JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 5, May-June 2004

Project Imperion: New Semantics, Façade and Command Design Patterns for Swing

Douglas Lyon, Computer Engineering Department, Fairfield University, Connecticut, USA

…Thus in the beginning the world
was so made that certain signs
come before certain events

- Cicero. 106–43 B.C.

Abstract

Project Imperion consists of a library of GUI elements that combine the command and facade design patterns to add new semantic meaning to some common Swing components. This simplifies the synthesis and maintenance of the code. It also shortens the code, while improving readability, speeding synthesis and easing maintenance. It also promotes better code organization, including the separation of business logic from GUI code and a logical grouping of the code along the same lines as its GUI's appearance.

Started in the late '90's, in the DocJava, Inc. *Skunkworks*, the Imperion project was meant only for menu items using AWT, targeting the single application of image processing programs. It has since been reworked and extended to support the use of 16 Swing components with built-in keyboard shortcuts. The Imperion project was named for the Latin root, imperium, meaning the power to command.

1 WHAT'S WRONG WITH THE CURRENT EVENT MODEL?

When programmers use the current event model, they typically jump around in the source code. For example, it is normal for programmers to create a graphic user interface using a 5 step process:

1. Make a new button (or *Component*)
2. Set the keyboard shortcuts.

3. Add the button to the container.
4. Add an *ActionListener* to the button
5. Create a long dispatch in an *actionPerformed* method.

Visiting code in several places in order to add a feature is a fruitful source of errors. The code is also hard to maintain, and less readable in that the programmer must jump back and forth in the code file to see declarations, installation of components into GUI containers and implementations of event listeners.

Such an approach does not scale well. When the number of different components gets large, these jump points can be many lines of code away from one another. To illustrate the programmers' jumping around in the code, consider the following example:

```
package gui.run.examples;

import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class CommandFrame extends JFrame
        implements ActionListener {
    //1. Make the new buttons
    JButton okButton = new JButton("OK");
    JButton cancelButton = new JButton("cancel");

    public CommandFrame() {
        Container c = getContentPane();

        //2. Set the shortcuts
        okButton.setMnemonic('o');
        cancelButton.setMnemonic('c');

        //3. Add the components to the
        //   container
        c.add(okButton);
        c.add(cancelButton);

        //4. Set up the ActionListeners
        okButton.addActionListener(this);
        cancelButton.addActionListener(this);
        c.setLayout(new FlowLayout());
        setSize(200, 200);
        show();
    }

    public void actionPerformed(ActionEvent e) {
        Object o = e.getSource();

        // 5. Add to a if-else dispatch
        if (o == okButton)
            System.out.println("ok");
        else if (o == cancelButton)
            System.out.println("cancel");
    }
    public static void main(String[] args) {
        new CommandFrame();
    }
}
```

Thus, the typical procedure is to put an instance of a button into a class member variable. During the initialization phase for the GUI, the shortcuts are established, components are added to the container and the listeners are added. During the event handling dispatch we check the event to see if it the button was selected. As a result, we have created a situation where we have to visit the code in several places. This complicated procedure for adding code also complicates maintenance and decreases reliability of the program. It also adds more code. The output of the *CommandFrame* is shown in Figure 1.

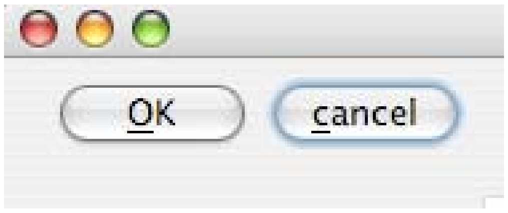

Figure 1. OK-Cancel Dialog.

In addition, using the above formulated *actionPerformed* method, one can expect a rather long dispatch. As an example, consider the following code, taken from [Lyon 1999]:

```
      public void actionPerformed(ActionEvent e) {
            if (match(e, mandelbrot_mi)) {
                mandelbrot();
                return;
            }
            if (match(e, goslab_mi)) {
                goslab();
                return;
            }
    // .... 17 if statements later....
            if (match(e, systemInfo_mi)) {
                systemInfo();
                return;
             }
            super.actionPerformed(e);
        }
```

Such long dispatches can be exceedingly difficult to maintain. And GUI elements are removed from their definitions by hundreds of lines of code (in programs of only modest complexity). Even more insidious is the invocation:

```
super.actionPerformed(e);
```

This can make reference to an unknown number of concrete super-classes that improve implementation reuse at a cost of increased fragility. In short, such a programming style does not scale well because it does not encapsulate complexity at the right level. The following section shows that adding function points in this way is a waste of time (i.e., unnecessarily difficult *and* wrong-headed).

