creating a text-based editor for eclipse

By Elwin Ho

Contact author at: Elwin.Ho@hp.com

June 2003

TABLE OF CONTENTS

creating a text-based editor for eclipse

purpose

The purpose of this white paper is to show you how to create an Eclipse editor with syntax coloring features and code completion functionality. Creating your own custom editor allows you to support a new language that the default editor does not. Think of this article as the entry point for creating such an editor. This white paper is intended for the intermediate Eclipse plug-in developer.

This paper does not go into details about the theory behind the Eclipse editor framework. For more detailed information, refer to the Eclipse online help and source code. [Reference 1]

Throughout this paper, there are links to code samples. These samples are the finished product for each section. These are provided so that you have working examples to build, run, and extend.

Eclipse is an open source framework for building Integrated Development Environment using a pre-built set of tools and GUI Components. It has generated interest from many developers since it was first released.

The Eclipse Workbench consists of:

- Editors – specific editors are associated with various types of documents
- Views – views support editors
- Perspectives – a combination of specific views and editors

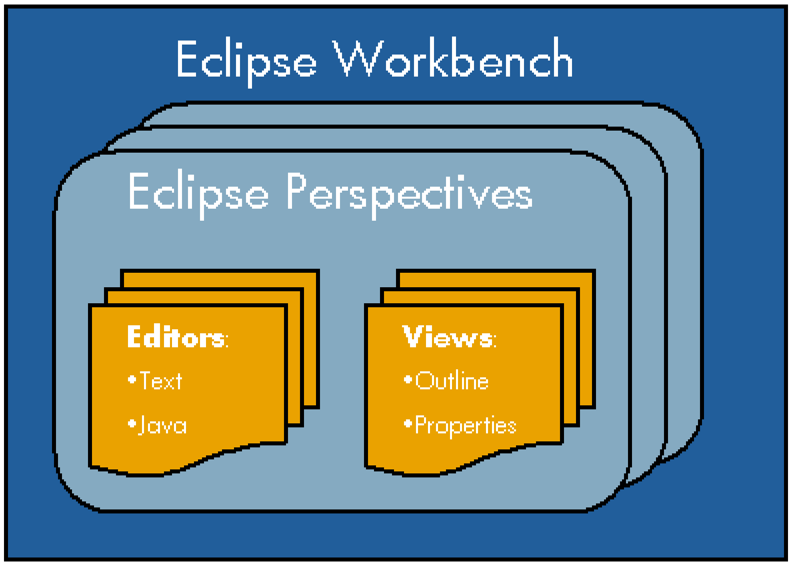


Figure 1. Eclipse Workbench

Below is a picture of the Eclipse Resource Perspective with a text document open in the Text Editor. As you can see, the Resource Perspective contains:

- Text Editor
- Navigator View
- Outline View
- Task View

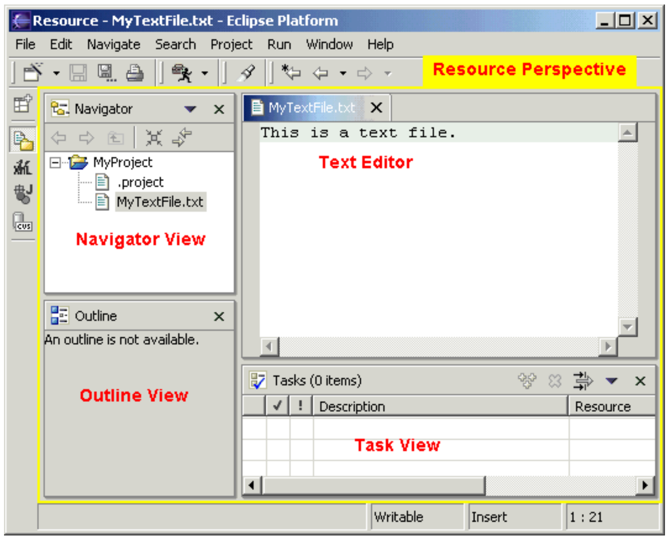


Figure 2. Eclipse Resource Perspective

For additional information on the Eclipse Workbench, or any terminology used in this paper, refer to the Workbench User Guide in the Eclipse online help (**Help > Help Contents**).

creating a simple text editor

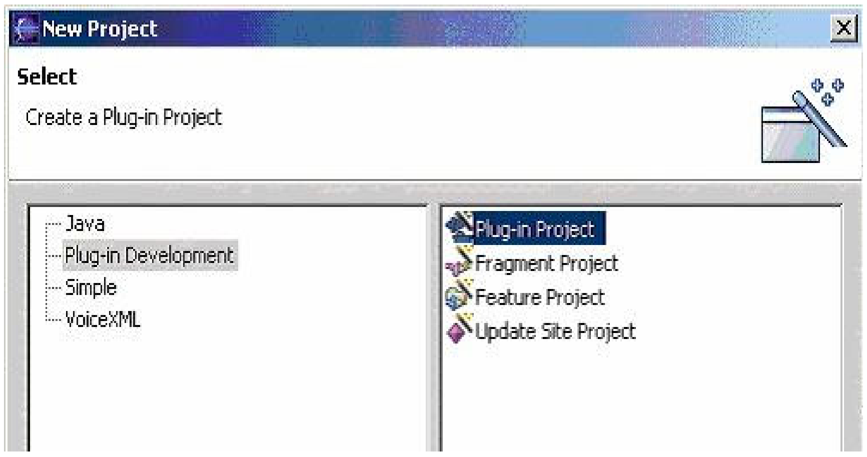
In this section, we describe how to create an Eclipse plug-in that provides basic text editor features. We also show you how to associate a custom icon with files created using this editor. All this functionality is a result of using Eclipse wizards. We don't have to write a single line of Java code!

To begin, download the sample project by clicking the link MyEditor1 and then import this project into your Eclipse workbench.

