Eclipse IDE Design Review

Neil Barooah

The development of Eclipse dates back to November 2001. It's initial codebase originated from IBM VisualAge and it was developed as a generic Integrated Development Environment (IDE). Eclipse provides a component-based platform that serves as a foundation for building tools for developers. At the time of its development, a number of powerful commercial development environments were available such as Microsoft Visual Studio, and Java-based IDEs such as Symantec's Visual Café, Borland's JBuilder, and others. Eclipse sought to provide an open source platform for creation of interoperable tools for application developers. This would allow developers to focus on writing new tools, instead of writing to code to deal with infrastructural issues like connecting to source code repositories, interacting with file system etc.

One of the primary goals of developing Eclipse was to focus on Java development tooling as key to enable growth in the open community. It aimed to address problems raised by users that dealt with the cohesiveness of IBM software tooling. IBM wanted to establish a common platform for all their development products to avoid duplicating common elements of infrastructure. It was aimed at enhancing integration between multiple tools built by IBM, so users can have a smooth experience. The most important aspect, perhaps, of the Eclipse design is the plug-in model. Each of the plug-ins is developed to perform a small number of tasks such as defining, testing, animating, publishing, compiling, debugging, and much more. These plug-ins are JAR files with a manifest that describes itself, its dependencies, and its functions. Even though the demographics of Eclipse has evolved as it becomes a more widely accepted tool, it remains an important productivity tool for programmers. According to a survey done by the Eclipse Foundation, 53.8% of Eclipse users identify their position as Programmers, while the next largest group is Systems Architect. Over half the respondents of the survey identified themselves as working in the high-tech or profession services/consulting industry.

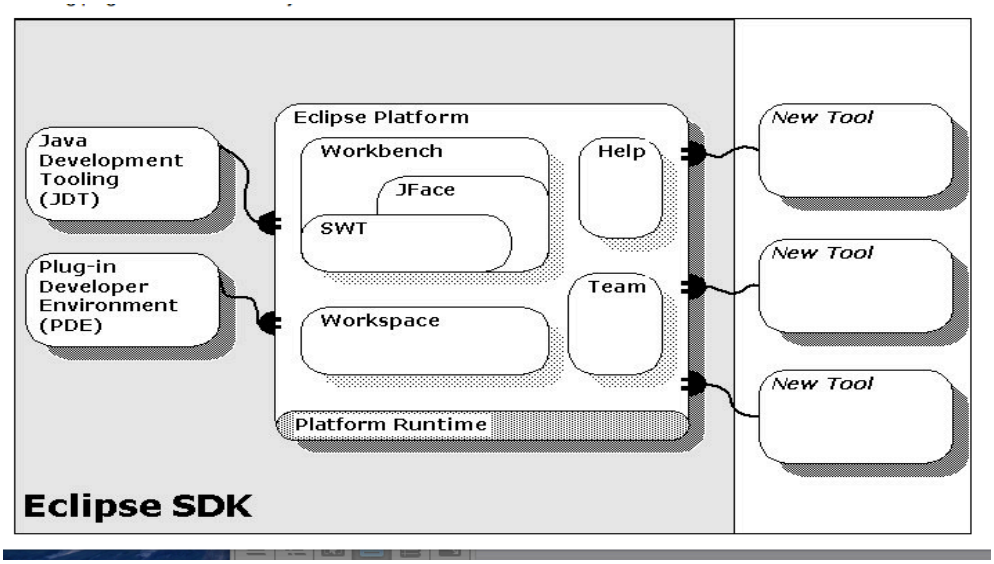

Figure 1: Architecture of Eclipse

The Eclipse platform is built around the idea of plug-ins. Plug-ins are the smallest unit in the Eclipse platform, and are structured bundles of code that add functionality. They supply code libraries and platform extensions. To allow people to build upon the platform, eclipse uses extensions and extension points. The 3 major elements of the Eclipse SDK architecture are the Platform, the JDT (Java Development Tools) and PDE (Plug-in Development Environment). The JDT provides plugins to extend Eclipse, while the PDE provides tooling for developing plugins to extend Eclipse, which are written in Java and sometimes contains non-code elements such as HTML files for online documentation. The architectural style of Eclipse is illustrated in Figure 1. As shown in the figure, the Eclipse platform provides a workbench to users. It consists of perspectives, views and editors. The workbench is built on the Standard Widget Toolkit (SWT) and Jface. The SWT in Eclipse is aimed at providing a native experience on the desktop. They use operating system calls to build UI components. JFace is a layer on top of SWT that supplies tools for typical UI tasks, such as frameworks for preferences and wizards. The Eclipse platform further contains an integrated help system based upon small units of information called topics as shown in Figure 1. These topics consist of a label and a reference to its location, which is an HTML documentation file, or an XML document describing additional links. Finally, the Team element in Figure 1 represents team support that allows common tasks such as interacting with source code repository and creating patches

