденциальность передаваемых данных, но у них нет собственных средств аутентификации. Они должны получать сертификаты из какого-нибудь центра сертификации.

Другое сетевое средство безопасности — *виртуальная частная сеть* VPN (Virtual Private Network) — не требует сертификатов. Оно передает сообщения, зашифрованные на выходе из локальной сети маршрутизаторами или брандмауэрами. Аутентификация сообщений и клиентов производится маршрутизаторами или брандмауэрами на входе в локальную сеть. Виртуальные частные сети требуют установки программного обеспечения на каждый сервер, маршрутизатор, брандмауэр и на каждую клиентскую машину, что вызывает обычные проблемы совместимости версий и необходимости их обновления на каждой машине.

Виртуальная частная сеть создается, чаще всего, на основе стандартов сетевого уровня IPSec, IPv6 или на основе протокола PPTP (Point-to-Point Tunneling Protocol).

Средства сетевой безопасности транспортного уровня могут использоваться при организации защищенных Web-служб, но сильно ограничивают их применение. Во-первых, эти средства привязывают Web-службу к определенному протоколу: HTTPS, PPTP, SMTP, хотя основное достоинство Web-службы — ее переносимость и независимость от протоколов и платформы. Во-вторых, эти средства шифруют все пересылаемое сообщение целиком, а SOAP-послания по пути следования могут обрабатываться промежуточными SOAP-узлами (actors). При этом каждый промежуточный сервер должен расшифровать послание, обработать его и снова зашифровать для дальнейшей пересылки. Ясно, что после этого ни о какой секретности говорить не приходится.

Безопасность на уровне XML

Для того чтобы зашифровать не всё SOAP-послание, а только его отдельные части, придется перейти на уровень языка XML и пойти привычным путем — ввести дополнительные элементы XML, описывающие зашифрованную часть послания.

Разработка средств описания зашифрованных XML-документов и их отдельных частей ведется консорциумом W3C давно и независимо от протокола SOAP и Web-служб, причем этим занимается сразу несколько рабочих групп.

- Рабочая группа, определяющая правила описания шифрования документов XML, выпустила предварительную версию рекомендации "XML Encryption Syntax and Processing". Будем называть ее короче "XML Encryption". Ее можно посмотреть по адресу **http://www.w3.org/TR/2002/xmlenc-core/**.
- Рабочая группа, разрабатывающая правила работы с цифровыми подписями на языке XML, ее сайт **http://www.w3.org/TR/xmldsig-core/**, предложила две рекомендации: RFC 3275 описывает синтаксис цифровой

подписи, а RFC 3076 — так называемую *каноническую форму XML* (Canonical XML).

- Третья рабочая группа рассматривает работу с цифровыми сертификатами и разрабатывает правила обмена ключами шифрования. Она разработала спецификацию XKMS (XML Key Management Specification) и работает над ее второй версией. Ее можно посмотреть на сайте **http://www.w3.org/TR/xlms2/**.

Рассмотрим подробнее эти рекомендации и спецификации.

Шифрование документов XML

Документы XML или их отдельные элементы шифруются с использованием обычных алгоритмов криптографии. Задача состоит в том, чтобы выделить зашифрованную часть документа и описать способ ее шифрования.

Рекомендация "XML Encryption" вводит новый XML-элемент `<EncryptedData>` и вложенные в него элементы. Все элементы определены в пространстве имен с идентификатором http://www.w3.org/2001/04/xmlenc#. Они описывают метод шифрования, сам зашифрованный элемент и тому подобную информацию. При помощи элемента <EncryptedData> можно зашифровать группу вложенных XML-элементов, отдельный элемент или только его содержимое.

Допустим, что в некотором XML-документе есть элементы

```
<person>
   <numb>XY-123456-27</numb>
</person>
```

и нам надо зашифровать элемент `<numb>`, содержащий секретный номер абонента. Это можно сделать двумя способами.

Первый способ — зашифровать только содержимое элемента:

```
<person>
   <numb>
      <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Content">
         <!— вложенные элементы с зашифрованным номером —>
      </EncryptedData>
   </numb>
</person>
```

Второй способ — зашифровать весь элемент `<numb>`:

```
<person>
   <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element">
```

```
        <!— вложенные элементы с зашифрованным элементом —>
    </EncryptedData>
</person>
```

Обратите внимание на то, как изменился атрибут `Type`, — вместо заключительного указания "content" атрибут содержит слово "Element".

Вообще же у элемента <EncryptedData> четыре необязательных атрибута:

- атрибут `Id` содержит идентификатор элемента <EncryptedData>, полезный для последующих ссылок на элемент;
- атрибут type содержит тип зашифрованной информации в виде строки URI;
- атрибут `MimeType` указывает MIME-тип зашифрованной информации обычной строкой или строкой URI;
- атрибут Encoding указывает кодировку содержимого в виде строки URI.

Остальные атрибуты описывают идентификаторы пространств имен; очень часто они выглядят так:

```
<EncryptedData
    xmlns="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
      "http://www.w3.org/2001/04/xmlenc#xenc-schema.xsd">
```

В элемент <EncryptedData> вкладываются четыре элемента.

- **Элемент** <EncryptionMethod Algorithm=""> **описывает** алгоритм шифрования своим обязательным атрибутом Algorithm. В этот элемент можно вложить элемент <KeySize>, содержащий длину ключа. Можно вложить и другие элементы, например, для записи параметров алгоритма RSA-OAEP специально предназначен вложенный элемент `<OAEPparams>`. Элемент `<EncryptionMethod>` можно опустить, если алгоритм шифрования был оговорен заранее.
- Элемент `<ds:KeyInfo>` описывает применяемые ключи шифрования. Как видно из его префикса, он определен в пространстве имен с идентификатором http://www.w3.org/2000/09/xmldsig#. Это пространство имен цифровой подписи, оно описано в следующем разделе. Дополнительно в этот элемент можно вложить элемент <EncryptedKey>, описывающий метод шифрования ключа, если ключ шифруется, и элемент `<AgreementMethod>`, описывающий метод обмена ключами.
- Элемент <cipherData> описывает сами зашифрованные данные.

- Элемент <EncryptionProperties> содержит дополнительную информацию во вложенных в него элементах <EncryptionProperty>.

Из этих элементов обязателен только элемент <cipherData>. Именно в нем, во вложенном в него элементе <CipherValue> содержится зашифрованная информация. Это двоичная информация, но она записывается в кодировке Base64.

Таким образом, минимальный состав элемента <EncryptedData> примерно таков:

```
<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#">
   <CipherData>
      <CipherValue>
         BNjivf7gTOXRRhdgB5h4JSxHJ7dlZudnZBrg=
      </CipherValue>
   </CipherData>
</EncryptedData>
```

Вместо элемента <CipherValue> в элемент <CipherData> можно вложить элемент <cipherReference URI="">, указывающий расположение зашифрованных данных своим атрибутом URI — строкой URI. В элемент <CipherReference> можно вложить элемент <Transforms>, а в него — один или несколько элементов <ds:Transform>, описывающих методы дополнительного преобразования извлеченной по ссылке зашифрованной информации. Например, извлеченную зашифрованную информацию можно переслать в кодировке Base64, для этого надо записать такое преобразование:

```
<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#">
   <CipherReference URI="http://www.some.com/CipherData.xml">
      <Transforms>
         <ds:Transform Algorithm=
              "http://www.w3.org/2000/09/xmldsig#base64"/>
      </Transforms>
```

```
    </CipherReference>

</EncryptedData>
```

Структура элемента <EncryptedKey>, описывающего зашифрованный ключ, повторяет структуру элемента <EncryptedData> с добавлением элемента <ReferenceList>, вложенные В который элементы <KeyReference URI=""> и/или элементы <DataReference URI=""> своим атрибутом URI указывают идентификаторы (значения атрибутов Id) элементов, использующих описываемый зашифрованный ключ.

Элемент <AgreementMethod> описывает способ генерации ключей и обмена ими. Сторона, генерирующая ключ, описывается вложенным элементом <OriginatorKeyInfo>, а сторона, получающая ключ — вложенным элементом <RecipientKeyInfo>. Факт пересылки ключа отмечается вложенным элементом <KA-Nonce>, чтобы не сгенерировать дважды один и тот же ключ.

Полная схема вложенности элементов показана в листинге 8.1.

Листинг 8.1. Вложенность элементов шифрования

```
<EncryptedData Id=""[?] Type=""[?] MimeType=""[?] Encoding=""[?]
  xmlns="http://www.w3.org/2001/04/xmlenc#"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/04/xmlenc#xenc-schema.xsd">

  <EncryptionMethod Algorithm=""> [?]
      <KeySize> [?]
      <OAEPparams> [?]
      <ds:DigestMethod Algorithm=""> [?]
  </EncryptionMethod>

  <ds:KeyInfo> [?]
    <ds:KeyName> (?]
    <ds:RetrievalMethod URI="" Type=""> [?]

    <EncryptedKey> [?]

      <EncryptionMethod Algorithm=""> [?]

      <CipherData>
```

```
        <CipherReference URI=""> | <CipherValue>
           <Transforms>
               <ds:Transform Algorithm="">
           </Transforms>
        </CipherReference>

    </CipherData>

    <ReferenceList> [?]
        <KeyReference URI=""> [*]
        <DataReference URI=""> [*]
    </ReferenceList>

  </EncryptedKey>

   <AgreementMethod Algorithm=""> [?]

     <KA-Nonce> [?]
     <ds:DigestMethod Algorithm="">

     <OriginatorKeyInfo> [?]
     <RecipientKeyInfo> [?]

   </AgreementMethod>

</ds:KeyInfo>

<CipherData>

   <CipherReference URI=""> | <CipherValue>
       <Transforms>
           <ds:Transform Algorithm="">
       </Transforms>
   </CipherReference>

</CipherData>

<EncryptionProperties> [?]
```

```
    <EncryptionProperty> [+]

  </EncryptionProperties>

</EncryptedData>
```

Цифровая подпись документа XML

Правила создания цифровой подписи документа **XML** определены в рекомендации RFC 3275. Суть рекомендации в том, что создаются дайджесты подписываемой информации, они вкладываются в элемент `<SignedInfo>` вместе с нужными пояснениями. Затем создается дайджест этого элемента, который подписывается цифровой подписью. Элемент `<SignedInfo>` определен в спецификации RFC 3275.

Все это вкладывается в корневой элемент цифровой подписи <signature>, тоже определенный в рекомендации RFC 3275. Непосредственно в элемент <signature> вкладывается два обязательных элемента `<SignedInfo>` и `<SignatureValue>`, а также необязательный элемент `<KeyInfo>` и нуль или несколько элементов <object>. Все элементы определены в пространстве имен с идентификатором http://www.w3.org/2000/09/xmldsig#.