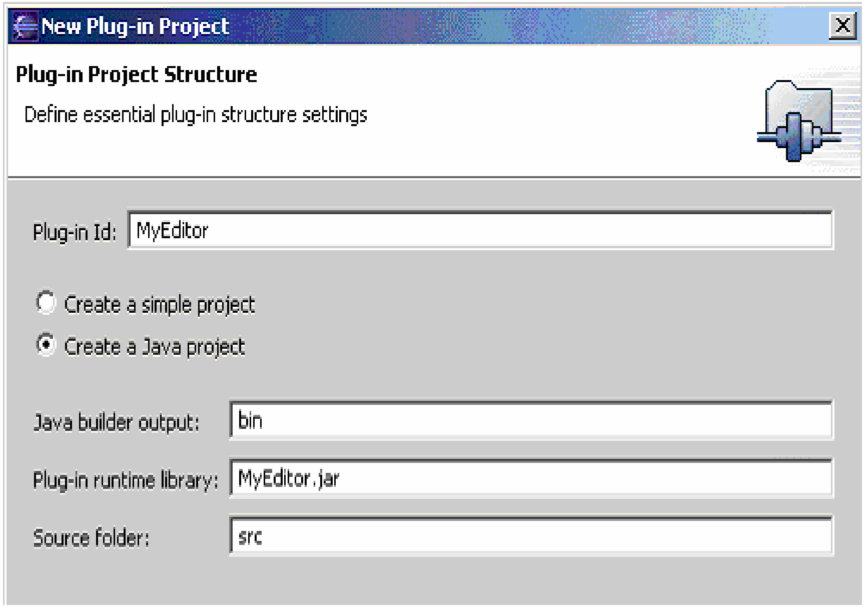
Create a Plug-in Development Project in Eclipse

To create a plug-in development project in Eclipse:

1) Select **File** > **New** > **Project** from the menu bar to open the **New Project Wizard**.
2) Select **Plug-in Development** from the left pane and **Plug-in Project** from the right pane. Click **Next**.



3) Enter a **Project name** and click **Next**. For example, we named our project `MyEditor`.


New Plug-in Project
Plug-in Project Structure
Define essential plug-in structure settings

Plug-in Id: MyEditor

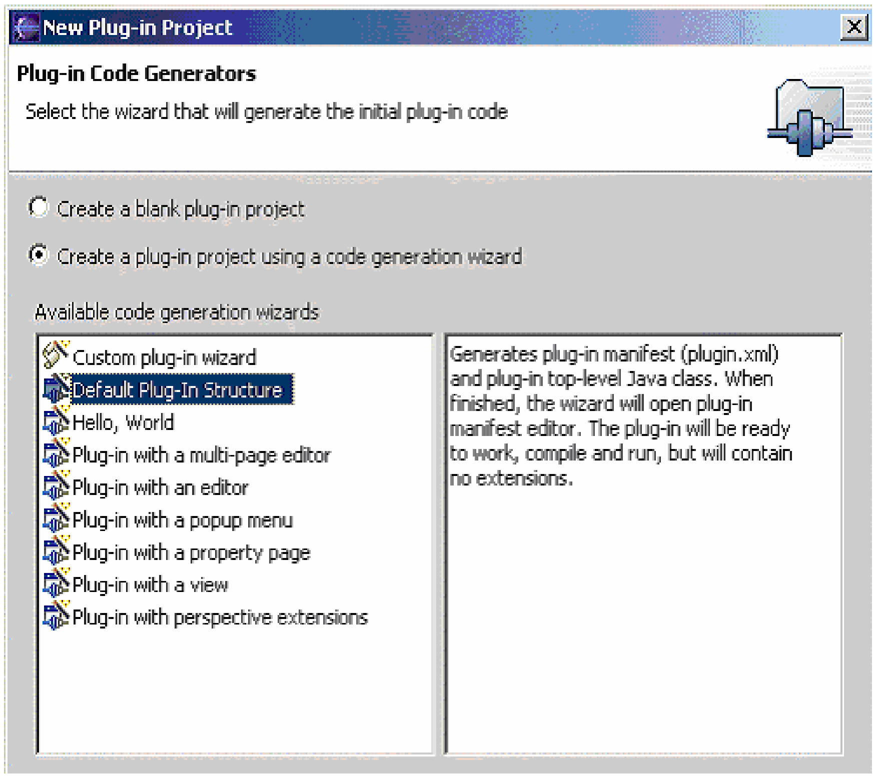
Create a simple project [✓ Create a Java project]

Java builder output: bin

Plug-in runtime library: MyEditor.jar

Source folder: src

4) Select the **Default Plug-In Structure** and click **Next**.


New Plug-in Project

Plug-in Code Generators

Select the wizard that will generate the initial plug-in code

( ) Create a blank plug-in project

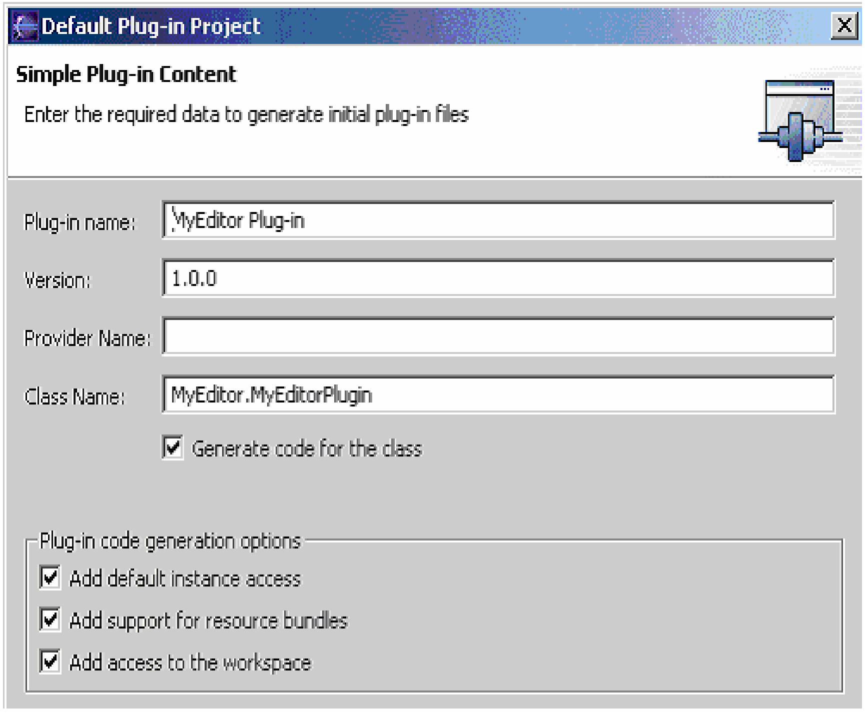
(●) Create a plug-in project using a code generation wizard

Available code generation wizards

- Custom plug-in wizard
- **Default Plug-In Structure**
- Hello, World
- Plug-in with a multi-page editor
- Plug-in with an editor
- Plug-in with a popup menu
- Plug-in with a property page
- Plug-in with a view
- Plug-in with perspective extensions

Generates plug-in manifest (plugin.xml) and plug-in top-level Java class. When finished, the wizard will open plug-in manifest editor. The plug-in will be ready to work, compile and run, but will contain no extensions.

