SPECIAL FOCUS: VRML & JAVA

U.S. $4.99 (Canada $6.99)

JAVA™ DEVELOPER'S JOURNAL

JavaDevelopersJournal.com

Volume: 3 Issue: 6

Make My PC JavaReady
by Rick Ross pg. 5

CORBACorner
OODBMS & CORBA
by David Knox pg. 56

The Grind
Avoiding the Web Application Front End Trap
by Java George pg. 66

Product Reviews
ProtoView JSuite
by David Jung pg. 43

Under the Sun
Java Multithreading
by David Nelson-Gal & Devang Shah pg. 40

Java News
RogueWave Ships StudioJ
.....
KAI Announces Debugging Tool
pg. 62

JDJ Focus Feature: Extending VRML 2 with Java — *Bruce Campbell*
When and how to use EAI in creating VRML 2 Worlds — 8

Integrating JavaScript into VRML: Glow Shapes Glow — *Guy Huggins*
An introduction to VRML programming — 18

How to Create and Use a Java Wizard Class — *Donald Fowler*
The preferred UI method for guiding users — 28

Adding a Middle Tier to Your Java Code *part 2* — *Sean Rhody*
Server Side Coding: *Building a simple Java component* — 34

Java Web Server and Dynamic Page Compilation — *Robert Tiffany*
JavaServer page technology introduces productivity gains — 46

Java Techniques: How to Make Flexible Threads — *Philip Rousselle & Mike McNally*
A robust approach to thread management — 50

Widget-izing Java's Graphic Interface Components — *Daniel Dee*
Technologies that will determine Java's role in computing — 54

Full Page Ad

Full Page Ad

Full Page Ad

EDITORIAL ADVISORY BOARD

Ted Coombs, Bill Dunlap, Allan Hess,
Arthur van Hoff, Brian Maso, Miko Matsumura,
Kim Polese, Richard Soley, David Spenhoff

Art Director: Jim Morgan
Executive Editor: Scott Davison
Managing Editor: Anita Hartzfeld
Associate Editor: M'Lou Pinkham
Editorial Assistant: Carolyn Emmett
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
Java Security Editor: Jay Heiser

WRITERS IN THIS ISSUE

Bruce Campbell, Daniel Dee, Donald Fowler, Guy Huggins, David Jung, George Kassabgi, David Knox, Mike McNally, David Nelson-Gal, Sean Rhody, Rick Ross, Philip Roussell, Devang Shah, Robert Tiffany, James Walker

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 Assistant: Erin O'Gorman
Advertising Intern: Jaclyn Redmond
Accounting: Ignacio Arellano
Senior Designer: Robin Groves
Designer: Alex Botero
Webmaster: Robert Diamond
Senior Web Designer: Corey Low
Customer Service: Rae Miranda
Sian O'Gorman
Paula Horowitz

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.

© COPYRIGHT

Copyright © 1997 by SYS-CON Publications, Inc. All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy or any information storage and retrieval system, without written permission. For promotional reprints, contact reprint coordinator. SYS-CON Publications, Inc. reserves the right to revise, republish and authorize its readers to use the articles submitted for publication.

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

Java and Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. SYS-CON Publications, Inc. is independent of Sun Microsystems, Inc.

GUEST EDITORIAL

Rick Ross

Make My PC JavaReady!

There's one form of power that is almost universally recognized in our society, the power of consumer spending. This is at the heart of all commerce, and anybody who tries to tell you otherwise must have something to sell you. I read today that more than 45 million American homes now have computers, and I'm sure that the number of computers used in businesses far exceeds that. I doubt that anyone will argue the fact that a lot of technology-driven consumer spending power is at work in our economy.

As a software developer you probably have significant influence over where some of this spending power is directed. The Java Lobby has started a new initiative in which you can exercise some of your influence to achieve a positive, proJava outcome. This initiative is called the "JavaReady PC Project," and you can learn all about it from the Java Lobby Web site at http://www.javalobby.org/javaready.

The "JavaReady PC Project" is a simple idea that does not depend on trust-busters, the courts or anything more complicated than your willingness to relate your support for Java to your own consumer spending decisions and to those that you influence. In short, you can help promote Java by favoring vendors that support Java. If computer resellers believe that supporting Java will give them an edge in their intensely competitive business, then they will definitely start supporting Java.

It's very easy for computer manufacturers to preload the hard drives of new computers with software that will help ensure a successful Java experience for their customers, and the required software is ABSOLUTELY FREE! If ever I heard of a win-win proposition, then this has got to be it. Savvy computer manufacturers install free Java software on the computers they sell, and that software makes those computers more attractive from your point of view as a consumer.

Now I know that some of you may be disappointed that this initiative does not involve any hot-headed accusations, name calling or other high-profile conflict of the sort that has become so common in the Java space. I apologize, and no doubt the mayhem will soon continue. This is an opportunity, however, for each and every one of us to make a difference by quietly leveraging the economic power that we control. It's not a matter of religion, nor is it a matter of which major company we might prefer to dominate the technology landscape. Instead, it is a simple and clear way to cast an economic vote for Java - a vote that will not be ignored.

So, if you want to support Java, then look for the JavaReady logo on the next PC you consider purchasing, and don't be afraid to speak up if you don't see this logo. Ask your preferred vendor to support Java by making their PC's JavaReady - and let them know that they may not remain your preferred vendor for long if they don't. It is so painlessly easy for vendors to make their PC's JavaReady that I can't think of any good reason why they wouldn't want to comply in order to win your business. The software is free, the Java Lobby has made it simple for them to get this free software, and they will save you unnecessary downloading time by pre-installing genuine, compatible Java before they ship your new system. Like many good ideas, this one is really simple - but it all depends on you.

Tell your friends, your family, your colleagues, tell everyone – but most important please tell your vendor "Make my new PC JavaReady!"

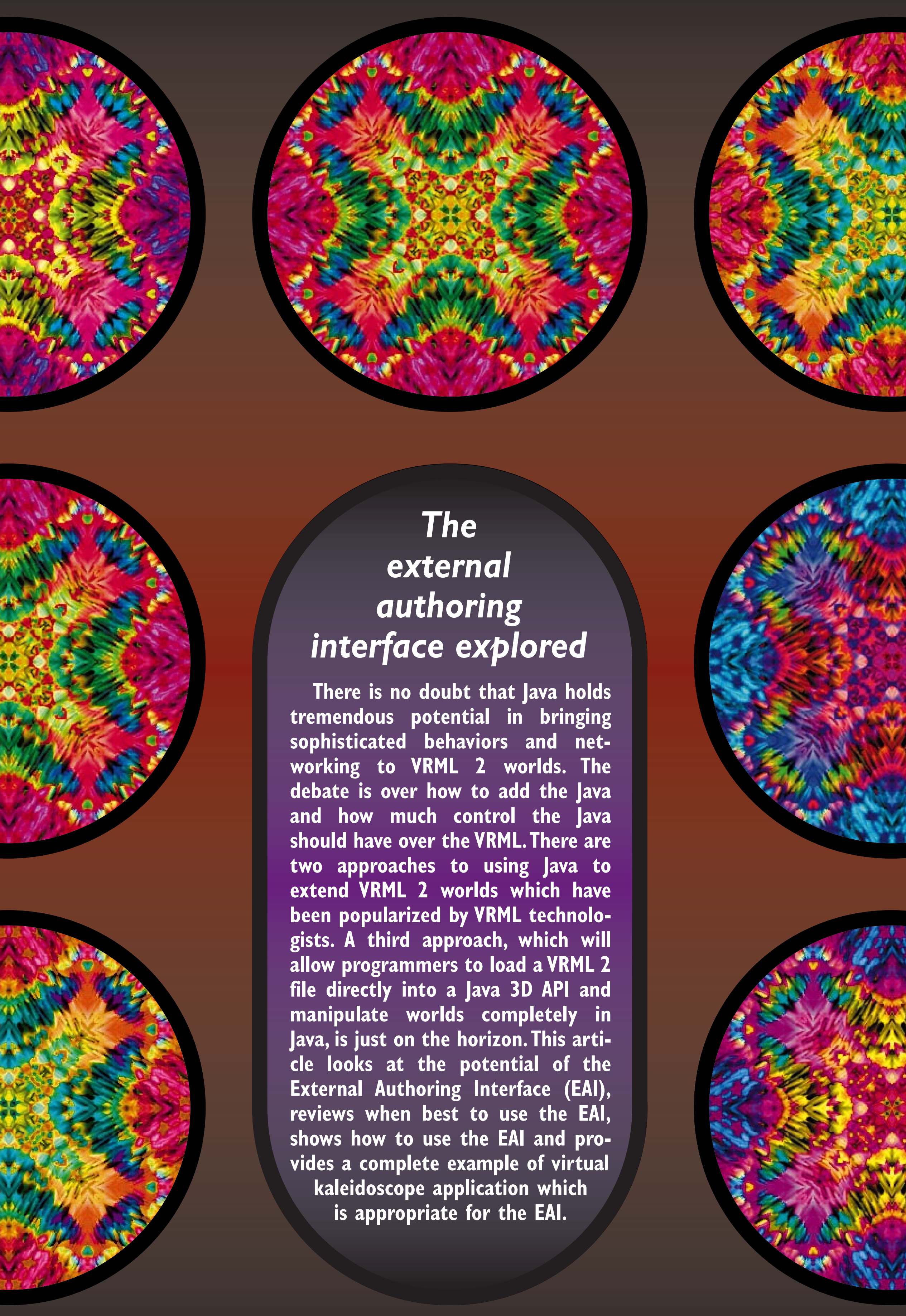
About the Author

Rick Ross is the President and founder of the Java Lobby (www.Javalobby.org), which currently has more than 17,700 members. He is also President of Activated Intelligence and can be reached at rick@activated.com.

Full Spr

read Ad

by Bruce Campbell



The external authoring interface explored

There is no doubt that Java holds tremendous potential in bringing sophisticated behaviors and networking to VRML 2 worlds. The debate is over how to add the Java and how much control the Java should have over the VRML. There are two approaches to using Java to extend VRML 2 worlds which have been popularized by VRML technologists. A third approach, which will allow programmers to load a VRML 2 file directly into a Java 3D API and manipulate worlds completely in Java, is just on the horizon. This article looks at the potential of the External Authoring Interface (EAI), reviews when best to use the EAI, shows how to use the EAI and provides a complete example of virtual kaleidoscope application which is appropriate for the EAI.

To me, VRML has two distinct purposes: To provide a pleasing architecture to 3D cyberspace and to provide educational and entertaining things to do within the confines of that architecture. Consider a virtual pool hall. VRML 1, as a standard, provides all the tools necessary to build an attractive and functional billiards room complete with marble floors, colorful walls, impressive artwork, cozy lighting and smoky haze.

But, visiting a virtual billiards parlor eventually loses its charm if there are no billiards to play. So, VRML 2 provides the context within which balls, racks, cues and chalk can be created and manipulated. Sensors, Timers, Interpolators, ROUTES and VRMLscript provide a primitive interface for making a pool hall respond to a virtual visitor's actions. But, as this article discusses, Java provides a more flexible yet more complicated tool for making a pool table come to life. Interpolators are great, but they must be pre-loaded with all of the possible behaviors that might take place in a world without the help of a programming language.

I enjoy creating visual 3D objects, embedding their physics and letting them interact without any additional human intervention. Consider 16 billiard balls behaving naturally in response to a single human-initiated event: a cue stick striking a cue ball. That's great stuff. Java makes such self-responding worlds possible. If you believe in the ability of Java to make your worlds come to life, you are halfway to making it happen. To get three-quarters of the way there, you have to choose how to add Java to your world. Then, of course, you have to learn how to program using Java.

I have given a lot of thought about how to add Java to my worlds. VRML gives us a new toy we have never had before: The ability to share visual 3D objects through Web servers located all around the planet. Although as planetary citizens we have barely taken advantage of this opportunity, it would be nice to continue to keep the dream alive and place Java code within the confines of each VRML object. The VRML 2 standards committee continues to pursue an approach to standardizing the way Java code self-contains itself within each 3D object in the VRML scene graph. Just as I can share VRML geometries with others on the Web using VRML with an intranode Java programming style, I can share behaviors with others as well since an object's behavior is self-contained in its Transform node. Imagine how quickly 3D cyberspace could come to life if each VRML author spent time studying and creating a different 3D object and made it available on the Web. With 500 or so interesting and naturally behaving 3D objects available, all VRML authors could put together sophisticated content in much shorter periods of time.

Intranode Java certainly shows long-term promise and should deliver the kind of 3D cyberspace many of us envision. Keep up with the latest work of the VRML Consortium to see intranode Java progress. This article, on the other hand, focuses on adding Java through an External Authoring Interface (EAI). I use the EAI to create my virtual pool halls, virtual solar systems and virtual board games. In this article I demonstrate using the EAI to create a virtual kaleidoscope. The virtual kaleidoscope project shows a good balance of using VRML 2 and Java, each for its intended purpose.

Before diving into an example, consider what makes the EAI different from intranode Java. The EAI was designed by Silicon Graphics (SGI) for use with their CosmoPlayer VRML 2 plugin viewer. You can read all about the EAI in the Developers' link at http://vrml.sgi.com. I think of the EAI as a presentation venue for a Java virtual machine. To me, although the EAI requires a .wrl file, the EAI is Java-centric. Intranode Java is VRML 2-centric. So, to the EAI VRML 2 extends Java. To intranode scripting within VRML 2, Java extends VRML 2. This is not a subtle differentiation. These differences affect the whole thought process I follow when I develop Web-based, 3D virtual worlds.

Consider what is required of your audience when you use the EAI. Each participant in your virtual world must download all the Java classes you use in your world. Yet, the classes they download can include network-aware classes that help them share 3D worlds with others. The classes they download can also include GUI controls created using the Java AWT. Using the EAI, you can provide additional controls beyond those available in their VRML viewer of choice. By focusing on Java, your world can quickly be updated to take advantage of any new Java classes or API made available by Sun or others on the Net.

But I'm sure you do not want to write your whole user interface using Java. Why not take advantage of the existing VRML viewers to give your audience the freedom to move about in your world in a manner they prefer? Using VRML 2 for your 3D presentation delivery is much better than any other available presentation tool on the net today. Yet, a 3D API from Sun shows much promise for the future. Check out http://java.sun.com/products/java-media/3D/forDevelopers/3Dguide/j3dTOC.doc.html to see Sun's 3D API specification.

One final note before I dig into the details of using the EAI. There is no reason you can't use both the EAI and intranode scripting in the same 3D virtual world. SGI builds the EAI with flexibility in mind.

Using the EAI

The process of using the EAI is best understood through example. Simplifying a bit, the overall process goes as follows:

1. Create the VRML 2 world of your dreams following the syntax of VRML 2.
2. Add additional empty Transform nodes wherever you might like to add new objects dynamically in response to your audience's actions.
3. Uniquely name the nodes you want to change dynamically with Java using the DEF keyword.
4. Extend Java's Applet class to create your own class which will be aware of your VRML 2 file.
5. Create an HTML document to connect your VRML 2 world from step 3 with your class in step 4.
6. Follow the architecture of the EAI to connect your class from step 4 to the browser object of the Web browser.
7. Follow the architecture of the EAI to provide attributes in your class in step 4 which map to the DEF nodes of step 3.
8. Add an additional attribute for each change you want to make to each attribute in step 7 (translation, rotation, scale, color, etc.).
9. Add an additional attribute for each attribute in step 7 you want to be mouse-click aware.
10. Create an event-handling routine for each attribute you specify in step 9.
11. Pre-package as text strings any new VRML nodes you may want to place into the world in response to user actions.
12. Create methods (or additional classes with methods) which use the attributes of your class from step 4 to dynamically change each attribute based on the state of the virtual machine.

Then, the Java Virtual Machine can run on its own as you rely on the EAI to pass events to your Applet-extended Java class. You just have to provide the interesting event-handling routines and embed the realistic physics in each object you want to behave on its own. Consider putting each of those objects into separate classes should you want to reuse them in other projects. As your project gets more complex, create an Animator class to manage the interactions between objects and a Timer class to

provide control over how fast things run. Long term, you can create a Java server and connect your world to others using Java networking classes. Java makes networking worlds relatively easy compared to other programming languages.

Using the EAI in a Virtual Kaleidoscope

Driving the EAI to its breaking point requires a tremendous programming effort. First things first. Here I provide you with an intermediate use of the EAI so you can see much of its design in use. My Kaleidoscope world works well on 486 or Pentium-based PCs and provides the user with an interesting array of tools to use to change the world. The Kaleidoscope world handles some behavior in the VMRL 2 file, yet extends user interactivity through Java. Basically, it is a good starting point for your study of the EAI.

The kaleidoscope, when first loaded in the CosmoPlayer VRML 2 viewer (downloadable from http://vrml.sgi.com), appears as shown in Figure 1. The kaleidoscope, in the upper left of Figure 1, consists of eight shapes chosen randomly from a shape palette. You design the eight shapes you want to use in the kaleidoscope and create them as standard VRML 2 Shape nodes. Each shape appears in a random color. The same eight shapes appear as a horizontal palette below the kaleidoscope. Below the eight shapes palette is a color palette. To the right appears a control panel with 16 buttons. The buttons provide tools a user can use to interact with the kaleidoscope. Figure 2 shows the significance of each control panel button.

A user interacts with the kaleidoscope by clicking on a shape in the kaleidoscope, redesigning the shape using the shape palette, color palette and control panel and then putting the new shape into the kaleidoscope in place of the old. The three white buttons from left to right allow a user to replace a shape, start the kaleidoscope animation and stop the kaleidoscope animation. The rest of the control panel, from top to bottom allows users to modify a shape they are designing by decreasing the intensity of color, increasing the intensity of color, increasing scale or rotating the shape. Figure 2 recaps the significance of the control panel buttons. Note that the left-most white button will choose a random new shape if a user clicks on it without designing his or her own new shape first.

You can build the kaleidoscope by following the steps outlined above. I will walk you through the basic steps here, but you should study the complete project code which includes the VRML 2 file (K.wrl), Java file (K.java) and HTML file (K.html). Not all lines of code are covered by this condensed explanation.

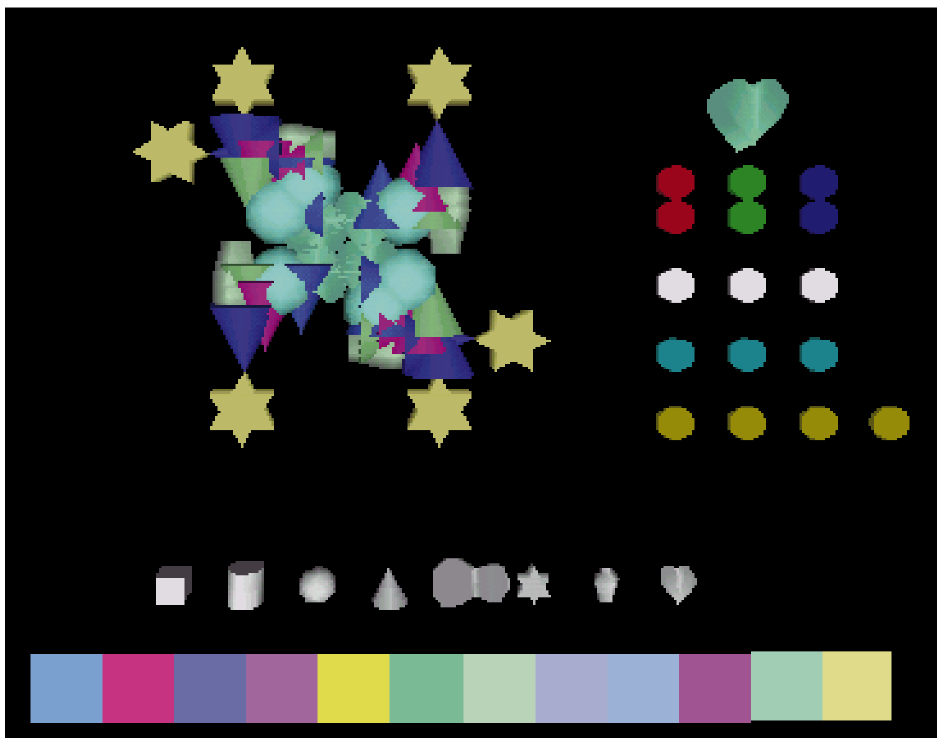


Figure 1: Kaleidoscope on first loading

1. Create the VRML 2 world of your dreams following the syntax of VRML 2.

Every visible object you see in Figure 1 exists as a DEF-defined Transform node in a single VRML 2 file. Note that each Shape node is a simple primitive VRML 2 shape except for the shapes you use in the kaleidoscope. Those eight shapes can be any shapes you want to design. I just happened to include a star, heart, guitar body and vase as examples. Each Transform node in the Kaleidoscope world contains a TouchSensor node in order to connect the shape to an event-handling routine that will affect the world upon a user's mouse click.

In my VRML 2 file, K.wrl, I use the following Transform naming conventions: The color controls have names that begin with TC, the white animation control buttons begin with T0, the scale controls start with TSCALE, the rotation controls start with TROT, the shape palette controls have a T followed by a shape name (such as TGUITAR) and the color palette items start with a CM.

Each kaleidoscope control is a Transform node similar to the following, which shows a color palette control's Transform node. Note that you can use the USE keyword to reuse geometry or appearance field nodes as I do to reuse the geometry for each color palette square:

```
DEF CM2 Transform {  children [
     DEF TSC2 TouchSensor {}
     Shape {…}
]  translation 2 0 0 },
```

The eight shapes in the kaleidoscope move according to simple OrientationInterpolator nodes and two of the eight nodes move according to PositionInterpolator nodes as well. You should put the movement in the VRML 2 file when it repeats over and over. You should put the kaleidoscope's behavior in the Java file instead if it lacks a highly repetitive motion. The kaleidoscope contains two separate grouping Transform nodes (named T1 and T2) which contain eight nodes each. Transform nodes T3, T4, T5 and T6 reuse T1 and T2 to simulate a mirror reflecting the images in the kaleidoscope. ROUTE statements connect the animation start and stop control buttons to the TimeSensor (named Time_T) and interpolators needed for the animation.

