Jet Info

ИНФОРМАЦИОННЫЙ БЮЛЛЕТЕНЬ № 11-12(66-67)/1998

JAVA В ТРИ ГОДА стр.2

НОВОСТИ ИНТЕРНЕТ стр. 36

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Java в три года

Александр Таранов,
Владимир Цишевский

СОДЕРЖАНИЕ

1. Введение

Более трех лет назад, в мае 1995 года, было официально объявлено о появлении в компании Sun Microsystems новой технологии создания сетевых систем — технологии Java, основными составляющими которой стали язык программирования, виртуальная машина и прикладной программный интерфейс. Java сразу же вызвала огромный интерес, масштабы которого трудно в точности оценить и еще труднее объяснить. Казалось бы, компания Sun не предложила ничего нового. Объектно-ориентированные языки развиваются со второй половины 60-х годов. Еще больше стаж идей и систем мобильного программирования. Подход к построению сетевых систем на основе так называемых активных агентов несколько лет развивала могущественная AT&T. Тем не менее, Java в момент своего появления была воспринята как долгожданное откровение.

Основных причин феноменального успеха Java, на наш взгляд, две. Во-первых, в Java-технологии объединено все или почти все необходимое для разработки современных приложений. Во-вторых, дело не только в возможностях той или иной технологии, но и в том, с чем она сопрягается. Важно не столько то, что и как нужно делать, сколько то, чего делать не нужно, потому что это уже есть. Java отлично «вкладывается» в WWW и СУБД, что автоматически на порядок повышает достоинства Java-приложений.

Сейчас, по прошествии трех лет, можно сказать, что основные принципы Java-технологии были выбраны весьма удачно. Язык Java почти не изменился. То же относится к виртуальной машине. Лишь окружение времени выполнения и, в частности, программные интерфейсы интенсивно развивались, что представляется вполне естественным.

В данной статье мы постараемся описать современное состояние Java-технологии. Предполагается, что читатель знаком с основами Java (см., например, [1] и [2]), поэтому основное внимание будет уделено нововведениям последнего года, а также другим важным чертам Java, о которых Jet Info еще не писал.

2. Новое в языке Java

Язык программирования — один из самых стабильных компонентов Java-технологии. Java выгодно отличается от монстров типа C++ отсутствием «лишних деталей». Требования практики заставили лишь немного отступить от первоначальной стройной концепции. Среди немногочисленных нововведений можно выделить следующие:

- появление внутренних (локальных) классов;
- появление объектных ссылок;
- стандартная реализация коллекций (наборов) объектов.

2.1. Внутренние классы

Внутренние классы появились в JDK (Java Development Kit, среда разработки для Java) версии 1.1, то есть около года назад. До этого описания классов могли присутствовать только на верхнем уровне Java-программ. Введение локальных классов преследует несколько целей:

- соблюдение концептуальной целостности при «скрещивании» объектного подхода и традиционной блочной структуры языков программирования;
- упрощение «обертывания» процедур и функций в объекты для передачи их в качестве параметров.

Первая цель носит скорее философский характер, хотя в ней есть и практические аспекты, а именно:

- упрощается обращение к компонентам других классов. Внутренний класс, в соответствии с правилами блочной структуры, может обращаться к компонентам содержащего его (внешнего) класса;
- глобальное пространство имен освобождается от лишних элементов.

Вторая цель более прозаична и является в первую очередь попыткой смягчить недостатки Java, связанные с некоторой слабостью механизма параметризации (вызванной, в свою очередь, слишком «пуристским» следованием канонам объектного подхода).

Чтобы проиллюстрировать работу с внутренними классами, обратимся, как это принято в литературе по объектному подходу, к реализации стека (см. листинг 1). Наша цель — построить такую реализацию стека, которая позволит последовательно перечислять его элементы, начиная с вершины.

Для этих целей существует интерфейс `java.util.Enumeration`, однако не имеет смысла засорять реализацию стека методами этого интерфейса, поскольку они являются необязательным дополнением к его (стека) функциональности.

```
public class FixedStack {
  Object array[];
  int top = 0;
  FixedStack(int fixedSizeLimit) {
    array = new Object[fixedSizeLimit];
  }
  public void push(Object item) {
    array[top++] = item;
  }
  public boolean isEmpty() {
    return top == 0;
  }
   •••
  // Другие методы класса (не показаны)
   •••
  // Этот класс определен как часть другого класса
  // он описан рядом с переменными, которые использует

  class Enumerator implements java.util.Enumeration {
    int count = top;
    public boolean hasMoreElements() {
      return count > 0;
    }
    public Object nextElement() {
      if (count == 0)
        throw new ArrayIndexOutOfBoundsException("FixedStack");
      return array[——count];
    }
  }
  public java.util.Enumeration elements() {
    return new Enumerator();
  }
```

Листинг 1. Пример определения внутреннего класса.

Вместо этого мы описали внутренний класс и вернули соответствующий объект в качестве результата метода `elements`.

Обратим внимание, что, как и положено в языках с блочной структурой, внутренние объекты имеют доступ к компонентам объемлющих объектов (в данном случае это массив элементов стека и текущая верхняя граница), однако для Java-компилятора подобный подход создает определенные проблемы. Чтобы во время выполнения такой доступ был возможен, приходится наделять внутренние объекты контекстом, передавая им при создании дополнительный служебный параметр — внешний объект, который сохраняется компилятором в скрытом компоненте (он доступен по имени `this$0`). В результате вложенные объекты оказываются связанными в список, в начале которого находится самый внутренний элемент.

Создатели языка Java ввели также анонимные внутренние классы. Такие классы являются носителями методов, связанных, например, с обработкой событий. В [2] мы подробно рассматривали механизм событий применительно к компонентам JavaBeans. При инициализации компонента может быть использован фрагмент программы, приведенный на листинге 2. Здесь аргументом метода `setActionListener` является объект анонимного класса, реализующего интерфейс `ActionAdapter`. Описание класса является частью выражения, начинающегося оператором `new`, за которым следует имя наследуемого класса или (как в данном случае) реализуемого интерфейса. Конечно, подобные «классовые литералы» хороши только тогда, когда содержат лишь несколько строк исходного текста.

Показателем своеобразной «старости» Java-технологии является тот факт, что для нее уже встают вопросы обратной совместимости. Чтобы программы с внутренними классами могли выполняться старыми Java-машинами, компилятор вынужден преобразовывать исходный текст, вынося все классы на верхний уровень и снабжая их квалифицирующими префиксами вида `имя_объемлющего_класса$`. Таким образом, байт-коды остаются по сути прежними, без каких-либо новых операций. Вероятно, более поздние версии виртуальных Java-машин будут действовать более оптимально.

```
package java.awt;

import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

class GUI extends Frame {
  // .........
  public GUI(){
    Button sb = new Button("earch");
    sb.addActionListener (
      new ActionListener () {
          public void actionPerformed(ActionEvent e){
              // действия, инициируемые нажатием кнопки ...
          }
      }
    );
  }
}
```

Листинг 2. Пример определения и использования анонимного внутреннего класса.

2.2. Слабые ссылки

В JDK 1.2 появился новый класс — `Reference`, «материализовавший» ссылки на объекты, то есть сделавший сами ссылки объектами. Основной метод этого класса, `get()`, позволяет перейти от ссылки к указуемому объекту.

Ссылки должны помочь более гибкой, чем это возможно посредством метода `finalize`, реализации «предсмертных» действий. Кроме того, ссылки могут использоваться в разного рода кэшах и отображениях, когда не исключено исчезновение обслуживаемых объектов. Основные свойства таких ссылок[1] :

- они не препятствуют попаданию указуемых объектов в мусор;
- сборщик мусора автоматически обнуляет их, если принимается решение об утилизации указуемых объектов;
- их нельзя изменять из программы, поскольку отсутствует метод `set()` — указуемый объект может быть задан только при создании ссылки.

Для обеспечения максимальной гибкости выделено три вида ссылок:

```
public abstract class Reference extends Object {
  void  clear(){
    //Обнуляет указуемый объект
  }

  boolean enqueue(){
    //Добавляет данную ссылку к зарегистрированной очереди
  }

  Object  get(){
    //Возвращает указуемый объект
  }

  boolean isEnqueued(){
    //Проверяет, помещена ли ссылка в очередь
  }
}
```

Листинг 3. Класс Reference.

[1] *Обычно в системах, включающих в себя сборщик мусора, такого рода ссылки называются слабыми (Weak Reference).*

- мягкие (soft);
- слабые (weak);
- призрачные (phantom).

Когда объект перестает быть доступным в традиционном для Java-программ смысле, сборщик мусора, вообще говоря, может утилизировать его, несмотря на наличие ссылок на него (сборщик мусора ровно так и поступает, если другими способами не удается получить достаточного количества свободной памяти). В первую очередь аннулируются призрачные ссылки, в последнюю — мягкие.

На листинге 3 приведено описание базового класса ссылок Reference, на листинге 4 — описание класса мягких ссылок SoftReference, на листинге 5 — класса очередей ссылок ReferenceQueue. Когда сборщик мусора утилизует указуемый объект, он не только обнуляет соответствующие ссылки, но и помещает содержащие их объекты в очереди, заданные при создании (см. первый конструктор класса SoftReference). Программа (в отдельном потоке или перед определенными операциями) может опрашивать эти очереди посредством методов poll() или remove(), узнавая таким образом о действиях сборщика мусора. Впрочем, даже если ссылка не попала в очередь утилизируемых, нет гарантии, что метод get() не вернет null, поскольку сбор мусора происходит асинхронно.

Отметим, что для призрачных ссылок get() всегда возвращает null, так что для них единственной содержательной операцией (помимо конструктора) является опрос очередей и выполнение необходимых «предсмертных» действий. Призрачные ссылки вполне оправдывают свое экзотическое название[2].

2.3. Коллекции

Введение в Java коллекций (наборов) важно как с утилитарной, так и с концептуальной точек зрения. Коллекции используются во многих программах, поэтому целесообразно выработать эталонный программный интерфейс к ним и предложить соответствующую реализацию, чтобы увеличить коэффициент повторной используемости кода со всеми вытекающими положительными последствиями. Кроме того, коллекции должны выступать в роли контейнеров, логически замыкая компонентную объектную модель JavaBeans.

Реализация коллекций — традиционно трудная проблема для языков программирования со статической типизацией, каковым является Java. Если описывать коллекции для каждого типа компонентов, происходит размножение кода. Если приписать компонентам некий общий тип, такой как Object, статический контроль окажется неэффективным и накладные расходы на проверки перекочуют в динамику, на этап выполнения программы. Чуда не произошло, авторы языка не предложили какого-то третьего пути, предоставив разработчикам в каждом конкретном случае выбирать меньшее из зол.

Еще одна проблема, присущая коллекциям, состоит в оптимизации их представления в зависимости от допустимых операций и других накладываемых ограничений. Имеется в виду возможность совпадения разных элементов коллекции (по этому критерию множества отличаются от мультимножеств), наличие операции сравнения элементов и, соответственно, возможность их сортировки, допустимость добавления новых элемен-

```
public class SoftReference extends Reference {
  SoftReference (Object referent, ReferenceQueue q)
    throws  NullPointerException {
    // Создает новую ссылку на данный объект и регистрирует ее
    // в данной очереди
  }
  SoftReference(Object referent)
    throws  NullPointerException {
    // Создает новую ссылку на данный объект без регистрации
  }
  Object get(){
    // Возвращает указуемый объект
  }
}
```

Листинг 4. Класс SoftReference.

[2] *Обычно в системах, включающих в себя сбор мусора, реализуются один вид слабых ссылок. Необходимость введения призрачных (или "фантомных") ссылок достаточно сомнительна, учитывая, что связанная с ними функциональность может быть реализована "штатными" средствами.*

```
public class ReferenceQueue extends Object {
  public ReferenceQueue(){
    // Создание новой очереди ссылок
  }
  public Reference poll(){
    // Опрос очереди на предемет наличия в ней ссылок.
    // Немедленно возвращает одну из ссылок, если таковая есть
  }
  public Reference remove(long timeout)
    throws IllegalArgumentException, InterruptedException
    {
      // Удаление из очереди следующей ссылки.
      // Если ссылок нет, ожидание не может превысить timeout
    }
  public Reference remove()
    throws InterruptedException{
    // Удаление из очереди следующей ссылки
    // с неограниченным ожиданием
  }
}
```

Листинг 5. Класс ReferenceQueue.

тов и т.п. Для решения этой проблемы авторы языка ввели два вида коллекций — множества и списки, определив в них различные подвиды.

С технической точки зрения коллекции представлены иерархией интерфейсов и классов. Корнем иерархии является интерфейс `Collection`, описание которого представлено в табл. 1. Этот интерфейс расширяют `Set` (множества), `SortedSet` (множества с поддерживаемым отношением порядка) и `List` (списки, допускающие операции с использованием позиций элементов в коллекции). Концептуально важными преемниками являются `BeanContext` и `BeanContextService`. В «связку» входят также интерфейсы `Map` и `SortedMap`, специфицирующие операции с отображениями, то есть с наборами пар (ключ, значение).

В реализации интерфейсов можно выделить два уровня. Первый составляют абстрактные классы, такие как `AbstractCollection`. Разработчик может наследовать их, дописывая или изменяя необходимые детали. На втором уровне располагаются конкретные представления коллекций, основывающиеся на хэш-таблицах, массивах изменяемого размера, сбалансированных бинарных деревьях или двусвязных списках. В табл. 2 сведены поддерживаемые комбинации интерфейсов и реализующих их классов.

Известно, что в ряде ситуаций полезен «полиморфизм представлений», когда один объект имеет несколько представлений, оптимизированных для выполнения специфических операций. В идеале эти представления должны поддерживаться на протяжении всего жизненного цикла объекта. Авторы Java предложили частичное решение проблемы полиморфизма представлений, введя «копирующие» конструкторы, создающие новый экземпляр коллекции из элементов существующего набора, быть может, отличающегося реализацией. Например, конструктор `TreeSet(Collection c)` позволяет представить коллекцию `c` в виде экземпляра класса `TreeSet`.

В Java-коллекциях использованы описанные выше новые возможности языка. Внутренние классы помогают реализовать итераторы. Слабые ссылки часто используются в отображениях (`Map`) специального назначения.

3. Окружение времени выполнения

Эффективность среды программирования определяется не столько языком, сколько стандартизованным окружением времени выполнения. Чем можно пользоваться без ущерба для мобильности программ? Насколько это сокращает сроки разработки приложений? Каков коэффициент повторного использования кода? Насколько гибкими являются стандартные средства, не загоняют ли они приложения в прокрустово ложе? Ответы на эти вопросы может дать лишь длительная практика.

В объектной среде окружение времени выполнения представляет собой набор стандартных классов. В данном разделе мы попытаемся рассмотреть Java-классы с точки зрения разработчика приложений для различных видов платформ.

Прежде всего отметим, что несмотря на внутренне присущую Java мобильность, вопрос мобильности приложений остается непростым. Существенно, насколько эффективно та или иная платформа реализует стандартные классы, а также насколько привычным оказывается пользовательский интерфейс.

Далее, необходимо учитывать, что разные платформы располагают существенно разными ресурсами. В таких условиях невозможно придерживаться единого стандарта; приходится определять семейство стандартов, ориентированных на виды платформ. В Java-технологии выделяется два вида стандартов: потребительский и корпоративный. Потребительские платформы по количеству наличных ресурсов стоят на одну или несколько ступенек ниже корпоративных. Первые представляют собой устройства со встроенным микропроцессором, вторые — полноценные компьютеры.

3.1. Java на потребительских платформах

Потребительские устройства можно разделить на несколько категорий в соответствии с их функциональными возможностями:

- простейшие: смарт-карты, игровые приставки;
- самостоятельные устройства: принтеры, цифровые фото- и видеокамеры, электронные секретари, младшие модели сотовых телефонов, пейджеры, факсимильные аппараты и др.;

`boolean add(Object o)`	Добавляет новый элемент в коллекцию, если ранее он там отсутствовал*
`boolean addAll(Collection c)`	Добавляет все элементы коллекции `c`*
`void clear()`	Удаляет все элементы коллекции*
`boolean contains(Object o)`	Возвращает `true`, если коллекция содержит указанный элемент
`boolean containsAll(Collection c)`	Возвращает `true`, если коллекция содержит все элементы коллекции `c`
`boolean equals(Object o)`	Сравнение с объектом `o`
`int hashCode()`	Вычисление свертки
`boolean isEmpty()`	Возвращает `true`, если коллекция пуста
`Iterator iterator()`	Создание итератора для данной коллекции
`boolean remove(Object o)`	Если коллекция содержит указанный элемент, он удаляется*
`boolean removeAll(Collection c)`	Удаляет все элементы коллекции c из данной коллекции, если они содержатся в данной коллекции*
`boolean retainAll(Collection c)`	Операция, обратная операции `boolean removeAll(Collection c)`. Удаляются все элементы коллекции, которые не содержатся в коллекции c*
`int size()`	Количество элементов в коллекции
`Object[] toArray()`	Создание массива, содержащего все элементы коллекции
`Object[] toArray(Object[] a)`	Создание массива, содержащего все элементы коллекции. Тип полученного массива совпадает с типом массива a

Примечание:
**Реализация метода может отсутствовать. При попытке вызвать такой метод для объекта, для которого этот метод не определен, возбуждается исключительная ситуация UnsupportedOperationException.*

Табл. 1. Интерфейс Collection.

Реализации/ Интерфейсы	Хэш-таблицы	Массивы изменяемых размеров	Сбалансированные бинарные деревья	Связанные списки
Set	*HashSet*		*TreeSet*	
List		*ArrayList*		*LinkedList*
List		*ArrayList*		*LinkedList*
Map	*HashMap*		*TreeMap*	

Таблица 2. Интерфейсы коллекций и реализующие их классы.

- устройства с открытым подключением: сетевые компьютеры и устройства других типов, обеспечивающие доступ в Интернет.

Для этих категорий в Java-технологии определены «встроенные» стандарты языка и прикладного программного интерфейса, а именно:

- для простейших — `Java Card`;
- для самостоятельных — `EmbeddedJava`;
- для устройств с открытым подключением — `PersonalJava`.

3.1.1. Java Card

Ресурсы смарт-карт чрезвычайно ограничены. Как правило, доступен 1 КБ оперативной памяти и 16 КБ постоянной. Естественно, виртуальную Java-машину в полном объеме в смарт-карту не поместить. Приходится идти на существенные урезания языка и окружения времени выполнения.

