READERS' CHOICE AWARDS ANNOUNCED!

JAVA™ DEVELOPER'S JOURNAL

JavaDevelopersJournal.com

Volume: 3 Issue: 8, 1998

Real Network Possibilities... Sun's JINI

Hot New Emerging picoJava
by Harlan McGhan pg. 50

Cosmic Cup
Web Products
by Ajit Sagar pg. 46

The Grind
Results of Impedence Mismatch
by Java George pg. 66

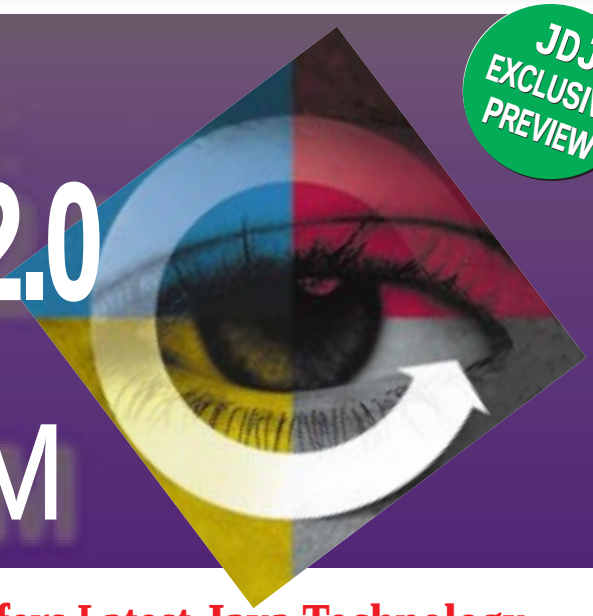
Product Reviews
SourceGuard
by David Reilly pg. 43
...
ProtoSpeed
by Jim Mathis pg. 52

RETAILERS PLEASE DISPLAY UNTIL OCTOBER 31, 1998

SYS-CON PUBLICATIONS

Sneak Peak: VisualAge 2.0 for Java from IBM

JDJ EXCLUSIVE PREVIEW



JAVA DEVELOPER'S JOURNAL

EDITORIAL ADVISORY BOARD

Ted Coombs, Bill Dunlap, David Gee, Arthur van Hoff, Brian Maso, Sean Rhody, Kim Polese, Rick Ross, Richard Soley, George Paolini

Editor-in-Chief: Sean Rhody
Art Director: Jim Morgan
Executive Editor: Scott Davison
Managing Editor: Anita Hartzfeld
Senior Editor: M'lou Pinkham
Editorial Assistant: Brian Christensen
Technical Editor: Bahadir Karuv
Visual J++ Editor: Ed Zebrowski
Visual Café Pro Editor: Alan Williamson
Product Review Editor: Jim Mathis
Games & Graphics Editor: Eric Ries
Tips & Techniques Editor: Brian Maso

WRITERS IN THIS ISSUE

Andrei Cioroianu, Scott Davison, Claude Duguay, George Kassabgi, Pascal Ledru, Jim Mathis, Lynn Monson, Jim Redman, David Reilly, Sean Rhody, Rick Ross, Ajit Sagar, Alan Williamson

SUBSCRIPTIONS

For subscriptions and requests for bulk orders, please send your letters to Subscription Department

Subscription Hotline: 800 513-7111

Cover Price: $4.99/issue.
Domestic: $49/yr. (12 issues) *Canada/Mexico:* $69/yr.
Overseas: Basic subscription price plus air-mail postage (U.S. Banks or Money Orders). *Back Issues:* $12 each

Publisher, President and CEO: Fuat A. Kircaali
Vice President, Production: Jim Morgan
Vice President, Marketing: Carmen Gonzalez
Advertising Manager: Claudia Jung
Advertising Assistants: Erin O'Gorman, Jaclyn Redmond
Accounting: Ignacio Arellano
Graphic Designers: Robin Groves, Alex Botero
Webmaster: Robert Diamond
Senior Web Designer: Corey Low
Customer Service: Sian O'Gorman, Paula Horowitz
Online Customer Service: Sian O'Gorman
Customer Service Interns: Angela Frasco, Ann Marie Milillo

EDITORIAL OFFICES

SYS-CON Publications, Inc.
39 E. Central Ave., Pearl River, NY 10965
Telephone: 914 735-1900 Fax: 914 735-3922
Subscribe@SYS-CON.com

JAVA DEVELOPER'S JOURNAL (ISSN#1087-6944) is published monthly (12 times a year) for $49.00 by SYS-CON Publications, Inc., 39 E. Central Ave., Pearl River, NY 10965-2306. Application to mail at Periodicals Postage rates is pending at Pearl River, NY 10965 *and additional mailing offices.*
POSTMASTER: Send address changes to: JAVA DEVELOPER'S JOURNAL, SYS-CON Publications, Inc., 39 E. Central Ave., Pearl River, NY 10965-2306.

ISSN # 1087-6944

Worldwide Distribution by
Curtis Circulation Company
739 River Road, New Milford NJ 07646-3048*Phone: 201 634-7400*

BPA Membership Applied For

SYS-CON PUBLICATIONS

FROM THE EDITOR

Sean Rhody

I Told You So

About two years ago a colleague of mine named Joe leaned over my cubicle wall and said, "Hey, I just downloaded this new language called Java. It's pretty cool!" At the time I can't remember being very excited about another programming language. I was a PowerBuilder maven and Joe was up to his eyeballs in C++. That probably accounts for some of my disinterest and Joe's initial drooling (sorry Joe, but you did). Two years and one large scale Java project later, I'm as much a convert as Joe.

That doesn't mean I want to rebuild everything that's ever been written in Java, nor does it mean I think PowerBuilder's obsolete (or C++ for that matter). I recently attended a conference where a technical representative from Sun was discussing Java for the enterprise. He asked, "Where should we use Java?" His answer was most appropriate, "Where you need it."

There's nothing a client hates more than an "it depends" answer. Unfortunately, it's often the truth. So it is with this answer. Where you're going to use Java depends on your needs and strategic direction. Should you use Java everywhere? Almost without a doubt, no. There are things Java is good at and there are things Java is not good at. There are also practical considerations, such as corporate infrastructure, that have nothing to do with Java's capabilities but impact where Java is and is not practical.

As an example of what's not practical, look at Corel's attempt to re-create its office suite in Java. In theory, Java is as suited for this as any language, more so than some with strong multitasking. But this was not the right place for Java. For one thing you need every ounce of speed on a machine to make these overprogrammed suites perform well. JIT compilers notwithstanding, native code is still faster right now.

Even worse, you know someone would get the bright idea to host this in a browser. Why buy a thousand copies when you can access a single copy over the LAN? It'd be a tossup as to who would shoot that guy first – the network administrators who were dealing with network overload or the users who were waiting hours for their new "improved" software to load.

Probably the biggest lesson that needs to be learned is that Java is part of an architecture, not an architecture unto itself. I hear companies saying, "We've got to go to Java," and I can understand their frustration and desire. The Internet has turned the safe, known world of client/server on its ear, and the closest thing to a standard that most of us can find is Java.

That's great. I'm all for Java being the language of the Net. It's compact, it's elegant and it's fun to program in. The problem is that you can't simply swap Java for whatever language you've been doing two-tier development in and expect to have a solution. For one thing, JDBC is still not as far along as ODBC or native drivers. For another, it's harder to provide the same rich GUI, at least on Windows platforms. Love it or hate it, Windows is still the overwhelming desktop today, and we need to be able to build better looking Java apps if Java is to become a dominant force on those desktops. Some of this is due to the browsers rather than to the language itself. I have to applaud the people who put HTML together as a document language, but as an application environment it leaves a lot to be desired.

So what do we do? It's pretty simple really. We need to put Java where it belongs. It's not the only tool we have, and we must have good reasons for selecting it over other languages and products. At the same time we need to push for improvements in the browsers and compilers, and hope that a JavaOS will actually make sense, both from a programmatic and, in an era of $700 PCs, an economic sense. Meanwhile, I need to call Joe and tell him he was right. I hope he doesn't rub it in.

About the Author

Sean Rhody is a senior consultant with Computer Sciences Corporation where he specializes in application architecture, particularly distributed systems. He is also Editor-in-Chief of **Java Developer's Journal.** *You can contact Sean at roadhog@nac.net.*

SYS-CON PUBLICATIONS

PHONE, ADDRESS & WEB DIRECTORY

CALL FOR SUBSCRIPTIONS

1 800 513-7111

International Subscriptions
& Customer Service Inquiries

914 735-1900

or by fax: 914 735-3922

E-Mail: Subscribe@SYS-CON.com
http://www.SYS-CON.com

MAIL All Subscription Orders or
Customer Service Inquiries to

Java Developer's Journal
JavaDevelopersJournal.com

Cold Fusion Developer's Journal
ColdFusionJournal.com

National Java Learning Center, Inc.

JDJ Buyer's Guide
JavaBuyersGuide.com

Web-Pro Developer's Supplement

SYS-CON Publications, Inc.
39 E. Central Ave.
Pearl River, NY 10965 – USA

EDITORIAL OFFICES
Phone: 914 735-1900
Fax: 914 735-3922

ADVERTISING & SALES OFFICE
Phone: 914 735-0300
Fax: 914 735-7302

CUSTOMER SERVICE
Phone: 914 735-1900
Fax: 914 735-3922

DESIGN & PRODUCTION
Phone: 914 735-7300
Fax: 914 735-6547

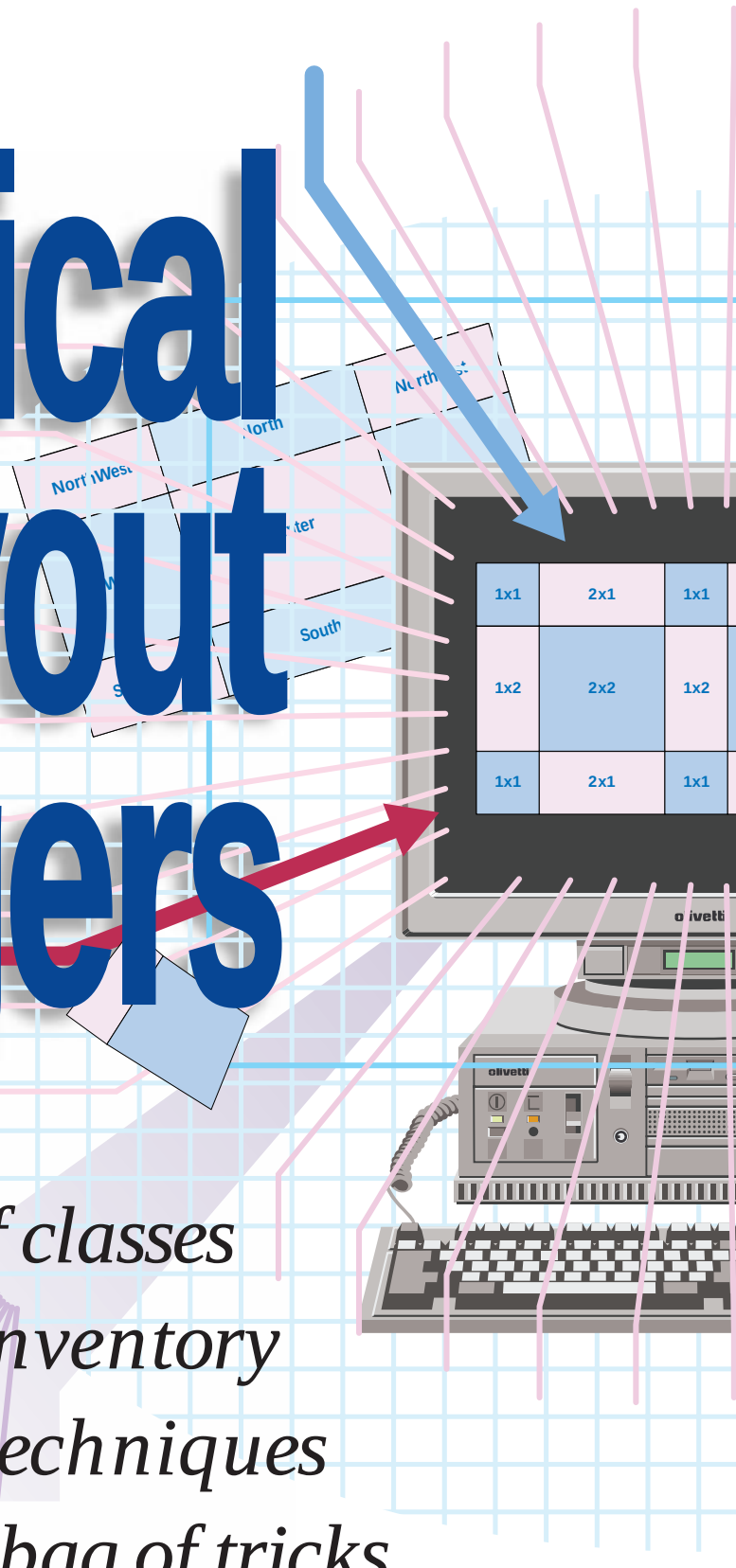
Worldwide Distribution by
Curtis Circulation Company
739 River Road,
New Milford NJ 07646-3048
Phone: 201 634-7400

DISTRIBUTED in the USA by
International Periodical Distributors
674 Via De La Valle, Suite: 204
Solana Beach, CA 92075
Phone: 619 481-5928

David Gee

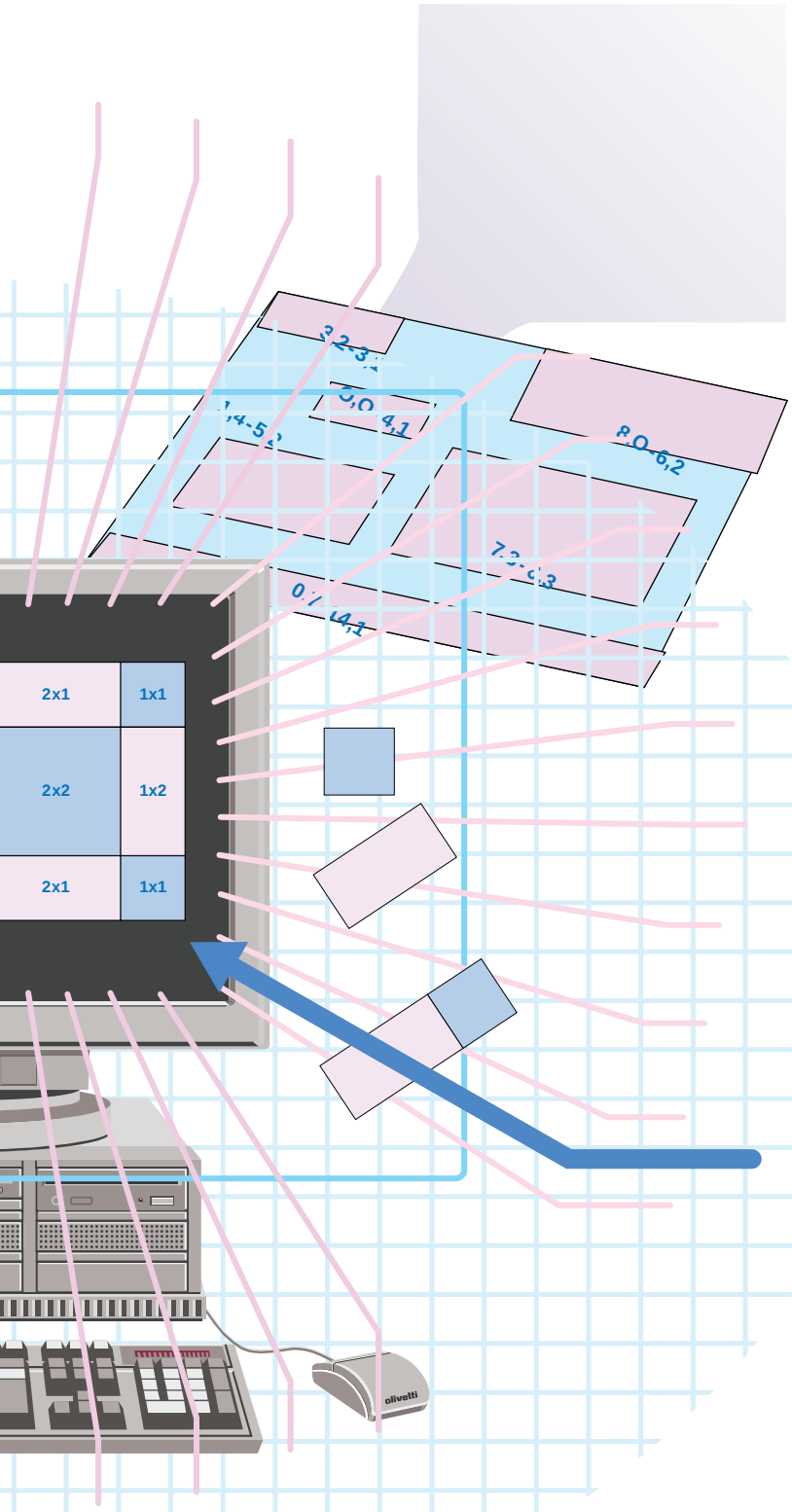
The Component Choice for e-business

Practical Layout Managers



by **Claude Duguay**

Add a trio of classes to your inventory and a few techniques to your bag of tricks



This article tries to take the mystery out of the black art of developing layout managers. Much of the coverage in books and magazines typically centers on trying to wrench the complicated GridBagLayout into submission or demonstrates the development of a layout manager with virtually no practical use. In the real world of software engineering, the need for applicable solutions takes precedence. This article will add a trio of reusable classes to your inventory and, hopefully, a few techniques to your personal bag of tricks.

If you've done any Java user interface development, you've probably used the BorderLayout and GridLayout to some extent. In fact, you probably figured out pretty quickly that nesting these layout managers typically gives you the control you need to get the job done, getting the layout to look the way you intended. Since one of Java's primary objectives is portability, you've probably managed to avoid hard-coding positions by setting the layout manager to null. If you haven't, you really should stop doing that. Unfortunately, some RAD tools tend to do this by default. But the payoff is considerable if you use proper layout managers.

Practical Layout Design

We can learn a little from the layout managers that ship with the Java Development Kit and a couple of the new layout managers released with the Java Foundation Classes. Table 1 gives a quick summary of those managers. Pay special attention to the descriptions.

The JFC also introduces the ScrollPaneLayout and ViewportLayout managers, but they are so tightly coupled to their respective JScrollPane and JViewport components that they are virtually useless in other contexts.

You'll notice a few things about these:

- Each has a simple purpose, described in a single sentence and easy to understand.
- None of them use exact positioning so they avoid tight coupling to a given display area.
- Each method has a natural organization behind it, often reflected in a real-life approach.
- All layout managers have the Layout suffix as part of their name.

Table 2 covers the implementation of new layout managers. We'll elaborate on their internal function after a quick look at the layout manager interfaces.

The LayoutManager Interface

The basic design of a LayoutManager is fairly simple. You need to declare your class as implementing the LayoutManager interface and then implement the following methods:

- addLayoutComponent(String, Component)
- removeLayoutComponent(Component)
- layoutContainer(Container)
- Dimension minimumLayoutSize(Container)
- Dimension preferredLayoutSize(Container)

The addLayoutComponent method is called when you use the add method in your container. Typically, the add method has only one parameter, so the addLayoutComponent method is called with the string set to null. With the BorderLayout, the label indicating position is passed using the add method with the string and component arguments. The layout manager handles the add with the addLayoutComponent method. The remove method is seldom used in a container but it works pretty much the same way, calling the removeLayoutComponent method. The layoutContainer method is called when you or the system calls the doLayout method. Two additional methods, getMinimumSize and getPreferredSize, are provided to determine the minimum and preferred layout sizes.

