Refactoring Object-Oriented Applications towards a better Decoupling and Instantiation Unanticipation

Soumia Zellagui, Chouki Tibermacine, Hinde Lilia Bouziane, Abdelhak-Djamel Seriai and Christophe Dony

LIRMM, CNRS, University of Montpellier, France
E-mail: {zellagui, tibermacin, bouziane, seriai, dony}@lirmm.fr

Abstract

Modularity in Object-Oriented (OO) applications has been a major concern since the early years of OO programming languages. Migrating existing OO applications to Component-Based (CB) ones can contribute to improve modularity, and therefore maintainability and reuse. In this paper, we propose a method for source code transformation (refactoring) in order to perform this migration. This method enhances decoupling by considering that some dependencies between classes should be set through abstract types (interfaces) like in CB applications. In addition, some anticipated instantiations of these classes "buried" in the source code are extracted and replaced by declarative statements (like connectors in CB applications) which are processed by a dependency injection mechanism. For doing so, a set of modularity defects has been defined. These defects are first detected in the source code. Then, some refactoring operations are applied for their elimination. An implementation of the method was successfully experimented on a set of open source Java projects. The results of this experimentation are reported in this paper.

1 Introduction

Modularity is a fundamental principle in software engineering, considered as an internal quality attribute that influences external quality attributes, such as maintainability and reuse [7]. A well-modularized system allows collaborative development of different parts (modules) of the same system by different developers. It also enables the substitution or debugging of a module without affecting other modules and the reuse of existing modules in different contexts.

Many existing (especially, business) software systems are built using the Object-Oriented (OO) development paradigm. However, many of these systems, especially large ones, are characterized by a high degree of coupling between their elements, which makes them difficult to maintain and reuse. Conversly, Component-Based (CB) paradigm has been recognized as an approach that emphasizes software modularity and reuse [4]. Therefore, it would be interesting to migrate OO systems into CB ones. This migration enables to benefit from CB development characteristics, in particular, decoupling and instantiation unanticipation [6].

Existing works that propose migration solutions [2, 3] consider that the modularity and reuse unit, i.e. the component, is a group of classes, called a cluster. In these works, if a user wants to develop a new application by reusing an independent class or subset of classes in a cluster, it is required to reuse the entire component. To the best of our knowledge, no solution proposes refactoring classes individually to make them component descriptors. In this paper, we propose a migration solution that considers each class in an OO application as a component descriptor in a target CB application. This solution follows two steps. The aim of the first step is to detect modularity defects, i.e. decoupling and unanticipation violation (presented in Section 2). The second step allows the elimination of these defects by automatically applying a composition of code refactoring operations (Section 3).

Section 4 discusses the results of an experimentation of this solution conducted on a set of open source Java projects. Section 5 surveys related works and Section 6 concludes the paper and presents future works.

2 Decoupling and Instantiation Unanticipation violation

While code decoupling and instantiation unanticipation principles are fundamental, they are not necessarily always respected in existing OO applications. Therefore, it is necessary to identify the symptoms of their violation in an existing OO code to enable their detection and elimintation.

2.1 Decoupling Violation

In CB programming, code decoupling means that components are assumed to communicate only through their interfaces/ports. Therefore, a component has not a direct access to a component with which it interacts. To have this decoupling in OO applications, assuming that each class will correspond to a component descriptor, each class must, for example, expose all its public methods in abstract types (provided interfaces). Then, other classes that use these methods should declare their dependence on these abstract types, which become their required interfaces.

However, most existing OO applications have multiple dependencies between their different classes (direct concrete types) for a cooperative business processing. In particular, it is possible for a field or a parameter to be typed with a concrete class of the application. These situations lead to code decoupling violation.

To deal with decoupling violation, we consider the two symptoms of the modularity defect: "Absence or Incompleteness of Provided Interfaces" (AIPI) and "Absence of Required Interfaces" (ARI). AIPI symptoms are identified when: **1)** a class defines a public non-static method[1] not declared in the interfaces implemented by this class (or no interface is implemented by this class), **2)** a class declares public fields or fields with no explicit visibility modifier, and **3)** a class declares global constants. ARI symptoms are identified when a class declares fields with a concrete class type[2].

