
Create a commercial-quality Eclipse IDE, Part 3: Fine-tune the UI

Skill Level: Intermediate

[Prashant Deva \(pdeva@placidsystems.com\)](mailto:pdeva@placidsystems.com)

Founder

Placid Systems

24 Oct 2006

This tutorial -- the final installment in this "[Create a commercial-quality Eclipse IDE](#)" tutorial series about integrated development environment (IDE) design -- shows how to fine-tune the UI of your IDE. It shows how to use additional elements in Eclipse to enhance your editor as well as demonstrates the differences between commercial-quality and amateur IDEs.

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 language or your own programming language. 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 problems associated with designing and implementing them.

This series uses the ANTLR Studio and Eclipse Java™ Development Tools (JDT) IDEs as case studies and examines their internals to help you understand what it takes to create a highly 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](#) introduces the architecture of an IDE and shows how to create the IDE's core layer. [Part 2](#) shows how to implement the UI component of your IDE. In this final

installment, you discover additional UI elements Eclipse provides to enhance your editor. This tutorial also discusses the differences between a commercial-quality IDE and an amateur one. It looks at the internals of the ANTLR Studio and Eclipse JDT IDEs to show how you can provide features that will differentiate your IDE from the rest.

Prerequisites

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

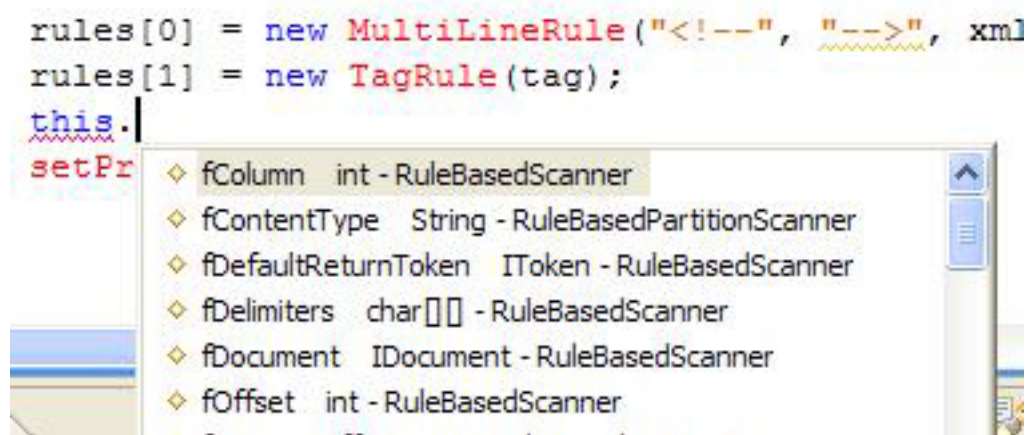
System requirements

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. Using code completion

No doubt, you use code completion when coding in your favorite IDEs. When you press **Ctrl+Space** in an editor, you expect to see a drop-down list containing the names of all the identifiers you can type in that location (see Figure 1). This functionality is no longer a feature but a standard among today's IDEs. If your IDE does nothing when a programmer presses **Ctrl+Space** on his or her keyboard, your IDE stands a good chance of being uninstalled.

Figure 1. Code completion in action within the Eclipse JDT



Now let's see how to actually implement this functionality in your IDE.

Implement code completion in your IDE

Eclipse developers seem to have assumed that nearly every editor of every IDE will be implementing code completion functionality. As a result, they provided classes to make adding the functionality easy. To provide code completion functionality in your editor, you must know the following two interfaces:

- `IContentAssistant`
- `IContentAssistProcessor`

The `Content Assistant` is the main facade for code completion. If your editor shows a list of completions at a specific moment (for example, when the user presses **Ctrl+Space**), it would call a method on an object that implements the `IContentAssistant` interface. Supporting classes handle the actual graphical user interface (GUI) of the pop-up list and the completions themselves. In fact, the `IContentAssistant` interface doesn't contain anything other than methods to show completions or context information and to install and uninstall itself from the editor.

The `IContentAssistProcessor` interface is responsible for calculating the completion proposals at a point in the editor. It also defines the characters on which it should auto-activate, such as the period (.) character, which, when pressed in the editor, shows the list of members of a type. You can define a different `IContentAssistProcessor` for each partition type of a document. For example, when you press **Ctrl+Space** within a Javadoc comment, you expect different completions from when you press **Ctrl+Space** within Java code.

Note that the `IContentAssistProcessor` is not responsible for implementing the GUI portion of the completions. That portion is handled by other classes within Eclipse, and you don't really need bother with it.

IContentAssistant

The `IContentAssistant` interface is relatively simple. It contains the following components:

install()/uninstall()

Methods for installing and uninstalling the content assistant on the source viewer

showPossibleCompletions

Shows the drop-down list containing the completions

showContextInformation

Shows the pop-up menu that displays the context information at a location

getContentAssistProcessor

Used to call the content assist processor for a particular content type in the document

Many extensions to the `IContentAssistant` are available for actions such as attaching listeners, repeated invocation mode, and common prefixes. However, you

don't need to worry about these, either, because Eclipse provides a class that implements them all for you: `org.eclipse.jface.text.contentassist.ContentAssistant`. You simply use the class in your IDE.