Элемент `<SignedInfo>`, как следует из его названия, описывает способ получения цифровой подписи. Это выполняется двумя обязательными вложенными элементами <CanonicalizationMethod> И `<SignatureMethod>`. Кроме того, элемент `<SignedInfo>` описывает подписываемую информацию, это делается третьим обязательным вложенным элементом `<Reference>`.

Элемент `<CanonicalizationMethod>` указывает способ приведения элемента `<SignedInfo>` к каноническому виду XML. Способ указывается обязательным атрибутом Algorithm. В этот элемент можно вложить произвольные элементы, характерные для данного метода приведения к каноническому виду.

Элемент `<SignatureMethod>` указывает алгоритм шифрования элемента <SignedInfo>. Алгоритм указывается обязательным атрибутом Algorithm, например:

```
<SignatureMethod
    Algorithm=http://www.w3.org/2000/09/xmldsig#dsa-sha1 />
```

Здесь указан комбинированный алгоритм шифрования **DSA-SHA1.**

В элемент `<SignatureMethod>` можно вложить произвольные элементы, характерные для выбранного алгоритма шифрования.

Элемент <Reference> может появиться несколько раз, поскольку подписываемая информация может состоять из нескольких документов XML, документов HTML или их частей, файлов с графическими изображениями, звуковых файлов и других ресурсов. Каждый ресурс, которым может быть и часть документа, описывается своим элементом <Reference> и указывается атрибутом URI этого элемента, например:

```
<Reference URI="http://www.some.com/pub/inv.xml#ident">
```

В этом примере создается дайджест части внешнего документа inv.xml. Полученная цифровая подпись (detached signature) хранится отдельно от подписанного документа.

Цифровая подпись может быть вложена в сам подписываемый документ (enveloped signature). В этом случае ссылка будет выглядеть так:

```
<Reference URI="#ident">
```

Для каждого ресурса создается свой дайджест.

Каждый элемент <Reference> содержит необязательный вложенный элемент <Transforms>, задающий вложенными элементами <Transform> методы предварительного преобразования подписываемого ресурса, например, преобразование к каноническому виду XML. Кроме того, элемент <Reference> должен содержать два обязательных элемента: элемент <DigestMethod> указывает алгоритм получения дайджеста обязательным атрибутом Algorithm, а элемент <Digestvalue> содержит дайджест в кодировке Base64. Например:

```
<Reference URI="file:/D:/sign.xml">
   <Transforms>
      <Transform Algorithm=
            "http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
   </Transforms>
<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
<DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
```

Здесь для получения дайджеста файла sign.xml выбран алгоритм SHA-1. Файл sign.xml предварительно приводится к каноническому виду XML.

Элемент <SignatureValue> содержит цифровую подпись в кодировке Base64.

Элемент <KeyInfo> описывает информацию об открытом ключе, с помощью которого проверяется подпись. Ключом может быть и цифровой сертификат. Элемент <KeyInfo> может отсутствовать, если получателю известен ключ. Информация содержится во вложенных элементах, некоторые элементы определены спецификацией. Это элементы <KeyName>, <KeyValue>, <RetrievalMethod>, <X509Data>, <PGPData>, <SPKIData>, <MgmtData>. Авторы цифровых подписей могут добавить другие элементы.

Элементы <object> определяют дополнительные свойства цифровой подписи, например, дату и время подписания, элементами <SignatureProperty>, <Manifest> и другими элементами.

Листинг 8.2 показывает схему вложенности элементов, описывающих цифровую подпись.

Листинг 8.2. Вложенность элементов цифровой подписи

```
<Signature Id="" [?]>

   <SignedInfo>

      <CanonicalizationMethod Algorithm="">

      <SignatureMethod Algorithm="">

      <Reference URI="" [?] > [+]

         <Transforms> [?]
            <Transform Algorithm=""> [+]
         </Transforms>

         <DigestMethod Algorithm="">

         <DigestValue>

      </Reference>

   </SignedInfo>

   <SignatureValue>

   <KeyInfo> [?]

      (<KeyName> | <KeyValue> | <RetrievalMethod> | <X509Data> |
      <PGPData> | <SPKIData> | <MgmtData>) [+]

   </KeyInfo>

   <Object Id="" [?]> [*]
```

```
    <SignatureProperties>
        <SignatureProperty Target=""> [+]
    </SignatureProperties>

    <Manifest>
        <Reference> [+]
    </Manifest>

  </Object>

</Signature>
```

Канонический вид XML

При создании цифровой подписи имеет значение каждый символ подписываемого документа. Даже разное количество пробелов между словами приведет к разным цифровым подписям. Для того чтобы цифровая подпись зависела только от содержания документа, но не от его оформления, документ сначала приводится к каноническому виду, а потом подписывается. Метод приведения к каноническому виду указывается атрибутом элемента <CanonicalizationMethod>.

Канонический вид документа XML и правила приведения к нему описаны в рекомендации **RFC** 3076, называемой "Canonical XML". По рекомендации **RFC** 3076 документ записывается в кодировке UTF-8, все ссылки разрешаются, пробельные символы нормализуются, заголовки **XML** удаляются, пустые элементы записываются парой из открывающего и закрывающего тега, записываются значения по умолчанию всех опущенных атрибутов и так далее. Если приведение к каноническому виду выполнено по этой рекомендации, то элемент <CanonicalizationMethod> должен выглядеть так:

```
<CanonicalizationMethod
    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
```

Средства Java для шифрования XML

Фирма IBM выпустила пакет интерфейсов и классов XSS4J (**XML** Security Suite for Java), реализующих спецификацию "XML Encryption", рекомендации RFC 3275, **RFC** 3076, методы авторизации и получения сертификатов, методы приведения к каноническому виду. Пакет XSS4J можно свободно скопировать, он доступен по адресу

http://www-106.ibm.com/developerworks/webservices/library/xmlsecuritysuite/.

Для применения классов и интерфейсов пакета XSS4J достаточно добавить полученный при копировании архив xss4j.jar в библиотеку классов J2SE SDK, например, в каталог $JAVA_HOME/jre/lib/ext/, или добавить путь к архиву в переменную CLASSPATH. Пакет XSS4J требует для своей работы наличия стандартного Java Security API, входящего в состав J2SE SDK. Кроме того, нужен XML-анализатор, например, Apache Xerces2, и XSLT-процессор, например, Apache Xalan-J. Пакет XSS4J входит в состав не раз упоминавшегося в этой книге набора IBM WSTK и в состав интегрированной среды разработки IBM WebSphere Studio.

Сообщество Apache Software Foundation выпустило пакет "Apache XML Security", доступный по адресу **http://xml.apache.org/security/**. В него входят средства шифрования, реализующие спецификацию "XML Encryption", и средства создания цифровой подписи по рекомендациям RFC 3275 и RFC 3076. Пакет можно использовать отдельно или в составе других инструментальных средств XML. Он входит, например, в набор средств создания Web-служб Apache Axis, рассмотренный нами в *главе 3*.

В листинге 8.3 показан пример клиента Web-службы, создающего SOAP-послание, подписывающего его цифровой подписью, отправляющего подписанное послание Web-службе, и ожидающего от нее ответа.

Перед тем как запустить клиентскую программу следует сгенерировать ключи и сертификат и записать их в стандартное хранилище ключей Java — файл .keystore. Это можно сделать утилитой keytool из набора J2SE SDK, набрав следующую командную строку:

```
keytool -genkey
```

и ответив на вопросы утилиты.

Листинг 8.3. Клиент Web-службы, отправляющий подписанное SOAP-послание

```
import org.apache.axis.*;
import org.apache.axis.client.*;
import org.apache.axis.message.*;
import org.apache.axis.utils.*;
import org.apache.axis.configuration.NullProvider;
import org.apache.axis.encoding.*;

import org.apache.xml.security.c14n.Canonicalizer;
import org.apache.xml.security.signature.XMLSignature;
import org.w3c.dom.*;
import org.xml.sax.InputSource;

import java.io.*;
```

```
import java.security.*;

import java.security.cert.X509Certificate;

public class ClientAxis{

   static SOAPEnvelope env = new SOAPEnvelope();

   public static void main(String[] args){

     if (args.length != 3){

       System.err.println("Usage: java ClientAxis" +
           " <keystorePassword> <privateKeyPassword>" +
           " <distinguishedName>");
     } System.exit(0);

      try{
         Options opts = new Options(args);

         Service service = new Service ();
         Call call = (Call)service.createCall();
         call.setTargetEndpointAddress(
               newjava.net.URL(opts.getURL()));

         SOAPBodyElement sbe = new SOAPBodyElement(
            XMLUtils.StringToElement(
              "http://localhost:8080/EchoService",
              "getName", ""));
         env.addBodyElement(sbe);

         sign(args[0], args[1], args[2]);

         System.out.println("\nRequest:");

         XMLUtils.PrettyElementToStream(env.getAsDOM(), System.out);

         call.invoke(env);

         MessageContext mc = call.getMessageContext();
```

```
        System.out.println("\nResponse:");
        XMLUtils.PrettyElementToStream(
            mc.getResponseMessage().getSOAPEnvelope().getAsDOM(),
            System.out);

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

private static void sign(String keystorePassword,
            String privateKeyPassword,
            String certificateAlias){
    try{
        env.addMapping(new Mapping(
            "http://schemas.xmlsoap.org/soap/security/2000-12",
            "SOAP-SEC"));

        env.addAttribute(Constants.URI_SOAP11_ENV,
                "actor", "some-uri");
        env.addAttribute(Constants.URI_SOAP11_ENV,
                "mustUnderstand", "1");

        SOAPHeaderElement header = new SOAPHeaderElement(
            XMLUtils.StringToElement(
                "http://schemas.xmlsoap.org/soap/security/2000-12",
                "Signature", ""));

        env.addHeader(header);

        Document doc = soap2dom(env);

        KeyStore ks = KeyStore.getInstance("JKS");
        FileInputStream fis = new FileInputStream(".keystore");

        ks.load(fis, keystorePassword.toCharArray());

        PrivateKey privateKey = (PrivateKey)ks.getKey("mykey",
```

```
            privateKeyPassword.toCharArray());

        Element soapHeaderElement =
             (Element)((Element)doc.getFirstChild()).
               getElementsByTagNameNS("*", "Header").item(0);

        Element soapSignatureElement =
             (Element)soapHeaderElement.
               getElementsByTagNameNS("*", "Signature").item(0);

        XMLSignature sig = new XMLSignature(doc,
               "http://xml-security",
               XMLSignature.ALGO_ID_SIGNATURE_DSA);

        soapSignatureElement.appendChild(sig.getElement());
        sig.addDocument("#Body");


        X509Certificate cert =
          (X509Certificate)ks.getCertificate(certificateAlias);


        sig.addKeyInfo(cert);
        sig.addKeyInfo(cert.getPublicKey());
        sig.sign(privateKey);

        Canonicalizer c14n = Canonicalizer.getInstance(

               Canonicalizer.ALGO_ID_C14N_WITH_COMMENTS);

        byte[] canonicalMessage = c14n.canonicalizeSubtree(doc);

        InputSource is = new InputSource(

           new java.io.ByteArrayInputStream(canonicalMessage));

        DeserializationContextImpl dser = null;

        AxisClient tmpEngine =
              new AxisClient(new NullProvider());
```

```
            MessageContext msgContext =
                  new MessageContext(tmpEngine);

            dser = new DeserializationContextImpl(is,
                        msgContext, Message.REQUEST);

            dser.parse();

        }catch(Exceptione) {
            e.printStackTrace();
            throw new RuntimeException(e.toString());
        }
    }

    private static Document soap2dom(SOAPEnvelope env)
            throws Exception{

        StringWriter writer = new StringWriter();

        SerializationContextImpl ctx =
                  new SerializationContextImpl(writer);

        env.output(ctx);
        writer.close();

        Reader reader = new StringReader(writer.getBuffer().toString());
        Document doc = XMLUtils.newDocument(new InputSource(reader));
        if (doc == null) throw new Exception();
        return doc;
    }
}
```

Существует еще много коммерческих и свободно распространяемых средств шифрования документов XML. Каждый месяц появляются новые средства, их списки можно просмотреть на сайтах Интернета, посвященных XML.

