Eclipse: A platform for integrating development tools

by J. des Rivières
J. Wiegand

Modern n-tier applications are developed using components implemented in many different technologies, including HTML, Java™, JavaServer Pages™ (JSP™), Enterprise JavaBeans™, connectors, COBOL or PL/1 programs, and relational database schemas. Creating an effective integrated development environment (IDE) for use in programming these applications presents some special challenges because a large number of different tool technologies have to be tightly integrated in support of development task flows. In order to meet these challenges, the Eclipse Platform was designed to serve as the common basis for diverse IDE-based products, providing open APIs (application programming interfaces) to facilitate this integration. This paper describes the overall architecture of the Eclipse Platform and the www.eclipse.org open source organization and consortium created to facilitate broad industry adoption of this platform.

Customers developing applications need a variety of different tools from various tool vendors to support the full software development life cycle. Developers can be more productive and effective if these tools work well together. Integrated development environments (IDEs) can aid in the integration of tools to facilitate the software development process and will succeed in doing so to the extent that the community of tool developers can be influenced to develop tools in ways that increase the likelihood of their interoperation with other tools.

The Eclipse Platform was created to address this issue by providing a common platform for diverse IDE-based products and facilitate their integration. The first part of this paper introduces and gives an historical perspective of IDEs, followed by a description of the technical aspects of the Eclipse Platform. In the second part, we discuss the efforts of the Eclipse community of tool developers to make the Eclipse Platform ubiquitous.

A brief history of commercial IDEs. In the early days of programming, the only software development tools that programmers really needed were a compiler for the language they were programming in and a link editor and loader to combine the compiled files into executable form. Programs were composed offline; debugging was done primarily with output statements inserted in the code. With the advent of time-sharing, programs started to be written and debugged interactively by using the computer as well. The earliest commercial IDEs were built for the programming languages BASIC[1] and APL.[2] The Emacs editor[3] is arguably the first language-neutral, extensible IDE, and to this day it maintains a loyal following who prefer it to the GUI (graphic user interface) -based IDEs that followed it.

The rise of personal computers saw the creation of a number of commercial IDEs geared towards pop-

ular programming languages, including Pascal (e.g., Borland Turbo Pascal**[4]), LISP (e.g., Xerox Interlisp[5]), Smalltalk (e.g., PARCPlace Smalltalk[6]), and C++ (e.g., Lucid Energize[7]). By the early 1990s, an IDE was standard fare for most programming languages. When the Java** programming language arrived in 1995, the market was soon flooded with Java IDE products, including Borland JBuilder**,[8] Symantec Visual Café,[9] Microsoft Visual J++**,[10] and IBM VisualAge* for Java.[11]

The use of IDEs is prevalent for one simple reason: they make software developers more productive. Because of this utilitarian appeal, IDEs continue to evolve as people find new ways to improve productivity. Over time, the set of tools integrated into the IDE has expanded from simple editors, compilers, and debuggers to include incremental compilers, browsers that present the program in meaningful ways (e.g., classes and methods arranged in a subclass hierarchy), automatic code completion, and visual editors for creating graphical UIs (user interfaces). This trend towards more sophisticated and powerful language-specific tools continues,[12] and nowadays includes facilities like built-in editor support for re-factoring code.

Another way to improve productivity is for the IDE to cover more of the software development life cycle. For example, most commercial IDEs include optional version and configuration management for source code files because this is a key concern for working programmers. IDEs are rapidly expanding into the other areas, such as software design with UML** (Unified Modeling Language) modeling tools (e.g., Rational Rose**[13]).

In reality, working programmers usually do much more than write and debug code in a single programming language. It is commonplace for a programmer to create and manipulate many non-code artifacts such as HTML (Hypertext Markup Language) pages. This means that the programmer ends up using additional tools not integrated with the IDE. In order to address these needs, most modern commercial IDEs are designed to be open and extensible so that new tools can be supplied by third parties (i.e., someone other than the IDE vendor). This is done by providing a mechanism for the IDE to discover add-in tools on start up, and by publishing APIs (application programming interfaces) for use by these tools to integrate their functions with the IDE. For example, Borland JBuilder has an Open Tools API, and IntelliJ** IDEA has a product called Plug-in API.[14] This kind of open-ended extensibility is essential in the commercial IDE arena because no IDE vendor could possibly provide a sufficient set of useful tools to satisfy all customer needs. Which third party tool will be bundled as an add-in for a particular IDE is determined by market forces.

Some IDEs start from a language-specific base and expand from there. For instance, Oracle JDeveloper Suite**[15] is a Java-centric IDE which expanded into UML modeling, well beyond a narrow Java focus. NetBeans** IDE[16] started as Java-specific, but later evolved into a language-neutral, open source IDE that is used within commercial IDE products, including Sun ONE Studio.[17] Microsoft Visual Studio** .NET[18] supports multiple languages within the same IDE and provides extensive language-neutral as well as language-specific APIs for use by tools. In the effort to expand their horizons, IDEs must overcome any language-specific biases which may have been built into them.

