http://help.eclipse.org/helios/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fguide%2Fintro%2Fpde\_overview.htm

The Eclipse platform is structured around the concept of **plug-ins**. Plug-ins are structured bundles of code and/or data that contribute functionality to the system. Functionality can be contributed in the form of code libraries (Java classes with public API), platform **extensions**, or even documentation. Plug-ins can define **extension points**, well-defined places where other plug-ins can add functionality.

Each subsystem in the platform is itself structured as a set of plug-ins that implement some key function. Some plug-ins add visible features to the platform using the extension model. Others supply class libraries that can be used to implement system extensions.

The Eclipse SDK includes the basic platform plus two major tools that are useful for plug-in development.  The Java development tools (JDT) implement a full featured Java development environment.  The Plug-in Developer Environment (PDE) adds specialized tools that streamline the development of plug-ins and extensions.

These tools not only serve a useful purpose, but also provide a great example of how new tools can be added to the platform by building plug-ins that extend the system.

**Workbench UI**

The workbench UI plug-in implements the workbench UI and defines a number of extension points that allow other plug-ins to contribute menu and toolbar actions, drag and drop operations, dialogs, wizards, and custom views and editors.

[Plugging into the workbench](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/workbench.htm) introduces the workbench UI extension points and API.

Additional UI plug-ins define frameworks that are generally useful for user interface development.  These frameworks were used to develop the workbench itself.  Using the frameworks not only eases the development of a plug-in's user interface, but ensures that plug-ins have a common look and feel and a consistent level of workbench integration.

The Standard Widget Toolkit (SWT) is a low-level, operating system independent toolkit that supports platform integration and portable API. It is described in [Standard Widget Toolkit](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/swt.htm).

Standard Widget Toolkit este o librarie independenta de sistemul de operare ce ofera o suita completa de elemente UI.

The JFace UI framework provides higher-level application constructs for supporting dialogs, wizards, actions, user preferences, and widget management.  The functionality in JFace is described in [Dialogs and wizards](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/dialogs.htm) and [JFace: UI framework for plug-ins](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/jface.htm).

Framework-ul JFace UI defineste un sablon standard pentru realizarea de casete de dialog, actiuni si alte elemente de UI.

## Who needs a platform?

On any given day, you can probably find an announcement about a strategic alliance, an open architecture, or a commercial API that promises to integrate all your tools, seamlessly move your data among applications, and simplify your programming life.

Down in the trenches, you're trying to apply enough import/export duct tape to let marketing say "suite" with a straight face.

Where is all this integration pressure coming from? Why is everyone trying to integrate their products into suites or build platforms to support open integration? Who needs these platforms?

### End users

Let's face it. End users do not call the support line to say, "What I really need is an open tools platform."

But they do ask why your product doesn't integrate with their other tools. They ask for features outside of the scope of your application because they can't get their data to a tool that would do the job better. They run into problems importing and exporting between different programs. They wonder why their programs have completely different user interfaces for doing similar tasks. Doesn't it seem obvious that their web site design tool should be integrated with their scripting program?

Your users want the freedom to pick the best tool for the task. They don't want to be constrained because your software only integrates with a few other programs. They have a job to do, and it's not managing the flow of files and data between their tools. They're busy solving their own problems. It's your job to make the tools work, and even better if you can make them work together.

### Software developers

Meanwhile, you are slaving on your tool implementing the next round of critical features, fixing bugs, and shipping releases. The last thing you need is another emergency import feature added to your list.

Wouldn't it be nice if you could just publish enough hooks to make integrating with your tool everyone else's problem? Unfortunately, unless you work for one of the giants, you just don't have enough clout to get away with that.

## What is Eclipse?

Eclipse is a platform that has been designed from the ground up for building integrated web and application development tooling. By design, the platform does not provide a great deal of end user functionality by itself. The value of the platform is what it encourages: rapid development of integrated features based on a **plug-in** model.