5) Choose **Create a plug-in project using a code generation wizard** and select the **Default Plug-In Structure** code generation wizard. Click **Next**.
6) Accept the **Simple Plug-in Content** default settings and click **Finish**.


Default Plug-in Project

Simple Plug-in Content

Enter the required data to generate initial plug-in files

Plug-in name: MyEditor Plug-in

Version: 1.0.0

Provider Name:

Class Name: MyEditor.MyEditorPlugin

[✓] Generate code for the class

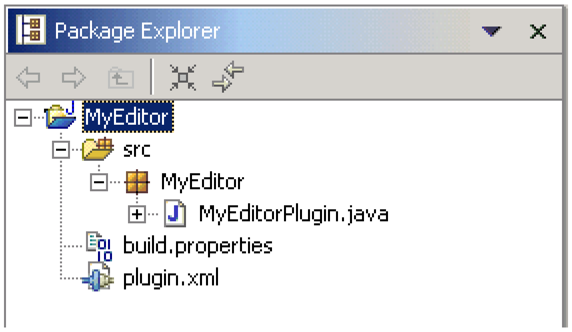
Plug-in code generation options

[✓] Add default instance access

[✓] Add support for resource bundles

[✓] Add access to the workspace

7) Our new editor plug-in project appears in the Package Explorer view.



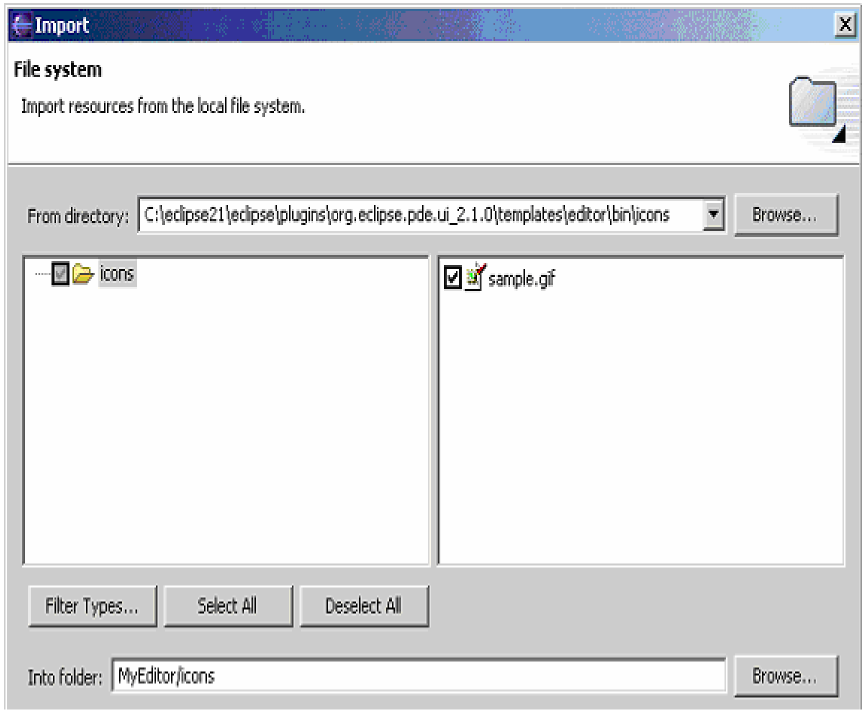
We have created a basic Eclipse plug-in development project. Now we can start customizing our editor.

Import an Icon to Associate with Our Editor's Files

Our next step is to import an icon for tagging the file type associated with our new editor. We are going to import a sample icon from the Eclipse sample project into our plug-in.

To import an icon into our plug-in:

1) Create a directory for the icon: highlight our project and select **File > New > Folder** from the menu bar.
2) Name our folder and click **Finish**. For example, we named our new folder `icons`.
3) Import the sample icon: highlight our new folder and select **File > Import** from the menu bar.
4) Select **File system** and click **Next**.



File system

Import resources from the local file system.

From directory: C:\eclipse21\eclipse\plugins\org.eclipse.pde.ui_2.1.0\templates\editor\bin\icons

☑ icons

☑ sample.gif

Into folder: MyEditor/icons

5) Browse to the `eclipse\plugins\org.eclipse.pde.ui_2.1.0\templates\editor\bin\icons` folder then,
   a) Click OK.
   b) Select `sample.gif` in the right pane.
   c) Make sure the **Into folder** field has the correct path: `MyEditor/icons`.
   d) Click **Finish.**

Now we are ready to create an extension point to register our plug-in with Eclipse.

Register Our Editor Plug-in with Eclipse

We must add an extension point to the plugin.xml file to register the editor plug-in with Eclipse.

To register your plug-in by adding an extension point the plugin.xml file:

1) Open the `plugin.xml` file: Right-click on the file in the Package Explorer, click Open. To see the source code, click the **Source** tab in the bottom of the Plug-in editor view.
2) Add the following extension point definition to the end of the `plugin.xml` file right before `</plugin>`. See the table below for an explanation of each attribute we are adding.

```
<extension
       point="org.eclipse.ui.editors">
    <editor
          name="MyEditor"
          icon="icons/sample.gif"
          extensions="myfile"
          contributorClass="org.eclipse.ui.editors.text.TextEditorActionContributor"
          class="org.eclipse.ui.editors.text.TextEditor"
          id="org.eclipse.ui.editors.text.texteditor">
    </editor>
 </extension>
```

Attribute	*Description*
`point=”org.eclipse.ui.editors`	Extension point for registering our editor.
`name=”MyEditor”`	Our editor's name.
`icon=”icons/sample.gif”`	Icon that is associated with files created by our editor. If you named your folder something different than `icons`, make sure to change this attribute to specify the correct folder.
`extensions=”myfile”`	Extension associated with files created by our editor.
`contributorClass=”org.eclipse.ui.editors.text.TextEditorActionContributor”`	Menu and toolbar action definitions.

