

Create a commercial-quality Eclipse IDE, Part 1: The core

Create professional, commercial-quality IDEs that plug into Eclipse

Skill Level: Advanced

[Prashant Deva](#)
Founder
Placid Systems

05 Sep 2006

This "[Create a commercial-quality Eclipse IDE](#)" series examines how to create professional, commercial-quality IDEs that plug into Eclipse. In this tutorial, learn how to create the core of the IDE.

Section 1. Before you start

About this tutorial

This "[Create a commercial-quality Eclipse IDE](#)" tutorial series shows you what it takes to churn out integrated development environments (IDEs) as Eclipse plug-ins for any of the existing programming languages or your own 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.

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

own IDE.

Part 1 looks at creating the base of the IDE, called the *core*, on top of which all the other components of the IDE are built. It also discusses the general architecture of an IDE and examines techniques employed in the commercial ANTLR Studio IDE to solve some of the problems you may encounter while designing a core.

Prerequisites

This tutorial assumes a basic knowledge of parsers and lexers (what they are and they do), ANTLR, and working with the Eclipse Java™ Development Tools (JDT).

System requirements

To try the code examples in this tutorial, you need the Eclipse software development kit (SDK) running Java Virtual Machine (JVM) V1.4 or later. In addition, to generate the parser and lexer examples, you need ANTLR (see [Resources](#)).

Section 2. Architecture

Although the IDEs available today seem wildly different from each other and are crammed with features, almost all of them share the same basic architecture. The IDE is generally divided into layers or modules, depending on the functionality of its different parts.

Two of the most common modules, which any IDE worth its salt has:

1. Core -- The *base* layer of any IDE, which consists of the parser/lexer that can recognize the language elements.
2. UI -- Where the IDE's UI-related code resides -- syntax highlighting, content outline, etc. Note that the UI layer is built on top of the core layer.

As this tutorial progresses, you'll notice that almost everything in an IDE can be classified within these layers. You can have additional modules, but these two are present in almost every IDE, and we will look at them in detail.

An IDE generally also has these other modules:

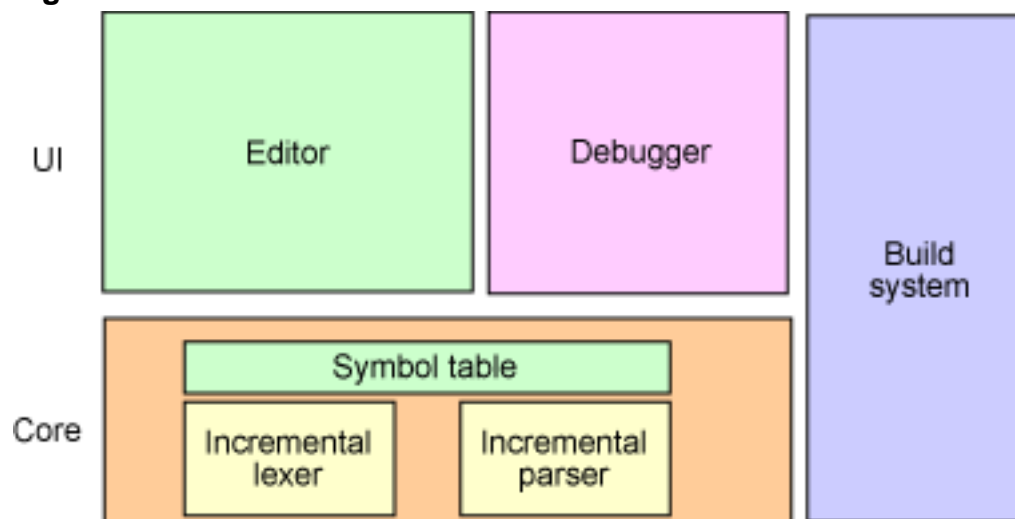
- Debug -- Code related to the debugger
- Doc -- Documentation in HTML format
- Build -- The build system, which may be complex enough to require a separate module

If your IDE is large and complex, you'll usually have the core and UI modules as separate projects, with one team working on each. This approach puts a well-defined boundary between the core and the UI, but makes it difficult to change and debug code. Because the UI layer sits on *top* of the core layer, your UI team members may at times find themselves in a position where they can't continue until the core team has finished. They may also encounter strange behavior in the UI due to bugs in the core. Thus, the core needs to be extremely robust. A great example of such a huge IDE is JDT, which includes a lot of modules, each coded by a separate team.