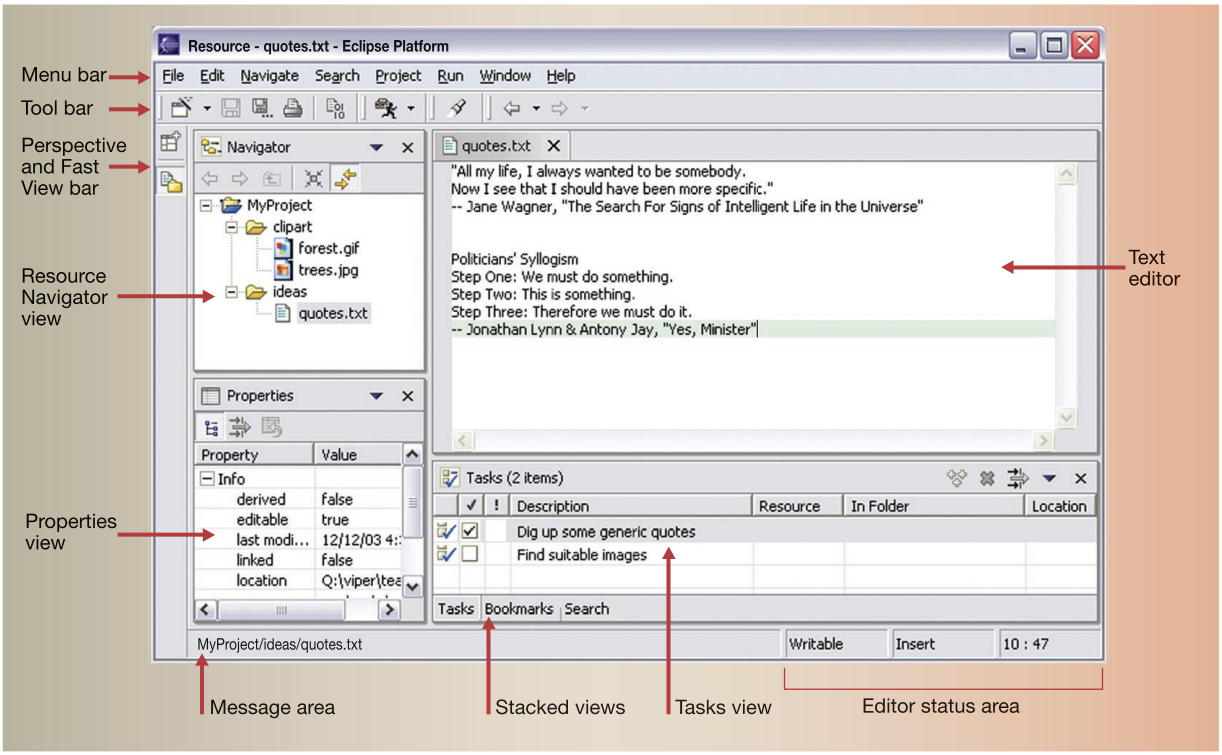
Eclipse Platform technical overview

In the following section, we give a high-level description of the Eclipse Platform.[19] The Eclipse Platform is an open-ended, language-neutral IDE. The open source Eclipse Platform 1.0[20] was released in late 2001 and began appearing in commercial products shortly thereafter, the first being IBM WebSphere* Studio Application Developer 4.0.[21] The Eclipse Platform is aptly described as "an IDE for anything and for nothing in particular."

Figure 1 shows a screen capture of the main workbench window as it looks with only the standard generic components that are part of the Eclipse Platform. The Navigator view (Figure 1, top left) shows the files in the user's workspace; the text editor (top right) shows the contents of a file; the Tasks view (bottom right) shows a list of to-dos; the Properties view (bottom left) shows various properties of the file selected in the Navigator view.

Although the Eclipse Platform has much built-in functionality, most of that functionality is very generic. It takes additional tools to extend the platform to work with new content types, to do new things with existing content types, and to focus the generic functionality on something specific. The Eclipse Platform is built on a mechanism for discovering, integrating, and running modules called *plug-ins*. A tool provider writes a tool as a separate plug-in that operates on

Figure 1 Eclipse Platform user interface



files in the workspace and surfaces its tool-specific UI in the workbench. When the platform is launched, the user is presented with an IDE composed from the set of available plug-ins. The quality of the user experience depends significantly on how well the tools integrate with the platform and how well the various tools work with each other.

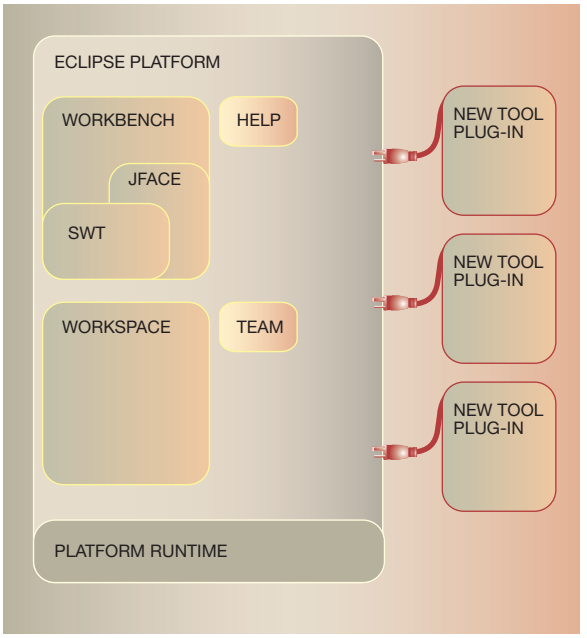
Platform design goals. The Eclipse Platform was designed and built to meet the following requirements:

- Support the construction of a variety of tools for application development.
- Support an unrestricted set of tool providers, including independent software vendors (ISVs).
- Support tools to manipulate arbitrary content types (e.g., HTML, Java, C, JSP** [JavaServer Pages**], EJB** [Enterprise JavaBeans**], XML [eXtensible Markup Language], and GIF [Graphic Interchange Format]).
- Facilitate seamless integration of tools within and across different content types and tool providers.
- Support both GUI and non-GUI-based application development environments.
- Run on a wide range of operating systems, including Windows** and Linux**.
- Capitalize on the popularity of the Java programming language for writing tools.

The Eclipse Platform's principal role is to provide tool providers with mechanisms to use and rules to follow that lead to seamlessly integrated tools. These mechanisms are exposed through well-defined API interfaces, classes, and methods. The platform also provides useful building blocks and frameworks that facilitate developing new tools. Figure 2 shows the major components and APIs of the Eclipse Platform.

Platform runtime and plug-in architecture. A plug-in is the smallest unit of Eclipse Platform function that can be developed and delivered separately. Usually a small tool is written as a single plug-in, whereas a complex tool has its functionality split across several plug-ins. Except for a small kernel known as the

Figure 2 Eclipse Platform architecture



Platform Runtime, all of the Eclipse Platform's functionality is located in plug-ins.

Plug-ins are coded in the Java language. A typical plug-in consists of Java code in a JAR (Java archive) library, some read-only files, and other resources such as images, Web templates, message catalogs, native code libraries, and so forth. Some plug-ins do not contain code at all. One such example is a plug-in that contributes on-line help in the form of HTML pages. A single plug-in's code libraries and read-only contents are located together in a directory in the file system. There is also a mechanism that permits a plug-in to be synthesized from several separate fragments, each in its own directory. This is the mechanism used to deliver separate language packs for an internationalized plug-in.

