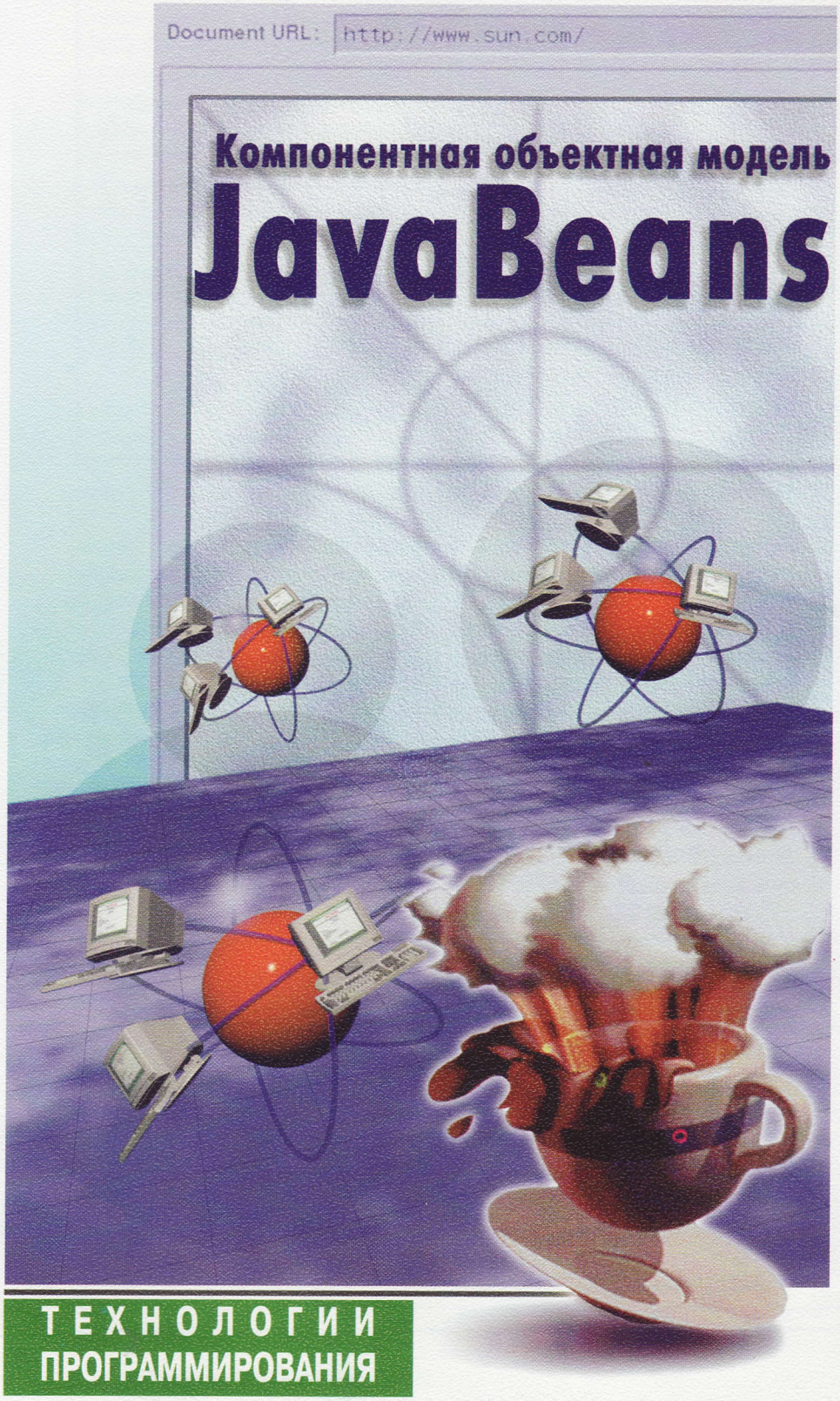
Jet Info

ИНФОРМАЦИОННЫЙ БЮЛЛЕТЕНЬ №19(50)/1997



ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Sun предлагает JavaEngine

26 августа 1997 г. на выставке Java Internet Business Expo отделение Sun Microelectronics компании Sun Microsystems обнародовало стратегию взаимодействия с производителями комплектного оборудования (OEM).

Комплексность – ключевой термин, характеризующий эту стратегию. Предполагается снабжать производителей всем необходимым для производства сетевых устройств в очень широком диапазоне: от смарт-карт и простых устройств для доступа в Интернет до Интернет-серверов и серверов для рабочих групп. Под "всем необходимым" понимается технологическая документация, аппаратные комплектующие и программное обеспечение. Такой подход позволяет компаниям-производителям создавать новые продукты, сочетая собственные разработки с готовыми решениями от Sun, концентрируя усилия на направлениях, придающих изделиям новые качества. Производители могут покупать микросхемы и проектировать собственные устройства и собственное программное обеспечение, но могут покупать и готовые платы.

В рамках реализации объявленной стратегии выпущен продукт JavaEngine 1, существующий в двух вариантах:

- готовая материнская плата;
- набор средств для проектирования.

На плате JavaEngine 1 (см. рис. 1) расположены процессор microSPARC-IIep (100 МГц), контроллер Ethernet (10/100 Мбит/с), графический контроллер, аудио-контроллер и оперативная память (от 8 до 64 Мб).

Программную часть продукта составляют операционная система JavaOS и настольное окружение HotJava Views, включающее Web-навигатор HotJava.

Заказчикам предоставляется также необходимое консультирование и обучение.

Платформа JavaEngine 1 ориентирована на три категории производителей:

- производителей персональных компьютеров, которые смогут наладить выпуск полного спектра оборудования для тонких клиентов;
- производителей UNIX-систем, которые с помощью JavaEngine 1 смогут внедрять тонкие клиенты в такие сегменты рынка, как образование, финансы и торговля;
- производителей специализированного оборудования, которые смогут создавать сетевые устройства, такие как интернет-телефоны или бездисковые станции для работы с текстом.

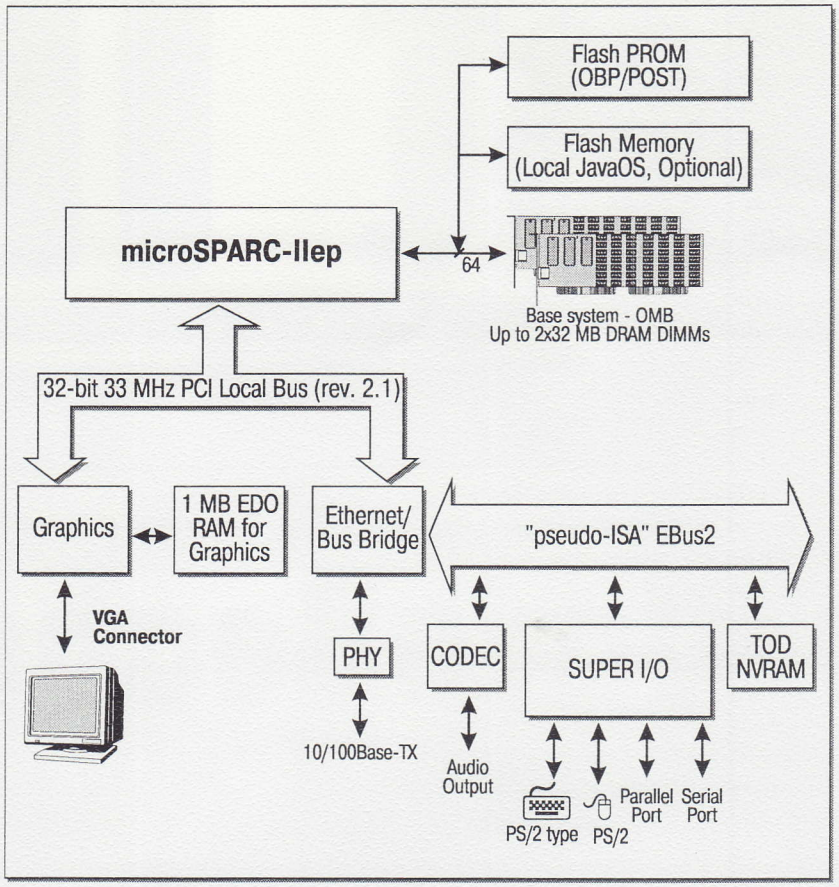


Рис. 1. Схема JavaEngine 1.

Отвечая на естественно возникающий вопрос: "Почему продукт JavaEngine 1 построен на microSPARC-IIep, а не на специализированных Java-процессорах?", президент Sun Microelectronics Чат Сильвестри сообщил, что microSPARC-IIep — это своего рода переходная модель, которая сегодня является идеальной платформой для миграции на Java-процессоры.

Новая стратегия работы с производителями распространяется и на компании, специализирующиеся на выпуске потребительских товаров. Для этой группы производителей предлагаются разработки и решения, ранее приобретенные вместе с компанией Diba. Сюда входят средства для обмена медицинской информацией с пациентами через Интернет (компания Healthway), телевизионные приставки для доступа в Интернет (компании Vestel и Teco), Web-телевидение для гостиниц (компания LodgeNet), системы для передачи справочной информации (компании CBS SportsLine, Publishers Clearinghouse, USA Today и другие).

Под солнцем военного здравоохранения

8 сентября 1997 года было объявлено, что подразделение Sun Microsystems Federal компании Sun Microsystems будет поставлять Unix-серверы и программное обеспечение к ним в рамках контракта с Министерством обороны США. Серверы Sun Microsystems предназначаются для военных медиков.

На сегодняшний день платформа SPARC/Solaris является одной из ведущих в области решений для медицины, включая обработку изображений и связь с Интернет. Понятно желание военных медиков вывести свои системы на передовой уровень, сделав ставку на передовые и, в то же время, проверенные продукты Sun Microsystems.

Согласно контракту, будут поставляться серверы UltraSPARC 2 и Ultra Enterprise 3000, операционная среда Solaris, сетевые средства, компиляторы и другие средства разработки. Заказан также ряд учебных курсов. В общем, омоложение серверной части военно-медицинских систем США будет носить комплексный характер.

На выставке
Java Internet Business Expo

Леонид Черняк

С 26 по 28 августа 1997 года в Нью-Йорке прошла первая выставка Java Internet Business Expo (JIBE). На ней побывало свыше 16 тысяч посетителей, она была высоко оценена профессиональной общественностью, компьютерной и бизнес-периодикой. В выставке и проходившей параллельно конференции приняли участие ведущие Java-игроки — компании Sun, IBM, Oracle, Netscape, Sybase, Marimba, Novell и другие. Выступления их руководителей (см. врезку) привлекли к себе всеобщее внимание.

Несомненный интерес вызвало сообщение компании Marimba о том, что она предложила консорциуму W3C (World Wide Web Consortium) новый протокол распространения и тиражирования информации DRP (Distribution and Replication Protocol). Этот протокол призван повысить производительность Интернет. Его поддержали компании Netscape, Novell, Sun Microsystems и другие.

Немало интересного было и на самой выставке.

Компания Sybase анонсировала два продукта, построенных на базе Java:

- инструментарий для разработки приложений PowerJ 2.0;
- программное обеспечение для управления транзакциями Jaguar CTS (Jaguar Component Transaction Server), поддерживающее спецификации Enterprise JavaBeans.

Еще два продукта представит в ближайшем будущем отделение Powersoft компании Sybase:

- инструментарий для разработки корпоративных Web-серверов PowerSite;
- новую версию PowerBuilder 6.0.

Компания GraphOn Corporation представила продукт GO-Joe — X-сервер, обеспечивающий доступ к Unix-приложениям для тонких клиентов, таких как JavaStation (Sun Microsystems). GraphOn использует трехуровневый архитектурный подход, где первый уровень — некоторое Unix-приложение, второй — работающий под Unix X-сервер GO-Joe, и третий — настольное устройство, которым может быть сетевой или персональный компьютер.

