

THE ARCHITECTURE OF THE ECLIPSE JDT & Existing Plugin TESTJDT3

Table of Contents

<u>1 The architecture of the Eclipse JDT.....</u>	<u>2</u>
<u>1.1 JDT Architecture Introduction.....</u>	<u>2</u>
<u>1.2 JDT Quality and Attributes.....</u>	<u>2</u>
<u>1.3 JDT Core.....</u>	<u>3</u>
<u>1.3.1 The Plug-in Manifest.....</u>	<u>3</u>
<u>1.3.2 JDT Core Extensions.....</u>	<u>3</u>
<u>1.3.3 Package: org.eclipse.jdt.core.....</u>	<u>5</u>
<u>1.3.4 Package: org.eclipse.jdt.core.dom.....</u>	<u>10</u>
<u>2 Using JDT in Econet.....</u>	<u>13</u>
<u>3 Existing Plugin TESTJDT3</u>	<u>14</u>
<u>3.1 Structure</u>	<u>14</u>
<u>3.2 Java Classes.....</u>	<u>15</u>
<u>3.3 Eclipse integration.....</u>	<u>15</u>
<u>4 References.....</u>	<u>16</u>

1 The architecture of the Eclipse JDT

1.1 JDT Architecture Introduction

The Java Development Tools (JDT) plug-in of the Eclipse platform provides a rich set of functionality to enable Java developers to use Eclipse as a Java IDE. The JDT is actually a set of plug-ins that contributes a Java compiler, debugger, as well many Java-specific user interface elements, to the Eclipse platform.

The JDT is structured into three major components:

- JDT CORE : the headless infrastructure for compiling and manipulating Java code
- JDT UI : the user interface extensions that provide the IDE.
- JDT Debug : program launching and debug support specific to the Java programming language.

In this paper, we present a broad overview of the architecture of the JDT. We have selected the one important plug-in of the JDT and expounded on their architecture. This plug-in are the Core plug-in. The Core plug-in provides functionality to compile, build, search, and format Java code. For this plug-in, we discuss how they extend the functionality of Eclipse and the patterns they use in doing so.

1.2 JDT Quality and Attributes

The JDT was developed to make Eclipse the top IDE for developing Java applications.

This was achieved to a great extent due to a set of JDT quality attributes that include modularity, extensibility, usability, and portability.

The extensibility quality attribute is inherited from the plug-in architecture of Eclipse platform. JDT plug-ins make contributions to the extension-points of the Eclipse platform, and also define new extension-points. Extension-points are discussed in the next sections of this document. As a default implementation, JDT plug-ins contribute several extensions to their own extension-points, thus making the Java IDE highly configurable.

The JDT Core plug-in provides a JCK-compliant Java compiler with incremental recompilation, which allows high performance and scales up to large projects.

The JDT plug-ins also incorporate many best practices in software engineering, such as integrated support for unit test (with JUnit plug-in) and refactoring, which makes Java programming with Eclipse truly productive.

1.3 JDT Core

JDT Core is the plug-in that defines the core Java elements and API. In this section, we'll look at its structure, extensions and some elements of its package.

1.3.1 The Plug-in Manifest

Each Eclipse plug-in contains a plug-in manifest file that describes the contents of the plug-in. When Eclipse starts up, it scans the plug-in directory and collects information on each plug-in by reading each one's manifest file. The manifest file is always called "plugin.xml". This allows the Eclipse platform to load plug-ins only when needed, as opposed to loading them all when Eclipse starts up.

The plug-in manifest file contains a declaration for the plug-in, the name of the library that has the binary class files for the plug-in, a list of other plug-ins that this plug-in depends on, and a list of extensions and extension points.

The plug-in manifest file for JDT Core indicates that the id for the plug-in is org.eclipse.jdt.core. This id can be used to reference the plug-in when creating other plug-ins that extend JDT Core.

JDT Core is dependent on the resources, runtime, and text plug-ins. JDT Core will not run if these three plug-ins are not installed. In addition, it has functionality that depends on the ant and team plug-ins, but this functionality will be ignored if those two plug-ins are not installed.

JDT Core defines three extension points. An extension point allows other plug-ins to extend certain features of a plug-in. JDT Core allows other plug-ins to extend the following functionality: the formatting of source code and the initialization of classpath variables and classpath containers.

1.3.2 JDT Core Extensions

JDT Core also defines eleven extensions from other plug-ins that provide much of the functionality of JDT. Some of these extensions will now be explained.