Each plug-in has a *manifest* file declaring its interconnections to other plug-ins. The interconnection model is simple: a plug-in declares any number of named *extension points*, and any number of *extensions* to extension points in other plug-ins. An extension point may have a corresponding API interface. Other plug-ins contribute implementations of this interface through extensions to this extension point. For example, the workbench plug-in declares an extension point for user preferences. Any plug-in can contribute its own user preferences by defining extensions to this extension point.

On start-up, the Platform Runtime discovers the set of available plug-ins, reads their manifest files, and builds a plug-in registry. The platform matches extension declarations by name with their corresponding extension point declarations. Any problems, such as extensions to missing extension points, are detected and logged. The resulting plug-in registry is available through the platform API. Plug-ins cannot be added after start-up.

Plug-in manifest files contain XML. An extension point may declare additional specialized XML element types for use in the extensions. This allows the plug-in supplying the extension to communicate arbitrary information to the plug-in declaring the corresponding extension point. Moreover, manifest information is available from the plug-in registry without activating the contributing plug-in or loading any of its code. This property is critical to supporting a large base of installed plug-ins, only some of which are needed in any given user session. Until a plug-in's code is loaded, it has a negligible memory footprint and impact on start-up time. Using an XML-based plug-in manifest file also makes it easier to write tools that support plug-in creation. The Plug-In Development Environment (PDE), which is included in the Eclipse SDK (software development kit), is such a tool.

A plug-in is activated when its code actually needs to be run. Once activated, a plug-in uses the plug-in registry to discover and access the extensions contributed to its extension points. For example, the plug-in declaring the user preference extension point can discover all contributed user preferences and access their display names to construct a preference dialog. This can be done by using only the information from the registry, without having to activate any of the contributing plug-ins. The contributing plug-in will be activated when the user selects a preference from a list. Activating plug-ins in this manner does not happen automatically; there are a small number of API methods for explicitly activating plug-ins. Once activated, a plug-in remains active until the platform shuts down. Each plug-in is furnished with a subdirectory in which to store data specific to the plug-in; this mechanism allows a plug-in to retain important state information between runs.

The Platform Runtime declares a special extension point for applications. When an instance of the platform is launched, the name of an application is specified through the command line; the only plug-in that gets activated initially is the one that declares that application.

By determining the set of available plug-ins for start-up and by supporting a significant exchange of information between plug-ins without having to activate any of them, the platform can provide each plug-in with a rich source of pertinent information about the context in which it is operating. This context cannot change while the platform is running, so there is no need for complex life-cycle events to inform plug-ins of context changes. A lengthy start-up sequence is avoided, as are many bugs stemming from unpredictable plug-in activation order.

The Eclipse Platform is run by a single invocation of a standard JVM** (Java Virtual Machine). Each plug-in is assigned its own Java class loader which is solely responsible for loading its classes (and Java resource bundles). Each plug-in explicitly declares its dependence on the other plug-ins from which it expects to directly access classes. A plug-in controls the visibility of the public classes and interfaces in its libraries. This information is declared in the plug-in manifest file; the visibility rules are enforced at runtime by the plug-in class loaders.

The plug-in mechanism is used to partition the Eclipse Platform itself. Indeed, separate plug-ins provide the workspace, the workbench, and so on. Even the Platform Runtime has its own plug-in. Non-GUI configurations of the platform may simply omit the workbench plug-in and the other plug-ins that depend on it.

The Eclipse Platform's update manager downloads and installs new features or upgraded versions of existing features (a feature being a group of related plug-ins that get installed and updated together). The update manager constructs a new configuration of available plug-ins to be used the next time the Eclipse Platform is launched. If the result of upgrading or installing proves unsatisfactory, the user can roll back to an earlier configuration.

The Eclipse Platform runtime also provides a mechanism for extending objects dynamically. A class that implements an "adaptable" interface declares its instances open to third-party behavior extensions. An adaptable instance can be queried for the adapter object that implements an interface or class. For example, workspace resources are adaptable objects; the workbench adds adapters that provide a suitable icon and text label for a resource. Any party can add behavior to existing types (both classes and interfaces) of adaptable objects by registering a suitable adapter factory with the platform. Multiple parties can independently extend the same adaptable objects, each for a different purpose. When an adapter for a given interface is requested, the platform identifies and invokes the appropriate factory to create it. The mechanism uses only the Java type of the adaptable object (it does not increase the adaptable object's memory footprint). Any plug-in can exploit this mechanism to add behavior to existing adaptable objects and to define new types of adaptable objects for other plug-ins to use and possibly extend.

Workspaces. The various tools which plug in to the Eclipse Platform operate on regular files in the user's workspace. The workspace consists of one or more top-level projects, where each project maps to a corresponding user-specified directory in the file system. The different projects in a workspace may map to different file system directories or drives although, by default, all projects map to sibling subdirectories of a single workspace directory.

A mechanism in the Eclipse Platform allows a tool to tag a project in order to give it a particular personality, or *nature*. For example, "Web site nature" tags are associated with a project that contains the static content for a Web site, and "Java nature" tags are associated with a project that contains the source code for a Java program. The project nature mechanism is open. Plug-ins may declare new project natures and provide code for configuring projects with that nature. A single project may have as many natures as required. This affords a way for tools to share a project without having to know about each other.

Each project contains files that are created and manipulated by the user. All files in the workspace are directly accessible by the standard programs and tools of the underlying operating system. Tools integrated with the platform are provided with APIs for dealing with workspace resources (projects, files, and folders). Workspace resources are represented by adaptable objects so that other parties can extend their behavior.

To minimize the risk of accidentally losing files, a low-level workspace history mechanism keeps track of the previous contents of any files that have been

changed or deleted by integrated tools. The user controls how the history is managed by means of space- and age-based preference settings. The workspace provides a marker mechanism for annotating resources. Markers are used to record diverse annotations, such as compiler error messages, to-do list items, bookmarks, search hits, and debugger breakpoints. The marker mechanism is open. Plug-ins can declare new marker subtypes and control whether they should be saved between runs.