Из Java Card изъяты потоки, динамическая загрузка классов, сбор мусора, менеджер безопасности, типы `char`, `double`, `float`, `long`, многомерные массивы. Из стандартных классов в ограниченной форме поддерживаются только `java.lang.Object` и `java.lang.Throwable` (что-что, а исключения поддерживаются всегда и везде).

Разумеется, в `Java Card` предусмотрена поддержка международного стандарта ISO 7816, а также промышленного стандарта EMV (Europay, Master Card Visa). Соответствующую функциональность обеспечивают пакеты `javacard.framework` (основные операции и интерфейс с платформой), `javacardx.framework` (файловая система), `javacardx.crypto` и `javacardx.cryptoEnc` (криптография).

Технологию `Java Card` лицензировали около полутора десятков компаний, в число которых входят Motorola Corporation и Dallas Semiconductor.

В качестве отправной точки для детального ознакомления с `Java Card` можно рекомендовать статью [3].

3.1.2. EmbeddedJava

Специфика EmbeddedJava привела к появлению двух новых терминов — *ромлет* (ROMlet) и *патчлет* (PATCHlet). Первый обозначает образ приложения, помещаемый в постоянную память, второй — «заплатку», которую можно наложить «по ходу дела». Естественным местом расположения для патчлетов является флэш-память, допускающая несложную замену.

Самостоятельные устройства обладают хотя и ограниченными, но все же на порядок большими ресурсами, чем смарт-карты, поэтому `EmbeddedJava` поддерживает практически все стандартные Java-пакеты, кроме `java.applet` (последнее представляется вполне естественным). Однако схожесть со стандартным окружением обманчива. `EmbeddedJava` основана на собственной реализации классов (см. табл. 3), главным свойством которой является модульность. Возможности, которые приложением не используются, удаляются из окружения времени выполнения, после чего производится компактификация кода и данных.

Как правило, операционной платформой для `EmbeddedJava` является ОС реального времени. Требуется, чтобы подобная ОС поддерживала динамическое распределение памяти и многопотоковость. Желательна, но не обязательна поддержка файловых систем, сетевых возможностей и графической оконной среды.

Многие производители интеллектуальных устройств и операционных систем реального времени энергично поддерживают Java-технологию вообще и спецификации `EmbeddedJava` в частности. Назовем такие компании, как Acorn, Chorus, Lucent Technologies, QNX, Wind River Systems.

На наш взгляд, `EmbeddedJava` — один из самых удачных компонентов Java. Здесь сочетаются следование стандарту на программный интерфейс, гибкость конфигурирования, опора на промышленные ОС реального времени, целостность набора инструментальных средств.

Компонент	Описание
Классы уровня приложения	EmbeddedJava devices включают в себя приложения управления встроенными устройствами и, возможно, их пользовательским интерфейсом. Поставляются обычно производителями самих устройств.
Классы EmbeddedJava	Эти классы имеют тот же протокол, что и соответствующие классы JDK 1.1, но реализация оптимизирована для использование во встраиваемых устройствах. Лицензия на эти классы предоставляется производителям устройств с EmbeddedJava. Включает все классы JDK 1.1 Java Platform Core API, кроме пакета java.applet. Предоставляется либо непосредственно компанией JavaSoft, либо через поставщика RTOS.
Специфические классы устройства	Классы, предназначенные для управления физическим устройством. Поставляются производителем устройства либо поставщиком RTOS.
Виртуальная машина	Виртуальная машина отвечает за загрузку классов Java, интерпретирует байт-коды и вызывает внешние методы. Предоставляется либо непосредственно компанией JavaSoft, либо через поставщика RTOS.
RTOS	RTOS поставляется третьими фирмами. Минимальное требование — обеспечить потребности работы виртуальной машины EmbeddedJava.
Аппаратура	Аппаратура поставляется производителями устройства. Минимальный набор включает процессор, ROM для кодов и оперативную память для размещения стека и данных. Дополнительно может включать в себя графическое устройство, сетевое оборудование и т.д.

Табл. 3. Компоненты EmbeddedJava.

3.1.3. PersonalJava

`PersonalJava` ориентирована на наиболее продвинутые потребительские платформы, приближающиеся по объему наличных ресурсов и функциональности к универсальным компьютерам. Здесь оперативную память считают на мегабайты, имеется полноценное подключение к сети, графический дисплей и клавиатура. В то же время определенный дефицит ресурсов все же дает о себе знать, поэтому приходится прилагать усилия по оптимизации как самой платформы `PersonalJava`, так и прикладных систем.

По сравнению с `EmbeddedJava`, `PersonalJava` еще ближе к стандартному варианту Java, поскольку обеспечивает поддержку аплетов. В табл. 4 сведены Java-пакеты, кратко пояснено их назначение и указан статус применительно к `PersonalJava`. Статус «модифицированный» означает, что пакет изменен с учетом специфики встроенной платформы. Например, в оконном инструментарии java.awt необходимо принять во внимание малые размеры и небольшую разрешающую способность дисплеев. Пакеты, помеченные как дополнительные, позволяют осуществлять доступ к аппаратным компонентам, таким, например, как таймер.

На базе `PersonalJava` функционирует `Personal WebAccess` — компактный, но достаточно богатый возможностями Web-навигатор, поддерживающий HTML версии 3.2, фреймы, таблицы и т.п. Таким образом, пользователи устройств с `PersonalJava` имеют массу возможностей, которые еще недавно были недоступны даже на мощных компьютерах.

Для более детального знакомства с `PersonalJava` можно рекомендовать статью[4].

3.2. Java в корпоративной среде

На наш взгляд, вопрос о том, насколько широко и где именно имеет смысл использовать Java в корпоративной среде, является спорным. Главными аргументами «за» широкое использование служат безопасность Java-программ и их мобильность. Главный аргумент «против» — относительно низкая скорость выполнения, неприемлемая для систем с высоким уровнем загрузки.

3.2.1. Корпоративные программные интерфейсы

Несмотря на все споры, Java-окружение времени выполнения для корпоративных платформ создано и постоянно обогащается, пополняясь новыми интересными элементами. Основными частями этого окружения являются:

- **Enterprise JavaBeans (EJB)**. Это компонентная объектная среда в корпоративном варианте, с компонентами не только на клиентской, но и на серверной сторонах. В идейном плане это центральная часть, с которой связаны надеж-

Пакет	Статус	Назначение
java.applet	Обязательный	Создание аплетов
java.awt	Модифицированный	Абстрактный оконный инструментарий
java.awt.datatransfer	Обязательный	Передача данных между приложениями
java.awt.event	Обязательный	Управление событиями
java.awt.image	Обязательный	Обработка изображений
java.beans	Обязательный	Программный интерфейс компонентной объектной модели JavaBeans
java.io	Модифицированный	Ввод/вывод. Поддержка файлов не обязательна
java.lang	Обязательный	Базовые классы
java.lang.reflect	Обязательный	Поддержка интроспекции
java.math	Необязательный	Целочисленная и вещественная арифметика произвольной точности
java.net	Обязательный	Сетевая инфраструктура
java.rmi	Необязательный	Удаленный вызов методов
java.rmi.dgc	Необязательный	Распределенный сбор мусора
java.rmi.registry	Необязательный	Каталог удаленных объектов
java.rmi.server	Необязательный	Серверная сторона удаленного вызова методов
java.security	Необязательный	Механизмы безопасности
java.security.acl	Неподдерживаемый	Списки управления доступом
java.security.interfaces	Необязательный	Управление ключами для цифровой подписи
java.sql	Необязательный	Реализация JDBC
java.text	Обязательный	Средства интернационализации
java.util	Обязательный	Служебные механизмы для других пакетов (например, Enumerator)
java.util.zip	Необязательный	Средства сжатия данных
com.sun.awt	Дополнительный	Оконный инструментарий для встроенных систем
com.sun.lang	Дополнительный	Вариант базовых классов для встроенных систем
com.sun.util	Дополнительный	Вариант служебных механизмов для встроенных систем

Табл. 4. Назначение и статус пакетов в PersonalJava.

ды на повторное использование кодов и создание приложений путем сборки из готовых компонентов.

- **Java Naming and Directory Interface (JNDI)**. Это стандартный программный интерфейс к корпоративной службе каталогов. Может использоваться, например, для поиска контейнеров и компонентов Enterprise JavaBeans.
- **Java Interface Definition Language (IDL)**. Это средства обеспечения совместимости с распределенной объектной моделью CORBA (см. [5]). Сюда входят компилятор из IDL в Java, а также облегченная реализация брокера объектных запросов, поддерживающая протокол IIOP.
- **Java Remote Method Invocation (RMI)**. Это средства для создания объектов, допускающих вызов своих методов из другой виртуальной Java-машины. Совокупность IDL и RMI позволяет строить на основе Java распределенные объектные системы.

- **Java Message Service (JMS)**. Это программный интерфейс для работы с сообщениями, позволяющий организовать очереди сообщений, реализовать распространение информации по схеме публикация/подписка и т.д.
- **Java Mail**. — программный интерфейс к почтовому протоколу SMTP. Позволяет встраивать в приложения почтовые средства.
- **Java Management API (JMAPI)**. — инструментарий для построения систем сетевого и прикладного администрирования. Поддержка протокола SNMP позволяет интегрировать JMAPI с другими системами управления, такими, например, как Tivoli Management Environment (TME).
- **Java Transaction API (JTA), Java Transaction Service (JTS)**. — механизмы высокоуровнего, основанного на открытых стандартах управления транзакциями в распределенной среде.
- **Java DataBase Connectivity (JDBC)**. — программный интерфейс для доступа к реляционным базам данных. Служит единой основой для реализации средств доступа более высокого уровня.

Сюда же следует отнести спецификации JSQL по непосредственному встраиванию SQL-операторов в Java. Первой реализовала эти спецификации компания Oracle.

Еще один важный механизм — механизм сервлетов (servlets), позволяющий динамически расширять функциональность серверов и получивший наименование по аналогии с аплетами, расширяющими функциональность клиентских систем, мы рассмотрим в разделе «Java на серверной стороне». Пока же остановимся на архитектурном аспекте использования Java в корпоративной среде.

3.2.2. Архитектура корпоративных систем, использующих Java

В статьях [6] и [7] описан опыт развертывания систем на основе Java-технологии в компании Sun Microsystems. Весьма показательно и поучительно, что Sun придерживается комплексного подхода, рассматривая Java как часть современных информационных технологий. Это проявляется, в частности, в архитектуре корпоративных Java-систем.

В статьях [8] и [9] были детально рассмотрены аппаратно-программные архитектуры с различным числом уровней, обоснованы достоинства трехуровневой архитектуры и «тонкого» клиента. Применительно к Java-технологии стандартное распределение функций по трем уровням и связи между ними показаны на рис. 1.

Приведенная на схема работоспособна в принципе, но в ситуации, когда клиенты и серверы приложений связаны (или разделены?) низкоскоростной глобальной сетью, задержки в работе пользователей, вызванные загрузкой аплетов или удаленным вызовом методов, становятся недопустимо большими. Примечательно, что в статье [9] вице-президент Sun Microsystems Уильям Дж. Радучел указывал на аналогичные проблемы при построении корпоративных систем в трехуровневой архитектуре, хотя тогда речи о Java-технологии еще не было.

Имеется еще несколько причин для развития трехуровневой модели и внесения в нее дополнительных компонентов. Компания Sun Microsystems выдвинула идею вебтопов (Webtop) — логических сетевых компьютеров (см. [6]), обеспечивающих работу пользователей в любом месте, в любое время, с любого физического устройства, обладающего достаточной функциональностью. Для реализации этой идеи необходимы серверы вебтопов, хранящие конфигурационную информацию и обеспечивающие начальную загрузку. Такие серверы должны располагаться поблизости от клиентов (в той же локальной сети) не только по соображениям эффективности, но и для лучшей управляемости. В такой ситуации естественно возложить на серверы вебтопов еще два вида действий:

- обслуживание запросов на локальные действия (доступ к файлам, печать и т.п.);
- кэширование программ и данных, запрашиваемых с удаленных серверов.

В результате получается уточненная четырехуровневая архитектура корпоративных систем на базе Java-технологии, показанная на рис. 2.

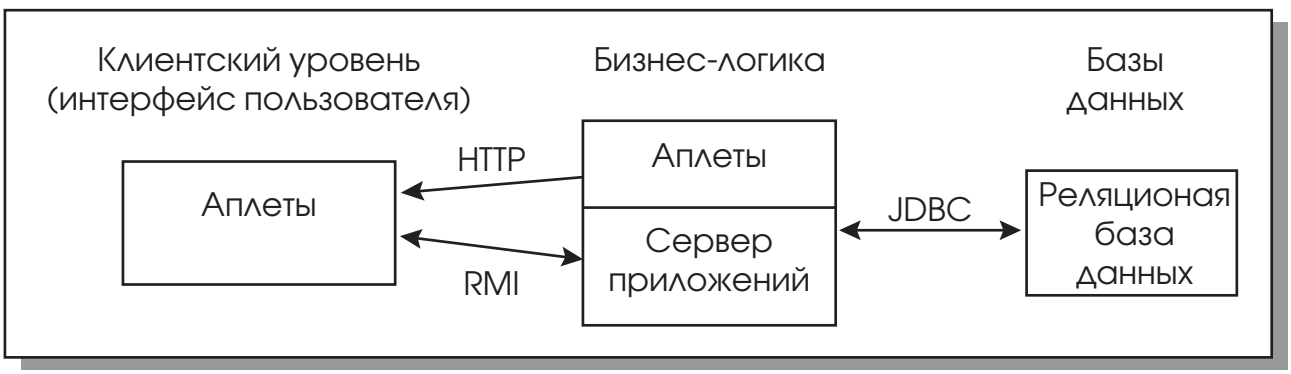

Рис. 1. Стандартная реализация трехуровневой архитектуры в Java-технологии.

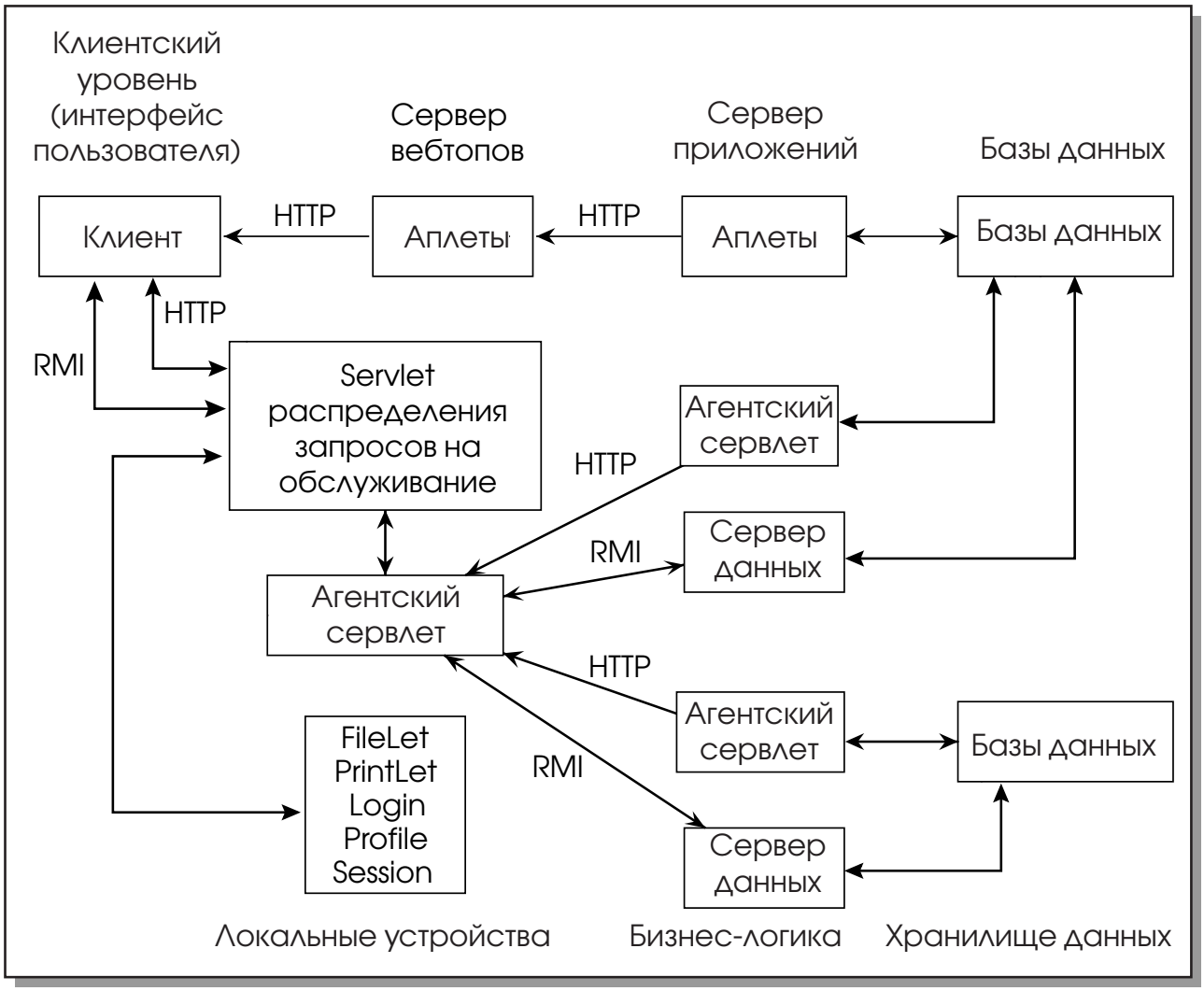

Рис. 2. Уточненная четырехуровневая архитектура корпоративных систем на базе Java-технологии.

Такая архитектура позволяет решить еще одну важную проблему — избежать конфликтов с моделью безопасности Java, разрешающей аплетам устанавливать сетевые соединения только с тем хостом, откуда аплет был загружен. Первоначально в статье [6] предлагалось ослабить требования безопасности, введя понятие «надежного аплета», возможности которого выходят за стандартные рамки. Надежный аплет должен быть подписан электронной подписью субъекта, пользующегося доверием. Введение уровня серверов вебтопов с соответствующей функциональностью дало возможность сохранить концептуальную целостность, не прибегая к трюкам вроде заверения аплетов.