Безопасность SOAP-посланий

Поскольку SOAP-послание представляет собой документ XML, к нему применимы все средства обеспечения безопасности, перечисленные в преды-

дущих разделах этой главы. Ничто не мешает зашифровать все послание целиком или его отдельные части, например, тело послания, или отдельные элементы XML, входящие в послание. SOAP-послание можно снабдить цифровой подписью как обычный документ XML.

Тем не менее, SOAP-послания следует составлять так, чтобы их могла прочитать и обработать любая Web-служба, имеющая право на это. Поэтому необходимы правила шифрования и подписи SOAP-посланий, единые для всех Web-служб. Такие правила разрабатываются как расширения протокола SOAP.

Первое расширение протокола SOAP, посвященное вопросам безопасности, было сделано фирмой IBM еще в 2000 году. Документ под названием "SOAP Security Extensions" опубликован в Интернете по адресу **http://www.trl.ibm.com/projects/xml/soap/wp/wp.html**. Он посвящен шифрованию, цифровой подписи и авторизации и вводит три соответствующих элемента XML <Encryption>, <Signature> И <Authorization> В пространстве имен с идентификатором http://schemas.xmlsoap.org/soap/security/. Эти элементы записываются как блоки заголовка <Header> SOAP-послания.

Элемент <Encription> содержит вложенные элементы <DecryptionInfo>, определенные в устаревшей версии спецификации "XML Encryption" с идентификатором пространства имен http://www.w3.org/2000/10/xmlenc. Поэтому этот элемент вряд ли будет использоваться в современных SOAP-посланиях.

Элемент <signature> содержит вложенный элемент <ds:Signature>, определенный в спецификации "XML Signature" с идентификатором пространства имен http://www.w3.org/2000/02/xmldsig#. Он не изменился со времени создания расширения "SOAP Security Extensions" и вполне может применяться в SOAP-посланиях.

У третьего элемента <Authorization> нет четкой структуры, в него можно вложить произвольные элементы, идентифицирующие автора послания.

Второе расширение протокола SOAP, сделанное фирмами IBM и Microsoft в феврале 2001 года, посвящено созданию цифровой подписи. Оно называется "SOAP Security Extensions: Digital Signature" и доступно по адре-су **http://www.w3.org/TR/SOAP-dsig/**. Расширение уточняет элемент <signature>, определенный в предыдущем расширении "SOAP Security Extensions". Уточненный элемент <signature>, принадлежащий пространству имен с идентификатором http://schemas.xmlsoap.org/soap/security/2000-12, тоже содержит вложенный элемент <ds:Signature>, но последний элемент определен в более новом пространстве имен с идентификатором http://www.w3.org/2000/09/xmldsig#.

Кроме того, расширение "SOAP Security Extensions: Digital Signature" задает Правила включения элемента <Signature> В заголовок <Header> SOAP-

послания, и правила его обработки промежуточными серверами и Web-службами.

Наконец, в июле 2002 года появилось новое расширение протокола SOAP — спецификация "WS-Security", разработанная фирмами IBM, Microsoft и VeriSign. Остановимся на ней подробнее.

Спецификация "WS-Security"

Спецификация "Web Services Security", сокращенно "WS-Security", доступна по адресу **http://www-106.ibm.com/developerworks/webservices/library/ws-secure/**. Несмотря на свое название, она не вносит какие-то новые правила шифрования или аутентификации. Она определяет всего один элемент <security> в пространстве имен с идентификатором http://schemas.xmlsoap.org/ ws/2002/04/secext.

Сразу же после выхода спецификации "WS-Security", в августе 2002 года, появилось приложение к ней, названное "WS-Security Addendum". Его можно посмотреть по адресу **http://msdn.microsoft.com/webservices/.** Оно изменило идентификатор пространства имен, теперь это http://schemas.xmlsoap.org/ ws/2002/07/secext. В приложении исправлены мелкие ошибки и сделаны некоторые дополнения к спецификации "WS-Security".

У элемента <security>, определенного в спецификации "WS-Security", может быть любое число любых атрибутов. В него можно вложить произвольные элементы XML, В ТОМ ЧИСЛе элементы <Encryption>, <Signature>, <Authorization>, определенные в расширениях протокола SOAP. Можно не использовать эти элементы, а вложить непосредственно элементы <EncryptedData>, <ds:signature>, определенные в спецификации "XML Encryption" и в рекомендации RFC 3275. Спецификация определяет и некоторые дополнительные элементы, которые можно вложить в элемент <Security>, и блоки заголовка SOAP-послания. Главное же в спецификации "WS-Security" то, что она расширяема, и в ее расширениях можно определять конкретные элементы.

Для аутентификации пользователя по его имени и паролю спецификация определяет элемент <UsernameToken> С вложенными элементами <Username> и <Password>. Необязательный атрибут туре элемента <Password> указывает тип записи пароля. Пока он принимает всего два значения: значение по умолчанию "PasswordText" — пароль записывается открытым текстом, и значение "PasswordDigest" — записывается дайджест пароля в кодировке Base64. Дайджест создается по алгоритму SHA-1.

Приложение к спецификации добавило два необязательных вложенных элемента, применяемых при создании дайджеста пароля: элемент <Nonce> содержит одноразовый 16-байтовый пароль, включаемый в дайджест, эле-

мент <created> содержит дату и время создания пароля, которые тоже включаются в дайджест. Вот полный пример заголовка SOAP-послания с записью о пользователе:

```
<soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">

  <soap:Header>

    <wsse:Security>

      <wsse:UsernameToken>

        <wsse:Username>Ivan</wsse:Username>

        <wsse:Password Type="wsse:PasswordDigest">
            vb2zDRGHjopcDfXn4Beg6Unm4Hm=
        </wsse:Password>

        <wsse:Nonce>
            5uW4ABku/m6/S5rnE+L7vg==
        </wsse:Nonce>

        <wsu:Created xmlns:wsu=

          "http://schemas.xmlsoap.org/ws/2002/07/utility">

            2002-11-13T09:00:00Z

        </wsu:Created>

      </wsse:UsernameToken>

    </wsse:Security>

  </soap:Header>

  <soap:Body>

    <!- Тело послания ->
```

```
    </soap:Body>

</soap:Envelope>
```

Для хранения двоичной информации, скажем, сертификата, спецификация "WS-Security" определяет элемент <BinarySecurityToken>. Вот, например, сертификат, полученный по протоколу X.509, и записанный в кодировке Base64:

```
<wsse:BinarySecurityToken
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
    Id="cert" ValueType="wsse:X509v3"
    EncodingType="wsse:Base64Binary">

        MIIEZzCCA9CgAwIBAgIQEmtJZc0...

</wsse:BinarySecurityToken>
```

Спецификация, точнее, приложение к ней, рекомендует хранить сертификаты именно в элементе <BinarySecurityToken>, а не в элементе <ds:KeyInfo>.

Защищаемые элементы могут располагаться не только внутри элемента <security>, но и в другом месте SOAP-послания, и даже в другом месте Интернета. В таком случае в элемент <security> вкладывается определенный В Спецификации элемент <SecurityTokenReference>, В КОТОРЫЙ ВКЛадываются элементы <Reference> со ссылками на защищаемые ресурсы. Ссылки даются в виде строки URI в атрибуте URI. Например:

```
<wsse:SecurityTokenReference
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">

    <wsse:Reference
        URI="http://www.some.com/f1.xml#token1" />

    <wsse:KeyIdentifier wsu:Id="akey"
        ValueType="wsse:X509v3" EncodingType="wsse:Base64Binary">

        MIIEZzCCA9CgAwIBAgIQEmtJZc0...
    </wsse:KeyIdentifier>

</wsse:SecurityTokenReference>
```

Элемент <KeyIdentifier> Призван заменить собой элемент <ds:KeyName>, обычно вкладываемый в элемент <ds:KeyInfo> при создании цифровой подписи.

Элемент <SecurityTokenReference> разрешается вкладывать не только в элемент <Security>, но и в элементы <ds:KeyInfo> цифровой подписи. Сам элемент <ds:KeyInfo> рекомендуется вкладывать непосредственно в элемент <Security>, а не В элемент <ds: Signature>.

При обмене SOAP-посланиями часто имеет значение дата и время отправки послания и его обработки, а также срок действия послания. Приложение "WS-Security Addendum" определяет блок заголовка <Timestamp>. Этот элемент определен в другом пространстве имен с идентификатором http://schemas.xmlsoap.org/ws/2002/07/utility.

В элемент <Timestamp> вкладываются элементы <Created>, <Expires> И <Received> с датой и временем создания, сроком действия и временем обработки SOAP-послания промежуточным сервером. Дата и время относятся к типу dateTime языка XSD. Их рекомендуется отсчитывать по правилам UTC, в противном случае следует использовать атрибут ValueType, в котором надо записать тип, описанный в схеме XML. Например:

```
<soap:Envelope
   xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
   xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">

  <soap:Header>

    <wsu:Timestamp>

       <wsu:Created>2002-11-13T08:42:00Z</wsu:Created>
       <wsu:Expires>2002-12-13T09:00:00Z</wsu:Expires>
       <wsu:Received Actor="http://some.com/" Delay="60 000">
             2002-11-13T08:44:00Z
       </wsu:Received>

    </wsu:Timestamp>
    ...
  </soap:Header>
  <soap:Body>
    ...
  </soap:Body>

</soap:Envelope>
```

Что дальше?