If you don't
have time to do it right
the first time, when
will you have time to
do it again?

– Unknown

2 BUILDING A BETTER BUTTON

This section presents an approach to building the same program by creating a new, more feature-rich button, called the *RunButton*. A *RunButton* instance is a button that knows how to run itself. Such a button makes use of what is known as the *command design pattern*. The command design pattern places an instance of a command into an instance of another class, call the *issuer*. For example, a button can have the role of the issuer, holding a reference to an instance of a command. Thus a *RunButton will* invoke a command (*run()*) on the command instance. The command instance has the role of either message forwarding the command to a specific recipient or of executing the command itself.

We define the *RunButton* class as a subclass of the *JButton* class. It works using the *façade design pattern* to map the ActionListener interface into the simpler *Runnable* interface. The *run* method is left undefined and this causes the class to be *abstract*.

```
package gui.run.examples;

import gui.ClosableJFrame;
import gui.run.RunButton;

import java.awt.Container;
import java.awt.FlowLayout;

public class ExampleRunButton {
    public static void main(String args[]) {
```

```
              // anonymous inner class
              // That uses the command pattern
              // also uses facade pattern
              // since the normal
              // requires an
              // actionListener-
              // actionPerformed(ActionEvent e)
              // now we just need a run method.
              // Semantics for the runButton now include
              // an implicit metaChar='['

              ClosableJFrame cf =
                      new ClosableJFrame(
                              "OK-CANCEL Frame");
              Container c = cf.getContentPane();
              c.setLayout(new FlowLayout());

            RunButton okButton = new RunButton("ok") {
                          public void run() {
                              System.out.println(getText());
                          }
                      };
            okButton.setMnemonic('o');
            c.add(okButton);
            RunButton cancelButton = new RunButton("cancel") {
                          public void run() {
                              System.out.println(getText());
                          }
                      };
            cancelButton.setMnemonic('c');
            c.add(cancelButton);
              cf.setSize(200, 200);
              cf.show();
        }
  }
```

The *main* method in the above code sample also produced Figure 1.

3 WHAT'S WRONG WITH THE STANDARD LISTENERS?

The astute reader may ask, "why start using non-standard GUI elements in the code? What's so wrong with using the command pattern without altering the interface from an *ActionListener* to a *Runnable*?" The primary problem is that there are just too many listeners to keep track of. As of j2se1.4.2, there are 68 known different interfaces that extend the *java.util.EventListener* interface. There are 234 known implementations of these interfaces in a wide variety of classes. This is too much complexity for the average programmer to keep in their head.

Using the *Facade* design pattern we are able to map several common components to take advantage of the simpler *Runnable* interface, as shown in Figure 2.

