

End of studies project

-

Memo

Matthieu AUBRY

Tima CAMARA

Aurélia COUVRAND

Jérémie GUIDOUX

Lydie THIERRY

Mathieu VÉNISSE

Supervisors :

Pascal ANDRÉ

Gilles ARDOUREL

Jean-Claude ROYER

March 27, 2009

Contents

1	Introduction : The Econet project and process B	3
1.1	Context of the project	3
1.2	Preamble	4
1.2.1	Motivations of the project	4
1.2.2	The Econet project	5
1.2.3	Why our project ?	6
1.3	The process B	7
1.3.1	Introduction	7
1.3.2	Architectural view	7
2	Division of labor	9
3	The architecture of the Eclipse JDT	11
3.1	JDT Architecture	11
3.1.1	Package org.eclipse.jdt.core	12
3.1.2	Package org.eclipse.jdt.core.dom	18
3.2	Using JDT in Econet	21
4	Study of the existing project	22
4.1	Existing Plugin TESTJDT3	22
4.1.1	Structure	22
4.1.2	Java Classes	24
4.1.3	Eclipse integration	24
4.2	Existing rules-based system and its properties	25
4.2.1	Introduction	25
4.2.2	Extracting components structure rules	25
4.2.3	Research of interfaces	30
4.2.4	Properties of a rules-based system	30
4.2.5	Conclusion	32
4.3	An experimentation : CoCoME	32
4.3.1	Introduction	32

4.3.2	CoCoME	33
4.3.3	Plug-in	36
5	Conception	38
5.1	Detailed Conception of Econet plugin	38
5.1.1	Rules-based system	38
5.1.2	Econet plugin structure	43
5.2	Annotations	49
5.2.1	How will we get these annotations?	49
5.2.2	How have implemented this and why?	50
6	Bibliography	51
6.1	Rainer Koschke's thesis summary	51
6.1.1	Extraction process	51
6.1.2	Extraction technique	53
6.1.3	Schwanke's Arch Approach	55
6.1.4	Semi-automatic techniques	59
6.1.5	Conclusion	62
6.2	References	64

Chapter 1

Introduction : The Econet project and process B

1.1 Context of the project

[This paragraph is heavily based on the document *econet_nantes.pdf*, which is on the [wiki COLOSS page](#)].

This project is an international one which includes four research teams that have complementary knowledge and background on the econet project domain. We present each team with its background.

- **France – Nantes : COLOSS**
Background : *Kmelia* for both the rich specification model of services and the existing analysis tools for the conformance verification.
- **France – Nantes :OBASCO**
Background : generation of Java code from protocol specifications. Some mechanisms have been identified that can help to build the environment for the component.
- **Czech Republic – Prague : DSRG**
Background : studied the verification of Java byte code of components against high-level specifications
- **Romania – Cluj : LCI**
Background : there are proposals and even approaches for extending OCL in order to support action specifications (the *Xactium tool XMF Mosaic* sees <http://www.xactium.com>).

The general impact of the project is to state some principles on the field of consistency between component code and component specifications. These principles are accompanied with techniques for code abstraction and behavioural properties checking. A specification model should extend behaviours to guards and assertions mechanism with OCL. Experimentation will be led on existing code.

An interesting perspective is to abstract the code of existing component platforms (such as *.NET*, *EJB* or *CORBA*) or Web services implementations to more abstract, i.e. academic models. This perspective can lead a comparison support and a benchmark for reverse engineering. A further perspective focus on the development of a platform that implements the principles : the move from reverse engineering theory into practice in the context of software components. The partners plan to integrate their approaches such that abstraction can deal with various specification models. The main benefits is to reuse the existing platforms for the specification and the verification of software components.

1.2 Preamble

1.2.1 Motivations of the project

Component-based software engineering is now a common approach in many areas of software development. This approach permits to make well-constructed software, it is based on a structural separation. A component must be autonomous and dialog with other components through his interfaces.

Today, there is no solution to check if a software is well-constructed or not. For example, we can use classes diagrams to describe an architecture, scenarios to describe behavioral features and the Java language to implement a solution but it is very difficult to check if the implementation is components oriented or not. It is difficult to check if the components are independent from each others. UML [0] specification or academic solution like ACME [1] are based on an abstract model when EJB [2], CCM [3], OSGi [4] or other industrial solutions are focusing on implementation. It is difficult to make links between these two specifications.

In the next part we will explain a solution proposed by the COLOSS team [5] to resolve this problem by using reverse engineering technologies and the MDA approach.

1.2.2 The Econet project

The solution proposed is called the *Econet project*. First we present the global scheme of this project.

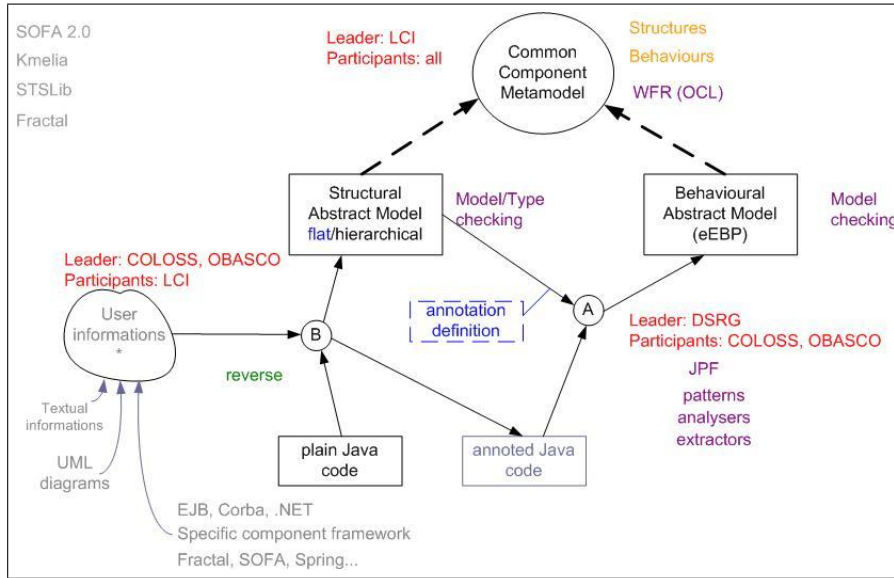


Figure 1.1: Global scheme

This project is divided in three independents parts :

- A component metamodel, described with the EMF [6], standard permits to represent components and dialogs between them without concerns about implementation solutions (EJB, OSGi, ...)
- **Process B** : this is the part that we will study in this project, as we can see on the scheme, plain Java code is in entry of this process and produce a structural abstract model conforms to a component meta-model. User informations can be added to simplify the extraction of components.
- **Process A** : This process analyse the structural abstract model given by the *process B* and extracts a dynamic behaviour specification. We don't explain this part of the *Econet project*.

1.2.3 Why our project ?

An implementation of the *process B* was proposed by the *COLOSS team* but it is just an experimental solution : all the functionalities are not implemented. The result produced by this process is not yet a model conforms to the EMF metamodel but textual informations that represents the structure of the application studied. This implementation is an eclipse plug-in.

We have to enhance and complement this implementation and to make possible the creation of the model that will be used by the *process B* in the future. For this work, the main API is Java Development Tools (JDT). This API is not used a lot and, therefore not documented. We have also to write a documentation which explain and detail the structure of this API and how to use it. This documentation is available in chapter 3 of this report.

If all this work was done we can implements a functionality that will permit to represent component structure with a graphical representation. We suppose that we can use the language *dot* [7] because it is very simple to generate a graph with this language or the java plug-in *prefuse* [8] to make prettiest graphs with more possibilities of representation.

1.3 The process B

1.3.1 Introduction

As we explain in the introduction, the *process B* permits to extract an abstract model conforms to a component metamodel from plain Java code. In this part we describe this process more precisely.

1.3.2 Architectural view

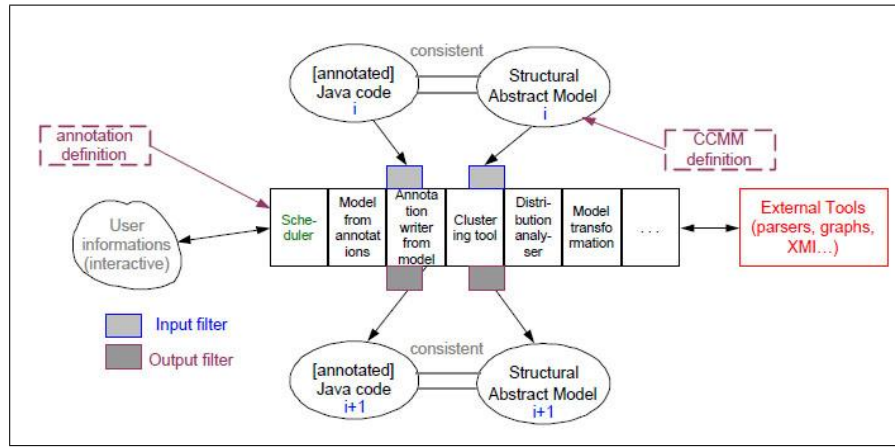


Figure 1.2: Scheme

Details :

A plain Java code (with possibles annotations) and a structural abstract model (not necessary) are in entry of the *process B*. Annotations permit to make links between the plain java code and the abstract model. As we can see on the precedent scheme, the *process B* is an iterative process represented by the markers i and $i+1$. Each iteration of this process visits one box and completes the java code and the abstract model. The java code and the structural abstract model are always consistent at the input and the output of the process. The process is a *Rules Based System* (RBS). We describe the RBS in section 4.2 of this report. Naturally, rules involve an order of application of them, without this order, rules could negate the work done by another. To resolve this problem we have to find the best application order for the rules, we will explain this in the same part. It is impossible to make the process fully automatic so, it will ask the user for additional informations to check which rules can be applied.

