JOURNAL OF OBJECT TECHNOLOGY

Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 1, January-February 2005

Resource Bundling for Distributed Computing

Douglas Lyon, Fairfield University, Fairfield CT, U.S.A.

Wantonly hacked by an endless stream of nameless,
faceless undergraduates,
both men and women,
often by more than one at the same time,
Kahindu fell into a hell-hole of depravity.

– DL, 1998

Abstract

This paper describes techniques for integrating programs and their resources. The goal is to distribute the programs, to a variety of platforms, without loosing the resources that they need in order to run. Programs so integrated are less fragile than their non-integrated counterparts. The techniques described include the use of a semi-automatic source code synthesizer, XML-based serialization and a base-64 GZIP encoded string format.

The approach is suitable for small data objects (i.e., icons, short audio signals, etc.). It has been used, with good success, on a variety of projects. One drawback of the technique is that an added step is required during program development in order to integrate resources into the code. Another drawback is that integrating resources into the source code can dramatically increase the size of the class files. On the other hand, once the class files are loaded, the resources are available in memory (and hence, quickly accessible).

The techniques described are a part of the Kahindu project, a joint project between the skunk works of DocJava, Inc. and Fairfield University. Kahindu is the name of a village in Kenya known for its fine coffee.

1 INTRODUCTION

One of the basic problems with Java programs is the fragility that results when data is decoupled from source code. For example, suppose that you write a program that seeks to make use of an icon in an interface. In order to load the icon, you write:

Cite as follows: Douglas Lyon: “Resource Bundling for Distributed Computing”, in *Journal of Object Technology*, vol. 4, no. 1, January-February 2005, pp. 45-58.

```
LookAndFeel.makeIcon(getClass(), "icons/ColorIcon.gif"));
```

Now suppose the GIF "ColorIcon" icon file is relocated, relative to the root of the source code. This can easily happen during the process of distribution or development. Even worse, the missing resource is not detected until run-time (perhaps days or even weeks after deployment). For example, if you try to deploy an application that attempts to obtain the above icon, without having it properly integrated into the application, you would throw an exception, like:

```
java.lang.NullPointerException
   at sun.awt.SunToolkit.getImageFromHash(SunToolkit.java:433)
   at sun.awt.SunToolkit.getImage(SunToolkit.java:490)
   at j2d.ImageUtils.getImageResource(ImageUtils.java:873)
   at j2d.ImageUtils.fetchIcon(ImageUtils.java:887)
```

Wouldn't it be nice if the compiler could make sure that our resources were present, before run-time? In this way, we trade off a run-time error for a compile-time error. We present some techniques that allow resources to be integrated directly into source code. The result is a self-contained resource without the normal source of fragility (i.e., source relocation). Hence, we no longer have a program that requires files to be located in particular places on the disk.

The presented technique can be used for any type of resource. Initially, we focus on icons. We present the ability to create icons in one of three ways: by grabbing the icon using a screen grabbing program, by drawing the icon using either the Kahindu drawing tools or another paint program and finally, by entering the icon using teletype graphics. Finally we present an encoding technique that automatically generates Java source code from serializable instances.

Icon Design by Grabbing

There are often icons that are available as a system resource that can be freely copied. These icons start as bit-maps and may be scaled in size to suite the application. The Kahindu program will accept icons of any size. Several applications are able to take snapshots of the computer screen. These applications vary from platform to platform. For example, Silicon Graphics workstations have an application called *Snap*, which will save a snapshot of the screen. On the Mac there is a keyboard shortcut, <shift-Moth-4> (or the *grab* application) which changes the cursor into a cross hair and allows the user to click and drag across the screen.

Depending on the platform, the screen shot will be saved to a file or to a *clipboard*. The clipboard enables the screen shot to be pasted into another application. Unfortunately, Kahindu is rather limited in the number of file types that it currently supports. Thus, a third-party application is required to convert the snapshot into an image that Kahindu supports. Currently, this means GIF, PPM or JPEG files. Suppose, for example, we wanted to grab the icon image from the system to symbolize magnification. Fig. 1-1 shows an image of the magnifier icon. This was grabbed using the screen capture facilities on a Mac.