В уточненной четырехуровневой архитектуре Java-компоненты функционируют и на клиентской, и на серверной сторонах, однако, как нам представляется, важнее всего то, что Java играет здесь роль программного обеспечения промежуточного слоя, «склеивая» разные уровни. Для ПО промежуточного слоя очень важен интеграционный потенциал, которым Java обладает в полной мере. На наш взгляд, мониторы транзакций и аналогичное ПО — идеальная точка приложения Java-технологии.

3.3. Java на серверной стороне

Первоначально Java трактовалась исключительно как технология для клиентских частей информационных систем, однако сейчас становится понятным, что она с равным успехом может применяться и на серверной стороне.

Сам по себе язык Java универсален и может использоваться для написания любых программ, но чтобы сделать Java-технологию «серверо-пригодной» необходимы два дополнительных механизма:

- программные интерфейсы с серверной ориентацией;
- средства расширения функциональности серверов, не нарушающие безопасность программных систем.

Эти механизмы были созданы, и мы переходим к их рассмотрению.

3.3.1. Программные интерфейсы с серверной ориентацией

Дадим несколько определений.

Сервис. Это реализация серверной части какого-либо протокола прикладного уровня, такого как HTTP, FTP и т.п.

Сервер. Это процесс в смысле операционной системы, содержащий виртуальную Java-машину, в которую загружены классы, обеспечивающие работу одного или нескольких сервисов.

Сервлет. Это класс, стандартным образом расширяющий функциональность какого-либо сервиса.

Точка подключения к сервису. Это точка, в которой сервис «слушает» запросы на обслуживание. Одному сервису может соответствовать несколько точек подключения.

Обработчик запросов. Это объект, предназначенный для обслуживания запроса, поступившего через некоторую точку подключения к сервису. Одной точке соответствует группа потоков, в каждом из которых выполняется один обработчик.

На рис. 3 показаны логические связи между введенными понятиями.

Функционирование на серверной стороне устроено следующим образом. Серверный процесс конфигурируется для автоматического запуска набора сервисов. Обычно сервер поддерживает административный сервис (для собственного динамического конфигурирования), а также несколько прикладных сервисов.

Каждому сервису выделяется своя группа потоков. Сервис инициализирует свои точки подключения, создает необходимое число потоков и создает в их рамках объекты-обработчики. Обработчики входят в бесконечный цикл ожидания и обслуживания запросов. Размер пула потоков может динамически изменяться в зависимости от нагрузки на сервис (сервисы, как и серверы, поддерживают административный интерфейс). Могут создаваться новые потоки и ликвидироваться простаивающие. Между точкой подключения и обработчиками существует промежуточное звено диспетчеризации, распределяющее поступающие запросы (рис. 4).

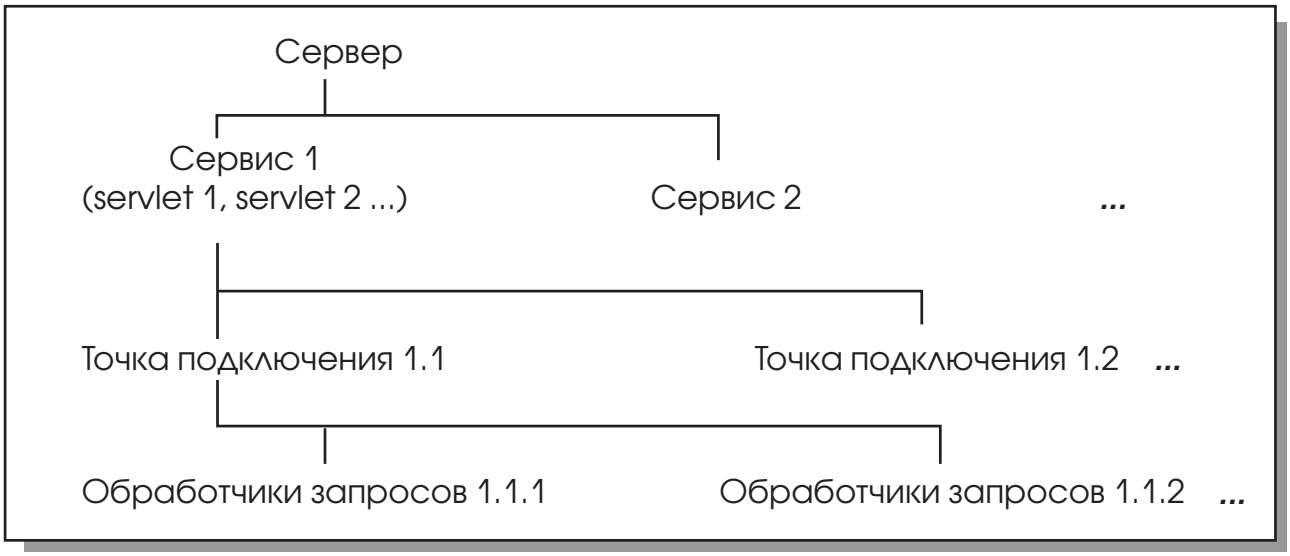

Рис. 3. Иерархия понятий в модели JavaServer.

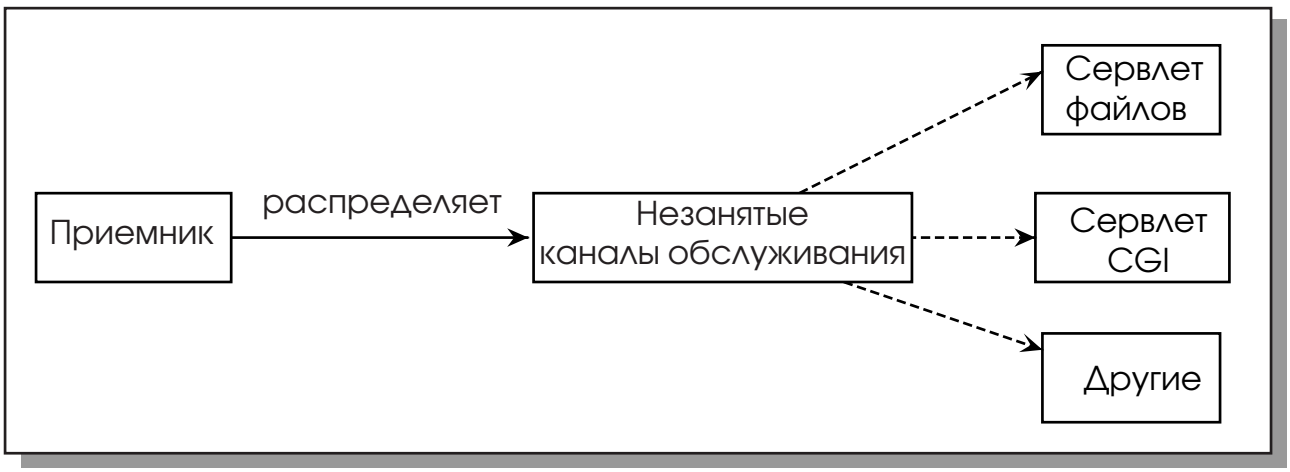

Рис. 4. Диспетчеризация запросов.

Для расширения возможностей сервисов используются сервлеты. Часть из них автоматически загружается при старте сервисов (в соответствии с конфигурацией последних), другие могут добавляться динамически. Во многих отношениях сервлеты аналогичны аплетам, но им не нужен графический интерфейс, они являются «аплетами без лица». На рис. 4 показано несколько сервлетов, обслуживающих специфические запросы.

Наличие механизма сервлетов — одно из главных достоинств серверной модели Java, делающее эту модель по-настоящему объектной. Появилась принципиальная возможность разделить логически независимые сущности, такие, например, как HTTP-сервис, итератор HTML-документов и средство поиска в одном документе, и затем комбинировать их в соответствии с потребностями пользователей, избегая дублирования программного кода. Точно так же можно организовать конвейеры в духе ОС Unix, состоящие их набора фильтров для преобразования передаваемых объектов к виду, удобному пользователю. Реализованный в JavaSoft Java Web Server — пример такого «правильного» сервера.

Каждому понятию серверной модели соответствует Java-сущность — интерфейс, класс или объект. Кратко опишем их.

Большинство интересующих нас сущностей помещено в пакет `sun.server`. Это классы `ServerProcess`, `Service`, `EndpointDescriptor`, `ConnectionEndpoint`, `ServiceHandler`.

Класс `ServerProcess` реализует серверный процесс. Упомянем методы `getAutoStartServiceNames()` (получить список автоматически запускаемых сервисов), `getProperties()` (получить свойства, ассоциированные с сервером), `getThreadGroup (String)` (получить группу потоков для сервиса), `isServiceRunning (String)` (проверить, выполняется ли сервис в рамках текущего серверного процесса), `restartService (String, int)` (перезапустить сервис в рамках текущего серверного процесса), `startService (String)` (запустить сервис с добавлением его к списку автоматически запускаемых), `stopService (String, boolean)` (остановить сервис и удалить его из списка автоматически запускаемых).

В классе `Service` обратим внимание на ряд переменных, описывающих конфигурацию и состояние сервиса (`serviceConfig`, `serviceRoot`, `params`, `port`, `minThreads`, `maxThreads`, `shutdownComplete`), а также на методы `init (ServiceConfiguration, ServiceConfiguration)` (инициализировать сервис с заданной конфигурацией), `getEndpoint()` (получить точку подключения), `createHandler()` (создать новый обработчик).

`EndpointDescriptor` — это абстрактный класс, предназначенный для администрирования сетевых сервисов. Каждый сервис доступен по определенному адресу транспортного уровня и с определенными характеристиками, для работы с которыми и служит класс `EndpointDescriptor` и его преемники, такие как `ConnectionEndpoint`. В частности, точки подключения могут соответствовать сервисам с установлением соединения, а также датаграммным и многоадресным сервисам. В классе `EndpointDescriptor` отдельного упоминания заслуживают методы `initialize (Properties)` (инициализировать точку подключения), `getAdminAppletClass()` (получить имя класса, объекты которого позволяют администрировать текущую точку подключения), `saveProperty(String, String)` (сохранить значения ключевого свойства).

Класс `ConnectionEndpoint` соответствует точкам подключения с установлением соединения (что на момент написания статьи являлось синонимом TCP-точек). Отметим два варианта метода `getServerSocket`, служащего для получения сокетов, используемых в коммуникациях с клиентами. Скорее всего, в реальных программах у класса будут свои преемники, конкретизирующие и дополняющие его методы.

Абстрактный класс `ServiceHandler` — это заготовка для создания собственных обработчиков. Благодаря архитектурным решениям, каждый обработчик функционирует в рамках отдельного потока и получает запросы только тогда, когда он свободен, так что он свободен и от проблем, связанных с обслуживанием последовательности асинхронно поступающих запросов. Это упрощает программирование обработчиков и повышает их надежность. Из методов класса `ServiceHandler` отметим два — `getBuffer()` (получить буфер, принадлежащий исключительно данному потоку) и `handleConnection(Socket)` (отработать одно соединения с клиентом).

Основными элементами иерархии, составляющей каркас реализации сервлетов, являются интерфейс `Servlet` и абстрактный класс `GenericServlet`. Интерфейс `Servlet` содержит три метода, поддерживающие жизненный цикл сервлета — `init (ServletConfig)` (инициализация), `service (ServletRequest, ServletResponse)` (обслуживание запроса) и `destroy()` (завершение), а также два информационных метода (`getServletConfig()` и `getServletInfo()`). Ключевую роль играет метод `service`, именно его будут определять и переопределять авторы сервлетов независимо от того, какой класс они возьмут в качестве предшественника.

Важнейшим частным случаем сервиса является HTTP. Для него определена конкретизация (остающаяся, впрочем, абстрактной) класса `GenericServlet` — `HttpServlet`. Этот класс предоставляет законченную реализацию метода service, учитывающую специфику протокола HTTP. Возможно, разработчикам сервлетов придется уточнить один из методов обработки HTTP-

запросов (GET, POST, PUT, DELETE) или какой-либо из методов жизненного цикла (init, destroy).

Абстрактные классы GenericServlet и HttpServlet играют роль заготовок, предоставляющих разумные реализации методов жизненного цикла и опроса/установки конфигурационных данных. Зачастую при наследовании этих класса достаточно переопределить только один-два метода. Наличие таких «почти конкретных» классов — важное достоинство серверной архитектуры Java.

Если вернуться к рис. 3, можно увидеть, что сервлеты функционируют в рамках сервисов (а не обработчиков), то есть в многопотоковой среде. Следовательно, при реализации сервлетов необходимо заботиться о синхронизации доступа к разделяемым ресурсам. Разумеется, синхронизация на уровне запросов (браться за следующий запрос только после завершения предыдущего) является слишком грубой и практически неприемлемой. Необходимы более тонкие методы и, что еще важнее, общие рекомендации по потоковой безопасности сервлетов.

Мы весьма кратко и фрагментарно описали программные интерфейсы с серверной ориентацией. Специалистами JavaSoft вопрос этот проработан гораздо глубже и детальнее. Определено большое количество интерфейсов и классов, облегчающих администрирование и реализацию серверов. В результате разработчики оказываются в весьма комфортных условиях.

3.3.2. Информационная безопасность на серверной стороне

На серверной стороне проблема информационной безопасности имеет две грани:

- безопасное выполнение ненадежных сервлетов;
- разграничение доступа к серверным ресурсам.

Первая проблема решается обычным для Java-технологии способом — заключением потенциально ненадежного элемента в «песочницу» (подробнее об этом см. далее в разделе «Эволюция модели безопасности». В данном случае в качестве «песочницы» выступает группа потоков. Это означает, помимо прочего, что вызов ненадежных сервлетов будет более медленным, чем в надежном случае, за счет более медленного обмена между потоками, принадлежащими к группам «надежных» и «ненадежных».

Разграничение доступа к серверным ресурсам также реализовано традиционным образом. Выделяется два этапа:

- идентификация/аутентификация субъектов доступа;
- проверка прав доступа субъектов к объектам.

Теоретически в Java-системе может использоваться любая схема аутентификации, однако на момент написания статьи реально поддерживалась лишь проверка подлинности в стиле ОС Unix. Впрочем, это чисто технический момент, допускающий последующее развитие и модификацию.

Для удобства администрирования пользователи могут объединяться в группы, как это делается в той же ОС Unix. Естественно, и пользователям, и группам соответствуют объекты, хранящие необходимую информацию.

Права пользователей определяются списками управления доступом. Существуют именованные списки, ассоциированные с конкретными объектами, а также общий список, описывающий права «по умолчанию».

Среди программных интерфейсов, связанных с разграничением доступа, центральное место занимает абстрактный класс sun.server.realm.Realm, описывающий область управления правами доступа. Область характеризуется набором пользователей, групп, ресурсов и применяемым протоколом аутентификации. Упомянем методы addAcl (String, Principal) (создать список с данными именем и владельцем), getAcl() (получить список доступа к административным привилегиям), getAcl (String) (получить список доступа с заданным именем), getUserNames() (получить список всех пользователей области), addGroup (String) (создать новую пустую группу с заданным именем).

Рис. 5. Афиширование сервиса в технологии Jini.

Для представления списков управления доступом используется класс `sun.security.acl.AclImpl`. Среди его методов следует выделить `AclImpl(Principal, String)` (конструктор, создающий список с заданными владельцем и именем), `addEntry (Principal, AclEntry)` (добавить к списку элемент по просьбе данного пользователя), `getPermissions (Principal)` (получить права доступа данного пользователя), `entries()` (получить итератор по элементам списка).

Таким образом, перед нами объектно-ориентированная реализация привычной схемы произвольного управления доступом[4] .

3.4. Jini

Jini — это исследовательский проект, преследующий цель кардинального упрощения разработки и администрирования распределенных систем, обладающих высокой степенью динамичности. Его инициатором стал один из основателей компании Sun Microsystems Билл Джой (Bill Joy).

С точки зрения реализации Jini представляет собой библиотеку классов, а также ряд спецификаций для создания и поддержки «федерации» виртуальных Java-машин, рассредоточенных по сети.

В сети выделяется система, играющая роль электронной доски объявлений. Каждый, кто хочет афишировать какой-либо сетевой сервис, вешает на эту доску объявление со своими координатами, Java-интерфейсом сервиса и другими параметрами. Кроме того (и это очень важно), на доску помещается агентская часть сервиса (рис. 5).

Если клиент нуждается в некотором сервисе, он запрашивает его, обратившись к доске объявлений и указав нужный Java-интерфейс, а также, быть может, дополнительные параметры (рис. 6).

В ответ на свой запрос клиент получает агентскую часть сервиса, которая и осуществляет взаимодействие с сервером (рис. 7). В общем случае агент предполагается интеллектуальным, то есть часть функций он может выполнять на клиентской системе, а за другими — обращаться к серверу. По сути агент представляет собой аплет, который, однако, попадает в клиентскую систему не прямо с

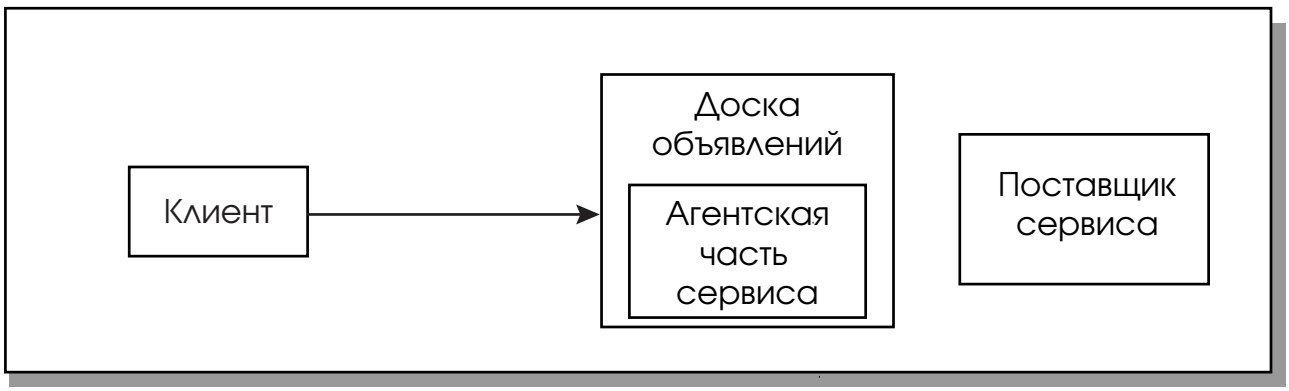

Рис. 6. Запрос сервиса в технологии Jini.

