Create a commercial-quality Eclipse IDE, Part 2: The user interface

Skill Level: Intermediate

Prashant Deva (pdeva@placidsystems.com)
Founder
Placid Systems

17 Oct 2006

This tutorial -- second in this "Create a commercial-quality Eclipse IDE" series -- shows how to create the UI of the IDE. It examines the editor user interface framework that Eclipse offers, as well as the SourceViewerConfiguration class and all the various classes related to it, used to implement and configure your IDE's UI.

Section 1. Before you start

About this series

This "Create a commercial-quality Eclipse IDE" series demonstrates what it takes to create integrated development environments (IDEs) as Eclipse plug-ins for any existing programming languages or your own. It walks you through the two most important parts of the IDE -- the core and the user interface (UI) -- and takes a detailed look at the challenges associated with designing and implementing them.

This series uses ANTLR Studio IDE as a case study and examines its internals to help you understand what it takes to create a professional commercial-level IDE. Code samples help you follow the concepts and understand how to use them in your own IDE.

About this tutorial

Part 1 of this series introduces the architecture of an IDE and shows how to create the IDE's core layer. This second installment shows how to implement the UI component. You'll learn the various UI-related objects in Eclipse and how to use and extend those to provide specific functionality in your IDE. The tutorial contains many code samples for easy copying into your IDE's code to get that functionality working in your IDE right away.

Prerequisites

This tutorial assumes a basic knowledge of creating plug-ins for Eclipse and using the Eclipse Plug-in Development Environment (PDE).

System requirements

To run the code samples in this tutorial, you need a copy of the Eclipse Software Development Kit (SDK) running Java™ Virtual Machine (JVM) V1.4 or later.

Section 2. Basic editor framework

What's the first thing that comes to mind when you think of a UI for an IDE? The editor! In fact, everything concerning the UI of an IDE revolves around the editor. This is particularly true with Eclipse plug-ins because basic UI functionality -- the workbench, tool bars, etc. -- are already implemented. You simply have to specialize everything related to your IDE.

We'll begin by looking at the architecture of a text editor inside Eclipse. Eclipse divides the concept of a text editor into two parts: the document and the viewer. While the *document* holds the actual content of the editor, the *viewer* is responsible for handling the content display. This nice Model-View-Controller (MVC)-style separation allows you to separate code that manipulates the content of the editor from the part that manipulates its display. For example, the parser/lexer runs in the background thread and communicates only with the document, while the syntax highlightter can communicate only with the viewer without concerning itself with the document.

The document

The document represents the "model" part of the text editor framework. It contains the actual textual content of the editor. It does not concern itself with the display of the text in any way.

A document allows you to set the actual textual content of an editor and contains methods to manipulate the content. It is defined by the `org.eclipse.jface.text.IDocument` interface. For manipulating documents, the `IDocument` interface provides the `replace()` method:

```
void  replace (int offset, int length, String text) throws BadLocationException
```

The method takes the offset in the document to replace, along with the length of text to replace and the text to replace it with. Note that the offsets start at 0. Listing 1 shows examples of using the `replace()` method to manipulate the content of the document.

Listing 1. Replace method examples

```
document.set("Hello"); // set the text of the document to 'Hello'

document.replace(document.length()-1,0 " World");
  //Adds the word ' World' to the document

System.out.println(document.get()); //Prints 'Hello World'

document.replace(1,4, "i"); //replaces 'ello' with 'i'

System.out.println(document.get()); //Prints 'Hi World'
```

The document also provides abstraction for *lines*. It does this with the help of *positions*, which are like stickers to a portion of text in the document. After you attach a position object to a range of text, the object updates itself automatically while the user edits the text of the document. For example, if you assign a position at offset 20 and the user adds a character to the document at offset 19, the position object updates itself to position 21.

One interesting part in the design of the `IDocument` interface is the support for multithreaded access. Note that none of the implementations of `IDocument` allow for multi-threaded access. Thus, you must take care of synchronizing all the access to the document yourself. But the designers of `IDocument` thought about this and tried to make things just a bit easier for us.

The design of `IDocument` is made to be "fail-fast." It throws a `BadLocationException` exception whenever you try to access text from it that is outside the bounds of the document. Because this is a checked exception, you must create a handler for it everywhere you want to access any text from the document. In this way, you can catch errors in a multithreaded program easily because if any thread attempts to access text outside the bounds, it fails with an exception -- thus, effectively notifying you that you missed synchronizing something somewhere. This functionality also allows you to catch bugs much more easily and earlier in the development process than if you were to silently let the operation fail.

The viewer

The viewer is responsible or displaying the content of the document and is defined by the `org.eclipse.jface.text.ITextViewer` interface. The `TextViewer` is built on top of the Standard Widget Toolkit (SWT) `StyledText` widget. The `Viewer` and `Document` classes are present to provide an MVC separation on top of the `StyledText` widget.

Now, while the `TextViewer` is designed as a generic implementation to handle any kind of text, Eclipse provides a special subclass of it specifically for displaying source code. This subclass is called the `SourceViewer`, and it is implemented by the `org.eclipse.jface.text.source.SourceViewer` class, which is the class you will use as your text viewer for this series.

All the UI-related activities for your editor, such as syntax highlighting, text hovers, and text folding, are handled in the source viewer itself. A `SourceViewer` uses a vertical ruler at the left of the text widget to display all the warnings and a **+/-** icon for folding, etc., when you open a Java file. It also uses an overview ruler to the right of the text widget to display all the error and warning markers in the document so you can navigate to them quickly.

`SourceViewer` also uses a `SourceViewerConfiguration` class, which allows you to selectively plug in customizable UI behavior to fit your needs. You'll see the `SourceViewerConfiguration` class in detail later.

The text editor

The previous two classes are part of the SWT/JFace framework, which Eclipse uses. The `org.eclipse.ui.editors.text.TextEditor` class ties the document and the viewer together and inserts Eclipse-specific functionality. Thus, while the viewer and document abstractions are present in the JFace framework, the `TextEditor` is part of the core Eclipse UI text framework. The `TextEditor` implements the `org.eclipse.ui.IEditorPart` class, which -- as the name implies -- denotes to the Eclipse Workbench that it is part of the workbench and is an editor. An editor is associated with an `IEditorInput` class, which defines the protocol of editor input.