Fig. 1-1. The Magnifier Icon

Using an application called Debabelizer [Debabelizer], we save the icon image to a GIF file and open it with the Kahindu program.

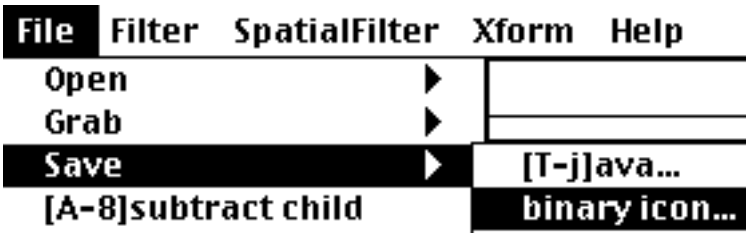


Fig. 1-2. Save As Binary Icon

Fig. 1-2 shows that the Kahindu program has a menu for saving the binary icon image.

```
public static byte iconName[][] = {

{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1},
{1,1,1,0,0,1,1,1,1,0,0,1,1,1,1,1},
{1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1},
{1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1},
{1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1},
{1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1},
{1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1},
{1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1},
{1,1,1,0,0,1,1,1,1,0,0,0,1,1,1,1},
{1,1,1,1,1,0,0,0,0,0,1,0,0,0,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0},
};
```

Fig. 1-3. Binary Icon Output as Java

Fig. 1-3 shows the binary icon output at the console as a two-dimensional static byte array. Such data takes very little space in the program (15x15 = 225 bytes) and its' space is allocated at compile time.

Icon Design by Drawing

Another method for obtaining a binary icon is to draw it. Several excellent paint programs are available that can help with drawing icons (including Kahindu).

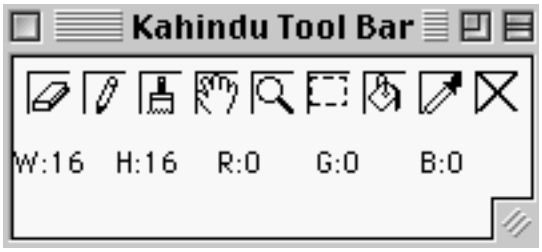


Fig. 1-4. The Kahindu Toolbar

Fig. 1-4 shows an image of the Kahindu toolbar. From left to right, the icons are identified as the eraser, pencil, paintbrush, hand, magnifier, marquee, paint can, eyedropper, and marker. To draw or modify an existing icon, use the eraser, pencil, brush, magnifier and eyedropper. The eraser will clear pixels in the icon. The pencil will set pixels to the value selected with the eyedropper. The brush will set a larger array of pixels than the pencil. Normally, the icons are small. The Kahindu toolbar icons are all 15x15 pixels in size. As a result, the magnifier is used to make the image easier to work on.

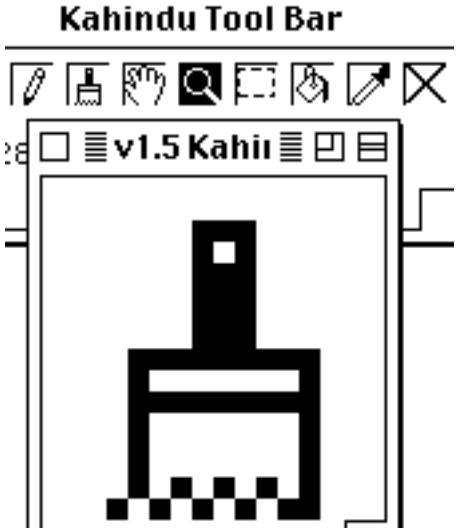


Fig. 1-5. Using the Magnifier to Expand the Paint Brush

Fig. 1-5 shows an example of the use of the magnifier icon to enlarge the paintbrush image. This type of enlargement works by doubling the number of pixels in the icon, and thus growing the icon. To keep the icon the same size, but to enlarge it on the screen, simply resize the frame. Once the frame is enlarged, use the drawing tools to modify the icon.

