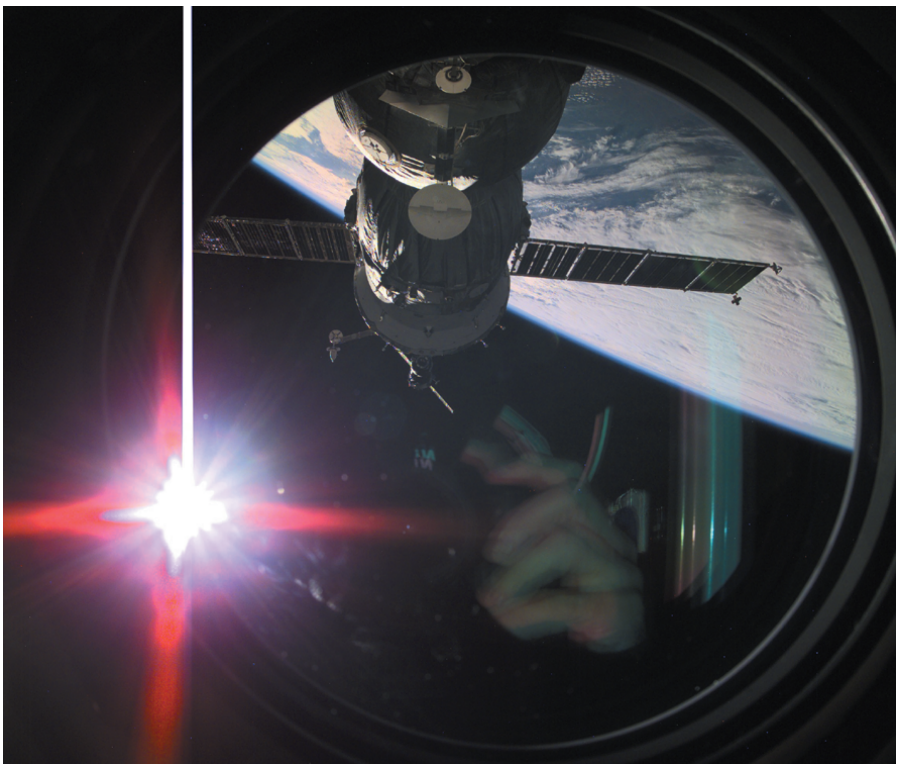
## Eclipse – Architecture and Plugin Development

# At the speed of light

The innovative Eclipse concept for independent plug-in development has been one of the major topics of interest within the software development community for over two years now. The number of products and projects is rising rapidly That prompted us to take a closer look at the basic architecture and experiment with developing our very own extension.

BY MARTIN RAEPPLE



The figures speak for themselves. Over four million downloads, more than 50 commercial products and over 200 Open Source projects with one common denominator: Eclipse as an open and non-proprietary platform for software tools. The assumption that an Integrated Development Environment, IDE, can only succeed by co-operation between a number of software tool manufacturers seems to have been confirmed for the erstwhile initiators, particularly IBM. Big Blue has consistently followed what was a new strategy for the enterprise, ever since the public release of the source code in November 2001.

The Open Source Community based around the mutual development project has been growing ever since. One of the major players is IBM's Swiss subsidiary, Object Technologie International (OTI), whose Technical Director, pattern guru Erich Gamma, was responsible for the technological development of Eclipse's predecessor, VisualAge for Java.

The fact that several other big names participate in the Eclipse Project Management Commission (PMC) such as Borland, Hewlett-Packard, SuSE or Red Hat guarantees a neutral stance. SAP is a recent addition to this illustrious circle. SAP sees Eclipse as a promising addition to a system architecture increasingly dominated by J2EE.

## Installation

Due to the rather large footprint of the ZIP archive (almost 65 Mbytes), there are several regional mirror sites for the Eclipse platform besides the original US based source.

The UK Eclipse community can access the Heanet server based in Ireland [1] for good download speeds and daily updates.