The only time you really encounter this class is in your `SourceViewerConfiguration` subclass to configure code completion. To configure code completion for your editor:

1. Override the `getContentAssistant()` method of `SourceViewerConfiguration`.
2. Create a new instance of the `ContentAssistant` class.
3. Set the document partitioning in the content assistant object.
4. Set the various content assist processors for the different document partition types.
5. If necessary, enable auto-activation. You can also set the delay in milliseconds after auto-activation is triggered.
6. If necessary, further customize the completion menu by setting elements such as the background color of the drop-down list (called the *Proposal Selector* in Eclipse terminology).

Listing 1 shows the code for performing these steps.

Listing 1. Enable code completion in your IDE editor

```
@Override
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer)
{
    ContentAssistant assistant= new ContentAssistant();

    assistant.setDocumentPartitioning(get\
ConfiguredDocumentPartitioning(sourceViewer));
    assistant.setContentAssistProcessor(new MyContentAssistProcessor(),
        IDocument.DEFAULT_CONTENT_TYPE);

    assistant.setAutoActivationDelay(0);
    assistant.enableAutoActivation(true);

    assistant.setProposalSelectorBackground(Display.getDefault().
        getSystemColor(SWT.COLOR_WHITE));

    return assistant;
}
```

IContentAssistProcessor

The only real work you must do while implementing code completion functionality is to implement the `IContentAssistProcessor` interface. Methods of the `IContentAssistProcessor` interface include:

`computeCompletionProposals()`

Computes the set of completions based on the offset provided

computeContextInformation()

Computes the context information based on the offset provided

getCompletionProposal/contextInformation

AutoActivationCharacters()

Returns the characters that, when a user types them in the editor, automatically trigger the completion/context information to appear

Other methods exist, but these are all you need for now. In fact, because the discussion focuses only on code completion right now, only the `computeCompletionProposals` and `getCompletionProposalAutoActivationCharacters` interfaces are of interest. And most of the logic behind code completion functionality lies in the `computeCompletionProposals()` method. Here, you must query your parse tree to see what identifiers are valid at the location.

The `computeCompletionProposals()` method returns an array of `ICompletionProposal` objects. An `ICompletionProposal` represents each completion listed in the auto-complete list. The array defines items such as the text to insert, how to insert it, and the image and text to display in the completion menu.

Listing 2 shows a sample class that implements these two methods.

Listing 2. Class for implementing the `computeCompletionProposals()` and `computeContextInformation()` methods

```
public class ASCompletionProcessor implements IContentAssistProcessor
{
    public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer,
        int offset)
    {
        List<ICompletionProposal> proposals =
            new ArrayList<ICompletionProposal>();

        //compute proposals at offset

        return proposals.toArray(new ICompletionProposal[proposals.size()]);
    }

    public char[] getCompletionProposalAutoActivationCharacters()
    {
        return new char[] {','};
    }
}
```

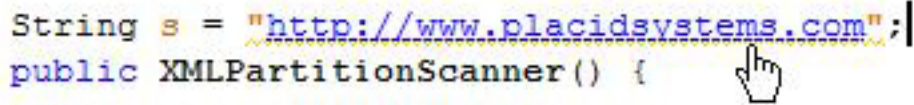
Section 3. Using hyperlink detectors

Here is a trick you may not know: If you type a URL such as `http://www.placidsystems.com` within the Eclipse Java editor and hover your cursor

over the URL's text while pressing and holding the **Ctrl** key, the text turns into a hyperlink that you can click (see Figure 2). This behavior occurs as a result of hyperlink detection functionality within the editor.

Figure 2. Hyperlink detector functionality makes text behave as links

```
String s = "http://www.placidsystems.com";|
public XMLPartitionScanner() {
```



```
IToken xmlComment = new Token(XML.COMMENT);
```

You can use hyperlink detection functionality for tasks other than simply detecting Web-site URLs. For example, if within a Javadoc in the Java editor, you type text such as `{@link MyClass#myMethod() }` and, while pressing and holding the **Ctrl** key, hover your cursor over the *MyClass* or the *myMethod* text, the text turns into a link that you can click. Clicking the text opens the corresponding type or method in a new editor window. The same happens when you hover over a variable or method name: Clicking the name moves the caret to its declaration.

To add hyperlink detection functionality to your IDE editor, you use two interfaces:

IHyperlinkDetector

Use this interface to detect hyperlinks within the editor.

IHyperlink

This interface represents the hyperlink itself and defines elements such as what should be done when the link is clicked and the text region the link occupies.

The IHyperlinkDetector and IHyperlink interfaces

The `IHyperlinkDetector` interface is simple, defining a single method:

```
IHyperlink[] detectHyperlinks(ITextViewer textViewer, IRegion region,
    boolean canShowMultipleHyperlinks);
```

The arguments stand for:

textViewer

The current text viewer

region

The region of text in which to look for hyperlinks

canShowMultipleHyperlinks

Used to tell whether you can show multiple hyperlinks to the user at the same time (in most cases, you will ignore this value and show a single hyperlink to the user.)