Продукт GO-Joe прошел бета-тестирование на нескольких тысячах настольных Java-системах, он лицензирован компанией Sun Microsystems и будет поставляться с каждой Java-станцией. Войдет он и в состав операционной среды Solaris.

Компания CST (Client/Server Technology) предложила семейство продуктов Jacada 5.0, предназначенных для построения клиентских Java-рабочих мест в корпоративных информационных системах. Jacada 5.0 позволяет автоматически, без какого-то либо предварительного знания Java, в течение нескольких часов создавать графические Java-клиенты. Это стало возможным благодаря использованию накопленной в CST базе знаний, которая содержит более 700 образцов прикладных интерфейсов.

Компания Sqribe Technologies представила средство генерации отчетов для хранилищ данных и баз данных — ReportMart, основанное на Java. Она продемонстрировала также продукт WebSqribe, который позволяет выполнять поиск отчетов по различным признакам с использованием навигатора.

Компания Finjan выпустила семейство продуктов Surfin для обеспечения информационной безопасности при работе с аплетами Java и ActiveX.

Из выступлений на конференции

Джим Баркдейл (Netscape Communications) заявил, что Интернет и 100% чистая Java сыграют ведущую роль в переосмыслении инвестиционной политики компаний в области программного и аппаратного обеспечения, они позволят выйти из бесконечной гонки по пути обновления вычислительных ресурсов. В начале 1998 года Netscape намеревается выпустить навигатор, построенный полностью на Java, который будет распространен не менее чем в 100 миллионах копий.

По мнению **Джона Томпсона (IBM)**, Java становится фундаментом для сетевых вычислений. Рост Web и быстрое развитие Java уже сегодня позволяют компаниям создавать корпоративные приложения на основе этих технологий.

Беатрис Инфант (Oracle) выразила уверенность, что независимость Java от типа платформы обеспечивает этому языку ключевую позицию при разработке компанией Oracle приложений клиент-сервер и поддержке ею спецификаций JavaBeans и Enterprise JavaBeans. Oracle планирует развивать корпоративные стандарты и интегрировать Java в свои СУБД.

Эрик Шмидт (Novell) высказал мнение, что на ближайшие 15 лет Java, вероятно, станет основным языком программирования. Развитие Интернет и принятие Java будет способствовать переходу компьютерной индустрии в новую фазу, характеризующуюся толстыми серверами, тонкими клиентами и интеллектальными сетями. В будущем Novell будет создавать продукты на базе Java. Первым из них станет новая версия IntranetWare.

Скотт МакНили (Sun Microsystems) утверждал, что Java-технологии начинают использоваться крупными компаниями. Так, Northern Telecom, Alcatel и Samsung пришли к соглашению об использовании PersonalJava при разработке устройств для Web-телефонии. МакНили сообщил о том, что IBM, Netscape и Sun создали совместный Java-центр (Java Porting and Tuning Center), функции которого будут состоять в повышении производительности Java и согласованном распространении версий Java. Центр, укомплектованный специалистами трех компаний, начнет свою деятельность на базе отделения JavaSoft компании Sun. Первым результатом работы Центра в четвертом квартале 1997 года должна стать настройка библиотеки классов JDK 1.1 с целью получения более высокой производительности. По планам, во втором квартале 1998 года должна быть выпущена новая версия JDK 1.2.

Мы перечислили лишь небольшую часть наиболее интересных продуктов, представленных на Java Internet Business Expo. Вероятно, под отчет о следующем подобном мероприятии придется выделять заметно больше места, поскольку Java-технология поддерживается очень большой армией разработчиков, выдающих "на гора" массу интересного.

Java для финансовых организаций

На выставке Financial Technology Expo (10-12 сентября 1997 года, Нью-Йорк) компания Sun Microsystems продемонстрировала ряд решений, ориентированных на организации, оказывающие финансовые услуги.

Компания представила собственные разработки, совместные проекты и продукцию партнеров. Они предназначены для использования в электронной коммерции, в хранилищах данных, в системах поддержки принятия решений, при интеграции программного обеспечения промежуточного слоя и традиционных систем, для дистанционного банковского обслуживания.

Ключевым элементом всей экспозиции является архитектура Sun Connect, опирающаяся на Java. Она служит основой для функционально полного и безопасного финансового обслуживания, использующего Web в качестве средства связи с пользователями.

В части экспозиции, получившей название "Финансовые организации будущего", Sun предлагает решения своих партнеров (см. табл. 1).

Партнерские решения используют Java-технологию и опираются на серверную платформу Ultra Enterprise.

Компания Sun Microsystems всегда занимала прочные позиции в финансовой сфере. Похоже, она не собирается останавливаться на достигнутом.

BEA Systems	Программное обеспечение промежуточного слоя, в том числе монитор транзакций TUXEDO и программное обеспечение для обработки транзакций в реальном времени.
BroadVision	Банковские Интернет- решения.
Chicago Board of Trade	Приложения для электронной торговли.
Fiserv	Решения для корпоративных хранилищ данных.
HNC Software	Системы поддержки принятия решений.
Innovision Corporation	Безопасная платформа для разработки Web-систем, включающая сервер OFX на основе Java.
Netsmart Technologies	Система электронных платежей SmartePay System, построенная на основе Java.
Oracle	Решения для дистанционного банковского обслуживания.
Sybase	Транзакционные системы.
Technology House	Системы обслуживания покупателей.
TriMark Technologies	Приложения для страховых компаний.

Табл. 1. Некоторые продукты партнеров Sun Microsystems, представленные на выставке Financial Technology Expo.

Открытые решения для встроенных систем

Компания Sun Microsystems сделала еще один шаг, расширяющий сферу ее деятельности. В пресс-релизе, опубликованном 10 сентября 1997 года, сообщается о заключении соглашения по поводу приобретения компании Chorus Systems, производящей ОС для телекоммуникационных систем и интеллектуальных устройств. По окончании процесса приобретения компания Chorus должна стать важной составной частью нового подразделения Sun — ESSG (Embedded Systems Software group). В ESSG войдет и коллектив, занимающийся разработкой и маркетингом JavaOS. Основной функцией ESSG станет разработка и производство открытого программного обеспечения для встроенных систем.

С приобретением Chorus Systems и созданием ESSG Sun становится компанией, способной предложить системное программное обеспечение для полного диапазона устройств - от телефонов до суперкомпьютеров. Цель Sun, согласно программе WebTone, состоит в том, чтобы любой человек, в любое время, из любого места, с помощью любого устройства мог подключиться к Интернет.

Дж. Хеберт, назначенный руководителем ESSG, видит свою задачу следующим образом: "Реализация программы WebTone не может быть обеспечена простым переносом операционных систем, принятых в настольных компьютерах, на компактные мобильные устройства. Мир встроенных систем более разнообразен, в нем требуются экономически оправданные, точные решения для самых разных устройств в диапазоне от мобильных ручных телефонов до больших телефонных коммутаторов".

ESSG начнет с производства ОС для телекоммуникационной инфраструктуры и потребительских электронных устройств. Позже производственная программа расширится встроенными системами реального времени для промышленной автоматики и систем управления технологическими процессами, для транспорта и медицины.

Консорциуму W3C

предложен протокол распространения и репликации данных

Компанией Marimba разработан протокол распространения и репликации данных (Distribution and Replication Protocol, DRP). Протокол DRP предлагается в качестве средства для оптимизации работы и повышения производительности сети Интернет. Особенность DRP заключается в том, что он содержит механизм обновления файлов, который исключает необходимость в повторной передаче по сети однажды полученной информации. Протокол обеспечивает ее актуализацию посредством передачи только изменений.

Идеи, заложенные в DRP, имеют много общего со свойствами продукта Castanet, выпускаемого компанией Marimba для корпоративных пользователей (см. рис. 1). С помощью двух составляющих — передатчика Castanet Transmitter на сервере и приемников Castanet Tuner на клиентских местах — автоматически выполняются распространение, установка, обновление и обслуживание программного обеспечения и данных.

Важность и своевременность протокола DRP подтверждается тем, что по мере роста числа распространяемых по сети материалов соответственно возрастает и частота их обновления. Существующие методы передачи данных оказываются неэффективными именно в части актуализации. Следовательно, они нуждаются в дополнении стандартизованным механизмом, предназначенным для обновления версий. Эту задачу и призван решить DRP.

DRP предлагает еще одну возможность оптимизации Интернет, а именно децентрализацию в процессе распространения данных. Если некоторые данные находятся одновременно на нескольких серверах, то тот сервер, к которому обращен запрос клиента, сможет информировать его о доступности запрашиваемой им информации на альтернативном сервере, расположенном ближе к клиенту.

Систему, использующую протокол DRP, можно представить себе следующим образом. Допустим, существует некоторая компания, распространяющая по подписке новостную или коммерческую информацию, располагающуюся на нескольких десятках HTML-страниц, каждая из которых содержит изображения и аплеты. Когда пользователь обращается к такому сервису сегодня, сервер должен определить, что изменилось с момента предыдущего запроса, а затем передать полноразмерные файлы. Это может потребовать установления нескольких сотен TCP-соединений. В итоге возрастают задержки в обслуживании и бесполезная загрузка сервера. Если же будет использоваться DRP, то пользователь мгновенно получит ссылку на файлы, которые требуют обновления, и вся загрузка сведется к одному TCP-соединению.

26 августа 1997 года спецификации протокола DRP (см. http://www.marimba.com/standards/drp.html) были совместно переданы в консорциум W3C (Worldwide Web Consortium) компаниями Marimba, Netscape Communications, Novell, Sun Microsystems и @Home Network.

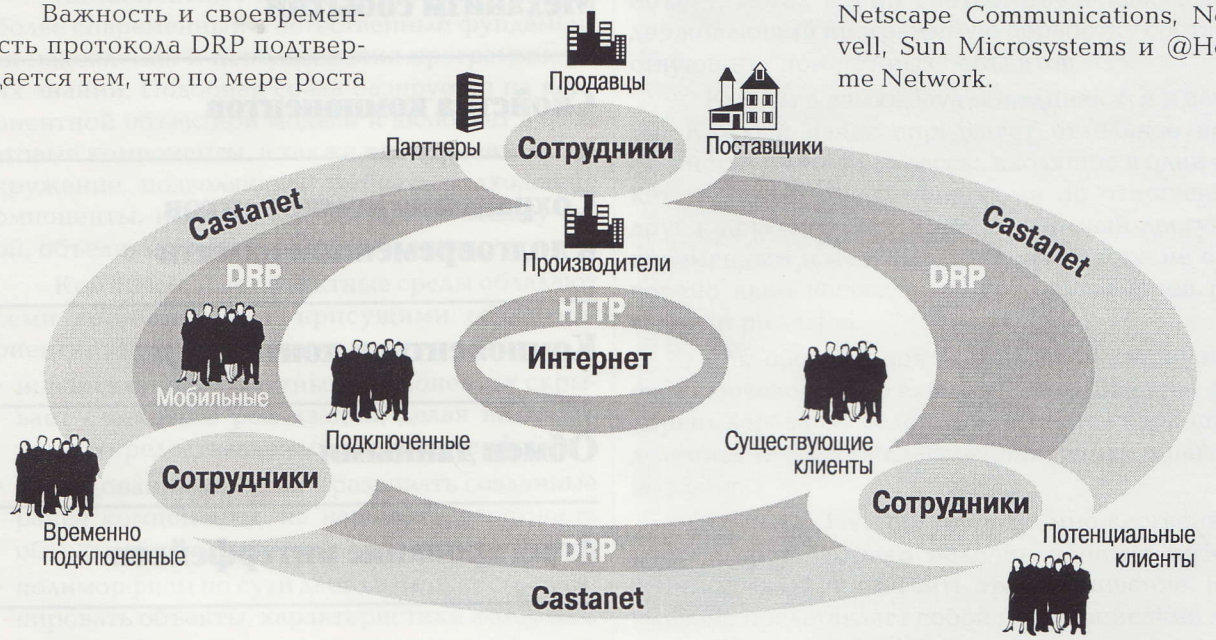


Рис. 1. Место протокола DRP в корпоративной сети.