Details of the toolbox :

- Model from annotations : permits to generate a model conforms to the EMF metamodel with an annotated java code in entry
- Annotation writer from model : permits to annotated a plain java code though a component model
- Clustering tool : takes a primitive abstract component based model in entry and produce a more precise one with the notion of components composition.
- Distribution analyser : enables the possibility to explore XML model and makes a components based model
- Model transformations : models transformations in general, patterns analyser or automatics transformations, etc.

Now that the context of the project is understood, we have to share out the differents tasks of our work; the next chapter draws up these tasks and the division of them between the members of the group.

The outline of this memo follows from the identified tasks.

Chapter 2

Division of labor

The first work we have to do was to understand the context, the aim and to study the tools of our project. Thus, we have beforehand analysed the JDT architecture (cf. chapter 3), the existing plug-in *TESTJDT3* (cf. section 4.1), the existing rules-based system (cf. section 4.2), the *CoCoME* experimentation (cf. section 4.3) and Rainer Koschke's thesis [11] (cf. section 6.1), what has been a transversal task during the project.

Once this study finished, we can enter the conception phase. But, it is not implementation time yet. Indeed, we have firstly to define the architecture of the new plug-in (cf. subsection 5.1.2), to specify the rules that will be implemented (cf. subsection 5.1.1) and to think about the annotations process (cf. section 5.2).

We know now the structure our plug-in, the algorithm of the rules and the way to integrate the annotations. Consequently, the implementation can begin : on the one hand, the plug-in structure and the process that selects and orders the rules, on the other hand, the processes of extraction of components (rules) and annotation.

The following Gantt diagram illustrates this allocation :

The outline of this memo respects the order of these tasks. Thus, the chapter 3 emphasizes on the JDT architecture. Then, the study of the existing project will be explained in chapter 4. Finally, the work done during the conception phase will detailed in chapter 5. At the end of this document, all the references we have approached are listed in chapter 6; the summary of Rainer Koschke's thesis is in this chapter (cf. 6.1).

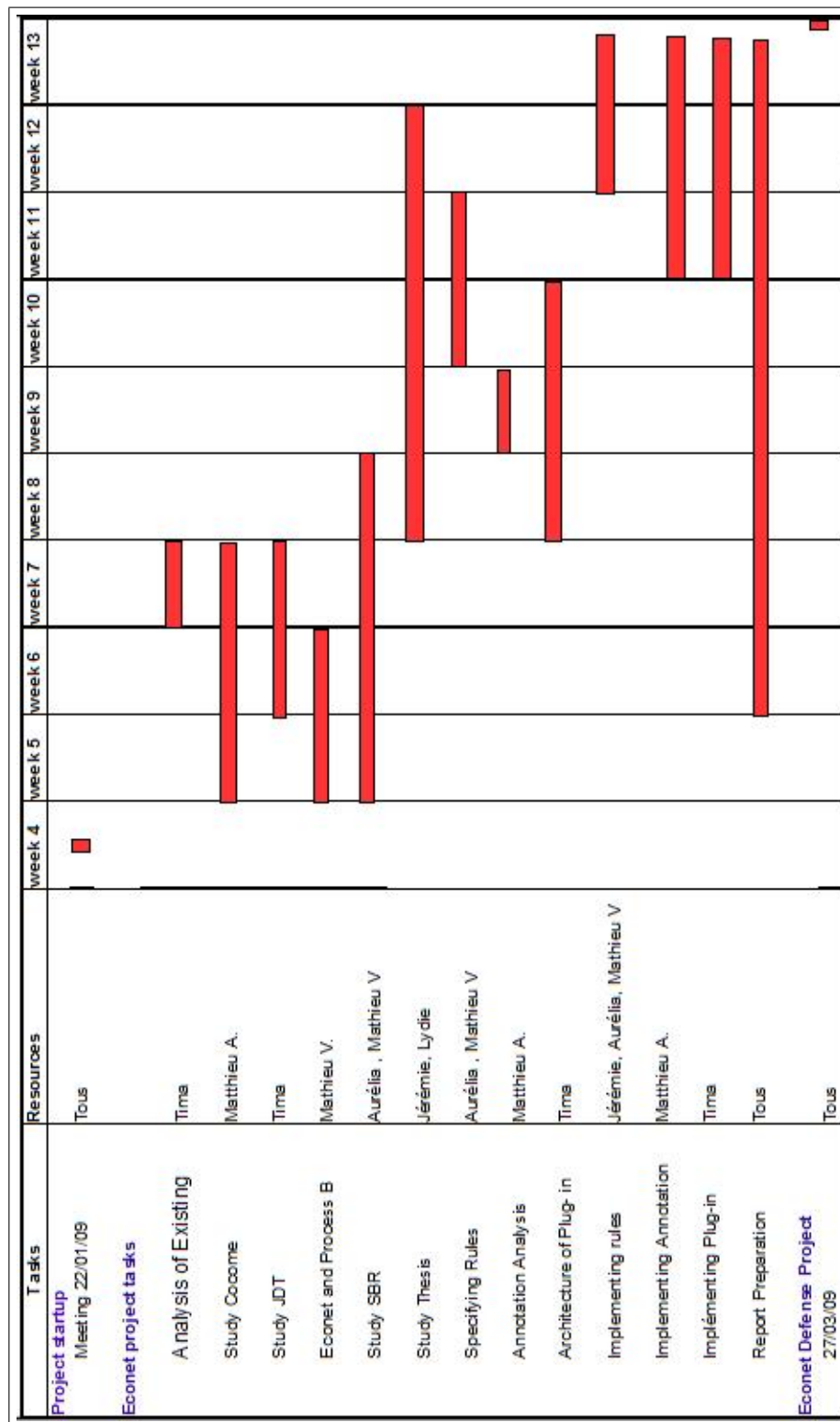


Figure 2.1: Planning

Chapter 3

The architecture of the Eclipse JDT

We did a study on JDT for two reasons : to understand the existing plug-in TESTJDT3 and developing new plug-in Econet. In this study we will make references to the document published by the University of Illinois at Urbana-Champaign, entitled JDT architecture.

3.1 JDT Architecture

The Java Development Tools (JDT) plug-in of the Eclipse platform provides a rich set of functionalities 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.

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.

In this document, we present a broad overview of the architecture of JDT which JDT Core. JDT Core is the plug-in that defines the core Java elements and API.

We present here some elements of package JDT Core.

3.1.1 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.

3.1.1.1 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.

3.1.1.2 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.

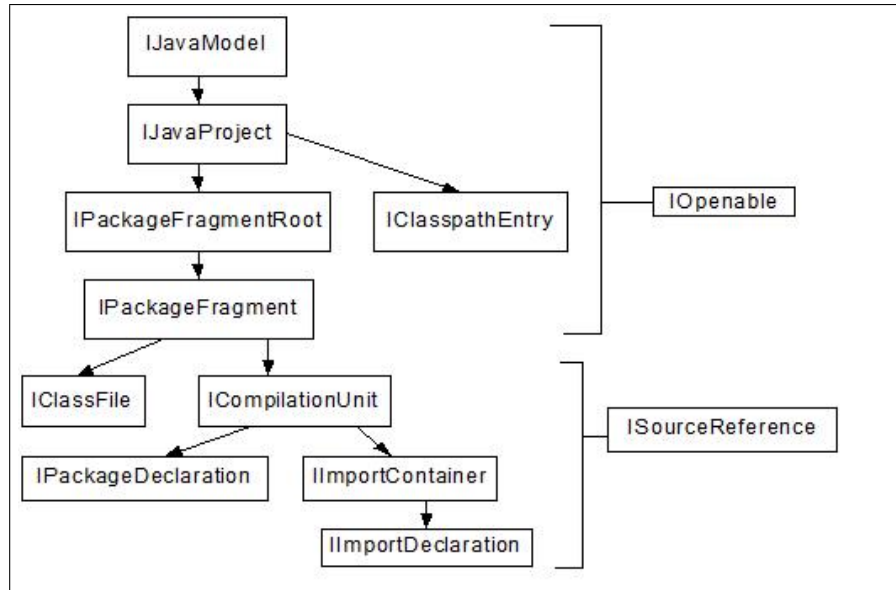


Figure 3.1: Java Model

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.

3.1.2 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.

3.1.2.1 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 :

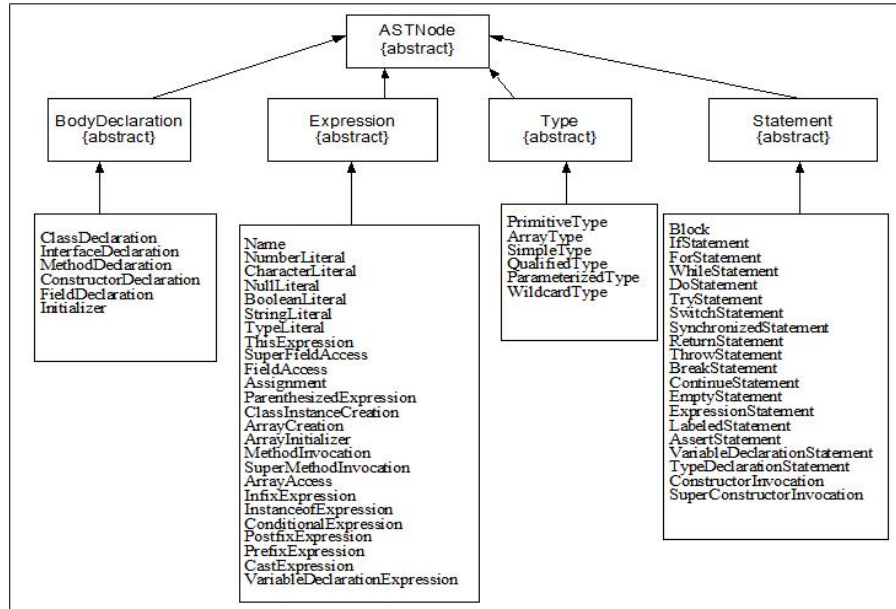


Figure 3.2: AST Hierarchy

- **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.