A Java SDK version 1.3 or later is required for the installation; the latest 1.4 version is recommended however, as it is the only version that allows you to use the Java debuggers so-called "Hot Code Replacement" feature in Eclipse.

Previously reserved for Visual Age developers, this feature allows you to swap broken code while running the debugger without having to re-launch the program. And this can save a lot of time, particularly if you are running an application server.

The GTK version of the platform, which some developers prefer for ergonomic reasons, requires at least version 2.0.6 of the GUI library. This can cause package dependency issues, and necessitate time-consuming updates on Red Hat 7.3 for example.

The Motif version is less choosy in this respect; simply add the *libXm.so.2.1* library from the main Eclipse installation directory */etc/ld.so.conf*.

```
/sbin/ldconfig
```

reloads the references to the dynamically linked program libraries. You can then launch the IDE by typing:

```
eclipse -data $HOME
```

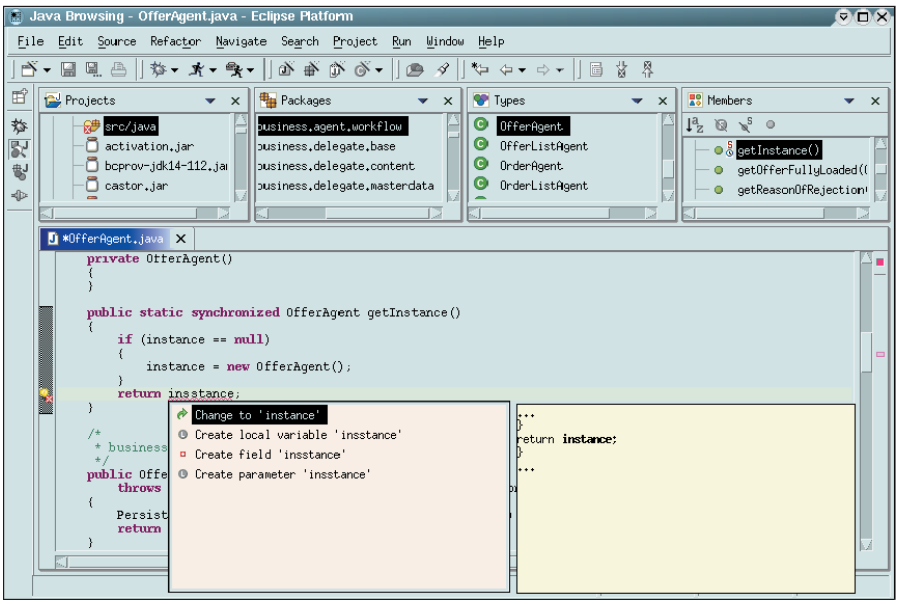



**Figure 1: Quick Fix: Code analysis and debugging while encoding**

The *-data* parameter tells Eclipse to create the workspace relative to the specified path – in this case the system user's home directory.

## Goals and Architecture

Eclipse has no direct primary connection to Java or any other programming language. On the contrary, the platform is aimed to flexibly integrate arbitrary functionality due to a highly modular architecture based on components. The architecture does not necessarily have anything to do with a program's development. Most components, better known as plug-ins, for free or commercial programs today are in fact development tools, that is compilers, editors, modeling tools, or debuggers.

This also applies to the Java programming language support in the form of the so-called Java Development Tools (JDT), an extension which is automatically installed and comprises a collection of plug-ins.

If you stick to the basic functionality of Eclipse as a raw runtime environment for plug-ins, a wide range of different scenarios are imaginable, such as a modularly extensible graphics program that has an basic editor component (a plug-in, of course) and allows the user to install additional graphics filters and formats – also as plug-ins.

This design has been successfully realized by market leaders in the past. This architecture differs from traditional IDEs where the basic functionality is a non-replaceable component of a monolithic runtime environment.

## Functionality

Viewed as a whole, the Eclipse Java IDE presents itself to the developer as an extremely "transparent" developer environment. Users migrating from VisualAge will appreciate the fact that their code no longer ends up in a proprietary repository, but can be stored directly at a user-configurable position on the filesystem. This makes the IDE accessible to fans of *sed*, *grep*, and other external tools that have by no means outlived their usefulness.