The LayoutManager2 Interface

Most of the shortcomings in the LayoutManager design stem from the need to use a String to describe positional or constraint information in the add method. The LayoutManager2 interface extends LayoutManager to deal with this more effectively. With the extended addLayoutComponent method, you can pass any object you want to handle specialized circumstances. The container add method calls this method automatically if it has the component, object argument signature.

The following methods in the LayoutManager2 interface extend the LayoutManager interface, and therefore require those methods

to be implemented as well:

- addLayoutComponent(Component, Object)
- Dimension maximumLayoutSize(Container)
- invalidateLayout(Container)
- int getLayoutAlignmentX(Container)
- int getLayoutAlignmentY(Container)

The getMaximumSize method in containers is supported by calling the maximumLayoutSize method in the layout manager. This functionality was missing in the earlier interface and seems like a natural extension. If the layout manager or supporting classes make changes to the container in important ways, the invalidateLayout method accommodates forcing a repaint operation. Finally, the getLayoutAlignmentX and getLayoutAlignmentY methods provide a way of determining how to align the component. The value returned indicates a relative position along the specified axis. A value of zero (0) indicates the origin and one (1) furthest from the origin. A value of 0.5, typically used as a default, indicates a position in the middle.

The AbstractLayout Class

When I developed the three layout managers presented here, I first wrote them independently. But as often happens in a development project, I noticed several common elements and revised the code to move these commonalties into a parent class. The resulting abstract class is reusable, as you might expect, and provides a number of common and default behaviors that help make the code much thinner. Figure 1 shows the class hierarchy for the layout managers and their AbstractLayout parent class. Let's take a quick look at the commonalities.

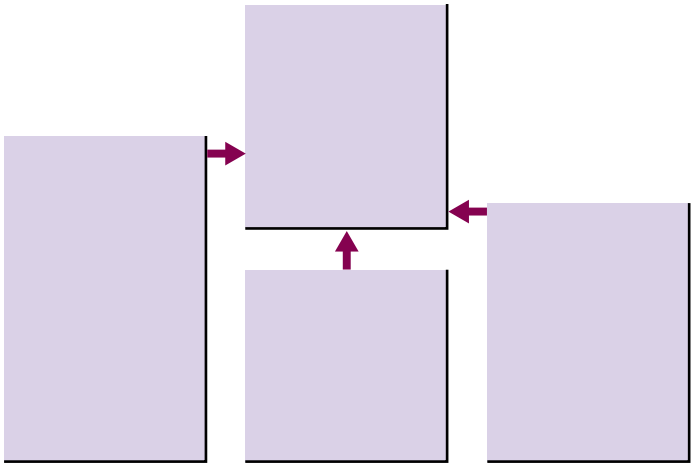


Figure 1: Practical layout managers class hierarchy

The layout managers allow you to specify horizontal and vertical gaps between components. They have accessor methods associated with them so the values can be retrieved and set outside the layout manager's constructor. Internally, they are represented as integer member variables named hgap (horizontal gap) and vgap (vertical gap). The accessors follow the JavaBean convention and are named with the get and set prefix. These methods will not be presented in this article, but you can find the entire code base on the ***Java Developer's Journal*** Web site.

The maximumLayoutSize method always returns a Dimension object with Integer.MAX_VALUE set for the width and height, leaving the maximum size unconstrained. We'll present specific code for the preferredLayoutSize method, which calculates the appropriate dimensions for the full layout manager for each respective layout manager. The similarity to minimumLayoutSize is so pronounced however, that it's not worth presenting them individually. The difference between them lies in the calls made to getPreferredSize for each component in the preferredLayoutSize method and the use of getMinimumSize in the minimumLayoutSize methods, respectively.

When implementing the LayoutManager2 interface, we need to provide the getLayoutAlignmentX and getLayoutAlignmentY methods. By convention, these always return a value of 0.5, which indicates centering as the default position for smaller containers. The layout managers we present always resize the component anyway, so they have no real effect on the layout behavior. If you have more specific needs, you can override this method in a subclass at your discretion.

The addLayoutComponent and removeLayoutComponent handling is not order dependent except in the CastleLayout, which has only nine positions and thus controls where in the component array each is stored. As such, we'll present only the CastleLayout code. The AbstractLayout class simply assumes that the container handles these values. The ScalingLayout manager extends this behavior and stores the constraints in a hash table.

Finally, the toString method is always implemented, as is the invalidateLayout method, which actually does nothing though it has to be present to implement the interface. This method is used only to force a repaint in specialized layout managers.

Layout Manager	From	Description
BorderLayout	JDK	Lays out a container, arranging and resizing its components to fit into five regions: north, south, east, west and center.
CardLayout	JDK	Acts like a stack of cards, treating each component in the container as a card where only one card is visible at a time.
FlowLayout	JDK	Arranges components in a left-to-right flow, like lines of text in a paragraph.
GridLayout	JDK	Lays out a container's components in a rectangular grid.
GridBagLayout	JDK	Lays out components based on a comprehensive constraint object that defines relative positional information
BoxLayout	JFC	Allows multiple components to be laid out either vertically or horizontally.
OverlayLayout	JFC	Arrange components on top of each other, overlapping where their dimensions require it.

Table 1

Layout Manager	Description
CastleLayout	Arranges components to fit into nine regions: north, south, east, west, northwest, northeast, southwest, southeast and center.
ProportionLayout	Lays out components in a proportional rectangular grid defined by a collection of specified row heights and column widths.
ScalingLayout	Lays out components in a scalable regular grid where components can occupy any specified rectangular region, possibly overlapping.

Table 2

The CastleLayout Class

The CastleLayout manager is designed to handle a number of real-life cases, such as:

- Setting up scroll bars or rulers in positions relative to the central panel
- Creating borders with sides and corners drawn by individual components

- Drawing legends, labels or other details relative to a central display
- Placing tool bars, logos and control elements around a central panel

Figure 2 shows how the various areas are laid out, by name.

The CastleLayout manager implements the LayoutManager2 interface, so it has to implement all the methods in the LayoutManager and LayoutManager2 interfaces. The most important methods are the preferredLayoutSize and layoutContainer, so we'll focus on those. There is some code reuse in these methods and a handful of private internal methods to make things easier. Take a look at Listing 1 to see how they work.

The instance variables and constants are listed first. The getComponent method defaults to CENTER if the name is not provided. We assume a null name is handled the same way as the BorderLayout default. The isVisible method returns true if the component is neither null nor invisible. If a component is invisible, it's considered unavailable during the layout call. The setBounds method avoids resizing an invisible component while properly handling visible components. The getSize method makes it possible to reuse the same code to get the preferred and minimum size of a given component.

The primary purpose of the interface method preferredLayoutSize is to figure out how big the container should be if it's contained in another layout manager or if the pack method is used to resize a dialog box or window frame. We need to figure out the maximum size of each of the north, south, east and west positions, depending on the components in those rows and columns. To do this we walk through the list of elements in a specific order, calculating the largest size at each stop. At the end we are left with the center area occupying the remaining space. Notice that the vertical and horizontal gaps are part of these calculations and that the insets for the layout are factored in as well.

Listing 2 shows the calculateLayoutSize method, which returns the four inset positions for the left/right widths and top/bottom heights. It calls the getSize private method with the correct type (PREFERRED or MINIMUM), which is passed to it by the calling method. The preferredLayoutSize method itself then takes the center component, vertical and horizontal gaps and the container's insets into account before returning the preferred dimensions. The minimumLayoutSize is virtually identical except that the MINIMUM constant is used instead of PREFERRED.

The layoutContainer method is actually responsible for the real work in a layout manager (see Listing 3 for the code). We do this in two passes, calculating the row and column positions for the north, south, east and west elements (which, naturally, include the corners) by calling the calculateLayoutSize again. Once we have those figured out, we can resize the components with the setBounds method. Like the preferred and minimum size calculations, we account for the vertical and horizontal gaps and the container's insets. Any remaining space is given to the center component.

The ProportionalLayout Class

The ProportionalLayout manager:

- Lets the programmer control the width of individual columns in a grid layout
- Lets the programmer control the height of individual rows in a grid layout
- Provides an easy way to create homogeneous rows or columns

Figure 3 shows how the rows and columns can vary. The values are shown as 1 and 2 for simplicity but they are completely arbitrary and relate only to each other. Any integer can be used, so long as the total of all rows or all columns doesn't go over Integer.MAX_VALUE.

Listing 4 shows how the row proportions are precalculated when

NorthWest	North	NorthEast
West	Center	East
SouthWest	South	SouthEast

Figure 2: CastleLayout manager arrangement

1x1	2x1	1x1	2x1	1x1
1x2	2x2	1x2	2x2	1x2
1x1	2x1	1x1	2x1	1x1

Figure 3: ProportionalLayout manager arrangement

the value is set. This might take place in the constructor or with an explicit call to the setRows method. There are two signatures for setRows, one of which allows you to set an arbitrary number of rows to 1. This provides the same behavior as the GridLayout manager and may be applied to one or both dimensions (vertical and horizontal). The setCols methods are not presented because they are virtually identical except for the orientation. The real trick lies in maintaining an integer array as well as a floating point array. The first stores the integer proportions used by the layout manager. The second normalizes those values into a range between 0 and 1 so that laying things out is easier.

The ProportionalLayout implements the LayoutManager interface so it has fewer methods than the other two layout managers. Given that we are inheriting from the AbstractLayout class, this matters little, but it's worth noting anyway. Like all layout managers, the preferredLayoutSize and minimumLayoutSize represent a good part of the work to be done. Listing 5 shows the preferredLayoutSize method. We loop through each row and column, accounting for the possibility that fewer than the maximum number of components might be used, and determine what the relative unit size should be. We then loop through each orientation to add up the width and height required. As you might expect, the minimumLayout size is virtually identical except for the call to getMinimumSize to get the dimensions.

Once again, the real workhorse is the layoutContainer method. Listing 6 shows how we start our calculation by determining what the unit size has to be. We use floating point numbers to avoid gross rounding errors during scaling. If we used integers directly, the boundary conditions would show up unevenly in our display, making some units noticeably larger or smaller than the average. We loop through the contained components and resize them based on the relative row and column sizes. All we have to do is call setBounds for each container and we're done.

The ScalingLayout Class

The ScalingLayout manager:

- Lays out components positionally without tight coupling to display resolution
- Allows scaling without destroying relative positioning of components
- Allows objects to overlap if necessary – drawing windows, for example
- Is ideal for drawing line art, where scale changes but not relative positions

Figure 4 shows how the ScalingLayout might look for six components placed at various positions in the scalable grid area. For clarity, the rectangle values are represented as x,y-width,height.

The ScalingLayout implements the LayoutManager2 interface. We use the java.awt.Rectangle object to describe constraints. The default Container behavior does not actually save this information. To handle this properly, we override the default addLayoutComponent and removeLayoutComponent to handle this explicitly. We use a HashTable object to store the Rectangle constraint associated with each object and remove it as appropriate. Listing 7 shows these simple methods. Notice how we check the instance type for the constraint parameter in the addLayoutComponent method to help developers easily find the problem if they use a different object type.

By now, the preferredLayoutSize and layoutContainer methods should be old friends. Not surprisingly, Listing 8 shows how the preferredLayoutSize is handled and the implication remains that the minimumLayoutSize method is almost identical except for the underlying calls to getPreferredSize. Since we know the grid is made up of uniform cell sizes, we find the largest vertical and horizontal unit sizes and simply multiply by the number of rows and columns to get the preferred and minimum dimensions.

The layoutContainer method is presented in Listing 9. We first calculate the vertical and horizontal unit size for each grid cell as a floating point value, accounting for the container insets and the vertical and horizontal gaps. Then we lay out each of the components in the order they are found in the Container. This is an important point if your objects overlap, since the last drawn components will overlap those that are drawn first. Notice that when we calculate the width and height, we work backward from the calculated *x* and *y* positions for the left and right or top and bottom positions. Rather than setting the component bounds to the *x*,*y* position and a calculated width and height, we figure out the right and bottom positions (based on the rectangle width and height) and then work out the relative bounds' width and height. Doing this the more obvious way results in occasional single-pixel gaps that you don't really want to see.

Summary

Figure 5 shows a digital clock example that is implemented almost entirely with layout managers, in particular the three presented in this article. The code is not presented here but can be found on the ***Java Developer's Journal*** Web site, along with the layout manager and related test harness code. The clock uses the JFC, so you'll need to have it installed to try it out. The CastleLayout is used to place the sides and corners; the ProportionLayout is used for each digit and colon. The ScalingLayout is used to position the digits and the colon in the central area. The whole thing is implemented as the main method in the Crystal class, given that it's the only new class required to draw the rounded green crystals. The rest of the code is static, and present only to make the nested layout code more readable. There's no actual timing code, since this was meant as an example, but it wouldn't take much to turn it into a valid prop for Jeff Goldblum if you're shooting the sequel to *Independence Day*, given that it can easily scale to fill the laptop screen for effect.

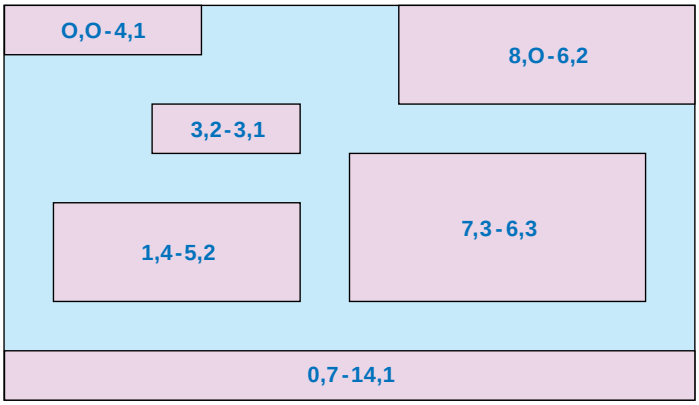


Figure 4: ScalingLayout manager arrangement

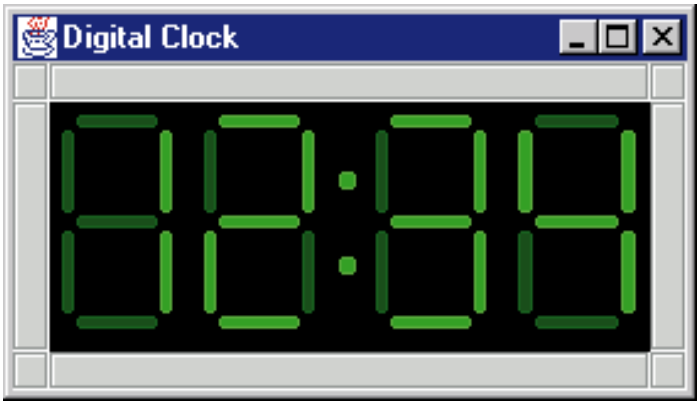


Figure 5: Digital clock example

Layout managers are rarely implemented by developers working on Java projects, yet they represent one of the simplest ways of controlling component positioning without tight coupling to the display resolution. While they have been discussed in various books and articles, few have presented a practical approach to developing solutions with reusable layout manager classes. In many cases it is simpler to develop a new layout manager than to nest the various existing managers to get the desired look and feel. My own experience with the GridBagLayout has never led to a satisfying, predictable result. If you've mastered and prefer it, you have my blessing.

I've tried to present enough information and example code to make developing custom layout managers an easy choice for you to make when the opportunity presents itself. You should now have a good foundation to help you make an informed choice next time you are presented with a problem that would benefit from a custom layout manager. Finally, if you take nothing more away from this article than the three layout managers implemented here, you should have gained a trio of useful components for your programming arsenal which will hopefully serve you well in your programming endeavors.

CODE LISTING

The complete code listing for this article can be located at **www.JavaDevelopersJournal.com**

About the Author

Claude Duguay has been programming since 1980. In 1988 he founded LogiCraft Corporation, and he currently leads the development team at Atrieva Corp. You can contact him with questions and comments at claude@atrieva.com.

claude@atrieva.com

Persistent User Interface for Multiuser Applications

Differentiating between profiles using object serialization

by **Andrei Cioroianu**

I'm starting my computer. I'm waiting for the operating system to be loaded. Now I can see the icons of my favorite applications. They're aligned on my desktop and in the same positions I left them.

I'm launching a development tool. It's creating its own windows and loading the toolbars I need. I select a project I want to work on from the list of those I used most recently. Everything is the same as it was when I closed this project.

What is hiding behind this mechanism? Lots of code that saves the coordinates of the windows, the position of the cursor in each window, the position of the visible text, the position of the selected text and many other parameters that determine the current state of the application's user interface. Each project has several files in which it keeps all this data. The application has its own config files. The programmers who made the tool worked very hard, and the persistence of the interface between two consecutive sessions is timesaving for users.

Now I'm launching one of my applications written in Java. Its window is placed somewhere on-screen at random coordinates or in the upper-left corner. This phenomenon doesn't appear at the Java applications that use property files, in which they keep the coordinates and dimensions of the windows. If the user interface is complex, however, then the property files become inadequate. You might be will-

ing to spend money and time to write the code for reading and writing files in an application-specific format, but maintaining that code will be a real pain. Each change of the file format will require modifications in the source code in at least two places (the read and write routine). Is there a simpler solution? Yes. You can use object serialization.

Why Choose Java?

Suppose you need an application that allows each user to connect to a network, depending on his/her position(s) in the company: manager, designer or developer (Figure 1). The users won't want to redefine their profile each time they run the application, so the profile should be serialized. So as not to grow the line count of the source code more than necessary, I won't introduce options for colors or fonts. For a real-world application, these options might be compulsory. A user who has a few complex profiles might want to identify them rapidly according to the background color of the application (which becomes the parameter of the profile). Using the simple application described in this article, the user will be able to differentiate the profiles, depending on the on-screen position of the application's window.

For a heterogeneous network you will choose Java. But the cross-platform support isn't the only advantage Java provides. Even if all users have the same operating system, they will still have displays with different resolutions, so fonts and controls will have different sizes. Few frameworks offer the flexibility of the Layout Managers from AWT. Sometimes you'll hear criticism that the AWT-based interfaces should be tested on all target platforms. It's impossible to write applications whose interfaces fit, on the first try, in all user preferences. However, an AWT dialog box can be resized, and all the UI components will be automatically reshaped. Users can resize the windows so the fonts and controls of their platform are correctly shown, instead of using font sizes that are established when the application is written. It remains only to serialize the dimensions of the windows so they don't have to start all over again each time they launch the application. It's unfair to criticize AWT, which has, in addition, a multithreading architecture. Since most native applications use dialog boxes that are modals with fixed dimensions, the code of these applications either isn't reentrant or is running in a single thread.

If you're still undecided about using Java, I'll give you one more reason. The Object Serialization API provided by Java is easy to use, flexible and scalable. You won't find something like this in other popular programming languages, such as C++, which produces native code, because the Serialization API is based on Reflection API. The latter provides information at runtime, which in the case of C++ native applications is available only at compile-time. I must say that MFC of Visual C++ provides some support for serialization, but the developers have to replace lots of TODOs with their own code. Unlike MFC, Java handles many issues automatically with the help of the reflection API. Sometimes, being interpreted means being superior.

