Abstract Syntax Tree

Summary

The Abstract Syntax Tree is the base framework for many powerful tools of the Eclipse IDE, including refactoring, Quick Fix and Quick Assist. The Abstract Syntax Tree maps plain Java source code in a tree form. This tree is more convenient and reliable to analyse and modify programmatically than text-based source. This article shows how you can use the Abstract Syntax Tree for your own applications.

By Thomas Kuhn, Eye Media GmbH
Olivier Thomann, IBM Ottawa Lab

November 20, 2006

Introduction

Are you wondering how Eclipse is doing all the magic like jumping conveniently to a declaration, when you press "F3" on a reference to a field or method? Or how "Replace in file" solidly detects the declaration and all the references to the local variable and modifies them synchronously?

Well, these—and a big portion of the other source code modification and generation tools—are based upon the Abstract Syntax Tree (AST). The AST is comparable to the DOM tree model of an XML file. Just like with DOM, the AST allows you to modify the tree model and reflects these modifications in the Java source code.

This article refers to an example application which covers most of the interesting AST-related topics. Let us have a look at the application that was built to illustrate this article:

Example Application

According to Java Practices [4], you should not declare local variables before using them. The goal of our application will be to detect contradicting variable declarations and to move them to their correct place. There are three cases our application has to deal with:

1. *Removal of unnecessary declaration.*
   If a variable is declared and initialized, only to be overridden by another assignment later on, the first declaration of the variable is an *unnecessary declaration.*

2. *Move of declaration.*
   If a variable is declared, and not immediately referenced within the following statement, this variable declaration has to be moved. The correct place for the declaration is the line before it is first referenced.

3. *Move of declaration of a variable that is referred to from within different blocks.* This is a subcase of case 2. Imagine that a variable is used in both a try- and a catch clause. Here the declaration cannot be moved right before the first reference in the try-clause, since then it would not be declared in the catch-clause. Our application has to deal with that and has to move the declaration to the best possible place, which would be here one line above the try-clause.

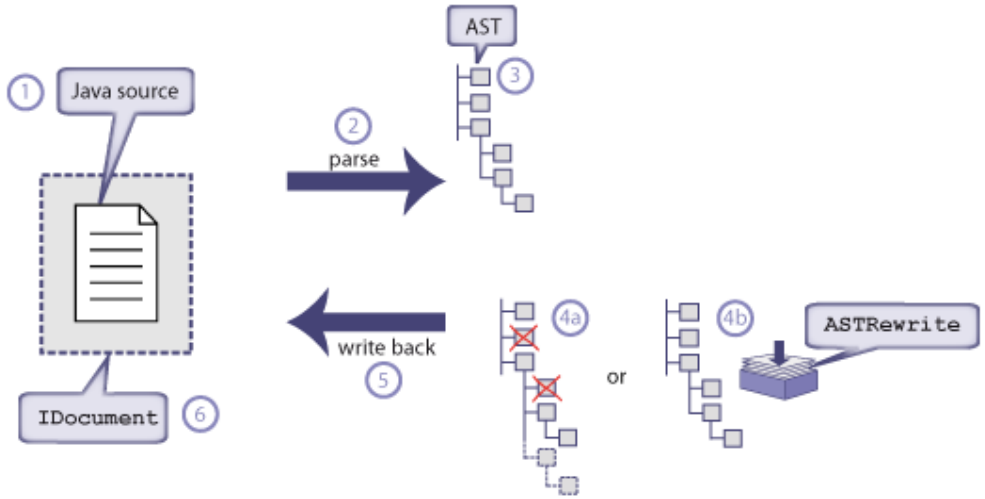
In Appendix A, *Code Fragments for Example Application Cases* code snippets to each of these cases are provided.

You can import the example application into your workspace [1] or install the plug-in using the Eclipse Update Manager [2].

