Component-Based Java Legacy Code Refactoring

Refactorización Basada en Componentes de Código Java Legado

Hugo Arboleda[1,*], *Andrés Paz*[1], *Jean-Claude Royer*[2]

[1] I2T Group, Universidad Icesi Calle 18 No. 122-135. Cali, Colombia.

[2]ASCOLA Group, Mines de Nantes – INRIA 4 Rue A. Kastler. 44307. Nantes, France.

(Recibido el 19 de octubre de 2012. Aceptado el 5 de agosto de 2013)

Abstract

Component-Based Software Engineering (CBSE) claims to improve software modularisation and to embed architectural concerns. Refactoring Java legacy code with CBSE in mind requires first assessing the compliance of legacy code with component programming principles. This paper presents a portfolio of rules to assess the compliance of Java legacy code with the *Communication Integrity* (CI) property, which is one of the major strengths of the CBSE approach. These rules are proposed with the objective of identifying implicit *component types* and thus provide a measure of the *componentisation* of an application. In order to help developers and legacy code maintainers when refactoring their applications, along with the rules, this work leads to define a set of refactoring actions. Additionally, the results of testing, comparing and analysing the outputs of refactoring several Java applications are also presented.

--------- *Keywords:* Component based programming, communication integrity, Java, refactoring

Resumen

La Ingeniería de Software Basada en Componentes (CBSE) pretende mejorar la modularización del software y la inserción de preocupaciones arquitecturales. Refactorizar código Java legado con CBSE en mente requiere evaluar primero el cumplimiento del código legado con los principios de la programación por componentes. En este artículo presentamos un portafolio de reglas para evaluar el cumplimiento de la propiedad de *Integridad de*

* Autor de correspondencia: teléfono: + 57 + 2 + 55 52 334 ext. 8035, fax: + 57 + 2 + 55 51 745, correo electrónico: hfarboleda@icesi.edu.co (H. Arboleda)

Comunicación en código Java legado; esta propiedad es una de las mayores fortalezas del enfoque CBSE. Proponemos estas reglas para identificar *tipos componente* y así proveer una medida de la construcción de componentes CBSE de una aplicación. Con el objetivo de ayudar a los desarrolladores y al personal responsable del mantenimiento de código legado cuando se hace necesario refactorizar sus aplicaciones, nuestro trabajo nos lleva a definir un conjunto de acciones de refactorización. En este artículo también presentamos resultados de pruebas, comparaciones y análisis de las salidas logradas luego de refactorizar varias aplicaciones Java.

---------- *Palabras clave:* Programación basada en componentes, integridad de comunicación, Java, refactorización

Introduction

Component Based Software Engineering (CBSE) [1] is a software engineering approach concerned with software architecture, modularisation and separation of concerns. The approach promotes the principle of making the architectural decisions explicit; it allows checking of architectural constraints and the use of strict programming principles such as the *Communication Integrity (CI)* property [2, 3]. Such a property states that two components can only communicate if a communication channel has been previously defined between them, *i.e.* there are no hidden communication channels. In software development, architectural erosion is the *"progressive gap observed between the planned and the actual architecture of a software system as implemented by its source code"* [4] and it appears as a side effect when software systems are maintained and the system finally violates the original architectural intents [5-7]. The CI property allows designers to explicitly specify and automatically check some architectural decisions, thus actively limiting the chances of architecture erosion.

This paper considers the problem of refactoring Java legacy code in order to generate component based software applications that satisfy the CI property. As part of previous work a first catalogue of rules to discover *component types* in Java legacy code is presented in [8]; a set of refactoring actions in order to convert, when possible, discovered *data types* into *component types* is presented in [9]. This paper highlights three technical contributions besides the general approach. First, a refined catalogue of rules to detect CI violations according to a light Java component model. Second, an advanced set of refactoring actions in order to solve the CI rule violations. Finally, in order to help developers and legacy code maintainers when refactoring their applications, advanced tool support for automatically identifying *component types*. Two software applications were developed, each one implementing a group of rules; it provides alternatives to discover *components* respecting *fully* or *partially* the CI property.