The SmartLogin Application

The main class of the application, SmartLogin, extends java.awt.Frame. The member variables are a constant -- okay, a string -- profile, which keeps the name of a file and several variables that reference the UI components of the application. The application's user interface will be serialized in the profile file.

```
public  static final String OK = "OK";
```

```
private transient String profile;
private TextField tName, tPassw;
private Checkbox  cMan, cDes, cDev;
private Button    bOk, bCancel;
```

The constructor of the SmartLogin class (Listing 1) receives as parameter a string taken from the command line, which represents the name of the user's profile. A reference to this string will be stored in the profile member variable. The constructor sets the title, size and the LayoutManager of the window. The last one is GridLayout(6,1). In the first two lines it inserts a label -- "Name:" – and a TextField -- tName – in which the users will type their name. In the third line are the three Checkboxes that define the user's profile, cMan, cDes and cDev. The next two lines contain a label --"Password:"-- and a TextField -- tPassw -- in which users will type the password. The last line groups two buttons -- bOk, and bCancel. The events which are fired when the buttons are pushed or the window is closed when intercepted by a SmartAdapter instance.

The SmartAdapter class (Listing 2) extends java.awt.event.WindowAdapter and implements java.awt.event.ActionListener, java.io.Serializable. The SmartLogin() constructor receives as parameter an instance, sd, of the SmartLogin class. The actionPerformed() method implements the method with the same name of the ActionListener interface. This method is called when one of the buttons is pushed. For the Ok button it calls the login() and serialize() methods of the SmartLogin instance. The windowClosing() method overrides the method with the same name of the WindowAdapter class. This method is called when the user closes the application's window. The Serializable interface has no methods. However, the SmartAdapter must "implement" it for it to be serializable. The SmartLogin class needn't implement Serializable, because one of its ancestors (java.awt.Component) implements it.

The login() method of the SmartLogin class simulates a login. It shows a message like the following one:

```
Hello Andrei
You have logged as designer and developer
```

The serialize() method first calls clearPassw() and then tries to serialize the user interface of the application. The clearPassw() method clears the password stored in a private variable of a TextField instance.

```
public void clearPassw() {
   tPassw.setText("");
   System.gc();
}
```

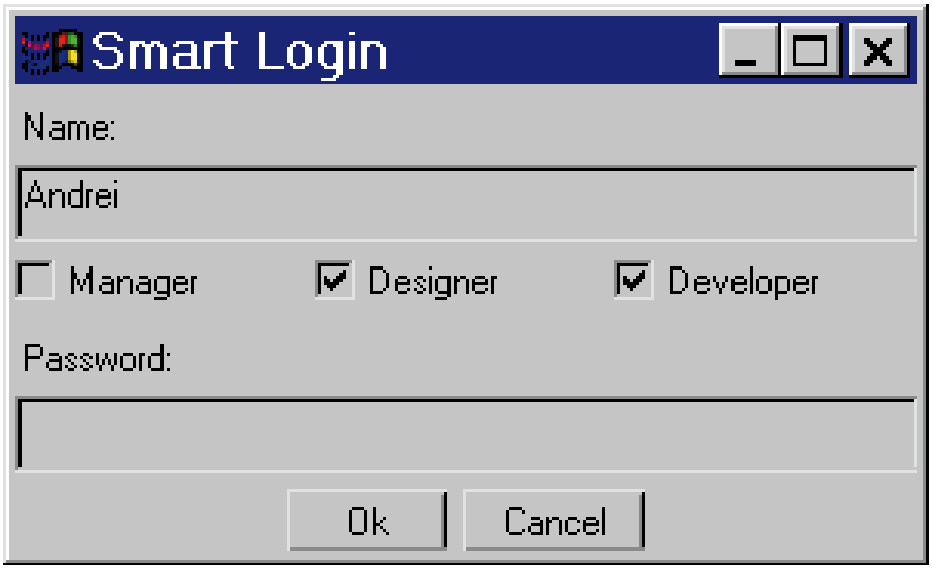


Figure 1: The user interface of SmartLogin app

After it calls clearPassw(), the serialize() method uses writeObject() of the java.io.ObjectOutputStream class to serialize the SmartLogin object, which represents the application's window. The writeObject() method is applied recursively on the member variables of the object passed as parameter and takes care not to serialize the same object twice. The OK and profile variables aren't serialized because one of them is static and the other one is transient.

```
FileOutputStream out = new FileOutput-
Stream(profile

+ ".ser");
ObjectOutputStream s = new ObjectOutput-
Stream(out);
s.writeObject(this);
s.flush();
s.close();
```

The main() method of the SmartLogin class first tries to deserialize the user interface. For this purpose the readObject() method of the java.io.ObjectInputStream class is used. If the deserialization succeeds, then the setProfile() method of the SmartLogin class is called to initialize the profile member variable, which wasn't serialized because it's transient. The show() method, which SmartLogin inherits from Frame, will show the window of the application.

```
FileInputStream in = new
FileInputStream(profile

                                    +

".ser");
ObjectInputStream s = new ObjectInput-
Stream(in);
sl = (SmartLogin) s.readObject();
s.close();
sl.setProfile(profile);
sl.show();
```

If the deserialization fails, then an exception is thrown. This is caught and shown at console only if its type isn't FileNotFoundException (the profile files may not exist if they haven't been created or have been deleted). A new instance of the SmartLogin class is created and the application's window is shown.

```
sl = new SmartLogin(profile);
sl.show();
```

See Reference 2 for more information about object serialization.

Control What Is Serialized

While the serialization mechanism is simple, using it without a minimum analysis may generate inconsistency, bad performances or even security holes. This is actually true for most applications whether they use object serialization or not. Fortunately, these problems can be easily solved.

You probably noticed that I used the transient keyword when I declared the profile member variable. Hence, this variable is ignored by the serialization mechanism because the name of the profile can't be part of the file without generating worry about whether an inconsistency will be determined. Nevertheless, this operation shouldn't be made when the SmartLogin application is running. When the user interface is deserialized, the profile variable is initialized with a default value. It's the programmer's responsibility to give it a correct value. This is the reason I called the setProfile() method.

Storing constants in the serialization stream is futile. Therefore, I declared the OK variable with the static keyword. As usual, the constants of a class and other

variables that need large amounts of memory (as precomputed tables) are declared static, to be shared between the instances of the class. Note that the static variables are ignored at serialization and mustn't be initialized in constructors because these aren't called at deserialization.

There is one more source of information, the password. For security reasons it must not be serialized. SmartLogin uses a TextField component to obtain the password. However, this component must be serialized because it's part of the user interface. The solution is to call the setText() method of the tPassw object to replace the password with an empty string. This operation is made by clearPassw(), which is called whether or not the user interface is serialized. The garbage collector is invoked as a supplementary cautious measure. This should be enough, but you can never be sure.

If you don't want a variable to be serialized, then you declare it static or transient. You might wonder how additional information could be stored in serialization streams. You can add write/readObject() methods to your serializable classes. These methods will be called by the write/readObject() methods of the ObjectOutput/InputStream classes, instead of the default serialization mechanism, which is still available via defaultWrite/ReadObject(). If this solution isn't sufficient, then you can implement the java.io.Externalizable interface instead of java.io.Serializable. The externalizable classes have total control of what is serialized. They also control the format of the serialized data. See "Object Serialization Specification" (Reference 2) for details.

Sometimes, you need to serialize information to which only authorized persons should have access. The easiest solution is to put filters between FileOutput/InputStream and ObjectOutput/InputStream. These filters should encrypt/decrypt the information, which you can store on disk or send over the network after encryption. However, the encryption algorithm should be chosen carefully because the format of the serialization stream is public. A safer solution is to combine encryption with the use of externalizable interface.

Be Sure Your Application Is Closing

The SmartAdapter class is responsible for the close of the application (Listing 2). This class is serializable because it implements java.io.Serializable. The constructor of the SmartLogin class will create a SmartAdapter instance that will be registered as the listener for some events (Listing 1). This object will be serialized together with the user interface because it is referenced by the components it was registered with. Still, if SmartAdapter doesn't implement Serializable, something strange will happen. When the application is run for the first time, everything will seem to be okay. But at the next run, with the same profile, the application won't close because the SmartAdapter instance couldn't be serialized at the first run because of a bug in java.awt.AWTEventMulticaster. (For more information, see Reference 3, "Serializing UI Components". I did report the bug to Sun, so it may already be corrected by the time you read this article.)

What Are the Advantages?

From the user's perspective, the UI persistence is timesaving. The application becomes friendlier because you don't have to repeat the same operations each time you start it. Resizing the application's window together with persistence might be an unusual solution for a well-known issue; the fonts and controls have different sizes on different computers even if the platform is the same.

> *"Lots of code determines the current state of the interface"*

From the programmer's point of view, object serialization is easy to use. You don't need to read "Object Serialization Specification" or have work experience with files in Java (this might be a common case because many Java programmers develop only applets, in which the use of streams is restricted for security reasons). But more importantly there is no supplementary cost for code maintenance.

Possible Inconveniences

Some people say that using serialization for the persistence of the coordinates of windows wastes space, but this isn't an issue in most cases. However, the transient keyword must be used to separate the application's user interface from the text or the tables edited in windows so that the application's data won't be serialized together with the interface. The data must be stored in separate files.

The SmartLogin application isn't perfect. A few users might try to log on with the same profile at the same time. If they modify the profile, the changes will be overridden on disk. This problem should be solved by the login() method, which should be able to detect the possible conflicts. The problem isn't relevant for this article.

Back to My Computer...

Before I stop it, I have to close all the applications so they can serialize their state. Only some of them can do that, and some of them provide "better" persistence than others. Wouldn't it be nice to have a button on the desktop that would serialize the user interface of all open applications? This way, the next time I start my computer, I won't have to restart them. This might sound utopian, because all native applications should cooperate with the operating system which doesn't provide a standard mechanism for serialization. Theoretically, you might think to save the image of the memory, the registers of the microprocessor and the state of the other hardware equipment. This won't work and wouldn't be practical. Imagine what would happen if the computer was connected to a network when it was closed. The applications must cooperate. If all the applications were written in Java and they knew how to serialize their state (another utopia), the listener of a simple button could save the state of all applications in a single bytestream that I could send to somebody, who could continue my work from the point I left it (this person could be me moving from one computer to another). The class files could then be downloaded from a Web server. This mechanism can work for a single Java application (an example is SmartLogin) and this might be enough. It's up to you to implement UI persistence for your applications.

References:

1. *The AWT Home Page*
   http://java.sun.com/products/jdk/awt/index.html
2. *Object Serialization Specification*
   http://java.sun.com/products/jdk/1.1/docs/guide/serialization/index.html
3. *Serializing UI Components*
   http://www.geocities.com/SiliconValley/Horizon/6481/AltUI11s.html

CODE LISTING

The complete code listing for this article can be located at **www.JavaDevelopersJournal.com**

About the Author

Andrei Cioroianu is an independent Java developer. He has a BS in mathematics-computer science and an MS in artificial intelligence. His focus is on 3D graphics (Java 3D), software components (JavaBeans) and user interface (AWT, JFC). You can reach Andrei for questions or comments at andcio@hotmail.com.

andcio@hotmail.com

JOutlookBar

For the user, alternative views in an application

by **Claude Duguay**

The Widget Factory is a series of articles (a regular column) dedicated to showing you how to develop sophisticated user interface components for your Java programs. We build on the foundation provided by Swing and the Java Foundation Classes, so the only assumption we'll make is that you can compile and run such programs (this code was tested under JDK 1.1.6 with Swing 1.0.2).

This first installment of *The Widget Factory* explores the development of a component called JOutlookBar, which resembles the Microsoft navigation bar provided first in their Outlook mail client and later in programs like FrontPage, Project and TeamManager.

Figure 1 shows what the finished product will look like. The Alan button is highlighted because the mouse is over it. The Dev, Ops and QA buttons allow you to switch contexts, presenting the user with lists of different icons. You'll also notice the down arrow inside the folder list, which allows you to scroll down when the list is longer than the display area. A complementary up-arrow button appears if the top part of the list is scrolled up.

The JOutlookBar component is primarily intended to present the user with alternative views in an application. This example demonstrates the basic mechanics only. The high-level interface is simple enough. You can add Action objects to the JOutlookBar component, based on arbitrary contexts, and have it become useful immediately.

Decomposition

Sometimes the simplest things are deceivingly simple. Let's decompose the component and see what makes it tick. Figure 2 shows the various nested pieces. The shading helps distinguish layers. As you can see, there are three kinds of buttons: the context buttons, which provide context navigation; the up- and down-arrow buttons; and the labeled icon buttons. We also have a number of nested panels and associated layout managers.

To manage context switching, and to encourage code reuse, the ContextLayout and ContextPanel handle everything at that particular level, making it possible to use these elements in alternative situations. The ListLayout is also usable outside this context. It lays out its children in a vertical sequence, adjusting position and/or width, but not height. We try to promote reuse and remove unnecessary coupling at every opportunity. The ScrollingPanel works like the JFC ScrollPane, but uses arrows at the top and bottom rather than a scroll bar.

Context Management

The ability to switch between contexts is one of the major features in the JOutlookBar. Switching contexts between icon listings and providing visual feedback through button positioning is key to the basic look and feel of this widget. With good design the coupling can be very loose and the various classes can be reused more effectively.

To keep it all generic, we develop the ContextLayout manager to handle button placement and center component management. We then put user-triggered mechanics into the ContextPanel class so we can use it outside this component if we want to. You can think of this arrangement as the View-Controller separation, where the layout manager handles the view and the ContextPanel handles user interaction.

We won't spend much time on layout manager design. Instead, check out "Practical Layout Managers" in this issue. The key methods here are addComponent, removeComponent, preferredLayoutSize, layoutComponent and setIndex.

Listing 1 shows the code for adding and removing components, and for setting the current active index. The context button is stored in the tabs vector and we use the constraint object argument to set the component that will be displayed when the button is pressed. These are stored in the panels vector. The setIndex method sets the current index value and calls layoutComponent to recalculate the layout for display.

Listing 2 shows how the preferred layout size is calculated. The code for getMinimumSize is almost identical. The getPreferredSize method calls getPreferredTabSize to determine what the largest button component dimensions are and then takes the insets into account. Notice that there is no restriction on the kind of component you use. Outside the JOutlookBar control, you could just as easily place labels here and change the context directly with the setIndex method.

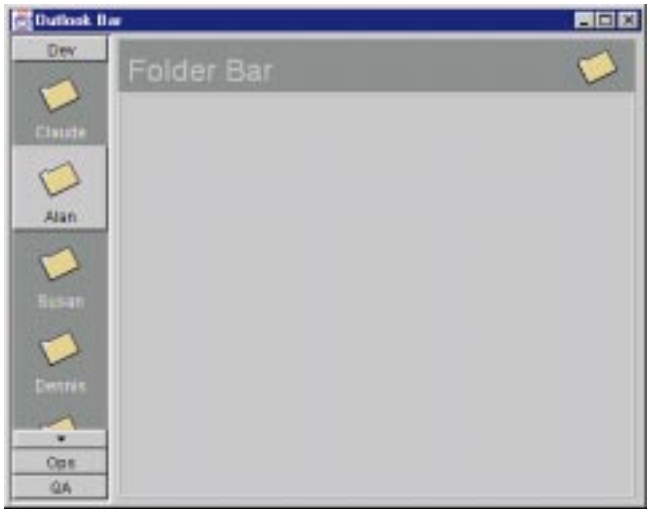


Figure 1: Outlook bar screen

Listing 3 shows how the components are actually laid out. We call two methods from layoutContainer to make things more readable. The layoutTabs method decides where the index position is and lays out buttons at the top and bottom of the display area. The layoutCenter method figures out where to put the center component that relates to the currently active index.

The ContextPanel provides a high-level view that encapsulates the behavior. Listing 4 shows the source code. The constructor sets the ContextLayout and adds a BevelBorder as a visual enhancement. Two methods are provided to

addTab and removeTab, where the internal code handles the registration and removal of button action listeners automatically. A setIndex method abstracts access to the ContextLayout equivalent and the actionPerformed method acts on user button selection.

Figure 3 shows how the ContextPanelTest class lets you view three context panels with colored panels in each.

The Scrolling Panel

The ScrollingPanel uses the JViewport paradigm to allow scrolling within the window area. It implements a constructor and only two methods. The setBounds call is intercepted to deal with window resizing and we implement the ActionListener interface to handle button presses. It's worth noting that we use the BasicArrowButton from the JFC. This is an undocumented class. There's no risk it will disappear anytime soon since it's heavily used by components like JScrollBar and JComboBox, but it's good to be cautious with anything that's not part of the official API.

Listing 5 shows code for the ScrollingPanel constructor, setBounds and actionPerformed. The constructor creates the arrow buttons and the main viewport and sets up the panel as an action listener for each button. The setBounds method handles resizing events by resetting the display to the top, removing the north button and adding a south button if necessary. The actionPerformed event handler does the scrolling work. Most of it is a matter of calculating the display area and determining whether the arrow buttons are required. If they are, they are added to the north and/or south position(s). Otherwise they are removed. The ScrollingPanel is fully reusable as is, but if you want to scroll horizontally, you'll have to extend the code.

The Icon List

To handle the list of icons in the outlook bar, we subclass JButton to provide a RolloverButton class. Our intention is primarily to handle mouse over events, so we register as a MouseListener and control the border and color drawing explicitly. The constructor also sets a number of JButton attributes to center the icon and text horizontally, and to put the icon above the text vertically. Listing 6 shows the constructor and code for handling the mouseEntered and mouseExited events.

The ListLayout manager provides the mechanics for placing components vertically above each other, allowing for various sizes if necessary. It allows us to control horizontal justification as well, though we are interested only in the BOTH setting in this context. Notice that it's generally wiser to build a generic layout manager than one that is tightly coupled to a specific application. If anything, layout managers are meant to be used in differing contexts, so this is clearly a good design objective.

Listing 7 shows the layoutContainer method for ListLayout. It handles various justification choices, but otherwise simply keeps track of the bottom position and lays one component under the previous component until none remain. This is useful primarily in containers that scroll vertically, like a list box.

