Eclipsing Your IDE

James W. Cooper

I have been hearing about the Eclipse project for some months, and decided I had to take some time to play around with it. Eclipse is a development project (www.eclipse.org) for building development environments. And much of Eclipse is itself written in Java. Developers make specific development environments by writing specific modules called Eclipse *plug-ins.* And while, not surprisingly, Eclipse has been used to build a Java development environment, you could build HTML, JSP, XML or other language development systems just as well. In fact, a C/C++ development system has just been released.

But the most surprising thing about Eclipse is that it is Open Source! You can get all the source code of the system and change or add anything you want.

How does this work? Well, the license terms are pretty broad, but you can get the source code and change it, and make a new product and distribute it under your own license. If you distribute the source code, you have to distribute all of it under that license. And this is what IBM has done. Its Websphere Development Studio product is based on Eclipse, but is a product IBM sells in object code form. Needless to say, IBM is a major supporter of the Eclipse project, but so are a lot of other big players.

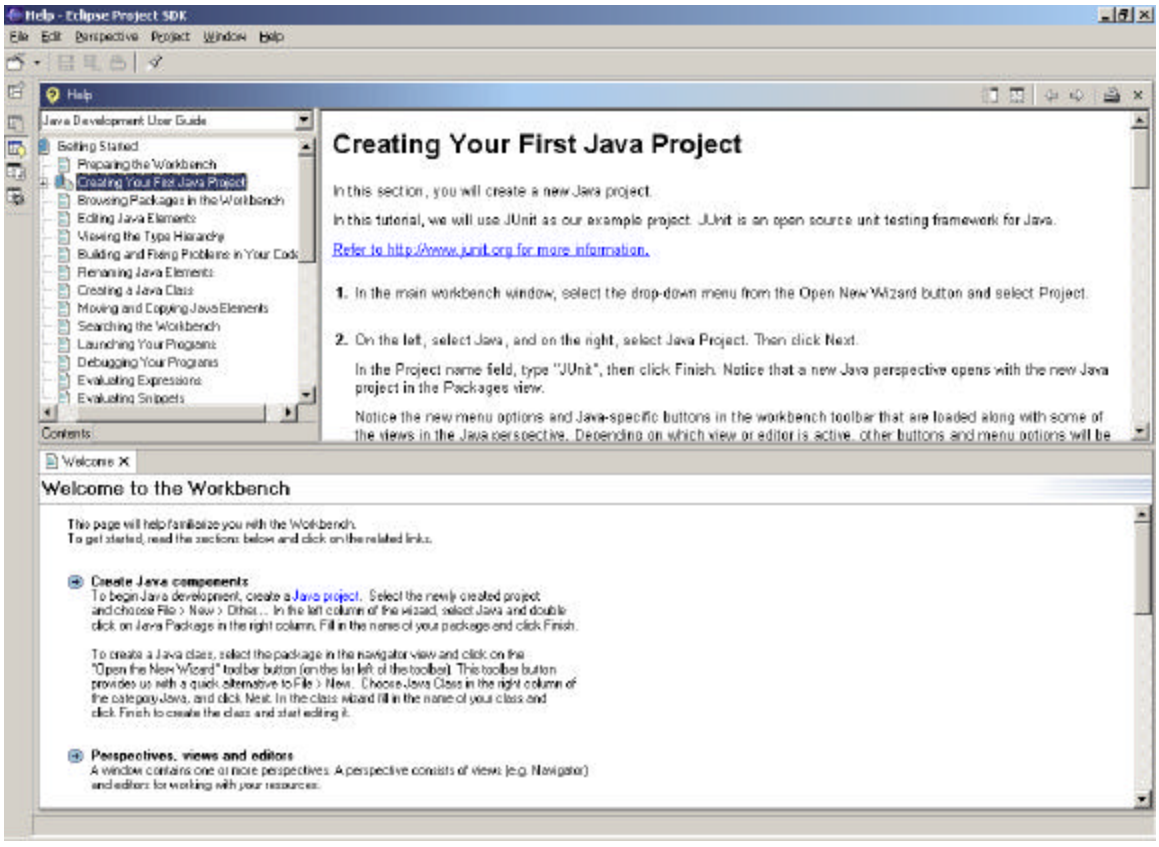

Figure 1 – The first Eclipse screen.

