# Integrating Web technologies in Eclipse

J. Ponzo
O. Gruber

In this paper we describe an approach and an implementation for integrating Web technologies in Eclipse, a Java- and component-based platform. First, we support embedding of UI (user interface) artifacts that are developed using either widget or markup technologies. Second, we provide support for DOM (Document Object Model) programming. We implement this approach by bridging the relevant foundation technologies—COM (Component Object Model) and XPCOM (Cross Platform Component Object Model)—which allows us to embed the engines of the two major browsers, Internet Explorer and Mozilla®. We discuss several possible applications of this work, such as seamless access to online help systems and Web-based development of administrative tools.

The world of end-user applications has been divided between Web applications and traditional applications. Whereas traditional applications use widget technologies for their user interface (UI), Web applications use markup rendering technologies, such as HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and scripting (e.g., JavaScript**). The advantages of combining both technologies in a single application has recently been demonstrated in products such as Lotus Notes*, Quicken**, and Microsoft Money 2005. It is our goal to offer these advantages in the Eclipse platform.[1]

Eclipse, an open-source project that started as a platform for developing IDEs (Integrated Development Environments), is a platform for developing applications based on software components. Through components, Eclipse provides integration frameworks such as the Eclipse Workbench for UI integration and the Eclipse Workspace for data integration. Recently, Eclipse has evolved toward the concept of a Rich Client Platform, the integration not only of tools but also of applications.[2] With this goal in mind, we feel that it is particularly important to appeal equally to Java** and Web developers.

To that end, we integrate Web technologies in Eclipse. Here we use the term "integration" to imply a tighter relationship than simple "embedding." The integration is performed in two steps: (1) embedding of UI artifacts, and (2) support for Document Object Model (DOM) programming. The UI artifacts that can be embedded in the Eclipse Workbench are developed by using either widget or markup

technologies. The DOM is the in-memory parsed tree of a markup document—most often HTML. Web browser engines dynamically render the DOM as they parse Web pages. DOM programming is the dynamic creation or manipulation of the in-memory tree, thereby controlling the rendering of the document. DOM programming is also used for handling user-interface events.

For our integration approach in Eclipse, we preserve the design principle of the Standard Widget Toolkit (SWT), the cross-platform widget toolkit of Eclipse.[1] SWT implements a thin veneer over existing native widget capabilities. This thin veneer varies with the environment because it depends on the windowing system and on the native widget toolkit. This approach is well suited for embedding Web browsers. Both Internet Explorer (IE) and Mozilla** provide embeddable engines that can be exposed to Java. We extend SWT to make use of these engines for the parsing and rendering of HTML documents as well as for providing full DOM application programming interfaces (APIs). We provide browser-specific DOM APIs, thus giving access to the full capabilities of each browser when needed.

In addition, it is important to promote a portable programming model for Web technologies, rather than a browser-specific one. To achieve this goal, we also provide the DOM 2 Core API[3] defined by the World Wide Web Consortium (W3C**). By using this API (W3C DOM2 for short), we ensure that Eclipse developers have portability across Web browsers and operating systems (Windows**, Linux** and Mac OS**).

Our approach is technically an extension of the approach for integrating native applications pioneered by SWT. SWT exposes a bridge in Java for the Microsoft native component framework COM (Component Object Model).[4] Our approach also supports the Mozilla component framework, XPCOM (Cross Platform Component Object Model).[5] With this foundation in place, it is relatively straightforward to define a SWT widget for embedding the different Web browsers, thus providing a common API for navigating HTML pages. However, providing a common DOM API across Web browsers is more challenging because of the slight variations between the existing DOM APIs in IE and Mozilla. We approach this in two steps. First, we map the native browser DOM APIs to Java. These straightforward but rather numerous mappings are automatically generated from the corresponding COM/XPCOM IDL (Interface Description Language) by a COM/XPCOM-IDL-to-Java-Class generator that we created. Second, we use these Java APIs to implement the W3C DOM2 API.