Как видно из предыдущего раздела, спецификация "WS-Security" определяет только самые общие правила составления блоков заголовка SOAP-послания, обеспечивающих безопасность SOAP-послания. Конкретные правила должны быть определены в расширениях этой спецификации. Корпорации IBM и Microsoft работают над дальнейшим развитием и детализацией правил обеспечения безопасности Web-служб. В настоящее время разрабатываются следующие спецификации, расширяющие спецификацию "WS-Security":

- Спецификация "WS-Policy" описывает методы проведения той или иной политики безопасности. Она определяет ограничения доступа, форматы кодирования, используемые алгоритмы шифрования, форматы файлов политики. Спецификация расширяема и может быть дополнена новыми элементами.
- Спецификация "WS-Trust" описывает методы установления доверительных отношений между Web-службами, их клиентами и посредниками.
- Спецификация "WS-Privacy", совместно со спецификациями "WS-Security", "WS-Policy" и "WS-Trust", определяет правила сохранения коммерческой тайны предприятия или учреждения при обмене SOAP-посланиями.

В дальнейшем IBM и Microsoft планируют выпустить следующий слой спецификаций, основанных уже на спецификациях "WS-Policy", "WS-Trust" и "WS-Privacy". Новые спецификации призваны конкретизировать и расширить требования предыдущих спецификаций.

- Спецификация "WS-Secure Conversation" будет описывать правила обмена конфиденциальной информацией, аутентификации сообщений, создания ключей и обмена ими.
- Спецификация "WS-Federation" будет рассматривать доверительные отношения в разнородных средах и в распределенных приложениях.
- Спецификация "WS-Authorization" создаст стандарт для авторизации клиентов и сообщений.

Все эти спецификации основаны на известных правилах и алгоритмах шифрования, аутентификации, авторизации. Они просто вводят стандарты применения этих правил в Web-службах. Работа только началась, нас ждет еще множество решений проблем безопасности Web-служб.

ГЛАВА 9

Развитие Web Services

Раз вы дочитали книгу до этого места, значит, поняли, что технология Web Services только начинает развиваться. Плодотворная идея представить информацию в виде документов XML и пересылать ее по Интернету, используя только протокол HTTP, находит множество воплощений. Десятки фирм и рабочих групп активно развивают эту технологию. Буквально каждую неделю появляются новые программные продукты и новые версии старых продуктов. То и дело обновляются протоколы и спецификации Web Services и возникают новые спецификации. Расширяется сфера применения Web Services, все больше фирм создают у себя Web-службы.

В этой главе я постараюсь дать обзор новейших тенденций развития технологии Web Services. Посмотрим сначала, как развиваются "три кита", на которых стоят Web-службы — протоколы SOAP, WSDL и UDDI.

Протокол SOAP

Близится к завершению работа над версией 1.2 протокола SOAP. Сейчас, когда вы читаете эти строки, уже должна выйти ее окончательная редакция. Описание протокола, сделанное в *главе 3,* основано на черновой редакции SOAP 1.2, опубликованной в июне 2002 года. Не думаю, что эта редакция будет сильно изменена, поскольку всякие дополнения к протоколу сейчас оформляются в виде отдельных документов — расширений (extensions) протокола SOAP.

В версию SOAP 1.2 внесено много мелких изменений: исключен атрибут `encodingStyle` из корневого элемента `<Envelope>`, атрибут `actor` заменен

атрибутом role, значения атрибута mustUnderstand записываются словами "true" и "false". Введены новые элементы и атрибуты, исключен HTTP-заголовок SOAPAction, определен новый MIME-тип application/soap+xml, изменено содержимое элемента <Fault>. Все это подробно изложено в *главе 3.*

Пока не выпущена окончательная редакция SOAP 1.2, производители программных продуктов, реализующих протокол SOAP, вынуждены основывать их на версии SOAP 1.1. Впрочем, уже в первой версии Apache Axis реализовано большинство конструкций SOAP 1.2. Как только версия 1.2 будет утверждена, производители сразу же воплотят ее в новых версиях своих продуктов.

Протокол SOAP, по-видимому, будет в дальнейшем заменен общим протоколом пересылки любых документов XML, названным "XMLP" (XML Protocol). Но разработка этого протокола сильно затянулась, она еще не вышла из начальной стадии. Вероятнее всего, Web-службы еще некоторое время будут действовать в рамках протокола SOAP 1.2. Его текущее состояние и процесс разработки можно посмотреть на сайте **http://www.w3.org/TR/xp/.**

Описание на языке WSDL

Во время написания книги действовала версия 1.1 языка описания Web-служб WSDL, но в июле 2002 года был опубликован черновой вариант версии WSDL 1.2. Он доступен по адресу **http://www.w3.org/TR/wsdl12**.

Основные цели новой версии — привести язык WSDL в соответствие с требованиями схемы XML и описать связь языка с версией SOAP 1.2. Кроме этого, в новую версию введены некоторые новые атрибуты, устаревшие атрибуты сделаны необязательными, уточнены многие правила описания Web-служб. Описание языка WSDL, сделанное в *главе 4,* основано на версии WSDL 1.2.

Несмотря на появление версии WSDL 1.2, ее внедрение произойдет еще не скоро. С одной стороны, судя по скорости разработки спецификации, ее окончательная редакция в ближайшее время не выйдет. С другой стороны, выпущено уже очень много программных средств, автоматически генерирующих WSDL-описания Web-служб и создающих клиентские приложения по этим описаниям. Все эти средства основаны на WSDL 1.1. Будет нелегко в кратчайший срок перевести их на новую версию.

Текущее состояние языка WSDL непрерывно отражается на сайте **http://www.w3.org/ws/desc/.**

Реестр UDDI

Реестры **UDDI** пока создаются и развиваются, основываясь на спецификации **UDDI** 2.0, изложенной в нескольких документах. В июле 2002 года вышла версия 3.0 спецификации UDDI. Она записана в одном документе "**UDDI** Version 3.0 Published Specification", доступном по адресу **http://uddi.org/pubs/uddi_v3.htm**.

По новой спецификации можно зарегистрировать данные сразу в нескольких реестрах, для чего введено понятие присоединенных (affiliate) реестров и изменен формат записи ключей. Усилена безопасность хранящихся в реестре сведений. В частности, реестры теперь могут хранить сведения, подписанные цифровой подписью. Новая спецификация позволяет реестру определить политику обработки хранящихся в нем сведений. Для этого введен элемент `operationalInfo`. Реестр теперь обеспечивает интернационализацию хранящихся в нем сведений.

Спецификация UDDI 3.0 вносит и другие изменения в структуру сведений, хранящихся в реестре, например, введены дополнительные правила поиска и сортировки с помощью шаблонов. Обеспечена согласованность документов **UDDI** с описанием Web-службы, сделанном на языке WSDL. Значительно расширен интерфейс UDDI API, в него внесено много новых функций. Эти изменения приведены в *главе* 5, описывающей версию **UDDI** 3.0.

Полное внедрение версии UDDI 3.0 связано с большими трудностями — надо привести в соответствие с ней огромное количество информации, уже хранящейся в реестрах UDDI. Поскольку версия 3.0 только расширяет версию 2.0, почти ничего в ней не меняя, владельцы реестров и поставщики программного обеспечения идут проверенным путем. Они выпускают и внедряют продукты, совместимые со старыми версиями UDDI.

Текущее состояние дел по разработке спецификации UDDI всегда можно посмотреть на сайте **http://www.uddi.org/**.

Фирменные разработки

Правила создания и функционирования Web-служб описываются не только протоколом SOAP, языком WSDL и реестром UDDI. Определение Web Services, приведенное в *главе 2,* вообще ничего не говорит об этих "трех китах". Оно даже не упоминает протокол HTTP. По этому определению для Web-служб характерно только использование документов XML и технологии WWW. Многие фирмы создают свои протоколы и спецификации Web-служб, основанные лишь на XML и WWW. В предыдущих главах уже упоминались протоколы и спецификации XML-RPC, DIME, WS-Routing, WS-

Inspection, WS-Security. Большинство нововведений принадлежит фирмам **IBM** и Microsoft и сообществу OASIS. Познакомимся подробнее с их деятельностью.

Наибольшую активность в развитии Web Services, пожалуй, проявляет фирма IBM. Она разработала и активно внедряет несколько новых спецификаций, ориентированных, главным образом, на организацию деловых связей и получения из них единого бизнес-процесса.

Язык описания потоков работ WSFL

Язык описания потоков работ WSFL (Web Services Flow Language) создан фирмой IBM с двумя целями: описать последовательность обработки информации несколькими Web-службами, образующую бизнес-процесс (flow model), и описать взаимодействие Web-служб в этом процессе (global model). Для этого устанавливаются связи между SEI-интерфейсами взаимодействующих Web-служб. При этом активно используется WSDL-описание этих Web-служб.

Спецификация WSFL опубликована в документе **http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf**.

Описание WSFL выполняется в виде документа XML с корневым элементом `<definitions>`, в который вкладывается любое число необязательных элементов `<flowModel>`, описывающих поток информации, и элементов `<globalModel>`, описывающих взаимные связи Web-служб. Кроме того, в корневой элемент можно вложить необязательные элементы <import> и <serviceProviderType>.

Бизнес-поток описывается элементами, вложенными в элемент `<flowModel>`. Начало потока указывается элементом `<flowSource>`, конец потока — элементом `<flowSink>`. Элементы <serviceProvider> описывают Web-службы, вовлеченные в поток, а элементы <activity> — операции, выполняемые этими Web-службами. Операции связаны либо по управлению, либо по обрабатываемым данным. Связь по управлению описывается элементом <controlLink>, связь по данным — элементом <dataLink>. Имена входящей в эту связь и выходящей из нее Web-службы задаются атрибутами source и target этих элементов.

Взаимодействие Web-служб описывается элементами, вложенными в элемент `<globalModel>`. Сами Web-службы описываются элементами <serviceProvider>, а связи между ними — элементами `<plugLink>`, в которые вкладываются элементы <source> и <target>, описывающие две Web-службы, между которыми устанавливается связь.

В листинге 9.1 приведена схема вложенности элементов языка WSFL.

Листинг 9.1. Вложенность элементов языка WSFL

```
<definitions name="">

   <import namespace="" location="" /> [*]

   <serviceProviderType name="">  [*]

      <portType name="" /> [*]
      <import namespace="" location="" /> [*]

   </serviceProviderType>

   <serviceProvider name="">
      <portType name="" />
   </serviceProvider>

   <flowModel name="" serviceProviderType="">  [*]

      <flowSource name="">  [?]
         <output name="" message="" />
      </flowSource>

      <flowSink name="">  [?]
         <input name="" message="" />
      </flowSink>

      <serviceProvider name="" type="" /> [+]

      <export lifecycleAction="">  [*]

           <source portType="" operation="">  [?]

           <target portType="" operation="">  [?]
              <map  />
           </target>

           <ref >  [*]
```

```
        <plugLink> [?]
            <source portType="" operation="" /> [?]
            <target portType="" operation="" /> [?]
            <map /> [?]
            <locator> [?]
        </plugLink>

    </export>

    <activity name=""> [*]

        <input name="" message="" />
        <output name="" message="" />
        <performedBy serviceProvider="" />

        <implement>
            <internal> | <export>
            <export portType="" operation="">
                <map />
            </export>
        </implement>

        <join condition="" />

        <materialize>
            <mapPolicy> | <construction>
        </materialize>

    </activity>

    <datalink name="" source= "" target="" /> [*]
    <controlLink name="" source="" target="" /> [*]

</flowModel>

<globalModel name="" serviceproviderType="">

    <serviceProvider name="" serviceProviderType="">
```

```
        <export>
          <source portType="" operation="" />
          <target portType="" operation="" />
        </export>

        <locator type="" service="" />

    </serviceProvider>

    <plugLink>
        <source serviceProvider="" portType="" operation="" />
        <target serviceProvider="" portType="" operation="" />
    </plugLink>

  </globalModel>

</definitions>
```

Как видно даже из такого краткого описания, язык WSFL позволяет описать во всех подробностях последовательную обработку SOAP-посланий несколькими Web-службами.