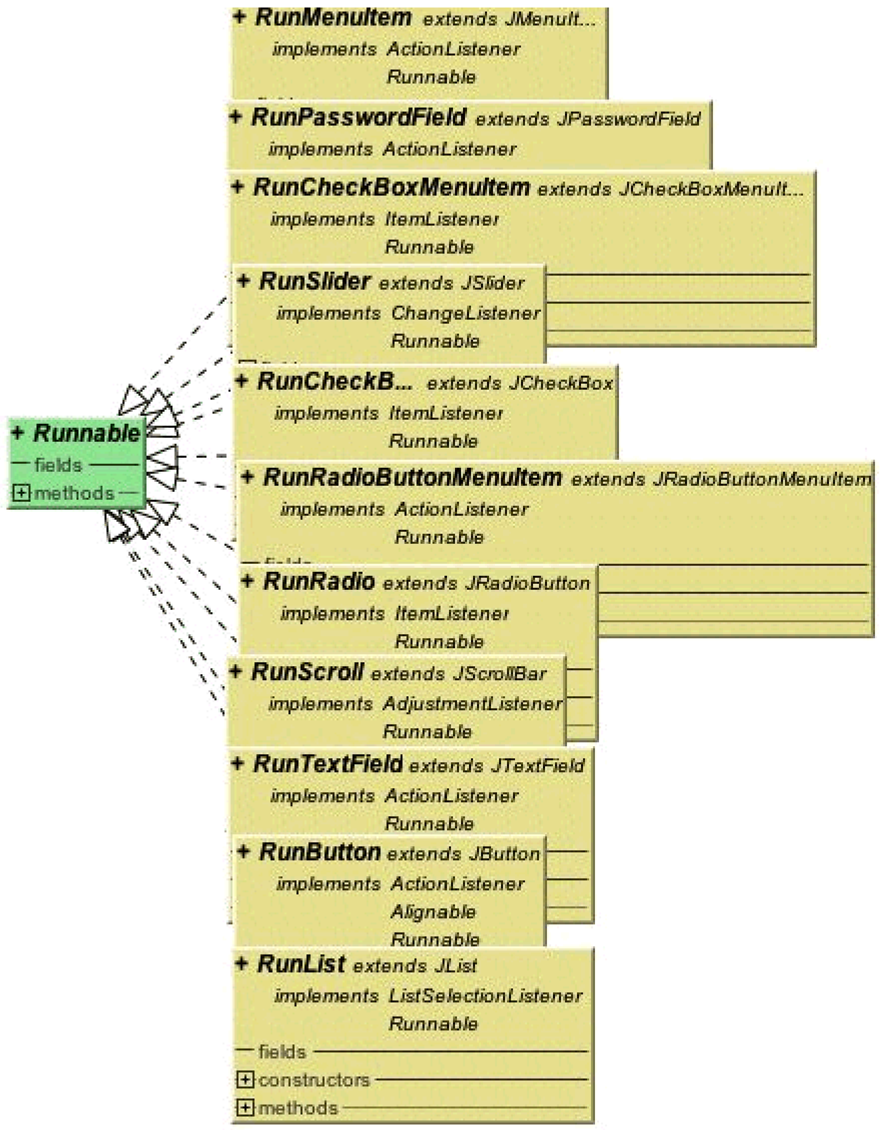
+ RunMenuItem extends JMenuIt...
implements ActionListener
Runnable
+ RunPasswordField extends JPasswordField
implements ActionListener
+ RunCheckBoxMenuItem extends JCheckBoxMenuIt...
implements ItemListener
Runnable
+ RunSlider extends JSlider
implements ChangeListener
Runnable
+ RunCheckB... extends JCheckBox
implements ItemListener
Runnable
+ Runnable
fields
methods
+ RunRadioButtonMenuItem extends JRadioButtonMenuItem
implements ActionListener
Runnable
+ RunRadio extends JRadioButton
implements ItemListener
Runnable
+ RunScroll extends JScrollBar
implements AdjustmentListener
Runnable
+ RunTextField extends JTextField
implements ActionListener
Runnable
+ RunButton extends JButton
implements ActionListener
Alignable
Runnable
+ RunList extends JList
implements ListSelectionListener
Runnable
fields
constructors
methods

Figure 2. Some Run Components

One of the basic questions that we must ask of a framework is: "how much complexity is too much?" If a feature set overwhelms us, then we no longer have the ability to program without constantly consulting documentation. This is a burden to the programmer. In short, simplicity is a desirable feature. By mapping several of the interfaces to the simpler *Runnable* interface, we can simplify our code. We dispense with the ability to obtain the target of the event, but, since the GUI element always knows that it is its' own target, it no longer needs these parameters. Thus, we have made a design decision to always make use of the command pattern when invoking GUI elements in the Imperion project.

4 WHY ADD KEYBOARD SHORTCUTS?

Keyboard shortcuts are often added to an interface as an afterthought. In fact, a GUI is really just a visual language for commanding a program. Adding more GUI elements to add more commands (i.e., feature points) is common practice. Programmers often ask, "Do we really need keyboard shortcuts too?". There are compelling reasons to add keyboard shortcuts, and to make these as easy as possible for the programmer to maintain them.

Screen space is consumed as a function of the number of displayed GUI elements. Even pop-up menus consume valuable screen space and make selection difficult. For example, menus with more than 10 items make scanning a list tedious. Also, many of the items may be irrelevant to the users' current need [Laurel]. This is the primary motivation for the use of a hierarchic menu. A hierarchic menu allows a grouping of items into relevant sub-menus. The danger is that the hierarchical menu becomes so deep that the user gets lost. According to Tognazzini, the hierarchical menu offers hundreds of extra options, but it takes much longer for the user to make a selection. Additionally, Tognazzini says that menu selections are generally faster than keyboard shortcuts [Tog].