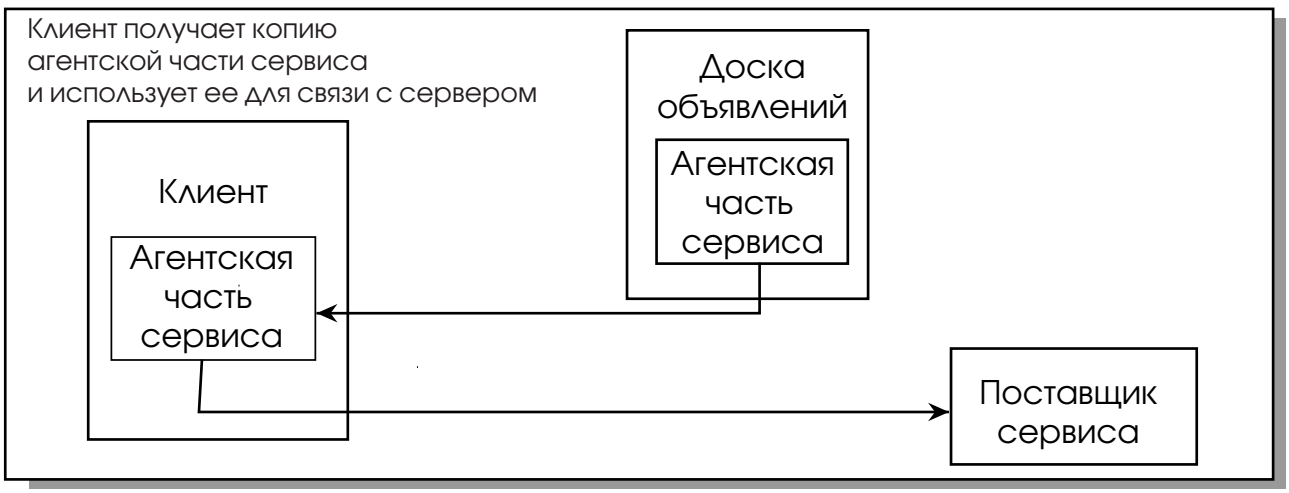

Рисунок 7. Вызов сервиса в технологии Jini.

[4] *Напомним, что под произвольным уаправлением доступом мы понимаем такую организацию, при которой ограничением доступа к объекту занимается сам владелец объекта.*

сервера, а через доску объявлений. В роли агента может выступать драйвер сетевого принтера, аплет для организации запросов к базе данных и т.п.

Разумеется, прежде чем разместить свою информацию на электронной доске объявлений, необходимо выяснить, где эта доска находится. Для этого система, инициализирующаяся в сетевом окружении, посылает многоадресный запрос, в ответе на который должны содержаться требуемые сведения. Получив их, система выполняет описанное выше афиширование сервиса, присоединяясь тем самым с «сетевой федерации». Теперь о сервисе знают и могут им воспользоваться.

Аналогичные действия по выяснению выполняют в начале своей работы и клиентские системы. В принципе они также могут поместить на доску объявлений свои характеристики, что может оказаться полезным для настройки сервисов. Вся процедура инициализации системы в сетевом окружении получила в технологии Jini название «выяснить и присоединиться». Процедура устроена таким образом, что система, ничего не зная о среде, в которой ей предстоит работать, быстро получает необходимые сведения, а также сообщает данные о себе, «присоединяясь к компании».

Над инфраструктурным уровнями выяснения/присоединения и поиска сервисов располагается прикладной уровень со стандартными сервисами. Пока прикладной уровень только формируется и включает `JavaSpaces` (средство для программирования в терминах потоков объектов, подробнее см. [10]), а также облегченный монитор двухфазного управления транзакциями. Получающаяся иерархия изображена на рис. 8.

Специалисты компании Sun Microsystems отлично понимают, что распределенное программирование на порядок сложнее обычного, локального. Разные части распределенной системы могут внезапно «умирать», между ними может рваться связь, острее встают проблемы информационной безопасности. Поэтому, кроме базовой инфраструктуры и приложений, было предложено несколько новых понятий и механизмов, представляющихся весьма перспективными.

Первым среди них следует назвать понятие аренды (см. [11]). В технологии Jini при запросе ресурса последний становится доступным не навсегда, а на согласованный между владельцем и арендатором срок («заключается договор об аренде»). Например, арендуется место на доске объявлений, где система размещает сведения о себе. Этот договор накладывает обязательства на обе стороны. В частности, владелец обязуется всячески способствовать сохранению доступности ресурса. По окончании срока ресурс автоматически освобождается (но арендатор, разумеется, может заранее попытаться продлить договор). Таким образом, даже если с арендатором что-то случится, ресурс не останется навсегда в собственности у «мертвой души». Путем управления сроками аренды можно добиться необходимого баланса между доступностью ресурсов и уровнем накладных расходов на организацию и продление аренды.

Процедуру аренды специфицирует интерфейс `Lease`. Обратим внимание на методы `renew(long)` (заявка на продление аренды) и `cancel()` (отказ от аренды).

С понятием аренды в распределенное программирование входит время, а сетевые системы становятся адаптивными, автоматически приспосабливающимися к изменениям в окружении. Конечно, сходные понятия предлагались и раньше, в рамках других разработок. Можно вспомнить, например, билеты системы Kerberos (см. [12]) или данные о маршрутизаторах в IPv6 (см. [13]). Тем не ме-

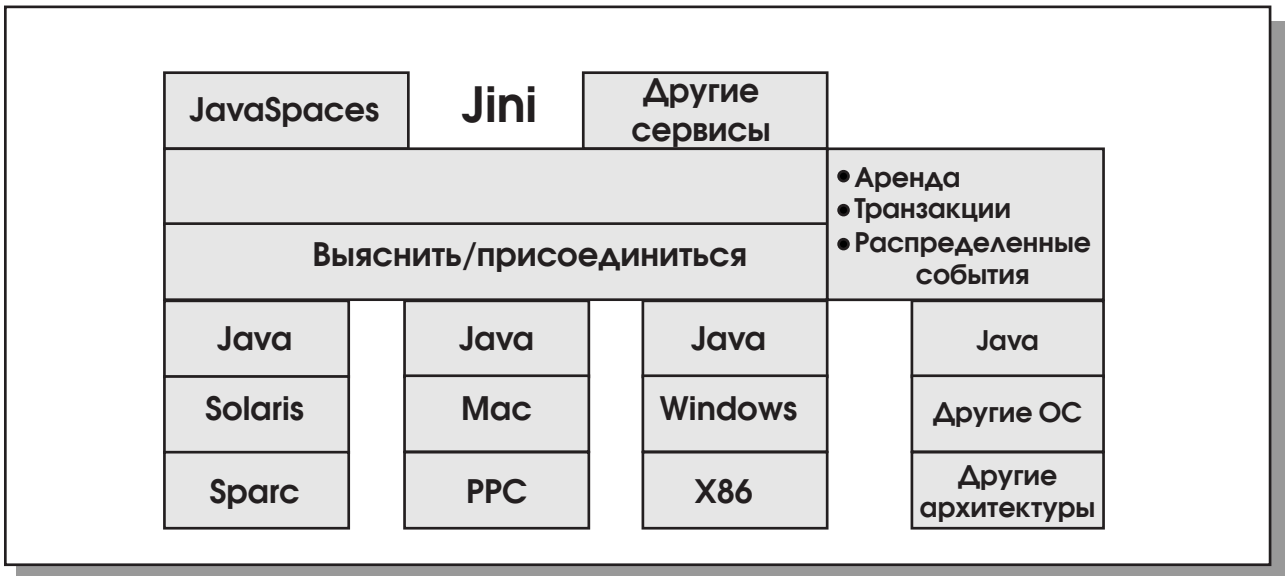

Рис. 8. Иерархия компонентов в технологии Jini.

нее, аренда, на наш взгляд, является полезным обобщением, а его систематическая поддержка способна радикальным образом повысить надежность программных систем, в особенности распределенных.

Еще одним обобщением, хотя и менее масштабным, является распределенный механизм событий (см. [14]). Здесь специфика распределенности состоит в том, что данные о событиях могут доставляться долго, ненадежно, с нарушениями хронологического порядка. Чтобы справиться с этими проблемами, целесообразно трактовать как, вообще говоря, различные сущности, объект, интересующийся событиями, и объект, получающий данные о них. Последний может организовать промежуточное хранение, упорядочивание, фильтрацию поступающих данных, освобождая «хозяина» от этой рутины (тяготеющей к транспортному, а не прикладному уровню). Наличие объектов-посредников позволяет организовать конвейер из фильтров, реализуя тем самым одну из самых красивых и важных идей ОС Unix.

В статье [2] как недостаток отмечалась неструктурированность пространства событий в модели JavaBeans. Распределенный механизм событий Jini в общем и целом построен по образцу JavaBeans, однако в нем введено понятие вида события, в соответствие которому может быть поставлен единый обработчик.

Предложенный в рамках Jini механизм транзакций носит достаточно традиционный характер и мы не будем на нем останавливаться. Для желающих получить более детальную информацию о Jini можно рекомендовать спецификацию [15] и статью [16].

В целом знакомство с Jini рождает противоречивые чувства. Пожалуй, ни одну из предлагаемых в рамках Jini идей нельзя назвать по-настоящему новой (что, впрочем, для промышленной системы недостатком не является). Есть сомнения в том, что облегченная реализация службы каталогов — доски объявления в Jini — обладает достаточной масштабируемостью. Авторы Jini указывают, что одна доска объявлений способна обслуживать до тысячи пользователей, что по современным меркам немного.

Трудно представить себе, что столь простыми средствами можно совершить переворот в распределенном программировании. Но, как говорится, дай бог, чтобы мы ошиблись.

3.5. Swing

Для Java-платформы определен набор базовых классов (Java Foundation Classes, JFC), являющихся обязательной частью окружения на клиентских системах и обслуживающих соответствующий архитектурный уровень корпоративных приложений. Главными элементами JFC являются:

- средства построения пользовательского графического интерфейса;
- средства поддержки двумерной графики;
- средства печати и перемещения данных между приложениями.

Мы остановимся на наиболее интересном, как нам кажется, элементе — средствах построения пользовательского графического интерфейса, получивших наименование Swing.

3.5.1. Идеи, положенные в основу Swing

Современный пользовательский интерфейс должен обладать по крайней мере следующими качествами:

- привлекательный внешний вид, привычный для пользователей;
- достаточная эффективность, исключающая заметные задержки;
- технологичность (возможность расширения, настройки и т.п).

Реализации, основанные на абстрактном оконном инструментарии (Abstract Windowing Toolkit, AWT), на наш взгляд, не обладали в должной мере ни одним из перечисленных качеств. Интерфейс Java-приложений выглядел простовато (хотя, конечно же, был функционально достаточным), реакция на движения мыши и нажатия кнопок казалась вязкой даже на старших моделях персональных компьютеров, а перенастройка с учетом специфики новой аппаратно-программной платформы (то есть по сути обеспечение мобильности программ) представлялась проблематичной. Разработчики Swing поставили перед собой цель разрешить выявившиеся проблемы.

Поскольку трудности с пользовательским интерфейсом в Java носили концептуальный характер, для их кардинального решения пришлось во многом пересмотреть понятийные основы и архитектуру соответствующих элементов.

Во-первых, разработчики Swing решили ориентироваться на компонентную модель JavaBeans и ассоциированные механизмы (в первую очередь имеется в виду механизм событий). Это сделало Swing технологичным и дало возможность воспользоваться интегрированной средой разработки, уже созданной для JavaBeans. Запрограммированные на 100%-чистой Java многочисленные и очень красиво сделанные меню, диалоговые панели, кнопки и т.п. представляют собой накопленное программистское и дизайнерское знание, наделенное мобильной реализацией.

Очень важной с идейной точки зрения является ориентация разработчиков Swing на парадигму Модель/Представление/Управление (Model/View/Controller, MVC). Суть ее в том, что все компоненты можно разделить на три категории (см. рис. 9). Компоненты вида «Модель» определя-

ют состояние и логику поведения объекта. Компоненты вида «Представление» (*View*) отвечают за отображение модели для внешнего мира (например, на дисплее компьютера). Компоненты вида «Управление» (*Controller*) обеспечивают получение моделью сигналов из внешнего мира (например, команды, вводимые пользователем).

В статье [2], в разделе «Агрегирование интерфейсов» обсуждалась концепция многогранной организации объектов в Java, подчеркивалась ее важность. К сожалению, пока она не играет сколько-нибудь заметной роли. Это привело к постепенной трансформации первоначальной интерфейсной парадигмы, не обеспеченной программной поддержкой. Сначала в умах разработчиков Swing объединились внешний вид и управление. В результате получилась схема, показанная на рис.10.

Данная схема остается достаточно содержательной. Она позволяет провести декомпозицию интерфейса и управления еще на два уровня, из которых нижний зависит от стиля организации интерфейса (Windows, Motif и т.д.), а верхний является платформно-независимым. Отметим, что разработчики Swing оформили и свой интерфейсный стиль, получивший название `Metal`.

Фактически при реализации Swing было произведено дальнейшее упрощение приведенной схемы и объединение модели и интерфейсной части. Выделение платформно-независимого уровня осталось, что является важным технологическим достоинством.

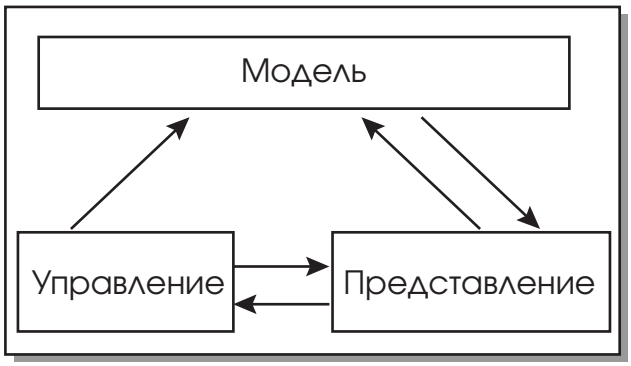

Рис. 9. Три категории интерфейсных компонентов – модель, представление и управление.

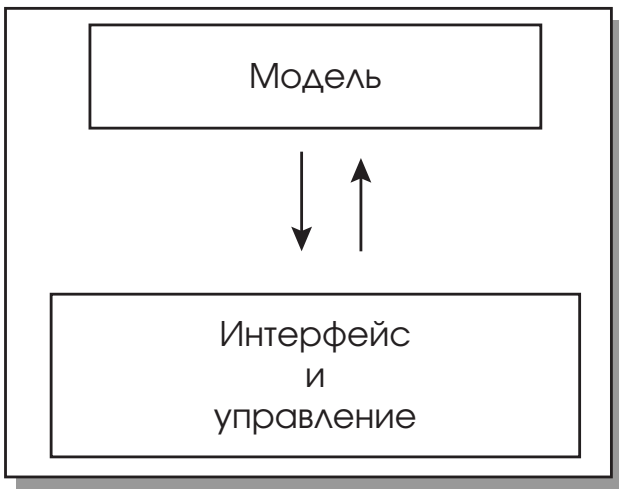

Рис. 10. Две категории компонента – модель и пользовательский интерфейс.

3.5.2. Объектная организация Swing

С технической точки зрения Swing представляет собой набор пакетов, сведенных в табл. 5. На момент написания статьи их место в Java-иерархии не было окончательно определено (обсуждались по крайней мере две возможности — com.sun.java и javax), поэтому в таблице вместо префикса поставлено многоточие, а в последующем тексте используются краткие имена пакетов.

Корнем объектной иерархии компонентов в Swing является абстрактный класс `JComponent` — «внук» `Component` и «сын» `Container` из AWT. Преемники этого класса могут играть роль и компонента, и контейнера, что типично для объектной интерфейсной среды.

Пакет	Назначение
`...swing`	Содержит интерфейсы и классы, определяющие множество реализованных на Java интерфейсных компонентов, обладающих высокой платформной независимостью.
`...swing.border`	Поддерживает рисование разного рода бордюров вокруг Swing-компонентов
`...swing.event`	Обслуживание событий в Swing-компонентах
`...swing.plaf`	Абстрактные классы для поддержки различных стилей интерфейса
`...swing.table`	Поддержка интерфейсного компонента «таблица»
`...swing.text`	Поддержка текстовых компонентов, в том числе редактируемых
`...swing.text.html`	Классы для создания редакторов HTML-текстов
`...swing.text.rtf`	Классы для создания редакторов RTF-текстов
`...swing.tree`	Поддержка интерфейсного компонента «дерево»
`...swing.undo`	Поддержка средств откатки для редактирующих приложений

Табл. 5. Пакет Swing.

JComponent определяет наиболее общие черты в поведении интерфейсных компонентов. Поле listenerList хранит список подписчиков событий в компоненте, поле ui — представление компонента в соответствии с выбранным стилем интерфейса (подробнее об этом в следующем пункте, «Настройка стиля интерфейса»). Конечно, это защищенные поля, манипулировать которыми можно только посредством вызова методов.

Методы класса JComponent можно разбить на несколько групп:

- рисование (paint, paintChildren, paintImmediately, repaint и т.п.);
- манипуляции с геометрическими характеристиками компонента (computeVisibleRect, contains, getAlignmentX, getHeight, reshape, setMaximumSize и т.п.);
- манипуляции с прочими характеристиками компонента (getComponentGraphics, getConditionForKeyStroke, hasFocus, isFocusTraversable и т.п.);
- обслуживание событий (addAncestorListener, firePropertyChange, registerKeyboardAction, processFocusEvent и т.п.);
- обслуживание компонентной иерархии (getTopLevelAncestor, getRootPane, getBorder, getInsets, getNextFocusableComponent и т.п.);
- манипуляции с подсказкой, которая появляется при движении мыши по компоненту (createToolTip, getToolTipText, setToolTipText и т.п.);
- операции со стилем интерфейса (setUI, updateUI, getUIClassID).

Мы видим, что функциональность интерфейсного компонента изначально сделана достаточно богатой, что существенно облегчает разработку новых компонентов.

Преемниками JComponent являются многочисленные абстрактные и обычные классы, названия которых говорят сами за себя: AbstractButton, JColorChooser, JFileChooser, JInternalFrame, JLabel, JList, JMenuBar, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JTable, JTextComponent, JToolBar, JToolTip, JTree и др.

