

The Eclipse 3.0 platform: Adopting OSGi technology

O. Gruber
B. J. Hargrave
J. McAffer
P. Rapicault
T. Watson

From its inception Eclipse was mainly designed to be a tooling platform, but with Version 3.0, Eclipse is now evolving toward a Rich Client Platform (RCP). This change, driven by the open-source community, brought a whole set of new requirements and challenges for the Eclipse platform, such as dynamic plug-in management, services, security, and improved performance. This paper describes the path from the proprietary Eclipse 2.1 runtime to the new Eclipse 3.0 runtime based on OSGi™ specifications. It details the motivation for such a change and discusses the challenges this change presented.

Although Eclipse¹ was initially created to serve as an open platform for tools, its architecture was designed so that its components could be used to build essentially any client application. Now in Release 3.0, Eclipse has reinvented itself, evolving toward a Rich Client Platform (RCP).² The RCP is a natural progression toward integrating not only tools but also applications and services. In particular, the RCP involved the minimal set of plug-ins needed for this transformation, promoting not only modular development through plug-ins but also dynamic network provisioning. This RCP evolution, however, introduced new requirements and challenges for Eclipse.

In particular, the RCP needed to embrace dynamic management of components. With larger and more complex runtime configurations and usage scenarios, requiring restarts of the platform for changes in the runtime configuration to take effect was no longer acceptable. A service framework would nicely complement the extension framework promoted by the Eclipse registry. Indeed, service and extension

frameworks correspond to complementary software design patterns. Finally, it was important that security be addressed in the context of an RCP.

Considering these technical challenges, we rapidly realized that our main challenge would be managing change itself. The necessary changes mandated a quantum leap in the design and architecture of the core platform, but the planning, target dates, and the requirement for maintaining backward compatibility (that is, compatibility with previous versions) all favored instead a rather timid evolutionary path. A deep-seated overhauling of the Eclipse core runtime³ would be both risky and difficult to achieve without disrupting the rest of the Eclipse 3.0 planning process.

©Copyright 2005 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/05/\$5.00 © 2005 IBM

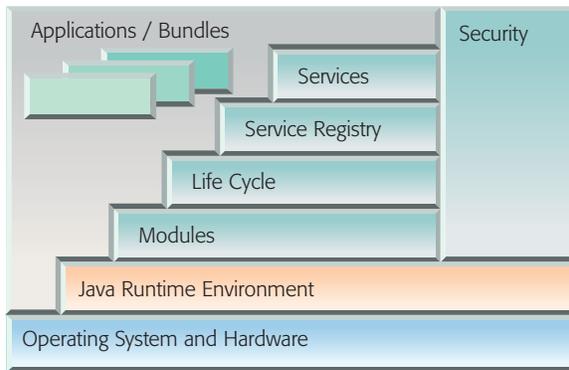


Figure 1
OSGi service platform architecture

We decided to try a new approach, one that would allow for fast experiments intended to scout for best solutions, but would also ensure that such solutions could be easily integrated into the main development stream of Eclipse 3.0. Specifically, we decided to leverage the existing Eclipse Technology Projects program.⁴ Technology projects are meant to explore adventurous ideas with a loose connection to the Eclipse mainstream.

Our technology project was called Equinox. From the start, we involved committers⁵ of the Eclipse platform team as well as external contributors. The goal was concrete: scouting practical solutions for direct inclusion in Eclipse 3.0. Through both Eclipse committers and external contributors, we achieved the right balance between fresh new ideas and an intimate knowledge of the Eclipse internal structure. The Eclipse committers provided invaluable connections to the other groups of Eclipse when it was time to transfer the results of Equinox back into mainstream Eclipse. Overall, this approach proved to be a successful one for managing core change in a mature open-source project.

In Equinox, we examined two primary technical avenues. One focused on using an Eclipse-specific runtime, evolving it to support the new requirements. The other involved evaluation of existing open standards providing similar functionality. In the end, we followed the latter course and chose an approach based on specifications from the OSGi^{**} Alliance.⁶ The OSGi Alliance provides an open standard that specifies the *OSGi Service Platform* (SP), a platform for network-provisioned software

components.⁷ Although the OSGi specifications seemed a natural choice, their adoption posed technical challenges as well as educational ones.

This paper traces the Equinox project, assuming familiarity with Eclipse 2.x.⁸ We briefly present OSGi technology in the following section and then discuss its adoption by Eclipse. The next section details the extensions to the OSGi specifications that we added to our implementation in order to support Eclipse. We also discuss the necessary changes to the Eclipse core runtime. Finally, we present conclusions derived from our work and discuss directions for future efforts.