Eclipse is primarily a Java product, but as you can see in Figure 1, it looks pretty professional. One of the reasons for this is that Eclipse uses neither the AWT nor Swing. Instead, Eclipse uses its on GUI widget set called the SWT. The SWT is a graphics library and widget set that is integrated with the native windows system but which provides an "OS-independent API." This means that Eclipse only runs on platforms where the underlying Jini code has been written to implement the SWT functions. However, since these platforms include Win32, and Linux a preponderance of today's workstation systems will run Eclipse.

Actually, I find the features in this free development system pretty amazing. While it doesn't yet have a GUI designer, it does have a very nice debugger and class hierarchy viewer.

Writing Code Using Eclipse

The Eclipse Workbench IDE is quite sophisticated. It features syntax highlighting and keyword completion. In addition, if you right-click on the code window, you can select Format from the pop-up menu, allowing the system to align and "prettify" your code. Some people would probably prefer that it do this automatically and others prefer that it never do it, so this is a nice compromise.

Eclipse actually compiles the code immediately each time you save it, so there is no real compile-edit cycle. If there is a syntax error, it will be marked each time you save the code.

Let's consider the really simple visual Swing Hello program in Figure 2. When you click on the "Click" button it copies the text in the text field to the JLabel widget.

Figure 2 – A simple Swing Hello program.

As you are no doubt aware, this is a really easy program to write, even without a GUI designer. It amounts to the following simple main program.

```
public class TempCalc extends JxFrame
                  implements ActionListener {
      private JTextField jtext;
      private JButton but;
      private JLabel lb;
      public TempCalc() {
            super("Hi window");
            //set up a 3 line grid layout
```

```
            setLayout(new GridLayout(3, 1));
            //create text field with default contents
            jtext = new JTextField("Hi there", 20);
            add(jtext);
            //create empty label
            lb = new JLabel("       ");
            lb.setFont(new Font("Arial", Font.BOLD, 14));
            add(lb);
            //create a button
            but = new JButton("Click");
            add(but);
            //call this action listener on click
            but.addActionListener(this);
            this.setSize(300, 200);
            this.setVisible(true);
      }
      //copy text field into label
      public void actionPerformed(ActionEvent e) {
            lb.setText(jtext.getText());
      }
      static public void main(String argv[]) {
            new TempCalc();
      }

}
```

However, it is only that simple because we created a JxFrame class that buries some of Swing's complexities. In this class we set the Close window box to work, set the look and feel and create a top-level Jpanel that all controls are added to:

```
public class JxFrame extends JFrame {
      private JPanel jp;
      public JxFrame(String title) {
            super(title);
            setCloseClick();  //make close box exit
            setLF();                //set look and feel
            jp = new JPanel();      //put JPanel inside layout
            getContentPane().add(jp);
      }
            //------
      private void setCloseClick() {
            //create window listener to respond to window close click
            addWindowListener(new WindowAdapter() {
                  public void windowClosing(WindowEvent e) {
                        System.exit(0);
                  }
            });
      }
      //------
      private void setLF() {
            // Force SwingApp to come up in the System L&F
            String laf = UIManager.getSystemLookAndFeelClassName();
            try {
                  UIManager.setLookAndFeel(laf);
            }
            catch (UnsupportedLookAndFeelException exc) {
                  System.err.println("No LookAndFeel: " + laf);
            }
```

```
            catch (Exception exc) {
                  System.err.println("Error" + laf + ": " + exc);
            }
      }
}
```

We also created setLayout and add methods that under the covers operate on the panel we have added to the frame:

```
      //puts added components inside jPanel
      public void add(JComponent c) {
            JPanel p = new JPanel();
            jp.add(p);
            p.add(c);
      }
      //------
      //sets the layout of the JPanel
      public void setLayout(LayoutManager lay) {
            if (jp != null)
                  jp.setLayout(lay);
            else
                  super.setLayout(lay);
      }
```