The methods defined in the `IHyperlink` interface are:

`open()`

Opens the hyperlink

`getHyperlinkRegion()`

Returns the region of text that the hyperlink occupies

`getTypeLabel()`

Used to label the hyperlink with a type; used when multiple hyperlinks must be shown

`getHyperlinkText`

Assigns a text to this hyperlink; used when multiple hyperlinks point to the same location

For our purposes, you implement just the `open()` and `getHyperlinkRegion()` methods. The bulk of the logic resides in the `open()` method because that method is responsible for implementing the behavior of the hyperlink.

Eclipse provides built-in `UrlHyperlinkDetector` and `URLHyperlink` classes used to detect URL hyperlinks within your editor. Your editor is configured with the `UrlHyperlinkDetector` by default. Of course, you can add your custom hyperlink detectors, as well. As an example, you'll create a simple hyperlink detector that detects any text inside a pair of angle brackets (<>) and considers it a link to some declaration inside your editor.

Create a simple hyperlink detector

To create a simple hyperlink detector for your editor:

1. Implement the `detectHyperlinks()` method of the `IHyperlinkDetector` class, as shown in Listing 3.

Listing 3. Implement the `detectHyperlinks()` method

```
public class MyHyperlinkDetector implements IHyperlinkDetector {
    public IHyperlink[] detectHyperlinks(ITextViewer textViewer, IRegion region,
        boolean canShowMultipleHyperlinks) {
        IRegion lineInfo;
        String line;
        try {
            lineInfo= document.getLineInformationOfOffset(offset);
            line= document.get(lineInfo.getOffset(), lineInfo.getLength());
        } catch (BadLocationException ex) {
            return null;
        }
        int begin= line.indexOf("<");
        int end = line.indexOf(">")
        if(end<0 || begin<0 || end==begin+1)
            return null;
    }
}
```

```

String text = line.substring(begin+1,end+1);
IRegion region = new Region(lineInfo.getOffset()+begin+1,text.length());

return new IHyperLink[] {new MyHyperLink(region,text);
}
}

```

2. Implement the IHyperLink class.
For this example, implement only the `open()` and `getHyperlinkRegion()` methods. In the `open()` method, simply query the parse tree for the declaration of the identifier and move the caret to that location in the editor, as shown in Listing 4.

Listing 4. Implement the IHyperlink class

```

class MyHyperlink extends IHyperlink {
    private String location;
    private IRegion region;

    public MyHyperlink(IRegion region, String text) {

        this.region= region;
        this.location = text;
    }

    public IRegion getHyperlinkRegion() {
        return region;
    }

    public void open() {
        if(location!=null)
        {
            int offset=MyAST.get().getOffset(location);
            TextEditor editor=getActiveEditor();
            editor.selectAndReveal(offset,0);
            editor.setFocus();
        }
    }

    public String getTypeLabel() {
        return null;
    }

    public String getHyperlinkText() {
        return null;
    }
}

```

3. Override the `getHyperlinkDetectors()` method of Source Viewer Configuration, and return an array containing your hyperlink detector class and the `URLHyperlinkDetector` class. This way, you get the functionality of both hyperlink detectors.

```

@Override
public IHyperlinkDetector[] getHyperlinkDetectors(ISourceViewer sourceViewer) {
    return new IHyperlinkDetector[] { new MyHyperlinkDetector(),
        new URLHyperlinkDetector() };
}

```

Section 4. The double-click strategy

By double-clicking a word in the text editor, you select the entire word. You can change this default behavior and enhance it to something specific to the programming language your editor supports. For example, clicking next to a brace in the Java editor selects everything within the matching pair of braces. To customize this behavior, you must implement the `ITextDoubleClickStrategy` interface.

Implement the `ITextDoubleClickStrategy` interface

This interface is simple and contains just one method: `void doubleClicked(ITextViewer textViewer)`. This method is called each time you double-click the text in the text editor. Theoretically, instead of changing the selection, you could perform some other operation on double-clicking. The only caveat here is that you are not allowed to modify the text in any way.

To see how this works, create a double-click strategy that selects all the text to the end of a line:

1. Create a class that implements the `ITextDoubleClickStrategy` interface and the `doubleClicked()` method such that it selects all text to the end of the line, as shown in Listing 5.

Listing 5. Implement the `ITextDoubleClickStrategy` interface

```
public class MyTextDoubleClickStrategy implements ITextDoubleClickStrategy {
    public void doubleClicked(ITextViewer text) {
        try {

            int startOffset = text.getSelectedRange().x;

            IDocument document= text.getDocument();
            IRegion line= document.getLineInformationOfOffset(position);
            int endOffset = line.getOffset() + line.getLength()

            if (startOffset == endOffset)
                return;

            text.setSelectedRange(startOffset, endOffset - startOffset);
        } catch (BadLocationException x) {
        }
    }
}
```

2. Override the `getDoubleClickStrategy()` method of `SourceViewerConfiguration` and return your double-click strategy, as shown in Listing 6. **Note:** Using the `getDoubleClickStrategy()` method, you can set different double-click strategies for different content

types. Thus, you could have a different strategy for comments, code, strings, etc.