Рассмотрим в качестве примера обобщенную реализацию кнопки — абстрактный класс AbstractButton. Наиболее интересным новым (по сравнению с JComponent) полем в нем является поле model — модель, ассоциированная с кнопкой. Соответственно, появились методы getModel и setModel. Кроме этого, присутствуют методы опроса/установки атрибутов — getIcon, getText, getActionCommand, isSelected, setSelected, setEnabled и т.п. Разумеется, присутствует и метод для программного нажатия кнопки — doClick.

Параллельно с иерархией интерфейсных компонентов существует иерархия их моделей. Правда, у последней нет содержательного корня (модели для JComponent), модели сразу подразделяются по видам, объединяющим сходные компоненты. Например, модель кнопки описывается интерфейсом ButtonModel. Эта модель соответствует обычным кнопкам, радиокнопкам и спискам выбора. Модель BoundedRangeModel предназначена для компонентов JProgressBar и JSlider, изображающих (хотя и по-разному) значение на заданном отрезке.

Рассмотрим подробнее интерфейс ButtonModel. С удивлением мы обнаруживаем, что большинство методов этого интерфейса продублировано в классе AbstractButton. В первую очередь имеются в виду методы обслуживания событий, а также некоторые методы опроса/установки атрибутов, причем в последнем аспекте не видно особой систематичности: в AbstractButton есть метод setEnabled, однако его «напарник» isEnabled отсутствует. Чисто «модельными», помимо isEnabled, оказываются лишь методы isArmed, isPressed, isRollover, setArmed, setPressed, setRollover (их назначение очевидно), а также setGroup (ButtonGroup), определяющий группу, к которой принадлежит кнопка (что полезно, например, для радиокнопок, когда возможен выбор лишь одного элемента из группы).

Подобный подход, на наш взгляд, трудно одобрить. Он затушевывает важное с концептуальной точки зрения различие между графическим компонентом и его моделью (прикладная программа может подписаться на события в компоненте или его модели, может опрашивать/изменять состояние с помощью методов компонента или модели и т.п.). Соображения простоты и удобства разработки приложений здесь едва ли применимы, а эффективность может только потеряться, поскольку, разумеется, реализация методов в графическом элементе сводится к обращениям к модели.

Частичное дублирование модельных методов имеет место и для компонентов, существенно превосходящих кнопки по сложности. Например, для таблиц продублированы методы getValueAt/setValueAt, не имеющие прямого отношения к графическому представлению данного интерфейсного компонента. Правда, обслуживание событий здесь целиком вынесено в модель.

Несмотря на отдельные недостатки, разработчики Swing проделали огромную и очень полезную работу. Выполнена реализация большого числа высокоуровневых интерфейсных компонентов. Для всех моделей и многих других содержательных сущностей определены прототипные

реализации (классы DefaultButtonModel, DefaultCellEditor, DefaultDesktopManager, DefaultListCellRenderer, DefaultListSelectionModel и т.д.). Имеются абстрактные классы (уже упоминавшийся AbstractButton, а также AbstractAction и AbstractListModel), позволяющие вести разработку не «с нуля». Наконец, предложено гибкое обслуживание событий, с которым связываются основные надежды на повышение эффективности интерфейса. Суть его в том, что при наступлении простых (и часто генерируемых) событий подписчикам передаются «облегченные» объекты (класса ChangeEvent), содержащие лишь данные об источнике), а в более сложных случаях событийные объекты становятся содержательнее (ListDataEvent, TableModelEvent, DocumentEvent и т.п.), в них передается дополнительная информация, такая, например, как координаты изменившихся клеток таблицы.

3.5.3. Настройка стиля интерфейса

Под стилем интерфейса (*look and feel*) понимается внешний вид графических компонентов (*look*) и их видимая реакция на пользовательский ввод (*feel*). Для подлинной мобильности приложений необходимо реализовать две возможности:

- динамическая настройка стиля интерфейса;
- создание новых стилей интерфейса.

Чтобы достичь сформулированных целей, разработчики Swing разделили интерфейс на два уровня. В предыдущем пункте мы рассмотрели верхний уровень, не зависящий от стиля. На нем располагаются «понятийные» компоненты, такие как «кнопка вообще» и «таблица вообще». На нижнем уровне находятся «кнопка для Windows» или «таблица для Motif». Для каждого компонента верхнего уровня в поле ui (которое является ограниченным свойством в смысле JavaBeans) хранится его низкоуровневое представление, зависящее от выбранного стиля. Именно это низкоуровневое представление обеспечивает реальный ввод/вывод; методы верхнего уровня являются лишь передаточным звеном, переправляющим запросы вниз.

Когда-то были популярны шутки типа «для удобства пассажиров с 1 января ко всем номерам автобусных маршрутов справа приписывается 0». Самое смешное состоит в том, что при построении объектных систем зачастую примерно так и поступают, причем с немалой пользой. В Swing классам верхнего уровня, имена которых начинаются на «J», соответствуют классы нижнего уровня, где убрано начальное «J», но в конец добавлено «UI» (JComponent — ComponentUI, JButton — ButtonUI и т.п.). Низкоуровневые классы описаны в пакете swing.plaf (Pluggable look-and-feel, сменные стили интерфейса). Наиболее содержательным из них является ComponentUI.

Обратим внимание на метод installUI (JComponent), формирующий низкоуровневое представление компонента. Этот метод устанавливает подразумеваемые характеристики компонента — цвет, фонт, бордюр и т.п., выбирает расположение, добавляет необходимые подкомпоненты, обеспечивает подписку на события, определяет реакцию на нажатия клавиш и т.д. Еще один интересный метод, contains (JComponent, int, int), в принципе позволяет определять непрямоугольные графические компоненты. Этот метод вызывает при движении мыши, чтобы узнать, «наехала» ли она на объект. Правда, если вычисления будут слишком «хитрыми», мышь рискует «завязнуть».

Для администрирования стилей служит класс UIManager. С помощью его методов можно установить/опросить текущий стиль, но, главное, он и ряд вспомогательных классов поддерживают таблицу соответствия между именами абстрактных низкоуровневых представлений и их реализаций (например, ButtonUI - motif.MotifButtonUI). Первое имя можно получить, вызвав метод getUIClassID в компоненте верхнего уровня, а затем по таблице соответствия динамически настроиться на текущий стиль.

Для создания собственных стилей нужно реализовать абстрактные классы из пакета swing.plaf или, что существенно проще, заняться наследованием и модификацией существующих стилей. Впрочем, по мнению авторов Swing, в этом нуждается весьма небольшое число разработчиков.

Для более подробного знакомства с архитектурой Swing и, в частности, с методами создания новых стилей интерфейса можно рекомендовать статью [17].

3.6. Механизм расширений

Как бы ни была богата базовая функциональность Java-окружения времени выполнения, во многих случаях ее оказывается недостаточно. Это показывает пример и самого базового набора Java-классов. В версии JDK 1.0 таких классов было 8, в версии 1.1 — 22, а в версии 1.2 — более 50. Это напоминает динамику роста курса доллара в России и, соответственно, рождает грустные предчувствия. Очевидно, в какой-то момент следует зафиксировать базовый набор, предложив одновременно дисциплину размещения и загрузки расширений.

Под расширениями понимаются пакеты классов, написанные на Java (и, быть, может, каких-то других языках программирования), которые могут использоваться разработчиками приложений для расширения функциональности базового Java-окружения. Классы расширений могут использоваться виртуальной Java-машиной практически наравне с базовыми (некоторые нюансы будут описаны ниже).

Среди расширений полезно выделить стандартные, обладающие тремя свойствами:

- наличие полной спецификации прикладного программного интерфейса;
- наличие инструментария для проверки Java-совместимости;
- наличие эталонной реализации.

Такие стандартные расширения можно использовать наравне с базовыми классами.

С технической точки зрения расширения представляют собой один или несколько обычных JAR-файлов. Это значит, что для изготовления расширений никакого специального инструментария не требуется.

Расширения можно подразделить на установленные и загружаемые. Установленные расширения должны быть предварительно размещены в каталоге `корневой_каталог_Java/lib/ext/`.

Определены также несложные правила размещения бинарного кода, если таковой имеется.

Установленные расширения доступны всем приложениям и аплетам, выполняющимся на данной Java-машине. Они ничем не отличаются от базовых классов.

На загружаемые расширения должна быть ссылка из описательной части JAR-файла аплета или приложения (для этого служит заголовок `Class-Path`). Загрузчик классов, модифицированный для поддержки механизма расширений, сначала пытается отыскать запрашиваемый класс среди базовых, затем просматривает установленные расширения и только после этого обращается к загружаемым. Последние доступны только в запросившем их приложении (или аплете).

Компания Sun Microsystems предполагает оформить в виде расширений целый ряд программных пакетов, в том числе:

- Java Naming and Directory Interface, JNDI;
- JavaMail;
- Java Commerce;
- Enterprise JavaBeans;
- Java Management;
- Java3D;
- Java Transaction Service;
- Java Media Framework

Механизм расширений, важный и полезный с концептуальной точки зрения, усложняет проблему мобильности программ. Понятие Java-платформы размывается. Очевидно, что со временем расширения станут многоуровневыми, так что «в нагрузку» к небольшому на первый взгляд приложению придется брать или покупать целый лес JAR-файлов. Но очевидно и то, что эта проблема существует для всех платформ и решить ее можно только при радикальном изменении технологии функционирования программных систем, в первую очередь распределенных.

С тонкостями механизма расширений и, в частности, с изменениями в программном интерфейсе загрузчика классов можно ознакомиться по спецификации [18].

4. Механизмы безопасности

Вопросы информационной безопасности Java-приложений и аплетов обсуждаются давно и весьма интенсивно. Рассматривались они и в Jet Info (см. [19], раздел «Java — безопасная программная среда для создания распределенных приложений»). Тем не менее, тему никак нельзя считать исчерпанной хотя бы потому, что разработчики Java-технологии продолжают вносить принципиальные изменения, не просто затрагивающие, но изменяющие основы, такие как модель безопасности. Попытаемся проанализировать ситуацию, складывающуюся в процессе завершения работы над JDK версии 1.2.

4.1. Защитные рубежи

Известный принцип гласит: «Надежная оборона должна быть эшелонированной». В согласии с этим принципом в Java-технологии предусмотрен целый ряд защитных рубежей, которые можно разделить на три группы:

- надежность языка;
- контроль при получении и загрузке программ;
- контроль при выполнении программ.

Java — простой язык, что позволяет надеяться на уменьшение числа «ошибок непонимания» по сравнению, например, с программами на C++. Далее, Java обеспечивает типовую (ударение на первом слоге) безопасность за счет средств статического и динамического контроля. Еще одно традиционно отмечаемое достоинство Java — автоматическое управление памятью, исключающее появление «висячих» указателей. Наконец, необходимо упомянуть о средствах статического разграничения доступа к программным компонентам, то есть о механизме пакетов и спецификаторах `protected`, `private`, `final`.

Контроль при получении и загрузке программ носит в Java многоступенчатый характер. Во-первых, программные компоненты могут снабжаться электронной подписью, что позволяет контролировать их целостность и аутентичность. Во-вторых, согласно [19], верификатор байткодов, кроме общей проверки формата поступившей информации, пытается убедиться в отсутствии следующих некорректных действий:

- подделка указателей (например, получение указателя как результат выполнения арифметической операции);

- нарушение прав доступа к компонентам классов (например, присваивание переменной, снабженной описателем final);
- использование объектов в каком-либо другом качестве (например, применение к объекту метода другого класса);
- вызов методов объектов с недопустимым набором параметров;
- недопустимое преобразование типов;
- вызов операции Java-машины с недопустимым набором параметров;
- некорректная операция с регистрами Java-машины (например, запись регистра с неопределенным содержимым);
- переполнение или исчерпание стека.

Выполняются также некоторые другие, более тонкие проверки, связанные, в частности, с корректностью обработки исключительных ситуаций.

Еще одной «ступенькой» контроля при получении и загрузке программ является загрузчик классов. Он поддерживает разделение пространств имен программ, поступивших из разных источников. В частности, пространство имен локальных программ отделено от пространства имен программ, полученных по сети. Порядок просмотра этих пространств при разрешении внешних ссылок гарантирует, что загружаемая программа не сможет «заслонить» библиотечные или локальные объекты.

Контроль при выполнении программ можно разделить на два уровня. На нижнем уровне виртуальная Java-машина не допускает выходов за границы массивов и аналогичных некорректных действий. На верхнем уровне менеджер безопасности выполняет содержательные проверки правомерности доступа, реализуя тем самым выбранную политику безопасности.

Несомненно, никакая другая технология программирования на сегодняшний день не обеспечивает аналогичной степени информационной безопасности. Правда, Java-безопасность достигается ценой существенного снижения эффективности.

Если продолжить деление на группы и уровни, то полезно выделить следующие два аспекта Java-безопасности:

- защита Java-окружения времени выполнения и ресурсов аппаратно-программной платформы от вредоносного программного обеспечения;
- разграничение доступа субъектов к ресурсам прикладного уровня.

Практически все перечисленные выше защитные рубежи относятся к первому аспекту. Лишь менеджер безопасности может быть использован для пользовательского разграничения доступа, а надежность языка способствует повышению устойчивости Java-программ. Конечно, собственная безопасность — вещь важная, но она интересна в основном разработчикам ограниченного числа программных средств, таких, например, как Web-навигаторы; для массы пользователей она является чем-то само собой разумеющимся.

Происхождение отмеченного «перекоса» понятно. Он коренится во взглядах на Java как технологию исключительно для клиентских систем, где пользователь один, и именно его (точнее, принадлежащие ему ресурсы) требуется обезопасить. С появлением Java на серверной стороне ситуация меняется, что, однако, пока не нашло должного отражения ни в модели безопасности Java, ни в наборе предлагаемых прикладных программных интерфейсов.

Конечно, на уровне Java-платформы невозможно решить все проблемы приложений и, в частности, обеспечить надежное разграничение доступа к прикладным ресурсам. Однако теоретически в Java-технологии могли бы специфицироваться (например, в виде стандартных расширений) программные интерфейсы для следующих защитных механизмов (см., например, [20], раздел «Основные программно-технические меры»):

- идентификация/аутентификация;
- управление доступом;
- протоколирование/аудит.

Пока в каком-то виде (точнее, в виде списков управления доступом, перспективы которых, впрочем, не ясны) специфицирован только второй из перечисленных механизмов (механизм менеджеров безопасности носит слишком общий характер, его трудно рассматривать как готовое или почти готовое решение). В принципе аутентификация может быть обеспечена криптографическими средствами, которые будут рассмотрены ниже, но желательна более детальная проработка этого вопроса. Протоколирование/аудит — пока, к сожалению, белое пятно в Java-технологии.

Разумеется, разработчики Java осознают стоящие перед ними проблемы (см., например, [21], раздел «Discussion and Future Directions»). Можно надеяться, что со временем адекватные решения будут найдены.

4.2. Эволюция модели безопасности

Как уже указывалось, основная цель мер безопасности в Java — обеспечить защиту Java-окружения от вредоносных программ. Для достижения этой цели в JDK 1.0 была предложена концепция «песочницы» (`sandbox`) — замкнутой среды, в которой выполняются потенциально ненадежные программы. Таковыми считались аплеты, поступившие по сети. Весь «родной» код (то есть программы, располагающиеся на локальном компьютере) считался абсолютно надежным и ему было доступно все, что доступно виртуальной Java-машине (рис. 11).

В число ограничений, налагаемых «песочницей», входит запрет на доступ к локальной файловой системе, на сетевое взаимодействие со всеми хостами, кроме источника аплета (хост, с которого аплет был получен) и т.д. При таких ограничениях безопасность в общем и целом обеспечивается, но возможности для работы у аплетов почти не остается.

Чтобы как-то справиться с этой проблемой, в JDK 1.1 ввели понятие электронной подписи, которую ставит распространитель аплета. Java-машина в соответствии со своей политикой безопасности делит распространителей и, соответственно, их аплеты на две категории — надежные и ненадежные (неподписанный аплет, естественно, считается ненадежным). Надежные аплеты были приравнены в правах к «родному» коду, в результате чего модель безопасности эволюционировала к виду, приведенному на рис. 12.

Таким образом, в JDK 1.1 произошло скачкообразное расширение прав аплетов. Это решило проблемы тех, кому прав не хватало (достаточно было попасть в число надежных распространителей), однако весьма грубое деление прав доступа — все или (почти) ничего — делало оборону неэшелонированной и, следовательно, уязвимой. Любая ошибка при определении «свой/чужой» становилась фатальной.

В JDK 1.2 по существу произошел отказ от модели «песочницы», хотя разработчики утверждают, что обобщили ее. Оформились три основных понятия:

- источник программы;
- право и множество прав;
- политика безопасности.

Источник программы определяется парой (универсальный локатор ресурсов — URL, распространители программы — те, кто подписал ее).

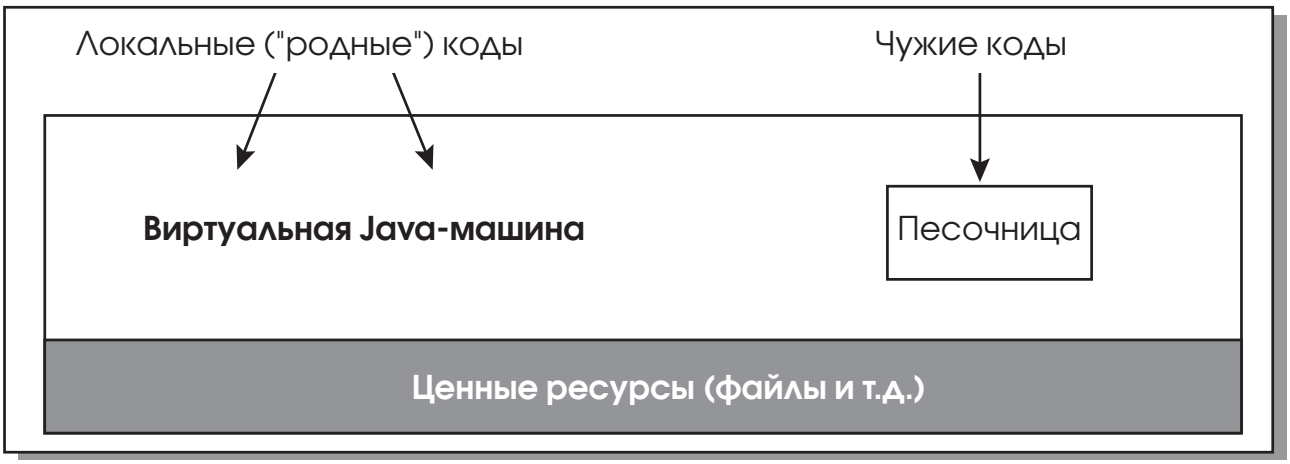

Рис. 11. Модель безопасности в JDK 1.0.