You can see how this program looks in the Eclipse environment in

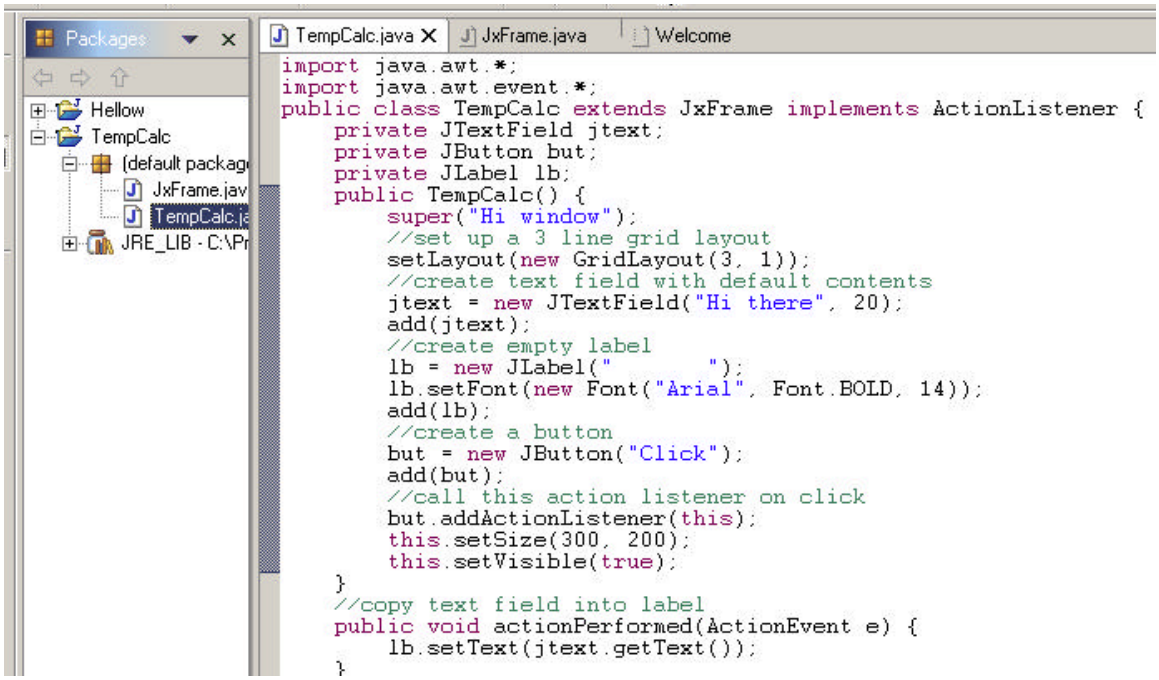

Figure 3 – The Eclipse coding environment, showing the syntax-highlighted Java pane and the class outline.

It also represents it as an outline as you see in Figure 4.

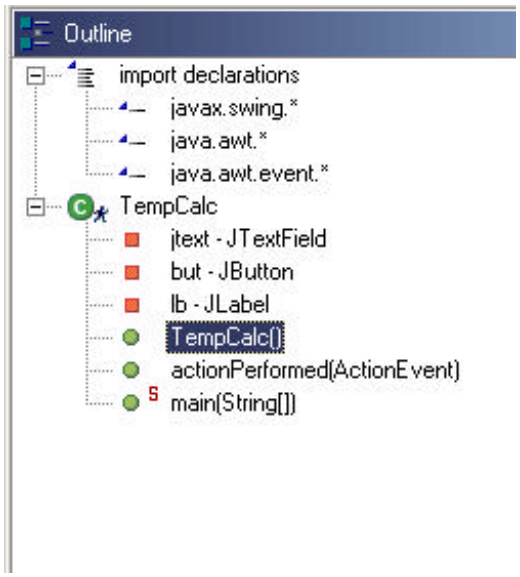

Figure 4 – The Eclipse Java outline of our simple program.

The Eclipse IDE also provides keyword completion. For any class instance, type the dot and, if you want, one or two characters of the beginning of the method name, and press Ctrl/Space. This is illustrated in Figure 5.