Integrating Web technologies significantly enhances the Eclipse platform. First, it enables the UI embedding of Web pages. This provides in-place integration of Web assets, such as Web applications or Web-based tools. One typical example is the integration of the HTML help system. Second, it enables a choice of UI technologies for plug-in developers. Eclipse views or perspectives may be developed using either markup or widget technologies. Java logic can interact with either Java widgets or HTML markup, providing a rich, seamless user experience that combines the strengths of both widget and markup paradigms. Our work opens up Eclipse to Dynamic HTML (DHTML)[6] for Java.

The rest of the paper is structured as follows. In the next section we review the necessary background on the Web browser technologies used by the IE and Mozilla browsers. In the following section we describe the design challenges we faced for integrating Web technologies in Eclipse. We first describe the bridging to the relevant foundation technologies—COM and XPCOM. Then, building on these bridges, we discuss our approach to the embedding of UI artifacts and to supporting DOM programming. In the section "Examples," we discuss several possible applications: online help systems, integrating development or administrative tools, and using Eclipse as an advanced application platform. In the last two sections we cover related work and conclusions.

## WEB BROWSER BACKGROUND

Web browsers have evolved from stand-alone markup viewers for hypermedia application platforms to today's desktop browsers—Mozilla and IE—which offer embeddable engines and the benefits of component frameworks.

XPCOM is the Mozilla binary framework for components and services. The Mozilla embeddable engine, Gecko,[7] is an XPCOM component. XPCOM is derived from Microsoft COM and runs on all operating systems that Mozilla supports, including Windows, Linux, and Mac OS. Mozilla as a whole is

designed as a collection of XPCOM components. Only after the XPCOM layer is initialized can an application load Gecko. Gecko's primary function is to provide markup rendering.

Gecko first parses markup documents into in-memory tree structures, called the Document Object Model (DOM). The rendering happens within a native window, either a parent or child window, as provided by the window manager of the platform on which Gecko runs. Within Gecko, the tree is manipulated through a DOM API, very similar to the one defined by the W3C consortium. Gecko then renders the DOM through its built-in support for HTML and style sheets (see ***Figure 1***). Through the DOM API, an HTML document can be created or incrementally modified. Whenever a DOM tree is modified, Gecko dynamically and incrementally re-renders that tree—this dynamic rendering is the foundation of DHTML.

Gecko also supports plug-ins that can be used to handle custom tags within HTML markup. These plug-ins are XPCOM components themselves. For example, when Gecko loads an HTML document, it extracts the JavaScript tags and passes them to its JavaScript interpreter, also an XPCOM component. Gecko uses the *<embed>* tag to handle browser plug-ins, triggering the loading of XPCOM components.

Although the IE architecture is similar to Mozilla's, it is based on COM, the Microsoft component framework from which XPCOM is derived. The rendering component of IE, the Web Browser Control, is a COM object that provides similar functionality to Gecko. The DOM APIs are provided as COM APIs. IE is as extensible as Gecko but uses COM for its plug-in model.

COM is a binary standard for object interoperability, based on virtual tables and calling conventions. COM is very similar to the binary model adopted by most C++ compilers. An interface maps to a virtual table that is implemented as an array of function pointers. All interfaces are subtypes of the root interface, called IUnknown. The IUnknown interface supports reference counting and casting from one interface to another (casting is querying an object, through one of its interfaces, to determine if it supports another interface). COM defines the concept of an object as supporting one or more

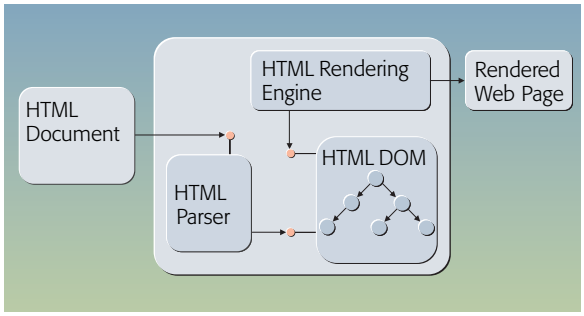


**Figure 1**
Web browser component

interfaces, but all interfaces of an object would return the same virtual table for the IUnknown interface. The address of this virtual table is considered the identity of the object.

COM objects are created by special kinds of objects known as factory objects. Factory objects, which are normal COM objects, declare themselves to the COM runtime as factories and identify themselves using globally unique identifiers. In fact, interfaces are also identified through globally unique identifiers. These unique identifiers are used when querying an object for an interface or asking the COM runtime to create an instance of a class. Best practices for COM include the rule that interfaces are immutable (once published, they will not be modified). After a component is included in a released product, any follow-on version of that component preserves the old interfaces for compatibility, and new interfaces are added as necessary.