The platform provides a general mechanism that allows a tool to track changes to workspace resources. By registering a "resource change listener," a tool is ensured to receive after-the-fact notifications of all resource creations, deletions, and changes to the contents of files. The platform defers the event notification until the end of a batch of resource manipulation operations. Event reports take the form of a tree of resource changes (or "deltas") that describe the effect of the entire batch of operations in terms of net resource creations, deletions, and changes. Resource deltas also provide information about changes to markers.

Resource tree deltas are particularly useful and efficient for tools that display resource trees because each delta points out where the tool may need to add, remove, or refresh on-screen widgets (small graphic elements). In addition, because a number of semi-independent tools may be operating on the resources of a project at the same time, this mechanism allows one tool to detect the activity of another in the vicinity of specific files or file types in which it has an interest.

Tools like compilers and link checkers must apply a coordinated analysis and transformation of thousands of separate files. The platform provides an *incremental project builder* framework. The input to an incremental builder is a resource tree delta capturing the net resource differences since the last build. Sophisticated tools may use this mechanism to provide scalable solutions. The platform allows several different incremental project builders to be registered for the same project and provides ways to trigger project- and workspace-wide builds. An optional workspace auto-build feature automatically triggers the necessary builds after each resource modification operation (or batch of operations).

The workspace save-restore process is open to participation from plug-ins wishing to remain coordinated with the workspace across sessions. A two-phase save process ensures that the important states of the various plug-ins are written to disk as an atomic operation. In a subsequent session, when an individual plug-in gets reactivated and rejoins the save-restore process, it is passed a workspace-wide resource delta describing the net resource differences since the last save in which it participated. This allows a plug-in to carry forward its saved state while making the necessary adjustments to accommodate resource changes made while it was deactivated.

Workbench and UI toolkits. The Eclipse Platform UI is built around a workbench that provides the overall structure and presents an extensible UI to the user. The workbench API and implementation are built from two toolkits:

- SWT (Standard Widget Toolkit)—a widget set and graphics library integrated with the native window system but with an OS (operating system) -independent API.
- JFace—a UI toolkit implemented using the SWT, which simplifies common UI programming tasks.

Standard Widget Toolkit. The Standard Widget Toolkit (SWT) provides a common OS-independent API for widgets and graphics implemented in a way that allows tight integration with the underlying native window system. The entire Eclipse Platform UI and the tools that plug in to it use SWT for presenting information to the user.

A perennial issue in widget toolkit design is the tension between portable toolkits and native window system integration. The Java AWT (Abstract Window Toolkit) provides low-level widgets such as lists, text fields, and buttons, but no high-level widgets such as trees or rich text. AWT widgets are implemented directly with native widgets on all underlying window systems. Building a UI using AWT alone means programming for the least common denominator of all OS window systems. The Java Swing toolkit addresses this problem by emulating widgets like trees, tables, and rich text. Swing also provides look-and-feel emulation layers that attempt to make applications look like the underlying native window system. However, the emulated widgets invariably lag behind the look and feel of the native widgets, and the user interaction with emulated widgets is usually different enough to be noticeable, making it difficult to build applications that compete head-on with applications developed specifically for a particular native window system.

The SWT addresses this issue by defining a common API that is available across a number of supported window systems. For each different native window system, the SWT implementation uses native widgets wherever possible; where no native widget is available, the SWT implementation provides a suitable emulation. Common low-level widgets such as lists, text fields, and buttons are implemented natively everywhere. But some generally useful higher-level widgets may need to be emulated on some window systems. For example, the SWT toolbar widget is implemented as a native toolbar widget on Windows** and as an emulated widget on Motif**. This strategy allows the SWT to maintain a consistent programming model in all environments, while allowing the underlying native window system's look and feel to shine through to the greatest extent possible.

The SWT also exposes native window system-specific APIs in cases where a particular underlying native window system provides a unique and significant feature that is unavailable on other window systems. Windows ActiveX** is a good example of this. Window system-specific APIs are segregated into aptly named packages to indicate that they are inherently nonportable.

Tight integration with the underlying native window system is not strictly a matter of look and feel. The SWT also interacts with native desktop features, such as the "drag and drop" function, and can use components developed with OS component models, such as Windows ActiveX controls. Internally, the SWT implementation provides separate and distinct implementations for each native window system. The Java native libraries are completely different, with each exposing the APIs specific to the underlying window system. (Contrast this to the Java AWT, which locates window system-specific differences in the C code implementation of a common set of Java native methods.) Because no special logic is buried in the native methods, the SWT implementation is expressed entirely in Java code. Nevertheless, the Java code looks familiar to the native OS developer. Any Windows programmer would find the Java implementation of the SWT for Windows instantly familiar because it consists of calls to the Windows API that they already know from programming in C; likewise, for a Motif programmer looking at the SWT implementation for Motif. This strategy greatly simplifies implementing, debugging, and maintaining the SWT because it allows all interesting development to be done in the Java language. Of course, this is of no direct concern for ordinary clients of the SWT because these native methods are completely hidden behind the window system-independent SWT API.