3.1.2.2 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.

3.2 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.

Chapter 4

Study of the existing project

4.1 Existing Plugin TESTJDT3

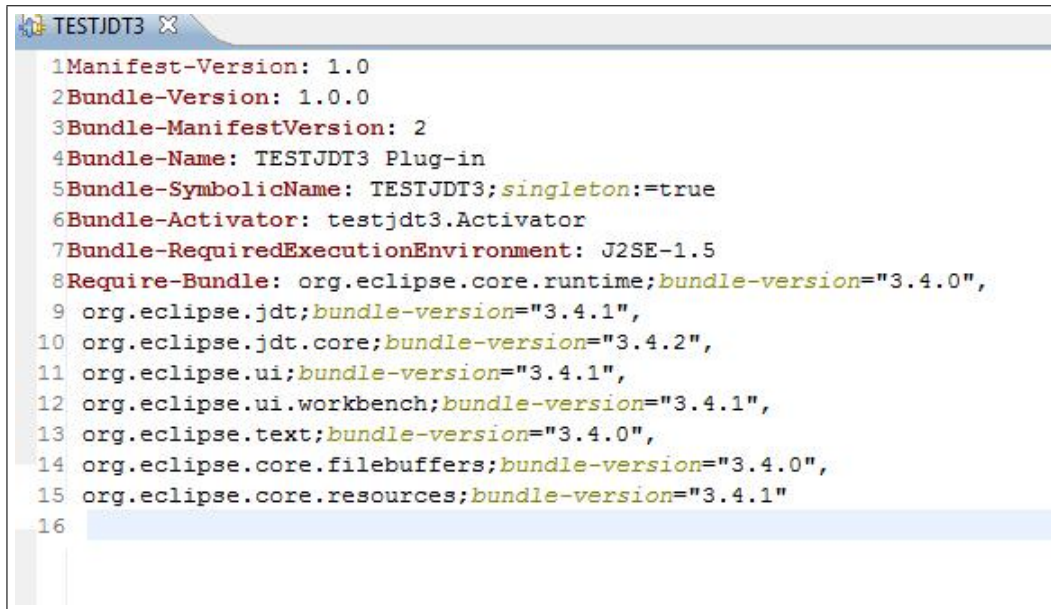
This section explains Econet project's existing plug-in. TESTJDT3 is the plug-in used to extract the component structure of a Java code.