## DESIGN CHALLENGES

For the integration of Web technologies in Eclipse, through either Mozilla or IE, we faced three challenges. We first had to provide bridges from Java to the underlying component frameworks, COM and XPCOM. Then we had to use SWT for embedding of HTML rendering within a widget-based user interface. Finally, we had to provide Java with a DOM API defined by the W3C consortium. In the following subsections, we discuss each of these challenges.

### Java-COM and Java-XPCOM bridges

As part of SWT, Eclipse includes a primitive bridge to COM for the Windows platform. It is a core technology that allows OLE (Object Linking and Embedding, Microsoft's earlier object-based tech-

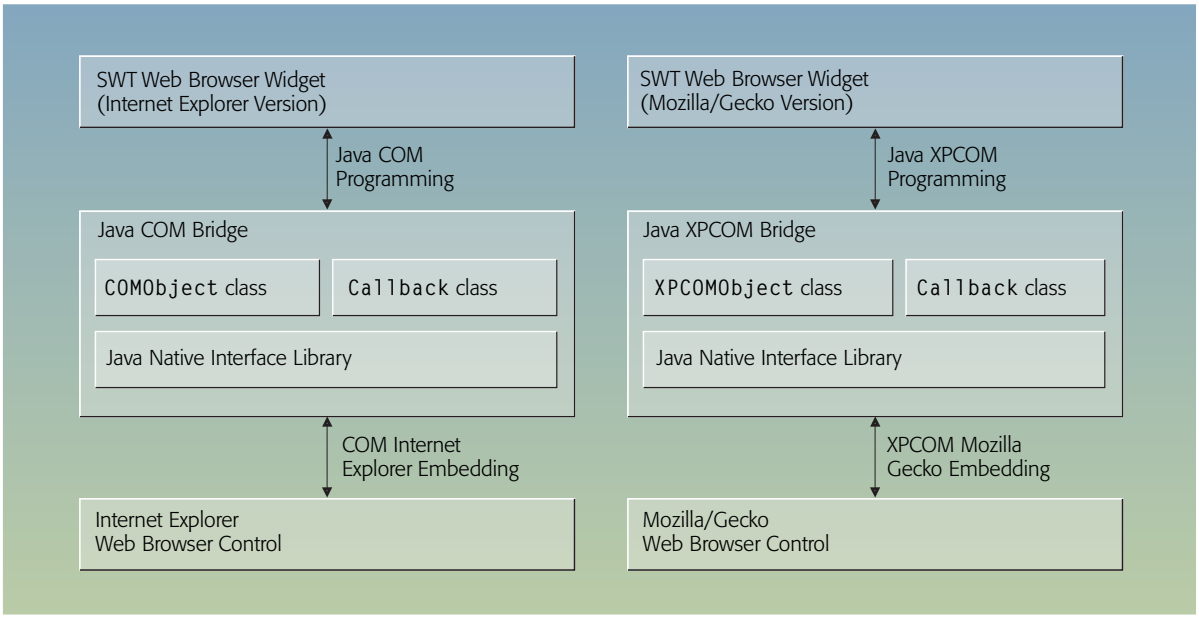


**Figure 2**
Embedding of Web browsers through COM/XPCOM programming in Java

nology) interoperability with SWT and thereby supports the visual embedding of OLE-compliant applications such as Microsoft Office.[8] However, the design of this bridge takes an ad hoc approach.

Specifically, Eclipse surfaces the COM framework through a class named `COM`. Its design imposes a single-thread execution limitation (which was kept throughout our implementation). This class incorporates all the constants that characterize the COM framework as well as the entry points to the COM runtime. It also provides support for creating object factories, which in turn create objects. Following the C background of the COM framework, reference counting is explicit, and it is the responsibility of the Java programmers to correctly increment or decrement the reference counts on COM objects. The `COM` class also supports calling methods on created COM objects through their virtual tables. The calls are carried through a set of fixed signature functions (function calls with a fixed set of parameters), with the virtual table index as the first parameter. Because this was not a general mechanism, it was necessary for us to make some minor additions. These included new fixed-signature function additions to the `COM` class, to accommodate calls to COM interfaces with method signatures not supported by the previous version of the `COM` class, and the creation of a several new COM-framework classes.