For our purposes, you simply need to subclass the `TextEditor` in your plug-in and override the methods to customize it according to your plug-in's needs. Using the `TextEditor`, you can specify the subclasses of `IDocument` and `ITextViewer` that you use in your plug-in, along with a few other classes discussed later.

To use a text editor in your plug-in, you must specify your subclass of `TextEditor` in the plugin.xml file using the `org.eclipse.ui.editors` extension point, as shown in Listing 2.

Listing 2. Specify the TextEditor subclass

```
<extension  point="org.eclipse.ui.editors">
        <editor
              id="my.ide.editor.MyEditor"
               class="my.ide.ui.MyEditor"
              extensions="c"
              icon="icons/fileIcon.gif"
              name="My Editor"
         />
</extension>
```

Attributes:

- **`id`** -- A unique name used to identify this editor
- **`class`** -- The actual subclass of `org.eclipse.ui.IEditorPart` that you're using in your plug-in
- **`extension`** -- The file extension for file types for which you are using this editor
- **`icon`** -- The icon used to display this file in various views, such as Navigator and Package Explorer
- **`name`** -- A name for this editor type

The `TextEditor` class is responsible for instantiating and configuring `SourceViewer`. Thus, if you're using some custom subclass of `SourceViewer` or want to do something special while the source viewer is being initialized, you must override the `createSourceViewer()` method in your subclass of `TextEditor`. Here's how:

1. Create a subclass of `TextEditor`.
2. Override the `createSourceViewerMethod()` method.
3. Create your custom instance of `SourceViewer`.
4. Call `getSourceViewerDecorationSupport` to ensure that decoration support has been created and configured for the viewer.
5. Write any custom configuration code you want.
6. Return the instance of `SourceViewer`.

Listing 3 demonstrate this process.

Listing 3. Override the createSourceViewer() method

```
public class MyEditor extends TextEditor
{
   @Override
    protected ISourceViewer createSourceViewer(Composite parent,  IVerticalRuler ruler,
        int styles)
    {
        ISourceViewer viewer = new MySourceViewer(parent, ruler,getOverviewRuler(),
                isOverviewRulerVisible(), styles,this);

        // ensure decoration support has been created and configured.
        getSourceViewerDecorationSupport(viewer);

        //do any custom stuff
        ...


        return viewer;
    }
}
```

The document provider

Now that you have seen that `SourceViewer` is instantiated inside the text editor, you can move on to creation of the document.

The document is created in a separate class of its own called the `Document Provider` class. The document provider creates a bridge between a file on a disk and its representation as a document in the memory. It handles tasks such as loading files from the disk and saving them, telling whether the document has changed since it was loaded, etc. Notice the asterisk (*), which appears in the tab of an editor window when you modify any text inside it: The document provider generates that notification.

Eclipse defines the document provider in the `org.eclipse.ui.texteditor.IDocumentProvider` interface. Several extension interfaces are also defined for it, with one for almost every release. But fear not -- the implementation is already done in the subclasses Eclipse provides. Your job is simply to extend a subclass and override two of its methods -- namely, `createDocument(Object input)`, which creates and configures a document for the specific input, and `createEmptyDocument()`, which creates an empty document. Although the document provider contains many more methods, 99 percent of the time, you will override only these two.

To create our own document provider class for your IDE:

1. Create a subclass of `FileDocumentProvider`.
2. Override the `createDocument()` method.
3. Within the method, call `super.createDocument` to create the document.
4. Put code to attach a document partitioner to the document. (Document partitioners and the code to attach them to the document are discussed later in the tutorial.)
5. Return the `Document`.
6. Override the `createEmptyDocument()` method.
7. Within that method, return a new instance of the subclass of `Document` used in your code -- for example, `return new MyDocument();`.

Listing 4 demonstrates this process.

Listing 4. Create a document provider class

```
public class MyDocumentProvider extends FileDocumentProvider
```

```
{

    @Override
    protected IDocument createDocument(Object element) throws CoreException
    {
        IDocument document = super.createDocument(element);

    //we will look at document partitions later in this tutorial
        if (document instanceof IDocumentExtension3)
        {
    IDocumentExtension3 extension3= (IDocumentExtension3) document;
    IDocumentPartitioner partitioner=
        new FastPartitioner(MyPlugin.getDefault().getMyPartitionScanner(),
        MyPartitionScanner.PARTITION_TYPES);
    extension3.setDocumentPartitioner(MyPlugin.MY_PARTITIONING, partitioner);
    partitioner.connect(document);
        }


        return document;
    }


    @Override
    protected IDocument createEmptyDocument() {
        return new MyDocument();
    }

}
```

Now, the question arises: Who creates the document provider itself? The `TextEditor`, which is responsible for creating the source viewer and everything related to the text editor. Therefore, it should create the document provider, too.

To use your subclass of `DocumentProvider`, override the constructor of `TextEditor` in your `TextEditor` subclass and call the `setDocumentProvider()` method with an instance of your document provider. Listing 5 shows this construction.