On the downside, many facilities appreciated by Visual Age users, such as internal version history and team development, through tight integration with major version control tools like CVS have been retained.

Release 2.1 which became available on March 27 saw further enhancements to one of Eclipse's major benefits, support for code refactoring.

This allows developers to upgrade direct access to local variables to class global, bean conform getting and setting methods just by clicking.

The automatic facility for creating methods for delegate modules [2] is also new. And modifications can be previewed before they are actually implemented.

Quick Fix (see Figure 1) is another remarkable function that not only parses code for errors while it is being written, but can also present the developer with various suggestions on how to remedy the situation, if so desired.

In the light of all this functionality, what reason could a developer possibly have for changing to a commercial variant?

GUI developers will miss a GUI builder, for example. And a powerful Java Server Pages (JSP) editor with syntax highlighting and automatic code completion is also missing. The picture is also marred by the lack of integration with a major J2EE application server.

## Docked – Plug-Ins for J2EE

If you look hard, you should quickly find a solution to the last issue amongst the reams of plug-ins. We would recommend adding a free plug-in by French software

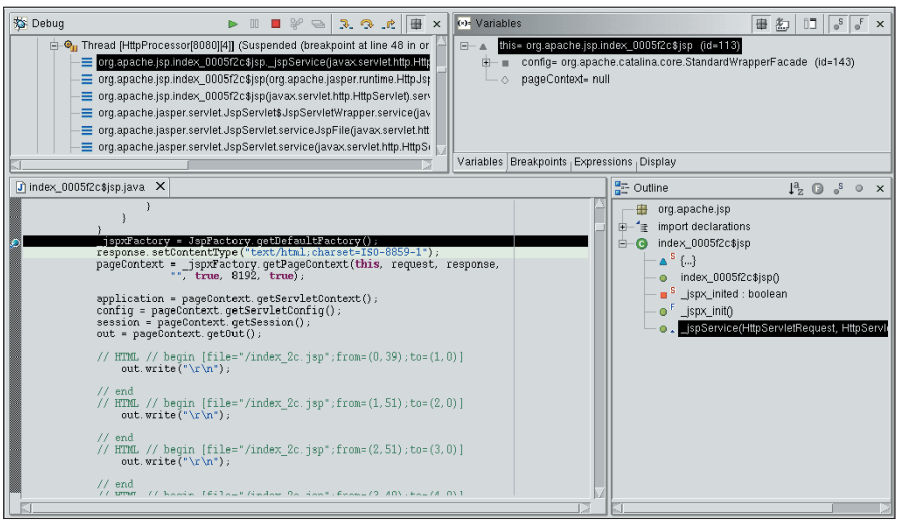



**Figure 2: JSP debugging with the Tomcat plug-in by Sysdeo**

developers, Sysdeo [3], to provide support for the well-known servlet and JSP engine, Tomcat. As is the case for all plug-ins, installation is as easy as falling off a log. Simply unpack the ZIP or TAR archive in the *plugins* subdirectory below Eclipse and re-launch the IDE – finished!

Sysdeo does not provide its own version of Tomcat, instead the plug-in assumes that a Tomcat has been installed. This allows for a high degree of flexibility with respect to the version you use, as the plug-in can be configured to support Tomcat 3.x through 5.x. Once installed, you will not want to do without the facility for starting and stopping the engine directly in Eclipse and the possibility to debug JSPs (see Figure 2).