Корпорация Microsoft создала свой язык описания бизнес-процессов, непохожий на язык WSFL. Он называется XLANG и разработан для сервера BizTalk. С языком XLANG можно ознакомиться на сайте **http://msdn.microsoft.com/library/**.

Язык описания бизнес-процессов BPEL4WS

Фирма IBM откликнулась на язык XLANG созданием языка BPEL4WS (Business Process Execution Language for Web Services), призванного объединить языки WSFL и XLANG. Спецификация языка BPEL4WS доступна по адресу **http://www-106.ibm.com/developerworks/library/ws-bpel/**.

Язык BPEL4WS создает композицию Web-служб, участвующих в бизнес-процессе, и описывает ее как одну Web-службу элементами <portType> языка WSDL. Такая композиция называется *процессом BPEL4WS* и записывается в теле элемента <process>. Каждый шаг процесса называется *действием* (activity). В число действий входит обращение к какой-либо Web-службе для выполнения Web-услуги <invoke>, ожидание результата <wait>, получение запроса от клиента <receive>, создание ответа <reply>, копирование данных <assign>.

Действия, описанные элементами <sequence>, <switch>, <while>, <flow>, <pick>, позволяют задать сложный нелинейный алгоритм обработки данных Web-службами, входящими в процесс BPEL4WS.

Действия <throw> и <catch> позволяют обработать исключительные ситуации, возникающие при выполнении других действий, основываясь на элементах <fault> описания WSDL.

Процесс BPEL4WS описывается, как всякая Web-служба, документом WSDL, в который записываются дополнительные элементы <serviceLinkType>, определенные в пространстве имен с идентификатором http://schemas.xmlsoap.org/ws/2002/07/service-link/. Эти элементы вкладываются непосредственно в корневой элемент <definitions> WSDL-описания. Они содержат один или два элемента <role>, содержащих ссылки на WSDL-описание тех Web-служб, которые выполняют действия, составляющие процесс.

Само же описание Web-служб, выполняющих действие, производится элементами <partner>, вложенными в элемент <partners>, который, в свою очередь, вкладывается в элемент <process>.

В листинге 9.2 показана схема вложенности элемента <process>.

Листинг 9.2. Элемент <process> языка BPEL4WS

```
<process>

   <partners> [?]
      <partner name="" serviceLinkType="" myRole="" /> [+]
   </partners>

   <containers> [?]

      <container name="" messageType="" /> [+]
         <message> [*]
      </container>

   </containers>

   <faultHandlers> [?]

      <catch faultName="" faultContainer="">

         <reply partner="" portType="" operation=""
```

```
            container="" faultName="" /> [*]

    </catch>

    <catchAll> [?]

</faultHandlers>

<correlationSets> [?]
    <correlationSet name="" properties=""> [+]
</correlationSets>

<compensationHandler>
     <compensate> | <Действия>
</compensationHandler>

<flow>

    <links>
       <link name="" /> [*]
    </links>

    <pick> [*]
      <onMessage> [+]
      <onAlarm> [*]
    </pick>

    <sequence> [*]

       <assign>

          <copy>
             <from container="" part="" />
             <to container=""   part="" />
          </copy>

       </assign>
```

```
<invoke partner="" portType="" operation=""
        inputContainer="" outputContainer="">

   <source linkName="" />
   <correlations> [?]
   <catch> [*]
   <catchAll> [+]
   <compensationHandler> [?]

</invoke>

 <receive partner="" portType="" operation=""
          container="">

    <source linkName="" /> [*]

 </receive>

 <reply partner="" portType="" operation=""
        container="">

   <correlations> [?]

   <target linkName="" /> [*]

 </reply>

</sequence>

<switch> [*]
   <case condition="" /> [+]
   <otherwise> [?]
</switch>

<throw faultName="" faultContainer="" />

<wait for="" until="" />
```

```
        <while condition=""> [*]

    </flow>

</process>
```

Фирма **IBM** активно продвигает язык BPEL4WS в качестве стандарта для описания взаимодействия Web-служб в бизнес-процессах. Она уже выпустила программный продукт BPWS4J, реализующий спецификацию BPEL4WS. Кроме интерфейсов и классов, реализующих BPEL4WS, в состав продукта входят примеры описаний на языке BPEL4WS, простой редактор для создания файлов с описаниями BPEL4WS, утилиты для проверки документов BPEL4WS. Пакет BPWS4J свободно доступен по адресу **http://www.alphaworks.ibm.com/tech/bpws4j/**.

Спецификация WS-Coordination

Спецификация WS-Coordination, разработанная фирмами **IBM,** Microsoft и BEA Systems, описывает самые общие правила взаимодействия компонентов распределенного приложения, которые можно затем детализировать другими, более конкретными спецификациями. Ее можно посмотреть на сайте **http://www.ibm.com/developerworks/library/ws-coor/**.

Компоненты распределенного приложения [10] часто работают на разном, плохо совместимом оборудовании и используют самые разные протоколы. Для координации их деятельности создаются специальные *согласующие протоколы* (coordination protocols). Главная задача спецификации — установить общие правила регистрации и активизации согласующих протоколов. Эту задачу должна решить *Web-служба координации* (coordination service). Она содержит в себе службу активизации и службу регистрации. У каждого компонента может быть своя Web-служба координации.

Web-служба координации активизируется при получении SOAP-послания от клиента с блоком заголовка `<CreateCoordinationContext>`. При этом происходит обращение к службе активизации, адрес порта которой записан во вложенном элементе <Activationservice>. Еще два вложенных элемента `<CoordinationType>` И `<RequesterReference>` описывают ТИП согласующего протокола и адрес клиента, приславшего запрос. Эти элементы и другие элементы, введенные в спецификацию, определены в пространстве имен с идентификатором http://schemas.xmlsoap.org/ws/2002/08/wscoor.

После активизации Web-служба координации посылает ответное SOAP-послание С блоком заголовка <CreateCoordinationContextResponse>, В КОторый вложен элемент `<RequesterReference>`, а также элемент `<CoordinationContext>`, содержащий сведения о созданном контексте Web-службы координации, в частности, ее адрес.

В каждое SOAP-послание, использующее Web-службу координации, вставляется блок заголовка <coordinationcontext>. В элемент <CoordinationContext> вкладывается элемент <CoordinationType>, описывающий тип согласующего протокола, и элемент <RegistrationService>, в котором, во вложенном элементе <Address>, записывается URI-адрес порта Web-службы координации.

Получив ответ, клиент регистрируется в Web-службе координации с помощью службы регистрации. Для этого он посылает Web-службе SOAP-послание с блоком заголовка <Register>, в который вложен элемент <Registrationservice> с содержимым, полученным от Web-службы координации, и элемент <ProtocolIdentifier>, который указывает тип согласующего протокола. Кроме того, вкладывается элемент <ParticipantProtocolService>, в котором указан адрес Web-службы, реализующей согласующий протокол, выбранный клиентом, и уже знакомый элемент <RequesterReference>.

На это Web-служба координации отвечает SOAP-посланием с блоком заголовка <RegisterResponse>, содержащим элемент <RequesterReference> и элемент <CoordinatorProtocolService>, содержащий адрес Web-службы, реализующей согласующий протокол, выбранный Web-службой координации.

После регистрации происходит взаимодействие Web-служб, автоматически использующее согласующие протоколы.

Спецификация WS-Coordination ничего не говорит о согласующих протоколах и способах реализации Web-службы координации. Это должны сделать другие спецификации. Одна из таких спецификаций — спецификация WS-Transaction, описывающая тип согласующего протокола, устанавливающего правила создания и выполнения транзакций.

СпецификацияWS-Transaction

Спецификация WS-Translation, созданная фирмами IBM, Microsoft и BEA Systems, предлагает две модели, детализирующие правила WS-Coordination: модель недолговременной транзакции с ограниченным числом участников AT (Atomic Transaction) и модель долговременной транзакции с большим числом участников BA (Business Activity). Спецификация представлена на сайте **http://www.ibm.com/developerworks/library/ws-transpec/**.

Модель AT управляет одной транзакцией по принципу "все или ничего". Согласующий протокол транзакции регистрируется по правилам спецификации WS-Coordination. В элементе <wscoor:CoordinationType>, определяющем тип согласующего протокола, указывается строка http://schemas.xmlsoap.org/ws/2002/08/wstx. Спецификация устанавливает правила завершения и от-

ката транзакции, например, предлагается двухфазная фиксация [10] транзакции.

После регистрации согласующего протокола по правилам спецификации WS-Coordination, клиенты начинают управление транзакцией, записывая в SOAP-ПОСЛаниях элементы <Prepare>, <ReadOnly>, <Commit>, <Rollback>, <OnePhaseCommit>, <phasezero>, **определенные в пространстве имен с идентификатором** http://schemas.xmlsoap.org/ws/2002/08/wstx. **Web-служба координации отвечает SOAP-посланиями с элементами** <Prepared>, <Aborted>, <Committed>, <Notified>, <PhaseZeroCompleted>, <Unknown>. **В каждый ИЗ этих элементов вкладываются элементы** <SourceProtocolService> **и** <TargetProtocolService>, содержащие адреса портов Web-служб, реализующих соответствующие протоколы. Для создания и выполнения транзакции в рамках Web-службы координации создаются соответствующие службы, выполняющие описанные этими элементами действия.

Модель BA охватывает, как правило, несколько транзакций, образующих бизнес-процесс, описанный, возможно, на языке BPEL4WS. Во время активизации и регистрации, в элементе <wscoor:CoordinationType>, определяющем тип согласующего протокола, указывается строка http://schemas.xmlsoap.org/ws/2002/08/wstx.

Модель BA задает правила распределения ресурсов между участниками бизнес-процесса и правила освобождения ресурсов, не используемых ими. Для этого спецификация WS-Transaction определяет элементы <Complete>, <Close>, <Cancel>, <Compensate>, <Forget>, используемые Web-службой координации для опроса участников бизнес-просесса, и элементы <Completed>, <Closed>, <Compensated>, <Faulted>, <Exited>, <Unknown>, применяемые участниками бизнес-процесса для ответа. Все эти элементы определены в пространстве имен с идентификатором http://schemas.xmlsoap.org/ws/2002/08/wsba. Получив ответ участника, Web-служба координации распоряжается ресурсами.