On the other hand, if your IDE is for a smaller language, you can combine the core and the UI into one Java Archive (JAR). This leads to faster changes and code that's easier to debug, although the process can be a little tough to manage with a team. One example is ANTLR Studio, which is an IDE for the popular ANTLR parser generator. Because ANTLR Studio was coded by one person, and ANTLR syntax isn't as complex as Java language syntax, it was easier to place all the layers in a single JAR, making them much easier to manage and debug.

Figure 1 shows a diagram of the architecture of ANTLR Studio.

Figure 1. Architecture of ANTLR Studio



As you can see, almost every module in ANTLR Studio can be divided into the core and UI categories. Because the build system isn't too dependent on either, it's shown as a separate vertical bar.

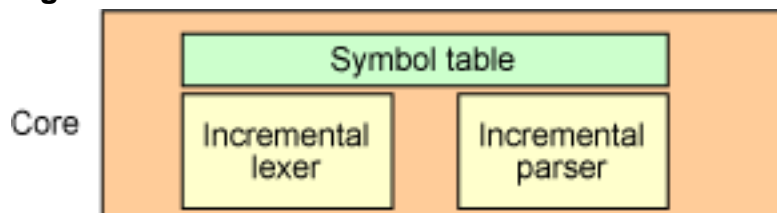
Section 3. The core

This tutorial looks in detail at implementing the core layer of the IDE. The core forms the foundation for the entire IDE: It's the part that *knows* about all the code in the IDE. If your IDE were a human body, the core would be its brain. Everything about your IDE -- its speed, stability, and all the features it can provide -- ultimately boils down to how well designed the core is.

Note that the core is the bottom layer of the IDE, so *everything* else in the IDE depends on it, directly or indirectly. Thus, a bug in the core can cause huge headaches when you're coding the other UI parts of the IDE.

Figure 2 shows a diagram of ANTLR Studio's core.

Figure 2. ANTLR Studio's core



You can see that it contains the following:

- Parser
- Lexer
- Symbol table

Almost all cores contain these basic elements. The *parser* and *lexer* are required to read the source code the user is typing, and the *symbol table* stores that information in a tree form. The other parts of the IDE query the symbol table only to find out about the code and perform their actions. The parser and lexer usually run in the background to keep updating the symbol table.

The *symbol table* is the part exposed to the other components. It's generally in the form of a parse tree with one node for each element of your language's grammar. Here are some common properties a node in the core's parse tree may contain:

- Start/end offset range -- Tells the position in the editor where the element is located

- Value -- The actual text associated with the node (for example, `int a;`)
- Parent element
- Child elements
- Other methods specific to the element type (for example, a node for "method declaration" may have methods to get parameters and return a type)

The next section explains how to implement the lexer and parser.

Section 4. Create a parser and lexer

You need to create a parser and a lexer for your IDE so it can recognize code that is typed into the document. To do so, you need to use a parser and lexer generator. Quite a few are available, such as FLex/Yacc, JavaCC, Anagram, and ANTLR. This series uses ANTLR, which is a free, open source parser generator.

For this tutorial, you'll create a parser/lexer for a small C-like language called Tiny C. The language lets you declare variables and function declarations. It currently recognizes only two native types: `char` and `int`. The declarations can contain simple statements like `while` and `if` loops, and assignment expressions. The language also allows single-line comments.

Listing 1 shows the ANTLR grammar for the parser.

