Crafting Application Public Interface (API) and especially managing API among different releases is difficult. Learn how to take advantage of Eclipse’s PDE API Tools to make this process easier and seamlessly integrated into your daily development. Note that this article is specific to Eclipse V3.4: Ganymede.

Before we get into detail about the Application Public Interface (API) Tools within the Eclipse Plug-in Development Environment (PDE), let’s talk a little bit about what it means to be API in Eclipse.

What is API?

Ever get the following warnings or errors in Eclipse and wonder what they mean?

Figure 1. Discourage access

<table>
<thead>
<tr>
<th>Description</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>A method getControl() from the type PartPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>A method getControl() from the type PartPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>A method getControl() from the type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>A method getControl() from the type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>A method getControl() from the type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>A method getControl() from the type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>The type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>The type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>The type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>The type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>The type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
<tr>
<td>The type PartContentPane is not accessible due to restriction: ActiveHelpSection.java.</td>
<td>ActiveHelpSection.java</td>
</tr>
</tbody>
</table>

The most probable reason for those warnings is that you’re accessing code that isn’t meant to be publicly accessed using some form of API. API elements, in general, are well documented and have a specification of some type. On the other hand, non-API elements are considered internal implementation details and usually ship without published documentation. The access to these internal elements is what Eclipse was notifying you about in the figure above. Eclipse was trying
to politely warn you that you are accessing code that may change and isn't officially supported. So then, what exactly is API?

Since Eclipse is based on the Java™ programming language, we have four types of API elements. Let’s look at each.

**API package**
A package that contains at least one API class or API interface.

<table>
<thead>
<tr>
<th>Table 1. Package naming conventions in the Eclipse platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naming Convention</td>
</tr>
<tr>
<td>org.eclipse.xyz.*</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>org.eclipse.xyz.internal.*</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>org.eclipse.xyz.internal.provisional.*</td>
</tr>
</tbody>
</table>

**API class or interface**
A public class or interface in an API package, or a public or protected class or interface member declared in or inherited by some other API class or interface.

**API method**
A public or protected method or constructor either declared in or inherited by an API class or interface.

**API field**
A public or protected field either declared in or inherited by an API class or interface.

Now that we are aware of the different kinds of API elements, let’s discuss API Tools and how it can manage these API elements for you.

**What is API Tools?**
The goal of API Tools is to help you maintain good APIs. API Tools accomplishes this by reporting API defects, such as binary incompatibilities, incorrect plug-in version numbers, missing or incorrect @since tags and usage of non-API code between plug-ins. Specifically, it's designed to:

- Identify binary compatibility issues between two versions of a software component or product.
- Update version numbers for plug-ins based on the Eclipse versioning scheme.
- Update @since tags for newly added classes, interfaces, and methods.
- Provide new javadoc tags and code assist to annotate types with special restrictions.
- Leverage existing information (in MANIFEST.MF) to define the visibility of packages between bundles.
- Identify usage of non-API code between plug-ins.
- Identity leakage of non-API types into API.

**Adding API Tools**
To use API Tools within your projects, you need to do two things: set an API baseline and add the API Tools nature to the projects of interest.
Setting an API baseline

To know whether you are breaking API, you need to set some type of baseline for compatibility analysis. In API Tools, this is called an API Baseline and may be set via the API Baselines preference page (see Figure 2). Setting an API Baseline is as simple as pointing to an existing Eclipse-based installation. API Tools will generate a baseline for you on the fly when it scans for plug-ins. Once you have a baseline set, you need to have your Eclipse projects take advantage of API Tools. Note, this process can also be done in a headless manner as part of your build system, but this is outside the scope of this article, and I recommend checking out the API Tools wiki for more information (see Related topics).

Figure 2. Adding an API baseline

Adding the API Tools project nature

To see any errors or warnings associated with API Tools, your projects need to have the API Analysis nature and builder added to them. This can be accomplished in two ways and depends whether you’re applying API Tools to existing projects. If you’re working with existing projects, the recommended approach is to use the API Tooling Setup wizard (see Figure 3). The wizard can be accessed by right-clicking your project and selecting PDE Tools > API Tooling Setup. In the wizard, simply click the projects you want converted to use API Tools and click Finish. That’s it.
The other approach to take advantage of API Tools is at the time of plug-in project creation. In Eclipse V3.4, the **New Plug-in Project** wizard was enhanced with an additional checkbox, *Enable API Analysis*, to add the API Analysis nature to your project.
Using API Tools

Now that we know how to set up projects to use API Tools, let's look at some examples on how API Tools can help us. API Tools has a set of annotations you can use on your various API elements to enforce restrictions.

Table 2. API restrictions

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Valid API elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@noimplement</td>
<td>Interfaces</td>
<td>Indicates that clients must not implement this interface. Any class using the implements or extends keyword for the associated interface will be flagged with a problem.</td>
</tr>
<tr>
<td>@noextend</td>
<td>Classes</td>
<td>Indicates that clients must not extend this class. Any class using the extends keyword for the associated class will be flagged with a problem.</td>
</tr>
<tr>
<td>@noinstantiate</td>
<td>Classes</td>
<td>Indicates that clients must not instantiate this class. Any code</td>
</tr>
</tbody>
</table>
that instantiates the associated class with any constructor will be flagged with a problem.

@nooverride
Methods
Indicates that clients must not redefine this method. Any subclass that defines a method that overrides the associated method will be flagged with a problem.

@noreference
Methods, constructors and fields (nonfinal)
Indicates that clients must not reference this method, constructor, or nonfinal field. Any code that directly invokes the associated method or constructor or references the associated nonfinal field will be flagged with a problem.