The Java Development Tools provide Java editors, wizards, refactoring support, debugger, compiler and an incremental builder. Eclipse has implemented it's own compiler in order to allow building tools on top of the compiler. The incremental builder, which is supported by its compiler, recompiles files that have changed and hence provides high performance. A builder within Eclipse takes inputs within the workspace and creates output files. So, the builder knows what types of files exist in the workspace, and how they are referenced to each other. When the incremental build is invoked, the builder describes any new, modified or deleted files and deleted source files have their corresponding class files deleted. The Plug-in Development Environment in Eclipse, on the other hand, provides the tools to develop, build, deploy and test plugins and other artifacts that are used to extend the functionality of Eclipse. It contains a PDE Build, which examines the dependencies of the plugins and generates Ant scripts to construct build artifacts.

Eclipse implements a number of interesting object-oriented design patterns. One of the simplest is the Singleton used in its platform. Singletons are patterns such that there is always only one instance of the target object, and there is usually global point of access to it. In Eclipse, some examples of Singleton patterns are methods to getting a Workbench, getting a Workspace and an Adapter Manager. Although a singleton saves time and resources since we can share a single instance of a class with many threads, it does have some visible disadvantages. The use of static for singletons lead to classloading issues and these may behave in an unpredictable fashion in a dynamic OSGi (Open Services Gateway initiative) environment. Having singletons also increase coupling as singletons are coupled

with clients. A solution to this problem is a Bridge pattern, which is meant to decouple an abstraction from its implementation. It uses encapsulation, aggregation, and can use inheritance to separate responsibilities. Another design pattern widely seen in the Eclipse Platform is Adapter. Adapters convert the interface of a class into another interface clients expect. Adapters let classes work together that cannot otherwise because of incompatible interfaces. In Eclipse, a lot of classes and interfaces implement IAdaptable, which is used to support extensions. An example is the Eclipse Properties view, which presents a set of properties of the selected view. It requires an interface to fetch the properties, and then display them. On the other hand, JFace uses the Adapter pattern in IContentProvider to get at the contents of a domain object and ILabelProvider for the text or image to display as illustrated in Figure 2. Even though we could extend another class instead of using an adapter, using adapters is cleaner as we need not extend too many interfaces or change objects that other objects are dependent on.

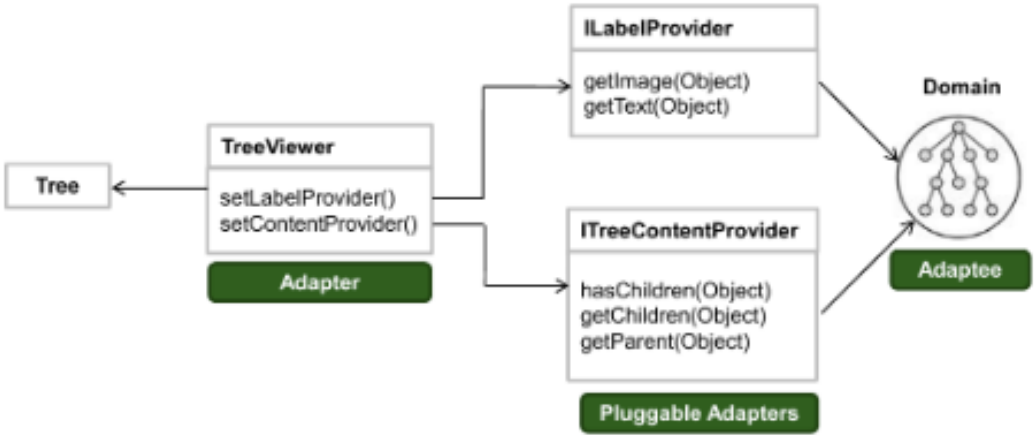

Figure 2: Illustration of Adapter in JFace.

The Workspace has two design patterns in work – Proxy and Bridge. A proxy design pattern provides a placeholder for another object to control access to it, while a bridge design pattern decouples an abstraction from its implementation so that the two can vary independently. Let us try to understand these patterns through IResource, as shown in figure 3. In a Workspace, each resource is represented by a handle, which acts like a key for a resource. Handles are small objects, and never change once created. These handles define the behavior of a resource, but don't keep any resource state information.

