ISSN: 2321-7782 (Online)
Volume 1, Issue 7, December 2013
International Journal of Advance Research in Computer Science and Management Studies
Research Paper
Available online at: www.ijarcsms.com

Plug in Technology with its Design Consideration: A New Way to Build Rich Internet Application

Madhulika Arora
PhD Scholar
Suresh Gyan Vihar University
Jaipur (Raj) - India

Abstract: Everybody knows that Html applications are lean and fast, but when it comes to complicated features they are unable to fulfill the expectation of the end user. For this reason everybody wants Rich Clients, which provide comfortable user interfaces. The open source development platform Eclipse really provides such as Rich Client architecture and supports the development process with a couple of components. Here it is very easy for the developer that he can focus his attention on creating the application logic. A Rich Client application consists of plug-ins, which are software components that extend the platform. Even the Eclipse IDE itself is a plug-in. It also provides the feature that new functionality can be integrated by extending other plug-ins extension-points.

Keywords: plug-in technology; eclipse; rich client platform; rich client application; design considerations.

I. Introduction

The Rich Client Platform provides a generalized model to assemble an application from components, which are provided by different software vendors. These vendors can build components by implementing the plug-in contract, whose primary purpose is to specify, which requirements a plug-in must fulfill to extend another plug-in, and wrapping their software into plug-ins. This paper will describe how the Eclipse plug-in model works, how single plug-ins can interact and how existing software patterns contributed to the development of this mechanism, to describe the plug-in model in this paper.

II. Plug-in Concept

To provide certain functionality, a plug-in is a computer program that interacts with another program. Typical examples are plug-ins for media players, like video or audio codec, or for graphical applications, like filters. The main program defines the interface and the plug-in contributes by implementing this interface. Generally spoken, there are two main differences in plug-in concepts: the form of plug-in execution (direct or indirect) and the plug-in identity. On the one hand, the direct approach is to provide enough information about existing plug-ins and by this way other plug-ins can execute this plug-ins directly.

This alternative reduces the overall amount of interfaces in an application, because a single direct execution replaces two execution calls that would be needed in an indirect plug-in execution architecture, where a service broker is responsible for executing plug-ins. The direct plug-in execution reduces the amount of data that has to be saved in the service broker and improves the performance of the application. The indirect plug-in execution architecture provides a service broker that does error checking when failures occur during the plug-in execution. So the service broker promotes reliability. In addition, it hides changes to the application programming interface (API) to make plug-ins more portable. For the reason that there is a single point of contact, this architecture even supports the modification of the application.

Plug-in discovery and registration is also a major concern of a plug-in architecture. A service broker can discover the identity of plug-ins instead of providing an interface where plug-ins can register their identities when they are added or removed. Self-registering of the plug-ins improves the performance during the service execution because the service broker knows about existing plug-ins. The disadvantage is that it requires the plug-in developer to add code to register itself and if the platform application programming interfaces changes, he will have to modify the code. Looking up plug-ins at runtime is less error-prone and thus makes the application more modifiable and reliable. Furthermore, the code required to look up plug-ins does not increase by adding new plug-ins. With the introduction of OSGi, Eclipse supports the direct plug-in execution on top of a service registry. Therefore plug-ins does not have to register them but are discovered during start-up. From this point on they can be invoked by other plug-ins.

A. Mechanism

It is very clear in the figure, that after registering the plug-ins on the host application it can provide services. Plug-ins does not usually work by them. Host application is responsible for the services which are provided by the plug-ins. End-users allowed to add and update plug-ins dynamically; this is made possible by making the plug-ins independent of host application. By this way it is possible make changes to the host application without any interference.

Sometimes third parties need some plug-ins to interact with the host application to provide a standard interface to open application programming interfaces. Third-party plug-ins can continuously work in proper way if there is a change in the original version if it is using a stable application programming interface. The life-cycle of obsolete applications can be extended here.