Listing 6. Override the `getDoubleClickStrategy()` method

```
@Override
public ITextDoubleClickStrategy \
getDoubleClickStrategy(ISourceViewer sourceViewer,
    String contentType) {
    return new MyTextDoubleClickStrategy ();
}
```

Section 5. Adding content formatting

To provide your editor the ability to format the code, you must use the content formatting application program interfaces (APIs) Eclipse provides. Following the usual strategy pattern, two interfaces are involved:

- `IContentFormatter`
- `IContentFormattingStrategy`

The `IContentFormatter` acts as a facade and merely chooses the formatting strategy to apply to a given region. The `IContentFormattingStrategy` actually does the formatting. Different partitions can have their own formatting strategy. Eclipse provides implementations of the `IContentFormatter` interface; you must simply implement the `IContentFormattingStrategy` interface.

The `IContentFormatter` interface

Eclipse provides two implementations of the `IContentFormatter` interface:

ContentFormatter

This implementation allows for two modes of operation: *partition-aware* and *partition-unaware*. When you use partition-aware mode, it determines the partitions of the document and applies a different formatting strategy for each one. Partition-unaware mode treats the entire document as one big partition of type `IDocument.DEFAULT_CONTENT_TYPE` and applies a single formatting strategy to the entire document.

MultiPassContentFormatter

The `MultiPassContentFormatter` is so named because it makes two passes of the document, taking in two kinds of formatting strategies:

1. **Master formatting strategy** -- The first pass, used to format the default content type

2. **Slave formatting strategy** -- Subsequent passes, used to format all the other content types

Note: This behavior is different from the `ContentFormatter` class, which takes a different formatting strategy for each content type.

So, which implementation should you choose when? If you want to format every one of your partitions in a different manner, use `ContentFormatter`. If you just want to format the main code and don't much care about the rest of the stuff, use `MultiPassContentFormatter`. The latter is useful in languages such as C, where you want to format the code, but not the comments, strings, etc.

Configure your editor to use `MultiPassContentFormatter`

To configure your editor to use a `MultiPassContentFormatter` interface:

1. Override the `getContentFormatter()` method in your `SourceViewerConfiguration` subclass.
2. Create an instance of `MultiPassContentFormatter`.
3. Set the document partitioning to the `MultiPassContentFormatter` object.
4. Set the master and slave formatting strategies.
5. Return the formatter.

Listing 7 shows this process.

Listing 7. Configure the editor for `MultiPassContentFormatter`

```
@Override
public IContentFormatter getContentFormatter(ISourceViewer sourceViewer) {

    MultiPassContentFormatter formatter=
        new MultiPassContentFormatter( getConfiguredDocumentPartitioning(sourceViewer),
            IDocument.DEFAULT_CONTENT_TYPE);
    formatter.setMasterStrategy(new MasterFormattingStrategy());
    formatter.setSlaveStrategy( new CommentFormattingStrategy(), IMyPartitions.COMMENTS);

    return formatter;
}
```

Section 6. Text hovers and tab widths

Text hovers

You've seen the yellow window that appears when you hover your mouse pointer over a method or class name describing its parameters or showing its documentation (see Figure 3). You can add them to your editor, too.

Figure 3. Text hover showing documentation for a Java interface



Add text hover functionality to your IDE editor

To add text hover functionality to your IDE editor, you must implement the `ITextHover` interface. This interface contains just two methods:

`getHoverRegion(ITextViewer textViewer, int offset)`

Given an offset in the editor, this method computes the region of text that will be used to compute the information for the hover window. For example, to provide a hover for each method, place the cursor inside the method, look at the offset, and return the region containing the entire method, which would then be used to compute the hover text.

`getHoverInfo(ITextViewer textViewer, IRegion hoverRegion)`

Given a region to calculate the information, this method returns a string containing the information to display.

The way in which you implement this interface depends on the kind of information you want to display. Most likely, after getting a "hover region," you will need to look the region up in your Abstract Syntax Tree (AST) to determine what kind of information (if any) can be shown at that particular location.

To configure your source viewer after you have implemented the `ITextHover` interface:

1. Override the `getTextHover()` method in your `SourceViewerConfiguration` subclass.
2. Return the implementation of `ITextHover`. You can return a different implementation for each partition by looking at the `contentType` string passed to the `getTextHover()` method.

The code for this process:

```
@Override
public ITextHover getTextHover(ISourceViewer sourceViewer, String contentType) {
    return new MyTextHover();
}
```

Tab widths

This one is simple: Every time the user presses the **Tab** key in the editor, the caret moves ahead by a certain number of characters. The default is four characters, so when the user presses **Tab**, the editor leaves four spaces blank.

