Writing an OS Shell in Java

Building a Facade

Andrew S. Downs (andrew@nola.template.com)

This paper describes the design and implementation of a Finder-like shell written in Java. Several custom classes (and their methods) areexamined: these classes provide a subset of the Macintosh Finder's functionality. The example code presents some low-level Java programming techniques, including tracking mouse events, displaying images, and menu and folder display and handling. It also discusses several high-level issues, such as serialization and JAR files.

Introduction

Java provides platform-dependent GUI functionality in the Abstract Windowing Tookit (AWT) package, and platform-independent look and feel in the new Java Foundation Classes (Swing) API. In order to faithfully model the appearance and behavior of one specific platform on another, it is necessary to use a combination of existing and custom Java classes. Certain classes can inherit from the Java API classes, while overriding their appearance and functionality. This paper presents techniques used to create a Java application (Facade) that provides some of the Macintosh Finder's capabilities and appearance. As a Java application, it can theoretically run on any platform supporting the Java 1.1 and JFC APIs. Figure 1 provides an overview of the Facade desktop and its functionality. Figure 2 shows a portion of the underlying object model. It contains a mixture of existing Java classes (such as Component and Window) and numerous custom classes. In the diagram, the user interface classes are separated from the support classes for clarity.

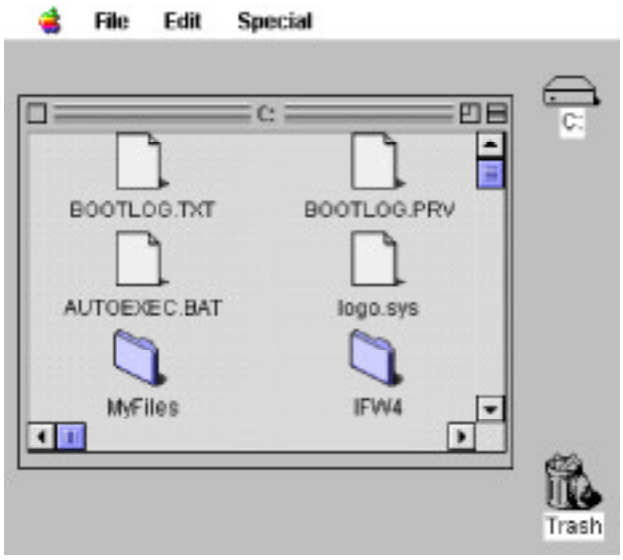

Figure 1. *Facade on Windows 95*

The object model

Facade's object model utilizes the core Java packages and classes where possible. However, in order to provide the appropriate appearance and speed, custom versions of several standard Java classes were added. For instance, the `DesktopFrame` class is used to display folder contents. Its functionality is similar to the `java.awt.Frame` class. However, additional behavior (i.e. window-shade) and appearance requirements dictated the creation of a "knockoff" class.