Listing 1. The parser's ANTLR grammar

```
class
TinyCParser
extends
Parser;

program
: (
declaration
)* EOF
;

declaration
:
(variable)
=>
variable
|
function
;
```

```
declarator
: id:ID
| STAR
id2:ID
;

variable
: type
declarator
SEMI
;

function
: type
id:ID
LPAREN
(formalParameter
(COMMA
formalParameter)*)?
RPAREN
block
;

formalParameter
: type
declarator
;

type:
(
"int"
|
"char"
| ID
)
;

block
:
LCURLY
(
statement
)*
RCURLY
;

statement
:
(declaration)
=>
declaration
| expr
SEMI
| "if"
LPAREN
expr
RPAREN
statement
(
TK_else
statement
)?
|
"while"
LPAREN
expr
RPAREN
statement
| block
```

```

;
expr:
assignExpr
;

assignExpr
: aexpr
(ASSIGN
assignExpr)?
;

aexpr
: atom
(PLUS
atom)*
;

atom:
ID
| INT
| CHAR_LITERAL
| STRING_LITERAL
;

```

Listing 2 shows the corresponding lexer.

Listing 2. The lexer

```

class TinyCLexer extends Lexer;

options {
    k=2;
    charVocabulary = '\3'..'377';
}

tokens {
    "int"; "char"; "if"; "else"; "while";
}

WS : ( ' '
      | '\t'
      | '\n' {newline();}
      | '\r' )
    { _ttype = Token.SKIP; }
;

SL_COMMENT :
    "//"
    (~'\n')* '\n'
    { _ttype = Token.SKIP; newline(); }
;

LPAREN
options {
    paraphrase="'('";
}
: '('
;

RPAREN
options {

```

```

        paraphrase="')'";
    }
    : ')';
;

LCURLY: '{';
;

RCURLY: '}';
;

STAR: '*';
;

PLUS: '+';
;

ASSIGN
    : '=';
;

SEMI: ';';
;

COMMA
    : ',';
;

CHAR_LITERAL
    : '\\'' (ESC|~'\\'' ) '\\'';
;

STRING_LITERAL
    : '\"' (ESC|~'\"')* '\"';
;

protected
ESC : '\\\'
    (
        'n'
        'r'
        't'
        'b'
        'f'
        '\"'
        '\\\'
        '\\\'
        '0'..'3'
        (
            : DIGIT options {warnWhenFollowAmbig = false;}
        )?
        | '4'..'7'
        (
            : DIGIT options {warnWhenFollowAmbig = false;}
        )?
    )
;

protected
DIGIT
    : '0'..'9';
;

INT : (DIGIT)+

```

```
    ;
ID
options {
  testLiterals = true;
  paraphrase = "an identifier";
}
: ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
```

The following is a valid block of code for this grammar:

Listing 3. Valid block of code

```
int a;
char myFunc()
{
  int b =a;
}
```

Section 5. The need for speed: An inside look at the functioning of the core

Determining how well designed the core is depends on one key factor: speed. Speed is your biggest challenge when designing the core. The performance of the entire IDE is ultimately defined by the core's performance. If the core isn't fast enough, you may find that you can't implement some of the features you envisioned because they're too slow to be of use.

Why is speed so important in the core? With 3-GHz (and faster) multicore processors available, should you even give it a second thought?

IDEs (and especially their editors, in this case) have a special requirement that you probably won't find in most other applications you write. If you've used any modern IDE, you must have noticed that as you type code in the editor, you see squiggly markers that indicate errors. In addition, an outline of the code appears in side views, you can get instant code completion if you use **Ctrl+Space**, etc. All this happens almost in real time while you type, without your having to press a button to update the IDE. You see no delay, regardless of how many lines of code you've written.

This is where speed comes into play. The IDE must keep up with your typing and update almost all of its data structures in less than a fraction of a second. And it can't take up too much processing power or the user will notice a lag while typing --

characters will appear on screen only after they've been typed, which will make your IDE more of a pain to use.

The art of IDE design involves keeping your data structures up to date while maintaining the IDE's responsiveness. Doing so is made even more difficult by the fact that people type quickly and expect characters on the screen to appear instantly.

Most IDEs (including the Eclipse JDT) take the following approach to solve this problem:

1. Run the parser/lexer in a background thread.
2. The thread runs after a fixed, short time interval (say, 200 ms) and runs the parser/lexer to update the symbol table.
3. The other views, such as outline view, run in their own thread and update themselves by looking at the symbol table, but their interval time is longer than that of the background thread (perhaps 1,200 ms).

