Internal Report 2010-17 September 2010

Universiteit Leiden

Opleiding Informatica

The NLP-Editor

A case study about

Eclipse Plug-In development

Derk Geene

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

The NLP-editor

An case study about Eclipse plug-in development

Derk Geene (s0110841)
Leiden University, April 2008
Supervisor: Bart Kienhuis

Introduction

The Computer Systems Group of the Leiden Instutute for Advanced Computer Science developes a design methodology that allows for fast mapping of nested-loop applications (e.g. DSP, Imaging, or Multi-Media), written in a subset of Matlab onto reconfigurable devices. This methodology is implemented as a tool chain called Compaan/Laura.

In this context an editor for nested loop programs is a very convenient tool. The topic of this case study is how to build this kind of an editor on top of the Eclipse Platform which is directed at building integrated development environments like the editor we have in mind. The goal is to build an editor that allows for easy development of nested loop programs using the possibilities of this platform. The editor will also give visual feedback to the user coming from the different stages of the Compaan/Laura tool chain.

Chapter one gives a short introduction on the Eclipse platform and the way in which it can be extended using plug-ins to develop an integrated development environment such as the NLP-editor. The second chapter describes how different editor functions are implemented using the eclipse platform. These functions include the creation of an outline using a visitor pattern (2.1), the use of markers to inform the user about a parse error (2.2), error-recovery (2.3), the extraction of linearization types from an additional file(2.4), the use of a rule based scanner to add special colourings (2.5) and extra information using a hover(2.6) to specific statements, the detection of scope errors (2.7) and the use of a scheduler to delay the parsing in order to prevent performance drops (2.8).

1. Platform

Eclipse is a platform that has been designed from the ground up for building integrated web and application development tooling. By design, the platform does not provide a great deal of end user functionality by itself. The value of the platform is what it encourages: rapid development of integrated features based on a plug-in model.
Eclipse offers a solution to the problems that arise each time an integrated environment is needed for a specific goal. The same requirements are their each time an integrated development environment is needed but in many cases the user ends up with programs have completely different user interfaces for doing similar tasks. There have been many announcements about strategic alliances and open architectures that promise to solve this problem, but most solutions appear to be a set of different tools which are put together with import/export duct tape to 'integrate' them.
The added value of the Eclipse platform is in the totally different approach that is used to solve the integration problem. It is not directed at providing end user functionality itself, but it is designed to allow developers to rapidly add the needed functionality to the Eclipse platform using plug-ins. At the core of Eclipse is an architecture for dynamic discovery, loading, and running of plug-ins. The platform handles the logistics of finding and running the right code. The platform UI provides a standard user navigation model.
Each plug-in can then focus on doing a small number of tasks. Typical tasks for an editor as we are building in this case study are defining, testing, compiling and debugging, but the tasks that a plug-in can provide are certainly not limited to these. Using plug-ins a developer can add almost every task in a relatively easy and transparent manner.

Plug-in Development Environment

The Nested Loop Programs editor is developed as a plug-in for the Eclipse platform. The development of plug-ins for the Eclipse platform is done using the Plug-in Development Environment (PDE) which provides a set of tools to create, develop, test, debug and deploy Eclipse plug-ins.
The PDE provides a new project creation wizards which makes the creation of a new plugin very easy. The PDE also provides form-based manifest editors. Using these multi-page editors the developer can centrally manage all manifest files of a plug-in or feature.
Other tools the PDE provide include special views such as the error-log, the plug-in registry and plug-in dependencies which provide the plug-in developer wit hall the information he needs.

PyDev

The NLP editor is build using the existing Eclipse Plugin PyDev as a basis. PyDev allows users to use the Eclipse Platform as a Python and Jython development environment. The Python parser was replaced with the Matlab parser as used in the Compaan/Laura tool chain. This was the basis for an editor especially designed for the editing of nested loop programs. Chapter two describes how all the functionality that was needed was than added to this basis.