Владимир Галатенко, Александр Таранов

Компонентная объектная модель JavaBeans

Коротко о языке Java

Основные понятия модели JavaBeans

Афиширование и выяснение интерфейсов

Механизм событий

Свойства компонентов

Сохранение компонентов в долговременной памяти

Компоненты и контейнеры

Обмен данными

Агрегирование интерфейсов

1. Введение

Накопление и использование программистских знаний — ключевой вопрос современных информационных технологий. Сейчас не имеет большого значения, насколько удобно в том или ином инструментальном окружении проектировать и реализовывать программы "с нуля" — в любом случае для большой информационной системы это будет сложный, многолетний процесс. Впрочем, как правило, это и не нужно. Очень многие вещи уже написаны (причем по многу раз), так что остается только настроить их должным образом и "проинтегрировать", то есть собрать в единую систему.

Если проводить аналогию со строительством, то в настоящее время наиболее перспективными представляются крупноблочные сборочные технологии. В программировании это направление развивается давно и под разными названиями — библиотеки стандартных программ, пакеты прикладных программ и т.п. Разработаны многочисленные "маги", реализующие фрагменты, специфичные для операционного окружения или предметной области.

Компонентные объектные среды — это наиболее современный и естественный фундамент для накопления и использования программистских знаний. Подобная среда базируется на компонентной объектной модели и включает в себя готовые компоненты, а также инструментальное окружение, позволяющее выбрать подходящие компоненты, настроить их и связать между собой, объединив в готовое приложение.

Компонентные объектные среды обладают всеми достоинствами, присущими объектно-ориентированному подходу:

- инкапсуляция объектных компонентов скрывает сложность реализации, делая видимым только предоставляемый вовне интерфейс;
- наследование позволяет развивать созданные ранее компоненты, не нарушая целостность объектной оболочки;
- полиморфизм по сути дает возможность группировать объекты, характеристики которых с некоторой точки зрения можно считать сходными.

Кроме того, объединение компонентов в рамках единого контейнера, с одной стороны, позволяет строить иерархии с небольшим числом сущностей на каждом уровне (это необходимое условие успешной работы с любой сложной системой), а, с другой стороны (и это очень важно с методологических позиций), предоставляет механизм образования новых понятий как во время разработки программной системы, так и в процессе ее использования.

JavaBeans — не единственная и не первая компонентная объектная среда, однако, учитывая огромную популярность Java-технологии, мы решили рассмотреть именно JavaBeans, сосредоточившись на компонентной объектной модели. Наше изложение основывается на версии спецификации JavaBeans [1] от 25 июля 1997 года. Используются также некоторые другие спецификации и их проекты [2, 3, 4, 5].

2. Коротко о языке Java

Мы позволим себе коротко напомнить читателям некоторые сведения о языке Java, которые понадобятся нам для дальнейшего изложения. Более полное описание языка и ассоциированной технологии можно найти, например, в статье [6].

Java — объектно-ориентированный язык. В его основе лежит понятие класса. Класс является шаблоном для создания объектов; он может содержать данные и методы. Существуют различные режимы доступа к элементам класса — private, protected, public.

Java — полностью объектно-ориентированный язык, каждому понятию которого (класс, объект, метод и т.п.) соответствует класс, поддерживающий программную обработку соответствующих "понятийных" объектов.

Классы в языке Java объединяются в пакеты. Каждый пакет определяет отдельное пространство имен. Все классы, входящие в один пакет, являются дружественными по отношению друг к другу, то есть имеют взаимный доступ к переменным и методам, если противное не оговорено явно посредством спецификаторов private или protected.

Для обозначения наследования используется ключевое слово extends. Класс Object — это корень дерева наследования. Имеется предопределенная иерархия классов, описанная в пакете java.lang.

В языке Java отсутствует множественное наследование, однако наличие понятия интерфейса позволяет смягчить это ограничение. Интерфейс представляет собой набор описаний методов. Классы могут реализовывать интерфей-

сы. Этот факт обозначается ключевым словом implements в заголовке класса.

Класс Class используется для получения во время выполнения информации о "классовых" свойствах объектов. Типичные методы этого класса — forName (получение объекта класса Class по текстовому имени), newInstance (порождение нового объекта данного класса), getMethods (получение массива объектов, описывающих public-методы класса, в том числе унаследованные).

Java-классы могут быть абстрактными, то есть не до конца конкретизированными. Это означает, что в классе описаны методы, определения которых отсутствуют. Такие методы (как и сам класс) должны снабжаться описателем abstract и конкретизироваться в производных классах.

Для обработки исключительных ситуаций, возникающих во время выполнения программы, в языке Java используется конструкция try/catch/finally. Для передачи информации об исключительной ситуации используются объекты классов — наследников класса Throwable.

Механизм потоков — обязательная черта современных операционных сред. В языке Java потоки представлены посредством класса Thread, интерфейса Runnable, спецификатора метода synchronized и методов класса Object wait и notify.

Java-программы подразделяются на два вида — самостоятельные приложения и аплеты. Последние выполняются в среде Web-навигатора и могут поступать по сети. Приложения и аплеты существенно различаются по мерам безопасности, принимаемым в процессе их работы (аплеты контролируются значительно жестче).

Java-компилятор транслирует исходные тексты программ в коды виртуальной Java-машины. Компилятор порождает файлы классов, содержащие интерпретируемые коды и дополнительную информацию, используемую на этапе выполнения. Спецификации Java-машины обеспечивают независимость скомпилированных программ от поддерживающей аппаратно-программной платформы.

3. Основные понятия модели JavaBeans

Среда JavaBeans является надстройкой над стандартной Java-технологией. Она наследует понятия и характеристики Java, такие как объектная ориентированность, многопотоковость, использование виртуальной машины, независимость от аппаратно-программной платформы, информационная безопасность и т.п. В JavaBeans нет ничего, не выразимого в терминах языка Java.

Основой среды JavaBeans является компонентная объектная модель, представляющая собой совокупность архитектуры и прикладных программных интерфейсов. Архитектуру образуют основные понятия и связи между ними. Прикладные программные интерфейсы характеризуют набор сервисов, предоставляемых элементами среды. Они описываются в терминах синтаксиса и семантики Java-классов и интерфейсов.

К числу основных понятий архитектуры JavaBeans относятся компоненты и контейнеры. Контейнеры могут включать в себя множество компонентов, образуя общий контекст взаимодействия с другими компонентами и с окружением. Контейнеры могут выступать в роли компонентов других контейнеров.

Неформально компонент ("кофейное зерно" — Java Bean) можно определить как многократно используемый программный объект, допускающий обработку в графическом инструментальном окружении и сохранение в долговременной памяти. С реализационной точки зрения компонент — это Java-класс и, возможно, набор ассоциированных дополнительных классов.

Каждый компонент предоставляет набор методов, доступных для вызова из других компонентов и/или контейнеров.

Компоненты могут обладать свойствами. Совокупность значений свойств определяет состояние компонента. Свойства могут быть доступны на чтение и/или запись посредством методов выборки и установки.

Компоненты могут порождать события (быть источниками событий), извещая о них другие компоненты, зарегистрировавшиеся в качестве подписчиков. Извещение (называемое также распространением события) заключается в вызове определенного метода объектов-подписчиков.

Типичным примером события является изменение свойств компонента. В общем случае компонент может предоставлять подписку на получение информации об изменении и на право запрещать изменение.

Методы, свойства и события образуют набор афишируемых характеристик компонента, то есть характеристик, доступных инструментальному окружению и другим компонентам. Этот набор может быть выяснен посредством механизма интроспекции.

Состояние компонентов может быть сохранено в долговременной памяти. Наличие методов для подобного сохранения выделяет компоненты JavaBeans среди произвольных Java-классов.

Компоненты JavaBeans могут упаковываться для более эффективного хранения и передачи по сети. Описание соответствующего формата является частью спецификаций JavaBeans.

Жизненный цикл компонентов JavaBeans можно подразделить на три этапа:

- разработка и реализация компонента;
- сборка приложения из компонентов;
- выполнение приложения.

Разработка и реализация компонентов JavaBeans по сути не отличается от создания произвольных Java-объектов, хотя и может включать реализацию специфических методов.

Сборка приложений выполняется, как правило, в инструментальном окружении, позволяющем проанализировать афишируемые характеристики компонентов, настроить значения свойств, зарегистрировать подписку на получение событий, организовав тем самым взаимодействие компонентов. Разработчик компонента может реализовать специальные методы для использования исключительно в инструментальном окружении (например, редактор свойств).

Компоненты взаимодействуют между собой и с инструментальным окружением. Взаимодействие осуществляется двумя способами — вызывом методов и распространением событий.

Спецификации JavaBeans описывают только локальное взаимодействие компонентов, осуществляемое в пределах одной виртуальной Java-машины. (Напомним, впрочем, что Java-апеты рассчитаны на передачу по сети, так что возможно собрать приложение из компонентов, первоначально распределенных по сети.) Удаленные объекты могут связываться по протоколам архитектуры CORBA [7], с помощью удаленного вызова методов (Remote Method Invocation — RMI) или иными способами, не относящимися к области действия спецификации JavaBeans (рис. 1).

Далее мы подробно рассмотрим, как описанные основные понятия реализуются в среде JavaBeans.

4. Афиширование и выяснение интерфейсов

В среде JavaBeans существуют способы динамического (то есть не по исходным текстам) выяснения характеристик компонентов. К таким характеристикам относятся:

- методы, доступные для вызова другими компонентами и инструментальным окружением;
- свойства, которые можно опрашивать и/или изменять;
- события, порождаемые данным компонентом.

Подобное выяснение в терминологии JavaBeans называется интроспекцией*.

Интроспекция используется прежде всего на этапе разработки, в рамках инструментального окружения, позволяя увидеть афишируемые характеристики и с их помощью настроить компонент и связать его с другими элементами приложения.