OSGi SERVICE PLATFORM RELEASE 3

The OSGi Alliance publishes specifications⁹ that define the SP depicted in *Figure 1*. The SP was initially targeted at residential Internet gateways with home automation applications. It consists of a small layer above a Java^{**} Virtual Machine (JVM^{**}) that provides a shared platform for network-provisioned components and services. It is shared by different providers of components, potentially across organizational boundaries. It provides an extensive security model and at the same time promotes cooperation and reuse between components. The most attractive features of the platform are its long-running design based on a service-oriented architecture and its ability to support dynamic updates with minimal perturbation of the running environment.

Component and service model

The SP embodies a component and service model. An OSGi component, or *bundle*, is a set of Java packages containing both classes and resources, essentially what would be traditionally packaged in a JAR (Java archive) file. The difference, however, is that the SP manages such bundles and their dependencies that are expressed as meta-data attached to each bundle.

Each bundle expresses its dependencies at the level of Java packages. A bundle explicitly imports the Java packages that it needs but does not itself define. Conversely, a bundle may export some Java packages that it defines. The SP automatically matches imports and exports based on name equivalence. This matching process is dynamic as bundles are installed or uninstalled. A bundle is said to be resolved when all its dependencies are met, that is,

when exporters for all the Java packages it imports are available. When a bundle is resolved, its exported Java packages automatically become available for matching to the import requirements of other bundles.

Above the component model, the SP provides a service-oriented architecture. In particular, it relies on the ability to activate a bundle. In other words, after a bundle is resolved, it may be started and stopped. A bundle may be activated if it defines an *activator class*. When a bundle is started, the SP creates an instance of that activator class, called the *activator object*, on which it calls the methods `start` and `stop`.

The `start` provides the bundle its context. Using that context, the bundle code may publish, find, and bind services. An OSGi service is a plain Java object. Once created, the service object may be registered. A registered service is identified through its Java interface and optional meta-data, provided as name-value pairs. A service may be located in the service registry through simple LDAP (lightweight directory access protocol)-like queries on interfaces or through meta-data. The service registry is fully dynamic; services may be registered or unregistered at any time by an active bundle.

Management agent model

The SP presents a simple and intuitive model for Java programmers, advocating components and services. It goes beyond a simple Java Runtime Environment (JRE) in that it also provides a shared platform with a managed configuration. As such, the *management agent* is a key component of the SP.

The management agent is a normal bundle, but with administrative privileges. It is responsible for managing the runtime configuration of the environment in which it runs. A management agent may be very simple or very complex. An example of a simple agent is one that installs a predetermined set of bundles from a bundle server on the Web. The set of bundles is assumed to be consistent, perhaps because a human administrator has tested that configuration. This would be the case for a small device like a cellular phone, where the service provider would have tested different configurations for different cellular phones.

In the presence of a more dynamic environment, the overall consistency of the runtime configuration is the sole responsibility of the management agent. This overall responsibility translates into the specific tasks that follow:

1. *Installing bundles that will resolve*—When a set of bundles is installed, the runtime attempts to match imports and exports based on names and versions. Some bundles may resolve while others may not. An agent wants to maximize resolved bundles. Indeed, unresolved bundles waste resources and contribute neither Java packages nor services. It is important to point out, however, that having unresolved bundles may be unavoidable at times.
2. *Managing the active configuration*—This is an explicit process involving the starting and stopping of bundles. It is the responsibility of the agent to decide which bundles to start and when. This is independent of overall shutdowns and restarts, because the SP offers a persistent environment wherein active bundles are automatically stopped and restarted across shutdowns and restarts.
3. *Overall security, that is, setting the permissions for bundles*—The SP is a secure environment that relies on Java 2 security. OSGi specifications define permissions for the entire set of APIs (application programming interfaces). There are administrative permissions, as mentioned above, but there are also permissions for importing or exporting a Java package and for publishing or finding a service. The permissions are associated with a bundle through the *bundle location*, which is a string associated with a bundle by the management agent when it first installs that bundle.
4. *Ensuring the correctness of name equivalence*—To better understand this responsibility, we must examine bundle dependencies in more detail. As mentioned, bundle dependencies are expressed at the level of Java packages; that is, a bundle may import or export Java packages. But the SP distinguishes between two very different types of packages, namely, *specification* and *implementation* packages.
 - a. Specification packages correspond to formal specifications. The very first property of specification packages is that a package name and version completely define the

contents of that package, across all providers. Indeed, providers may not augment or reduce the contents of specification packages in any way. The second property of specification packages is that versions are forever compatible with previous versions, as dictated in Reference 10.

