Short Draft About The Java Component Extractor

Jean-Claude Royer
ASCOLA, Mines de Nantes - INRIA
Nantes, France
Jean-Claude.Royer@mines-nantes.fr

June 11, 2010

Abstract

This short draft gives the rules used by the Java Component Extractor to qualify component types in Java source code.

1 Extraction Rules

The JCE (for Java Component Extractor) is an Eclipse plugin developed in the ASCOLA team during a collaborating project [4]. The underlying component model relies on the following assumptions: *i)* component types are true types which can be instantiated to generate components, *ii)* they communicate via a strict message passing policy, *iii)* they can be either concrete or abstract component types *iv)* they support subtyping, and *v)* composites are built from a structure containing subcomponents. A subcomponent is a component enclosed in a composite component. Since the approach relies on static analysis of source code we extract component types rather than component instances. The extracting process is based on a semantic property, the so-called communication integrity property [7, 2]. Basically, a component type ($CTypes$) must respect this property, if not it is qualified as a data type ($DTypes$). A set of rules, see below, qualify the types as either a component or exclusively a data type. Following the work on ArchJava [2, 3, 1], the tool considers that a component can neither be passed as a reference in a method call, nor be part of a data type, nor occurred in a generic construction, nor be an exception class, nor contains an inner data type. These are preliminary properties to ensure that the flow of communication is compliant with the architecture, that is, there is no hidden communication channel. The JCE tool analyze the main Java elements: classes (abstract or not), interfaces, inner classes, subtyping and composition relationships, and method signatures. We consider generic classes as ordinary classes, this is only a partial check. However, parametrization are fully analyzed and lead to some specific rules.

1.1 The Main Rules

The rules from 1-a) to 1-d) are general rules coming from the component model. Rules 1-e) to 1-h) address specific problems of generics, inner classes, exceptions and external types. Rules 2 to 5 are devoted to, respectively, structure, communications, interfaces and subtyping.

Components:

1-a) This first rule searches the wrong profiles in the application. If a type of interest is passed as parameter of a method or returned by a method it is considered a data type, otherwise it is considered a component type. The idea is that static checking of communication integrity is possible only when all uses of a component are explicit (as opposed to accessing the component through a pointer). This is a standard assumption in CBSE *e.g.* [2], [6].

1-b) Exception to rule 1-a: a component may be passed to or returned by constructor. This is often necessary to allow building composite components, and as explained in Section **??** this is consistent with modern framework principles which distinguish: creating, initializing and using a component.

1-c) A type occurring in the parts of a $DTypes$ is a $DTypes$. This follows from the prohibition of encapsulating a component reference into a value. It is not difficult to see that the value can be captured and a method of the data type can indirectly send a message to the internal component.

1-d) A subtype of a $DTypes$ is considered a data type. This follows from rule 1-a, since instances of the subtype could be used as parameters using subtyping rules.

1-e) Arrays and generics constructions add several complications[1]. Using arrays or generics instantiation as formal parameters in signatures or in field declarations of data types would allow to indirectly access the stored references. Thus, we should consider the effective parameters of arrays and generics as data types. There is also another specific case for the use of a generic instantiation as a superclass or a super interface. In this case the formal parameter should be flag as a $DTypes$ but also the subclasses and implementation of the generic instantiation (from rule 1-d).

1-f) To analyze inner classes could be useful. One reason is that they are generally used in GUIs, or to implement some specific features like simulating multiple inheritance. There are four kinds of inner classes, static inner class, on one side, and (member, local or anonymous) inner class, on the other side. If the inner class is a $DTypes$, one of its instances could escape from its context and could allow access to the private enclosing component structure or to the enclosing component reference itself. In this case, in addition to other rules[2] about wrong profiles, composition and subtyping, the enclosing class should be declared a $DTypes$.

[1] ArchJava considers array constructions but it says nothing about the use of generics.

[2] For technical reasons, local and anonymous classes are not yet caught by our prototype.

