IBM

Building templates with the Eclipse Plug-in Development Environment

Create templates to save your users time and trouble

Chris Aniszczyk, Software Engineer, IBM, Software Group

Summary: This article introduces the creation of templates in Eclipse so you can enhance the experience of your clients. We will develop a simple template as an example of the flexibility PDE's templating system provides.

Date: 06 Feb 2007
Level: Intermediate
Activity: 1618 views
Comments: 0 (Add comments)

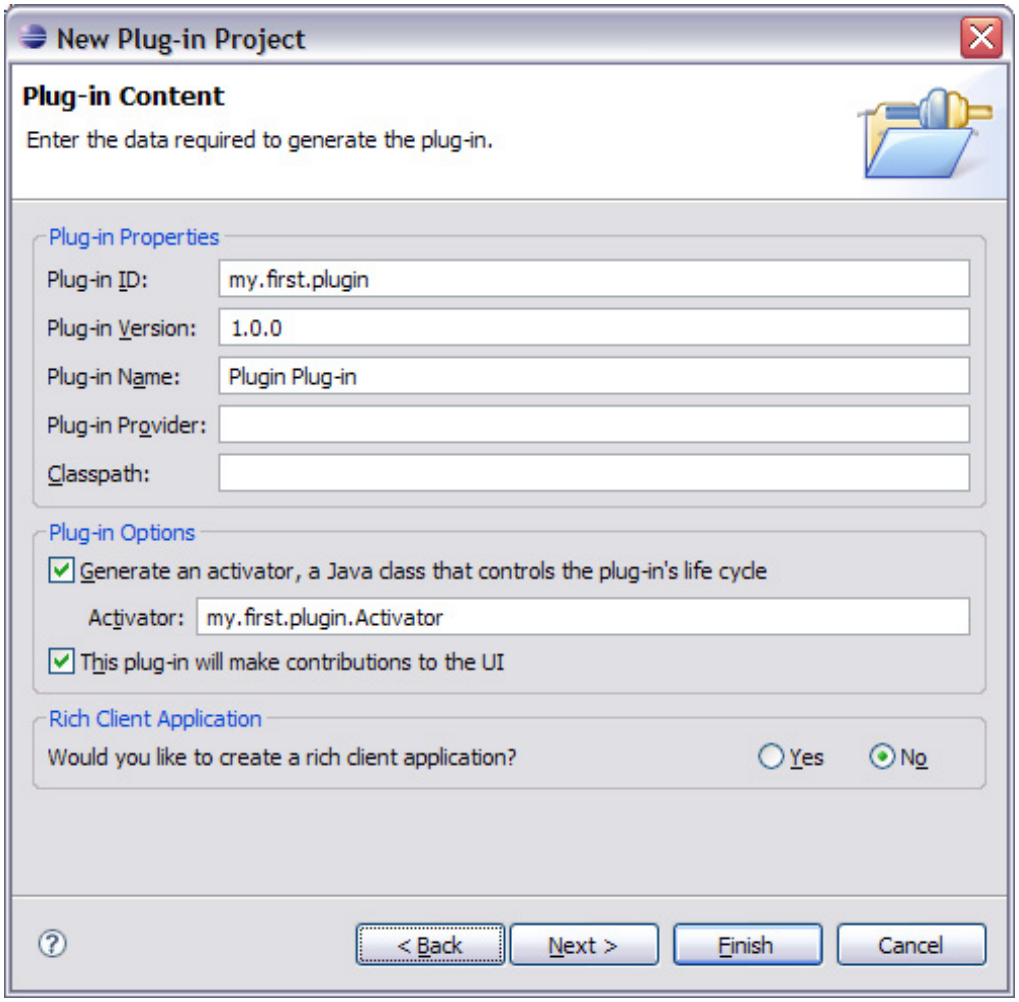
Average rating

Background

Eclipse has been very successful since its inception, and a lot of that success is due to the various projects that make up the Eclipse platform. One of these projects is called the Plug-in Development Environment (PDE). If you have ever created a plug-in in Eclipse before, you've worked with PDE. PDE is a set of tools to help create, package, and manage plug-ins.