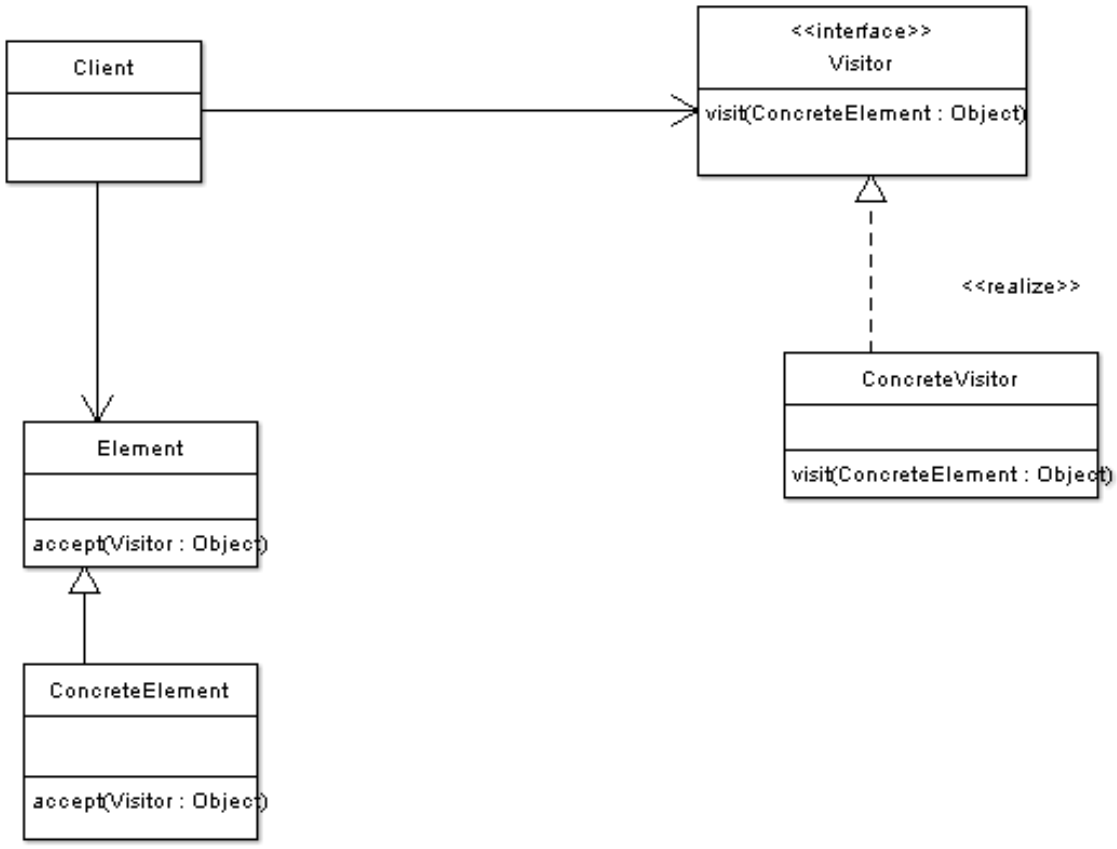
2. Editor features

2.1 Creating an outline using a visitor design pattern

The outline view displays an outline of a structured file that is currently open in the editor area and lists structural elements. The contents of the outline view in eclipse are editor specific. In the case of the NLP editor the structure of the file is shown by using the syntax tree that is generated by the JJTree/JavaCC Matlab parser.

To keep the implementation of the outline separated from the parser a visitor pattern is used. A visitor design pattern is a way of separating an algorithm (in this case: the adding of an item to the outline view of the eclipse workbench) from an object structure (in this case: the syntax tree). The advantage of this separation is the ability to add new operations to the syntax tree without having to modify the syntax tree. That's why the visitor pattern will also turn out to be very convenient for the adding of other functionality to the editor.

In a nutshell the visitor allows for adding new virtual functions to a family of classes without modifying the classes themselves; instead, a visitor class is created that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.



Step 1: Adding a visitor to the MatlabParser

The first things needed are a Visitor interface that has a visit() methods for each type of node of the syntax tree, and accept() methods in the syntax tree that can take the visitor object as an argument. The accept() method of the node calls back the visit() method for its class. The visitor class for the outline will contain visit() methods that add the nodes to the outline.