One advantage of using custom classes can be improved speed. This holds true for

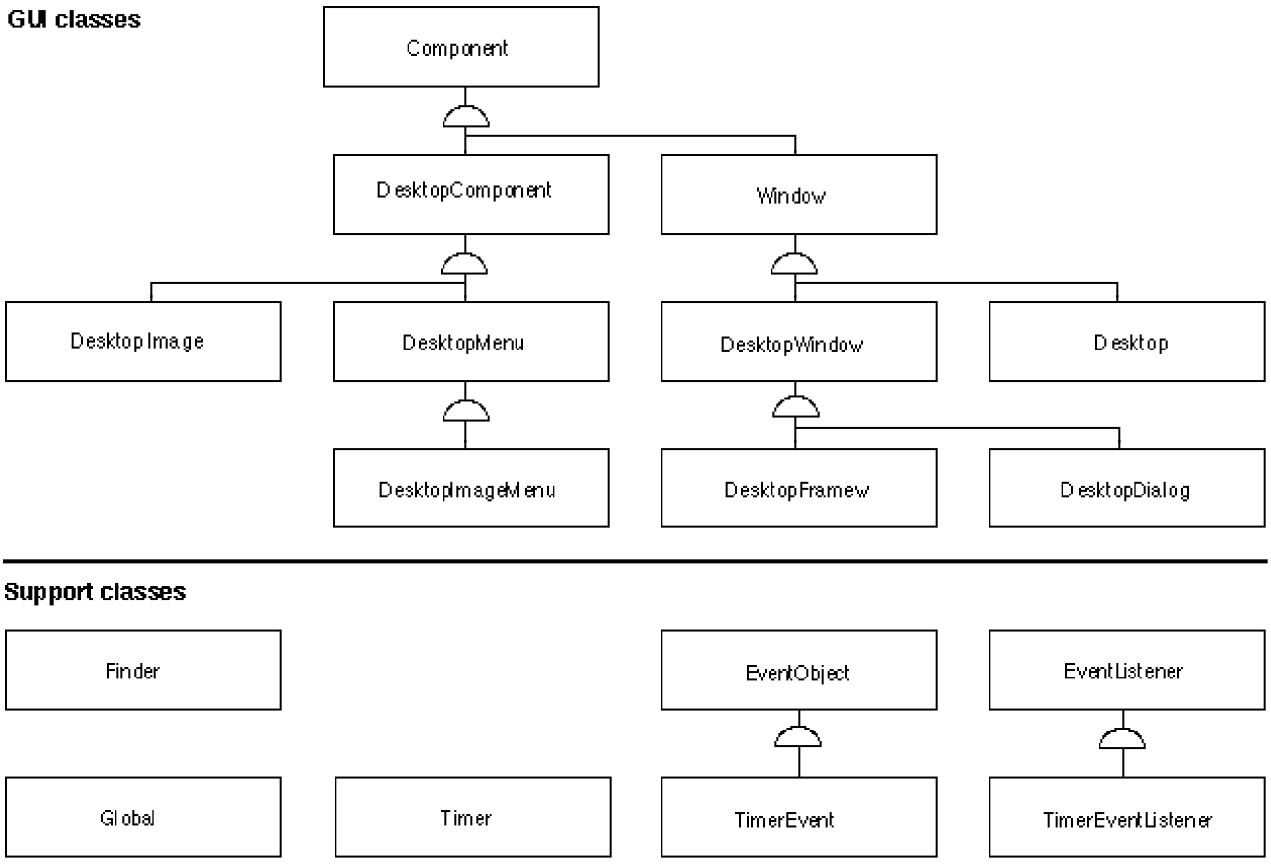

Figure 2. *Facade object model (partial)*
Only the inheritance structure is shown; relational attributes are not shown.

classes inheriting from `java.awt.Component`: such classes do not require the creation of platform-specific "peer" objects at runtime. Less baggage results in a faster response. For example, the Facade menu display and handling classes respond very quickly.

In the custom classes, most drawing uses primitive functions to draw text, fill rectangles, etc. Layout and display customization is assisted through the use of global constant values which typically signify offsets from a particular location (as opposed to hardcoded coordinates).

Several of the classes depicted in the object model will be discussed in this paper.

Appearance and functionality

Facade relies on several graphical user interface classes in order to provide a Finder-like interface. These classes fall into the following categories:

- GUI
  - Desktop
  - Menus
  - Icons
  - Folders
- Support
  - Startup
  - Timing
  - Constants

Facade provides a subset of the following operations:

- Menu and menu item selection
- Displaying volume / folder contents
- Dragging icons
- Opening and closing windows
- Managing the trash
- Saving the desktop state at shutdown (exit)

- Restoring the desktop state at startup
- Launching applications
- Dialog display
- Cut / Copy / Paste
- Changing the cursor
- Emulating the desktop database

Several of these operations (and the classes which implement them) will be discussed in the following sections.

The desktop

The `Desktop` class is a direct descendant of the `java.awt.Window` class, which provides a container with no predefined border (unlike the `java.awt.Frame` class). This approach allows the drawing and positioning of elements within the Desktop area, without needing to hide the platform-specific scrollbars, title bar, etc.

However, this approach means that special care and feeding is required to make the menubar work properly. In the current implementation, menubar (and menu) drawing is done directly in the `Desktop` class, using values obtained from those respective classes. In other words, menus don't draw themselves, `Desktop` does it for them.

