# On Plug-ins

## and Extensible Architectures

DORIAN BIRSAN, ECLIPSE

Extensible application
architectures such as Eclipse
offer many advantages,
but one must be careful to
avoid "plug-in hell."

In a world of increasingly complex computing requirements, we as software developers are continually searching for that ultimate, universal architecture that allows us to productively develop high-quality applications. This quest has led to the adoption of many new abstractions and tools. Some of the most promising recent developments are the new *pure plug-in architectures*.

What began as a callback mechanism to extend an application has become the very foundation of applications themselves. Plug-ins are no longer just add-ons to applications; today's applications are made entirely of plug-ins. This field has matured quite a bit in the past few years, with significant contributions from a number of successful projects.

This article identifies some of the concepts around the foundation of pure plug-in architectures and how they affect various stakeholders when taking a plug-in approach. The intention is to discuss the general issues involved, but the article also presents some lessons learned from Eclipse (www.eclipse.org) to better illustrate some of the concepts.

## MOTIVATION FOR PLUG-INS

Customers' requirements control the creation and deployment of software. Customers demand more and better functionality, they want it tailored to their needs, and they want it "yesterday." Very often, large shops prefer to develop their own in-house add-ons, or tweak and replace existing functions. Nobody wants to reinvent the wheel, but rather to integrate and build on existing work, by writing only the specialized code that differenti-

# On Plug-ins
## and Extensible Architectures

ates them from their competition. Newer enterprise-class application suites consist of smaller stand-alone products that must be integrated to produce the expected higher-level functions and, at the same time, offer a consistent user experience. The ability to respond quickly to rapid changes in requirements, upgradeability, and support for integrating other vendors' components at any time all create an additional push for flexible and extensible applications.

Down in the trenches, developers must deal with complex infrastructures, tools, and code. The last thing they need is to apply more duct tape to an already complex code base, so that marketing can sell the product with a straight face. The new plug-in architectures are very attractive to developers because they can focus on providing modular functionality to users. Anyone can quickly customize applications by mixing and matching the plug-ins they need, or write new plug-ins for missing functions. The price to pay for such flexibility is managing these plug-ins. We discuss this and other issues later in the article, and you can decide if it is worth the price.

## TRADITIONAL PLUG-INS VERSUS PURE PLUG-INS
Most people are familiar with traditional plug-ins, down-loadable software bundles that extend the functionality of hosting applications such as Web browsers or text and graphical editors. They are not compiled into the application, but linked via well-defined interfaces and extension mechanisms. Most often, building a plug-in does not require access to the source code of the application. Plug-ins implement functions that the host application can recognize and activate when needed.

In the new pure plug-in architectures, *everything* is a plug-in. The role of the

hosting application is reduced to a runtime engine for running plug-ins, with no inherent end-user functionality. Without a directive hosting application, what is left is a universe of federated plug-ins, all playing by the rules of engagement defined by the framework and/or by the plug-ins themselves.
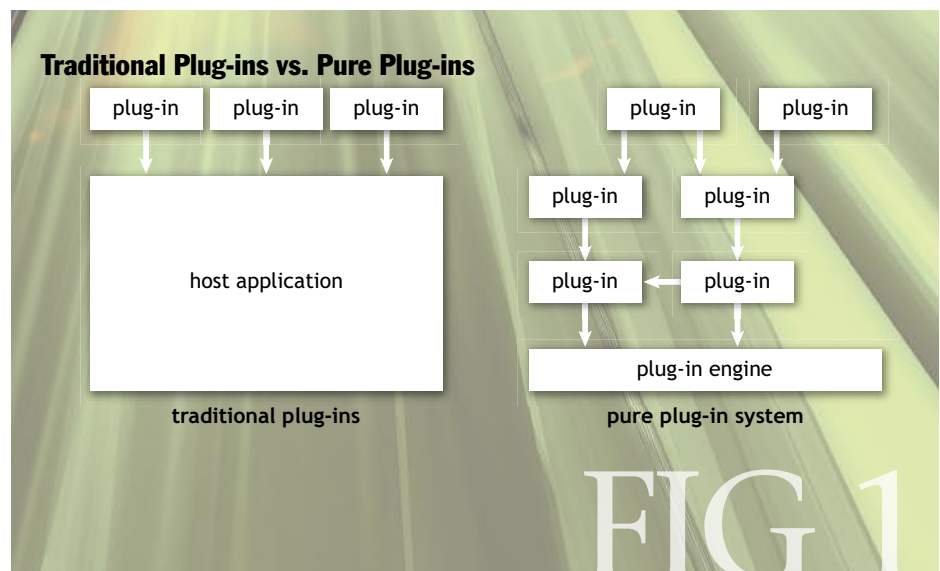
To support composing a larger system that is not pre-structured, or to extend it in ways you don't foresee, the architecture must support the extensibility of plug-ins by plug-ins. In the absence of a hosting application, plug-ins themselves become hosts to other plug-ins by providing well-defined hook points where other plug-ins can add functionality. These hook points are commonly known as *extension points*. When a plug-in contributes an implementation for an extension point, we say that it adds an extension. Much like defining any contractual obligation, an extension model provides a structured way for plug-ins to describe the ways they can be extended and for client plug-ins to describe the extensions they supply.