The adding of this visitor and the accept functions is very easy because the JJTree treebuilder of JavaCC can generate a visitor implementation. To enable this feature the visitor option is enabled in the JJTree input file:

```
options {
  VISITOR=true;
}
```

After using JJTree and JavaCC to create a new parser, all different classes of the abstract symbol tree (AST) nodes will contain a jjtAccept function like this:

```
public class ASTInteger extends SimpleNode {
  ...

  /** Accept the visitor. **/
  public Object jjtAccept(MatlabParserVisitor visitor, Object data) {
    return visitor.visit(this, data);
  }
}
```

Also a MatlabVisitor interface has been generated by JJTree that looks like this:

```
package org.totic.pydev.parser.matlab;

public interface MatlabParserVisitor
{
  public Object visit(SimpleNode node, Object data);
  public Object visit(ASTNLP node, Object data);
  ...
  public Object visit(ASTInteger node, Object data);
}
```

Step 2: Creating the outline visitor

The next step is to build a concrete implementation of the visitor for the outline. The outline is a tree that needs a model. This model consists of items. Those items are being added using a new OutlineVisitor that traverses the syntax tree:

```
class ParsedItem {
  /**
   * Traverses the parsed tree. Fills the array list with the
   * items we are interested in.
   */
  static class OutlineVisitor implements MatlabParserVisitor {
    ArrayList fill;
    ParsedItem parent;

    public OutlineVisitor(ParsedItem parent, ArrayList fill) {
      this.parent = parent;
      this.fill = fill;
    }

    /**
     * Example of a visit function, visit functions exist for all the
     * different type of AST nodes.
     */
    public Object visit(SimpleNode node, Object data) {
      for (int i = 0; i < node.jjtGetNumChildren(); i++) {
        fill.add(new ParsedItem(
          parent,
         (SimpleNode) node.jjtGetChild(i)
        ));
      }
      return null;
    }
  }

  public ParsedItem[] getChildren() {
    if (children == null) {
      ArrayList allMyChildren = new ArrayList();
      OutlineVisitor v = new OutlineVisitor(this, allMyChildren);
      try {
        if (token != null)
          token.jjtAccept(v,  null);  //  traversal  fills  in  the
children
          children = new ParsedItem[allMyChildren.size()];
          for (int i = 0; i < allMyChildren.size(); i++)
            children[i] = (ParsedItem) allMyChildren.get(i);
      } catch (Exception e) {
      }
    }
    return children;
  }
}
```

Step 3: Add line-numbers to the syntax tree nodes to allow for highlighting code and using the outline as a navigation aid.

Now the Outline view of the workbench displays an outline of the code file that is currently open in the editor area, but at this point when an element in the outline view is selected the corresponding code in the editor is not being highlighted.

To add this functionality the items of the syntax tree must contain their corresponding start and end positions in the source file. This information is added to the nodes of the syntax tree by changing the jjtreeOpenNodeScope and jjtreeCloseNodeScope functions.

```
void jjtreeOpenNodeScope(Node n) {
  Token t = getToken(1);
  jjtree.pushNodePos(t.beginLine, t.beginColumn);
  n.setBeginLineNumber(t.beginLine);
  n.setBeginColumnNumber(t.beginColumn);
}

void jjtreeCloseNodeScope(Node n) {
  jjtree.setNodePos(n);

  Token t = getToken(0);

  n.setEndLineNumber(t.endLine);
  if (t.toString().equals("\n")) {
    n.setEndColumnNumber(t.endColumn-1);
  } else {
    n.setEndColumnNumber(t.endColumn);
  }
}
```

Each time a new node scope is started the jjtreeOpenNodeScope function is called using the current node as its input. The new setBeginLineNumber and setBeginColumnNumber functions add the correct starting information to the node that can be used later when the node is visited by the OutlineVisitor. The same holds for the setEndLineNumber and setEndColumnNumber functions that are added to the function that is called when the scope is node scope is closed.
The setter and getter functions for the begin- and end-information of the node are added to SimpleNode which is the base class for all the node types of the syntax tree.