Listing 5. Override the TextEditor constructor

```


public class MyEditor extends TextEditor
{
    public MyEditor ()
    {
        super();
        setDocumentProvider(new MyDocumentProvider());
    }
}
```

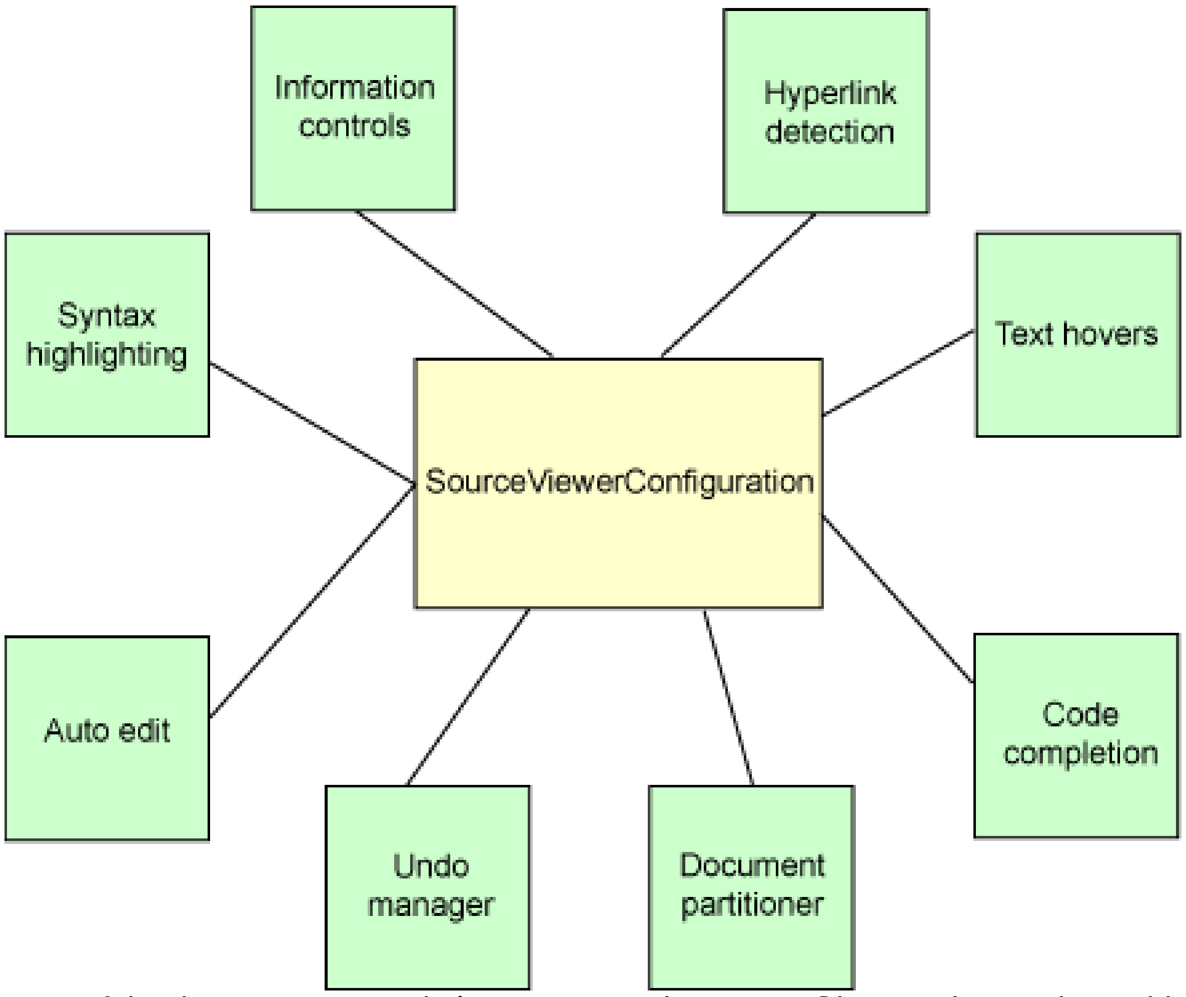
Section 3. Source viewer configuration

The `SourceViewerConfiguration` class is where the rest of the functionality comes together. The `SourceViewerConfiguration` class provides methods to access most of the UI-related helper objects the Eclipse framework provides and that would be useful while creating the editor. Of course, you must create and configure these objects.

Configure your source viewer

Think of the `SourceViewerConfiguration` class as a sort of control panel or central hub for your editor, in which you can plug in all the custom UI-related behavior for your editor, such as syntax highlighting and text hovering, and individually configure them, as shown in Figure 1. The functionality you don't want to customize you can simply leave at its default setting.

Figure 1. SourceViewerConfiguration acts as central hub for UI-related functionality



Most of the time, you must subclass `SourceViewerConfiguration` and override the methods for the behavior you want to implement.

The `SourceViewerConfiguration` class is initialized in `TextEditor`, so if you have a custom subclass of it, you must override the `initializeEditor()` method in your plug-in's `TextEditor` subclass and call `setSourceViewerConfiguration` with an instance of your `SourceViewerConfiguration` subclass, as shown in Listing 6.

Listing 6. Override the initializeEditor() method

```
@Override
protected void initializeEditor() {

    super.initializeEditor();
    setSourceViewerConfiguration(new MySourceViewerConfiguration());
    ...
}
```

Section 4. Document partitions

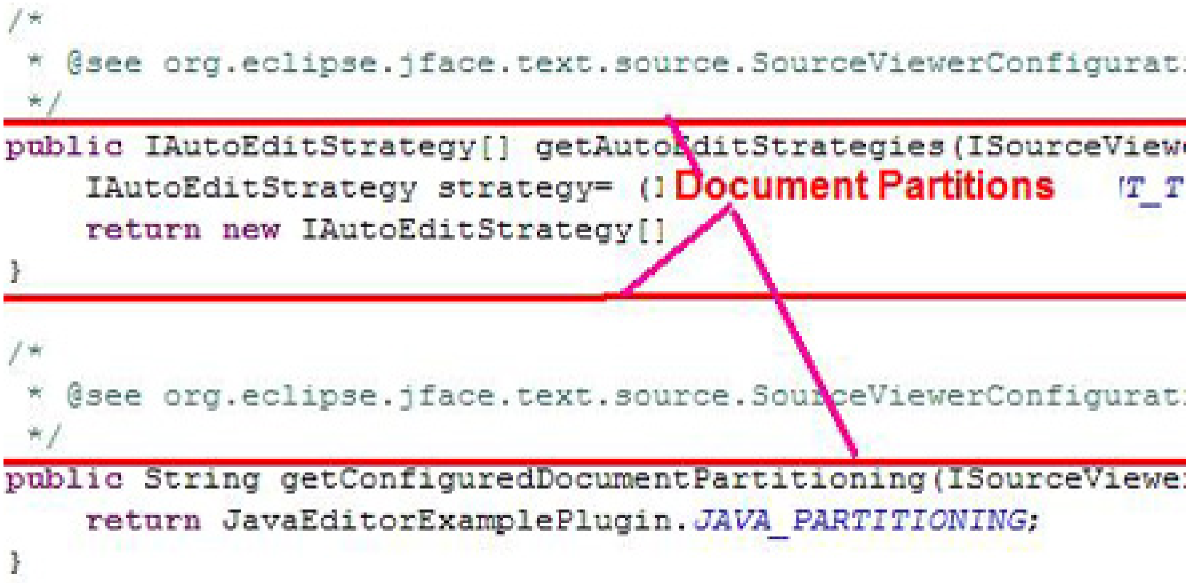
Partition types

Eclipse allows documents to be partitioned into separate *content types*. Examples of content types include:

- Strings
- Single-line comments
- Multiline comments

Figure 2 marks the multiline and Java content partitions in a document.

Figure 2. Partitions in a document



```
/*
 * @see org.eclipse.jface.text.source.SourceViewerConfigurat
 */
public IAutoEditStrategy[] getAutoEditStrategies(ISourceView
    IAutoEditStrategy strategy= (                          T_T
    return new IAutoEditStrategy[]
}

/*
 * @see org.eclipse.jface.text.source.SourceViewerConfigurat
 */
public String getConfiguredDocumentPartitioning(ISourceViewe
    return JavaEditorExamplePlugin.JAVA_PARTITIONING;
}
```

These partitions allow the editor to behave differently, depending on the various content types. For example, you may not want code completion within strings or comments, you may not want to highlight keywords that appear within comments, or you may want to format such keywords differently.

Rule-based partitioning

Eclipse provides simple rule-based scanners you can use to partition a document easily into some of the common content types, such as single-line comments, multiline comments, strings, and character constants. To define your own partition scanner, follow this process:

1. Create a subclass of `org.eclipse.jface.text.rules.RuleBasedPartitionScanner`
2. Define string constants in the class for different partition types
3. Override the default constructor, and define the various rules

Listing 7 provides an example of this process.

Listing 7. Define a custom partition scanner

```
public class MyPartitionScanner extends RuleBasedPartitionScanner {

    //string constants for different partition types
    public final static String MULTILINE_COMMENT= "multiline_comment";
    public final static String SINGLELINE_COMMENT= "singleline_comment";
    public final static String STRING ="string";
    public final static String[] PARTITION_TYPES = new String[] {MULTILINE_COMMENT,
        SINGLELINE_COMMENT, STRING};

    /**
     * Creates the partitioner and sets up the appropriate rules.
     */
    public MyPartitionScanner() {
        super();


        IToken multilinecomment= new Token(MULTILINE_COMMENT);
        IToken singlelinecomment= new Token(SINGLELINE_COMMENT);
        IToken string = new Token(STRING);

        List rules= new ArrayList();

        // Add rule for single line comments.
        rules.add(new EndOfLineRule("//", singlelinecomment));

        // Add rule for strings and character constants.
        rules.add(new SingleLineRule("\", "\", string, '\\'));
        rules.add(new SingleLineRule("'", "'", string, '\\'));


        // Add rules for multi-line comments and javadoc.
        rules.add(new MultiLineRule("/*", "*/", multilinecomment, (char) 0, true));

        IPredicateRule[] result= new IPredicateRule[rules.size()];
        rules.toArray(result);
        setPredicateRules(result);
    }
}
```

This example defines rules for single-line and multiline comments and strings. But how do you connect this scanner to the document?

Connect your scanner to your document

Remember the `DocumentProvider` class? Here is its code again.

Listing 8. The DocumentProvider class

```
    @Override
    protected IDocument createDocument(Object element) throws CoreException
    {
        IDocument document = super.createDocument(element);

    //we will look at document partitions later in this tutorial
        if (document instanceof IDocumentExtension3)
        {
      IDocumentExtension3 extension3= (IDocumentExtension3) document;
      IDocumentPartitioner partitioner=
        new FastPartitioner(MyPlugin.getDefault().getMyPartitionScanner(),
        MyPartitionScanner.PARTITION_TYPES);
      extension3.setDocumentPartitioner(MyPlugin.MY_PARTITIONING, partitioner);
      partitioner.connect(document);
        }


        return document;
    }
```

Here, you instantiate the `FastPartitioner` class, which takes a partition scanner and an array of partition types in its constructor. In the next line, you must pass in a string that will serve as an ID for this partitioner. Listing 9 shows how this string is usually defined in the plug-in class.

Listing 9. Define the string in the plug-in class

```
public class MyPlugin extends AbstractUIPlugin {
    public final static String MY_PARTITIONING = "___my__partitioning____";

    private MyPartitionScanner fPartitionScanner;

    public MyPartitionScanner getMyPartitionScanner() {
        if (fPartitionScanner == null)
            fPartitionScanner= new MyPartitionScanner();
        return fPartitionScanner;
    }
}
```

Override the methods in your source viewer

You must also override methods in the `SourceViewerConfiguration` class to tell it about the custom partitioning and content types. Listing 10 shows such an override.

Listing 10. Override methods to identify partitioning

```
@Override
public String getConfiguredDocumentPartitioning(ISourceViewer sourceViewer) {
        return MyPlugin.MY_PARTITIONING;
    }

@Override
public String[] getConfiguredContentTypes(ISourceViewer sourceViewer) {
  return new String[] { IDocument.DEFAULT_CONTENT_TYPE, MyPartitionScanner.
    MULTILINE_COMMENT ,
    MyPartitionScanner. SINGLELINE_COMMENT, MyPartitionScanner. STRING };
    }
```

This way, your partition scanner is connected to the document and takes care of automatically dividing the document into partitions while you're typing.

