Eclipse RCP: A Platform for Building Platforms

by Wayne Beaton
August 29, 2006

The Eclipse RCP is composed of numerous components. Each of these components contributes some part of the overall functionality of the environment. In fact, almost all of the Eclipse RCP is made of components; every part of RCP, with the exception of a small amount of bootstrap code is a component. Components are more commonly known as "plug-ins" in Eclipse circles (or "bundles" in OSGi terminology). The name "plug-in" implies that the functionality contained within the component is somehow second class, or an add-on to "built-in" functionality. This is not the case; Eclipse RCP treats all plug-ins as equals and there is no explicit notion of "built-in" vs. "custom". The plug-ins that you create to contribute your application's behaviour are run alongside those that make up Eclipse RCP.

A natural way to start the development of a rich client application is to begin with a single plug-in. Within a single plug-in, you can define the entire user interface, business logic, and object model for your application. Building an application in this manner provides exposes the developer to a lot of useful infrastructure. More than just a collection of windowing components, the RCP application provides the framework for managing your application's user interface including higher level notions of editors and views (with the ability to reposition and stack), tool bars, menues, selection management, and must more. However, building an application in this manner only scratches the surface of the power that Eclipse enables. This is a fine first foray into Eclipse application building; the natural next step is to start building real components.

One of the more powerful features of the Eclipse platform is plug-ins. In fact, the Eclipse SDK is itself a collection of plug-ins; there is very little in Eclipse that is not a plug-in (just a relatively small amount of bootstrap code). This makes Eclipse very extensible. New functionality can be added to the SDK by creating one or more plug-ins that are dynamically discovered and installed by the framework. Plug-ins are discovered by virtue of being present; there is no manifest that needs to be updated when you add a plug-in.

Like the Eclipse SDK, RCP applications are an assembly of plug-ins. Plug-ins come in many different shapes and sizes. You can build an entire Eclipse RCP application using a single plug-in added to the collection of plug-ins that implement RCP itself. Or you can build an RCP application as a collection of many plug-ins that each implement some piece of the application.

There are many benefits to building your RCP application as a collection of many plug-ins, including:

Lazy-loading
: Eclipse will load a plug-in when it is required. If you partition your application into multiple plug-ins you can decrease your application's startup time and improve its use of memory by leveraging this feature. At start up time, only the initially required subset of the plug-ins will load rather than your entire application; this will reduce the amount of time and memory required at startup.

Updates
: Eclipse can update individual plug-ins when an update is required. If your application is partitioned into multiple plug-ins, only those plug-ins that actually require updating can be downloaded and installed. This will decrease the amount of time, and resources required to update your application.

Extensibility
: The very act of partitioning an application into multiple plug-ins encourages the creation of extension points which can later be leveraged in ways that you may not currently be considering.

Reuse
: By making your application a collection of components, you provide an opportunity to reuse those components in other applications without needing to make code changes.

Better design

Making multiple components encourages better design. When you partition your application, you're forced to think about important issues like how you will define interfaces between the components and how you're going to facilitate conversation between them. Unfortunately there's no guarantee that you'll end up with a great design, but it's certainly encouraged.

An obvious way to partition your application is to separate your domain-specific business logic and object models into one plug-in and your user interface into another. By doing so, you are effectively layering your application along the lines recommended by the model-view-controller pattern. That is, your application's business logic -- contained within its own plug-in -- is unconcerned with the specifics of the user interface (ideally, it is void of any notion of any particular user interface technology or language). A second plug-in provides the user interface code (view and controller layers) and is void of any implementation of business logic. The "user interface" plug-in has the "business logic" plug-in as a dependency and makes use of the models defined in it. This style of architecture is very similar to what you might find in a Java EE application: a web module (WAR file) contains servlets as controllers and JSPs as views; other JAR files contain the business logic and object model.

The Eclipse component model provides a great service in this style of architecture. By imposing a visibility model, Eclipse actively discourages the developer from including user interface code in the model (of course, real programmers can figure out ways around this). The collection of "user interface" plug-ins all have the "business logic" plug-in (and others) as dependencies: the views can see the model, but not the other way around. By encouraging this style of separation, the level of coupling in the code is reduced which makes the code far less brittle in the face of change and infinitely more reusable.

When your application code is partitioned into multiple plug-ins, creating an RCP application is a process of assembling those plug-ins into a coherent whole. Defining the set of plug-ins required by application is one of the roles of a product configuration. A product configuration is also the place where branding information, including the location of the splash screen, window icons, about image and text, and more.

Building Platforms

One of the greater strengths of Eclipse is its ability to dynamically discover and load components. This grants the developer considerable power to build very extensible applications. Taken to the extreme, you can build your RCP applications as foundations with your own open application programming interfaces (API).

Consider an RCP application that manages a "To do" list. The core functionality of this application provides an object representation of a "To do task" and the ability visualize a list of tasks. The visual representation of the list is shown in Figure 1.

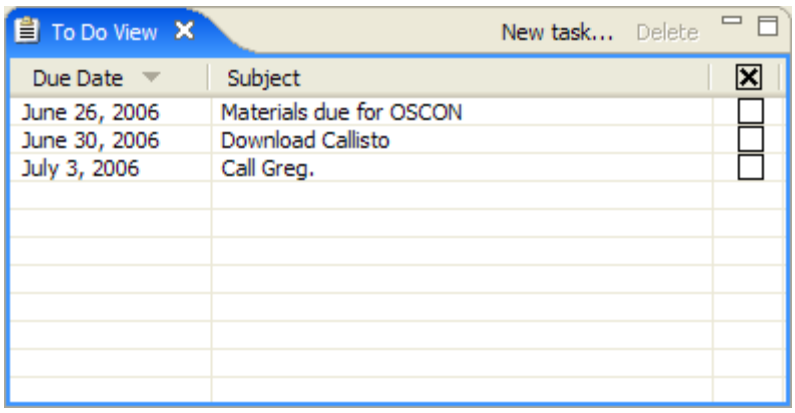

Figure 1 - The "To do" view

By itself, this application is moderately interesting. But by opening up the application to extension, significant power is granted to a broader community. It would be relatively easy to build this application with a hardcoded mechanism for storing the task information. But rather than hardcoding, you might consider opening the application to allow other developers to hook their own storage mechanism into your code through an extension point. Your initial implementation can leverage the extension point to provide the ability to store information in a local database. Another developer might extend your application to store information on a remote server using a web service or some other mechanism.

Conclusion

Eclipse RCP is a powerful framework for building rich client applications and more. An initial look at RCP reveals tight integration into the host platform with a native look and feel, window management, customizability featuring stackable editors and views, and more. But this only scratches the surface. At the heart of RCP lies an OSGi-compliant component model which opens a world of power and flexibility. Components are dynamically discovered and loaded as required; they can be updated and extended.

An evolutionary approach to adopting Eclipse RCP is natural. Your initial foray into Eclipse RCP will likely be focus on the user interface aspects. As you become more familiar with the component model, the partitioning of application code into multiple plug-ins follows. Ultimately, with the development of domain-specific platforms based on multiple extensible plug-ins, the real power of Eclipse RCP is unlocked.