To get the highlighting to work the getPosition function of ParsedItem is changed as follows:

```
public IOutlineModel.SelectThis getPosition() {
  IOutlineModel.SelectThis position = new IOutlineModel.SelectThis(
    token.getBeginLineNumber(),
    token.getBeginColumnNumber()-1,
    token.getEndColumnNumber() - token.getBeginColumnNumber()+1);
  );
);
```

2.2 Parse error markers

A very important feature of the editor is the ability to inform the code writer about the errors he is making. During the course of editing the error information has to be brought to the user in an understandable way. In Eclipse the resource marker mechanism can be used to manage this kind of information.
A resource marker is like a yellow sticky note stuck to a resource. A marker can contain information about a problem (e.g., location, severity) and the user can quickly jump to the problem location by clicking the marker in the problems view which contains a list of all the warnings and errors.

The PyParser class is the main parser class of the project. It uses MatlabParser to analyse the document. PyParser is connected to PyEdit, the editor view, and listens to changes in the document. On every document change, MatlabParser is asked to generate a new syntax tree. Clients that need to know when a new syntax tree has been generated can register as a parseListener. In the case a correct syntax tree has been generated (i.e., MatlabParser returns without an error) fireParserChanged() calls the parserChanged() function of all the listeners. But in the case that MatlabParser returns a ParseException fireParserError() is called which in turn calls the parserError() functions of all the listeners.

```
public class PyParser {
  IDocument document; // document to be parsed
  PyEdit editorView; // the editor associated
  SimpleNode root; //ast for the last successful parsing
  ArrayList parserListeners; // listeners that get notified
  private MatlabParser parser = new MatlabParser(); // the parser

  protected void fireParserError(Throwable error) {
    if (parserListeners.size() > 0) {
      Iterator e = parserListeners.iterator();
      while (e.hasNext()) {
        IParserListener l = (IParserListener) e.next();
        l.parserError(error);
      }
    }
  }

  public void reparseDocument() {
    IEditorInput input = editorView.getEditorInput();
    IFile original = (input instanceof IFileEditorInput) ?
      ((IFileEditorInput) input).getFile() : null;
    try {
      parser = new MatlabParser(inString);
      root = parser.NLP();
      fireParserChanged();
    } catch (ParseException parseErr) {
      fireParserError(parseErr);
    }
  }
}
```

In the case MatlabParser throws as parseException the parseError() fucntion of all the PyParser listeners is called. One of the registered listeners is the editor, PyEdit. The parserError function of the PyEdit class figures out where the error occurred and creates a marker containing the error message from the MatlabParser on this position.

```
public class PyEdit extends TextEditor implements IParserListener {
  // This event comes when parse ended in an error
  public void parserError(Throwable error) {
    ...

    int errorStart;
    int errorEnd;
    int errorLine;
    String message;

    // Figure out where the error is in the document
    Token errorToken = parseErr.currentToken.next != null
      ? parseErr.currentToken.next
      : parseErr.currentToken;

    message = parseErr.getMessage();

    IRegion startLine =
      document.getLineInformation(errorToken.beginLine - 1);
    IRegion endLine =
      document.getLineInformation(errorToken.endLine - 1);
    errorStart = startLine.getOffset() + errorToken.beginColumn - 1;
    errorEnd = endLine.getOffset() + errorToken.endColumn;
    errorLine = errorToken.beginLine;

    // Create a marker for the error
    Map map = new HashMap();
    map.put(IMarker.MESSAGE, message);
    map.put(IMarker.SEVERITY, new Integer(IMarker.SEVERITY_ERROR));
    map.put(IMarker.LINE_NUMBER, new Integer(errorLine));
    map.put(IMarker.CHAR_START, new Integer(errorStart));
    map.put(IMarker.CHAR_END, new Integer(errorEnd));

    MarkerUtilities.createMarker(original, map, IMarker.PROBLEM);

    ..
  }
}
```

This method to display the parse error works very well but has a disadvantage: when MatlabParser encounters an error it throws a parseException and it stops. This means that only one error can be detected and the document will only be parsed till the position where this error occurs. How this problem can be solved is the topic of the next section.

2.3 Multiple error markers by using error recovery

Each time the MatlabParser encounters an error is throws a parseException and stops. To add the possibility to show multiple errors this behaviour has to be changed.

MatlabParser is created using the JavaCC parser generator. JavaCC allows compiler writers to deal with error recovery. To use error recovery the production rules for JavaCC needs some changes.