The remainder of this paper is organised as follows. First, the background for this study and the description of the main elements of the reference model. Then, in Components Qualification the list and explanation of the rules used in the tool to prevent communication integrity violations in Java legacy code. Subsequently, in Component Extraction Results, are presented two rule sets and their applications in several open-source projects. The Refactoring Process presents a list of refactoring actions. Finally, related work and conclusions are presented, including a summary of our contribution and future work.

Background

Reference component model

There are many proposed models that implement the main CBSE principles. The authors focus on models with interface and hierarchical

composition, leaving aside the notion of ports and connectors. In [10] several component definitions are discussed, the present work was based on the definition given in [11]. This definition implies that: *i)* a software component is a unit, *ii)* it specifies an interface (or interfaces) of services it provides, *iii)* it specifies context dependencies, and *iv)* it may be part of a larger composite component. A composite component is built from other components; a component that is not a composite is called a primitive component. As in [12, 13], it is valuable to consider the notion of *subtyping* as a formal way to organise types in the applications.

In the refactoring approach a strict component model with a straightforward implementation in Java is considered. The underlying component model relies on the assumptions that component types *i)* are true types, which means they can be instantiated to generate components, *ii)* communicate via a strict message passing policy based on method calls, *iii)* can be either concrete or abstract component types, *iv)* support subtyping, and *v)* composites are built from a class structure containing subcomponents.

The communication integrity property

In order to illustrate the CI property, consider Listing 1, an excerpt of an application exposed in [14]. In the Primitive class the getIt() method allows access to the otherPart attribute from the outside. Thus, according to the CI property, the OtherPart class cannot be considered as a component type since it can be accessed from outside its enclosing parent. Assuming that Primitive is a component type, the setIt(...) method enables communication between Primitive instances and the otherPart argument, hence it could also violate the CI property. The Composite class has a public field, which is an array of Data instances. Thus, one can access these Data instances from objects holding Composite instances. According to the CI property, the Data class is considered as a data type because it can be accessed from outside its enclosing parent, the Composite. This is also true for the SubData class, which is considered as a data type because of polymorphism. The data part attribute is enclosed in a data type thus it can be indirectly accessed from everywhere in the program and it should also be considered as a data type. In the Composite class the getIt() method is private. If one restricts its uses to this or super then the Primitive instance cannot escape from its enclosing parent. Thus, the Primitive class can be considered as a component type. Similarly, the Composite class is considered as a component type since there is no possible violation of the CI property. As this simple example shows, the effects of these rules are complex and difficult to predict. A tool is required to help designers understand their applications to software projects.

Listing 1 Java Code Sample

```
public class Primitive {
 private OtherPart otherpart;
 public OtherPart getIt()
 { return this.otherpart; }
 public void setIt(OtherPart dp) { ... }
} // end Class Primitive
public class OtherPart { ...}
public class Composite {
 private Primitive prime;
 public Data [] datas;
 public Composite()
 { this.prime = new Primitive(); }
 private Primitive getIt()
 { return this.prime; }
} // end Class Composite
public class Data {...}
public class SubData extends Data {
 private DataPart datapart;
 ...
} // end Class SubData

public class DataPart {...}
```

Discovering components from Java Code

Since the present work considers the static analysis of source code, one cannot extract the exact dynamic information about components, as only the information of types is examined. The set of types (classes, interfaces, generics) in the Java source code is called the *types of interest*. The component model recognises data types (*DTypes*) and component types (*CTypes*). An instance of a *CType* is a *component*, while a *value* is an instance of a *DType*. The types of interest are a disjunction of two sets, *DTypes*, *CTypes* and *ETypes*, the latter are the external types to the project under study. The *composition structure* of a type is defined as the types of its fields or attributes. The authors consider the maximal structure, that is, all the defined attributes and the inherited ones are collected, but the super private fields are not considered since, in Java, they cannot be accessed in the subclasses. A *visible* member in a type is a public or default member, and conversely private and protected ones are called non visible. For component types, an additional constraint is added: non-visible members can only be called on this and super. A *service* is a visible method. A method *signature* is defined by a name, typed parameters and resulting type (as usual the authors use void for procedures). *Provided services* are all the available services defined in the type. The *required services* of a given type are those methods that are defined in another type and are called in the source code of the considered type. Communications occur dynamically when a component requires the service of another component; a communication link denotes such a communication. There is a *communication channel* between the two component types if a block of code of the source component type contains a call to a method of the target component type. *Subtyping relationships* are computed from the two Java subtyping relationships: extends and implements.