In a simplified way, figure 1 illustrates the main structural difference between traditional and pure plug-in architectures.

## PLUG-IN RUNTIME CONSIDERATIONS
As already mentioned, plug-ins are not stand-alone programs; their execution is orchestrated by a runtime engine. The plug-in runtime engine—let's call it the *kernel*—is the core engine that is started when a user launches an application. At a minimum, the kernel must provide runtime support for the basic plug-in infrastructure by:
• Finding, loading, and running the right plug-in code.



**Traditional Plug-ins vs. Pure Plug-ins**

traditional plug-ins — plug-in, plug-in, plug-in → host application

pure plug-in system — plug-in, plug-in, plug-in, plug-in, plug-in, plug-in → plug-in engine

FIG 1

- Maintaining a registry of installed plug-ins and the functions they provide.
- Managing the plug-in extension model and inter-plug-in dependencies.

Optionally, the kernel can supply other utility services, such as logging, tracing, or security. The user should also be able to turn off or remove parts of the application, or dynamically install new functions. For obvious reasons, it is desirable for the kernel to be small and simple, yet robust enough to build industrial-strength functionality on top of it, without many (or any) hacks.

To illustrate the basic workings of a plug-in environment, let's take a behind-the-scenes look at the Eclipse plug-in runtime structure and control flow. While mostly known as a powerful Java integrated development environment, Eclipse is actually a universal plug-in architecture for creating "anything, but nothing in particular." The runtime engine itself is implemented as a number of core plug-ins, except for a tiny bootstrap code. This code starts the core plug-ins to initialize the plug-in registry and the extension model and to resolve plug-in dependencies. Other than the core plug-ins, no other plug-in code is run at this time. All the needed plug-in metadata is read from the plug-in manifest files (plugin.xml and/or manifest.mf).

Once the initialization is complete, the runtime kernel is ready to run applications. The extension mechanism, shown in figure 2, kicks in and an entire application is built progressively, from inside out, as follows:

1. The runtime core plug-in defines an *applications* extension point so that any plug-in can declare itself an application by contributing an extension to it.

2. An *applications* extension must provide a concrete implementation for the extension point's callback interface, *IPlatformRunnable*.

3. Eclipse's default *applications* extension is contributed by the IDE plug-in, by providing a concrete implementation class that, when called by the extension point handling code, creates the workbench graphical user interface and runs the event loop until it exits.
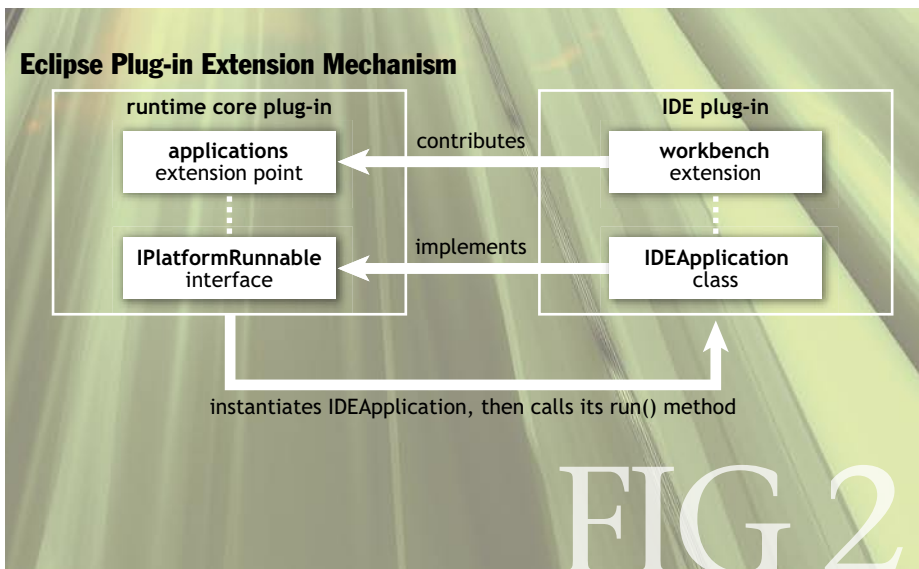
This workbench window looks like any standard application: it has menus, toolbars, views, and editors, and you can launch wizards, open preference dialogs, and so on. The basic workbench by itself, however, is just a frame for displaying visual parts and other user interface elements. Its power comes from the extension points (e.g., views, wizards, editors, action sets, help support) it defines, to which other plug-ins, or the workbench plug-in itself, can contribute extensions.