Let's take the forStatement production for example:

```
void forStatement () : {}
{
  (
    <FOR>Identifier() "=" complexExpression()<COLON>Integer()<COLON>
      complexExpression() "," <NEWLINE>
    [ listOfAllStatements() ]
    <END> <NEWLINE>
  )
  {
  }
}
```

Each time the parser detects a problem in the for statement, it throws the exception ParseException. To recover from this exception we catch this exception and call a special function called error_skipto. The new production for the for-statement:

```
void forStatement () : { }
{
  try {
    <FOR> Identifier() "=" complexExpression()<COLON>Integer()<COLON>
      complexExpression() "," <NEWLINE>
    [ listOfAllStatements() ]
    <END> <NEWLINE>
  } catch (ParseException e) {
    error_skipto(END);
  }
  {
  }
}
```

The error_skipto function adds the exception to a new list that contains all the exceptions and then it skips everything that is not of the kind given as input to the error_skipto function. In the case of the for-production above, when an error occurs everything that is not of the kind END will be skipped. After this, the parsing of the rest of the file will continue as usual. All the exceptions that are found during the parsing process will be available to the editor so they can al be added as a marker.

The error_skipto function:

```
void error_skipto(int kind) {
  ParseException e = generateParseException();
  exceptionList.add(e);

  Token t;
  do {
    t = getNextToken();
  } while (t.kind != kind);
  getNextToken();
}
```

Because the errors are being caught the PyParser will now call the ParserChanged function in stead of the ParserError function, also in the case there are one or more parse errors. So now the ParserChanged function of the editor has to add the exceptions on the exceptionList as a marker to the editor view. This will typically look like this:

```
public void parserChanged(SimpleNode root) {
  ...
  Enumeration errors = parser.getExceptionList().elements();
  ParseException e;
  while ( errors.hasMoreElements() ) {
    e = (ParseException)errors.nextElement();
    Map map = new HashMap();
    map.put(IMarker.MESSAGE, new String(e.toString()));
    map.put(IMarker.SEVERITY, new Integer(IMarker.SEVERITY_ERROR));
    map.put(IMarker.LINE_NUMBER, new Integer(e.getLineNumber()));
    map.put(IMarker.TEXT, new String("hallo"));

    MarkerUtilities.createMarker(original, map, IMarker.PROBLEM);
  }

  ...
}
```

Using error recovery the editor view now contains all the parse errors that MatlabParser encounters!

2.4 Get linearization types from KPN file

In the Compaan toolchain the matlab files get a kpn file with a corresponding name that contains a Kahn Proces Network (KPN). These are basically xml documents that contain a model of computation in which a group of processing units are connected by communication channels to form a network of processes.

For the NLP-editor some of the information contained in the KPN file is needed to give a certain color to the ipd-statements in the matlab file. This color corresponds the linearization type of this ipd statement.

To fetch this information from the .kpn file a new parser is added to the project. The Kpnparser opens the kpn file, if there is any, that has the same filename as the matlab file that is currently open in the editor view. When the timestamp of the kpn file is newer than the matlab file it means that it is generated after the matlab file, so it can be used. In this case a xml parser is used to create a document object model of the xml file.

Now a visitor is used to visit all the ipd-statements in the syntax tree. The linktype is added to the node and to a special datastructure that is used to give a certain color to every ipd statement in the editor view.

