The Heart of Eclipse

DAN RUBEL, INSTANTIATIONS

A look inside an extensible plug-in architecture

ECLIPSE is both an open, extensible development environment for building software and an open, extensible application framework upon which software can be built. Considered the most popular Java IDE, it provides a common UI model for working with tools and promotes rapid development of modular features based on a plug-in component model (figure 1). The Eclipse Foundation (<http://www.eclipse.org>) designed the platform to run natively on multiple operating systems, including Macintosh, Windows, and Linux, providing robust integration with each and providing rich clients that support the GUI interactions everyone is familiar with: drag and drop, cut and paste (clipboard), navigation, and customization. You can think of Eclipse as a “design center” supported by a development team of 300 or more developers whom you can leverage when developing your own software.

Although designed as a universal tool-integration platform, Eclipse is not only for creating IDEs. It is just as helpful for constructing general-purpose applications—for example, workflow, help systems, or contact management systems (figure 2). In fact, NASA used Eclipse RCP (Rich Client Platform) to build Maestro, a software program for managing remote vehicles on space missions.

ARCHITECTURAL OVERVIEW OF ECLIPSE RCP

At the heart of Eclipse is an architecture for dynamically discovering, loading, and running components, or *plug-*

ins, the basic unit of functionality in Eclipse. The plug-ins determine the platform functionality and whether it operates as an IDE or a general-purpose application.

The Eclipse IDE is separated into layers, each consisting of smaller-grained plug-ins. The Java IDE is layered on top of a more general-purpose tools platform, which in turn is layered on the Eclipse RCP framework (figure 2). Eclipse RCP provides plug-ins that form a very basic application infrastructure, including services such as the windowing system, the help system, preference pages, and an update manager (see table 1).

The development objectives for Eclipse and Eclipse RCP are to:

- Foster cross-platform development for Windows, Macintosh, Linux, and other operating systems
- Support native-user interface elements (“widgets”) rather than emulate them on supported operating systems
- Reduce the memory footprint and startup time of components and plug-ins
- Create reusable modules for general application development

The Eclipse teams strive to reduce the effort of designing an application by providing a sort of “Chinese menu” of readily available modules (plug-ins), thus decreasing development time and time-to-market, while still building world-class applications. Eclipse plug-ins are loosely

The Heart of Eclipse

coupled, and developers are free to snap them together in different configurations using only those necessary to solve a particular problem.

Plug-ins for RCP can be programmed to the Eclipse portable APIs and run unchanged on any supported operating system. This allows developers to focus on designing plug-ins that perform important business tasks, such as workflow, processing, diagramming, reporting, publishing, or any other business requirement. There are many plug-ins generally available for Eclipse, or users can design their own. The Eclipse platform handles the logistics of finding and running the right code, and the platform UI provides a standard model for navigation. (For a list of plug-ins available, go to <http://www.eclipse-plugincentral.com>.)

Eclipse RCP contains five fundamental parts:

- UI Workbench with editors, perspectives, and views.
- SWT (Standard Widget Toolkit), a low-level graphics library of Java GUI components with native implementations that hide differences between platforms or operating systems so that the developer can program to a single API for all platforms.

- JFace, a GUI abstraction layer for displaying objects that is layered on top of SWT.
- Platform Runtime, which defines the extension-point model facilitating loose coupling between plug-ins and just-in-time lazy loading and initialization.
- OSGi (Open Services Gateway Initiative), a framework used by Eclipse for plug-in discovery and lifecycle management, including loading and unloading plug-ins without requiring the application to be restarted.

ECLIPSE UI WORKBENCH

The Eclipse UI Workbench organizes the user experience and supplies the structures in which an application interacts with the user. From the user's standpoint, an application window consists visually of *editors*, *views*, and *actions* that can be rearranged by the user to better suit the task. *Perspectives* manifest themselves in the selection and arrangements of editors, views, and actions visible on the screen (figure 3). The UI Workbench frees developers from having to reinvent a high-performance user interface for each Java-client application running on different operating systems.