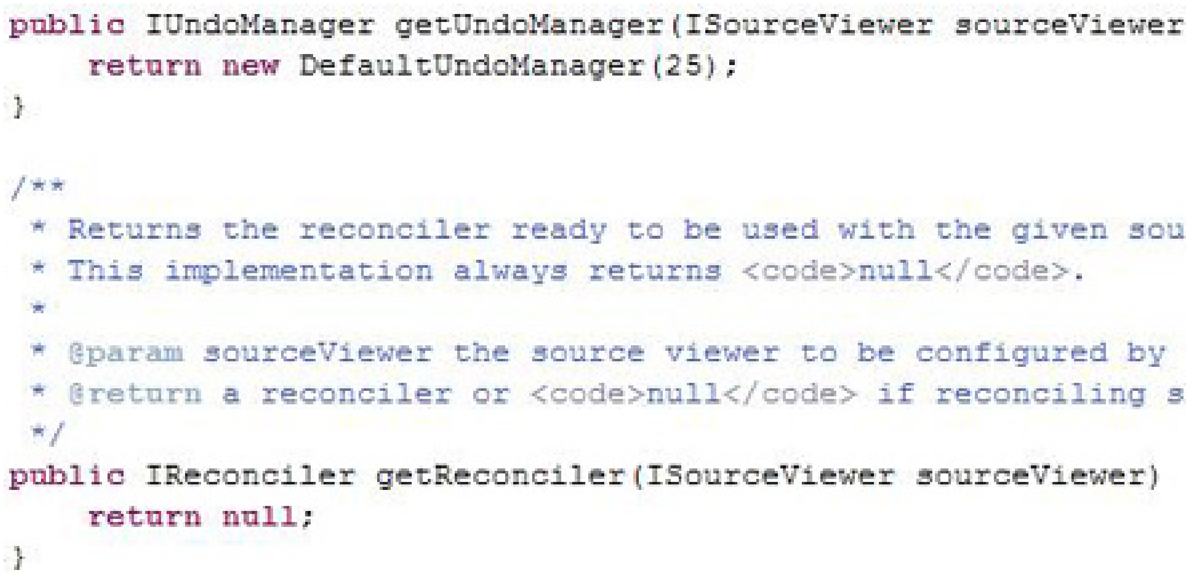
Section 5. Syntax highlighting

Let's move on to the more interesting part: syntax highlighting.

The presentation reconciler

For syntax highlighting, Eclipse provides a presentation reconciler, which divides the document into a set of tokens that define the background and foreground color, along with the font style for a section of text, as shown in Figure 3. Although this may sound the same as the document partitions described earlier, it is, in fact, quite different.

Figure 3. Syntax highlighting in Eclipse text editors



```
public IUndoManager getUndoManager(ISourceViewer sourceViewer
    return new DefaultUndoManager(25);
}

/**
 * Returns the reconciler ready to be used with the given sou
 * This implementation always returns <code>null</code>.
 *
 * @param sourceViewer the source viewer to be configured by
 * @return a reconciler or <code>null</code> if reconciling s
 */
public IReconciler getReconciler(ISourceViewer sourceViewer)
    return null;
}
```

The document partitions define the content types within a document, while the presentation reconciler divided those individual content types into tokens for sets of characters that share the same color and font style. For example, `while` may represent a keyword token for Java content, but it won't be represented in the same way within a multiline comment content.

The presentation of the document must be maintained while the user continues to modify it. This is done through the use of a damager and a repairer. A *damager* takes the description of how the document was modified and returns a description of regions of the document that must be updated. For example, if a user types `/*`, all the text in the document up to `*/` must be repaired. The *repairer* takes care of actually repairing the region of the document that the damager output by using the rules for dividing the region into tokens and the color and font information associated with the tokens.

Although this process may sound complicated, the Eclipse text framework does most of the work for you. Here's what you must do.

Step 1: Define a class for color values

Define a class that will hold all the color values for the tokens, as shown in Listing 11.

Listing 11. Define a class for color values

```
public class MyColorProvider {

    public static final RGB MULTI_LINE_COMMENT= new RGB(128, 0, 0);
    public static final RGB SINGLE_LINE_COMMENT= new RGB(128, 128, 0);
    public static final RGB KEYWORD= new RGB(0, 0, 128);
    public static final RGB TYPE= new RGB(0, 0, 128);
    public static final RGB STRING= new RGB(0, 128, 0);
    public static final RGB DEFAULT= new RGB(0, 0, 0);


    protected Map fColorTable= new HashMap(10);

    public void dispose() {
        Iterator e= fColorTable.values().iterator();
        while (e.hasNext())
            ((Color) e.next()).dispose();
    }


    public Color getColor(RGB rgb) {
        Color color= (Color) fColorTable.get(rgb);
        if (color == null) {
            color= new Color(Display.getCurrent(), rgb);
            fColorTable.put(rgb, color);
        }
        return color;
    }
}
```

Step 2: Create rules for the document tokens

Next, create a set of rules to describe the various tokens in the document. Listing 12 shows such rules.

Listing 12. Rules for describing document tokens

```
public class MyCodeScanner extends RuleBasedScanner
{

        private static String[] fgKeywords = { "while", "for", "if", "else"};

        public MyCodeScanner(MyColorProvider provider) {
```

```
        IToken keyword= new Token(new
            TextAttribute(provider.getColor(MyColorProvider.KEYWORD)));
        IToken string= new Token(new
            TextAttribute(provider.getColor(MyColorProvider.STRING)));
        IToken comment=
            new Token(new
            TextAttribute(provider.getColor(MyColorProvider.SINGLE_LINE_COMMENT)));
        IToken other=
            new Token(new TextAttribute(provider.getColor(MyColorProvider.DEFAULT)));

        List rules= new ArrayList();

        // Add rule for single line comments.
rules.add(new EndOfLineRule("//", comment));

        // Add rule for strings.
        rules.add(new SingleLineRule("\"", "\"", string, '\\'));
        rules.add(new SingleLineRule("'", "'", string, '\\'));

        // Add generic whitespace rule.
        rules.add(new WhitespaceRule(new MyWhitespaceDetector()));

        // Add word rule for keywords.
        WordRule wordRule= new WordRule(new MyWordDetector(), other);
        for (int i= 0; i < fgKeywords.length; i++)
            wordRule.addWord(fgKeywords[i], keyword);
        rules.add(wordRule);

        IRule[] result= new IRule[rules.size()];
        rules.toArray(result);
        setRules(result);

        }
}
```

Step 3: Add the scanner to a presentation reconciler

Finally, input the scanner to a presentation reconciler in the subclass of `SourceViewerConfiguration`. Specify a `DamagerRepairer` for each partition of the document.