1.3.2.1 Java Builder Extension

A builder is associated with a project and is used to incrementally build a project whenever a resource is changed. The changes to the project can also be batched so that the receiver gets one notification containing all the changes that occurred. Builders work by processing the list of changes they receive, updating the internal build state, and generating resources. They are notified whenever a resource is changed or the user selects a build action, and they build incrementally, so only those resources that are dependent on the changes that occurred are rebuilt.

The extension declaration for the JDT builder in the plugin.xml file indicates that the class that implements the builder is org.eclipse.jdt.internal.core.builder.JavaBuilder. This class is a subclass of IncrementalProjectBuilder and must implement the build() method. The build() method takes as arguments an integer that represents the type of build to run (this will usually be AUTO_BUILD or INCREMENTAL_BUILD), a Map object of

builder-specific arguments, and an IProgressMonitor object that can be used to show the progress to the user and to let the user cancel the action if so desired.

The build() method calls the getLastDelta() method with an IProject argument in order to get any changes that have occurred to the specified project since the last time the build() method was called. The builder class then does the necessary processing and returns a list of IProject objects, which specify the projects that the builder would like deltas on the next time it is run.

The IncrementalImageBuilder, which is also in the org.eclipse.jdt.internal.core.builder package, is used to do the actual building when an incremental build is done. This class sets certain resources as derived resources. Derived resources are those resources that are built by the builder. It creates a new output file (of type IResource) for the source file being built, builds the output file, and then calls setDerived() with a value of true on the output file. It is important for other plug-ins to know which resource files are derived so that they do not put those files under version control.

When creating a new workbench project, a file named “.project” is also created for the project. This is an XML file that contains metadata about the project so that one can export the project. The following is part of the XML that is created for Java projects and lets Eclipse know which builder to invoke on the project:

```
<buildSpec>
  <buildCommand>
    <name>org.eclipse.jdt.core.javabuilder</name>
    <arguments>
    </arguments>
  </buildCommand>
</buildSpec>
```

The builder given is the id of the JavaBuilder class that JDT defines.

1.3.2.2 Java Marker Extensions

A marker is a mechanism for adding notes and metadata to a resource. Each marker has an associated string that contains the type of marker it is and an associated identifier that uniquely identifies it within a resource. There are five standard marker types that are defined by the Resources plug-in. These are: marker, taskmarker, problemmarker, bookmark, and textmarker. When a new marker is defined as an extension it is defined as one or more of these types. Markers can also be persisted by Eclipse if the user specifies this option in the plugin.xml file. Finally, markers may have attributes attached to them. The names of these attributes are also listed in the plugin.xml file.

The first marker extension in JDT Core is the Java Problem marker. It has id “problem” and is of type problemmarker and textmarker. This means that the marker will be shown in the Problems view and that it is a text-based marker. The marker is also persistent. The Java Buildpath Problem marker has id “buildpath_problem” and is of type problemmarker and textmarker. Like the Java Problem marker, it too is persistent. The Java Transient Problem marker is a non-persistent marker of type textmarker with id

“transient_problem”. Finally, the Java Task Marker is a persistent marker with id “task” and is of type taskmarker. This means that the marker will appear in the Tasks views. The IJavaModelMarker interface defines constants for the different attributes and ids of the JDT Core markers. Attributes for the markers can be set using the setAttribute() and getAttribute() methods. These attributes are key/value pairs. The value can be a String, an integer, or a boolean. There are also several other attributes that are not in the plugin.xml file, but are nonetheless used extensively. They are defined as constants in the IMarker interface. These markers are used to indicate where in the text the marker occurs. The attributes used for this are IMarker.LINENUMBER, IMarker.CHAR_START, and IMarker.CHAR_END. If a user double clicks on a problem marker in the Problems view, the editor will automatically highlight the text indicated by the marker attributes.

1.3.2.3 Java Nature Extension

A nature groups a project with functionality specific to that project, such as a builder. Also, a nature can choose to hide certain actions from projects, if those actions do not apply to the specific project. A nature extension defines a new nature with an id that is used to refer to it. It also specifies a class that implements the org.eclipse.core.resources.IProjectNature interface.