Editors allow the user to open, edit, and save domain objects, such as a record of a person's name, age, contact information, and social security number. Editors follow an open-save-close lifecycle much as file-system-based tools do, but they are more tightly integrated into the workbench. When active, an editor can contribute actions to the workbench menus and tool bar.

Views provide information about objects related to the user's current task, such as a list of objects that can be edited. A view may

assist an editor by providing information about the document being edited. For example, if the editor implements a standard interface, then the content outline view shows a structured outline for the content of the active editor. A view may augment other views by providing information about a selected object. Views have simpler lifecycles than editors. Modifications made in a view (such as changing a property value) are saved immediately, and the changes are reflected immediately in other related parts of the UI.

Actions permit user commands to be defined independently from their exact location in the UI. An action represents a command that can be triggered by the user with a button, menu item, or tool-bar item. Each action knows its own UI properties (label, icon, tool tip, etc.) used to construct appropriate widgets for presenting the action. This separation allows the same action to be used in several places in the UI and means that it is easy to change where an action is presented in the UI without

TABLE 1

Details of Eclipse

Platform Runtime	Dynamically discovers plug-ins and maintains information about the plug-ins and their services in a platform registry. Plug-ins are loaded and launched when required, according to user operation of the platform. The runtime is implemented using the OSGi framework.
Resource management (workspace)	Defines API for creating and managing resources (projects, files, and folders) that are produced by tools and kept in the file system.
Eclipse UI Workbench	Implements the user cockpit for navigating the platform. It defines extension points for adding UI components such as views or menu actions. It supplies additional toolkits (JFace and SWT) for building user interfaces. The UI services are structured so that a subset of the UI plug-ins can be used to build rich client applications that are independent of the resource management and workspace model. IDE-centric plug-ins define additional function for navigating and manipulating resources.
Help system	Defines extension points for plug-ins to provide help or other documentation as browsable books.
Team support	Defines a team programming model for managing and versioning resources.
Debug support	Defines a language-independent debug model and UI classes for building debuggers and launchers.
Other utilities	Other utility plug-ins supply functions such as searching and comparing resources, performing builds using XML configuration files, and dynamically updating the platform from a server.

The Heart of Eclipse

having to change the code for the action itself.

Perspectives organize views and editors in an arrangement suitable for a specific user task. Views and editors can be tiled, stacked, or detached for presentation on the screen. A perspective controls initial view visibility, layout, and action visibility, but once the perspective has been opened, the user has complete control and can easily rearrange and customize a perspective to better suit a particular task. At any time, the user can quickly switch to a different perspective to work on a different task. Multiple perspectives can be opened in a single window or in separate windows, all controlled by the user.

The Eclipse workbench window offers a set of visual design choices about how a UI is organized. It has a menu bar, tool bar, shortcut bar, and a status line, along with certain colors and gradients that give things an integrated

and distinctive appearance. Views have titles with actions in their title bar; the workbench has File, Edit, Window, and Help listings. Part of every perspective is set aside as an editor area. For a general-purpose application, each of these UI elements can be visible or suppressed, based upon the needs of the application.

The user's ability to configure and customize is an important part of the UI. For example, the workbench allows the user to rearrange elements in the perspective by dragging and dropping views and maximizing a view by double-clicking on its title bar. A general-purpose application might want to decide whether and how the user can affect the layout. A user might want to rearrange how a person's information appears on the screen—for example, by making it smaller to see more people's information or even rearranging the order of the information.

New functionality integrates into the UI's editors-views-perspectives paradigm in well-defined ways. All views and editors implement common interfaces and access common APIs so that users may customize their appearance, content, and location in the application window. Extension points allow plug-ins to augment the workbench by adding new types of editors, views, actions, and perspectives.