1-g) Exceptions can be viewed as data structures enriched with a `throw/try-catch` mechanism. They should be checked as for ordinary types but in addition an exception type occurring in a `catch` clause is a $DTypes$. A strict and pragmatic rule[3] is to consider exceptions as always data types.

1-h) "External" types of interest ($ETypes$, not defined in the Java project) are ignored. We chose to ignore all types of interest not defined in the Java project (*e.g.*, ignore all external libraries as `java.io.*`, or `org.eclipse.*`). One reason is that we want to extract the provided services of the components, and their structure. This requires having access to the source code (the Java reflective API could help, but we favor a more generic solution). Also, there are good chances that `Object` will be passed as parameter of, or returned by, some method, turning it into a data type (rule 1-a). This would, in turn, qualify all types of interest as data types (rule 1-c). Nevertheless, it is convenient to extend some external data types. Finally, we cannot hope to restructure the entire world and would like to limit ourselves to the application at hand. Thus we consider that the external world do not introduce communication integrity problems. This is surely wrong and the designer has the responsibility to ensure this. One first, but restricted way, is to check for downcast from an external data type to a component type, (see Section 1.2.5 for more details). To provide more precise checking in this case is still an open problem.

Compositions:

2-a) The composition structure of components is extracted from the fields (component that is part of another component). We choose to consider the maximal structure, that is collecting all the defined attributes and the inherited ones, but not the inherited private fields. We also consider only the $CTypes$, array and generics of $CTypes$. This is another point to focus on the important information from a component point of view.

Communications:

3-a) There is a communication between two component types if a method of one component type makes a call to a method of the other. If the method "returns" `void` it is a one way communication, otherwise, it is a two ways if it returns a data type. There are two kinds of communication: *i)* the communication is between a component and one of its subcomponents or *ii)* it is between two sibling subcomponents of the same enclosing component. These are the two legal cases for communications, all other situations should be considered as ill-formed. Other communications techniques than direct call are possible. For example in Java, using the reflective API. We did not consider these cases here as they require more advanced knowledge of an application to know how communications are implemented in it. It also enables more dynamism and we focus on static information only.

Required/provided services:

[3]This simple rule is not yet implemented in the current prototype.

4-a) Provided services are all the publicly available methods defined in the component type. In Java, these methods are the `public`, `protected` and `default package` ones.

4-b) The required services of a component type are those methods that are called in the component type.

The application is assumed to be typed-checked, there is no need to check the correspondence between required and provided services.

Subtyping:

5-a) Subtyping relationships are computed from the language inheritance relationships. In Java, there are two such relationships: `extends` and `implements`. Component types may inherit from component types but not data types (see also rule 1-c). A data type may inherit from data types or component types.

As explained in the formal model we should also check subtyping relations for structures, provided and required services. We discuss the issue of checking this constraint in Section 1.2.4.

1.2 Complementary Information

The extraction analysis provides additional information to warn the architects or the designers to more specific problems or additional constraints. This pertinent information helps in analyzing the extraction result and detecting potential problems or bad practice. This section discusses these potential problems.

1.2.1 Direct Accesses

The programmer should use messages with the adequate method to ensure true encapsulation and do not access to a field in a wrong direct style. More precisely, the rule detects the `expression.field` where a public (or default-package or protected) `field` is used in a default-package manner. Although, an access through `this` or `super` is obviously considered legal.

1.2.2 Component Composition Cycles

A component composition cycle is a cyclic dependency between the structure of the component types. The main reason to prohibit such cycles is that it increases the coupling between the elements of the cycle. In CBSE the structure of components is a tree or a forest, that is no component sharing[4] and no cycle. One reason behind that is that often users expect to display their component structures as nested boxes. A cycle in a component structure implies a cycle in the corresponding type structure, but the opposite is not true since relation between components and component types is many to one. The control looks for cycles in the component types structure to warn the designer

[4]Fractal [5] is one component language allowing sharing of component, we think this an exception rather than a well-accepted practice.

about the risk of cycles. This could be restrictive in case of implementing bidirectional communication between enclosing and subcomponents with only direct calls.

