Eclipse Platform Technical Overview

Object Technology International, Inc.
July 2001

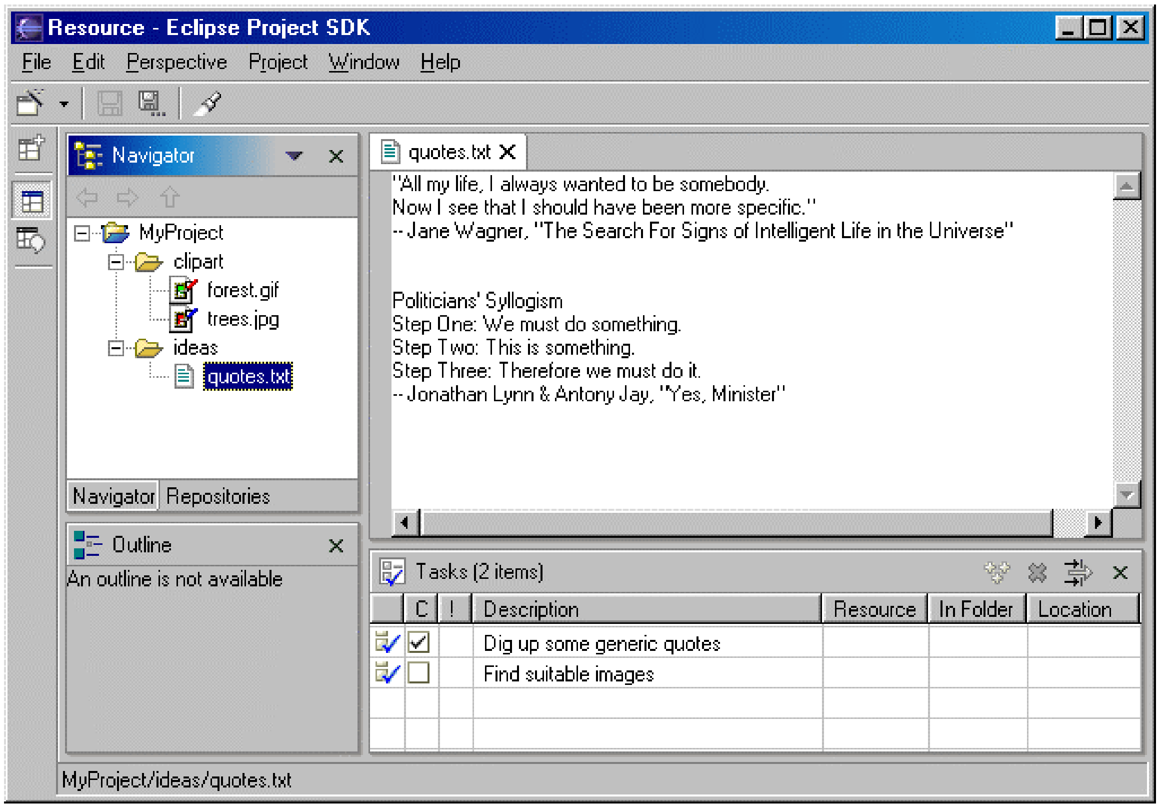
Abstract: The Eclipse Platform is designed for building integrated development environments (IDEs) that can be used to create applications as diverse as web sites, embedded Java™ programs, C++ programs, and Enterprise JavaBeans™. This paper is a general technical introduction to the Eclipse Platform. Part I presents a technical overview of its architecture. Part II is a case study of how the Eclipse Platform was used to build a full-featured Java development environment.

Contents

Introduction

The Eclipse Platform is 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.



"All my life, I always wanted to be somebody.
Now I see that I should have been more specific."
-- Jane Wagner, "The Search For Signs of Intelligent Life in the Universe"

Politicians' Syllogism
Step One: We must do something.
Step Two: This is something.
Step Three: Therefore we must do it.
-- Jonathan Lynn & Antony Jay, "Yes, Minister"

An outline is not available

	C	!	Description	Resource	In Folder	Location
✓	☑		Dig up some generic quotes			
✓	☐		Find suitable images			

Figure 1. Eclipse Platform UI.

The navigator view (Figure 1, top left) shows the files in the user's workspace; the text editor (top right) shows the content of a file; the tasks view (bottom right) shows a list of to-dos; the outline view (bottom left) shows a content outline of the file being edited (not available for plain text files).

Although the Eclipse Platform has a lot of 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 files in the workspace and surfaces its tool-specific UI in the workbench. When the Platform is launched, the user is presented with an integrated development environment (IDE) composed of 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.