2. Add additional empty Transform nodes wherever you might like to add new objects dynamically in response to your audience's actions.

The Transform nodes that make up the kaleidoscope (their names start with a W) do not contain any shapes when the VRML 2 file initially loads in the Web browser, yet do contain TouchSensor nodes. Transform node W9 will contain the shape of the node the user is designing at any time. W9 also contains no shape when the world initially loads.

3. Uniquely name the nodes you want to change dynamically with Java using the DEF keyword.

Note that all the nodes I want to change have been given unique names in step 1 above.

4. Extend Java's Applet class to create

your own class which will be aware of your VRML 2 file.

After importing all the classes from the Java language and EAI VRML packages you need in order to compile your code, you extend the Applet class with your own class. For Kaleidoscope world, I created a class I name K with the line:

```
public class K extends Applet implements
EventOutObserver {
```

5. Create an HTML document to connect your VRML 2 world from step 3 with your class in step 4.

The HTML file, K.html (see Listing 1), connects the K.wrl file with the Java code using the HTML tags:

```
<center>
<embed src="k.wrl" border=0 height="500"
 width="500"></center>
<applet code="K.class" mayscript></applet>
```

6. Follow the architecture of the EAI to connect your class from step 4 to the browser object of the Web browser.

I connect to Netscape Navigator with these four lines:

```
JSObject win = JSObject.getWindow(this);
JSObject doc = (JSObject)
win.getMember("document");
JSObject embeds = (JSObject) doc.getMem-
ber("embeds");
browser = (Browser) embeds.getSlot(0);
```

7. Follow the architecture of the EAI to provide attributes in your class in step 4 which map to the DEF nodes of step 3.

For Kaleidoscope world, you need 14 Node type attributes, some of which can take advantage of array indexing. The variable named root[] provides an indexed Node for each shape in the kaleidoscope. The variable named shape[] provides an indexed Node for the shape of each root[] node. The next six Node arrays create attributes for the kaleidoscope nodes, color palette nodes, shape palette nodes, scale control nodes and rotation control nodes. For example, the rotation controls are connected to the K class through the line:

```
Node snsrro[] = {null,null,null,null};
```

The last six Node attributes set up attributes for the color controls.

Each attribute connects to the VRML 2 file through the getNode() method of the browser object. For example, in the init() method of the K class, you can connect the increase green color control to its appropriate Transform node using the line:

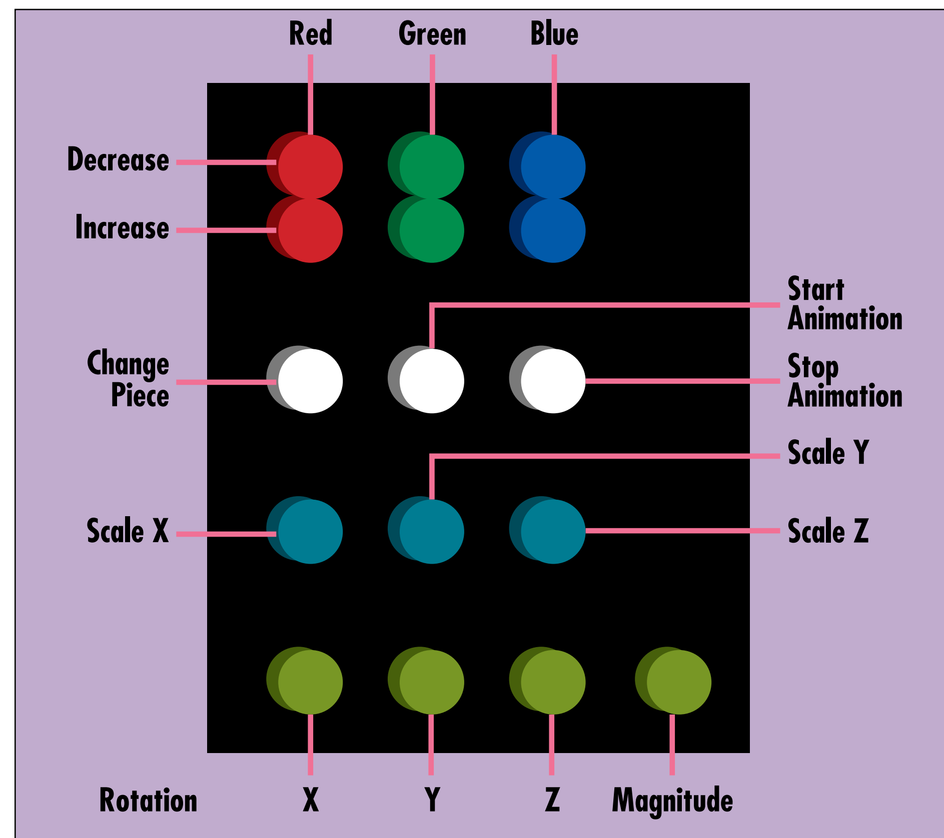


Figure 2: Control panel buttons

```
snsrg = browser.getNode("TSCG");
```

8. Add an additional attribute for changes you want to make to each attribute in step 7 (translation, rotation, scale, color, etc.).

For Kaleidoscope world I decided to package each change of translation, rotation, scale or color in a string which follows the syntax of valid VRML 2 nodes. So, instead of changing each design feature separately, I name removeChildren[] and addChildren[] attributes to allow me to remove VRML nodes and replace them with new ones in a single line of code. Both are of type EventInMFNode. The EAI provides types for changing translation, rotation, scale or color one at a time without having to create VRML 2 syntax strings.

9. Add an additional attribute for each attribute in step 7 you want to be mouse-click aware.

You must also create a separate EventOutSFTime type attribute to enable each touch sensor you want users to be able to use in your world. These attributes connect to touch enabled timers in the init() method of the K class as with the line tTimeg = (EventOutSFTime) snsrg.getEventOut("touchTime"); which connects the touch sensor from the green increase intensity control to its appropriate timer.

10. Create an event-handling routine for each attribute you specify in step 9.

You connect your time attributes to your event-handling routines through the use of an integer as with the line tTimeg.advise(this, new Integer(25));

Then, in the callback() method of your K class (all EAI applications must override this exact callback() method in order to work correctly), you create a routine that will run each time the appropriate touch sensor is activated. In the case of the green intensity control, the event handling routine appears as in the following. Assuming you have assigned the active touch sensor the value 25, the objgreen variable increases by .1, which makes the node being designed greener (if, of course, the node is not already at maximum green intensity):

```
else if (whichNum.intValue()==25) {
        objgreen = objgreen + .1;
        if(objgreen>1) {objgreen = 1;}
}
```

11. Pre-package as text strings any new VRML nodes you may want to place into the world in response to user actions.

For Kaleidoscope world, you create eight different shapes you want to add to the kaleidoscope in different combinations. Each of these eight shapes is associated with a shape attribute which converts from a simple string using the createVrmlFromString method of the browser class. You

remove unwanted nodes, for example shape[8], using lines of code like removeChildren[8].setValue(shape[8]); and add new nodes using lines of code like addChildren[8].setValue(shape[8]);

You must make sure you remove the exact same node as exists in the VMRL 2 scene graph at any time. Otherwise, you cannot add a new node in its place.

12. Create methods (or additional classes with methods) which use the attributes of your class from step 4 to dynamically change each attribute based on the state of the virtual machine.

In Kaleidoscope world, much of the processing works on manipulating strings which create VRML 2 Transform and Shape nodes. All translations take place within the VRML 2 file which makes sense since the kaleidoscope's movement is very repetitive in nature.

Much of the power of the EAI takes advantage of moving VRML 2 objects according to complex logic embedded in Java classes. I would use Java to create kaleidoscope behaviors that were based on real-world physics. In that case, my virtual machine would have to perform collision detection and move the shapes according to the logic that responds to collisions.

Note: Kaleidoscope world was created specifically for the EAI included with Silicon Graphics' CosmoPlayer 1.0 release of their VRML viewer. I compiled my Java code using JDK 1.0.9. In the 1.0 implementation, I perform multiple EAI changes to a node in response to a single touch event. In version 2.0 of CosmoPlayer, multiple changes to a single node within a single event-handling routine are not guaranteed to be handled by the viewer in the order they appear in the Java code. So, a removeChildren.setValue() method is not guaranteed to take place before the related addChildren.setValue(). To avoid any console warnings, you could add another white control button whose event-handling routine would contain the necessary remove node functionality. Then, the add node routine could exist separate from the remove node routine.

Conclusion

The EAI fills in the gaps between the built-in functionality of a VRML 2 viewer, model specification of VRML 2 file syntax and programmability of Java through the use of an embedded Applet on an HTML Web page. Once a VRML 2 file has been read into a Java class structure, the world model can interact with a Java Virtual Machine capable of all kinds of new, creative processing including processing on multiple machines across a network. Creating interactive worlds with the EAI is thus open-ended, yet every Java class you involve in the processing must be delivered to each user. If your user's VRML viewer already contains the processing logic to perform an action to your virtual world, you should attempt to use the appropriate VRML 2 mechanism for enabling that action. Java through the EAI is appropriate for extending a VRML viewer's capabilities perhaps only until the next VRML viewer version that contains the necessary enhancement.

Complete project code for Kaleidoscope may be found at www.sys-con.com/vrml.

About the Author

Bruce Campbell is a virtual reality and human interface research scientist working at the Human Interface Technology Laboratory at the University of Washington in Seattle. He enjoys teaching groupware and Web-related technologies through writing books and lecturing in front of a live audience. Bruce can be reached at bdc@hitl.washington.edu

bdc@hitl.washington.edu

1/2 Ad

Glow Shapes Glow

Building three objects that glow and dim as you click on them

by **Guy Huggins**

The purpose of this article is to explain how to change the appearance of some shapes as a result of the user clicking on one of them. This article assumes that you are a beginning VRML developer.

VRML and JavaScript

One is a language used to describe 3D worlds while the other is the most popular scripting method for the Internet. When you put these two together you get a way to change the attributes of the virtual world as the result of outside events. The new VRML 2.0 standard finally offers the VRML developer avenues to create interactivity within the VRML world as a result of user activity. This powerful and flexible ability is made possible by the new Script node that basically allows the programmer to use code that does not exist within the native VRML syntax.

For this article we will create a world that displays three primitive shapes. A click of the mouse on any of these shapes will change the appearance of the one clicked as well as the other two. To make this happen we will discuss how to "sense" a click from the mouse using the TouchSensor node. Then we shall look at class specifiers and type specifiers that specify criteria for what kinds of data can be received or sent from node to node. After that, we will look at the Script node and, finally, routing data between nodes. Also, for our example, I assume that the reader is familiar with the fundamentals of the VRML specification as well as the basics of JavaScript.

First, we need to set the stage. Using the code in Listing 1, I have created the three primitive shapes that have come to symbolize VRML: the red cube, green sphere and blue cone. The resulting world is seen in Figure 1. Looking at the code you should notice that each object is contained in a Transform node. This grouping node can contain many other nodes within its children field. So far, I have placed only one node as a child to the Transform node, the Shape node. Like the Transform node, the Shape node is a grouping node that can contain other nodes. I use the Appearance, Material and three different "shape" nodes to build this scene. Later, however, we will add the TouchSensor and Script nodes to the list of children.

Now we need to add an element that is capable of "sensing" the actions of the user's mouse. The VRML 2.0 specification includes the TouchSensor node that is capable of detecting whether the mouse pointer is either over an object or has clicked on an object. Think of the TouchSensor node as the VRML equivalent of the JavaScript event methods "onMouseOver" and "onClick". The TouchSensor node "senses" mouse actions for all of the shapes contained as children within the group. In other words, the TouchSensor node acts as a catchall sensor for its siblings. So, now we need to add this node as a child node in each one of the Transform nodes. Listing 2 shows the code after these additions. Now our three primitive shapes are capable of being clicked.

Next, we need to add code that says: "When an object is clicked, make that object glow and make the other objects dull." But before we can do that, we need to digress and understand the concept of class specifiers and type specifiers. Each node in the VRML standard has a definition. Included in the definition are the class specifiers and type specifiers for each item. The class specifiers are eventIn, eventOut, field and exposedField. These define how accessible the items of the node are to other nodes. Think of them as setting a "scope" for different items in the node. The eventIn specifier defines an event that the node is capable of receiving. In opposition, the eventOut specifier defines an event that the node can generate. These specifiers are enforced very strictly. A field class specifier is a private member of a node and cannot receive or generate events to any other

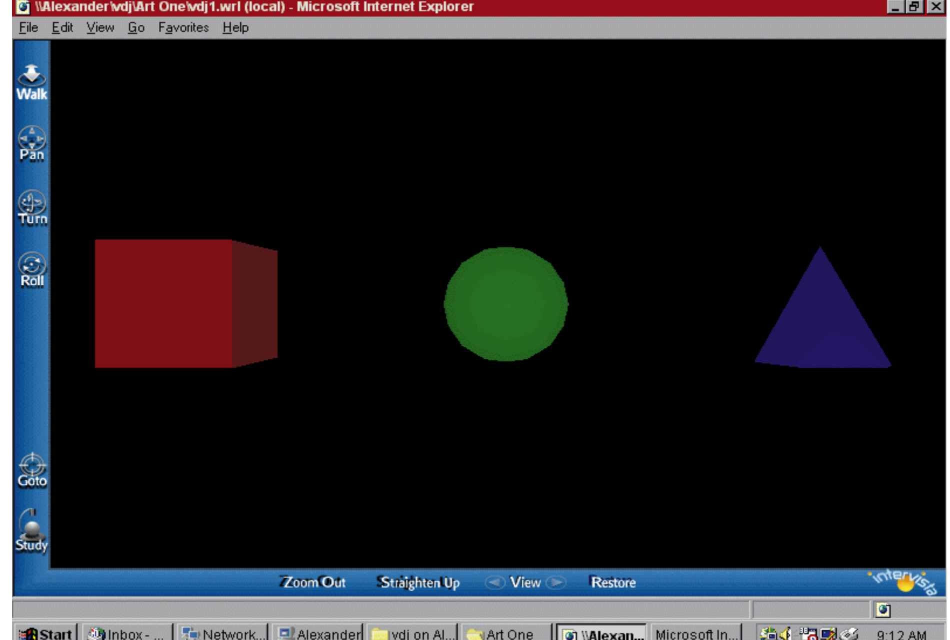


Figure 1

nodes or elements. Its contents are used only within the node itself. Finally, there is the exposedField specifier. This specifier is a hybrid of eventIn and eventOut. When it receives an event (eventIn) it automatically generates a corresponding eventOut. However, it is not necessary to have any element "catching" the resulting eventOut from an exposedField. What the class specifiers allow us to do is set up mail slots for our nodes. Think of eventIns as a node's incoming mail slot that receives messages and events from other VRML elements. EventOuts are a node's outgoing mail slot which sends messages and events to other VRML elements. The field specifier is like mail kept in the top drawer, hidden from the view of the other VRML elements, and exposedFields are like a revolving door: they just receive an event in and send it out to any other node that may be waiting for it.

Along with the class specifiers are type specifiers. Type specifiers define what kinds of data the class specifiers are capable of sending and receiving. All type specifiers start with "SF" or "MF". An "SF" type means that this item can contain only a single value. An "MF" type means that this item can contain multiple values. Following the "SF" or "MF" is the identifier that tells us the type of data being defined. For example, "SFString" means that this particular item can only hold a single string value. "MFInt32" means that this particular item

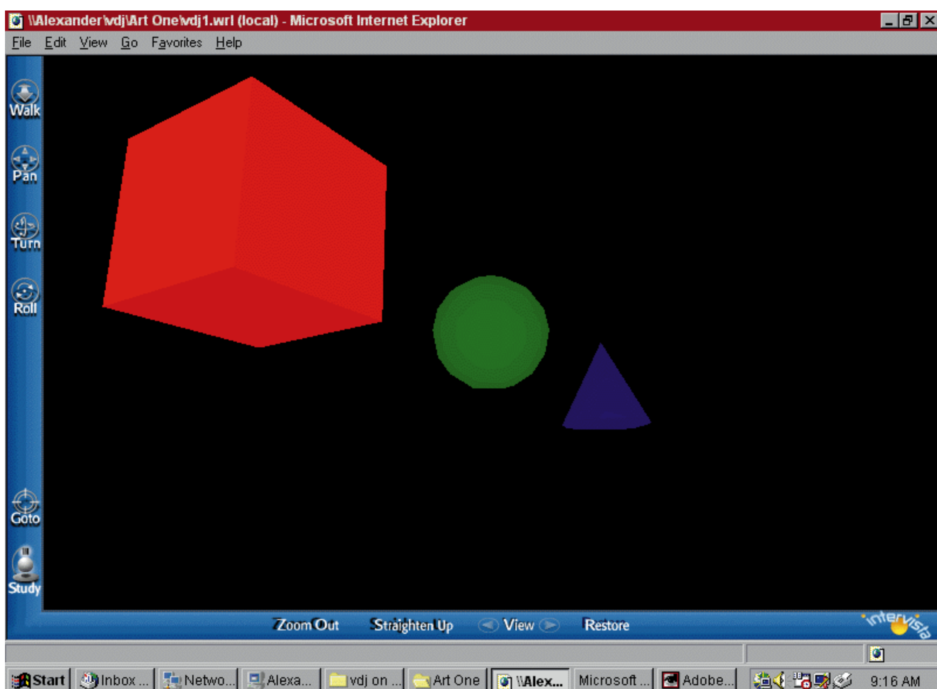


Figure 2

can hold multiple integer values. Think of type specifiers as having two distinct halves. The first half defines how many values can be held and the second half defines the type of data.

These two specifiers work together to restrict what kinds of data can be received as an eventIn or generated as an eventOut. The same works for an exposedField and a field. Therefore, looking at the definition, we see that the TouchSensor node has seven items: six eventOuts and one exposedField. Note that it does not have any eventIns. This only makes sense considering the purpose of this node. This node only sends messages to other nodes about the mouse actions that it senses and it does not need the ability to receive messages.

Listing 1: Creating shapes.

```
Viewpoint{
 position 0 0 10
}

/// Here is my Box at -5,0,0 (relative to the origin)
////////////////////
//////////////////////////////////////////////////////
Transform{
 translation -5 0 0
 children[
  Shape{
   appearance Appearance{
                material DEF boxColor Material{
                                     ambientIntensity 0
                                     diffuseColor .2 0 0
        emissiveColor .2 0 0
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Box{
       size 2 2 2
     }
  }
 ]
}

/// Here is my Sphere at 0,0,0 (relative to the origin)
/////////////////////
//////////////////////////////////////////////////////////////////

Transform{
 translation 0 0 0
 children[
  Shape{
   appearance Appearance{
                material DEF sphereColor Material{
                                     ambientIntensity 0
                                     diffuseColor 0 .2 0
        emissiveColor 0 .2 0
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Sphere{
       radius 1
     }
  }
 ]
}

/// Here is my Cone at 5,0,0 (relative to the origin)
//////////////////////////////////////////////////////
Transform{
 translation 5 0 0
 children[
  Shape{
   appearance Appearance{
                material DEF coneColor Material{
                                     ambientIntensity 0
                                     diffuseColor 0 0 .2
        emissiveColor 0 0 .2
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
```

Now it is time to introduce the Script node. The Script node is sort of a "roll your own" node, meaning that developers have the ability to add any number of eventIn, field and eventOut items they need. Our Script node will be placed as a child within the Transform node, just like the TouchSensor node before it. The Script node contains a native exposedField item "URL". It is this item that we will use to "link" our JavaScript code to out VRML code. This can be accomplished by using a true URL reference to a JavaScript file located on a Web server, or you can embed the JavaScript code directly in the VRML code, much like embedding JavaScript into an HTML document. So now we add the Script node and the JavaScript code to our VRML document as shown in Listing 3. Notice that all of the Script nodes I have added are identical; the only difference is a slight change in the colors assigned by the script itself. The Script node contains the required URL field that houses my JavaScript code. All of the other items were created by me. I manufactured an eventIn item with a single Boolean value and named it "clicked". I also manufactured three other items as eventOuts, each capable of sending single color values that I named "b_color", "s_color" and "c_color", respectively. The script itself, like the rest of this example, is rather elementary. The function "clicked( )" simply declares the

```
   geometry Cone{
       bottomRadius 1
      height 2
     }
  }
 ]
}
```

Listing 2: TouchSensor node as child node.

```
Viewpoint{
 position 0 0 10
}

/// Here is my Box at -5,0,0 (relative to the origin)
/////////////////////////////////////////////////////

Transform{
 translation -5 0 0
 children[
  DEF boxSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
               material DEF boxColor Material{
                                     ambientIntensity 0
                                     diffuseColor .2 0 0
        emissiveColor .2 0 0
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Box{
       size 2 2 2
     }
  }
 ]
}

/// Here is my Sphere at 0,0,0 (relative to the origin)
///////////////////////////////////////////////////////////////

Transform{
 translation 0 0 0
 children[
  DEF sphereSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
               material DEF sphereColor Material{
                                     ambientIntensity 0
                                     diffuseColor 0 .2 0
        emissiveColor 0 .2 0
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Sphere{
       radius 1
     }
  }
 ]
}

/// Here is my Cone at 5,0,0 (relative to the origin)
/////////////////////////////////////////////////////

Transform{
 translation 5 0 0
 children[
  DEF coneSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
               material DEF coneColor Material{
                                     ambientIntensity 0
                                     diffuseColor 0 0 .2
        emissiveColor 0 0 .2
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Cone{
       bottomRadius 1
      height 2
     }
  }
 ]
}
```

Listing 3.

```
///////////////////////////////////////////////////////
Example for VRML Developers Journal - Guy D. Huggins   /////
///   D   E   L   T   A      V - http://www.deltav.net
/////////////////////////////////////////////////////

/// Let's get a little closer
/////////////////////////////////////////////////////

Viewpoint{
 position 0 0 10
}

/// Here is my Box at -5,0,0 (relative to the origin)
/////////////////////////////////////////////////////
Transform{
 translation -5 0 0
 children[
  DEF boxSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
               material DEF boxColor Material{
                                     ambientIntensity 0
                                     diffuseColor .2 0 0
        emissiveColor .2 0 0
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Box{
       size 2 2 2
     }
  }
```

FOCUS ON VRML

variables "b_color", "s_color" and "c_color" and assigns to them new single-color values.