Workflow

A typical workflow of an application using AST looks like this:

Figure 1. AST Workflow



1. *Java source*: To start off, you provide some source code to parse. This source code can be supplied as a Java file in your project or directly as a `char[]` that contains Java source
2. *Parse*: The source code described at 1 is parsed. All you need for this step is provided by the class `org.eclipse.jdt.core.dom.ASTParser`. See the section called “Parsing source code”.
3. The *Abstract Syntax Tree* is the result of step 2. It is a tree model that entirely represents the source you provided in step 1. If requested, the parser also computes and includes additional symbol resolved information called "bindings".

4. *Manipulating the AST*: If the AST of point 3 needs to be changed, this can be done in two ways:

   a. By directly modifying the AST.
   b. By noting the modifications in a separate protocol. This protocol is handled by an instance of `ASTRewrite`.

   See more in the section called “How to Apply Changes”.

5. *Writing changes back*: If changes have been made, they need to be applied to the source code that was provided by 1. This is described in detail in the section called “Write it down”.
6. `IDocument`: Is a wrapper for the source code of step 1 and is needed at point 5

The Abstract Syntax Tree (AST)

As mentioned, the Abstract Syntax Tree is the way that Eclipse looks at your source code: every Java source file is entirely represented as tree of AST nodes. These nodes are all subclasses of `ASTNode`. Every subclass is specialized for an element of the Java Programming Language. E.g. there are nodes for method declarations (`MethodDeclaration`), variable declaration (`VariableDeclarationFragment`), assignments and so on. One very frequently used node is `SimpleName`. A `SimpleName`
is any string of Java source that is not a keyword, a Boolean literal (`true` or `false`) or the `null` literal. For example, in `i = 6 + j;`, `i` and `j` are represented by `SimpleNames`. In `import net.sourceforge.earticleast`, `net sourceforge` and `earticleast` are mapped to `SimpleNames`.

All AST-relevant classes are located in the package `org.eclipse.jdt.core.dom` of the `org.eclipse.jdt.core` plug-in.

To discover how code is represented as AST, the AST Viewer plug-in [5] is a big help: Once installed you can simply mark source code in the editor and let it be displayed in a tree form in the AST Viewer view.

Parsing source code

Most of the time, an AST is not created from scratch, but rather parsed from existing Java code. This is done using the `ASTParser`. It processes whole Java files as well as portions of Java code. In the example application the method `parse(ICompilationUnit unit)` of the class `AbstractASTArticle` parses the source code stored in the file that `unit` points to:

```
protected CompilationUnit parse(ICompilationUnit unit) {
        ASTParser parser = ASTParser.newParser(AST.JLS3);
        parser.setKind(ASTParser.K_COMPILATION_UNIT);
        parser.setSource(unit); // set source
        parser.setResolveBindings(true); // we need bindings later on
        return (CompilationUnit) parser.createAST(null /* IProgressMonitor */); // parse
}
```

With `ASTParser.newParser(AST.JLS3)`, we advise the parser to parse the code following to the Java Language Specification, Third Edition. JLS3 includes all Java Language Specifications up to the new syntax introduced in Java 5. With the update of Eclipse towards JLS3, changes have been made to the AST API. To preserve compatibility, the `ASTParser` can be run in the deprecated JLS2 mode.

`parser.setKind(ASTParser.K_COMPILATION_UNIT)` tells the parser, that it has to expect an `ICompilationUnit` as input. An `ICompilationUnit` is a pointer to a Java file. The parser supports five kinds of input:

Entire source file: The parser expects the source either as a pointer to a Java file (which means as an `ICompilationUnit`, see the section called “Java Model”) or as `char[]`.

- `K_COMPILATION_UNIT`

Portion of Java code: The parser processes a portion of code. The input format is `char[]`.