JDT Core defines a Java nature and all Java projects that are created have this nature associated with them. This nature is also added to the project’s .project file. The following is part of the XML that is created for Java projects and lets Eclipse know which nature the project has:

```
<natures>
  <nature>org.eclipse.jdt.core.javanature</nature>
</natures>
```

The class that implements the IProjectNature interface for JDT Core is org.eclipse.jdt.internal.core.JavaProject. This class implements the methods configure(), which adds a Java builder to the project, and deconfigure(), which removes a Java builder from a project.

1.3.3 Package: org.eclipse.jdt.core

The jdt.core package contains the model used to represent the various objects involved with Java development. It also contains the plug-in class for JDT core.

1.3.3.1 The Java Core Plug-in Class

Each Eclipse plug-in contains one class that represents the plug-in to the Eclipse plug-in loader. This plug-in class is responsible for accessing and storing state information for the plug-in, as well as accessing any resources associated with the plug-in. The plug-in class contains methods that are run when the plug-in is loaded, as well as methods that are automatically called when the plug-in is about to be terminated. These methods are called start() and stop(), respectively. All plug-in classes must extend either Plugin or AbstractUIPlugin. The former is extended by non-UI plug-ins, while the latter is extended by UI plug-ins.

The plug-in class for JDT Core is `JavaCore`. It is located in the `org.eclipse.jdt.core` package and extends the `Plugin` class, since it is not a UI-based plug-in. The `JavaCore` class is usually accessed through its static methods, which allow one to create different nodes that are part of the Java model (such as new projects, classes, folders, and files). The JDT Core plug-in does not maintain any state or preference information, and so does not implement the `start()` or `stop()` methods.

1.3.3.2 Detailed Description of the Java Model

The Java model is used to represent the elements of the Java language as a tree. When programmers need to access different Java elements (such as a source file, or a method) they do so by using the Java model to get the correct node in the tree.

The following description of the `jdt.core` package will explain the Java model, starting from the topmost elements in the model to elements comprising single source files and parts of source files. In addition code examples will be given to show how to get handles to these objects, as well as how to manipulate them.

- **IJavaElement**

The Java model is represented as a tree. Each node in the tree is of type `IJavaElement`. Many operations on Java model elements take in an `IJavaElement` and return an `IJavaElement`. Some useful operations for an `IJavaElement` are `getParent()` which return the parent of the element in the model as an `IJavaElement`. To get the name of the element use `getElementName()`. To get the type of the element use `getElementType()`. This method returns an `int` that represents the type of the element. The `int` is one of the static constants declared in the `IJavaElement` class. Two other useful functions are `getAncestor()` and `getPath()`. The `getAncestor()` method returns the closest ancestor of the element matching the specified type, which is given as a parameter to the method. The `getPath()` method returns an `IPath` object representing the path to the parent of the element.

- **IOpenable and ISourceReference**

While all the elements in the Java model implement the `IJavaElement` interface, they can all be divided based on whether they implement the `IOpenable` interface or the `ISourceReference` interface.

The `IOpenable` interface is implemented by those elements in the Java model that must be opened before they can be manipulated. These elements are generally files on the system. The elements in the Java model that implement this interface are usually high up in the hierarchy, such as `IPackageFragmentRoot` and `IProject`. Three operations that are very important for elements that implement this interface are `open()`, `close()`, and `save()`. If an element in the Java model that implements this interface is obtained and accessed by a client, the Java model will automatically open the element as needed. To open an element, all of its parent elements must also be opened. However, this is also done automatically by the Java model. The Java model will not open its children though, unless they are accessed by a client.

When an `IOpenable` element is opened, a buffer is created in memory that contains the contents of the corresponding file. A client can access this buffer by calling the

`getBuffer()` method on the `IOpenable` element, which returns an `IBuffer` class. Clients may manipulate the `IBuffer` directly to change the contents of the underlying file, but it is safer to use the corresponding Java model element instead to do so. The Java model contains a cache of opened buffers. If a buffer has been edited and not saved, it will not be removed from the cache. Otherwise, buffers will be moved out in LRU order to make room for new opened buffers. Therefore, clients should save buffers frequently because having too many buffers open at the same time will use up memory quickly.

In contrast to `IOpenable`, the `ISourceReference` interface is implemented by elements that correspond to source code. They are not represented by files, but are contained in them. These elements are further down in the tree that represents the Java model. The elements in the Java model that implement this interface are `IClassFile`, `ICompilationUnit`, `IPackageDeclaration`, `IImportDeclaration`, `IImportContainer`, `IType`, `IField`, `IMethod`, and `Initializer`.