- b. Implementation packages are very different and have none of the preceding properties. For implementation packages, there is no entity protecting the package name or guarding its contents. Furthermore, there are usually no globally accepted version numbers, and versions may grow incompatible over time. Therefore, a Java package name and version may not represent the same contents with different providers. It is consequently crucial that the management agent ensure that name equivalence is used properly. In other words, it is the responsibility of the management agent to ensure that name equivalence is correct between the import and export statements of the bundles it installs.

Runtime environment

The service platform is a thin layer above a JVM. It requires a minimal Java runtime environment that is compatible with J2SE** (Java 2 Platform, Standard Edition) and many Java profiles in J2ME** (Java 2 Platform, Micro Edition). The SP is a single-JVM environment; it is not a distributed environment across processes or across machines.

The SP hosts a persistent runtime configuration. From within this environment, the management agent creates and maintains the runtime configuration by installing, updating, or uninstalling bundles. The SP also remembers this configuration across shutdowns. In particular, it remembers the installed bundles, and it also remembers which bundles were active at the time of shutdown and should therefore be restarted automatically at the next startup.

Because of the persistent nature of the runtime configuration, the SP manages the physical contents of the installed bundles. As mentioned previously, a management agent uniquely identifies a bundle by a location. This location is used by the SP to determine the contents of that bundle in a platform-dependent way. For example, the location could be a URL (Uniform Resource Locator) from which the con-

tents could be obtained as an input stream. Alternatively, the URL could be a file URL and point to a directory where the bundle contents can be found.

The approach is quite flexible, but there are constraints. The first constraint is that the platform must be able to understand the location string in order to access the contents, unless these contents are provided directly through an input stream. The second constraint is that the platform must be able to create a class loader for the bundle contents. Indeed, the SP uses a class loader for each bundle combined with a directed delegation mechanism to support Java package imports.

ECLIPSE 3.0

When the Equinox project began, the goal was to explore more advanced technologies for the Eclipse platform. Improving on the Eclipse 2.1 runtime was a natural first choice. However, as noted previously, adopting an open standard rather than extending an Eclipse-specific technology rapidly became attractive, to the mutual benefit of both the Eclipse and open-standard communities. Our eventual decision was to adopt OSGi technology because this technology had very strong points in its favor. As a fully dynamic environment, the OSGi technology provides a solution to one of the most serious limitations of the Eclipse 2.x platforms. OSGi technology also defines a service framework and a security framework. However, the adoption of this technology also presented serious challenges because the Eclipse and OSGi environments had architectural, design, and philosophical differences.

This section discusses these challenges and describes the solutions we adopted initially in Equinox and which were later included in Eclipse 3.0. Eclipse 3.0 does not embed the full set of OSGi specifications, but rather a small subset centered around the module, life-cycle, and service concepts. Our implementation also includes extensions required to support Eclipse-specific needs. These extensions have been made, however, in a way which is fully compatible with OSGi specifications. These extensions are in turn being proposed to the OSGi Alliance for inclusion in its next version (Release 4).

Architecture overview

Eclipse 3.0 adopted the OSGi Service Platform (SP) as a foundation, evolving and improving its own architecture accordingly. It is important to point out

that Eclipse provides a core runtime that lies above the SP, as depicted in *Figure 2*. The SP provides the core model for components and services, but Eclipse provides the concept of plug-ins, which are fully managed by the Eclipse core runtime.

Above the SP, all components are bundles, specifically including the components implementing the Eclipse core runtime itself, which includes the plug-in registry, the backward compatibility component, the configurator, and the update manager. The plug-in registry and the configurator work closely together to deliver the traditional Eclipse environment for plug-ins. The configurator manages the runtime configuration, installing or uninstalling specific plug-ins from the available set of downloaded plug-ins. The update manager downloads plug-ins from Eclipse Web sites onto the local file system; these plug-ins are later installed in the runtime configuration by the configurator.

To achieve this division of responsibilities between the configurator and the update manager, we used the semantics of the bundle locations. The location is merely an identification string in Eclipse. The platform is not able to locate the bundle contents from locations; instead bundle contents are provided indirectly through a special input stream that refers to the directory on the local file system where the bundle can be found. As a side note, our implementation does also support input streams that directly provide contents, in which case the SP reads the stream and expands its contents into the local file system.