It is important to note that the name of my JavaScript function is identical to the eventIn item I created called "clicked". Naming the JavaScript function identical to an eventIn of its hosting Script node is how you "call" that particular JavaScript function when its corresponding eventIn is received. What this basically says is, "When I receive the 'clicked' eventIn I need to execute the JavaScript function 'clicked'." Also note that the names of my three JavaScript variables are identical to the three eventOut items I created. Once again, the names of the variables and the eventOuts are the same to relate them to one another. Notice now that when you place your mouse pointer over an object in the world it changes to denote that the object is now clickable as the result of the TouchSensor node's presence. But when you click it, nothing happens. There is still a piece missing.

Currently, we have a mechanism in place that detects the click but once the event occurs there is no transport method in place that takes that click event and gives it to our script. What we need to have happen is something like this: "When the user clicks on the object, execute the script found in the Script node. Take the values assigned within the script and send them to the appropriate Appearance node to change the

```
  DEF boxScript Script{
  eventIn SFBool clicked
  eventOut SFColor b_color
  eventOut SFColor s_color
  eventOut SFColor c_color
  url "javascript:
   function clicked()
   {
    b_color = new SFColor(1, 0, 0);
    s_color = new SFColor(0, .2, 0);
    c_color = new SFColor(0, 0, .2);
   }"
 }
 ]
}

/// Here is my Sphere at 0,0,0 (relative to the origin)
/////////////////////////////////////////////////////
Transform{
 translation 0 0 0
 children[
  DEF sphereSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
              material DEF sphereColor Material{
                                    ambientIntensity 0
                                    diffuseColor 0 .2 0
        emissiveColor 0 .2 0
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Sphere{
       radius 1
     }
  }
    DEF sphereScript Script{
  eventIn SFBool clicked
  eventOut SFColor b_color
  eventOut SFColor s_color
  eventOut SFColor c_color
  url "javascript:
   function clicked()
   {
    b_color = new SFColor(.2, 0, 0);
    s_color = new SFColor(0, 1, 0);
    c_color = new SFColor(0, 0, .2);
   }"
 }
 ]
}

/// Here is my Cone at 5,0,0 (relative to the origin)
/////////////////////////////////////////////////////

Transform{
 translation 5 0 0
 children[
  DEF coneSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
              material DEF coneColor Material{
                                    ambientIntensity 0
                                    diffuseColor 0 0 .2
        emissiveColor 0 0 .2
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Cone{
       bottomRadius 1
      height 2
     }
  }
    DEF coneScript Script{
  eventIn SFBool clicked
  eventOut SFColor b_color
  eventOut SFColor s_color
  eventOut SFColor c_color
  url "javascript:
   function clicked()
   {
    b_color = new SFColor(.2, 0, 0);
    s_color = new SFColor(0, .2, 0);
    c_color = new SFColor(0, 0, 1);
   }"
 }
 ]
}
```

Listing 4: Adding routing statements.

```
Viewpoint{
 position 0 0 10
}

/// Here is my Box at -5,0,0 (relative to the origin)
/////////////////////////////////////////////////////

Transform{
 translation -5 0 0
 children[
  DEF boxSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
              material DEF boxColor Material{
                                    ambientIntensity 0
                                    diffuseColor .2 0 0
        emissiveColor .2 0 0
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Box{
       size 2 2 2
     }
  }
  DEF boxScript Script{
  eventIn SFBool clicked
  eventOut SFColor b_color
  eventOut SFColor s_color
  eventOut SFColor c_color
  url "javascript:
   function clicked()
```

brightness of all the objects." The transport mechanism that does this for us is routes.

Routes join the eventOuts from one node to the eventIns of other nodes. These can exist as one-to-one, one-to-many or many-to-one. In order to do this, however, you must name each one of the corresponding nodes using the DEF identifier. Note that in Listing 3 I have named the TouchSensor nodes, the Script nodes and the Appearance nodes for all three objects. These are the nodes that will be involved in the routing statements. Finally, we will add the routing statements to our VRML document as depicted in Listing 4. Essentially, what the routing statements for the box object say is, "Route the click event from the box's sensor (an eventOut from boxSensor) to the eventIn 'clicked' of the box's script (boxScript). Once this is received into the Script node (boxScript), execute the function 'clicked'. As a result of this function, generate the eventOuts b_color, s_color and c_color with the single value colors that were assigned to them in the script. Then, route the eventOut b_color (from boxScript) to the eventIn 'diffuseColor' of the box object (boxColor). Route the eventOut s_color (from boxScript) to the eventIn 'diffuseColor' of the sphere object (sphereColor). Finally, route the eventOut c_color (from boxScript) to the eventIn 'diffuseColor' of the cone object (coneColor)." When we look at this world in the VRML browser we see that when we click on an object, it glows while the others get dim. Figure 2 shows this result.

This simple example shows how to use several VRML elements with JavaScript to build three objects that glow and dim as the user clicks on the varying objects. It is important to note that the result depends on the VRML browser being used. Like HTML, VRML is first interpreted by the browser and then rendered on the screen. Therefore, it is important for the developer to use methods that will be correctly interpreted by the VRML browser. This example was created to use Intervista's World View 2 for its ability to interpret JavaScript. While all VRML 2.0-compliant browsers will read and render the VRML the same, they are not all capable of interpreting the same scripting languages.

About the Author

Guy Huggins is a speaker, writer and trainer on Internet development and networking subjects, as well as a Partner at DELTA V, an Internet and Web-development company based in Arlington, TX. Guy is a Level II Microsoft Sitebuilder, a Microsoft Certified Professional and a guest speaker at Internet World conferences at home and abroad. He can be reached at ghuggins@deltav.net

ghuggins@deltav.net

```
    {
     b_color = new SFColor(1, 0, 0);
     s_color = new SFColor(0, .2, 0);
     c_color = new SFColor(0, 0, .2);
    }"
  }
  ]
}

///Here is my Sphere at 0,0,0 (relative to the origin)
/////////////////////////////////////////////////////

Transform{
 translation 0 0 0
 children[
  DEF sphereSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
               material DEF sphereColor Material{
                                     ambientIntensity 0
                                     diffuseColor 0 .2 0
        emissiveColor 0 .2 0
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Sphere{
       radius 1
     }
  }
    DEF sphereScript Script{
  eventIn SFBool clicked
  eventOut SFColor b_color
  eventOut SFColor s_color
  eventOut SFColor c_color
  url "javascript:
   function clicked()
   {
    b_color = new SFColor(.2, 0, 0);
    s_color = new SFColor(0, 1, 0);
    c_color = new SFColor(0, 0, .2);
   }"
 }
 ]
}

/// Here is my Cone at 5,0,0 (relative to the origin)
/////////////////////////////////////////////////////

Transform{
 translation 5 0 0
 children[
  DEF coneSensor TouchSensor{
  }
  Shape{
   appearance Appearance{
               material DEF coneColor Material{
                                     ambientIntensity 0
                                     diffuseColor 0 0 .2
        emissiveColor 0 0 .2
        shininess .2
        specularColor 0 0 0
        transparency 0
       }
              }
   geometry Cone{
       bottomRadius 1
      height 2
     }
  }
    DEF coneScript Script{
  eventIn SFBool clicked
  eventOut SFColor b_color
  eventOut SFColor s_color
  eventOut SFColor c_color
  url "javascript:
   function clicked()
   {
    b_color = new SFColor(.2, 0, 0);
    s_color = new SFColor(0, .2, 0);
    c_color = new SFColor(0, 0, 1);
   }"
 }
 ]
}

/// Here are the ROUTE statements
/////////////////

ROUTE boxSensor.isActive TO boxScript.clicked
ROUTE boxScript.b_color TO boxColor.set_diffuseColor
ROUTE boxScript.s_color TO sphereColor.set_diffuseColor
ROUTE boxScript.c_color TO coneColor.set_diffuseColor
ROUTE sphereSensor.isActive TO sphereScript.clicked
ROUTE sphereScript.b_color TO boxColor.set_diffuseColor
ROUTE sphereScript.s_color TO sphereColor.set_diffuseColor
ROUTE sphereScript.c_color TO coneColor.set_diffuseColor
ROUTE coneSensor.isActive TO coneScript.clicked
ROUTE coneScript.b_color TO boxColor.set_diffuseColor
ROUTE coneScript.s_color TO sphereColor.set_diffuseColor
ROUTE coneScript.c_color TO coneColor.set_diffuseColor
```

Creating & Using a

by Donald Fowler

Over the past several years, wizards have become the preferred UI method of guiding a user through a series of steps. In this article we explore the methodology involved in creating a general wizard class. We then discuss the supporting classes and interactions we should consider to successfully complete a wizard.

As it turns out, designing and implementing this wizard framework exposes many of the real-world design and programming issues we face when creating Java applications.

Overview

The first step in designing our wizard class is to come up with an interface. An interface is a prototype for a class in which we describe the class' functionality without getting into implementation details. When designing an interface, think about how someone would use the class and jot down some pseudo-code. Translate these ideas into member functions to create the interface. One thing to remember is that the initial pass at an interface will rarely be the last. You will probably end up modifying and improving the interface several times during the course of development.

Since there are several different ways we could go about designing our wizard interface and corresponding implementation class, we need to decide what features are important to us. For instance, what types of containers do we want our wizard to be capable of displaying? What events do we want our wizard to support? Do we want to support multicasting of events, and, if so, how? Do we want to have random access to our wizard pages or is sequential access sufficient? What do we want our wizard buttons to look like?

There are three main elements to a wizard.

- First, there are the navigation buttons (next, previous, cancel, etc.).
- Second, there are the information and controls presented by the wizard, which I refer to as "pages."
- Finally, there is the dialog, typically modal, that contains the active page and the navigation buttons.

The Layout

We now need to think about how the wizard dialog should be laid out. Because the wizard navigation buttons are independent of the content pages, we want to create two distinct areas within the dialog. The first area, which will occupy the upper region of the dialog, will consist of the information that changes between each wizard page. The second area, which will occupy the lower region of the dialog, will contain the wizard navigation buttons. This is where Java's layout manager classes will come in handy.

Messaging

The next issue we need to deal with is messaging. We need to provide a mechanism in our wizard class to notify the outside world of events. To do this we will need to create our own event and listener classes so that the appropriate event occurs when one of our navigation buttons is clicked.

If you are experienced in using the AWT, you are probably familiar with event listeners and event adapters. For instance, Java provides a class called WindowAdapter to accept events that occur on a window. The class is provided mainly for convenience reasons, providing empty methods for all the various window events. To actually do something, you must extend the WindowAdapter class and tell the window about it by calling the AddWindowListener function.

We are going to use a similar structure in our wizard framework. Our wizard class will allow a user to set a particular WizardAdapter to have the wizard messages sent to. We are going to design our wizard so that there can be only one WizardAdapter active at any given time. Note, however, that this does not prevent us from multicasting the events, if desired, since you can add that functionality to the WizardAdapter class. Why not allow multiple listeners to be registered with the wizard class as they can in Java's Window class? Consider the events that our wizard class is producing. In response to these events (next, previous, finish, etc.) we might validate some of the fields in the current page, then tell the wizard class to display the next page or the previous page, or to close the dialog. Can you see some of the potential problems if these events were to be processed by independent listeners? What if the first listener told the wizard to advance to the next page, while the next listener, not knowing about the first, did the same thing? In this case you would get the unintended result of advancing two pages whenever the next button was clicked.

By limiting the wizard class to one WizardAdapter, we require a central point of control by the user of the wizard class. This is important because it is the user of the class, not the wizard class itself, that knows how the wizard should be used. If users want events to be multicast, they can implement that multicasting inside their WizardAdapter, simply forwarding on the wizard events to all interested parties. The lesson here is to provide functionality appropriate for the object being modeled. Don't add capabilities just because you think they're cool.

Example Classes

To show how you use the wizard class, I created a sample vacation wizard. This wizard is relatively simple, yet illustrates most of the features of the wizard class, as well as some interesting "gotchas."

Figure: 1

To effectively use the wizard class, several application-specific support classes are created, as shown in Figure 1. An application-specific version of the WizardAdapter class (VacationWizardAdapter) is created to handle messaging, while a class called VacationWizard is used to create the wizard content pages and control the interactions between wizard pages. The VacationWizard class knows about both the wizard class and the VacationWizardAdapter class. The VacationWizardAdapter takes a VacationWizard instance in its constructor so that it knows whom to send messages to. The three classes used together provide a flexible and extensible mechanism for handling wizards.

The Details

Let us begin by taking a closer look at the interface definition for the wizard class (see Listing 1). From this interface definition we can see that in addition to the basic wizard navigation support, we also want our wizard to be able to support advanced features such as hidden panels, keyboard navigation, help button and messaging.

In implementing our wizard class, one of the first issues we must confront is what existing Java class we want to extend. As I mentioned in the overview, a modal dialog is typically used as the container for the various wizard elements. To get some exposure to some of the new JFC classes, we will extend the JFC class Jdialog.

```
public class Wizard extends
com.sun.java.swing.JDialog
    implements WizardControllerInterface
```

Notice that this declaration says we are going to implement the WizardInterface that we listed earlier. Therefore, our class definition must define all the member functions we defined in our WizardInterface.

The Layout

As I alluded earlier, Java's layout classes come in handy when implementing our Wizard class. Using a GridBagLayout, we can lay out the wizard dialog's contents as shown in Figure 2. The uppermost area is itself a Jpanel that uses its own layout manager, CardLayout, to manage the wizard pages. I won't go into the details of laying out controls using the GridBagLayout here, but look at the source code if you are interested.

Messaging

How do we go about handling the messaging? First, we need to create a new event type called a WizardEvent by extending Java's EventObject class (see Listing 2). We will then need to create a WizardListener class by extending the Java's EventListener class. Our WizardListener class will handle the events shown in Listing 3.

Inside our wizard class we need to create ActionListeners for each of our wizard navigation buttons, such as:

```
previous_.addActionListener(new PreviousBut-
tonListener());
```

Our listener classes simply fire off an appropriate event when one of our navigation buttons is clicked. Listing 4 shows how one of these functions, PreviousButtonListener, is implemented.

In addition, we need to add KeyListeners for each of our wizard navigation buttons since we want to support keyboard control of our navigation buttons using the Enter Key.

```
next_.addKeyListener(new NextEnterKeyListen-
er());
```

All we are doing here is checking to see if the Enter Key is pressed while the focus is on one of our navigation buttons. If it is, we fire off the appropriate event. Listing 5 shows how the NextEnterKeyListener is implemented.

Page Navigation

How do we accomplish the task of moving between pages in the wizard class? As mentioned earlier, the Java Layout Manager called CardLayout is used to switch between wizard pages. The CardLayout is ideal for use with wizards because it allows easy access to all the pages in the layout either sequentially or randomly via a name. (See the complete listing for details.)

The wizard class offers the ability to hide certain pages of the wizard if desired. As we will see in our example, many times we might want to create a wizard that shows certain pages conditionally. This is accomplished by using the wizard functions hidePanel and unHidePanel. PanelAttribute class is used inside the wizard class to maintain state about each page of the wizard. If a page is marked as hidden, that page is bypassed by the forward and next logic of the wizard. (See the complete listing for details.)

Miscellaneous

Our wizard class also has the ability to display messages. The user of the wizard class might want to display a message stating that all the required fields of the wizard have not been filled in. A user might also want to display a specific field validation message, such as "value must be between 10 and 50." The wizard class handles these situations using a specific function called displayRequiredFieldMessage, as well as provides a generic error message function called displayErrorMessage that takes a title and a message string.

An Example

To create an actual wizard of your own, you need to decide on the content and order of the wizard pages. Once you have figured that out, you need to decide what container you will use for the wizard pages. Our wizard class allows you to use any class derived from the AWT's Component class. For the sample wizard I used a variety of components such as the AWT's Panel and ScrollPanel class, as well as the new JFC's Jpanel class.

Once you have your wizard pages designed, you need to deal with any required interactions between the various controls on the same page as well as the interactions between pages. For instance, in the VacationWizard example, the second page of the wizard is displayed only if Hawaii has been selected as the destination. Therefore, we need to be able to conditionally hide/unhide this wizard page based on the value of the radio buttons on the first wizard page.

To check the values of various controls at the appropriate time, messages need to be sent when a wizard page is activated and deactivated. This is the job of our VacationWizardAdapter. Since the VacationWizardAdapter is the class that is getting the messages (previous, next, finish, etc.) from

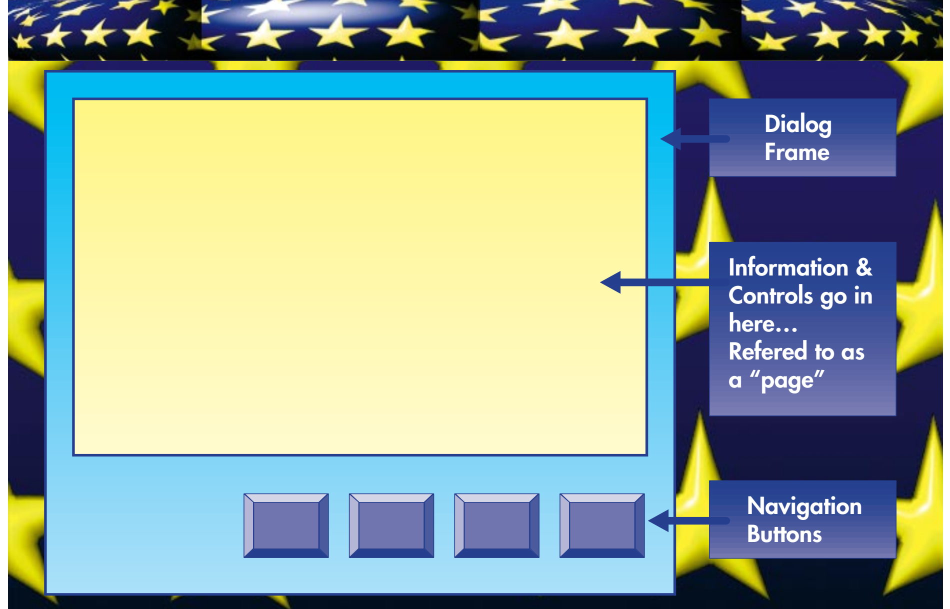


Figure: 2

the Wizard Class, it is responsible for calling the initializePage and finalizePage functions of the VacationWizard class, passing the current wizard page as a parameter. The finalizePage function can veto moving to the next page by returning false. This is a handy mechanism for validating fields and preventing the wizard from moving forward until all the appropriate fields and values are filled in.

The code in the VacationWizard class deals mostly with creating the content pages for the wizard. However, there is one interesting thing to look at. While creating and testing the wizard pages, I discovered some undesired behavior with the AWT's TextArea class. I wanted to use a non-editable TextArea to display some explanatory text at the top of each wizard page (see Figure 3). At first glance everything looked okay. However, when I used the tab key to tab around the dialog, I noticed that once the TextArea received the focus, it did not want to give it back. Although I'm sure the TextArea was happy, the rest of my controls were feeling left out.

I imagine this behavior stems from the fact that in a standard editable text area, the tab key really means to put in a tab character. Fortunately, you can get around this problem by deriving a new class from TextArea. I called my new class UnfocusedTextArea (see Listing 6). After searching the documentation, I found a handy function in the AWT's Component class, from which TextArea descends, called isFocusTraversable. The key to solving the problem lies in overriding this function in our UnfocusedTextArea class so that it simply returns false. This means that the text area will never actually get the focus -- which is exactly what we want to happen in this case.

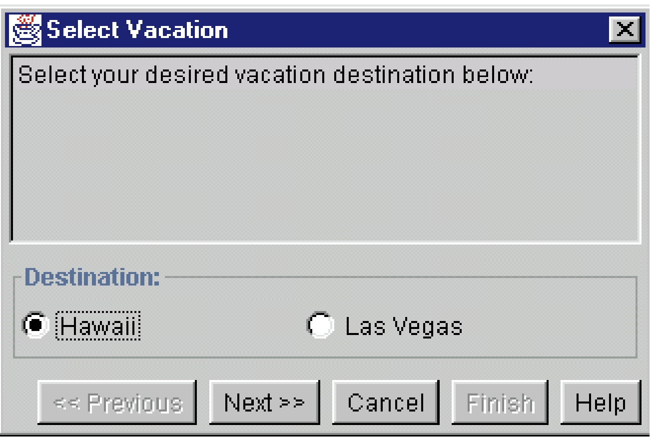


Figure: 3

Conclusion

There are many different ways to design a wizard class and its supporting classes. I have presented one way that provides a flexible and extensible framework for creating real-world wizards. One thing to keep in mind when building applications of your own is that most real-world problems require more than one class to model. In most cases it is best to solve problems with mini-frameworks that use a general-purpose class or classes (like the Wizard class) combined with one or more specializing classes (like the VacationWizard and VacationWizardAdapter classes).