Деятельность организации WS-I

Список протоколов и спецификаций Web Services далеко не ограничен перечисленными выше документами. Можно назвать еще множество аббревиатур, относящихся к Web-службам: SAML, WSXL, WSCL, WSCM, WSEL, WSML, WSUI, WS-License, WS-Referral. Пока я их перечисляю, возникают новые спецификации. Такую тенденцию необходимо упорядочить и стандартизировать. С этой целью весной 2002 года по инициативе IBM и Microsoft была создана организация, названная "Web Services Interoperability Organization", сокращенно WS-I. Ее целью была провозглашена выработка стандартов, обеспечивающих совместимость программных продуктов разных

производителей, относящихся к разным платформам, операционным системам и языкам программирования. К организации WS-I уже примкнули все крупные производители компьютерного оборудования и программного обеспечения. Официальный сайт организации **http://www.ws-i.org/.**

Организация WS-I вырабатывает четыре типа документов: профили взаимодействия, инструменты для тестирования, образцы применения и примеры приложений, построенных по этим образцам.

Профиль взаимодействия (interoperability profile) или просто *профиль* — это набор протоколов и/или спецификаций, взятых в определенный момент их развития, вместе с правилами их наилучшего использования. Профиль содержит готовое проверенное решение, которое могут сразу применить создатели Web-служб.

Основой для первого профиля, разработанного организацией WS-I, послужил набор взаимодействующих протоколов и спецификаций XML Schema 1.0, SOAP 1.1, WSDL 1.1 и UDDI 1.2. Этот профиль назван "WS-Basic".

Профиль WS-Basic

Профиль WS-Basic Profile создан фирмами IBM, Microsoft и BEA Systems. Его черновая версия опубликована на странице **http://www.ws-i.org/Profiles/Basic/2002-10/BasicProfile-1.0-WGD.htm**. Профиль рассматривает четыре группы вопросов: пересылку SOAP-посланий, описание WSDL, регистрацию UDDI и безопасность. Рассмотрим их подробнее.

Пересылка SOAP-посланий

Организация WS-1 выбрала для профиля WS-Basic следующие версии протоколов:

- SOAP 1.1;
- XML 1.0 (Second Edition);
- HTTP/1.1.

Большинство рекомендаций по использованию протокола SOAP 1.1 относится к сообщениям об ошибках `<Fault>`. Профиль рекомендует не использовать в элементе `<Fault>` других вложенных элементов кроме `<faultcode>`, `<faultstring>`, `<faultactor>` и `<detail>`. Имена этих элементов рекомендуется записывать без префиксов. В элементе `<faultcode>` следует использовать только коды ошибок, определенные SOAP 1.1.

Профиль не рекомендует употреблять атрибут `encodingStyle` в элементах тела SOAP-послания <Body> и вообще во всех элементах, определенных в пространстве имен с идентификатором http://schemas.xmlsoap.org/soap/envelope/.

Само SOAP-послание не должно содержать описаний DTD, инструкций по обработке и никаких элементов после элемента <Body>.

Все элементы, вложенные в элемент <Body>, должны быть записаны расширенными именами с префиксами. Это облегчает синтаксический разбор послания.

Профиль рекомендует использовать для пересылки SOAP-посланий только протокол HTTP версии 1.1, причем только метод POST. Если применен другой HTTP-метод, то сервер должен вернуть код ответа "405 Method not Allowed". Послание с сообщением об ошибке `<Fault>` должно иметь код ответа "500 Server Error", а послание без элемента `<Fault>` — код ответа "200 OK" или "202 Accepted".

Сервер должен вернуть ответ с кодом "415 Unsupported Media Type", если в поле `Content-Type` заголовка запроса не был указан MIME-тип `text/xml`.

Сервер должен послать сообщение с кодом ответа "307 Temporary Redirect", если он направил сообщение другому серверу.

Не следует использовать расширения протокола HTTP, сделанные в рекомендации RFC 2774.

Как видите, большинство этих рекомендаций учтено в версии SOAP 1.2.

Описание WSDL

Профиль WS-Basic содержит следующие версии спецификаций:

- WSDL 1.1;
- XML Schema 1.0.

Профиль дает множество рекомендаций по оформлению описания Web-служб, сделанного на языке WSDL.

Большинство рекомендаций относится к элементу `<import>` описания WSDL. Его бездумное применение приводит к большим трудностям при разборе описания Web-службы. Профиль рекомендует записывать элемент `<import>` первым в элементе <definitions> и использовать его только для включения других описаний или определений на языке XSD. В первом случае надо обращать особое внимание на значение атрибута targetNamespace включаемого описания. Оно должно соответствовать значению атрибута namespace элемента <import>. В последнем случае элемент <import> следует включать в элемент <types>.

Элемент <types> следует записывать первым в элементе <definitions> или сразу после элемента <import>. В элементе <types> не следует применять имена элементов из пространств имен, на которые нет ссылок в элементе <import>.

При описании посланий документного стиля в элементе <message> следует записывать не более одного элемента <part>, описывающего тело <Body> SOAP-послания. При этом надо обязательно использовать атрибут element, указывающий на XSD-определение типа элемента.

При описании посланий процедурного стиля можно записывать сколько угодно элементов <part>, при этом следует использовать их атрибут type и строго соблюдать порядок следования элементов <part>. Результат выполнения процедуры должен содержать только один элемент <part>, может быть, сложного типа.

В описании нельзя использовать операции типов `solicit-response` и `notification`.

Профиль рекомендует применять для описания пересылки посланий (SOAP binding) только протокол SOAP 1.1. В элементе <soap:binding> надо атрибутом transport указать значение "http://schemas.xmlsoap.org/soap/http". Так сделано в листинге 4.2. В атрибуте use следует указывать значение по умолчанию "literal".

В элементах <soap:fault> и <soap:headerfault> обязательно надо использовать атрибут part, а в элементе <soap:fault> еще и атрибут name.

Ни в коем случае нельзя смешивать в одном описании документный и процедурный стиль.

Большая часть этих рекомендаций уже учтена в версии WSDL 1.2.

Регистрация UDDI

В профиль WS-Basic включены следующие версии спецификаций UDDI 2.0:

- UDDI Version 2.04 API Published Specification;
- UDDI Version 2.03 Data Structure Reference;
- Version 2.0 UDDI XML Schema 2001;
- UDDI Version 2.03 Replication Specification;
- Published Specification, Version 2.03 Replication XML Schema 2001;
- UDDI Version 2.03 XML Custody Schema;
- UDDI Version 2.01 Operator's Specification, Published Specification.

Основная рекомендация профиля WS-Basic — максимально согласовать описание Web-службы, сделанное на языке WSDL, с описанием UDDI. Для этого профиль советует установить взаимно однозначное соответствие между элементами `<wsdl:port>` описания и элементами `<uddi:bindingTemplate>` описания UDDI. При этом значение атрибута

location элемента <wsdl:port> должно совпадать со значением атрибута accessPoint элемента <uddi:bindingTemplate>.

Если такое взаимно однозначное соответствие установлено, то следует пометить элемент <uddi:businessService> как совместимый с профилем WS-Basic. Для этого в элементе <keyedReference>, вложенном в элемент <categoryBag>, следует указать категорию ws-i-org:conformsTo И дать ей значение "http://wwww.ws-i.org/profiles/base/1.0".

Описание технических деталей <tModel> необходимо основывать на описании WSDL. Для этого в элементе <overviewURL>, вложенном в элемент <tModel>, следует указывать адрес описания WSDL, как это сделано в листинге 5.5. Кроме того, надо пометить элемент <tModel> как совместимый с описанием WSDL. Для этого надо во вложенном элементе <categoryBag> указать категорию uddi-org: types и дать ей значение "wsdlSpec".

Многие рекомендации профиля WS-Basic уже учтены в версии UDDI 3.0.

Безопасность

Для обеспечения безопасности профиль WS-Basic рекомендует протокол HTTPS, то есть следующие протоколы:

- HTTP Over TLS (рекомендация RFC 2818);
- TLS Protocol Version 1.0 (рекомендация RFC 2246);
- SSL Protocol Version 3.0;
- Internet X.509 Public Key Infrastructure Certificate and CRL Profile (рекомендация RFC 2459).

Это старые испытанные средства, их применение давно отработано на практике. Профиль дает лишь одну рекомендацию по использованию этих протоколов — оговаривать с Web-службой детали их применения.

Как видите, профиль WS-Basic дает ценные практические рекомендации по эффективной организации работы Web-службы. Эти рекомендации уже воплощаются в следующих версиях спецификаций Web Services. Работа только началась. Организация WS-I готовит новые профили, образцы их применения и примеры приложений. Вы можете ознакомиться с ними на сайте организации **http://www.ws-i.org/.**

Что дальше?

Вот и последняя страница книги. Вы познакомились с новейшей Web-технологией, широко распространяющейся по компьютерному миру и бур-

но развивающейся вширь и вглубь. Станет ли эта технология доминирующей, затмит ли она другие Web-технологии — неизвестно. Полувековая история развития вычислительной техники знает много шумных открытий, которым прочили роль панацеи от всех программистских бед. Большинство этих технологий благополучно забыто, другие заняли свое место и скромно работают на благо переработки информации. Скорее всего, то же самое произойдет и с Web Services. Но пока ниша Web-служб четко не определена. От нас, от нашего труда и нашей активности зависит место и область применения Web Services. Успехов вам в ваших проектах!

Список использованной литературы

1. Брогден Б., Минник К. Электронный магазин на Java и XML. — СПб.: Питер, 2002. — 400 с, ил.
2. Валиков А. Н. Технология XSLT. — СПб.: БХВ-Петербург, 2002. — 544 с, ил.
3. Гэри Д. JavaServer Pages. — М.: "Вильямс", 2002. — 464 с, ил.
4. Даконта М., Саганич А. XML и Java 2. — СПб.: Питер, 2001. — 384 с, ил.
5. Кэй М. XSLT. Справочник программиста. — СПб.: Символ-Плюс, 2002. - 1016 с, ил.
6. Мак-Лахлин Б. Java и XML. — СПб.: Символ-Плюс, 2002. — 544 с, ил.
7. Монсон-Хейфел Р. Enterprise JavaBeans. 3-е изд. — СПб.: Символ-Плюс, 2002. — 672 с, ил.
8. Фридл Дж. Регулярные выражения. — СПб.: Питер, 2001. — 352 с, ил.
9. Хабибуллин И. Ш. Самоучитель Java. — СПб.: БХВ-Петербург, 2001. — 464 с, ил.
10. Хабибуллин И. Ш. Создание распределенных приложений на Java 2. — СПб.: БХВ-Петербург, 2002. - 704 с, ил.
11. Холл М. Сервлеты и JavaServer Pages.— СПб.: Питер, 2001.— 496 с, ил.
12. Шапошников И. В. Web-сервисы Microsoft.NET. — СПб.: БХВ-Петербург, 2002. — 336 с, ил.
13. Яворский Дж., Перроун П. Дж. Система безопасности Java. Руководство разработчика. — М.: "Вильямс", 2001. — 528 с, ил.
14. Bequet H. et al. Beginning Java Web Services. — Wrox Press, 2002. — 411 p.