You can change this default by overriding the `getTabWidth()` method of the `SourceViewerConfiguration` subclass and returning an integer that specifies the tab width. The following code configures the editor to use a tab width of six spaces:

```
@Override
public int getTabWidth(ISourceViewer sourceViewer) {
    return 6;
}
```

Section 7. From a good IDE to great IDE

So far, this series has shown the architecture and the core of an IDE. It has also provided an in-depth look at the Eclipse APIs you can use to create the UI for your IDE. Say you're about to create the next great IDE for Ruby or Python, but wait -- there's a catch. If you were to follow exactly what you've learned to this point and implement all of the UI formatting and customization, you'd end up with a good IDE. But you won't end up with a *great* IDE.

Now, I'll let you in on some of the biggest secrets in IDE design. I'll show you how to make your IDE great -- how to make it such that your users will turn into fans and talk about your creation in forums and newsgroups.

You might find me discussing some of the features of ANTLR Studio here and there, but I can't explain how to go about implementing the features unless I explain the features themselves. The section that follows presents features of commercial-quality IDEs and, for some features, compares the approaches that JDT and the ANTLR Studio take to implementing them.

Syntax highlighting

If you were to plainly implement syntax highlighting functionality by using the Eclipse APIs in [Part 2](#), you'll be able to highlight the keywords, comments, and strings in your language easily. The Eclipse APIs allow you to add very basic functionality. However, you would not be able to add highlighting such as you could with JDT or ANTLR Studio.

The JDT approach

Apart from the usual Java keywords and comments, JDT can color elements such as method calls, variable names, field names, and class names (see Figure 4). It can even differentiate between method calls and method declarations, fields and static fields, etc, coloring each one differently.

Figure 4. Normal highlighting in Eclipse APIs and enhanced highlighting in JDT

```
final static String XML_DEF
final static String XML_COM
final static String XML_TAG

XMLPartitionScanner() {
    IToken xmlComment = new Token(
    IToken tag = new Token(XML_TAG)

    IPredicateRule[] rules = new I
    rules[0] = new MultiLineRule("
    rules[1] = new TagRule(tag);

    setPredicateRules(rules);
}
```

Here's how JDT achieves this enhanced functionality:

- It uses the Eclipse API and its own mechanism.
- For keywords such as *int* and *return*, JDT uses the normal Eclipse APIs to color them.
- For elements such as method calls and local variables, JDT runs a background thread that after a certain interval of time, checks its AST and analyzes the modified portions of the document to find out the semantics of that portion of code. It then applies the appropriate color to it.
- This check results in the appearance of certain colorings -- like those for method calls and static fields -- after a short lag, while elements like keywords and comments are highlighted instantly.
- Because the keywords are highlighted instantly anyway and the delay for

highlighting other elements is short, the user doesn't notice it.

The ANTLR Studio approach

ANTLR Studio can differentiate between the various sections of an ANTLR grammar and color each one in a different way. Thus, identifiers that are keywords in that section are painted as keywords only in that section, not in others (see Figure 5). Even comments in one section are painted differently from comments in another.

Figure 5. ANTLR Studio can color different sections of an ANTLR grammar differently

```

WS : ( /* '\r' '\n' can be match
      '\r' in one iteration ;
      handle any flavor of n
      that allows both "\r\n"
      newline is ambiguous. I't
      */
      options {
        generateAmbigwarnings=
      }
      ;
      '\t'
      '\r' '\n' {newlineO;}
      '\r' '\n' {newlineO;}
      '\n' {newlineO;}
      ;
      { $setType(Token.SKIP); }
      ;
COMMENT :
( SL_COMMENT | t:ML_COMMENT { $set

```

The process that ANTLR Studio uses to achieve this enhanced functionality:

- ANTLR Studio runs a proprietary analysis algorithm in the same thread as the editor. The main feature of this analysis is that there is virtually no delay in scanning the damaged portions of the document.
- Running in the same thread results in no delay in syntax coloring, and you still get all the colors you want.

Although this approach is the best from a user point of view (because everything is instantaneous), you must be wary because even the slightest delay in the UI thread can cause a significant lag between the user typing the character and the character's appearance on the screen. Even the slightest lag will make your IDE unusable. So, weigh the benefits against the danger before you put everything in a single thread.

Code completion

What can possibly be done more to code completion, you ask? Well, if you use the Eclipse APIs, you'll be able to add code completion nicely in your editor. But let's see

how commercial IDEs tackle this functionality.

The JDT approach

JDT uses a unique process to implement code completion:

- Its code completion window automatically arranges the members most likely to be used on top, irrespective of their alphabetical position.
- JDT accepts camel-case input, so if you type `NPE` in Eclipse JDT V3.2, then press **Ctrl+Space**, JDT shows a window containing `NullPointerException`.
- JDT can recognize subclasses, so if you type `List a = new`, then press **Ctrl+Space**, JDT shows `ArrayList` at the top (see Figure 6).

Figure 6. JDT shows `Vector` and `ArrayList` classes as possible completions



- JDT rearranges the most frequently used subclass on the top, so that those classes are available as the first selection.
- When you're typing a new class that is not yet in the list of imports and you press **Ctrl+Space**, you can still see that class name available for completion. Selecting the class adds an import for it automatically.