Section 6. Reconciler

Remember the lexer and parser you created in Part 1 of this series to scan and analyze the text in the editor? Now, you will actually run them in the editor.

The Eclipse reconciler

Eclipse provides a framework called the Eclipse *reconciler*, which allows you to run a lexer or parser in a background thread. This thread records all the changes to the text and is invoked periodically.

Note: This reconciler is not the same as the presentation reconciler you worked with earlier.

After you install a reconciler on a text editor, it creates a queue used to record all the changes that occur to the text. Each change is represented by a `DirtyRegion` object, and all these dirty regions are added to a `DirtyRegionQueue`. A `DirtyRegion` object contains the following elements:

- Length of the region
- Offset where the region begins
- Text that was changed
- The content type of the region

This is enough information to run even an incremental parser.

The great thing is that if several edits of the same kind occur sequentially, the reconciler can merge them into a single `DirtyRegion` object. For example, if you are continuously typing into the editor, the reconciler merges the small, individual dirty regions for each character into a single large dirty region, which can significantly speed up the analysis, as you won't need to make several passes.

The IReconciler interface

The reconciler is defined by the `org.eclipse.jface.text.reconciler.IReconciler` interface, which basically contains just one method to access the `IReconcilingStrategy`. The `ReconcilingStrategy` is basically an Eclipse abstraction for accepting any type of reconciler or scanner. It only contains methods that set the document and ask for a "reconcile" passing in the dirty region. Note that the passing of dirty regions is enabled only if you set the reconciler to Incremental mode. Otherwise, the reconciler simply passes in the entire document to you each time. You must implement the `IReconcilingStrategy` interface depending on your parser and lexer objects.

Run the reconciler

Here is how the reconciler works along with the reconciling strategy:

1. Instantiate a background thread.
2. Call the `reconcile()` method on the reconciling strategy with the dirty region or the entire document, depending on whether the reconciler is set to Incremental.
3. Wait for an interval of time (say, 500 microseconds [ms]).
4. Return to Step 2.

Listing 13 provides sample code to show how to instantiate and install a reconciler on a source viewer. Again, you must override a method in the `SourceViewerConfiguration` class. For this example, you use the `MonoReconciler` implementation of `IReconciler`, which is what you use most of the time, anyway, unless you have more than one reconciling strategy (in which case, you must use `org.eclipse.jface.text.reconciler.Reconciler`).

Listing 13. Instantiate and install the reconciler

```
@Override
public IReconciler getReconciler(ISourceViewer sourceViewer) {

        MonoReconciler reconciler = new MonoReconciler(new MyReconcilingStrategy(),true);
        reconciler.install(sourceViewer);
        return reconciler;

}
```

Section 7. Undo Manager

Although in most cases you won't need to implement a custom Undo Manager, it's useful to know how it works.

Basic Undo Manager functioning

The Undo Manager is defined by the `org.eclipse.jface.text.IUndoManager` interface. This interface contains the following methods:

- **`beginCompoundChange/endCompoundChange()`** -- This method tells the Undo Manager that all changes recorded until `endCompoundChange` is called are to be marked as "undo" in one piece, thus essentially accumulating those changes in one big compound change.
- **`connect/disconnect()`** -- This method connects to and disconnects from a text viewer.
- **`undo/redo()`** -- This method performs undo/redo operations on the current operation in its undo/redo stack.
- **`reset()`** -- This method resets the undo stack.
- **`undoable/redoable()`** -- This method reports whether any undo/redo operation can be performed.
- **`setMaximalUndoLevel()`** -- This method sets the maximum length of the undo stack.

Another interface -- `IUndoManagerExtension` -- which has been defined since Eclipse V3.1, contains a single method: `getUndoContext()`.

An undo context defines the "context" in which the user is currently working. The Eclipse Workbench uses undo contexts to filter the undo/redo stack, so that when a user wants to perform an undo/redo operation, only those operations that are part of the current undo context are available. In other words, a text editor's context is related to the editor itself, while the resource navigator's context is related to the workspace model objects, etc.

These interfaces are implemented by the `org.eclipse.jface.text.DefaultUndoManager` class, which is returned by the `getUndoManager()` method of the `SourceViewerConfiguration` class.

```
public IUndoManager getUndoManager(ISourceViewer sourceViewer) {
        return new DefaultUndoManager(25);
}
```

Although you won't have to create a custom implementation of `IUndoManager` in most cases, you can override this method in your `SourceViewerConfiguration` subclass and return your own implementation. You can also return `null` if you don't want any undo/redo functionality in your editor.

Section 8. Auto edits

Remember the way the statements in the Java editor indent themselves automatically or the ending braces and string quotes appear automatically while typing? You can do that in your editor, as well, with little effort -- thanks to the all-powerful Eclipse text framework.

The Eclipse text framework

In Eclipse developer lingo, this functionality is called *auto edits*. To use auto edits, you must implement the `org.eclipse.jface.text.IAutoEditStrategy` interface, which contains just one method:

```
void
customizeDocumentCommand(IDocument
\
document,
DocumentCommand
command);
```

This is called each time you make a change to the document with the `DocumentCommand` object containing the information about that change. So, adding indents or making any change to the text being changed is as simple as modifying the `DocumentCommand` object. Listing 14 provides a sample implementation of `IAutoEditStrategy`, which automatically puts ending quotation marks for strings and character constants when you type the opening quotation mark and also places the caret between both the quotes.

Listing 14. Enabling auto edit functionality

```
public class MyAutoEditStrategy implements IAutoEditStrategy
{
public void customizeDocumentCommand(IDocument document, DocumentCommand command)
{
        if(command.text.equals("\""))
        {
            command.text = "\"\"";
            configureCommand(command);
        }
        else if(command.text.equals("'"))
        {
            command.text = "''";
            configureCommand(command);
        }
    }


    private void configureCommand(DocumentCommand command)
    {
        //puts the caret between both the quotes

        command.caretOffset = command.offset + 1;
        command.shiftsCaret = false;
    }

}
```

Listing 15 shows a more elaborate example that automatically inserts an ending brace (}), along with the correct indentation of the line.