Part I: Eclipse Platform Technical Overview

If you build it, they will come.
—W.P. Kinsella, *Field of Dreams (1989)*

The Eclipse Platform (or simply "the Platform" when there is no risk of confusion) is 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, EJB, XML, and GIF).
- 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 via 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.

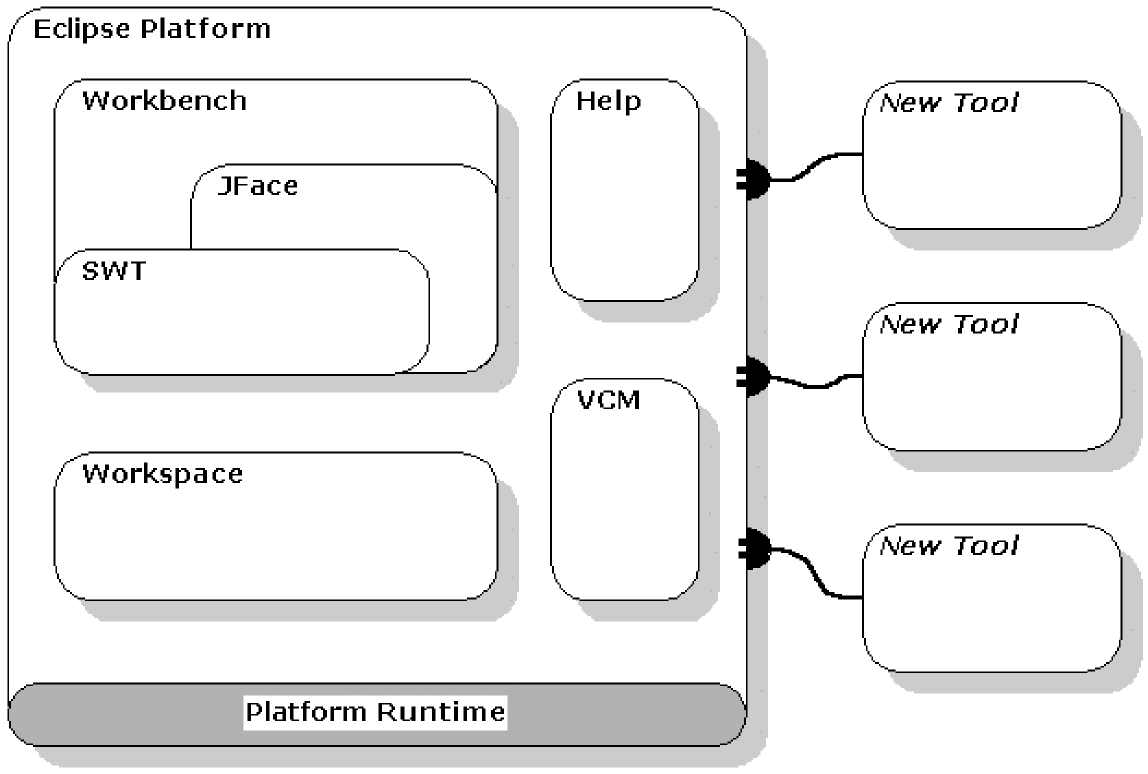


Figure 2. Eclipse Platform architecture.

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

Plug-ins are coded in Java. A typical plug-in consists of Java code in a JAR library, some read-only files, and other resources such as images, web templates, message catalogs, native code libraries, etc. Some plug-ins do not contain code at all. One such example is a plug-in that contributes online help in the form of HTML pages. A single plug-in's code libraries and read-only content are located together in a directory in the file system, or at a base URL on a server. There is also a mechanism that permits a plug-in to be synthesized from several separate fragments, each in their own directory or URL. 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 one or more extension points in other plug-ins.

A plug-in's extension points can be extended by other plug-ins. 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.

An extension point may have a corresponding API interface. Other plug-ins contribute implementations of this interface via extensions to this extension point. Any plug-in is free to define new extension points and to provide new API for other plug-ins to use.

On start up, the Platform Runtime discovers the set of available plug-ins, reads their manifest files, and builds an in-memory 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 via 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 of any of its code. This property is key 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 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, 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 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 plug-in-specific data; this mechanism allows a plug-in to carry over important state 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 via 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 up front, 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 when the context changes. A lengthy start-up sequence is avoided, as is a common source of bugs stemming from unpredictable plug-in activation order.

