A
Report
on
Abstract Syntax Tree
of
Java Development Tooling Library

EE 564 – Enterprise Software Development

Submitted To:
Professor Daqing Hou
Clarkson University

Submitted By:
Chandan Raj Rupakheti
Clarkson University

Date: March 26, 2007

Abstract

Java Document Object Model (JDOM) also known as Abstract Syntax Tree (AST) provides a tree structure for the Java program elements for static as well as runtime operation on Java source code. The library helps manipulation of Java source code as well as generation and compilation of the created source code in Eclipse framework. Provided under Java Development Tooling library in Eclipse, it works as primary tool for static program analysis of Java source code.

This report summarizes the detailed study performed on AST with the evolution trend of the framework by comparing three versions of the library. It then further illustrates on the exception handling mechanism and concludes by presenting the statistics of the explored packages and classes and provides recommendation for a new user on how to approach the problem of learning big framework like JDT AST library.

1. Introduction

The Eclipse platform is delivered with a full featured Java integrated development environment (IDE). Java development tooling (JDT) allows users to write, compile, test, debug, and edit programs written in the Java programming language. It's easiest to think of the JDT as a set of plug-ins that add Java specific behavior to the generic platform resource model and contribute Java specific views, editors, and actions to the workbench of Eclipse. Among the components of the library, AST serves as the primary component for different kinds of operation on Java source code. A new source code can be created on the fly or existing source code can be read and modified dynamically. All this is possible due to AST library. Programmer has full control on creation and manipulation of source code through this framework.

This report is divided into several section. Section 2 illustrates on JavaModel and its associated classes, Section 3 presents the evolution trend of the software by comparing three versions of this library in CVS repository, Section 4 illustrates on Exception Handling and associated classes, Section 5 contains statistics on number of packages and classes studied, Section 6 provides recommendation for new user on learning the framework and finally Section 7 concludes the report.

2. Java Model

The Java model is the set of classes that model the objects associated with creating, editing, and building a Java program. The Java model classes are defined in org.eclipse.jdt.core. These classes implement Java specific behavior for resources and further decompose Java resources into model elements [1].

2.1 Java elements

The package org.eclipse.jdt.core defines the classes that model the elements that compose a Java program. The JDT uses an in-memory object model to represent the structure of a Java program. This structure is derived from the project's class path. The model is hierarchical. Elements of a program can be decomposed into child elements [1].

The [Table 2.1] summarizes the different kinds of Java elements and [Figure 2.1] shows the segment of inheritance hierarchy of IJavaElement interface.

Element	Description
IJavaModel	Represents the root Java element, corresponding to the workspace. The parent of all projects with the Java nature. It also gives you access to the projects without the java nature.
IJavaProject	Represents a Java project in the workspace. (Child of IJavaModel)
IPackageFragmentRoot	Represents a set of package fragments, and maps the fragments to an underlying resource which is either a folder, JAR, or ZIP file. (Child of IJavaProject)
IPackageFragment	Represents the portion of the workspace that corresponds to an entire package, or a portion of the package. (Child of IPackageFragmentRoot)
ICompilationUnit	Represents a Java source (.java) file. (Child of IPackageFragment)
IPackageDeclaration	Represents a package declaration in a compilation unit. (Child of ICompilationUnit)
IImportContainer	Represents the collection of package import declarations in a compilation unit. (Child of ICompilationUnit)
IImportDeclaration	Represents a single package import declaration. (Child of IImportContainer)
IType	Represents either a source type inside a compilation unit, or a binary type inside a class file.
IField	Represents a field inside a type. (Child of IType)
IMethod	Represents a method or constructor inside a type. (Child of IType)
IInitializer	Represents a static or instance initializer inside a type. (Child of IType)
IClassFile	Represents a compiled (binary) type. (Child of IPackageFragment)
ITypeParameter	Represents a type parameter. (Not a child of any Java element, it is obtained using IType.getTypeParameter(String) or IMethod.getTypeParameter(String))

ILocalVariable	Represents a local variable in a method or an initializer. (Not a child of any Java element, it is obtained using ICodeAssist.codeSelect(int, int))

Table 2.1: Different kinds of Java Element

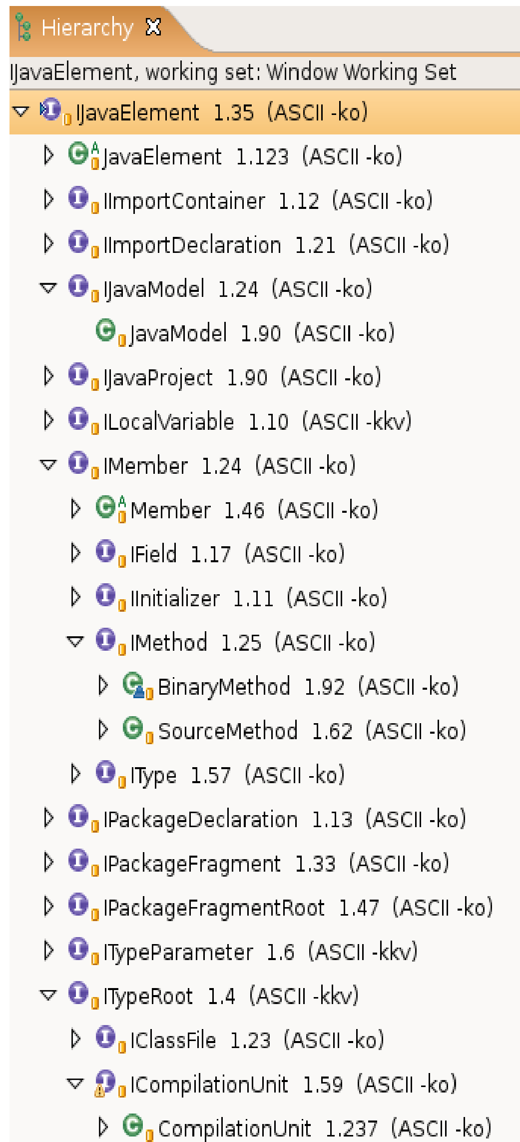

Figure 2.1: IJavaElement inheritance hierarchy

All Java elements support the IJavaElement interface. Some of the elements are shown in the Packages view. These elements implement the IOpenable interface, since they must be opened before they can be navigated. The [Figure 2.2] below shows how these elements are represented in the Packages view[1].