4.1.1 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
```

Figure 4.1: MANIFEST.MF

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.

4.1.2 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* : class 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.

4.1.3 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.

4.2 Existing rules-based system and its properties

4.2.1 Introduction

We have to study carefully the work that have been done before thinking of new rules to implement or modifying the existing ones. Thus, we have to understand the aim and conditions of each potential rule, and make sure that the system is well-formed, regarding three properties that we will define later.

This report deals with the rules that allow us to extract a components-based structure; we are going to explain each of them, detailing the code of the ones which have already been implemented.

In the first subsection, we will explain each rule allowing the extraction of a components structure. We will finish up this section by defining the properties the system has to check in a second section.

4.2.2 Extracting components structure rules

We are going to explain the rules that have been suggested in the documents [architecture-extraction.pdf](#) and `paper.pdf`. Some hypotheses have been established to extract such a structure :

- We consider only static architectures that are created when instantiated;
- we have to respect the encapsulation;
- there is no use of component factory;
- no special implementation pattern, such as EJB, is used.

Other hypotheses have been made out regarding the Java code :

- The program is a source code;
- the code is contained in a unique project, which can import other projects or libraries;
- there is a unique type of interest per compilation unit;
- generic types are forgotten;
- a component type in Java can be an interface, a concrete class or an abstract one, even if it must be instantiated;
- static methods are not considered.

We have now focus our study, so we are able to propose some rules to select the potential applicants for the role of component. Here are these rules.

4.2.2.1 Elimination of implementation classes

This rule is the first step of our work; indeed, it allows us to extract what we call the *types of interest*. We mean by this naming that we consider as potential candidates only the main types defined in the project we study; external and primitives data types are forgotten.

In TESTJDT3, the method *getTypesOfInterest()*, an *ASTActionDelegate*'s member, implements this rule. Let us explain its code.

Before detailing this method, we have to focus on *getUnitsOfInterest()* that selects the compilation units of the analysed project to instantiate the attribute *units* of the AST.

It has none parameter and returns a vector of *ICompilationUnit* (cf. [Eclipse documentation about the interface ICompilationUnit](#)). It browses each internal package fragment root of the AST (extracted by *getAllPackageFragmentRoots()*) and selects their children. The compilation units of each one that contains Java resources are added to the vector to return.

Here are now the details of *getTypesOfInterest()*.

This method takes none parameter and returns a vector of *IType* (cf. [Eclipse documentation about the interface *IType*](#)). Each compilation units of the AST is parsed to extract, thanks to the method *getTypes()* of the interface *ICompilationUnit*, its main type (regarding the hypotheses that have been made, there is a unique type of this level per compilation unit). If the type extracted is a class or an interface, it is added to the vector of *IType* that will be returned.

4.2.2.2 Respect of encapsulation or communication integrity

A component is defined as a deployment autonomous entity, which encapsulates codes and describes, by interfaces, allowed interactions with other components. That is why we expect to respect the encapsulation or the communication integrity.

Now that we have collected the *types of interest*, we want to flag some of them as data types. Thence, subrules have been defined to locate this kind of type.

Identify parameter types in methods

This rule considers methods which are not constructors neither static methods. The type of each parameter that belongs to the set of types of interest is marked as data type. This prevents from violating the encapsulation and the communication integrity.

Each subtype of a type flagged as data type has to also been marked to ensure it is not passed as a parameter. We state that a component type can be passed as a parameter of a constructor. Furthermore, having decided to use none component factory, a component type cannot be the return type of a method.

The class *ASTActionDelegate* provides the implementation of this rule with the method *implementR1(IType it)*.

Its parameter *it* of type *IType* represents an element of *typesOfInterest*. It has no return type.

All the methods of *it* are checked to analyse the type of their parameters and the return one. The extraction of the provided services is done here; we will explain the instructions in the paragraph dealing with the search of interfaces. If the method considered is not a constructor or a main method, its signature is parsed to get the type of its parameters and the one that is returned in order to flagged them as data types.

The method *propagateDATA()* in the class *TypesTable* allows to mark the subclasses of the data types. It is called in the method *run(IAction action)* of the class *ASTActionDelegate* on its attribute *table* types as *TypesTable* after *implementR1(IType it)* is executed.

Identify parameter types in static methods

This rule is, for the moment, only a potential one. As specified in the hypotheses concerning the Java code, we do not consider static methods yet; indeed, none opinion have been established about the connection between the use of such a method and a components-based structure. Nevertheless, we will give the possibility to the user to check the parameters of these methods.

Getters and setters

We estimate that a class disposing of several getters and setters has to be marked as data type; indeed, to handle or modify an attribute of a component breaks the encapsulation. However, it can be useful in order to add a binding. That is why a ceiling percentage could be inserted : if the ratio between attributes having getter(s) and/or setter(s) and those without any is over this percentage, the type is flagged as data type; under, it is a component.

Study the attributes types

A class is flagged as data type if its attributes refer only data types and types that are not types of interest. This rule will be study to ensure none serious applicant for the role of component is marked as data type; the preceding work that has been made advises us to analyse all the communications of a system to verify if this rule is valid.

Regarding interfaces in Java meaning

A type that implements or includes fields referring to a Java interface can be considered as a component. This means that such types that have been marked as data types may target not interesting interfaces regarding a components approach.

Experiences on components-oriented projects have to be done to check this rule.

Enumerations and implementation of data structures

Enumerations and classes implementing data structures are data types.

Public attributes

If a class's attributes are public, it violates the encapsulation, so the class cannot be a component.

4.2.2.3 Composite analysis

At this stage, some data types may have not been flagged. Thus, we will have to test by experiences if our rules and conditions are strong enough, in order to be sure that only components are browsed to find their structures.

The potential composite structures will be extracted thanks to the analysis of the fields. Each type of interest is parsed to collect recursively its structure. The inherited fields have to be collected too.

Another way to extract the structure of a component type consists in analysing the instantiation of the components, from the main program, and following the constructors calls of the components.

The extraction of these structures is implemented in the class *ASTActionDelegate* by the method *implementP3()*.

It takes no parameter and does not have a return type. The *types of interest* are browsed to collect the main method in order to set the root of the AST associated with the analysed project. Then, each type which is not flagged as data type, is parsed to extract its structure.

4.2.2.4 Extraction of communications

This rule consists in extracting the communications from the code of the methods. We estimate that a communication exists from a type *A* to a type *B* by the method *m*, if and only if *m* appears in the code of a method of *A* and *B* provides the method *m*.

In the class *ASTActionDelegate*, the method *implementP4* allows to identify the communications.

It takes neither parameters nor a return type. An AST parser is created to browse the code of the methods of each type of interest.

4.2.3 Research of interfaces

Considering the components types have been identified, we have to specify the required and provided interfaces of each component.

The extraction of the required ones follows from the preceding analyse; the methods such as the method m are collected as required services.

To identify the provided ones, each type of interest has to be parsed to extract the *public* and *default package* methods.

Regarding a communication from A to B by m , we will have to check if a required service m for A is really a provided one for B .

The extraction of the provided services is done in the method *implementR1(IType it)* : the methods of *it* are collected; those which are *public* or *default package* are added to the set of provided services.

The required ones would have had to be collected in the method *implementP5required()*; finally, we just have to look to the extracted methods for each type by *implementP4()*.

4.2.4 Properties of a rules-based system

Such a system cannot be functional without defining an order in the application of its rules. Once defined, two predicates have to be checked :

- is the system consistent?
- is the completeness ensured?

Let us discuss about these properties.

4.2.4.1 Application order

A process will be set up to allow the user defining his own order. However, we have to guide him, establishing a logical order that ensures the two next properties.

- It seems obvious that the study of a project begins with the selection of the *types of interest*.
- In order to centrre the rest of the extraction on the more serious applicants for the role of component, the next step that emerges is to flag the data types.
- Since we do not want to analyse the structure of a data type, now that they are marked, the extraction of potential composite structures can be launched.
- The communications between the *types of interest* can be extracted now.
- The interfaces of the components can be searched and divided into the required and provided ones.

Because the user could want to analyse only a part of a project (defined in a process), such an order could be not essential; indeed, two rules can work on different subsets, so the result of one would not have any effect on the other.

4.2.4.2 Consistency

We mean by this term that the application of a rule does not inhibit a preceding one. Let us discuss about the coherence of a system regarding the defined rules.

After the study of the code of *TESTJDT3*, we notice that none of the rules erases the effect of another. Thus, the consistency of the system is ensured, regardless of the order defined to apply the rules.

4.2.4.3 Completeness

This property means that the defined rules have to consider all the possibilities they could meet. Nowadays, we do not have given a ruling concerning this predicate yet.

4.2.5 Conclusion

As a conclusion, we first emphasize on the difficulty to understand an existing study; the first thought of the problem has led to constraints, questions, hypotheses, etc. that we have not consider. Thus, the existing rules are sometimes complex to interpret. Moreover, we have to apprehend the drawn up architecture to be able to understand the code of the methods.

That is why experiences are indispensable. Indeed, some rules have still to be validated or have to be strengthen. Thus, one of the next steps in the analyse of our rules-based system consists in comparing a manual study of a project with one resulting of the execution of *TESTJDT3*.

4.3 An experimentation : CoCoME

4.3.1 Introduction

CoCoME is the abbreviation of Common Component Modeling Example.

This is a program representing a sell system. We won't detail how it works because it's not what we need. We will focus on the architecture of the program, we want to analyze components, interfaces and other elements we need, in order to have a base example to use and to see what will give us the result of the plug-in we are working on.

4.3.2 CoCoME

The major component is TradingSystem.

At the base of the program we have two major components named Inventory and CashDeskLine. Inventory and CashDeskLine are connected with two interfaces:

- CashDeskConnectorIf is the provided interface of Inventory and required by CashDeskLine;
- SaleRegisteredEvent is provided by CashDeskLine and required by Inventory.

Bank is provided by CashDeskLine and is the external interface of the complete program.

This is the global architecture of the program :

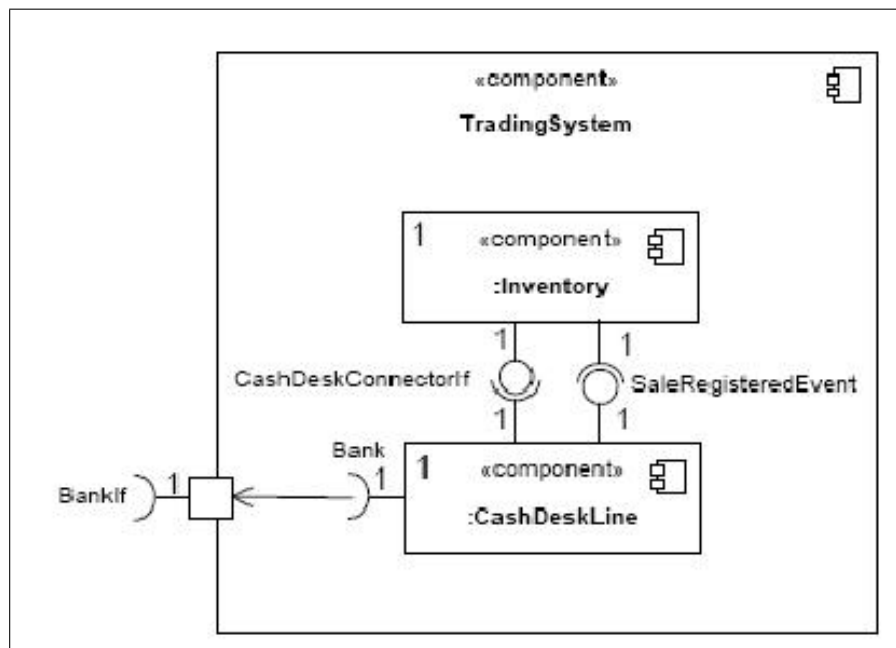


Figure 4.2: Global program architecture

We won't show other diagram of the program as this is not the goal of this document. We only remember major details of the program.

4.3.2.1 Inventory

This component contains four components:

- Gui, this is the global user interface which require the informations provided through the interfaces StoreIf and ReportingIf provided by the Application component.
- Application, it makes the link between Inventory and the other contained components. That's why it has the interfaces SaleRegisteredEvent and CashDeskConnectorIf. It needs informations from Data, so it has three interfaces required, those are named EnterpriseQueryIf, PersistenceIf, StoreQueryIf.
- Data has the required interface JDBC
- DataBase.

4.3.2.2 Data

- Enterprise provide EnterpriseQueryIf
- Persistence provide PersistenceIf
- Store provide StoreQueryIf

These three components manage informations from their name (the component Enterprise manage information from the enterprise, ...).

Each of the components contains base classes. As for the component **Enterprise** :

- TradingEnterprise
- ProductSupplier
- Product

4.3.2.3 Application

- Reporting
- Interface provided \rightarrow ReportingIf (for Gui)
- Interface Required \rightarrow EnterpriseQueryIf, PersistenceIf, StoreQueryIf
- Store
- Interface provided \rightarrow CashDeskConnectorIf (for Inventory), StoreIf (for Gui)
- Interface Required \rightarrow PersistenceIf, StoreQueryIf, SaleRegisteredEvent
- ProductDispatcher
- Interface provided \rightarrow ProductDispatcherIf
- Interface Required \rightarrow EnterpriseQueryIf, PersistenceIf, StoreQueryIf

4.3.2.4 Gui

- Reporting
- Store

For the rest of the program we will not focus on the part about the CashDesk, but we'll see major components.

4.3.2.5 CashDeskLine

- CashDesk
- EventBus
- Coordinator

We will not detail all of the components present in CashDesk, they are principally components used for controllers like ScannerController, CardReaderController.

4.3.3 Plug-in

Now we will talk about the result we are having by passing the CoCoME in the plug-in, and we will check if results seems correct to our previous analyze of the architecture.

After we've used the plug-in, we obtain : around 59 components for all of the program, which is really more important than we should have. Moreover we have some components like Store which are not detected as components by the plug-in and this for every classes that composes his package. There are some other component identified but some should not be, like some classes used for tests of the database in order to fill it. They are also tag as ROOT component and we don't think it's could be true event if they were components.

We've just take these examples, maybe they aren't relevant, but they are chosen to point on incoherences in the plug-in. So we must take the rules and determines why we have those results and correct them.

We did not find any cycle in the program, this should be good, maybe we could make an example with a cycle in order to test the rules why manage this point.

Most of the binding are not resolved.

Seems that most of the interfaces are identified, for the whole program, the plug-in has identified 120 interface, but due to the result, we probably have more than we have really, 120 seems to be pretty huge, but it certainly is coherent if we take in consideration the number of components identified. But has we only need to see the key "interface" in the program, this should be strange to identify other interfaces.

In the same way, communication are few in comparison with the number of interfaces but most of them are pertinents.

To finish with this, we will simply say that most of the plug-in identify what we need, but it needs to be refined in order to match everything we need and correctly.

What have we get?

First of all, we have 9 elements which are not present in the results, mainly they are enumeration classes. 6 are sure to be enumeration.

Then we have around 45% of the interfaces which are flagged as `DataType`, this implies that the first version of the rule about `DataType` is not accurate. We also have 9 elements not identified that are interfaces. An example is :