However, this does require an additional entry for the web application context in your Tomcat configuration file, *server.xml*. The temporary directory for the servlets generated by any JSPs must be re-directed to a project repository accessible to Eclipse. The following example shows how to do so:

```
workDir="<path-to-sources>➥
/work/org/apache/jsp"
```

**Listing 1: plugin.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="XMLProcessing"
   name="XMLProcessing Plug-in"
   version="1.0.1"
   provider-name="Martin Raepple"
   class="net.raepple.plugin.XMLProcessing">

   <runtime>
      <library name="XMLProcessing.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui"/>
   </requires>

   <extension
         point="org.eclipse.ui.popupMenus">
      <objectContribution
            objectClass="org.eclipse.core.resources.IFile"
            nameFilter="*.xml"
            id="XMLProcessing.contribution1">
         <menu
               label="%menu_entry"
               path="additions"
               id="XMLProcessing.menu1">
            <separator
                  name="group1">
            </separator>
         </menu>
         <action
               label="%action_count_elements"
               class="net.raepple.plugin.popup.actions.CountElements"
               menubarPath="XMLProcessing.menu1/group1"
               enablesFor="1"
               id="XMLProcessing.newAction">
         </action>
      </objectContribution>
   </extension>
</plugin>
```

If you add the path to the project properties as an additional source code directory, there is nothing to prevent debugger access to the servlets.

Complex, multi-level J2EE applications will tend to integrate a database that the developer will normally want to keep an eye on. The Quantum DB Plugin [4] proved to be a useful, free alternative. Many major database systems for Linux, such as MySQL, Postgres, or DB2 can easily be integrated as plug-ins for the IDE, allowing the developer to create history files for queries, query spreadsheet metadata, and display a tree-view of the whole database.

## Writing Your Own Plug-Ins

If you are raring to start programming that IDE extension you have been missing for so long, Eclipse rewards your keenness by providing comprehensive support for programming your own plug-ins. Besides the JDTs, a comfortable Plug-In Development Environment, PDE, is installed. The PDE provides easy access via the *File -> New -> Project -> Plug-In Development* menu, including a wizard and templates to introduce developers to Eclipse's component architecture. The PDE Guide, which is available via online help, also provides a useful introduction to this aspect of Eclipse.

Besides the Java classes themselves, the so-called manifest file *plugin.xml* is the central component of each plug-in. It provides details on the general configuration of the smallest executable units, and describes how they integrate with the platform. Integration can occur at various, clearly defined positions within the Eclipse framework. The new editors (*org.eclipse.ui.editors*), menu entries (*org.eclipse.ui.actionSets*), or a workbench preference page (*org.eclipse.ui.preferencePage*) are examples of these extension points.

In the following section we will be looking at an achievable example of using a plug-in to extend the platform's XML functionality. The aim is for users to be able to analyze the elements and their occurrences of an XML document that forms part of a project.

To allow for this, an additional "XML Processing" item, and a "Count Elements" sub-item that provides the

previously described functionality will be displayed in the drop-down menu for files with the *.xml* suffix. We will be using a plug-in that docks at the *org.eclipse.ui.popupMenus* extension point for drop-down or popup menus (see Figure 3) to implement this facility.

The manifest file for Listing 1 is fairly self-explanatory. In addition to superordinate details such as a unique reference ID, the JAR archive with the classes (<*runtime*>) and dependencies (<*requires*>) with respect to other plug-ins, which are also referenced by their Ids, one or multiple extension points (<*extension*>) can be defined. Each extension points has additional attributes that permit precise configuration of the plug-in.

In our example, the drop-down menu entry is only displayed while the mouse is pointing at a file resource (*objectClass = "org.eclipse.core.resources.IFile"*) with the *.xml* extension (*nameFilter = "*.xml"*).

The XML <*action*> tag specifies the class to call for this selection as *class = "net.raepple.plugin.popup.actions.CountElements"*. To impacting performance, Eclipse will not typically load plug-in code immediately on launching, but wait for the code to be called.

The Plug-In Manifest Editor *plugin.xml* provides convenient editing facilities – one of the PDE's major features. If you need to separate the development of plug-ins and accompanying classes from other projects, again you should stipulate the *-data <Plug-In-Workspace>* parameter when launching Eclipse.

The entry point for this plug-in's functionality is the *CountElements* class specified by the <*action*> element when selecting a menu item (see Listing 2). It is automatically created by the Wizard, and implements the *IObject ActionDelegate* Java interface.

At this point, developers would do well to investigate Eclipse's own GUI library (Standard Widget Toolkit, SWT). Suffice to say that the interface needs to implement the *run()* and *selectionChanged()* methods, which are called when the selected object (the XML file) is accessed and allow any further steps to be initialized.

These methods are designed to open a Wizard type dialog box for a selection (see Figure 4), and prompt the user for an element name. The appropriate class, *XMLElementCountWizard* (available only online from [5] to save space), derives from the SWT *Wizard* component and overwrites its *addPages()* and *performFinish()* methods. The former first adds a single dialog page of the *XMLElementCountPage* type which inherits the attributes of the abstract *WizardPage* class. User-specific GUI elements are created by overwriting the *createControl()* method that the Wizard calls.

After the entry has been confirmed, *performFinish()* is called. As the plug-in now has all the information it needs, that is the XML file and the element name, it can use JAXP and the DOM API to ascertain the required value elegantly and with very little effort. The results are presented to the developer in the form of a message (see Figure 5).

In addition to the classes we defined, a main class is automatically generated when creating a plug-in. This class contains methods that provide access to the data of the runtime environment (workbench and workspace).

In the case of multilingual extensions it additionally manages a typical Java resource bundle that parses the content of property files containing language specific texts. In the case of texts that directly affect the plug-in configuration, such as menu items, % can be prefixed in *plugin.xml*.

Doing so will automatically look up a corresponding translation according to Java's search rules for language resources. To allow this, a *plugin.properties* file must exist in the main plug-in directory. In our example, the file has the following content:

```
menu_entry = XML Processing
action_count_elements = Count Elements
```

Additional language support can be provided by files with a corresponding country identifier and a translation (such as *plugin_de.properties*).

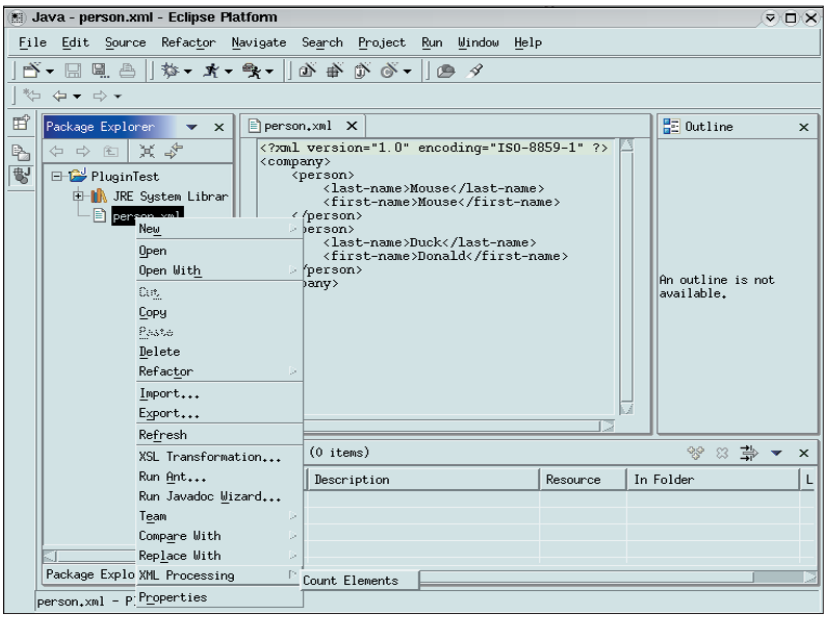



Figure 3: Launching a plug-in via the drop-down menu

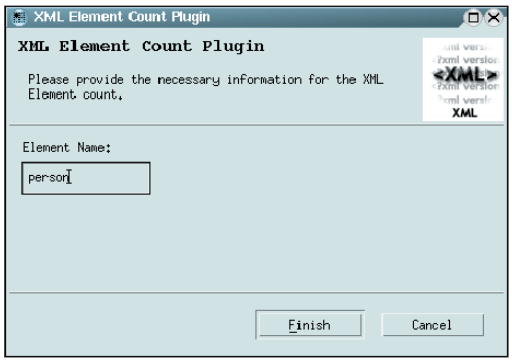



Figure 4: The XML Element Count plug-in meets the Wizard

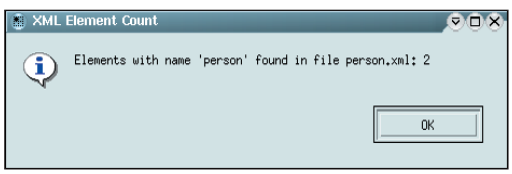



Figure 5: A message informs the developer of the results

Debugging techniques for a plug-in development project are slightly different from the procedures for tradition Eclipse projects.

To shift to debugging mode, you need to launch a second workbench (Run -> Debug as -> Run-time Workbench) as a runtime environment for your test candidate. However, the first workbench is still used to set breakpoints and control the program execution.

After resolving any issues, there is no need to hide your new plug-in from the rest of the world. You could simply compile a Zip or Tar archive and put it on the Web, where other programmers can download it.

The Eclipse Update Manager does provide a slightly more elegant approach. The Update Manager checks dependencies and versions, operating system incompatibility and various other issues when installing new extensions.

This reduces the plug-in users task to simply supplying a URL, where the code can be downloaded to the Update Manager (*Window -> Open Perspective -> Install/Update*). Plug-in providers can setup an update site using *New Project -> Plug-in Development -> Update Site Project*.

**Listing 2: CountElements.java (Excerpt)**