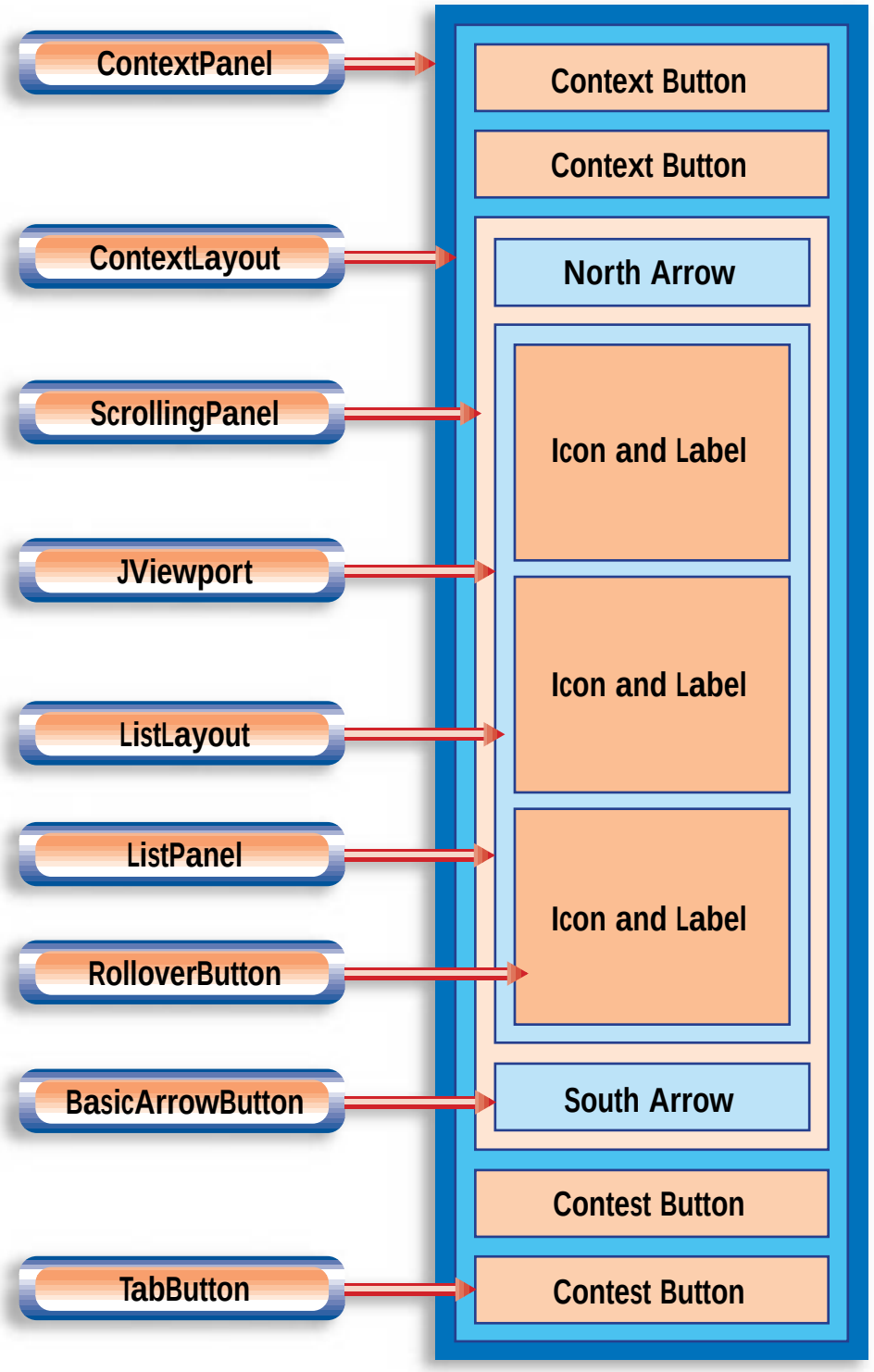


Figure 2: Nested components

Wraping It Up

The JOutlookBar class wraps everything together so you don't have to pay attention to the underlying code. You already know everything that's happening under the hood, of course, and by now you've probably thought of a few alternate uses for the classes we've already discussed. All that aside, we need to make it as easy as possible to use the component,

ADVERTISER INDEX

Advertiser		Page
Borland		19
www.borland.com	408 431-1000	
Bristol Technology		75
www.bristol.com	203 438-6969	
Coriolis		77
www.coriolis.com	800 410-0192	
Greenbrier & Russel		25
www.gr.com/java	800 453-0347	
Halcyon		35
www.halcyonsoft.com	888 333-8820	
IBM		58&59
www.ibm.com	800 426-5900	
IEC-EXPO		73
www.iec-expo.com	888 222-8734	
ILOG		17
www.ilog.com	415 688-0200	
Installshield		13
www.installshield.com	800 374-4353	
Inno Val		38
www.innoval.com	914 835-3838	
Keo Group		22&37
www.keo.com	978 463-5900	

Advertiser		Page
KL Group Inc.		B/C
www.klg.com	800 663-4723	
Live Software		41
info@livesoftware.com	619 643-1919	
Net Dynamics		79
www.netdynamics.com	650 462-7600	
ObjectShare		43
www.objectshare.com	800 973-4777	
Object Matter		50
www.objectmatter.com	305 718-9101	
ObjectSpace		4
www.objectspace.com	972 726-4100	
Object Management Group		53
www.omg.org	508 820-4300	
Progress/Cohn & Godly		21
www.apptivity.com	800 477-6473	
ProtoView		3
www.protoview.com	609 655-5000	
Roguewave		15
www.roguewave.com	800-487-3217	
Sales Vision		47
www.salesvision.com	704 567-9111	

Advertiser		Page
Silverstream		83
www.silverstream.com	888 823-9700	
Sockem Software		65
www.sockem.com	814 696-3715	
Stingray Software Inc.		2
www.stingsoft.com	800 924-4223	
SunTest		11
www.suntest.com	415 336-2005	
Sybex Books		63
www.sybex.com	510 523-8233	
The Object People		23
www.objectpeople.com	919 852-2200	
SYS-CON Publications		71
www.sys-con.com	800 513-7111	
Thought, Inc.		48
www.thought.com	415 836-9199	
Visionary Solutions, Inc.		50
www.visolu.com	215 342-7185	
WebMethod		33
www.wbmethods.com	888 831-0808	
Zero G. Software		6
www.zerog.com	415 512-7771	

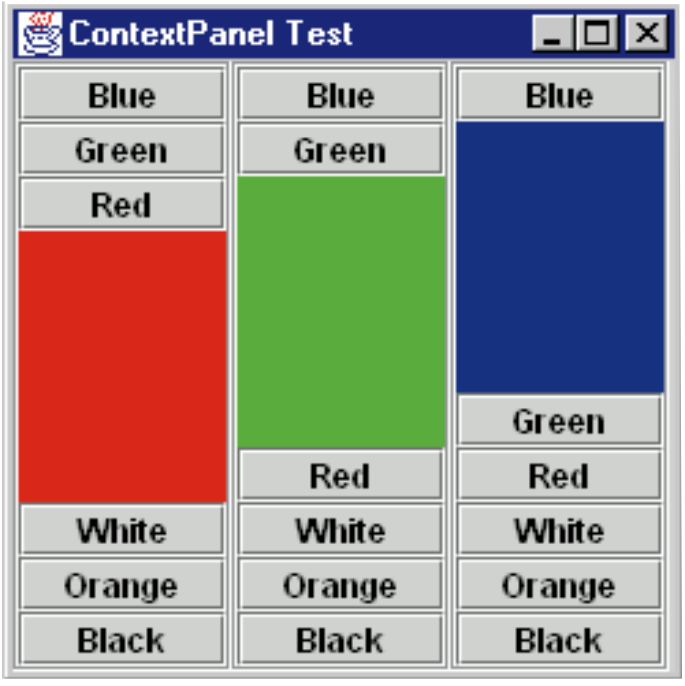


Figure 3: Context panel

so the interface looks like this:

```
addIcon(String context, Action action)
```

Okay, that should be easy enough for us, right? Listing 8 shows the code for this approach. The context is created only if it has not been seen before; otherwise we look it up and use the existing view. We then create a RolloverButton with the name and large icon value from the Action object. The large icon property is not a default property, so no icon will be present unless this value is associated. Take a look at the code for SelectAction to get a sense of how this works.

If you don't know anything about Action objects under the JFC, you'll want to read up on it. It's the preferred method for handling user interface events. You can subclass AbstractAction and implement the ActionListener interface to deal with what happens when the user selects the icon. By setting the text and icon properties in the Action object, you can use them interchangeably in this context, in toolbars or in dropdown menus.

In Closing

This is the first installment of a column I hope will bring you insight and provide building blocks you can use in future user interface design and development projects. We don't have a lot of room to explore everything in great detail, but often what you really need is the right idea and a good place to start. The JFC is a great foundation to build on and I hope the widgets coming out of our little factory will answer some of your programming questions. Next month we'll develop a JWizard framework that lets you build wizards without having to worry about the mechanics.

About the Author

Claude Duguay has been programming since 1980. In 1988 he founded LogiCraft Corporation, and he currently leads the development team at Atrieva Corp. You can contact him with questions and comments at claude@atrieva.com.

claude@atrieva.com

Listing 1.

```
protected Vector tabs = new Vector();
protected Vector panels = new Vector();

public void setIndex(Container parent, int index)
{
  this.index = index;
  layoutContainer(parent);
}

public void addLayoutComponent(Component tab, Object panel)
{
  if (panel == null) return;
  tabs.addElement(tab);
  panels.addElement(panel);
}

public void removeLayoutComponent(Component comp)
{
  for (int i = 0; i < tabs.size(); i++)
  {
    if (tabs.elementAt(i) == comp)
    {
      tabs.removeElementAt(i);
      panels.removeElementAt(i);
      return;
    }
  }
}
```

Listing 2.

```
public Dimension preferredLayoutSize(Container target)
{
  Insets insets = target.getInsets();
  Dimension tab = getPreferredTabSize();
  int h = tab.height *
    (tabs.size() + 1) + (tabs.size() * hgap);
  return new Dimension(
    tab.width + insets.left + insets.right + (hgap * 2),
    h + insets.top + insets.bottom);
}

private Dimension getPreferredTabSize()
{
  int w = 0;
  int h = 0;
  Dimension size;
  Component comp;
  for (int i = 0; i < tabs.size(); i++)
  {
    comp = (Component)tabs.elementAt(i);
    size = comp.getPreferredSize();
    if (size.width > w) w = size.width;
    if (size.height > h) h = size.height;
  }
  return new Dimension(w, h);
}
```

CODE LISTING

The complete code listing for this article can be located at **www.JavaDevelopersJournal.com**

or•a•cle \'or-e-kel,'är- \ *n.*
1. a person believed to make infallible predictions

You've got to have faith?

by **Alan Williamson**

The year isn't long, is it? Time seems to be whipping along at a tremendous pace. It seems like only a couple of weeks ago that we were at JavaOne talking over all things Java with anyone prepared to listen. We spoke to Sun, IBM and Oracle, to name but a few of the big boys. Now, Oracle...well, there's a tale to be told. But first I'd better introduce this column. I got a bit excited there.

For those of you new to this fine journal, let me introduce myself. My name is Alan Williamson, and I write this *Straight Talking* column, which is now creeping into its second month. So don't worry...you haven't missed much yet; I'm only warming up.

What's the purpose of this column, you ask? What am I supposed to be conveying in this little piece of magazine real estate? Well, here in my world I want to take you behind the marketing hype and give you a real-world view of Java. I'm one of Java's biggest fans, and I stand up for it at every opportunity. I'm not blind or blinkered to its problems, however. Journeying down the Java development road isn't always the blissful trek we are led to believe. Promises of no more pointer crashes may be true, but a host of new problems have been introduced that make chasing a pointer problem seem like an absolute busman's holiday. I intend to highlight some of them, and, I hope, save you time if and when you hit the same circumstances.

Each month I'll take another character attribute and, as a bit of fun, apply it to the world of Java. Last month I looked at trust. This time let's take a peep at faith.

What is faith? Collins' *English Dictionary* (1993) tells me it's a strong belief, without proof. Sounds like the definition I've been working with for a number of years. As a developer I generally have faith in what bigger companies tell me. For example, although we employ a number of people, my company, N-ARY Limited, is small fry when compared to the corporate giants of, say, Borland (sorry, Inprise!), Symantec, Sun or Oracle. Both Borland (sorry, there I go again, Inprise) and Symantec have been developing compilers and development tools since day one. So to say they know a thing or six about this subject would be an understatement. As Steve Martin uttered in the classic movie *The Man with Two Brains*: "When a woman who has just had brain surgery says she has a headache, you have to listen." Therefore, if one of these companies offers assistance, since it is coming straight from the horse's mouth, you feel you really ought to follow it.

But every so often even these guys fall over their own arrogance, and here I am referring specifically to Oracle and their attitude toward JDBC, Java's database classes. So this month's public health warning is JDBC drivers.

Before I begin destroying Oracle completely, I want it known that I believe them to have one of the best database engines on the market today, and they really have some cutting-edge technology. Oracle8 is a fine piece of software. However, they don't have a clue about Java. As one of the biggest database companies in the world, their apparent neglect of the Java community is something we should be concerned about. Sure, they openly promote all these new Java add-ons and extensions you can buy from them, but if they can't even code a proper JDBC driver for their own database, then doubt has to be cast on their other Java products.

Let me illustrate this with a problem that has faced around 90% of Java developers who have tried to hook their code into an Oracle database. (I have no way of proving this figure, of course, but if the newsgroups, discussions with third-party driver companies and Oracle themselves are to be believed, then I estimate the percentage to be very high.) Now this problem, sadly, is not unique to Oracle, but at least the other database companies are openly trying to rectify the problem. Oracle doesn't appear to admit, officially, that a problem exists. Nonetheless, they illustrate the power and flexibility of Java with their handling of it.

The JDBC API is a set of classes that allows developers to safely code database applications without having to worry about the specific implementation details of the back-end database. Thus it allows the developer the ability to switch to, say, an Oracle database to replace an MS-Access alternative without having to recompile their source code. A fine piece of technology, assuming you've stuck to standard SQL, that on the whole works extremely well.

The secret to the success of JDBC is the driver. Now this is a piece of software that sits between your application and the database. It's the communication bridge. Its main role in life is to pass SQL queries to the database and then collate the resulting data from the execution. It doesn't sound difficult, but this area is fraught with danger.

JDBC drivers come in four types. Some are written in Pure Java, and may be run on any Java-enabled platform; some are native libraries written for a specific platform. The driver type gives an indication of the amount of native code employed. For example, types 3 and 4 drivers are generally Pure Java solutions. Choosing the driver is an important part of the development cycle. A badly chosen driver will be slow and in some instances may crash; a good driver will be fast and efficient, and will outlive the life of your application.

Drivers that employ native libraries generally have a higher chance of crashing than drivers that are implemented in Pure Java. Therefore, the tendency to favor type 3 and 4 drivers is never a bad thing.

So...back to our story. We had been implementing a major server-side application employing over 80 servlets, all accessing an Oracle database. For the sake of convenience, all of our developers installed a local Oracle server on their NT boxes, which is used to develop and test. Early on, warning bells began to ring. There were reports of Java hanging, and General Protection Faults were observed on a daily basis. Dr. Watson became a friend to us all. Our database expert had spoken to Oracle and they

1/3 Object Matter

assured us it was the JDBC driver, but have no fear, Oracle8 will take you on the road to recovery. Hooray, we thought. We had Oracle8, and a brand spanking new Sun Solaris box to run it on which was due to arrive any day, so we lived with the problems.

In the meantime, JavaOne popped up, so our database expert and I went over there. We knew we had problems with Oracle, and we knew we weren't alone: whenever we used either of Oracle's JDBC drivers, we got GPFs after a short time of use. At this stage of the game we had committed to Oracle and couldn't really change our back end. But we were very worried about the stability. We had heard of third-party JDBC drivers, but thought, well, if Oracle can't even keep theirs up long enough, then the outside companies probably won't have any better luck.

We echoed this concern to one of the senior Oracle developers at JavaOne. He assured us that although the version of Oracle we were using, 7.3.3, had problems, Oracle8 should address them. He seemed to know about the problem and was very confident in his advice, so we were as well. We had reassurance from Oracle. We had faith that Oracle, being the size they are and dominating the database market, wouldn't let a problem like this sit for long.

We flew home, happier. Eventually, Oracle8 and the new server arrived. We eagerly installed both, and spent about two days configuring the back-end database engine to all our tables and ensuring it was running as efficiently as it could. We also installed the JDBC drivers straight off the CD-ROM. We then copied our Java packages onto Solaris and ran the test server up.

You know the feeling you get when you open a present expecting it to be the one item you've had your heart set on for weeks? And the letdown you have when you discover something else? Well, we had the same feeling.

We ran our tests for all of two minutes. This was as long as the driver stayed up. We couldn't believe it. Devastated didn't come close to what we were feeling. What were we going to do? I rang up Oracle and asked for support. They said, "Sorry, you didn't buy any." I told them, "Excuse me, but I am not paying for me to tell you about your problems. If it's something we've done wrong, then I will pay for support, but otherwise, no." Alas, the telephone support didn't see it quite like this and hung up. Charming. Now what?

Well, the power of the Internet came into play. I looked around and discovered a lot of people who had the exact same problem. I even found one chap who had collated Dr. Watson logs, Java dumps, code snippets and repeatability tests, and he got nowhere with Oracle. He did, however, point me in the direction of a San Francisco-based Java company known as WebLogic. I e-mailed them and told them my problem. They e-mailed back – within 15 minutes, I have to say – and told me not to worry, they had heard our problem a thousand times, to try this driver and it will fix our problem. I felt like a dying man who had just met a set of doctors who could cure me. It was wonderful.

I downloaded the 30-day trial, installed it within six minutes and fired up our application. Believe it or not, it is still running after two months of continual use. We didn't have to touch a single line of our code, nor did we have to change any tables. It worked, and worked well.

But the Oracle driver wouldn't work for love or money. We told Oracle, and they refused to talk to us unless we bought support. Brilliant. Here we had spent all this money on Oracle8 licenses, and when it came to the crunch, the whole system wasn't working due to a small JDBC driver. When we swapped in a third-party JDBC driver, it worked.

WebLogic was fantastic, and after talking with them for a while I discovered that the Oracle driver is one of their best product lines. No wonder. They sympathized with us and told us many others had also faced this apparent lack of support from Oracle.

The moral of the tale is that blind faith is very dangerous to have. Java is much more than simply coding classes; it's the deployment that can make or break the end application. And if it's a database-dependent application, the whole system can pivot around one small piece of software that isn't even in your control. Choose your JDBC driver very carefully, and don't stick to the one shipped as standard with your database. You could waste a lot of time.

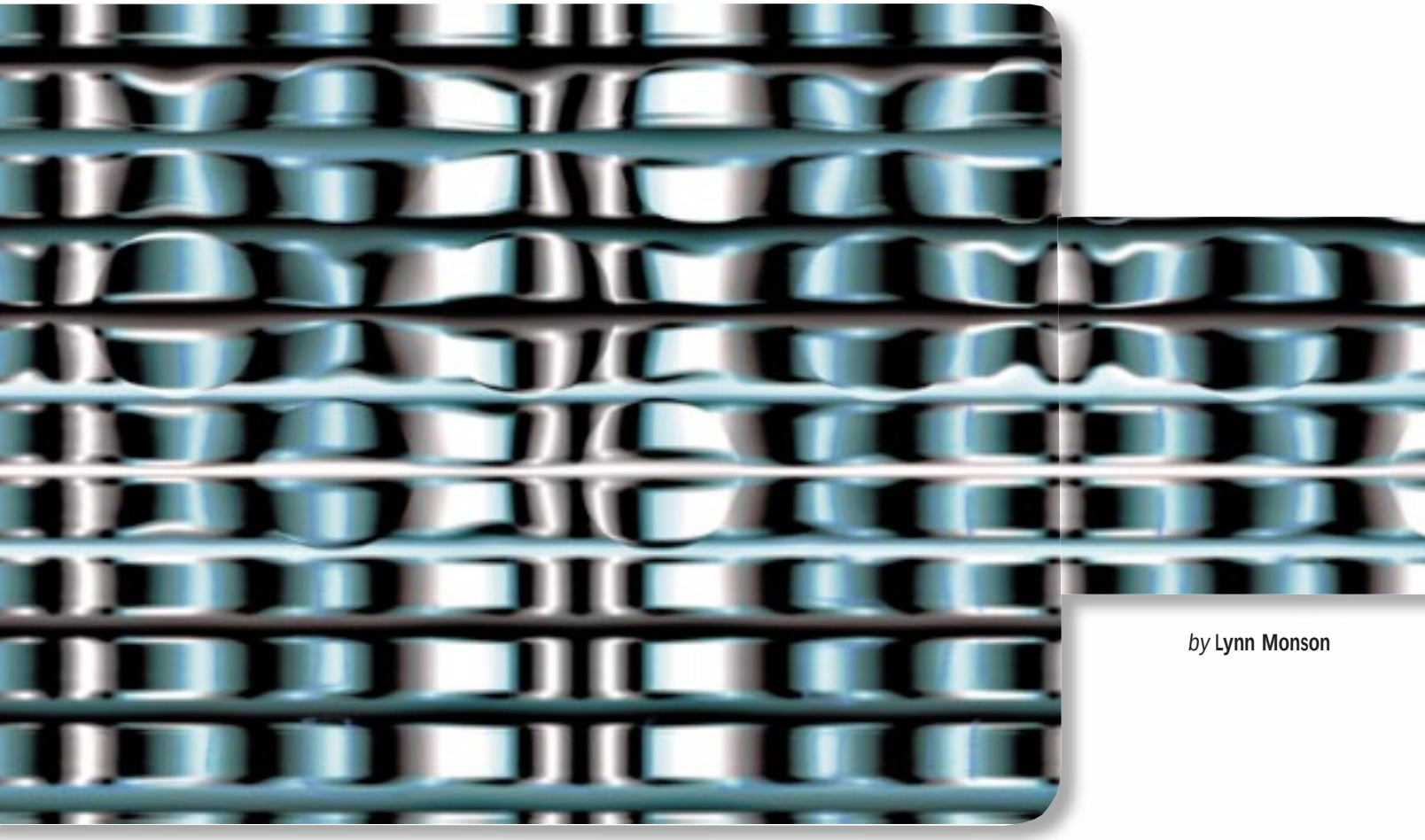
Sadly, we have joined the mass of Java developers who no longer have faith in Oracle. I think Oracle needs to take a leaf out of Sun's book with regard to how they treat developers. Once they get over this stupid support policy they have, maybe we can all work together and produce a complete package that works from end to end. I was reminded by a colleague that if Oracle were Microsoft, Mr. Gates would probably have bought WebLogic and deployed it as his own. This would definitely be a solution, as at the end of the day what all developers want is a system that works, something they can have faith in.