1.2.3 Boundary Violations

The *boundary analysis* identifies the communications which are not conform to the structure of the component types. In a strict component framework, components may communicate directly if they are in the same scope or boundary of a composite structure. A component may communicate with its composite component or with sibling components (see Section **??**). But the Java code may not always respect these well-formedness constraints. By identifying communication channels and structural relationships between component types, we are able to pinpoint some anomalies. We herein provide a lightweight method coping with the fact that we use type information instead of runtime instance and we do not want to stress too much the designers with too many problems. More precisely we distinguish four degrees of boundary problems. Let consider that in the context of component type `S'`, there is a communication to component type `S`, *i.e.* one method of `S'` contains a message `(exp:S).mesg(someparameters)`.

- If `S == S'` we consider it as a correct message.
- If `S` is the type occurring in the structure of `S'` then if `exp == NOFIELD` this problem is raised. This case must also look for "inherited" component fields.
- If `S'` occurs in the structure of `S` then a `REVERSE` problem is raised.
- If it exists a type `C` with `S` and `S'` as sibling types then a `SIBLING` problem is raised.
- Else, if the sender type is an enclosing type, it means that `S'` is trying to send a message to `S` which is out of its context then an `EXTERN` problem is triggered.

These four cases correspond to various seriousness boundary violations or bad practices and are resolved using different solutions. The EXTERN problem refers to a possible communication between two component instances not in the same boundary. A `NOFIELD` problem denotes a message to a non subcomponent. The `REVERSE` problems means a communication from a subcomponent type to the enclosing type. The `SIBLING` covers various problems which can be solved in various ways: move the communication in the enclosing component, dependency injection, using ports, or using composition.

1.2.4 Subtyping Violations

The rule for subtyping 5-a) should check more about structures and required services. Since these controls are either constraining or could be considered as non relevant they are separated from the main rules. The subtyping structure principle is rather more liberal in Java than our structural co-variance. In the case of provided services, since our legacy code is assumed to be compiled thus safely type-checked, the subtyping relation hold. However, this is not the case with required services, which are not explicit

in a usual legacy application. The rule will give information about the violations of the structural subtyping and of the contra-variant rule for required services.

1.2.5 Suspect Downcasts

The communication integrity property is lost when inheriting from an external data types. The rule 1-a) detects the methods defined or redefined with a wrong profile against the component type. In other words, a component type can add or redefined some methods but it cannot do it with wrong signatures. A violation of the communication integrity is still possible. One case is casting the component to the super data type and later by downcasting it into the component type. A *suspect downcast* is a cast expression `(T) exp` where `exp` if of type an external type (ETypes) and `T` is a CTypes. Thus we think convenient to check the places where a suspect downcast is done. The rule locates the expressions where there is a downcast from an external type to a component type. These expressions open the risk of a component reference capture but are not always infraction to communication integrity.

1.2.6 Shared Types

CBSE prohibits component sharing but allow to share values. We propose an analysis to detect the data types which are shared between at least two component types. The result can be viewed as a warning. A deeper alias analysis will find the real problems (if exists). The aliases checking can help in disambiguating some boundary problems of Section 1.2.3 but also to detect hidden channels resulting from the sharing of a data value. We do not proceed to this advanced analysis since we want to get first a correct static architecture. Furthermore such kind of analysis could be done by some existing approaches like AliasJava [1].

1.3 The Extracting Process

Theses rules are implemented by procedures which exploit information provided by the abstract syntax tree built from the source code. By default all the types are initially $CTypes$. The rules which are responsible to flag types as $DTypes$ are run first (1-a, 1-b, 1-e, 1-g, and 1-h). Then the propagation rules (1-c, 1-d, and 1-f) are applied to propagate the $DTypes$ along subtyping, composition and inner class. Finally the complementary information is obtained by running some specific procedures.

2 The Java Component Extractor

This section reviews our tool prototype and presents some of the experiences we did with it.