15. Bequet H. Professional Java SOAP. — Wrox Press, 2002. — 400 p.
16. Cable S. et al. Professional Java Web Services. — Wrox Press, 2002. — 600 p.
17. Chappell D., Jewell T. Java Web Services. — O'Reilly & Ass., 2002. — 276 p.
18. Deitel H. M. et al. Java Web Services For Experiences Programmers. — Prentice Hall, 2002. — 500 p.
19. Developing Java Web Services with Java APIs for XML using WSDP. — Syngress Media, 2002. — 501 p.
20. Englander R. Java and SOAP. — O'Reilly & Ass., 2002. — 300 p.
21. Irani R., Basha S. J. AXIS: Next Generation Java SOAP. — Wrox Press, 2002. - 250 p.
22. Jasnowski M. Java XML and Web Services Bible. — John Wiley & Sons, 2002. - 1000 p.
23. Java Web Services Unleashed. — SAMS Publ., 2002. — 752 p.
24. Mogha R., Niit, Preetham V. V. Java Web Services Programming. — John Wiley & Sons, 2002. - 555 p.
25. Nagappan R. et al. Developing Java Web Services. — John Wiley & Sons, 2002 - 800 p.
26. Simeonov G. et al. Building Web Services with Java: Making Sense of XML, SOAP, USDL, and UDDI. - SAMS Publ., 2001. - 600 p.
27. Wesley A. A. Programming Web Services with Java. — Manning Publ., 2002. - 325 p.

Предметный указатель

T

U

V

W

X

А

Б

Г

Д

З

И

К

Л

М

О

П

Р

С

Т

У

Ф

Ц

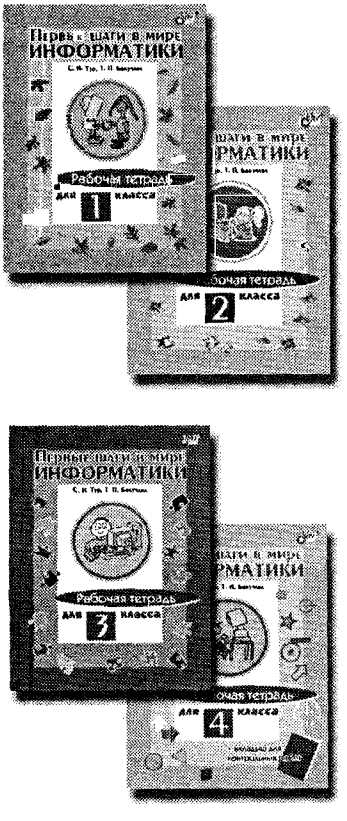
Э

Первые шаги в мирЕ ИНФОРМАТИКИ

Курс «Первые шаги в мире информатики» апробировался в течение 5 лет в школах Ленинградской области. В 2000 г. курс получил гриф: «Рекомендовано экспертным советом Комитета общего и профессионального образования Ленинградской области», имеет статус авторской программы.

Авторская программа «Первые шаги в мире информатики» с 1 по 9 годы обучения рассчитана для общеобразовательных и специализированных школ и является основным звеном в цепи непрерывного курса обучения информатике и информационным технологиям с 1 по 11 классы.

Курс может изучаться учащимися любого начального уровня развития и имеет полное программно-методическое обеспечение.



Рабочие тетради для учеников 1–4 класса начальной школы предназначены для проведения уроков по курсу «Первые шаги в мире информатики» с использованием педагогических программных средств «Страна „Фантазия"» (авторы Тур С. Н., Бокучава Т. П) и содержат теоретический материал и задачи для самостоятельного решения.

Наличие в каждом уроке дополнительного задания позволяет проводить занятия в безмашинном варианте. Тетради содержат уроки, позволяющие проводить диагностическое тестирование на развитие памяти, внимания, саморегуляции.

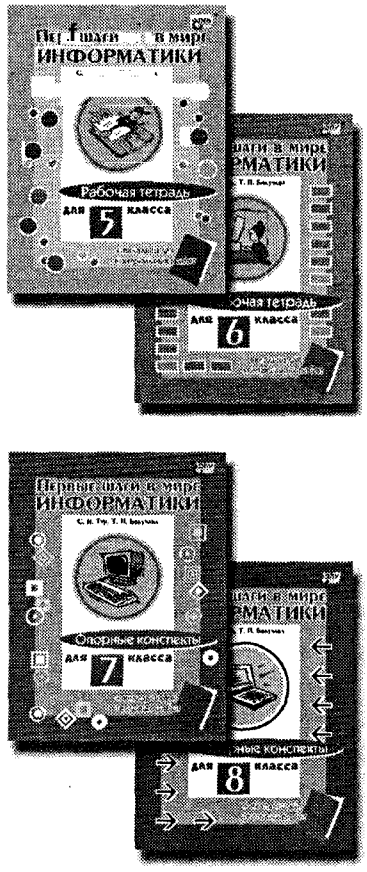
Уроки спланированы так, чтобы строго соблюдались возрастные санитарно-гигиенические нормы работы на компьютере.

Рабочая тетрадь для учеников 4 класса укомплектована тетрадью-вкладышем с самостоятельными и контрольными работами для двух вариантов.

ПЕРВЫЕ ШАГИ В МИРЕ ИНФОРМАТИКИ

Курс «Первые шаги в мире информатики» апробировался в течение 5 лет в школах Ленинградской области. В 2000 г. курс получил гриф: «Рекомендовано экспертным советом Комитета общего и профессионального образования Ленинградской области», имеет статус авторской программы.

Авторская программа «Первые шаги в мире информатики» с 1 по 9 годы обучения рассчитана для общеобразовательных и специализированных школ и является основным звеном в цепи непрерывного курса обучения информатике и информационным технологиям с 1 по 11 классы.



Рабочие тетради для учеников 5–6 классов и опорные конспекты для ученика 7–8 классов общеобразовательной школы предназначены для проведения уроков по курсу «Первые шаги в мире информатики» и содержат теоретический материал и задачи для самостоятельного решения.

Приводятся общие сведения о вычислительной технике, кодировании информации и программировании на языке QBasic. Описываются системы счисления, создание алгоритмов, работа с текстовым и графическим редакторами. Рассматриваются операционные системы, файлы, каталоги, команды MS-DOS, основные понятия Windows, файловые оболочки, архивирование файлов, вирусы и антивирусная защита, электронные таблицы.

В прилагаемых вкладышах представлены самостоятельные, контрольные и тестовые работы для двух вариантов.

Уроки спланированы так, чтобы строго соблюдались возрастные санитарно-гигиенические нормы работы на компьютере.

Книги издательства "БХВ-Петербург" в продаже:

Серия "В подлиннике"

Андреев А. и др. MS Windows XP: Home Edition и Professional	848 с.
Андреев А. и др. Windows 2000 Professional. Русская версия	700 с.
Андреев А. и др. Microsoft Windows 2000 Server. Русская версия	960 с.
Андреев А. и др. Новые технологии Windows 2000	576 с.
Андреев А. и др. Microsoft Windows 2000 Server и Professional. Русские версии	1056 с.
Ахаян Р. Macromedia ColdFusion	672 с.
Браун М. HTML 3.2 (с компакт-диском)	1040 с.
Вебер Дж. Технология Java (с компакт-диском)	1104 с.
Власенко С. Компакт-диск с примерами к книгам серии "В подлиннике": "MS Office XP в целом", "MS Access 2002", "MS Word 2002", "MS Excel 2002"	32 с.
Власенко С. Microsoft Word 2002	992 с
Гофман В., Хомоненко А. Delphi 6	1152 с.
Долженков В. MS Excel 2002	1072 с
Закер К. Компьютерные сети. Модернизация и поиск неисправностей	1008 с.
Колесниченко О., Шишигин И. Аппаратные средства PC, 4-е издание	1024 с.
Мамаев Е. MS SQL Server 2000	1280 с.
Матросов А. и др. HTML 4.0	672 с.
Михеева В., Харитонова И. Microsoft Access 2000	1088 с.
Михеева В., Харитонова И. Microsoft Access 2002	1040 с.
Новиков Ф., Яценко А. Microsoft Office 2000 в целом	728 с.
Новиков Ф., Яценко А. Microsoft Office XP в целом	928 с.
Ноутон П., Шилдт Г. Java 2	1072 с.
Пауэлл Т. Web-дизайн	1024 с.
Персон Р. Word 97	1120 с.
ПитцМ., Кирк Ч. XML	736 с.
Пономаренко С. Adobe Illustrator 9.0	608 с.
Пономаренко С. Adobe Photoshop 6.0	832 с.
Пономаренко С. CorelDRAW 9	576 с.
Пономаренко С. Macromedia FreeHand 9	432 с.
Русеев С. WAP: технология и приложения	432 с.
Секунов Н. Обработка звука на PC (с дискетой)	1248 с.
Сузи Р. Python (с компакт-диском)	768 с.
Тайц А. М., Тайц А. А. Adobe PageMaker 7.0	784 с.
Тайц А. М., Тайц А. А. Adobe InDesign	704 с.
Тайц А. М., Тайц А. А. CorelDRAW 9: все программы пакета	1136 с.
Тайц А. М., Тайц А. А. CorelDRAW 10: все программы пакета	1136 с.
Тихомиров Ю. Microsoft SQL Server 7.0	720 с.

Уильямс Э. и др. Active Server Pages (с компакт-диском)	672 с.
Усаров Г. Microsoft Outlook 2002	656 с.
Ханкт Ш. Эффекты CorelDRAW (с компакт-диском)	704 с.