Menu creation

The following code snippet (Listing 1), taken from the `Desktop` constructor, creates the Apple menu, and populates it with one item. The menu is an instance of the `DesktopImage` class. The Apple icon (`imgApple`) was loaded further up in this method. A similar sequence creates and populates the File menu.

```
// ---------------------------------------
// • Desktop constructor
// ---------------------------------------

Desktop() {

   // <snip>

   // Create the menus and their menu items

   DesktopImageMenu dim;
   Vector vectorMenuItems;
   DesktopMenuItem dmi;

   // Calculate the y-coordinate of the menus (constant).
   int newY = this.getY() +
     this.getMenuBarHeight() -
     ( this.getMenuBarHeight() / 3 );

   // Calculate the x-coordinate of the menus (variable).
   int newX = this.getX();

   // Create the Apple menu.
   dim = new DesktopImageMenu();
   dim.setImage( imgApple.getImage() );
   dim.setX( imgApple.getX() );
   dim.setY( imgApple.getY() );

   // Create menu item "About..."
   dmi = new DesktopMenuItem(
     Global.menuItemAboutMac );
   dmi.setEnabled( true );

   // Add the menu item to the Vector for this particular menu
   dim.getVector().addElement( dmi );

   // ...then tell the menu to calculate the item position(s).
// Note that the Desktop's FontMetrics object gets used here
   dim.setItemLocations(
     this.getGraphics().getFontMetrics() );

   // Add the Apple menu to the menubar's Vector.
   // The menubar was instantiated further up in this method.
   dmb.getVector().addElement( dim );

   // For the next menu, calculate its x-coordinate.
   newX = imgApple.getX();
   newX += ( 2 * Global.defaultMenuHSep );
   newX += ( (DesktopImageMenu)
     dmb.getVector().elementAt( 0 )
     ).getImage().getWidth( this );
     // Line wrap.

   DesktopMenu dm;

   // Create the File menu...
   dm = new DesktopMenu();
   dm.setLabel( Global.menuFile );
   dm.setX( newX );
   dm.setY( newY );

   // ...and all its menu items.
   dmi = new DesktopMenuItem(
     Global.menuItemNew );
   dmi.setEnabled( true );
   dm.getVector().addElement( dmi );

   dmi = new DesktopMenuItem(
     Global.menuItemOpen );
   dmi.setEnabled( false );
   dm.getVector().addElement( dmi );

   dmi = new DesktopMenuItem(
```

```
      Global.menuItemClose );
    dmi.setEnabled( false );
    dm.getVector().addElement( dmi );

    // Tell the menu to calculate the item position(s).
    dm.setItemLocations(
      this.getGraphics().getFontMetrics() );

    // Add the File menu to the menubar's Vector.
    dmb.getVector().addElement( dm );

    newX += ( 2 * Global.defaultMenuHSep );
    newX += theFontMetrics.stringWidth(
      dm.getLabel() );
  // <etc.>
}
```

Listing 1: *Creating menus in the Desktop constructor.*

Here, the variable `dim` is an instance of the class `DesktopImageMenu`. This specialized menu class simply adds an instance variable that contains an `Image` (in this case, the Apple) to the `DesktopMenu` class. Its behavior is the same as other menus, except that instead of a `String` it displays its `Image`. The `x` and `y` coordinates of this menu are determined by the same values assigned to the image when it was loaded.

This object contains one menu item, the "About" item. That item is enabled, and added to the `Vector` for the menu. This `Vector` contains all the menu items for that menu. This approach provides an easy way to manage and iterate over the menu items.

Once the `Vector` has been setup, its contents are given their x-y coordinates for drawing and selection purposes through the `setItemLocations()` method. Although this calculation can be done when the menu is selected, the code runs faster if those numbers are calculated and assigned at startup time.

Finally, the menu is added to the `Vector` for the menubar. The menubar uses a `Vector` to iterate over its menus, in the same way the menus iterate over their menu items. This will become apparent when examining the mouse-event handling code for the `Desktop` class.

The same approach is shown for the File menu, except that File contains a `String` instead of an `Image`, as well as several menu items.

Menu and menu item selection

Figure 3. *Menu item selection*

The code in Listing 2 handles mouseDown events in the menubar. The first two lines reset the instance variables used to track the currently active menu and menu item. Since the user has pressed the mouse, the old values do not apply. Next, the `Graphics` and corresponding `FontMetrics` objects for the `Desktop` instance are retrieved. They will be used in drawing and determining what selection the user has made.

Next, test to see whether the event occured within the bounding rectangle of the menubar. This line uses the `java.awt.Component.contains()` method. If the x-y coordinates of the mouse event are inside the menubar, then determine which menu (if any) the user selected.

Determining the menu selection uses the menubar's Vector of menus. Iterate over the Vector elements, casting each retrieved element to a DesktopMenu object. (Vector.elementAt() returns Object instances by default, which won't work here.) The menu has its own bounding rectangle, which is compared to the event x-y coordinates. In addition, in order to duplicate the Finder, a fixed pixel amount is subtracted from the left-side of the bounding rectangle, and added to the right-side. This allows the user to select near, but not necessarily directly on, the menu name (or image). Notice also in the (big and nasty) if conditional that the DesktopMenu retrieved from the

Vector is checked for an Image (this handles the Apple menu). If it has one, the image width is used instead of the String width.