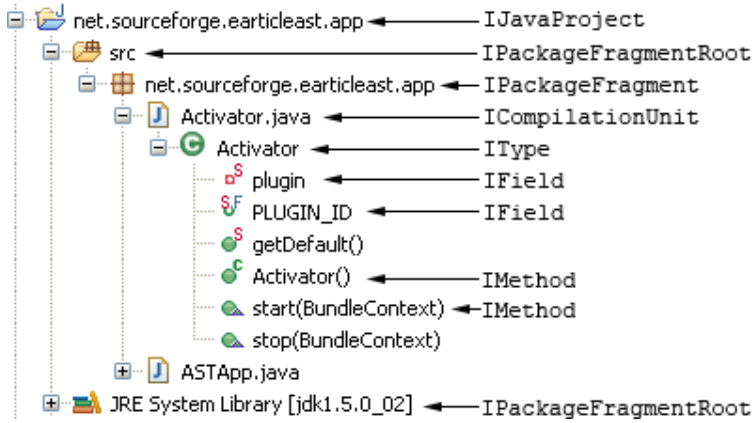
- `K_EXPRESSION`: the input contains a Java expression. E.g. `new String()`, `4+6` or `i`.
- `K_STATEMENTS`: the input contains a Java statement like `new String();` or `synchronized (this) { ... }`.
- `K_CLASS_BODY_DECLARATIONS`: the input contains elements of a Java class like method declarations, field declarations, static blocks, etc.

Java Model

The Java Model is a whole different story. It is out of scope of this article to dive deep into its details within. The parts looked at will be the ones which intersect with the AST. The motivation to discuss it here is, to use it as an entry point to build an Abstract Syntax Tree of a source file. Remember, the `ICompilationUnit` is one of the possible parameters for the AST parser.

The Java Model represents a Java Project in a tree structure, which is visualized by the well known "Package Explorer" view:

Figure 2. Java Model Overview



The nodes of the Java Model implement one of the following interfaces:

- `IJavaProject`: Is the node of the Java Model and represents a Java Project. It contains `IPackageFragmentRoots` as child nodes.
- `IPackageFragmentRoot`: can be a source or a class folder of a project, a `.zip` or a `.jar` file. `IPackageFragmentRoot` can hold source or binary files.
- `IPackageFragment`: A single package. It contains `ICompilationUnits` or `IClassFiles`, depending on whether the `IPackageFragmentRoot` is of type source or of type binary. Note that `IPackageFragment` are not organized as parent-children. E.g. `net.sf.a` is not the parent of `net.sf.a.b`. They are two independent children of the same `IPackageFragmentRoot`.
- `ICompilationUnit`: a Java source file.
- `IImportDeclaration`, `IType`, `IField`, `IInitializer`, `IMethod`: children of `ICompilationUnit`. The information provided by these nodes is available from the AST, too.

In contrast to the AST, these nodes are lightweight handles. It costs much less to rebuild a portion of the Java Model than to rebuild an AST. That is also one reason why the Java Model is not only defined down to the level of `ICompilationUnit`. There are many cases where complete information, like that provided by the AST, is not needed. One example is the Outline view: this view does not need to know the contents of a method body. It is more important that it can be rebuilt fast, to keep in sync with its source code.

There are different ways to get an `ICompilationUnit`. The example applications are launched as actions from the package tree view. This is quite convenient: only add an `objectContribution` extension to the point `org.eclipse.ui.popupMenus`. By choosing `org.eclipse.jdt.core.ICompilationUnit` as `objectClass`, the action will be only displayed in the context menu of a compilation unit. Have a look at the example application's `plugin.xml`. The compilation unit then can be retrieved from the `ISelection`, that is passed to the action's delegate (in the example, this is `ASTArticleActionDelegate`).

Another, programmatic, approach is to get the project handle from the IDE and to look for the compilation unit. This can be done by either step down the Java Model tree to collect the desired `ICompilationUnits`. Or, if the qualified name of a type within the compilation unit is known, by calling the `findType()` of the Java project:

```
IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
IProject project = root.getProject("someJavaProject");
project.open(null /* IProgressMonitor */);

IJavaProject javaProject = JavaCore.create(project);
IType lwType = javaProject.findType("net.sourceforge.earticleast.app.Activator");
ICompilationUnit lwCompilationUnit = lwType.getCompilationUnit();
```

How to find an AST Node

Even a simple "Hello world" program results in a quite complex tree. How does one get the `MethodInvocation` of that `println("Hello World")`? Scanning all the levels is a possible, but not very convenient.