Fig. 1-6. Elements of One Icon Can Be Used to Create Another

Fig. 1-6 shows how the bristles of the brush were reused to make a face. The design of icons is an art and it takes great care to find icons that have cross-cultural meaning. For example, icons should probably not contain English language characters, as these are not well known in all cultures.

Icon Design by Typing

The programmer can design icons by hand-keying them into the byte array. The *IconFrame* class resides in the Kahindu *gui* package. It contains several icons, some of which were typed by hand. Sometimes this is the easiest way to enter an icon, as it gives fine-grained control over the value and location of each pixel.

```
private static byte xImage[][] = {
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
{0,0,0,1,1,1,1,1,1,1,1,1,1,0,0},
{1,1,0,0,1,1,1,1,1,1,1,1,0,0,1},
{1,1,1,0,0,1,1,1,1,1,1,0,0,1,1},
{1,1,1,1,0,0,1,1,1,1,0,0,1,1,1},
{1,1,1,1,1,0,0,1,1,0,0,1,1,1,1},
{1,1,1,1,1,1,0,0,1,0,0,1,1,1,1},
{1,1,1,1,1,1,1,0,0,0,1,1,1,1,1},
{1,1,1,1,1,1,1,0,0,0,1,1,1,1,1},
{1,1,1,1,1,1,0,0,1,0,0,1,1,1,1},
{1,1,1,1,1,0,0,1,1,0,0,1,1,1,1},
{1,1,1,1,0,0,1,1,1,1,0,0,1,1,1},
{1,1,1,0,0,1,1,1,1,1,1,0,0,1,1},
{1,1,0,0,1,1,1,1,1,1,1,1,0,0,1},
{0,0,0,1,1,1,1,1,1,1,1,1,1,0,0},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
};
```

Fig. 1-7. Hand Typing an Icon

Fig. 1-7 shows the marker icon, as it was hand-encoded into the *IconFrame*. The shift by exactly one pixel during each entry of the zeros in the array, as well as the precise centering of the mark, is simplified by hand entry. In fact, some programs (such as Maple) can output Teletype graphics, like that shown in Fig. 1-7. The Internet is a good source of Teletype graphics, as they are still e-mailed occasionally.

Saving the Icon as Java

Now that you have obtained the Java source code needed for an image, you must assign it a name. This is done in the *IconFrame* when the static array is formulated in the Java code. For example:

```
private static byte pencil[][] = {
```

```
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1},
{1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1},
{1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1},
{1,1,1,1,1,1,1,1,1,0,1,0,0,0,1,1,1},
{1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1},
{1,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1},
{1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1},
{1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1},
{1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1},
{1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1},
{1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1},
{1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1},
{1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1},
{1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,1},
{1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1},
};
```

Once the byte array is formulated, an *IconComponent* is created, using the *getIconComponent* method:

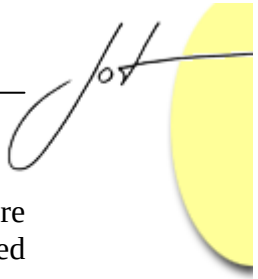
```
package gui;
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import java.util.*;

public class IconFrame
   extends ClosableFrame implements ActionListener {

   private Panel iconPanel = new Panel(new FlowLayout());


   IconComponent eraserIcon =
              getIconComponent(eraser);
   IconComponent pencilIcon =
              getIconComponent(pencil);
```

The *IconComponent*instances are added, using:

```
   private void addIcons() {
         addIcon(eraserIcon,iconPanel);
         addIcon(pencilIcon,iconPanel);
         addIcon(brushIcon,iconPanel);
         addIcon(handIcon, iconPanel);
         addIcon(magnifyingGlassIcon,iconPanel);
```