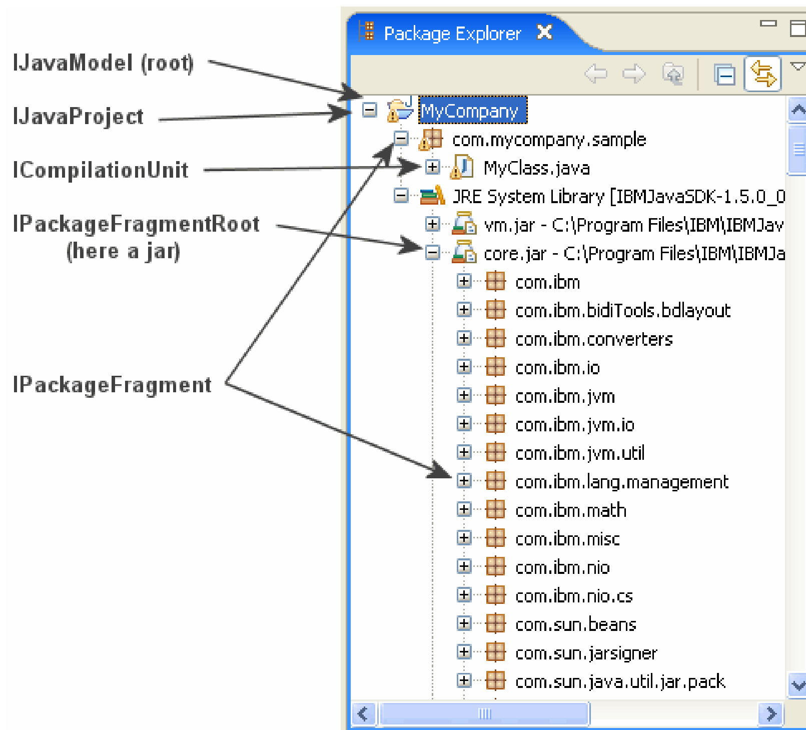

Figure 2.2: Package view of Java Elements

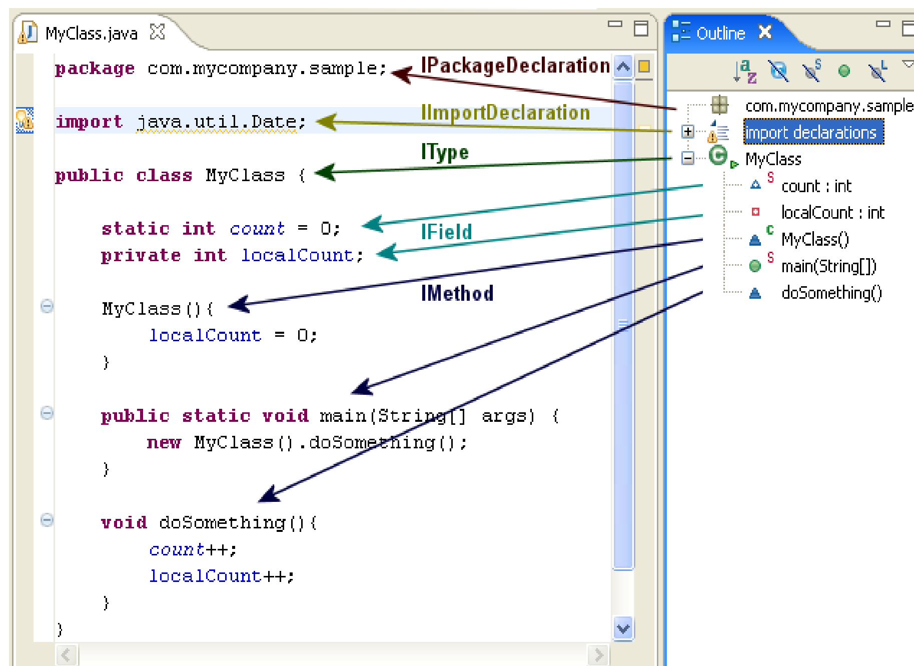

Figure 2.3: Java Compilation Unit in Outline View

The Java elements that implement IOpenable are created primarily from information found in the underlying resource files. The same elements are represented generically in the resource navigator view. Other elements correspond to the items that make up a Java compilation unit. The [Figure 2.3] above

shows a Java compilation unit and a content outliner that displays the source elements in the compilation unit[1]. These elements implement the ISourceReference interface, since they can provide corresponding source code. As these elements are selected in the content outliner, their corresponding source code is shown in the Java editor.

IJavaModel can be considered the parent of all projects in the workspace that have the Java project nature and therefore can be treated as an IJavaProject[1].

2.2 Abstract Syntax Tree (AST)

The Java DOM/AST is the set of classes that model the source code of a Java program as a structured document. It is an umbrella owner and a factory for abstract syntax tree node. See [2] for more details on factory method. An AST instance serves as the common owner of any number of AST nodes, and as the factory for creating new AST nodes owned by that instance.

Abstract syntax trees may be hand constructed by clients, using the *newTYPE* factory methods to create new nodes, and the various *setCHILD* methods to connect them together.

Each AST node belongs to a unique AST instance, called the owning AST. The children of an AST node always have the same owner as their parent node. If a node from one AST is to be added to a different AST, the subtree must be cloned first to ensures that the added nodes have the correct owning AST.

There can be any number of AST nodes owned by a single AST instance that are unparented. Each of these nodes is the root of a separate little tree of nodes. The method *ASTNode.getRoot()* navigates from any node to the root of the tree that it is contained in. Ordinarily, an AST instance has one main tree (rooted at a *CompilationUnit*), with newly-created nodes appearing as additional roots until they are parented somewhere under the main tree. One can navigate from any node to its AST instance, but not conversely.

The class *ASTParser* parses a string containing a Java source code and returns an abstract syntax tree for it. The resulting nodes carry source ranges relating the node back to the original source characters.