```
application.productdispatcher.ProductDispatcherIf -> Data type  
application.productdispatcher.OptimisationSolverIf -> undefined
```

After reading each of these interface, the only major difference between them is the presence of a “public” before the method of the `OptimisationSolverIf`. But after reading the other interfaces, we think it is because the first element of the contained methods is a `DataType` or not.

For the component, we find too much component, we have not a value, but we have all of the exception classes, or elements of the interfaces that are marked as component. But it is not what we want, we should have get a flagged component for each of the implementation like `StoreImpl` for the `Store` component.

Around 50% of the elements does not have any binding found.

Our study is now in the conception phase. We will have to think about the architecture of the plugin again, to allow an evolution in a hierarchical system.

Chapter 5

Conception

5.1 Detailed Conception of Econet plugin

In this section, the different stages of the Econet plug-in conception will be explained. Firstly, the rule-based system will be explained, entities used in the process, then UML conception that concluded with the realization of the first version of itself;

5.1.1 Rules-based system

The plug-in architecture is built, so we have to specify the rules which permit us to extract a components structure from a plain java code.

5.1.1.1 Specification of the rules

- **Get the types of interest**

Name : *GetTypesOfInterest*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : set of IType

Explanations : this rule corresponds to the method *getTypesOfInterest* of the existing plug-in. It uses the method *getUnitsOfInterest* from the class *ASTActionDelegate* to find the compilation units of the project. This rule returns a vector of IType and will allow the instantiation of the attribute *typesOfInterest* of the class *ASTActionDelegate*, browsing each compilation unit of a project which is analysed and by recovering the high-level type of this unit.

- **Flag data types**

Name : *FlagDataTypes*

Extends : *GroupOfRules*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : this rule enables the recognition of the data types. Its sub-rules are detailed below.

- **From no static method parameters (without constructor)**

Name : *FlagMethodsParametersType*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : this rule enables to parse the methods, other than the constructors and the static methods, of the set *typesOfInterest* and to flag as data types the type of their parameters.

- **From static methods parameters**

Name : *FlagStaticMethodsParametersType*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : this rule enables to parse the static methods of the project. Note that the relationship between static methods and a component approach is not defined yet ; the method will be implemented in case of the user need it.

- **From getters and setters**

Name : *FlagFromGettersAndSetters*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : this rule enables to flag as data type a class which contains more than 70% of attributes having getter(s) and/or setter(s). The value of 70% is not a final choice, experimentations will show us if we have to modify it.

- **From the types of the attributes**
 Name : *FlagFromNonComponentAttributes*
 Extends : *SimpleRule*
 In parameter(s) : *ASTActionDelegate*
 Out parameter(s) : void
 Explanations : this rule enables to flag as data type classes with attributes referencing only implementation classes or data types. To valid this rule, we have to study the communications in an experimentation.
- **From enumerations**
 Name : *FlagEnumerations*
 Extends : *SimpleRule*
 In parameter(s) : *ASTActionDelegate*
 Out parameter(s) : void
 Explanations : this rule enables to flag enumerations as data types.
- **From the visibility of the attributes**
 Name : *FlagFromAttributesVisibility*
 Extends : *SimpleRule*
 In parameter(s) : *ASTActionDelegate*
 Out parameter(s) : void
 Explanations : as the rule about getters and setters, a threshold is introduced. If the percentage of public attributes is higher than this threshold, the type is flag as a data type. The value is initialized with the value of 70%. This value will be modify if the experimentations are not satisfying.

- **Flag components**

Name : *FlagComponents*

Extends : *GroupOfRules*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : This rule enables to flag the components. Its details are below.

- **Flag components from Java interfaces**

Name : *FlagFromJavaInterfaces*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : This rule enables to flag as component a type which implements a Java interfaces or contains fields that refer to such an interface.

- **From the type declaration**

Name : *FlagFromTypeDeclaration*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : this rule has been extracted from the composite structures extraction rule of the plug-in TESTJDT3 : types of *typesOfInterest* that are still unresolved are browsed; if they are declared as classes, they are flagged as components. The experimentation on CoCoME has not been conclusive, that is why this rule may be modified, or even disappear.

- **Extraction of composite structures**

Name : *AnalyseCompositeStructures*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : this rule enables the study of components complex structures. From the main program, this rule follows the constructors calls and finds components which are in other component. Of course this rule only follows the constructor of the components types.

- **Extraction of the communications between components**

Name : *FindComponentsLinks*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : this rule enables the analysis of sent messages. We consider that a component A communicates with a component B if A provides a method m and B calls it in one of its methods. This rule makes the different links between the identified components. As the previous rule, the one that recovers the types of interest has to be executed before.

- **Identification of equired and provided interfaces**

Name : *FindInterfaces*

Extends : *SimpleRule*

In parameter(s) : *ASTActionDelegate*

Out paramter(s) : void

Explanations : this rule enables to find the required and provided interfaces of the components. It browses the *ASTActionDelegate* and the methods of the components, and study the methods with the visibility *public* or *package*.

5.1.2 Econet plugin structure

So far, the system's rules have been defined. However, entities necessary for the extraction process of a Java component structure have to be defined.

After reflection, the extraction program has been broken down in a number of classes.

One needs:

- A main class that instructs start command to the others; it initially collects the project whose component structure is to be extracted and calls upon the rules manager to apply in order rules provided by the user. This class will contain information regarding data types of Java code and an AST whose structure will be used for the extraction rules
- A class that will play the rules manager, applying rules chosen by the user and provided by extraction policy on the process. These rules are of type simple and/or complex.
- A policy rules extracting class that collects rules to be applied during process in the rules manager.
- A component class that will model a component, its type and structure.

The following class diagram was then established :

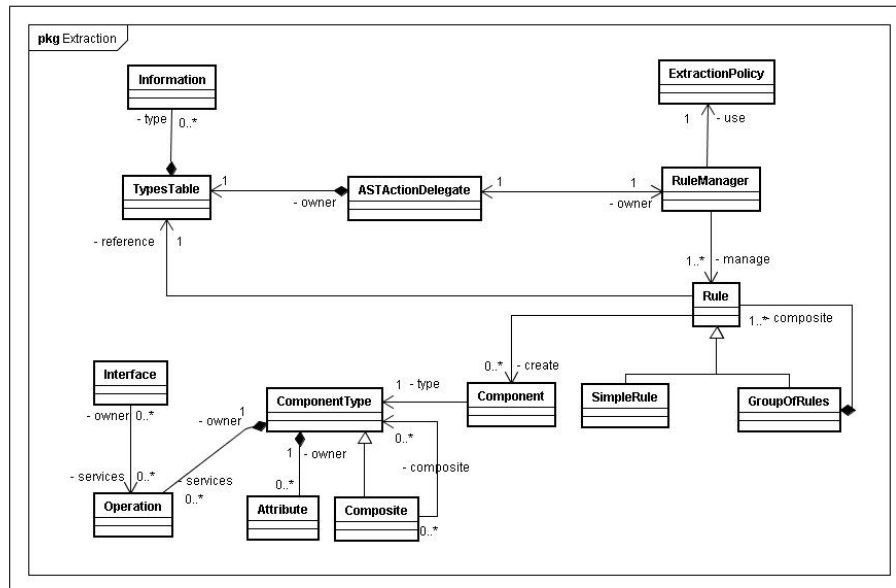


Figure 5.1: Plug-in class diagram

Class names were chosen as follows:

- ***ASTActionDelegate*** : The main class.
- ***TypesTables*** : Class containing persistent data names and information.
- ***Information*** : Class detailing information about a type of persistent data structure
- ***RuleManager*** : Class acting as the rules manager
- ***ExtractionPolicy*** : Class that collects in a file the order of the rules as defined by the user.
- ***Rule*** : Abstract class defining extraction rule.
- ***SimpleRule*** : A ***Rule*** subclass representing a simple rule. .
- ***GroupRule***: A ***Rule*** subclass representing a set of rules simple and/or complex.
- ***Component*** : Class defining a component associated with a component type ***ComponentType***, that holds attributes ***Attribute***, operations ***Operation*** interfaces ***interfaces*** and whose structure could be composite ***Composite***.

The following figure illustrates the ins and outs of the extraction program.

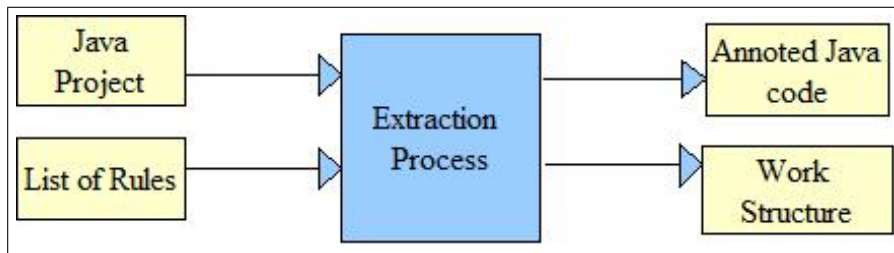


Figure 5.2: Ins and outs of the extraction program

The program must:

- Take as an argument the name of a Java project existing in the workspace environment and a file containing order of the rules to apply in the process chosen by the user.
- Supply an annotated Java code as well as a work structure in return. The work structure will contain information about the extraction process on rules applied throughout the process, their results, temporary elements (classes, types, interfaces, methods) with their appearance level (stage in the process) and verification of the structure result.

Our renamed Econet project, is a plug-in intended to work on Eclipse. It will therefore be implemented in Java, and will comply to inherent construction norms of plug-ins.

5.1.2.1 Eclipse

Eclipse is an IDE: Integrated Development Environment. Eclipse is the project principal tool and is a very powerful development platform.

Eclipse kernel is actually composed of :

- A base capable of loading modules (plug-ins)
- Integrated modules enabling management of a set of resources (projects, files, folders,...)
- Modules enabling creation of coherent graphical interfaces. Using this library, plug-ins keeps a homogeneous aspect.

The Java development part is indeed a set of plug-ins that constitute the first usage of the base. Their principal characteristic is to be delivered with Eclipse. Note that plug-ins forming C++ development environment are also available, although not in Eclipse standard version, it requires a separate download.

Eclipse could therefore be used as the basis of any given development tool, no matter the language and the file formats supported.

Eclipse is found in different guides, workbench user guide, Java development user guide, Platform plug-in developer guide, JDT plug-in developer guide, PDE guide, Eclipse UML plug-in user guide.

5.1.2.2 Plugin integration in Eclipse environment

This section discusses the integration of the plug-in in Eclipse. Similarly to former TESTJDT3 plug-in, the new plug-in named “Econet” is displayed on Eclipse menu bar in its integration.

However, the process procedure is different; it was designed to facilitate its operation. In order to simplify several actions, Eclipse offers a wizards system. Thus, one of the newest features of the plug-in is the use of wizard to define the extraction process.

These wizards are built like a sequence of dialogs providing all required options. Four new wizards were created.

- LoadWizard : Retrieve the Java project whose component structure is to be extracted.

The figure belows illustrates LoadWizard class.

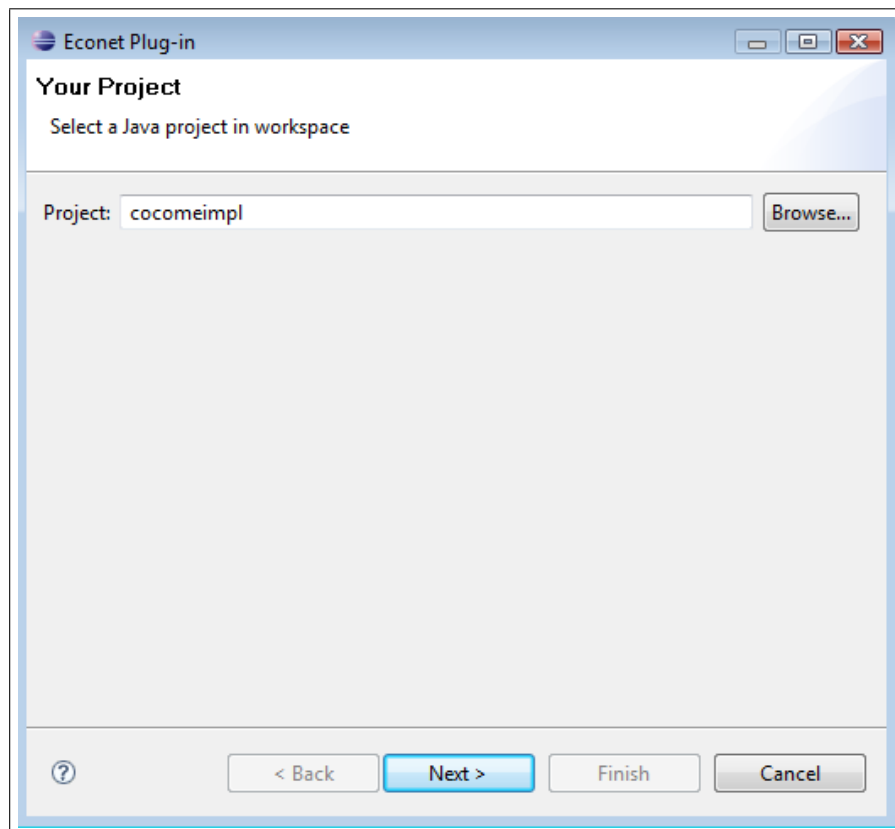


Figure 5.3: LoadWizard class

- RuleWizard : Provide the choice for the rules applied during the extraction process.

The figure belows illustrates RuleWizard class.

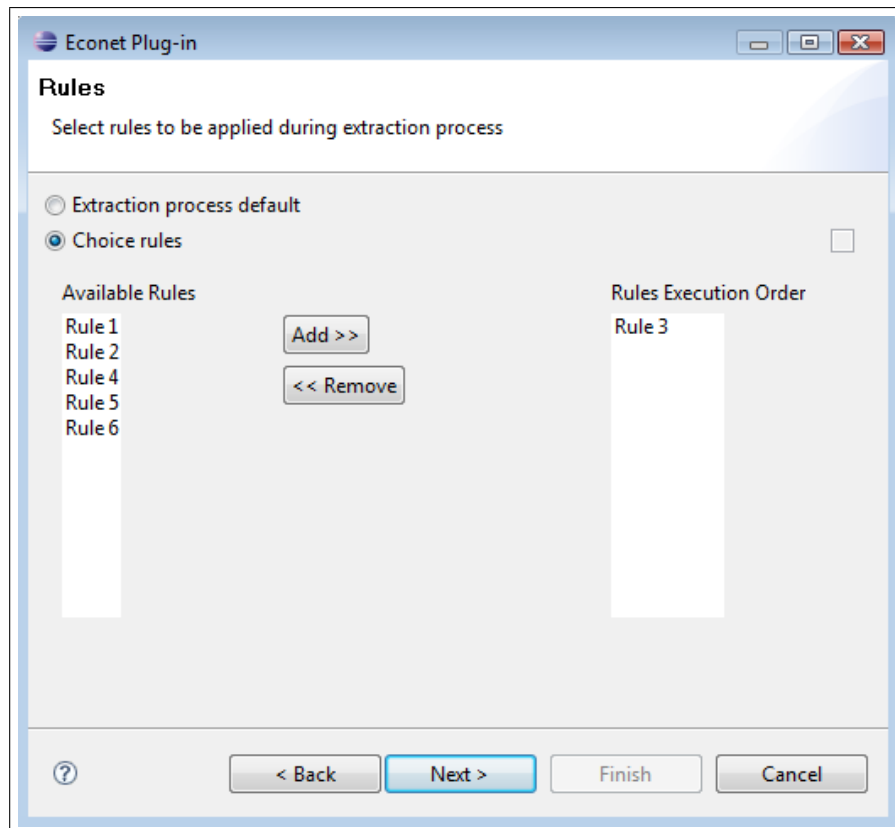


Figure 5.4: RuleWizard class

- ExtractionProcessWizard: Launch the extraction process.

The figure belows illustrates ExtractionProcessWizard class.

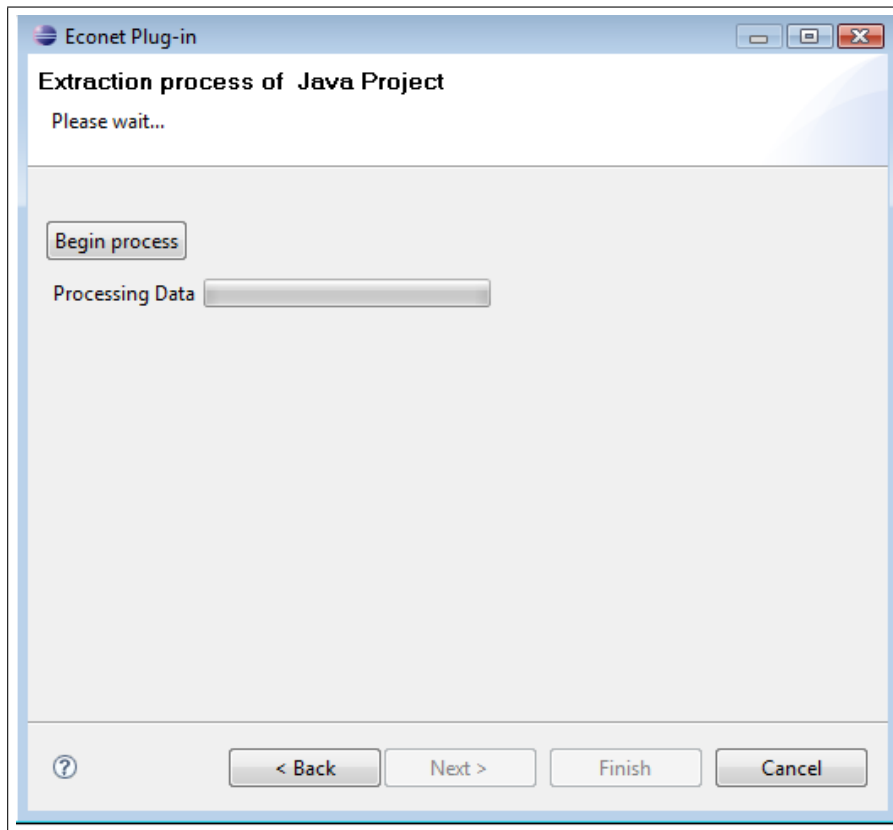


Figure 5.5: ExtractionProcessWizard class

The wizard structure enable the user to select the project name to be analysed and extraction rules directly instead of having to type it in a dialog.

To install the plug-in one, simply copy « Econet » folder in Eclipse « plugins » folder. Finally, running Eclipse will complete the installation.

The plug-in re-used was functioning, although its was greatly modified and enhanced in order to improve its structure, implement the rules, and its utilisation.

Classes such as `ASTActionDelegate`, `TypesTables` strongly inspired the final version of the plug-in. Features have been added over the earlier version: integration of Econet wizard, choice of rules, generation of Java annotated code.

5.2 Annotations

5.2.1 How will we get these annotations?

After some reflexion, we were thinking about two way of getting them:

- Getting all informations after all of the rules were done, and then getting each of the informations we needed in order to make a logical structure of the program.
- Getting informations while we are using the rules, in order to catch each element as soon as they are analyzed and make the structure in “real-time”.

We will describe each of them, this way we will be able to choose which one would correspond the most with our methodology, and what we really need.

1. In this method, we first will need to get all of the components which were flagged. So that we could do the first step of the annotation `@InComponent(annotationSrc,componentName)` and `@InitMethod(annotationSrc,componentName)`. We should identify each of the interface used, and match them with the components that used them in order to do the link between a component and his interfaces. This will perform the fill of the annotation `@provided(...)` and `@required(...)`. After that, as we have the rule “composite” we can find each sub-component of each components, and then finish with the annotation `@InComponent` to modify the last part `componentName` in order to match the composite.