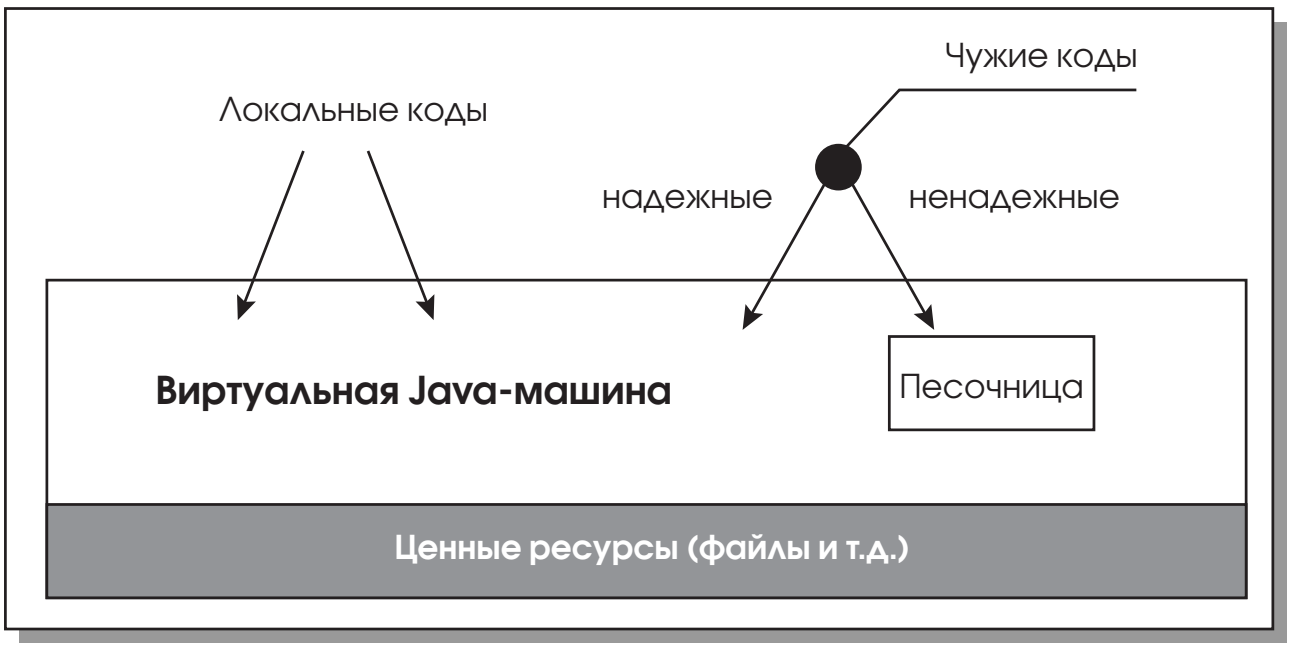

Рис. 12. Модель безопасности в JDK 1.1.

URL может указывать на файл в локальной файловой системе или же на ресурс удаленной системы.

Право — это абстрактное понятие, за которым, как обычно в Java, стоят классы и объекты. В большинстве случаев право определяется двумя цепочками символов — именем ресурса и действием. Например, в качестве ресурса может выступать файл, а в качестве действия — чтение. Важнейшим методом «правовых» объектов является `implies()`. Он проверяет, что одно право (запрашиваемое) следует из другого (имеющегося).

Политика безопасности задает соответствие между источником и правами поступивших из него программ. Вообще говоря, в JDK 1.2 «родные» программы не имеют каких-либо привилегий в плане безопасности и политика по отношению к ним может быть любой.

Формально можно считать (если хочется чувствовать идейное родство с прежними версиями JDK), что каждому источнику соответствует своя «песочница». Видимо, разработчики механизмов безопасности в JDK 1.2 такого родства желали, поэтому они ввели (точнее, использовали) понятие домена защиты, понимая под ним совокупность источника программ и ассоциированных прав доступа. Но содержательных операций над доменами не определено, так что с нашей точки зрения данное понятие в JDK 1.2 является избыточным, а проведение аналогий между доменами и «песочницей» — неправомерным. По сути мы имеем традиционный для современных операционных систем и систем управления базами данных механизм прав доступа со следующими особенностями:

- Субъектом доступа является не пользователь, а источник программы. Впрочем, формально можно считать, что во время исполнения программы ее источник становится пользователем (и это весьма глубокая и прогрессивная трактовка, если ее развить в правильном направлении);
- Нет понятия владельца ресурса, который (владелец) мог бы менять права; последние задаются исключительно политикой безопасности. Впрочем, формально можно считать, что владельцем всего является тот, кто формирует политику;
- Механизмы безопасности снабжены объектной оберткой.

Еще одним важнейшим понятием в модели безопасности JDK 1.2 является контекст выполнения. Когда виртуальная Java-машина проверяет права доступа объекта к системному ресурсу, она рассматривает не только этот (текущий) объект, но и предыдущие элементы стека вызовов. Доступ предоставляется только тогда, когда нужным правом обладают все объекты в стеке (возможно, принадлежащие разным доменам защиты). Разработчики Java называют это реализацией принципа минимизации привилегий.

На первый взгляд учет контекста представляется логичным. Нельзя допускать, чтобы вызов какого-либо метода расширял права доступа хотя бы по той причине, что доступ к системным ресурсам осуществляется не напрямую, а с помощью системных объектов, имеющих все права (см. рис. 13). Наличие нескольких уровней объектов-посредников — норма, которую нужно принимать во внимание.

Увы, эти доводы противоречат одному из основных принципов объектного подхода — принципу инкапсуляции. Если объект **A** обращается к объекту **B**, он не может и не должен знать, как реализован **B** и какими ресурсами он пользуется для своих целей. Возможно, **B** работает с временными файлами или обращается к удаленной базе данных, но **A** это не касается. Если **A** имеет право вызывать какой-либо метод **B** с некоторыми значениями аргументов, **B** обязан обслужить вызов. В противном случае при формировании политики безопасности придется учитывать возможный граф вызовов объектов, что, конечно же, нереально.

Разумеется, разработчики Java осознавали эту проблему. Чтобы справиться с ней, они ввели понятие привилегированного интервала программы. При выполнении такого интервала контекст (нижняя часть стека) игнорируется. Привилегированная программа отвечает за себя, не интересуясь предысторией. Аналогом привилегированных программ являются файлы с битами переустановки идентификатора пользователя/группы в ОС Unix, что лишний раз подтверждает традиционность подхода, реализованного в JDK 1.2. Известны угрозы безопасности, которые привносят подобные файлы. Теперь это не лучшее средство ОС Unix перекочевало в Java. Такова оборотная сторона попыток минимизировать привилегии. Если на проходной слишком дотошный вахтер, в заборе появляются дырки.

Еще одна проблема, присущая выбранному подходу, — тяжеловесность реализации. В частности, при порождении нового потока (`thread`) с ним приходится ассоциировать зафиксированный «родительский» контекст и, соответственно, проверять последний в процессе контроля прав доступа.

Но, возможно, хуже всего то, что этот подход не обобщается на распределенный случай, хотя бы потому, что контекст имеет лишь локальный смысл (как и политика безопасности). Конечно, информационная безопасность распределенных систем — очень сложная проблема, но решать ее надо. В JDK 1.2 такой попытки не сделано.

В общем и целом состояние механизмов безопасности в JDK 1.2 можно оценить как промежуточное. Во-первых, для защиты от вредоносного программного обеспечения использованы традиционные механизмы безопасности, которые обычно применяют для разграничения поль-

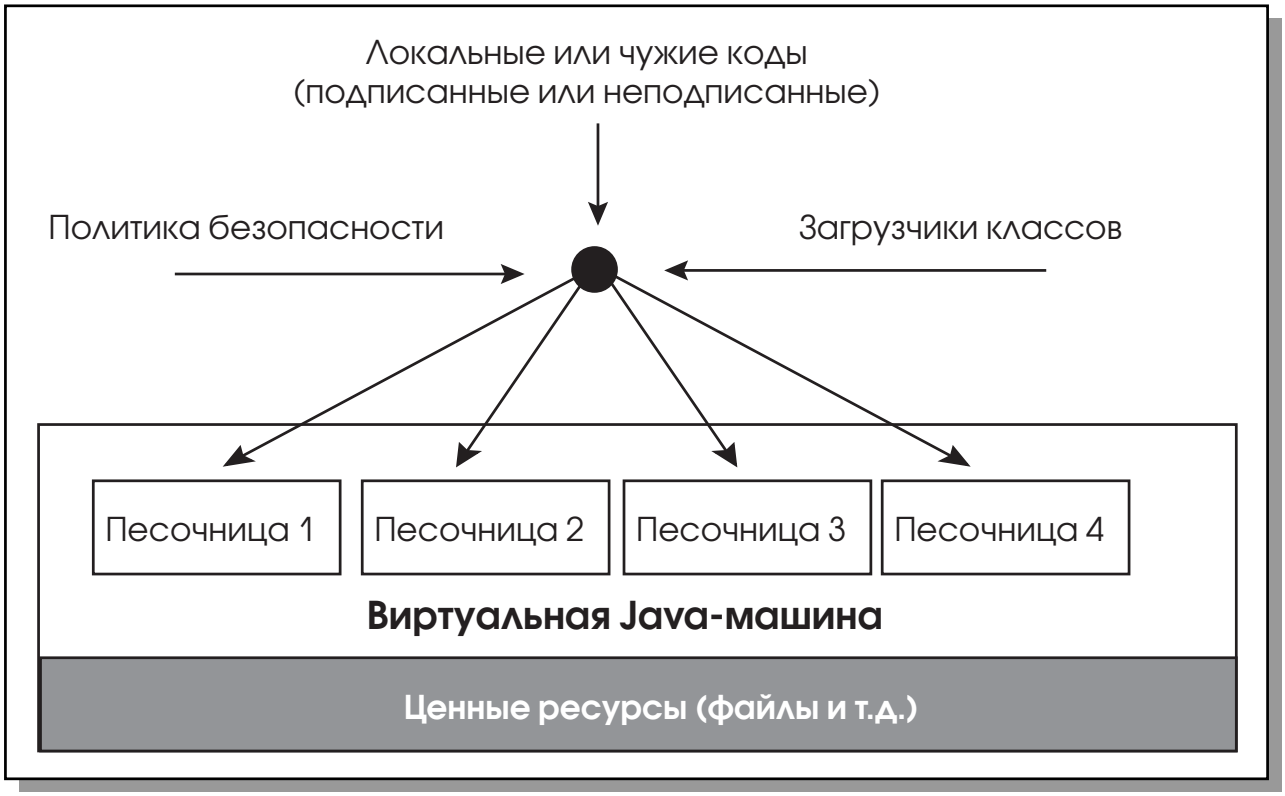

Рис. 13. Модель безопасности в JDK 1.2.

зовательского доступа. Пройдено как бы полвитка спирали. Гранулированность доступа стала сколь угодно тонкой, но те, для кого такая гранулированность действительно нужна (пользователи), пока не поддерживаются. Во-вторых, половинчатой является и объектная ориентированность механизмов безопасности. Реализация, разумеется, оформлена в виде интерфейсов и классов, однако доступ по-прежнему разграничивается к необъектным сущностям — ресурсам в традиционном понимании. Наконец, не ясно, как применять предлагаемые средства в распределенных приложениях.

4.3. Криптографическая архитектура Java

Криптография — неотъемлемая часть современных средств информационной безопасности. Без криптографии невозможны надежная аутентификация, контроль целостности и, конечно, сохранение конфиденциальности.

Криптографическая архитектура Java (Java Cryptography Architecture, JCA) разработана для предоставления следующих сервисов:

- постановка/проверка электронной подписи;
- вычисление хэш-функции;
- генерация пар ключей открытый/секретный;
- создание сертификатов, подтверждающих аутентичность открытых ключей;
- хранение ключей, а также сертификатов надежных партнеров;
- преобразование ключей из представления со скрытой структурой в общепонятное представление и наоборот;
- управление параметрами криптографических алгоритмов;
- генерация параметров криптографических алгоритмов;
- генерация (псевдо)случайных чисел;
- симметричное шифрование;
- выработка общего ключевого материала.

Кроме того, криптографическая архитектура должна удовлетворять следующим технологическим требованиям:

- обеспечивать независимость от алгоритмов и их реализаций;
- обеспечивать взаимную совместимость реализаций;
- обеспечивать расширяемость набора алгоритмов и их реализаций.

Чтобы лучше понять соотношение между криптографическими сервисами, алгоритмами и реализациями, обратимся к рис. 15. Вообще говоря, каждый сервис может обеспечиваться несколькими алгоритмами, каждый из которых, в свою очередь, может иметь несколько реализаций. Например, для вычисления хэш-функции предназначены алгоритмы MD5/SHA-1 (равно как и российский ГОСТ «Функция хэширова-

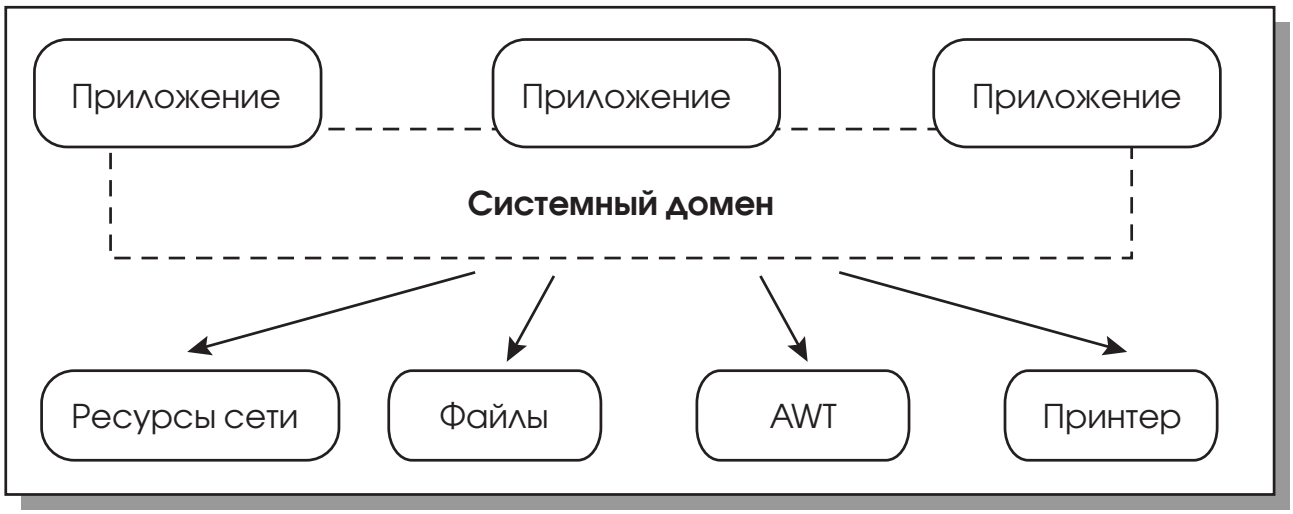

Рис. 14. Взаимодействие прикладных и системных объектов.

ния»), для выработки и проверки электронной подписи — алгоритмы RSA/DSA, российский ГОСТ «Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма» и т.д.

Программный интерфейс сервисов построен так, чтобы отразить их функциональность в алгоритмически-независимой форме. Для обозначения реализаций используется понятие поставщика криптографических услуг — пакета или группы пакетов, содержащих реализацию. В JDK 1.2 имеется встроенный поставщик — «SUN», поддерживающий весь спектр сервисов. Приложение имеет возможность выбирать алгоритмы и поставщиков услуг из числа доступных.

Одним из средств обеспечения взаимной совместимости реализаций (то есть различных поставщиков криптографических услуг) являются функции преобразования ключей из представления со скрытой структурой в общепонятное представление (и наоборот), позволяющие использовать общий ключевой материал.

Традиционный для криптографии нюанс состоит в том, что два последних элемента в списке сервисов (симметричное шифрование и выработка общего ключевого материала) подвержены экспортным ограничениям США, поэтому они, в отличие от остальных перечисленных сервисов, оформлены как расширение (Java Cryptography Extension, JCE), являющееся отдельным продуктом.

Криптографическая архитектура Java и в плане предоставляемых сервисов, и в технологическом плане носит достаточно традиционный характер. Полезно сопоставить JCA и архитектуру IPsec (см. [13], раздел «Архитектура средств безопасности»). Нам же остается лишь в очередной раз пожалеть по поводу строгости российского законодательства в области криптографии.

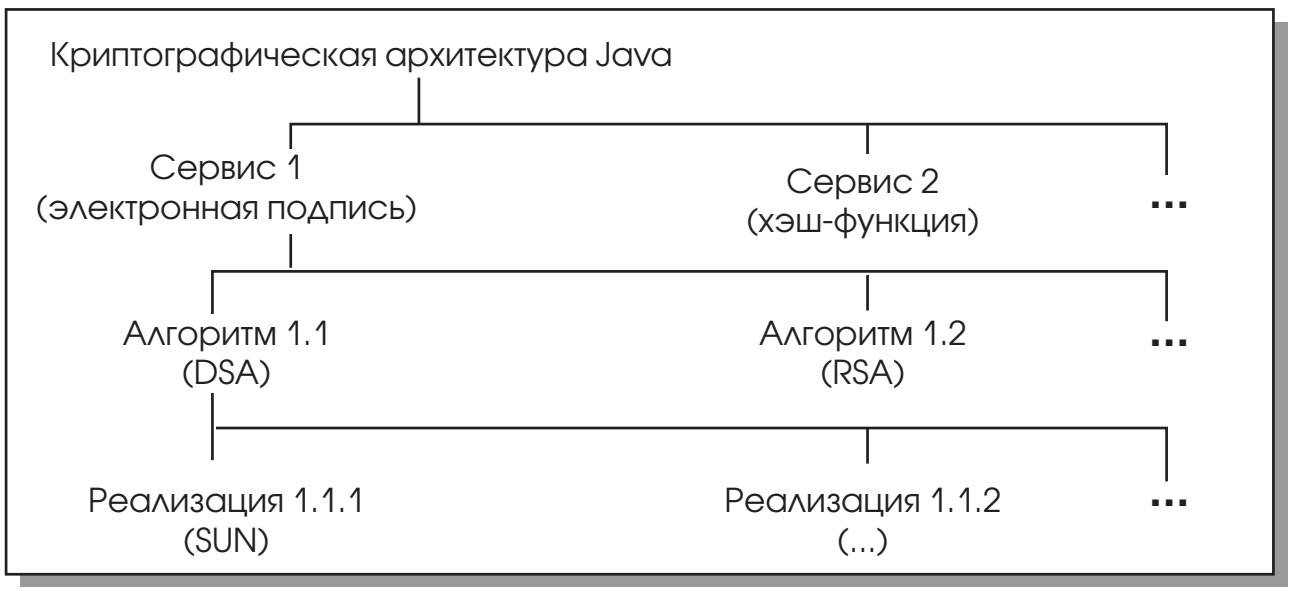

Рис. 15. Иерархия криптографических сервисов, алгоритмов и реализаций.