Most COM APIs further require that calling applications also provide COM objects that implement COM interfaces for callbacks. We therefore must be able to implement COM objects in Java. This requires the ability to create virtual tables at a binary level and expose them to COM. This is done with minimal C native code with most of the implementation in Java, specifically in the following two Eclipse classes: `Callback` and `COMObject` ("native" here means that we use Java Native Interface, or JNI, to invoke C modules).

As ***Figure 2*** illustrates, the `Callback` class provides native code with function pointers back to Java. In other words, it creates a C function stub that can forward a function call to a specific Java method of a specific object. The supported method signatures are limited to either multiple parameters of type integer or a single parameter of the type array of integers. This `Callback` class is used to construct Java virtual tables for COM objects.

The `COMObject` class is the superclass of all Java classes exported to COM. It can allocate and fill a native virtual table by using the `Callback` class and dispatching a set of abstract methods to be overloaded by a subclass. These abstract methods follow a simple naming pattern, $\text{method}_0$ to $\text{method}_n$, corresponding to the index in the virtual table. Each

instance of `COMObject` provides support for exporting one COM interface. One or more interfaces can be used to construct more complex COM objects.

Garbage collection issues, that is, coordinating the scopes of persistence between the explicit reference counting in COM and automated garbage collection in Java, are all of vital importance. In our case we had to consider these issues in both directions: Java proxies (the client) for native COM objects and Java objects (the server) implementing exported COM objects.

For Java proxies, Java programmers are expected to count references correctly, adding or releasing references. The Java proxy may then count Java references. The reference count has to go to zero before the proxy may actually become garbage from the perspective of Java garbage collection. When the Java reference count goes to zero, the actual reference to the underlying native COM object may be released. When the reference count is zero, the Java proxy is then invalid and should refuse method invocations.

For Java objects implementing exported COM objects, the situation is reversed. The reference count represents the number of native references (uses) held by native COM objects. It is important that the Java objects are not freed by the Java garbage collector as long as that count is not zero. Because of the way the `COMObject` class is implemented, instances of `COMObject` cannot be freed by the Java garbage collector until they have been "disposed." Therefore, the Java implementation of a COM object is responsible for counting its references accurately (across all its exported COM interfaces) and should dispose of the corresponding `COMObject` instance only after the count reaches zero.

XPCOM is very similar to COM, but the Java-COM bridge needed modifications. For practical reasons and the overall acceptance of our work within the Eclipse community, we decided to create a new bridge for XPCOM rather than modify the existing COM bridge. Because XPCOM was derived from COM, it was very easy to model our XPCOM bridge on the existing COM one. This new bridge allows us to interact with the XPCOM framework, the foundation of the Mozilla browser. With these two bridges, we have the necessary foundation in place.

### Embedding of UI artifacts

To embed markup-based UI artifacts in the Eclipse user interface, we needed to embed the rendering of Web pages within SWT. Our goal was to create a Web-browser widget, eliminating the differences among browsers and providing a simple and intuitive interface for browsing hyperlinked markup documents. As with most browsers, the widget has a concept of a current document and a history of visited URLs. The widget API provides support for setting and getting the current URL, navigating back and forth along the history, refreshing the current document, and stopping any ongoing download.

Our widget, `org.eclipse.swt.browser`, derives from the SWT `Composite` class. The implementation of our widget is different for each platform although the public interface remains the same. On Windows, we use OLE support, a higher-level API than the bare COM. First, we create a window (based on the `OleFrame` class) as a child element of our composite widget. Second, we create an instance of `OleControlSite`, mapped to the COM component named `Shell.Explorer`. This creates the embedded IE Web browser, using the `OleFrame` window to render the HTML documents "in place." This has the side effect of initializing `OleAutomation`, through which we can control the embedded IE and thereby implement our widget behavior.