Once the user's menu selection has been found, it is redrawn with a blue background. Then, the menu items belonging to that menu are drawn inside a bounding rectangle (black text on white background). The menu item coordinates, and the corresponding bounding rectangle coordinates, were calculated when the menu was created, saving some CPU cycles here.

```
// ------------------------------------------------------
// • mousePressed
// ------------------------------------------------------

public void mousePressed( MouseEvent e ) {

  // New click means no previous selection.
  this.activeMenu = -1;
  this.activeMenuItem = -1;

  Graphics g = this.getGraphics();
  FontMetrics theFontMetrics =
    g.getFontMetrics();

  if ( dmb.contains(e.getX(), e.getY()) )
  {
  // Handle menu selection.
    for ( int i = 0;
        i < dmb.getVector().size();
        i++ ) {

      // Get menu object.
      DesktopMenu d = (DesktopMenu)
        dmb.getVector().elementAt( i );

      // Determine if we're inside this menu.
      // This could be done with one or more Rectangles.
      // Note the (buried) conditional operator: it accounts
      // for both text and image (e.g. the Apple) menus.
      if ( ( e.getX() >= d.getX() -
        Global.defaultMenuHSep )
        && ( e.getX() <= ( d.getX() +
        ( d.getLabel() == null ?
        ((DesktopImageMenu)
        d).getImage().getWidth(this) :
        theFontMetrics.stringWidth(
        d.getLabel() ) ) +
        Global.defaultMenuHSep ) )
        && ( e.getY() >= this.getY() )
        && ( e.getY() <=
        this.getMenuBarHeight() ) )
      {

        // Draw menubar highlighting...
        g.setColor( Color.blue );

        // Save the current Rectangle surrounding the menu.
        // This will speed up painting on the following line,
        // and when the user leaves this menu.
        activeMenuRect = new
          Rectangle( d.getX() -
          Global.defaultMenuHSep,
          this.getY(),
          ( d.getLabel() == null ?
          ( ( DesktopImageMenu )d
          ).getImage().getWidth(this) :
          theFontMetrics.stringWidth(
          d.getLabel() ) ) + ( 2 *
          Global.defaultMenuHSep ),
          getMenuBarHeight() );

        g.fillRect( activeMenuRect.x,
          activeMenuRect.y,
          activeMenuRect.width,
          activeMenuRect.height );

// Draw menu String or Image.
        g.setColor( Color.black );
        if ( d.getLabel() != null )
          g.drawString( d.getLabel(),
            d.getX(), d.getY() );
        else
          g.drawImage(
            ( (DesktopImageMenu)d
            ).getImage(),
            d.getX(), d.getY(), this );

// Get menu item vector.
        Vector v = d.getVector();

// If the Trash is full, enable the menu item.
// This code can easily be moved; it is included
// here for illustration.
        DesktopMenuItem dmi =
          ( DesktopMenuItem )
          v.elementAt( 0 );
        if ( dmi.getLabel().equals(
            Global.menuItemEmptyTrash ) )
          {
          DesktopImage di = (
            (DesktopImage)
            this.vectorImages.elementAt(
            1 ) );
          String path = di.getPath();
          File f = new File( path,
            Global.trashDisplayString);
          if ( f.exists() ) {
            String array[] = f.list();

            if ( array == null ||
                array.length == 0 )
              dmi.setEnabled( false );
            else
              dmi.setEnabled( true );
            }
          }

// Draw menu background.
```

```
        g.setColor( dmb.getBackground() );
        g.fillRect(
          d.getItemBounds().getBounds().x,
          d.getItemBounds().getBounds().y,
          d.getItemBounds().getBounds
          ().width,
          d.getItemBounds().getBounds
          ().height );

    // Draw menu items.
        for ( int j = 0; j < v.size();
            j++ ) {
          g.setColor( Color.black );

          if ( !( ( DesktopMenuItem )
              v.elementAt( j )
              ).getEnabled() )
            g.setColor( Color.lightGray );

            g.drawString( (
              ( DesktopMenuItem )
              v.elementAt( j )
              ).getLabel(), (
              ( DesktopMenuItem )
              v.elementAt( j )
              ).getDrawPoint().x,
              ( ( DesktopMenuItem )
              v.elementAt( j )
              ).getDrawPoint().y );
          }

          g.setColor( Color.black );

    // Outline the menu item list bounding rectangle.
          g.drawRect(
            d.getItemBounds().getBounds(
            ).x,
            d.getItemBounds().getBounds(
            ).y,
            d.getItemBounds().getBounds(
            ).width,
            d.getItemBounds().getBounds(
            ).height );

    // Add horizontal drop shadow.
          g.fillRect(
            d.getItemBounds().getBounds(
            ).x + 2,
            d.getItemBounds().getBounds(
            ).y +
            d.getItemBounds().getBounds(
            ).height,
            d.getItemBounds().getBounds(
            ).width, 2 );

    // Add vertical drop shadow.
          g.fillRect(
            d.getItemBounds().getBounds(
            ).x +
            d.getItemBounds().getBounds(
            ).width,
            d.getItemBounds().getBounds(
            ).y + 2, 2,
            d.getItemBounds().getBounds(
            ).height );

          // Set current menu.
              this.activeMenu = i;

          // Once found, we're done.
              break;
            }
          }
        }
      }
```

Listing 2. *The mousePressed method handles menu selections.*

The `mouseDragged()` method (not shown) is responsible for the highlighting and unhilighting of menus and menu items: that is where menu items get redrawn with white text on a blue background, as depicted in Figure 3.

Displaying folder contents

Folders in Facade use a combination of core Java classes and Swing classes. The container itself descends from `java.awt.Window`, and the `paint()` method performs the low-level drawing calls (drawing the title bar, close box, etc.). The content is displayed inside a `JScrollPane`.