```
public class CountElements implements IObjectActionDelegate
{
   private IFile xmlFile;

   public CountElements()
   {
      super();
   }

   public void setActivePart(IAction action, IWorkbenchPart targetPart)
   {
   }

   public void run(IAction action)
   {
      XMLElementCountWizard wizard = new XMLElementCountWizard(xmlFile);
      WizardDialog dialog = new
WizardDialog(XMLProcessing.getDefault().getWorkbench().getActiveWorkbenc
hWindow().getShell(),wizard);
      dialog.create();
      dialog.open();
   }

   public void selectionChanged(IAction action, ISelection selection)
   {
      this.xmlFile = null;
      if (selection instanceof IStructuredSelection) {
         IStructuredSelection structuredSelection =
(IStructuredSelection) selection;
         if (structuredSelection.size() == 1) {
            Object selectedResource =
structuredSelection.getFirstElement();
            if (selectedResource instanceof IFile)
               this.xmlFile = (IFile) selectedResource;
         }
      }
   }
   public IFile getXmlFile()
   {
      return xmlFile;
   }
}
```

## Conclusion

Open Source works. This is becoming increasingly apparent, thanks to successful projects such as Eclipse, that have managed to achieve the fine balance between the potential that commercial suppliers offer and the interests and goals of the Open Source developer community.

If the technical concept also supports a collaborative approach to software development with an innovative and well thought out component architecture, the resulting product is often well-suited for professional use.

Version 2.1 of Eclipse has seen the product develop into an extremely stable and fully-featured platform that empowers developers, allowing them to create their own extensions quickly and effectively.

In future one would hope for a loosening of the close ties to IBM with respect to platform development. Some initial efforts in non-Java areas such as C++ or C# are already noticeable. ■

**INFO**


[1] Irish based Heanet Eclipse mirror site: *http://eclipse.ftp.heanet.ie/downloads/*

[2] Sun J2EE Patterns Catalog: *http://developer.java.sun.com/developer/restricted/patterns/J2EEPatternsAtAGlance.html*

[3] Sysdeo Tomcat plug-in: *http://www.sysdeo.com/eclipse/tomcatPlugin.html*

[4] Quantum DB plug-in: *http://sourceforge.net/projects/quantum/*

[5] Listings for this article: *ftp://www.linux-magazin.de/pub/listings/magazin/2003/06/Eclipse*



**THE AUTHOR**

*Martin Raepple is a Senior Consultant at Avinci in Frankfurt, Germany. He has authored and co-authored several special-interest publications. His major topics are J2EE, EAI, and security.*