For Mozilla, the steps are quite different because we have to use XPCOM. There is no equivalent to OLE in XPCOM. We need to initialize manually the XPCOM bridge and get access to the XPCOM Component Manager[9] in order to create a new instance of the Mozilla Web browser, composed of several XPCOM objects. However, the overall logic is similar to the IE version. The Web browser renders markup in a child window of our SWT composite widget. The widget controls the embedded Web browser through the Web browser's own API.

The embedded browser directly renders the HTML documents in the child window without any interaction with the surrounding SWT. This means that performance is unaltered. This also means that there is no dependency between the native toolkit used to render SWT widgets and the native toolkit used by the embedded browser. The use of a child window provides enough isolation. This means that

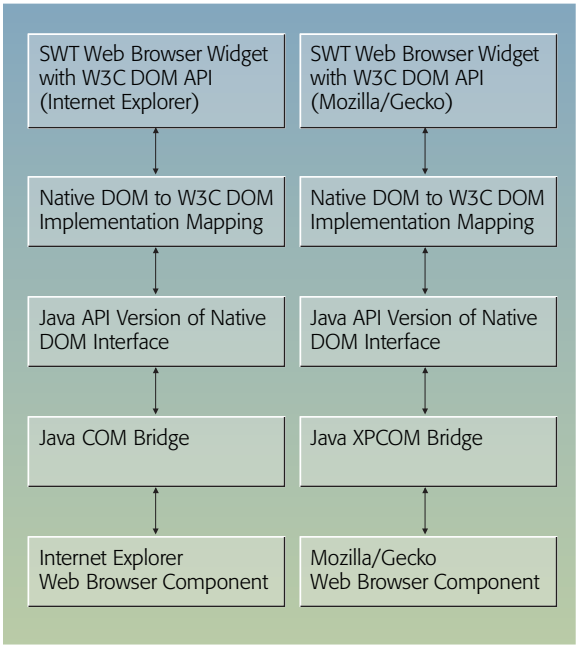


**Figure 3**
Implementing support for W3C DOM2 programming

any locally available Web browser could be used without widget toolkit compatibility.

### Support for DOM programming

DOM programming enables full control over the manipulation of the tree of HTML elements as well as the handling of DOM events. Our main goal was to provide one unique DOM API across platforms and browsers. We decided to implement the W3C DOM2 API in order to promote the standard as the common interface for programming Web browsers.

*Figure 3* illustrates the mapping from the SWT widget to the Web-browser component. This mapping is implemented in a number of steps.

Following SWT design philosophy, we decided to first map the native browser-specific DOM APIs to Java. This is illustrated in Figure 3 in the mapping layer labeled "Java API Version of Native DOM Interface." We wrote generators that automatically produce the Java proxies for the DOM APIs, defined in COM type libraries for IE and defined in XPCOM IDL for Mozilla. Given the large size of the HTML DOM2 API, an automated approach was needed, especially because the generated proxies are simple pass-through proxies. The advantage of this approach is that the full browser-specific DOM APIs are available, if needed.

Because of the limited portability of the browser-specific DOM APIs, we also provide the W3C DOM2 API that simplifies platform and browser migration. This is depicted in Figure 3 in the mapping layer labeled "Native DOM to W3C DOM Implementation Mapping." The mapping layer between the browser-specific APIs and the W3C DOM2 Core API is a very thin layer for both IE and Mozilla. Mozilla is very close to being compliant with the W3C DOM2 standard, while IE is not, but both are close enough so that the mapping is quite straightforward. Using the W3C DOM2 API, applications are provided with the following features:

- The ability to add, remove or modify any HTML element (node in the DOM tree) in the document
- Each element has its own methods and properties that can be programmed
- Each element has events to which a listener can hook.

The DOM API is exposed in a widget called `org.eclipse.swt.browser.DOMBrowser` that is implemented as a subclass of the basic browser widget called `org.eclipse.swt.browser`. The `org.eclipse.swt.browser.DOMBrowser` widget is shown in Figure 3 in the top layer called "SWT Web Browser Widget with W3C DOM API."

## EXAMPLES

In this section we describe some integration scenarios that use our technology to illustrate the power of combining Java and Web technologies.

### Example 1

Our first example is an online help system. As in many application environments, Eclipse has a help system, which is best provided through Web pages. Advanced help topics are best supported through dynamic pages, generated by Web applications running on a Web application server. For that purpose, Eclipse embeds the Apache Jakarta Tomcat servlet container as a plug-in.[10] With Eclipse 3.0, the help system creates an external help browser, with no visual integration or any ability to integrate with Java logic.