The Eclipse Platform is run by a single invocation of a standard Java virtual machine. Each plug-in is assigned its own Java class loader that is solely responsible for loading its classes (and Java resource bundles). Each plug-in explicitly declares its dependence on 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 itself 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 components or upgraded versions of existing components. 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 plugged 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 *project nature* mechanism allows a tool to tag a project in order to give it a particular personality, or nature. For example, the web site nature tags a project that contains the static content for a web site, and the Java nature tags 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 to the standard programs and tools of the underlying operating system. Tools integrated with the Platform are provided with API for dealing with workspace *resources* (the collective term for 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 content of any files that have been changed or deleted by integrated tools. The user controls how the history is managed via 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 guaranteed to receive after-the-fact notifications of all resource creations, deletions, and changes to the content 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 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, since each delta points out where the tool may need to add, remove, or refresh on-screen widgets. In addition, since 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 build 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 on 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 state 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 - a widget set and graphics library integrated with the native window system but with an OS-independent API.
- JFace - a UI toolkit implemented using SWT that simplifies common UI programming tasks.

SWT

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. Java AWT 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 to 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 shrink-wrapped applications developed specifically for a particular native window system.

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 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.

SWT also exposes native window system-specific API in cases where a particular underlying native window system provides a unique and significant feature unavailable on other window systems. Windows ActiveX® is a good example of this. Window system-specific API is segregated into aptly named packages to indicate the fact that it is inherently non-portable.

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, and can use components developed with OS component models, like Windows ActiveX controls.

Internally, the SWT implementation provides separate and distinct implementations in Java for each native window system. The Java native libraries are completely different, with each surfacing the APIs specific to the underlying window system. (Contrast this to 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 natives, 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 SWT for Windows instantly familiar, since 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 SWT because it allows all interesting development to be done in Java. Of course, this is of no direct concern for ordinary clients of SWT since these natives 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 API 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 independently from their exact whereabouts in the UI. An action represents a command that can be triggered by the user via 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, and means that 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 higher-level semantics than 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 synch with changes to that domain. These viewers are configured with a content provider and a label provider. The content provider knows how to map the viewer's input element to the expected viewer content, and how to parlay domain changes into corresponding viewer updates. The label provider knows how to produce the specific string label and icon needed to display any given domain element in the widget. Viewers can optionally be configured with element-based filters and sorters. Clients are notified of selections and events in terms of the domain elements 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, and 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. 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.

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 in which tools interact with the user. 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 (see Figure 1). The workbench API is dependent on the SWT API, and to a lesser extent on the JFace API. The workbench implementation is built using both SWT and JFace; Java AWT and Swing are not used.

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. 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 lifecycle much like file system based 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 some 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 shows 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 lifecycle 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 (see Figure 1); 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 visibility, layout, and action visibility. The user can quickly switch perspective 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, online help, and team support tasks. Additional perspectives are supplied by other plug-ins.

Tools integrate into this editors-views-perspectives UI paradigm in well-defined ways. The main extension points allow tools to augment the workbench:

- Add new types of editors.
- Add new types of views.
- Add new perspectives, which arrange old and new views to suit new user tasks.

The Platform's standard views and editors are all contributed using these mechanisms.

Tools may also augment existing editors, views, and perspectives:

- Add new actions to an existing view's local menu and tool bar.
- Add new actions to the workbench menu and tool bar when an existing editor becomes active.
- Add new actions to the pop-up content menu of an existing view or editor.
- Add 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 functionality that the plug-in provides.

Once 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 as a source for selections. 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 selected in the currently active editor or view.

UI Integration

Tools written in Java using the 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 via 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 document as an editor (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 (Windows only). 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.[1] This option provides loose UI integration, but allows tight integration below the UI.[2]

Team Support

The team support component of the Eclipse Platform adds version and configuration management (VCM) capabilities to projects in the workspace, and augments the workbench with all the necessary views for presenting VCM concerns to the user. Most tools simply operate on resources in the workspace. Such tools can treat VCM as an orthogonal capability that is taken care of entirely by the team support component provided they understand that components outside their control may periodically replace resources. Tools that require seamless VCM integration at the UI may add VCM actions to their domain specific presentations.

The team support model has these advanced features:

- Multiple heterogeneous repositories.
- Lightweight model of team collaboration.
- Resource versions.

The team support model centers on *repositories* that store version-managed resources on shared servers. The Eclipse Platform is designed to support a range of existing repository types.[3] A single workspace can access different types of repositories simultaneously.

A *stream* maintains a team's shared configuration of one or more related projects and their folders and files. A team of developers share a stream that reflects their ongoing work and all their changes

[1]In the 0.9 release (June 2001) of the Eclipse SDK, this works for Windows but not Linux.