2.5 Using a rule based scanner for colouring the ipd statements according to their linearization type

The actual coloring is done by adding a new rulebased scanner to the editor configuration.

```
package org.totic.pydev.editors;
public class PyEditConfiguration extends SourceViewerConfiguration {
  ...
  private MyRuleBasedScanner getMyCodeScanner() {
    MyRuleBasedScanner codeScanner = new MyRuleBasedScanner();
    IToken IOMminToken = new Token(new TextAttribute(colorCache
                          .getNamedColor("BLUE")));
    IToken IOMplusToken = new Token(new TextAttribute(colorCache
                          .getNamedColor("GREEN")));
    IToken OOMminToken = new Token(new TextAttribute(colorCache
                          .getNamedColor("YELLOW")));
    IToken OOMplusToken = new Token(new TextAttribute(colorCache
                          .getNamedColor("RED")));
    ...
    LocationRule locationRule = new LocationRule(
      new GreatKeywordDetector(), defaultToken);

    Iterator it = kpnparser.fLocations.entrySet().iterator();
    while (it.hasNext()) {
      Map.Entry pairs = (Map.Entry)it.next();
      Integer[] location = (Integer[])pairs.getKey();

      String[] temp = (String[])pairs.getValue();
      if (temp[1].equals("IOM-")) {
        locationRule.addLocation(location[0]-1, location[1],
          location[2], IOMminToken);
      } else if (temp[1].equals("IOM+")) {
        locationRule.addLocation(location[0]-1, location[1],
          location[2], IOMplusToken);
      } else if (temp[1].equals("OOM-")) {
        locationRule.addLocation(location[0]-1, location[1],
          location[2], OOMminToken);
      } else if (temp[1].equals("OOM+")) {
        locationRule.addLocation(location[0]-1, location[1],
          location[2], OOMplusToken);
      }
    }

    rules.add(locationRule);

    ...
  }
}
```

The getMyCodeScanner is extended with a new rulebasedscanner that adds colourings to the ipd statements using the special datastructure (fLocations) that contains the locations of the all the ipd statements with their corresponding types. Depending on their type (IOM-, IOM+, OOM- or OOM+) the statements get a different color.

2.6 Adding hover information to the ipd statements

The linearization types are also added to the editor view as a text hover that appears when the user has his mouse on one of the ipd statements. This is implemented using the same datastructure containing all the ipd statements with their corresponding type that is supplied by the KpnParser.

To add the new hovers to the editor a class called MyTextHover is added that implements ITextHover interface as defined by the eclipse platform. A new function getTextHover adds our new texthover class to our editor.

```
public class PyEditConfiguration extends SourceViewerConfiguration {
  public ITextHover getTextHover(
    ISourceViewer sourceViewer, String contentType)
  {
    return new MyTextHover(kpnparser);
  }
}
```

The MyTextHover class implements the function getHoverInfo which displays the correct hover for each ipd statement:

```
package org.totic.pydev.editors;
public class MyTextHover implements ITextHover {
  …

  public String getHoverInfo( textViewer, hoverRegion) {
    IDocument document = textViewer.getDocument();

    if (hoverRegion != null) {
      try {
        if (hoverRegion.getLength() > -1) {
          Iterator it = kpnparser.fLocations.entrySet().iterator();
            while (it.hasNext()) {
              // check if the location of the hover corresponds with
              // the location of the iterated ipd
              if ( locations match ) {
                return "Linkname: " + linkinfo[0] +
                       "\nLinearization: " + linkinfo[1] +
                       "\nSize: " + linkinfo[2];
              }
            }

            return textViewer.getDocument().get(
              hoverRegion.getOffset(), hoverRegion.getLength());
        }
      } catch (BadLocationException x) {
      }
    }
  }
}
```

2.7 Scope error detection

Parse errors are show in the workbench nicely now, however there are more mistakes that can be made by the programmer. This section explains how scope errors are being detected and marked in the editor view.

An typical example of a scope error is shown in the following code fragment:

```
for j = 1 : 1 : N,
   for i = j : 1 : N,
      [ out_0 ] = ReadMatrix_Zeros_64x64;
      [ r_1( j, k) ]  = opd( out_0 );
   end
end
```

The identifier k on the fourth line is not defined in the current scope (assuming that it is not defined before this fragment). This error is not found by the parser. Support for the detection of scope errors is added by using the visitor pattern again.

```
static class KpnVisitor implements MatlabParserVisitor {
  Stack scopeVars; // contains identifiers that are in scope

  public Object visit(ASTforStatement node, Object data) {
    SimpleNode n;
    n = (SimpleNode) node.jjtGetChild(0);
    scopeVars.push(n.getName());

    _visitChildren(node);

    scopeVars.pop();
    return null;
  }

  public Object visit(ASTIdentifier node, Object data) {
    String name = node.getName();
    if (name.length() == 1) {
      if (scopeVars.search(name) == -1) {
        if (name.charAt(0) >= 'a' && name.charAt(0) <= 'z') {
          KpnWarning warning = new KpnWarning(
            node.getBeginLineNumber(),
            node.getBeginColumnNumber(),
            "Identifier " + name + " not in scope"
          );
          this.kpnparser.warningList.add(warning);
        }
      }
    }
    _visitChildren(node);
    return null;
  }
  ...
}
```