There is a better solution: every `ASTNode` allows querying for a child node by using a visitor (visitor pattern [6]). Have a look at `ASTVisitor`. There you'll find for every subclass of `ASTNode` two methods, one called `visit()`,

the other called `endVisit()`. Further, the `ASTVisitor` declares these two methods: `preVisit(ASTNode node)` and `postVisit(ASTNode node)`.

The subclass of `ASTVisitor` is passed to any node of the AST. The AST will recursively step through the tree, calling the mentioned methods of the visitor for every AST node in this order (for the example of a `MethodInvocation`):

- `preVisit(ASTNode node)`
- `visit(MethodInvocation node)`
- ... now the children of the method invocation are recursively processed if visit returns true
- `endVisit(MethodInvocation node)`
- `postVisit(ASTNode node)`

In our example application, the `LocalVariableDetector` is a subclass of `ASTVisitor`. It is used, amongst other things, to collect all local variable declarations of a compilation unit:

```
public boolean visit(VariableDeclarationStatement node) {
        for (Iterator iter = node.fragments().iterator(); iter.hasNext();) {
                VariableDeclarationFragment fragment = (VariableDeclarationFragment) iter.next(
                // ... store these fragments somewhere
        }
        return false; // prevent that SimpleName is interpreted as reference
}
```

If `false` is returned from `visit()`, the subtree of the visited node will not be considered. This is to ignore parts of the AST.

In the example, `process(CompilationUnit unit)` is called from the outside to start visiting the compilation unit. The method is fairly simple:

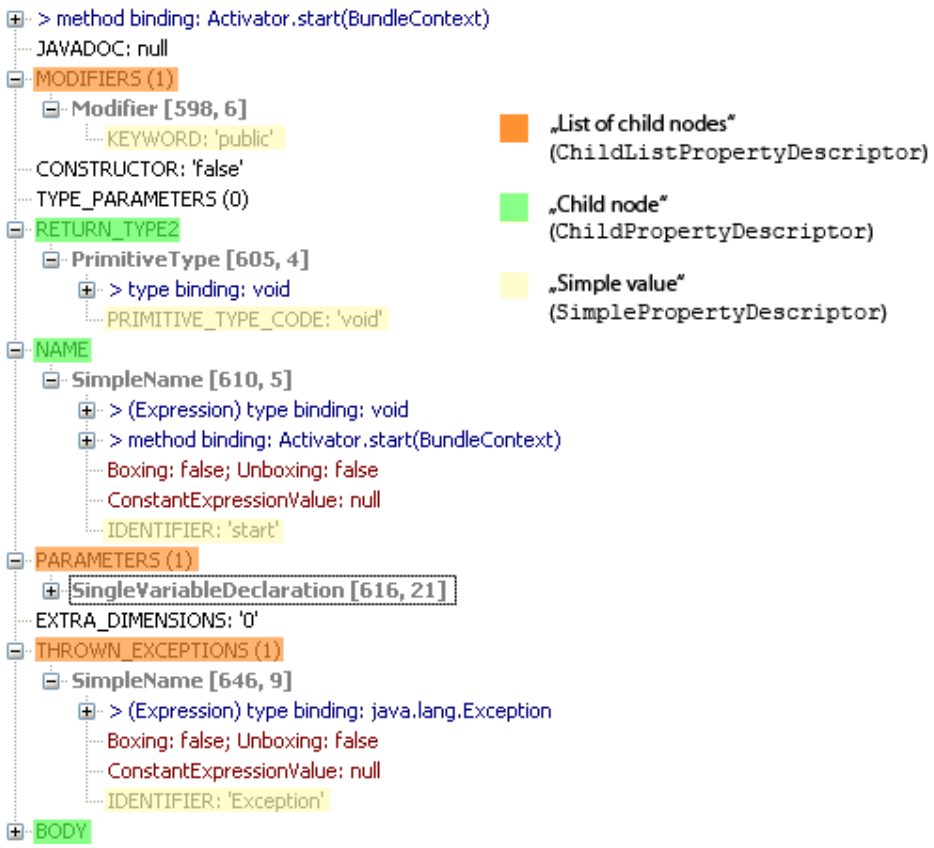
```
public void process(CompilationUnit unit) {
        unit.accept(this);
}
```

