# Econet project
# -
# Rules-based system

Analysis of the existing system and its properties

Aurélia COUVRAND

March 12, 2009

# Contents

# Chapter 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 section, we will explain each rule allowing the extraction of a components structure. We will finish up defining the properties the system has to check in a second section.

This version is not the final one, the last section will be padded by Mathieu VÉNISSE.

# Chapter 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, whiwh 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 center our study, so we are able to propose some rules to select the potential applicants for the role of component. Here are these rules.

## 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.

## 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.

### 2.2.1 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.

### 2.2.2 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.

### 2.2.3 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.

### 2.2.4 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.

### 2.2.5 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.

### 2.2.6 Enumerations and implementation of data structures

Enumerations and classes implementing data structures are data types.

### 2.2.7 Public attributes

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

## 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.

## 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.

## 2.5 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()*.

# Chapter 3

# 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.

## 3.1 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 centrer 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.

## 3.2 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.

## 3.3 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.

# Chapter 4

# 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*.

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.