The workbench API is implemented using both SWT and JFace for a platform native user experience. AWT (Abstract Window Toolkit) and Swing are not used because AWT provides too small a set of native widgets, and Swing provides emulated and not

Eclipse Workbench Screenshot

FIG 3

native widgets; furthermore, at the time both lacked some of the performance needed. Today client applications can use AWT and Swing alongside SWT and JFace, although typically only SWT and JFace are used.

SWT

SWT provides a common, operating system-independent API for widgets and graphics. The entire Eclipse platform UI and the tools that plug into it use SWT for presenting information to the user. SWT is implemented to allow tight integration with the underlying native window system. It provides an infrastructure for modular, native widgets consistent with the platform the application targets. This results in a responsive, high-quality user experience.

A low-level graphics library of Java GUI components with native implementations, SWT hides differences between platforms or operating systems so that a developer can program to a single API for all platforms using a consistent set of high-performance widgets. This means developers can write the code once and make it look like a Macintosh or Windows application UI. As part of the UI Workbench, SWT also contributes to giving the user a responsive, high-quality experience with the Java-client application that feels native to the platform. For small, lightweight applications to a low-level API where a tiny footprint is important, such as a mobile application, developers might choose SWT over JFace.

The tension between portable toolkits and native window system integration is always an issue with widget toolkit design. Java AWT provides low-level widgets (lists, text fields, and buttons), but no high-level widgets (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 to the least common denominator of all operating system window systems.

The Java Swing toolkit addresses this issue by emulating widgets such as 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. Emulated widgets invariably lack the look-and-feel of native widgets, however. Users interacting with emulated widgets notice enough difference from shrink-wrapped software applications and their native widgets that it is hard for software with an emulated UI to compete.

To overcome this problem, SWT defines a common API available across a number of supported window systems. Wherever possible, SWT uses native widgets for each different native window system. When a native widget is

available on one platform but not another, SWT provides a suitable emulation. SWT implements common low-level widgets such as lists, text fields, and buttons everywhere using native widgets; some generally useful higher-level widgets, however, may need to be emulated on some window systems. This approach allows SWT to maintain a consistent programming model across all environments and lets the look-and-feel of the underlying native window system show through as much as possible.

Internally, the SWT implementation provides separate and distinct implementations in Java for each native window system while keeping the API identical across all platforms. The Java native libraries are completely different for each platform, with each tying the specific underlying window system to the common API with a minimum amount of code. Tight integration with the underlying native window system is not strictly a matter of look-and-feel. SWT also interacts with native desktop features such as drag-and-drop, allowing the user to drag-and-drop content from SWT widgets to other native applications not based on SWT.

In cases where a particular underlying native window system provides a unique and significant feature that cannot be emulated on other window systems, SWT exposes the native window system-specific APIs. Windows ActiveX is an example. The window system-specific API is segregated into appropriately named packages to indicate that it is inherently nonportable. Because these window system-specific APIs are implemented with a minimum amount of code, the SWT implementation is expressed entirely in Java code that looks familiar to the native operating system developer. Windows programmers using C will find the Java implementation of SWT for Windows ActiveX instantly familiar, since it consists of calls to the Windows API that they already know.

This strategy greatly simplifies implementing, debugging, and maintaining SWT because it allows all interesting development to be done in Java. On the other hand, this is not a concern for ordinary SWT clients because native widgets are hidden behind the window system-independent SWT API.

JFACE

JFace is a Java UI toolkit with classes for handling many common UI programming tasks at a higher level. SWT and JFace are separate because they solve different problems: SWT provides a common API across all platforms, whereas JFace builds upon this common yet simple API to provide an easy-to-use UI abstraction layer. Using JFace, programmers interact with their own domain objects

The Heart of Eclipse

rather than their more primitive underlying UI presentation. For example, to display a list of records, you implement an interface describing what should be displayed for a given record, and from that point on you manipulate lists of records rather than lists of strings representing those records.

JFace is window-system-independent in both its API and implementation and includes the usual UI toolkit components of image and font registries, dialog, preference, wizard frameworks, and progress reporting for long-running operations. Three of its more interesting features are *viewers*, *content providers*, and *label providers*.