- **IJavaModel**

The `IJavaModel` interface is at the root of the Java model. It acts as a container for all the Java projects in the workspace. The following code is used to get a handle to this object:

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
IJavaModel model = JavaCore.create(workspace.getRoot());
```

Usually, with an `IJavaModel` object, you will want to get a specific Java project to manipulate. The `getJavaProjects()` method returns an array of `IJavaProject` objects corresponding to all the Java projects in the workspace. Alternatively, if you know which project you want, you can get the project directly by calling the `getJavaProject()` method with the name of the project as a parameter.

- **IJavaProject**

The `IJavaProject` interface represents a Java project. The following code is used to get an `IJavaProject` object that corresponds to a Java project in the workspace named “TESTJDT3”.

```
IJavaProject project = model.getJavaProject("TESTJDT3");
```

The children of an `IJavaProject` are `IPackageFragment` objects (which represents a package in the project), `IPackageFragmentRoot` objects (represents jar files in the project), and `IClassPathEntry` objects (representing the values of the classpath environment variable for the project).

There are several actions one might want to perform on an `IJavaProject` object. To find a Java element in the project, one first constructs an `IPath` object that represents the path of the Java element. Then one calls the `findElement()` method with the `IPath` object. The method returns an `IJavaElement` object, or null if no object is found with the specified path. `IPath` objects are defined in the `org.eclipse.core.runtime` package, and describing them is beyond the scope of this document. One can also find types in the project by calling the `findType()` method with the fully qualified name of the type as the parameter. The return value will be an object of type `IType`, or null if none is found.

Another useful method in this class is `newTypeHierarchy()`. This method takes in an `IRegion` object and an `IProgressMonitor` object as parameters. It returns an `ITypeHierarchy`. The `IProgressMonitor` object can be used to kill the operation if it is taking too long. If this feature is not needed, one can just enter `null` as the parameter. An `IRegion` object contains a set of `IJavaElement` objects, and the children of all the elements in the set.

- **IPackageFragment**

The `IPackageFragment` interface represents all or part of a package in a project. A package can contain either class files or source files. Class files are represented by `IClassFile` objects, and source files are represented by `ICompilationUnit` objects. The following code can be used to get an `IPackageFragment` object from an `IJavaProject` object:

```
IPath path = new Path("/TESTJDT3/packageOne");
IPackageFragment pack = project.findPackageFragment(path);
```

In the above code, the path can either be an absolute path, or can be relative to the workspace of the user.

We mentioned above that we can get an `ITypeHierarchy` object from a project. Here is some sample code that gets that object and represents the type hierarchy for the package object created above:

```
IRegion region = JavaCore.newRegion();
region.add(pack);
ITypeHierarchy typeHierarchy = project.newTypeHierarchy(region, null);
```

Normally, with an `IPackageFragment` object, we will want to get the class files and source files that are in the given package. `getCompilationUnits()` returns an array of the source files in the package represented as `ICompilationUnit` objects. `getClassFiles()` does the same thing for the class files in the package. If we know which source file we want, we can call `getCompilationUnit()` and pass it the name of the source file we want as a `String`. The corresponding method for class files is `getClassFile()`. Finally, one can create a new source file in the package by calling the `createCompilationUnit()`. This method takes in the name of the source file to create, the contents of the source file (as a `String` object), a boolean value that will force an existing file with the same name to be overwritten if it is true, and an `IProgressMonitor` that can be used to kill the operation in progress.

- **ICompilationUnit**

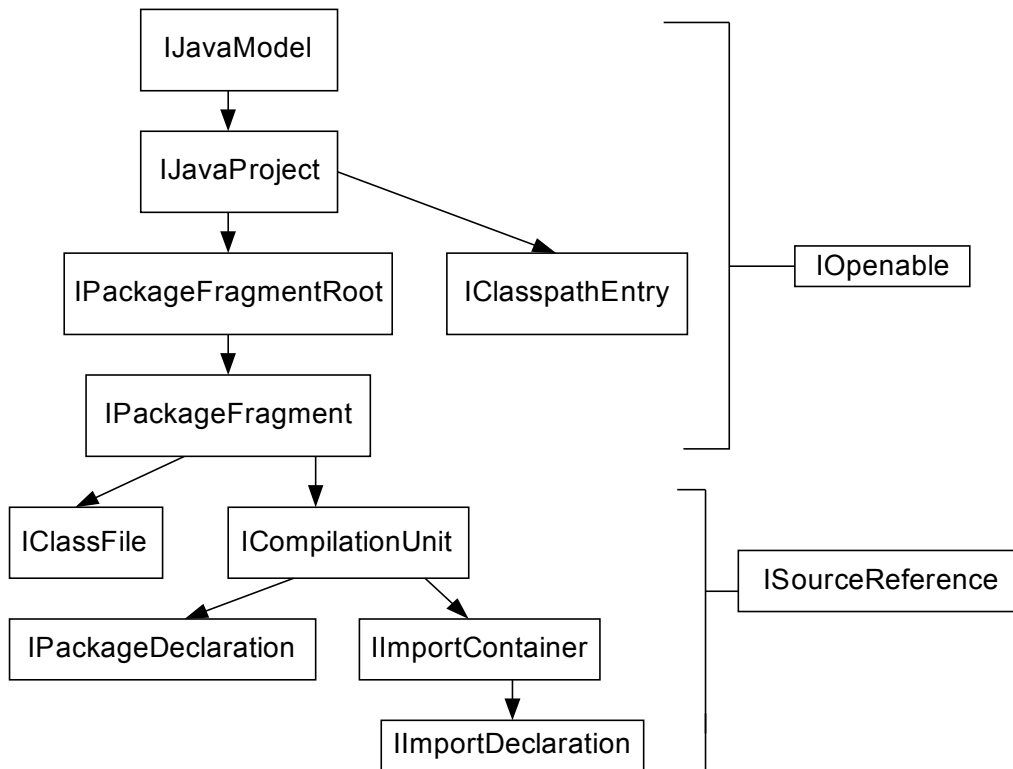
An `ICompilationUnit` object represents a Java source file. The following code gets an `ICompilationUnit` object representing the source file named "MyClass.java".