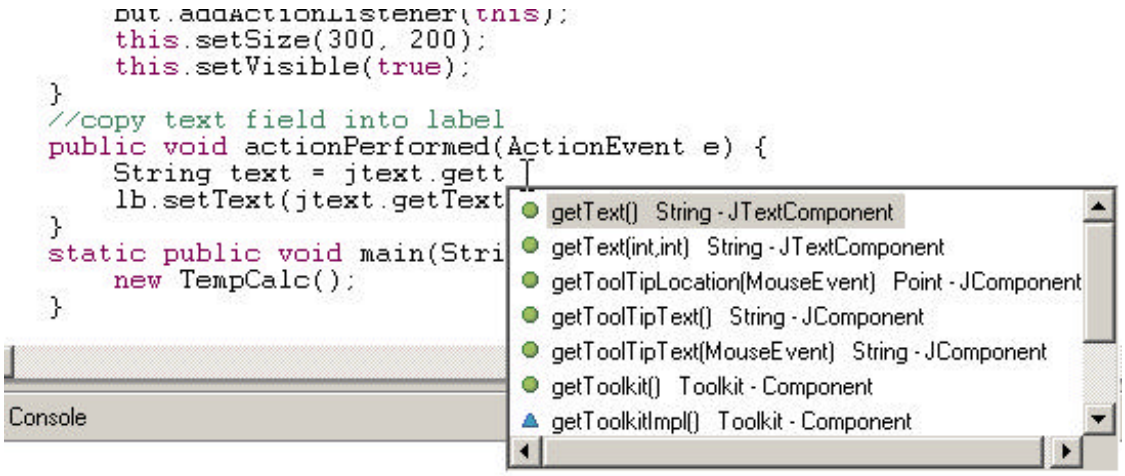

Figure 5 – Keyword completion in the Eclipse IDE.

Debugging

Now, I have a pretty good simplistic development environment using just Visual SlickEdit. It provides projects, make facilities and the ability to run programs form within the IDE. However, it does not provide any way to debug, and while using *System.out.println* stands many of us in good stead, having a real debugger with breakpoints can be very valuable. For instance, when I began writing the JxFrame class I show above, the program kept crashing in the *setLayout* method with a null reference exception. By putting a breakpoint in the setLayout method and in the constructor, I found that the setLayout method was called from the super method before the JPanel was instantiated.

```
//sets the layout of the JPanel
public void setLayout(LayoutManager lay) {
	if (jp != null)
		jp.setLayout(lay);
	else
		super.setLayout(lay);
}
```

Thus, by providing a test for null, I was able to provide for both cases.

Of course, you can also examine all the variables during debugging. When I put a breakpoint in the actionPerfomed method, the variables window showed the data in Figure 6.

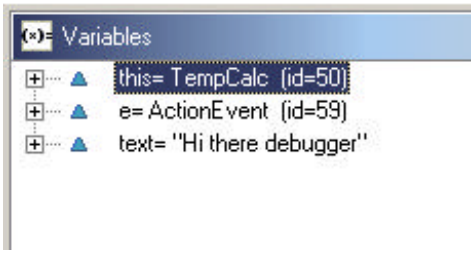

Figure 6 – Debugger variable display when breakpoint in actionPerformed event is encountered.

You can also right-click on any variable and select Display from the pop-up menu, and see that current value of that variable.

Ah, Sweet History of Life

Now, if like most programmers, you occasionally find that you have taken a turn down the wrong pathway and ended up in a programming cul-de-sac, where nothing works anymore, you probably say things like “If only I could go back to the last version that worked and start over.” You probably say a lot of other things, too, but I don’t think Eclipse deals any better with expletives than do other development systems.

But as far as history goes, Eclipse has a stunning, if slightly hidden feature. If you right click in the package list on any filename, and then select Replace with | Local History, you get the stunning display shown in Figure 7.