About the Author

Donald Fowler works as a software design developer at Rogue Wave Software where he is currently the technical lead for the Software Parts Manager product line. He has 15 years of software design and development experience specializing in GUI programming and 7 years' experience in object-oriented design and programming. Donald can be e-mailed at donald@roguewave.com.

donald@roguewave.com

Listing 1: WizardInterface

```
import java.awt.Component;

//Interface for a Wizard Class
public interface WizardInterface
{
//
public void setWizardAdapter(WizardAdapter adapter);
public WizardAdapter getWizardAdapter();

public void setTitle(String title);

//Optional Help Button
public void setHelpVisible(boolean state);
public boolean isHelpVisible();

//Enter key behavior
public void setEnterKeyActive(boolean state);
public boolean isEnterKeyActive();

//Panels can be hidden - if hidden, they are skipped by previ-
ous,next
public void hidePanel(int index);
public void hidePanel(Component panel);
public boolean isPanelHidden(int index);
public boolean isPanelHidden(Component panel);
public void unHidePanel(int index);
public void unHidePanel(Component panel);

public int getNumberOfPanels();
public int getCurrentPanelIndex();
public Component getCurrentPanel();
public Component getPanel(int index);

public int getPanelIndex(Component panel);


//allows you to go directly to any panel in the wizard
//returns false if panel is hidden and could not be displayed
public boolean setCurrentPanel(int index);

public void doPrevious();
public void doNext();
public void doFinish();
public void doCancel();
public void doHelp();

public boolean isPreviousEnabled();
public boolean isNextEnabled();
public boolean isFinishEnabled();
public boolean isCancelEnabled();
public boolean isHelpEnabled();

}
```

Listing 2: WizardEvent

```
import java.util.*;

public class WizardEvent extends java.util.EventObject {
    Wizard wizard_;
```

```
    public WizardEvent(Object source, Wizard wiz){
        super(source);
        wizard_=wiz;
    }
    public Wizard getWizard(){
        return wizard_;
    }
}
```

Listing 3: WizardListener

```
import java.util.EventListener;

public interface WizardListener extends EventListener {
    public void nextButtonClicked(WizardEvent evt);
    public void previousButtonClicked(WizardEvent evt);
    public void cancelButtonClicked(WizardEvent evt);
    public void finishButtonClicked(WizardEvent evt);
    public void helpButtonClicked(WizardEvent evt);
    public void wizardActivated(WizardEvent evt);
}
```

Listing 4: PreviousButtonListener Class

```
private class PreviousButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            //broadcast WizardEvent out to everyone
            WizardEvent evt;
            evt=new WizardEvent(e,thisWizard_);
            wizardAdapter_.previousButtonClicked(evt);
        }//actionPerformed
    }//PreviousButtonListener
```

Listing 5: NextEnterKeyListener

```
private class NextEnterKeyListener extends KeyAdapter {
        public void keyReleased(KeyEvent k) {
            if (enterKeyActive_){
                if (k.getKeyCode() ==
java.awt.event.KeyEvent.VK_ENTER) {
                    WizardEvent evt;
                    evt=new WizardEvent(k,thisWizard_);
                    wizardAdapter_.nextButtonClicked(evt);
                }//if keycode
            }//enterKeyActive_
        }//keyReleased
    }//NextEnterKeyListener
```

Listing 6: UnfocusedTextArea Class

```
private class UnfocusedTextArea extends TextArea {
        public UnfocusedTextArea(String s, int row, int col, int
visibility){
            super(s,row,col,visibility);
        }

        public boolean isFocusTraversable(){
            return false;
        }

    }
```

1/2 Ad

JDJ FEATURE

Adding a Middle Tier to Your Java Code Using Jaguar CTS

PART 2
Server Side Coding

by **Sean Rhody & James A. Walker**

Last time we discussed the general capabilities of Jaguar CTS from Sybase. This month we're going to look at what it takes to build a very simple component and place it inside Jaguar.

Since a middle tier isn't much good without something for it to talk to, we're going to use the Pubs2 database, which is an example database provided with Sybase SQL Server 1.1 Specifically, we'll use the authors table which is described in Figure 1.

Imports

All of the example code is written in one file. It provides methods to do select, insert, update and delete on the authors table. To work with Jaguar, in addition to whatever usual imports you use, you must import a number of packages developed by Sybase. For our example, we use what is shown in Table 1.

Methods

We'll declare a class called Pubs2CompImpl, which stands for Pubs2 component implementation. The "Impl" on the end of the class name is sort of a Jaguar standard for the implementation of server components – the actual component name will be Pubs2Comp. In addition to the constructor for this class, there are eight methods.

The code listing shows the class definition. The constructor for Pubs2CompImpl is defined to throw a JException (Jaguar Exception), which will happen for certain events. There are also several class variables, as shown in Table 2.

In actuality, it's not difficult to write a component for Jaguar. Some of the classes that Jaguar provides such as Connection, Statement, ResultSet and JCMCache make it easy to get data from an SQL database and send it on to the client, as you'll see in a moment.

Listing 1 shows a couple of variables used to connect to the SQL Server database that we are using as our target. We've used the Sybase Pubs2 example database, which should be available at least to the Sybase community. For those of you who have never heard of this sample database, it contains information about an imaginary publishing company, particularly things like authors, books, royalties, etc. Our examples are all very straightforward (I hope) so there should be no trouble following along.

The password and user name should be familiar to any database developer. What might be new to some is that we're connecting with JDBC, and in particular with Sybase JConnect (hence the tds portion of the DATASERVER string). The string tells you what is shown in Table 3.

In our example the values are coded, but there is no reason why a component cannot be created that will accept these parameters in some setup function. This would allow the application to drive what database to connect to.

The constructor for our sample class is shown in Listing 2. This function illustrates the use of the Jaguar Connection Manager (JCM). As we mentioned briefly in Part One of this series, one of the services Jaguar provides is a Connection Cache. This allows a number of logical connections to be managed over a smaller number of physical connections. Since database connection is one of the biggest bottlenecks in application responsiveness, the cache provides a way to speed up the process.

JCM is a static class implemented by Jaguar. The getCache() function returns a connection to the database that is stored in the _cache variable. This function can throw an exception, which must be caught. Should an exception occur, we use the Jaguar static class method writeLog() to send a message to the server log and set the cache to null. Finally, if we've been unsuccessful in connecting to the database, we throw a JException that will be caught by the client.

Listing 3 shows the qt() function, which encloses a string in double quotes. This is useful for dealing with strings and character data in SQL statements.

The doQuery() function is shown in Listing 4. In our sample code, there are two types of operations – those that return result sets (select statements) and those that do not (insert, update and delete statements). The doQuery() function is used to take an SQL statement and return a result set. It takes a string containing the select statement, a statement variable and a Connection variable. It uses the statement object to execute the SQL statement, and returns a result set or throws an exception. Basically, the function is used to encapsulate the common logic of creating a result set.

Listing 5 shows the selAuthors() function, which makes use of the doQuery() function to select all of the data from the authors table. It starts by building an SQL statement into the SQL variable (I apologize to all of you purists who hate select star – it was an attempt to keep things simple). Within the try block, it then attempts to get a connection from the cache. If that is successful, it creates a statement using the createStatement() function. A result set is generated using the doQuery() function. Next comes a very innocent looking function that allows you to send a result set to a

Package	Description
com.sybase.jaguar.util.*	Utility methods for server and client components
com.sybase.jaguar.sql.*	Methods for processing result sets
com.sybase.jaguar.server.*	Methods for server components
com.sybase.jaguar.jcm.*	Methods for working with connection caches

Table 1

Data Type	Variable Name	Description
final String	DB_USER	Constant username for connecting to the database
final String	DB_PWD	Constant password for connecting to the database
final String	DATASERVER	Constant connection string for connecting to the data server
String	sql	An SQL Statement
static Connection	con	A connection object
static Statement	stmt	A statement object
static ResultSet	rs	A result set object (the data returned from a query)
JCMCache	_cache	A connection to the Jaguar cache manager

Table 2

Section	Meaning
jdbc	driver type
sybase	database type
tds	protocol type - TDS stands for tabular data stream and is a Sybase protocol associated with JConnect
89.86.200.1	The address of the server
5000/pubs2	The port the server listens on, and

Table 3

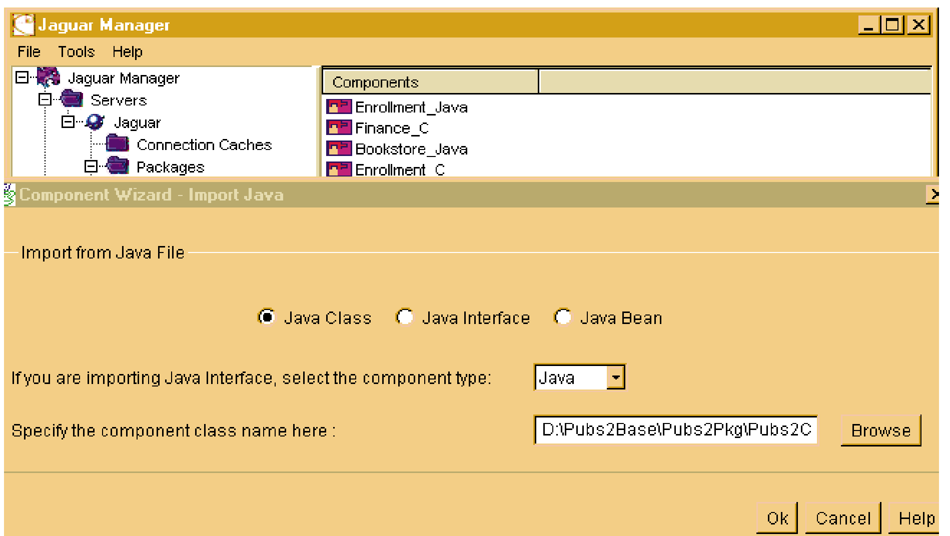


Figure 1: Registering the component in Jaguar

client application. In this sense, Jaguar is particularly well suited to Java because the result set on the client side looks just as if it came from a database (which, in fact, it did here). There are other functions that allow you to manually create and forward a result set in a Jaguar server component so you're not limited to what you can pull out of a database. For example, you could process the results of a query and create a summary of the information, and send that instead of a year's worth of information. After the result set is sent, the statement and the connection are closed.

The remaining listings show other functions that we've created to illustrate the concepts of insert, update and delete. Since these are probably familiar to you from a conceptual standpoint, and the code is very similar to the code used in Listings 4 and 5, we won't go into too much detail. Instead, we'll look at how we get the code into Jaguar and close by discussing how we will get the code to the client, which will be the subject of Part 3 of this article.

Registering the Component in Jaguar

It's not particularly difficult to register components in Jaguar. You start Jaguar manager and connect to the server. Then you have several options on how you want to make a component available to the client. First of all, it's important to understand that for a component to be available to a client application it must be part of a package that is in turn part of the server. Server can get kind of confusing in this context, because while you can have multiple named servers on a single machine, they all have the same source for components. You can, however, set up one named server for development and another for deployment, and place packages into deployment only after they have been thoroughly tested in the development named server.

Note that each named server listens on a different port and can be started and stopped independently of each other, but they all share the same shared memory space. Think of them as analogs to database instances.

You can either create a component, place it in a package and then place the package into a server, or you can create a component directly in a package that is directly inside a server. One reason you might want to progress in steps is because you have to mark any method that returns a result set as doing so. Unfortunately, there's no way that Jaguar can tell that you want to return a result set. If your component has a large number of methods, this can be tedious. Also, if you have just reregistered an existing component after some changes, this will allow you time to set all of these attributes before some overenthusiastic developer tries to test the changes. As a side note, creating a second named server that listens on a different port allows you to do the equivalent of putting Jaguar in single user-mode. Because all the clients expect to connect to a particular port (the default is 7878), if you run a maintenance instance on some other port (say 5858), you can register components without interference from developers.

Figure 1 shows the registration of a component. You have three choices initially: ActiveX, Java or C/C++. After choosing Java and selecting a logical component name (it does not have to match your class name), you have a choice of selecting a class file or a JavaBean as the source. By default, Jaguar looks in its \html\classes directory for the source files, so you should put your component here or below here in a package directory if the component is to be part of a package.

Once you've told Jaguar where the component is, it pulls in all of the methods that are declared void (the void declaration is a restriction in 1.1 that will be lifted in the next major release according to Sybase). After you mark the methods as returning result sets, you are almost done.

Part Three

In Part Three, we'll show you how to generate Java stubs for use by client programs. Then we'll provide a fairly simple Java application that will connect to Jaguar and exercise the methods that we've created here.

About the Authors

Sean Rhody is a respected industry consultant and a leading authority on PowerBuilder. Sean is also editor-in-chief of the **PowerBuilder Developer's Journal** *and is the editor of* **PowerBuilder 6.0: Secrets of the PowerBuilder Masters.** *You can contact Sean at roadhog@compuserve.com or roadhog@nac.net.*

James A. Walker is a senior consultant with Sybase's NorthEast Professional Services. Currently, he is on a project that utilizes Jaguar CTS and PowerJ. You may reach him at walker@sybase.com.

Listing 1: Variable initialization.

```
final String  DB_USER     = "pubs2_user";      // Database user
name.
final String  DB_PWD      = "pubs2_user";      // Database user
password.
final String  DATASERVER =
"jdbc:sybase:Tds:89.86.200.1:5000/pubs2"; // URL to the dataserv-
er.

// other variables
String sql;
static Connection con = null;
static Statement stmt = null;
static ResultSet rs    = null;
JCMCache _cache = null;
```

Listing 2: Constructor.

```
public Pubs2CompImpl() throws JException
{
    try
```

```
    {   // Get a cache reference.
      _cache = JCM.getCache(DB_USER, DB_PWD, DATASERVER);
    }
    catch (Exception e)
    {
      Jaguar.writeLog(true, "Pubs2CompImpl constructor getCache():
" +
e.getMessage());
      _cache = null;
    }

    /*
    ** If we can't get a handle to the connection cache, log an
error and throw
    ** a Jaguar exception. The JException gets caught by the com-
ponent stub class,
    ** which in this case is Pubs2Comp.class.
    */
    if (_cache == null)
   {
      Jaguar.writeLog(true, "Pubs2CompImpl(): Could not get con-
nection cache reference.");
      throw new JException("Pubs2CompImpl constructor: Could not
create connection
cache.");
    }
  }
```

Listing 3: The qt function.

```
public String qt(String str)
{
    return("\"" + str + "\""); // Enclose the string in quotes.
}
```

Listing 4: The doQuery function.

```
private ResultSet doQuery(String sql, Statement stmt, Connection
con) throws JException
{
    int rows;

    try {
      rows = stmt.executeUpdate("set quoted_identifier off");
      boolean results = stmt.execute(sql);
      rs = stmt.getResultSet();
    }
    catch (SQLException sqle) {
      Jaguar.writeLog(true, "doQuery(): " + sqle.getMessage());
    }
    catch (Exception e) {
      Jaguar.writeLog(true, "doQuery(): " + e.getMessage());
    }
    return rs;
}
```

Listing 5: The selAuthors Function.

```
public void selAuthors() throws JException
{
    sql =       "SELECT * ";
    sql = sql + "FROM authors ";
    sql = sql + "ORDER by au_lname";

    try {
      Connection con = _cache.getConnection(JCMCache.JCM_FORCE);
      Statement stmt = con.createStatement();
      ResultSet rs = doQuery(sql, stmt, con);
      JContext.forwardResultSet(rs);
      stmt.close();
      _cache.releaseConnection(con);
    }
    catch (Exception e) {
      Jaguar.writeLog(true, "selAuthors(): " + e.getMessage());
    }
 }
```

Listing 6: The selAuthorsByLname function.

```
public void selAuthorsByLname(String au_lname) throws JException {

    // Build the sql statement and put strings in double quotes.
    sql =       "SELECT * ";
    sql = sql + "FROM authors ";
    sql = sql + "WHERE au_lname = " + qt(au_lname) + " ";
    sql = sql + "ORDER by au_lname";

    try {
      Connection con = _cache.getConnection(JCMCache.JCM_FORCE);
      Statement stmt = con.createStatement(); // Create a connec-
tion statement.
      ResultSet rs = doQuery(sql, stmt, con); // This returns a
result set so call doQuery().
      JContext.forwardResultSet(rs);          // Forward the
result sets to the client.
      stmt.close();                            // Release all
statement resources.
      _cache.releaseConnection(con);          // Release the con-
nection back into the pool.
    }
    catch (Exception e) {
      Jaguar.writeLog(true, "selAuthorsByLname(): " + e.getMes-
sage());
    }
  }
```

Listing 7: The insAuthors function.

```
public void insAuthor(String au_id, String au_lname, String
au_fname,
                        String phone, String address,  String
city,
                        String state, String country,  String
postal_code) throws JException {

    sql = "INSERT INTO authors VALUES(";
    sql = sql + qt(au_id) + "," + qt(au_lname) + "," +
qt(au_fname) + ",";
    sql = sql + qt(phone) + "," + qt(address)  + "," + qt(city) +
",";
    sql = sql + qt(state) + "," + qt(country)  + "," +
qt(postal_code) + ")";

    try {
      Connection con = _cache.getConnection(JCMCache.JCM_FORCE);
      Statement stmt = con.createStatement();
      doActionQuery(sql, stmt, con);
      stmt.close();
      _cache.releaseConnection(con);
    }
    catch (Exception e) {
      Jaguar.writeLog(true, "insAuthor(): " + e.getMessage());
    }
  }
```

Listing 8: The delAuthors Function.

```
public void delAuthor(String au_id) throws JException {

    sql =       "DELETE ";
    sql = sql + "FROM authors ";
    sql = sql + "WHERE au_id = " + qt(au_id);
```

```
    try {
      Connection con = _cache.getConnection(JCMCache.JCM_FORCE);
      Statement stmt = con.createStatement();
      doActionQuery(sql, stmt, con);
      stmt.close();
      _cache.releaseConnection(con);
    }
    catch (Exception e) {
      Jaguar.writeLog(true, "delAuthor(): " + e.getMessage());
    }
  }
```

Listing 9: The updAuthors function.

```
public void updAuthor(String au_id, String au_lname, String
au_fname,
                        String phone, String address,  String
city,
                        String state, String country,  String
postal_code) throws JException {

    sql =       "UPDATE authors ";
    sql = sql + "SET au_lname    = " + qt(au_lname) + ",";
    sql = sql + "    au_fname    = " + qt(au_fname) + ",";
    sql = sql + "    phone       = " + qt(phone)    + ",";
    sql = sql + "    address     = " + qt(address)  + ",";
    sql = sql + "    city        = " + qt(city)     + ",";
    sql = sql + "    state       = " + qt(state)    + ",";
    sql = sql + "    country     = " + qt(country)  + ",";
    sql = sql + "    postalcode  = " + qt(postal_code);
    sql = sql + "WHERE au_id = " + qt(au_id);

    try {
      Connection con = _cache.getConnection(JCMCache.JCM_FORCE);
      Statement stmt = con.createStatement();
      doActionQuery(sql, stmt, con);
      stmt.close();
      _cache.releaseConnection(con);
    }
    catch (Exception e) {
      Jaguar.writeLog(true, "updAuthor(): " + e.getMessage());
    }
  }
```

Listing 10: The doActionQuery function.

```
private void doActionQuery(String sql, Statement stmt, Connection
con) throws JException {

    int rows = 0;

    try {
      rows = stmt.executeUpdate("set quoted_identifier off");
      rows = stmt.executeUpdate(sql.toString());
      rows = stmt.executeUpdate("commit");
    }
    catch (SQLException sqle) {
     Jaguar.writeLog(true, "doActionQuery(): " + sqle.getMessage());
    }
    catch (Exception e) {
      Jaguar.writeLog(true, "doActionQuery(): " + e.getMessage());
    }
  }
```

1/2 Ad

Java Multithreading

by **David Nelson-Gal, *Director,*** *Java Technology Group, Sun Microsystems*
& Devang Shah, *Engineer, Sun Microsystems*

Although Java is cross-platform, its performance and quality are greatly influenced by the features of the underlying native platform. Since the Java language and environment includes concurrency and multithreading (MT) as integral components, the native operating system's multithreading model and environment will greatly influence Java applications' quality and performance on that system.

Sun's Java Virtual Machine (JVM), central to the performance and scalability of the Java Development Kit (JDK) for Solaris, is designed to take full advantage of multiprocessor computing systems by using the native multithreading capabilities of Solaris. It performs bytecode interpretation using native multithreading and fast synchronization and, later this summer, will feature an improved memory system. These features provide developers with significant performance boosts required for successfully developing and deploying Java applications that deliver solid performance and fast response times under peak loads.

Java Multithreading Implementation Comparisons

Understanding the architectural advantages of one native MT environment/architecture over another is critical to an understanding of the advantages of one Java implementation over another. Since a typical JVM runtime is implemented on top of the traditional platform, a richer, architecturally superior MT platform will obviously translate to a superior Java MT environment for Java applications on that platform.

The native OS threads model greatly influences Java application performance because, on Solaris and other platforms, Java threads get mapped directly onto operating system native threads. In this sense, the Java thread model is a platform-independent abstraction on top of native threads that attempts to hide as many platform-specific details from the applications developer as possible and provide a single, cross-platform abstraction.

However, there are platform-specific differences between native Solaris threads and native threads on other platforms that affect Java threads' performance and resource consumption on each platform.