`class="org.eclipse.ui.editors.text.TextEditor"`	Our editor's main class. This is the default text editor from the Eclipse framework. This provides basic text editing features.
`id="org.eclipse.ui.editors.text.texteditor"`	Unique id for this contribution

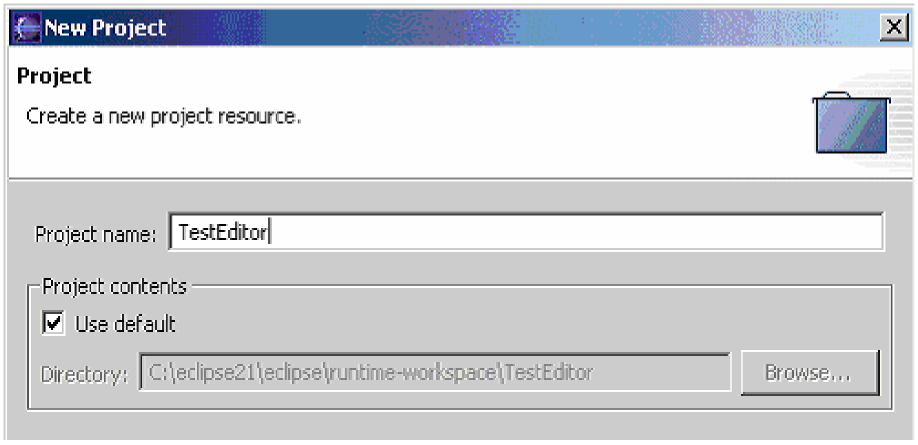
Build, Run, and Test the Project

To build and run our project:

1) Build our plug-in project by selecting **Project > Rebuild All** from the menu bar.
2) Run our project by selecting **Run > Run As > Run-time Workbench**. This will start a new instance of Eclipse that uses our new editor.

To test our editor:

1) Create a simple project by selecting **File > New > Project** from the menu bar.
2) Select **Simple** from the left pane and **Project** from the right pane. Click **Next**.
   e) Enter a **Project name** and click **Finish**. For example, we called our project `TestEditor`.

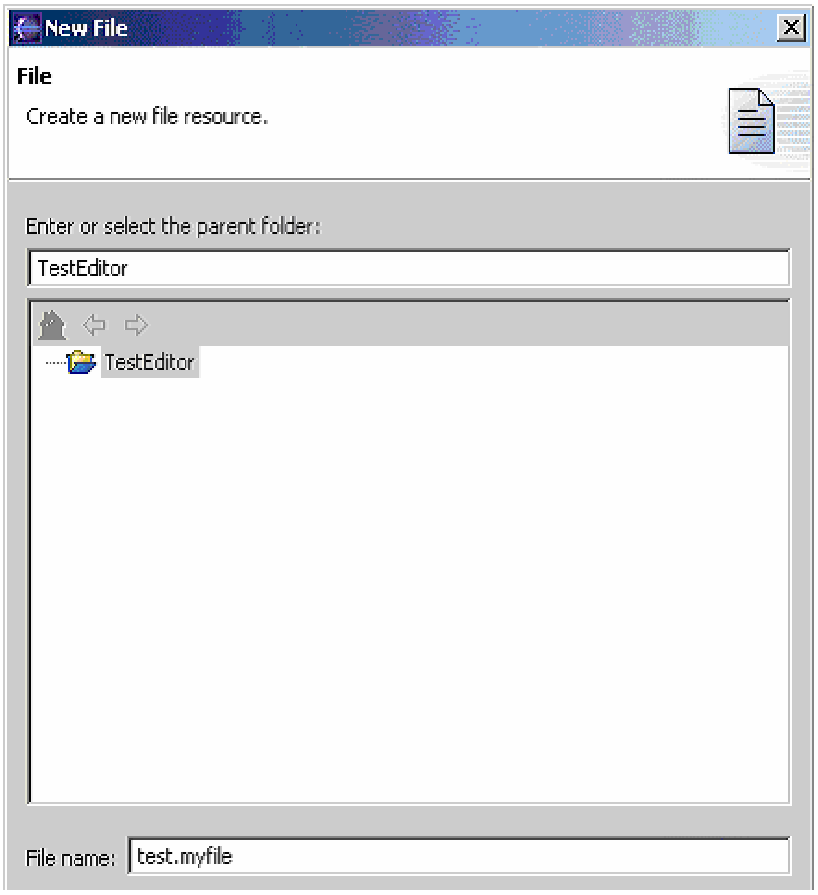


Project name: TestEditor

☑ Use default

Directory: C:\eclipse21\eclipse\runtime-workspace\TestEditor

   f) Select **File > New > File** from the menu bar to create a new file.
   g) Name our file with the `.myfile` extension. For example, we named our file `test.myfile`.


New File

File

Create a new file resource.

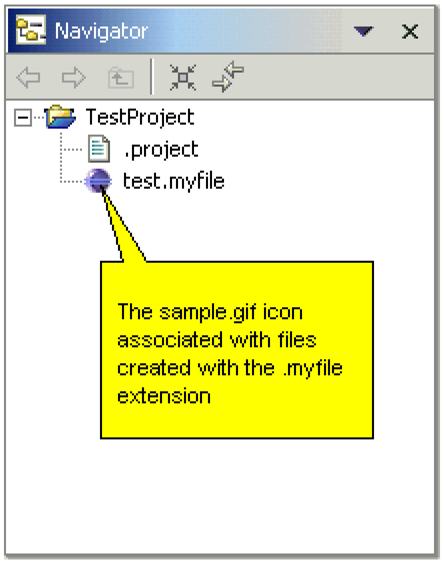
Enter or select the parent folder:

TestEditor

TestEditor

File name: test.myfile

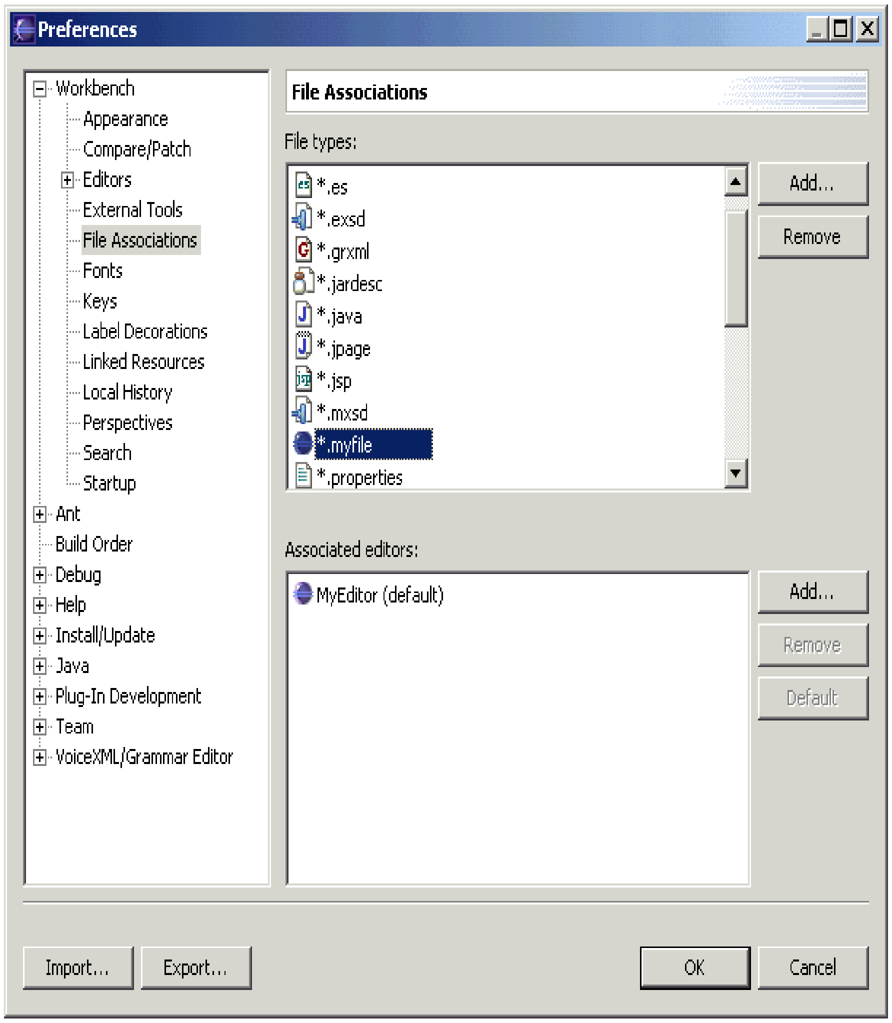
h) Our new file with the sample icon appears in the Navigator view.


Navigator
TestProject
.project
test.myfile
The sample.gif icon associated with files created with the .myfile extension

To view our editor file associations:

1) Select **Window** > **Preferences** from the menu bar.

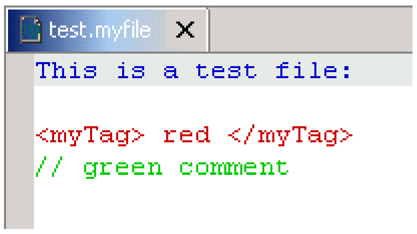
2) In the Preferences dialog box, expand the **Workbench** tree and highlight **File Associations**. Notice that files with the `.myfile` extension is associated with MyEditor.



adding syntax coloring to the text editor

We can extend our editor with different functionality. Let's add syntax coloring to our editor that designates:

- Red for `<myTag>` elements and content
- Green for single-line comments (preceded by //)



```
This is a test file:

<myTag> red </myTag>
// green comment
```

Eclipse contains APIs that help us add syntax coloring to our editor. Eclipse uses the damage, repair, reconcile model. Every time the user changes the document, the presentation reconciler determines which region of the visual presentation should be invalidated and how to repair it.

- Damage is the text that must be redisplayed.
- Repair is the method used for redisplaying the damaged area.
- The process of maintaining the visual presentation of a document as changes are made in the editor is known as reconciling.

To customize the damage/repair model to our editor, we extend the default `TextEditor` class from Eclipse and then configure it to use our own `SourceViewerConfiguration` class. The `SourceViewer` is the central class for configuring the editor with pluggable behavior such as syntax coloring and code assistance. By default, the `SourceViewer` in the default text editor does not support syntax coloring. We can create our own `SourceViewer` class by extending the `SourceViewerConfiguration` class and overriding some of its methods to add the syntax-highlighting feature to our editor. The following graphic shows the relationship of these basic classes and the functions they provide.

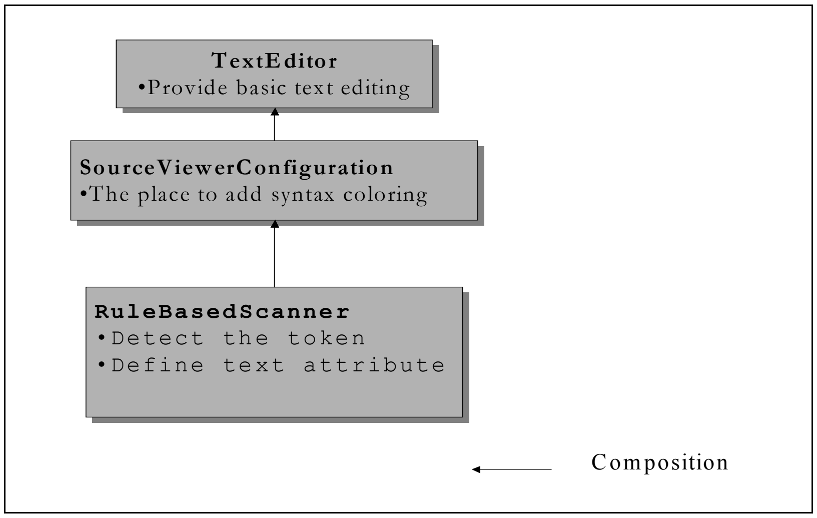

TextEditor
•Provide basic text editing
SourceViewerConfiguration
•The place to add syntax coloring
RuleBasedScanner
•Detect the token
•Define text attribute
Composition

Figure 3: Relationship of damage/repair relate classes

For more information on these classes, please refer to the “Configuring a source viewer” and “Syntax coloring” topic in the Editors chapter of the *Platform Plug-in Developer Guide* in the Eclipse online help.