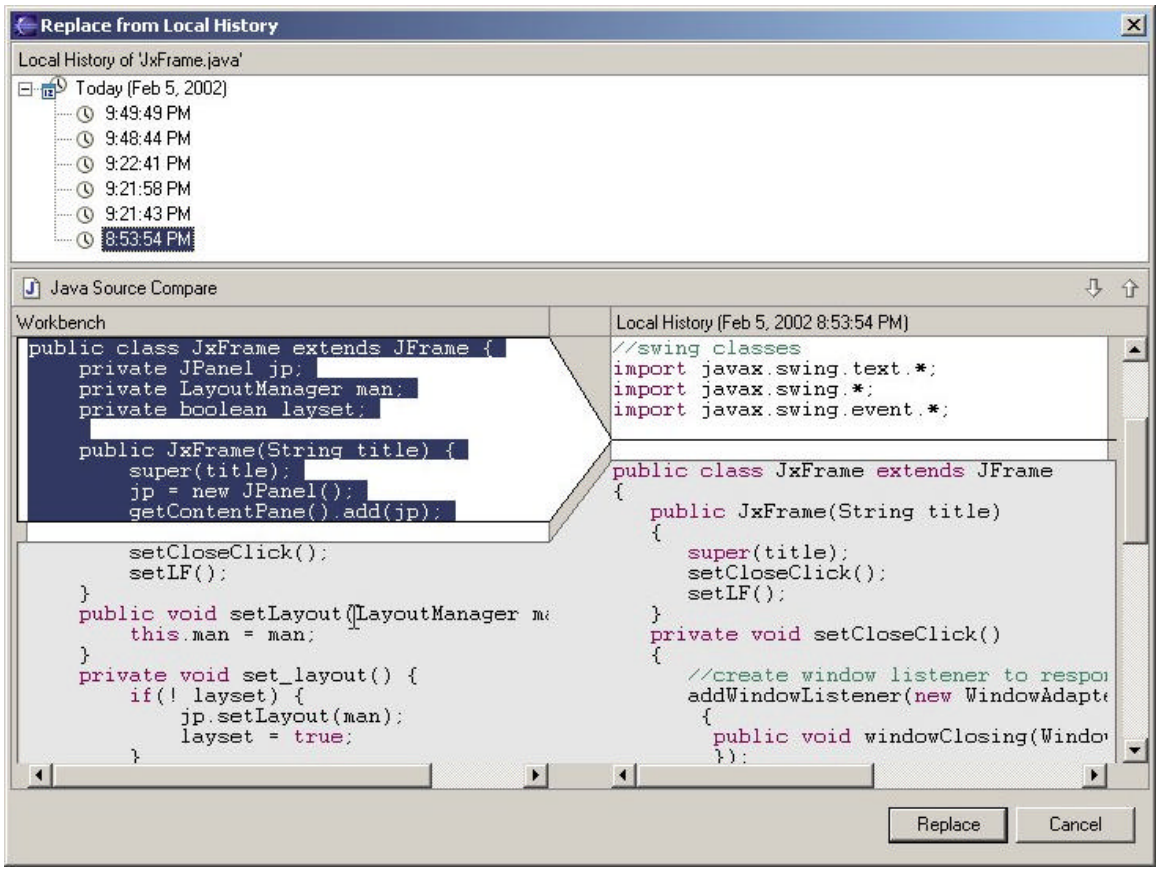

Figure 7 – The recent history of a class. You can select any earlier version and see the changes between it (on the left) and the current version (on the right).

You can see the changes between any earlier version and the current version, and replace it. I found this feature invaluable as I tried to create the simplest possible graphical example. I also found that this local history difference record extends across invocations of Eclipse. You can always find the earlier versions.

Refactoring

We discussed the general idea of refactoring in the August, 2000 column. Refactoring just means rearranging your code to make it more efficient. The Eclipse workbench supports at least two kinds of refactoring: class renaming and method extraction. For example, we might decide that the class JxFrame could be named to something more evocative. So, we just select that class from the package list, and right-click and select rename. The system renames the class and all references to it in the project automatically as you see in Figure 8.