The Kpnvisitor is extended with a stack called scopeVars. This stack contains all the identifiers that are in the current scope. When a for statement is visited, the identifier of

that statement is added to the scope before the code within the for-loop is visited. After that the identifier is deleted from the scopeVars stack.

When an identifier is visited the scopeVars stack is checked to contain this identifier. If this is not the case a warning is added to the warninglist containing a the location of the error and a message that says to the user that this particular identifier is not defined within the current scope.

The warnings that are generated in this way are shown in the editor by creating markers in the parserChanged function.

```
public void parserChanged(SimpleNode root) {
  ...

  // KPN WARNINGS
  warnings = kpnparser.getWarningList().elements();
  KpnWarning kpnwarning;

  while ( warnings.hasMoreElements() ) {
    kpnwarning = (KpnWarning )warnings.nextElement();

    Map map = new HashMap();
    map.put(IMarker.MESSAGE,                                        new
String(kpnwarning.getWarningText()));
    map.put(IMarker.SEVERITY, new Integer(IMarker.SEVERITY_WARNING));
    map.put(
      IMarker.LINE_NUMBER, new Integer(kpnwarning.getLineNumber()));
    map.put(IMarker.TRANSIENT, new Boolean(true));
    map.put(IMarker.LOCATION, "line " + kpnwarning.getLineNumber());

    MarkerUtilities.createMarker(original, map, IMarker.PROBLEM);
  }

  ...
}
```

2.8 Parser scheduler

To keep the performance of the application in line with the user requirements a parser scheduler is added that delays the parsing. When a user starts typing a new thread is started that will sleep for 1000 milliseconds, each time a new keystoke is detected this value is reset. When the user stops typing the parsing is done.

```
public class ParserScheduler {
  public ParsingThread parsingThread;
  private PyParser parser;

  public void parseNow(boolean force) {
    if(!force){
      if(state != STATE_WAITING_FOR_ELAPSE && state != STATE_DOING_PARSE){
        //waiting or parse later
        state = STATE_WAITING_FOR_ELAPSE; // the parser will reset it later
        timeLastParse = System.currentTimeMillis();
        parser.reset(false, TIME_TO_PARSE_LATER);
        checkCreateAndStartParsingThread();
      }else{
        //another request... we keep waiting until the user stops adding requests
        boolean created = checkCreateAndStartParsingThread();
        if(!created){
          parsingThread.okToGo = false;
        }
      }
    }
   ...
  }

  private boolean checkCreateAndStartParsingThread() {
    if(parsingThread == null){
      parsingThread = new ParsingThread(this);
      parsingThread.setPriority(Thread.MIN_PRIORITY); //parsing is low priority
      parsingThread.start();
      return true;
    }
    return false;
  }

  public void reparseDocument() {
    parser.reparseDocument();
  }

  public int getIdleTimeRequested() {
    return parser.getIdleTimeRequested();
  }
}
```

Each time parseNow is called there is a check in which state the scheduler is. If the scheduler is not currently waiting or parsing a thread is created that will wait for some time and then do the parsing. If there is a thread that is waiting or doing the parsing the scheduler will keep waiting. In this way the parser is not started on every keystroke which prevents a drop in performance when using the editor.

Conclusions

In this case study an editor for nested loop programs was build on top of the very popular Eclipse platform. This platform encourages the rapid development of integrated features based on a plug-in model. Although this particular project has not been an example of rapid application development, we can conclude that the Eclipse platform certainly provides everything that is needed for the rapid development of an editor like the NLP-editor. Significant new features were added and integrated without impact to other tools. The integration of the editor with the parser is possible because of the use of a visitor pattern. The native support for a visitor in the Java Compiler Compiler allows for easy integration of the parser in the Eclipse platform that provides all the visual and navigation aids that are required in a modern editor.