When you're creating the code for this functionality in JDT, keep in mind that you must provide these features yourself. Eclipse won't do it for you.

The ANTLR Studio approach

Like JDT, ANTLR Studio also has a unique approach to code completion functionality:

- ANTLR Studio doesn't require you to press **Ctrl+Space** or any other key combination. Completions appear automatically and usually contain what's needed.
- You needn't define grammar rules before they appear in code completions. This is especially useful because in ANTLR Studio, grammar rules are used before they are defined, (see Figure 7).

Figure 7. ANTLR Studio shows completions for rules even though not declared



Note: The red squiggly line shown below the first reference to the *mexpr* rule shows that it is not declared.

- There is no delay in the editor, even when windows continue to appear while you're typing. Users don't notice any form of slowdown in the editor, even with the huge number of calculations performed on each keystroke.

To achieve its code completion functionality, ANTLR Studio runs the parser in the main UI thread and doesn't need to wait for a background thread to finish. It also uses a fully incremental lexer and parser, which significantly reduces the number of calculations to be performed, allowing it to run the analysis in the UI thread.

Memory requirements

As the number of files in the projects of your IDEs grows, your IDE will require more and more memory and, thus, run more slowly. You cannot keep all the files in memory at all times. You must use a lazy, delayed loading approach to load only what's needed and discard those elements as soon as they're no longer necessary.

ANTLR Studio has only one or two grammar files per project. Most of the time, they have no dependencies with each other. Although ANTLR Studio can read token definitions from external files, and for those situations, the files are loaded lazily and only when needed. They are also monitored for changes at regular intervals.

Speed

If your IDE is slow, your editor lags while you're typing, or projects take ages to load, it will never succeed in the marketplace. You cannot compromise on speed. If such delays occur, take another good look at your design. Try to use an incremental parser and lexer as well as the delta algorithms described in [Part 1](#). Use multiple threads: If your IDE isn't fast, it should at least give the illusion of being fast.

As of this writing, Eclipse V3.2 loads a huge workspace almost instantaneously. It does this by processing files in the background and making heavy use of delayed loading. Although this behavior results in nearly instant loads, it does slow the IDE a bit.

Elements not discussed that you should implement

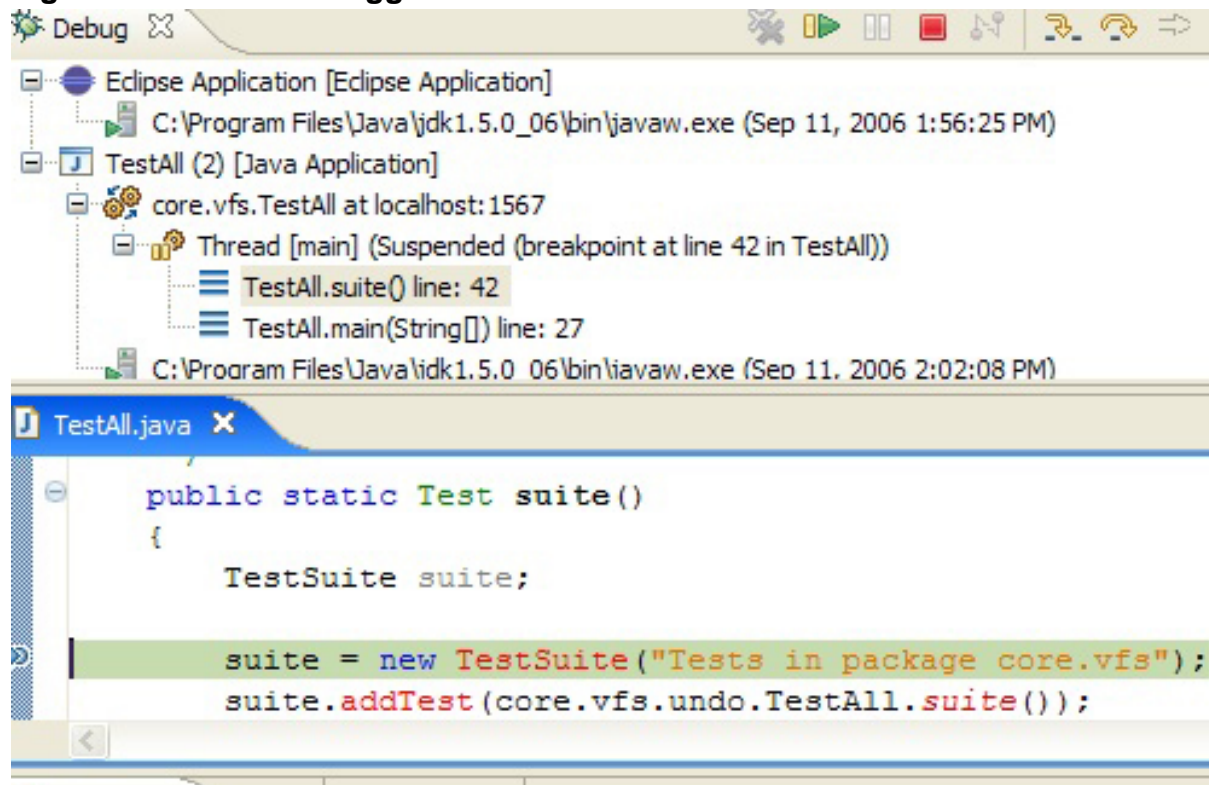
No matter how many installments this tutorial series could be, IDEs are such a huge

topic that there will always be something left to cover. Unfortunately, to code decent IDEs, you must know most of what there is to know about them. Here's a brief summary of the elements this tutorial series didn't cover in detail, but that you should implement in your IDE to make it successful.