About the Author

Alan Williamson is on the board of directors at N-ARY Limited, a UK-based Java software company specializing solely in JDBC and Java servlets. He recently completed his second book, focusing purely on Java servlets. In his first book he looked at using Java/JDBC/servlets to provide an efficient database solution. You can e-mail him at alan@n-ary.com.

alan@n-ary.com

Caching & WeakReferences



by Lynn Monson

By careful crafting of caching algorithms, we can begin to tailor the runtime memory management behaviors in our applications without giving up the type safety and memory protections given to us by Java.

Java brought garbage collection to mainstream programming. Never before have commercial software developers been so aware of the need and benefit of using a collector. Notwithstanding, the benefits of garbage collection in Java are far from being completely realized. As larger and more complex applications are built in Java, it's becoming apparent that some very flexible memory management schemes are both needed and possible.

In this article I explore a cache you can build on JDK 1.2 beta 3 that uses WeakReferences to cooperate with Java's garbage collector. The cache uses WeakReferences to take advantage of available system memory without clogging it with unfreeable objects. When memory is reclaimed, the cache will free those objects that are not in use, resulting in a better-performing application. Objects can be cached in memory but don't have to trigger memory thrashing.

References and WeakReferences

Before talking about the cache in detail, we need to explore JDK 1.2's Reference objects. Introduced in the java.lang.ref package, the Reference class stands as the fulcrum for a body of memory management classes in JDK 1.2. The intention behind these classes is to allow an application developer to interact with the memory management policies of Java in a type-safe and extensible way.

The basic idea is to put data inside a java.lang.ref.Reference class (or subclass) instead of referring to the data directly through a variable. The data can be retrieved as needed from the Reference class, but the application doesn't keep a permanent finger on the data. Under certain conditions, if the garbage collector determines that only the Reference object is currently using the data, it may free the data. This can happen even if the application is still using the Reference object that wraps the data.

To illustrate, consider the following code:

```
Reference myReference = new WeakReference( new String("some string") );
```

In this example the application has created a WeakReference, a subclass of the base Reference class. Inside the WeakReference is stored the string "some string." The application refers to the WeakReference through the variable myReference, but has no direct access to the string.

Whenever the string is needed, the application calls the get() method on the Reference object as follows:

```
String myString = (String) myReference.get(); // returns "some string"
```

The application can then use the string. For References to do their job, however, the application needs to release its lock on the string object by either exiting the scope where "myString" exists or explicitly setting the variable to null:

```
myString = null;
```

When memory gets tight, the garbage collector may determine that the only reference to the string "some string" is through the Reference object "myReference". When this occurs, the string may be freed even though the Reference object that has stored the string is still in use. This is in stark contrast to the way memory management happens for any other class in Java, where transitive references are sufficient to keep an object from being freed. Reference objects are special, and are specifically handled by the garbage collector.

The garbage collector doesn't free a Reference's interior object directly, but instead invokes the clear() method on the Reference. Invoking this method is the signal to the Reference that its interior data will be freed. Once cleared, the Reference object will return null from the get() method. An application can detect that the interior data has been freed as seen here:

```
String myString = (String)myReference.get();
 if ( myString == null )
 {
  // do something since
  // the interior data was freed
 }
```

It's worth noting some awkwardness in the Reference terminology. The Java language has a language construct, called a *reference*, that should not be confused with the Reference class. The language construct is the way a variable "refers" to its data, while the class is a first-order entity in the system. The Reference class is used to "wrap up" and manipulate the concept of a language reference, a process known as *reification*. The interior data managed by the Reference class is called the *referent* of the class, meaning the thing to which the Reference class refers.

The specific conditions under which a Reference class is cleared vary from one subclass of Reference to another. Some subclasses are silently cleared while others are not, allowing an application to take action. The basic idea, however, stays the same.

References and ReferenceQueues

Another aspect of the Reference class hierarchy is the use of ReferenceQueues for monitoring state transitions. By registering a Reference with a ReferenceQueue, an application indicates that the Reference should be put into the queue when a significant state transition occurs. The application then pulls the References out of the queue. What constitutes a "significant state transition" varies from one subclass of Reference to another, but is commonly defined as clearing the Reference. That is, after the Reference is cleared, it is put into the queue.

The ReferenceQueue itself is monitored in two different ways. An application can poll for items in the queue or can block waiting for something to enter the queue. The latter is particularly useful in multithreaded applications when there are auxiliary system resources associated with Reference objects. An application dedicates a thread to monitoring the ReferenceQueue; when an object is placed into the queue, it frees up whatever resources are associated with the reference.

Caching

Before designing our Reference-based cache, let's put down a few requirements. In a typical cache, arbitrary objects are stored, identified by a unique key value and retrieved later. Each key/value pair is a cache entry. The entries are kept in memory to improve performance, but there is no requirement that any given object be in the cache.

To minimize memory consumption, a simple cache will limit the number of entries it can hold. A sophisticated cache will take advantage of available memory by using a variable number of cache entries. When memory gets tight, items are not in use or memory is being reclaimed, the sophisticated cache releases some (or all) of its cache items. In this way the cache can use available memory without creating sandbars that the memory manager has to work around. Our cache will release items when they are not in use and the garbage collector is reclaiming memory.

We will base our cache on JDK 1.2's collection classes. Our cache will implement the java.util.Map interface. which offers a simple put()/get() interface. An object is added via put() and retrieved via get(). We don't want the user of the cache to see any use of Reference objects, so we define the put/get methods to accept the cached objects directly.

Listing 1 shows the skeletal implementation of the cache. This simple implementation will store and retrieve cached objects under an applications control. As you can see from the listing, the class is a trivial subclass of Hashtable.

This simple cache does not cooperate with the memory manager. All cached objects are kept in memory until the cache is explicitly cleared. Worse yet, the cache grows in size with each new item put in it. There is no size limit. To correct these deficiencies, we introduce WeakReferences into our cache implementation. When an application stores an item in the cache, we won't store it directly. Instead, we put it inside a WeakReference and store that instead. When the garbage collector runs, it is free to collect the cached data if it is not in use.

Listing 2 shows the new implementation. As you can see, introducing References into the design has changed the implementation substantially. The class is no longer a subclass of Hashtable, keeps a private copy of all cached data, and does some Reference manipulation. These changes are necessary because of our requirement that users of the cache be shielded from the use of Reference objects.

To ensure that no Reference objects surface outside the cache, we have to guarantee that all of the access points into and out of the cache are protected. Objects passed into the cache are immediately wrapped by a WeakReference. The WeakReference is stored, but is always stripped off before an object is handed out of the cache. To accomplish this, the new cache implementation makes four core changes:

1. The Cache class is changed to inherit from AbstractMap. This class allows us to supply the Map data on demand, stripping off WeakReference wrappers as we go.
2. The put() method is adapted to wrap the incoming object with a WeakReference. Since the put() method is supposed to return the previous item that corresponds to the cache key, some clerical work is done to strip off any old WeakReference layers.
3. Items put into the cache are stored in a private Hashtable. This Hashtable contains WeakReferences indexed by the cache keys passed to the Cache.put() method.
4. The cache implements the AbstractMap .entries() method. This is the method by which the Map data is supplied on demand, and where the cache removes the WeakReference layer. When the layer is removed, an additional check is done to determine whether the WeakReference has been cleared. If it has, the WeakReference is removed from the internal Hashtable and is not passed back as a result of the entries() method.

> *"Careful crafting of caching algorithms can tailor runtime memory management behaviors without giving up saftey and protection of Java"*

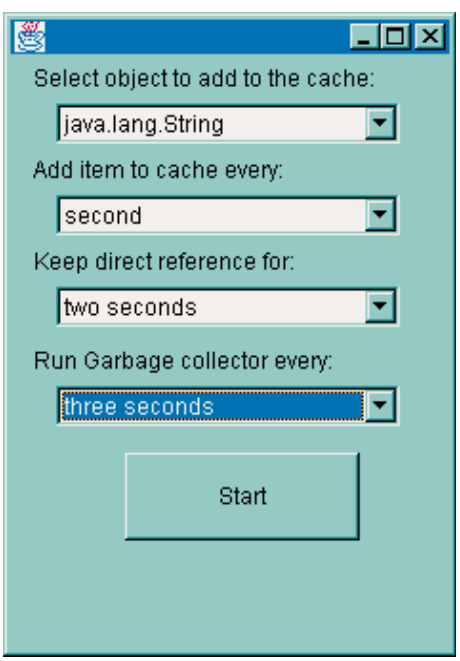


Figure 1: The CacheTest main window

That's all we need for the basics. The AbstractMap class drives all other operations from the values returned by the entries() method. With that in place, consumers of the cache see a simple put/get interface and can call the other abstract Map methods such as containsKey and containsValue. Additionally, the cache can be enumerated over, compared for equality with other caches, searched, etc. The only operations not supported by the cache are the collection-based deletion operations. For clarity, I've left that code out of these examples, but implementing them is a small extension; we simply change the collection returned from the entries() call to forward its modifications to the private Hashtable of the cache. All in all, we inherit a pretty complete system from the base collection classes.

Race Conditions

As you examine the entries() code, you may question whether the cache accounts for potential race conditions with the garbage collector. After all, if the garbage collector is running at the instant the entries() method is trying to determine which items are still available, isn't that an error? The answer is no. To understand why, you should first know that Reference objects are cleared automically. That is, if the Reference.get() method returns a non-null value that the application immediately uses, the garbage collector will not clear out the object. Put another way, the garbage collector performs the test and clear operations on the Reference as an indivisible, uninterruptible operation.

In our cache, once the entries() code has determined that a given Reference contains a non-null referent, a direct Java reference to that interior object is maintained and stored. This prevents the garbage collector from freeing the interior object.

To clarify this just a little, consider the following code:

```
if ( myReference.get() != null )
  myHashtable.put( "some key",
myReference.get() );
```

This example has a bug. The problem is that the garbage collector may run between the first call to myReference.get() and the second. It's possible that the Reference is cleared in the process. The bug won't happen often, but when it does, it will be hard to find. The example can be corrected as follows:

```
Object o = myReference.get();
if ( o != null )
 myHashtable.put( "some key", o );
```

In this example the object managed by myReference is directly referred to. This prevents any race conditions from freeing the data before it is put into myHashtable.

To Queue or Not to Queue

The attentive reader may wonder why the WeakReferences are explicitly tested for null rather than using a ReferenceQueue for the same purpose. The answer lies in the way Reference objects are queued. The only guarantee provided by the ReferenceQueue for WeakReferences is that each WeakReference instance that has been cleared will eventually be put in the queue. There is no guarantee that the object will be put into the queue at the time it was cleared, or that it will occur in a timely fashion. Since we want to ensure that the user of our cache never sees a cache item suddenly become null, we can't rely on a ReferenceQueue to remove our cache items fast enough.

Testing the Cache

To test the cache, we want to populate the cache with objects, simulate the irregular use of items in the cache, occasionally run the garbage collector and see if the right things remain cached. To achieve this, I wrote the CacheTest program.

This program is implemented by spinning off threads for each of several tasks. A thread that periodically adds items to the cache is started. As each item is added to the cache, another thread keeps a direct reference to the item for a brief time. Another thread periodically runs the garbage collector and prints out whatever is still left in the cache. The implementation of these tasks is found in the class files CacheItemGenerator.java, TimedReference.java and TimedGarbageCollector.java.

The user interface for CacheTest is shown in Figure 1. The main screen presents several options. The first pulldown lets you select what type of item should be added to the cache. The second is used to determine how often a new item is added to the cache. The third determines how long a direct reference to the cached item is maintained, and the last pulldown determines how often the garbage collector is run. Try selecting different combinations.

After choosing a set of parameters, press the start button and the machine will be set in motion. After each run of the garbage collector, CacheTest dumps the contents of the cache to standard output. You can identify which items are still in the cache by their names. Cache items are named sequentially, "Key 1, Key 2," etc. Listing 3 shows a sample run where all timeouts are set to one second.

The CacheTest threads can be halted at any time by pressing the stop button. This allows you to reconfigure the cache parameters and run another test.

You should adapt the test program to try caching your own classes. This is easy to do by changing the options in the first pulldown menu. When CacheTest is running, it uses the default class loader to load whatever class is identified in the pulldown. The only requirement is that the class has a default constructor.

Other Kinds of References

The cache presented here could be adapted in several ways. In this implementation I've used WeakReferences for the

cache items. This has the advantage of freeing items in the cache when they are not in use and when memory is being reclaimed. For many applications this is desirable. For others, however, items in the cache should be freed only if system memory is running low. In those environments it is inappropriate to free items in the cache merely because they don't happen to be in use.

To address that need, the cache can switch from using WeakReferences to SoftReferences. SoftReferences are cleared by the garbage collector only when their interior objects are not in use and when memory is running low. Additionally, the SoftReferences are subject to a least recently used algorithm. This meets the above objective. With beta 3 of JDK 1.2, however, I've had some variableness and problems with SoftReferences. These could be my own bugs, but because SoftReferences are debuggable only in low-memory situations, I haven't bothered to track down the problem. Caveat emptor.

If you have very particular cache needs, you may also want to investigate using a GuardedReference. A GuardedReference is not cleared by the garbage collector. Instead, the GuardedReferences are put into a ReferenceQueue when the garbage collector sees that the interior object is not in use. It's up to the application developer to pull the objects from the queue and clear them.

Summary

Once you take a garbage collection facility as a given in your programming, you can begin to consider new types of memory management that weren't open to you before. The Reference class hierarchy introduced in JDK 1.2 offers the facilities for building these new memory management structures. In particular, by careful crafting of caching algorithms, we can begin to tailor the runtime memory management behaviors in our applications without giving up the type safety and memory protections given to us by Java. With that in hand, we can begin to build better commercial applications than we have ever built before.

CODE LISTING

The complete code listing for this article can be located at **www.JavaDevelopersJournal.com**

About the Author

Lynn Monson is a software Architect at Novell, Inc. He has some eclectic interests, including distributed collaboration, machine learning and pattern based,object oriented, and Internet architectures. Lynn can be reached at lmonson@computer.org.

lmonson@computer.org.

Listing 1.

```
import java.lang.ref.*;
import java.util.*;

public class SimpleCache extends Hashtable
{
}
```

Listing 2.

```
import java.lang.ref.*;
import java.util.*;


public class Cache extends AbstractMap
{
 private Map map = new Hashtable();
 public synchronized Set entries()
 {
  Map    newMap;
  Iterator iter;

  newMap = new Hashtable();
  iter = map.entries().iterator();
  while( iter.hasNext() )
  {
   Map.Entry me = (Map.Entry)iter.next();
   Reference ref =
    (Reference)me.getValue();
   Object o = ref.get();
   if ( o==null )
   {
    // Delete cleared reference
    iter.remove();
   }
   else
   {
    // Copy out interior object
    newMap.put( me.getKey(), o );
   }
  }

  // Return set of interior objects
  return newMap.entries();
 }


 public synchronized
  Object put( Object key, Object value )
 {
  Reference ref =
   new WeakReference( value );
  ref = (Reference)map.put( key, ref );
  if (ref!=null)
   return(ref.get());
  return null;
 }
}
```

Listing 3.

```
Garbage collecting
Added class java.lang.String
 Cache still holds key 0
Added class java.lang.String
Garbage collecting
Releasing direct reference
 Cache still holds key 1
 Cache still holds key 0
Releasing direct reference
Added class java.lang.String
Garbage collecting
 Cache still holds key 2
Releasing direct reference
```

Direct Applet to Applet Communication with RMI

Pass remote objects in remote calls to allow bidirectional communication between the applets and the server and among the applets themselves

by **Pascal Ledru**

It's widely known that an applet isn't allowed to create a network connection to a computer that's not the one from which the applet itself was loaded. This has led to the idea that two applets aren't allowed to communicate directly with each other unless they're located on the same host. This article provides a brief overview of Java's Remote Method Invocation (RMI), describes a small conferencing program and shows how direct applet-to-applet communication can be established with RMI.

RMI's Overview

Three approaches for programming distributed systems can be identified. The first one consists of using a low-level API such as the java.net package. The second one uses middleware such as OMG's CORBA. The third one consists of using a high level API such as RMI.

In a nutshell, the advantages of using RMI over other approaches are:

- **Transparency:** From the programmer's perspective, remote objects and methods are somewhat like local objects and methods; the only differences are that remote classes must implement the java.rmi.Remote interface and remote methods must throw the java.rmi.RemoteException exception.
- **Integration:** As RMI is a standard component of Java, it supports distributed programming in a seamless manner with the other aspects of the language.
- **Automatic object serialization:** Java provides the facility to transfer data not only among hosts, but also classes and objects.
- **Automatic class loading:** If an object is of a class that is not available on the given host, the class is dynamically loaded at runtime.
- **Distributed garbage collection:** As objects are moved among hosts, RMI keeps track of which remote objects are no longer referenced by clients and deletes them automatically.

The ability to pass objects among hosts makes it extremely easy to program applications in which some classes are both clients and servers. For example, a method of a remote class R1 can be called with a parameter being a remote class R2. The remote class R1 will then be able to call any of the methods of the remote class R2.

A Conferencing Program

In this section I'll walk you through the complete code of a small conferencing program and show how it takes advantage of the facilities provided by RMI to achieve direct applet-to-applet communication. The program consists of two basic remote objects (implementing the "java.rmi .Remote" interface):

- A server that accepts requests from clients who wish to participate in a discussion, and requests to identify which clients are already active
- An applet that participates both as a client of the server and as the entity notified when other applets are ready to communicate (thus the applet is also a server)

To explain briefly, each applet registers its reference in the server, which broadcasts these references back to all other applets. All applets now have a reference to all other applets, which enables them to communicate directly without having to access the server. (To prove this, the system allows an applet to shut down the server.) Each applet displays the list of other available applets, allows the user of the system to send messages to other users and displays incoming messages.

Defining the Remote Interfaces

The first step in writing an applet or an application using RMI is to define the remote interfaces to the remote objects in the application. A remote interface extends the interface java.rmi.Remote and declares all the methods that may be invoked remotely. The two remote interfaces of the conferencing program are the Talker and TalkerServer interfaces (see Listings 1 and 2). The Talker interface is used by the applet to receive notifications from the server and messages from other applets while the TalkerServer is used by the server to receive commands from the applet. The Talker interface is implemented by the TalkerImpl class and the TalkerServer interface is implemented by the TalkerServerImpl class.