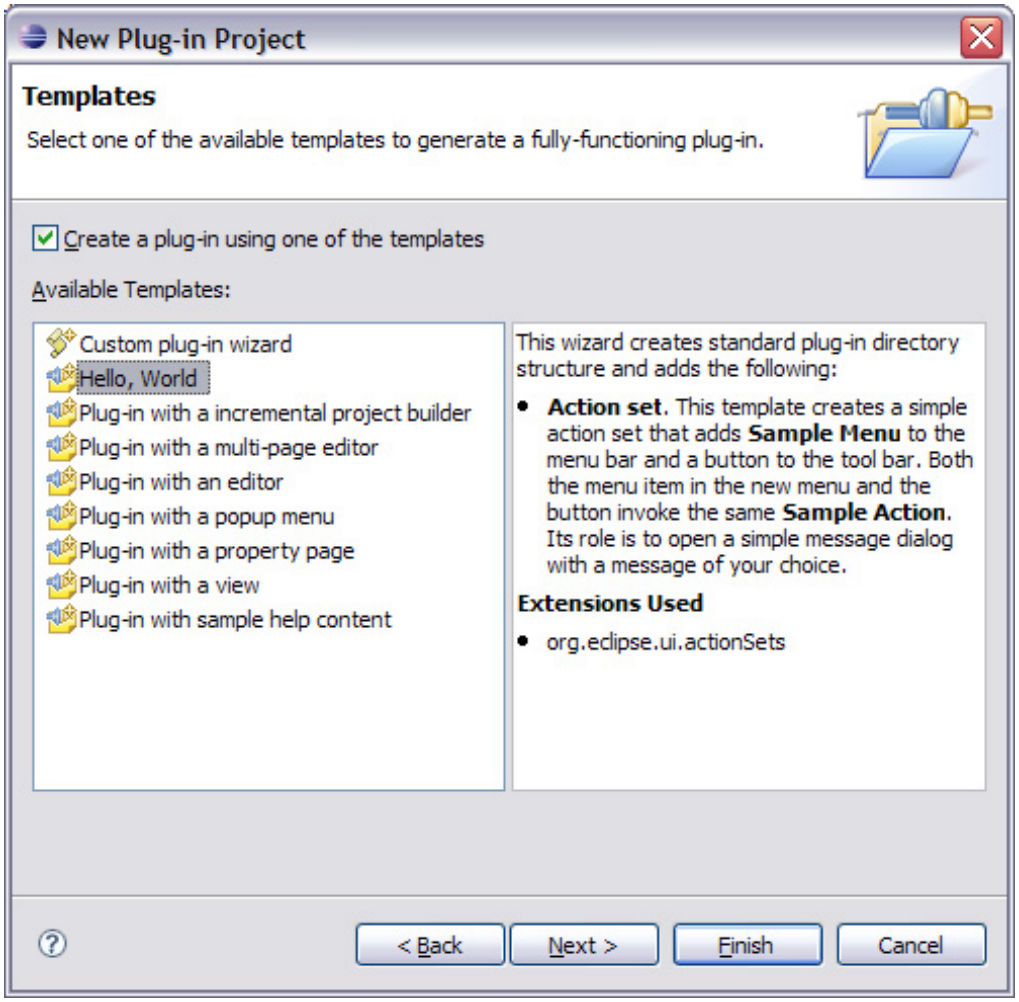
We will focus on the templating functionality offered in PDE. If you remember the day you tried to create your first plug-in in Eclipse, you were making your way through the New Plug-in Project wizard.

Figure 1. PDE plug-in wizard



Then, on your way to creating your first plug-in, you stumbled on gold.