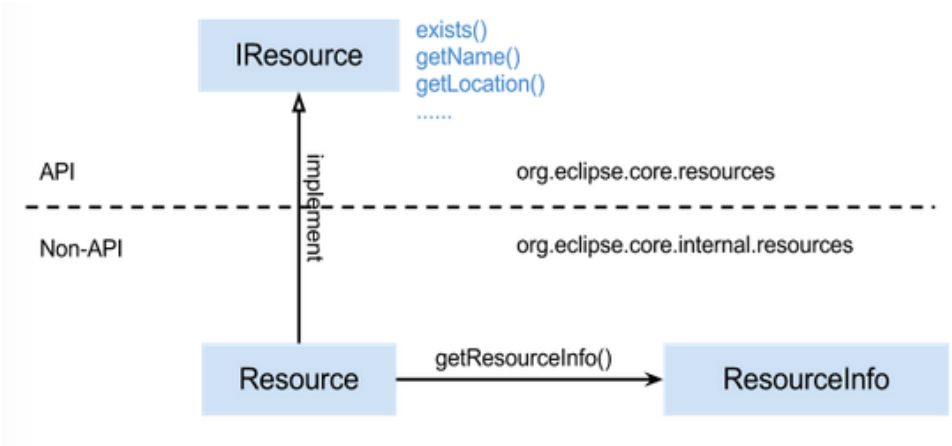

Figure 3: Illustration of Proxy and Bridge in IResource

In case of IResource as shown in Figure 3, Resource acts as the proxy for ResourceInfo and all requests sent to ResourceInfo is handled through its proxy Resource. The Resource acts as the implementer for IResouce. Since there is only one implementer for a handle, this is a Bridge. Another design pattern seen in Workspace is the Composite pattern, which composes objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. In the Eclipse Workspace, IWorkspace is the root interface and it is a composite of IContainers and IFiles, as illustrated by Figure 4.

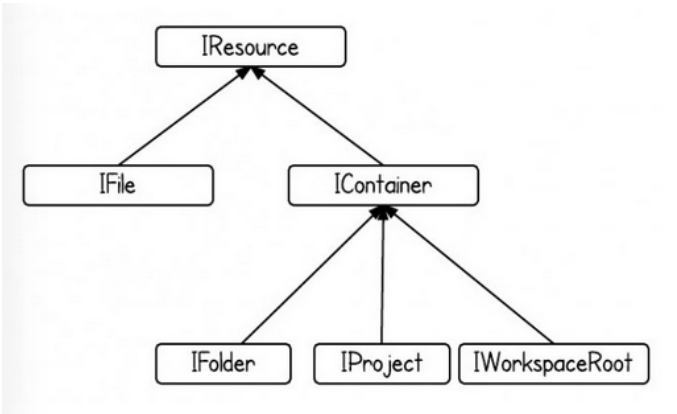

Figure 4: Illustration of Composite pattern in IResource

Apart from composite, the SWT implements the strategy design pattern. A strategy defines a family of algorithms, encapsulates each one, and makes them interchangeable. In SWT, a RowLayout is created to setup Shell's layout, which is an example of a strategy (Figure 5).

```
RowLayout rowLayOut = new RowLayout()
Shell shell = new Shell();
shell.setLayout(rowLayOut);
```

Figure 5: Illustration of Strategy in SWT

Similarly, JFace implements the strategy design pattern when filtering an sorting of elements in views. Another design pattern dominant in the Eclipse design is command, which encapsulates a request as an object, thereby parameterizing clients with different requests, queue or log requests, and support for undoable operations. In JFace, the run() method in IAction encapsulates the code to be executed when it is called by the Invoker. It has contains more than just one function as it stores additional data and can be used to attach multiple menus, tool bars and buttons. This is illustrated in Figure 6.

```
IAction action = new Action("Exit") {
    @Override
    public void run() {
        System.exit(0);
    }
};

action.setDescription("Exits the VM");
menu.add(action);
```

Figure 6: Illustration of Command in JFace

Just like with good implementation of SOLID principles in Eclipse as discussed in previous sections of this paper, there is ample evidence of GRASP in Eclipse. In particular, the use of Open Closed principle stands out, which says that software entities like classes, modules and functions should be open for extension but closed for modifications. An obvious example is the Eclipse Extension Point method, which defines extension points where other plug-ins can later add functionality. Another prominent software design that Eclipse implements is Dependency Injection, which is how certain objects can access other objects from the outside. For instance, in Eclipse an implementation of a view that requires a parent composite, an input object or a service, such as a logger. Dependency injection deals with the problem of how to retrieve certain objects while implementing something.

Given the wide use of SOLID and GRASP principles in Eclipse that greatly enhances functionality, the growing popularity of Eclipse as a primary software development tool in the software industry is no surprise. There is clear evidence of good use of singletons and adapters in Eclipse Platform, composite, bridge and proxy in Workspace and Resource, strategy and composite in SWT, and strategy and command in JFace. These are SOLID design principles that gives allows us to describe Eclipse as superior quality software. However, as discussed earlier, the use of singleton leads to higher coupling which is a violation of GRASP principles. This can be solved effectively with the use of bridge and proxy.

Sources

1. Brown, Amy, and Greg Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. Mountain View, CA: CreativeCommons, 2011. Print.

2. "Eclipse Documentation." *Eclipse Documentation*. N.p., n.d. Web. 22 Apr. 2016.

3. "Design Patterns Used in Eclipse Platform." *Design Patterns Used in Eclipse Platform*. N.p., n.d. Web. 22 Apr. 2016.