2.2 Instantiation Unanticipation Violation

In CB applications, instantiation unanticipation means that the implementation of a component does not include a connection to another given component, *i.e.* an instantiation of a class which is another component descriptor. In fact, a component requiring a service can be connected to any other component providing such a service. This connection should be established only by a third party, who is the developer of the application/component that uses the two components to be connected. To comply with this connection fashion in OO applications, constructor calls should not be used. Instead, declarative annotations should be defined; these are processed by a (depndency injection) mechanism that manages instances at runtime.

To deal with instantiation unanticipation violation, we consider the symptoms of the modularity defect of type EAI (Existence of Anticipated Instantiations). EAI symptoms are identified when a reference to a created object **1)** is stored in a field/local variable, **2)** is a returned value of a method, or **3)** is an argument of a method invocation. In the present work, we consider that these instantiations are not surrounded by a control flow statement.

[1]Methods of the language's standard API are ignored.

[2]Fields whose types are defined in a library are ignored.

The detection of decoupling and instantiation unanticipation violations in an OO application is done by a ("control flow"-insensitive) static analysis of its source code.

3 Refactoring Operations

This section presents a set of refactoring operations to correct modularity defects introduced in the previous section. Table 1 gives an overview of these operations and, for each one, the treated symptom.

Table 1: Refactoring Operations

Symptom	**Operation**
Public fields or fields with no explicit visibility modifier	Change visibilities
Global constants	Move declarations
Public non-static methods not exposed in interfaces	Expose methods
Fields typed with concrete classes	Create required interfaces
Anticipated instantiations	Use dependency injection

Changing the visibility of a field This operation considers the AIPI symptom when a class field is public or has no explicit visibility modifier, i.e., has the package default visibility for Java, for example. In this case, the field visibility is simply changed to private, and a pair of setter/getter methods is inserted to acces this field (only a getter method in the case of a public final field). The resulted methods will be exposed via interfaces as explained through the next refactoring operation.

Exposing Methods through Interfaces

This type of refactoring deals with the AIPI symptom when a class A defines a public non-static method m and its declaration does not exist in any interface. The idea here is to add this declaration to an interface I. This (changed) interface should not be implemented by any other class. Otherwise, i.e., when all the interfaces implemented by A are also implemented by other classes, a new interface I' is created and m's signature is added to it.

We take into consideration the particular case where distribute the exposed methods on several interfaces. We calculate *LCOM (Lack of Cohesion of Methods)* metric to evaluate the cohesion of each signature added to an interface and the other existing methods in this interface.

Someone can find that the use of *Default Methods* in Java 8 can be useful to eliminate this type of modularity defect. But the idea here consists in exposing only the declarations of methods not their implementations.

Moving a Constant Declaration

To deal with a global constant declaration (AIPI symptom type), we move this declaration to one of the interfaces implemented by the class declaring the constant. We create a new interface or use one from the resulted ones after the application of the previous refactoring operation.

Creating Required Interfaces

This refactoring is used when a field is typed with a concrete class A. It consists of the following steps: search all invocations to external methods whose receiver is saved in the considered field, collect the signatures of these methods, create a new interface (considered as the required interface), add signatures to this interface and replace the type of the field by the newly created interface. The last step consists of adding inheritance links between the required interface and the provided interfaces implemented by A (the provided interface extends the required interface). The class' required interfaces will be as many as the number of concrete classes used as types for its fields. However a single required interface is created for two fields with the same type. By applying this type of refactoring and the former ones, the required and provided interfaces are henceforth defined explicitly in the source code.

Using Dependency Injection

In Dependency injection (DI), a (client) class does not depend on a specific implementation (concrete class). The implementation class is instantiated and injected at runtime by an object container, such as Spring's one[3].

The first case that we deal with is the one where an instantiation is made inside a method/constructor, the reference to the created instance is stored in a field, and the instantiation does not take any argument or it takes attainable ones (arguments whose values can be calculated by a static analysis). Its refactoring is done through the following steps: save the arguments of the constructor call if any, delete the instantiation statement from the method/constructor body, replace it by an annotation used by the used DI framework. For example, the @Autowired annotation is used on fields in Spring (the field must be non-final). The annotation @Autowired enables the automatic dependency injection based on the type.