As a result, it is common to associate keyboard shortcuts with GUI elements. This enables the user to decide how to invoke a process. It also can lead to faster execution of application commands by skilled users.

5 WHAT IS WRONG WITH THE PRESENT KEYBOARD SHORTCUT SYSTEM?

At present, a programmer must isolate a reference to a GUI element and explicitly make an instance of a *Keystroke* that will be associated with the GUI element. Consider the following code fragment:

```
RunButton sleepButton = new RunButton(
        "sleep") {
    public void run() {
        System.out.println(getText());
    }
};
sleepButton.setMnemonic('s');
c.add(sleepButton); // add button to container
```

The *RunButton* instance is stored in the *sleepButton* variable. This variable is needed so that the programmer can explicitly set the mnemonic that will be associated with a button press. Such a programming practice requires that a new variable be introduced for each

GUI element, and that it be explicitly named by the programmer. These variables have a nasty habit of propagating through the program, adding both code and complexity.

6 ADDING NEW SEMANTIC MEANING TO GUI ELEMENTS

This section shows how the Imperion project adds keyboard shortcuts to GUI elements by modifying their in-line text. Consider the following code sample:

```
        c.add(new RunButton("[sleep") {
               public void run() {
                   System.out.println("alarm!");
               }
           });
```

The new *RunButton* has an *implicit* mnemonic s. The rule is that the first character after the '[' will be the mnemonic keyboard shortcut for the Imperion component. Also, that the meta-character '[' will be stripped from the final appearance of the component. The following example also produces the image shown in Figure 1:

```
package gui.run.examples;
import gui.ClosableJFrame;
import gui.run.RunButton;
import java.awt.Container;
import java.awt.FlowLayout;

public class ExampleRunButton {
    public static void main(String args[]) {
        ClosableJFrame cf =
                new ClosableJFrame(
                        "OK-CANCEL Frame");
        Container c = cf.getContentPane();
        c.setLayout(new FlowLayout());

        c.add(new RunButton("[ok") {
            public void run() {
                System.out.println(getText());
            }
        });
        c.add(new RunButton("[cancel") {
            public void run() {
                System.out.println(getText());
            }
        });
        cf.setSize(200, 200);
        cf.show();
    }
}
```

A mnemonic shortcut always uses a meta character (this is not true for keyboard accelerators). For example, on Windows or a mac, the meta character is the ‘alt’ key. Thus, to issue the ‘OK’ command, the user types: “alt-o”. The mnemonic shortcuts are only active in a *context*. Thus, in a hierarchic menu system, only the visible mnemonics are active. This facilitates navigation of the menus.

Mnemonic shortcuts are different from keyboard accelerators in that accelerators do not require a meta-character and they can be invoked from any context. Thus, keyboard accelerators map to visual commands in a global way. The advantage of global invocation is that you can issue commands with fewer keystrokes. The disadvantage of global invocation is that it is easier to have conflicting accelerators in the interface. For example, multiple menu items can map to the same accelerator. In such a case, the last one to be added to the GUI is the one that gets executed. In the Imperion system accelerators are defined with the begin curly brace ‘{‘ and the end curly brace ‘}’.