With our integration technology, a more unified experience is possible. The Web-browser widget can be used to embed help pages in a way that is fully integrated in the Eclipse Workbench environment. Any Web page with embedded JavaScript or CSS will display correctly because it is directly rendered by the embedded browser. Any limitation is due to the embedded browser, either Mozilla or IE. Beyond the visual integration, Java-based DHTML allows for coordination between help pages and the rest of the Eclipse platform. For example, HTML event handlers can affect the layout of perspectives (an Eclipse concept), integrate with the Workbench navigation subsystem, or impact menus and toolbars.

### Example 2

Our next example shows how our integration technology expands the tool-integration capability of Eclipse to include Web-based tools. Specifically, we consider tools for developing Apache applications. Central to such tools are WYSIWYG (what you see is what you get) editors that enable page designers to visually build HTML pages. With our integration technology, such editors can embed an actual Web browser to render the HTML pages. This avoids the need to build page preview technologies in Java and also provides an in-place rendering by the same Web browser technology that will render the page when it is deployed.

We are now able to simplify the deployment of Web-based applications, such as Web-based administrative tools for Apache. This can start with integrating the UI of Web tools in the Eclipse Workbench for a seamless experience. But our technology enables a much more powerful integration because the Web pages can be programmatically tied to the rest of the Eclipse platform. For instance, the Web pages showing the installed applications on the server can be linked with the Eclipse environment. Typically, if an application on the server is also a project in the Workspace, right-clicking the application on the Web page may provide a navigation menu to the local project, such as being able to open the application meta-data (e.g., `web.xml` in J2EE**). This would be done by hooking a Java listener to the DOM of the Web page on the particular nodes representing server-side applications. This requires an intimate knowledge of the HTML structure of the Web pages. This approach can be used to adapt existing Web applications to the surrounding Eclipse environment.

### Example 3

In another scenario new Web applications can be designed for the Eclipse platform as a Web-enabled client platform. The Web application developer would create an Eclipse plug-in as the front-end view of the application. For instance, an application can develop an Eclipse perspective, but the editors or views would be Web pages. Some pages may be static and local, whereas others may be dynamic and generated on the server. As explained above, those Web pages could be hooked with listeners for tighter integration to the Eclipse platform. But we can enhance this further by incorporating other features such as a pub-sub (publish-subscribe) mechanism between Web pages and shared data in the Eclipse environment.

In a previous paper, we discussed how to improve Web application performance and responsiveness by generating Web pages that contain a shared data model.[11] Using our technology, Web pages can be further enhanced by extending the range of data services with which a Web page can be integrated. The Eclipse plug-in in the preceding scenario can replicate the shared data using different replication protocols such as SyncML (an open-standard protocol for synchronizing data among machines, from handheld devices to corporate servers). Then, the plug-in may actually download the HTML pages, populating them locally through DOM manipulations with the relevant data from the shared data source. Data filtering and sorting can be easily applied to enhance the user experience. Additionally, if the data is modified through one view, the changes can be easily propagated to other views, providing a de facto local MVC (Model-View-Controller) paradigm among multiple views sharing one data model.

This type of integration takes the Web experience a step further. The user experiences not only a richer interface but also a more responsive system. Through DOM manipulations, round trips to the server are avoided, and rich and responsive interfaces that are seamlessly integrated into the Eclipse environment are provided. A front-end application can work in a disconnected mode, caching not only the data but also the Web pages. When connected, the front-end plug-in can communicate with server-side logic to manage data and Web-page updates.

## RELATED WORK

Our approach extends Eclipse's Rich Client Platform properties by integrating Web technologies in a way that is consistent with the SWT design philosophy, which promotes direct access to the underlying native widget toolkit and windowing system.

The integration of widget and Web technologies has been previously attempted. It was tried in Java with Swing.[12] Swing introduced the idea of HTML rendering as part of the Swing toolkit, but the rendering was done in Java—respecting Swing's all-Java philosophy. The Mozilla Blackwood project[13] was another attempt.