Более подробную информацию о криптографической архитектуре Java и ее реализации можно найти в статье [22].

4.4. Объектная организация механизмов безопасности в JDK 1.2

Механизмы безопасности в JDK 1.2 оформлены в виде четырех основных пакетов и трех пакетов расширения:

- `java.security` — содержит интерфейсы и классы, составляющие каркас механизмов безопасности. Сюда входят рассмотренные выше средства разграничения доступа, а также криптографические средства, свободные от экспортного контроля США (выработка электронной подписи, вычисление хэш-функции и т.п.).
- `java.security.cert` — средства поддержки сертификатов X.509 версии 3.
- `java.security.interfaces` — средства генерации RSA- и DSA-ключей.
- `java.security.spec` — средства спецификации ключевого материала и параметров криптографических алгоритмов.
- `javax.crypto` — интерфейс и классы для симметричного шифрования (пакет расширения JCE 1.2).
- `javax.crypto.interfaces` — интерфейсы средств выработки ключей для алгоритма Диффи-Хелмана (пакет расширения JCE 1.2).
- `javax.crypto.spec` — классы для управления ключами и параметрами криптографических алгоритмов (пакет расширения JCE 1.2).

Наиболее важные с концептуальной точки зрения интерфейсы и классы сосредоточены в пакете `java.security`. Их мы и рассмотрим.

4.4.1. Источник программы

Класс `CodeSource` описывает источники программ. Источник характеризуют URL и набор сертификатов. Предикат `implies()` устанавливает отношение частичного порядка между источниками. Источник **B** может пользоваться правами **A** (**B** не слабее **A**), если URL(**B**) соответствует элементу в дереве с корнем URL(**A**), а все сертификаты **A** присутствуют и в **B**.

4.4.2. Право и множество прав

Права доступа обслуживаются несколькими классами. Класс `Permission` описывает одно право, класс `PermissionCollection` — множество однородных прав, класс `Permissions` — множество множеств (точнее, коллекция коллекций) прав.

Абстрактный класс `Permission` описывает абстрактное право. Конкретным правам, рассчитанным на определенные типы ресурсов, соответствуют классы — преемники `Permission`, такие как `java.io.FilePermission` (права доступа к файлам), `java.net.SocketPermission` (права на взаимодействие в удаленными системами) и т.п. Каждое право имеет имя, обозначающее контролируемый ресурс или группу ресурсов (например, все файлы заданного каталога). Кроме того, право может характеризовать допустимые действия с ресурсом (чтение файла, прием сетевых соединений от удаленной системы и т.д.).

Обратим особое внимание на три метода класса `Permission` - `newPermissionCollection()`, `implies(Permission)` и `checkGuard(Object)`.

Метод `newPermissionCollection()` создает пустую коллекцию «правовых» объектов данного класса. Он (метод) необходим для добавления нового элемента к совокупности `Permissions`, если коллекция нужного типа отсутствует.

Метод `implies(Permission)` — идейная основа механизма прав доступа в JDK 1.2. Он определяет, является ли право-аргумент следствием права, задаваемого текущим объектом. Вообще говоря, наличие этого метода может навести на мысль, что при проверке прав доступа используется аппарат логического вывода, во всей его мощи и тяжеловесности. На самом деле это не так, поскольку право задается не предикатами, а двумя цепочками символов, которые характеризуют два множества — ресурсов и действий. Право **B** следует из права **A**, если ресурсы(**B**) входят в ресурсы(**A**), а действия(**B**) входят в действия (**A**).

С точки зрения эффективности проверки следования прав решающее значение имеет дисциплина именования ресурсов. Имена должны отражать естественную иерархию ресурсов. Например, для прав `FilePermission` допускаются имена следующего вида:

- имя_файла (обозначает один файл);
- имя_каталога (обозначает один каталог);
- имя_каталога/имя_файла (заданный файл в заданном каталоге);
- имя_каталога/* (обозначает все файлы в заданном каталоге);
- * (все файлы в текущем каталоге);
- имя_каталога/ — (все файлы в заданном каталоге и его подкаталогах);
- (все файлы в текущем каталоге и его подкаталогах);
- <<ALL FILES>> (все файлы в файловой системе).

Подобная дисциплина именования соответствует организации файловых систем. Более того, за счет снижения выразительной силы имен по сравнению, скажем, с шаблонами, используемыми в ОС Unix (такими, хотя бы, как

`*.txt`), удается избежать сравнительно сложных процедур сопоставления цепочек символов. В принципе, от упрощения имен страдает гранулированность доступа (нет возможности разрешить аплету «текстовый редактор» чтение/запись только текстовых файлов), но в ситуации, когда субъектами доступа являются источники программ, данное обстоятельство не имеет особого значения.

Отметим, что метод `implies(Permission)` (с очевидной модификацией семантики) присутствует и в классе `PermissionCollection`. Здесь эффективность его реализации еще важнее, поскольку речь идет о количестве перебираемых элементов. Если устранять избыточные элементы при выполнении метода `add(Permission)`, можно поддерживать коллекцию в «нормальной форме», минимизируя тем самым перебор.

Возвращаясь к классу `Permission`, рассмотрим метод `checkGuard(Object)`. Назначение этого метода и включающих его механизмов состоит в том, чтобы осуществить проверку прав доступа силами и в контексте потребителя (а не поставщика) ресурса. Это может быть полезно, когда у поставщика ресурса не хватает информации для проверки (например, потому, что передавать контекст потребителя слишком дорого).

Последовательность действий, выполняемых потребителем и поставщиком ресурса приведена на рис. 16. Поставщик создает и возвращает потребителю защищающий объект, содержащий запрашиваемый ресурс и требуемые права доступа. Ресурс извлекается потребителем путем применения метода `getObject`, который, в свою очередь, вызывает `checkGuard()`. Стандартная реализация этого метода в классах — преемниках `Permission` сводится к вызову `SecurityManager.CheckPermission(this)`.

Таким образом, потребитель ресурса, «дернув за веревочку» `getObject()`, пускает в действие механизм контроля прав доступа в своем (потребителя) контексте.

Механизм защищающих объектов носит весьма общий характер. В принципе он позволяет ассоциировать с актом доступа произвольные действия. В качестве аналогии можно привести триггеры и правила в системах управления базами данных, а также разного рода обработчики (*hook*) в операционных системах. В то же время, в Java этот механизм (в предложенном виде) выглядит чужеродной заплатой, призванной как-то скрыть проблемы, возникающие при передаче контекстов. Во-первых, не используются преимущества, которые дает объединение в Java языковой и операционной среды. Доступ оказывается непрозрачным, с потерей информации (обратите внимание на преобразование типа после вызова `getObject()`). Во-вторых, не очевидно, что защищающие объекты решают те проблемы, ради которых они были задуманы. Если тяжело передавать контекст от потребителя ресурса к поставщику, то, скорее всего, осуществлять доступ к ресурсу на стороне потребителя тоже будет накладно. Значит, на деле `getObject()` будет служить только для проверки прав, что увеличивает непрозрачность.

Вернемся, однако, к более удачным (и более важным с практической точки зрения) аспектам объектной организации механизма прав доступа.

Чтобы оценить детальность проработки и полноту охвата проблемы прав доступа в JDK 1.2, полезно рассмотреть иерархию «правовых» классов, представленную на рис. 17. Можно видеть, что разграничению доступа подвергаются все виды системных ресурсов.

4.4.3. Политика безопасности

Политика безопасности в JDK 1.2 устанавливает соответствие между источниками программ и их правами доступа. В Java-машине в каждый момент времени она представлена одним объектом класса, являющегося преемником абстрактного класса `Policy`.

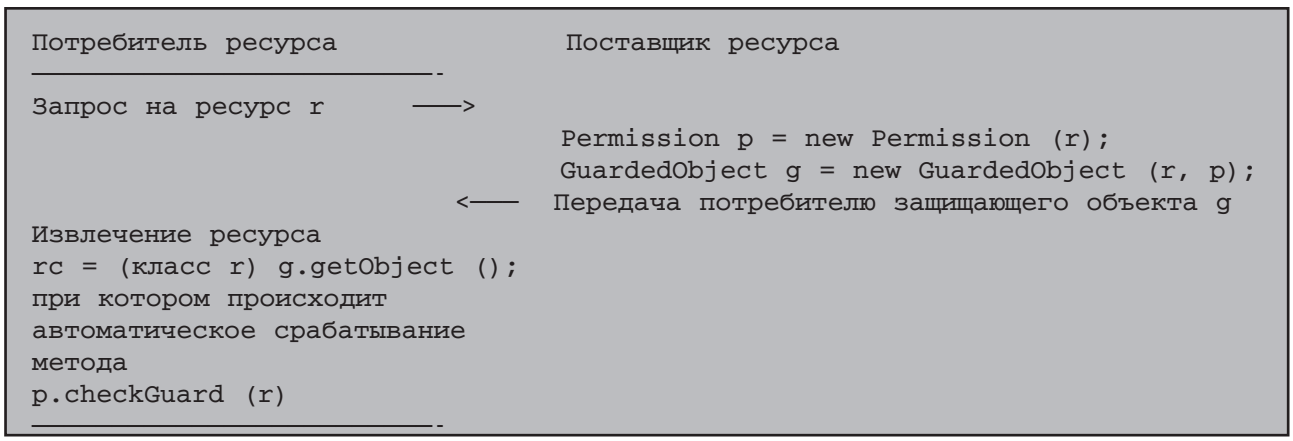

Рис. 16. Последовательность действий при использовании защищающего объекта.

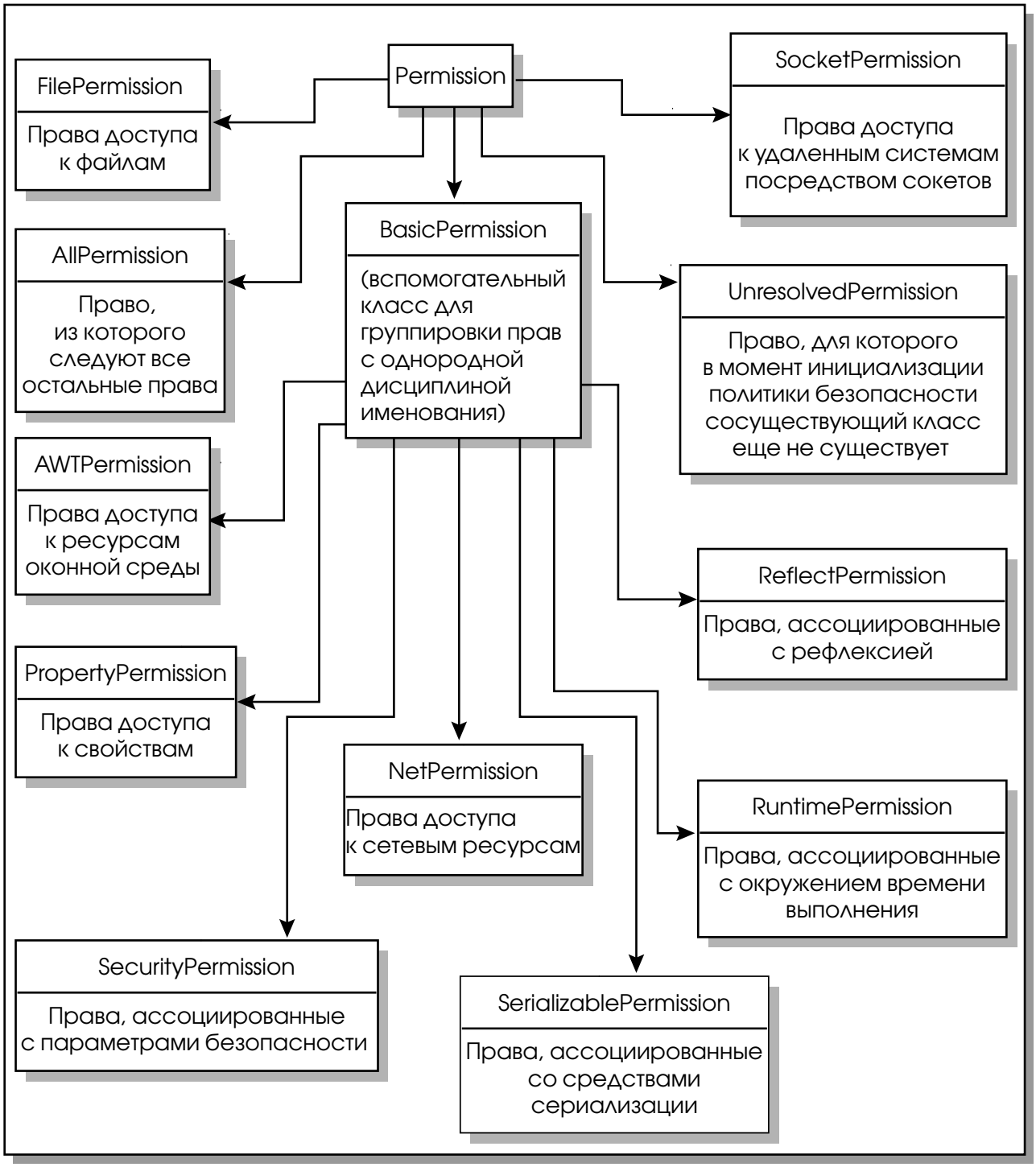

Рис 17. Иерархия классов, описывающих права доступа.

Методы класса `Policy` позволяют получить/установить текущую политику (обратим внимание на то, что эти методы являются статическими, а для их выполнения необходимо обладать соответствующими правами класса `SecurityPermission`), «освежить» текущую политику (например, заново прочитать конфигурационную информацию), а также выяснить права доступа, ассоциированные с заданным источником программ.

С практической точки зрения наибольший интерес представляет стандартная реализация политики безопасности. Она основана на конфигурационных файлах, имеющих текстовое представление. Права задаются строками вида

```
grant источник_программ { права_доступа };
```

Кроме того, в строке с ключевым словом `keystore` задается место хранения криптографи-

ческих ключей и сертификатов (необходимых для проверки аутентичности электронных подписей).

4.4.4. Проверка прав доступа

Вообще говоря, для контроля прав доступа в JDK 1.2 можно пользоваться двумя средствами:

- встроенным менеджером безопасности, получившим название `AccessController`;
- динамически изменяемым менеджером безопасности, присутствовавшим и в более ранних версиях JDK — `SecurityManager`.

Предпочтительным является первый вариант. Его мы и рассмотрим.

Класс `AccessController` предоставляет единый метод для проверки заданного права в текущем контексте — `checkPermission (Permission)`. Это, конечно, лучше (по причине параметризуемости), чем множество методов вида `checkXXX`, присутствующих в `SecurityManager`.

Способ, которым `AccessController` осуществляет проверку прав доступа, был кратко описан выше, в разделе 4.2. Здесь мы изложим его более формально, попутно поясняя новые понятия.

Пусть текущий контекст выполнения состоит из N стековых фреймов (верхний соответствует методу, вызвавшему `checkPermission(p)`). Проверка производится по следующему алгоритму (см. листинг 6).

Сначала в стеке ищется фрейм, не обладающий проверяемым правом. Проверка производится до тех пор, пока либо не будет исчерпан стек, либо не встретится «привилегированный» фрейм, порожденный в результате обращения к методу `doPrivileged(PrivilegedAction)` класса `AccessController`. Если при порождении текущего потока выполнения был сохранен контекст `inheritedContext`, проверяется и он. При положительном результате проверки метод `checkPermission(p)` «молча» возвращает управление, при отрицательном возбуждается исключительная ситуация `AccessControlException`.

Класс `AccessController`, помимо своей основной функции — проверки прав доступа, «по совместительству» выполняет еще два вида действий:

- оформление привилегированных интервалов программ;
- сохранение текущего контекста.

Оформление привилегированных участков программ выполняется с помощью метода `doPrivileged(PrivilegedAction)` (см. листинг 7). В контексте данной статьи приведенный на листинге 6 фрагмент примечателен тем, что в нем использован анонимный класс, реализующий интерфейс `PrivilegedAction`.

Метод `doPrivileged()` вызывает метод `run()` своего объекта-параметра, помечая соответствующий стековый фрейм как привилегированный.

За сохранение текущего контекста отвечает метод `getContext()`. Он возвращает результат класса `AccessControlContext`, для которого также, как и для `AccessController`, определен метод `checkPermission(Permission)`, использованный на листинге 6.

4.4.5. Криптографические интерфейсы и классы

Объектная организация криптографической подсистемы Java естественным образом отражает описанную выше криптографическую архитектуру (см. раздел «Криптографическая архитектура Java»). Каждому сервису соответствует абстрактный класс, описывающий его (сервиса) программный интерфейс, а также класс, описывающий программный интерфейс поставщика сервиса. Примеры таких пар — `MessageDigest` и `MessageDigestSpi`, `Signature` и `SignatureSpi`. Правда, в силу исторических причин имеет место странный порядок наследования — не поставщик наследует сервис, а наоборот, но это, конечно, не влияет на криптостойкость реализаций.