Figure 2. PDE plug-in wizard templates



Oh, my -- templates! PDE offers a variety of templates to start your Eclipse plug-in creation adventure. The purpose of this article is to discuss how to create these templates so you can reduce the learning curve your end users face when using your extension points or code.

PDE Tidbits

PDE comprises of two main parts: UI and Build. The UI component is responsible for all the wizards and editors -- and more! -- you see while developing plug-ins. It also contains the templating infrastructure we're discussing in this article. The Build component is responsible for the building and packaging of plug-ins.

Creating templates

Our goal

The way I have found I learn things best is through a nice simple example. Well, guess what? That's exactly what we're going to do today. We're going to develop a

simple template that creates a simplistic view. It is my hope this lays the groundwork for future template creation endeavors.

Setting up your plug-in

The first leg of our journey is to create a new plug-in project (**File > New > Project > Plug-in Project**). Take advantage of templates. Make sure your project has a dependency on org.eclipse.pde.ui. Once this is done, we can go to the Extensions tab of the plug-in editor and begin creating our template.

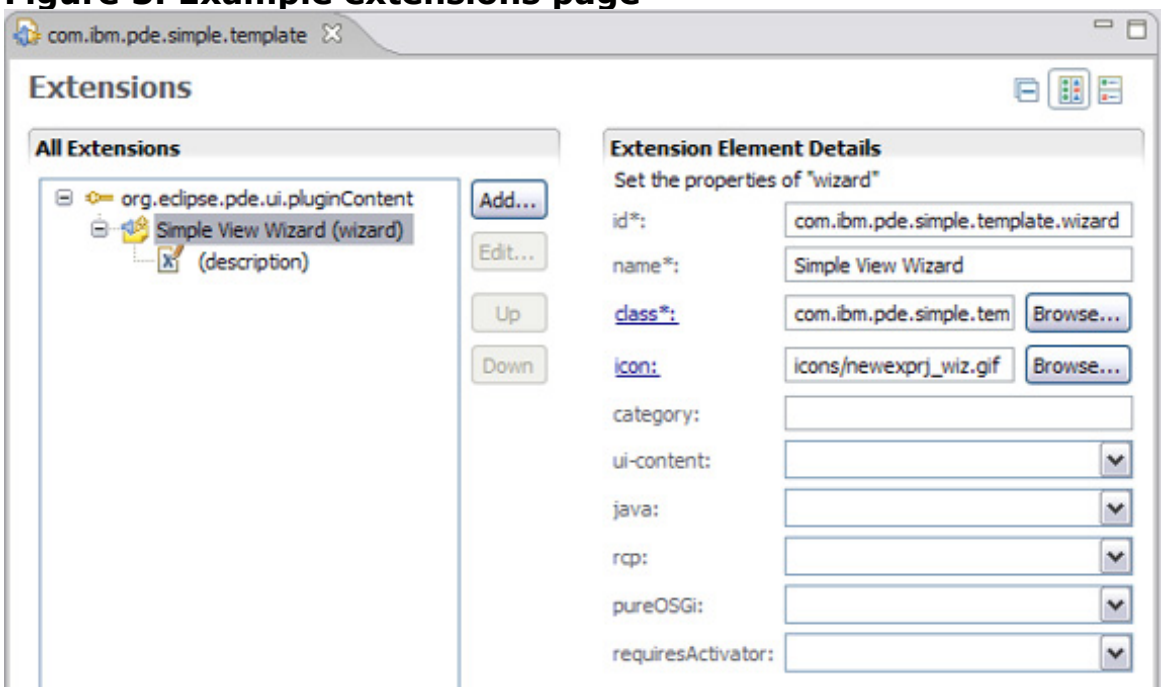
Template wizards

PDE templates

All the templates in the Plug-in Project wizard are owned by PDE and the source for these templates is freely available on Eclipse's CVS repository.

