Automatic Refactoring of Component-based Software by Detecting and Eliminating Bad Smells
A Search-based Approach

Salim Kebir[1,3], Isabelle Borne[2] and Djamel Meslati[3]
[1]*Ecole Nationale Superieure d'Informatique, BP 68 M Oued-Smar, Algiers, Algeria*
[2]*IRISA, Université de Bretagne-Sud, Vannes, France*
[3]*Laboratoire d'Ingénierie des Systèmes Complexes (LISCO), Université Badji Mokhtar, Annaba, Algeria*

Keywords: Automatic Refactoring, Search-based Software Engineering, Component-based Software Engineering, Genetic Algorithm, Bad Smells.

Abstract: Refactoring has been proposed as *a de facto* behavior-preserving mean to eliminate bad smells. However manually determining and performing useful refactorings is a though challenge because seemingly useful refactorings can improve some aspect of a software while making another aspect worse. Therefore it has been proposed to view object-oriented automated refactoring as a search-based technique. Nevertheless the review of the literature shows that automated refactoring of component-based software has not been investigated yet. Recently a catalogue of component-relevant bad smells has been proposed in the literature but there is a lack of component-relevant refactorings. In this paper we propose detection rules for component-relevant bad smells as well as a catalogue of component-relevant refactorings. Then we rely on these two elements to propose a search-based approach for automated refactoring of component-based software systems by detecting and eliminating bad smells. Finally, we experiment our approach on a medium-sized component-based software and we assess the efficieny and accuracy of our approach.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Due to organizational and market pressures it is not conceivable to develop a software by keeping permanently in mind the idea that it should be easily maintained or changed to fulfill new requirements, as it forces programmers to focus on an extra time-consuming task. This translates into the emergence of bad smells (Fowler et al., 1999), also called design defects or code anomalies. As a consequence, software becomes hard and too costly to maintain. In order to overcome this problem in object-oriented software systems, refactoring has been proposed to provide behavior-preserving means to eliminate bad smells and improve the design of a software (Fowler et al., 1999). However, manually determining and performing useful refactorings is a tough challenge (Seng et al., 2006). In order to address it, it has been proposed to view automated refactoring of object-oriented software as a search-problem where an automated system can discover useful refactorings (O'Keeffe and Cinnéide, 2006). This can be achieved by searching for a sequence of usefull refactorings that improve the overall quality of the system.

The review of the literature shows that automated refactoring of component-based software has not been investigated yet. Recently a catalogue of component-relevant bad smells has been proposed by Garcia et. al. (Garcia et al., 2009) and extended by Macia et. al. (Macia et al., 2013) but there is a lack of component-relevant refactoring operations to overcome these bad smells. Thus refactoring has to be rethought to take into account the different structural aspects that components and interfaces exhibit.

Our contribution in this paper is twofold : first, we propose detection rules for component-relevant bad smells as well as a catalogue of component-relevant refactoring to get rid of them. Second, we rely on these two elements to propose a search-based approach for automated refactoring of component-based systems.

This paper is organized as follows : In Section 2, we present a detailed description of the problem. Section 3 describes our approach with focus on the bad smells detection rules, the proposed refactorings and the genetic algorithm we use. Section 4 contains a dis-

Kebir, S., Borne, I. and Meslati, D.
Automatic Refactoring of Component-based Software by Detecting and Eliminating Bad Smells - A Search-based Approach.
In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2016)*, pages 210-215
ISBN: 978-989-758-189-2
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

cussion of the experimental study that we performed. Finally, Section 5 concludes the paper and presents future perspectives.

2 PROBLEM DESCRIPTION

We address the automated refactoring of component-based software. In concrete, the solution to this problem consists in the detection and elimination of bad smells by operating refactoring operations at the component level.The entries of this problem are : the source code, a set of component-relevant bad smells and a set of component-relevant refactorings. In the following, we give more details on these three elements.

2.1 The Source Code

The source code of a software is the most reliable and accurate source of information describing this latter. However in the context of automated refactoring, the source code in its textual form can not be considered as such because it requires highly expensive parsing operations which degrades the overall process performances. In order to avoid theses costs, source code must be first reified in an intermediate structure called *the source code model*. Such a structure must be designed to allow to measure some properties that we need later during the extraction of bad smells detection rules. It must also be suitable to simulate actual refactoring and check if they do not lead to incoherent situations.

2.2 Component-relevant Bad Smells

Recently, Garcia et. al. (Garcia et al., 2009) identified four representative component-relevant bad smells that they encountered in the context of reverse-engineering and refactoring of large industrial systems. In order to detect such smells, they provide architects with UML diagrams and concrete textual definitions of each bad smell. More recently, in the same perspective, Macia et. al. (Macia et al., 2013) extended this catalogue.