Debugging

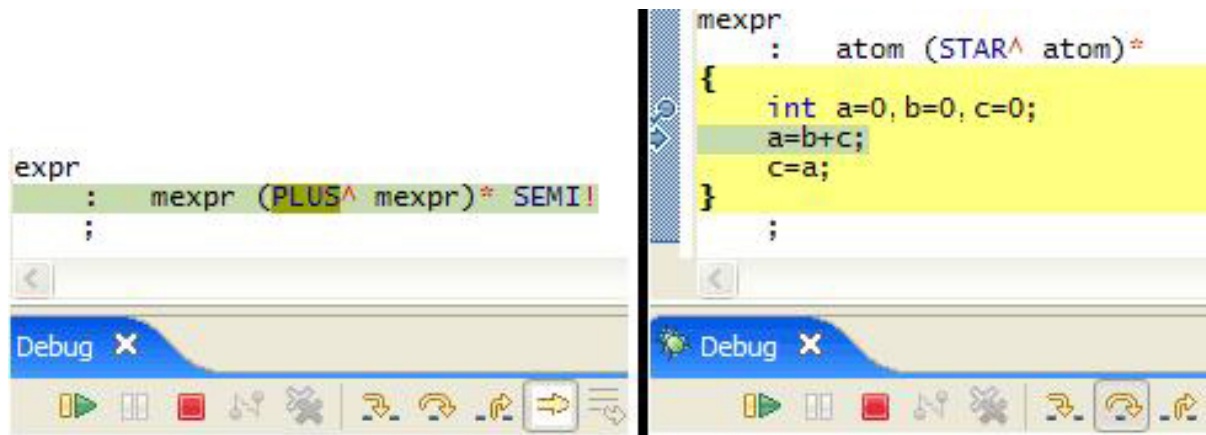
This series didn't discuss debugging technology, but you definitely need a debugger in your IDE if it's to succeed (see Figure 8). The debugger APIs in Eclipse are a huge topic worthy of tutorials of their own. You must also do an immense amount of coding apart from simply using the Eclipse API to implement the debugger for the programming language your IDE supports.

Figure 8. The JDT debugger



In this regard, ANTLR Studio's Fluid Debugger integrates with the Eclipse JDT debugger and provides a seamless debugging experience with the Java code so you don't feel that you're in a separate IDE. Using Fluid Debugger, you can step over rules and Java code embedded in ANTLR grammar files (see Figure 9). It does this using the JSR 45 standard, along with proprietary debugging algorithms.

Figure 9. ANTLR Studio's Fluid Debugger



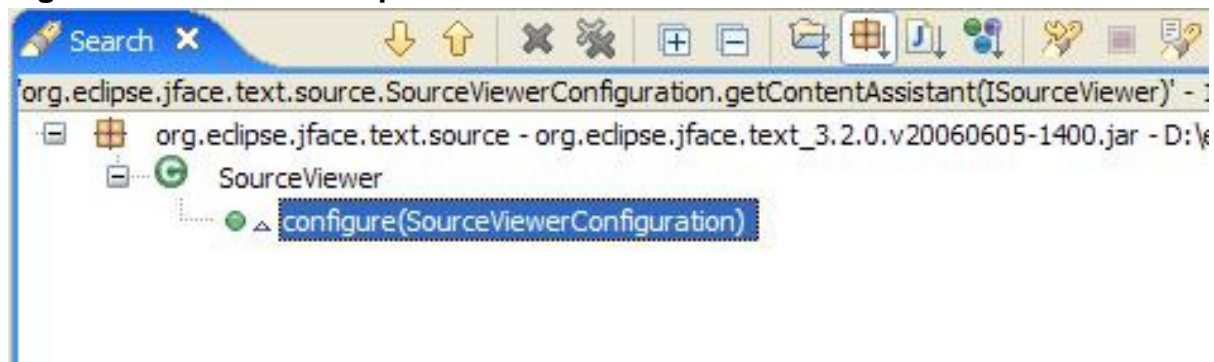
Build system

Your IDE definitely needs a build system so users can get a working executable from the source code. Both JDT and ANTLR Studio compile the code in the background whenever you save a file. Because this behavior is the de-facto standard for Eclipse IDEs and is convenient for users (because there's no extra, explicit build step), I recommend following the same model, instead of providing an explicit **Build** button.

Search

Implement code search in your IDE so users can search elements such as variable references and method calls. In large projects, this functionality is invaluable, as Figure 10 shows.