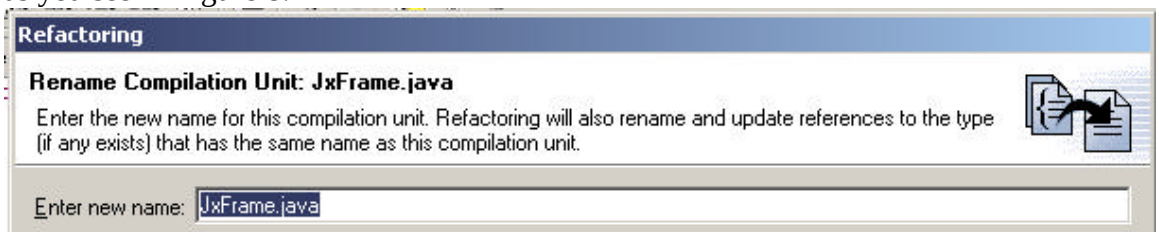

Figure 8 – Renaming a class using Eclipse's refactoring system.

Finally, you can select code from a method and move it to a new method to simplify your code. Just select the code and select Extract Method from the right click menu as shown in Figure 9.

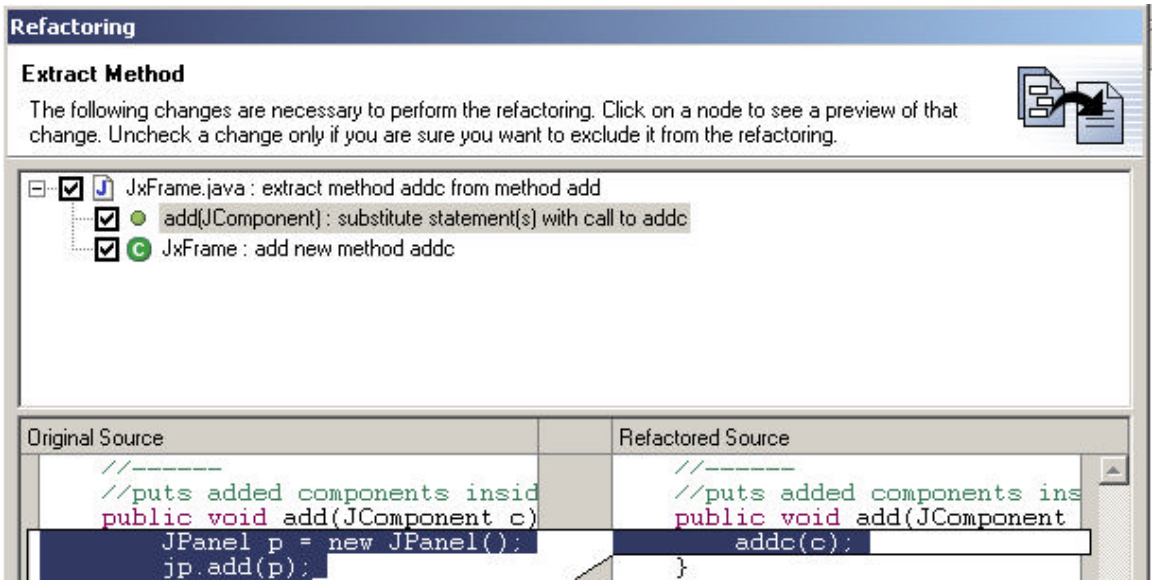

Figure 9 – Extracting a method using Eclipse's refactoring system.

Summary

I think Eclipse is pretty powerful. It runs on two major platforms, and is certainly as powerful as any Linux tool you'll find. Certainly on Win32, there are a lot of nice commercial products, but this one is extremely impressive. And further, there is a lot of synergy in open source development projects, as Lawrence Lessig persuasively describes in his book, *The History of Ideas.* Open source tools develop faster and frequently contain fewer errors because there are so many people contributing to their success.

I'm certainly going to keep Eclipse on my workstations and keep up with its growth and development.

References

1. Fowler, Martin, *Refactoring*, Addison-Wesley, 1999.
2. Lessig, Lawrence, *The Future of Ideas: The Fate of Commons in a Connected World*. Random House, 2001.