2. In this method, the principle is yo flag each element that we are checking with its proper annotation and then, each time there is a new information we modify each annotation related with the newly element checked. In this method we need to care about the possibility of being incoherent. This should not happened if each of our rule are coherent.

After reflexion, we think that the first method is better for our project, because first, if this method is running after all rules were being applied, this implies that this method is independent of the rules, so that we don't have to modify our method if the rules changes. Then, another point is that each rules are connected, we could not have a directly a good result if we use only one rule, so if we were including part of the method inside rules, we would have been blocked in the system of annotation if the user was not choosing one rule.

5.2.2 How have implemented this and why?

We have done a new class that contains each of these methods, to works it only need the ASTActionDelegate were every informations are contains. First as there is no information about the Ipath of each element contained in the AST, we will use the directory of the system. So first we get the root directory of the project in a string, then we append it the system file separator. When we've done that we are ready to do each method for the annotations.

doComponent which will see in the table attribute of the AST if each of the element is a component, and if the current element checked is a component then it will modify it in order to get the annotation @InComponent, and @InitMethod. We can only make the match with string, because we can't check the structure of the component, as we've said before we don't have the IPath so that we can't have more information about the component (like the matching with IMethod, or other element we would have needed).

The procedure for the other method are quite the same so we won't develop them.

Chapter 6

Bibliography

6.1 Rainer Koschke's thesis summary

Meanwhile, in order to better understand the concepts of software architecture and components, and to achieve our system of rules extraction, we need a study on a thesis.

This thesis, entitled « Atomic Architectural Component Recovery for Program Understanding and Evolution - Evaluation of Automatic Re-Modularization Techniques and Their Integration in a Semi-Automatic Method » is structured into two main parts. The first part deals with automatic techniques and the second part with a semi-automatic method for atomic component detection.

6.1.1 Extraction process

One of the main work of this thesis is that on the extraction of a structure consisting of a source code.

There are basically two kinds of static components that are to be detected by architecture recovery: Subsystems and atomic components. The two of them differ in their level of granularity: Subsystems may comprise architectural quarks, atomic components, and lower-level subsystems whereas atomic components consist of related global constants, variables, subprograms, and/or user-defined types only.

- Atomic Components

An atomic component can be seen as a named set of architectural quarks. In our relational model introduced in the previous section, we can capture this as follows :

 - atomic components are represented by a new entity type
 - the fact that an entity E belongs to an atomic component AC is expressed by a part-of relationship: E is a part of AC.
- Subsystems

Subsystems are a means to represent hierarchical sets of related elements (architectural quarks, atomic components, and other subsystems) whereas atomic components can be thought of as flat sets of related architectural quarks. Subsystems must contain at least one atomic component three quarters a component with architectural quarks only is considered an atomic component.

The model below shows the extract of the extraction process :

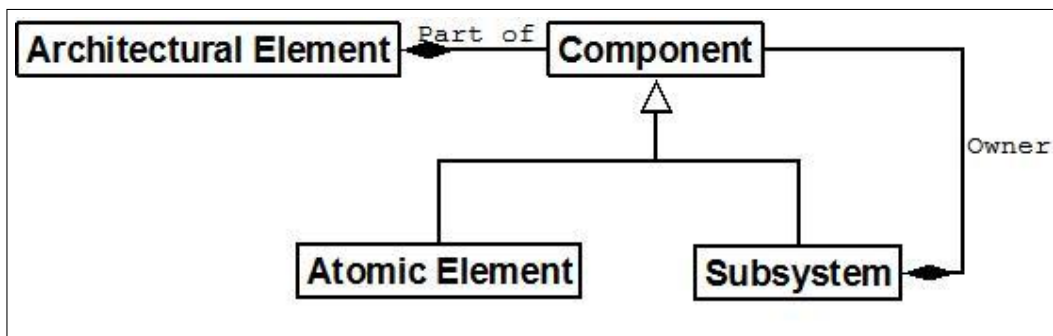


Figure 6.1: Extraction process

6.1.2 Extraction technique

Two types of techniques used techniques automatic and semi-automatic.

6.1.2.1 Automatic Techniques

Basic Techniques

Algorithm For *Global Object Reference*, we can use the generic algorithm, which will also be used for other methods that produce disjoint clusters. The algorithm iterates over the subprograms and groups them with their relevant connected entities. What a relevant connected entity is depends upon the respective technique; in terms of the algorithm, this is decided by a generic parameter that yields for each subprogram all entities that have to be part of the same atomic component as the subprogram. This function could also exclude frequently used objects as proposed by Yeh et al. for the *Global Object Reference* heuristic.

Generic algorithm to detect disjoint atomic components

Generic parameter :

- function *connected_entities*: Entity \textcircled{R} set of Entities

Input :

- input view V

Output :

- disjoint clusters

Algorithm :

1. *put each base entity in V into a set of its own* :
for each entity E in V loop
 new_set (E);
end loop;
2. *clustering* :
for each entity E in V where subprogram (E) loop
 for each entity E' in connected_entities (E) loop
 union (find (E), find (E'));
 end loop;
end loop;
3. *results* :
 each remaining disjoint set is a cluster

- *new_set (e)* defines a new set $\{e\}$.
- *union (s1, s2)* unites the two sets *s1* and *s2*; after the call, the two set identifiers denote the same set, i.e., $s1 \cup s2$.
- *find (e)* yields the set that contains *e*; since all entities will be initially put into a set and since the sets are disjoint, there is exactly one such set.

In the ideal situation, i.e., when the system is properly decomposed, each module contains one single atomic component. When we count on good design, we can group all declarations of a module together to an atomic component which represents the abstract functionality of the module. This is the underlying clustering criterion of the Same Module heuristic.

6.1.3 Schwanke's Arch Approach

The techniques described above compare pairs of entities by their direct relationships in order to decide whether they belong to the same atomic component. However, a complementary source of information is the environment of the compared entities as stated by Schwanke (1991) :

“If two procedures use several of the same unit-names, they are likely to be sharing significant design information, and are good candidates for placing in the same module.”

And not only what kind of entities (unit-names) they commonly use increases their relatedness but also by which common entities they are used. For example, the implementations of a sine and a cosine function will both have a float parameter and result type, but they also will likely be used in the same context, i.e., have common callers. Schwanke's approach takes this into account.

Schwanke's work is aimed at module detection. Subprograms are clustered into modules based on a similarity metric.

Similarity Clustering algorithm