[2]Of course, AWT and Swing would need to be present in the configuration of the underlying Java runtime environment that runs the Eclipse Platform.

[3]The Eclipse Platform does not provide its own repository technology. The 0.9 release (June 2001) of the Eclipse Platform includes support for CVS repositories accessed via either pserver or ssh protocols. Back end interfaces to other commercial VCM products are under consideration.

integrated to date. In effect, a stream is a shared workspace that resides in a repository.

Each team member works in a separate workspace and makes changes to private copies of the resources. Other team members do not immediately see these changes. At convenient times, a developer can synchronize their workspace with the stream. As a team member produces new work, they share this work with the rest of the team by *releasing* those changes to the stream. Similarly, when a team member wishes to get the latest work available, they *catch up* to the changes released to the stream by others. Both synchronization operations can be done selectively on resource subtrees, with an opportunity to preview incoming and outgoing changes. When the repository type supports an optimistic concurrency model, as CVS does, conflicting incoming and outgoing changes are detected automatically. The developer resolves the conflict, usually by merging the incoming changes into their local workspace, and releases the result.

This model supports groups of highly collaborative developers who work on a common base of files and who frequently share their changes with each other. It also supports developers who work offline for long periods, connecting to their repository only occasionally to synchronize with the stream.

A repository typically may have any number of streams, which are distinguished by name. In a CVS repository, for instance, a stream maps to the main trunk ("HEAD") or to a branch. Streams act as lightweight containers for building product releases, and as safe places to store a developer's personal work and early prototype versions outside the team's main development stream. When the same project appears in different streams, the resources evolve independently; changes released to one stream have no affect on other streams.

Repositories also support the notion of immutable resource *versions*. A version of a file resource captures the content of the file. A version of a project captures the configuration of folders and files and their specific versions (i.e., a baseline). In a CVS repository, for example, a project version corresponds to a distinctive tag assigned to a set of file versions. New versions of changed files are created automatically when those changes are released to a stream. New project versions may be created as a snapshot of either the current configuration of a project in a stream, or the current configuration of a project in the workspace.

Heterogeneous repositories and repository types are supported; each project in the workspace can be managed in a different type of repository. The workspace records stream-specific synchronization information for a project's resources. This information allows detection of incoming and outgoing file creations, deletions, and content changes.

Team support integrates with the workbench in straightforward ways. It contributes a team support perspective that includes VCM-specific views for repositories, resource histories, and resource synchronization state. It also adds version names and other VCM-specific decorations to the labels used for resources in key workbench views.

Help

The Eclipse Platform Help mechanism allows tools to define and contribute documentation to one or more online books. For example, a tool usually contributes help style documentation to a user guide, and API documentation (if it has any) to a separate programmer guide.

Raw content is contributed as HTML files. The facilities for arranging the raw content into online books 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 needing to edit or rewrite them.

The add-on navigation structure presents the content of the books as a tree of topics. Each topic, including non-leaf topics, 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, 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 organizing a help system, 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 indicate where to insert its topics into a pre-existing topic tree at predefined insertion points.

Epilogue

In summary, 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 specialized by the set of available tool plug-ins. However, rather than being the end of the story, it is really just the beginning. 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.

This brief overview has omitted a number of other interesting aspects of the Eclipse Platform such as scripting support and integration with the ANT build tool. Further details about the Eclipse Platform API, extension points, and standard components can be found in the *Platform Plug-in Developer Guide*, which is available as online help for the Eclipse SDK.