Eclipse provides a common user interface (UI) model for working with tools.  It is designed to run on multiple operating systems while providing robust integration with each underlying OS.  Plug-ins can program to the Eclipse portable APIs and run unchanged on any of the supported operating systems.

At the core of Eclipse is an architecture for dynamic discovery, loading, and running of plug-ins. The platform handles the logistics of finding and running the right code. The platform UI provides a standard user navigation model.  Each plug-in can then focus on doing a small number of tasks well. What kinds of tasks? Defining, testing, animating, publishing, compiling, debugging, diagramming...the only limit is your imagination.

### Open architecture

The Eclipse platform defines an open architecture so that each plug-in development team can focus on their area of expertise. Let the repository experts build the back ends and the usability experts build the end user tools. If the platform is designed well, significant new features and levels of integration can be added without impact to other tools.

The Eclipse platform uses the model of a common workbench to integrate the tools from the end user's point of view. Tools that you develop can plug into the workbench using well defined hooks called **extension points**.

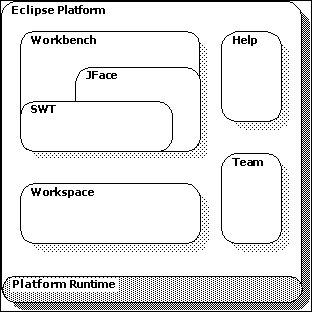
The platform itself is built in layers of plug-ins, each one defining extensions to the extension points of lower-level plug-ins, and in turn defining their own extension points for further customization. This extension model allows plug-in developers to add a variety of functionality to the basic tooling platform. The artifacts for each tool, such as files and other data, are coordinated by a common platform resource model.

The platform gives the users a common way to work with the tools, and provides integrated management o f the resources they create with plug-ins.

Plug-in developers also gain from this architecture.  The platform manages the complexity of different runtime environments, such as different operating systems or workgroup server environments.  Plug-in developers can focus on their specific task instead of worrying about these integration issues.

### Platform structure

The Eclipse platform itself is structured as subsystems which are implemented in one or more plug-ins.  The subsystems are built on top of a small runtime engine. The figure below depicts a simplified view.



The plug-ins that make up a subsystem define extension points for adding behavior to the platform.  The following table describes the major runtime components of the platform that are implemented as one or more plug-ins.

## latform SDK roadmap

### Runtime core

The platform runtime core implements the runtime engine that starts the platform base and dynamically discovers and runs plug-ins. A **plug-in** is a structured component that describes itself to the system using an OSGi manifest (**MANIFEST.MF**) file and a plug-in manifest (**plugin.xml**) file. The platform maintains a registry of installed plug-ins and the functionality they provide.

A general goal of the runtime is that the end user should 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.

The platform runtime is implemented using the OSGi services model. While implementation details of the runtime may not be important to many application developers, those already familiar with OSGi will recognize that an Eclipse plug-in is, in effect, an OSGi bundle.

The best way to get a feel for the runtime system is to build a plug-in. See [Plug it in: Hello World meets the workbench](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/firstplugin.htm) to get started building a plug-in. To understand the nuts and bolts of the runtime system, see [Runtime overview](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/runtime.htm).

### Resource management

The resource management plug-in defines a common resource model for managing the artifacts of tool plug-ins. Plug-ins can create and modify **projects**, **folders**, and **files** for organizing and storing development artifacts on disk.

[Resources overview](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/resInt.htm) provides an overview of the resource management system.

### Workbench UI

The workbench UI plug-in implements the workbench UI and defines a number of extension points that allow other plug-ins to contribute menu and toolbar actions, drag and drop operations, dialogs, wizards, and custom views and editors.

[Plugging into the workbench](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/workbench.htm) introduces the workbench UI extension points and API.

Additional UI plug-ins define frameworks that are generally useful for user interface development.  These frameworks were used to develop the workbench itself.  Using the frameworks not only eases the development of a plug-in's user interface, but ensures that plug-ins have a common look and feel and a consistent level of workbench integration.

