Notes et règles d'extraction

Jean-Claude Royer

today

1 Préambule

Il s'agit de quelques notes générales explicatives du principe et processus. D'autres informations sur l'implémentation seront bientôt disponibles. Une chose importante est que les règles peuvent évoluer, il faut donc proposer une architecture très modulaire fournissant un bon ensemble de services généraux pour ce genre de problème.

2 Architecture extraction

For re-engineering Java applications into component assemblies several questions have to be raised. To summarise the main questions:

- We need to identify or extract the boundaries of components, for primitive and composite, this result in a tree structure describing the architecture. Primitive component are leaves of the tree structure extracted from the components set.
- Communications are, usually in such applications, reduced to binary communications and we classify services into required and provided services. The second task is to extract the communications between the different components in the architecture.
- Last we need the interfaces of each component, here we can consider only two interfaces (required and provided) which are sets of methods.

We consider plain Java code thus generally it is not possible to simply extract a component architecture from it. We choose to propose some simples rules allowing to get a minimal architecture which can be refined or improved manually or automatically with more re-engineering effort. Hence, we propose rules to extract related component information from the code and some controls which can help in detecting potential problems or non-expected situations. Running the extracting plugin will produce various information which is detailed below. A given information is extracted using a rule and a set of checking is provided to indicate more or less serious problems in the extracted information. This information is used to extract the protocols later on or it could also be used by an architect to re-engineer the system.

However, we add some hypotheses on the Java code. We consider Java source code, binary Java code could be analysed using more or less the same way, or first decompiling and then analysing the result with our tool. We consider that the code to analyse is contained in a unique project (an Eclipse project) which can import predefined project or libraries. We assume that in a Java file, what JDT called a compilation unit, there is only one main type which could be a component type or a data type.

This constraint is simple to relax, but we don't want to analyse code which is too badly structured. We don't consider generic types, this is a main future extension of this work. To relax this restriction is not easy since Java 1.5 introduces a sophisticated type system and we have to distinguish, generic definition, instantiation of them, their use in fields, inheritance and methods. A component type in Java could be an interface, a concrete or an abstract class. However, a component type must be instantiated and this is only possible with concrete classes. We also do not analyse static methods since until now we do not get a precise feeling of their use regarding a component approach. Nevertheless, the component extracting task remains a challenge.

The first rule we consider is that a component cannot be passed as an effective parameter of a method but it is possible for constructors. The main reason is: we expect to respect encapsulation or communication integrity [2]. This is an important property since it implies the existence of some communication channels which can be identified with a static code analysis. Thus we analyse a Java project and extract the main types defined in it and forgetting external data types or primitives ones. These types are called the *types of interest*. Then a type of interest which is used as parameter type in a method is flagged as a data type. Furthermore the subclasses of this type are also flagged as data types since their instances could be used as effective parameters using subtyping rules.

The second rule is to extract a possible composite structure from the analysis of the fields. Ideally the composite structure (or part-of relationship) should be a tree or a forest. The analysis simply browses the candidate types and collects recursively their structure. However the process has to collect the inherited structure from the super classes, that means to collect private, public, protected and default-package fields in the inheritance up to `Object`. We collect all kinds of fields since, even if a field is not accessible in a class, a public accessor can be defined to read or write it. One possible problem here is to get a structure which is not a tree, there may be some cycles, in this case we have not a true component architecture. However this cyclic structure could be used by another more elaborated process to propose, after some changes, a compliant component architecture. The `checkCycle` function is responsible for providing this information and the `checkPublicFields` informs about public fields which are also a way to break component encapsulation.

A more specific case of this rule concerns the analysis of the main entry points. There may be several, usually devoted to testing activities. However they often provide an instantiation of an object structure without defining explicitly a class for this structure. Our assumption here is: we consider that there is no such point or equivalently the structure is defined by the class fields. The other way would be to analyse constructor calls in the main entry points and this implies to analyse the constructor calls of all the class. This analysis is a bit more difficult but the main problem is that it provides several ways to create a component not the maximal component structure.

The third rule is to extract communications from the code of methods. We consider that there is a communication from type `E` to type `R` with message `mesg`, if and only if `mesg` occurs in the implementation of `E` and its local sender type `R` provides the `mesg` service. Here `mesg` is a signature in the Java sense.