The most important extension point we'll use to create templates is org.eclipse.pde.ui.pluginContent. This provides the ability to contribute wizards that create additional content for PDE plug-in projects. After the plug-in manifest and key files have been created, these wizards can be used to add more files and extensions to the initial structure. Our implementation of this wizard is going to add content based on a parametrized template customized based on the user choices in a wizard. Now let's get started with this extension point.

Figure 3. Example extensions page



In Figure 3, we define a new wizard with ID (com.ibm.pde.simple.template.wizard),

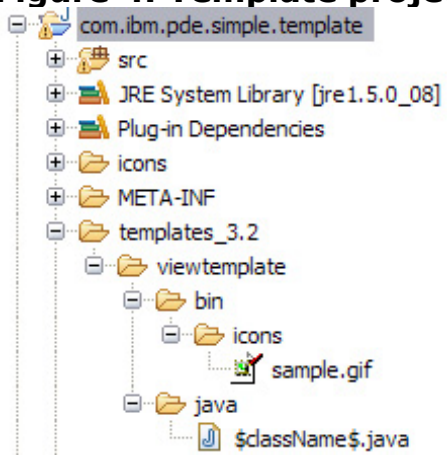
name (Simple View Wizard), icon and class definition (see the code below). The important method of the class definition is `createTemplateSections()`, which is responsible for returning template sections that drive the creation of content. The next section will discuss how to create our template file(s) and afterward, what exactly resides in a template section.

Listing 1. SimpleViewTemplateWizard.java

```
package com.ibm.pde.simple.template;

import org.eclipse.pde.ui.IFieldData;
import org.eclipse.pde.ui.templates.ITemplateSection;
import org.eclipse.pde.ui.templates.NewPluginTemplateWizard;

public class SimpleViewTemplateWizard extends NewPluginTemplateWizard {

                protected IFieldData fData;

    public void init(IFieldData data) {
        super.init(data);
        fData = data;
        setWindowTitle("Simple View Wizard");


    public ITemplateSection[] createTemplateSections() {
        return new ITemplateSection[] {new SimpleViewTemplateSection()};
    }

}
```

Template files

Figure 4. Template project structure



Conditional logic in templates

As you see in our template, it is possible to have simple conditional logic in our templates. The templating system supports simple and nested `if` statements. This can be helpful if you desire to make your templates flexible based on user input.

The next leg of our journey is to create a template we can use to generate code. To do this, we have to set up our project properly. If you glance at Figure 4, notice there is a templates_3.2 folder. The name of this folder is significant as it dictates to PDE what versions of Eclipse this template is applicable for. If you wanted your template to run on V3.1 and greater, you would name your folder templates_3.1. The child of this templates folder is viewtemplate and signifies the grouping of files, known as a *section*, you want associated with your template. You can name this folder anything you want.

Under the section folder, we have two folders: bin and java. These contain content that will be copied when the template is created. The content in the java folder is important because this is the class that represents our view. It's named $className$.java because the templating system will automatically swap out the variable `$className$` for the user-desired name.

Inside this class file, we also have variable substitutions we want our users to customize. The names of these variable substitutions are flexible and will be discussed in the next section, which focuses on template sections.

Listing 2. $className$.java

```
package $packageName$;

import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.part.ViewPart;

public class $className$ extends ViewPart {

                public void createPartControl(Composite parent) {
                                Label label = new Label(parent, SWT.CENTER);

%if importantMessage
                                String message = new String("$message$!!!");
%else
                                String message = new String("$message$");
%endif
                                label.setText(message);
                }

                public void setFocus() {}

}
```

Template sections

Template sections are the Java classes that contain the UI and control logic that drive the input to your template. Template sections must implement the `ITemplateSection` interface. For your convenience, PDE has an abstract class, `OptionTemplateSection`, that can save you a lot of time. There are quite a few things you can do with template sections (I'll leave it to you as an exercise to explore the included code sample), we're going to focus on the most important ones. The first step is to make sure we're adding wizard pages via the `addPages(...)` method. In this simple case, we only have one page to add, on the first page.