This approach allows us to share bundles throughout multiple runtime configurations, which is very important for three reasons. First, it supports more efficient handling of libraries. If bundles are kept in JAR files, the JVM does not know how to load native libraries from them, requiring that native libraries be manually extracted and managed somewhere on disk. Second, it allows the installation of different Eclipse-based products, running as different runtime instances, but sharing some or all of their plug-ins on the local file system. Third, it supports the Eclipse *self-hosting* philosophy.

Self-hosting relates to the fact that the tools to develop plug-ins are themselves developed as plug-ins. In other words, Eclipse provides a set of plug-ins, called the Plug-in Development Environment

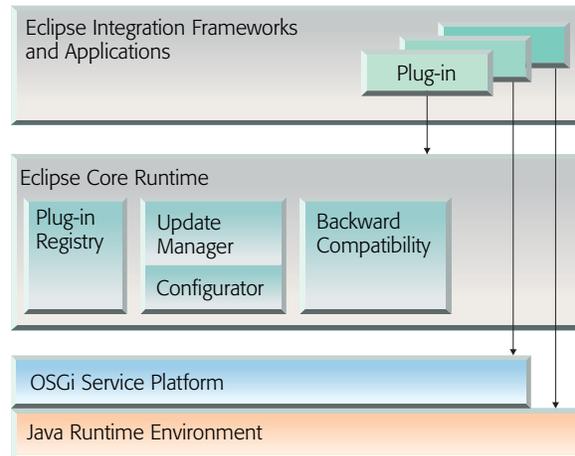


Figure 2
Eclipse architecture overview

(PDE), aimed at developing plug-ins. With PDE, there is a *host instance* of Eclipse running PDE and a *target instance* of Eclipse running the developed plug-ins. A typical scenario for developing new plug-ins is the following.

The developed plug-ins are projects in the workspace of the host Eclipse. These plug-ins are developed to run within the same runtime configuration as the host Eclipse. When debugging, the host Eclipse launches a target Eclipse, with the same runtime configuration as itself plus the plug-ins developed in the workspace. This runtime configuration is computed by the PDE and stored in a configuration file, which is used by the configurator of the target Eclipse to create the correct runtime configuration. None of the plug-ins are actually copied. Rather they are all shared between the host and target instances of Eclipse.

When not in self-hosting mode, Eclipse follows a persistent approach; it simply restarts in the same state as that from which it was shut down. This requires remembering the runtime configuration across shutdowns, which is the traditional OSGi approach. However, Eclipse also allows the configuration to be modified while Eclipse is shut down, in particular, by adding or removing plug-ins directly in the file system. To support this capability, the configurator must compare the last-known configuration and the current on-disk configuration, installing or uninstalling bundles accordingly.

```

<plug-in> <!--org.eclipse.core.runtime plug-in -->
  <extension-point id="applications"
    name="%applicationsName"
    schema="schema/applications.exsd"
  />
</plug-in>

<plug-in> <!--org.eclipse.ui.ide plug-in -->
  <extension
    id="workbench"
    point="org.eclipse.core.runtime.applications">
    <application>
      <run class="org.eclipse.ui.internal.ide.IDEApplication"></run>
    </application>
  </extension>
</plug-in>

```

Figure 3
XML definitions for the Application extension point and one of its extensions

This separation of the configurator and the update manager provides a much cleaner architecture than was the case in earlier versions of Eclipse. Previously, the functionality of the configurator was buried deep in the Eclipse platform, making the Eclipse core runtime aware of and dependent on the details, or *features*, of a runtime configuration as seen by the Eclipse update manager. The new OSGi-based implementation allows a clean separation between the core runtime and management issues. This is important because the goal is for Eclipse 3.0 to be able to operate regardless of the particular management protocol that is used for its dynamic provisioning.

Plug-ins versus bundles

Plug-ins are bundles that are managed by the Eclipse core runtime. Some plug-ins may declare extensions or extension points. This extension framework provides support for extensibility and is one of the core concepts of Eclipse. The extension framework and the service framework are complementary, and both leverage the component framework. More specifically, the extension framework defines the concepts of *extension point* and *extension*.