```
place each routine in a group by itself
repeat
    identify the two most similar groups
    combine them
until the existing groups are satisfactory
```

Clustering criterion. In each iteration, the most similar groups are combined using the similarity metric described below.

Similarity between subprograms. The group similarity used to combine groups in this algorithm is based on a similarity between subprograms. Given two subprograms A and B , the similarity metric used during clustering is defined as follows:

$$Sim(A, B) = \frac{(Common(A, B) + k * Linked(A, B))}{(n + Common(A, B) + d * Distinct(A, B))}$$

wherein $Common(A, B)$ reflects the common features of A and B and $Distinct(A, B)$ reflects the distinct features. $Linked(A, B)$ is 1 if A calls B or B calls A , otherwise it is 0. The two parameters $k \geq 0$ and $d \geq 0$ are weights given to $Linked$ and $Distinct$ in *Similitude* have to be ascertained by experiments on a sample of the subject system. The parameter $n \geq 0$ is used for normalization purposes; it will be considered 0 in the following.

Group similarity. Based on the similarity between subprograms, the similarity for groups is defined as the maximum similarity between any pair of group members one from each):

$$GSim(A, B) = \max(Similitude(a, b)) | a \in A, b \in B$$

Extensions. An extension to this approach was proposed by Schwanke himself in joint work with Hanson in 1994. In the extension, they use a nearest neighbor approach to classify components.

In order to distinguish the original approach from 1991 from the extension of 1994, the former approach is called the **Arch approach** and the latter the **iArch approach** following the terminology Schwanke used in his papers. The original Arch approach was extended to detect atomic components by Jean-François Girard, Georg Schied, and me in many ways. The enhancements are so manifold that the extension can be considered a new approach.

6.1.3.1 Similarity Clustering

Similarity Clustering is the most general approach described in the thesis. It can detect abstract data types (ADT), abstract data objects (ADO), hybrid components, as well as groups of related routines. All connection-based techniques can be subsumed under *Similarity Clustering*. *Similarity Clustering* goes beyond other approaches in that it also considers relations to common third entities and informal aspects.

Similarity Clustering can be used in two different modes :

- search for specific user-defined patterns
- search for similar patterns of already found atomic components

Whereas connection-based techniques always yield the same candidates, *Similarity Clustering* can be adjusted by the maintainer to different search patterns by changing its edge weights. The adjustable parameters of *Similarity Clustering* offer more flexibility. On the other hand, when the maintainer wants to search for atomic components similar to those already found, these parameters can be automatically calibrated by the set of known components using traditional optimization techniques, such as simulated annealing or Gauß-Seidel optimization. The sample used to calibrate *Similarity Clustering* can be ascertained with other techniques or withments have been applied. Experiments with calibration methods for the subject systems used in this thesis indicate that a sample of 20-30% of the groupable base entities of a system grouped to atomic components is a sufficient training set (a base entity is said to be *groupable* if it actually belongs to an atomic component; recall that not all base entities were grouped to components by the software engineers for our subject systems). However, the data have not shown that one could improve the recall rate of *Similarity Clustering* by using larger samples. This is probably because of the diversity of characteristics among atomic components. For example, some atomic components may be properly encapsulated such that high weights for record components and variables references will yield good results. Some others may be permissive atomic components for which higher weights for record components and variables references will also add many nonaccessor functions that break the information hiding principle. *Similarity Clustering* after specific user-defined adjustments have been applied. Experiments with calibration methods for the subject systems used in the thesis indicate that a sample of 20-30% of the groupable base entities of a system grouped to atomic components is a sufficient training set (a base

entity is said to be *groupable* if it actually belongs to an atomic component; recall that not all base entities were grouped to components by the software engineers for our subject systems). However, the data have not shown that one could improve the recall rate of *Similarity Clustering* by using larger samples. This is probably because of the diversity of characteristics among atomic components. For example, some atomic components may be properly encapsulated such that high weights for record components and variables references will yield good results. Some others may be permissive atomic components for which higher weights for record components and variables references will also add many nonaccessor functions that break the information hiding principle.

Similarity Clustering is one of the most effective techniques as far as the recall rate is concerned. On the other hand, it has also more false positives than other approaches (except for *Arch* which has more false positives). In earlier variants, the number of false positives was even worse (Girard et al. 1997c). *Rainer Koschke* additions resulted in a substantial reduction of the false positives in comparison to results previously published.

Another advantage of *Similarity Clustering*, as a hierarchical clustering method, is that it yields a dendrogram of clustered entities instead of a set of flat candidates. This is in particular useful for validation. In the quantitative evaluation of *Similarity Clustering*, branches of the dendrogram were cut and converted into candidates using the same similarity threshold. However, this assumes that the same threshold is suitable for all components. Using a single threshold was necessary in a fair comparison to other automatic techniques; in an interactive approach, one does not need a threshold at all. Hence, less false positives can be expected for a hierarchical view.

There are also some drawbacks of *Similarity Clustering*. For all techniques other than *Similarity Clustering*, there is one single criterion used for clustering. Hence, the reason why a technique has grouped entities together is obvious. This is less obvious for *Similarity Clustering* when the similarity metric considers several aspects at the same time. This complicates validation of candidates proposed by *Similarity Clustering*.

When *Similarity Clustering* considers informal information, it may happen that entities are clustered that are not even transitively connected to each other just because they have similar names. This may be useful when groups of related subprograms are to be detected. However, for ADT and ADO detection, the entities are always at least transitively connected via call, type, or reference relationships. Fortunately, unconnected entities can

be easily filtered from candidates if this is necessary. In the quantitative comparison reported in this chapter, a filter for candidates with unconnected entities was not used. Using such a filter will probably lead to less false positives. Furthermore, future extensions of *Similarity Clustering* should try to use informal information only for nodes that are either first or second-degree neighbors. Then, informal information would only be an additional hint but not a sufficient criterion for two nodes to be in the same component. This would also reduce the time complexity for computing the similarity matrix as discussed next.

The computational effort needed for *Similarity Clustering* is higher than for all other techniques. This is mainly due to establishing the similarities among the entities while clustering as such is comparatively fast. Rainer Koschke give hints on how the complexity can be reduced. It turned out that time and space complexity for *Similarity Clustering* is basically linear to the number of entities, n , when informal information is excluded (assuming an upper constant limit of neighbors an entity can have). However, when informal information is used, each entity has to be compared to any other entity resulting in a time complexity of $O(n\check{s})$. If the proposal above to use informal information only for first and second-degree neighbors is put into action, however, the complexity can be reduced to $O(n)$.

6.1.4 Semi-automatic techniques

6.1.4.1 Combined and Incremental Techniques

In the first part of the thesis, the basic techniques and their evaluation have been described. The evaluation found that showed basic techniques fail to match the quality of its people. Therefore, improvements can be expected by combining these basic techniques. In addition, new techniques can be added to these existing techniques. On the other hand, the different basic techniques are appropriate for certain types of atomic elements, but not for others (for example, to detect ADO: Abstract Data Object). They can also be refined by means of control and analysis of data streams. In all cases it is important that the responsible action.

Several combinations are possible through a combination of some operators such as union, intersection, and difference. The information provided by the user can be captured and used to enrich the analysis. The thesis also describes a method of voting, as an addition to the combination of operators, in which the agreement of each technique with a given element, the underlying is expressed in metric units. An atomic component candidates can then

be evaluated by adding the various measures.

To validate a candidate, the user has several choices: he or she may add to, accept, and reject partially and completely. All this information is retained for the next iteration.

To distinguish the candidates reference components, a graphical view of resource use can be valuable. A view that shows the breakdown of the components is called point of view. All results of the basic techniques can also be represented by components views. There are essentially two types of information contained in these respects: the positive and negative information. Positive information results from a decision of the user that some elements are consistent, while the negative information occurs when the user decides that certain elements are not compatible or mutually exclusive.

Finally, the user can manipulate a component as follows :

- creation : create a new component
- award : add an entity to a component
- rejection: delete a related entity from its components (which does not imply that the entity and its components are mutually exclusive from now on)
- exclusion : mark two mutually exclusive
- confirmation: confirmation of the information is added to a user

6.1.4.2 Semi-automatic techniques for the detection

Some techniques introduced above are fully automatic, which is desirable especially for large systems. However, the evaluation revealed that none of them has the quality of human detection. There are basically two ways to improve the quality of the detection process. We can look for more sophisticated techniques or include the user in the process.

However, despite this, the user remains the final judge. Because of the complexity, vagueness, and to a certain degree of subjectivity, it is questionable whether we can always find specific techniques that correspond to all cases. The semi-automatic method is that the process can be divided into tasks for the computer or the person responsible. The results of each task performed by one of two partners, human or computer is used for the other partner in the next task. The user controls the detection process by the selection of analysis and metrics, and validation of candidates proposed by the automatic technique.

The task of the computer analysis includes automatic calculation of parameters of the proposed candidates, presentation of results, and the accounts of the user decisions. Candidates are ranked according to parameters selected by the user, the candidates are presented to the user for acceptance. The presentation is a crucial and non-trivial, it must be so that the validation of the user as quickly as possible. In each iteration, the user selects and combines different analysis to find areas that could not be found by previous studies.

Several tests may be selected and applied in parallel. Then the intersection, union, and the analysis of these differences can be established automatically and the user can review and validate them.

Rigi in the thesis is used for the presentation and user interaction (Müller, 1994). Rigi is a customizable graph editor developed for reverse engineering and offers many features.

The analysis can be selected from Rigi through lists, menus and easily done with simple mouse clicks, so that the user does not need to learn a complex language. The result of the analysis is represented by a single node of the hierarchical analysis that is the root of the number of candidates.

Data are used to evaluate and rank the candidates who were proposed by the analysis. The metric used for classification is a composite of normal during the weighted sum of various parameters.

The composite metric is used to guide the user through the large number of candidates. The user can navigate by clicking on the candidates on the nodes or view the entire hierarchy of the node as a whole. In this way, the user can see all the candidates at once and still immediately find promising candidates.

Candidates may be accepted or refused, returned, either individually or as a whole by direct manipulation. The atomic elements can be renamed by the user to give them a meaningful name.

The strategy for recovery of components consists of two main parts: the detection of atomic elements and identify relationships between components of these elements. In the first phase of the above method, various techniques are combined using the combination of the operators described above.

Once the components of atoms detected, their relationships can be analyzed by the application of component analysis. The recovered components are documented by the responsible and saved for future maintenance. They can be used to explain the system to a higher level of abstraction. Ideally, the module decomposition of the system will be restructured to conform to the atomic structure of the component, ie, a module contains exactly one atomic element. If entities are extracted from a component, the technique of voting may be used to determine whether the system should be restructured.

6.1.5 Conclusion

Beyond the structure-based techniques, there are other automated techniques based on data flow information and domain information. The semi-automatic methods include the user in the process. Most semi-automatic techniques are bottom-up approaches that begin at the global level down to the code.

The evaluation revealed the following :

- The effectiveness of a technique depends on the characteristics of the system, such as hidden information, the decomposition into modules, etc.
- None of the investigative techniques at a rate sufficient to recall, the highest point, we obtained was 75% of abstract data objects. In the worst case, the best technique is only a reminder of 34%.

Extended the technique called "Similarity Clustering" can detect all kinds of atomic elements. It can be used to find the styles defined by the user and search for models of components already found. The settings of "Similarity Clustering" provides great flexibility. Another way to combine technology was introduced as the vote in which the agreement of each technique is probed, weighted, and all these data are summarized in total agreement on whether a group is a promising candidate. Analysis, selected by the user, are used to nominate candidates who are then validated by the user. The information added by the user is used by the techniques of the next iteration.

The recovery of the information architecture from source code is not only necessary for the understanding of the system. It is also necessary to validate the architectural specifications, unless the code is generated automatically from the specification and generation itself is reliable. A software architect may specify certain aspects like the structure of the system (atomic elements, subsystems, hidden etc), protocols, components, or configurations, such as design patterns. To validate these specifications, it is necessary to recover the architecture as construction and compare it with the specification. Thus, three major aspects must be addressed by research in the architecture of compliance :

- Specifications :

What is to be specified in a software architecture?

What are the methods, notations and tools to specify a software architecture?

What type of analysis are supported by these methods and notations?

- Recovery :

How can we retrieve information architecture that needs to be validated?

- Validation :

How can architecture be built to verify compliance with the specification?

This thesis describes the architecture recovery of components by the assessment, improvement, and the combination of techniques in an interactive and progressive process. The information collected by the methods and techniques described in this thesis are useful for understanding the program, reverse engineering and reengineering techniques, and the validation software.

6.2 References

- [0] – http://fr.wikipedia.org/wiki/Unified_Modeling_Language
- [1] – <http://www.cs.cmu.edu/~acme/>
- [2] – http://fr.wikipedia.org/wiki/Enterprise_JavaBeans
- [3] – <http://www.ibm.com/developerworks/webservices/library/co-cjct6/>
- [4] – *OSGi Service Platform Core Specification release 4*, August 2005, The OSGi Alliance
- [5] – <http://www.sciences.univ-nantes.fr/info/perso/permanents/attiogbe/COLOSS/>
- [6] – <http://www.eclipse.org/modeling/emf/>
- [7] – <http://www.graphviz.org/>
- [8] – <http://prefuse.org/>
- [9] – Eclipse Home Page : <http://www.eclipse.org/>
- [10] – David Boxer, Ashutosh Galande, Thuc Si Mau Ho of University of Illinois at Urbana-Champaign : JDT Architecture, Publication 2004 41p.
- [11] – Rainer Koschke : Atomic Architectural Component Recovery for Program Understanding and Evolution - Evaluation of Automatic Re-Modularization Techniques and Their Integration in a Semi-Automatic Method