Listing 3. SimpleViewTemplateSection.java

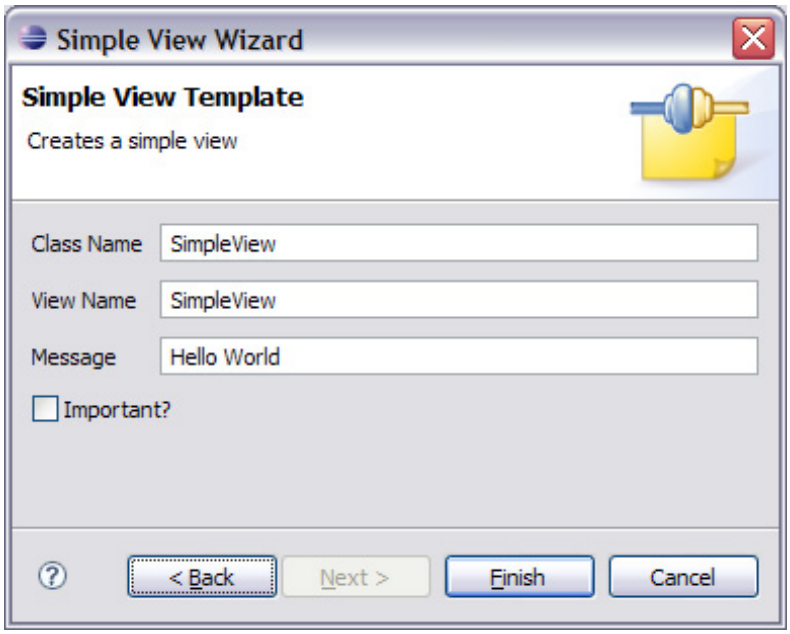
```
...
public void addPages(Wizard wizard) {
                WizardPage page = createPage(0, IHelpContextIds.TEMPLATE_INTRO);
                page.setTitle("Simple View Template");
                page.setDescription("Creates a simple view");
                wizard.addPage(page);
                markPagesAdded();
}
...
```

The next step is to provide clients with a UI so they can take advantage of the template. This is done by declaring variables used by the template and also by adding options via the `addOption(...)` method. In our class, we have a convenience method `createOptions()`, which gets called in the constructor to create the options.

Listing 4. SimpleViewTemplateSection.java

```
...
private static final String KEY_CLASS_NAME = "className";
private static final String KEY_VIEW_NAME = "viewName";
private static final String KEY_MESSAGE_NAME = "message";
private static final String KEY_IMP_MESSAGE_NAME = "importantMessage";
...
private void createOptions() {
                addOption(KEY_CLASS_NAME, "Class Name ", "SimpleView", 0);
                addOption(KEY_VIEW_NAME, "View Name", "SimpleView", 0);
                addOption(KEY_MESSAGE_NAME, "Message", "Hello World", 0);
                addOption(KEY_IMP_MESSAGE_NAME, "Important?", false, 0);
}
...
```

Figure 5. Simple view options wizard page

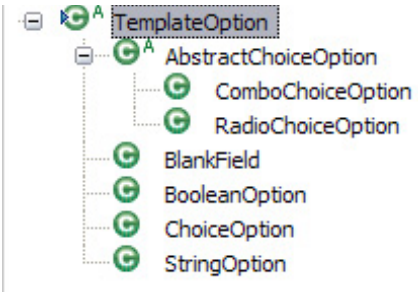


The templating system provides several methods to help you add options; you can see the implementation of these methods in the `BaseOptionTemplateSection` class. However, if you require more flexibility than what the templating system offers, I will point you to the `registerOption(...)` method. It is possible to create your own set of options, for example, if you wanted a combo choice option, you would instantiate a `ComboChoiceOption` and invoke the `registerOption(...)` method (see Table 1 for descriptions of options and Figure 6 for the full type hierarchy).