Figure 10. Search in Eclipse JDT



QuickFixes

QuickFixes are becoming the norm in modern IDEs (see Figure 11). Note that you must have a robust error-handling mechanism that can detect errors and provide solutions for them.

Figure 11. QuickFixes in the Java IDE



Refactorings

Refactorings ease the development load. A successful IDE must have a fair number of refactorings for programmers to fall in love with it.

Integrations

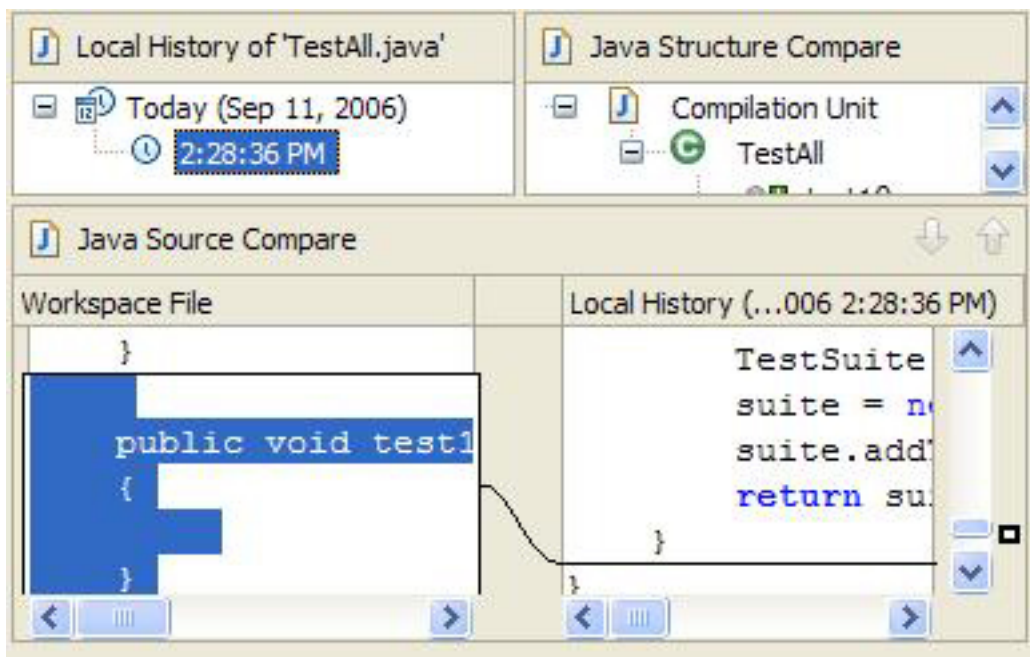
Because you're coding on top of Eclipse, make sure that you follow Eclipse conventions and provide a highly integrated experience so users don't have to learn new ways of performing certain tasks. Heavy integration is especially important if the language your IDE supports is a sort of domain-specific language (DSL) on top of existing languages, such as the Java programming language.

One of my mantras while designing ANTLR Studio was that it should not feel like ANTLR Studio at all, but rather like Eclipse JDT, so users feel that they're working in the same Java environment. The more highly integrated the experience of your IDE, the more experienced users will find it easier to switch to your product.

Diff functionality

Your IDE should provide a visual diff feature that is aware of the programming language of your IDE. For example, JDT has an excellent Java-aware diff utility that can, for example, show exactly which methods have been added, removed, or modified between two files (see Figure 12). It can also ignore irrelevant items, such as whether two methods are in a different location in two files, because in the Java programming language, the order in which methods are defined doesn't matter.

Figure 12. The JDT diff utility is Java-aware



Documentation

Finally, don't forget to provide high-quality documentation for your IDE's features. I have seen many otherwise great IDEs with little documentation, leaving users to flail about, guessing how to use the tools. The only thing worse than writing documentation is using a tool without documentation.

Section 8. Conclusion

The features and techniques outline in this "[Create a commercial-quality Eclipse IDE](#)" series are designed to help Eclipse IDE developers build easy-to-use tools. Designing IDEs is a major task, and there are details I haven't covered. Like a good -- or even bad -- action movie, there are opportunities for sequels.

Until then, design a great IDE, create a nice Web site documenting and touting it, and make a name for yourself.

Resources

Learn

- Learn more about the [Eclipse Foundation](#) and its many projects.
- For an excellent introduction to the Eclipse platform, see "[Getting started with the Eclipse Platform.](#)"
- Expand your Eclipse skills by checking out 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.
- [developerWorks podcasts](#) include interesting interviews and discussions for software developers.
- Stay current with [developerWorks technical events and webcasts](#).

Get products and technologies

- Download the [Eclipse PDE](#) to try the example code shown in the tutorial.
- Download an evaluation copy of [ANTLR Studio](#) to get a feel of the advanced features discussed in this tutorial.
- Check out 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

- Read [Prashant Deva's blog](#) for tips on IDE design and information about the internals of ANTLR Studio.
- Check out the [blog of Cyrus N](#), the designer of the Intellisense functionality in Microsoft Visual Studio, for excellent tips on designing the core of IDEs and the internals of Visual Studio.
- The [Eclipse newsgroups](#) offer 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.