An extension point is defined by a plug-in within the namespace (a set of names that is defined according to some naming convention) of that plug-in. For example, if the plug-in `org.eclipse.core.runtime` defines an extension point `Application`, then the full name of the point is `org.eclipse.core.runtime.Application`. In other words, the name of a plug-in

defines a namespace for the extension points that the plug-in defines. An extension point is a recipient for extensions and defines the schema of the extensions it accepts. In turn, the schema of an extension defines, in the XML (Extensible Markup Language) sense, the characteristics of that extension. Both extensions and extension points of a plug-in are defined in an XML *manifest* for that plug-in.

The XML definitions for the above `Application` extension point and one of its extensions are shown in **Figure 3**. The first XML snippet appears in the manifest of the `org.eclipse.core.runtime` plug-in. The second XML snippet appears in the manifest of the `org.eclipse.ui.ide` plug-in, which defines the `workbench` application.

More details on the extension framework are beyond the scope of this paper, but it is important to briefly discuss this framework in comparison to the service framework provided by the SP. Extensions and services correspond to two different design patterns. Extensions address the extensibility of a component, whereas services address the more traditional requirement of interoperability. Not only did we need to keep both patterns, but both patterns are valuable for an RCP.

Besides extensions and extension points, an Eclipse plug-in may have *fragments*. Conceptually, a fragment adds contributions to the class path (a listing of locations where Java can expect to find class files) of its host plug-ins. The OSGi specifications define

no equivalent support. Without specialized support, a management agent would have to update the host plug-in after having modified its contents on disk. We decided this approach was too complex and instead added specific support for fragments to our SP, extending the concept of bundles. A typical role for fragments involves delivering resources, such as natural language property files or operating-system-dependent libraries.

A bundle may be either a host plug-in or a plug-in fragment. When a bundle is a fragment, it identifies its host bundles using symbolic names and versions. Symbolic names and versions for bundles do not exist in the OSGi specifications; thus, they are another extension required in our implementation. When a host is resolved and has fragments, the class paths of the fragments are added to the class path of the host. Therefore, installing a fragment may require reloading its hosts, although this can be avoided in many practical cases.

Plug-in dependencies

Eclipse and the SP have quite different understandings of runtime configurations. Although this initially appeared to be a difficult challenge, we found that the two complementary approaches actually resolved each other's limitations.

Eclipse relies on a radically different dependency model, based on requiring plug-ins. Rather than simply importing the Java packages it needs, a plug-in requires other plug-ins. In other words, a plug-in does not state which Java packages it needs, but rather indicates where they can be found. Because plug-ins often embed both their API and their implementation, the Eclipse model captures implementation dependencies. Furthermore, because most plug-in names are in fact prefixed with their provider names, a plug-in dependency relates to an implementation from a specific provider. This provides a very robust description of a product release. The Eclipse approach works best for implementation packages for which the correctness of name equivalence is hard to ensure across bundle providers, or potentially across organizations.

In comparison, the SP captures more specification dependencies and provides a very flexible matching mechanism. In particular, an import may be matched to any export from any bundle, as long as the name and version match. This flexible approach

works well with specification dependencies, and is especially well-suited for a service-oriented architecture that advocates a strong independence between specification APIs and implementations. Examples include specifications such as W3C** (World Wide Web Consortium),¹¹ DOM (Document Object Model),¹² SAXP (Simple API for XML Parser),¹³ or Servlet APIs.¹⁴

We decided to support both models in our runtime. A bundle may import and export specification packages, following the OSGi model. However, a bundle developer is encouraged not to use import or export statements for implementation packages, but rather to use our new mechanism designed after the Eclipse plug-in dependency. This new mechanism works as follows. A bundle B_1 may *require* another bundle B_2 . This means that B_1 will see all packages that bundle B_2 *provides*. The bundle B_2 provides packages by specifying them in its manifest, similar to the way it would specify packages that it exports.

Bundle B_1 requires bundle B_2 by name and version, reusing the same mechanism put in place for fragments. The bundle names typically follow the Java naming convention for Java packages (domain prefixed). The version model uses four tokens: *major*, *minor*, *service*, and *qualifier*. Versions may be compared at each of these different levels. Major versions may be incompatible. Minor versions are compatible. Service and qualifier versions are minor compatible evolutions, such as bug fixes.

The combination of the two models is quite powerful and incorporates the strengths of both approaches, providing greater flexibility when semantically correct and controlled flexibility when required. One very important feature is that the combined models allow concurrent loading of different versions of the same bundle. For implementation packages, each bundle is a different namespace for the packages it provides. Therefore, although they have the same package name, the Java packages provided by two versions of the same bundle are considered different. This is not possible in the OSGi model and enables handling of the more complex runtime configurations that may be required in large systems.