The second case that is treated is the one where an instantiation is made inside a method/constructor and the obtained reference is stored in a local variable. As in the previous case, we suppose that the constructor call does not take any argument or take attainable ones. This local variable is removed from the method body and turned into a private field of the class (this refactoring, transforming a local variable to a field, is failure-safe as it has been experimented in the literature [9]). This field will be treated following the previous case. Renaming this local variable, before moving it, could be another additional refactoring.

In contrast to the previous case, since what is transformed is a local variable and not a field, we use here a lazy initialized DI so that the created field is injected when it is first requested (during the execution of the method/constructor where it was originally declared as a local variable), rather than at startup.

The last case is where instances' references are stored in fields/local-variables while using non-literal values as arguments in their instantiation. To deal with this case, first, a new "default" constrcutor is created in the instantiated class, and the initial constructor call, in the instantiation, is replaced by this new constructor call. Then, a new method that contains exactly what the initial constructor contains is added to the instantiated class. Finally, the instantiation statement is treated following one of the two previous cases, and an invocation statement of the new method is added to the instantiating class.

Anonymous object instantiations, i.e., instantiations which play the role of arguments in method invocations or returned values, for instance, are considered the same as instantiations made as right-hand-side expressions of assignments to local variables. They are processed following the same procedure than the two previous cases.

4 Experiments

We have implemented the described approach[4] using Spoon[5], an open-source library for Java source code analysis and transformation. We conducted some experiments to evaluate the truthfulness of the stated hypothesis of migrating OO applications into CB ones in order to improve their maintainability and reusability quality characteristics. These experiments were conducted to answer the following research questions:

1. What is the efficiency (precision) of the detection phase?
2. To what extent does the proposed approach improve software maintainability?
3. To what extent does the proposed approach improve reusability?

4.1 Used Data & Metrics

For our study, the latest versions of four open source Java projects were used. Table 2 provides a brief description of these projects, which are of different sizes, varying from 5 to 23 KLOC, 50 to 214 concrete types and 1 to 36 abstract types, and developed by different teams to avoid the influence of development team habits on the results.

Table 2: Data collection

System	Description	LOC	No interfaces + Abstract classes	No classes
Jasml	Java classes visualization tool	5732	1 + 0	50
CoCoME	Commercial application	5779	21 + 0	99
FreeCS	Chat server	23012	17 + 6	139
Log4j	A Logging API	20129	20 + 16	214

The first research question deals with measuring the efficiency of the detection phase. To answer this question, we measured precision, a well-known metric in information retrieval. Precision assesses the ratio of true modularity defects to all defects detected by our approach. To obtain the set of relevant defects (that should be identified by any defect detection approach), we analysed the four projects manually.

[3]https://spring.io/

[4]https://cloud.lirmm.fr/index.php/s/QXEV11bUGvYz1Ss

[5]http://spoon.gforge.inria.fr/

Table 3: Detected defects (In each row, M = results of Manual analysis; A = results of Automatic analysis).

System		Public & package fields / All fields	Public non-static methods not exposed / All public non-static methods	Fields of concrete class type	Instantiations that can be injected / All instantiations	Average
Jasml	M	420/487	48/49	26	41/67	
	A	447/487	49/49	27	46/67	
	Precision	93.96%	97.96%	96.29%	89.13%	94.33%
CoCoME	M	32/285	221/338	25	82/106	
	A	34/285	223/338	29	85/106	
	Precision	94.11%	99.1%	86.20%	96.47%	93.97%
FreeCS	M	380/888	571/926	83	79/299	
	A	402/888	837/926	88	86/299	
	Precision	94.52%	68.22%	94.31%	91.86%	87.23%
Log4j	M	365/910	750/1015	150	105/246	
	A	370/910	753/1015	167	133/246	
	Precision	98.65%	99.6%	89.82%	78.95%	91.75%

To answer the second research question, we used the *Maitainability Index (MI)* metric that measures the maintainability of a software system, and which was successfully used in many recent works, such as [5]. High MI values indicate that the system is easy to maintain. MI is calculated using the following formula:

*MI1 = 171 - 5.2ln(V) - 0.23*C- 16.2ln(LOC) + (50 * sin(sqrt(2.46*NOLComments))*

V is the Halstead's volume [1], which is a measure of the mental effort required to develop or maintain a program based on its length, number of operators and operands. *C* is the cyclomatic complexity value; *LOC* is the number of lines of code and *NOLComments* is the number of lines of comments. In the case of systems which do not have considerable comments, the above formula can be simplified to omit the involvement of NOLComments. For our study, the tool used to calculate the MI value is JHawk[6].