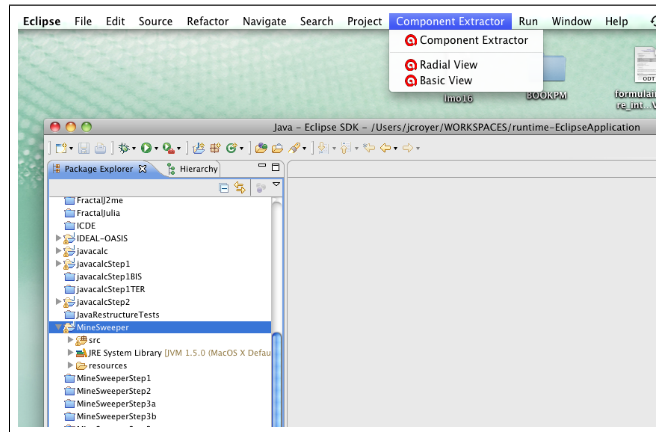

Figure 1: Java Component Extractor Menu.

2.1 The extracting tool

The tool is implemented as an Eclipse plug-in[5] and is available as a new menu in the window tool bar of Eclipse, see Figure 1. The `ComponentExtractor` item computes an instance of the component model by processing the extraction rules of Section 1.1 and performs the complementary analysis. During the component model extraction process, the user must choose where to find the Eclipse project to analyze in the workspace, whether he wants a detailed output or not, and which of the detected types he wants to process[6]. The extraction process produces various kinds of information, either in synthetic and interactive graphical views or in detailed text files. The main output, the component model, is separated from additional information to avoid burden the user with less important data.

There are two graphical views, differing mainly in their layout and also differ by the information they display. The basic one uses a spring layout which gives a "classical" graph view of the component model (see for example Figure Figure 2). The second view uses a radial layout, nodes are arranged on concentric circles around a, user-selected, central node (a type of the component model). The graph may be dynamically configured to show *i)* component types and/or data types, *ii)* structure relationships and/or communication relationships and/or subtyping relationships[7]. For example, the graph of Figure 2 shows the $ITypes$ with the structure and communication relationships. In this example, the dark gray boxes are data types, the light gray boxes are component types, plain arrows go from a composite component type to a subcomponent type, dashed arrows illustrate a communication (*i.e.* one or more method call(s)), from the caller to the callee. The width of the dashed arrow indicates the

[5]Information about the tool is available at
`http://www.emn.fr/x-info/jroyer/JAVACOMPEXT/index.html`

[6]However, in a first run, it is best to accept the default option which considers all types.

[7]Once you choose a view, a set of buttons allows you to select the kind of nodes and links you want to display.

"strength" of the communication link (the number of services called on this communication). A context window on a communication link shows what services are involved in this communication. A context window (lower right window, for component `Grid` in the figure) on a component, shows the services it provides and requires. An wrong profile appears as a provided service, red colored, and with a star before its name.

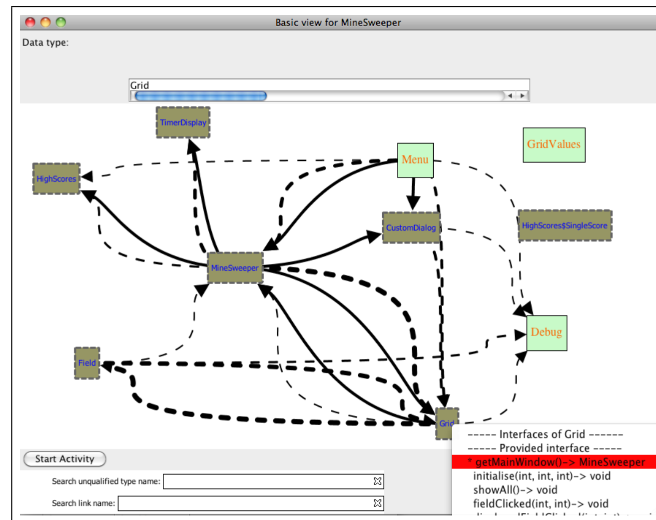

Figure 2: First analysis of MineSweeper.