```
ICompilationUnit source = pack.getCompilationUnit("MyClass.java");
```

With this object, there are many useful methods we can call. We can get all the types in the class with a call to `getTypes()`, which returns an array of all the types. If we know

the type we want, we can call `getType()` with a `String` parameter that is the name of the type. The topmost type matching the `String` will be returned. We can get the package declaration with a call to `getPackageDeclarations()`, which will return an array of `IPackageDeclaration` objects. This array will usually be of size one, since most Java classes are part of one package. There is also a method to get the import declarations, as well as methods to create new import declarations for the class.

The following diagram shows the Java model as a tree with the topmost elements of the tree. An arrow from one element to another means that element being pointed to is a child element of the other element. For instance, `IJavaProject` is a child element of `IJavaModel` and can be accessed from an `IJavaModel` object.



There are many more elements in the Java model. Some of the most useful are `IType`, `IField`, `IMethod`, and `IMember`. These elements all have similar operations that one can perform on them, and one is referred to the Javadoc for more information.

1.3.4 Package: `org.eclipse.jdt.core.dom`

The `jdt.core.dom` package comprises an Abstract Syntax Tree (AST) for the Java language, as well as objects that perform operations on the AST.

1.3.4.1 Detailed Description

- **AST class:**

The AST class is the owner of the AST. Any new AST nodes created by using an object of this class will be owned by that object.

The AST class also acts as a factory for producing `ASTNode` classes. A node of any type can be created using a method of the form `newXXX` where `XXX` is the name of the syntax element to be created. Each node that is created in this way does not have any type name or value specified. The node also has no parent.

Finally, AST provides a utility method `resolveWellKnownType()`. This method takes in a String which names a well known type. It returns an `ITypeBinding`, which is an interface that represents a well known type. It will be discussed further below.

- **ASTNode class:**

The `ASTNode` class is the superclass for the many AST node types. An `ASTNode` represents a syntactic element in the Java language. Each node has links to each of its children, as well as to its parent node. Therefore, the AST can be traversed either from the top down, or from the bottom up. In addition, each `ASTNode` object contains the range in the source file where the syntactic element can be found. The `getStartPosition()` method returns an index into the source file where the element starts, and the `length()` method returns the number of characters that comprise the element.

There are three static variables in the class that are used as flags. One of particular interest is the `ASTNode.MALFORMED` flag, which indicates that the syntactic element contains a syntax error. The other static variables are used to represent different nodes. They are used in the `createInstance()` method of the AST class to specify the AST node to create.

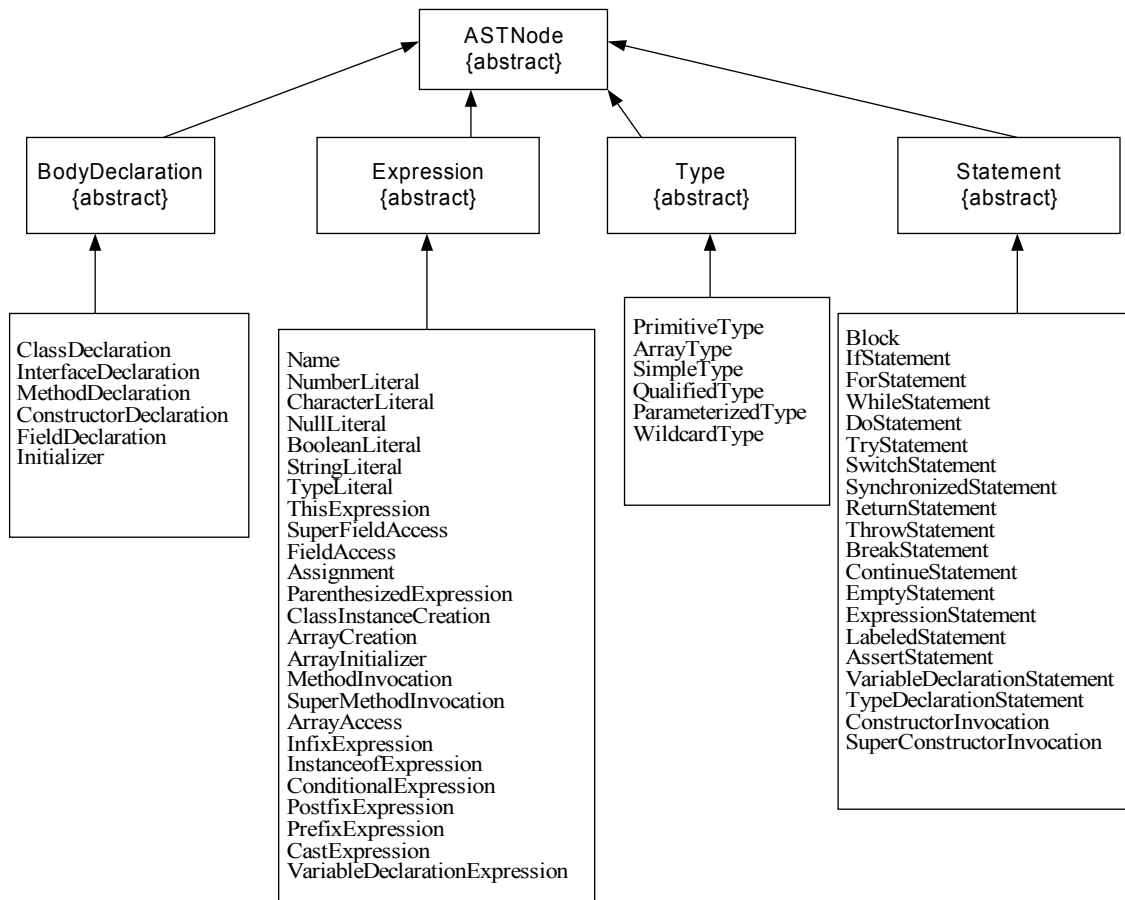
- **ASTParser class:**

The `ASTParser` class is responsible for converting source code into an AST. There are no constructors to use for this class. Instead a static factory method called `newParser()` is used to create a new `ASTParser`. The argument to this method specifies the level of the Java Language Specification to use. The `setSource()` method specifies the source to compile. There are three overloaded versions of this method. One of them takes in an array of characters, which will contain the source to parse. The `createAST()` method will create the AST from the source that was given to the object. It returns an object of type `ASTNode`, which will represent the root of the produced AST. It also takes in an `IProgressMonitor` object that can be used to cancel the operation in progress, if so desired. If this functionality is not required, null can be passed in as the argument. A useful method that this class provides is `setProject()`. This method takes in an

IJavaProject object that is used to specify a Java project on the workbench. This Java project will be used to resolve types in the source string that otherwise could not be resolved by the compiler.