Obtaining Information from an AST Node

Every subclass of `ASTNode` contains specific information for the Java element it represents. E.g. a `MethodDeclaration` will contain information about the name, return type, parameters, etc. The information of a node is referred as *structural properties*. Let us have a closer look at the characteristics of the structural properties. Beneath you see the properties of this method declaration:

```
public void start(BundleContext context) throws Exception {
        super.start(context);
}
```

Figure 3. Structural properties of a method declaration

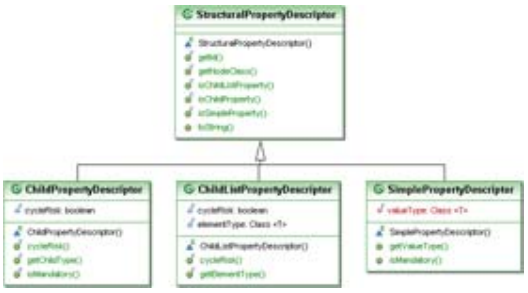


Access to the values of a node's structural properties can be made using static or generic methods:

1. *static methods*: every node offers methods to access its properties: e.g. `getName()`, `exceptions()`, etc.
2. *generic method*: ask for a property value using the `getStructuralProperty(StructuralPropertyDescriptor property)` method. Every AST subclass defines a set of `StructuralPropertyDescriptors`, one for every structural property. The `StructuralPropertyDescriptor` can be accessed directly on the class to which they belong: e.g. `MethodDeclaration.NAME_PROPERTY`. A list of all available `StructuralPropertyDescriptors` of a node can be retrieved by calling the method `structuralPropertiesForType()` on any instance of `ASTNode`.

The structural properties are grouped into three different kinds: properties that hold simple values, properties which contain a single child AST node and properties which contain a list of child AST nodes.

Figure 4. StructuralPropertyDescriptor and subclasses



view full size

- `SimplePropertyDescriptor`: The value will be a `String`, a primitive value wrapper for either `Integer` or `Boolean` or a basic AST constant. For a list of all possible value classes of a simple property, see Appendix C, *Simple properties value classes*

- `ChildPropertyDescriptor`: The value will be a node, an instance of an `ASTNode` subclass
- `ChildListPropertyDescriptor`: The value will be a `List` of AST nodes

Bindings

The AST, as far as we know it, is just a tree-form representation of source code. Every element of the source code is mapped to a node or a subtree. Looking at a reference to a variable, let's say `i`, is represented by an instance of `SimpleName` with "i" as `IDENTIFIER` property-value. Bindings go one step further: they provide extended resolved information for several elements of the AST. About the `SimpleName` above they tell us that it is a reference to a local variable of type int.

Various subclasses of `ASTNode` have binding information. It is retrieved by calling `resolveBinding()` on these classes. There are cases where more than one binding is available: e.g. the class `MethodInvocation` returns a binding to the method that is invoked (`resolveMethodBinding()`). Furthermore a binding to the return type of the method (`resolveTypeBinding()`) and information about whether the method invocation is involved into a boxing (`resolveBoxing()`) or unboxing (`resolveUnboxing()`) is offered.

Since evaluating bindings is costly, the binding service has to be explicitly requested at parse time. This is done by passing `true` the method `ASTParser.setResolveBindings()` before the source is being parsed.

```
int i = 7;
System.out.println("Hello!");
int x = i * 2;
```

the reference of the variable `i` is represented by a `SimpleName`. Without bindings you would not know nothing more than this:

Bindings provide more information:

Bindings allow you to comfortably find out to which declaration a reference belongs, as well as to detect whether two elements are references to the same element: if they are, the bindings returned by reference-nodes and declaration-nodes are identical. For example, all `SimpleNames` that represent a reference to a local variable `i` return the same instance of `IVariableBinding` from `SimpleName.resolveBindings()`. The declaration node, `VariableDeclarationFragment.resolveBinding()`, returns the same instance of `IVariableBinding`, too. If there is another declaration of a local variable `i` (within another method or block), another instance of `IVariableBinding` is returned. Confusions caused by equally named elements are avoided if bindings are used to identify an element (variable, method, type, etc.).