Components qualification

The principles and rules presented in this article mostly come from ArchJava [12], but have been modified and extended. ArchJava is a language extension to Java that seeks to integrate a software's architecture with its implementation. As in the ArchJava language, the present work avoids hidden communications that have been established via data sharing, see AliasJava [15] for a solution, and ignores the use of the Java reflection API. A source code analyser has been created, which is able to identify violations of our CI rules in Java code and then to classify in *DType* or *CType* the types of interest that were found.

The communication integrity rules

The CI rules, described below, prevent subcomponents from escaping their enclosing parent component and are used to distinguish *DType* from *CType* in Java legacy code.

Rule. Wrong Signatures:

1-a) Types passed as parameters of, or returned by, services enclosed in *CTypes* or *DTypes* are *DTypes*; the service signature is qualified as a *wrong signature*.

1-b) The rule 1-a also applies to any constructor, regardless of its modifier.

1-c) Non-visible methods can have component types in their signatures, as long as they are called on this or super.

Rule. Composition:

2-a) Types occurring in the structure of *DTypes* are *DTypes*.

2-b) Types in visible fields of *CTypes* are *DTypes* since their instances are publicly available.

2-c) Types in static fields of *CTypes* are *DTypes* since their instances are shared by several instances.

2-d) *CTypes* can have non-static and non-visible fields of *CTypes* but they should only be accessed *via* this or super, to prevent components escaping from their parent components.

Rule. Subtyping:

3-a) Subtypes of *DTypes* are *DTypes*. This follows from rule 1-a, since instances of the subtype could be used as parameters or result, using polymorphism resulting from subtyping.

3-b) Exception: The above rule does not apply for *ETypes* since it is convenient to extend existing libraries. The communication integrity property can be lost when inheriting from external data types. Inherited methods, required redefinitions or downcastings are possible problems. To provide a better checking in case of extending external types is still an open problem [16]. One restricted way is to check for suspicious downcasts (see below).

3-c) *DTypes* can be a subtype of *CTypes*, but if it inherits from an inner class, rule 5-a should apply on the subtype.

Rule. Arrays and Generics:

4-a) Actual types of arrays and generics in services are *DTypes*.

4-b) Actual types of arrays and generics in visible and static field declarations are *DTypes*.

4-c) The rule 4-b also applies to non visible fields of *DTypes* (from 2-a).

4-d) Formal parameters of generics in the case of generic realisation used as a superclass or super interface are *DTypes* (from 4-a).

4-e) In addition to 4-d: The subclasses and implementations of the generic realisation (from rule 3-a) are *DTypes*.

Rule. Nested Classes:

5-a) Parent classes with *DType* inner classes are *DTypes*. If an inner class is a *DType*, one of its instances could escape from its context and could allow access to the enclosing component reference itself.

Rule. Exception Classes:

6-a) Exception types are *DTypes*. This is a strict and pragmatic rule.

Rule. Enumeration Classes: An enumeration class (enum) defines a public class with a set of public constants.

7-a) Enumeration classes are *DTypes*.

Component extraction results

The rules defined in the previous section could at first seem strict. In order to "relax" the component extraction and the refactoring processes, two different sets are defined, which are concerned to particular and well-defined interests. By using each set of rules separately some concerns can be considered and others ignored as proof of the usability of the presented approach and tool support. This also helps maintenance engineers perform incremental refactoring.