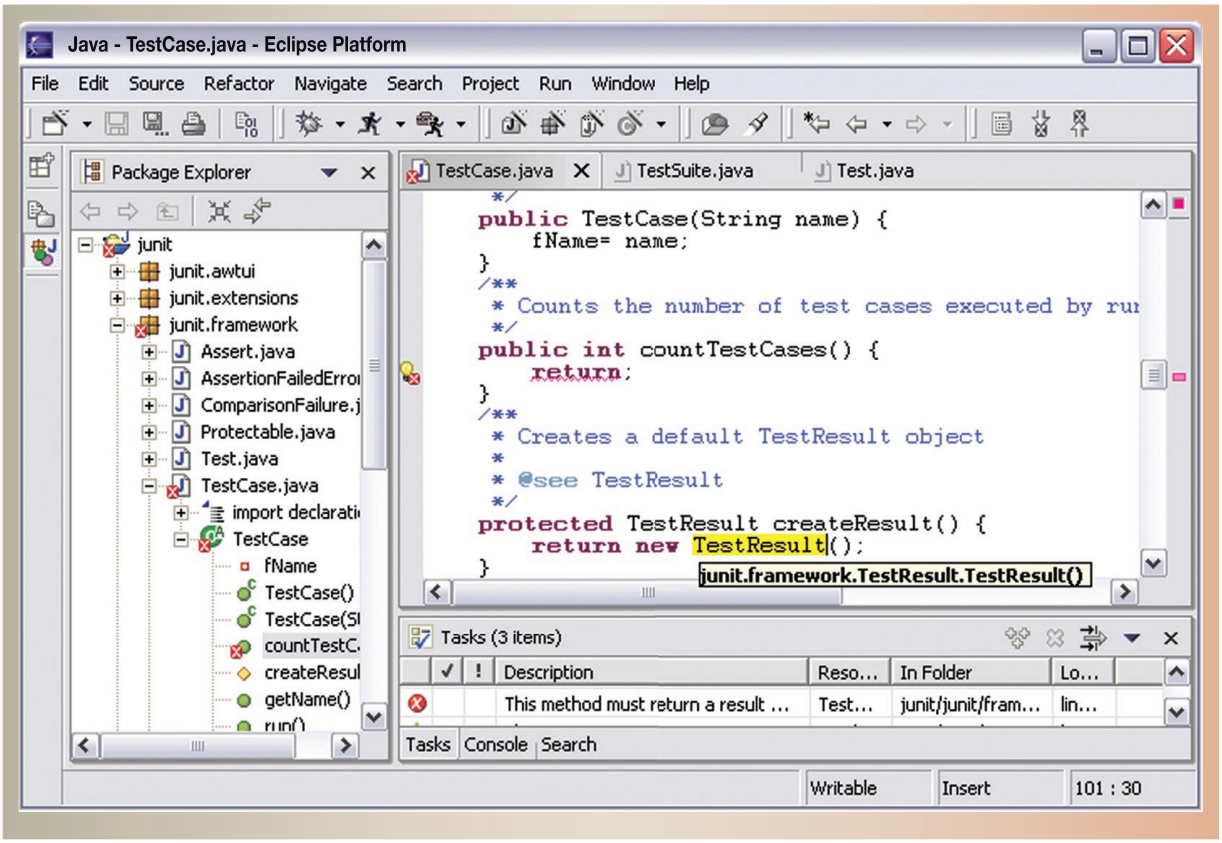
JFace. JFace is a UI toolkit with classes for handling many common UI programming tasks. JFace is window-system-independent in both its APIs and implementation and is designed to work with SWT without hiding it. JFace includes the usual UI toolkit components of image and font registries, dialog, preference, and wizard frameworks, and progress reporting for long-running operations. Two of its more interesting features are actions and viewers.

The action mechanism allows user commands to be defined independent of their exact whereabouts in the UI. An action represents a command that can be triggered by the user through a button, menu item, or item in a tool bar. Each action knows its own key UI properties (label, icon, tool tip, etc.), which are used to construct appropriate widgets for presenting the action. This separation allows the same action to be used in several places in the UI, making it is easy to change where an action is presented in the UI without having to change the code for the action itself.

Viewers are model-based adapters for certain SWT widgets. Viewers handle common behavior and provide semantics of a higher level than those available from the SWT widgets. The standard viewers for lists, trees, and tables support populating the viewer with elements from the client's domain and keeping the widgets in synchronization with changes to that domain. The standard text viewer provides a document model to the client and manages the conversion of the document to the information required by the SWT-styled text widget. Multiple viewers can be open on the same model or document; all are updated automatically when the model or document changes in any of them.

Eclipse workbench. Unlike SWT and JFace, which are both general purpose UI toolkits, the workbench provides the UI personality of the Eclipse Platform and supplies the structures for the interaction of the user and the tools. Because of this central and defining role, the workbench is synonymous with the Eclipse Platform UI as a whole and with the main window the user sees when the platform is running. The workbench API depends on the SWT API, and to a lesser extent on the JFace API. The workbench implementation is built by using both SWT and JFace; Java AWT and Swing are not used.

Figure 3 Workbench user interface showing Java perspective



The Eclipse Platform UI paradigm is based on editors, views, and perspectives. From the user's standpoint, a workbench window consists visually of views and editors (see Figure 1). Perspectives manifest themselves in the selection and arrangements of editors and views visible on the screen. Editors allow the user to open, edit, and save objects. They follow an open-save-close life cycle much like file system tools, but are more tightly integrated into the workbench. When active, an editor can contribute actions to the workbench menus and tool bar. The platform provides a standard editor for text resources; more specific editors are supplied by other plug-ins.

Views provide information about an object that the user is working with in the workbench. A view may assist an editor by providing information about the document being edited. For example, the standard content outline view uses a JFace tree viewer to present a structured outline for the content of the active editor if one is available. A view may augment other views by providing information about the currently selected object. For example, the standard properties view presents the properties of the object selected in another view. Views have a simpler life cycle than editors: modifications made in a view (such as changing a property value) are generally saved immediately, and the changes are reflected immediately in other related parts of the UI. The platform provides several standard views; additional views are supplied by other plug-ins.

A workbench window can have several separate *perspectives*, only one of which is visible at any given moment. Each perspective has its own views and editors that are arranged (tiled, stacked, or detached) for presentation on the screen. Some may be hidden at any given moment. Several different types of views and editors can be open at the same time within a perspective. A perspective controls initial view vis-

ibility, layout, and action visibility. The user can quickly switch perspectives to work on a different task and can easily rearrange and customize a perspective to better suit a particular task. The platform provides standard perspectives for general resource navigation, on-line help, and team support tasks. Additional perspectives are supplied by other plug-ins.

Tools integrate into this "editors, views, and perspectives" UI paradigm in well-defined ways. The main extension points allow tools to augment the workbench by adding new types of editors, new types of views, and new perspectives, which arrange old and new views to suit new user tasks. The platform's standard views and editors are all included by use of these mechanisms. Tools may also augment existing editors, views, and perspectives by adding new actions to an existing view's local menu and tool bar, adding new actions to the workbench menu and tool bar when an existing editor becomes active, adding new actions to the pop-up content menu of an existing view or editor, or adding new views, action sets, and shortcuts to an existing perspective.

The platform takes care of all aspects of workbench window and perspective management. Editors and views are automatically instantiated as needed and disposed of when no longer needed. The display labels and icons for actions contributed by a tool are listed in the plug-in manifest so that the workbench can create menus and tool bars without activating the contributing plug-ins. The workbench does not activate the plug-in until the user attempts to use the functionality that the plug-in provides.