2.3 Component-relevant Refactorings

In general, refactorings are often associated with a set of bad smells (Fowler et al., 1999) by analogy to medical diagnostic-treatments. Nevertheless, in the context of component-based software, object-oriented refactoring seem not to be adequate to refactor component-based software due to the additional level of abstraction introduced by components and interfaces.

3 SOLUTION APPROACH

In our approach, automated refactoring is implemented using a search-based technique. We decompose our approach in three steps: (i) extraction of relevant information from source code to construct the source code model, (ii) formulation of a detection rule and a refactoring for each component-relevant bad smell and finally (iii) exploration of the solutions space using a genetic algorithm. Figure 1 depicts these three steps. Next, we will see in detail each step.

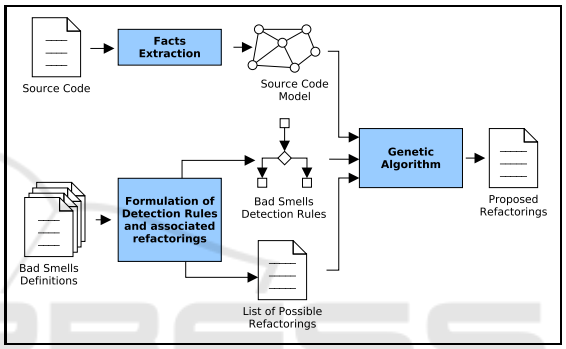

Figure 1: Overall view of our approach.

3.1 Facts Extraction

During this step, we construct from source code and additional artifacts (e.g. XML Configuration files) the source code model in accordance with the metamodel established in figure 2.

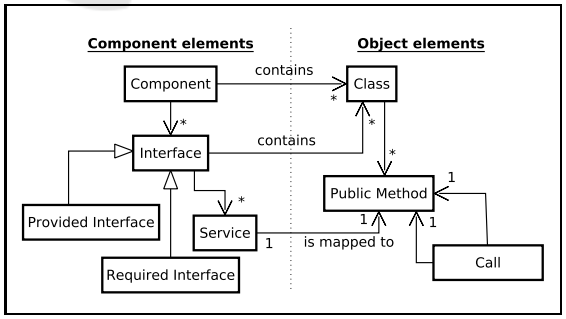

Figure 2: Source Code Metamodel.

We constructed this metamodel according to a recent survey conducted by Vale et al. (Vale et al., 2016). In this surveys, it is stated that components are often considered as sets of classes and interfaces are those classes which have a link with some classes

from the outside of the component (e.g. a method call or attribute use from the outside).

In order to perform the extraction of these information, we have designed and implemented an extraction engine that rely on the API provided by Eclipse JDT[1]. The extraction engine depends on the component model, the underlying programming language and additional component-model specific resources like XML and Manifest configuration files. At this moment, we have successfully defined and implemented an extraction engine for OSGi component-based applications.

3.2 Formulation of Detection Rules and Associated Refactorings

In this section we will revisit component-relevant bad smells. Moreover, we propose to detect each bad smell by refining its description into an *informal rule*, and then extract from these rules *measurable properties* whose range $\in [0,1]$ and pertain to internal attributes and metrics of the constituents of source code metamodel.

Furthermore, for each bad smell we propose a refactoring to eliminate it. Similarly to Fowler's approach (Fowler et al., 1999) we describe significant properties of each refactoring using the following template.

Table 1: Refactoring template.

The **context** summarizes the situation in which the refactoring is needed. That is, it explain *when* performing the refactoring.
The **summary** of the refactoring must reflect in a concise manner *what* action is performed by the refactoring and *where* it have to be performed. The **mechanics** describes *how* to perform the refactoring.

3.2.1 Ambiguous Interface

Definition. Components suffering from this bad smell offer only a *single*, general entry-point. Such interface are referred to as ambiguous (Garcia et al., 2009). Moreover it *dispatches* requests to internal services not belonging to any interface (Garcia et al., 2009).

Detection Rule. According to the previous definition, to judge whether a component suffers from ambiguous interface, we need to know the number of its interfaces and the number of their services. The lower are these two numbers are low, the more the component has ambiguous interfaces. Therefore these information alone are not sufficient to assess how much the interface is ambiguous. Indeed, we also need to know about how much each interface dispatches requests to other internal services not belonging to any interface. Thus, we define the following rule to assess how much a component suffers from this bad smell :

$$AI(C) = \frac{\frac{1}{|C.p|} + \frac{|C.p|}{\sum_{i \in C.p} |SOS(i)|} + \sum_{i \in C.p, k \in C}^{j \in C \setminus C.p} \frac{|SOC(i,j)|}{|SOC(i,k)|}}{3} \quad (1)$$

where :

- $C.p$ denotes the set of provided interfaces of the component C.
- $SOS(i)$ denotes the set of services belonging to the interface i.
- $SOC(i,c)$ denotes the set of outgoing calls from the services belonging to an interface i to public methods belonging to the class c.