Table 1. PDE template options

Option (class name)	Description
BlankField	Used to create blank space on the template section wizard page
StringOption	Used to collect a string from the user in the template section wizard page
BooleanOption	Used to collect a boolean choice from the user in the template section wizard page
RadioChoiceOption	Used to collect a set of radio choices from the user in the template section wizard page
ComboChoiceOption	Used to collect a set of combo choices from the user in the template section wizard page
ChoiceOption	Deprecated -- please use RadioChoiceOption or ComboChoiceOption

Figure 6. Option hierarchy



The final step is to write the code that creates and populates the extension point(s) you're interested in. In our simple example, we are only concerned with the `org.eclipse.ui.views` extension point. In the PDE templating system, the `updateModel(...)` method is called when the template is being created, and it is expected of you as the template creator to create your extensions -- and other things -- here. As a side note, it is also possible to create multiple extension points even though the code listing below shows just the creation of one.

Listing 5. SimpleViewTemplateSection.java

```
...
protected void updateModel(IProgressMonitor monitor) throws CoreException {
                                IPluginBase plugin = model.getPluginBase();
                                IPluginModelFactory factory = model.getPluginFactory();

                                // org.eclipse.core.runtime.applications
                                IPluginExtension extension = \
                                createExtension("org.eclipse.ui.views", true);

                                IPluginElement element = factory.createElement(extension);
                                element.setName("view");
                                element.setAttribute("id", getStringOption(KEY_CLASS_NAME));
                                element.setAttribute("name", getStringOption(KEY_VIEW_NAME));
                                element.setAttribute("icon", "icons/sample.gif");

                                String fullClassName =
                                                getStringOption(KEY_PACKAGE_NAME)\
                                                +"."+getStringOption(KEY_CLASS_NAME);

                                element.setAttribute("class", fullClassName);
                                extension.add(element);

                                plugin.add(extension);
                }
...
```

org.eclipse.pde.ui.templates

The org.eclipse.pde.ui.templates extension point is important to note before we conclude. This extension point provides the gateway to the custom plug-in wizard. If

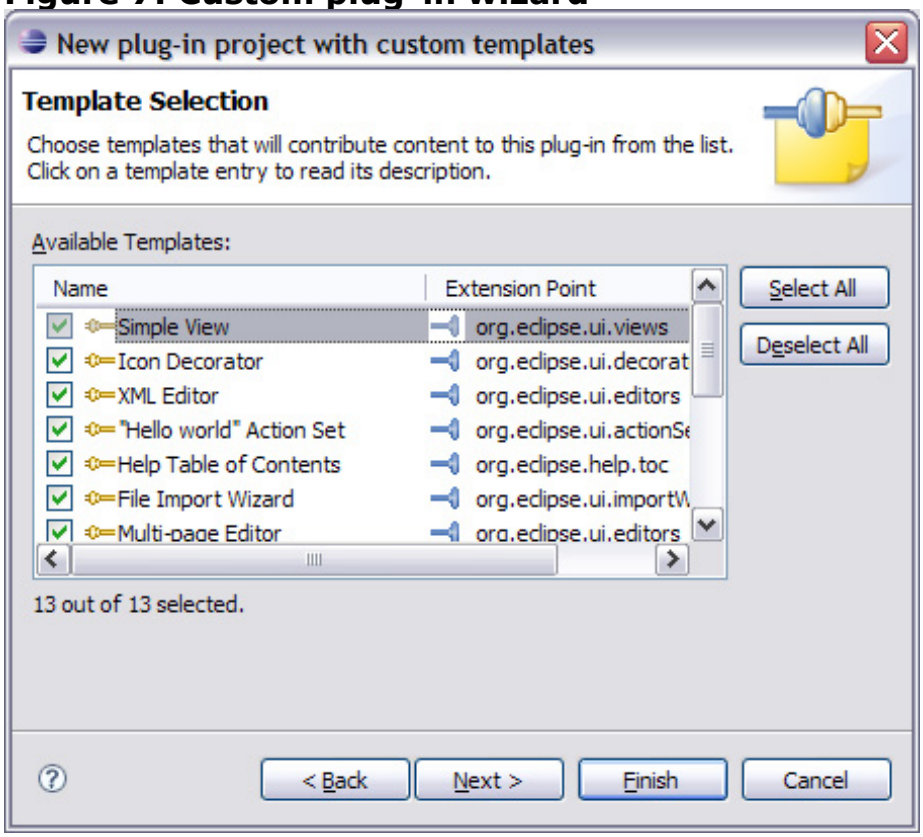
you have a specific extension point that you're templating, I recommend you extend this as it allows clients to use your template along with others (see Figure 7). In the case of the example used here, we can simply reuse the `OptionTemplateSection` implementation (`SimpleViewTemplateSection`). I just want to stress that if you created a custom extension point, it would help your users if you provided a template for them.