Серия "Мастер"

CD-ROM с примерами к книгам "Ресурсы MS Windows NT Server 4.0" и "Сетевые средства Windows NT Server 4"	
Microsoft Press. Электронная коммерция. B2B-программирование (с компакт-диском)	368 с.
Microsoft Press. Visual Basic 6.0	992 с.
Microsoft Press. Ресурсы MS Windows NT Server 4.0	752 с
Айзекс С. Dynamic HTML (с компакт-диском)	496 с.
Анин Б. Защита компьютерной информации	384 с.
Асбари С. Корпоративные решения на базе Linux	496 с.
Березин С. Факс-модемы: выбор, подключение, выход в Интернет	256 с.
Березин С. Факсимильная связь в Windows	250 с.
Борн Г. Реестр Windows 98 (с дискетой)	496 с.
Бухвалов А. и др. Финансовые вычисления для профессионалов	320 с.
Валиков А. Технология XSLT	432 с.
Габбасов Ю. Internet 2000	448 с.
Гарбар П. Novell GroupWise 5.5: система электронной почты и коллективной работы	480 с.
Гарнаев А. Microsoft Excel 2000: разработка приложений	576 с.
Гарнаев А. Excel, VBA, Internet в экономике и финансах	816 с.
Гарнаев А., Гарнаев С. Web-программирование на Java и JavaScript	1040 с.
Гордеев О. Программирование звука в Windows (с дискетой)	384 с.
Гофман В., Хомоненко А. Работа с базами данных в Delphi	656 с.
Дарахвелидзе П. и др. Программирование в Delphi 5 (с дискетой)	784 с.
Дронов В. JavaScript в Web-дизайне	880 с.
Дубина А. и др. MS Excel в электронике и электротехнике	304 с.
Дубина А. Машиностроительные расчеты в среде Excel 97/2000 (с дискетой)	416 с.
Дунаев С. Технологии Интернет-программирования	480 с.
Жарков С. Shareware: профессиональная разработка и продвижение программ	320 с.
Зима В. и др. Безопасность глобальных сетевых технологий	320 с.
Киммел П. Borland C++ 5	976 с.
Костарев А. PHP в Web-дизайне	592 с.
Краснов М. DirectX. Графика в проектах Delphi (с компакт-диском)	416 с.
Краснов М. Open GL в проектах Delphi (с дискетой)	352 с.
Кубенский А. Создание и обработка структур данных в примерах на Java	336 с.
Кулагин Б. 3ds max 4: от объекта до анимации	448 с.
Купенштейн В. MS Office и Project в управлении и делопроизводстве	400 с.
Куприянов М. и др. Коммуникационные контроллеры фирмы Motorola	560 с.
Лавров С. Программирование. Математические основы, средства, теория	304 с.
Лукацкий А. Обнаружение атак	624 с.

Матросов А. Maple 6. Решение задач высшей математики и механики	528 с.
Медведев Е., Трусова В. "Живая" музыка на PC (с дискетой)	720 с.
Мешков А., Тихомиров Ю. Visual C++ и MFC, 2-е издание (с дискетой)	1040 с.
Миронов Д. Создание Web-страниц в MS Office 2000	320 с.
Мещеряков Е., Хомоненко А. Публикация баз данных в Интернете	560 с.
Михеева В., Харитонова И. Microsoft Access 2000: разработка приложений	832 с.
Новиков Ф. и др. Microsoft Office 2000: разработка приложений	680 с.
Нортон П. Разработка приложений в Access 97 (с компакт-диском)	656 с.
Одинцов И. Профессиональное программирование. Системный подход	512 с.
Олифер В., Олифер Н. Новые технологии и оборудование IP-сетей	512 с.
Подольский С. и др. Разработка интернет-приложений в Delphi (с дискетой)	432 с.
Полещук Н. Visual LISP и секреты адаптации AutoCAD	576 с.
Понамарев В. COM и ActiveX в Delphi	320 с.
Пономаренко С. Adobe InDesign: дизайн и верстка	544 с.
Попов А. Командные файлы и сценарии Windows Scripting Host	320 с.
Приписное Д. Моделирование в 3D Studio MAX 3.0 (с компакт-диском)	352 с.
Роббинс Дж. Отладка приложений	512 с.
Рудометов В., Рудометов Е. PC: настройка, оптимизация и разгон, 2-е издание	336 с.
Русеев Д. Технологии беспроводного доступа. Справочник	352 с.
Соколенко П. Программирование SVGA-графики для IBM	432 с.
Тайц А. Каталог Photoshop Plug-Ins	464 с.
Тихомиров Ю. MS SQL Server 2000: разработка приложений	368 с.
Тихомиров Ю. SQL Server 7.0: разработка приложений	370 с.
Тихомиров Ю. Программирование трехмерной графики в Visual C++ (с дискетой)	256 с.
Трельсен Э. Модель COM и библиотека ATL 3.0 (с дискетой)	928 с.
Федоров А., Елманова Н. ADO в Delphi (с компакт-диском)	816 с.
Федорчук А. Офис, графика, Web в Linux	416 с.
Чекмарев А. Windows 2000 Active Directory	400 с.
Чекмарев А. Средства проектирования на Java (с компакт-диском)	400 с.
Шапошников И. Web-сайт своими руками	224 с.
Шапошников И. Интернет-программирование	224 с.
Шапошников И. Справочник Web-мастера. XML	304 с.
Шилдт Г. Теория и практика C++	416 с.
Яцюк О., Романычева Э. Компьютерные технологии в дизайне. Логотипы, упаковка, буклеты (с компакт-диском)	464 с.

Серия "Изучаем вместе с BHV"

Березин С. Internet у вас дома, 2-е издание	752 с.
Тайц А. Adobe Photoshop 5.0 (с дискетой)	448 с.

Серия "Самоучитель"

Ананьев А., Федоров А. Самоучитель Visual Basic 6.0	624 с.
Васильев В. Основы работы на ПК	448 с.
Гарнаев А. Самоучитель VBA	512 с.
Герасевич В. Самоучитель. Компьютер для врача	640 с.
Дмитриева М. Самоучитель JavaScript	512 с.
Долженков В. Самоучитель Excel 2000 (сдискетой)	368 с.
Исагулиев К. Macromedia Dreamweaver 4	560 с.
Исагулиев К. Macromedia Flash 5	368 с.
Кетков Ю., Кетков А. Практика программирования: Бейсик, Си, Паскаль (с дискетой)	480 с.
Кирьянов Д. Самоучитель Adobe Premiere 6.0	432 с.
Кирьянов Д. Самоучитель MathCAD 2001	544 с.
Коркин И. Самоучитель Microsoft Internet Explorer 6.0	288 с.
Котеров Д. Самоучитель PHP 4	576 с.
Культин Н. Программирование на Object Pascal в Delphi 6 (с дискетой)	528 с.
Культин Н. Самоучитель. Программирование в Turbo Pascal 7.0 и Delphi, 2-е издание (с дискетой)	416 с.
Леоненков А. Самоучитель UML	304 с.
Матросов А., Чаунин М. Самоучитель Perl	432 с.
Омельченко Л., Федоров А. Самоучитель Microsoft FrontPage 2002	576 с.
Омельченко Л., Федоров А. Самоучитель Windows 2000 Professional	528 с.
Омельченко Л., Федоров А. Самоучитель Windows Millennium	464 с.
Пекарев Л. Самоучитель 3D Studio MAX 4.0	370 с.
Полещук Н. Самоучитель AutoCad 2000 и Visual LISP, 2-е издание	672 с.
Полещук Н. Самоучитель AutoCAD 2002	608 с.
Понамарев В. Самоучитель Kylix	416 с.
Секунов Н. Самоучитель Visual C++ 6 (сдискетой)	960 с.
Секунов Н. Самоучитель C#	576 с.
Сироткин С. Самоучитель WML и WMLScript	240 с.
Тайц А. М., Тайц А. А. Самоучитель Adobe Photoshop 6 (с дискетой)	608 с.
Тайц А. М., Тайц А. А. Самоучитель CorelDRAW 10	640 с.
Тихомиров Ю. Самоучитель MFC (сдискетой)	640 с.
Хабибуллин И. Самоучитель Java	464 с.
Хомоненко А. Самоучитель Microsoft Word 2002	624 с.
Шапошников И. Интернет. Быстрый старт	272 с.
Шапошников И. Самоучитель HTML 4	288 с.
Шилдт Г. Самоучитель C++, 3-е издание (сдискетой)	512 с.

Серия "Компьютеритворчество"

Деревских В. Музыка на PC своими руками	352 с.
Дунаев В. Сам себе Web-дизайнер	512 с.
Дунаев В. Сам себе Web-мастер	288 с.

Людиновсков С. Музыкальный видеоклип своими руками	320 с.
Петелин Р., Петелин Ю. Аранжировка музыки на PC	272 с.
Петелин Р., Петелин Ю. Звуковая студия в PC	256 с.
Петелин Р., Петелин Ю. Музыка на PC. Cakewalk Pro Audio 9. Секреты мастерства	420 с.
Петелин Р., Петелин Ю. Музыка на PC. Cakewalk. "Примочки" и плагины	272 с.
Петелин Р., Петелин Ю. Музыкальный компьютер. Секреты мастерства	608 с.
Петелин Р., Петелин Ю. Персональный оркестр в PC	240 с.

Серия "Учебное пособие"

Бенькович Е. Практическое моделирование динамических систем (с компакт-диском)	464 с.
Гомоюнов К. Транзисторные цепи	240 с.
Дорот В. Толковый словарь современной компьютерной лексики, 2-е издание	512 с.
Культин Н. C/C++ в задачах и примерах	288 с.
Культин Н. Turbo Pascal в задачах и примерах	256 с.
Порев В. Компьютерная графика	432 с.
Робачевский Г. Операционная система Unix	528 с.
Сафронов И. Бейсик в задачах и примерах	224 с.
Солонина А. и др. Алгоритмы и процессоры цифровой обработки сигналов	464 с.
Солонина А. и др. Цифровые процессоры обработки сигналов фирмы MOTOROLA	512 с.
Угрюмов Е. Цифровая схемотехника	528 с.
Шелест В. Программирование	592 с.

Серия "Знакомьтесь"

Надеждин Н. Карманные компьютеры	304 с.
Надеждин Н. Портативные компьютеры	288 с.
Надеждин Н. Знакомьтесь, цифровые фотоаппараты	304 с.

Серия "Быстрый старт"

Васильева В. Персональный компьютер. Быстрый старт	480 с.
Гофман В., Хомоненко А. Delphi. Быстрый старт	288 с.
Дмитриева М. JavaScript. Быстрый старт	336 с.
Культин Н. Microsoft Excel. Быстрый старт	208 с.
Хомоненко А., Гридин. В. Microsoft Access. Быстрый старт	304 с.

Первые шаги в мире ИНФОРМАТИКИ

Программирование в среде Logo

Факультативный курс

Курс «Первые шаги в мире информатики» апробировался в течение 5 лет в школах Ленинградской области. В 2000 г. курс получил гриф: «Рекомендовано экспертным советом Комитета общего и профессионального образования Ленинградской области», имеет статус авторской программы.

Факультативный курс «Программирование в среде Logo» предназначен для проведения компьютерных уроков с использованием интегрированной среды языка Logo, содержит теоретический материал и задания для закрепления изученного материала. Наличие в каждом уроке дополнительного задания позволяет осуществить индивидуальный и дифференцированный подход к обучению. Рабочая тетрадь ученика содержит вкладыш с самостоятельными и контрольными работами для двух вариантов.

«Программирование в среде Logo» является отдельным модулем курса Тур С. Н., Бокучава Т. П. «Первые шаги в мире информатики» и предназначен для обучения информатике в 5—6 классах.