Eclipse registry

The Eclipse registry manages plug-ins. More specifically, given a set of plug-ins, the Eclipse registry

manages their *contributions* (i.e., their extensions and extension points).

In earlier versions of Eclipse, the registry was computed at startup. All installed plug-ins were processed and their contributions added to the registry. Each plug-in then queried the registry to discover the contributed extensions to its extension points. The plug-in would usually process that list of extensions, building internal data structures. Nothing would change until the next shutdown-restart cycle of the Eclipse platform. With Eclipse 3.0 this is no longer the case.

The first change in Eclipse 3.0 is that plug-ins may come and go, and thus the list of plug-ins changes over time. We originally planned to use the life-cycle events defined by the OSGi specifications to determine which plug-ins are installed or uninstalled. Indeed, the SP triggers *events* when a new bundle is installed or an existing bundle is uninstalled. However, this is not sufficient; it is also important to know whether a bundle was resolved or unresolved. Indeed, the registry can only consider the contributions of resolved bundles. If a bundle is unresolved, the registry must discard its contributions, even if the bundle is still installed in the runtime configuration.

We simply added two new life-cycle events to our OSGi implementation: *resolved* and *unresolved*. The registry could then listen for those two events and dynamically add or remove contributions of plug-ins. Thus, the registry is able to maintain itself properly, but the challenge becomes to broadcast to plug-ins the changes in the registry itself. As contributions come or go, the registry needs a way to broadcast life-cycle events in order for plug-ins to react appropriately. We favored the *delta* mechanism, well-known to Eclipse developers. A delta groups change events into a data structure, as opposed to sending individual events for each change. The delta mechanism allows anyone to listen to such life-cycle events. Each delta applies to a plug-in namespace and includes change events for added or removed extensions and extension points in that namespace.

Eclipse execution model

Eclipse advocates an automated model for activation. Plug-in developers must implement the start and stop methods for their bundles, but they do not

have to worry about when or how these methods are called. This is a simple programming model that further enables a lazy and efficient approach for our runtime, allowing it to scale up to thousands of plug-ins. The idea is to avoid overeager creation of class loaders and initialization of plug-ins.

The first issue is to delay the creation of the class loaders for resolved bundles. The problem is as follows. When a plug-in becomes resolved, the Eclipse registry has to parse its plug-in manifest (`plug-in.xml`) in order to discover its contributions. However, the OSGi API does not allow accessing the contents of a bundle without creating a class loader. In fact, most OSGi implementations seem to be eager to create class loaders when bundles resolve.

We avoided the problem in two steps. First, we adopted a lazy approach that delays the creation of class loaders until absolutely necessary—the first class load. Second, we introduced a new API that allows accessing the content of a bundle without requiring a class loader to be created. This enables the Eclipse registry to access the plug-in manifest and add its contributions, without activating the plug-in or even triggering a class loader creation.

The second issue is deciding when to activate a plug-in. We consider that a plug-in needs to be activated before any loading of a class that it defines. This ensures that a plug-in will be initialized before any of its functionality can be used. Our implementation hooks into the class loader creation (on first class load) and triggers plug-in activation, calling the bundle start method.

Backward compatibility

An essential requirement of evolving a core runtime of a mature system is to offer strong backward compatibility. It was extremely important to be able to run Eclipse 2.x plug-ins on the new runtime because most plug-ins would have to run in backward compatibility mode until made aware of the dynamic registry. Even as Eclipse 3.0 was shipped, many current plug-ins were still static, unable to react to events regarding extensions and extension points appearing or disappearing in the Eclipse registry.

Despite the importance of backward compatibility, we wanted to make it optional so that it could be *deprecated* some day. (Deprecation is the declara-

tion that a component should not be used in subsequent designs, but remains available to support existing designs that incorporate it.) Thus, backward compatibility is simply provided through an optional plug-in (`org.eclipse.core.runtime.compatibility`), which, when installed, allows an Eclipse 2.1 plug-in to be loaded without any modification, even to its meta-data. In other words, an Eclipse 2.1 plug-in may simply be dropped into the file system, and the plug-in will work the next time Eclipse is started.