Creating the Remote Server

The next step is to define the TalkerServerImpl class that extends java.rmi.UnicastRemoteObject. UnicastRemoteObject provides support for point-to-point object references using TCP streams. The TalkerServerImpl class (see Listing 3) has a constructor, a main method and an implementation for the methods declared in the TalkerServer interface. The main method starts the remote server process, installs a security manager, creates a registry on port 2006, creates a TalkerServer remote object and finally makes this object available via the registry. The implementation of all the methods is trivial. Register adds a client to a private vector and notifies every registered client that a new member has joined the group. UnRegister notifies every client that one of them has left the session and removes it from the vector. Lookup returns a copy of the vector, and shutdown terminates the server. Although these methods are remote, they differ from local methods only in throwing RemoteException.

Implementing the Talker Interface

TalkerImpl (see Listing 4) implements the Talker interface. The constructor saves references of the applet and the current user's name. The applet is used to update the user interface; the name is used by

other users of the system. Remote users call the getName method to identify the current user and call the recvMsg method to send him messages. The add and remove methods are called by the server to update the system's list of users.

Creating the User Interface

The user interface (see Listing 5) consists of the following components displayed in an applet (Figure 1):

- A TextField to enter the name of the user
- A List to show the names of remote users
- A TextArea to display the sent messages
- A TextArea to display the received message
- Three buttons to start participating in the system, stop participating and shut down the server, respectively

Although the code of the applet is longer than the code of the other classes, it is fairly straightforward. The init method initializes a connection with the server and contains the code necessary to lay down the components within the applet. An ActionListener is added to each button. StartListener initializes the list of persons already connected and registers the current user in the server; StopListener unregisters the user from the server. ShutdownListener allows the user to shut down the server to prove that once connections among users are established, the server is not needed anymore. A KeyListener is also added to the TextArea handling the messages to be sent. The initList, addTalker and removeTalker methods update the list of users. The addToMsg method displays an incoming message.

Installing the System

This system is based on an applet accessing a server and other applets with RMI. Therefore, it will work only with a browser supporting all of the features of JDK1.1. For this experiment I used HotJava running on several PCs while the server was running on a UNIX machine. The system is easy to configure; the only requirement is that the server must be located on the same host as the HTTP daemon. The HTML file is shown here:

```
<HTML>
<TITLE>Talk</TITLE>
<APPLET code="TalkerApplet.class" width=550
height=450>
</APPLET>
</HTML>
```

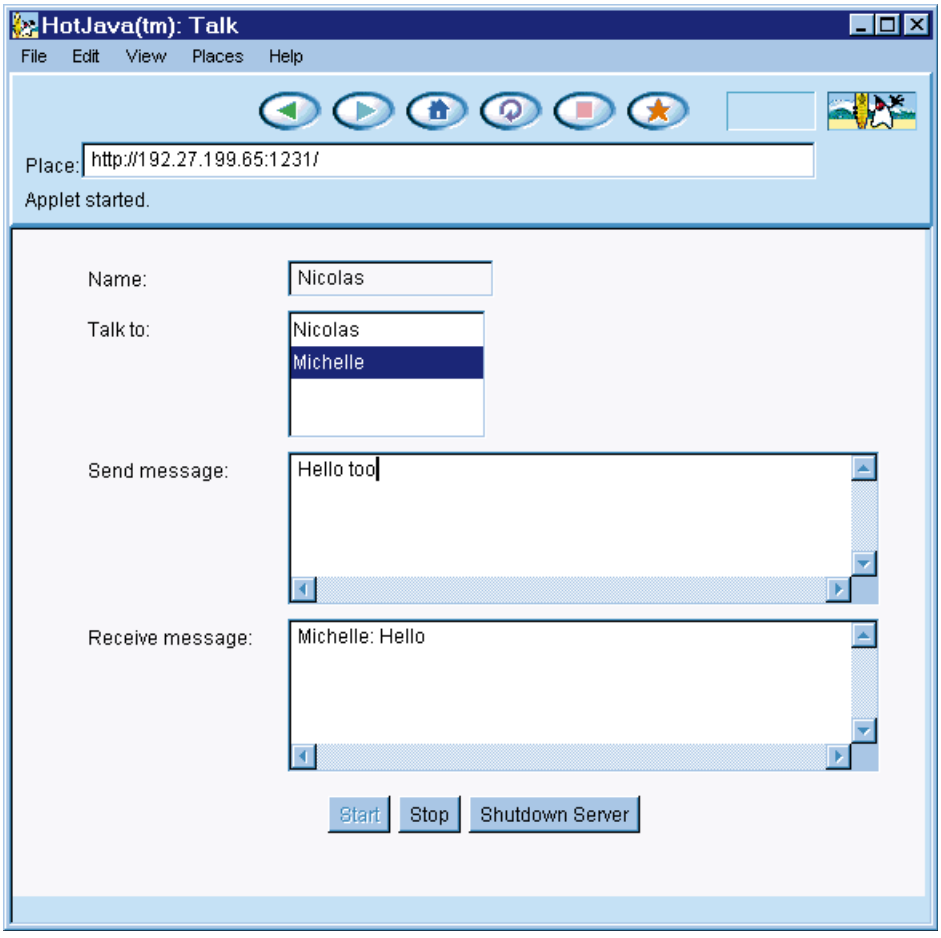


Figure 1

Conclusion

In this article, I have shown how to take advantage of the facilities provided by RMI to pass remote objects in remote calls to allow both bidirectional communication between the applets and the server and among the applets themselves. This short example also demonstrates that direct applet-to-applet communication is, in fact, possible.

About the Author

Pascal Ledru is a software engineer specializing in networking applications at Aerospatiale, Inc. He is also working on his Ph.D. in computer science at the University of Alabama in Huntsville. Pascal may be reached at pledru@worldnet.att.net.

pledru@worldnet.att.net

Listing 1.

```
import java.rmi.*;
public interface Talker extends Remote {
  public String getName() throws RemoteException;
  public void recvMsg(String from, String msg) throws RemoteException;
  public void add(Talker talker) throws RemoteException;
  public void remove(Talker talker) throws RemoteException;
}
```

Listing 2.

```
import java.rmi.*;
public interface TalkerServer extends Remote {
  public void register(Talker talker) throws RemoteException;
  public void unRegister(Talker talker) throws RemoteException;
  public Talker[] lookup() throws RemoteException;
  public void shutdown() throws RemoteException;
}
```

CODE LISTING

The complete code listing for this article can be located at **www.JavaDevelopersJournal.com**

SourceGuard
by 4thpass

Protect your code from reverse engineering!

by **David Reilly**

Software developers invest a great deal of time, effort and money to bring a product to market. Whether it's a complete Java application, an applet for a Web site or a JavaBean component that performs some cool new function, you as a developer have a right to protect the inner workings of your Java classes from prying eyes. After all, you don't want someone looking at your super-secret algorithm or trying to analyze your classes to find potential security holes. Many developers feel confident that when they compile Java source code to bytecode, their classes will be safe. After all, who can read bytecode like a third-generation language?

Unfortunately, other developers don't have to. Decompilers allow unscrupulous developers to reverse-engineer your product and to extract source code from it. They allow others to access your or your company's intellectual property, the code that is at the heart of your application. It's a sad fact of life, but reverse engineering does occur. While there are laws to protect against intellectual property theft, the best defense is not to allow it to happen in the first place, by protecting your code from such tools. That's where SourceGuard comes in!

How Does SourceGuard Work?

SourceGuard takes existing Java class files and modifies them to protect against decompilation. You create a project, define the level of protection on a class, method and field level, and then allow SourceGuard to modify your classes. SourceGuard is capable of renaming classes, methods and fields, as well as removing extra debugging information and modifying the control flow of bytecode.

When SourceGuard is run, either manually or from the command line, it produces new copies of your classes that can then be redistributed. SourceGuard doesn't modify your original source code, only the compiled classes. This means that you as a developer don't need to be concerned with writing cryptic code or changing your variable names. SourceGuard does it all for you, and then produces protected classes.

Installing SourceGuard

SourceGuard is easy to install, and requires a JDK 1.1.5+ virtual machine with JFC. After you've downloaded the software from 4thpass's Web site, you need to run the installation application. SourceGuard uses InstallShield, which makes installation simple. You'll need to specify a password to use the trial edition, which is sent to you via e-mail when you register. There are two installation options: you can use either a Windows executable or an installation class file. If you run the Windows executable, SourceGuard will ask you to select a Java Virtual Machine from a list it detects. Then you simply specify the installation path. Otherwise you can run the installation class file with whatever Java virtual machine you like.

SourceGuard
Professional Editiion
4thpass
810 32nd Avenue South
Seattle, WA 98144
Phone: 206-329-7460 Fax: 206-329-7480
Web: www.4thpass.com
E-mail: sales@4thpass.com
Requirements: Platform: JDK1.1.5 with JFC 1.0.1
Price: $599 download, $649 CD-ROM shipment

Using SourceGuard

Using SourceGuard to protect your applications and applets is quite simple. After compiling your Java source code, you can run SourceGuard. It's a straightforward process to apply SourceGuard to Java classes, thanks to the Project Wizard that guides you through the task of creating project files. It requires you to specify a directory location for your Java classes, an output directory and your classpath. It's important to specify the correct classpath for any

Original source code

```
// Calculate difference in dates long numericalDifference =
    m_Date.getTime() - currentDate.getTime();

// Divide by 1000 to find number of seconds difference
    numericalDifference = numericalDifference / 1000;

// Get seconds
int seconds = (int) numericalDifference % 60;

// Get minutes
numericalDifference = numericalDifference / 60;
int minutes = (int) numericalDifference % 60;
```

Decompilation with Mocha

```
i1 = B.getTime() - object.getTime();
i1 /= 1000;
j1 = (int)i1 % 60;
i1 /= 60;
k1 = (int)i1 % 60;
i1 /= 60;
```

Decompilation with Java

```
l = (this.B.getTime() - ((java.util.Date)(obj)).getTime());
l = (l / 1000);
i4 = (((int)l) % 60);
l = (l / 60);
i3 = (((int)l) % 60);
l = (l / 60);
```

Table 1: Original and decompiled source code, after SourceGuard protection

Figure 1: Specifying a Java classpath for external packages, and setting an output directory

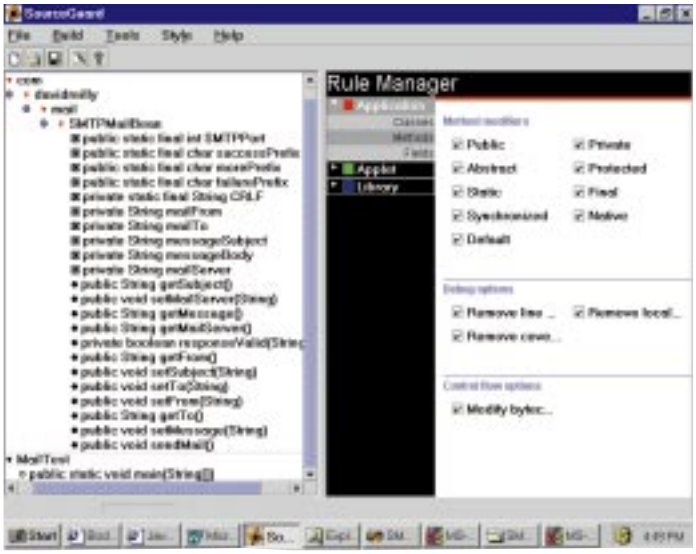


Figure 2: SourceGuard Class and Rule Manager allows you to customize the level of protection on a class, field and method basis.

external libraries you use; otherwise you can run into problems. This includes the basic JDK packages as well. Figure 1 shows the Project Wizard in action, and how to set the classpath and output directory.

Once your project is complete, you can begin to assign rules to your classes. SourceGuard is extremely flexible, and allows you to assign different rule categories. Each class can have a category applied to it, or a class can be left unprotected. SourceGuard defaults to three categories, but you can add more as required. For example, if you'd like to treat the class with your main method differently from your proprietary packages, you can assign minimal protection for one and maximum protection for another. From the Class View menu, which can be seen in Figure 2, allows you to specify which methods and fields should be protected, and which should be left alone.

Once you've created and customized your project settings, protecting your Java classes is a breeze. Simply select the build menu option, or click on the build icon, and SourceGuard will create protected classes and place them in your project's output directory. This process can even be automated as part of your build process. SourceGuard provides a batch file and a Unix script, which takes as a parameter the name of a SourceGuard project. Simply run SourceGuard from your build tool and it will automatically protect your projects.

Protection from Decompilers

SourceGuard offers protection from decompilers and other reverse-engineering tools that seek to recreate the source code for Java class files. Several decompilers for Java are freely available. The two decompilers I used for testing were Mocha and DejaVu (distributed as part of the Object Engineering Workbench for Java). After creating a protected application, an applet and a JavaBean, I used the popular Mocha decompiler, and the results were reassuring. SourceGuard had taken my carefully named variables and replaced them with meaningless names. SourceGuard can also do the same with method and class names. As you can see from Table 1, decompiled code can be difficult to understand. When compiled with Mocha or DejaVu, variables have been replaced with meaningless names.

Decompilation with DejaVu

SourceGuard can also give you added protection that goes beyond renaming variables, classes and methods. It has the option to modify the control flow of Java class files, using its Bytecode Range Modification feature. When a method contains try/catch blocks, SourceGuard modifies the bytecodes to help prevent decompilation. With bytecode range modification enabled, Mocha and DejaVu were unable to correctly decompile methods that contained try/catch statements. This feature isn't enabled for all rule categories by default, however. You should make sure it is turned on if this level of protection is appropriate for your project, because it will make decompilation much more difficult.

Conclusion

SourceGuard offers significant protection against decompilation of Java classes by others. While not impossible, it makes the task of understanding decompiled code much less likely. The success of protection will vary from project to project; in general, however, the code produced is much more difficult to interpret than without SourceGuard's protection. If you have a small applet that has few methods and fields, it may be possible for someone to make rough assumptions about how the applet works. However, more complex examples with long methods become quite difficult to understand. Without the original source in front of me, it would be an extremely frustrating and time-consuming process. Furthermore, for methods that contained try/catch blocks and were protected by bytecode range modification, Mocha and DejaVu failed to produce valid source code.

While it doesn't prevent people from trying to interpret your applications, it does make the task extremely difficult and tips the effort-to-reward ratio in the original developer's favor.

Running your source code through a decompiler makes you aware of just how vulnerable unprotected classes can be. I'd feel much more confident that my source code is safe when it's protected against reverse engineering and decompilation by SourceGuard. Given the choice between releasing unprotected classes and classes run through SourceGuard, I'd take SourceGuard any day!

Resources

1. *SourceGuard, from 4thpass*; www.4thpass.com
2. *Mocha the Decompiler*; www.brouhaha.com/~eric/computers/mocha.html
3. *DejaVu the Decompiler*, distributed as part of OEW; www.isg.de/OEW/Java/

About the Author

David Reilly has worked on network protocols and Web-related programming at Bond University, Australia. Since his conversion to Java in 1996, he has worked almost exclusively with the language, finding it both a joy to use and the most productive way to produce portable applications. David can be contacted by e-mail at java@davidreilly.com.

java@davidreilly.com

Web Products

Closely linked Java APIs that support Web and browser based application development

by **Ajit Sagar**

Our dreams of having the world at our fingertips have been realized in large measure by the advent of the World Wide Web and Web browsers. The Java Platform gained much of its popularity due to its inherently distributed nature and its implicit support for the Web. The Java-based products that are defined under the platform for facilitating Web-based development are a major factor in this support.

This month we will peer into The Cosmic Cup to look at the products from Sun Microsystems that support the Web- and browser-based application development. Please note that while a wide range of Web products and APIs are available from several vendors, this column currently focuses on those provided by JavaSoft/Sun Microsystems Inc. These are base products, closely linked with the APIs defined under the scope of the Java Platform, which is why I want to focus on them.

The Web Products

Web products that facilitate the development of Web-based applications are listed below and illustrated in Table 1. Subsequent paragraphs examine the products individually. Links for detailed information on all products mentioned may be obtained from Sun's Java Web site at http://java.sun.com/products.

- HotJava Browser
- HotJava HTML Component
- HotJava Views
- Java Plugin
- Personal WebAccess
- Java Web Server
- JavaServer Engine

HotJava Browser

HotJava Browser (HJB) is a modular browser that may be used for creating and deploying Web-enabled applications and several different environments and devices. It's a lightweight browsing solution that can scale to provide a solution for a variety of devices ranging from NCs to desktop PCs.

HJB allows developers to custom-build a browser with a highly customizable interface, depending on the application or device it targets. Customization is achieved through the use of text-based properties files. It provides flexibility in the look and feel for the UI for the application. The HotJava Browser supports various Internet standards and protocols including: Java applets, HTTP, HTML, tables and frames, persistent cookies, mulitmedia formats, file and mail transfer protocols, secure sockets layer (SSL) and internationalization.

HotJava Browser is a commercial product available from Sun Microsystems, Inc.

HotJava HTML Component

The HotJava HTML Component is a JavaBeans component that provides functionality for parsing and rendering HTML. It may be viewed as the HotJava Browser without the user interface. Instead, its interaction with other components is provided via the JavaBeans paradigm. It is a full-fledged browsing component that offers support for the same Internet standards and protocols as the HotJava Browser.

This product provides Web access to thin-client devices such as screen phones and NCs. The proprietary user interface for these devices is meant to be provided by the vendor. The HotJava HTML Component is also targeted to ISVs and corporate in-house developers who would like to incorporate HTML viewing into their applications. In addition, it may be used to plug and play with other JavaBeans components to create full-featured applications.

HotJava HTML Component is a commercial product available from Sun Microsystems.

HotJava Views

HotJava Views (HJV) software incorporates an environment and a set of tightly integrated corporate communication tools for e-mail, calendar management, name directory and Web browsing. It is used for enterprise deployment of a server-centric WebTop client using a graphical suite of deployment tools.

HJV software is centrally installed, configured and managed at the server, which significantly reduces client administration. It provides support for multiple platforms and supports internationalization and localization. The HJV software consists of the following components:

- **Selector/sliding panels:** Customizable GUI elements for accessing applications and information.
- **MailView:** A simple IMAP4/SMTP-compliant mail client that enables editing, sending and saving e-mail while providing extensive integration with other tools.
- **CalendarView:** A personal and group calendaring tool that provides calendaring and scheduling capabilities.
- **NameView:** A LDAP/JNDI-compliant tool for accessing corporate-name databases. It enables users to view an enterprise name directory that can be created from within HotJava Views or downloaded from existing enterprise directory databases.
- **WebView:** This is an HTML-compliant Web-browsing tool for the intranet. It may be used as a constrained browser that permits access to predesignated URLs only, or as a full-capability browser that enables access to any URL.
- **Administration tools:** Comprehensive tools (applets) for configuring and managing WebTop clients. These may be used to configure other HJV components as well as user profiles.

HotJava Views software is bundled with Sun's JavaStation and with network computers made by other manufacturers.

Java Plugin

Java Plugin (formerly known as Java Activator) runs Java applets or JavaBeans components in an HTML page using Sun's Java Virtual Machine (JVM) inside Microsoft Internet Explorer (IE) on Win32 platforms, or Netscape Navigator on Win32 and Solaris platforms.

Since Web browsers are usually a couple of steps behind the latest release of the JDK, they cannot use the features available in the latest Java Runtime Environment. For example, Netscape and IE browsers do not cur-

rently support the latest features of JDK 1.1 such as JavaBeans, RMI and JNI. Java Plugin ensures that enterprise developers can use all the features and functionality of the latest Java. JDK ensures that they can deploy 100% Pure Java applets on their intranet. The current Java Plugin is ready for JDK 1.2 and the Java HotSpot virtual machine with an architecture that makes upcoming features available today.