The example application uses variable bindings for this purpose: for every declaration, a manager object is created and added to a map. The binding of the declaration figures as key, the created manager as value.

```
for (Iterator iter = node.fragments().iterator(); iter.hasNext();) {
        VariableDeclarationFragment fragment = (VariableDeclarationFragment) iter.next();

        IVariableBinding binding = fragment.resolveBinding();
        VariableBindingManager manager = new VariableBindingManager(fragment);
        localVariableManagers.put(binding, manager);
}
```

Then, if a `SimpleName` is visited, the application checks, whether the binding of this `SimpleName` occurs in the map. If so, the `SimpleName` is a reference to a local variable.

```
public boolean visit(SimpleName node) {
        IBinding binding = node.resolveBinding();
        if (localVariableManagers.containsKey(binding)) {
                VariableBindingManager manager = localVariableManagers.get(binding);
                manager.variableRefereneced(node);
        }
}
```

Error Recovery

Since Eclipse 3.2, the DOM/AST support has the ability to recover from code with syntax errors. In order to trigger it, you need to use the `ASTParser#setStatementsRecovery()`. As its name says it, the error recovery is

done at the statement level. This however implies that the code is not completely messed up as it cannot recover from all kinds of syntax errors. This being said a missing semi-colon is no longer an problem anymore to retrieve the statements of a method. Let's take an example:

```
public static void main(String[] args) {
        System.out.print("Hello");
        System.out.print(", ")
        System.out.println("World!");
}
```

With this source code, before Eclipse 3.2, the method body would be empty. Now with Eclipse 3.2 and its error recovery it is possible to get a `RECOVERED` expression statement that contains the method invocation. Not only can you have nodes that can be traversed by a visitor, but in some cases it is even possible to get bindings for the `RECOVERED` statement. So in the example, you would end up with two statements that have no problems and one `RECOVERED` statement.

Any instance of a subclass of `ASTNode` can be tagged with some bits that provide information about the way the node was created. This bits can be retrieved by calling the method `ASTNode#getFlags()`. The whole list of bits are:

- `MALFORMED`: indicates node is syntactically malformed
- `ORIGINAL`: indicates original node created by ASTParser
- `PROTECT`: indicates node is protected from further modification
- `RECOVERED`: indicates node or a part of this node is recovered from source that contains a syntax error

So when traversing a AST tree, you might want to check the flags of the traversed nodes. A node flagged as `RECOVERED` might not contain the expected nodes.

How to Apply Changes

This section will show how to modify an AST and how to store these modifications back into Java source code.

New AST nodes may have to be created. New nodes are created by using the class `org.eclipse.jdt.core.dom.AST` (here `AST` it is the name of an actual class. Do not confuse with the abbreviation "AST" used within this article). Have a look at this class: it offers methods to create every AST node type. An instance of `AST` is created when source code is parsed. This instance can be obtained from every node of the tree by calling the method `getAST()`. The newly created nodes can only be added to the tree that class `AST` was retrieved from.

Often it is convenient to reuse an existing subtree of an AST and maybe just change some details. AST nodes cannot be re-parented, once connected to an AST, they cannot be attached to a different place of the tree. Though it is easy to create a copy from a subtree: `(Expression) ASTNode.copySubtree(ast, manager.getInitializer())`. The parameter `ast` is the target `AST`. This instance will be used to create the new nodes. That allows copying nodes from another `AST` (established by another parser run) into the current `AST` domain.