The Standard Widget Toolkit (SWT) is a low-level, operating system independent toolkit that supports platform integration and portable API. It is described in [Standard Widget Toolkit](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/swt.htm).

The JFace UI framework provides higher-level application constructs for supporting dialogs, wizards, actions, user preferences, and widget management.  The functionality in JFace is described in [Dialogs and wizards](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/dialogs.htm) and [JFace: UI framework for plug-ins](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/jface.htm).

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

## The runtime plug-in model

The platform runtime engine is started when a user starts an application developed with Eclipse. The runtime implements the basic **plug-in** model and infrastructure used by the platform. It keeps track of all installed plug-ins and the functionality that they provide.

A plug-in is a structured component that contributes code (or documentation or both) to the system and describes it in a structured way. Plug-ins can define **extension points**, well-defined function points that can be extended by other plug-ins. When a plug-in contributes an implementation for an extension point, we say that it adds an **extension** to the platform. These extensions and extension points are declared in the plug-ins's manifest (**plugin.xml**) file.

Using a common 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. Defining an extension point is much like defining any other API. The only difference is that the extension point is declared using XML instead of a code signature. Likewise, a client plug-in uses XML to describe its specific extension to the system.

A general goal of the runtime is that the end user should not pay a memory or performance penalty for plug-ins that are installed, but not used. The declarative nature of the platform extension model allows the runtime engine to determine what extension points and extensions are supplied by a plug-in without ever running it. Thus, many plug-ins can be installed, but none will be activated until a function provided by a plug-in has been requested according to the user's activity. This is an important feature in providing a scalable, robust platform.

The plug-in **org.eclipse.ui.workbench** defines most of the public interfaces that make up the workbench API. These interfaces can be found in the package [**org.eclipse.ui**](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/package-summary.html) and its sub packages. Many of these interfaces have default implementation classes that you can extend to provide simple modifications to the system. In our hello world example, we will extend a workbench view to provide a label that says hello.

The interface of interest is [**IViewPart**](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/IViewPart.html), which defines the methods that must be implemented to contribute a view to the workbench. The class [**ViewPart**](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/part/ViewPart.html) provides a default implementation of this interface. In a nutshell, a view part is responsible for creating the widgets needed to show the view.

The standard views in the workbench often display some information about an object that the user has selected or is navigating. Views update their contents based on actions that occur in the workbench. In our case, we are just saying hello, so our view implementation will be quite simple.

Before jumping into the code, we need to make sure our environment is set up for plug-in development...

### Plug-ins and bundles

The mechanics for supporting plug-ins are implemented using the [OSGi](http://www.osgi.org) framework. From this standpoint, a plug-in is the same thing as an OSGi **bundle**. The bundle and its associated classes specify and implement the process for Java class-loading, prequisite management, and the bundle's life-cycle. For the rest of this discussion, we use the terms **plug-in** and **bundle** interchangeably, unless discussing a particular class in the framework.

#### Plugin

The [**Plugin**](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/runtime/Plugin.html) class represents a plug-in that is running in the platform. It is a convenient place to centralize the life-cycle aspects and overall semantics of a plug-in. A plug-in can implement specialized functionality for the **start** and **stop** aspects of its life-cycle. Each life-cycle method includes a reference to a [**BundleContext**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleContext.html) which can supply additional information.

The **start** portion of the life-cycle is worth particular discussion. We've seen already that information about a plug-in can be obtained from the plug-in's manifest file without ever running any of the plug-in's code. Typically, some user action in the workbench causes a chain of events that requires the starting of a plug-in. From an implementation point of view, a plug-in is never started until a class contained in the plug-in needs to be loaded.