On complement of the graphical views, detailed textual data is output in four files, suffixed `_BOUNDS.txt`, `_FINAL.txt`, `_REQUIRES.txt`, and `_METRICS.txt`. The `_BOUNDS.txt` file contains the result of the boundary analysis, as explained in Section 1.2.3. The boundary problems are labeled: `NOFIELD`, `REVERSE`, `SIBLING`, and `EXTERN`. The `_FINAL.txt` file is a table of information about all type of interest. It gives their binary Java name, whether they are component type or data types and information about their Java nature (class or interface). It also explicits the relations with other component types: superclasses and composite types. The `_REQUIRES.txt` file provides, for each component type, the subtyping checking information. It gives the result of the structural co-variance and what are the required signature which are not correctly subtyped in the ancestor component types. Finally, the `_METRICS.txt` file provides component type information only and some metrics. It describes each component type with its binary java name, the structure, the provided and the required interfaces. There are also some numeric information about the number of component and data types, the number of classes and interfaces, the number of provided and required services, the number of composition and subtyping relation and the number of communication channels.

2.2 A Simple Example

In this section we demonstrate how to use the information provided by the tool to help restructuring a simple Java project. The goal is to improve the component structure of the project and the tool will help in making this structure explicit. We do not explicit a precise restructuring process but illustrates how to use the information provided by the tool. This example is based on the `MineSweeper` game: “Minesweeper is a single-player computer game. The object of the game is to clear an abstract minefield without detonating a mine. The game has been written for many system platforms in use today” (from Wikipedia) We use the implementation provided by Tim Van den Bulcke[8]. There is nothing special about this example, it has nearly 800 lines of code and 10 classes. The code was not yet compliant to good Java practices, with regards to, for instance, naming conventions, documentation or data encapsulation. The restructured application was tested by playing a party.

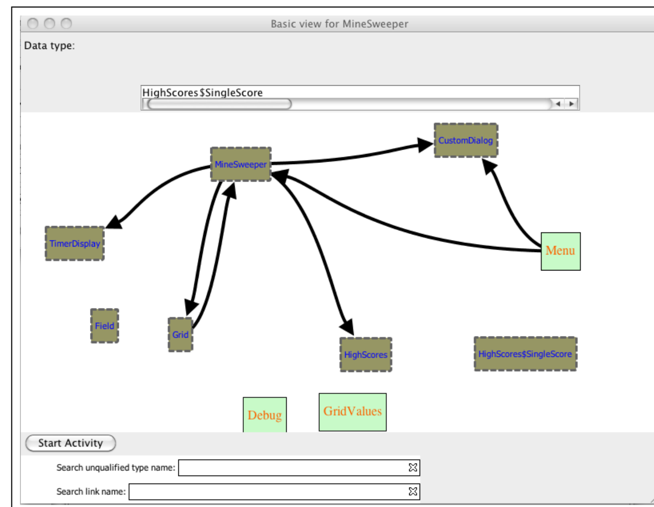

Figure 3: Initial compositions of MineSweeper.

The initial extraction is displayed in Figure 3, configured to show only the composition links. One may first want to look at the wrong signatures, there are two in this case: `Grid/getMainWindow() -> MineSweeper` (see also Figure 2 and `Field/getGrid() -> Grid`. They cause `MineSweeper` and `Grid` to be flagged as $DTypes$. `CustomDialog`, and `TimerDisplay` are $DTypes$ because they are parts of `MineSweeper`. `Field` is a $DTypes$ because it is used as a formal parameter of an array in the `Grid` data type. `HighScores` is a $DTypes$ for two reasons: It is a subcomponent type of `MineSweeper` and it has an inner $DTypes$ (`SimpleScore`).