This way, although you won't *always* remain up to date, you're *mostly* up to date, and the IDE is fast enough that the user won't notice. Also, because each view runs in a background thread, the user thread on which the user is typing remains mostly unaffected, so the user doesn't notice any lag while typing.

ANTLR Studio handles this situation a little differently. ANTLR Studio is always up to date, which means that at every keystroke, the entire symbol table is updated. There is no lag between the symbol table and what the user types in the editor, so the user doesn't notice any lag before characters appear on the screen. As a result, ANTLR Studio can provide features that would otherwise not be possible -- and that make a great case study. The next section looks inside ANTLR Studio.

Section 6. Incremental parser and lexer

In ANTLR Studio, the parser/lexer runs in the same thread as the editor. In a nutshell, this is what happens at every keystroke in ANTLR Studio's editor:

1. The parser and lexer are invoked. They scan the entire document, forming a tree of all the code in the editor.

2. The syntax highlighter is invoked to scan and color the line on which the user is typing.
3. The context in which the user is typing is checked. If any auto-completions are available, they're presented to the user.

All this happens with no lag, regardless of whether the editor contains 10 lines or 10,000 lines. This performance is possible thanks to a trick.

[Figure 2](#), which shows ANTLR Studio's core, includes the words *incremental parser* and *incremental lexer*. What difference does that make?

Instead of going through the entire document each time, an incremental parser or lexer figures out which portions of the document have been changed since the last time and processes only that part of the document, leaving the rest as is. If you type one character, the incremental parser and lexer scan only that little portion of the document. This saves a huge amount of time and allows the parser and lexer to run in the same thread.

Disadvantages of an incremental parser and lexer

Ironically, although the incremental parser/lexer result in a better IDE for the end user, the disadvantages affect you, the developer:

- Developing an incremental parser and lexer is a tough and complex process.
- Doing so adds huge amounts of complexity to the underlying code.
- You must change the design of all your components in a fundamental way, which, in turn, makes them complicated.

To take full advantage of the incremental nature of the parser and lexer, you need to make all the other components in your IDE update in an incremental way, as well. Coming back to the original problem of updating *all* the data structures of the IDE within a fraction of a keystroke, you solve it by having the data structures update incrementally, not entirely. Having an incremental parser and lexer is just a base for letting you have the entire IDE update incrementally.

Section 7. The delta

As an example, consider the content outline view. Under normal circumstances, it

behaves as follows:

1. It asks the core for a tree of the code after a certain time interval (say, 1,200 ms).
2. It takes the new tree and updates the entire view.
3. The view flashes because the entire view is updated and redrawn. If this happens every 1.2 seconds, it's a major distraction for the user.

As a demonstration, open Eclipse, and create a new Java project (or use an existing one). Make sure the outline view is open. Create a Java file, and type some code in it. The outline view updates without any flashing, which is what is expected of a high-quality IDE like the JDT.

Taking a tree and telling the view to refresh is far easier to code than having it compare the changes and updating only the necessary parts. But again, that results in a far better end-user experience. This brings up another problem: Once you get the tree from the parser, you need to check what changes have been made to it from the previous tree, so you can pass those changes to the components that need them. The data structure that holds the information about the changes is called the *delta*. Thus, you need to calculate the delta of the parse tree after each parse.

The delta must be calculated immediately after the parse and in the same thread. If, like ANTLR Studio, you plan to run the parser in the same thread as the editor, you need to make sure the delta calculation algorithm is fast enough that the user doesn't notice any lag while typing.

Here is what a simple delta structure for the Tiny C language might look like:

Listing 4. Simple delta structure for Tiny C

```
interface Delta {  
    Declaration[] getAddedDeclarations();  
    Declaration[] getRemovedDeclarations();  
    Declaration[] getModifiedDeclarations();  
}
```

If an outline view needs to update itself, it does the following:

1. Call `getRemovedDeclarations()` and remove the nodes for all the declarations that have been removed.
2. Call `getAddedDeclarations()` and add the nodes for all declarations

that have been added.

3. Call `getModifiedDeclarations()` and change the text for the nodes that have been modified.

Although this process adds complexity, it's faster than refreshing the entire tree -- and the user won't notice any flashing.