After an editor or view becomes an active part of a perspective, it can use workbench services for tracking activation and selection. The "part service" tracks view and editor activation within the perspective, reporting activation and deactivation events to registered listeners. A view or editor can also register with the *selection service*. The selection service feeds selection change events to all parties that have registered interest. This is how, for example, the standard properties view is notified of the domain object currently selected in the active editor or view.

UI integration. Tools written in the Java language using the Eclipse Platform APIs achieve the highest level of integration with the platform. At the other extreme, external tools launched from within the platform must open their own separate windows in order to communicate with the user and must access user data by means of the underlying file system. Their integration is therefore very loose, especially at the UI level. In some environments, the Eclipse Platform also supports levels of integration between these extremes.

- The workbench has built-in support for embedding any OLE (Object Linking and Embedding) document as an editor (for Windows only). This option provides tight UI integration.
- A plug-in tool can implement a container that bridges the Eclipse Platform API to an ActiveX control so that it can be used in an editor, view, dialog, or wizard (for Windows only). The SWT provides the requisite low-level support. This option provides tight UI integration.
- A plug-in tool can use AWT or Swing to open separate windows.[22] This option provides loose UI integration but allows tight integration below the UI level.

Team support. The Eclipse Platform allows a project in the workspace to be placed under version and configuration management with an associated team repository. The platform has extension points and a repository provider API that allow new kinds of team repositories to be plugged in. The function provided by a particular team repository product invariably affects the user's workflow; for example, by adding explicit steps for retrieving files from the repository, for returning updated files to the repository, and for comparing different file versions. The exact effect on the user's workflow varies somewhat for each kind of repository. Accordingly, the Eclipse Platform takes a hands-off view and allows each team repository provider to define its own workflow so that users already familiar with the team repository product can quickly learn to use it from within Eclipse.

The platform supplies basic hooks to allow a team repository provider to intervene in certain operations that manipulate resources in a project. These hooks provide good support for both optimistic and pessimistic models. At the UI level, the platform supplies placeholders for certain actions, preferences, and properties, but leaves it to each repository provider to define these UI elements. There is also a simple, extendable configuration wizard that lets users associate projects with repositories, which each repository provider can extend with UI elements for collecting information specific to that kind of repository.

Multiple team repository providers can coexist peacefully within the platform. The Eclipse Platform includes support for CVS (Concurrent Versions System) repositories accessed via either pserver or SSH (Secure Shell) protocols.

Help mechanism. The Eclipse Platform help mechanism allows tools to define and contribute documentation to one or more online guides. For example, a tool usually contributes help documentation to a user guide and API documentation (if it has any) to a separate programmer's guide. Raw content is contributed as HTML files. The facilities for arranging the raw content into online guides with suitable navigation structures are expressed separately in XML files. This separation allows pre-existing HTML documentation to be incorporated directly into online books without the need to edit or rewrite it.

The navigation structure presents the contents of the guides as a tree of topics. Each topic can have a link to a raw content page. A single book may have multiple alternate lists of top-level topics allowing some or all of the same information to be presented in completely different organizations; for example, it may be organized by task or by tool.

The XML navigation files and HTML content files are stored in a plug-in's root directory or subdirectories. Small tools usually put their help documentation in the same plug-in as the code. Large tools often have separate help plug-ins. The platform uses its own internal documentation server to provide the actual Web pages from within the document web. This custom server allows the platform to resolve special inter-plug-in links and extract HTML pages from ZIP archives.

When a help system is organized, the creation of a full topic tree is only possible when the set of tools to be documented is closed. With the Eclipse Platform, the set of tools is open-ended, and consequently the structure of the help documentation needs to be modular. The platform help mechanism allows tools to contribute both raw content and sets of topics, and to define insertion points to indicate where to insert its topics into a pre-existing topic tree.

Just the foundation. As described above, the Eclipse Platform provides a nucleus of generic building blocks and APIs like the workspace and the workbench, and various extension points through which new functionality can be integrated. Through these extension points, tools written as separate plug-ins can extend the Eclipse Platform. The user is presented with an IDE that is customized by the set of available tool plug-ins. Tools may also define new extension points and APIs of their own and thereby serve as building blocks and integration points for yet other tools.

The tools plugged in to the platform supply the specific capabilities that make it suitable for developing certain kinds of applications. The Eclipse project itself provides a number of optional components that sit atop the Eclipse Platform. The most extensive of these are the Java development tools (JDT), which add the capabilities of a full-featured Java IDE to the Eclipse Platform. Figure 3 shows what the workbench normally looks like when the user is writing a Java program. Comparing Figure 3 to Figure 1 gives a sense of how JDT integrates its capabilities into the workbench.

The JDT is implemented by a group of plug-ins, with the UI in one plug-in and the non-UI infrastructure in a separate core plug-in. This separation of UI and non-UI code allows the JDT core infrastructure to be used in GUI-less configurations of the Eclipse Platform, and by other GUI tools that incorporate Java capabilities but do not need the JDT UI. The Java UI plug-in makes extensive use of workbench extension points to contribute special editors, views, perspectives, and actions that allow the user to work with Java programs in Java-specific terms. The Java compiler and underlying Java structure model can be invoked programmatically from other tools through the Java model API defined by the JDT core plug-in. Both the JDT core and UI plug-ins also declare extension points so that other tools can extend them in predefined ways.

Eclipse in practice

The Eclipse Platform provides a solid technical foundation for integrated tools that support diverse application domains across the full development life cycle. Yet a technically sound foundation alone is no assurance of success. In this section, we look at how the Eclipse Platform has gained acceptance in practice, ensuring that useful Eclipse-based tools are built.

Eclipse open source project. One key factor in the Eclipse Platform's success is that it is run as an open source project. The www.eclipse.org Web site, the public base of operations, has the following features:

- *Free download area*—Offers current releases of the Eclipse Platform in a ready-to-use (binary) form.
- *Newsgroups*—Provide general discussion about Eclipse. These lists are open to anyone and are a primary resource for help and advice for anyone using the Eclipse Platform.
- *Source code repository*—The source code for the Eclipse Platform is in a CVS repository. Anyone can browse the contents and revision history of any source file. There are also build scripts that allow anyone to recompile and re-create the Eclipse Platform from the source code.
- *Bug-tracking database*—There is a Bugzilla-based bug database for tracking defects and problems with Eclipse Platform releases. Users can use the database to discover known problems or to report a new bug they have found (or request a new feature they would like).
- *Development plans, proposals, and developer mailing lists*—Development plans and proposals are posted on the Web site, and there are developer mailing lists for communication between developers working on the Eclipse Platform. Although this information is primarily of interest to the Eclipse development team (which is distributed around the world), it is also available to anyone interested in following how the next Eclipse Platform release is taking shape.