Listing 15. Automatically insert an ending brace and indentation

```
public void customizeDocumentCommand(IDocument document, DocumentCommand command)
{
        if (command.text.equals("{"))
        {
            int  line = document.getLineOfOffset(command.offset);
            int indent = getIndentOfLine(document, line);
            command.text = "{" + "\r\n" + indent + "}";
            configureCommand(command);
        }
    }


    public static  int findEndOfWhiteSpace(IDocument document, int offset, int end)
        throws BadLocationException
    {
           while (offset < end) {
               char c= document.getChar(offset);
               if (c != ' ' & c !=  '\t') {
                   return offset;
               }
               offset++;
           }
           return end;
    }

 public static String getIndentOfLine(IDocument document, int line)
            throws BadLocationException
    {
        if (line > -1)
        {
            int start = document.getLineOffset(line);
            int end = start + document.getLineLength(line) - 1;
            int whiteend = findEndOfWhiteSpace(document, start, end);
            return document.get(start, whiteend - start);
        }
        else
        {
            return '';
        }
    }
```

This is how the code works:

1. Determine whether the text being added is an opening brace ({). If it is, go to Step 2.
2. Retrieve the line containing the offset of the document being modified.
3. Pass it to the `getIndentOfLine()` method.
4. Get the indent.
5. Add the indent, along with the closing brace to the **text** field of the `DocumentCommand` object.

The `getIndentOfLine()` method works as follows:

1. Retrieve the beginning and ending offset of the line.
2. Send it to the `findEndOfWhiteSpace()` method.
3. The `findEndOfWhiteSpace()` method returns the offset of first character that is *not* a whitespace, thus effectively returning the indent of the line.
4. Return the indent.

To use auto edits, you must override yet another method in the `SourceViewerConfiguration` class and set the strategies for each of your document content types, as shown in Listing 16.

Listing 16. Set overrides for the SourceViewerConfiguration class

```
public IAutoEditStrategy[] getAutoEditStrategies(ISourceViewer sourceViewer,
    String contentType) {
        IAutoEditStrategy strategy= (IDocument.DEFAULT_CONTENT_TYPE.equals(contentType)
        ? new MyAutoIndentStrategy() : new DefaultIndentLineAutoEditStrategy());
        return new IAutoEditStrategy[] { strategy };
    }
```

Eclipse provides the `DefaultIndentLineAutoEditStrategy` class to catch all line breaks and insert an indent on new lines that are the same as the lines before them.

Section 9. The annotation model

Learn how to use the Eclipse annotation model, as well as the annotations in various rulers.

Using the Eclipse annotation model

You use the annotation model, as its name suggests, to mark ranges of text in a document. All the errors, warnings, and folding **+/-** buttons you see inside the Java editor of JDT are simply visual representations of annotations. Because you will be using these elements in your editor, too, it's useful to know about the Eclipse annotation model.

Eclipse defines the annotation model of a document through the `org.eclipse.jface.text.IAnnotationModel` interface. The interface contains the following methods:

- **`addAnnotation/removeAnnotation()`** -- Use this method to add or remove annotations. Annotations are added along with a *position* in the document they annotate.
- **`connect/disconnect()`** -- Use this method to connect the annotation model to or disconnect it from the document.
- **`getPosition()`** -- This method retrieves the position associated with a specific annotation.
- **`getAnnotationIterator()`** -- Use this method to gain access to all the annotations that this model manages.
- **`add/removeAnnotationModelListeners()`** -- Use this method to add or remove annotation model listeners. The listeners are notified whenever the annotation model is modified.

As you can see, the annotation model interface is quite simple, containing methods for adding and removing text. You do not have to worry about implementing this interface because Eclipse does it for you.

Eclipse defines the implantation of the annotation model in the `ResourceMarkerAnnotationModel` class, which uses the concepts of markers. *Markers* are assigned an image and a range of text to monitor. Thus, the icons for errors, warnings, and those squiggly lines you see in the editor are actually defined by markers. You can change how these markers appear by going to **Window > Preferences > General > Editors > Text Editors > Annotations** in the Eclipse Workbench.

The annotation model is attached to the `Document` of the `SourceViewer` when the `TextEditor` initializes the `SourceViewer`. Listing 17 shows a snippet from the `AbstractTextEditor`, which demonstrates how this process works.

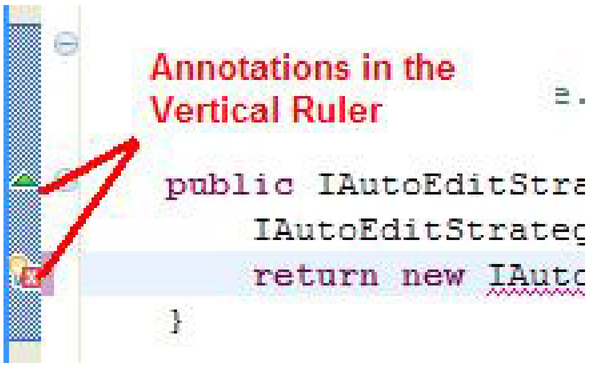
Listing 17. Excerpt from the AbstractTextEditor

```
private void initializeSourceViewer(IEditorInput input) {

    IAnnotationModel model= getDocumentProvider().getAnnotationModel(input);
    IDocument document= getDocumentProvider().getDocument(input);

    if (document != null) {
        fSourceViewer.setDocument(document, model); //put the model
        ...

}
```

Vertical ruler association

When the code has run, it's now the `SourceViewer`'s job to associate the annotation model with the vertical ruler, as shown in Figure 4.

Figure 4. Annotations in the vertical ruler



```
public IAutoEditStra
    IAutoEditStrateg
    return new IAutc
}
```

Listing 18 shows how the process works.