Once the Macintosh look-and-feel is activated, the scrollbars take the appearance shown in Figure 4. The icons within the scrollpane are added to a grid layout. This approach works well, except that the tendency of the scrollpane is to take up the entire display area (ignoring the title bar, etc.). So, on resize of the container, the scrollpane is also resized. A `java.awt.Insets` object is used to make this as painless as possible: the `Insets` values are used as the buffer area around the scrollpane.

Like most of the Facade classes, `DesktopFolder` implements the `MouseListener` and `MouseMotionListener` interfaces so it can receive mouse events. Within the methods required for those interfaces, the x-y

coordinates are checked to determine if a close, zoom, shade, grow, or drag operation is taking place, and the container gets re-shaped appropriately.

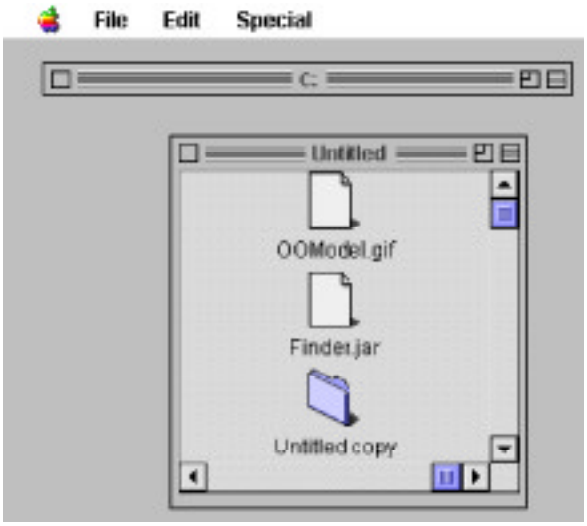

Figure 4. *Displaying folder contents*

```
// ------------------------------------------------------------
//     • addFileTree
// ------------------------------------------------------------

public void addFileTree( String s ) {

   // Instantiate the instance variables for this object's content.
   // The Mac L&F requires both scrollbars.
   pane = new JScrollPane();
   pane.setHorizontalScrollBarPolicy(
   JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
   pane.setVerticalScrollBarPolicy(
    JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

   // Make sure the argument passed in is a valid directory.
   String defaultVolume = s;

   if ( defaultVolume == null )
     return;

   File theDir = new File( defaultVolume );

   if ( !theDir.isDirectory() )
     return;

   // Retrieve the contents of this folder/directory.
   contents = theDir.list();
   int tempLength = 0;

   if ( contents != null ) {
     tempLength = contents.length;

   // Use a Swing panel inside the Swing scrollpane.
   // Default to a grid layout simply because it looks the best.
   JPanel p = new JPanel();
   p.setLayout(
     new GridLayout( 5, 3, 5, 5 ) );

   // If the contents contain full path specifications, there
   // will be some extraneous characters that we don't want
   // to display.
   char pathSeparatorChar =
     theDir.pathSeparatorChar;
   char separatorChar =
     theDir.separatorChar;
   Vector v = new Vector();
   v.addElement( Global.spaceSep );

   int loc = 0;
   int k = 0;

   for ( int j = 0; j < tempLength; j++ )
   {

     // For each item, separator chars should become
     // spaces for display purposes.
     File tempFile =
       new File( theDir, contents[ j ] );

     contents[ j ] = new String(
       contents[ j ].replace(
       pathSeparatorChar, ' ' ) );
     contents[ j ] =
       new String( contents[ j ].replace(
       separatorChar, ' ' ) );

     for ( int l = 0; l < v.size(); l++ )
     {
   // Parse root volume name.
   // Remove leading '%20' character.
       loc = contents[ j ].indexOf(
         ( String )v.elementAt( l ) );

       if ( loc == 0 )
         contents[ j ] = new String(
           contents[ j ].substring( (
           ( String )v.elementAt( l )
           ).length() ) );

   // Volume name includes first '%20'.
   // Rework this for MacOS.
       loc = contents[ j ].indexOf(
         ( String )v.elementAt( l ) );

   // Build the final display String from substrings.
       while ( ( loc > 0 ) &&
           ( loc < contents[ j ].length()
           + l ) ) {
         String s1 = new String(
           contents[ j ].substring( 0,
           loc ) );
         String s2 = new String(
           contents[ j ].substring( loc
           + ( ( String )v.elementAt( l )
           ).length() ) );
           contents[ j ] = new String(
           s1 + " " + s2 );
```

```
          loc = contents[ j ].indexOf(
            ( String )v.elementAt( 1 ) );
        }
      }

    // Now build the appropriate icon.
      Image theImage;
      int tempWidth = 0;
      DesktopFrameItem d;
      ImageIcon ii = new ImageIcon();

    // Files obviously look different than folders.
    // Set the appropriate Image.
    // This version does not handle mouse clicks on
    // documents.
      if ( tempFile.isFile() ) {
        d = new DesktopDoc();
        ii = new ImageIcon( (
          ( DesktopDoc )d ).getImage() );
      }
      else {
        d = new DesktopFolder();
        ii = new ImageIcon( (
          (DesktopFolder)d ).getImage());
        d.addMouseListener(
          ( DesktopFolder )d );
      }

    // Set the display Strings.
      d.setLabel( contents[ j ] );
      d.setText( contents[ j ] );

    // Set the path for the item. It is built from the parent folder
    // path and filename.
      d.setPath( this.getPath() +
        System.getProperty(
        "file.separator" ) + contents[j] );

    // And set the icon.
      d.setIcon( ii );

    // Swing methods for positioning the icon and label.
      d.setHorizontalAlignment(
        JLabel.CENTER );
      d.setHorizontalTextPosition(
        JLabel.CENTER );
      d.setVerticalTextPosition(
        JLabel.BOTTOM );

      ii.setImageObserver( d );

    // Add the completed item to the panel.
      p.add( d );
    }

  // Set the panel characteristics, then add it to the scrollpane.
    p.setVisible( true );

    pane.getViewport().add( p, "Center" );
    pane.setVisible( true );

  // A container within a container within a container...it works.
    panel = new JPanel();
    panel.setLayout( new BorderLayout() );
    panel.add( pane, "Center" );
    this.add( panel );

  // Use the Insets object for this window to set the viewing
  // size of the panels and scrollpane.
    panel.reshape( this.getInsets().left,
      this.getInsets().top,
      this.getBounds().width -
      this.getInsets().right -
      this.getInsets().left,
      this.getBounds().height -
      this.getInsets().top -
      this.getInsets().bottom );

  // Ready for display.
    panel.setVisible( true );
    this.pack();
    this.validate();
}
```