Advantages of Java Multithreading on Solaris

Java on Solaris leverages the multithreading capabilities of the operating system kernel while allowing developers to create powerful Java applications using thousands of user-level threads, if needed, for multiprocessor or uniprocessor systems through a very simple programming interface.

The Java on Solaris environment supports the many-to-many threads model. As illustrated in Figure 1, the Solaris two-level architecture separates the programming interface from the implementation by providing an intermediate layer, called lightweight processes (LWPs). LWPs allow application developers to rapidly create very fast and cheap threads through a portable application-level interface. Developers simply write applications using threads. The runtime environment, as implemented by a threads library, multiplexes and schedules runnable threads onto "execution resources," the LWPs.

Individual LWPs operate like virtual CPUs that execute code or system calls. LWPs are dispatched separately by the kernel, according to scheduling class and priority, so they can perform independent system calls, incur independent page faults and run in parallel on multiple processors. The threads library implements a user-level scheduler that is separate from the system scheduler. User-level threads are supported in the kernel by the kernel-schedulable LWPs. Many user threads are multiplexed on a pool of kernel LWPs.

Solaris threads provide an application with the option to bind a user-level thread to an LWP or to keep a user-level thread unbound. Binding a user-level thread to

Solaris MT Architecture

Though accessing Solaris-specific features from Java applications is not recommended, this list is included here to illustrate the richness of the Solaris MT architecture:

- The ability to define bound or unbound threads for user- or system-level control of application concurrency: note that a bound Java thread may be created only via native methods.
- In addition, the application can control application concurrency through a programmatic interface.
- The ability to bind a user-level thread (through native methods) to an LWP that is dedicated to a single processor: this feature is useful to real-time applications running on multiprocessor systems.
- Synchronization primitives that have interprocess scopes.
- Synchronization primitives that can be placed in files and can have lifetimes beyond that of the creating thread.
- Direct native support for Java's daemon threads: daemon threads are threads that run in the background and have dedicated exit semantics enabling them to terminate independently of the processes that use them. Daemon threads are useful to libraries that need to create threads that are unknown to applications. The Solaris JVM does not utilize direct native support for Java's daemon threads but may do so eventually.

Note: In general, accessing native Solaris features using native methods from a Java application is not recommended. Such usage could make the Java application nonportable because it would not be 100% Pure Java and would be tied to the Solaris platform only.

Comparative Overview of Selected Threading Environments.

PRIVATE Feature	Multithreaded Kernel	User Threads Libraries	Standard POSIX Interface[2]	Architecture	MT Development Tools	Thread-Aware Debugger	Thread Programming Class	Shipping Applications Now
AIX	Yes	Yes	Yes	one to one	Yes	Yes	Yes	Yes
Generic DCE	Varies (1)	Yes	No (Draft 4)	many to one	No	No	Yes	Yes
OSF/DCE	Yes	Yes	No (Draft 6)	many to one	Yes	Yes	Yes	Yes
NT	Yes	Yes	No	one to one	NA	Yes	Yes	Yes
OS/2	No	Yes	No	one to one	No	Yes	Yes	Yes
Solaris	Yes	Yes	Yes	many to many	Yes	Yes	Yes	Yes
HP-UX	Yes	No	No	n/a	No	No	No	No
IRIX	No	No	No	n/a	No	No	No	No

Notes:

No entries for the high-level features indicate that the specified environment is not present, with the exception of the user threads libraries where it indicates "not thread-safe." The threads implementations shown in this table are the most recent versions available on the supporting platforms, except for the HP-UX and IRIX information, which is not quite current.

1. There have been a number of drafts to the POSIX 1003.1c specification. Draft 10 is the only specification to have been endorsed by POSIX as the standard.

2. Depends on the operating system on which DCE is implemented.

NA = Not Available

n/a = Not Applicable

Figure 1

an LWP establishes an exclusive connection between the two. Thread binding is useful to applications that need to maintain strict control over their own concurrency, such as those that require real-time response.

Since most Java applications would not require it, there is no Java API to perform the binding. If required, a Solaris native method call can be made to perform the binding. Therefore, all Java threads are unbound by default. Unbound user-level threads defer control of their concurrency to the threads library, which automatically expands and shrinks the pool of LWPs to meet the demands of the application's unbound threads

The Solaris two-level model delivers unprecedented levels of flexibility for meeting many different programming requirements. Certain programs, such as window programs, demand heavy logical parallelism. Other programs, such as matrix multiplication applications, must map their parallel computation onto the actual number of available processors. The two-level model allows the kernel to accommodate the concurrency demands of all program types without blocking or otherwise restricting thread access to system services.

Java on Solaris

Solaris 2.6 is bundled with Sun's native-threaded Java Virtual Machine 1.1 and includes a Just-In-Time (JIT) compiler. Working in conjunction with the JVM, the JIT Compiler for Solaris recognizes method compilation opportunities and serves to reduce bytecode interpretation overhead.

Because the Solaris JVM uses Solaris native threads, which provide multiprocessor support and true application concurrency to Java applications, Java threads become true operating system threads and provide the following benefits:

- Java threads run in parallel, providing much greater performance for parallelized Java applications on multiprocessor machines.
- Java threads harness true operating system concurrency, providing greater performance for multithreaded Java applications on both multiprocessors and uniprocessors.
- Java applications interoperate with existing multithreaded applications in the Solaris environment.
- The Solaris JVM is fully compatible with the Java Developer Kit 1.1 from JavaSoft and includes a JIT compiler, which substantially increases Java application performance.

The Java on Solaris design uses system resources efficiently as needed. Applications can have thousands of threads with minimal thread-use overhead. Threads execute independently, share process instructions and share data transparently with the other threads in a process. Threads also share most of the operating system state of a process, can open files and permit other threads to read them and allow different processes to synchronize with each other in varying degrees.

Evaluating Multithreading Capabilities with Real-World Application

While benchmarks can be a valuable tool to assess the performance and stability of an application, the true test is in real-world implementation.

The latest in JVM technology for Solaris is currently being evaluated as part of an early access program by several real-world Internet applications, including Vitria's BusinessWare application integration software and Volano's VolanoChat, which allows Web developers and businesses to create easily customized chat rooms for their public Websites and corporate intranets.

As a result, these highly threaded, highly networked, multiuser applications are realizing tremendous gains in Java performance, further demonstrating that the Java on Solaris threading model delivers the best combination of speed, concurrency, functionality and kernel resource utilization.

Because of the flexible manner in which the Solaris Java Virtual Machine maps Java threads to its kernel, there are no predefined limits on the number of threads that can be used per application, according to John Neffenger, CTO, Volano LLC. "We have reason to believe that the new version of the Solaris JavaVirtual Machine coming out from Sun later this summer will be the platform with all the speed, scalability, and stability that Java deserves."

ADVERTISER INDEX

Advertiser		Page
Borland		55
www.borland.com	408 431-1000	
Bristol Technology		59
www.bristol.com	203 438-6969	
Cold Fusion Developer's Journal		48
www.sys-con.com	914 735-1900	
Coriolis		63
www.coriolis.com	800 410-0192	
Data Representations		49
www.datarepresentations.com	888 307-9550	
Draw Computing		23
www.openworlds.com	215 38820390	
InstallShield Software Corporation		15
www.installshield.com	800 374-4353	
KL Group		BC&11
www.klg.com	800 663-4723	
Kuck & Associates, Inc.		21
www.ibm.com	888 524-0101	
Ligos		19
www.ligos.com	415 437-6137	
Mecklermedia		61
www.internet.com	800 500-1959	
MindQ		32
wwwmindq.com	800 646-3008	
Object Matter		42
www.objectmatter.com	305 718-9101	
ObjectShare		17
www.objectshare.com	800 973-4777	
ObjectSpace		67
www.objectspace.com	972 726-4100	
Object Management Group		65
www.omg.org	508 820-4300	
Platinum Technology		16
www.platinum.com/vrcreator	800 291-6509	
JDJ Online		39
www.sys-con.com	914 735-1900	
PreEmptive Solutions		53
www.preemptive.com	216 732-5895	
Progress/Cohn & Godly		37
www.apptivity.com	800 477-6473	
Progress Software		27
www.protospeed.progress.com	800 477-6473	
ProtoView		3
www.protoview.com/java	811 231-8588	
Roguewave		2
www.roguewave.com	800-487-3217	
Silicon Graphics		25
www.cosmo.sgi.com	888 91-COSMO	
Slangsoft		13
www.slangsoft.com	972 3-751-8127	
SunTest		4
www.suntest.com	415 336-2005	
Thought, Inc.		45
www.thought.com	415 836-9199	
Visionary Solutions, Inc.		42
www.visolu.com	215 342-7185	
VRML Developer's Journal		33
www.sys-con.com	914 735-1900	
Zero G. Software		6 & 7
www.zerog.com	415 512-7771	

1/4 Ad

1/4 Ad

ProtoView JSuite
by ProtoView

Plenty of good JavaBeans™ to pick from

by **David Jung**

The world is based on objects. In the world of programming, objects are your friends, especially in an environment like Java. Objects in Java are known as JavaBeans™, or just Beans. After all, you can't make Java without Beans. ProtoView is definitely a company that wants you to use their Beans. They have a collection of JavaBeans that will help you extend your Java development and enhance your applications.

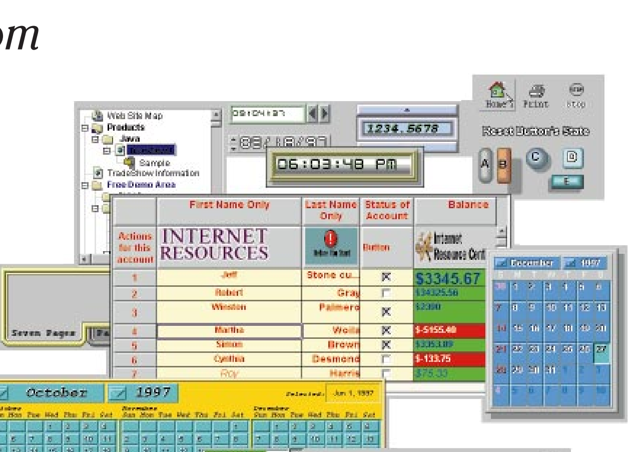
ProtoView is no stranger to the object-based, component market. They also market ActiveX components that can be used with Microsoft Visual Basic, Internet Explorer and any other ActiveX Java Virtual Machine.

With ProtoView's release of JSuite, they have successfully raised the bar on usable JavaBeans. JSuite consists of five main JavaBeans components with over a score of different API functions. These components are CalendarJ, DataTableJ, TabJ, TreeViewJ and WinJ Component Library.

CalendarJ

Have you ever had to write a scheduling program or appointment manager and wanted to include a calendar? Even if you haven't, you've probably had to deal with a date field and would have liked to attach a calendar to a button. The CalendarJ Bean is made up of two different Beans, Calendar and DatePlus components. The Calendar component gives you the ability to display a single month calendar with dropdowns for changing the month and year. It also allows you to display a calendar with three, six, or all twelve months on one form.

The DatePlus component is designed to display and edit a text string that represents a date. Built into the component is the ability to validate the date your user enters in without having to program a date validation routine. For example, if they enter 02 in the month field, the component will not allow you to enter 29, 30, or 31 for the day. (Note: It will allow you to enter 29 assuming the year is a leap year.) Also built into the component is reuse of the Calendar component in the form of a dropdown calendar. When your user clicks on the button, you can display a one, three, six, or twelve-month calendar. When the user selects a date from the calendar, it will display the selected in the DatePlus field.



DataTableJ

Displaying information in a tabular format is very common. This component is more extensible than your average "grid" component. Most grids allow you to add data to the grid, resize rows and columns and probably colors of rows and columns. The DataTableJ allows for in-cell editing, check boxes within the cell, dropdown list boxes, graphics within a cell and more. With all this extensibility, you might think it was a difficult object model to work with. This is not true. It was very easy to add and update information. This version has full JDBC support and dbAnywhere object libraries (Visual Cafe Support) built into the component.

TabJ

Ever since Quattro Pro for Windows, the use of tabs has been very prevalent in Windows applications. This component allows you to easily segment your information into logical groups. The tab placeholders can be placed along any side of the tabs: left, right, top or bottom. You can place any number of tab placeholders on the tab component. If all the tabs can't be viewed on the form at one time, a navigation bar is provided to slide back and forth through the tabs. One caveat is that this component is only available for JDK 1.1 environments.

TreeJ

The tree-type display structure is not a new idea, it's just that Windows 95 made it popular. Normally, most WebArchitects might attempt to construct this functionality through JavaScript. If you like a challenge, go right ahead, but why should you when you have a fine component like TreeJ? TreeJ allows you to create, expand and collapse the structure by pressing the + and – buttons. You can assign any 16x16 pixel icon you like to be displayed on the tree's node. You can edit a node's description either in-line like Windows

ProtoView JSuite
ProtoView Development Corporation
2540 Route 130
Cranbury, NJ 08512
Sales (800) 231-8588
Fax (609) 655-5353
E-Mail info@protoview.com
Web: http://www.protoview.com
$399 ($1,299 With Source Code)

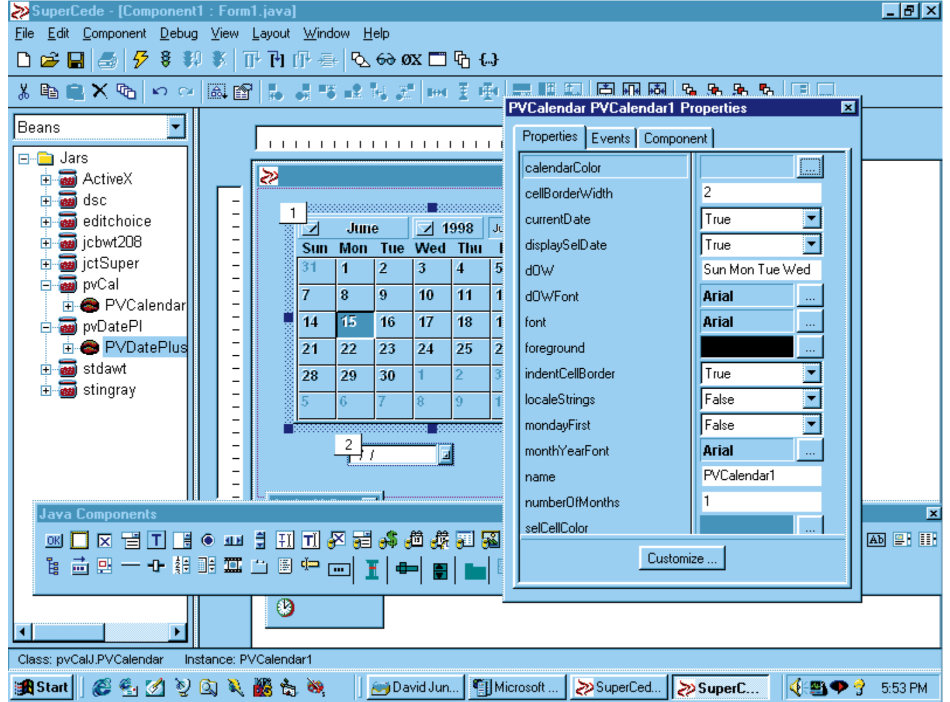


Figure 1

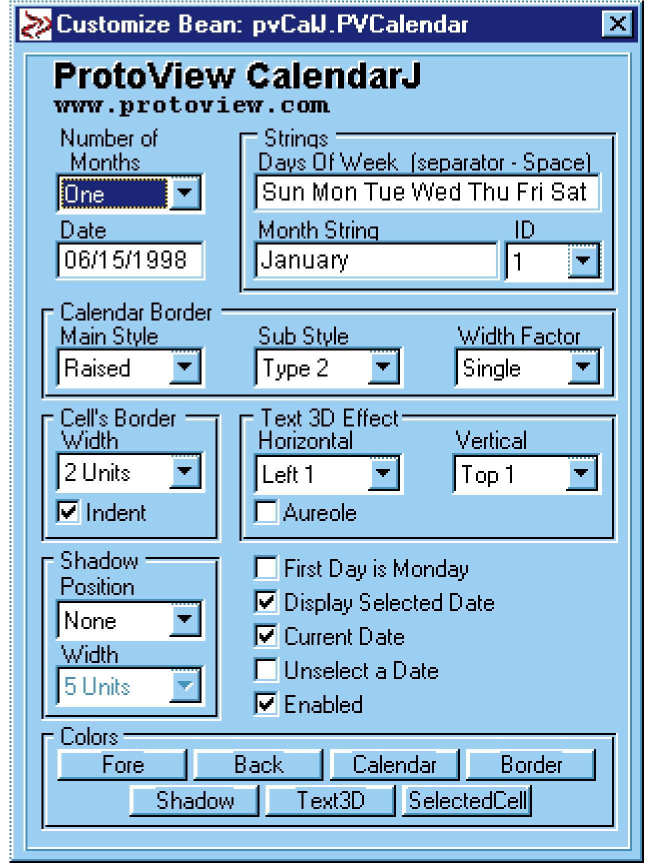


Figure 2

Explorer or through a text box like Windows Registry.

WinJ

The WinJ components consist of 13 different Beans ranging from Buttons to Time display controls. What sets these components apart from other objects that come with your Java development environments? It's their extensive customization. For each component, you can change the component's borders (width, type and bevel), drop shadows around the components, font style and display background.

What's not provided by most Java development environments is the ability to change the looks of most components. WinJ Buttons can look rectangular, circular or square with images. There are components that allow for different types of data entry, like currency, mask edit for Social Security or phone numbers and various date and number entries.

Ease of Use

The object model for these Beans has been thought out very well. The methods and procedures are clearly documented in the HTML-based help files. If your Java development environment is JavaBean-compatible, like SuperCede and Visual Café for Java, the use of these Beans is even easier. You can drag the component you want on to your form and modify the Bean's properties in a custom dialog box. Figure 1 illustrates SuperCede diplaying Properties window of the PVCalendar Bean. When you press the Customize button at the button of the Properties window, you see the JSuite's implementation of the Properties windows, as shown in Figure 2.

Documentation

Don't expect a thick manual when you purchase these components because ProtoView ships the documentation on HTML pages. It is very advantageous to ProtoView because they can provide you with an updated document when there is a change to a component without having to mail out new manuals. This can be frustrating for you, the developer, because HTML documents are great for the Web but can be cumbersome for component documentation.

Finding information in a manual isn't always easy but at least printed information has an index to rely on. HTML documents rely on hypertext linking which doesn't always translate well to development documentation. It would have been a better service to provide that documentation in a PDF (portable document format) file. That way, searching for information can be more accurate and printing out information can be more exact.

Conclusion

As part of an applet, they don't take a long time to initialize and start over a standard 28.8 modem connection. The best part is they are written in "100% Pure Java". This is an important issue to some, but a bigger concern is whether they will run in both Navigator/Communicator and Internet Explorer and they do. As the JavaBean market starts to grow, it's anyone's guess when it will grow as exponentially as the Visual Basic component market. In the here and now, the JavaBeans found in the JSuite tool set are definitely components to be looked into. They are very rich in features and have a lot to offer any Java developer. Whether you purchase the JavaBeans separately as needed or purchase them as the suite, you won't be disappointed.

About the Author

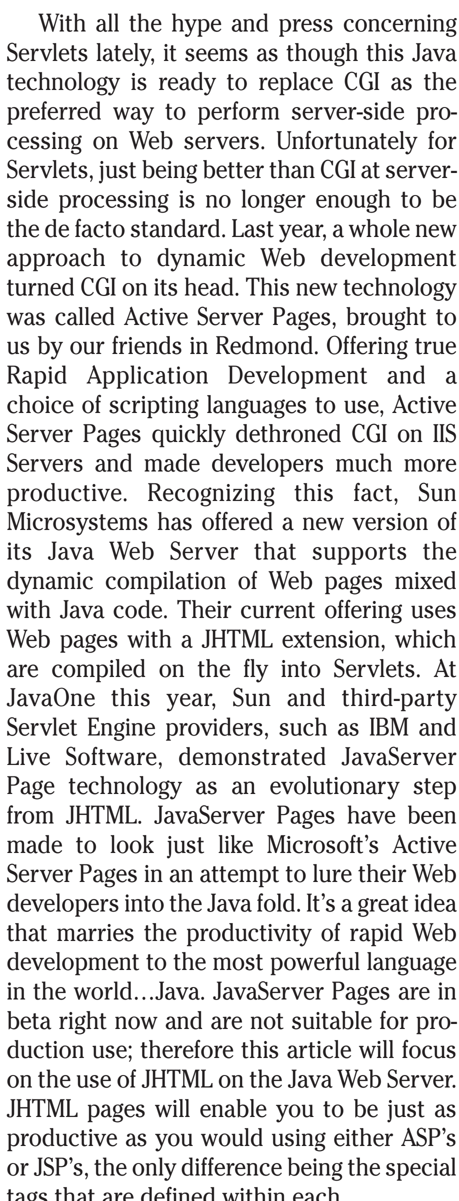
David Jung is a senior programmer analyst for a national medical center in Southern California. He is a key architect for all client/server development for the organization. He is also co-author of several Visual Basic books, including "Visual Basic 5 Client/Server How-To" and "Visual Basic 5 Superbible Set" (Waite Group Press). David can be reached at davidj@vb2java.com.

davidj@vb2java.com

Dynamic Page Compilation with the Java Web Server

Java Replacing CGI to perform server-side processing on Web

by **Robert Tiffany**



With all the hype and press concerning Servlets lately, it seems as though this Java technology is ready to replace CGI as the preferred way to perform server-side processing on Web servers. Unfortunately for Servlets, just being better than CGI at server-side processing is no longer enough to be the de facto standard. Last year, a whole new approach to dynamic Web development turned CGI on its head. This new technology was called Active Server Pages, brought to us by our friends in Redmond. Offering true Rapid Application Development and a choice of scripting languages to use, Active Server Pages quickly dethroned CGI on IIS Servers and made developers much more productive. Recognizing this fact, Sun Microsystems has offered a new version of its Java Web Server that supports the dynamic compilation of Web pages mixed with Java code. Their current offering uses Web pages with a JHTML extension, which are compiled on the fly into Servlets. At JavaOne this year, Sun and third-party Servlet Engine providers, such as IBM and Live Software, demonstrated JavaServer Page technology as an evolutionary step from JHTML. JavaServer Pages have been made to look just like Microsoft's Active Server Pages in an attempt to lure their Web developers into the Java fold. It's a great idea that marries the productivity of rapid Web development to the most powerful language in the world…Java. JavaServer Pages are in beta right now and are not suitable for production use; therefore this article will focus on the use of JHTML on the Java Web Server. JHTML pages will enable you to be just as productive as you would using either ASP's or JSP's, the only difference being the special tags that are defined within each.