Java Plugin is a free product available from Sun Microsystems.

Personal WebAccess

Personal WebAccess is a customizable, compact Web browser for devices running the PersonalJava platform. It supports Internet standards including HTML 3.2, HTTP 1.1, tables, frames and Java applets. It has a small footprint that makes it suitable for consumer devices such as desktop screen phones or Web phones, set-top boxes, car navigation systems, high-end cell phones and other mobile hand-held devices.

Personal WebAccess is designed as a collection of JavaBeans components. It allows device manufacturers the flexibility to choose the level of functionality they need with choices ranging from a base HTML rendering engine to a full-fledged Web browser. Personal WebAccess will support SSL and I18n in its next release.

This product offers a user interface designed to accommodate the smaller displays of consumer devices. The interface is customizable, allowing device manufacturers the flexibility to create a fully tailored Web browser that fits the look and feel of their product line.

Personal WebAccess is a commercial product offered by Sun Microsystems and was developed in collaboration with Spyglass.

Java Web Server

The Java Web Server (JWS, formerly known as Jeeves) is a cross-platform Web server written in Java. It provides implicit support for the Java servlet API, SSL, digital signatures, Access Control Lists (ACLs) and proxy support. The salient features of JWS are:

- **Cross-platform functionality:** Since JWS is completely written in Java, it is truly cross-platform.
- **Servlet API:** JWS inherently provides support for the Java servlet API. Servlets are protocol-independent and platform-independent server-side components, written in Java, that are basically an alternative for CGI scripts.
- **Page compilation:** Page compilation allows server code to be embedded in an HTML file. As a result, changes can be made to the server with no recompilation of the code, thus allowing the option to customize Web content from the client side.
- **Session tracking:** Session tracking is a mechanism for building a sophisticated, stateful model on top of the Web's stateless protocol. With this feature, session state is maintained by the server.
- **Administration applet:** The administration applet is a set of GUI-based tools that provide a single point of control and facilitate installation management of a Web site. It is used to install the JWS.
- **Presentation templates:** Presentation templates are a content management feature that allow HTML content to be independent of the overall look and feel.
- **SSL support:** JWS provides standard SSL (secure socket layer) support for secure communication.
- **Secure area sandboxes:** JWS supports secure area sandboxes that allow Java servlets, like applets, to be isolated in predefined spaces so they are secure.
- **Digital signature:** JWS provides support for digital signatures that allow servlets to run securely outside the sandbox.
- **Access control list (ACL):** JWS supports ACLs, which are used to control access to specific Web site files or servlets.
- **Proxy support:** JWS supports proxy disk caching.

The JWS is available as a commercial product from Sun Microsystems.

JavaServer Engine

The JavaServer Engine is a collection of reusable Java classes that automate connection management, security and administration to simplify the development and deployment of network-enabled server-based applications. It is a product targeted at value-added resellers, system integrators and ISVs.

Developers can leverage the JavaServer Engine in their applications and are provided simplified connection management, data security and user authentication. The product also provides a Web-based server. The JavaServer Engine also provides applet-based administration and native support.

The JavaServer Engine provides a finer granularity of control over the HTTP protocol as compared to JWS, which is an out-of-the-box Web application solution. Application extensibility is accomplished through servlets.

The JavaServer Engine is a commercial product from Sun Microsystems; pricing and availability have not been announced yet.

Product	Description
HotJava Browser:	A lightweight Web browser that is targeted to developing Internet-aware and intranet-aware applications and devices
HotJava HTML Component:	A JavaBeans component that supplements the HotJava browser for displaying HTML; supports Internet standards such as HTML 3.2, frames and tables, HTTP
HotJava Views:	Enables enterprise deployment of a server-centric cross-platform WebTop client
Java Plugin:	Provides enterprise customers with the ability to specify use of Sun's implementation of the Java Runtime Environment instead of the browser's default JVM
Personal WebAccess:	Offers a compact Web browser for devices that run the PersonalJava platform; supports Internet standards including HTML 3.2, frames, tables, cookies
Java Web Server:	A Web server used for developing network servers in the Java programming language; includes extensive support for servlets
JavaServer Engine:	A collection of reusable Java classes that automate connection management, security and administration to simplify the development and deployment of network-enabled server-based applications

Table 1: The Web products

Cosmic Reflections

The Web and the Internet have been instrumental in defining the Java Platform APIs as well as in the development of products initially offered as Java-based solutions to industry problems. The very nature of the Java Platform makes it an ideal environment for development of solutions in the client/server world. It will be interesting to see how these products evolve from the client/server realm to enterprise solutions-based on distributed computing.

About the Author

Ajit Sagar is a member of the technical staff at i2 Technologies, in Dallas, Texas. He holds an MS in computer science. Ajit focuses on networking, UI and middleware architecture development. He's a Java certified programmer with 8 years' experience. You can e-mail him at Ajit_Sagar@i2.com.

Ajit_Sagar@i2.com

JDJ SPREAD

JDJ SPREAD

Emerging picoJava Processor Architecture

Drives new markets for networked appliances in the home, the car and in your hand

by **Harlan McGhan**

When Sun Microsystems introduced the Java programming language in May 1995, it handed developers the programmer's equivalent of the holy grail. Java's promise of write once, run anywhere proved real. For the first time, developers could write their source code just once, run it across multiple platforms without modification and generate bit-for-bit identical results. Now, less than three years later, Java technology has become more than just a wildly popular development environment; it has enabled the creation of a whole class of new computing devices called "thin-clients" that were not imaginable three years ago.

The Java Model

Following a close examination of the Java environment, Sun concluded that it could significantly enhance the Java paradigm by migrating the execution of bytecode instructions from software in the Java Virtual Machine to hardware in a Java processor. Even with this Java processor implementation, however, the Java Virtual Machine and its overhead remains. Indeed, it is required to execute Java code, because the Java Virtual Machine is a fairly complex piece of code that does far more than simply execute bytecode instructions. Its functions also include:

- **Verification:** Prior to passing the code along for execution, the Java Virtual Machine establishes that the program is legal and well behaved. This is an important part of the overall Java security model, making the transmission of computer viruses impractical in Java programs.
- **Class loader and garbage collection:** The Java programming language loads and purges code as necessary. These modules provide the runtime memory management operations necessary to keep the system running efficiently.
- **Thread Manager:** The Java Virtual Machine keeps track of the various specific concurrent strands of operation and sees that they execute without bumping into each other.

Java Without Execution Overhead

The Java Virtual Machine executes Java programs through a translation mechanism, using either an Interpreter or a just-in-time (JIT) compiler to convert bytecode instructions into machine instructions that the underlying CPU can understand and execute. This conversion process inevitably adds some overhead to the task of program execution, that is, it takes time (affecting performance) and consumes resources (affecting cost).

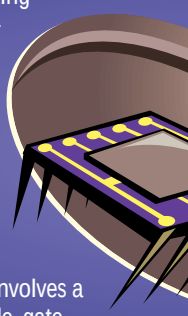
Fortunately, the burden of overhead is not fixed; it may be shifted depending on circumstances. If ample time is available for program execution (because, e.g., the application is I/O bound), a simple interpreter that consumes little in the way of machine resources can be used to translate bytecode programs. If ample hardware resources are available (e.g., in a power desktop or server system), an advanced JIT compiler that enables high-performance execution can be used.

The bottom line for program translation, however, is that it involves overhead in some form or other, either time (if resources are not available) or resources (if time is not available). The difficulty comes when both are in short supply, as is often the case in embedded applications. Under these circumstances developers generally regard Java programming as impractical, however desirable the principle might be

Smart Industrial Sensors Using the JavaChip

Embedded computer systems, common in manufacturing and factory automation, are used to control or monitor parts of a manufacturing process. They vary from large systems that control a complete machine to small systems, often single-board computers, that may monitor a single process sensor such as an oven temperature, tank level or chamber pressure. While not as immediately obvious or as sexy as the smart appliances or set-top boxes, industry estimates put the total market for factory automation equipment at over $3 billion by the year 2000.

Factory automation sensors are becoming smarter. They're better at passing real-time and near real-time information and using networks. This is important since, unlike traditional business information, much of the plant floor data is transient and the many decisions involved in controlling equipment are made rapidly, based on fast-changing information. However, while many sophisticated manufacturing facilities use an Ethernet network with TCP/IP on the plant floor to connect vital information sources, most of the sensors and small pieces of equipment are left out of the loop. Instead, a number of industrial networks, such as DeviceNet, ProfiBus and FieldBus, have been developed to handle the data communication for these sensor systems. The success of these industrial networks can be attributed in part to the lower hardware cost and simpler networking software. Usually these systems hide the networking complexity from the software developer, who may then concentrate on the problem of building the sensor interface and the logic to support it (rather than networking and information transport).



While information can be passed from these separate industrial networks to an Ethernet system on the plant floor, this process involves a router or, more commonly, a crude gateway. This creates data disconnects between the two networks. Integration on the plant floor would be improved by having these sensors and systems on the same networking architecture -- Ethernet and TCP/IP – which the JavaChip provides. Single-board computers based on the JavaChip bring the promise of almost transparent networking capabilities in a

for using the technology.

Java processors were developed to address precisely this problem. They are based on a new core CPU technology called *picoJava*, that reads and executes bytecode instructions directly in hardware. Consequently, Java programs execute on a Java processor without translation. In effect, for a Java processor, bytecode instructions are already machine instructions. Direct execution of bytecode instructions completely eliminates the requirement to sacrifice either time or machine resources. Java programs running on a Java processor execute as efficiently as possible, combining the high performance of quality JIT compilers with the negligible resource requirements of the simplest interpreters.

Java processors, therefore, enable a new breed of thin clients powered by Java that are more powerful and less expensive than would otherwise be possible. These devices are able to run Java code at machine speeds without using the expensive combination of a high-performance CPU, a sophisticated JIT compiler and an expansive memory subsystem. Eliminating these otherwise essential costs without compromising performance delivers the optimum in price/performance to the customer.

Legacy Code Compatability

Even with the wide acceptance and popularity of Java technology, Sun recognizes that Java is a new development environment, and that as a result even emerging, thin-client systems, such as network computers, may have to run some amount of existing code written in other languages. To make it easier for thin clients to handle legacy code, the picoJava technology at the core of Sun's JavaChip processor has been designed, not only with the capability to execute code written in any programming language, but also non-Java code as well as any comparable RISC processor. As a result, Java processors can extend the life of legacy applications, letting companies leverage their existing software resources while simultaneously easing their transition to Java.

Picking the Right Time to Transition to Java Processors

Given the ability of Java processors to execute Java and non-Java code, there is no reason to wait until all, or almost all, of the code base has migrated to the Java language before switching to a Java processor. You don't have to wait until a majority of the code base consists of Java programs. Rather, it makes sense to start thinking about using a JavaChip processor as soon as Java programs make up any noticeable fraction of the application workload -- especially if the you expect that more and more of the future workload will consist of Java programs and less and less code will be written in other languages. The non-Java programs will run as well as they would on any other processor, and the Java programs will run much better.

Java Processors: The Best Choice for the New Computing Environment

Java is enabling the creation of a whole host of new computing devices, including network computers and other thin clients such as set-top boxes, smart phones and home automation systems. Because all of these devices are meant to be high-volume, low-cost products, relatively small differences in price and performance can mean the difference between success and failure in the mass market. When unit volumes are measured in the millions, there is no such thing as a "minor" cost savings -- every penny saved is millions of pennies saved, every dollar saved adds up to millions of dollars. An MB of RAM can be purchased for as little as $4 today, and no doubt still less tomorrow. But while memory may be cheap and getting cheaper, it will never be free. Since JavaChips enable Java programs to run faster using less memory, a JavaChip can provide critical efficiency for high volume devices, returning many millions of dollars to the bottom line of those companies with the foresight to adopt them.

Java processors, then, are ideal to power an emerging generation of new, high-volume computing devices. Indeed, they have been designed with these types of devices in mind, delivering the kind of economy, power and compatibility critical to their success. If Java was the fuel that ignited the explosion of new computing devices, JavaChips will be the engines that drive them into the future.

The picoJava architecture has been licensed from Sun by IBM, LG Semicon, Fujitsu, Rockwell and NEC. Siemens Semiconductor has licensed the Instruction Set Architecture for use in Smart Cards. In addition, Sun has recently received an endorsement from Visa International regarding a Java specification for smartcard applications. In a roundtable discussion at the JavaOne conference, Fujitsu, Siemens and IBM all predicted that products would be available by the end of 1998.

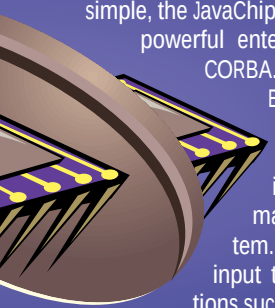
About the Author

Harlan McGhan is the group manager of architecture marketing for Sun Microelectronics, responsible for product roadmaps, product definitions, competitive analysis, and both SPARC and JavaChip processor evangelism. McGhan has a BA in philosophy from Michigan State University and an MA and ABD in logic and history of science from Princeton University. You can e-mail him at harlan.mcghan@Eng.sun.com.

harlan.mcghan@Eng.sun.com

relatively low-cost system. Java supports traditional network techniques such as network "sockets." Furthermore, the object-oriented architecture allows much simpler information transport, especially Java's Remote Method Invocation (RMI), which allows an object on the sensor to call a remote object in a manner almost transparent to the developer.

While these techniques are effective and simple, the JavaChip can also support the more powerful enterprise standards such as CORBA. With a CORBA ORB for EmbeddedJava or the JavaChip's MicroJava, these sensors can now interconnect to the entire manufacturing software system. This might include direct input to the front-office applications such as planning and scheduling. With this wide range of options designers using JavaChip-based systems can improve information management and integration. Using JavaChip-based single-board computers, even the smallest of sensors can be fully and smoothly integrated into the manufacturing network without changes in network topologies.



On both large and small embedded systems the cross-platform capabilities of Java simplify development and lower the total system cost. Complete applications can be developed using the productive Java development environments on UNIX and Windows, and subsequently moved in bytecode form to a JavaChip-based embedded system or any embedded system supporting Java. Java bytecode libraries may require little or no special porting to support embedded systems, further decreasing development cost.

A combination of ease of development, built-in standard TCP/IP networking and low cost for development and integration make JavaChip-based solutions attractive for integrating data sources into the information system on the plant floor.

About the Author

Jim Redman is the president of ErgoTech Systems, Inc., a company focused on developing Java applications and toolkits for plant-floor automation. This includes links to low-level systems and hardware, and also network links -- including CORBA support -- for enterprise distribution of factory automation information. He may be reached at JRedman@ergotech.com.

ProtoSpeed
by Progress Software

An essential client/server traffic monitoring tool

by **Jim Mathis**

ProtoSpeed 2.0
Progress Software
14 Oak Park
Bedford, MA 01730
Phone: 800 477-6473 ext. 470
Fax: 617 280-4095
Web: www.progress.com
Price: $495 introductory pricing (regular $995)

Progress Software recently released the second version of its ProtoSpeed distributed application debugging and monitoring tool for testing and deploying Internet or intranet applications. ProtoSpeed is actually a combination of three related but distinct components: a protocol interaction monitor/debugger, the JWatch remote Java applet debugger from Intermetrics and an environment for creating custom network monitoring applications.

Installation

ProtoSpeed 2.0 is available now for Windows 95 and Windows NT, with a version for Solaris in beta testing. I reviewed the Windows version on an NT 4.0 system. ProtoSpeed installs quickly from the supplied CD-ROM. The only problem I had with installation was the step in configuring the location of my Java Development Kit. If you have multiple Java Development Kits installed on your machine, you must configure the location of your preferred JDK (version 1.1.2 or newer) manually. In my case I had Symantec Café 1.8 and Symantec Visual Café 2.5 installed. Through some trial and error I discovered that I needed to specify the path to the VisualCafePDE/Java folder in order to use the JDK that was supplied with Visual Café. Once this problem was resolved, the rest of the installation proceeded smoothly.

Application Traffic Viewing

The unique aspect of ProtoSpeed, and perhaps its most common use, is monitoring and debugging the information transferred between a client and a server. Typically, client/server developers use LAN protocol analyzers that eavesdrop on the network to monitor client/server interaction. Rather than eavesdropping, ProtoSpeed uses a server proxy to monitor the traffic between the client and server. A proxy is a server that intercepts all Internet requests, passes each request to the intended Internet server, receives the response and, in turn, forwards the response to the original requester. With the rise of corporate intranets and the firewalls to isolate them from the general Internet, most client applications now include support for proxies. A different proxy is required for each application protocol. ProtoSpeed supports the most common Internet application protocols, including HTTP for Web servers, FTP for file transfer, and SMTP, POP3 and IMAP4 for e-mail systems. You can run several different proxies at once for different protocols or ports. For example, if you are testing an e-mail package, you can run both an SMTP proxy and a POP3 proxy at the same time.

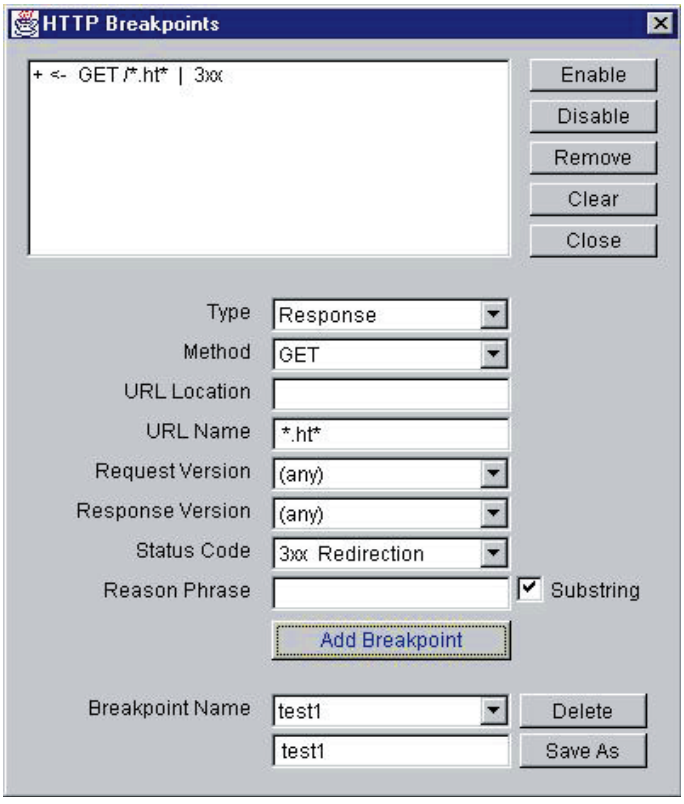


Figure 1: Setting an HTTP breakpoint

To use ProtoSpeed for Internet protocol debugging, you must configure your clients to use ProtoSpeed's server proxy. Complete instructions are included for configuring Netscape Communicator and Microsoft Internet Explorer browsers. You will need to consult your user manual for other clients, such as e-mail and FTP programs, to find the steps to configure them to use a proxy. Because the proxy receives and forwards each packet, you can set breakpoints that are triggered when a certain type of packet is received. When a breakpoint is triggered, the packet flow between client and server is suspended, and you can view or modify the packet contents before resuming operation. ProtoSpeed provides easy-to-use breakpoint dialog windows for each supported protocol. Using these dialogs you can select almost any possible packet condition. Figure 1 shows a breakpoint set for an HTTP interaction.