Once the icons are added to the *iconPanel* instance, they will be available upon program start-up (i.e., without the need to access the file system). For use in Swing, programmers can get an instance of a *javax.swing. Icon* by using:

```
Icon icon = Icons.getMagnifierIcon();
```



Event processing occurs both in the *IconFrame* instance and in the frames that are associated with the icon's state. For example, in the *PaintFrame*, there is a method called *mouseDragged* which checks the *iconFrame* instance for the selected icon:

```
public void mouseDragged(MouseEvent e) {
            e.consume();
      IconComponent ic = iconFrame.getSelectedIcon();
      if (ic == iconFrame.eraserIcon) erasePoint();
      if (ic == iconFrame.brushIcon) brushPoint();
      if (ic == iconFrame.pencilIcon) pencilPoint();
      if (ic == iconFrame.eyeDropperIcon) getColor();

           setP1(e);
           repaint();
}
```

The *IconComponent* instances act just like other components in the AWT, except that they know how to invert their own appearance. They can be added to panels and layout managers can arrange them, just like other components. The *IconComponent* instance keeps a private copy of an image of its own appearance.

2 ON THE XML ENCODING OF SERIALIZABLE OBJECTS

This section shows how an instance of a serializable class can be transformed into a static variable. The static variable is used to reconstruct the instance, at run-time, thereby integrating the resource directly into the source code. Consider the *Address* class, a fragment of which follows:

```
public class Address implements Comparable,
        Serializable {
    private String title = null;
    private String userId = null;
    private String password = null;
    private String firstName = null;
    private String lastName = null;
    private String street = null;
    private String company = null;
    private String address1 = null;
    private String address2 = null;
    private String address3 = null;
    private String homePage = null;
    private String emailAddress = null;
    private String homePhone = null;
    private String businessPhone = null;
    private String faxPhone = null;
    private String city = null;
    private String state = null;
    private String zip = null;
```

An instance of the *Address* converts itself to XML using the *java.beans.XMLEncoder*:

```
public String toXml() {
        ByteArrayOutputStream baos = new
        ByteArrayOutputStream();
        XMLEncoder e = new XMLEncoder(baos);
        e.writeObject(this);
        e.flush();
        return baos.toString();
}
```

The *XMLEncoder* shows:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_03" class="java.beans.XMLDecoder">
 <object class="xml.adbk.Address">
  <void property="address1">
   <string>1313 Mocking bird lane</string>
  </void>
  <void property="city">
   <string>munsterville</string>
  </void>
  <void property="state">
   <string>ny</string>
  </void>
  <void property="zip">
   <string>12181</string>
  </void>
 </object>
```

A static method in a utility class converts the above code into a static string that can be embedded into a Java program:

```
static String Address761092 = "<?xml version=\"1.0\"
       encoding=\"UTF-8\"?> " +
            "<java version=\"1.4.2_03\"
       class=\"java.beans.XMLDecoder\"> " +
            " <object class=\"xml.adbk.Address\"> " +
            "  <void property=\"address1\"> " +
            "   <string>1313 Mocking bird lane</string> " +
            "  </void> " +
            "  <void property=\"city\"> " +
            "   <string>munsterville</string> " +
            "  </void> " +
            "  <void property=\"state\"> " +
            "   <string>ny</string> " +
            "  </void> " +
            "  <void property=\"zip\"> " +
            "   <string>12181</string> " +
            "  </void> " +
            " </object> " +
            "</java>";
```

This can be decoded using:

```
public static Object decodeXml(String xml) {
        ByteArrayInputStream bais = new
ByteArrayInputStream(xml.getBytes());
        XMLDecoder d = new XMLDecoder(bais);
        return d.readObject();
    }
```

Thus, we can test the XMLDecoder using:

```
private static void testXmlDecoder() {
        Object o = decodeXml(Address761092);
        System.out.println("decode shows:" + o);
        Address a = (Address)o;
        a.toXml();
    }
```

It is possible to serialize full-color images using XML, but this is really inelegant. For example, an instance of the *ShortImageBean*:

```
public class ShortImageBean implements Serializable {
    private short r[][];
    private short g[][];
    private short b[][];
…
```

becomes:

```
private static String ShortImageBean8330147 ="<?xml
version=\"1.0\" encoding=\"UTF-8\"?> "+
"<java version=\"1.4.2_03\" class=\"java.beans.XMLDecoder\">
"+
" <object class=\"j2d.ShortImageBean\"> "+
"  <void property=\"b\"> "+
"   <array class=\"[S\" length=\"64\"> "+
"    <void index=\"0\"> "+
"     <array class=\"short\" length=\"64\"/> "+
"    </void> "+
"    <void index=\"1\"> "+
"     <array class=\"short\" length=\"64\"> "+
"      <void index=\"1\"> "+
"       <short>155</short> "+
"      </void> "+…
```

This uses 6 lines for every pixel! Such expansions are too cumbersome even to contemplate. The following section addresses this concern by using a base-64 GZIP encoding technique.

3 ON THE BASE-64 ENCODING OF SERIALIZABLE OBJECTS

This section shows how to GZIP compress, and base-64 encode, an instance of an object, so that it is suitable for storage into a Java string. This is far more efficient (in terms of

memory and CPU time) than creating XML strings. The technique suffers from the fact that the output is not readable by humans and cannot be hand-edited.

The following example turns a full color image into a base-64 GZIP encoded string, then decoded and displayed for view.

```
private static void testBase64GzipEncodeDecoding() {
        ShortImageBean sib = new
       ShortImageBean(NumImage.getImage());
        String s =
                Base64.encodeObject(sib, Base64.GZIP |
       Base64.DONT_BREAK_LINES);
        Object o = Base64.decodeToObject(s);
        ShortImageBean sibTest = (ShortImageBean)o;
        ImageFrame imf = new ImageFrame("sib test");
        imf.setImage(sibTest.getImage());
        imf.setSize(200,200);
        imf.show();
        System.out.println(s);
    }
```

The following code shows how the GZIP base-64 encoder can be used to synthesize a Java static variable declaration:

```
/**
     * Input a serializable object and get back a very
     * long string that is a base64 encoded gzipped version
     * of the serialized object suitable for compilation into
     * a Java program.
     *
     * @param object
     * @return
     */
    public static String getCompactJava(Serializable object) {
        String instanceName = getUnqualifiedClassName(object)
                + object.hashCode();
        String s = "static String " +
                instanceName +
                "=\n\"" +
                Base64.encodeObject(object,
Base64.GZIP|Base64.DONT_BREAK_LINES)
                + "\";";
        return s;
    }
The output is too long to reproduce in full here. An
abbreviated version follows:
static String ShortImageBean12960684=
"H4sIAAAAAAAAAO2a+3NVVxXHF1hrIS8gQKiFqkChgsKUadJ…";
```

The following code example demonstrates how to decode the embedded object:

```
private static void testBase64GzipDecoding() {
```

```
        Object o =
Base64.decodeToObject(ShortImageBean12960684);
        ShortImageBean sibTest = (ShortImageBean)o;
        ImageFrame imf = new ImageFrame("sib test");
        imf.setImage(sibTest.getImage());
        imf.setSize(200,200);
        imf.show();
        System.out.println(ShortImageBean12960684);
    }
```

More generally, we write:

```
public static String encodeImage(Image img) {
        ShortImageBean sib = new ShortImageBean(img);
        return getCompactJava(sib);
    }
public static Image decodeImage(String s) {
        Object o = Base64.decodeToObject(s);
        ShortImageBean sib = (ShortImageBean)o;
        return sib.getImage();
    }
```

One drawback to this technique is that string constants can be too long, causing the java compiler to choke. For example:

```
javac Images.java
Images.java:11: constant string too long
```

Even editing such files can cause some IDEs (like IntelliJ) to bomb out. The question of how to address this problem (perhaps by breaking up the strings) remains open.

As a final example, we present a series of icons used in our new GUI for image processing. These icons are integrated into an *Images* class:

```
public class Images {
    private static String ShortImageBeanExit =

        "H4sIAAAAAAAAAFvzloG1uIhBKMsoRS84I7+oxDM3MT3VKTUx79Jz7
        fNOD1JnMjGwRDMwJpUwMEdHB/swsCfnZBYUpBaVMPD6pJdm6jtDuNZ
        ANemFDHUMjEBGEZhRUVBaBNZUsy52nZC1iykTA0NFAQMDgwBQnCk6+
        H2zHtvT2A2/4MIM3xn+44RODN9LQcaywVWewwpb6KiSAUkNAwGVDHB
        1+FXCVCB04HMnA5I66qgk1nbifUR8KFE7jlpwQHSVTnjgd2Du4AYmd
        D1oQn/8IahS7ff83UwMjFGwzJACTL9gA1nAqRth9Dm8CfocmsprWGE
        LHVUyIKlhIKCSAa4Ov0qYCoQOfO5kQFJHHZXE2k68j4gPJWrHEe4Ej
        aoSX4I+hzWlLsWbUpeiqcTtYnqpxB4H2FVii1fstmNLK7jciZn+KFV
        JrO3E+4j4UKJ2HOFOqagq8aXUpQBz65GonQgAAA==";
    private static String ShortImageBeanOpen =

        "H4sIAAAAAAAAAFvzloG1uIhBKMsoRS84I7+oxDM3MT3VKTUx79Jz7
        fNOD1JnMjGwRDMwJpUwMEdHB/swsCfnZBYUpBaVMPD6pJdm6jtDuNZ
        ANemFDHUMjEBGEZhRUVBaBNZUsy52nZC1iykTA0NFAQMDgwBQnCk6+
        H2zHtvT2A2/4MIM5+AQEwBFS0HGsqGoZGD4j4Zxq0RXh08lqjpcKs9
        hqMOnElUdITOJU0nYTAYizWQg2kxklcSZia7+HA53opt8DqvKc0hyK
        OqwmImq8hxOlajuROEDcwc3MKHrQRP64w9BlWq/5+9mYmCMgmWGFGD
        6BRvIAjIQyehrcIgJgKJYVIICHxkSq5KBYS8elQwo6nCrRJgKUUdYJ
```

```
            Uwd/VQyEKmSgWgzESohgLBKmHokdVjdyYCiEq4bw+8MKCqRbMEwkwH
            FtddwqkR15zUMlaMpdTSlDvqUCgCbGTxOnQgAAA==";
        private static String ShortImageBeanRevert =

            "H4sIAAAAAAAAAFvzloG1uIhBKMsoRS84I7+oxDM3MT3VKTUx79Jz7
            fNOD1JnMjGwRDMwJpUwMEdHB/swsCfnZBYUpBaVMPD6pJdm6jtDuNZ
            ANemFDHUMjEBGEZhRUVBaBNZUsy52nZC1iykTA0NFAQMDgwBQnCk6+
            H2zHtvT2A2/4MIM5/DDUpCxbGgqQQCDh0UlDBBSiQyopZIUdw6USmS
            fIPsLDChUyYBNFw6VCLVwUTwqz6GYjVslcgwRUokGgLmDG5jQ9aAJ/
            fGHoEq13/N3MzEwRsEyQwow/YINZAEZiGT0NfwQq0oQwOBhUQkDhFQ
            iA2qpJMWdA6US2SfI/kKPfnJUMmDThUMlQi1cFI/Kayhm41aJHEOEV
            KKB0ZQ6yFQi+wTZX7jjdYSkVADMCBJtnQgAAA==";
        private static String ShortImageBeanSave =