Viewers provide object-oriented wrappers around their associated SWT components. They provide higher-level semantics than 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 sync with changes to that domain. These viewers are configured with a content provider and a label provider and can optionally be configured with element-based filters and sorters.

Content providers know how to map the viewer's input element to expected viewer content and how to translate domain changes into appropriate viewer updates.

Label providers produce the specific string label and icon needed to display any given domain element in the widget.

Clients are notified of selections and events in terms of the domain ele-

ments they provide to the viewer. The viewer implementation handles the mapping between domain elements and SWT widgets, adjusting for a filtered view of the elements and re-sorting when necessary. The standard viewer for text supports common operations such as double-click behavior, undo, coloring, or navigating by character index or line number. Text viewers provide a document model to the client and manage the conversion of the document to the information required by the SWT-styled text widget.

Existing domain objects can be displayed in the Eclipse UI by implementing the appropriate interfaces. 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. If you are show-

ing a list of record information, for example, and the user selects a particular element in the list, then internal notifications are generated informing other user interface elements so that they can update their content based on the new selection.

PLATFORM RUNTIME

The Platform Runtime facilitates loosely coupled program modules (plug-ins) through the use of extensions and extension points. Plug-ins declare how they extend other plug-ins and how other plug-ins can extend them. Using this information, the Platform Runtime dynamically determines the minimum set of plug-ins necessary for a given task and loads only those plug-ins; this reduces the memory footprint of the application and improves the startup speed.

The Platform Runtime uses the OSGi framework to determine which plug-ins are available when an application is running. This information, along with the plug-in extension and extension-point information, provides each plug-in with information about which plug-ins extend it and how. Because this information is not hard-coded into each plug-in, but instead dynamically discovered at time of execution, programmers can more easily recombine plug-ins in different ways to solve different problems without having to reinvent the wheel.

OSGi FRAMEWORK

The OSGi framework is a thin layer that allows multiple Java-based components to cooperate in a single JVM (Java Virtual Machine). It focuses on the plug-in lifecycle so that plug-ins can be installed, updated, or removed on the fly without disrupting the operation of the device. Its software components are libraries or applications that can dynamically discover and use other components (see figure 4).

Components are lazily loaded on an as-needed basis, reducing the startup time and overall memory footprint of the application. Many standard component interfaces are available for OSGi, including common functions such as HTTP servers, configuration, logging, security, user administration, and XML. Plug-in-compatible implementations of these components can be obtained from different vendors with different optimizations. OSGi adds the following functionalities to the Eclipse RCP platform:

- A powerful and rigidly specified Java-based class-loading layer defining the class-loading policies.
- A lifecycle layer that adds bundles that can be dynamically installed, started, stopped, updated, and uninstalled. Bundles rely on the module layer for class

loading but add an API to manage the modules in runtime.

- A service registry providing a cooperation model for bundles that facilitates service discovery without loading and executing the bundle.

PLUG-INS AND THEIR OPERATION

A plug-in, also known as a bundle, is the smallest functional unit of the Eclipse platform that can be developed and delivered separately. Plug-ins are coded in Java. Very small applications might be written as a single plug-in, whereas complex applications blend their functionality across several plug-ins. Except for the Platform Runtime, all of the Eclipse platform's functionality is located in plug-ins. They can provide support for editing and manipulating additional types of resources such as Java files, C programs, Word documents, HTML pages, and JSP (JavaServer pages) files. Plug-ins determine the ultimate functionality of the platform or application. That's why the Eclipse SDK ships with additional plug-ins to enhance its functionality.

A typical plug-in contains a manifest file, 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, etc. A plug-in, however, may not contain code at all. A plug-in that contributes online help in the form of HTML pages is one example. The plug-in code libraries and read-only content are located together in a directory in the file system or at a base URL on a server. Another mechanism permits a plug-in to be synthesized from separate fragments, each in its own directory or URL. This is the mechanism used to deliver separate language packs for an internationalized plug-in.