Принципиальная возможность интроспекции была изначально заложена в Java-технологии. Файлы классов содержат достаточно информации для выяснения всех необходимых характеристик объектов. Воспользоваться этой информацией можно с помощью класса Class, па-

*** – Перед нами пример неудачного англоязычного термина, когда лучше оставить не совсем русский, но не вызывающий ложных ассоциаций "научный" перевод. Introspection – это самонаблюдение, самоанализ; в спецификациях JavaBeans, однако, данный термин используется для обозначения процесса выяснения свойств компонента внешними субъектами (обычно инструментальным окружением). Возможно, перевод "просвечивание" был бы более удачным, чем интроспекция.**

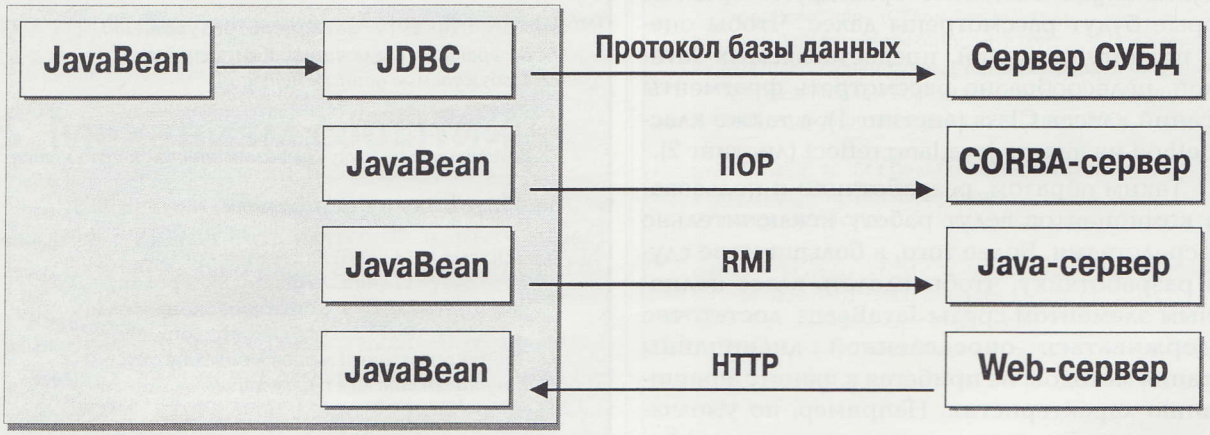


Рис. 1. Нелокальное взаимодействие компонентов.

```
public final class Class extends Object
   implements Serializable {
   // Информация о классе или интерфейсе

   public native Class getSuperclass ();
   // Возвращает класс-предшественник данного класса

   public native Class [] getInterfaces ();
   // Возвращает интерфейсы, реализуемые классом

   public Field [] getFields ()
       throws SecurityException;
   // Возвращает public-поля данного класса, в том числе
   // унаследованные. Возбуждает исключительную ситуацию,
   //  если доступ к этой информации запрещен политикой
   // безопасности.

   public Constructor [] getConstructors ()
       throws SecurityException;
   // Возвращает public-конструкторы данного класса.
   // Возбуждает исключительную ситуацию, если доступ
   // к этой информации запрещен политикой безопасности.

   public Method [] getMethods ()
       throws SecurityException;
   // Возвращает public-методы данного класса, в том числе
   // унаследованные. Возбуждает исключительную ситуацию,
   //  если доступ к этой информации запрещен политикой
   // безопасности.

   public Method getMethod (String name, Class
                                 parameterTypes [])
       throws NoSuchMethodException,
              SecurityException;
   // Возвращает public-метод с заданными именем и типами
   // параметров. Возбуждает исключительную ситуацию,
   // если такого метода нет, или если доступ к этой
   // информации запрещен политикой безопасности

   public Method [] getDeclaredMethods ()
       throws SecurityException;
   // Возвращает все методы, продекларированные в данном
   // классе. Возбуждает исключительную ситуацию, если
   // доступ к этой информации запрещен политикой
   // безопасности
   . . .
}
```

Листинг 1.

кета java.lang.reflect и некоторых других средств, которые будут рассмотрены далее. Чтобы оценить полноту сведений, предоставляемых Java-средой, целесообразно рассмотреть фрагменты описаний класса Class (листинг 1), а также класса Method из пакета java.lang.reflect (листинг 2).

Таким образом, разработчики и пользователи компонентов ведут работу исключительно Java-средствами. Более того, в большинстве случаев разработчику, чтобы сделать класс полноценным элементом среды JavaBeans, достаточно придерживаться определенной дисциплины описания методов, не прибегая к явному афишированию характеристик. Например, по умолчанию в число афишируемых попадают все public-методы компонента.

Способность компонента среды JavaBeans по существу без дополнительных усилий со стороны разработчика предоставлять информацию о своем интерфейсе называется рефлексией. Рефлексия базируется на дисциплине определения методов. Эта дисциплина состоит в следовании заданным шаблонам при выборе имен методов, а также типов формальных параметров и результатов. Далее, по ходу изложения, мы будем приводить эти шаблоны.

Среда JavaBeans проектировалась таким образом, чтобы в типичных случаях и действия разработчиков, и внутренняя реализация оставались простыми; дополнительные усилия и утяжеление компонентов должны требоваться только тогда, когда этого желает сам программист. В данном случае, если разработчику не хватает выразительной силы рефлексии, он может реализовать интерфейс BeanInfo, явным образом описывающий афишируемые характеристики компонента.

Интерфейс BeanInfo содержит методы, позволяющие получить объекты-описатели характеристик компонента. В число этих методов входят getBeanDescriptor, getMethodDescriptors и т.д. (см. листинги 3 и 4). Поскольку реализация

```
public final class Method
   extends Object implements Member {
// Информация о методе класса или интерфейса

   public Class getDeclaringClass ();
   // Возвращает класс или интерфейс,
   // содержащий декларацию данного метода

   public String getName ();
   // Возвращает имя метода в виде цепочки символов

   public native int getModifiers ();
   // Возвращает модификаторы (public, ...),
   // использованные при описании метода

   public Class [] getParameterTypes ();
   // Возвращает типы формальных параметров метода

   public Class getReturnType ();
   // Возвращает тип результата метода

   public Class [] getExceptionTypes ();
   // Возвращает исключительные ситуации,
   // возбуждаемые данным методом

   public String toString ();
   // Возвращает цепочку символов, описывающую метод

   public native Object invoke (Object obj,
                                Object args [])
       throws  IllegalAccessException,
               IllegalArgumentException,
               InvocationTargetException,
               NullPointerException;
   // Применяет данный метод к объекту obj
   // с заданным списком параметров
   . . .
}
```

Листинг 2.

методов может быть сколь угодно изощренной, у разработчика появляется возможность ассоциировать с компонентом ресурсы (например, файлы), содержащие описательную информацию. Класс SimpleBeanInfo, входящий в пакет java.beans, является "пустой" реализацией интерфейса BeanInfo, отрицающей наличие у компонента каких-либо афишируемых методов, свойств и событий. Разработчик может создать производный класс и выборочно переопределить методы класса SimpleBeanInfo.

Класс Introspector реализует процесс интроспекции. По заданному компоненту он конструирует объект класса BeanInfo (см. листинг 5). Действует Introspector следующим образом. Сначала он пытается найти класс, имя которого получается из имени класса компонента приписыванием текста "BeanInfo". Если такой класс находится, а его методы возвращают непустые дескрипторы, соответствующая информация используется при конструировании результирующего объекта BeanInfo. В противном случае Introspector полагается на механизм рефлексии и анализирует имена и типы параметров public-методов класса компонента и его предшественников.

Характерная особенность Java-технологии состоит в наличии стройной модели безопасности. Применительно к интроспекции действуют два защитных рубежа:

- Получение объектов Field, Method и Constructor возможно только путем применения методов класса Class, которые вызывают системный менеджер безопасности.
- Доступ к полям и методам объектов производится с применением стандартных правил языка Java.

Кроме того, аплеты подвергаются дополнительному контролю.

Детальный анализ модели безопасности Java выходит за рамки данной статьи. Здесь мы отметим лишь, что компонентная объектная среда не привносит каких-либо новых, специфических угроз, поскольку она полностью описывается в терминах языка Java.

5. Механизм событий

Согласно спецификациям JavaBeans, у каждого события есть источник и, быть может, один или несколько подписчиков (получателей).

Источник обязан:

- выбрать имя метода, вызываемого в компонентах-подписчиках при распространении события. Этот метод должен содержаться в интерфейсе, который является расширением интерфейса EventListener (данное расшире-

```
public class BeanDescriptor
   extends FeatureDescriptor {
   // Описатель компонента объектной среды

   public BeanDescriptor (Class beanClass);
   // Создает описатель по классу компонента

   public Class getBeanClass ();
   // Возвращает класс компонента

   public Class getCustomizerClass ();
   // Возвращает класс-настройщик компонента
   // (см. раздел "Настройка свойств")

   . . .
}
```

Листинг 3.

```
public interface BeanInfo {
// Интерфейс, который нужно реализовать,
// чтобы явным образом афишировать характеристики
// компонента объектной среды

   public abstract BeanDescriptor
       getBeanDescriptor ();
   // Возвращает описатель компонента

   public abstract EventSetDescriptor []
       getEventSetDescriptors ();
   // Возвращает описатели событий,
   // возбуждаемых компонентом

   public abstract PropertyDescriptor []
       getPropertyDescriptors ();
   // Возвращает описатели афишируемых
   // свойств компонента

   public abstract MethodDescriptor []
       getMethodDescriptors ();
   // Возвращает описатели афишируемых
   // методов компонента

   public abstract Image getIcon (int iconKind);
   // Возвращает иконку, ассоциированную с компонентом

   . . .
}
```

Листинг 4.

```
public class Introspector extends Object {
// Выяснение характеристик компонента объектной среды

   public static BeanInfo
       getBeanInfo (Class beanClass)
   throws IntrospectionException;
   // Выясняет афишируемые характеристики компонента.
   // При нештатном ходе процесса интроспекции
   // возбуждает исключительную ситуацию

   public static String [] getBeanInfoSearchPath ();
   // Возвращает массив пакетов,
   // в которых будут разыскиваться классы BeanInfo

   public static void
       setBeanInfoSearchPath (String path []);
   // Устанавливает массив пакетов,
   // в которых будут разыскиваться классы BeanInfo

   . . .
}
```

Листинг 5.

ние мы будем называть интерфейсом события);

- реализовать метод регистрации подписчиков события и метод аннулирования регистрации;
- при распространении события вызвать метод, описанный в интерфейсе события, во всех компонентах-подписчиках.

В свою очередь, подписчик должен предпринять следующие действия:

- выполнить реализацию интерфейса события, то есть по сути реализовать метод обработки события (напомним, что имя этого метода выбрал источник);
- зарегистрироваться в качестве подписчика события.

Рассмотрим перечисленные шаги более подробно.

5.1. Действия, выполняемые источником события

Источник события по своему выбору назначает имя метода, вызываемого в компонентах-подписчиках при распространении события. Чтобы сделать возможной автоматическую интроспекцию компонентов на предмет распространяемых ими событий (то есть для поддержки рефлексии), данный метод описывается в расширении пустого интерфейса java.util.EventListener, играющего роль этикетки. Пример расширения приведен на листинге 6.

По соглашению, опять-таки направленному на поддержку рефлексии, имя интерфейса-расширения должно оканчиваться цепочкой символов "Listener".

```
interface KeyPressedListener
   extends java.util.EventListener {

   void KeyPressed (KeyPressedEvent kpe);
   // Метод, вызываемый в подписчиках
   // при распространении события

}
```

Листинг 6.

```
public class EventObject extends Object
   implements Serializable {

   protected transient Object source;
   // Поле событийного объекта, хранящее информацию
   // об источнике. Слово transient означает, что поле
   // является временным и при сохранении объекта
   // в долговременную память не записывается

   public EventObject (Object source);
   public Object getSource ();
   public String toString ();
   // Возвращает представление событийного объекта
   // в виде цепочки символов
}
```

Листинг 7.

```
public class KeyPressedEvent
   extends java.util.EventObject {

      protected transient int KeyCode;

      KeyPressedEvent (java.awt.Component source,
                                    int Key) {
         super (source);
         KeyCode = Key;
      }

      public int getKeyPressed () {
         return KeyCode;
      }
}
```

Листинг 8.

```
public abstract class KeyPressedEventSource {

   private Vector listeners = new Vector ();
   // Массив для хранения набора подписчиков

   public synchronized void
    addKeyPressedListener (KeyPressedListener kpl) {
   // Зарегистрировать подписчика

      listeners.addElement (kpl);
   }

   public synchronized void
    removeKeyPressedListener (KeyPressedListener kpl) {
   // Аннулировать регистрацию

      listeners.removeElement (kpl);
   }

   protected fireKeyPressed (int Key) {
   // Распространение события (оповещение подписчиков)

      Vector l;
      KeyPressedEvent kpe =
         new KeyPressedEvent (this, Key);

   // Создадим локальную копию набора подписчиков
   // на момент возникновения события.
   // В процессе распространения события набор подписчиков
   // (но не локальная копия!) может изменяться
      synchronized (this)
         {l = (Vector) listeners.clone ();}

   // Оповестим подписчиков о наступлении события

      for (int i = 0; i < l.size(); i++) {
         ((KeyPressedListener)
            l.elementAt(i)).KeyPressed (kpe);
      }
   }
}
```

Листинг 9.