The ISEC Set

The ISEC set does not check the wrong constructor signatures. To restrict constructors is a major issue in OO programming; this set of rules allows component types in constructors, which enables in fact a violation of the communication integrity property. This set does not consider the use of the static modifiers for fields and class members in the source code under study. These static modifiers are considered not so important in OOP and CBSE applications. Finally, the wrong signatures are checked on services of *CTypes* and *DTypes*. This last point was to simplify both the checking process and the refactoring process. Since the set is stricter on *DTypes*, it is easier to convert a *DType* into a *CType*. This set includes the rules 2-a, 2-d, 3-*, 4-a, 4-c, 4-d, 4-e, 6-a; including 1-a, 1-c for all types, and 1-c, 2-b, 2-c, 4-b, 5-a for those without the static cases.

The AJ Set

The AJ set includes the rules for checking all static modifiers and the rule for checking wrong

constructors. The idea is to include a more complete set of rules but still checking wrong signatures on services of *CTypes* and *DTypes*. This set includes the rules 2-*, 3-*, 4-*, 5-a, 6-a; including 1-* for all types.

Experiments

The two tools implementing each set of rules were run on several examples of various sizes, coming from different repositories and illustrating different application domains. The examples can be found on SourceForge [17] (e.g. Metrics), and some others on the Jakarta Project [18] (e.g. REGEXP). Some specific applications (e.g. MineSweeper, Javacalc) were also collected, and some others (e.g. JavaCompExt, NIM game, and simplification) were designed by the authors of this work. The tools were run on over 20 examples, ranging from simple examples of 100 Lines Of Code (LOC) to real size applications of 230 KLOC (thousands of lines of code).

The percentage of component types relative to the total number of types (*%R = #CTypes/(#CTypes+#DTypes)*) gives a partial evaluation of the CBSE quality relevant in the context of this work. As a preliminary remark, types respecting the sets of CI rules were found in every application tested, even in traditional OOP applications. The main entry types of the applications, the main class, test classes or top layer classes, are often considered as component types. The main reason for this is that any other types in the application do not use them, and if they are designed in a good object-oriented manner they are not responsible for violations of the CI rules.

For some applications, which were designed with CBSE in mind, the results are better in terms of number of discovered *CTypes*. For instance, the CoCoMe-OASIS [19] has a ratio of 57%, It was designed with an explicit architecture and implemented with a component-based approach. However, for some others, which claim to be CBSE applications, the results are poor in terms of number of discovered *CTypes* (for instance, COCOME-RCOS with a ratio of 31%). There are various reasons for this bad score. The first is that the component models which are often used as a reference to develop the applications are not compliant with our component model, for example in relation to the concept of hierarchical components and the implementation of composites. Another reason is that designers and programmers violate the CI property and/or do not respect the initial architecture.

Although our component model is not compatible with all the existing component model implementations, the CI property, which is the base of our approach, is compliant with some other implementations like event-oriented programming. The use of message-oriented middleware, like Java Message Service, is compatible with our approach. In message oriented middleware the information is communicated via events, which are instances of classes encapsulating information. Nevertheless, the event types are analysed and qualified correctly according to the rules. One example was the CoCoME-impl project, which uses JMS.

The refactoring process

Several small and middle-sized applications were processed, and target plain Java source code, which can be tested to verify their behaviour: NimI, NimF, Javacalc, Simplification, MineSweeper (a detailed refactoring conducted with the ISEC rule set is described in [8]), Regexp, Metrics, JavaCompExt. This set of projects represents a total of almost 20 KLOC.

The main objective was to remove violations of the CI rules on some of the types of interest, transforming them from *DTypes* into *CTypes*. These applications are generally provided without explicit component architecture, and the refactoring process generally does not target a specific architecture.

Refactoring actions

During the refactoring of the applications several recurrent actions to fix CI rule violations were

processed. The actions described below are intended to restructure the applications under study by removing violations of the CI rules while preserving the original behaviour.

Wrong constructor signature

In this case, the type (T) occurs as a parameter type of the constructor definition and this constructor is called in its type definition or in another type. The general method for removing wrong constructor signatures is to erase T from the set of parameter types while creating the T instance inside the constructor of the enclosing type. Two situations are possible: *i)* the constructor can provide default values for the T instance, *ii)* the T instance can be configured with values passed to the constructor. In either case, this impacts on the constructor calls, which must be changed accordingly.