Compilation units created by *ASTParser* from a source document can be serialized after arbitrary modifications with minimal loss of original formatting. An example is shown in Figure 2.4.

```
Document doc = new Document("import java.util.List;\nclass X {}\n");
ASTParser parser = ASTParser.newParser(AST.JLS3);
parser.setSource(doc.get().toCharArray());
CompilationUnit cu = (CompilationUnit) parser.createAST(null);
cu.recordModifications();
AST ast = cu.getAST();
ImportDeclaration id = ast.newImportDeclaration();
id.setName(ast.newName(new String[] {"java", "util", "Set"});
cu.imports().add(id); // add import declaration at end
TextEdit edits = cu.rewrite(document, null);
UndoEdit undo = edits.apply(document);
```

Figure: 2.4: A code snippet for creating AST from ASTParser

2.3 AST Node

ASTNode is an abstract superclass of all Abstract Syntax Tree (AST) node types. An AST node represents a Java source code construct, such as a name, type, expression, statement, or declaration. Each AST node belongs to a unique AST instance, called the owning AST. The children of an AST node always have the same owner as their parent node. If a node from one AST is to be added to a different AST, the subtree must be cloned first to ensure that the added nodes have the correct owning AST[1].

When an AST node is part of an AST, it has a unique parent node. Clients can navigate upwards, from child to parent, as well as downwards, from parent to child. Newly created nodes are unparented. When an unparented node is set as a child of a node (using a *setCHILD* method), its parent link is set automatically and the parent link of the former child is set to *null*. For nodes with properties that include a list of children (for example, *Block* whose *statements* property is a list of statements), adding or removing an element to/for the list property automatically updates the parent links. These lists support the *List.set* method; however, the constraint that the same node cannot appear more than once means that this method cannot be used to swap elements without first removing the node[1].

ASTs must not contain cycles. All operations that could create a cycle detect this possibility and fail. ASTs do not contain "holes" (missing subtrees). If a node is required to have a certain property, a syntactically plausible initial value is always supplied[1].

The hierarchy of AST node types has some convenient groupings marked by abstract superclasses:

- expressions - *Expression*
- names - *Name* (a sub-kind of expression)
- statements - *Statement*
- types - *Type*
- type body declarations - *BodyDeclaration*

Abstract syntax trees may be hand constructed by clients, using the *newTYPE* factory methods to create new nodes, and the various *setCHILD* methods to connect them together.

The class *ASTParser* parses a string containing a Java source code and returns an abstract syntax tree for it. The resulting nodes carry source ranges relating the node back to the original source characters. The source range covers the construct as a whole.

Each AST node carries bit flags, which may convey additional information about the node. For instance, the parser uses a flag to indicate a syntax error. Newly created nodes have no flags set.

Each AST node is capable of carrying an open-ended collection of client-defined properties. Newly created nodes have none. *getProperty* and *setProperty* are used to access these properties.

AST nodes are thread-safe for readers provided there are no active writers. If one thread is modifying an AST, including creating new nodes or cloning existing ones, it is **not** safe for another thread to read, visit, write, create, or clone *any* of the nodes on the same AST. When synchronization is required, user should consider using the common AST object that owns the node; that is, use *synchronize (node.getAST()) {...}*.

AST also support the visitor pattern[3]; see the class *ASTVisitor* [1] for details. [Figure 2.5] shows the inheritance hierarchy of ASTNode.

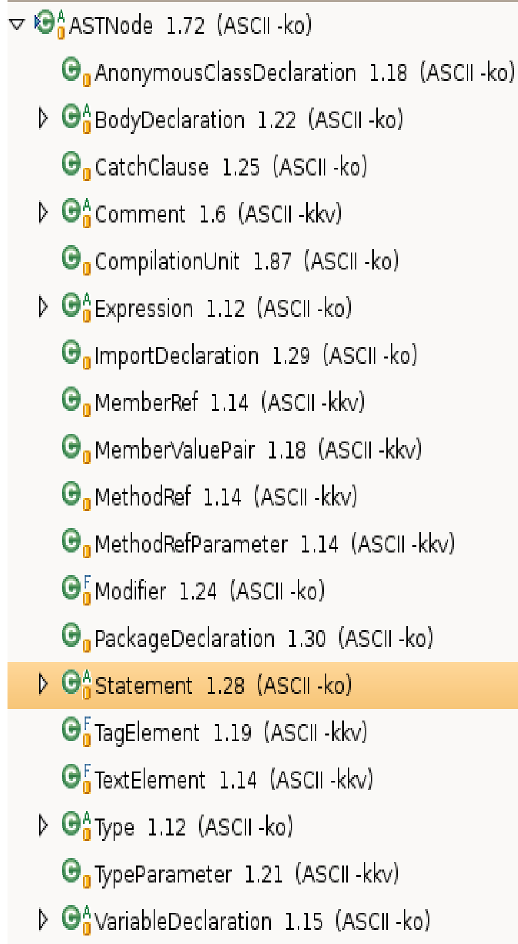

Figure a: ASTNode Hierarchy (Root)

