Notes et règles d'extraction

Jean-Claude Royer

today

TODO: **PBA: la creation/instanciation est particuliere, verification controle ?, pour l'instant ni discute ni implemente**

1 Architecture extraction

1.1 Principles

In summary: we target a component model with some general principles but not too specific features related to a given component language. The idea is to identify entities that could be component and the flow of communications between these entities. This analysis could be performed on a general object-oriented application, without specific patterns or other linguistic construction, and a designer get a map of the application and the important information to restructure it in a better component way. Component programming promotes more strict rules than in classic object-oriented programming. In such a programming model we use usual data types and component types. The first rule is: we focus on first class component instances with explicit component types. Thus for instance, using the package structure to implement a component application, while it could be used, is out of the scope of this model. As an example, in the CoCoME reference implementation of the component instances is done using the Java packages hierarchy. This architecture could be found analysing the imported packages. The second rule is about communication integrity [3]. The reason is: in component programming the communication channels have to be explicit and there is no other way to communicate between component instances. The benefit is to get consistent the static architecture of communication and the implementation. Two components communicate only *via* an existing communication chanel. The third rule is about communication means, there are various ways to implement them ranging from simple method call to complex connector mechanisms. We consider that a communication is a method call occurring in the context of the emitter code and with an object as receiver. Lastly, many component models consider that the structure (or the part-of relationship) is a tree (or more generally a forest). However this hypothesis is not often explicit and the reasons to used it are not clear. The main reason is to get a well-founded recursive data structure for components. From that, algorithms can be easily defined, for instance graphical views of the architecture has nested boxes with communication links. In our abstract model we consider also a subtype relation between types, the semantic of subtyping is based on the substitution principles [6]. It means that if `R` is a subtype of `T`, whenever a `T` is used an `R` can be used in place.

Note that some of the properties we want to use, like communication integrity and boundary analysis are unsolvable. Our approach relies on a static analysis and use type information to proceed. Thus it often only provides necessary conditions which are not sufficient to eliminate some wrong situations.

These principles are useful to measure the degree of "componentization" of a source code. If we consider an architecture as a graph with as nodes the types and as edges the relations, the general properties of the graph are:

- It is bipartite with component types and data types.
- The subgraph of the component type structure is a directed acyclyc graph (DAG).
- Subtyping relations define a DAG, and a data type has only sub data types
- There are communication channels between component types: At each level of the structure, a root component with some direct sub components define a composite component. Communications exists between the sub components and possibility with the root component. But there is no communication involving a sub component and another component in another composite. This is what usually is called the boundary of composite components. The root component represents the frontier and meadiates bewteen the outside and the inside of the composite.

1.2 About Communication Integrity

Communication integrity [5] is a key property which was implemented in ArchJava. It ensures that the component implementation does not communicate in ways that could violate reasoning about control flow in the architecture. *Intuitively, communication integrity in ArchJava means that a component instance A may not call the methods of another component instance B unless B is A's subcomponent, or A and B are sibling subcomponents of a common component instance that declares a connection or connection pattern between them [3].* ArchJava enforces communication integrity in the cases of direct method calls and method calls through ports using both static (compile-time) and dynamic (runtime) checks. It also places restrictions on the ways the components are used. In particular, subcomponents are not allowed to escape the scope of their parent component. In the present work we are interested but the following rule: a component type cannot be passed has a method argument or return as a result. In case of violation the component implementing the method could capture the component reference and then build a communication links not existing in the static architecture. Using this programming style has some benefits, mainly it ensures consistency between the architecture and the implementation of the application. But it also introduce some constraints, for instance to dynamically configure components [7].

1.3 Hypothesis

Extracting an architecture is a static analysis of the code, it provides useful information but sometimes not so accurate than runtime analysis. See related work for a more detailed comparison between the different approaches.

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 DAG structure describing the architecture. Primitive component are leaves of the 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.

- We need the interfaces of each component, here we can consider only two interfaces (required and provided) which are sets of methods.
- Last, we analyse the messages between components to check that the boundaries are preserved.

We consider plain Java code thus generally it is not possible to simply extract a component architecture from it. The approach proposed here is also applicable to other languages, mainly object-oriented languages. However there is no standard way to consider interfaces and their relationships with classes thus it could be a non trivial adaptation of our current work.

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. TODO: **pas clair si externe DATA TYPE =... sinon ?**

One first point is that Java has four levels of data access `private`, `protected`, and `public` but also default package without keyword. The second point is the complex imbrication of classes in Java. In a Java file, what JDT called a compilation unit, there may be several types defined in it. Usually one main public type and possibly several default-package types. In addition Java has four notions of nested classes, *static member*, *inner*, *local* and *anonymous*. In all these cases TODO: **Que dire!!!, implementation issues but it could be used to define public service of one component**
. 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 discard code using the Java reflective API since we can dynamically program complex things without obeying to any static contract (typing, communications, etc). Nevertheless, the component extracting task remains a challenge. Another thing is the existence of interfaces in Java, an interface may have several sub interfaces and can implemented by several classes. On the other hand a class has only several subclasses. Both relations, `extends` and `implements`, have a subtyping semantic and do not allow cycles. The main difference between interfaces and classes is that the first ones do not really participate in the concrete realisation of instances, they are only used to type expressions and to define constants. Classes define the instantiation structure, and the reader may be aware of the fact that even abstract classes can define a concrete structure. The difference between both abstract and concrete classes is that the first ones cannot be directly instantiated but indirectly using the `super` constructor.