Если источник желает распространять несколько различных событий, допускается описание в одном интерфейсе соответствующего числа методов их обработки.

Метод обработки события должен иметь один аргумент, которым является так называемый событийный объект — преемник класса java.util. EventObject. Посредством этого объекта

```
public class MyListener
   implements KeyPressedListener {

      public void KeyPressed (KeyPressedEvent kpe) {
         . . .
      }

}
```

Листинг 10.

подписчику передается информация об источнике и другие характеристики события. Определение класса EventObject приведено на листинге 7. Листинг 8 содержит возможное описание класса KeyPressedEvent (окончание "Event" — еще одно требование рефлексии).

Реализация методов регистрации подписчиков, аннулирования регистрации, а также собственно распространения события может быть выполнена способом, приведенным на листинге 9. Отметим, что при распространении события вызов методов подписчиков производится синхронным образом, в рамках потока источника события.

Обратим внимание на два аспекта программного текста, приведенного на листинге 9. Во-первых, в источнике необходимо обеспечить безопасность работы в многопотоковой среде. Методы add/remove выполняются в рамках потоков подписчиков, поэтому они нуждаются в синхронизации. В методе fire также следует учитывать возможность регистрационных действий параллельно с распространением события. Отсюда три вхождения ключевого слова synchronized.

Во-вторых, регистрационные методы должны поддерживать рефлексию и определяться по следующим шаблонам:

```
public void add<имя интерфейса события>
      (<имя интерфейса события> подписчик);
public void remove<имя интерфейса события>
      (<имя интерфейса события> подписчик);
```

Определение метода fire — внутреннее дело источника события.

5.2. Действия, выполняемые подписчиком события

Реализация интерфейса события — основное действие, выполняемое подписчиком. Его содержательная сторона зависит от специфики подписчика. Чисто технические моменты отражены на листинге 10.

Регистрация подписки производится обращением к соответствующему add-методу источника события.

Общая схема взаимодействия источника и подписчиков события представлена на рис. 2. Если проводить аналогию с обычной подпиской на газеты и журналы, то базовый механизм, описанный в спецификациях JavaBeans, соответствует оформлению подписки в редакции каждого издания (а не в отделении связи). В спецификациях упоминается также о возможности реализации адаптеров — посредников, берущих на себя централизованное оформление подписки и реализацию определенной дисциплины распространения событий. При наличии адаптера схема взаимодействия источников и подписчиков событий может выглядеть так, как показано на рис. 3. Очевидно, подобная схема облегчает жизнь всем взаимодействующим сторонам (не считая адаптера, который необходимо реализовать).

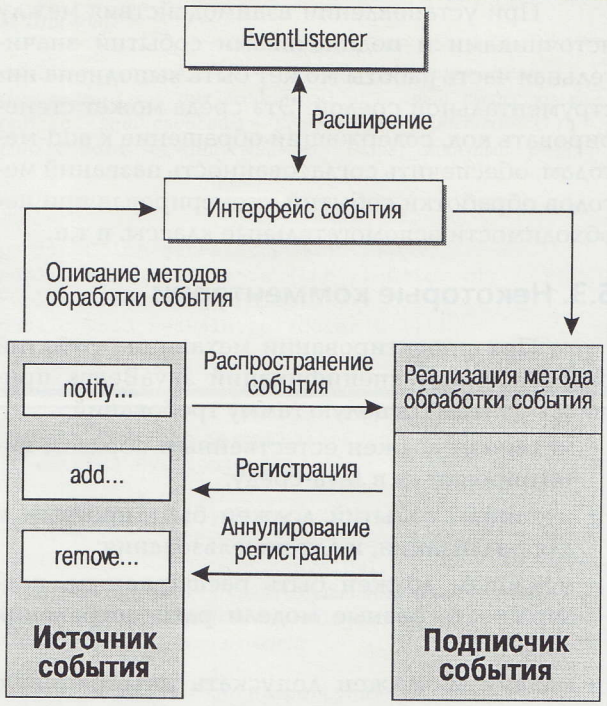


Рис. 2. Схема взаимодействия источника и подписчиков события.

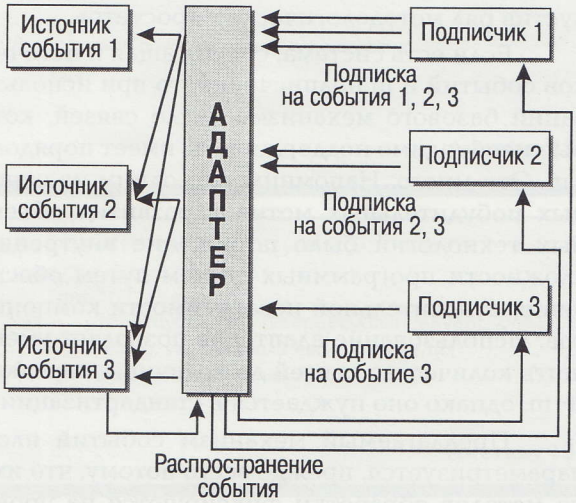


Рис. 3. Схема взаимодействия источников и подписчиков событий при наличии адаптера.

При установлении взаимодействия между источниками и подписчиками событий значительная часть работы может быть выполнена инструментальной средой. Эта среда может сгенерировать код, содержащий обращение к add-методам, обеспечить согласованность названий методов обработки событий, сгенерировав при необходимости вспомогательные классы, и т.п.

5.3. Некоторые комментарии

При проектировании механизма событий разработчикам спецификаций JavaBeans пришлось учитывать целую гамму требований:

- механизм должен естественным образом интегрироваться в Java-среду;
- механизм событий должен быть простым и для реализации, и для использования;
- механизм должен быть расширяемым, поддерживать разные модели распространения событий;
- механизм должен допускать интеграцию с другими компонентными средами.

Таким образом, задача перед разработчиками стояла трудная, и, на наш взгляд, они не сумели найти достаточно хорошего решения, допустив ряд методологических просчетов.

Если есть система, содержащая n источников событий и m подписчиков, то при использовании базового механизма число связей, которые необходимо поддерживать, имеет порядок n * m. Это много. Напомним, что одним из основных побудительных мотивов развития объектных технологий было понижение внутренней сложности программных систем путем обеспечения относительной независимости компонентов. Использование адаптеров позволяет уменьшить количество связей до величины порядка n + m, однако оно нуждается в стандартизации.

Предлагаемый механизм событий плохо параметризуется, прежде всего потому, что имена методов обработки фиксируются на уровне исходных текстов. Из-за этого трудно писать программы, систематически обрабатывающие различные события. Приходится как минимум наращивать объем программ, что вступает в противоречие с основными положениями Java-технологии.

Пространство событий не структурировано, хотя некоторые действия в некоторых частях среды Java (например, в абстрактном оконнном инструментарии, AWT) в этом направлении предприняты. Возможность иерархической организации элементов — одно из необходимых условий успешного создания и сопровождения сложных систем. Здесь это условие нарушено (конечно же, иерархии классов в данном случае недостаточно).

Складывается впечатление, что в предлагаемом виде механизм событий является слишком простым, в нем не хватает ряда понятий. По существу это способ диспетчеризации вызовов методов. Какая-либо "событийность" в такой трактовке попросту отсутствует. Обычно события возникают с частотой, определяемой факторами, внешними по отношению к программной системе, так что необходимо минимизировать по крайней мере время распространения события. При предлагаемом синхронном вызове методов оценить время распространения не представляется возможным. Более того, необходимо позаботиться о противодействии атакам на доступность, производимым путем регистрации подписчика, метод обработки которого не возвращает управления. Часть этих проблем можно решить, реализовав интеллектуальный многопотоковый адаптер, но тогда придется отдельно решать вопросы безопасности, так как ряд защитных мер в Java-технологии основан на прослеживании стека вызовов, и опасность может грозить со стороны зловредного источника событий.

При подготовке спецификаций JavaBeans пришлось пересмотреть механизм событий, первоначально предусматривавшийся абстрактным оконным инструментарием. Весьма возможно, что и версия 1.01 спецификаций JavaBeans не является в этом смысле окончательной. Впрочем, для систем небольшого и среднего масштаба она представляется вполне удовлетворительной.

6. Свойства компонентов

Свойства компонентов (такие как цвет изображения, размер и т.п.) можно представлять себе в виде полей объектов; в большинстве случаев так они и реализуются. Тем не менее, спецификации JavaBeans, в соответствии с последовательным объектно-ориентированным подходом, предусматривают доступ к свойствам только посредством специальных методов выборки и установки, так что в принципе свойства могут быть устроены сколь угодно сложным образом.

Между свойствами разных компонентов могут существовать связи двоякого рода. Во-первых, при изменении одного свойства может потребоваться модификация других свойств или иных характеристик. Такие свойства называются связанными. Во-вторых, некоторые изменения могут трактоваться как некорректные и запрещаться. Соответствующие свойства называются ограниченными. Тем самым спецификации JavaBeans предоставляют средства контроля целостности объектной среды.

Свойства компонентов могут подвергаться настройке. Обычно настройка производится в инструментальном окружении при сборке при-

ложений, однако, вообще говоря, она может осуществляться и во время выполнения.

Перейдем к детальному рассмотрению заявленных тем.

6.1. Методы и события, ассоциированные со свойствами

Свойства компонентов могут быть скалярными и индексируемыми. Выборка и установка скалярных свойств осуществляется с помощью методов get/set:

```
public <тип_свойства> get<имя_свойства> ();
public void set<имя_свойства> (<тип_свойства> p);
```

(такой шаблон заголовков методов необходим для поддержки рефлексии).

Например, если свойством компонента является цвет, соответствующие методы могут описываться как

```
public Color getColor ();
public void setColor (Color c);
```

(класс `Color` *определяется в пакете* `java.awt`*).*

Особый шаблон предусмотрен для выборки булевых свойств:

```
public boolean is<имя_свойства> ();
```

Вообще говоря, свойства могут быть доступны только на чтение или только на запись; тогда для них определяется лишь один из методов — get или set соответственно.

Индексируемые свойства образуют массивы с целочисленными индексами. С этими массивами можно работать покомпонентно или как с единым целым. Шаблоны соответствующих методов и их примеры приведены на листинге 11 (под <типом_свойства> здесь понимается тип элемента массива).

Спецификации JavaBeans предусматривают наличие связанных свойств, после изменения которых возбуждается событие propertyChange. Другие компоненты могут подписаться на это событие и, следовательно, получать информацию о производимых изменениях, анализируя объект-параметр PropertyChangeEvent, фрагмент описания которого приведен на листинге 12.

Метод propertyChange, вызываемый для обработки изменения значения свойства, описан в интерфейсе PropertyChangeListener (листинг 13). Источник события, в соответствии с общими правилами (см. раздел "Механизм событий"), должен реализовать методы addPropertyChangeListener и removePropertyChangeListener, обеспечивая регистрацию подписчиков.

Вспомогательный класс PropertyChangeSupport, входящий в пакет java.beans, реализует рутинные действия, характерные для обслуживания связанных свойств (см. листинг 14). Естественно, разработчики компонентов могут воспользоваться этим классом.

```
// Шаблоны

public <тип_свойства> get<имя_свойства> (int i);
public void set<имя_свойства> (int i, <тип_свойства> p);
public <тип_свойства> [] get<имя_свойства> ();
public void set<имя_свойства> (<тип_свойства> p []);

// Примеры

public Color getPalette (int i);
public void setPalette (int i, Color c);
public Color [] getPalette ();
public void setPalette (Color c []);
```

Листинг 11.

```
public class PropertyChangeEvent
   extends EventObject {

   public PropertyChangeEvent (Object source,
                String propertyName, Object oldValue,
                Object newValue);
   // Конструктор. Создает событийный объект из источника
   // события, имени изменяемого свойства,
   // старого и нового значений

   public String getPropertyName ();
   // Возвращает имя изменяемого свойства

   public Object getNewValue ();
   // Возвращает новое значение свойства

public Object getOldValue ();
// Возвращает прежнее значение свойства

. . .
}
```

Листинг 12.

```
public interface PropertyChangeListener
   extends EventListener {
      public abstract void
         propertyChange (PropertyChangeEvent pce);
      // Метод, вызываемый после изменения
      // связанного свойства

}
```

Листинг 13.