Listing 3: *Building a folder's display area.*

The trash

The trash is simply a directory located at the root of the volume (for example, `c:\Trash`). Items can be dragged onto the trash can icon, the mouse button released, and the item (and its contents, if it is a folder) are moved to the trash directory. If the item has an open window, that window is destroyed. Emptying is accomplished through the "Empty Trash..." menu item in the Special menu. Figure 4 shows a folder being dragged to the trash. The cursor did not get captured in the image, but it is positioned over the trash icon, ready to release the folder.

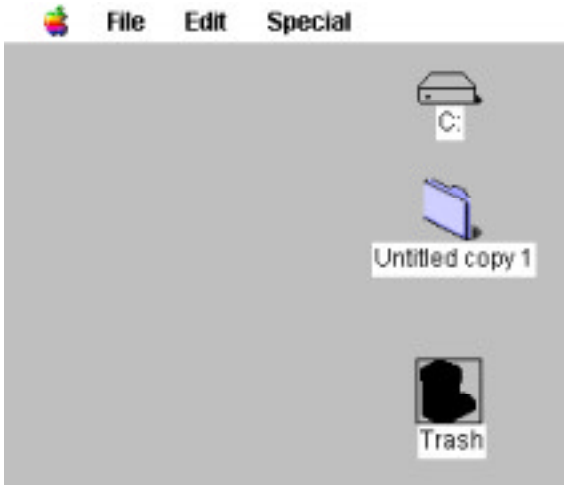

Figure 5. *Dragging a folder to the Trash*

The code in Listing 4 handles the "Empty Trash..." menu item. If a match is found between the selected item and the constant assigned as the Trash label, the path to the Trash is retrieved. Once the code determines the "Trash" is indeed an existing directory, the `emptyDir()` method is called. Once `emptyDir()` returns, the icon is changed from full to empty. Note that the trash icon is always present in the `vectorImages` object. This `Vector` contains the known images (icons) for items on the desktop. The trash is always at position 0, the first position. The root volume is at position 1. This allows quick, though hardcoded, access to the images when repainting occurs, or an image needs swapping.

```
// ------------------------------------------------------------
//     • handleMenuItem
// ------------------------------------------------------------

public boolean handleMenuItem() {

  boolean returnValue = false;

 // Handle active menu item.
  if ( ( this.activeMenu != -1 ) &&
       ( this.activeMenuItem != -1 ) ) {

   // Get menu object.
     DesktopMenu dm = ( DesktopMenu )
       dmb.getVector().elementAt(
       this.activeMenu );

   // Get menu item.
     String s = ( ( DesktopMenuItem )
       dm.getVector().elementAt(
       this.activeMenuItem ) ).getLabel();

   // <snip>
     if ( s.equals(
          Global.menuItemEmptyTrash ) ) {

     // The path to the trash is assumed to be of the form:
     //    <root vol>/Trash
       DesktopImage di = ( ( DesktopImage )
         this.vectorImages.elementAt( 1 ) );
       String path = di.getPath();
       File f = new File( path,
         Global.trashDisplayString );

     // Once we have the object that references the Trash dir...
       if ( f.exists() && f.isDirectory() )
       {

      // Call the method which empties it.
         boolean b = this.emptyDir( f );

      // If successful, change the icon back to empty.
         if ( b ) {

        // Save the current bounds. We really want the x-y.
           Rectangle r = ( ( DesktopImage )
             this.vectorImages.elementAt(0)
             ).getBounds();

        // Change the icon. imgTrash and imgTrashFull are two
        // instance variables, each referencing the appropriate
        // icon.
           this.vectorImages.setElementAt(
             this.imgTrash, 0 );
           ( ( DesktopImage )
             this.vectorImages.elementAt( 0
              ) ).setBounds( r );

        // Show the change.
           this.repaint();
         }
       }
     }
   }
}
```