In general, the plug-in that defines an extension point is responsible for looking up its extenders (using the extension registry), instantiating the associated callback class, and invoking the appropriate interface methods. Under the covers, the runtime kernel is responsible for providing the extension registry lookup functionality, creating a class loader for the extender plug-in, loading it, and starting it. This ensures the extender plug-in is initialized and active before its classes are created and run.

Finally, it is worth noting that some extension points may require more than one callback interface and others do not require any. For example, the help system extension point *toc* is used for contributing documentation tables of contents and does not require any callbacks. When help is invoked, the help plug-in processes the *toc* extensions by creating a larger, integrated table of contents from all contributing plug-ins.

## CONFIGURING AND DISCOVERING PLUG-INS

An important consideration when deploying a plug-in system is configuring and discovering the plug-ins. Unlike monolithic applications that are typically installed in one folder on a user's machine, plug-in-based applications may have a higher degree of freedom for installation

### Eclipse Plug-in Extension Mechanism

**runtime core plug-in**
applications extension point

*contributes*

**IDE plug-in**
workbench extension

IPlatformRunnable interface

*implements*

IDEApplication class

instantiates IDEApplication, then calls its run() method

FIG 2

# On Plug-ins
## and Extensible Architectures

layout and plug-in discovery. This flexibility can also be a source of major headaches for installers and plug-in management.

Consider a multiuser shared installation where the entire product is installed by the central administrator on behalf of a community of users. The individual users do not need to perform any installation. They are simply given a local "shortcut" to invoke the shared installation (the shortcut can be in the form of a command-line invocation or a true desktop shortcut encapsulating the command-line invocation). If you're like most users, you will want to install some other cool new features, especially when the third-party plug-in offerings are plentiful (and often free). Obviously, the new plug-ins, private to that user, cannot be installed in the read-only, shared install location, so the product should allow users to install and configure extra plug-ins in a location where they have more privileges.

Configuring only a subset of plug-ins for a particular user based on access criteria, such as user roles, adds more challenges. Other configuration issues arise in scenarios where a common set of plug-ins is shared by many applications running on the same machine, or where some plug-ins are not installed locally but run from a remote location (e.g., an application server provider).

In theory, the platform must find the available plug-ins on its own and gracefully cope with missing plug-ins or those that are dynamically coming and going. In practice, many solutions for these problems involve predefined files and directory locations for key configuration files or for the plug-ins themselves. More advanced plug-in systems offer pluggable configuration and plug-in discovery mechanisms. Usually some basic bootstrap code is run to start a *configurator* plug-in, which will then discover the other plug-ins based on its own strategy. For example, Eclipse provides an Update Manager configurator plug-in that picks up plug-ins from the eclipse/plugins folder, as well as from other local plug-in folders linked from the eclipse/links folder or dynamically added when users install new plug-ins to locations of their choice. Because the configurator is pluggable, anyone can plug in another configurator, which can provision and configure the plug-ins from a remote server.

## CHALLENGING ISSUES

Pressured by time, budgets, or slick marketing claims, you may be tempted to adopt a pure plug-in architecture without being aware of its potential pitfalls. Alternatively, you might truly believe in Murphy's law and therefore won't use plug-ins because they're doomed to fail, anyway. I tend to follow those who walk the middle road: if something can fail, then it first should be understood and then fixed. The rest of the article presents issues that are likely to pose some challenges when you employ plug-ins, and that will help you ask questions or provide answers when evaluating a plug-in architecture.

**Installing and updating.** Many modern products automatically detect when they are out of date with respect to available service or product version. Either on start-up, or as a result of an explicit update action, the products compare the current installation level against some network-based baseline. The product then automatically downloads required fixes or upgrades and applies them, often as part of the product execution. Additionally, users can, and most often will, install additional plug-ins from various sources to extend the functionality provided by their current application.