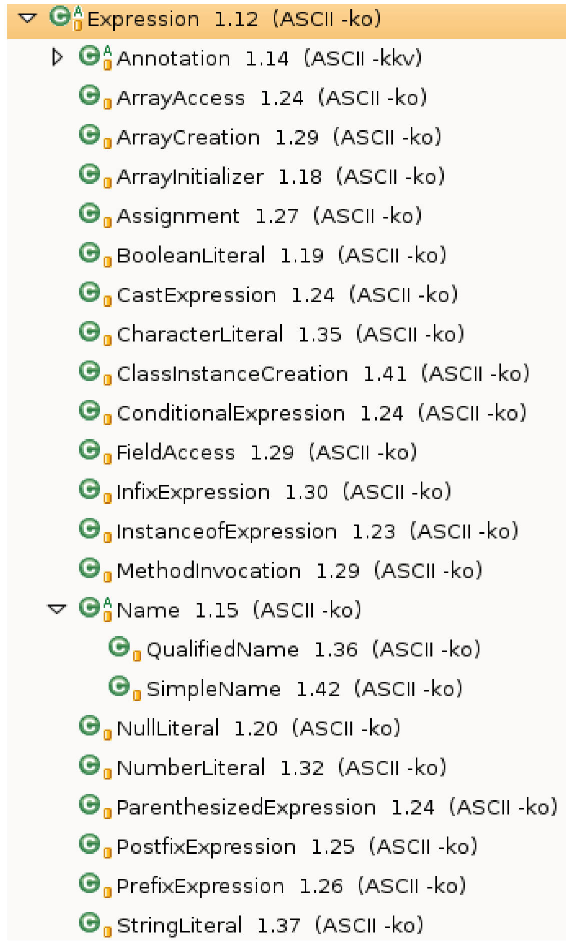

Figure d: Expression Node Hierarchy

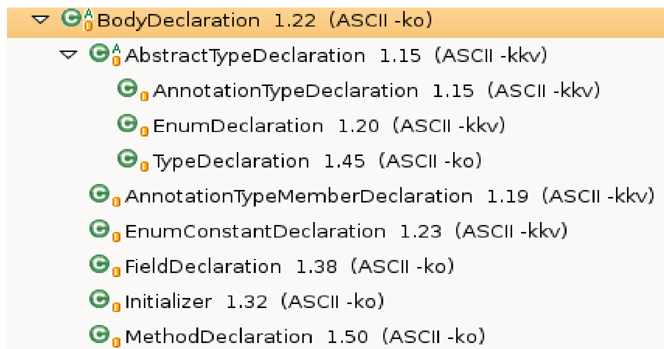

Figure b: BodyDeclaration Node Hierarchy

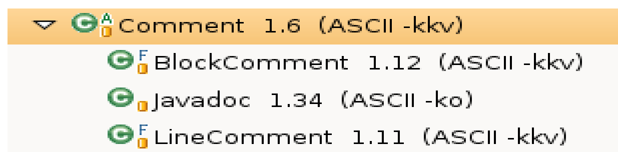

Figure c: Comment Node Hierarchy

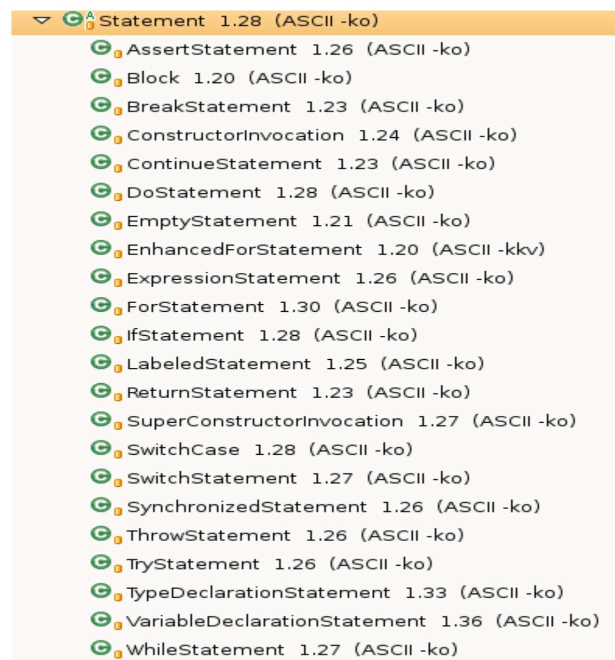

Figure e: Statement Node Hierarchy

Figure 2.5: AST Node Inheritance Hierarchy

2.3 Creating Java element from scratch

It is possible to create a CompilationUnit from scratch using the factory methods on AST. These method names start with *newNODE*. The following is an example that creates a *HelloWorld* class[1].

The first snippet is the generated output:

```
package example;
import java.util.*;
public class HelloWorld {
      public static void main(String[] args) {
            System.out.println("Hello" + " world");
      }
}
```

The following snippet is the corresponding code that generates the output.

```
AST ast = new AST();
CompilationUnit unit = ast.newCompilationUnit();
PackageDeclaration packageDeclaration = ast.newPackageDeclaration();
packageDeclaration.setName(ast.newSimpleName("example"));
unit.setPackage(packageDeclaration);
ImportDeclaration importDeclaration = ast.newImportDeclaration();
QualifiedName name = ast.newQualifiedName(
                        ast.newSimpleName("java"),
                        ast.newSimpleName("util"));
importDeclaration.setName(name);
importDeclaration.setOnDemand(true);
unit.imports().add(importDeclaration);
TypeDeclaration type = ast.newTypeDeclaration();
type.setInterface(false);
type.setModifiers(Modifier.PUBLIC);
type.setName(ast.newSimpleName("HelloWorld"));
MethodDeclaration methodDeclaration = ast.newMethodDeclaration();
methodDeclaration.setConstructor(false);
methodDeclaration.setModifiers(Modifier.PUBLIC | Modifier.STATIC);
methodDeclaration.setName(ast.newSimpleName("main"));
methodDeclaration.setReturnType(ast.newPrimitiveType(PrimitiveType.VOID));
SingleVariableDeclaration variableDeclaration = ast.newSingleVariableDeclaration();
variableDeclaration.setModifiers(Modifier.NONE);
variableDeclaration.setType(ast.newArrayType(
                        ast.newSimpleType(ast.newSimpleName("String"))));
variableDeclaration.setName(ast.newSimpleName("args"));
methodDeclaration.parameters().add(variableDeclaration);
org.eclipse.jdt.core.dom.Block block = ast.newBlock();
MethodInvocation methodInvocation = ast.newMethodInvocation();
name = ast.newQualifiedName( ast.newSimpleName("System"),
                  ast.newSimpleName("out"));
methodInvocation.setExpression(name);
methodInvocation.setName(ast.newSimpleName("println"));
InfixExpression infixExpression = ast.newInfixExpression();
infixExpression.setOperator(InfixExpression.Operator.PLUS);
StringLiteral literal = ast.newStringLiteral();
literal.setLiteralValue("Hello");
infixExpression.setLeftOperand(literal);
literal = ast.newStringLiteral();
literal.setLiteralValue(" world");
infixExpression.setRightOperand(literal);
methodInvocation.arguments().add(infixExpression);
ExpressionStatement expressionStatement =
ast.newExpressionStatement(methodInvocation);
```

```
block.statements().add(expressionStatement);
methodDeclaration.setBody(block);
type.bodyDeclarations().add(methodDeclaration);
unit.types().add(type);
```

2.4 Adding new concrete AST node types

There are several things that need to be changed when a new concrete AST node type is added to AST hierarchy. Lets call it *FooBar*. Here are the steps that should be considered:

1. Create the FooBar AST node type class. The most effective way to do this is to copy a similar existing concrete node class to get a template that includes all the framework methods that must be implemented.
2. Add node type constant ASTNode.FOO_BAR. Node constants are numbered consecutively. Add the constant after the existing ones.
3. Add entry to ASTNode.nodeClassForType(int).
4. Add AST.newFooBar() factory method.
5. Add ASTVisitor.visit(FooBar) and endVisit(FooBar) methods.
6. Add ASTMatcher.match(FooBar,Object) method.
7. Ensure that SimpleName.isDeclaration() covers FooBar nodes if required.
8. Add NaiveASTFlattener.visit(FooBar) method to illustrate how these nodes should be serialized.
9. Update the AST test suites. The next steps are to update AST.parse* to start generating the new type of nodes, and ASTRewrite to serialize them back.