Wrong sign

A general solution is to remove the need for the method with the wrong signature by substituting its source code. Obviously this is far from a good general solution, but it can be successful if the method is only called once. If E = T and the method is not used outside of T then it can become non visible (making this method private or protected). If the method is used elsewhere or defined in a type other than T then there are two sub cases depending if whether there is a wrong argument or a wrong resulting type.

Wrong argument signature

The action is to replace the wrong method by a new method without the wrong argument type, but with some new parameters available from the T instance. The calls of this wrong method signature must be changed and an attribute of type T must be defined in the context of the call in order to replace the argument.

Wrong result signature

In this case one solution is to make the method non visible, to define a local attribute of type T and to add a public void method which sets the attribute with the method call. Complications arise since the T value is usually accessed to provide information. Thus the full solution needs to delegate the required services to new provided services defined in the enclosing type.

Data type encapsulation

In this case either the enclosing type becomes a *CType* or it is removed.

Visible field

In this case a *CType* contains a visible field of a class, interface, array or generic type. Making the field non visible is the recommended solution to this violation, often many public or default package modifiers are overused. However, it can lead to other modifications if this part is accessed outside of its enclosing type. In this case, defining accessors adds new wrong methods and the wrong signature case above applies.

Static field

Removing the static nature and making the field non-visible or removing this part from the enclosing type will solve this violation. However, if the static field is intended to be shared, its type should remain as a *DType* and no refactoring actions must be processed.

Data type subtyping

Either the super type becomes a *CType* or the subtyping link must be removed. This action could apply to classes, interfaces or generics.

Array and generics

The general principle is to avoid *CTypes* in arrays or generics occurring in visible methods, visible or static parts or as super type. One possibility is to change the scope modifier or the supertype link. An alternative approach is to define a container, using a class in a compliant CBSE way. Nevertheless, it may be difficult to respect the CI rules; this will be discussed in future work.

Inner data type

If the inner type is a *DType* it must be refactored as a *CType* or the inner structure must be changed. For instance the solution may be to extract the inner class from the enclosing context, or to change to a static nested class.

Exception

To change an exception into a *CType* is to remove its exception nature, generally acquired by inheritance from an exception class.

Enumeration

As above there is no solution without removing the enum qualifier.

Refactoring with the two rule sets

The component type ratio (%R) is considered as a simple measure of the componentisation of an application. In table 1, for each row, the first line of data represents the original application, its size given by its LOC, and the componentisation ratio (%R) obtained after processing it with the rule set referred to in the column header. The two subsequent lines represent two different refactoring alternatives; the first is guided by the ISEC rule set and the second by the AJ rule set. An initial exception is the Simplification application; due to the use of singletons and of a purely functional style it was not possible to propose a CBSE refactoring without completely changing the programming paradigm or introducing bad programming practises.

Table 1 Component type ratio before and after refactoring

Project	*Size (LOC)*	*ISEC %R*	*AJ %R*
NimF	89	25	25
_ISEC	127	100	50
_AJ	92	100	100
Niml	123	50	50
_ISEC	100	100	50
_AJ	125	100	100
Javacalc	189	8	8
_ISEC	253	42	42
_AJ	242	42	42
Simplification	454	29	29
MineSweeper	795	30	30
_ISEC	794	90	40
_AJ	825	70	70
REGEXP	3232	43	37
_ISEC	3225	81	75
_AJ	3250	50	50
JavaCompExt	7602	52	36
_ISEC	7611	58	36
_AJ	7721	55	51
Metrics	14470	42	41
_ISEC	14459	52	51
_AJ	14481	48	45

Related work

A survey about architectural degeneration is presented in [20]. One approach is *architectural recognition* [21], which analyses Java source code and generates a model of information containing components and connectors. Mendonça and Kramer analysed the limits of some recovery tools and identified the requirements for effective architecture recovery of legacy systems. This is a complementary but more coarse-grained approach than detecting potential communication integrity violations. The above survey also discusses refactoring support in development environments, like in Eclipse [22]. The refactoring actions suggested in this paper are more advanced than those provided by such an environment. In [23], the authors combine architecture recovery and change dependency analysis, but the components they consider are files, not true programmed components.