For plug-in-based applications, the installation and update process can be a real nightmare: on the one hand, there are the traditional installation issues that arise in any application—ability to roll back changes, migrate existing program data and preferences, or ensure the installation is not corrupted. On the other hand, because plug-ins may originate from various providers that are not related to each other, the resulting configuration has likely never been tested. This poses a number of interesting challenges that we address in the context of the other issues discussed in the next sections.

**Security.** Systems can never be too secure, and you need to pay particular attention to securing a system based on plug-ins. Since arbitrary plug-ins can be installed—for example, by downloading them from the Web—and are allowed unlimited access to the system they plug into, security in a plug-in environment must be carefully planned. On top of this, some plug-ins require support for executing custom install code during installation, so they can have control over some parts of their installation. To prevent software security accidents or failures, the plug-in framework must address the issues of downloading from third parties and controlling a plug-in's access to other code and data. Supporting digitally signed plug-ins or secure connections helps, but it still relies on trusting the download source or the plug-in provider. Some programming environments, such as Java,

offer built-in runtime security mechanisms that can be used effectively to close some of the security gaps.

The cold reality is that, unless you are careful about what you install, it is almost impossible to be confident that the installed plug-ins are not ill-intentioned. Of course, a pure plug-in architecture also means that even a well-intentioned plug-in with serious bugs can do as much damage as an ill-intentioned one when installed.

**Concurrent plug-in version support.** Without a doubt, managing concurrent plug-in versions and dependencies is one of those problems that can keep architects, developers, and installation folks awake at night. Most of you have probably experienced "DLL hell" at some point and will look with suspicion on something that has the potential of being a "plug-in hell."
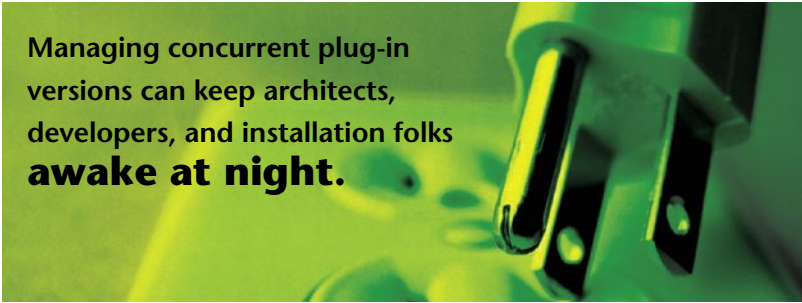
Any serious plug-in model defines some versioning scheme, but allowing multiple versions of a plug-in to be concurrently installed and executed is an issue that must be considered early on in the design, and it must be properly enforced by the install and update processes.

To illustrate the complexity of this area, let's start by considering the scenario of an application suite that integrates two stand-alone plug-in products, each installed in a separate location. It is possible to have the same plug-in, at the same version or at different versions, installed in two locations. The runtime kernel must deal with the two plug-in instances by either running them both or dropping one. When both versions are running, and the plug-ins contribute user interface elements such as menus or preference pages, the result may be a very confusing interface, with duplicate menu entries or preference pages. There is more trouble: would the user need to be exposed to this anomaly? How would you update or uninstall these plug-ins?

The difficulty of the problem is amplified by the other important role plug-in versions play here: they are part of plug-in dependency specifications. Typically, a plug-in requires the functionality provided by other plug-ins, and very often there are strict criteria about what versions are required. For example, a plug-in may require a certain level of an XML parser contributed by another plug-in, or it may need to contribute an extension to a particular version of an extension point only. Version dependency can be in the form of a fixed version number such as 3.0.1, or as a range of versions—say, 3.0.1 or newer. Properly resolving plug-in dependencies at runtime is critical to the correct functioning of the application. Managing plug-in dependency graphs is also critical to ensure a consistent configuration state after installing or updating plug-ins, as well as for properly rolling back changes.

The install/update process may need to abort installation of plug-ins that leads to an unresolved dependency, or search for and install the missing plug-ins. Either way, it is possible to end up with multiple versions of the same plug-in, so we're back to the problem of managing concurrent versions.