Listing 4*: Handling the "Empty Trash..." menu item.*

Emptying the trash

`emptyDir()` is displayed in Listing 5. This method will empty the specified directory recursively. As it finds each item in the directory, appropriate action is taken. If the item is a file it is immediately deleted. If it is a directory, `emptyDir()` is called again,

with the subdirectory name as its argument. Eventually, all of the files in the subdirectory are deleted, and then the subdirectory itself is removed.

```
// ------------------------------------------------------------
//     • emptyDir
// ------------------------------------------------------------

public boolean emptyDir( File dir ) {

  // Recursive method to remove a directory's contents,
  // then delete the directory.
   boolean b = false;

  // Get the directory contents.
   String array[] = dir.list();

  // If the path is screwed up, this will catch it.
   if ( array != null ) {

    // Iterate over the directory contents…
     for ( int count = 0;
          count < array.length; count ++ ) {
       String temp = array[ count ];

     // Create a new File object using path + filename.
       File f1 = new File( dir, temp );

     // Delete files immediately.
     // Call this method again for subdirectories.
       if ( f1.isFile() ) {
         b = f1.delete();
       }
       else if ( f1.isDirectory() ) {
         b = this.emptyDir( f1 );
         b = f1.delete();
       }
     }
   }

   return b;
}
```

Listing 5: *Emptying a directory.*

Saving and restoring state

Facade uses a variant on the standard Java serialization mechanism for saving the `Desktop` state. The code in Listing 6 illustrates the saving and restoring of open folder windows. Both of these methods are in the `Desktop` class. Note that rather than flatten entire `DesktopFrame` objects, this code saves only specific attributes from each object. The primary reason for this approach is that it avoids any exceptions thrown when attempting to serialize the Swing components and Images (which are not serializable by default) within each `DesktopFrame`. Another reason is the relative ease with which this approach may be implemented. Plus, it reduces the amount of data written to disk. One disadvantage is that the current scroll position of any open window is lost.

```
// ------------------------------------------------------------
//     • saveState
// ------------------------------------------------------------

private void saveState() {

  // The Desktop data will be stored in a file at the root level.
  DesktopImage di = ( ( DesktopImage )
    this.vectorImages.elementAt( 1 ) );
  String s = new String( di.getPath() +
    "Desktop.ser" );

  try {
    FileOutputStream fos =
      new FileOutputStream( s );
    ObjectOutputStream outStream =
      new ObjectOutputStream( fos );

  // Use a temporary Vector to hold the open DesktopFrames.
  // This should not be necessary when shutting down, but it
  // is a good habit to follow.
    Vector v = new Vector();
    v = ( Vector )
      this.vectorWindows.clone();

  // Use another temporary Vector to hold the attributes
  // to save.
    Vector temp = new Vector();

    for ( int i = 0; i < v.size(); i++ ) {

    // For each DesktopFrame, we'll save its path, display
    // string, and its bounding Rectangle. If window-shading is
    // in effect on an object, restore the full height before
    // saving.
      DesktopFrame df = ( DesktopFrame )
        v.elementAt( i );
      temp.addElement( df.getPath() );
      temp.addElement( df.getLabel() );

      if ( df.getShade() ) {
        df.restoreHeight();
      }

      temp.addElement( df.getBounds() );
    }

  // Write the Vector contents to the .ser file.
    outStream.writeObject( temp );
    outStream.flush();
```

```
      outStream.close();
    }
    catch ( IOException e ) {
      System.out.println(
        "Couldn't save state." );
      System.out.println( "s = " + s );
      e.printStackTrace();
    }
  }

// ------------------------------------------------------------
//     • restoreState
// ------------------------------------------------------------

  private void restoreState() {

    // The Desktop data is stored in a file at the root level.
    // This step occurs near the end of the Desktop constructor,
    // and so can use the root volume to build the path.
    DesktopImage di = ( ( DesktopImage )
      this.vectorImages.elementAt( 1 ) );
    String s = new String( "\\Desktop.ser" );

    try {

      // Open the file, and read the contents. In this version
      // we know it's just one Vector.
      FileInputStream fis =
        new FileInputStream( s );
      ObjectInputStream inStream =
        new ObjectInputStream( fis );
      Vector v = new Vector();
      v = ( Vector )inStream.readObject();

      inStream.close();

      Rectangle r = new Rectangle();

      for ( int i = 0; i < v.size(); i++ ) {

      // Iterate over the Vector contents. We know the save
      // format:
      // each field corresponds to a specific attribute of a
      // DesktopFrame.

      // Create a new DesktopFrame using the retrieved path.
        s = ( String )v.elementAt( i );

        DesktopFrame df =
          new DesktopFrame( s );

      // Set its label using the next Vector element.
        i++;
        s = ( String )v.elementAt( i );
        df.setLabel( s );

      // Set its bounding Rectangle using the next Vector
      // element.
        i++;
        r = ( Rectangle )v.elementAt( i );
        df.setBounds( r );

      // Prepare and display the DesktopFrame.
        df.pack();
        df.validate();
        df.show();
        df.toFront();
        df.repaint();

      // Add the object to the Desktop's Vector of open
      // windows.
        Desktop.addWindow( df );
      }
    }
    catch ( ClassNotFoundException e ) {
      System.out.println(
        "ClassNotFoundException in
        retrieveState()." );
    }
    catch ( IOException e ) {
      System.out.println(
        "Couldn't retrieve state." );
      System.out.println( "s = " + s );
      e.printStackTrace();
    }
  }
```