To achieve this, we automatically translate an Eclipse 2.1 plug-in into an Eclipse 3.0 plug-in. In particular, this means generating the bundle manifest dynamically. This can be achieved quite easily by mapping plug-in requirements into bundle requirements. Once the manifest is generated, an Eclipse 2.1 plug-in can be loaded by our SP just like any other bundle. We also create an activator for the bundle, which interconnects the new OSGi life cycle and the old Eclipse 2.1 life cycle, creating and activating the Version 2.1 plug-in object as necessary.

CONCLUSION

The move from Eclipse toward an RCP has received very positive reaction so far. At the Eclipse conference in Anaheim in 2003, the very large attendance at RCP-related tracks pleasantly surprised everyone. It confirmed that the Eclipse community perceives Eclipse as an integration platform not only for tools but also for applications and services. We feel confident that this positive reaction, combined with the already existing momentum of OSGi technology in the industry, together suggest exciting times to come for Eclipse, the OSGi Alliance, and the open-source community at large.

One very important lesson drawn from this project regards managing drastic change in an already successful open-source project. Although it may still be early to draw absolute conclusions, using the Eclipse technology project framework with both internal and external contributors provides an attractive approach, especially if the project is focused on technologies intended for direct inclusion back into the main development stream. External contributors are key to bringing new ideas, new skills, and of course additional manpower that allows for the involvement of just a limited number of committers. Involving committers provides ex-

perience and wisdom about the current state of the project as well as better understanding of the trade-offs of various possible solutions. Committers also make final acceptance realistic. Without the involvement of committers throughout the process, real understanding of the solutions and their final acceptance become less likely, regardless of their merits.

Another very interesting aspect of Equinox was the adoption of an open standard. We believe that adopting the OSGi specifications has already proved beneficial to both communities. Involving the OSGi Alliance within Equinox was very beneficial from the start, with OSGi experts providing valuable know-how and insights on problems and potential solutions. In the long term, we are proposing some of our extensions to the OSGi Alliance for adoption in the next release, R4. We also hope that the OSGi Alliance will benefit from the new usage scenarios that Eclipse brings. Additionally, we look forward to seeing better tools for helping to develop, debug, and monitor applications for OSGi platforms.

Equinox began in early 2003. After approximately eight months of exploring different technical avenues, the OSGi technology was adopted as the favored option within Equinox. Three months later, Eclipse accepted this conclusion, and the Equinox OSGi-based runtime was integrated into the main Eclipse code base. Thus, the Eclipse core runtime was overhauled in less than a year, with minimal impact on the layers above the core runtime. Overall, Eclipse 3.0 shipped with the new runtime only eighteen months from the start of this project.

One key aspect of this success was certainly the strong commitment of Equinox to backward compatibility. In particular, we worked hard to produce a backward compatibility layer that allows Eclipse 2.1 plug-ins to run without modification. Unfortunately, this also meant that progress toward some of our other goals had to be delayed, because leveraging some of the most salient features of OSGi involves more pervasive impact throughout the Eclipse programming model and corresponding tools. At the time of this writing we are just beginning phase 2 of the Equinox project, aimed at enhancements requiring these more deep-seated changes.

This second phase involves interesting and challenging issues. First, security needs to be addressed.

Although the SP is a secure environment, we need to understand and master the implications of security at the level of the Eclipse core runtime and above (integration frameworks). In addition, the impacts of a dynamic environment on plug-ins have not been fully mastered. We need tools to help develop dynamic plug-ins and to diagnose erroneous ones. Furthermore, the programming model should evolve to integrate the OSGi service framework. Finally, scalability is a never-ending pursuit, including both upward and downward scalability, that is, the ability both to manage large complex configurations and to understand how to scale down the Eclipse RCP for small pervasive devices.

**Trademark or registered trademark of Massachusetts Institute of Technology, Sun Microsystems, Inc., or The OSGi Alliance.