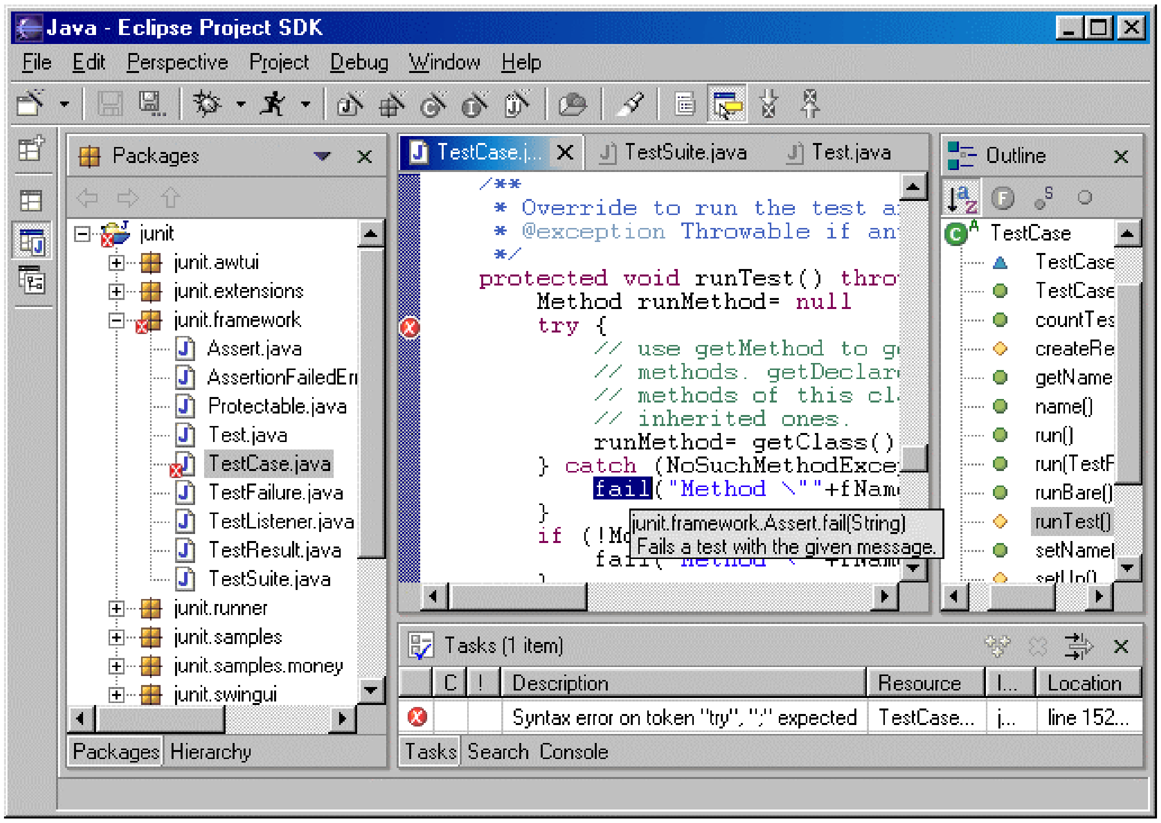
Part II: Case Study of Using the Eclipse Platform - Java Development Tooling

The proof of the pudding is in the eating.
—*folk saying*

As mentioned in Part I, the Eclipse Platform by itself is an IDE for anything, and for nothing in particular. The tools plugged in to the Platform supply the specific capabilities that make it suitable for developing certain kinds of applications. This part is a case study of a real tool, the Java development tooling (JDT), which adds Java program development capability to the Platform. The JDT is included in the Eclipse SDK.

JDT Features

Before going behind the scenes to see how the JDT is put together, it helps to have a sense of what the JDT does and what it looks like to the user. Figure 3 shows what the workbench normally looks like when the user is writing a Java program.



```
/**
 * Override to run the test a
 * @exception Throwable if an
 */
protected void runTest() thro
    Method runMethod= null
    try {
        // use getMethod to g
        // methods. getDeclar
        // methods of this cl
        // inherited ones.
        runMethod= getClass()
    } catch (NoSuchMethodExce
        fail("Method \""+fNam
    }
    if (!Mc
        fai
```

junit.framework.Assert.fail(String)
Fails a test with the given message.

C	!	Description	Resource	I...	Location
		Syntax error on token "try", ";" expected	TestCase...	j...	line 152...

Figure 3. Workbench showing Java perspective.

The JDT adds the capabilities of a full-featured Java IDE to the Eclipse Platform (some of which are visible in Figure 3). The following is a brief summary of those features:

- Java projects
  - Java source (*.java) files arranged in traditional Java package directories below one or more source folders.
  - JAR libraries in the same project, another project, or external to the workspace.
  - Generated binary class (*.class) files arranged in package directories in a separate output folder.
  - Unrestricted other files, such as program resources and design documentation.
- Browsing Java projects
  - In terms of Java-specific elements: packages, types, methods, and fields.
  - Arranged by package, or by supertype or subtype hierarchy.
- Editing
  - Java source code editor.
  - Keyword and syntax coloring (including inside Javadoc comments).
  - Separate outline shows declaration structure (automatic live updates while editing).
  - Compiler problems shown as annotations in the margin.
  - Declaration line ranges shown as annotations in the margin.
  - Code formatter.
  - Code resolve opens selected Java element in an editor.
  - Code completion proposes legal completions of method, etc. names.
  - API help shows Javadoc specification for selected Java element in pop-up window.
  - Import assistance automatically creates and organizes import declarations.
- Refactoring
  - For improving code structure without changing behavior.
  - Method extraction.
  - Safe rename for methods, etc. also updates references.
  - Preview (and veto) individual changes stemming from a refactoring operation.
- Search
  - Find declarations of and/or references to packages, types, methods, and fields.
  - Search results presented in search results view.
  - Search results reported against Java elements.
  - Matches are highlighted as annotations in the editor.
- Compare
  - Structured compare of Java compilation units showing the changes to individual Java methods, etc.
  - Replace individual Java elements with version of element in the local history.
- Compile
  - JCK-compliant Java compiler.
  - Compiler generates standard binary *.class files.
  - Incremental compilation.
  - Compiles triggered manually upon demand or automatically after each change to a