Помимо связанных, спецификации JavaBeans описывают ограниченные свойства, перед изменением значений которых распространяется событие vetoableChange с параметром PropertyChangeEvent. Подписчики этого события могут отклонить планируемое изменение, возбудив исключительную ситуацию PropertyVetoException. Метод set должен отреагировать на подобное вето, вернув прежнее значение, "извинившись" перед уже оповещенными подписчиками (то есть вызвав их методы vetoableChange с обратной парой новое/старое значение) и передав исключительную ситуацию инициатору изменения. Соответственно, заголовок set-метода для

```
public class PropertyChangeSupport
   extends Object implements Serializable {
// Вспомогательный класс для обслуживания
// связанных свойств

   public PropertyChangeSupport (Object sourceBean);
   // Конструктор

   public synchronized void
   addPropertyChangeListener
       (PropertyChangeListener pcl);
   // Регистрация подписчиков

   public synchronized void
       removePropertyChangeListener
           (PropertyChangeListener pcl);
   // Аннулирование регистрации

   public void firePropertyChange
       (String propertyName, Object oldValue,
        Object newValue);
   // Конструирование событийного объекта и
   // распространение события. Если новое и старое значения
   // совпадают, никаких действий не предпринимается

}
```

Листинг 14.

ограниченных свойств приобретает следующий вид:

```
public void set<имя_свойства> (<тип_свойства> p)
      throws PropertyVetoException;
```

Синтаксически связанные и ограниченные свойства аналогичны, но реализация последних требует гораздо большей аккуратности и от источников (set-методов), и от подписчиков события vetoableChange. Источнику рекомендуется воспользоваться вспомогательным классом java.beans.VetoableChangeSupport, аккуратно выполняющим все необходимые действия. Подписчикам будет проще, если сделать свойство и ограниченным, и связанным. В таком случае до изменения (при обработке события vetoable-Change) подписчики заботятся только о голосовании "за" и "против", а после изменения (при обработке события propertyChange) они выясняют, каким же стало новое значение.

6.2. Настройка свойств

Компонент объектной среды особенно полезен тогда, когда его можно настроить. Обычно настройка выполняется во время сборки приложения в инструментальном окружении.

Для несложных компонентов можно представить себе схему настройки, при которой окружение путем интроспекции выявляет афишируемые свойства и порождает электронный бланк, каждая клетка которого соответствует одному свойству. Редактирование содержимого клетки выполняется соответствующим редактором свойства. Редакторы свойств, принадлежащих стандартным типам, входят в Java-окруже-

```
public interface PropertyEditor {

   public abstract void setValue (Object value);
   // Устанавливает редактируемый объект (свойство)

   public abstract Object getValue ();
   // Возвращает текущее значение свойства

   public abstract boolean isPaintable ();
   // Истина, если свойство имеет графическое представление
   // (реализован метод paintValue)

   public abstract void paintValue (Graphics gfx,
                                    Rectangle box);
   // Отрисовывает графическое представление свойства
   // в заданной области экрана

   public abstract String getAsText ();
   // Возвращает текстовой представление значения
   // свойства, доступное для редактирования

   public abstract void setAsText (String text)
       throws IllegalArgumentException;
   // Устанавливает значение свойства
   // по текстовому представлению

   public abstract boolean supportsCustomEditor ();
   // Истина, если поддерживается специализированный
   // редактор свойства

   public abstract Component getCustomEditor ();
   // Возвращает специализированный редактор свойства,
   // которым, вероятно, воспользуется окружение

   public abstract void addPropertyChangeListener
       (PropertyChangeListener pcl);
   // Регистрация подписчиков, информируемых
   // об изменении значения свойства

   public abstract void removePropertyChangeListener
       (PropertyChangeListener pcl);
   // Аннулирование регистрации

   . . .
}
```

Листинг 15.

ние; в более сложных случаях должен существовать специализированный редактор, поставляемый с компонентом или с инструментальным окружением.

Класс-редактор свойства должен реализовывать интерфейс PropertyEditor (см. листинг 15). Обязательными для реализации являются метод setValue () и один из методов прорисовки и редактирования свойства — в графическом или текстовом представлении. По общим правилам, при изменении значения свойства редактор должен возбуждать событие propertyChange. Отсюда необходимость в реализации методов add/remove.

Для установления ассоциаций между типами данных и их редакторами служит класс PropertyEditorManager. Он поддерживает каталог зарегистрированных редакторов; если же явная регистрация отсутствует, PropertyEditor-

```
public class PropertyEditorManager extends Object {

 public static void registerEditor (Class targetType,
                        Class editorClass);
 // Регистрация редактора для типа targetType

 public static PropertyEditor findEditor
   (Class targetType);
// Возвращает редактор для типа targetType

   . . .
}
```

Листинг 16.

```
public interface Customizer {

   public abstract void setObject (Object bean);
   // Устанавливает настраиваемый объект

   public abstract void addPropertyChangeListener
       (PropertyChangeListener pcl);
   // Настройщик должен возбуждать событие
   // propertyChange при изменении значения свойства

   public abstract void removePropertyChangeListener
       PropertyChangeListener pcl);
}
```

Листинг 17.

Manager пытается отыскать класс, имя которого образовано приписыванием к имени типа текста "Editor" (см. листинг 16).

Для настройки сложных компонентов с большим числом специфических свойств может потребоваться специализированный класс, облегчающий действия пользователей по сравнению со стандартным редактированием бланка. Такой класс-настройщик должен быть прямым или косвенным преемником класса java.awt.Component, одновременно реализуя интерфейс java.beans. Customizer (описание последнего приведено на листинге 17).

Чтобы известить окружение о наличии настройщика, компонент должен предоставлять класс BeanInfo и, в частности, реализовывать метод getCustomizerClass класса BeanDescriptor (см. раздел "Афиширование и выяснение интерфейсов").

```
public interface Serializable {

   private void writeObject
       (java.io.ObjectOutputStream out)
           throws IOException;
   // Запись объекта в долговременную память

   private void readObject
       (java.io.ObjectInputStream in)
           throws IOException,
                  ClassNotFoundException;
   // Чтение объекта из долговременной памяти — метод,
   // обратный writeObject ()

}
```

Листинг 18.

На наш взгляд, механизм свойств специфицирован в среде JavaBeans весьма квалифицированно. В простых ситуациях разработчику компонента не нужно делать ничего; в сложных же случаях он имеет практически полную свободу для самых изощренных реализаций.

7. Сохранение компонентов в долговременной памяти

7.1. Сохранение и восстановление компонентов

Среда JavaBeans полагается на стандартные механизмы сохранения Java-объектов. Таких механизмов два. Интерфейс java.io. Serializable (см. листинг 18) предусматривает сохранение в стандартной форме графа объектов, доступных из данного. Впрочем, методы writeObject/readObject и здесь предоставляют пользователям определенную свободу. Интерфейс java.io.Externalizable (см. листинг 19) делает эту свободу абсолютной — Java-машина берет на себя лишь сохранение классовой информации, все остальное отдается на откуп пользователю. Реализация интерфейса Externalizable целесообразна в первую очередь тогда, когда есть необходимость представить компонент JavaBeans как элемент другой объектной среды, такой, например, как OLE или OpenDoc.

При сохранении объектов в долговременной памяти приходится учитывать стандартные проблемы, связанные с различным временем жизни компонентов, с изменением их версий и с обеспечением их информационной безопасности; мы, однако, на этих проблемах останавливаться не будем.

Создание нового экземпляра компонентов также сопряжено с некоторыми тонкостями. Экземпляр может создаваться в разных контекстах, например, в контексте приложения или в контексте инструментального окружения. Метод instantiate класса java.beans.Beans (см. лис-

```
public interface Externalizable
   extends Serializable {

   public abstract void writeExternal
     (ObjectOutput out) throws IOException;
   // Сохранение объекта в нестандартном формате

   public abstract void readExternal
     (ObjectInput in) throws IOException,
                      ClassNotFoundException;
   // Чтение нестандартно сохраненного объекта —
   // метод, обратный writeExternal ()

}
```

Листинг 19.

```
public class Beans extends Object {

   public static Object instantiate
     (ClassLoader cls, String beanName)
        throws IOException, ClassNotFoundException;
   // Создание экземпляра компонента

   public static boolean isDesignTime ();
   // Истина, если работа идет в инструментальном
   // окружении

   public static boolean isGuiAvailable ();
   // Истина, если работа идет в графическом окружении

. . .

}
```

Листинг 20.

тинг 20) наделен достаточным интеллектом, чтобы учесть контекст и сделать все корректно; в этой связи пользователю рекомендуется избегать создания экземпляров иными методами.

Помимо instantiate, класс Beans содержит некоторые другие методы, облегчающие управление компонентами объектной среды (см. листинг 20).

7.2. Упаковка компонентов

Упаковка сохраненных Java-объектов (и, в частности, компонентов объектной среды) важна прежде всего для эффективной доставки их по сети. Спецификации JavaBeans рекомендуют (но не предписывают) использовать для упаковки формат JAR (Java-архив).

Java-архивы могут содержать файлы классов, результаты сериализации объектов, изображения, справочную информацию и другие ресурсы. С архивом может быть ассоциирован manifest-файл, описывающий его содержимое, в том числе зависимости между компонентами.

На примере сохранения компонентов можно видеть, насколько упрощается жизнь разработчиков за счет продуманной организации Java-технологии. Проблема сохранения решена на уровне стандартного Java-окружения, так что все надстройки (такие как JavaBeans) могут полагаться на существующие механизмы.

8. Компоненты и контейнеры

Представляется удивительным, что спецификация [3], регламентирующая фундаментальное отношение компоненты/контейнер, не вошла в число первоочередных и дорабатывается только сейчас, в рамках новой версии JavaBeans с рабочим названием Glasgow.

Механизм контейнеров необходим для достижения по крайней мере двух целей:

- организации иерархической логической структуры компонентов в рамках объектной среды;
- организации единого контекста для совокупности компонентов, предоставляющего им набор сервисов для взаимодействия между собой и с окружением.

Первый пункт означает, в частности, инкапсуляцию совокупности компонентов, так что с точки зрения окружения она выглядит как единое целое с набором методов, предоставляемых контейнером. Кроме того, применительно к иерархической структуре возможен систематический обход и обработка ее элементов.

Второй пункт имеет противоположное назначение — инкапсуляцию окружения. Контейнер выступает в роли оболочки, скрывающей от компонентов особенности внешней среды и предоставляющей им свой контекст.

Чтобы избежать употребления перегруженного в Java-среде термина "контейнер", авторы спецификации употребляют сочетание "BeanContext". Мы не будем этого делать, поскольку, помимо предоставления общего контекста, у контейнера есть и другие функции; надеемся, что к путанице это не приведет.

Реализация механизма контейнеров использует служебный интерфейс java.util.Collection, который предполагается включить в одну из ближайших версий Java-среды. Фрагмент описания этого интерфейса, содержащий типичные методы для работы с наборами, приведен на листинге 21.

```
public interface Collection {
   public abstract boolean add (Object o)
       throws ...;
   // Включает объект в набор. Возвращает "ложь",
   // если объект там уже был

   public abstract void addAll (Collection c)
       throws ...;
   // Включает все элементы одного набора в другой

   public abstract boolean remove (Object o)
       throws ...;
   // Удаляет объект из набора

   public abstract void clear () throws ...;
   // Удаляет все элементы из набора

   public abstract boolean contains (Object o);
   // Проверяет, входит ли данный объект в набор

   public abstract Iterator iterator ();
   // Возвращает итератор набора, позволяющий
   // перебирать элементы

   public abstract Object [] toArray ();
   // Преобразует набор в массив

   . . .
}
```

Листинг 21.