Now that you have an idea of the available API restriction annotations, let’s look at some examples of how this would work in the real world.

### API restriction examples

Let’s start with a really simple example, like a plug-in with an API that allows you to produce widgets.

#### Listing 1. IWidget.java

```java
package org.eclipse.example.widgetfactory;

/**
 * A simple widget
 * @noimplement
 */
public interface IWidget {
    public String getName();
    public long getId();
}
```

In this example, a widget simply has a name and an identifier. We annotated the interface with a restriction to tell clients not to implement this interface as we want our clients to implement the abstract widget listed below.
Listing 2. AbstractWidget.java

package org.eclipse.example.widgetfactory;

/**
 * An abstract widget
 */
public abstract class AbstractWidget implements IWidget {
  /**
   * @nooverride
   */
  public long getId() {
    return Math.round(Math.random());
  }
}

The abstract widget we have has a method that is annotated with an API restriction telling clients not to override this method. Let's create a default widget people can use.

Listing 3. DefaultWidget.java

package org.eclipse.example.widgetfactory;

/**
 * A default widget, this class isn't meant to be extended.
 * Implementation is provided as-is.
 */
public class DefaultWidget extends AbstractWidget {
  public String getName() {
    return "The default widget";
  }
}

The default widget we have provided for our clients is simply as-is. It's meant to be used, but not extended. We are free to extend this class within our own plug-in and create other default widgets people could use. So, let's pretend we are a client of this widget API and see what happens. Let's create two classes: MyWidget (implements IWidget) and MyDefaultWidget (extends DefaultWidget). As a client of the widget API, what would I see? For the MyWidget class, we will see a warning that we are implementing an API interface that isn't meant to be implemented by clients (see Figure 5). For the MyDefaultWidget class, we will see warnings about illegally extending the DefaultWidget class and also illegally overriding the getId() method.
This simple example should give you a clear idea of how clients of your API will be presented with usage information about your API.

**Binary incompatibilities**

A difficult problem when declaring API is to know when you actually break API from version to version of your software. For example, an easy way to break API is to change the method signature of a previously defined API method. To illustrate, I'll do that in the Eclipse code base with the popular PluginRegistry class.
In this case, I modified an existing API method, `findEntry(String id)`, and API Tools is smart enough to realize that this would break clients of this API class. There are many other ways to break API, and API Tools provides checks for all of these. As a bonus, all these binary-incompatible changes are configurable using the API Tools preferences page.
Versioning

Another difficult problem that surfaces when dealing with API is version management. There are two sides to version management in my opinion: One is coming up with a versioning strategy and the other is enforcing it. API Tools follows the Eclipse Versioning Guidelines (see Related topics), which are heavily based on the OSGi versioning scheme. To keep things simple, version numbers are composed of four segments — three integers and a string, respectively named major.minor.service.qualifier.

Each version segment captures a different intent. The major segment indicates breakage in the API, the minor segment indicates externally visible changes like new API, the service segment indicates bug fixes and the change of development stream, and the qualifier segment indicates a particular build.

The difficult and error-prone part has always been remembering when to change the version number of your plug-in due to whatever changes you made (like adding a new API method or fixing a bug). A common problem in the past was that developers used to simply bump the minor
version when anything changed instead of doing it when new API changed — for example, when the 3.4 stream opened for the Eclipse SDK, it was common to just bump the minor version, assuming changes would be made in the future to actually cause a need to modify the minor version. However, with API Tools, version management isn't error-prone anymore because API Tools is able to enforce and tell you what the proper version should be of your plug-in based on an API Baseline. For example, imagine I added a method to an API class of a plug-in that had a version of 3.3.0, what would API Tools show you?

**Figure 9. Version errors**

API Tools presents us with two errors, indicating that we should update the \@since tag for the method to clearly state that the method is part of the 3.4 API. The other error indicates that the plug-in version be bumped in the MANIFEST.MF file. If we look at this specific error in detail, we see that API Tools even kindly offers a proper quick fix to update the version number to the correct version.
Conclusion

This article provided a brief introduction to API Tools and showcased some of its features. It's important to note that this article scratches the surface of what API Tools can do for your projects. For example, you can use API Tools in a headless environment and also have API Tools generate reports for you that catch API violations. I hope you grasp the importance of API and versioning and start using API Tools in your projects.
Related topics

- Learn more about the The Eclipse Plug-in Development Environment (PDE).
- Learn more about the PDE API Tools at the Eclipse Foundation.
- Read "Eclipse platform API rules of engagement" for additional history and background on Eclipse API.
- See "Eclipse Platform API Specification" for a list of Eclipse APIs.
- Check out the Eclipse Ganymede release train page at Eclipse.org.
- Learn more about how Eclipse versions its plug-ins at the Eclipse version-numbering page.
- Interested about OSGi? Check out the OSGi Alliance Web site.
- Interested in what's happening inside the Eclipse community? Check out PlanetEclipse.
- Check out the available Eclipse plug-ins at Eclipse Plug-in Central.
- Check out EclipseLive for webinars featuring various Eclipse technologies.
- Want to meet Eclipse committers and learn more about Eclipse projects? Attend EclipseCon, Eclipse's premiere conference.
- Check out the latest Eclipse technology downloads at IBM alphaWorks.
- Check out the "Recommended Eclipse reading list."
- Download Eclipse Platform and other projects from the Eclipse Foundation.
- Browse all the Eclipse content on developerWorks.
- New to Eclipse? Read the developerWorks article "Get started with Eclipse Platform" to learn its origin and architecture, and how to extend Eclipse with plug-ins.
- Expand your Eclipse skills by checking out IBM developerWorks' Eclipse project resources.

© Copyright IBM Corporation 2008
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)