Because a Java development tool is not required, the Java Web Server and your favorite HTML editor are all you'll need to get started. If you don't own the Java Web Server, you can download a 30-day trial version at http://java.sun.com/javastore/jserv/buy_try.html. To begin, create a basic Web page in the public_html directory of the Java Web Server and save it with the extension .jhtml. You will not be able to view this JHTML page by double-clicking on it; you must first open your browser and then type in the proper URL to view it. You'll notice that whenever you make changes to your JHTML page that it will take about five seconds for it to open in your Web browser. This is because the Java Web Server compiles your JHTML page into a Servlet every time you make a change to it. The good thing to know is that unlike Active Server Pages that are interpreted every time they're requested, your JHTML-created Servlet is compiled and then stays resident in memory for fast execution.

The first topic covered in this article is the new Java Tags that you'll mix in with your HTML Tags. The most commonly used tag is the <java> tag. This tag tells the Java Web Server that it will find Java code to compile in between <java> and </java>. If you want to insert Java code inside the structure of an HTML tag, you will put your code between two ticks ` ` instead of two <java> tags. To print the value of your Java code, you would insert that code inside the <java type=print> </java> tags. To import Java classes into your Web page, you will first use the <java type=import> tag followed by the classes you wish to import. A number of other Java tags are available, but the ones I've just covered are the only ones we'll need to proceed.

There are a few things that I recommend you put at the top of your JHTML pages to ensure that you never have any problems utilizing all the features that the Servlet API provides. For starters, add the code in Listing 1 to the top of your page. Then add the code in Listing 2 to see an example of using a Java loop to dynamically build a table so that you can get a feel for how your code should look in the context of a Web page.

The next topic we'll discuss is the use of cookies. JHTML pages can send and receive cookies to a user's browser to store information. The code in Listing 3 demonstrates their use. You've probably noticed the use of the code "out.print();" in Listing 3. This code is used to send data back to the browser from inside two <java> </java> tags.

Cookies are often utilized to track sessions or maintain state. An easier way to do this is to use the Session Tracking feature that is built into the Java Web Server. Session variables use either cookies or URL rewriting to store your Java variables throughout the lifetime of a particular session with a particular user. Using Session variables eliminates the need for the manual use of cookies to maintain variables and state. To use Session variables on a particular page, you must first include this line of code:

```
<java>
HttpSession session =
request.getSession(true);
</java>
```

With this code added, you can add, retrieve and remove Session variables using the code shown in Listing 4.

Another way that data is maintained between Web pages or sent to the server is through the use of form submissions and QueryStrings. With JHTML pages, you can receive data that is sent from another page using the getParameter() method. Listing 5 shows the code to retrieve submitted data.

Many times you may find yourself submitting data to another JHTML page whose sole purpose is to take that data and insert it into a database. Usually, after doing such a database operation, you will want to redirect the user to another Web page. This is accomplished using the sendRedirect() method. An important thing to remember is that you can use it only before you send any data back to the Web browser; once you've sent out any kind of data, the sendRedirect method will fail. Below is the code required to redirect a Web page:

```
<java>request.sendRedirect("index.html")</java>
```

An important thing to note when using this feature is that the closing </java> tag

sends out a CRLF to the browser, which is enough to make your redirect code fail. Listing 6 is an example of code that will fail. Listing 7 shows you how to make the redirect code work when you have </java> tags above it.

The next thing to discuss is how to reference core and third-party classes from a JHTML page. Accessing core Java classes from your JHTML page is straightforward and requires no special settings. You've already seen the code to import core classes in Listing 1. Accessing third-party classes, as well as classes you've built yourself, is a little tricky and requires a few extra steps. The first thing you must do is create a "classes" directory off your Java Web Server directory: C:\JavaWebServer1.1\classes. A CLASSPATH is not required, since the Java Web Server picks up the "classes" directory on startup. Subdirectories of your "classes" directory will mirror your different Java Packages. The Java Web Server requires that all of your classes be inside a package. The SendMail class that will be used in the next example references a package called smtp. A proper directory path enabling this class to work would be C:\JavaWebServer1.1\classes\smtp\SendMail.class. To reference this class in your JHTML file, you must first import it at the top of your page just as you do with core classes. In the case of the SendMail class residing in the smtp package, your code will look like Listing 8.

Finally, because nearly everyone uses databases with their Web pages, Listing 9 illustrates an example of JHTML database access. This article should provide you with enough information to get started on your own with JHTML.

About the Author

Robert Tiffany is a senior technology consultant with Insource Technology in Houston, TX. He is currently working on an e-commerce Website using servlet technology for the George Bush Presidential Library. Robert has worked on a lot of Internet/intranet/extranet development with both Active Server Pages and Enterprise Java technologies. Past employers have included Boeing, Microsoft and Real Time Data. You can reach Robert at robertt@insource.com.

robertt@insource.com

Listing 1.

```
<java type=import>
java.io.*
javax.servlet.*
javax.servlet.http.*
</java>
```

Listing 2.

```
<java type=import>
java.io.*
javax.servlet.*
javax.servlet.http.*
</java>

<html>
<head>
<title>Java Loop</title>
</head>
<body>

<table>
<java> for (int I = 0; I < 5; I++) { </java>
 <tr>
  <td>
   <java type=print> "<b>" + I + "</b>" </java>
  </td>
 </tr>
<java> } </java>
</table>

</body>
</html>
```

Listing 3.

```
<java type=import>
java.io.*
javax.servlet.*
javax.servlet.http.*
</java>

<html>
<head>
<title>Java Cookies</title>
</head>
<body>

<java>
//sending cookies
Cookie outCookie = new Cookie("Name", "Value");
outCookie.setMaxAge(2 * 24 * 60 * 60);
response.addCookie(outCookie);

//receiving cookies
Cookie inCookie[] = request.getCookies();
for (int I = 0; I < inCookie.length; I++) {
 out.print(inCookie[I].getName());
 out.print(inCookie[I].getValue());
}
</java>

</body>
</html>
```

Listing 4.

```
<java>
session.putValue("Item", "Shirt");  //adds an Item called Shirt
session.getValue("Item") //gets the Item and returns Shirt
session.invalidate //kills this session
if (session.isNew()) {      //checks to see if you're new
 ...do this...;
}
</java>
```

Listing 5.

```
<java>
String x = request.getParameter("Name");
out.print(x);
</java>
```

Listing 6.

```
<java type=import>
java.io.*
java.net.*
</java>

<java>request.sendRedirect("index.html")</java>
```

Listing 7.

```
<java type=import>
java.io.*
java.net.*
</java><java>request.sendRedirect("index.html")</java>
```

Listing 8.

```
<java type=import>
```

```
smtp.SendMail
</java>

<html>
<head>
<title>Send some mail</title>
</head>
<body>

<java>
SendMail x = new SendMail();
x.smtpHost = "mx2.insource.com";
x.smtpFrom = "robertt@insource.com;
x.smtpToEmail = "briant@insource.com;
x.smtpToName = "Brian Towles";
x.smtpSubject = "The Java Web Server can send email";
x.smtpMessage = "Hello World";
String result = x.Send();
out.print(result);
</java>

</body>
</html>
```

Listing 9.

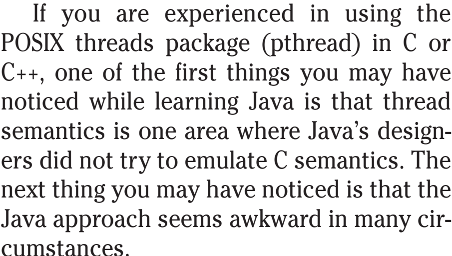
```
<java type=import>
java.sql.*
</java>

<html>
<head>
<title>dbselect</title>
</head>
<body>

<java>
try {
 //load oracle driver
 DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriv-
er());
 //connect to database
Connection cn =
DriverManager.getConnection("jdbc:oracle:thin:@host:1521:orcl",
"scott", "tiger");

 //query
 Statement st = cn.createStatement();
 ResultSet rs = st.executeQuery("select ename from emp");
</java>

<table>
<java> while (rs.next()) { </java>
 <tr>
  <td>
   <java type=print>rs.getString(1)</java>
  </td>
 </tr>
<java> } </java>
</table>
<java>
rs.close();
st.close();
cn.close();
} catch(Exception e){       out.print(e.getMessage());}
</java>
</body>
</html>
```

1/2 Ad

Making Threads Flexible

Using Inner Classes and the Reflection API for Robust Thread Management

by **Philip Rousselle & Mike McNally**



If you are experienced in using the POSIX threads package (pthread) in C or C++, one of the first things you may have noticed while learning Java is that thread semantics is one area where Java's designers did not try to emulate C semantics. The next thing you may have noticed is that the Java approach seems awkward in many circumstances.

For instance, consider the timer() method shown in Listing 1. You might want to add a method like this to a long-running application to reassure yourself that your Java Virtual Machine has not gotten hung. Since this method contains an infinite loop, it only makes sense to run it in its own thread. The POSIX threads model allows you to pass a function pointer and an argument pointer to the pthread_create() method. The specified function is executed in its own thread of control and the argument pointer can be used to pass data structures into or out of the method. If you are used to programming with this threads model, you would expect to be able to invoke the timer() method using a construct like:

```
new Thread(this.timer(5)).start();
```

Unfortunately, Java's semantics do not support this. Instead, to start a thread in Java, your class must implement the Runnable interface. (A similar option, which in the interest of clarity we will not discuss, is to extend the Thread class.) You can then pass an instance of your Runnable class to the Thread constructor. Calling the resulting Thread object's start() method will cause the run() method of your object to be invoked in its own execution thread. The Runnable interface does not allow the run() method to return a result, take any parameters or throw any exceptions. The run() method must also be public to satisfy the Runnable interface. With these semantics in mind, the AppTimer class of Listing 2 shows a naive solution to the problem of invoking the timer() method in a separate thread and passing it a parameter.

The programming approach (or "design pattern") illustrated by Listing 2 is simple. When you have a method you wish to execute in a thread, do the following:

- Change your class definition to include "implements Runnable."
- Add a public run() method to your class that returns void, takes no parameters and throws no exceptions.
- At the place in your code where you want to start the method, put its parameters in instance variables, pass the "this" reference to the Thread constructor and call start() on the new Thread.
- Code your run() method to invoke the target method and pass it the instance variables [or move the code from the target method into run()].

This design pattern does get the job done. That is, the AppTimer class in Listing 2 is successful in running timer(periodLength) in its own thread. In spite of this, the approach is unappealing for several reasons. The main drawback is that it forces you to expose a public run() method that is not really intended for use by clients. Someone looking at Listing 2 could easily be confused about how AppTimer's designer expected the class to be used. If someone mistakenly passes an instance of this class to a Thread constructor and starts it, it will function incorrectly. However, that type of usage is normally inferred by implementing Runnable(). There are also other problems with this approach. It is difficult to use in classes that already expose a public run() method for client use, and it is awkward if a class contains several thread driven-methods.

A more elegant AppTimer implementation, using a design pattern that overcomes these shortcomings, is shown in Listing 3. It defines an inner helper class (TimerHelper) that hides the run() method. An inner class is a class that is defined inside a top-level class as a member of the top-level class. As long as the inner class and its constructors are not public, instances cannot be created outside its defining scope and it is not part of the interface of the top-level class.

Using inner classes to manage threads within a class is an attractive alternative to the approach taken in Listing 2. Not only does it relieve the top-level class of having to implement Runnable and expose a spurious and confusing public run() method, but it also imparts considerable flexibility. The helper class associated with any method can be customized to provide a variety of services such as facilities to stop the method, query its progress or retrieve its result.

The design pattern of Listing 3 generalizes as follows:

- If you have a class, <class>, with a method, <<method>(<signature>)> that you want to run in its own thread of control, create an inner class within <class> named <Method>Helper which implements Runnable.
- Create a constructor for <Method>Helper with the signature <Method>Helper (<class>, <signature>).
- Code the constructor to place its parameters in instance variables, construct a Thread passing the "this" reference and call start() on the new Thread Code <Method>Helper's run() method to pass the parameters saved by the constructor to the top-level class's <method>.
- At the place in the code where you wish to start <method>, construct a new <Method>Helper(this,<parameters>).

While this approach is a considerable improvement compared to trying to place all of a top-level class's thread management logic in its run() method, it still has some drawbacks. The main problem is that it may lead to top-level classes that are littered with inner helper classes. Since all the helper classes have similar logic, the code can become tedious to develop, read and maintain.

Fortunately, you can use the capabilities provided by Java's Reflection API (introduced in JDK 1.1) to implement a single class that performs all the thread management functions of this design pattern. The

Reflection API allows a method name and signature (represented as a String and an array of Class objects) to be passed among methods. Code receiving such information can determine, at runtime, if a particular class has a method with the specified name and signature, and invoke it if so. These facilities are used in the ThreadHelper class shown in Listing 4 and the much simplified AppTimer implementation of Listing 5 to launch a thread into the timer() method.

The ThreadHelper constructor receives a reference to an Object that contains a method to be executed in an independent thread, the name of the target method and an Object array containing the parameters to be sent to the method. By calling the getClass() method on each parameter [getClass() is inherited by all classes from Object], ThreadHelper builds an array of Class objects that identifies the signature of the target method. Calling getDeclaredMethod() on the target object's class and passing the target method's name and signature returns the Method object for the target method. The Method object is passed to ThreadHelper's run() method, along with its parameters, in instance variables. When ThreadHelper passes its "this" reference to the Thread constructor and calls start() on the new Thread, its run() method is started in a new thread of control. The run() method retrieves the target method's Method object and parameter and invokes the target method.

Because ThreadHelper is intended for general use, it should provide more function than the specialized TimerHelper of Listing 3. Specifically, it needs to allow its client to retrieve any result that might be returned by the target method and to allow the client to (optionally) block and await the completion of the method. These capabilities are provided by ThreadHelper's join() method. This join() method is equivalent to the join() function provided in C by the POSIX pthread package. The boolean completed variable is used by the join() function as a synchronization semaphore. If completed has not been set, join() uses the wait() function to block. When the target method returns, run() calls signalCompletion() to set the semaphore and use notifyAll() to wake up any threads that might be waiting in join(). These actions should be performed only in a "synchronized" method, which is why they are not simply performed in run().

"Not only does using inner classes to manage threads relieve the top-level class of having to implement Runnable and expose a spurious and confusing public run() method, but it also imparts considerable flexibility"

It is also possible, of course, to add a variety of useful methods to ThreadHelper. Some functions like isComplete() or kill() are trivial to add. Others like setTimeOut() are more difficult. The ThreadHelper class shown in Listing 4 illustrates the basic concepts involved in implementing a general helper class to encapsulate thread management primitives. This technique can be used to relieve other classes from having to implement Runnable or use inner classes to exploit Java's powerful multithreading capabilities.

ThreadHelper does have one subtle but significant limitation. The parameters passed to methods by way of the ThreadHelper constructor must be classes rather than Java primitive types. That is, the timer() method of Listing 1 must be modified to take an input parameter of type Integer rather than int. This allows the version of AppTimer shown in Listing 5 to pass the input parameter as "new Integer(periodLength)" rather than simply periodLength. This limitation comes from ThreadHelper calling getClass() for each item in the parameter array. Primitive types (like int) don't inherit from Object and therefore do not have a getClass() method.

The ThreadHelper class is more complex than TimerHelper (see Listing 3), and if you rarely use thread-driven logic, then occasionally incorporating it into specialized inner classes may be adequate. However, the Reflection API makes it possible to provide a general solution with only slightly more effort.

About the Authors

Philip Rousselle develops systems and network management software for Tivoli Systems. He is currently designing techniques for efficiently communicating event data (alerts) over large distributed systems. He is also investigating how network topology information can be used to enhance systems management software. He can be reached at philr@tivoli.com.

Mike McNally is the lead designer of distributed monitoring products for Tivoli Systems in Austin, TX. He can be reached at m5@tivoli.com.

philr@tivoli.com m5@tivoli.com

Listing 1.

```
void timer(int interval) {

  long startTime = System.currentTimeMillis();

    while(true) {
      try {
            Thread.sleep(interval * 1000);
          }
         catch(InterruptedException ex) { }
               System.out.println(
                 "The application has been running for "
                 + (int) ((System.currentTimeMillis()
                         - startTime) / 1000) + " seconds");
         }
}
```

Listing 2.

```
public class AppTimer implements Runnable {
```

```
  int periodLength;

  public AppTimer(int newPeriodLength) {

    periodLength = newPeriodLength;
    Thread timer = new Thread(this);
    timer.setDaemon(true);
    timer.start();
  }

  public void run() { timer(periodLength); }

  // void timer(int) see Listing One
```

Listing 3.

```
public class AppTimer {

  class TimerHelper implements Runnable {

    AppTimer target;
    int      periodLength;

    TimerHelper(AppTimer newTarget,
                int newPeriodLength) {

      target = newTarget;
      periodLength = newPeriodLength;
      Thread thread = new Thread(this);
      thread.setDaemon(true);
      thread.start();
    }

    public void run() {

      target.timer(periodLength);
  }

  public AppTimer(int periodLength) {

    TimerHelper helper =
             new TimerHelper(this, periodLength);
  }

  // void timer(int) see Listing 1
}
```

Listing 4.

```
import java.lang.reflect.*;

public class ThreadHelper implements Runnable {

  Object   targetObject;
  Method   targetMethod;
  Object[] parameters;
  Thread   thread;
  Object   result;
  boolean  completed = false;

  public ThreadHelper(Object   newTarget,
                      String   methodName,
                      Object[] newParameters)
    throws java.lang.NoSuchMethodException {

    targetObject          = newTarget;
    parameters            = newParameters;
    Class[] parameterTypes =
                            new Class[parameters.length];

    for (int i = 0; i < parameters.length; i++)
        parameterTypes[i] =
                        parameters[i].getClass();

    targetMethod = targetObject.getClass().
                     getDeclaredMethod(
                     methodName, parameterTypes);

    thread = new Thread(this);
    thread.setDaemon(true);
    thread.start();
  }

  public void run() {

    try {
          result = targetMethod.invoke(
                       targetObject, parameters);
        }
    catch(InvocationTargetException ex) {
                              System.err.println(ex);
    }
    catch(IllegalAccessException ex) {
                              System.err.println(ex);
    }
    signalCompletion();
  }

  synchronized void signalCompletion() {

    completed = true;
    notifyAll();
  }

  public synchronized Object join() {

    while (!completed) {
              try
              {
                wait();
              }
              catch(InterruptedException ex) { }
    }
    return result;
  }
}
```

Listing 5.

```
public class AppTimer {

  public AppTimer(int periodLength) {

    Object[] parm = { new Integer(periodLength) };
    try {
        new ThreadHelper(this, "timer", parm);
    }
    catch(java.lang.NoSuchMethodException ex) {
        System.err.println(ex);
    }
  }

  // void timer(Integer) see Listing 1
}
```

Widget-izing Java's Graphical User Interface Components

Using Java Interface Construct to Enforce Uniformity

by **Daniel Dee**

What Is a Widget?

A widget is a reusable graphical user-interface (GUI) component that operates synergistically with callbacks (a mechanism by which a user's action on a software application's GUI is connected to the code implementing the application's response to this action. The Implementing Callback article last month showed how the callback mechanism can be implemented in Java and how standard AWT components can be extended to support it. Like callback, widget is a familiar concept to X Toolkit and Motif programmers. This follow-up article will introduce the reader to the Widget interface class that helps in the implementation of the widget extension to AWT or other GUI components.

Advantages of Using the Widget Interface Class

The Implementing Callback article showed how the use of the callback mechanism allows the programmer to separate the application's GUI code from the code implementing the application's response to any user interaction with the GUI. To make use of Callbackable and CallbackList classes introduced in that article, GUI component classes have to be widget-ized. However, there's no mechanism in place that requires the widget developer of new GUI components to conform to a standard widget structure. A standard structure has many advantages. In a way, it is a contract between the widget developer and the widget user, in that it guarantees to the widget user the existence of certain widget attributes, such as names. The solution is to use the Java language's interface construct to implement a Widget interface class to enforce the necessary uniformity.

Defining the Widget Interface Class

To support callback, an AWT component where the event originated has to be extended or subclassed to hand off events to the centralized event handler rather than processing the event itself. The programmer supplies the application response code by extending the Callbackable class, and registers this code with the extended AWT component. To simplify the identification of the component that triggers the callback, it is sometimes convenient for the programmer to assign a name to that component that is retrievable at a later time. A Widget should implement at least two methods:

```
public abstract void addCallback(
String callbackName,
Callbackable callback );
public abstract String getName();
```

Listing 1 shows the actual implementation of the Widget interface. Widget naming is implemented as a protected variable and can be assigned as a parameter in the constructor. This can be partially enforced only by adding a constructor with just the name parameter, which in essence calls the default constructor and saves the widget name.