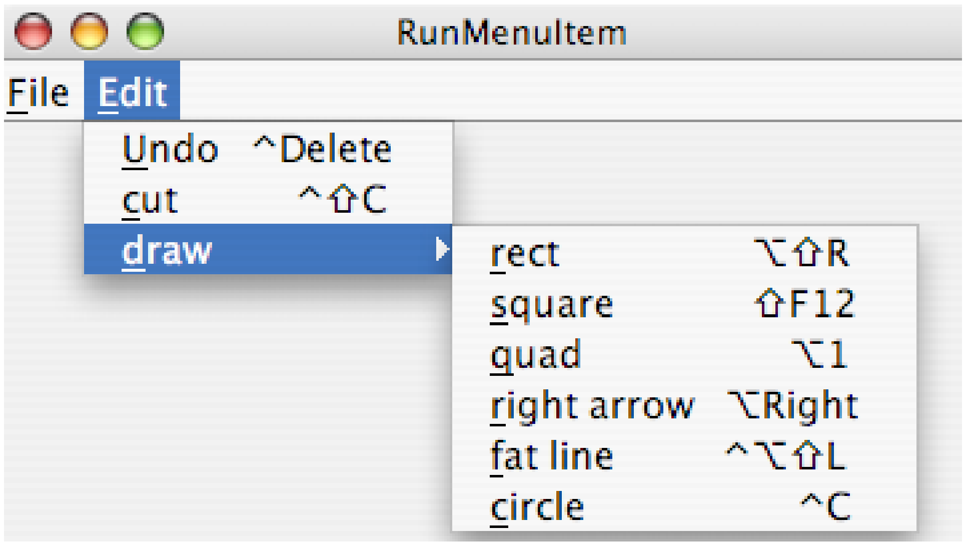

Figure 3. RunMenuItems with Mnemonics and Accelerators

Figure 3 shows that the key strokes “ctrl-c” will draw a circle, but that “ctrl-C” will do a cut. Also, these keystrokes may be used from anywhere in the frame. Compare this with the context sensitive mnemonics that require the user type “alt-e,d,c” to obtain a circle. While more keystrokes are involved, the hierarchic nature of the mnemonics enables the organization of keystrokes according to their function. Global keyboard accelerators are more likely to collide than mnemonics. Collisions create semantic errors. For example, if “ctrl-c” was used for both the cut and the circle commands, it would only invoke the command associated with the keystroke added last. It should probably be a run-time error to add duplicate keystrokes, but Imperion will not catch the error, at present (nor, for that matter, does Swing). The following code will set up the GUI depicted in Figure 3:

```
package gui.run.examples;

import gui.ClosableJFrame;
import gui.run.RunMenu;
import gui.run.RunMenuItem;

import javax.swing.JMenuBar;
import java.awt.Container;

public class RunMenuItemExample {
    public static void main(String args[]) {
        new RunMenuItemExample();
    }

    private RunMenuItemExample() {
        ClosableJFrame cf = new ClosableJFrame(
                "RunMenuItem");
        Container c = cf.getContentPane();
        cf.setJMenuBar(getMenuBar());
        c.setLayout(new java.awt.FlowLayout());
        cf.setSize(200, 200);
        cf.setVisible(true);
    }

    private JMenuBar getMenuBar() {
        JMenuBar mb = new JMenuBar();
        mb.add(getFileMenu());
        mb.add(getEditMenu());
        return mb;
    }

    private RunMenu getDrawMenu() {
        RunMenu drawMenu = new RunMenu("[draw");
        drawMenu.add(new gui.run.RunMenuItem(
                "[rect{alt shift R}") {
            public void run() {
                System.out.println(getText());
            }
        });
        drawMenu.add(new RunMenuItem(
                "[square{shift F12}") {
            public void run() {
                System.out.println(getText());
            }
        });
        drawMenu.add(new RunMenuItem(
                "[quad{alt 1}") {
            public void run() {
                System.out.println(getText());
            }
        });
        drawMenu.add(new RunMenuItem(
                "[right arrow{alt RIGHT}") {
            public void run() {
                System.out.println(getText());
            }
        });
        drawMenu.add(new RunMenuItem(
                "[fat line{ctrl alt shift L}") {
            public void run() {
                System.out.println(getText());
            }
        });
        drawMenu.add(new RunMenuItem(
                "[circle{ctrl C}") {
            public void run() {
                System.out.println(getText());
            }
        });
        return drawMenu;
    }

    private RunMenu getEditMenu() {
        RunMenu editMenu = new RunMenu("[Edit");
        editMenu.add(new RunMenuItem(
                "[Undo{control DELETE}") {
            public void run() {
                System.out.println(getText());
            }
        });
        editMenu.add(new RunMenuItem(
                "[cut{ctrl shift C}") {
            public void run() {
                System.out.println(getText());
            }
        });
        editMenu.add(getDrawMenu());
        return editMenu;
    }

    private RunMenu getFileMenu() {
        RunMenu fileMenu = new RunMenu("[File");
        fileMenu.add(new RunMenuItem(
                "[open{alt shift X}") {
            public void run() {
                System.out.println(getText());
            }
        });
        fileMenu.add(new RunMenuItem(
                "[save{alt control DELETE}") {
            public void run() {
                System.out.println(getText());
            }
        });
        fileMenu.add(new RunMenuItem(
                "[close{PAGE_UP}") {
            public void run() {
                System.out.println(getText());
            }
        });
        fileMenu.add(new RunMenuItem(
                "[export{NUMPAD0}") {
            public void run() {
                System.out.println(getText());
            }
        });
        return fileMenu;
    }
}
```