Delta combining

Say you're running the parser in the same thread as the editor. You have three other views open, which utilize the parse tree produced by the parser. To reproduce the parse tree on each keystroke, you'd need to update all three views on each keystroke, too. That would generate lag because each view would do its own processing on the parse trees. You'd essentially update all data structures *and* redraw all the views on each keystroke. Oops -- that's what you set out to *prevent* from happening!

How about if you run the views on a different thread and update them after a certain time interval, but keep the parser running on the UI thread itself? Here's what happens:

1. On each keystroke, the core generates a parse tree and a delta for that parse tree *relative to the previous one*.
2. The content outline view invokes a thread after 1,200 ms, which asks the core for a delta.
3. But the content outline view holds a parse tree that was generated five keystrokes earlier, so the current delta doesn't apply to it.
4. The view goes out of sync.

How do you solve this? (Think for a minute before looking at the solution.)

Here's what you do:

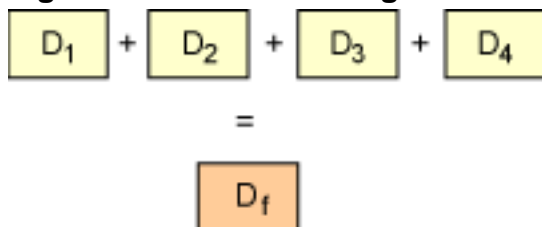
1. Each view has a queue associated with it.
2. Each time the core generates a delta, it keeps a copy in the queue of each view.
3. When the view wants to update itself, it applies each of these deltas *in succession* to the parse tree it's holding -- and voila! It's up to date. You can optimize this by looking at all the deltas first and canceling the nodes

that have the same node added and later deleted in any of the subsequent deltas in the queue.

4. Once the view is done, it clears the queue, and the whole process starts over.

This technique is called *delta combining* (see Figure 3) because you essentially collect and combine all the deltas into one and execute that single delta on the parse tree. Note that you have to keep the queue synchronized so that the core and the UI component don't try to write to it at the same time.

Figure 3. Delta combining



Here's some code to demonstrate how this technique works. You may need to add some extra synchronization-related code, depending on the design of your IDE.

Listing 5. Delta combining

```
// interface for components that want to collect deltas
interface DeltaListener{
    void newDeltaCreated(Delta delta);
}

class DeltaUpdatingView extends MyView implements DeltaListener
{
    private BlockingQueue<Delta> queue;

    public void newDeltaCreated(Delta delta)
    {
        queue.put(delta);
    }

    public void onUpdate()
    {
        for(Delta delta:queue.toArray(new Delta[queue.size()]))
        {
            //apply delta to the UI component
        }

        //clear the queue
        queue.clear();
    }
}
```

Section 8. Error recovery

Error recovery of the parser is something you must look at closely when designing the core. Most of the time, when users enter code, errors occur. Your parser needs to recover from those errors gracefully and continue parsing until the end of the document.

The bad news is that there are no shortcuts around this. Although ANTLR provides some basic error handling, you must manually put checks in the parser to make it recover from the errors. This is how all IDEs work, including JDT and ANTLR Studio.

As far as speed is concerned, you need to be careful: ANTLR throws an exception each time it encounters an error, and exceptions are slow. Try to make the most out of each exception thrown by consuming tokens until you see valid input.

As an example, let's add some simple error-recovery code to the Tiny C language. This is what you do if you encounter an error while parsing:

1. For a *statement*, skip ahead until you find a semicolon (;).
2. For a *block* of code, skip ahead until you find a right curly brace (}).

Here is the corresponding grammar code, with error handling added:

Listing 6. Grammar code with error handling

```
statement
: (declaration) => declaration
  | expr SEMI
  | "if" LPAREN expr RPAREN statement
    ( TK_else statement )?
  | "while" LPAREN expr RPAREN statement
    block
  ;

exception catch [RecognitionException ex]
{
  consume();
  while (LA(1) != Token.EOF_TYPE && LA(1)!=SEMI ) {
    consume();
  }

  if(LA(1)==SEMI)
    consume();

  return;
}

block
: LCURLY ( statement )* RCURLY
```

```
        ;
    exception catch [RecognitionException ex]
    {
        consume();
        while (LA(1) != Token.EOF_TYPE  && LA(1)!=RCURLY ) {
            consume();
        }

        if(LA(1)==RCURLY)
            consume();

        return;
    }
```

Note that error handling is an important part of the core. If the error handling is bad, the whole IDE suffers, and the user may not be able to get any code completions or outlines. Imagine a situation where you type `int` in the middle of a Java file, and the remaining lines are underlined with red squiggles because the parser can't recover from an error (see Figure 4). You probably wouldn't use that IDE, would you?