There are two APIs to track modifications on an AST: either you can directly modify the tree or you can make use of a separate protocol, managed by an instance of `ASTRewrite`. The latter, using the `ASTRewrite`, is the more sophisticated and preferable way. The changes are noted by an instance of `ASTRewrite`, the original AST is left untouched. It is possible to create more than one instance of `ASTRewrite` for the same AST, which means that different change logs can be set up. "Quick Fix" makes use of this API: this is how for every Quick Fix proposal a preview is created.

Example 1. Protocolling changes to a AST by using `ASTRewrite` .

```
 rewrite = ASTRewrite.create(unit.getAST()); // unit instance of CompilationUnit
// ...
VariableDeclarationStatement statement = createNewVariableDeclarationStatement(manager, ast);
int firstReferenceIndex = getFirstReferenceListIndex(manager, block);
ListRewrite statementsListRewrite = rewrite.getListRewrite(block, Block.STATEMENTS_PROPERTY);
statementsListRewrite.insertAt(statement, firstReferenceIndex, null);
```

The example shows, how a child is added to a child list property value. If a single-child property is set, no list rewrite is necessary. For example, to set the name of a `MethodInvocation`, the code would look like this:

```
rewrite.set(methodInvocation, MethodInvocation.NAME_PROPERTY, newName, null);
```

or

```
rewrite.replace(methodInvocation.getName() /* old name node*/, newName, null)
```

To set a simple property value, call `set()` like shown above.

Let us have a look at the second way to change an AST. Instead of tracking the modifications in separate protocols, we directly modify the AST. The only thing that has to done before modifying the first node is to turn on the change recording by calling `recordModifications()` on the root of the AST, the `CompilationUnit`. Internally changes are logged to an `ASTRewrite` as well, but this happens hidden to you.

Example 2. Modifying an AST directly.

```
unit.recordModifications();
// ...
VariableDeclarationStatement statement = createNewVariableDeclarationStatement(manager, ast);
block.statements().add(firstReferenceIndex, statement);
```

The next section will tell how to write the modifications back into Java source code.

Write it down

Once you have tracked changes, either by using `ASTRewrite` or by modifying the tree nodes directly, these changes can be written back into Java source code. Therefore a `TextEdit` object has to be created. Here we leave the code related area of the AST, and enter a text based environment. The `TextEdit` object contains character based modification information. It is part of the `org.eclipse.text` plug-in.

How to obtain the `TextEdit` object differs for the two mentioned ways only slightly:

- If you used `ASTRewrite`, ask the `ASTRewrite` instance for the desired `TextEdit` object by calling `rewriteAST(IDocument, Map)`.
- If you changed the tree nodes directly, the `TextEdit` object is created by calling `rewrite(IDocument document, Map options)` on `CompilationUnit`.

The first parameter, `document`, contains the source code that will be modified. The content of this container is the same code that you fed into the `ASTParser`. The second parameter is a map of options for the source code formatter. To use the default options, pass `null`.

Obtaining an `IDocument` if you parsed source code from a `String` is easy: create an object of the class `org.eclipse.jface.text.Document` and pass the code string as constructor parameter.

If you initially parsed an existing Java source file and would like to store the changes back into this file, things get a little bit more tricky. You should not directly write into this file, since you might not be the only editor that is manipulating this source file. Within Eclipse, Java editors do not write directly on a file resource, but on a shared working copy instead.

```
ITextFileBufferManager bufferManager = FileBuffers.getTextFileBufferManager(); // get the buffe
IPath path = unit.getJavaElement().getPath(); // unit: instance of CompilationUnit
try {
        bufferManager.connect(path, null); // (1)
        ITextFileBuffer textFileBuffer = bufferManager.getTextFileBuffer(path);
        // retrieve the buffer
        IDocument document = textFileBuffer.getDocument(); (2)
        // ... edit the document here ...

  // commit changes to underlying file
```

```
        textFileBuffer
                .commit(null /* ProgressMonitor */, false /* Overwrite */); // (3)

} finally {
        bufferManager.disconnect(path, null); // (4)
}
```

1. Connect a path to the buffer manager. After that call, the document for the file described by `path` can be obtained.
2. Ask the buffer for the working copy by calling `getTextFileBuffer`. From the `ITextFileBuffer` we get the `IDocument` instance we need.
3. Store changes to the underlying file.
4. Disconnect the path. Do not modify the document after this call.