One could choose to start the restructuring with the `Grid` type, trying first to solve

[8] `http://timvandenbulcke.objectis.net/minesweeper-in-java`

the `Field.getGrid() -> Grid` profile. Formal priority rules could be defined based, for instance, on the number of wrong signatures and the level in the inheritance and composition hierarchies. This is one point where complementary metrics would be useful. `Field.getGrid() -> Grid` method is used in `MineSweeper` by the `mouseClicked` and `showValue` methods. A possible solution is to introduce a field `grid:Grid` in `MineSweeper`, replacing calls to `getGrid()` by accesses to `this.grid`, and setting the value of this field in the constructor. The `Grid/getMainWindow() -> MineSweeper` wrong signature may be solved by defining a new method in the `Grid` class as a relay for communication from `Field` to `MineSweeper`.

After these modifications, a new analysis with the tool gives the new model in Figure 4 (with communication links this time). There are only two data types now and there are no wrong signatures.

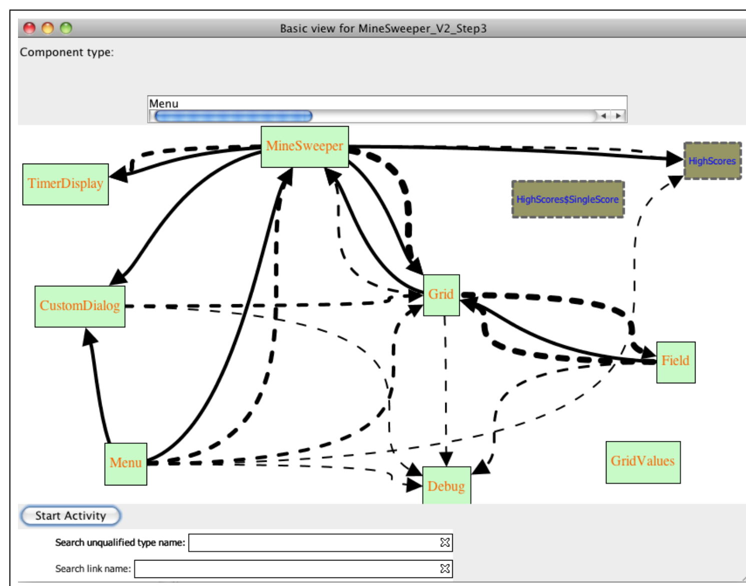

Figure 4: Second analysis of MineSweeper.

The boundary analysis done at this step shows 21 problems: 8 EXTERN and 13 REVERSE. All the EXTERN problems arise from a `Debug` class which is used as a kind of test. Generally programmers define main programs or other tests without encapsulating them into components. A simple refactoring into a component should be possible. All the REVERSE problems are messages send to a reference of a field of type `Field[][]` which is used to represent the grid. In this case a better design without connectors is not possible. There are several downcast localized in the class `HighScore` and related to its inner class `SingleScore`. In this case the use of the composite pattern or a simple parametrization would be better and avoid the need of these casts.

2.2.1 ArchJava Examples

Using the tool we also analyzed several applications from small to middle size. In this section we relate the main points of these experiments. We extracted component information from the examples compiled with the ArchJava preprocessor[9] and which are compliant with the static condition associated to the communication integrity property. We are able to find the component types and the compositions except for one example where most of the types are flagged as $DTypes$. We also found the connectors as predicted by the ArchJava architecture, they appear as component type enclosing both port types associated to the connectors ends. Most of the calls between the component types are made trough the Java reflexive API and are therefore not detected by our tool.

References

[1] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *ECOOP '04 — Object-Oriented Programming European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Oslo, Norway, 2004. Springer-Verlag.

[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] P. André, N. Anquetil, G. Ardourel, J.-C. Royer, P. Hnetynka, T. Poch, D. Petrascu, and V. Petrascu. Javacompext: Extracting architectural elements from java source code. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009), tool demonstration*, pages 377–378, Lille, France, October 2009.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, 36(11-12), 2006.

[6] L. Chouambe, B. Klatt, and K. Krogmann. Reverse engineering software-models of component-based systems. In *CSMR*, pages 93–102. IEEE, 2008.

[7] 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.

[9] http://archjava.fluid.cs.cmu.edu/index.html