One can also specify compiler options to use to parse the source string. The method to do this is called `setCompilerOptions()`. This method takes in a Map object. All the keys and values of the Map are expected to be String objects, where the key is a compiler option, and the value is the desired value for that option. An argument of type null sets the options back to their defaults.

- **The AST hierarchy:**



- **ASTVisitor class:**

To perform operations on an AST, we use the ASTVisitor class. ASTVisitor is an abstract class. It provides two operations to be performed on every node of an AST. The visit() method returns true if the node has children that will be visited after the current node is visited. The endVisit() method is similar to visit() except that the children of the node will be visited before the node itself is visited. The default implementation provided in the ASTVisitor class does nothing in the endVisit() methods, and returns false for the visit() methods. The developer who wishes to implement a visitor for the AST must subclass ASTVisitor and then define the operations for each node to be visited in the appropriate method.

In addition to all the type-specific visit operations, there are two operations that perform work on an ASTNode in general, and not on specific types within the AST hierarchy. The preVisit() method is used to visit an ASTNode before the type-specific visit operation is called on that node. The postVisit() method visits the ASTNode after the type-specific visit operation on that node.

1.3.4.2 Patterns Used

AST uses the following patterns to perform operations mentioned above.

- **Factory Pattern:**

The AST class utilizes the Factory design pattern to create new ASTNode objects. The method createInstance() takes an integer value that represents a certain node. The method will return an instance of that node. Each class that it returns inherits from the ASTNode class.

The ASTParser class also uses the Factory pattern. To create a new parser, one calls the newParser() method instead of using a constructor. This method will create the appropriate parser based on the level of the JLS specified.

- **Visitor Pattern:**

The ASTVisitor class uses the Visitor pattern. The Visitor pattern is generally used to perform operations on a structure with lots of small nodes in them. Rather than putting the functionality for the nodes in the nodes themselves, they are moved to a visitor class, which implements the functionality for all of the nodes. In general, we would use a visitor class if the operations we are performing on the structure involve most or all of the nodes of that structure. When subclassing ASTVisitor, we will usually define operations on all of the nodes, since the nodes represent syntactic elements in the Java language, and we do not want to leave out processing any of these elements. However, it is conceivable that one would construct an ASTVisitor that implements only some of the operations on the nodes. For instance, one could write a visitor that traverses an AST and outputs all the class names in the AST. In this case, we would only be interested in nodes that represent classes, or nodes that can contain classes as descendants. The rest can be ignored and the default operations on them will be performed.

2 Using JDT in Econet

This study of JDT will enable us to understand the Eclipse development environment, to capture the TESTJDT3 plugin structure and develop a new Econet plugin in the sense that we will be able to transform the structure of a Java code without modifying its logic. For illustration, we could:

- Manipulate Java classes and/or interfaces resources to extract information regarding their structure: attributes, methods.
- Create an AST to analyze compilation units.
- Interrogate AST data on implementing extraction rules.
- Modify code in order to position annotations.

3 Existing Plugin TESTJDT3

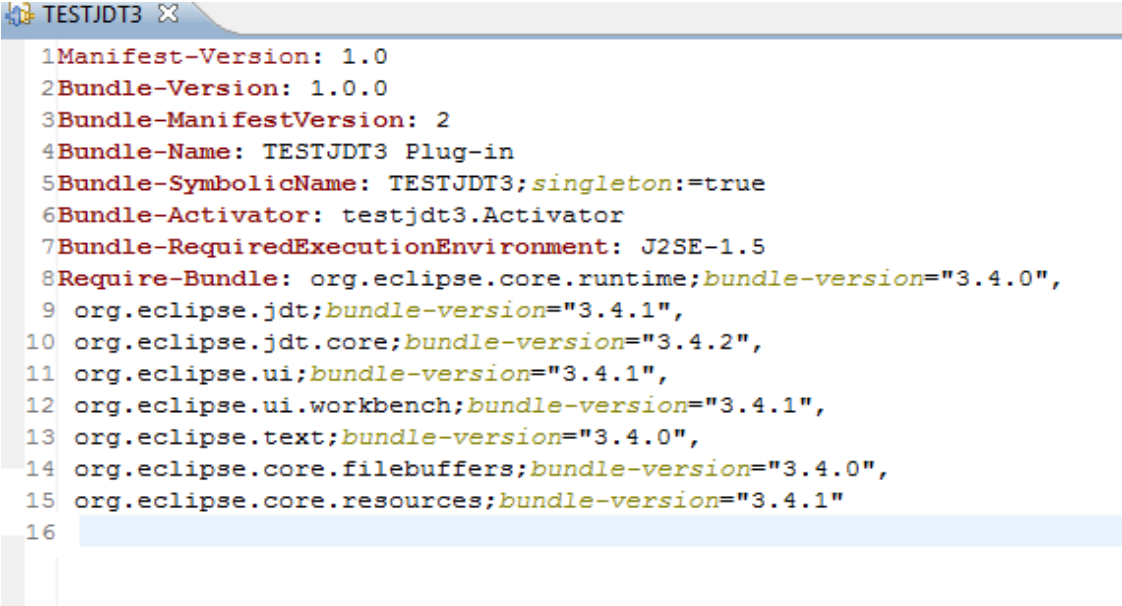
This section explains Econet project's existing plug-in. TESTJDT3 is the plug-in used to extract the component structure of a Java code.

3.1 Structure

The plug-in is structured as a classic JAR file containing in addition to its Java classes two files(META-INF/MANIFEST.MF and plug-in.xml)

MANIFEST.MF is the configuration file describing the functioning of the plug-in Jar archive. It is used to by Eclipse kernel, Equinox, to obtain information about the plug-in that particularly serves to manage the life cycle and the relationship with others plug-ins.

The figure below illustrates the content of the MANIFEST.MF file of TESTJDT3