    source file (i.e., workspace auto-build).
  - Compiler problems presented in standard tasks view.
- Run
  - Run Java program in separate target Java virtual machine.
  - Supports multiple types of Java virtual machine (user selectable).
  - Console provides stdout, stdin, stderr.
  - Scrapbook pages for interactive Java code snippet evaluation.
- Debug
  - Debug Java program with JPDA-compliant Java virtual machine.
  - View threads and stack frames.
  - Set breakpoints and step through method source code.
  - Inspect and modify fields and local variables.
  - Expression evaluation in the context of a stack frame.
  - Dynamic class reloading where supported by Java virtual machine.

JDT Implementation

The JDT is implemented by a group of plug-ins, with the user interface in a UI 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.

Figure 4 illustrates key connections between the JDT and the Platform.

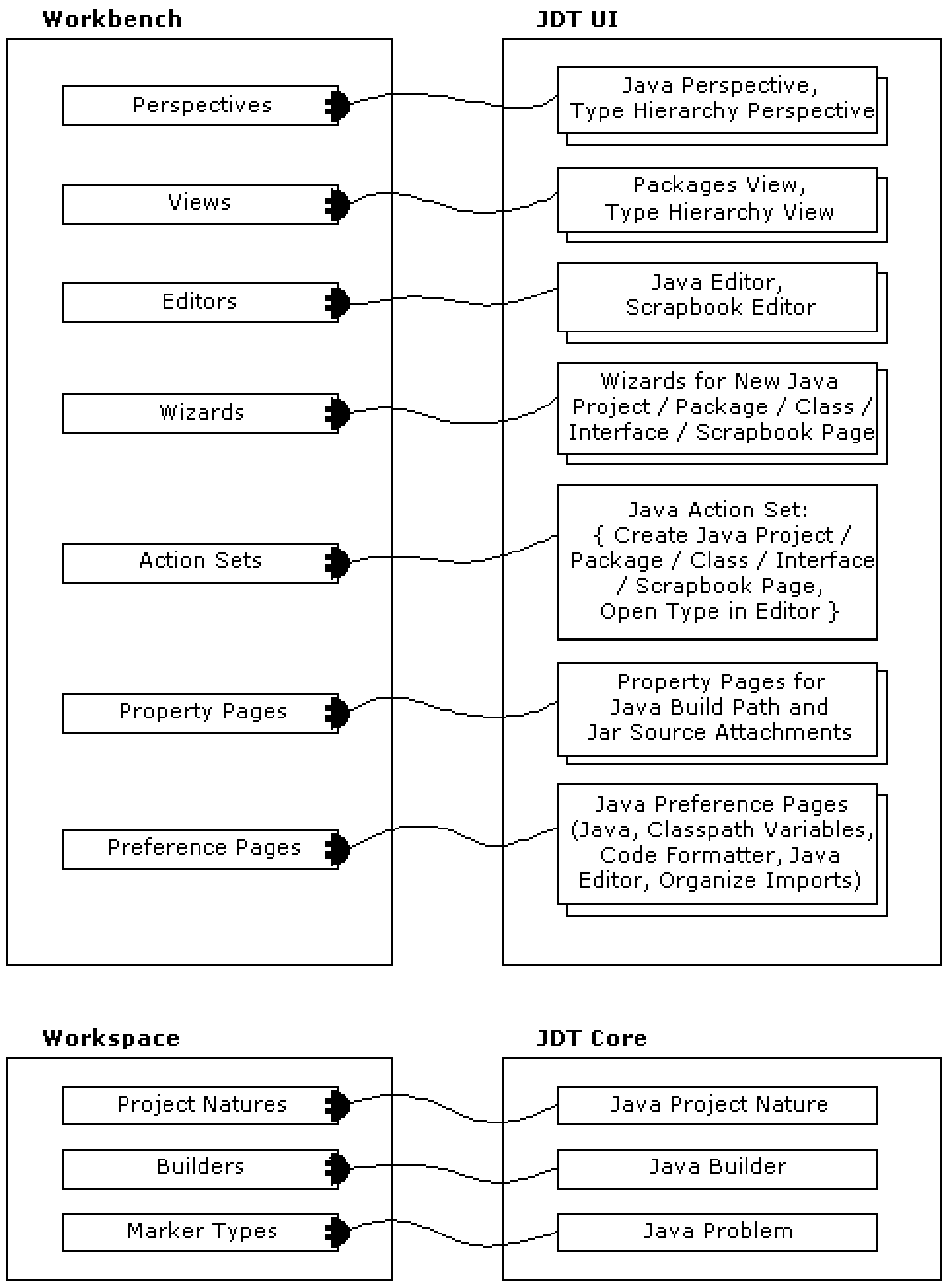

Workbench
JDT UI
Perspectives
Java Perspective, Type Hierarchy Perspective
Views
Packages View, Type Hierarchy View
Editors
Java Editor, Scrapbook Editor
Wizards
Wizards for New Java Project / Package / Class / Interface / Scrapbook Page
Action Sets
Java Action Set: { Create Java Project / Package / Class / Interface / Scrapbook Page, Open Type in Editor }
Property Pages
Property Pages for Java Build Path and Jar Source Attachments
Preference Pages
Java Preference Pages (Java, Classpath Variables, Code Formatter, Java Editor, Organize Imports)
Workspace
JDT Core
Project Natures
Java Project Nature
Builders
Java Builder
Marker Types
Java Problem

Figure 4. Key connections between JDT and Eclipse Platform.

Java Projects

At the workspace level, the JDT defines a special *Java project nature* that is used to tag a project as a Java project.

Each Java project has a single special *classpath* file (named ".classpath") that records Java-specific information about the project. This information includes the locations of the project's source folder(s), pre-compiled JAR libraries, and the output folder for compiler-generated binary class files.

Java Compiler

The Java nature configures each Java project with a *Java incremental project builder* that invokes a built-in *Java compiler*.[4]

In the case of an initial or full build, the Java compiler translates all Java source files found in the project's source folder(s) into corresponding binary class files in the project's output folder. The JDT declares a new marker subtype for *Java problems*. As the compiler detects errors, it annotates the offending source files with Java problem markers. A project specifies which JAR libraries it depends on. This allows the JDT to target various Java runtime configurations, such as CLDC, J2SE, and J2EE.