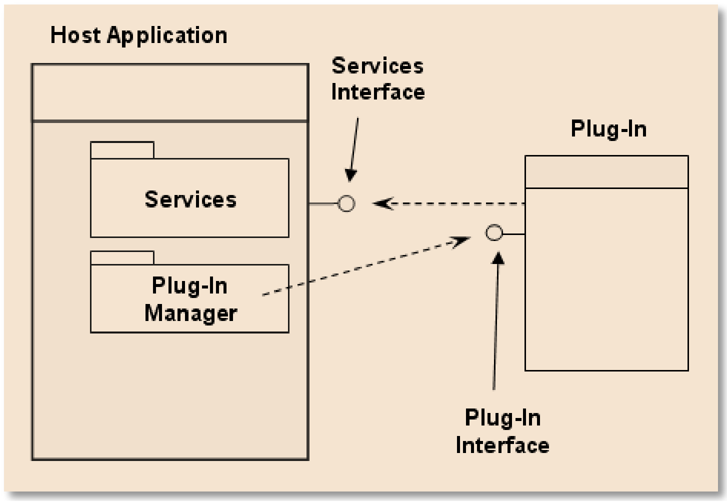

Figure 1: Host Application and Plug-In interface

B. Traditional Plug-Ins Versus Pure Plug-Ins

In the traditional plug-ins, it uses the downloadable software bundles so that it can provide the enhanced functionality of the hosting applications. This functionality is achieved by linked to extension mechanisms with well-defined interfaces although they are not compiled into the particular application. Mostly it is not necessary to access to the source code of the application at the time of building the plug-in. Only those functions can be implemented via plug-in, which are known to the host application. These functions are made activated when needed. Most people are familiar with that there is everything is a plug-in if we talk about new pure plug-in architectures.

At the time of composing a larger system which is not restructuring, it is necessary that architecture should be the way that it supports the extensibility in regard of plug-ins by plug-ins. If there is no host application then some plug-ins provide the functionality of hosts, by providing the guidance to other plug-ins in well-defined popper points that at which points other plug-in can provide the additional functionality. The points which are talking about are the extension points. Whenever a plug-in implements the extension point, means it adds an extension. To explain the extensions which are provided by the plug-ins to client plug-ins, there are extension model defined. Extension models also describes that in which ways, they can be extended.

C. *The Eclipse Plug-Ins Model*

A Plug-in can be seen component that extends the platform with certain services and functionalities. It can seamlessly be integrated into the platform at deployment time. At runtime, the component is wrapped into a plug-in runtime class. This class is responsible for the management and configuration of the plug-in. The main purpose of this class is to perform some special tasks during the activation or deactivation of the plug-in.

At deployment time, all required resources of a plug-in must be copied to the plugins directory of the Rich Client application. These resources consist of the manifest file, the jar file with the assembled class files and other resources like images, documentation, etc. The plug-in is activated when other plug-ins request one of its functionalities. A user interaction usually results in the instantiation and initialization of such a plug-in.

"Our knowledge for solving software engineering problems in the right manner is more and more being put in a nutshell in tools which provide a good interface to do so. These tools provide the greatest of benefits when they operate in surroundings that can provide complete integration with existing elements such as editors, compilers, debuggers, profilers and visualizes. A major challenge is to develop some tools which can enhance the way of developing projects, which can help in changing the future environments".

D. *The Manifest File*

The Manifest file documents the XML elements and attributes that define plug-ins. The manifest file provides the information about the unique id org.eclipse.ui.intro of this plug-in, its provider name, its name its version number, and its main class org.eclipse.ui.intro.internal.IntroPlugin. The "%" signs in the name and provider-name attribute are a reference to a properties file that is located in the same folder. The runtime element defines the name of the target library as well as the resources that should be exported to this library.

The very small Eclipse core has the task to load and execute the plug-ins, which are located in the plugins folder. The Eclipse core itself is a plug-in and contains the library runtime.jar and a plugin.xml file. The runtime.jar library belongs together with the plug-in org.eclipse.platform with the Java archive startup.jar and the plug-in org.eclipse.core.boot with the Java archive boot.jar to the absolute minimum requirement of an Eclipse based application. Moreover, boot.jar is responsible to finish the installation process of a newly installed platform. Another basic requirement is the plug-inorg.apache.xerces with its Xerces-DOM implementation. The DOM is required to be intelligent to interpret the manifest files.

E. *Dependency and Extension*

As far as Eclipse model is concerned, a plug-in may have the relation with another plug-in. This relation can be of two types:

1. **Extension:** There are two main types of plug-ins in this relationship. These are extender plug-in and host plug-in. The functionality of a host plug-in is extended by the extender plug-in.
2. **Dependency:** There are also two main types of plug-ins in this relationship. These are prerequisite plug-in and dependent plug-in. The functionality of a dependent plug-in is supported by prerequisite plug-in.