            "H4sIAAAAAAAAAFvzloG1uIhBKMsoRS84I7+oxDM3MT3VKTUx79Jz7
            fNOD1JnMjGwRDMwJpUwMEdHB/swsCfnZBYUpBaVMPD6pJdm6jtDuNZ
            ANemFDHUMjEBGEZhRUVBaBNZUsy52nZC1iykTA0NFAQMDgwBQnCk6+
            H2zHtvT2A2/4MIM5/DDUpCxbFCVuAHxKhkoUInsslGVMDb5KnEDTDO
            JVUmsmXghMHdwAxO6HjShP/4QVKn2e/5uJgbGKFhmSAGmX7CBLCADk
            Yy+hh+iqMQNMFXuxQEZ8JqJXyWyyxjAKuD2j0CVMDb5KtFDHp/t2GM
            Uv5mo4BqJqW40pQ4XlcM4pQIAdEPpCJ0IAAA=";

        public static Icon getRevertIcon() {
            return new
    ImageIcon(Utils.decodeImage(ShortImageBeanRevert), "revert");
        }

        public static Icon getSaveIcon() {
            return new
    ImageIcon(Utils.decodeImage(ShortImageBeanSave), "save");
        }

        public static Icon getExitIcon() {
            return new
    ImageIcon(Utils.decodeImage(ShortImageBeanExit), "exit");
        }

        public static Icon getOpenIcon() {
            return new
    ImageIcon(Utils.decodeImage(ShortImageBeanOpen), "open");
        }
    }
```

The output of the new interface, with the integrated icons, is shown in Fig. 2-1.

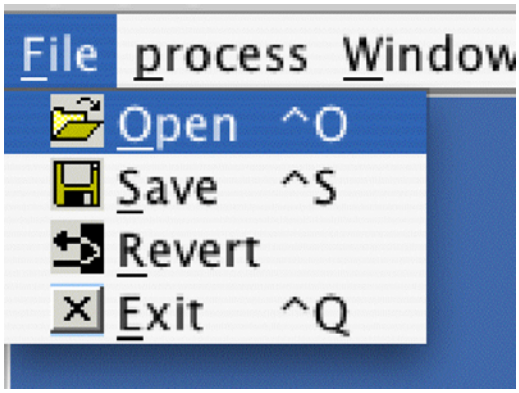


Fig. 2-1. The Application with Integrated Icons

4 CONCLUSION

In summary we look to avoid storing resources in external files in order to speed resource loading and improve reliability of our programs. This is a natural outcome of resource integration with source code.

The idea of storing icons in source code is not new [Lyon], but the idea of storing serialized XML encoded objects in Java source is (as far as I know).

There are definite limitations to the technique of storing objects inside of a Java program. For one, there is a reliance on classes implementing the serializable interface. For example, the *Image* class does not implement the serializable interface. On the other hand, writing ad-hoc techniques for storing images has the benefit of creating a nice structure for hand editing.

It is equally clear that large arrays of data could create a cumbersome burden during compilation. In fact, some compilers will not allow a static array that is larger than 64K bytes. It is generally recognized that this is not a limitation of the Java language specification, but rather of the implementation of the compiler.

Other failures to serialize into Java source can result from using inner classes. For example:

```
private static void testXmlImageEncoding() {
        class Foo implements Serializable {
           int i =10;
        }
        System.out.println(getJava(new Foo()));
    }
```

Cannot work because *Foo* is an inner class.

REFERENCES

[Debabelizer] Debabelizer is a useful program for batch image conversion and processing, available from: Equilibrium, 475 Gate Five Road, #225, Sausalito, CA 94965. Phone: (415)332-4343.

[Harder] Robert W. Harder, "Base64", A Public Domain Java class providing very fast Base64 encoding and decoding in the form of convenience methods and input/output streams. See http://iharder.sourceforge.net/base64/

[Lyon] Douglas A. Lyon, *Image Processing in Java*, Prentice Hall, Upper Saddle River, NJ, 07458. 1999. Available from http://www.docjava.com

About the author

After receiving his Ph.D. from Rensselaer Polytechnic Institute, **Dr. Lyon** worked at AT&T Bell Laboratories. He has also worked for the Jet Propulsion Laboratory at the California Institute of Technology. He is currently the Chairman of the Computer Engineering Department at Fairfield University, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. E-mail Dr. Lyon at Lyon@DocJava.com. His website is http://www.DocJava.com.