Интерфейс java.beans.BeanContextChild содержит описания методов, позволяющих ассоциировать с компонентом объемлющий контейнер и опрашивать эту ассоциацию (см. листинг 22). Таким образом, связи, ведущие вниз (от контейнера к компоненту), обслуживает интерфейс Collection, а связи, ведущие вверх, — интерфейс BeanContextChild.

С отношением компоненты/контейнер ассоциировано событие beanContextChanged. Соответствующий интерфейс (BeanContextListener) описан на листинге 23.

Вообще говоря, распространение этого события может происходить в несколько приемов: контейнер, получив извещение от компонента, передает его своим подписчикам, в число которых, вероятно, входит объемлющий контейнер, и т.д. Чтобы распознать подобные "вторичные" события и определить первоисточник, предусмотрены соответствующие методы событийного объекта BeanContextEvent (см. листинг 24).

Изменения совокупности компонентов, входящих в контейнер, обслуживает событийный объект BeanContextMembershipEvent. Он содержит информацию о разности ("дельте") между старым и новым составом контейнера, то есть о том, какие компоненты были добавлены или, напротив, удалены (листинг 25).

Интегрирующим элементом рассматриваемой спецификации является интерфейс java.beans.BeanContext, описывающий связи, идущие как вверх, так и вниз (за счет расширения интерфейсов BeanContextChild и Collection соответственно). Интерфейс BeanContext позволяет также опросить предоставляемые контейнером сервисы и ресурсы. Содержит он и методы для регистрации подписчиков событий (см. листинг 26).

Разумеется, кроме синтаксиса специфицируется семантика методов интерфейса BeanContext.

При добавлении компонента методом add (), унаследованным у интерфейса Collection, контейнер "привязывает" этот компонент к себе, вызывая в нем метод setBeanContext с аргументом this (полноценные компоненты должны реализовывать интерфейс BeanContextChild). В свою очередь, компонент может протестовать против включения в контейнер, возбуждая исключительную ситуацию PropertyVetoException. Если это случится, контейнер обязан отменить добавление, возбудив исключительную ситуацию IllegalArgumentException. При успешном добавлении компонента контейнер распространяет подписчикам событие beanContextChanged с соответствующим объектом-параметром. Контейнер должен подписаться у нового компонента на события, связанные со свойствами послед-

```
public interface BeanContextChild
   extends BeanContextListener {

   void setBeanContext (BeanContext bc)
       throws PropertyVetoException;
   // Ассоциирует контейнер с компонентом.
   // Если компонент считает, что новый контейнер ему
   // не подходит, он возбуждает исключительную ситуацию.
   // Интерфейс BeanContext описывается ниже, на листинге 26

   BeanContext getBeanContext ();
   // Опрашивает ассоциацию

   . . .
}
```

Листинг 22.

```
public interface BeanContextListener
   extends java.util.EventListener {

   void beanContextChanged (BeanContextEvent bce);
   // Метод, вызываемый в объектах-подписчиках
   // при изменениях контейнера

}
```

Листинг 23.

```
public abstract class BeanContextEvent
   extends java.util.EventObject {

   public BeanContext getBeanContext ();
   // Возвращает контейнер, распространяющий событие

   public synchronized void setPropagatedFrom
       (BeanContext bc);
   // Устанавливает первоисточник события

   public synchronized BeanContext
       getPropagatedFrom ();
   // Возвращает первоисточник события

   public synchronized boolean isPropagated ();
   // Истина, если событие является вторичным

}
```

Листинг 24.

```
public abstract class BeanContextMembershipEvent
   extends BeanContextEvent {

   public boolean isDeltaMember (Object o);
   // Истина, если заданный компонент был добавлен
   // или удален

   public Object [] getDeltas ();
   // Возвращает добавленные или удаленные компоненты

   public boolean isChildrenAddedEvent ();
   // Истина, если имело место добавление компонентов

   public boolean isChildrenRemovedEvent ();
   // Истина, если имело место удаление компонентов

}
```

Листинг 25.

```
public interface BeanContext
   extends java.beans.BeanContextChild,
       java.util.Collection {

   Object instantiateChild (String beanName)
       throws IOException, ClassNotFoundException;
   // Создает новый экземпляр компонента, заданного
   // именем, и включает его в себя как в контейнер

   Object hasService (Class serviceClass,
                BeanContextChild requestor);
   // Истина, если контейнер предоставляет
   // запрашиваемый сервис

   Object getService (Class serviceClass,
                BeanContextChild requestor);
   // Возвращает сервисный объект запрашиваемого класса

   public java.net.URL getResource (String name,
                BeanContextChild requestor);
   // Возвращает универсальный локатор ресурса
   // с заданным именем

   void addBeanContextListener
       (BeanContextListener bcl);

   void removeBeanContextListener
       (BeanContextListener bcl);
   . . .
}
```

Листинг 26.

него, чтобы отслеживать по крайней мере свойство beanContext и не допустить нарушения целостности иерархической структуры. Кроме того, компонент регистрируется как подписчик событий, возбуждаемых контейнером.

При удалении объекта из контейнера производятся обратные действия. В частности, вызывается метод setBeanContext с аргументом null. Если компонент находится в состоянии, не позволяющем произвести удаление, он возбуждает исключительную ситуацию PropertyVetoException, заставляя тем самым контейнер отказаться от удаления.

Отметим, что контейнерная реализация методов интерфейса Collection должна быть безопасной в многопотоковой среде.

Рассмотренная спецификация заполняет очень важную методологическую брешь в JavaBeans. Хотелось бы надеяться, что переход к новой версии прикладного программного интерфейса Java, включающей интерфейс BeanContext, произойдет в ближайшее время.

9. Обмен данными

Компоненты объектной среды обычно взаимодействуют не только с объемлющим контейнером, но и между собой. Механизм событий — это одна грань такого взаимодействия. В дополнение необходима возможность обмена структурированными элементами данных.

Соответствующая спецификация [5] готовится специалистами компаний Sun Microsystems и Lotus Development. И хотя совместная работа только началась, мы сочли необходимым осветить выдвинутые предложения ввиду исключительной важности затрагиваемой темы.

Центральным понятием предлагаемых спецификаций является информационная шина. Компоненты могут подключаться к шине, помещать на нее данные (это делают поставщики) и считывать данные (это делают потребители). Обмен носит асинхронный характер: поставщик, поместив данные, не заботится о том, когда они будут считаны. Для идентификации элементов данных используются их имена.

Информационная шина описывается классом InfoBus. Методы этого класса (реализуемого в рамках виртуальной Java-машины) порождают экземпляры шины, осуществляют подключение компонентов к подходящим экземплярам, отслеживают список шин и подключенных к ним компонентов, распространяет события, обслуживающие обмен данными, и т.п. Фрагмент описания класса InfoBus представлен на листинге 27.

Процесс информационного взаимодействия компонентов в спецификациях InfoBus можно подразделить на пять фаз:

- подключение к шине;
- прослушивание шины;
- установление контакта между поставщиком и потребителем, передача элемента данных;
- выяснение формата элемента данных;
- интерпретация элемента данных.

Для подключения к шине компонент должен реализовать интерфейс InfoBusMember, получить ссылку на экземпляр шины и войти в число ее членов. Фрагмент интерфейса InfoBusMember приведен на листинге 28. Обратим внимание, что шина, ассоциируемая с компонентом, трактуется как его свойство, изменение значения которого может отслеживаться на основе механизма событий.

Для облегчения реализации интерфейса InfoBusMember спецификации предлагают класс InfoBusMemberImpl, который, в дополнение к "обязательным", предоставляет еще два удобных метода — joinInfoBus () и leaveInfoBus () (см. листинг 29).

Для реализации второй фазы (прослушивание шины) поставщик должен подписаться на информацию о запросах данных, воспользовавшись методом addDataProducer () класса InfoBus. Аналогично, потребитель должен подписаться на информацию о наличии данных, обратившись к методу addDataConsumer (). Компонент может одновременно являться и поставщиком, и потребителем (типичный пример — промежуточное звено конвейера).

Третью фазы информационного взаимодействия компонентов обслуживают преимущественно событийные объекты. В соответствии с моделью, принятой в спецификации InfoBus, поставщик оповещает потребителей о появлении нового элемента данных. Потребители запрашивают у поставщиков данные, когда в них возникает нужда. Обмен элементом данных становится возможным, если поставщик и потребитель используют для него одно и то же имя. Ответственность за надлежащий выбор имен лежит на разработчике приложения.

Класс InfoBusEvent является базовым для событийных объектов поставки/приема. Его описание приведено на листинге 30.

Поставщик объявляет о наличии новых данных, рассылая всем потребителям шины событийный объект класса InfoBusItemAvailableEvent. Методы этого класса позволяют получить информацию о поставщике и ассоциированный элемент данных.

Потребитель, желающий получить данные, рассылает объект класса InfoBusItemRequested-

```
public class InfoBus extends Object {

   public static synchronized InfoBus
       open (Component c);
   // Получение ссылки на экземпляр шины.
   // Аргумент используется для определения контекста
   // (контейнера), для которого подходящая шина, возможно,
   // уже существует. При необходимости создается
   // новый экземпляр шины

   public static synchronized InfoBus
       open (String busName);
   // Получение ссылки на экземпляр шины. Аргумент задает
   // желательное имя экземпляра. Обычно используется
   // не-компонентами (например, инструментальным
   // окружением)

   public synchronized void join (InfoBusMember member)
       throws  PropertyVetoException,
               InfoBusMembershipException;
   // Включение заданного компонента в число членов шины.
   // Компонент, желающий подключиться к шине, должен
   // реализовать интерфейс InfoBusMember. Экземпляр
   // шины устанавливается в качестве значения свойства
   // InfoBus нового члена

   public void leave (InfoBusMember member)
       throws PropertyVetoException;
   // Выведение компонента из числа членов шины.
   // Обычно вызывается самим компонентом

   public void propertyChange
       (PropertyChangeEvent event);
   // Обработка события, вызванного изменением значения
   // свойства InfoBus у какого-либо члена шины. Служит
   // для обеспечения целостности связей между шинами и
   // их членами

   public void addDataProducer
       (InfoBusDataProducer producer);
   public void addDataConsumer
       (InfoBusDataConsumer consumer);
   // Обслуживание подписки на события в экземпляре шины,
   // запрашиваемой поставщиками и/или потребителями
   // элементов данных

   public void fireItemAvailable (String dataItemName,
       InfoBusDataProducer producer);
   // Распространение среди потребителей события,
   // состоящего в том, что на шине появился элемент
   // данных с указанным именем, помещенный заданным
   // поставщиком

   public DataItem findDataItem (String dataItemName,
       InfoBusDataConsumer consumer);
   // Распространение среди поставщиков события,
   // состоящего в том, что заданный потребитель
   // нуждается в элементе данных с указанным именем
   . . .

}
```

Листинг 27.

```
public interface InfoBusMember {

   public void setInfoBus (InfoBus infobus)
       throws PropertyVetoException;
   // Установка шины, ассоциированной с компонентом.
   // Обычно вызывается методом InfoBus.join()

   public void addInfoBusListener
       (VetoableChangeListener vcl);
   // Обслуживание подписки на возможность запрета
   // изменения свойства InfoBus

   public void addInfoBusListener
       (PropertyChangeListener pcl);
   // Обслуживание подписки на информацию об изменении
   // свойства InfoBus
   . . .

}
```

Листинг 28.

```
public class InfoBusMemberImpl
   extends Object implements InfoBusMember {

   public void joinInfoBus (String busName)
       throws InfoBusMembershipException;
   // Подключение к шине с заданным именем

   public void leaveInfoBus ()
       throws PropertyVetoException,
              InfoBusMembershipException;
   // Отключение от шины
. . .

}
```

Листинг 29.

```
public class InfoBusEvent
   extends java.util.EventObject {

   InfoBusEvent (String itemName,
       InfoBusEventListener source);
   // Конструктор. Задаются имя ассоциированного
   // элемента данных и источник события

   public String getDataItemName ();
   // Возвращает имя ассоциированного элемента данных
}
```

Листинг 30.

```
public class InfoBusItemRequestedEvent
   extends InfoBusEvent {

   InfoBusItemRequestedEvent (String itemName,
                 InfoBusDataConsumer consumer);
   // Конструктор. Устанавливает пустое значение
   // свойства DataItem и вызывает конструктор
   // InfoBusEvent

   public void setDataItem (DataItem item);
   // Ассоциирует элемент данных с событийным объектом

   public DataItem getDataItem ();
   // Возвращает ассоциированный элемент данных

   public InfoBusDataConsumer getSourceAsConsumer ();
   // Возвращает источник события
}
```

Листинг 31.

```
public interface Aggregate
   extends java.rmi.Remote, java.io.Serializable {

   Aggregate getInstanceOf
       (Class requestedInterface);
   // Возвращает объект, реализующий методы
   // заданного интерфейса (класса)

   boolean isInstanceOf (Class requestedInterface);
   // Истина, если заданный интерфейс (класс) входит
   // в число поддерживаемых

   Enumeration getTypes ();
   // Возвращает набор поддерживаемых интерфейсов
   // (классов)
   ...

}
```

Листинг 32.