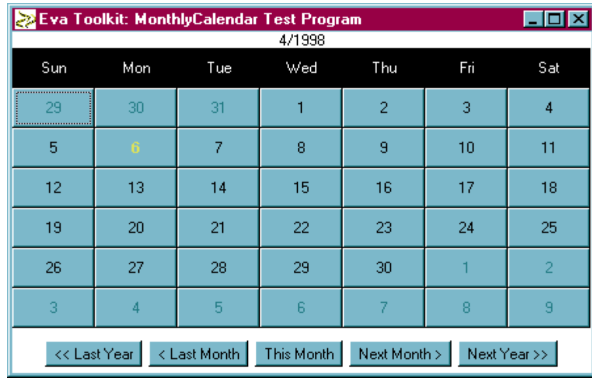


Figure 1: MonthlyCalendarTest program

Implementing the Widget Interface in AWT Button

Listings 2 and 3 show the enhanced Button classes for Java 1.0 and 1.1, respectively. Note that these classes differ from the version presented in the Implementing Callback article only by requiring them to implement the Widget interface. If the programmer inadvertently omits support for widget name or callback registration, the Java compiler flags these errors.

The MonthlyCalendar Class and Building Composite Widgets

Using the Widget interface and widget-ized AWT and other GUI components, it's very easy to build a suite of more complex widgets. As an example, Listing 4 shows the MonthlyCalendar class, which displays a one-month calendar of the specified month and year. Listing 5 shows the MonthlyCalendarTest program. Figure 1 shows how the MonthlyCalendarTest program looks when run. MonthlyCalendar is implemented as a Widget. Note its addCallback method, which essentially passes any registered callback down to the Buttons representing the days of the month. Note also that because event handling is delegated to the callback mechanism, MonthlyCalendar is Java version-independent.

Conclusion

Java's interface language construct makes it easy to enforce uniformity of implementation of widget classes supporting callbacks. By enforcing uniformity, the widget user is guaranteed the existence of certain widget attributes. This, in turn, makes it easy to implement more complex widgets based on standard AWT components. Next month's article will introduce a far more complex widget (a TreeViewer).

Download Source Code

Source code for this article can be downloaded from http://www.sys-con.com. A fully implemented version containing the extended AWT code and a WallCalendar class (built on top of MonthlyCalendar) can also be downloaded for a nominal cost from Wigitek Corporation at www.wigitek.com.

About the Author

Daniel Dee has more than 10 years' experience in the development of GUI software toolkits, using X Windows (versions 10 and 11) and Java. He is currently the president of Wigitek Corporation, a software development and consulting firm. He has an MSEE from the University of the Philippines and an MS in computer system engineering from the University of Massachusetts.

Daniel@wigitek.com

OODBMS & CORBA

Transaction Management in OODB Platforms and CORBA

by **David Knox**

When is a transaction not a transaction? The constant discussion over language objects versus CORBA objects echoes another interesting issue/question – is a CORBA transaction (e.g. a CORBA request within a potentially nested set of OTS-managed transactions) exactly the same as a database object? Does an OTS transaction reflect precisely one database transaction. In this issue David Knox explains why the answer is "not necessarily" with an in-depth discussion of transaction management in object-oriented databases and in CORBA.

Richard Soley
Editor, CORBACORNER
President and Technical Director of the Object Management Group, Inc.

Object-oriented database platforms offer several benefits. The first one I think of is that I don't have to write code to handle the transformation of an object to a row in a table. The object model is the data model. Navigation from reference to reference is efficient because object access is in the OO language itself. How complicated that can get depends on your intent, your design and the OODB platform you're using. Most Java API's for OODBMS platforms are maturing quickly, and there are some interesting variations and parallels forming.

ObjectStore is a physical page-based implementation. This means that when an object is referenced, a physical image of the disk area that the object is stored on is brought into memory. The physical disk address becomes the object's identifier. Other OODBMS platforms, like Versant, use logical models. Instead of bringing a page of disk into memory, they use tree traversal to find the object and bring it into memory. There are good things relative to both paradigms. ObjectStore allows (at the time of this writing) one transaction per session and one session per process.

Versant's product allows multiple sessions per process. Both allow multiple threads in a session. Yet no matter what sort of session model the platform uses, whether it is a physical or logical implementation, or how implicit the transaction boundaries become, we, the programmers, must still be involved in transaction management. For instance, you may want to cause a rollback based on a programmatic exception and return an informative exception to the user/client.

Adding CORBA to the fray makes things more complicated because there needs to be some concept of coordination between CORBA transactions and database transactions. The patterns used to implement the service can make it or break it. Three things to pay close attention to going into the problem are interface granularity, scaling issues and use cases.

The granularity of the CORBA interface can make all the difference in the complexi-

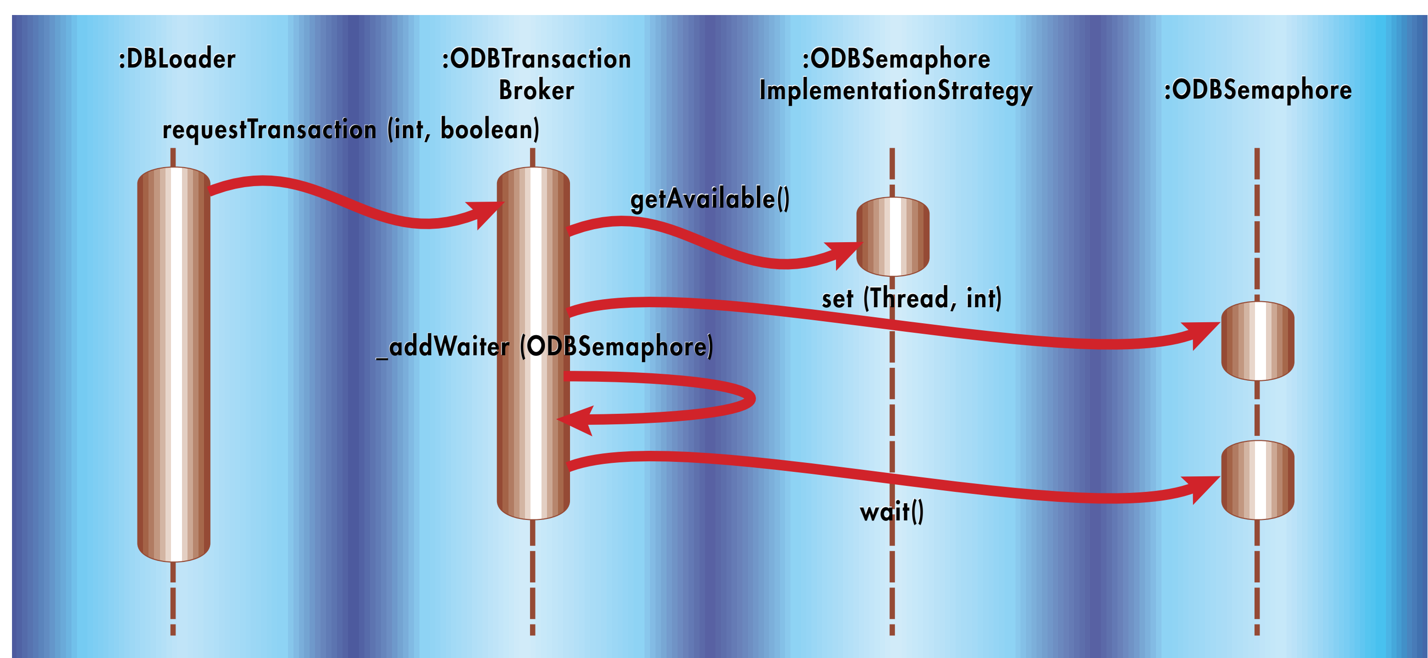


Figure 1: Object model

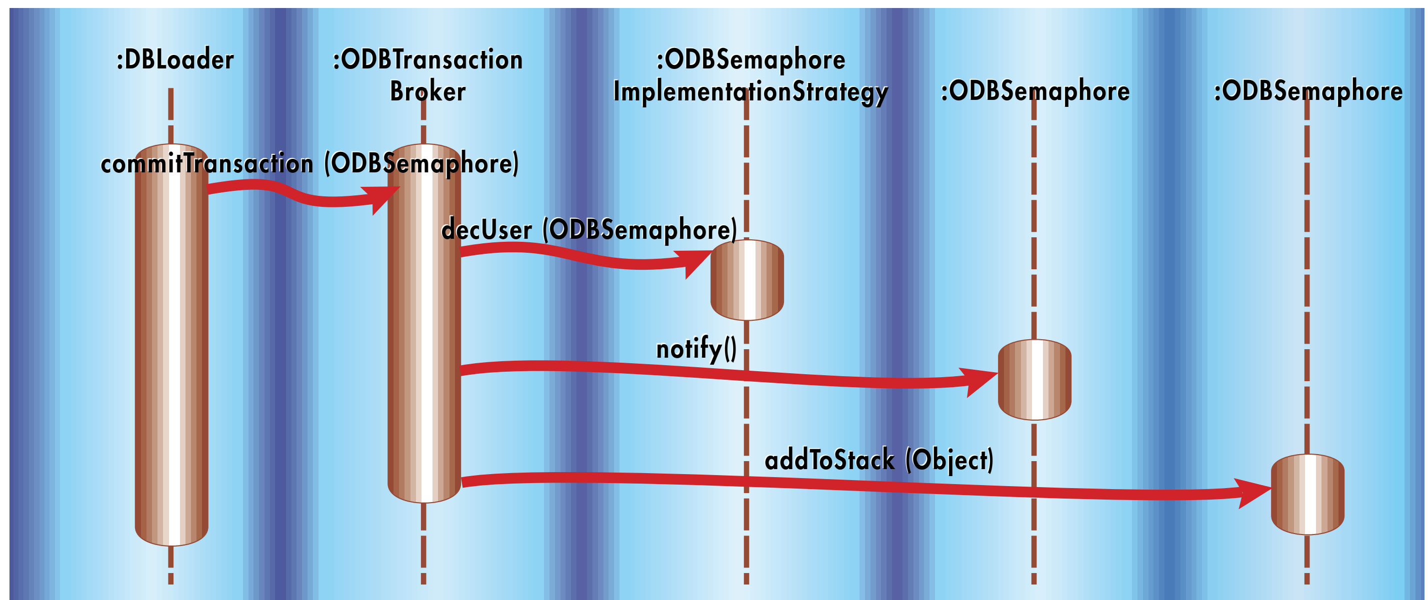


Figure 2: Transaction request message trace

ty of transaction coordination. Some interface implementations are very fine-grained, using accessor and mutator methods defined in the IDL for each data member. Others are more coarsely grained and deliver structures of data to a client. The client manipulates the data in the structure and returns it to the interface.

Scaling issues are a beast in their own right. Some applications achieve scaling by horizontal partitioning using multiple clones of the same CORBA service to distribute load. This is a means to push some of the concurrency issues back into the database engine because it implies having several processes hitting the same database. Others achieve scaling by vertical partitioning in which services are built using a per-method or per-client service. This also implies multiple processes hitting the same database.

It pays to understand your use cases in depth. Some services manage read operations while others are built to manage write operations. This directly addresses both scaling and concurrency versus alleviating concurrency issues as a result of addressing scaling.

It is certain, if you ever implement an OODB with CORBA, you will use some combination of the above paradigms. For instance, you could implement a read-only service that is horizontally partitioned and is coarse-grained, but as you plan your implementation, remember that all affect concurrency. Study your use cases carefully. Let granularity and scaling fall in once your use cases are well understood.

If you have a good conceptualization of the previous issues, then the conceptualization of transaction coordination should follow. A CORBA transaction starts when the client request enters the server's domain, and ends when the reply leaves the server. Should the CORBA transaction and the database transaction be parallel? If your data structure is shallow and your interface is fine-grained, the answer could be yes. This would mean the CORBA implementation is a persistent object as well. But if your data structure has associative depth, then you probably shouldn't make the implementation objects persistent objects and your interface should be coarse grained. The benefits start at performance and continue into maintenance. Consider the implications if the acting CORBA interface is associated with another CORBA interface. If the CORBA transactions and database transactions are parallel, you are potentially stuck with a two-phase commit. Thanks for playing.

There is no necessary relationship between a CORBA transaction and a database transaction. A CORBA transaction could cause no database transaction or it could be the precursor of many. But to relax the relationship is to bring into question how, in pattern, the CORBA object can be decoupled from the persistent object. The answer begins with a concept called "Persistence Aware." ObjectStore implements the idea directly. But conceptually an object that is persistence-aware is an aggregate of persistence-capable objects. It is aware of which objects in its immediate membership are persistent. Control of database transactions begins at this point, one layer behind the CORBA interface.

If you intend to pass persistent data to your clients via a CORBA interface, then CORBA structures are necessary. In general, it is not a good idea to make your CORBA structures persistent. Each persistent object contains one transient CORBA structure. The structure is declared in the persistent object as transient and the persistent object is responsible for moving data elements back and forth before and after a database event. Most vendors provide hooks (Versant will by the end of the year) in persistence-capable objects that the database engine can call when events, like a fetch or a flush, occur. ObjectStore defines postInitializeContents and preFlushContents methods for every persistence-capable class. Both are there to be overridden. The pattern of containment and the hooks allow easy mapping to occur automatically between CORBA structure and persistent object.

When a client calls into the object via the CORBA interface, the CORBA structure is returned. If the data is fresh, there is potentially no database transaction necessary. The structure is a cache that is refreshed within the boundaries of a database transaction. The database engine will call the preFlush method when the data is being saved, and it will call the postInitialize method when the data is fetched. The data will, in a sense, be transferred back and forth automatically between its transient representation and its persistent representation. The provision of methods guaranteed to be called before and after database events provides a hands-off environment with respect to each transient structure. Responsibility for starting and committing the transaction should be based on the data structure being returned. It can lie in the persistence-aware object or it can be pushed back into the persistence-capable objects. The base heuristic is that the transaction boundary is defined as late as practical. Postponing the transaction could allow you to avoid the transaction altogether. Avoidance

could turn throughput into something nearly nonmeasurable.

The issue of transaction control and management becomes more complicated when the service you're building is multi-threaded. When threads are implemented well, the increase in performance makes the cost of writing the code diminish quickly, especially when using CORBA. With the CORBA layer decoupled from the database layer, the CORBA transactions become relatively easy to implement, synchronization being key. However, the database transactions become a bit more complicated because you've got n threads running though your objects' accessors and mutators. It will always be economical to allow multiple threads into the same transaction. For instance, you could allow read requests into a write transaction. Consequently, you need to know when a transaction is in progress and what sort (Read or Write) of transaction it is. The thing to do is to encapsulate transaction management in a subsystem. Within it there needs to be:

- An object that keeps track of the current transaction state
- An object that queues and brokers the transaction requests
- An object that can open, load and close the database
- Pool management for the semaphores
- A background thread that pulls requests from the broker and processes them. When the thread asks the broker for a transaction, what gets returned is a sort of heavy semaphore. It contains everything that the thread will need, from a database perspective, to do its job.

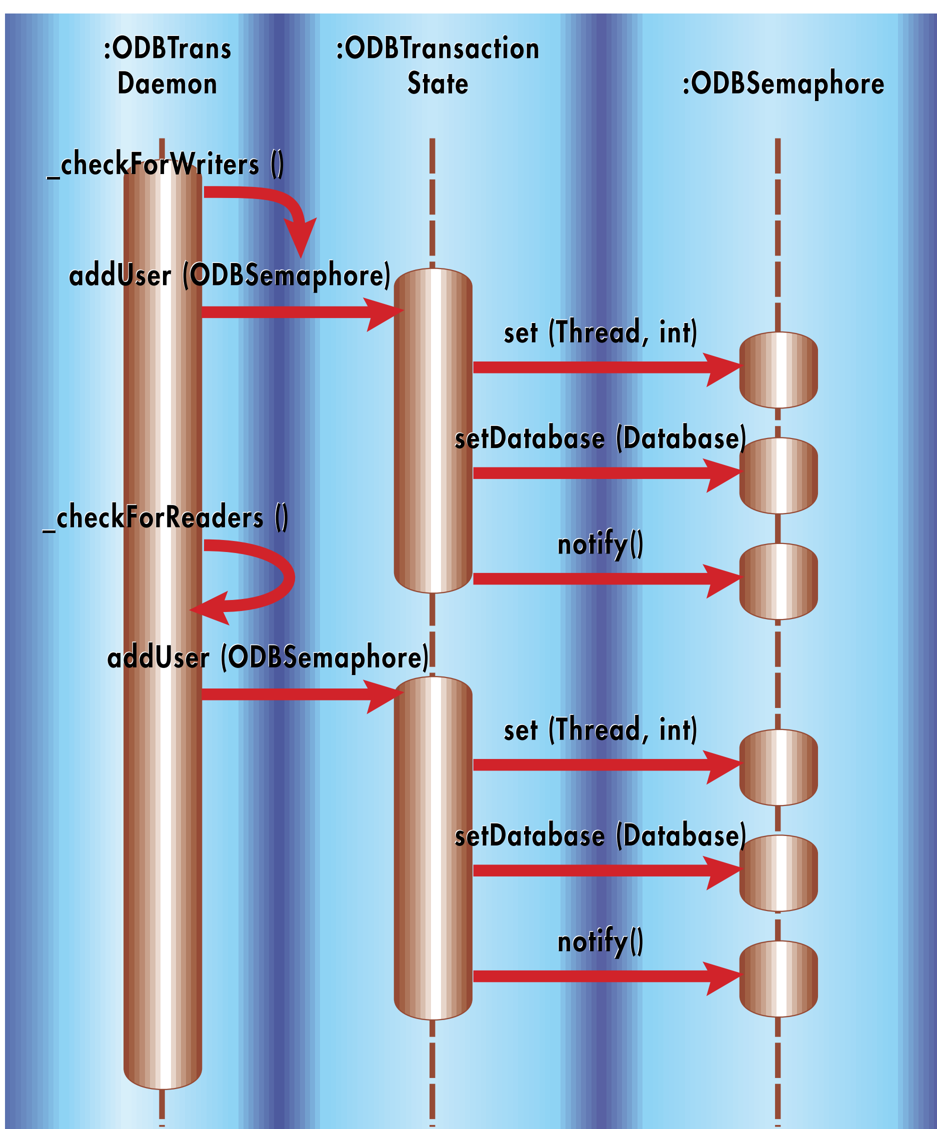


Figure 3: End transaction message trace

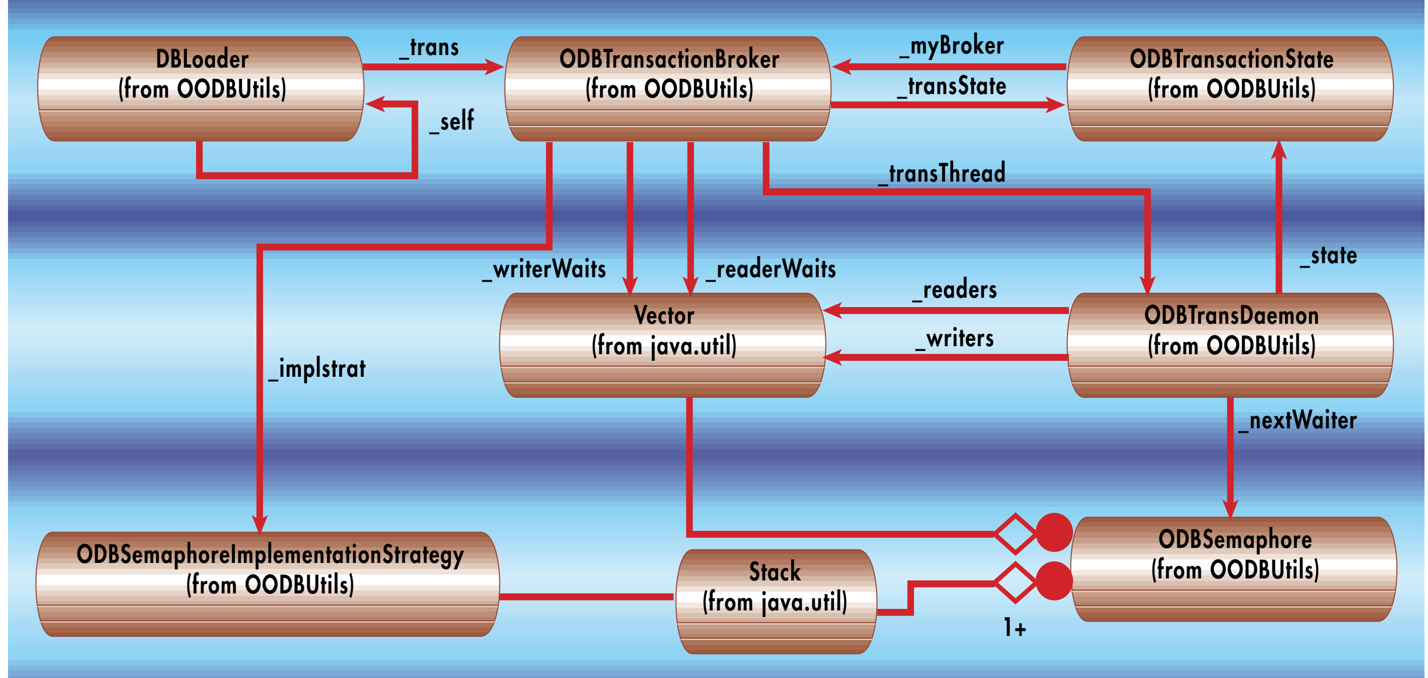


Figure 4: Transaction processing message trace

Object pool management consists of a stack or queue and a strategy to allocate new semaphores to storage in case of fault. Semaphores are not created and destroyed per demand. They are pooled for reuse. The ImplementationStrategy is the manager of the pool.