The **start** method has been a convenient place to implement initialization and registration behavior for a plug-in. However, it is important to realize that your plug-in can be started in many different circumstances. Something as simple as obtaining an icon to decorate an object can cause one of your plug-in's classes to be loaded, thus starting your plug-in. Over-eager initialization can cause your plug-in's code and data to be loaded long before it is necessary. Therefore, it's important to look closely at your plug-in's initialization tasks and consider alternatives to performing initialization at start-up.

* **Registration** activities such as registering listeners or starting background threads are appropriate during plug-in start-up if they can be performed quickly. However, it is advisable to trigger these actions as part of accessing the plug-in's data if the registration activities have side-effects such as initializing large data structures or performing unrelated operations.
* **Initialization** of data is best done lazily, when the data is first accessed, rather than automatically in the start-up code. This ensures that large data structures are not built until they are truly necessary.

#### Bundle Context

Life-cycle management is where the OSGi "bundle" terminology and the platform's "plug-in" terminology meet. When your plug-in is started, it is given a reference to a [**BundleContext**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleContext.html) from which it can obtain information related to the plug-in. The [**BundleContext**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleContext.html) can also be used to find out about other bundles/plug-ins in the system.

**BundleContext.getBundles()** can be used to obtain an array of all bundles in the system. Listeners for **BundleEvent** can be registered so that your plug-in is aware when another bundle has a change in its life-cycle status. See the javadoc for [**BundleContext**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleContext.html) and [**BundleEvent**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleEvent.html) for more information.

*Prior to 3.0, a plug-in registry (****IPluginRegistry****) was provided to supply similar information. For example, it could be queried for the plug-in descriptors of all plug-ins in the system. This registry is now deprecated and* [***BundleContext***](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleContext.html) *should be used for this purpose. The platform registry is now used exclusively for information about extensions and extension points.*

#### Bundle Activator

The [**BundleActivator**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleActivator.html) interface defines the start and stop behavior implemented in [**Plugin**](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/runtime/Plugin.html). Although the [**Plugin**](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/runtime/Plugin.html) class is a convenient place to implement this function, a plug-in developer has complete freedom to implement the interface for [**BundleActivator**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleActivator.html) in any class appropriate for the plug-in's design. In fact, your plug-in need not implement this interface at all if it does not have specific life-cycle management needs.

#### Bundles

Underneath every plug-in lies an OSGi bundle managed by the framework. The [**Bundle**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/Bundle.html) is the OSGi unit of modularity. Fundamentally, a bundle is just a collection of files (resources and code) installed in the platform. Each bundle has its own Java class loader, and includes protocol for starting, stopping, and uninstalling itself. From the Eclipse platform point of view, [**Bundle**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/Bundle.html) is merely an implementation class. Plug-in developers do not extend the bundle class, but use [**Plugin**](http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/runtime/Plugin.html) or other [**BundleActivator**](http://www.osgi.org/javadoc/r4v42/org/osgi/framework/BundleActivator.html) implementations to represent the plug-in.

# PDE – Concepte

* PDE Overview

# Plug-in Development Environment Overview

The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.

PDE also provides comprehensive OSGi tooling, which makes it an ideal environment for component programming, not just Eclipse plug-in development.

PDE is broken into three main components:

* [UI](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm#ui) - A rich set of models, tools and editors to develop plug-ins and OSGi bundles
* [API Tooling](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm#api) - Tooling to assist API documentation and maintenance
* [Build](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm#build) - Ant based tools and scripts to automate build processes

Discover the latest features in the [What's New](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/whatsNew/pde_whatsnew.htm) section.

## PDE UI

PDE UI provides a editors, wizards, launchers, views and other tools to create a full featured environment to develop and deploy Eclipse plug-ins, fragments, features, update sites, RCP products and OSGi bundles.