Figure 4. Users see squiggles if parser's error recovery isn't good

```
void execute() throws BuildException

(dir == null)

throw new BuildException("dir attribute

(dir.isFile())

throw new BuildException("Unable to crea
```

Be sure you give error handling a high priority when designing the core. Use lots of test cases that check how well an IDE can recover from errors. This will become more complex relative to the complexity of the language for which the IDE is built. For example, because the syntax of Java is far more complex than that of ANTLR, a Java IDE requires much more complex error-handling code than an IDE for ANTLR.

Section 9. Design decisions when designing the core

Now that you've seen the problems associated with designing the core, examine some of the major design decisions you need to make while creating the core's

architecture:

Incremental or nonincremental

As described, the choice of having an incremental or nonincremental core affects your IDE in fundamental ways. You need to decide from the beginning whether to design an incremental core.

UI or non-UI thread

If you decide to use an incremental core, you must decide whether to run the parser/lexer in a background thread or in the same thread as the editor. Running in the UI thread saves you from a *lot* of synchronization-related problems, but then the speed of the core will become your worst enemy (if you run in the UI thread on each keystroke, a time limit is imposed on the document analysis). Even a delay of a few milliseconds can cause a lag between the user typing a letter and it appearing on the screen, which can make using your IDE unbearable.

If you do pull it off, you can provide some cool features in your IDE that others running in a background thread will find it almost impossible to include. For example, running the analysis in the UI thread allows ANTLR Studio to provide TypeOnce functionality, which doesn't require the user to press **Ctrl+Space** to show auto-completions and even works when the rule name being typed hasn't been defined.

Partially incremental

Instead of having an incremental core, with everything in the IDE updating incrementally, you can have a partially incremental core. This means that instead of having both an incremental parser and an incremental lexer, you just have an incremental lexer. In the analysis part, the lexer takes 80 percent of the time, so having an incremental lexer speeds your analysis quite a bit. This way, your IDE won't need to be so complex while at the same time being fast. This is the approach taken by the NetBeans IDE.

Error-recovery strategy

You need to identify the most common form of errors your parser/lexer will face as users type code. You must then add custom code to make sure the parser can recover from these errors gracefully. Otherwise, the user's experience in your IDE will be diminished because the IDE won't have information about the code at critical points when the user needs to perform an operation, such as code completion.

Section 10. Conclusion

This first part of the "[Create a commercial-quality Eclipse IDE](#)" series has examined the general architecture of an IDE and taken a detailed look at issues related to implementing the IDE's core. By now, you should understand the importance of having a robust, high-performance core and the related design issues with running a parser/lexer in the UI or non-UI thread. This tutorial also looked at problems associated with the UI portions that communicate with the core and the importance of robust error handling in the core.

[Part 2](#) of this series looks at the UI and the API framework that Eclipse provides, which makes it much easier to write IDEs.

Resources

Learn

- Read "[Understanding ANTLR Grammar Files](#)" to get familiar with ANTLR quickly.
- Look at the [Harmonia Research Project](#) for papers on incremental parsers and lexers.
- Check out the [ANTLR 2005 workshop demonstration of ANTLR Studio](#), which contains a lengthy portion on the interior designs of ANTLR Studio.
- Visit the developerWorks [Java technology zone](#) for lots of articles on thread synchronization in Java.
- 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 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](#).

Get products and technologies

- Download the latest version of [ANTLR](#).
- Get a free trial of [ANTLR Studio](#).
- Download the [Eclipse SDK](#).
- Get the [NetBeans Incremental Lexer](#) for use in your own IDE.
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Discuss ANTLR at the [antlr-interest mailing list](#).
- 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.