All these kind of relationships are declaratively specified at manifest files of the plug-in with using the XML extension and elements which are required for this purpose. In addition a reachable plug-in may as well stay deactivated when its functionality is not used during the lifetime of the running instance.

Dependency

For the reason that dependency statements are compile and runtime directives, the statements in source lead to the loading of prerequisite plug-ins at runtime and to the augmentation of the class path at compile time.

Extension

To make the functionalities of a plug-in available to the user, a plug-in has to be intelligent to contribute menu items, toolbar items, etc. to the user interface. Due to the specification of Eclipse, this has to be possible without recompiling the application or altering the user interface classes, in this case org.eclipse.ui, where the menu is setup. So the org.eclipse.ui plug-in has to be augmented. As it does not know anything about the plug-ins which requires it to be altered, it has to define a set of configuration and behavioral requirements. This set can be thought of slots, also called extension-points in the Eclipse terminology. So in this way one plug-in can define a couple of extension points, for example to allow its menu and its toolbar items to be altered. Furthermore, one extension-point allows one or more extension plug-ins to be plugged into it. These extension points allow extending the host plug-in by adding processing elements. In the role of the host plug-in, the Intro plug-in, offers an extension point to add Introduction related configurations to its presentation layer and the Intro plug-in, in the role of the extender plug-in, adds a standard configuration and implements the actions which should be executed when invoking this configuration. In this case, the extension is purely descriptive. It tells the plug-in the text, which it should display when showing this item and the link that it should follow when clicking on it. A single extension-point can be extended one or more times by a single extension (e.g.: various tutorial items of the JDT) and one or more times by a couple of extensions for example JDT, PDE, etc. Callback objects, through which the extender and host plug-ins communicate, are not always required because the extension model allows providing the necessary classes, which are needed by the host plug-in to change its behavior, at compile-time. The extension can be purely descriptive.

As already mentioned, not all extensions lead to a callback in the extender plug-in. Some extensions may be used to set flags in the host plug-in or to describe a certain behavior. The configExtension of the JDT, for example, only tells the host plug-in about the URL that it should follow when the user clicks on a certain link. So the extension does not provide a callback object, but it causes an object, in this case the link on the introduction page, to come into existence.

The creation of new top-level workbench menu items, to the configExtension extension-point, the action Sets allows at the point the extension. For the reason that processing a click on a top-level menu only resides in exposing the lower-level menu entries, this action does not need further processing from the extender plug-in and therefore no custom callback object. When designing an application that is based on the plug-in architecture of Eclipse, one achieves great flexibility when it comes to exposing parts of the application. The application can be parameterized through XML tags and therefore be extended with complex extension structures.

III. THE ARCHITECTURE OF RICH CLIENT

An RCP application can be seen as three layers-

1. Presentation layer
2. Business layer
3. Data layer

1. Presentation layer- The presentation layer more often holds presentation logic components and beside this, it has User Interface.
2. Business layer- The business layer holds business entity components with the dealing workflow. It also consists of business logic.
3. Data layer- The data layer holds basically service manager components. It also facilitates the user with data access. Rich client applications mainly have a presentation layer. The main work of this presentation layer is to contact a remote business layer which is hosted on server machines from beginning to end services. For example, an application which is used for data entry can send all of the data to the server where it is processed and stored.

Some applications may be difficult that only communicates with other services and stores the data to consume or send back information. These applications perform most of the processing themselves. For example spreadsheet software like Microsoft Excel that stores both the data and state locally, beside this it performs complex local tasks itself, and only communicates with remote servers at the time when it needs to fetch and update linked data. The guidelines for the data and business layers in such applications do not different from other applications; it is just like general applications. These types of rich clients can have their own data access layers and business layers.

IV. DESIGN CONSIDERATION FOR DEVELOPMENT WITH RICH CLIENT APPLICATION

At the time of designing of rich client application, the main objective of the software designer is to design the formation in such a way so that it could reduce complication. This can be done easily by separating tasks into different areas of concern and by choosing an appropriate technology for his application. As these both works have a great impact on the application, it should be done carefully. The design should fulfill all the requirements for the application in terms of security, performance, ease of maintenance and reusability. These all things are essential for a robust application.