Eclipse source materials are made available under the Common Public License (CPL), one of the licenses approved by the Open Source Initiative (OSI). This royalty-free license allows anyone to use and redistribute Eclipse for commercial or non-commercial purposes.

The Eclipse code base is developed and maintained by a technical meritocracy. A developer with a proven record of valuable contributions is rewarded with increased responsibility and the opportunity for further contributions. Interested developers can contribute to the project without needing to be an employee of any particular company; rather, their ability to contribute is based on their skills and the technical merits of their contributions. Key contributors are visible and recognized by the community. Committers are the subset of developers responsible for the code and are the only ones allowed to create new versions in the repository. Committers are typically full-time employees paid by their companies to work on Eclipse.

The Eclipse project management committee (PMC) provides technical leadership. The PMC collects community input and requirements, develops release plans, and generally coordinates activity across the range of platform subcomponents.

The development process is iterative and engages the wider Eclipse community. The release schedule is partitioned into milestone cycles of fixed duration (currently six weeks). Each milestone cycle is like a small release cycle: it includes planning, development, testing, and a milestone delivery. The specific steps in each cycle evolve based on input from the community. Each milestone is shipped with a description of new and noteworthy features in an effort to draw the community's attention to the most recent milestone and to encourage them to use it. By providing milestones at regular intervals, the Eclipse Platform is effectively in continuous beta. This sets up a positive feedback loop that further encourages community participation and growth.

Eclipse consortium. The Eclipse open source project is backed by the Eclipse consortium, a group of companies that have made a commitment to releasing Eclipse Platform-compatible offerings and to supporting the community of users, researchers, and developers. The consortium has steadily expanded from nine founding members in November 2001 to over 40 in the following two years. IBM (a founding member of the Eclipse consortium) originally developed the Eclipse Platform and contributed it to the open source project in November 2001. The team that originally developed the Eclipse Platform became the project's initial set of committers. Besides the Eclipse Platform, the seed contribution included JDT and specialized tools needed to develop Eclipse plug-ins. Right from the start, the Eclipse open source project was able to be entirely self-sustaining; no proprietary tool or "special sauce" is required to develop and maintain Eclipse or to develop new plug-ins for Eclipse.

The Eclipse consortium supports other Eclipse-based open source development efforts at www.eclipse.org. The Eclipse tools project fosters the creation of a wide variety of tools and frameworks for the Eclipse Platform, including a graphical editor framework (GEF subproject), a modeling framework (EMF [Eclipse Modeling Framework] subproject), C/C++ development tools (CDT subproject), and automated software quality tools (Hyades subproject). The Eclipse technology project provides channels for open source developers, researchers, academics, and educators to participate in the long-term evolution of Eclipse (and beyond). Current research efforts in-

clude aspect-oriented software development (AspectJ and AJDT subprojects), alternative Eclipse runtimes (Equinox project), a collaboration framework (Koi subproject), advanced software configuration management (Stellation subproject), XML Schema tools (XSD [XML Schema Definition language] subproject), and tools for model-driven software development (GMT [Generative Model Transformer] subproject).

Eclipse community. The Eclipse community is remarkably diverse. We describe some of the constituencies to explain why Eclipse appeals to them and to see how their participation enriches the Eclipse community.

A large, ready-made audience for Eclipse consists of Java developers who need a Java IDE to help them develop Java programs. Java developers can quickly have a high quality, ready-to-use, Java IDE for the price of a free download. Naturally, this is a big attraction in the Java community, including computer science students. These Eclipse users are a valuable source of bug reports, new feature requests, newsgroup traffic (both questions and answers), and beta testers.

The transition from Java developer to Eclipse plug-in developer is relatively smooth. Eclipse is written in the Java language, and the standard Eclipse download contains the specialized tools needed for developing plug-ins. A segment of these Java developers go on to "scratch their own itch" and develop new tools in the form of Eclipse plug-ins. These tools are often made available by their owners to the general Eclipse community in one form or another. Another segment of these Java developers go on to apply their Java skills to tracking down and fixing bugs in Eclipse. Again, the built-in PDE support makes it easy to work with the source code of the plug-ins that make up Eclipse, and to produce patches to attach to the bug report for later consideration by the Eclipse committer responsible for the affected component. Individual developers who are knowledgeable in the ways of building Eclipse plug-ins are an important resource.

Many software engineering research projects can benefit from building atop a ready-made, commercial quality, full source code base such as that provided by the Eclipse Platform, allowing the researchers to focus their efforts on their area of expertise and interest. On the commercial side, for many companies with IDE products, the IDE is just the matrix in which to embed the unique special-purpose tools that the company is offering. Building the matrix is onerous because it involves writing the frameworks and tool infrastructure that provide services common to all IDEs. The Eclipse Platform provides these common services in a language-neutral way. Using the Eclipse Platform instead of investing in creating (and maintaining) their own comparable IDE infrastructure lets companies focus their efforts on providing the essential tools that their customers need to work with the companies' main products. For example, IBM WebSphere Studio Application Developer is an Eclipse-based IDE with special tools for creating and debugging J2EE applications; Hewlett-Packard provides an Internet usage manager component development environment based on Eclipse; and QNX Momentics** [23] is an Eclipse-based development suite for the Neutrino** real-time operating system. Comprehensive IDE products built atop the Eclipse Platform make salient issues of UI complexity and scalability; in time, this pressure allows other developers to follow with increased confidence that the edifice will hold up under the load of a large number of plug-ins.