A big plus for ProtoSpeed over LAN analyzers and other protocol monitors is the extensive set of data content viewers for the complex data types often found in the protocol data streams. You can view ActiveX objects, Java class files, JAR and CAB archives, ZIP files, images and MIME attachments. Support for these most common data

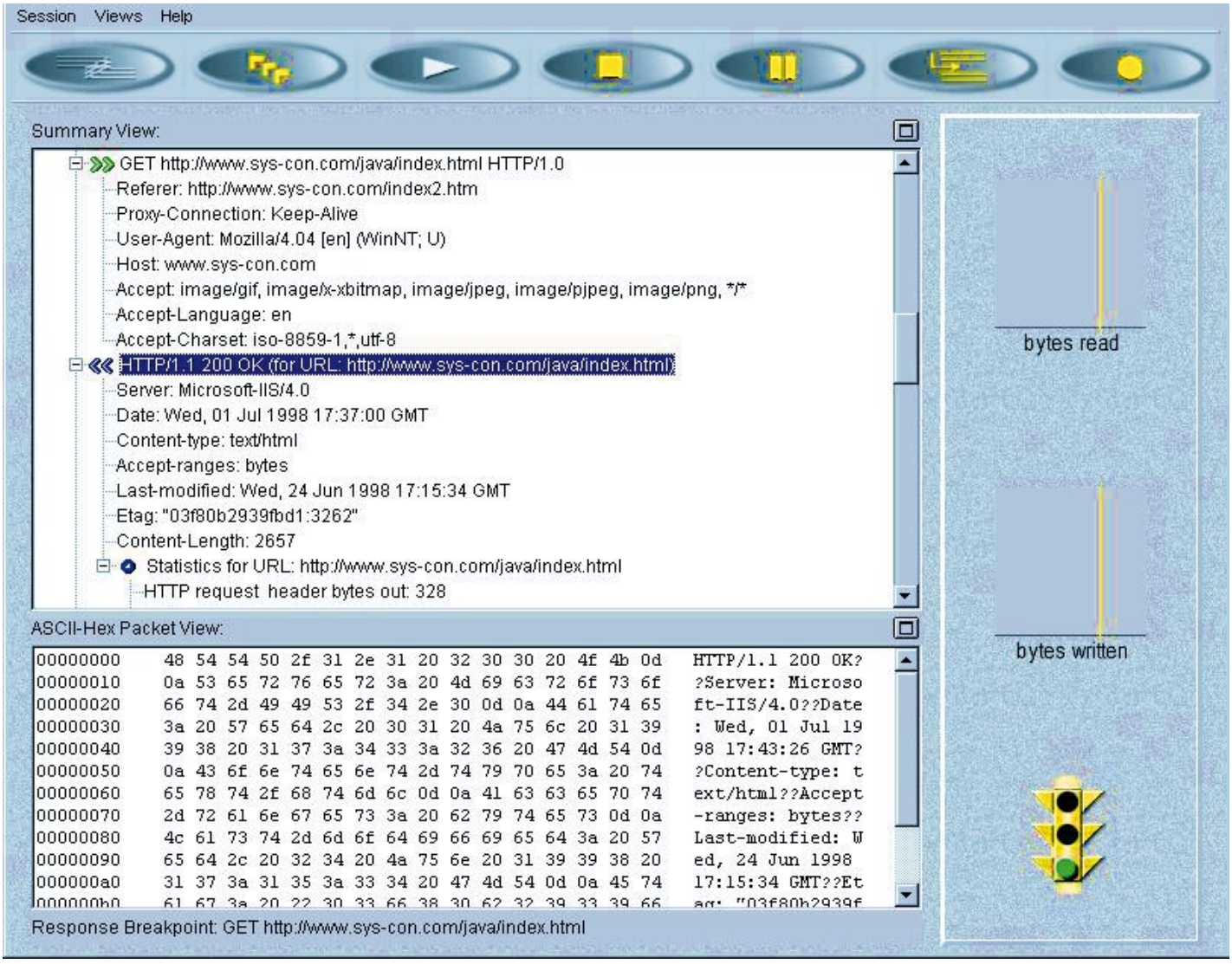


Figure 2: Packet trace for an HTTP transaction

types makes inspection of downloaded Java applets and ActiveX objects for potential security problems a breeze. You can also easily use ProtoSpeed to debug problems with cookies, understand the loading order of items on a Web page, check the format of posted data and monitor for rejected requests and other error conditions. Figure 2 shows an HTTP packet trace when accessing the ***Java Developer's Journal*** home page.

Remote Java Debugger

To go along with its Internet protocol debugger, Progress Software bundles the JWatch distributed Java debugger from Intermetrics into its ProtoSpeed package. With this debugger you can debug multiple applications or applets, local and remote, running in a browser, outside a browser or on a server. The distributed Java debugger provides a source-level debugger and also the ability to debug Java programs where you do not have the source, such as applets downloaded off the Web.

The debugging functions commonly available in an integrated development environment debugger are also available in ProtoSpeed's debugger. You can display thread groups, threads, stack frames, variables, packages, classes and methods. You can set breakpoints, single-step, run, pause and kill threads. If source code is available, it's displayed allowing line-by-line debugging. There can be multiple windows, each focusing on a different thread, or multiple windows for the same source file.

The remote Java debugger component of ProtoSpeed is compatible with Sun's JDK version 1.1.2 or newer, and can debug applets and applications running under either the Sun JVM (e.g., under AppletViewer) or Microsoft's JVM that is part of Internet Explorer.

Custom Network Event Monitoring

The most advanced feature of ProtoSpeed is its Network Event Manager. This manager lets you take advantage of ProtoSpeed's interception of network traffic to write your own packet monitoring code. As each packet is received and processed by ProtoSpeed, your handler is called to continue processing. These handlers can be running on the same machine as ProtoSpeed or on a remote machine. They can be written in Java or other languages such as Visual Basic. While quite powerful, this feature is probably beyond the needs of most developers.

Summary

ProtoSpeed is one of the best client/server traffic monitoring tools available. It is an essential tool for network developers, network managers or anyone else needing a clear understanding of exactly what information is being exchanged between their clients and servers.

About the Author

Jim Mathis is a freelance Java and JavaScript consultant by night and a communications system architect by day. He has been active in the Internet community from its very beginnings and wrote one of the first implementations of TCP/IP. A former Apple employee, Jim concentrates on Macintosh as a platform. You can write to him at jmathis@ais.net.

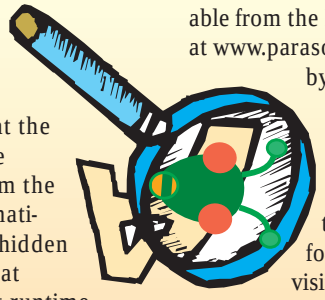
jmathis@ais.net

IBM STORY

IBM STORY

ParaSoft Announces jtest!™ 2.0

(Monrovia, CA) - ParaSoft Corporation has released jtest! 2.0, a completely automatic white-box testing tool for Java developers. With jtest! developers can test programs at the class or module level giving them the power to automatically find bugs hidden in their code that cause uncaught runtime exceptions to occur.



jtest! tests at the module level, which means more effective code produced by all manner of Java developers including API, library and JavaBean developers, servlet or server-side developers, and enterprise and large-scale developers.

Free evaluations are available from the ParaSoft Web site at www.parasoft.com/jtest or by calling 888 305-0041. Call the same number to purchase or for more information, e-mail fo@parasoft.com or visit their Web site at www.parasoft.com.

Rogue Wave Software Ships zApp Developer's Suite™ 3.1

(Boulder, CO) - Rogue Wave Software, Inc., has announced the release of zApp Developer's Suite 3.1, an enhanced version of the company's set of object-oriented visual tools for building portable, native GUIs.

zApp combines ease of visual programming with the power, flexibility and performance of C++. The product encapsulates and extends standard native windowing systems to provide a rich set of functionality and true native look, feel and performance. The 3.1 release adds Gauge and Spin controls, enhanced color handling for Win32 and a reference-counted font class. It also has new documentation and support for the latest operating system and compiler releases, including MSVC 5 and HP aCC.

Rogue Wave has also announced its Third Quarter earnings, ending June 30, 1998. Revenue for the quarter was $11.1 million, up 22 percent from the $9.2 million achieved in the third fiscal quarter of 1997. Their year-to-date earnings were up 41 percent from the prior year. Excluding $0.6 million nonrecurring reorganization expenses, net earnings for the quarter were $390,000 or $0.04 per share.

For more information, contact Rogue Wave's investor relations by e-mail at ir@roguewave.com, by fax at 303 443-7780, by phone at 800 758-5804, or visit their Web site at www.rwav.com.

SYS-CON Publications Announces *Cold Fusion Developer's Journal*

(Pearl River, NY) - SYS-CON Publications has announced the October launching of ***Cold Fusion Developer's Journal***, geared toward Allaire Corporation's new Web application platform.

Initial distribution will be 20,000 copies to newsstands, paid subscribers and other single-copy outlets.

For more information, contact Corey Low at clow@sys-con.com or Traci Massaro (Allaire Corporation) at tmassaro@allaire.com. SYS-CON's Web site address is www.sys-con.com.

Sun Certifies Multilizer Java Edition™ 1.1 as 100% Pure Java™

(Helsinki) - Innoview Data Technologies has announced its 100% Pure Java certified localization tool's Multilizer Java Edition 1.1 standards-compliant support for development environments that support the Java platform. Sun Microsystems' blessing lets potential customers and end users know that the product is portable across all Java-compatible systems.

For general information, visit http://usa.multilizer.com/main.htm. Free evaluations are available at http://usa.multilizer.com/download. You can also call Erik Lindberg, Multilizer's product manager at 358 9-476-20553 or e-mail him at erik.lindberg@multilizer.com.

MKS Releases Source Integrity™

(Waterloo, ON) - MKS has announced the release of MKS Source Integrity Professional Edition 3.1, which allows globally distributed software development teams to use the World Wide Web to collaborate on mission critical software projects. MKS Source Integrity lists for $1,299.

For more information, call MKS tollfree at 800 265-2797 or visit their Web site at www.mks.com.

NetDIVE Releases Client/Server System

(San Francisco, CA) - CallSite™, a system for connecting visitors of a company's Web site (via Web-based Paging buttons) to the members of that company in real time for instant communication in voice or text, also has the ability to share documents. The advantages of CallSite are instantaneous response, cross-platform capability and voice communication.

For more information, call NetDIVE at 415 474-3756, e-mail info@netdive.com or visit www.netdive.com.

BOOMA WebForms™ Beta Available

(Sandy, UT) - The public beta of BOOMASoft's BOOMA WebForms is available for download from its Web site at www.boomasoft.com/download.htm. The product is an easy-to-use and cost-effective solution for Web site developers to create and manage forms on their Web sites.

For more information, call BOOMASoft at 801 495-3200 or visit www.boomasoft.com.

TakeFive Software Introduces Source Code Engineering Tools

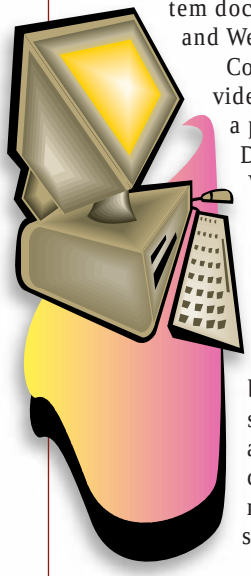
(Cupertino, CA) - TakeFive Software has announced availability of SNiFF+™ 3.0, a family of integrated source code comprehension, navigation, analysis and management tools. It is now available in two versions, SNiFF+ and SNiFF+ Cross. The tools enable programming teams to quickly understand and develop complex software systems, resulting in shortened development cycles and higher quality software.

The product is available for a single user for $1,750 per license or $2,195 for a multiple-user license, across multiple platforms it lists at $3,295.

For more information, call tollfree 800 418-2535 or visit their Web site at www.TakeFive.com.

Interface Technologies Releases CodeVizor™ 1.0

(Raleigh, NC) - Interface Technologies, Inc. (ITI) has announced the release of its new software development tool, CodeVizor. The tool allows software developers to quickly visualize and document the class structure of C++ and Java code by generating a color-keyed diagram of their project's class hierarchy. CodeVizor can view and print the class hierarchy, and export it as a graphics file. Class-level documentation can be produced by exporting the class hierarchy for individual classes as a graphics file that can be pasted into reports, system documentation and Web pages.



CodeVizor provides six views of a project: DerivationView, HierarchyView, PackageView, FileView, DiagramView and SourceView. The built-in color syntax editor allows changes to be made on the spot.

For more information, contact Ken Bagnal, director of marketing and sales, tollfree at 800 224-4965, ext. 116, or visit www.iftech.com.

Quadbase Systems Offers EspressChart™1.4

Quadbase Systems Announces EspressChart™1.4 (Santa Clara, CA) - Quadbase Systems Inc., a supplier of Web-based tools and open client/server DBMS technologies, has developed EspressChart 1.4, a tool that allows you to easily create and publish graphic charts on the Web. EspressChart consists of Chart Designer, Chart Viewer, Chart Server and a powerful API.

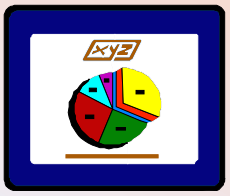
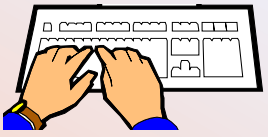


Chart Designer enables visual chart creation and editing using a browser enabled with Java technology. Chart Server performs file I/O and access to JDBC/ODBC data sources on the Web server. Chart Viewer allows end users to view charts with a Web browser. Web/database developers and/or end users can design a professional-looking 2D/3D chart with just a few keystrokes and mouse clicks. The resulting chart template can either be incorporated into applets/applications using Chart API or directly into an HTML page.



EspressChart runs on Windows 95/NT, Solaris, Unix and IBM platforms.

For more information, visit Quadbase's Web site at www.quadbase.com or contact Amy Schultheis, Quadbase's product marketing manager, at 408 982-0835.

Inprise Offers Advanced Java Training and Application Deployment

(Scotts Valley, CA) - Inprise Corp. and Referentia Systems has announced the availability of a next-generation, computer-based multimedia training tool for professional software developers interested in learning advanced Java application development techniques. Referentia for JBuilder is an easy-to-use, modular and integrated learning system for Borland JBuilder 2, Inprise's family of visual development tools for building corporate and enterprise software applications with Java.

Inprise also launched a new JBuilder Web site devoted to helping developers build, manage and deploy sophisticated Pure Java applications using JBuilder 2. The site is located at www.inprise.com/jbuilder/deployment/.

Inprise has also announced its stock buy-back program, authorized by its board of directors. The board's resolution authorizes Inprise to repurchase up to one million shares of stock for up to 10 million dollars.

Along with this announcement came Inprise's earnings for its second quarter, ending June 30, 1998. The company reported a 58 percent increase over the prior year in sales of VisiBroker middleware, aimed at the corporate enterprise computing market.

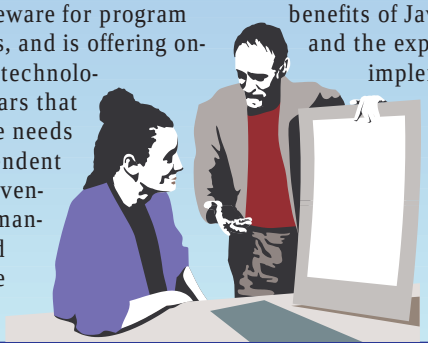
All figures reported reflected pooling of interests with Visigenic Software, Inc., acquired in February, 1998. In June, the company received shareholder approval to change its name from Borland International to Inprise Corporation to reflect its mission of "integrating the enterprise."

Net income for the quarter was $1.9 million, compared with a net loss of $2.5 million in the same quarter for 1997. Net loss for the six months was $11.6 million, which included $19.3 million in restructuring and acquisition costs from the purchase of Visigenic and a tax benefit of $3.8 million.

For more information, contact Larry Lieberman at Referentia at 808 396-3319, or by e-mail at larry@referentia.com. Contact Inprise's Bret Smith at 408 431-1341 or by e-mail at bsmith@inprise.com, or visit their Web site at www.inprise.com.

Sun Provides Java Education for Upper Management

(Palo Alto, CA) - Responding to an increasing need for high-level knowledge of the Java programming language, Sun Microsystems has developed a new self-paced Java technology courseware for program managers, and is offering on-site Java technology seminars that target the needs of independent software vendors, IT managers and CIOs. The courses were developed for managers who want quick access to the benefits of Java computing, and the expertise to implement Java technology in their environments. The classes are written in the Java programming language. These new services support Sun's "The Road to JavaSM" initiative, a road map for companies to build Java technology competence, and to reap the business benefits from Java computing and the Internet.



For more information, visit Sun's Web site at www.sun.com, or e-mail Santosh Ramdev at sramdev@upstart.com or call him at 510 420-7986.

Impedance Mismatch Hinders Development of Network Applications

THE GRIND

by **Java George**

"The output of each tool must match the input of the other tools"

Remember electrical science in high school? The teacher always harped on impedance matching, or in my case mismatching. I never became an electrical engineer, but I did remember one thing about impedance matching that translates very well to building network applications: the output of each tool must match the input of the other tools. If they don't match, you have a software impedance mismatch and you spend a lot of time and energy trying to make it match.

The problem of impedance mismatching becomes particularly acute when we try to build Java applications with lots of disparate tools and loosely connected pieces. A development team attempting to build a network application today typically assembles a wide assortment of tools and components: a Java IDE, the latest version of the JDK, some JDBC drivers for the current version of the target database, an object request broker for server-side logic and so on.

This is usually described as a best-of-breed approach. It sounds great. What organization doesn't want the best for itself, right? But it's not easy to do. No sooner do the developers try to assemble some pieces than they encounter the impedance mismatch because few – if any – of these best-of-breed pieces talk directly to each other.

And the problems only get worse. What happens when the new version of the IDE comes out and it no longer works with the older JDK you were using? You can move to the newer JDK, but will the JDBC driver support it? If you have to upgrade your JDBC driver, is there one available for the database you're using? Will the Object Request Broker you want to use be compatible with any of this right out of box? Will the new version of the JDK be supported in the end user's browser? There's only one way to find out – try, test and keep trying and testing for as long as it takes.

The best-of-breed approach and the inherent impedance mismatches that result create problems in four areas:

1. **Quality** – How will the team ensure quality without blowing both the budget and the delivery date?
2. **Performance –** Impedance mismatches in electrical science are highly inefficient. Ditto for software development.
3. **Openness –** This is a roll of the dice, completely dependent on the selected pieces.
4. **Productivity/maintainability –** Very difficult at best, due to all the hand coding required to smooth over the impedance mismatches.

Fortunately, there's an alternative: an integrated environment supplied by a single vendor that does all the impedance matching for you. Here the only difficult decision is choosing an integrated development environment vendor in the first place. After that, most of the necessary pieces come prewired to do what you need. In terms of quality, performance, openness and productivity/maintainability, most of the burden shifts to the integrated environment vendor. Your main worry is whether the vendor will be capable of moving this solution forward and keeping up with the requirements.

While it's true that the Java platform deals with these kinds of issues better than other platforms, impedance mismatch can still be daunting for a development team in the throes of a project. Use of an integrated development environment, however, decreases the impact of impedance mismatch. Just upgrade the integrated environment and in one move you've upgraded most or all of the pieces.

Today's early adopters naturally want to hand-sew the solution. The notion of an open platform understandably fuels the desire to join different, relatively compatible products. Having done this over and over again and achieved miserable levels of productivity and maintenance, however, I suggest jumping to an integrated solution. You'll gain substantial benefits in the process. After all, we're trying to build applications: the business logic, the UI and the events and workflow. We don't win extra points for doing it the hard way.

Java George is George Kassabgi, director of developer relations for Progress Software's Apptivity Product Unit. You can e-mail George at george@apptivity.com.

George@sys-con.com