Listing 18. Associate the annotation model with the vertical ruler

```
public void setDocument(IDocument document, IAnnotationModel annotationModel) {
    ...
    // create a visual annotation model from the supplied model
    ...

    if (fVerticalRuler != null)
        fVerticalRuler.setModel(fVisualAnnotationModel);
}
```

Listing 19 shows how the text editor creates the vertical ruler itself.

Listing 19. The text editor creates the vertical ruler

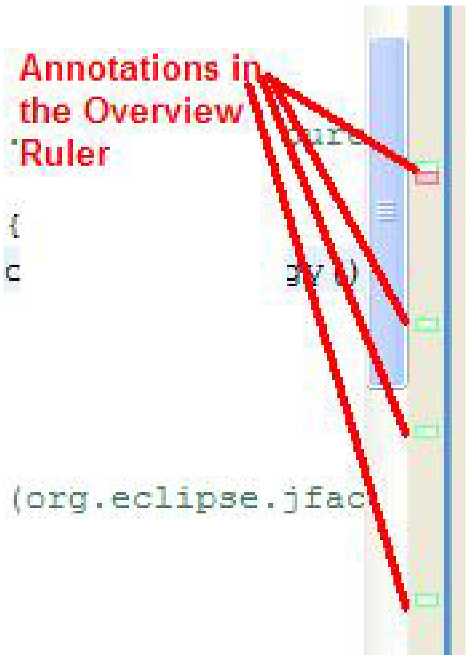
```
protected IVerticalRuler createVerticalRuler() {
    CompositeRuler ruler= new CompositeRuler();
    ruler.addDecorator(0, new AnnotationRulerColumn(VERTICAL_RULER_WIDTH));
    if (isLineNumberRulerVisible())
        ruler.addDecorator(1, createLineNumberRulerColumn());
    return ruler;
}
```

Thus, the `AnnotationRulerColumn`, which is added to the ruler, handles the display of annotation images at proper locations on the ruler.

The overview ruler

The overview ruler appears on the right side of the editor and displays the annotations for the entire document, as shown in Figure 5, so the user can easily navigate to it. The types of annotations shown in it are determined by the `SourceViewerDecorationSupport` class.

Figure 5. Annotations in the overview ruler

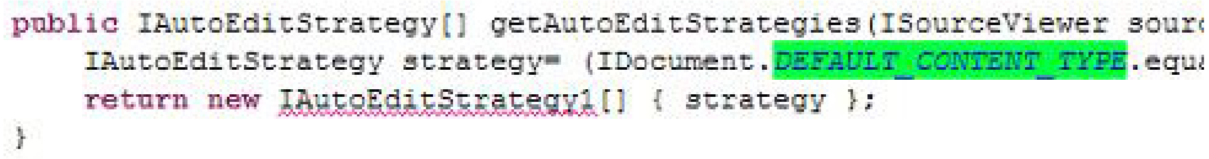


The constructor for the overview ruler takes an instance of `IAnnotationAccess`, which provides information about a particular annotation such as its type and the color in which it is displayed.

Text annotations

The source viewer shows annotations in text using squiggly lines or different background colors to highlight that portion of text, as shown in Figure 6.

Figure 6. Annotations in the text editor



```
public IAutoEditStrategy[] getAutoEditStrategies(ISourceViewer sour
    IAutoEditStrategy strategy= (IDocument.DEFAULT_CONTENT_TYPE.equ
    return new IAutoEditStrategy1[] { strategy };
}
```

The preferences for this functionality are also handled by the `SourceViewerDecorationSupport` class. Listing 20 shows how the `TextEditor` along with the `SourceViewerDecorationSupport` create the

`SourceViewer`.

Listing 20. Create the SourceViewer

```
protected ISourceViewer createSourceViewer(Composite parent, IVerticalRuler ruler,
    int styles) {

    ...
    ISourceViewer sourceViewer= new SourceViewer(parent, ruler, fOverviewRuler,
        isOverviewRulerVisible(), styles);
    fSourceViewerDecorationSupport= new SourceViewerDecorationSupport(sourceViewer,
        fOverviewRuler, fAnnotationAccess, sharedColors);
    configureSourceViewerDecorationSupport();

    return sourceViewer;
}
```

Section 10. Conclusion

You have learned how to implement some of the UI-related functionality of your IDE from within Eclipse. This tutorial introduced you to the text editor classes and examined the role of the `SourceViewerConfiguration` class as the central hub for installing most of the UI-related functionality. You even saw the way documents are divided into partitions and how the annotation model of a document works. You added syntax highlighting to your editor, along with auto-indenting for braces.

Part 3 of this series introduces more UI objects. It also takes another look at ANTLR Studio and discusses some of the design problems encountered while creating the UI of a commercial-quality IDE.

Have fun experimenting with your own IDE.

Resources

Learn

- "Folding in Eclipse Text Editors" explains this process.
- "Getting Your Feet Wet with the SWT StyledText Widget" introduces the SWT `StyledText` widget.
- "Into the Deep End of the SWT StyledText Widget" shows how to customize the SWT `StyledText` widget.
- Take a look at the Eclipse Platform Plug-in Developer Guide for help with developing Eclipse plug-ins.
- Learn more about the Eclipse Foundation and its many projects.
- For an excellent introduction, check out "Getting started with the Eclipse Platform."
- Expand your Eclipse skills by visiting IBM developerWorks' Eclipse project resources.
- Browse all of the Eclipse content on developerWorks.
- 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.
- Stay current with developerWorks technical events and webcasts.
- To listen to interesting interviews and discussions for developers, be sure to check out developerWorks podcasts.

Get products and technologies

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

Discuss

- The Eclipse newsgroups has many resources for people interested in using and extending Eclipse.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Prashant Deva
Prashant Deva is the founder of Placid Systems and the author of the ANTLR Studio plug-in for Eclipse. He also provides consulting related to ANTLR and Eclipse plug-in

development. He has written several articles related to ANTLR and Eclipse plug-ins, and he frequently contributes ideas and bug reports to Eclipse development teams. He is currently busy creating the next great developer tool.