The Blackwood project is a collection of technologies that promote Java access to the Mozilla/Gecko Web browser component. One of the main features of Blackwood is BlackConnect, an object request broker (ORB) that provides first-class support for XPCOM in Java. Although BlackConnect goes beyond the SWT-style integration with XPCOM that we describe here, it interposes a layer of abstraction that masks the direct access to the native programming model. For example, it deals with threading issues and supports out-of-process invocation. The Blackwood project also provides programming access to the Gecko Web Browser Component via a scaled-down version of the DOM 2 API called JavaDOM.

The Blackwood project is probably the closest attempt to ours, regarding an integration of the Java and Web technologies. However, our goals and design points are different. (For example, we have a legacy to contend with, that is, the SWT design and especially its approach to integration with COM.) Our approach is one of a straightforward and direct mapping of native interfaces of the underlying platform. This approach helps with cross-platform compatibility, limiting non-Java code to a bare minimum. The Eclipse bridge to XPCOM corresponds to the lowest layer of the BlackConnect bridge, that which provides direct access to the binary standard of XPCOM—the virtual table layout. BlackConnect adds the creation and management of the stubs and proxies. In contrast, the COM and XPCOM programming in Eclipse is identical to the programming done in C/C++. This means that Eclipse programmers are responsible for reference counting and multithreading issues.

Aside from design issues at the XPCOM level, our overall goal was different. Following the Eclipse goal of portable frameworks, we wanted to promote a portable framework for Web technologies across Web browsers and operating systems. Hence, our approach handles both IE and Mozilla. It provides a common DOM programming framework based on W3C DOM Level 2 that is mapped to the browser-specific DOM APIs of IE or Mozilla. However, we also provide direct access to these browser-specific DOM APIs, allowing the possibility of leveraging specific capabilities when necessary.

Microsoft has pioneered some of the same concepts with COM and IE. Microsoft platform APIs provide complete integration between the COM component framework, the native widget toolkit, and the DHTML support in IE. But most of this support was previously unavailable to Java developers. Our work now brings Eclipse up to a similar level of functionality.

## CONCLUSION

In this paper, we describe an approach to integrating Web technologies in Eclipse. The core of our implementation, the native bridges and the UI embedding (Web-browser widget), has been incorporated into Release 3.0 of Eclipse. We are currently working with the Eclipse Foundation to include our support for DOM programming. This work is already in use today as a foundation technology within the IBM Workplace Client Technology.[14]

In general terms, our work opens Eclipse to DHTML-style programming in Java. Prior to this integration, DHTML was restricted to HTML page authoring. Eclipse applications now have the same level of functionality and integration as native Windows applications using IE and COM. Our approach has several advantages. The use of Java for programming logic avoids the performance issues of embedded JavaScript in HTML. The tight integration with the rest of the Eclipse environment enables applications that are richer in UI and functionality. Additionally, Eclipse applications are also portable across operating systems and Web browsers.

As our scenarios illustrate, this integration technology is a breakthrough for the Eclipse community. It enables the integration of Web-based tools and applications, thus expanding the integration role of Eclipse. It provides freedom of choice for plug-in

developers regarding UI technologies. Looking forward, it brings Java and Web technologies together as a solid foundation for the next generation of the Web. It opens the path for the Eclipse Rich Client Platform to be a first-class Web client, fronting Web application servers. The path leads to dynamic provisioning of Web application front ends that support a richer and more responsive user experience as well as a disconnected mode of operation. Our approach keeps down the total cost of ownership because it allows us to use existing Web applications while leveraging the power of the Rich Client Platform.

This work relates to two browsers, Mozilla and IE. However, our approach could be extended to other Web browsers—especially in the pervasive-computing world. Any browser that provides an embeddable engine can be quite easily integrated if the engine follows either COM or XPCOM conventions. Our approach requires that a browser provide support for UI embedding and DOM programming. If the DOM programming that a browser supports greatly deviates from the DOM2 API, such as browsers based on WAP (Wireless Application Protocol), we can extend our approach and expose only those DOM APIs that are available. For browsers that do not provide COM or XPCOM compliance, we can implement a direct JNI binding to whatever browser interfaces are available.