The following guiding principle should keep in mind when designing rich client applications-

- Appropriate technology should be chosen on the basis of application requirements-
  Suitable technologies for designing the rich client application are XAML, Browser Applications, Windows Forms, WPF, and OBA etc.
- Presentation logic should be separated from interface implementation-
  The use of separate components like presentation logic, interface implementation within the application makes the application robust as this can reduce dependencies, promote reusability, and make maintenance and testing easier. That’s why some design patterns like Supervising Presenter and Presentation Model that separate user interface logic from user interface rendering provides ease of maintenance, improves testability and promotes reusability.
- Presentation flows and presentation tasks should be identified-
  This will help us to design each step and each screen with the help of multi-screen or Wizard process.
- Designing should be in the manner so that it could provide a usable and suitable interface-
  The features of application such as navigation, layout, localization and choice of controls should have maximum usability and accessibility.
- There should be separate concerns for all layers-
  There should be separate components for each work like components for data accessing code should be separately located in a data layer, extract business rules and other tasks which are not related to presentation should be located into a separate business layer.
- Presentation logic should be in a way that it can be reused-
  There are a lot of components that can be reused in other application such as helper classes, generalized client-side validation functions and libraries that contain templates can be reusable in several applications.
- Round trips should be avoided when accessing remote layers-
  Whenever remote layers are accessed, they freeze or block the user interface so coarse grained methods should be used or executed asynchronously if possible.
- Client should be loosely couple from any remote services it uses-
  For communicating with services which are located on separate physical tiers, a message-based interface should be used.
- Tight coupling should be avoided for the objects which are located in other layers-
  Abstraction should be used which is provided by abstract base classes, or messaging when communicating with other layers of the application or common interface definitions.

V. CONCLUSION

With the collaboration of Eclipse, Rich Client Platform can simplify a lot of tasks, especially because of the powerful framework. Also the management of preferences and the handling of user actions are quite intuitive. At the beginning of a project a lot of time has been spent to get familiar with these complex frameworks. However, especially for large or long-living applications this pays off heavily in the long run. Eclipse platform is a plug-in management kernel of Eclipse framework. This plug-in is present in every Eclipse deployment. This is core firm plug-ins. The identities are hard-coded in core plug-ins and necessarily activated in each running instance of Eclipse. If we talk about Non-core plug-ins, it is only activated by other plug-ins when required. One of the best features of this plug-in architecture is its extensibility. If the application is developed using plug-in technology then new features can be dynamically extended to it. Beside this, features of the application can be developed in parallel by different teams, since these features can be implemented as separate components. With lots of ease of development this technology also provides clear development direction. Developers have a clear roadmap for development since the plugin framework ideally provides a well-defined interface and documentation. And finally a plugin typically has one function, and so developers have a single focus which is necessary for worthy development.

References

1. Khemavast, T.,Muthorst, D., Kinderreich, A., Anwar, T. Architecting a plug-in based TSM client application. 5th Malaysian Conference in Software Engineering (Mysec), Johor Bahru, Malaysia), December 2011 pages 101-106.
2. Storey M.-A., Michaud J., Mindel M. Improving the Usability of Eclipse for Novice Programmers. Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange, 35-39.
3. DavidSaff, Michael D. Ernst. Continuous Testing in Eclipse. Electronic Notes in Theoretical Computer Science Volume 107, 12 December 2004, Pages 103–117.
4. S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. IEEE Transactions on Software Engineering, 27(5):449–464, May 2001.
5. William G. Griswold, Coping with Crosscutting Software Changes Using Information Transparency, Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, p.250-265, September 25-28, 2001.
6. Gregor Kiczales , Erik Hilsdale , Jim Hugunin , Mik Kersten , Jeffrey Palm , William Griswold, Getting started with ASPECTJ, Communications of the ACM, v.44 n.10, p.59-65, Oct. 2001.
7. W. Y. Leong. Using the atlas metaphor to assist cross-cutting software changes. Masters Thesis, March 2002.

AUTHOR(S) PROFILE

Ms Madhulika Arora, is Assistant Professor at Centre for E-Governance, Jaipur. She received MCA degree in 2006 and presently she is pursuing Ph.D. from Gyan Vihar University, Jaipur. Her areas of interest are Programming Concept and Programming Languages like C, C++, and Java. Her papers had been published in various International and National Journals, Conferences.