One important feature of the code is that the visual organization of the menus and menu items is reflected in the structure of the code. That is, one does not have to guess what the function of the *getDrawMenu* is. A logical grouping of menu item constructions into menu factory methods improves maintainability and readability. The use of the anonymous inner classes promotes short method bodies (since people don't generally like to see long anonymous inner classes). The short method bodies, in turn, promote method forwarding to the business logic (improving code reuse and promoting the separation of GUI code and business logic).

7 HOW DOES IMPERION DO IT?

This section describes the Imperion technique for modifying the keyboard shortcut used on a component. Consider the following constructor for the *RunMenuItem*:

```
public RunMenuItem(String l, Icon i) {
    super(l, i);
    addActionListener(this);
    ShortcutUtils.addShortcut(this);
}
```

At the heart of the mechanism is the invocation to *ShortcutUtils.addShortcut*.

```
public static void addShortcut(
         AbstractButton ab) {
     addMnemonic(ab);
     addAccelerator(ab);
}
```

The *addMnemonic* will scan for the mnemonic meta character (i.e., '[') and add the following character as a mnemonic, if the meta character is present. The *addMnemonic* method is implemented as:

```
private static void addMnemonic(
        AbstractButton ab) {
    String s = ab.getText();
    if (!hasMnemonic(s)) return;
    ab.setMnemonic(getMnemonicChar(s));
    ab.setText(stripMnemonicMetaChar(s));
}

private static boolean hasMnemonic(String s) {
    int mc = s.indexOf(META_CHARACTER);
    if (-1 == mc)
        return false;
    if (s.length() == mc - 1) return false;
    return true;
}
```

The basic idea is that *hasMnemonic* will return true only if there is a '[' present and there is a character following it. If both these conditions are met, then the character is stripped out and used to add a mnemonic. The label on the component is then reset.

8 CONCLUSION

The use of the command and facade design patterns to formulate a new GUI library is not new. In fact, this was described in detail in my most recent book [Lyon 2004]. The embedding of keyboard shortcuts into the text of GUI elements using a code is not new either [Lyon 1999].

However, the combination of the facade and command patterns, along with the embedding of keyboard shortcuts into the text of GUI elements, is new.

A logical grouping of menu item constructions into menu factory methods improves maintainability and readability. The use of the anonymous inner classes promotes short method bodies (since people don't generally like to see long anonymous inner classes). The short method bodies, in turn, promote method forwarding to business logic (improving code reuse and promoting the separation of GUI code and business logic).

One of the drawbacks of the Imperion system is that the components map their actions to only a single listener (themselves). This would seem, on the surface, to be a big limitation. For example, what if many components are interested in a text-field action? I suggest that if this occurs, we are too tied to the components. That text-field should be treated as a controller in a model-view-controller design pattern. A control should alter a model. Views that are interested in the model should be observing the model, *not* the controller. It should come as no surprise, therefore, that having components listen to their own event is an easy limitation to live with. In fact, I contend that this is preferable, as it limits inter-object associations, which is, in my view, a primary metric of object-oriented complexity.

Another limit of Imperion (and of Swing, for that matter) is that the semantic error of duplicate keyboard accelerators is not caught. To catch the duplicates, you need code that goes to the root of the content pane and then forms a list of all accelerators in all components. Overcoming this limitation requires addressing some open questions. For example: what is the policy for accelerator duplication? That is, should we throw a run-time exception, put up a dialog box, print a warning to the console, etc. Also at question is the issue of responsibility. Sun has already made it the implicit responsibility of the programmer to check for conflicts during the test phase of the development. Since conflict checks are time consuming for the programmer, perhaps an optional display feature (with a nice GUI) would not be amiss. In any case, this remains a topic of future work.

REFERENCES

[Laurel] *The Art of Human-Computer Interface Design*, edited by Brenda Laurel, Addison Wesley, 1990.

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

[Lyon 2004] Douglas A. Lyon. *Java for Programmers*, Prentice Hall, 2004. Available from http://www.docjava.com.

[Tog] Bruce Tognazzini. *Tog on Interface*, Addison Wesley, 1992.

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.