The manifest file of a plug-in defines how that plug-in interacts with the system. This includes information uniquely identifying the plug-in and version. It also includes what services that plug-in consumes and what services that plug-in provides to other plug-ins. This declarative approach to modularization allows the Eclipse Platform Runtime to load only the plug-ins necessary for the current operation of the application.

CHALLENGES OF BUILDING APPLICATIONS USING ECLIPSE RCP

Developers wanting to use Eclipse RCP face several technical issues, including getting up to speed and writing loosely coupled code:

Getting up to speed. Eclipse is a very well-designed and well-supported object-oriented framework. It rep-

The Heart of Eclipse

resents a very large API and therefore can have a steep learning curve for someone new to the environment. Many books, training materials, and online articles are available to provide a general understanding and direct developers where to dig for more details. How to use a particular function or feature is well documented in code via the Javadoc explaining the API.

Stick to the API. Many developers are used to writing code that accesses internal code in other modules. Taking the same approach for an Eclipse RCP application means the code will not be maintainable, because the internal code changes as the Eclipse platform evolves.

Loosely coupled code. Many developers are used to writing code interacting with other modules but not writing a manifest declaring *how* their code interacts with other modules. Writing plug-ins that declare how they interact with other plug-ins promotes flexibility and reusability of the code being written and is a cornerstone of Eclipse-based development. Developers need to think in terms of what “services” a plug-in provides and how it extends or consumes “services” of other plug-ins.

Testing tools. The availability of testing tools can also be an issue. For example, few automated GUI testing tools are available for Eclipse RCP applications; JUnit is a standard Java test framework but does not easily address UI testing without additional functionality layered on top of it. Fortunately, some third-party solutions are emerging.

Installers. Eclipse RCP doesn’t have an installer, making deployment and application updates more difficult. Commercial third-party installers, however, can handle this well.

Plug-in quality, maturity, and support. For some, Eclipse is rich enough as an application framework that it doesn’t require other plug-ins to add capabilities. Many organizations, however, need to enhance Eclipse RCP with features and functions that match their business requirements. This means they will have to dip into the Eclipse plug-in ecosystem, which brings up several issues. A parallel can be found in the commercial software world where freeware and shareware are offered. The quality, maturity, and support of these products are obvious considerations.

Solutions are emerging that address identification, quality, and support issues. SourceLabs validates, integrates, and supports open source products. Eclipse Plug-in Central (<http://www.eclipseplugincentral.com>) is a clearinghouse for Eclipse plug-ins. Yoxos is another example of Eclipse distribution that includes open source plug-ins available in managed scenarios and supported.

COMMERCIAL THIRD-PARTY TOOLS

For developers choosing to create a rich client for their applications, there are commercial tools, such as RCP Developer (<http://www.instantiations.com/rcpdeveloper>), available. These tools help with not only the GUI design and coding, but also UI testing, creating help documentation, and packaging and deployment of the application. The update manager in Eclipse can keep users up to date with the latest versions of their applications.

CONCLUSION

Many organizations are standardizing on RCP as the architecture for their internal IT applications, thus enabling them to run as a manageable, integrated suite rather than as disassociated units of functionality. Eclipse RCP will have serious implications for the desktop computing strategies of enterprise organizations because it incorporates application development tools that are easy to use, provides an open source extensible application framework, and creates applications that can run on a variety of platforms. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

DAN RUBEL is the CTO at Instantiations. An expert in the design and application of object-oriented technologies, he has more than 15 years of software development experience, including eight years of experience with Java and four years with Eclipse. He is the primary architect and product manager for several commercial products. He has a B.S. from Bucknell and is co-author of the book *Eclipse: Building Commercial Quality Plug-ins* (Addison-Wesley Eclipse series).

© 2006 ACM 1542-7730/06/1000 \$5.00