3. Evolution

This part of the report focuses on the evolution trend of the AST library. The comparison is made between the latest version (Mar 22, 2006-head) with version 300 and version 235 of org.eclipse.jdt.core library from CVS repository.

3.1 AST Re-write

AST re-write is needed for modifying code by describing changes to AST nodes. The AST rewriter collects descriptions of modifications to nodes and translates these descriptions into text edits that can then be applied to the original source. The key thing is that this is all done without actually modifying the original AST, which has the virtue of allowing one to entertain several alternate sets of changes on the same AST. The rewrite infrastructure tries to generate minimal text changes, preserve existing comments and indentation, and follow code formatter settings.

The following code snippet illustrated usage of this class:

```
Document document = new Document("import java.util.List;\nclass X {}\n");
ASTParser parser = ASTParser.newParser(AST.JLS3);
parser.setSource(doc.get().toCharArray());
CompilationUnit cu = (CompilationUnit) parser.createAST(null);
AST ast = cu.getAST();
ImportDeclaration id = ast.newImportDeclaration();
id.setName(ast.newName(new String[] {"java", "util", "Set"}));
ASTRewrite rewriter = ASTRewrite.create(ast);
TypeDeclaration td = (TypeDeclaration) cu.types().get(0);
ITrackedNodePosition tdLocation = rewriter.track(td);
ListRewrite lrw = rewriter.getListRewrite(cu, CompilationUnit.IMPORTS_PROPERTY);
lrw.insertLast(id, null);
TextEdit edits = rewriter.rewriteAST(document, null);
UndoEdit undo = edits.apply(document);
assert "import java.util.List;\nimport java.util.Set;\nclass X {}".equals(
           doc.get().toCharArray());
// tdLocation.getStartPosition() and tdLocation.getLength()
// are new source range for "class X {}" in doc.get()
```

The versions 235 and 300 do not have AST Rewrite functionality while the latest version have it. The packages *org.eclipse.jdt.core.dom.rewrite.** and *org.eclipse.jdt.internal.core.dom.rewrite.** contains this functionality that are missing in version 235 and 300 inside *org.eclipse.jdt.core.dom.** package.

3.2 ASTNode

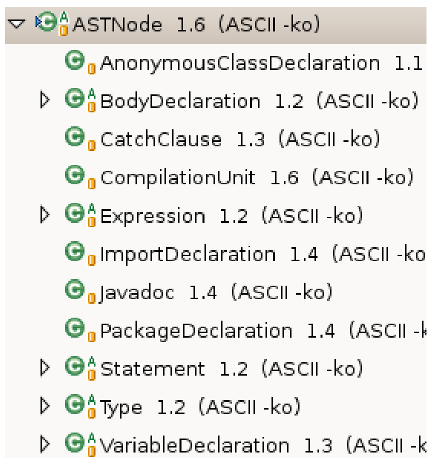

Figure 3.1.a: Version 235

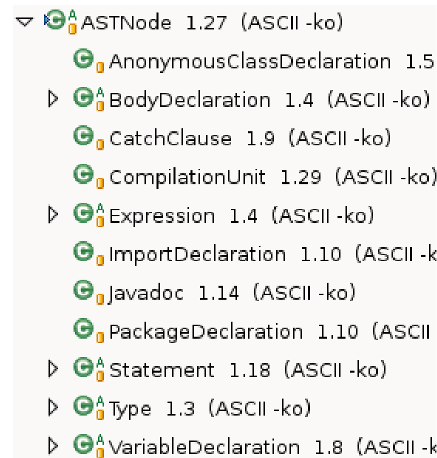

Figure 3.1.b: Version 300

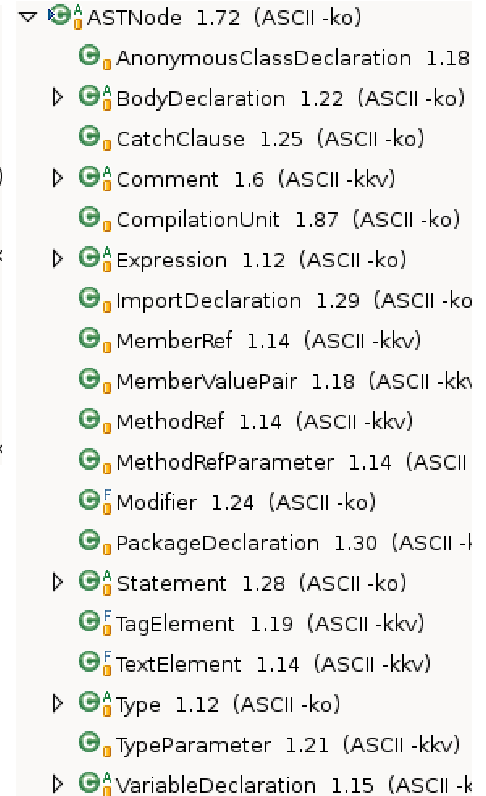

Figure 3.1.c: Version Head

There has been major changes in the ASTNode hierarchy. [Figure 3.1] shows the inheritance hierarchy for ASTNode for the three versions. Versions 235 and 300 have same inheritance structure whereas the recent version has added nodes like *Comment*, *MemberRef, MethodRef, MethodRefParameter, Modifier, TagElement, TextElement* and *TypeParameter* as direct subclass of *ASTNode*.

Comment is an abstract base class for all AST nodes that represent comments. There are exactly three kinds of comment that is represented by three concrete AST Nodes: line comments (*LineComment*), block comments (*BlockComment*), and doc comments (*Javadoc*).