During software development, programers often reuse existing APIs to write client code. This requires the reuse of all API's classes even if the client application uses only a small fraction of this API. By answering the third research question, we want to validate the assumption that our approach allows shrinking API classes by keeping only the used classes and discarding the other ones, resulting in a reduction of API size in memory. To do this, we used Log4j as an API, on which our approach was applied, and collected four client applications from sourceforge.net that use this API (jdbcLogDriver, VaadingLog4j, Jag and Marauroa), in addition to CoCoME[7] which also uses Log4j. The sizes of these applications range from 8 to 190 classes.

4.2 Protocol & Results

Research Question 1 (efficiency of detection): We asked four master students and one Ph.D. candidate, who were not involved in this work before, to analyse the source code of these systems manually. We gave the Jasml and CoCoMe systems to the two master students. We asked the other two master students to divide the FreeCS system and each of them analyzed half of the packages. The Ph.D. student was assigned to analyse the Log4j system. The five students used, as a reference specification, a detailed description of the modularity defects we wrote. They produced Excel files containing the number of occurrences of each modularity defect for each class and the total number of defects in the entire project. We report the results of the detection phase in Table 3. It provides for the four systems the number of existing defects, the result of manual analysis (M) in the first line of each row, the defects detected by our implementation (A for automatic) in the second line, and the precision in the third line.

Table 3 shows that a large percentage of the results obtained with our approach are validated manually (from 87.23% in average for FreeCS to 94.33% for Jasml). Recall was not measured because in the analyzed applications, there are no false negatives: defaults which are detected manually (relevant) and not detected (retrieved) with our process.

Research Question 2 (improvement in maintainability): We have calculated the aforementioned formula for each class of the analyzed applications. Table 4 shows the MI scores before and after applying the approach on the four projects. MI represents the average of the classes' MI value.

Table 4: MI values before and after applying the refactoring

System	MI before	MI after	Improvement factor
Jasml	125.49	147.95	1.17
CoCoME	125.12	161.64	1.29
FreeCS	110.21	120.89	1.09
Log4j	114.52	122.68	1.07

From the results of Table 4 and 5, we can observe that MI is improved, with an improvement factor that ranges between 1.02 and 1.29, in the resulting systems after applying the approach. The improvement in maintainability, according to this metric, is between 10 and 30%, which is not an insignificant score, regarding the size of these systems. Then, in order to check if during the evolution of a single system, the proposed refactorings keep stable this improvement in maintainability, we evaluated MI for six versions of Log4j API, which where developed over a period of 17 years.

The MI values for the six versions before and after applying the proposed refactorings are depicted in Table 5. From Table 5, we can see that there is an increase in MI values of Log4j, before applying our approach, in all the analyzed

[6]http://www.virtualmachinery.com/index.htm
[7]http://www.cocome.org/

versions. This is justified by the fact that from a version to another, new functionalities are added to the system or bugs are corrected, but developers of this system pay attention to its maintainability. As an example of modifications that have been performed in version 1.0.4 and contributed to improve the maintainability of version 1.1.3: *FileAppender* class from *org.apache.log4j* package has been splitted into three classes (*ConsoleAppender*, *WriterAppender* and *FileAppender*) and the MI value for this class passed from 120.47 to 122.75 (the average MI for the three new classes).

Table 5: MI values of Log4j versions

Version	# Classes	MI before	MI after	Imp. factor
1.0.4	121	111.31	121.59	1.09
1.1.3	132	112.25	122.47	1.09
1.2.1	157	114.81	120.66	1.05
2.0	76	116.08	119.64	1.03
2.4	93	117.66	121.56	1.03
2.8	135	118	121.16	1.02

As we can observe, in all the versions, the maintainability is improved. However, the improvement is greater in the first versions. This is explained by the fact that starting from the (major) version 2.0, the structure of Log4j has completely changed, and its maintanability was substantially improved. In the following versions, the system keeps a good MI score, even if this is slightly improved by our refactorings. This shows that our refactorings give better results on old legacy systems, compared to new, potentially refactored, ones.