Managing Comments

One of the most frustrating part of modifying an AST is the comment handling. The method `CompilationUnit#getCommentList()` is used to return the list of comments located in the compilation unit in the ascendant order. Unfortunately, this list cannot be modified. This means that even if the AST Rewriter is used to add a comment inside a compilation unit, the new comment would not appear inside the comments' list.

In order to add a comment the following code snippet can be used:

```
CompilationUnit astRoot= ... ; // get the current compilation unit

ASTRewrite rewrite= ASTRewrite.create(astRoot.getAST());
Block block= ((TypeDeclaration) astRoot.types().get(0)).getMethods()[0].getBody();
ListRewrite listRewrite= rewrite.getListRewrite(block, Block.STATEMENTS_PROPERTY);
Statement placeHolder= rewrite.createStringPlaceholder("//mycomment", ASTNode.EMPTY_STATEMENT);
listRewrite.insertFirst(placeHolder, null);

textEdits= rewrite.rewriteAST(document, null);
textEdits.apply(document);
```

The methods `CompilationUnit#getExtendedLength(ASTNode)` and `CompilationUnit#getExtendedStartPosition(ASTNode)` can be used to retrieve the range of a node that would contains preceding and trailing comments and whitespaces.

If a comment is a javadoc comment defined prior to a method declaration, a field declaration or a type declaration (including enum, annotations, interfaces and classes), they can also be retrieved by calling the `getJavadoc()` method on the corresponding declaration.

Conclusions

This article has shown how to use the Eclipse AST for static code analysis and code manipulation issues. It touched the Java Model, explained Bindings and showed how to store changes made to the AST back into Java source code.

For remarks, questions, etc. enter a comment in the bugzilla entry of this article [7].

Resources

[1]
Download the Packed Example Project. Use the option "Existing Projects into Workspace" from the "Import" Wizard to add it to your workspace.

[2]
To install the plug-in, obtain using the Eclipse Update Manager. Update Site: http://earticleast.sourceforge.net/update.

[3] Java Tool Smithing, Extending the Eclipse Java Development Tools .

[4] Java Practices .

[5] AST Viewer Plug-in .

[6] Wikipedia: Visitor Pattern .

[7] AST Article bugzilla entry .

A. Code Fragments for Example Application Cases

In the introduction, three typical cases for our example application have been presented (see the section called "Example Application"). Clarifying code before / after code snippets follow to further clarify these cases.

1. *Removal of unnecessary declaration.*

   Before:

   ```
   int x = 0;
   ...
   x = 2 * 3;
   ```

   After:

   ```
   ...
   int x = 2 * 3;
   ```

2. *Move of declaration.*

   Before:

   ```
   int x = 0;
   ...
   System.out.println(x);
   ...
   x = 2 * 3;
   ```

   After:

   ```
   ...
   int x = 0;
   System.out.println(x);
   ...
   x = 2 * 3;
   ```

3. *Move of a declaration of a variable, that is used within different blocks.*

   Before:

   ```
   int x = 0;
   ...
   try {
           x = 2 * 3;
   } catch (...) {
           System.out.println(x);
   }
   ```

   After:

   ```
   ...
   int x = 0;
   try {
           x = 2 * 3;
   } catch (...) {
           System.out.println(x);
   }
   ```

B. Complete list of bindings

- `IAnnotationBinding`
- `IMemberValuePairBinding`
- `IMethodBinding`
- `IPackageBinding`
- `ITypeBinding`
- `IVariableBinding`

C. Simple properties value classes

Below the list of all classes of which simple property values can be instance of (in Eclipse version 3.2).

- `boolean`
- `int`
- `String`
- `Modifier.ModifierKeyword`
- `Assignment.Operator`
- `InfixExpression.Operator`
- `PostfixExpression.Operator`
- `PrefixExpression.Operator`
- `PrimitiveType.Code`