Remember when you were first learning Eclipse and had to create a view? The view template was highly valuable in understanding how things work. Extend the same courtesy to your users.

Listing 6. plugin.xml

```
<extension
         point="org.eclipse.pde.ui.templates">
      <template
            class="com.ibm.pde.simple.template.SimpleViewTemplateSection"
            contributingId="org.eclipse.ui.views"
            id="com.ibm.pde.simple.template"
            name="Simple View"/>
</extension>
```

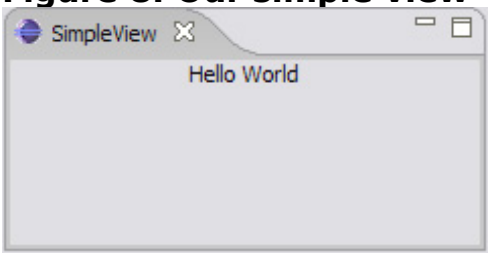
Figure 7. Custom plug-in wizard



Conclusion

After all this hard work, we are blessed with the ability to create a project and have it use our simple template. Once this project is created, we can easily look at the code and start to understand the intricacies of the template. We can also launch a runtime workbench environment and see the result of the template.

Figure 8. Our simple view



Our goal was to introduce you to PDE's templating system, and this was accomplished with a hands-on example. I encourage you to download the example listed below to see how it works. I hope it is evident to you that PDE's templating system offers a powerful way to quickly bootstrap project creation -- or maybe show off an example implementation of your own extension point -- for your end users. If you feel that the PDE templating system is lacking in certain areas, we gladly take feedback in the form of Bugzilla entries. We're always looking for ways to improve PDE.

Download

Description	Name	Size	Download method
SimpleView Template Plug-in	os-eclipse-pde.zip	12KB	HTTP

Information about download methods

Resources

Learn

- Learn more about Plug-in Development Environment (PDE) at Eclipse.org.
- Stay current regarding Eclipse happenings by visiting Planet Eclipse.
- Learn more about the Eclipse Foundation at Eclipse.org.

- Learn about another templating language in Eclipse called JET by reading "Create more -- better -- code in Eclipse with JET."
- For an excellent introduction to the Eclipse platform, see "Getting started with the Eclipse Platform."
- Expand your Eclipse skills by visiting IBM developerWorks' Eclipse project resources.
- developerWorks offers interesting podcasts for software developers.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Download the Eclipse Platform and get started with Eclipse now.
- See the latest Eclipse technology downloads at IBM alphaWorks.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- The PDE mailing list should be your first stop for discussing PDE.
- The Eclipse newsgroups has many resources for people interested in using and extending Eclipse.
- Platforms newsgroups should be your first stop to learn about the intricacies of PDE. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Chris Aniszczyk is a software engineer at IBM Lotus focusing on OSGi related development. He is an open source enthusiast at heart, and he works on the Gentoo Linux distribution and is a committer on a few Eclipse projects (PDE, ECF, EMFT). He's always available to discuss open source and Eclipse over a frosty beverage.