Cited references

1. Eclipse.org, The Eclipse Foundation, <http://www.eclipse.org/>.
2. E. Burnette, *Rich Client Tutorial*, <http://www.eclipse.org/articles/index.html>, and references therein.
3. For a detailed description of the role of the runtime, see Eclipse Runtime, The Eclipse Foundation, <http://help.eclipse.org/help30/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/runtime.htm>.
4. Eclipse Technology Project Home Page, The Eclipse Foundation, <http://www.eclipse.org/technology/>.
5. Committers are Eclipse developers who have made frequent and valuable contributions to a project or a component of a larger project, and thus have been granted access to the relevant source-code repository as well as voting rights allowing them to affect the future of that project. For more detailed information see The Eclipse Project—Top Level Project Charter, The Eclipse Foundation, <http://www.eclipse.org/eclipse/eclipse-charter.html>.
6. The OSGi Alliance, <http://www.osgi.org/>.
7. *About the OSGi Service Platform*, OSGi Alliance (July 12, 2004), http://www.osgi.org/documents/osgi_technology/osgi-sp-overview.pdf.
8. *Eclipse Platform Technical Overview*, The Eclipse Foundation (February 2003), <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>.
9. OSGi Service Platform, Release 3 Specifications, OSGi Alliance, http://www.osgi.org/resources/spec_download.asp.
10. *Java Product Versioning Specification*, Sun Microsystems, Inc. (November 30, 1998), <http://java.sun.com/j2se/1.3/docs/guide/versioning/spec/VersioningTOC.html>.
11. World Wide Web Consortium (W3C), <http://www.w3.org/>.
12. Document Object Model (DOM), World Wide Web Consortium, <http://www.w3.org/DOM/>.
13. Class SAXParser, Sun Microsystems, Inc., <http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/SAXParser.html>.
14. Java Servlet Technology, Sun Microsystems, Inc., <http://java.sun.com/products/servlet/>.

Accepted for publication November 11, 2004.

Published online April 26, 2005.

Olivier Gruber

IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (ogruber@us.ibm.com). Dr. Gruber received a Ph.D. degree in the field of object systems from the University Pierre et Marie Curie in Paris, France (1992). For the next two years he led a European project on large-scale persistent object systems at the French national research institute for computer science (INRIA). He joined IBM Research in 1995 and since then has alternated between research phases on advanced object systems and transfer phases with high impact on IBM business. Since 2002, he has been a decisive force working toward the recently announced IBM Workplace. In particular, he was involved in the origins of the Equinox project and advocated the adoption of OSGi technology.

B. J. Hargrave

IBM Software Group, 11501 Burnet Road, Austin, TX 78758 (hargrave@us.ibm.com). Mr. Hargrave has over 18 years of experience as an IBM software architect and developer. His focus is small computer operating systems (kernels, file systems, development tools, application binary interface specifications) and Java technology. He holds multiple patents for JVM performance improvements and is the IBM expert and lead developer for OSGi technologies. He holds a B.S. degree in computer science from Rensselaer Polytechnic Institute and an M.S. degree in computer science from the University of Miami. He has been a leader in the development of the OSGi technology since its inception and was named an OSGi Fellow during the 2002 OSGi World Congress for his technical contributions and leadership. He is currently Chief Technical Officer of the OSGi Alliance and chair of the OSGi Core Platform Expert Group.

Jeff McAffer

IBM Software Group, Ottawa Laboratory, 2670 Queensview Drive, Ottawa, Ontario K2B 8K1 (Jeff_McAffer@ca.ibm.com). Dr. McAffer leads the Equinox project. He is one of the architects of the Eclipse platform and has been involved in the project from the beginning. His current interests lie in helping to realize Eclipse's original vision as a platform for composing general sets of application function, involving, in particular, such areas as dynamic plug-ins and alternate runtime models. Previous lives included work in distributed/parallel object-oriented computing (Server Smalltalk, massively parallel Smalltalk, etc.) as well as expert systems, and meta-level architectures. He received a Ph.D. degree from the University of Tokyo.

Pascal Rapicault

IBM Software Group, Ottawa Laboratory, 2670 Queensview Drive, Ottawa, Ontario K2B 8K1 (Pascal_Rapicault@ca.ibm.com). Mr. Rapicault has been a developer with IBM Ottawa labs (formerly Object Technology International) since 2002. He played a key role in Equinox and the successful adoption of OSGi into Eclipse 3.0. He continues to work on the Eclipse 3.x platform. Dr. Rapicault holds a Master's degree from the ESSI (France) and a Ph.D. degree from the University of Nice (France).

Thomas Watson

IBM Software Group, 11501 Burnet Road, Austin, TX 78758 (tjwatson@us.ibm.com). Mr. Watson is a lead developer of IBM's implementation of the OSGi Framework. He has been

involved in the Equinox project and the adoption of the OSGi Framework as the new component runtime for Eclipse 3.0. His current interests lie in participating in the OSGi Alliance to help define a modularity layer that can be used in a broad range of environments, including Eclipse and Web application servers. His previous experience in OSGi includes design and development of embedded web container and service gateway software that controls home appliances. Before his involvement with OSGi and the Pervasive Computing Division, he worked for the Network Computer Division. ■