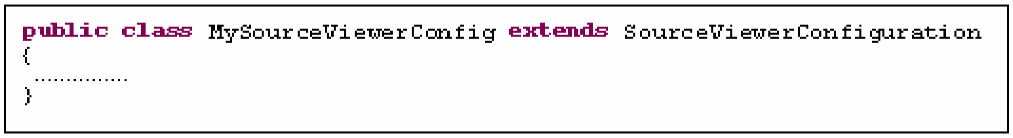
Modify the Code

To begin, download the sample project by clicking the link MyEditor2 and then import this project into your Eclipse workbench. This sample project adds three classes to the default `MyEditorPlugin`, `MyEditor`, `MyRuleScanner`, and `MySourceViewerConfig`. These classes provide the following basic features:

- `MyEditorPlugin` – The standard class for plug-ins that integrate with the Eclipse platform UI. It is created by the project wizard.
- `MyEditor` – extends the default TextEditor class in Eclipse. We change the default SouceViewer in this class.
- `MyRuleScanner` – extends the `RuleBasedScanner` class in Eclipse and actually sets the rules for our editor (makes specified tags red and single-line comments green) that will be used in the damage/repair class.
- `MySourceViewerConfig` – extends the `SourceViewerConfiguration` class in Eclipse, instantiates the rule scanner, and defines the reconciler action.

The following steps detail the classes added for our new color syntax:

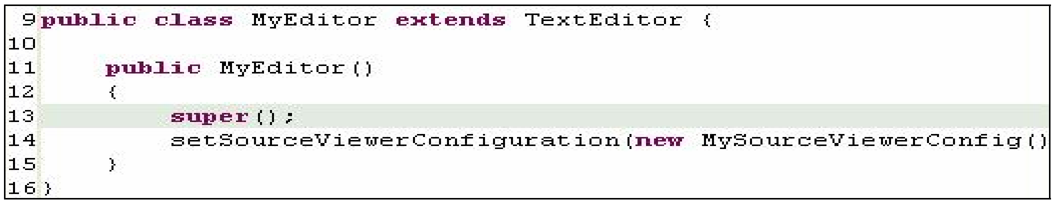
Define a new class that extends the Eclipse SourceViewerConfiguration class:



```
public class MySourceViewerConfig extends SourceViewerConfiguration
{
 ...............
}
```

Listing 1 MyScouceViewConfig.java

Extend the basic `TextEditor` as our editor and configure `MySourceViewerConfig` as our editor source viewer in the `MyEditor` class:

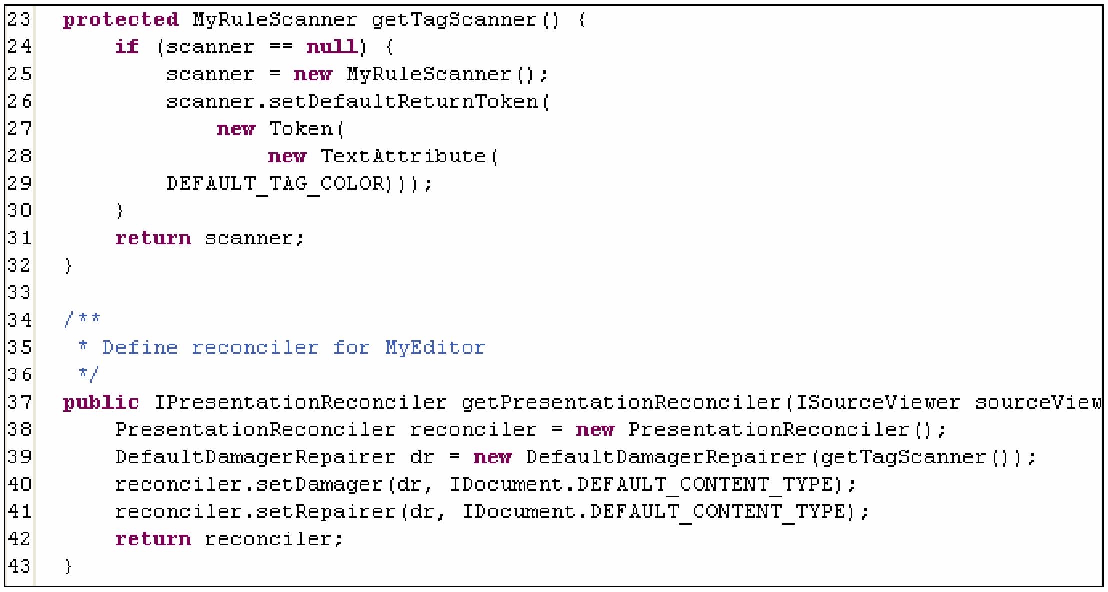


```
public class MyEditor extends TextEditor {

    public MyEditor()
    {
        super();
        setSourceViewerConfiguration(new MySourceViewerConfig()
    }
}
```

Listing 2. MyEditor.java

Now set up the damage/repair/reconcile model by overwriting the `SourceViewerConfiguration.getPresentationReconciler()` method. Configure the default `DamagerRepairer` as our damage/repair handler as shown in line 39 in the text below. It uses `MyRuleScanner` as parameter (return by `getTagScanner` method in line 23 to 31). The default `DamagerRepairer` takes the token that we define in `MyRuleScanner` to repair the text that includes the text attribute, in this case the text color. (In list 4, line 21 to 27)

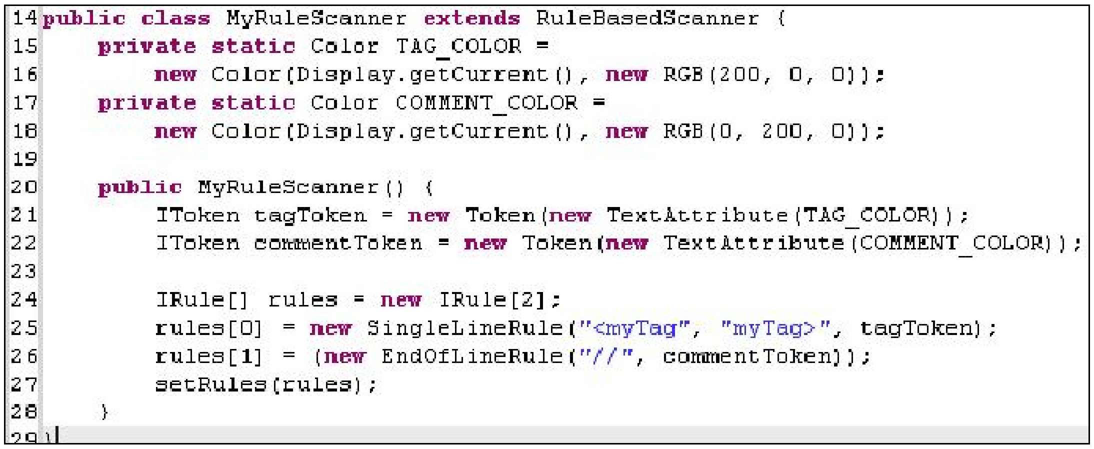


```java
    protected MyRuleScanner getTagScanner() {
        if (scanner == null) {
            scanner = new MyRuleScanner();
            scanner.setDefaultReturnToken(
                new Token(
                    new TextAttribute(
            DEFAULT_TAG_COLOR)));
        }
        return scanner;
    }

    /**
     * Define reconciler for MyEditor
     */
    public IPresentationReconciler getPresentationReconciler(ISourceViewer sourceView
        PresentationReconciler reconciler = new PresentationReconciler();
        DefaultDamagerRepairer dr = new DefaultDamagerRepairer(getTagScanner());
        reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
        reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);
        return reconciler;
    }
```