MemberRef is an AST node for a member reference within a doc comment (Javadoc). The principal uses of these are in "@see" and "@link" tag elements, for references to field members (and occasionally to method and constructor members).

MethodRef is an AST node for a method or constructor reference within a doc comment (*Javadoc*). The principal uses of these are in "@see" and "@link" tag elements, for references to method and constructor members.

MethodRefParameter is an AST node for a parameter within a method reference (*MethodRef*). These nodes only occur within doc comments (*Javadoc*).

Modifier is an AST node that represents access modifier like private, public, protected, etc.

TagElement and *TextElement* are respectively the *Javadoc* tags and text in comments.

TypeParameter node represents parameterized type that has been added in JLS3 API.

3.3 BodyDeclaration

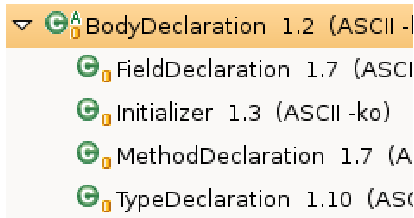

Figure 3.2.a: Version 235

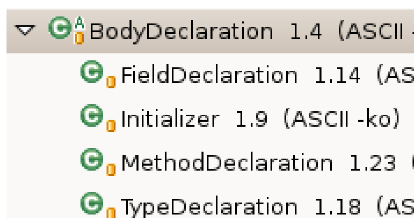

Figure 3.2.b: Version 300

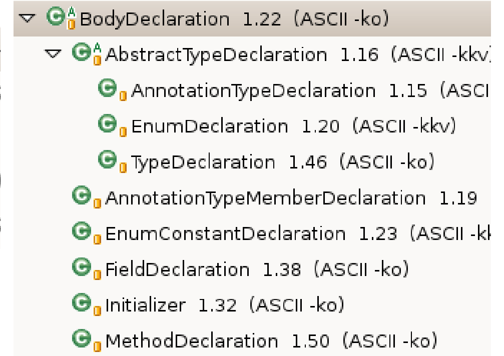

Figure 3.2.c: Version Head

Body declaration remained same in versions 235 and 300 but has gone through major revision in latest version as shown in [Figure 3.2]. *AbstractTypeDeclaration* has been added as abstract node type that is furthered subclassed into *AnnotationTypeDeclaration*, *EnumDeclaration* and *TypeDeclaration*. *TypeDeclaration* has been removed from direct inheritance hierarchy of *BodyDeclaration* and has been moved *AbstractTypeDeclaration* as its subclass. Other nodes added in the hierarchy are *AnotationTypeMemberDeclaration* and *EnumConstantDeclaration*.

3.4 Expression

Figure 3.3.a: Version 235

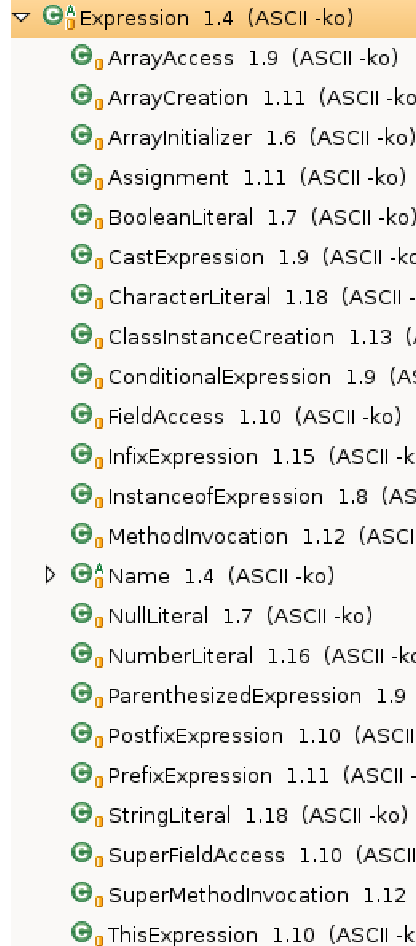

Figure 3.3.b: Version 300

Figure 3.3.c: Version Head

The difference between versions 235 and 300 is that there is addition of *InstanceofExpression* nodes in the subclass hierarchy of Expression nodes in version 300 as shown in [Figure 3.3].

The difference between 300 and latest version are addition of Annotation node and its subclass tree containing *MarkerAnnotation*, *NormalAnnotation* and *SingleMemberAnnotation*.