1.4 Rules Details

To start our analysis the first task is to get all the compilation unit in the project and to extract the type names. The enumeration, primitive types, and external types are discarded from this list considering either they cannot be component type or we only reused it as a black box. Thus we analyse a Java project and extract the main types defined in it, these types are called the *types of interest*.

1.4.1 First Rule.

The first rule we consider is that a component type cannot be passed as an effective parameter or return as a result of a method but it is possible for constructors. The main reason is: we expect to respect encapsulation or communication integrity. This is an important property since it implies the existence of some communication channels which can be identified with a static code analysis. Then a type of interest which is used as parameter type or as a resulting type in a non constructor method is flagged as a data type. Here we use a propagation mechanism to mark also subclasses and subinterfaces since this property can be transmitted by subtyping: Let a method `msg : D -> R`, then `D` is a data type and every subtypes of it are data types. It implies that:

- If `D` is a class all its subclasses are also flagged as data types since their instances could be used as effective parameters using the subtyping principle.
- If `D` is an interface all its subclasses and its sub interfaces are flagged.

This rule has some important consequences, as example, a public writer like `setField(Type t)` implies that `Type` is considered as a data type since one of its references could be captured. Regarding elimination of the resulting types from component types, let be `aFct()` returning a component instance then we can build expressions like: `cmp.aFct().aService();`. This defines an implicit possibly new communication channel. It shows that this rule is stronger than the classic encapsulation principle and prohibit direct or indirect setters.

The `checkPublicFields` informs about the existence of public and default-package fields which are an obvious way to break component encapsulation.

1.4.2 Second Rule.

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 DAG. 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 public, protected and default-package fields in the inheritance up to `Object`. Private fields are not really inherited since they cannot be write in the subclasses.

The possible problem here is to get a structure which is not a DAG, 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 the cycle information. Thus the absence of cyclic structure is a sufficient condition to ensure that no runtime cycle can be build between component instances.

We choose to consider the maximal structure, that is collecting the defined attributes and the inherited ones. However in a non perfect world, or in usual Java programs, the real situation may need a less strict rule. This is the role of the 6th rule.

At this step, classes are either data types or component type. The interfaces which are not tagged has data type are then tagged as component type. After that every type of interest is tagged either as a data type or as a component type.

1.4.3 Third Rule.

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. Note that we collect method calls as well as static function calls and static final attribute references. In this process we collect all messages between the types of interest, even those concerning data types. This is important in a re-engineering perspective or to detect connectors implemented by data types. Note also that a message can be one way (resulting type is `void`) or one way with return (resulting type is a data type).

1.4.4 Fourth Rule.

The fourth rule is twofolds since it computes the required and provided services of each type. For the required services: They come 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. This analysis also collects the static functions and the static final attributes as soon as they have the public or default-package modifiers. 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. This checking must cope with the subtyping relation but it is rather straight.

1.5 Fifth Rule.

The `boundaryAnalysis` 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 boundary 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. TODO: **probleme heritage ici**

1.6 The Sixth Rule.

A specific situation 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. Another case is that a class can have several different constructors. Thus the other way would be to analyse all the constructor calls and the main entry points of all the class. TODO: **A completer, je devrais pouvoir le faire, mais comment comparer avec structure maximal.**

1.7 Implementation

Our work has been conducted under the Eclipse environment and Java 1.5. The implementation of our system relies on the JDT [1] plugin included in the Eclipse framework. The JDT provides a rich set of functionalities to enable Java developers to annotate programs, to analyse Java code or to extend the compiler or the debugger. It is a set of plugins and the one needed for our analysis is JDT core which defines the Java model for source and binary code. It also provides tools to extract information from Java projects and to parse Java code which are the basic functionalities our plugin required. The sub-package `dom` define a general AST (Abstract Syntax Tree) parser and some facilities like a general AST visitor class.

The plugin produces textual results but there are also two graphical views. There are interactive and graphical views which allow to browse the components, their structure and their communications. It is possible to search for some component type names or to only display some set of relations. The implementation of this plugin relies on the prefuse [4] library. Two layouts are provided: a force and a radial one. TODO: **We could also put one figure and comment the result.**

2 Examples

2.1 ArchJava Analysis

ArchJava [3, 2] is a small, backwards-compatible extension to Java that integrates software architecture specifications into Java implementation code. It extends a practical implementation language to incorporate architectural features and enforce communication integrity. In ArchJava, a *component* is a special kind of object capable of communicating with other components in a structured way. The communication is performed using logical communication channels called *ports*. Each port is allowed to declare methods qualified by the keywords `requires` and `provides`. Only the provided methods have to be implemented in a component. The hierarchical software architecture is expressed with *composite components* made of connected subcomponents. To connect two or more ports in an architecture, the `connect` primitive is employed. This primitive binds each required method in a port to a provided method with the same signature in other ports.

We analyze the various small examples available on the web site of ArchJava. Note that ArchJava defines a pre-processor thus we analyze the Java source code after the Java code generation. As expected this code contains the components but also the port definitions, and no cyclic structure in the composition. However it only catches some communications from the composite to the port. The reason is that communications between ports are not done by a method call but using the Java reflective API. Thus our communication analysis cannot find them.

References

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

[2] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 334–367. Springer Verlag, 2002.

[3] 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.

[4] J. Heer. The prefuse visualization toolkit. http://prefuse.org/.

[5] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[6] J. Paslberg and M. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[7] S. Pavel, J. Noyé, and J.-C. Royer. Dynamic configuration of software product lines in archjava. In Robert L. Nord, editor, *Software Product Lines: Third International Conference*, number 3154 in Lecture Notes in Computer Science, pages 90–109, Boston, MA, USA, September 2004. Springer-Verlag.