```

1Manifest-Version: 1.0
2Bundle-Version: 1.0.0
3Bundle-ManifestVersion: 2
4Bundle-Name: TESTJDT3 Plug-in
5Bundle-SymbolicName: TESTJDT3;singleton:=true
6Bundle-Activator: testjdt3.Activator
7Bundle-RequiredExecutionEnvironment: J2SE-1.5
8Require-Bundle: org.eclipse.core.runtime;bundle-version="3.4.0",
9 org.eclipse.jdt;bundle-version="3.4.1",
10 org.eclipse.jdt.core;bundle-version="3.4.2",
11 org.eclipse.ui;bundle-version="3.4.1",
12 org.eclipse.ui.workbench;bundle-version="3.4.1",
13 org.eclipse.text;bundle-version="3.4.0",
14 org.eclipse.core.filebuffers;bundle-version="3.4.0",
15 org.eclipse.core.resources;bundle-version="3.4.1"
16

```

The file syntax is described in the OSGi specification, it contains information about the plug-in regarding:

- the MANIFEST version number: 1.0
- Its own version, currently 1.00
- The MANIFEST version : 2
- The name: Plugin TESTJDT
- The symbolic name: TESTJDT3
- The name of its unit Activator used at start-up and shut-down of the plug-in: testjdt3.Activator
- Its execution environment : J2SE-1.5
- And, login of necessary units and their version for utilisation. This section has to contain all the plug-ins required at compilation, as well as all the plug-ins supplying extensions points used by our plug-in.

The file `plug-in.xml` is unique to Eclipse (it is not part of OSGi), it serves to realised Eclipse extension functionalities. Via this , optional, file the plug-in declares extension points and allow other to connect to them.

3.2 Java Classes

The extraction process of the component structure architecture of a Java code uses the following Java classes

- *Activator* : the activator class controls the plug-in life cycle.
- *ASTActionDelegate* : experiment with JDT to parse and extract component boundaries.
- *DisplayText* : general class to simply display some texts.
- *TypesTable* : table for storing informations about the types.
- *MyIType* : auxiliary class for additional services of ItypeBinding
- *GenericASTParser* : A generic ASTParser which could be configure with the visitor type.
- *Fields* : cl'ass for storing structure with full information
- *Communications* : class to store communications.
- *InfoCom* : class to store method reference from calls in the code
- *Decision* : class for decision.
- *Provided* : class to store per type name the set of required services
- *Information* : class for information about the types.
- *Utility* : simple utility class.

3.3 Eclipse integration

Whilst the plug-in is loaded in Eclipse, a new tab appears on the menu bar with TESTJDT3 as a label. This menu contains an item Visit that triggers the extraction process.

To extract Java code, the process consists of a simple click on the Visit tab, followed by providing the name of the project to be parsed. However, this project has to exist in the configuration workspace or at least appears in the project list on the project explorer window.

4 References

1. Eclipse Home Page: <http://www.eclipse.org/>
2. David Boxer, Ashutosh Galande, Thuc Si Mau Ho of University of Illinois at Urbana-Champaign : JDT Architecture, Publication 2004 41p.