3.5 Statement

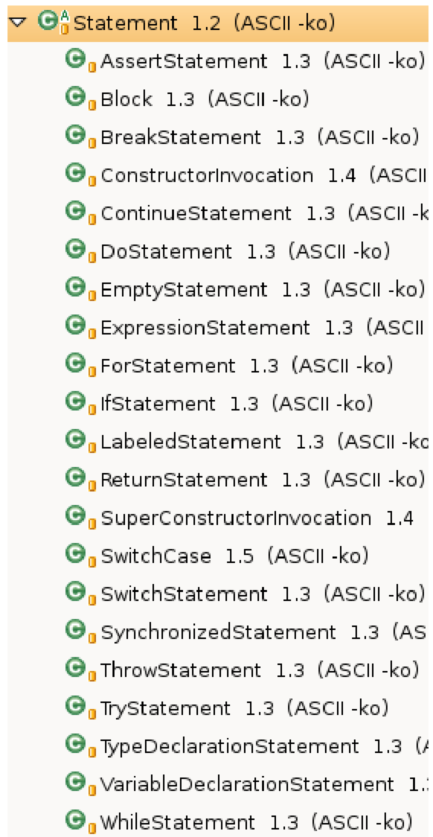

Figure 3.4.a: Version 235

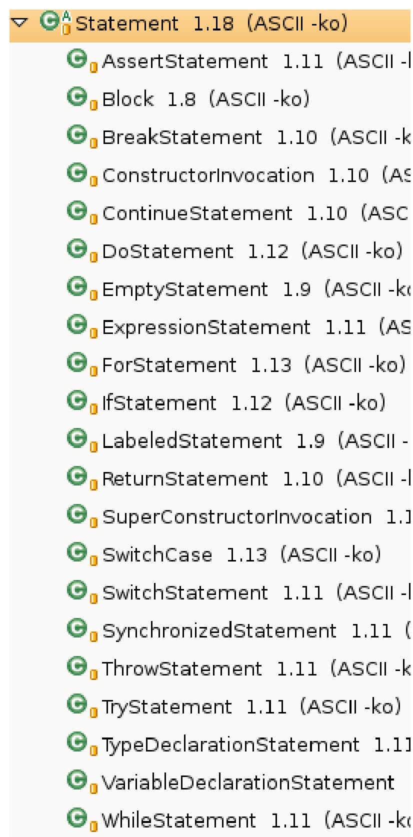

Figure 3.4.b: Version 300

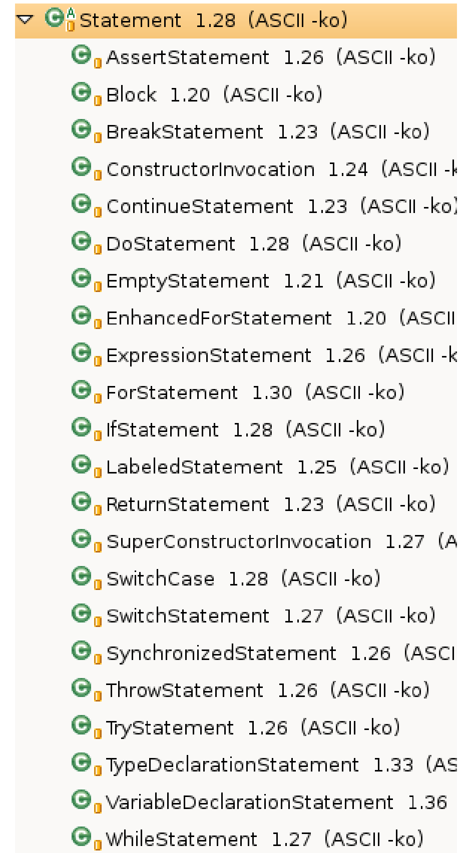

Figure 3.4.c: Version Head

There is no difference in the subclass hierarchy of versions 235 and 300. However, there is an addition on *EnhancedForStatement* in latest version that is not present in the former two versions as shown in [Figure 3.4].

EnhancedForStatement AST node type is added in JLS3 API. It has following structure:

```
EnhancedForStatement:
    for ( FormalParameter : Expression )
                          Statement
```

3.6 Type

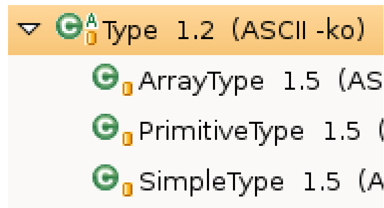

Figure 3.5.a: Version 235

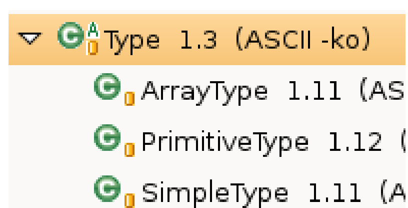

Figure 3.5.b: Version 300

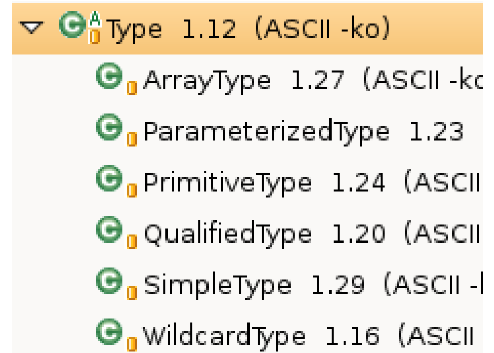

Figure 3.5.c: Version Head

Versions 235 and 300 have same subclass hierarchy for Type class as shown in [Figure 3.5]. However, latest version has added *ParameterizedType*, *QualifiedType* and *WildCardType* in the subclass hierarchy of Type node.

ParameterizedType is added in JLS3 API. These nodes are used for type references (as opposed to declarations of parameterized types.) Its structure is as follows:

```
ParameterizedType:
   Type < Type { , Type }
```

QualifiedType is added in JLS3 API. It has following structure:

```
QualifiedType:
   Type . SimpleName
```

WildCardType is added is JLS3 API. It has following structre:

```
WildcardType:
   ? [ ( extends | super) Type ]
```

4. Exception Handling

Exception handling in JDT framework has not changed much. There has been very little revisions in exception handling which includes addition of new error state constants and change in Javadoc. Here are the few exception classes used in JDT:

4.1 ClassFormatException

Exception thrown by a class file reader when encountering a error in decoding information contained in a .class file. Exception hierarchy is shown in [Figure 4.1].

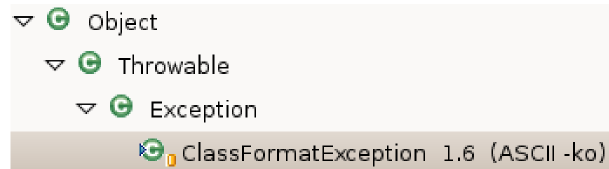

Figure 4.1: ClassFormatException Hirarchy

4.2 CoreException

It is a checked exception that represents a failure. Core exceptions contain a status object describing the cause of the exception. Exception hierarchy is shown in [Figure 4.2].

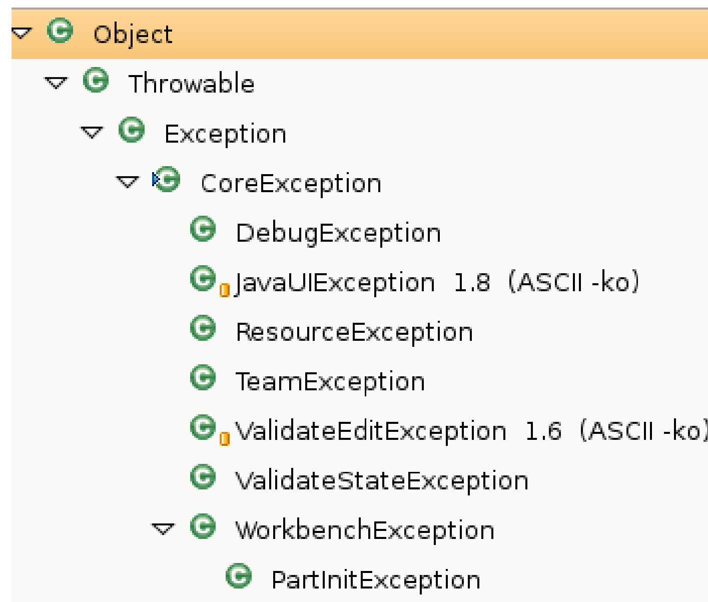

Figure 4.2: CoreException Hierarchy

4.3 DOMException

Unchecked exception thrown when an illegal manipulation of the JDOM is performed, or when an attempt is made to access/set an attribute of a JDOM node that source indexes cannot be determined for (in case the source was syntactically incorrect). It is now deprecated as the JDOM was made obsolete by the addition in of the more powerful, fine-grained DOM/AST API found in the org.eclipse.jdt.core.dom package after 2.0 version of Eclipse. Exception hierarchy is shown in [Figure 4.3].