Event, описание которого приведено на листинге 31. Обратим внимание на метод setDataItem, позволяющий поставщику, получившему событийный объект, "привязать" к нему элемент данных, выдав его тем самым инициатору запроса.

Формат элемента данных определяется интерфейсами, которые этот элемент реализует. Базовым интерфейсом элементов данных является DataItem, в число преемников которого входят CollectionAccess, DbAccess и др. Поскольку проблема обмена структурированными данными уже решена в рамках абстрактного оконного инструментария (пакет java.awt.datatransfer), представляется естественным и в спецификациях InfoBus пойти тем же путем, что и было сделано.

Спецификации InfoBus только начали свой путь к официальному утверждению. Многие детали еще не определены. Тем не менее, основополагающие решения приняты, и они представляются весьма удачными. В частности, понятие экземпляра шины позволяет структурировать пространство поставщиков и потребителей, уменьшая сложность системы и повышая эффективность обмена данными.

10. Агрегирование интерфейсов

В разных ситуациях компонент объектной среды должен поворачиваться к пользователю разными гранями. Например, в инструментальном окружении необходимо получать информацию об афишируемых характеристиках объекта. В настоящее время такую информацию предоставляет специальный информационной объект, косвенно (по имени класса) ассоциированный с исследуемым и реализующий интерфейс BeanInfo. Однако более естественной была бы не косвенная, а прямая ассоциация, позволяющая единообразно осуществлять доступ к разным "проявлениям" объектов.

Решению сформулированной задачи служит очень важная в концептуальном плане спецификация [4]. Центральное место в ней занимает понятие агрегата — сущности, обладающей "многогранным" поведением, динамически унаследованным у совокупности классов (интерфейсов). В агрегат входят представители соответствующих классов, а также координатор, способный по запросу выдавать нужного представителя.

Спецификации не предусматривают внесения каких-либо изменений в язык Java. Агрегат и входящий в него координатор представлены интерфейсом Aggregate (см. листинг 32). Интерфейс Aggregate содержит методы, позволяющие получить ссылку на объект требуемого класса и опросить поддерживаемый агрегатом набор классов.

Формально каждый представитель является самостоятельным объектом, принадлежащим своему классу, однако с идейной точки зрения более правильно считать, что метод getInstanceOf () возвращает разные проявления одного (агрегатного) объекта.

11. Заключение

Спецификации JavaBeans в совокупности с предлагаемыми дополнениями образуют целостную архитектуру компонентной объектной среды, позволяющей накапливать и многократно использовать программистские знания. Уникальным достоинством JavaBeans является Java-фундамент, предоставляющий современный объектный язык, гарантирующий мобильность и информационную безопасность разрабатываемого программного обеспечения.

Компоненты среды JavaBeans оказываются мобильными вдвойне. Мост к ActiveX автоматически "вкладывает" их в эту среду, а средства Java IDL помогают организовать взаимодействие с CORBA-брокерами объектных запросов.

Разработка спецификаций JavaBeans следовала и следует традициям открытости, заложенным на предыдущих этапах развития Java-технологии. Партнерами JavaSoft в этой работе выступали такие известные компании, как Apple Computer, Borland International, IBM, Informix Software, Lotus Development, Netscape Communications, Novell, Oracle, Silicon Graphics, Sybase, Texas Instruments и многие другие. Спецификации доступны для свободного ознакомления, внесения замечаний и предложений.

Политика партнерства способствовала созданию многочисленных инструментальных окружений, поддерживающих процессы разработки и интеграции компонентов JavaBeans. Назовем Data Director for Java (Informix Software), Visual Age for Java (IBM), BeanMachine (Lotus Development), Cosmo Code (Silicon Graphics). На подходе JBuilder (Borland International), Visual JavaScript (Netscape Communications), Java Studio и Java Workshop (SunSoft), PowerJ (Sybase) и др.

Разработчики, делающие ставку на JavaBeans, не рискуют проиграть. Эта компонентная объектная среда вступает в пору зрелости, обладая всеми качествами, необходимыми для успешного продвижения.

12. Литература

1. The JavaBeans 1.01 specification. — Sun Microsystems, July, 1997.
2. Java Core Reflection. API and Specification. — Sun Microsystems, February, 1997.
3. L. Cable, G. Hamilton. A Draft Proposal to define an Extensible Runtime Containment and Services Protocol for JavaBeans (Version 0.97). — JavaSoft, August, 1997.
4. L. Cable, G. Hamilton. A Draft Proposal for a Object Aggregation/Delegation Model for JavaBeans (Version 0.8). — JavaSoft, August, 1997.
5. InfoBus Specification. Draft Version 0.04 (First Public Review). — Sun Microsystems, Inc., Lotus Development Corporation, July, 1997.
6. А. Таранов, В. Цишевский. Java как центр архипелага. — Jet Info, 1996, N 9.
7. Ю. Пуха. Объектные технологии построения распределенных информационных систем. — Jet Info, 1997, N 16.

Новая версия Java WorkShop

16 сентября 1997 года компания Sun Microsystems объявила о выпуске новой версии инструментального окружения Java WorkShop 2.0 — более мощного и удобного средства для создания компонентов JavaBeans и Java-приложений. Это интегрированное окружение считается наиболее популярным средством разработки на Java, оно работает под управлением большинства известных операционных систем: Solaris, Microsoft Windows 95 и NT, Novell IntranetWare, HP-UX, SCO-Unix.

Новая версия Java WorkShop представляет собой графическую среду с усовершенствованным интерфейсом. Она поддерживает все этапы разработки программного обеспечения. Основные возможности Java WorkShop 2.0:

- Поддержка спецификаций JDK 1.1 и компонентной модели JavaBeans.
- Наличие графического конструктора Java GUI Builder для разработки пользовательских графических интерфейсов и импорта компонентов JavaBeans.
- Новая пользовательская модель, ядром которой служит редактор, обеспечивающий быстрое написание и просмотр кодов, выделение синтаксических конструкций, автоматическое форматирование текстов программ и другие удобные возможности.
- Fast Java compiler — копилятор, ускоряющий сборку приложений в 10 — 15 раз.
- Отладчик, поддерживающий многопотоковый режим и обладающий средствами для удаленной отладки.
- Java profiler — средство выявления узких мест в приложениях.
- Project manager — средство для организации коллективной разработки компонентов JavaBeans, аплетов и приложений.

В период подготовки промышленной версии Java WorkShop 2.0 она испытывалась исследовательскими компаниями (Aberdeen Group, Business Research Group, IDC, META Group, NC.Focus, Patricia Seybold Group и Software Productivity Group) и партнерами (GemStone, HP, KL Group, MindQ, NCR, Netscape, Novell, Object Design, SCO, and TV Objects). Полнота и эффективность набора функциональных средств Java WorkShop 2.0 получили высокую оценку.

Лидеры индустрии связи выбирают PersonalJava

Компании Alcatel, Nortel и Samsung выбрали PersonalJava в качестве стандартной платформы для Web-телефонов. Все три компании планируют начать производство Web-телефонов в середине 1998 года.

Web-телефон отличается от привычного аппарата наличием средств для пользования электронной почтой и для просмотра Web-страниц. Web-телефон на платформе PersonalJava обеспечивает пользователям простой доступ в Интернет и автоматическую загрузку интересующих его аплетов, выдающих, например, биржевые сводки или прогноз погоды.

Тот факт, что компании, которые контролируют рынок телекоммуникационных и потребительских технологий, оцениваемый в 110 миллиардов долларов, отдали предпочтение PersonalJava, свидельствует о большом успехе Sun Microsystems. Несомненно, старт PersonalJava можно назвать блестящим.

Выражая свое удовлетворения решением компаний Alcatel, Nortel и Samsung, президент Sun Microsystems Скотт МакНили тем не менее заметил: "Web-телефоны — это всего лишь начало глобального проникновения Java на рынок домашних и потребительских товаров. Это пример для других разработчиков. Теперь они получили возможность создавать качественно новые конкурентоспособые изделия". Действительно, PersonalJava позволяет создавать аплеты и приложения, которые могут выполняться на любом потребительском устройстве, подключенном к сети.

PersonalJava

PersonalJava — это новая среда для Java-приложений (Java Application Environment), ориентированная на потребительские устройства домашнего и офисного применения и на мобильные устройства. PersonalJava состоит из ядра и стандартных прикладных программных интерфейсов. Функциональные возможности PersonalJava являются подмножеством Java. PersonalJava была создана специально для работы в ограниченных условиях и дополнена специфическими возможностями, необходимыми в потребительских приложениях.

PersonalJava предоставляет преимущества как потребителям, так и проектировщикам.

С потребительской точки зрения существенным является экономическая эффективность. Во-первых, PersonalJava продлевает жизненный цикл устройств, поскольку появляется возможность обновления функций путем загрузки программного обеспечения по сети. Тем самым снижается риск морального старения. Во-вторых, благодаря компактности байт-кодов Java, выдерживается умеренность требований к аппаратным ресурсам устройств.

Для разработчиков преимущества заключаются в том, что PersonalJava работает на виртуальной Java-машине, а это значит, что приложения не ограничены типами процессоров и ОС. Простота разработки, повторное использование кодов и информационная безопасность, изначально заложенная в Java-архитектуру, продолжают список качеств, которые привлекают к себе программистов.

Предварительная версия спецификации PersonalJava была опубликована 2 июля 1997 года (см. http://java.sun. com/products/personaljava/). За три летних месяца к ней было зарегистрировано более 10 000 обращений, что свидетельствует о несомненной заинтересованности разработчиков.

Jet Info
ИНФОРМАЦИОННЫЙ БЮЛЛЕТЕНЬ
Издается с 1995 года
Издатель: компания Джет Инфо Паблишер

Главный редактор: Галатенко В.А. (galat@jet.msk.su)
Технический редактор: Демочкин С.И. (serged@jet.msk.su)

Россия, 103006, Москва, Краснопролетарская, 6
тел. (095) 972 11 82, 972 13 32
факс (095) 972 07 91
e-mail: JetInfo@jet.msk.su

Подписной индекс по каталогу Роспечати 32555