As the compiler encounters each source file, it adds information to an in-memory dependency graph. This allows subsequent builds of the project to be handled more efficiently. The workspace incremental project builder framework maintains a resource delta tree with changes since the last time a given builder was invoked. The next time the Java incremental project builder is called, it uses this resource delta tree to determine the limited set of source files that need to be recompiled because they were changed, removed, or added. The compiler uses its dependency graph to further widen the recompilation set to include any other source files that might compile differently as an consequence. The compiler then deletes obsolete class files and Java problem markers, and compiles only the computed subset of source files. The JDT participates in *workspace saves* so that the dependency graph can be preserved on disk between sessions, otherwise the next session would require a full build just to rediscover the dependency graph. This incremental strategy allows the JDT to run builds very frequently, such as after every source file save operation, even for projects containing hundreds or thousands of source files.

Java Model

The *Java model* provides API for navigating the *Java element tree*. The Java element tree represents projects in terms of Java-centric element types:

- Package fragment roots corresponding to a project's source folders and JAR libraries.
- Package fragments corresponding to specific packages within a package fragment root.
- Compilation units and binary classes corresponding to individual Java source (*.java) and binary class (*.class) files.
- Various types of Java declarations that appear within a compilation unit or class file:
    - Package declarations.

[4]In the 0.9 release (June 2001) of the Eclipse SDK, the Java compiler is JCK 1.3 compliant.

- Import declarations.
- Class and interface declarations.
- Method and constructor declarations.
- Field declarations.
- Initializer declarations.

For most Java-specific tools (including the Java UI) navigating and operating on a Java project via the Java model is more convenient than navigating the underlying resources. Java elements are represented by adaptable objects so that other parties can extend their behavior.

A Java project's classpath file and underlying resources define the Java element tree. It is infeasible to keep the Java element tree in memory as it is an order of magnitude larger than the workspace resource tree and would require reading and parsing all Java source files to construct. Instead, the Java element tree is built piecemeal and only on demand. The Java compiler parses individual compilation units to extract declaration structure. The Java element tree maintains an internal, limited size cache of recently analyzed compilation units. The Java element tree registers a resource change listener with the workspace so that it can remove obsolete cache entries when source files get deleted or changed. The Java element tree issues its own deltas, which are analogous to workspace resource tree deltas.