Listing 3 SourceViewerConfiguration.java

`MyRuleScanner` class builds rules for detecting different kinds of tokens such as single line comments (//). It adds rules to detect the patterns we specify and associates the text attributes, that is, the colors with these patterns. It then returns a token for the reconciler action. For example, in line 17 and line 26 we associate the color green (RGB 0, 200, 0) with a comment line in listing 4.



```java
14 public class MyRuleScanner extends RuleBasedScanner {
15     private static Color TAG_COLOR =
16         new Color(Display.getCurrent(), new RGB(200, 0, 0));
17     private static Color COMMENT_COLOR =
18         new Color(Display.getCurrent(), new RGB(0, 200, 0));
19
20     public MyRuleScanner() {
21         IToken tagToken = new Token(new TextAttribute(TAG_COLOR));
22         IToken commentToken = new Token(new TextAttribute(COMMENT_COLOR));
23
24         IRule[] rules = new IRule[2];
25         rules[0] = new SingleLineRule("<myTag", "myTag>", tagToken);
26         rules[1] = (new EndOfLineRule("//", commentToken));
27         setRules(rules);
28     }
29 }
```

Listing 4 MyRuleScanner.java

Build, Run, and Test the Project

Build and run our project the same way we did previously.

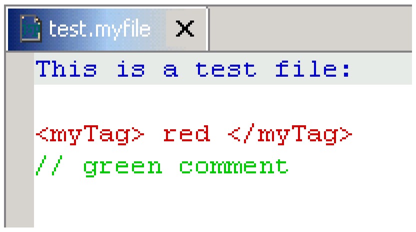
To test the syntax-highlighting feature:

1) Create a test file. For example, we named our file `test.myfile`.

2) Type the following lines in the new file:

```
<myTag> red </myTag>
// green comment
```

The first line appears in red and the second line in green.



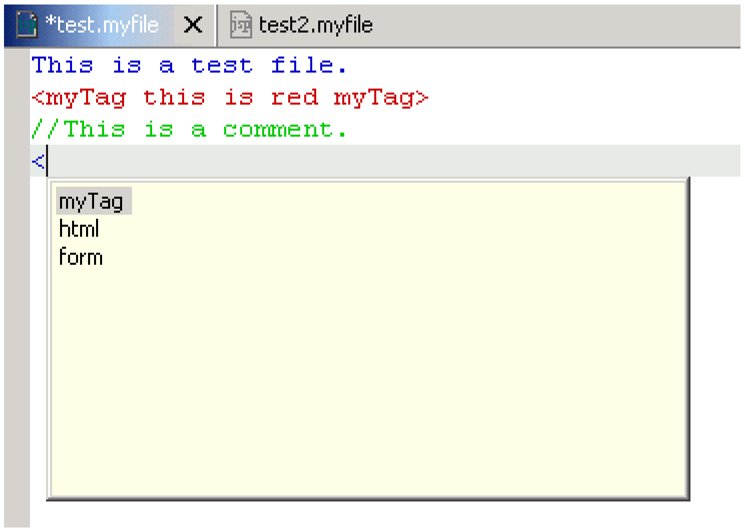
```
This is a test file:

<myTag> red </myTag>
// green comment
```

adding content assistance to the text editor

Content assistance is a feature that allows an editor to provide context-sensitive content completion upon user request. To pop up a content assistance box, press CTRL-space or enter key characters when editing a file. We can double-click a selection in the content assistance box to insert that particular text into our file.

Now that we have added syntax coloring, we detail how to add a content assistance feature to our editor. Our content assistance box contains the options `myTag`, `html`, and `form`.



```
This is a test file.
<myTag this is red myTag>
//This is a comment.
<
```

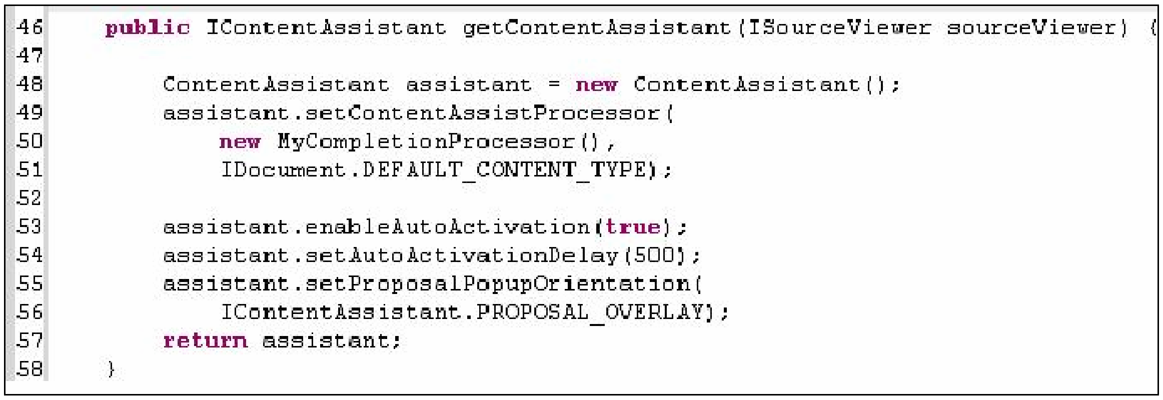
Modify the Code

To begin, download the sample project by clicking the link MyEditor3 and then import this project into your Eclipse workbench. This sample project adds one new class (`MyCompletionProcessor`) to the three we added in the last section. The four classes are:

- `MyCompletionProcessor` – provides the text completion action (new class)
- `MyEditor` – extends the `TextEditor` class in Eclipse

- MyRuleScanner – extends the RuleBasedScanner class in Eclipse and actually sets the rules for our editor
- MySourceViewerConfig – extends the SourceViewerConfiguration class in Eclipse and instantiates the content assistant.