Some of the PDE tools include:

* **Form-Based Manifest Editors**: Multi-page editors that centrally manage all manifest files of a plug-in or feature.
* **RCP Tools**: Wizards and a form-based editor that allow you to define, brand, test and export products to multiple platforms.
* **New Project Creation Wizards**: Create a new plug-in, fragment, feature, feature patch and update sites.
* **Import Wizards**: Import plug-ins and features from the file system.
* **Export Wizards**: Wizards that build, package and export plug-ins, fragments and products with a single click.
* **Launchers**: Test and debug Eclipse applications and OSGi bundles.
* **Views**: PDE provides views that help plug-in developers inspect different aspects of their development environment.
* **Miscellaneous Tools**: Wizards to externalize and clean up manifest files.
* **Conversion Tools**: Wizard to convert a plain Java project or plain JARs into a plug-in project.
* **Integration with JDT**: Plug-in manifest files participate in Java search and refactoring.

# Extensions and Extension Points

A basic rule for building modular software systems is to avoid tight coupling between components. If components are tightly integrated, it becomes difficult to assemble the pieces into different configurations or to replace a component with a different implementation without causing a ripple of changes across the system.

Loose coupling in Eclipse is achieved partially through the mechanism of **extensions** and **extension points**. The simplest metaphor for describing extensions and extension points is electrical outlets. The outlet, or socket, is the extension point; the plug, or light bulb that connects to it, the extension. As with electric outlets, extension points come in a wide variety of shapes and sizes, and only the extensions that are designed for that particular extension point will fit.

When a plug-in wants to allow other plug-ins to extend or customize portions of its functionality, it will declare an extension point. The extension point declares a contract, typically a combination of XML markup and Java interfaces, that extensions must conform to. Plug-ins that want to connect to that extension point must implement that contract in their extension. The key attribute is that the plug-in being extended knows nothing about the plug-in that is connecting to it beyond the scope of that extension point contract. This allows plug-ins built by different individuals or companies to interact seamlessly, even without their knowing much about one another.

The Eclipse Platform has many applications of the extension and extension point concept. Some extensions are entirely *declarative*; that is, they contribute no code at all. For example, one extension point provides customized key bindings, and another defines custom file annotations, called *markers*; neither of these extension points requires any code on behalf of the extension.

Another category of extension points is for overriding the default behavior of a component. For example, the Java development tools include a code formatter but also supply an extension point for third-party code formatters to be plugged in. The resources plug-in has an extension point that allows certain plug-ins to replace the implementation of basic file operations, such as moving and deletion.

Yet another category of extension points is used to group related elements in the user interface. For example, extension points for providing views, editors, and wizards to the UI allow the base UI plug-in to group common features, such as putting all import wizards into a single dialog, and to define a consistent way of presenting UI contributions from a wide variety of other plug-ins.

# Plug-in

A **plug-in** is used to group your code into a modular, extendable and sharable unit.

Plug-ins are modular as each plug-in contains some portion of code. The plug-in specifies other plug-ins (or java packages) it requires to be available to run and it also specifies the set of java packages it provides. An Eclipse based program or [product](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/concepts/product.htm) will contain multiple plug-ins, which can be added, replaced or removed to alter the functionality of the program.

Plug-ins are extendable using [extensions and extension points](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/concepts/extension.htm). A plug-in can provide one or more extension points so other plug-ins can add to the functionality of the plug-in. A plug-in may also provide extensions to connect to other plug-ins.

Plug-ins are sharable. A plug-in can be exported as a directory or as a jar which can be added to other applications. Plug-ins can be grouped into [features](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/concepts/feature.htm) which can be distributed and installed into applications.

Eclipse plug-ins are based on OSGi bundles. OSGi is used to manage the plug-ins in an Eclipse application. A plug-in must contain a manifest file with valid OSGi headers for plug-in name and version. Extensions and extension points functionality added by Eclipse in addition to OSGi. To use extensions you must provide a plugin.xml file. PDE provides a full featured [project](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/guide/tools/project_wizards/new_plugin_project.htm) and [editor](http://help.eclipse.org/helios/topic/org.eclipse.pde.doc.user/guide/tools/editors/manifest_editor/editor.htm) for creating and editing these files.