Our Web browser middleware built on DOM programming bridges the divide between the widget toolkit technology, used by traditional client-based applications, and the Web browser technology. The ability to use both widget and browser technologies in an integrated fashion provides application developers with a wider range of application integration possibilities. Using our technology, a legacy Web application can be integrated within any tool framework or Rich Client Platform application built with Eclipse. Similarly, existing Web pages can be dynamically modified to integrate with other Eclipse platform features, including presentation and data services. Our Web browser middleware also provides a rich palette of UI functionality to developers building new applications that require a dynamic UI layout engine beyond that found in the rest of the Eclipse SWT. Our support promotes W3C Web standards as the primary interface to program browser middleware and thus eliminates Web-specific dependencies and provides portability. Our technology accelerates the seamless integration of Web technologies in Eclipse, which will lead to new applications that share the benefits of these two worlds.



## CITED REFERENCES

1. *Eclipse.org*, Eclipse Foundation, http://www.eclipse.org/.
2. *Eclipse Rich Client Platform*, Eclipse Foundation, http://www.eclipse.org/rcp/.
3. W3C Document Object Model (DOM) Level 2 Specification, World Wide Web Consortium, http://www.w3.org/DOM/DOMTR#dom2.
4. D. Box, *Essential COM*, Addison Wesley, Boston, MA (1997).
5. D. Turner and I. Oeschger, *Creating XPCOM Components*, The Mozilla Organization (2003), http://www.mozilla.org/projects/xpcom/book/cxc/.
6. *HTML and Dynamic HTML*, Microsoft Corporation, http://msdn.microsoft.com/workshop/author/dhtml/dhtml_node_entry.asp.
7. *Embedding Mozilla*, The Mozilla Organization, http://www.mozilla.org/projects/embedding/.
8. K. Brockschmidt, *Inside OLE,* Microsoft Press (1995).
9. R. Parrish, *XPCOM Part 1: An Introduction to XPCOM*, developerWorks, IBM Corporation, http://www.ibm.com/developerworks/webservices/library/co-xpcom.html.
10. Apache Jakarta Tomcat, The Apache Foundation, http://jakarta.apache.org/tomcat/index.html.
11. J. Ponzo, et al."On Demand Web-Client Technologies," *IBM Systems Journal* **43**, No. 2, 297–315 (2004).
12. Java Foundation Classes (JFC/Swing), Sun Microsystems, Inc., http://java.sun.com/products/jfc/index.jsp.
13. *Blackwood Project: Java-to-Mozilla Bridge*, The Mozilla Organization, http://www.mozilla.org/projects/blackwood/.
14. *IBM Workplace Client Technology, Rich Edition*, IBM Corporation, http://www.lotus.com/products/product5.nsf/wdocs/workplaceclienttech.

*Accepted for publication February 11, 2005.*
*Published online April 26, 2005.*

***John Ponzo***
*IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (jponzo@us.ibm.com)*. Mr. Ponzo, an IBM Distinguished Engineer, has a B.S. degree in computer science from Manhattan College and an M.S. degree in computer science from Polytechnic University. His major areas of interest are the Web and its evolution, server-side and client-side

programming models, runtime architecture and design, and Web development tools. He is also interested in "rich client" applications, which provide a richer user experience and portal-like client-side aggregation. He made key contributions to several strategic IBM products, such as WebSphere® Studio and WebSphere Application Server. His research helped seed the Lotus Workplace vision, and he was closely involved with the making of the Lotus Workplace product, which is based on the Eclipse Rich Client Platform effort.

*Olivier Gruber*
*IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York, 10532 (orgruber@us.ibm.com)*. Dr. Gruber received his Ph.D. in the field of object systems from the University Pierre et Marie Curie in Paris, France, in 1992. For two years he was with the French national research institute for computer science (INRIA) where he led a European project on large-scale persistent object systems. He joined the IBM Research Division in 1995. During 1996–97, he led the core team of the first research prototype of an e-business Web server. That work, which demonstrated the importance and usability of Java, dynamic Web pages, personalization, and enterprise software components, opened the way for the development of WebSphere® Application Server. In the period 1998–2002, Dr. Gruber experimented with object technologies in support of ubiquitous access to information and applications by mobile users through pervasive devices. He also participated in the initiation of the Equinox project, which helped reshape Eclipse into a Rich Client Platform and led to the adoption of the OSGi™ technology. ■