In the context of refactoring tool support is needed to evaluate the quality of applications and to guide the restructuring process. There is a lack of tools for assessing the quality of component-based source code. Metrics based tools, architecture recovery tools, Java analysers and architecture compliance tools are some immediate candidates, however, none of them are devoted to the purpose addressed in this work (see [8] for a deeper discussion).

ArchJava [3, 8] is closely related to the present work. The latter introduces new rules for checking generics, enumerations, and exceptions. The main difference regarding the refactoring process is that the present work defines an inference system mining for data types in pure Java code and propagating this information along inheritance and composition, and coping with most of the Java features. The approach addressed in this paper makes explicit, more rigorous and automated, the distinction between ordinary classes (for data structure) and component types.

In [12, 18], ArchJava is claimed to be incremental, which is found true within certain and well defined limits. The incremental refactoring process in ArchJava only considers *i)* to choose a class, *ii)* transform it into a component type and then *iii)* use the compiler and check the violations of the communication integrity rules. This is a coarse-grained restructuring, and communication integrity enforcement can fail for several reasons as discussed with the rule sets exposed in this work. Discarding the generic, exception and enumeration rules, more fine-grained situations were also identified where CI violations can occur and the refactoring actions to solve them, than the ArchJava approach. For instance, the concept of wrong signature is crucial in the analysis and the refactoring process of a Java application.

Table 2 presents a comparative summary between the features of our approach and those of the approaches we have mentioned.

Table 2 Comparison with related approache

Feature	*Our approach*	*Arch Java*	*Architectural Recognition [21]*
Component-based	X (True programmed components)	X	X (File-based components)
Rule-based approach	X (Extends the rules of ArchJava)	X	
Mines for data types and component types	X		
Incremental refactoring process	X	X	
Detects potential communication integrity violations	X	X	
Provides refactoring actions	X (coarse- and fine-grained restructuring)		X (coarse-grained restructuring)

Conclusion

The Communication Integrity (CI) property is an approach to maintain the software architecture's consistency of CBSD applications. However, the CI property has not been significantly used in refactoring processes besides the formal analysis and practical experiences from ArchJava. The

present work proposes in this context a catalogue of rules to ensure the CI property in Java legacy code according to a light, Java component model. Several differences with ArchJava can be noted. The approach presented in this paper considers strict static checking, fine-grained detection of components and small refactoring steps. A fine-grained and incremental approach helps ensure the refactoring steps are reliable. New rules for subtyping, generics, exception and enumerations are also considered. Groupings of our rules were tested in two rule sets to compare their applicability in identifying and distinguishing data types from component types.

Experiments were conducted on more than 40 projects, which showed consistency in the qualification of components. To further complete the present work, a set of refactoring actions is provided with the intent of removing the CI rules violations in Java legacy code and, thus, to increment the componentisation ratio. In this regard, our experiments with our two sets of rules throw useful information, particular to each set of rules, in order to guide the selection of the refactoring actions to apply. In an in-depth analysis of several projects, some limits of the approach were identified. For instance, the fact that pure functional programming is not compliant with it.

The respect of the communication property can bring some value in evaluating and refactoring Java applications. However, this is not a simple task and tools with a good set of rules are required. The present study shows that there are still some improvements to be done with component types in constructors. Simplifying some rules is also possible, as demonstrated with the prohibition of component types in data type signatures. Future work in this setting will focus on the tool support built to provide assistance for the qualification of component types, and the theoretical side of this work and will consider questions such as: what degree of safety these rules ensure.