Figure 4.3: DOMException Hierarchy

4.4 JavaModelException

JavaModelException is a checked exception representing a failure in the Java model. Java model exceptions contain a Java-specific status object describing the cause of the exception.

This class is not intended to be subclassed by clients. Instances of this class are automatically created by the Java model when problems arise, so there is generally no need for clients to create instances. Exception hierarchy is shown in [Figure 4.4].

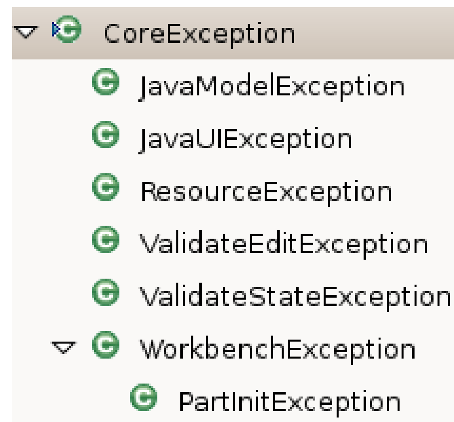

Figure 4.4: JavaModelException

4.5 InvalidInputException

Exception thrown by a scanner when encountering lexical errors. This class is not intended to be instantiated or subclassed by clients.

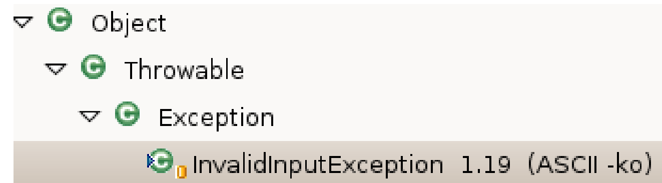

Figure 4.5: InvalidInputException Hierarchy

Apart from these, java.lang's *IllegalArgumentException* and *RuntimeException* are most popular exception that has been used throughout JDT library.

5. Statistics

[Table 5.1] summarizes the list of packages and their classes that has been studied for this project.

Package Name	ClassName	LOC
org.eclipse.jdt.core.dom	Annotation	182
org.eclipse.jdt.core.dom	AnonymousClassDeclaration	190
org.eclipse.jdt.core.dom	ArrayType	240
org.eclipse.jdt.core.dom	Assignment	441
org.eclipse.jdt.core.dom	AST	2868
org.eclipse.jdt.core.dom	ASTNode	2741
org.eclipse.jdt.core.dom	ASTParser	1147
org.eclipse.jdt.core.dom	ASTVisitor	2574
org.eclipse.jdt.core.dom	CompilationUnit	1058
org.eclipse.jdt.core.dom	Comment	130
org.eclipse.jdt.core.dom	Expression	140
org.eclipse.jdt.core.dom	ImportDecleration	377
org.eclipse.jdt.core.dom	MemberRef	271
org.eclipse.jdt.core.dom	MethodRef	316
org.eclipse.jdt.core.dom	Modifier	707
org.eclipse.jdt.core.dom	Type	172
org.eclipse.jdt.core.dom.rewrite	ASTRewrite	659
org.eclipse.jdt.core.compiler	InvalidInputException	39
org.eclipse.jdt.core.compiler	IProblem	1273
org.eclipse.jdt.core.compiler	IScanner	151
org.eclipse.jdt.core	ICompilationUnit	734
org.eclipse.jdt.core	IJavaElement	366
org.eclipse.jdt.core	IJavaModel	259
org.eclipse.jdt.core	JavaModelException	174
org.eclipse.jdt.core	JavaCore	4644
org.eclipse.jdt.core.util	ClassFormatException	55
org.eclipse.core.runtime	CoreException	99
org.eclipse.jdt.core.jdom	DOMException	40

Table 5.1: List of packages and classes studied with their lines of code.

6. Recommendation

Understanding a big framework is not an easy task. First and foremost, these frameworks document tend to become incomplete and the programmers will have no choice than to look at the source code for deeper understanding of these framework. Similarly, to understand Abstract Syntax Tree/JDOM library, the first approach would be to look at the sample examples given in the help files. Once a programmer gets familiar with the underlying code then the first class to explore would be AST and then ASTNode. These two classes are the key classes for whole AST library as they have factory methods for producing children of other concrete types. After one can understand these two classes then classes like ASTVisitor, ASTParser can be explored for further details on other functionality provided in the library. Slowly and steadily, the library implementation becomes clearer at each study.

7. Conclusion

AST library is a part of bigger framework called Java Development Tooling Framework. It has number of classes and packages delivering specific functionality to the programmer. The document of the framework only provides the starting information on the library and the library has capacity to do beyond what is illustrated in these documentation. To understand these functionality, one has to study the provided source code of the framework. The report presents on few of the starting classes for the exploration with code snippets and pictures wherever possible. The evolution trend discussed in the Section 3 clearly presents how a framework tends to evolve over time. The causes may be the update of Java Language Specification or the bugs detected in the library usage, this frameworks has gone through lots of major revisions. Section 5, clearly summarizes how a framework component are linked with one another. To understand AST, one has to also understand other associated libraries.

In this way, the report summarizes an effort of understanding Abstract Syntax Tree provided under Java Development Tooling framework in Eclipse.

References

[1] http://help.eclipse.org/help32

[2] http://en.wikipedia.org/wiki/Factory_method

[3] http://en.wikipedia.org/wiki/Visitor_pattern