Listing 6: *Saving and restoring open DesktopFrames.*

Packaging (Facade in a JAR)

Facade can be run from a Java Archive (JAR) file, or as a set of separate classes plus images. In addition, the Macintosh look-and-feel classes must be present (usually in a separate JAR). The system environment variables need to be setup properly in order for the Java runtime to find the classes.

The JAR file for Facade was created like this:

```
jar cf Facade.jar /Facade/*.class /Facade/
*.gif
```

Once the environment variables are set, Facade can be invoked as follows:

```
java Facade
```

Other classes

In addition to the `java.awt` classes already discussed, there are other packages and classes used often in Facade.

This program relies heavily on the `java.util.Vector` class for maintaining a semblance of order among the various

objects. Several `Vector` instances are used for tracking icons and open windows. Figure 6 illustrates the core concept of a `Vector`: it is a growable array.

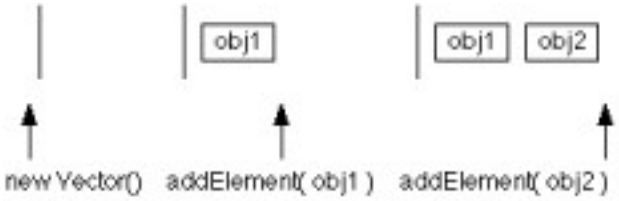

Figure 6. *Creating and populating a Vector. The arrow represents a pointer to the current element at the end of the Vector.*

The `java.io.File` class provides most of the file system operations for display and management. Refer to the references provided in the bibliography for API details. One platform-specific detail that was not discussed is the translation of a root path to a form that is valid for a File directory object. For example, calling the `File.list()` method on the path `"c:\"` will not return any file or folder names, but calling the same method for the path `"/"` will return the root directory contents.

Facade also uses the Java Foundation Classes (Swing) to provide a consistent look and feel across platforms, primarily for window scrolling operations. Swing runs a bit slow right now, but as peformance improves Swing classes can be substituted for some of the Facade custom classes. For example, the code that handles folder content display can be modified to show a tree structure. This capability should be easy to implement using the Swing classes, with little additional low-level drawing code.

Conclusion

Writing Java classes to model operating system functionality requires some choices, particularly in the selection of pre-defined GUI classes. Although many classes are available, you will need to create others, since performance of the pre-defined classes may be slow, or the behavior may not be exactly what you need. Both the appearance and behavior for custom GUI elements must be written.

This paper touched on several key areas: desktop layout, menu and mouse handling, icon display and movement, the trash, and saving and restoring the desktop state. All of these functional areas can be modeled easily using the Java 1.1 classes, supplemented by JFC and custom-built classes.

Bibliography

[1] Englander, Robert. *Developing Java Beans*. O'Reilly & Associates, Inc., Sebastopol, CA. 1997.

[2] Flanagan, David. *Java in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA. 1997.

[3] Niemeyer, Patrick and Peck, Joshua. *Exploring Java*. O'Reilly & Associates, Inc., Sebastopol, CA. 1997.

[4] Weiner, Scott and Asbury, Stephen. *Programming with JFC*. John Wiley & Sons, Inc., New York, NY. 1998.