There is no simple, general solution to managing concurrent versions of plug-ins. Eclipse has adopted a reasonable trade-off convention for concurrent plug-in versions: only versions of plug-ins that contribute code libraries but no plug-in extensions (no user interface contributions, no documentation, and so on) are allowed to coexist in the same runtime instance. For all the other plug-ins, the latest version is usually picked up, unless a

> Managing concurrent plug-in versions can keep architects, developers, and installation folks **awake at night.**

configuration file precisely defines what to run. The main advantage of this approach is that it still allows various levels of the same code to coexist at runtime, but hidden to the end users, who will get a consistent user interface. The downside is that this will not cover all user scenarios, and that plug-ins are not treated uniformly as elsewhere in Eclipse. This special treatment of plug-ins is not just a runtime/install issue, but also something that developers and product packagers must consider.

**Scalability, up and down.** Another challenge of working with plug-in architectures is scalability. Just as the problems associated with multiple versions and dependencies can quickly escalate, so too can the sheer number of interacting plug-ins quickly become a problem. For example, when Eclipse was first designed, it was thought that a product, a large one for that matter, would consist of a few hundred plug-ins. A few releases later, some enterprise-class products built on Eclipse are known to have passed the thousand plug-in mark, so the platform goal has been revisited to support scaling between 5,000 and 10,000 plug-ins.

When designing a plug-in system for scalability, developers must consider various mechanisms that make start-up faster and have a smaller memory footprint. A general principle of runtime is that the end user should

# On Plug-ins
## and Extensible Architectures

not pay a memory or performance penalty for plug-ins that are installed but not used. A plug-in can be installed and added to the registry, but the plug-in will not be activated unless a function provided by the plug-in has been requested according to the user's activity. In general, this requires support for plug-in declarative functionality. It is often realized in practice via plug-in manifest files, so no code has to be loaded for obtaining the function contributed by plug-ins. Caching of the plug-in registry and the plug-in manifests/declarations can reduce processing during start-up in subsequent application launches, improving response time. What is gained in performance and memory footprint, however, is lost in code complexity: more code needs to be written to cache the data and to synchronize the cache with changes in the installation configuration.

Scalability problems almost always surface during plug-in install/update operations. The larger the product, the larger the download size of patches and upgrades. For product upgrades, the download time can be improved by simply downloading only the plug-ins that changed. Another installation scalability problem can arise during an interactive install/update operation. Quite often plug-in boundaries are established for development reasons (such as function reuse) and present the wrong level of granularity in terms of what the user sees as the unit of function, and therefore as an installation unit. Considering that enterprise-level applications can scale up to hundreds and thousands of plug-ins, with complex dependencies, the user can be overwhelmed by various installation choices. A possible solution is to introduce a packaging and installation component that groups a number of plug-ins to offer a higher level of function. For example, in Eclipse, Update Manager does not install plug-ins directly; it processes *features*. Features are bundles of plug-ins and are considered deployment units with install/update semantics.

The other side of scalability is scaling down, so products can run on devices with limited resources, such as cellphones and PDAs. This usually means rethinking the core framework plug-ins to refactor them into smaller plug-ins, some of them optionally deployable, so that one can run with a minimal configuration.

## OVERCOMING THE BUMPS

General discussion about plug-in architecture and design issues may not mean much for you or your company's bottom line unless you're a participant or plan to be one—either by creating a plug-in framework or developing plug-ins for fun and profit, or by using a plug-in-based application. Employing a plug-in framework is not without bumps, but they can be overcome with proper preparation.

While traditional plug-ins have been around for some time, the pure plug-in models have only recently emerged as robust, enterprise-level quality application development environments. As many major industry players are rapidly adopting plug-in technologies for their software lines, we can expect further research and development into improving the current architectures. Some of the areas that will receive increased focus will be:

- Security at all levels, including installation, update, and runtime.
- Performance: speed and low resource usage.
- Improved tools for development, testing, packaging, and deployment of plug-ins.
- Improved installation and updates, especially by merging changes without taking the system down.
- Remote management of various processes (provisioning, configuration, and so on).
- Convergence, compatibility, or interoperability of various plug-in frameworks.
- Wider range of deployment platforms (from desktop PCs and high-end servers to digital mobile phones and embedded devices).

Increasingly, much of this effort will be happening in the open source domain or standards bodies. Q

### LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

**DORIAN BIRSAN** has been working at the IBM Toronto Labs for almost 10 years, leading a range of technical projects on application development tools and writing a number of patents in the field. He has played an active role in Eclipse since its inception, leading the user assistance and the update/install teams. Mathematician at heart, turned computer scientist, and later husband and father, he holds a B.Math. in computer science and combinatorics and optimization from the University of Waterloo, as well as an M.Sc. in computer science from the University of British Columbia, where he was a National Science and Engineering Research Council of Canada Fellow.