In the following paragraphs, we report on a case study we have conducted in order to evaluate the benefits brought by the proposed approach on the maintenance effort in API migration. API migration is a kind of software adaptation, and is part of the software maintenace activities. The effort in API migration is measured in this case study in terms of the number of tokens in the modified lines of code in a client application. The two APIs (source and target) of our study are XOM [8] and JDOM [9] which are XML document manipulation APIs. We performed an API migration, from XOM to JDOM, of a client application named SleepXomXML [10]. Measurements have been made before and after applying our approach on this application and for the two APIs.

Table 6: API migration results

	SleepXOMXML		Refactored SleepXOMXML	
	XOM	JDOM	Refact. XOM	Refact. JDOM
Number of tokens in modified lines	254	600	269	380
Difference	346		111	

The results of this case study are shown in Table 6. In this table, we can see the number of tokens in the lines of code that have been adapted in the version of the client application, before its refactoring using our approach: 254 tokens. The number of these tokens in the new code (after its migration to JDOM) become 600 (third column). (Difference = 346 tokens.) When considering the application after its refactoring with our approach, the number of tokens in the modified lines is slightly more, 269. But the number of tokens in the new code became much less, 380. We can deduce from the table that in this case study, the difference in the number of tokens has been reduced by more than 3 times (from 346 to 111).

[8]http://www.xom.nu/

[9]http://www.jdom.org/

[10]http://altsol.gr/sleepxomxml/

Research Question 3 (improvement in reusability:) To determine which classes are really used by the Log4j client applications, we analysed these applications's source code manually. Table 7 shows the result of this analysis.

The first column presents the number of classes/interfaces used directly in the client source code. Both *CoCoME* and *jdbcLogDriver* use only one class which is *Logger*. *VaadingLog4j* uses three classes (*Category*, *Priority* and *LoggingEvent*) and *Marauroa* uses two interfaces (*Appender* and *LoggerRepository*) and seven classes. These directly used classes/interfaces have dependencies with other classes/interfaces of Log4j. Their number is depicted in the second column. The proportion of the API used code is 70% for *jdbcLogDriver, VaadingLog4j, Marauroa* and *CoCoME* (the same classes/interfaces are used by these client apps). *Jag* uses directly only one class, *LogLevel*. This is explained by the fact that *Jag* uses another logging implementation which is Apache Commons Logging.

In terms of memory usage, the Log4j API size is 1.3 MB. By applying our approach and keeping only the used classes/interfaces by the client applications, this size can be reduced to 0.7 MB for *jdbcLogDriver, VaadingLog4j, Marauroa* and *CoCoME* and 8KB for *Jag*.

4.3 Threats to validity

The obtained results in the detection phase depend on the specification of modularity defects and on the profile of students. We tried to be as accurate as possible in the description of defects and we have chosen students who have some experience in Java programming. Another aspect can biais the results and is related to the number of persons involved in the experiments: one student was assigned to one system or to a part of a system. Several persons should be assigned per system to have more accurate results. In our study, we gave these students large periods of time (2 weeks in average) to carefully check the defects; and asked them to analyse each class individually and indicate the time spent on that class. The individual results can be checked in the previous repository[11].

Besides, we tried to collect systems of different sizes and developped by different teams to diversify the data. It is sure that with a larger set of systems we may obtain more precise results. However, since the results were all positive with the four studied systems, which vary in size, our intuition, on the interest of transforming OO code into CB one using the proposed refactoring operations, is strengthened.