A cache-based Java element tree works well for simple navigation but cannot support broad searches or other patterns of traversal that visit declarations across a large number of different compilation units. The Java model addresses this by maintaining internal indexes on disk. The indexes are composed of summary entries that associate a declared or referenced name with the path of the corresponding file. Given a name, these indexes can be efficiently searched to identify files that contain at least one occurrence. The individual files can be read and parsed if further precision is required or if line numbers are needed. The Java model uses a resource change listener to keep track of files currently in need of indexing. The actual work of indexing individual files happens in a low priority background thread.

Java UI

The JDT UI defines a *Java perspective* for users developing Java code. This perspective contains the following Java-specific workbench contributions (amongst others):

- Packages view.
- Type hierarchy view.
- Actions for creating Java elements.
- Java editor and Java outline.

The packages view, which shows compilation units within a Java project, or class files within a JAR library, cuts along structural lines (like the standard workbench resource navigator view). The elements and relationships shown in the tree viewer come directly from the Java model.

In contrast, the type hierarchy view cuts across structural lines to show classes and interfaces arranged by the supertype-subtype relationship defined by the Java language. The type hierarchy is built using a sequence of index-based Java model searches combined with parsing the relevant compilation units to extract direct supertype names. The elements presented in the viewer come directly from the Java model.

The actions bring up wizards for creating a new Java project, package, class, or interface. These actions operate on the Java model.

The Java editor is registered as the editor of choice for files of type *.java. This editor collaborates with the standard workbench content outline view by providing a tree of Java elements for the current declaration structure. The Java editor makes extensive use of the JFace text viewer toolkit to implement the following features:

- Partitioning - The document is partitioned into regions of Java code and Javadoc comments using a rules-based scanner.
- Keyword and syntax coloring - Coloring rules are applied to visually distinguish tokens within each region type. The coloring is maintained with a presentation reconciler.
- Marginal annotations - The margin of the text viewer shows declaration line ranges, problem markers, and debugger breakpoints. These annotations are adjusted automatically as the text is edited.
- Formatting - Controls automatic indenting and redistributes whitespace within and between lines.
- Code assist - Proposes region-specific Java (or Javadoc) completions at a given document position. This relies on special support from the Java compiler.
- Content outline - Updates as editing takes place. This is done periodically as a background activity using a reconciler. Selections and manipulations in the content outline are immediately reflected in the editor buffer.
- Method level edit - The editor can also present a single method (or other kind of declaration) rather than the entire source file.

Refactoring operations such as "safe rename" rely on index-based Java model searches and special compiler support to locate and rewrite parts of the program affected by the change.

Java Run and Debug

The JDT UI also adds workbench menu and tool bar actions for running and debugging Java programs, and provides a *debug perspective* better suited to that task. This perspective includes a processes view that shows all currently running and recently terminated processes, and a console view that allows developers to interact with the selected running process via its standard input and output streams.

A target Java virtual machine is launched as a separate process to run the Java program. The JDT supports launchers for different types of Java virtual machines. Other tools can contribute specialized launchers via a JDT-defined extension point.

Scrapbook pages are represented as text files (type *.jpage) that get edited by a special text editor that knows how to run Java code snippets. This involves turning the selected statement, expression, or declaration into a main class, compiling it, downloading it to a running Java VM, running it, and extracting the print string of the result and displaying it in the snippet editor. This feature relies on special support from the Java compiler, but does not require special support from the target Java VM. The same evaluation technology is used to evaluate Java expressions in the debugger in the context of a stack frame.

When a Java program is launched in debug mode, a debug view shows the processes, threads, and stack frames. When the debugger needs to show source code to the user, it opens an editor using the workbench provided mechanisms. During single stepping, the debugger instructs the editor which source code line to highlight. Other debug-specific views show the list of breakpoints, the values of variables, and the fields of objects. Breakpoints are represented by a special type of marker.

The Java debugger works with any JDPA-compliant target Java VM. Where the Java VM supports dynamic class reloading, fixes made to the running program are installed immediately so that the debug session can continue with the fixes in place. The debugger registers a resource change listener so that it can discover which binary class files in the project's output folder need to be reloaded into the target VM.

Epilogue

This has been only the briefest glimpse of how the JDT supplies the specific capabilities that make the Eclipse Platform suitable for developing Java programs. 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 Java 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 pre-defined ways.

JDT is included in the Eclipse SDK; further details can be found in the *Java Development User Guide* and *JDT Plug-in Developer Guide*, which is available as online help for the Eclipse SDK.

2001-07-24b