The fourth rule is twofolds since it computes the required and provided services of each type. For the required services: They comes directly from the previous analysis of communications. For the provided services: An analysis of each type is done to extract the public or default package methods of the type. An additional checking (`checkRequiredProvided`) has to be done here to check if a required services of a component type is effectively a provided service of the right component type. This is a strict requirement, the opposite checking would be wrong, since a provided service may not necessarily be connected in a given architecture.

The `checkStructureCommunications` checking has the role to identify the communications which are not conform to the structure of the components. In a strict component framework components communicate directly if they are in the same scope or level of the composite structure. But in a plain Java code we could find three exclusive situations: i) there is no path in the composite structure between the emitter and the receiver, or ii) there is a common ancestor for both instances, or iii) the communication is between two components occurring at the same level of the structure. The first case is a violation of encapsulation and not compliant with a composite architecture. In the second case we can easily restructure the communication into a component compliant way. The third case is the best one.

The implementation of our system relies on the JDT [1] plugin included in the Eclipse framework.

3 Principe des règles

```
%%%% 22/09/2008 => depose sur le wiki econet
About the extracting process

We want to analyse Plain Java code and try to extract
a set of structural information related to a component based development.

First: there is no strict rule but some heuristics
(try to see about uncertainty and other statistical computations).

Second: we have to consider some light and general component hypothesis,
try to extract something without such hypothesis make no sense.

Third: an experimental validation has to be done since it is no obvious to
determinate the right set and the correct ordering of the rules.

We consider general hypothesis:
- static architecture only and created at initialisation time
- respect communication integrity (or encapsulation)
- no component factory
- no special implementation pattern (EJB, for instance)
- more ?

We have discuss some general steps composed of some more specific rules.
I try to expose these first, and a set of complementary rules or ideas we can
use.

%---------------
P1: Elimination of implementation classes
Collect all the classes and interfaces inside a set of packages given as input.
This process must ignore the external classes, Java API,
other framework and inner classes.
It will produce a set of elements called the TYPES OF INTEREST.
```

```
DANS TESTJDT2: ASTActionDelegate.getTypesOfInterest()
recupere les compilations units

%---------------
P2: Identification of data types
The goal is to mark some of the types of interest
as data types in the application.
We consider several rules, the first one seems mandatory.

R1: Identify parameter types in methods
(not constructors or static methods) these are data types.
The reason of this rule is that it is a usual practise
which avoids breaking encapsulation or violating communication integrity.
The component types are not resulting types of methods
since we don't consider component factory.

R2: Identify parameter types in static methods (and ?)

R3: Getter and setters implies a data type class
(? not sure, may be useful to add a binding)

R4: Class with attributes referring only implementation
 classes and data types are data types (It needs to analyse
the communications network to be sure of that)

R5: We can consider as component type a class
  which implements interfaces and includes fields that target interfaces

R6: Enumerations are data types,
or classes implementing data structures (describe the patterns)

R7: A class ublic data field is not a component type

%---------------
P3: Composite analysis
The composite structure should be found by
the analysis of the instantiation of component types,
starting from the main program and following
recursively the constructors calls of the component types.
Not sufficient ? there are some data types may be here ...

%---------------
P4: Look at the communications
From the code of methods (constructors, static ones ?)
we will identify the message sending pattern r.send(args):
```

```
it occurs in the context of type T, with r:R
and args are data types values.
 If T and R are components types we add a link from T to R.
Note that we consider a set of component types here.

%---------------
P5: Find the interfaces
    Assuming we have the component type names
the problem is to compute the interface,
for each component type name:
- This is an interface
        - This is a class which implements several interfaces
  - This is a class which inherits from a class
inside the context and see case 2
  - This is a class which inherits from an external class ...
- This is a class, extract its public methods
(several cases here with inner classes, inheritance, ...)
- (not sure here I we need something else)
We need the keywords ‘‘public’’ somewhere.

%---------------
Some other rules or ideas

SOR1: In an implementation of a component type the attributes could be link,
to be sure of that it must exist communications related to this link.

SOR2: Analyse the package structure (I am not sure that we can extract
something useful from there since
the logic behind Java package and composite structure is quite different).
```

References

[1] *Java Development Tooling*, 2008. `http://www.eclipse.org/jdt/`.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 187–197. ACM Press, 2002.