[11]https://cloud.lirmm.fr/index.php/s/QXEV11bUGvYz1Ss

Table 7: Number of used classes/interfaces in Log4j

Client applications	API's directly used types (classes + interfaces/abstract classes)	API's indirectly used types (classes + interfaces/abstract classes)	API (classes + interfaces/abstract classes)
jdbcLogDriver	1	145 + 28	214 + 36
VaadingLog4j	3	143 + 28	
Jag	1	1	
Marauroa	7 + 2	137 + 28	
CoCoME	1	145 + 28	

5 Related Works

Allier et al [2] proposed a method to automate the process of migrating OO Java applications into CB OSGi ones. This method makes component interfaces operational by the use of two design patterns: Adapter and Façade. In another work, Alshara et al [3] proposed another approach to automatically transform Java applications into OSGi ones. This approach takes as input a Java application and the description of its CB architecture. Then, the code transformation consists of replacing the dependencies (inheritance and instantiation) between classes belonging to different clusters (components) by interactions via interfaces. Shatnawi et al [11] proposed an approach that aims at recovering software components from OO APIs. In this approach, groups of API classes that are able to form components are identified. This identification is based on the probability of classes to be reused together by clients, and the structural and behavioral dependencies among classes.

In these works, the modularity unit, and therefore the reusability unit, is a group of classes (a cluster). If a user wants to develop a new application using an independent class or a subset of classes in a cluster, she/he is obliged to reuse the entire component. To optimize the level of reuse, we defend here the idea of refactoring the classes individually, to make them component descriptors.

In [8], Fowler defined 22 refactorings for Java programs and initially introduced the concept of bad smells in code as an indicator when (and where) to apply refactorings. Shah et al [10] proposed an algorithm that uses various refactoring techniques to automatically remove unwanted dependencies in Java programs. This algorithm is designed to eliminate modularity defects represented by four types of anti-patterns: circular dependencies between packages, subtypes knowledge, abstraction without decoupling and degenerated inheritance. They classified dependencies between classes in four categories and for each category they specified a refactoring operation. For decoupling classes using interfaces, Steimann et al [12] proposed a fully automated refactoring approach for the introduction of new interfaces. This refactoring calculates from variable decalarations, the minimal types (interfaces), containing all the method declarations needed from the chosen reference and all other references it gets possibly assigned to.

These works share the same goal, which is improving the modularity of an application. Our method has the same goal but with another requirement which is having at the end of the process a class that complies with a component descriptor, in which the decoupling is "pushed further", through the declaration of dependencies as abstract types only, and *via* instantiation unanticipation.

6 Conclusion and Futur Work

In this paper, we presented a method for improving the modularity of object-oriented source code, by focusing on what component-based development brought to programming, *i.e.* decoupling and instantiation unanticipation. Our method was experimented on a set of Java projects to evaluate its efficiency in the detection of modularity defects, and the improvement it brings to maintainability and reusablity. The results of this experimentation showed that there is a potential in using the proposed process in migrating existing legacy OO applications.

As a future work, we plan to perform our analysis on a larger set of applications (with larger sizes). In addition, we envisage to take into consideration other OO mechanisms, like inheritance and instantiation in nested classes, which bring new instances of defects in the analyzed projects. From a tool-support point of view, we project to integrate our solution to the Eclipse IDE as a monolithic refactoring operation and experiment its usability.

References

[1] R. E. Al Qutaish et al. An analysis of the design and definitions of halstead metrics. In *IWSM*, 2005.

[2] S. Allier et al. From object-oriented applications to component-oriented applications via component-oriented architecture. In *WICSA* 2011.

[3] Z. Alshara et al. Migrating large object-oriented applications into component-based ones: instantiation and inheritance transformation. In *GPCE* 2015.

[4] A. Bertolino et al. An architecture-centric approach for producing quality systems. In *Quality of Software Architectures and Software Quality*, 2005.

[5] J. Börstler et al. Beauty and the beast: on the readability of object-oriented example programs. *Software Quality Journal*, 2016.

[6] L. Fabresse et al. Foundations of a simple and unified component-oriented language. *CLSS Journal*, 2008.

[7] N. E. Fenton et al. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., 1998.

[8] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999.

[9] M. Gligoric et al. Systematic testing of refactoring engines on real software projects. In *ECOOP*, 2013.

[10] S. M. A. Shah et al. On the automation of dependency-breaking refactorings in java. In *ICSM*, 2013.

[11] A. Shatnawi et al. Reverse engineering reusable software components from object-oriented apis. *JSS*, 2016.

[12] F. Steimann et al. Decoupling classes with inferred interfaces. In *SAC*, 2006.