References

1. N. Medvidovic, R. Taylor. “A Classification and Comparison Framework for Softwar Architecture Description Languages”. *IEEE Transactions on Software Engineering*. Vol. 26. 2000. pp. 70-93.
2. C. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, W. Mann. “Specification and Analysis of System Architecture Using Rapide”. *IEEE Transactions on Software Engineering*. Vol. 21. 1995.pp. 336-355.
3. J. Aldrich, C. Chambers, D. Notkin. *ArchJava: connecting software architecture to implementation.* Proceedings of the 24th International Conference on Software Engineering (ICSE’02). Orlando, FL, USA. 2002. pp. 187-197.
4. R. Terra, M. Valente, K. Czarnecki, R. Bigonha. *Recommending Refactorings to Reverse Software Architecture Erosion.* 16th European Conference on Software Maintenance and Reengineering (CSMR). Szeged, Hungary. 2012. pp. 335-340.
5. D. Perry, A. Wolf. “Foundations for the Study of Software Architecture”. *Software Engineering Notes*. Vol. 17. 1992. pp. 40-52.
6. J. van Gurp, J. Bosch.“Design Erosion: Problems & Causes”. *Journal of Systems and Software*. Vol. 61. 2002. pp. 105-119.
7. M. Lindvall, D. Muthig. “Bridging the Software Architecture Gap”. *IEEE Compute*. Vol. 41. 2008. pp. 98-10.
8. H. Arboleda, J. Royer. *Component types qualification in Java legacy code driven by communication integrity rules*. Proceedings of the 4th India Software Engineering Conference (ISEC’11). New York, NY, USA. 2011. pp. 155-164.
9. H. Arboleda, J. Royer. *Java Component Refactoring Based on Communication Integrity Violations*. 9th Belgian-Netherlands Software Evolution Seminar. Lille, France. 2010. pp. 115-129.
10. I. Crnkovic, S. Sentilles, A. Vulgarakis, M. Chaudron. “A Classification Framework for Software Component Models.” *IEEE Transactions on Software Engineering*. Vol. 37. 2011. pp. 593-615.
11. J. Bosch, C. Szyperski, W. Weck. C*omponent-Oriented Programming.* European Conference on Object-Oriented Programming (ECOOP) Workshops 2003. Darmstadt, Germany. 2003. pp. 34-49.
12. J. Aldrich, C. Chambers, D. Notkin. *Architectural Reasoning in ArchJava*. Proceedings Eureopean

Conference on Onject-Oriented Programming (ECOOP) 2002. Málaga, Spain. 2002. Vol. 2374. pp. 334-367.

13. M. da Silva, P. de Castro, C. Rubira. *A java component Model for Envolving Software Systems*.18th IEEE International conference on Automated Software Engineering (ASE). Montreal, Canada. 2003. pp. 327-330.

14. J. Gosling, B. Joy, G. Steele, G. Bracha. *The Java Language Specification*. 3rd ed. Ed. Addison-Wesley. Santa Clara, California, USA. 2005. pp. 175-248.

15. J. Aldrich, C. Chambers. *Ownership Domains: Separating Aliasing Policy from Mechanism*. Object-Oriented Programming European Conference (ECOOP'04). 2004. Vol. 3086. pp. 1-25.

16. M. Abi, J. Aldrich, W. Coelho."A case study in re-engineering to enforce architectural control flow and data sharing." *Journal of Systems and Software*. Vol. 80. 2007.pp. 240-264.

17. Slashdot Media. SourceForge. Available: http://sourceforge.net/. Accessed in october 18, 2012.

18. The Apache Software Foundation. The Apache Jakarta Project. Available: http://jakarta.apache.org/. Accessed in october 18, 2012.

19. A. Cansado, D. Caromel, L. Henrio, E. Madelaine, M. Rivera, E. Salageanu. "A Specification Language for Distributed Components Implemented in {GCM}/ ProActive". *CoCoME.* Vol. 5153. 2007. pp. 418-448.

20. L. Hochstein, M. Lindvall. "Combating architectural degeneration: a survey". *Information & Software Technology.* Vol. 47. 2005. pp. 643-656.

21. N. Mendonça, J. Kramer. "Requirements for an effective architecture recovery framework". *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*. ACM. New York, NY, USA. 1996. pp. 101-105.

22. D. Gallardo. *Refactoring for everyone.* IBM developerWorks Technical library. 2003. Available: http://www.ibm.com/developerworks/library/os-ecref/. Accessed in october 18, 2012.

23. C. Stringfellow, C. Amory, D. Potnuri, A. Andrews, M. Georg. "Comparison of software architecture reverse engineering methods". *Information & Software Technology*. Vol. 48. 2006. pp. 484-487.