The companies that produce Eclipse-based IDE products also open doors for ISVs to sell their unique tools as Eclipse plug-ins augmenting the IDE product that the customer has already purchased. In some cases, this allows an add-on tool writer to sell the same plug-in for use in several different IDE products. For example, Instantiations CodePro Studio [24] provides additional tools for Java development that augment IBM WebSphere Studio or any Eclipse-based IDE that also includes JDT. By addressing customer needs not satisfied by the Eclipse-based IDE product, ISVs enhance the value of the product in addition to finding a market for their own tools. Thus, each of the different constituencies has its own reasons for using Eclipse, and each gives back to the Eclipse community either directly or indirectly. Like Eclipse itself constituencies are open-ended: new constituencies arise naturally and push Eclipse in hitherto unexplored directions. For instance, a number of parties would like to use the Eclipse Platform to build applications other than IDEs. Indeed, a number of them have modified the Eclipse Platform to meet their needs in this area. While this was not an original design goal for the Eclipse Platform, it is equally true that much of the platform is not particularly IDE-specific. One of the challenges for the next (3.0) release of the Eclipse Platform is to find a way to directly satisfy this new constituency while continuing to meet the expectations and needs of the existing Eclipse community.

Conclusion

The rosier future that customers are hoping for is one where they are more effective at developing their software because they have a wide spectrum of well-integrated tools from diverse tool vendors that support all aspects of the software development life cycle. The Eclipse Platform is a technical solution designed to enable such a future. Given the continued active participation of a diverse and growing community of developers who want Eclipse to evolve to meet their needs, there is reason to hope that the customers will get what they are hoping for.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Borland International, Inc. Corporation, Sun Microsystems, Inc., Borland Software Corporation, Microsoft Corporation, Rational Corporation, JetBrains S. R. O. Corporation, Oracle Corporation, Linus Torvalds, The Open Group, Object Management Group, or QNX Software Systems Ltd. Corporation.

Cited references and notes

1. J. G. Kemeny and T. E. Kurtz, *BASIC Instruction Manual*, Dartmouth College, Hanover NH (June 1964).
2. A. D. Falkoff and K.E. Iverson, *APL\360*, IBM Corporation (November 1966).
3. R. M. Stallman, "Emacs, the Customizable, Extensible Display Editor," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, ACM, New York (1981), pp. 147–160, http://www.gnu.org/software/emacs/emacs-paper.html.
4. *Turbo Pascal v1.0 (IBM PC Version)*, Borland Software Corporation, Scotts Valley, CA (Nov. 1983).
5. W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *Computer* **14,** No. 4, 25–34 (April 1981).
6. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA (1983).
7. R. P. Gabriel, N. Bourbaki, M. Devin, P. Dussud, D. N. Gray, and H. B. Sexton, "Foundation for a C++ Programming Environment," *Proceedings of C++ at Work-90*, ACM, New York (September 1990), http://www.dreamsongs.com/NewFiles/Energize.pdf.
8. *Borland JBuilder 1.0*, Borland Software Corporation, Scotts Valley, CA (1997), http://www.borland.com/jbuilder/.
9. *Symantec Visual Café 1.0*, Symantec Corporation, Cupertino, CA (1997), http://www.symantec.com/.
10. *Microsoft Visual J++ 1.0*, Microsoft Corporation, Redmond, WA (1997), http://www.microsoft.com.
11. *IBM VisualAge for Java 1.0*, IBM Corporation, NY (1997), http://www.software.ibm.com/ad/vajava.
12. S. P. Reiss, "Software tools and environments," *ACM Computing Surveys* (CSUR) **28,** No. 1, 281–284 (March 1996).
13. *Rational Rose 2000*, Rational Software Corporation, Cupertino, CA (2000), http://www.rational.com/products/rose/index.jsp.
14. *Introduction to IDEA 3.0 Plug-Ins*, JetBrains Inc., Prague, Czech Republic (2003), http://www.intellij.com/docs/PlugIns.pdf.
15. *Oracle9i JDeveloper*, Oracle Corporation, Redwood Shores, CA (2001), http://otn.oracle.com/products/jdev/index.html.
16. *NetBeans IDE*, www.netbeans.org (2000), http://www.netbeans.org/products/ide/.
17. *Sun ONE Studio*, Sun Microsystems, Mountain View, CA (2003), http://wwws.sun.com/software/sundev/.
18. *Microsoft Visual Studio: Extending Visual Studio*, Microsoft Corporation, Redmond, WA (2003), http://msdn.microsoft.com/vstudio/using/building/addin/default.aspx.
19. This section contains material adapted from Eclipse Platform Technical Overview, which appears on the www.eclipse.org Web site at http://eclipse.org/whitepapers/eclipse-overview.pdf.
20. *Eclipse Platform 1.0*, www.eclipse.org (November 2001), http://eclipse.org.
21. *IBM WebSphere Studio Application Developer Version 4.0*, IBM Corporation, NY (2001), http://www.ibm.com/websphere/eclipse.
22. In Version 2.1 of Eclipse (released in March 2003), this works for Windows but not for Linux.
23. *QNX Momentics*, QNX Software Systems Ltd., Ottawa, Canada (2003), http://www.qnx.com/products/ps_momentics/.
24. *Instantiations CodePro Studio*, Instantiations Inc., Portland, OR (2003), http://www.instantiations.com/codepro/default.htm.

Accepted for publication December 5, 2003.

Jim des Rivières *IBM Software Group, 2670 Queensview Drive, Ottawa, Ontario K2B 8K1, (jim_des_rivieres@ca.ibm.com)* is one of the architects of the Eclipse Platform and JDT infrastructure, with a special focus on the overall design of the Eclipse APIs. Prior to Eclipse, Jim was involved with IBM VisualAge/Smalltalk and was an architect of IBM VisualAge Micro Edition. Prior to joining OTI (Object Technology International) in 1993, Jim was at Xerox PARC where he co-authored the book "The Art of the Metaobject Protocol." His interests include API design and evolution, programming languages, and digital photography. Jim is with IBM OTI Labs in Ottawa, Canada.

John Wiegand *IBM Software Group, 15350 S.W. Koll Parkway, Beaverton, Oregon, 97006, (john_wiegand@us.ibm.com)* is the principal architect for the Eclipse Platform infrastructure. John played a central role in the development of IBM VisualAge/Java, IBM VisualAge Micro Edition, and Eclipse. His interests are in the areas of performance, scalability, compilers, and other challenging issues. John is the Eclipse project PMC lead and also leads the Eclipse Platform and PDE subprojects. John is with IBM OTI Labs in Beaverton, Oregon.