Важными с концептуальной точки зрения являются классы `Provider` и `Security`. В первом собраны общие методы поставщиков криптогра-

```
i = N;
while (i > 0) {
  if (метод, породивший i-й фрейм, не имеет проверяемого права) {
    throw AccessControlException
      } else if (i-й фрейм помечен как привилегированный) {
        return;
      }
  i = i - 1;
};
// Проверим, есть ли проверяемое право у унаследованного контекста
inheritedContext.checkPermission (p);
```

Листинг 6. Алгоритм работы метода checkPermission класса AccessController.

```
обычный программный код
. . .

AccessController.doPrivileged (new PrivilegedAction () {
  public Object run () {
    // привилегированный интервал,
    // то есть программный код,
    // выполняющийся без учета прав доступа
    // вызывающих объектов
    . . .
  }
}

. . .
обычный программный код
. . .
```

Листинг 7. Пример оформления привилегированного интервала.

фических услуг, во втором — методы управления поставщиками и параметрами алгоритмов.

Подробное описание криптографических интерфейсов и классов Java можно найти в статье [22].

5. JavaOS

JavaOS — это семейство небольших по размеру, эффективных операционных систем, оптимизированных для поддержки Java-среды. По сравнению с реализацией Java над универсальными ОС, JavaOS обеспечивает не только экономию ресурсов (что важно в первую очередь для встроенных систем), но и снижение затрат на администрирование (что сулит существенную экономию для корпоративных систем).

Семейство JavaOS включает два основных элемента:

JavaOS for Business. (развивается совместно с корпорацией IBM);

JavaOS for Consumers.

JavaOS for Business рассчитана на работу в корпоративной среде и включает как клиентские, так и серверные компоненты (см. рис. 18). Клиентская часть предназначается для поддержки выполнения Java-приложений и аплетов, серверная — для администрирования клиентских систем (точнее, информации о них, расположенной на серверной стороне).

Характерная черта JavaOS — стремление к максимальной платформной независимости. Такая независимость способствует мобильности самой JavaOS и построенных на ее основе программных систем, что очень важно в условиях большого разнообразия аппаратных модификаций и частой смены моделей. Для достижения платформной независимости внутри JavaOS выделены технологические интерфейсы — с платформой (JavaOS Platform Interface, JPI) и периферийными устройствами (JavaOS Device Interface, JDI). Все, что выше этих интерфейсов, может быть написано на Java и сделано мобильным.

Уменьшение объемов платформно-зависимых частей JavaOS упрощает администрирование корпоративных конфигураций, делая их более однородными. Это важное средство снижения общей стоимости владения клиентскими частями информационных систем.

JavaOS for Consumers предназначена для поддержки среды PersonalJava, то есть для работы на интеллектуальных устройствах с открытым сетевым подключением, но с ограниченными ресурсами. Логическая структура JavaOS for Consumers и ассоциированного окружения изображена на рис. 46. Здесь наряду с Java-средствами могут использоваться универсальные механизмы систем реального времени.

В основе всех вариантов JavaOS лежит операционная система ChorusOS с микроядерной архитектурой. Такой выбор представляется весьма удачным. ChorusOS — это операционная система реального времени, обладающая высокой модульностью. Она способна предоставить те и только те сервисы, которые необходимы в конкретном окружении. Тем самым аппаратные ресурсы используются оптимальным образом. С другой стороны, она следует спецификациям Real-time POSIX на программный интерфейс, что позволяет, помимо JavaOS, поддерживать другие программные системы реального времени с открытой архитектурой, что и продемонстрировано на рис. 19.

Для желающих детально познакомиться с JavaOS можно рекомендовать справочное руководство [23].

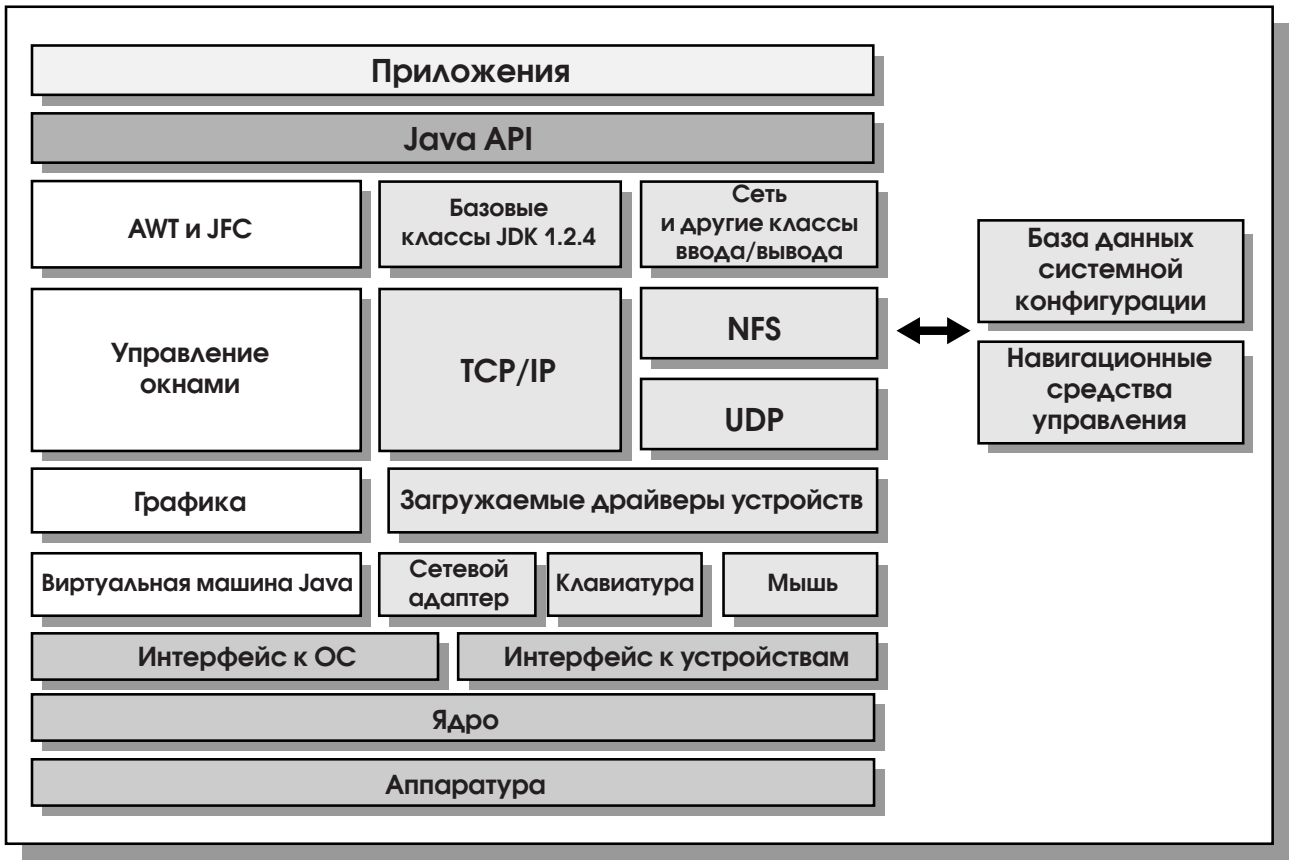

Рис. 18. Логическая структура JavaOS for Business.

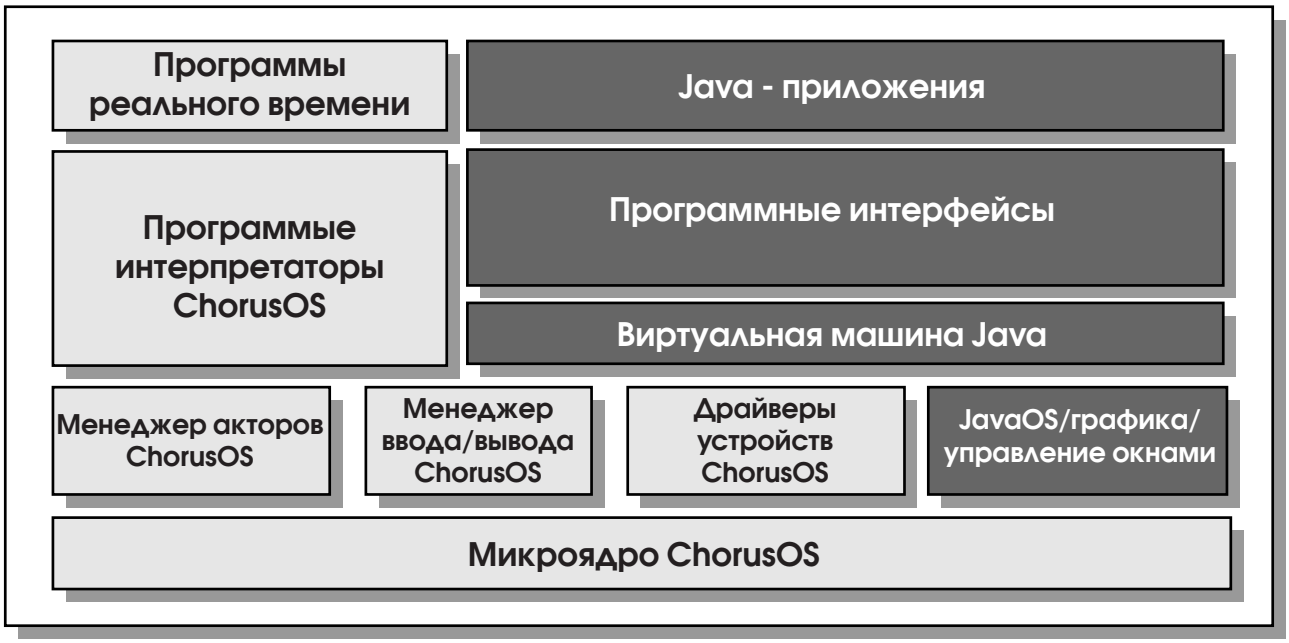

Рис. 19. Логическая структура JavaOS for Consumers.

6. Заключение

Java — одно из самых выдающихся явлений в информационных технологиях 1990-х. Java-технология оказала и оказывает огромное влияние на архитектуру современных программных систем, на стиль мышления программистов и компьютерных бизнесменов. Был задан новый темп работы, достигнут новый уровень открытости, мобильности, взаимной совместимости и информационной безопасности.

Более чем через три года с момента официального объявления Java-технология вступила в пору зрелости. Накоплен широкий спектр прило-

жений, создана достаточно полная аппаратно-программная «вертикаль»: JavaStation — JavaOS — программные интерфейсы — инфраструктурные компоненты — приложения. Java-технология на самом деле работает на всем спектре аппаратных платформ — от смарт-карты до суперкомпьютера.

Теперь на первый план выходят такие прозаические вещи, как стабильность спецификаций и качество реализации заявленных продуктов. От них в первую очередь зависит, в какой степени Java оправдает обещания одних и надежды других. Ближайшие год-два, на наш взгляд, станут в этом отношении решающими. Если «критическая масса качества» не будет накоплена, Java рискует стать одним из довольно многочисленных повзрослевших вундеркиндов, не сумевших до конца реализовать свой потенциал.

Области применения Java-технологии можно разделить на четыре категории:

- встроенные системы;
- клиентские системы;
- серверные компоненты;
- программное обеспечение промежуточного слоя.

На наш взгляд, особенно хороши перспективы Java в первой и последней категориях. Недостатки Java, такие как относительно низкая скорость выполнения программ, здесь не имеют принципиального значения, зато достоинства проявляются в полной мере. То, что компания Sun Microsystems приобрела компании Diba и Chorus, еще больше усиливает позиции Java на рынке встроенных систем.

На серверной стороне Java (с соответствующими расширениями), вероятно, станет одним из самым употребительных средств для написания хранимых процедур. Язык Java, конечно, может довольно широко использоваться при реализации серверов, для которых объем ввода/вывода существенно превосходит объем действий по обработке данных.

Разумеется, Java-технология была и будет активно использоваться на клиентской стороне. Вопрос заключается в размере той доли, которую составят Java-компоненты. На наш взгляд, пока, даже если принимать во внимание только технические аспекты, нет оснований говорить о том, что в скором времени Java будет доминировать в этой области.

Сейчас в моде слово «культовый» — культовый актер, культовый фильм и т.п. Java — это, несомненно, культовая технология. Ее успехи станут всеобщими успехами, ее неудачи болезненно отразятся на многих.

Будем надеяться, что «направление главного удара» будет выбрано правильно, и успехов окажется много больше, чем неудач.

Литература

1. Java как центр архипелага, Таранов А. и Цишевский В., Jet Info, 1996.
2. Компонентная объектная модель JavaBeans, Галатенко В. и Таранов А., Jet Info, 1997.
3. Inderstanding Java Card 2.0, Chen Z., Javaworld, Mar. 1998.
4. PersonalJava Technology White Paper, Sun Microsystems, 1998.
5. Объектные технологии построения распределенных информационных систем, Пуха Ю., Jet Info, 1997.
6. Опыт внедрения Java-технологии в компании Sun Microsystems, Peddada T., Littlepage J., и Ferris C., Jet Info, 1997.
7. Implementing a Multitier, Services-Based Architecture on the Java Platform at Sun. A Case Study, Sun Microsystems, 1998.
8. М. Технология «клиент-сервер» и мониторы транзакций, Ладыженский Г., «Открытые системы», 1994.
9. Открытые сетевые решения 1990-х годов, Радучел У., Jet Info, 1996.
10. JavaSpaces Specification. Revision 1.0 Beta, Sun Microsystems, 1998.
11. Distributed Leasing Specification. Revision 1.0 Beta, Sun Microsystems, 1998.
12. Сервер аутентификации Kerberos, Вьюкова Н. и Галатенко В., Jet Info, 1996.
13. Сетевые протоколы нового поколения, Галатенко В., Макстенек М., и Трифаленков И., Jet Info, 1998.
14. Distributed Event Specification. Revision 1.0 Beta, Sun Microsystems, 1998.
15. Transaction Specification. Revision 1.0 Beta, Sun Microsystems, 1998.
16. Jini Architecture Overview. White paper, Waldo J., Sun Microsystems, 1998.
17. An Overview of Swing Architecture, Fowler A., Sun Microsystems, 1998.
18. Support for Extensions and Applications in the Version 1.2 of the Java Platform, Sun Microsystems, 1998.
19. Информационная безопасность в Интранет: концепции и решения, Галатенко В. и Трифаленков И., Jet Info, 1996.
20. Информационная безопасность — обзор основных положений, Галатенко В., Jet Info, 1998.
21. Java Security Architecture (JDK 1.2). Draft Document (Revision 0.9), Gong L., Sun Microsystems, 1998.
22. Java Cryptography Architecture. API Specification & Reference, Sun Microsystems, 1998.
23. JavaOS for Business Version 2.0. Reference Manual, Sun Microsystems, IBM Corporation.

НОВОСТИ ИНТЕРНЕТ

Java 2

Sun microsystems

8 декабря 1998 года представители компании Sun Microsystems сообщили о выпуске новой версии среды разработки Java, получившей название Java 2 (см. http://-www.sun.com/smi/Press/sunflash/-9812/sunflash.981208.9.html). Тем, кто постоянно следит за развитием Java-технологии, эта версия больше известна под рабочим именем JDK 1.2, использовавшемся в статье "Java в три года". При подготовке JDK 1.2 и в язык, и в окружение времени выполнения было внесено много нового (см. упомянутую статью), так что использование названия Java 2 представляется вполне оправданным.

В тестировании JDK 1.2 приняли участие десятки тысяч разработчиков, что говорит о заинтересованности компьютерного сообщества и о поддержке Java в корпоративной среде. Быстрому и эффективному тестированию, несомненно, способствовала и открытость, распространившаяся и на финальный продукт, который можно получить, обратившись по адресу http://java.sun.com/jdk/.

Среди новинок, реализованных в Java 2, представители компании Sun Microsystems на первое место ставят улучшенные средства информационной безопасности. Они подробно рассмотрены в разделе "Механизмы безопасности" материала номера. Важное значение имеют и меры, направленные на повышение производительности, такие как оптимизация динамического распределения памяти, улучшенная поддержка потоков и "родных" интерфейсов, а также новый JIT-компилятор.

С новыми интерфейсными решениями, вошедшими в проект Swing (см. раздел "Swing" материала номера), связаны надежды на улучшения качества и эффективности Java-интерфейса. Не менее важны и средства интернационализации/локализации, поддержка коллекций (см. раздел "Коллекции"), возможность описания удаленно вызываемых методов (Java IDL API).

Разумеется, платформа Java 2 свободна от сложностей, связанных с проблемой 2000 года.

День 8 декабря оказался чрезвычайно богат на объявления. Было сообщено о выпуске ряда стандартных расширений для платформы Java 2 (см. http://www.-sun.com/smi/Press/sunflash/9812/sunflash.981208.8.html, а также раздел "Механизм расширений" в материале номера). В число доступных расширений вошли:

- Java 3D, программный интерфейс трехмерной графики;
- Java Naming and Directory Interface (JNDI), интерфейс службы имен и каталогов;
- программный интерфейс к Java сервлетам;
- JavaMail, программный каркас для разработки платформно-независимых почтовых приложений;
- Java Media Framework (JMF), спецификация архитектуры обработки мультимедийных данных.

Выпуском Java 2 компания Sun Microsystems достойно продолжила энергичное, продуманное, открытое развитие Java-технологии.

Java в интеллектуальных сетях

Летом 1998 года компания Sun Microsystems выступила с инициативой разработки стандартизованного Java-каркаса для перспективных интеллектуальных сетей. Инициатива получила наименование J.A.I.N — Java in Advanced Intelligent Networks. Идея состояла в упрощении слияния голосовых и IP-сетей, в обеспечении мобильности сервисов, в сокращении стоимости и сроков разработки новых приложений. Основным архитектурным элементом J.A.I.N является компонентная объектная модель JavaBeans (см. Jet Info, 1997, 19).

12 января 1999 года было объявлено (см. http://www.sun.com/-smi/Press/sunflash/9901/sunflash.-990112.4.html) о присоединении к инициативе J.A.I.N таких компаний, как IBM, Bellcore и Trillium, которые помогут распространить J.A.I.N на новую область. Имеется в виду создание среды разработки прикладных сервисов, функционирующих в рамках интеллектуальных сетей, и, в частности, стандартизация прикладных программных интерфейсов, не зависящих от сетевой инфраструктуры.

Ожидается, что в ближайшее время к инициативе J.A.I.N примкнут и многие другие компании.

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

Главный редактор: Галатенко В.А. *(galat@jet.msk.su)*
Технический редактор: Антонов А.Н. *(silver@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

Jet infosystems