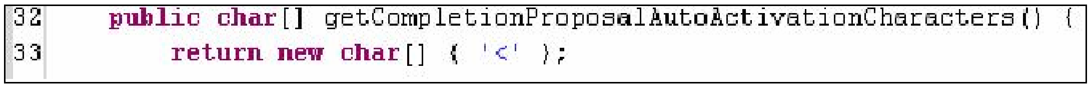
As in syntax coloring, we need to customize content assistance object in the MySourceViewerConfig class to implement the content assistance. In the MySourceViewerConfig class, we use Eclipse's ContentAssistant class to enable content assistance as shown in line 48 below. The MyCompletionProcessor class provides the completion action. We set the assistance properties (for example, orientation and delay time) in lines 53 to 56.



```
46      public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {
47
48          ContentAssistant assistant = new ContentAssistant();
49          assistant.setContentAssistProcessor(
50              new MyCompletionProcessor(),
51              IDocument.DEFAULT_CONTENT_TYPE);
52
53          assistant.enableAutoActivation(true);
54          assistant.setAutoActivationDelay(500);
55          assistant.setProposalPopupOrientation(
56              IContentAssistant.PROPOSAL_OVERLAY);
57          return assistant;
58      }
```

List 5 MySourceViewerConfig.java

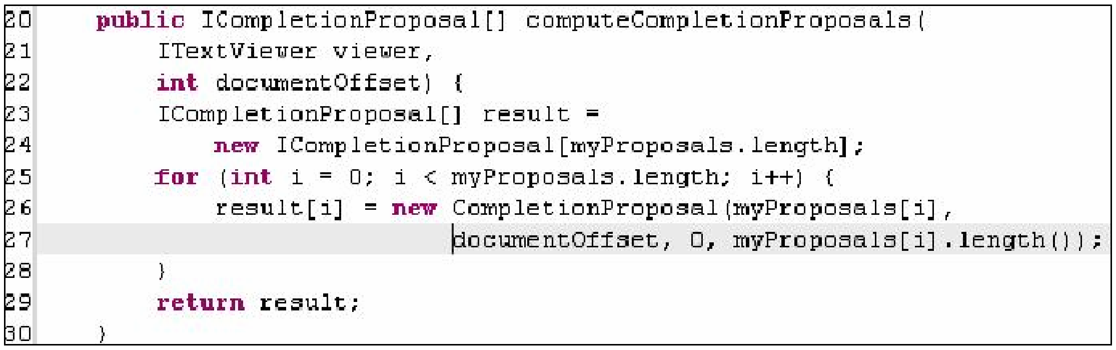
Inside MyCompletionProcessor, we must define the trigger function. This function tells the system that whenever the "<" is typed, code assistance should be invoked. We do this by overriding getCompletionProposalAutoActivationCharacters() in the MyCompletionProcessor class.



```
32      public char[] getCompletionProposalAutoActivationCharacters() {
33          return new char[] { '<' };
```

List 6 MyCompletionProcessor.java

We also implement the computeCompletionProposals method to return the completion proposal for the content assistance. In this example, we simply return a list of hard-coded text in line 25 to 27.



```
    public ICompletionProposal[] computeCompletionProposals(
        ITextViewer viewer,
        int documentOffset) {
        ICompletionProposal[] result =
            new ICompletionProposal[myProposals.length];
        for (int i = 0; i < myProposals.length; i++) {
            result[i] = new CompletionProposal(myProposals[i],
                                documentOffset, 0, myProposals[i].length());
        }
        return result;
    }
```

List 7 MyCompletionProcessor.java

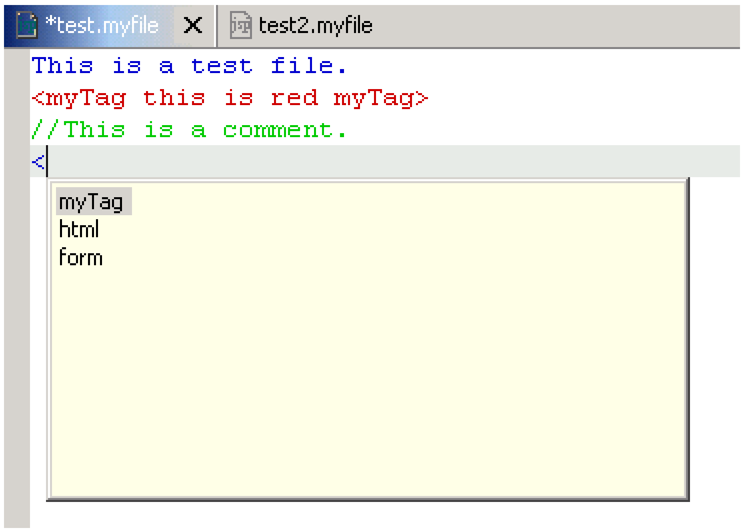
Build, Run, and Test the Project

Once again, build and run our project.

To test the content assistance feature:

1) Open `test.myfile`.

2) On a new line, type <. Double-click on one of the selections to add it to our file.



```
This is a test file.
<myTag this is red myTag>
//This is a comment.
<
```

We can also get the content assistance box to pop up by simply:

- Pressing CTRL+Space
- Selecting **Edit** > **Content Assist** from the Eclipse menu bar

summary

The editor is the most frequently used tool in an IDE. It has a major impact on the success of any IDE. Any features we can add to help developers increase their accuracy and efficiency greatly enhances the usefulness of our editor. The Eclipse platform provides a rich set of APIs and frameworks to allow different vendors to create their own unique development environments. Once we have created our own editor, we can add many features besides the syntax coloring and content assistance demonstrated in this white paper. Some of the possible features we can add include content outlining, annotation, and text hovering. The Eclipse online help provides more information about these features.

reference:

1. Eclipse on line help [http://dev.eclipse.org/help21/index.jsp]
2. Eclipse news group [http://www.eclipse.org/newsgroups/index.html]
3. Wiki wiki [http://eclipsewiki.swiki.net]