The DBLoader is responsible for starting things up. It has a static instance() method so it is easily available in the process. It creates the TransactionBroker. All requests for a transaction and subsequent transaction calls come through the DBLoader.

The TransactionBroker is the middle manager. It creates two vectors to keep read and write requests separate. Because there is a break in the operation sequence, there is a hashtable to store exceptions in. If there is an exception while the request is being processed, it is placed in the hashtable. The requesting thread will check the hashtable after its notify() is called from TransactionState. TransactionBroker creates the TransactionState and subsequently the TransDaemon. When a request is received, it is manifested in the form of the ODBSemaphore.

The TransactionBroker requests a semaphore from the ImplementationStrategy and initializes it with a reference to the requesting thread and the type of transaction that is being requested. The TransactionBroker adds the semaphore to the appropriate vector and calls wait () on the semaphore. TransactionBroker also has the job of notifying TransDaemon when a transaction ends.

The TransDaemon is derived from java.lang.Thread. TransDaemon is given references to the request vectors and works both from the front. The current heuristic gives write requests priority over read requests. If it finds nothing in either vector, it yields. If a request is found, then a check is made with the TransactionState to find out if there is already a transaction started. If the system is in a transaction and the new request is a writer, call wait(). Here the Daemon waits to be notified by the TransactionState that the transaction has ended. Under all conditions the logic proceeds to three rules:

1. If the system is not in a transaction state and there is a writer, pass the request to TransactionState to start the transaction. This rule starts a write transaction.
2. If the system is in a transaction state and there is no writer, pass the request to TransactionState to join the transaction.
3. If the system is not in a transaction state and there is no writer, pass the request to TransactionState to start the transaction. This rule starts a read transaction.

The TransactionState is, as one would suspect, a state machine: it approves a request. When the TransDaemon pulls a request from one of the request vectors, eventually the request will go to TransactionState to either join the current transaction or to start a new one. When that happens, the TransactionState notifies the waiting semaphore/request and it completes its trip back through the TransactionBroker.

For general implementation, there are holes in the rules. In a mixed-transaction-type situation there is an apparent probability of starving readers out. One extension I've been considering is how to always allow readers into a write transaction. That way, if there is a steady stream of write requests, the readers can still join in the game. Readers are allowed in until the current writer intends to commit. At that point, transaction management calls for a checkpoint. All waiting readers are queued until the next transaction starts up. It is important that the clients are completely uncoupled from the database and see only CORBA. The database is loosely coupled to CORBA and sees only idl-defined, behaviorless structures. CORBA interfaces are uncoupled from the database and see only in-process calls that return idl-defined

Bristol
1/2 Ad

structures. Another extension entails the invention of a heuristic object that would act as a transaction legislator. Other objects involved in transaction processing would ask the heuristic object what is allowed. Answers come back in the form of true or false.

I've described a pattern that is extensible and flexible. It provides a means to uncouple the dependencies between an OODB platform and CORBA, and places the database functionality in the back of the process. It will be interesting to see how it evolves both in my domain and from the perspective of other developers. Some things, though, are not as flexible. No matter what OODB platform you choose, the first things that must be done by all is to fall into good design habits.

Analyze and understand your use cases. If you can separate reads and writes into different services, do so. It relieves one concurrency factor. Analyze and understand your scaling issues. Derive a sense of how many clients there could be, and interject into the consideration how interactive your interface needs to be. If the service you are building has a high degree of interaction, the number of clients becomes less meaningful. A few clients can cause a great deal of traffic. When you can, design your interface to function in terms of structures versus fine-grained accessors and mutators. It lessens the degree of interaction.

About the Author

David Knox has a BS in Mathematics from Metropolitan State College of Denver. He works for Galileo International, Inc., developers of one of the largest computerized airline reservations systems in the world. David works in Infrastructure and Middleware organization. His responsiblities include Research & Development and the first deployment of CORBA technology. You can reach David at David.Knox@Den.Galileo.com.

David.Knox@Den.Galileo.com

Listing 1.

```
public class MyPersistentObjectExample  {

 // NOTE: this is example code only
  // Don't expect it to compile as is.

  private int _foo;

  private String _bar;

  private transient
        A_CorbaStruct _myCorbaStruct;

  public MyPersistentObject(int foo,
                          String bar) {

      this._foo = foo;

      this._bar = bar;

      _myCorbaStruct = new
              A_CorbaStruct(foo, bar);

  }

  // The transient data is the working data

  public void setFoo(int foo) {

      _myCorbaStruct.foo = foo;

  }
  public void setBar(String bar) {
      _myCorbaStruct.bar = bar;
  }
  // Clients use the transient data
  // it is guaranteed to be current.

  public int getFoo() {
      return(_myCorbaStruct.foo);
  }
  public String getBar() {
      return(_myCorbaStruct.bar);
  }

  public A_CorbaStruct
      getPersistentObjectExampleAttrs() {

      return(_myCorbaStruct);
  }
  public void
      setPersistentObjectExampleAttrs(
              A_CorbaStruct attrs) {
      _myCorbaStruct = attrs;
  }

  public void postInitializeContents() {
      // move persistent contents to tran-
sient
      _myCorbaStruct = new A_CorbaStruct(
                        _foo, _bar);
  }
  public void preFlushContents() {
      // Move transient contents to persistent
      this._foo = _myCorbaStruct.foo;
      this._bar = _myCorbaStruct.bar;
  }

  public void save() {
     // This will force a flush
     this._foo = this._foo;
  }
}
```


SYS-CON PUBLICATIONS

PHONE, ADDRESS & WEB DIRECTORY

CALL FOR SUBSCRIPTIONS

1 800 513-7111

International Subscriptions & Customer Service Inquiries

914 735-1900

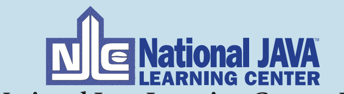
or by fax: 914 735-3922

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

MAIL All Subscription Orders or Customer Service Inquiries to

Java Developer's Journal
http://www.JavaDevelopersJournal.com

VRML Developer's Journal
VRMLJournal.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

NobleNet, Inc., Unveils Nouveau™

(Southboro, MA) - NobleNet, Inc., has announced the launch of Nouveau, the industry's first distributed application development environment designed to automate the generation and integration of diverse CORBA, COM, Java and RPC-based applications.

By supporting C, C++, Java, ActiveX and 4GLs, Nouveau enables IT organizations to experience greater functionality, increased productivity and improved performance without retraining existing staff. Nouveau is currently priced at $5,000, with its first shipment scheduled for August 1998. For more information, visit NobleNet's Website at www.noblenet.com.

KAI Introduces New Multithreaded Java™ Debugging Tool

(San Francisco, CA) - Kuck & Associates, Inc., has announced the arrival of Assure™, a new tool for debugging multithreaded Java applications. Assure is based on breakthrough dynamic analysis technology and enables application developers to identify timing-related problems in their Java source code.

Until recently, developers of enterprise class applications had very few tools available to successfully identify race conditions, deadlocks and other timing-related bugs. Assure finds these bugs automatically; developers can spend more time developing and less time chasing bugs. Assure is easy to use and requires no modification of user source code.

Assure is available directly from KAI for x86 platforms under Windows NT, Windows95 and Solaris operating systems for $495. For more information, contact assurej@kai.com or see KAI's Website at www.kai.com.

Compuware Corporation Announces NuMega DevPartner for Java

(Farmington Hills, MI) - Compuware Corporation has announced NuMega DevPartner for Java, a suite of development tools that automatically detect, diagnose and facilitate resolution of Java performance problems and runtime errors. Using DevPartner for Java, developers can improve the performance, reliability and delivery time of Java applications and components.

The first version of DevPartner for Java will consist of NuMega TrueTime Java Edition and NuMega JCheck. DevPartner for Java will be available in July of 1998. The list price in the U.S. is $599. For more information, see Compuware's Website at www.compuware.com.

Simplicity for Java

(Stamford, CT) - Data Representations, Inc., has announced the general availability of Simplicity for Java™ a rapid application design tool for Java 1.1, which allows developers to build Java applications and applets interactively. Simplicity features:

- The Code Sourcerer™ which interviews the user to determine what should happen in response to events and writes the appropriate Java source code
- An Integrated Design Environment (IDE) that organizes all of the components of a project

Simplicity is able to run on any Java-enabled platform. Its list price is $89. For more information or a free tryout version, check out www.datarepresentations.com.

J Street Mailer™ Release Two Available

(Harrison, NY) - InnoVal Systems Solutions has released a new production version of J Street Mailer, a full-function e-mail client written entirely in Java. J Street Mailer supports both POP3 and IMAP4 mail servers.

The J Street Mailer is currently available for $49. Java Lobby members and students and faculty of accredited institutions will receive a discounted price. Additional information, including a comprehensive list of features and a screen shot, may be found at InnoVal's home page at www.innoval.com.

Riverton Introduces HOW 2.0 for Java

(Cambridge, MA) - Riverton Software Corporation, a leading provider of component-based development technology for developers of business applications, has announced the introduction of HOW 2.0 for Java™. This release offers a component-modeling tool that works with leading Java development environments, including Microsoft Visual J++ and Sybase PowerJ.

HOW 2.0 for Java automatically generates middle-tier business components as JavaBeans™ from developers' applications models. These components can use DCOM or CORBA as their distribution protocol, and can be modified and elaborated in any Java-enabled IDE. The list price for HOW 2.0 begins at $2,995. For more information, see Riverton Software's Website at www.riverton.com.

Rogue Wave Software, Inc., Ships StudioJ

(Boulder, CO) - Rogue Ware Software, Inc. has announced that it's shipping StudioJ, a new suite of JavaBeans components and classes that integrates best-of-breed technology from its Stingray division. In addition to being one of the most complete and flexible sets of components and classes for Java™ technology development, StudioJ includes full source code that gives developers the ability to get "under the hood" and also to add customized extensions.

StudioJ is available from Rogue Wave Software for $995. For more information, call Rogue Wave Software tollfree at 800 487-3217 or visit their Website at www.roguewave.com.

Sales Vision

Novera™ and The Allied Group Partner to Offer Unique Interactive Retail Solutions

(Burlington, MA) - Novera Software, Inc., the leader in server-side Java, has announced that the company has partnered with The Allied Group to create Java kiosk applications for the retail market based on Novera's new jBusiness™ Solutions. This relationship will allow The Allied Group, a developer and integrator of interactive software applications, to offer their customers networked kiosk applications that are easily developed, deployed and managed across the enterprise.

The strength of the product lies in its capability to improve the quality of server-side Java application development, deployment and management across the enterprise. jBusiness facilitates development of intranet, extranet, Internet and platform-independent applications on various hardware architectures, operating systems and disparate object models. For more information, visit Novera's Website at www.novera.com or call tollfree 1-888-NOVERA1.

Object Design Launches Major Upgrade of Pure Java, Pure Object Database

(Burlington, MA) - Object Design, Inc., has announced the availability of ObjectStore® PSE Pro release 2.0 which is designed for portability and performance. ObjectStore PSE Pro database management system (DBMS) maximizes developer productivity for rapid time-to-market and provides high-performance data management for a wide range of Java applications and platforms.

A free 30-day trial version is available from Object Design's Website. The Developer Edition costs $245 and an end-user costs $95. ObjectStore PSE Pro runs on any Java-supported platform (including Windows, Unix, OS/2 and Macintosh).

For more information, visit Object Design at www.objectdesign.com.

Quadbase Systems Inc. ships PulsePoint

(Santa Clara, CA) - Quadbase Systems Inc. has begun shipment of PulsePoint 1.1, a Web-based report/charting tool targeting the business intelligence market. PulsePoint is a query-tool/report-writer that allows users to do adhoc as well as canned querying and reporting using a Java-enabled Web browser.

Some of its features include a graphical query builder, report designer, query/report engine, interactive graphic chart designer/viewer, menu designer, administration module and a Windows-style drag and drop dynamism.

The list price starts at $2,000. For more information, see Quadbase's Website at www.quadbase.com.

An In-Depth Look at Java Security

(Sebastopol, CA) - O'Reilly has announced the release of the book *Java Security* by Scott Oaks, which explores Java security for Java programmers. It includes detailed coverage of:

- Security managers
- Class loaders
- The access controller
- The Java security package
- Message digests, certificates and digital signatures
- The differences between versions 1.1 and 1.2

Java Security helps the programmer master the tools Java provides; programmers deploying software written in Java must know how to grant classes the privileges they need without granting privileges to untrusted classes.

For more information, see O'Reilly's Website at http://java.oreilly.com.

Tadpole Technology Targeting North American Utilities

(Cambridge, England) - Tadpole Technology has announced the appointment of GeoData Solutions, Inc., as reseller for its Java-based geographic information systems (GIS) solution in the North American market.

Tadpole's Java solution, Designated Cartesia Gateway, provides utilities with enterprise-wide access to corporate GIS from any computer fitted with a Java-enabled browser. The applied use of Cartesia as a vehicle to access corporate GIS throughout an organization provides signiciant gains in workforce efficiency, service levels, and cuts in IT budgets.

GeoData Solutions' core operations span project planning, data modeling, migration strategy development and Smallworld GIS training.

Designated Cartesia Gateway offers powerful performance, flexibility, integration and connectivity, platform independence, ease of use and low cost of ownership, and is highly configurable.

For more information, find Tadpole Technology on the Web at www.tadpole.com and GeoData Solutions at www.geodata-gis.com.

Metamata Launches Components for Large-Scale Java™ Development

(Fremont, CA) - Metamata, Inc., has announced the availability of the first four components of its integrated, advanced productivity and quality suite that complements and enhances standard visual development environments (IDE's) for building mission-critical applications in Java. Metamata Browse™ is an intelligent Java source code browser. Metamata Debug™ combines a Java command line interpreter and a fully functional debugger specialized for debugging large-scale applications. Metamata Audit™ is a source code quality analysis tool that evaluates code for programming errors and style issues against standard Java principles and coding practices. Metamata Metrics™ calculates static metric measurements incrementally on different portions of Java source code.

A basic version of Metamata Browse is free; an advanced version is $95. Metamata Debug is $250, Metamata Audit is $395 and Metamata Metrics basic version is $500 while the advanced version is $995.

For more information, visit the company's Website at www.metamata.com.

Avoiding the Web Application Front End Trap

THE GRIND

by **Java George**

A "nice, clean, simple little HTML front end connecting to some tool's proprietary Web application back end?" Think again!

Many developers are discovering that the front end of a Web application can be a dangerous trap. Sure, it seems simple at first: just grab one of the HTML application development tools and knock out a quick front end and connect to the tool's back end. This works well as long as the application remains a simple HTML application that isn't going anywhere.

But, developers are quickly discovering that what begins, say, as a simple little intranet application becomes so popular that suddenly people are clamoring to put it onto the extranet, and even on the Internet where it can be accessed by the general public. As soon as the developer tries to redeploy the application, he or she falls into the front-end trap. The nice, clean, simple little HTML front end connecting to some tool's proprietary Web application back end can't support the functionality required for new deployment options.

Suddenly, you need a lot of functionality that you didn't think you would need when the application was first deployed. This extra functionality calls for Java on the front end, but the simple HTML front end doesn't effectively support Java. You're trapped. The only recourse is to rebuild the entire application.

In other cases, the developers use a Java front end. Often times, the chosen architecture/tools do not support an HTML front end very well, which you still need. In addition, the resulting solution will not be open and/or scalable. In yet other situations, we find that the back-end solution for the Java and the HTML front end are different, which leads to a situation where the developers cannot reuse business objects on the server.

Developers fall into such traps when they don't think ahead and allow for multiple deployment options. The various options—Internet, intranet, extranet—serve different audiences, have different requirements and must reflect the different computing environments. For example, developers have better knowledge of and, possibly, control over the desktop systems that will connect to an intranet application. Similarly, developers can count on intranet and extranet users to have faster network connections than dial-up Internet users. When dealing with intranet/extranet users, the state capabilities and interaction of a Java front end are both feasible and welcome. HTML, on the other hand, has the advantage when dealing with dial-up users on the Internet.

The plurality of end-user interface types will become even more important over time. End-users will want to access anything from anywhere. This means access from the office, the home, a hotel, a taxicab or a plane. They'll be using Windows'98, Mac, PalmPilot, Windows CE, WebTV, etc. Developers can't fall into the trap of having to completely rebuild the front-end application for each deployment scenario.

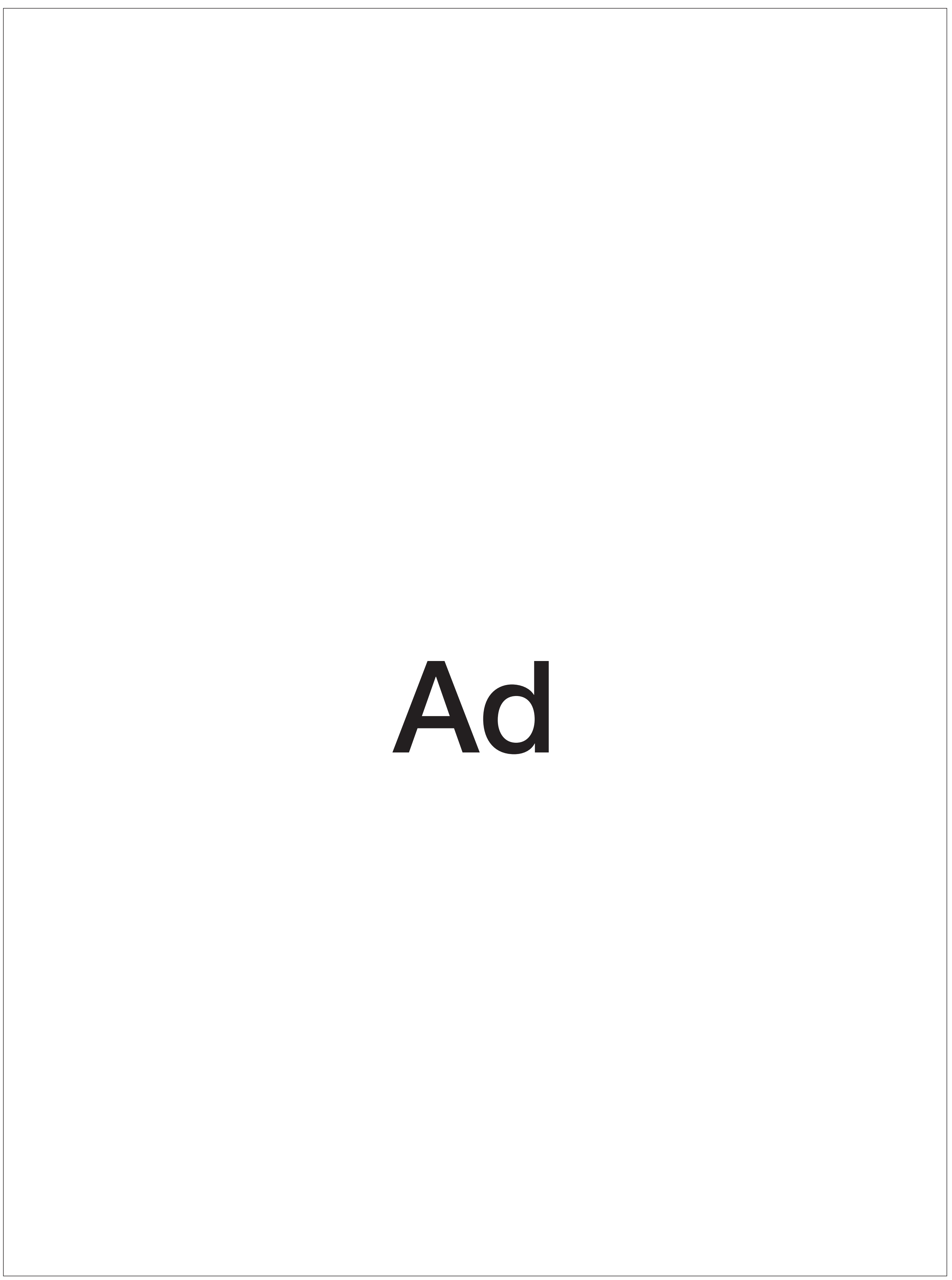
Fortunately, there is a solution. You need to support both Java and HTML front ends and variations of each depending on the case (e.g., DHTML). And the way to do that is to opt for a Java application server built on top of open platforms. Java application servers are extremely well suited for the Java/HTML front end, since a Java front end to a Java application server is an apples to apples connection. In addition, the servlet API with the JDK provides very robust HTML support. Java Server Pages makes it even stronger while remaining an open solution. Developers are praising the servlet API for its support for all popular Web Servers and its extremely effective architecture and design.

My preferred approach is to use Java Application Server with CORBA and open platform services, plus an HTML front end using standard servlet and JSP APIs via HTTP, and a Java front end via IIOP protocol. Using open platform services such as Java Transaction Service (JTS) allows the development team to select a service implementation. Anything else is either not supportive of intra/extra/Internet combinations or based on proprietary architectures that will shut down your options, or both.

In the end, developers cannot simply build the front end of the Web application for the immediate problem at hand. We know that Web applications take on a life of their own. The developer might be building an intranet application this month but within a few months the application may need to be deployed on an extranet or even the Internet. Unless you want to continuously rebuild applications to run with different front ends under different deployment options, you must build front-end flexibility into the application from the start.

Java George is George Kassabgi, director of developer relations for Progress Software's Apptivity Product Unit.

Joe@sys-con.com


Ad

KL Group
Full Page Ad