Proposed Refactoring: Pull Interface. In a component suffering from *ambiguous interface*, there may be classes that offer services but are not defined as provided interfaces. This reduces analyzability and understandability since a user must look into the implementation of the component to know about the services it offers. The proposed refactoring namely *Pull Interface* consists in creating a new provided interface for a component using the underlying component model mechanisms to turn such classes into provided interfaces.

3.2.2 Connector Envy

Definition. Components with *Connector Envy* encompass extensive interaction-related functionality between two or more other components (Garcia et al., 2009).

Detection Rule. According to the previous definition, a component suffering from this bad smell delegates the majority of its requests to other components. Thereby, the number of its incoming and outgoing calls should be relatively high. Consequently we define the following rule to assess how much a component suffers from connector envy :

$$CE(C) = \frac{\sum_{i \in C}^{j \notin C} (|SOC(i,j) \cup SOC(j,i)|)}{\sum_{i \in C}^{\forall k} (|SOC(i,k) \cup SOC(k,i)|)} \quad (2)$$

[1]Eclipse JDT. http://eclipse.org/jdt/

Proposed Refactoring: Push Component. A component with *connector envy* only delegates calls from a component to another and does not have a proper responsibility. Thus it should be integrated to one or the other. This bad smell reduces reusability insofar the component can not be reused elsewhere. The proposed refactoring namely *Push Component* consists in integrating a component into another by moving all the classes belonging to a component into another one and deleting the old component. By applying this refactoring, components with *connector envy* are eliminated. Thus, the lack of reusability is not relevant anymore.

3.2.3 Scattered Parasitic Functionality

Definition. This bad smell occurs in a system where multiple components are responsible for realizing the same high-level concern and, additionally, some of these components are individually responsible for an additional unrelated concern (Garcia et al., 2009).

Detection Rule. Given a set of components, in order to detect this bad smell, we need to measure the overall cohesion of this set of components and the individual cohesion of each component. In one hand, the higher is the overall cohesion, the more a functionality is scattered among this set of components. In the other hand, the higher is the cohesion of each component, the less this set of components suffer from scattered parasitic functionality. So in order to detect this bad smell, we propose the following rule to assess if a set of components $S = \{C_1, C_2, ..., C_n\}$ suffer from scattered parasitic functionality :

$$SPF(S) = \frac{1}{2} \cdot (LCC(S) + \sum_{C_i \in S} \frac{1 - LCC(C_i)}{|S|}) \quad (3)$$

where :

- $LCC(c)$ denotes the cohesion of the classes belonging to the component c according to the *Loose Class Cohesion* metric proposed in (Bieman and Kang, 1995).

Proposed Refactoring: Merge Components. In a system suffering from *Scattered Parasitic Functionality*, several components may be individually responsible for implementing a wide scope concern. The latter should be encompassed in a single component. This violates the separation of concerns principle since a concern is scattered among a set of elements. The proposed refactoring namely *Merge Components* consists in merging two or more components into a new one by creating a new component containing all the classes belonging to several components and deleting the old ones.

3.2.4 Component Concern Overload

Definition. Components with *concern overload* are responsible for realizing two or more unrelated architectural concerns (Garcia et al., 2009).

Detection Rule. This bad smell can be easily detected by measuring the cohesion of the component. The lower this measure, the more the component is suffering from concern overload. So, we propose this rule to assess how much a component is overloaded with many concerns :

$$CCO(C) = 1 - LCC(C) \quad (4)$$

Proposed Refactoring: Extract Component. In a single component suffering from *Component Concern Overload*, the separation of concerns principle is violated since an element is responsible of two or more concerns. The refactoring proposed here namely *Extract Component* consists in extracting a new component from an existing one by creating a new component containing a subset of classes from the set of classes belonging to a given component.

3.2.5 Overused Interface

Definition. Also called *Fat Interfaces* (Romano et al., 2014), these are interfaces whose clients invoke different subsets of their services (Macia et al., 2013).

Detection Rule. This bad smell can be detected by measuring for each client of a given interface, the number of services invoked together. The higher is this number, the less the interface is overused. Thus we propose in a similar manner to (Romano et al., 2014) to detect this bad smell by measuring the average of the ratio of services invoked from all the clients of a given interface using the following rule :

$$OI(i) = \frac{1}{|CLIENTS(i)|} \cdot \sum_{C_k \in CLIENTS(i)} \frac{|SOC(C_k, i)|}{|SOS(i)|} \quad (5)$$

where :

- $CLIENTS(i)$ denotes the set of clients using the interface i.

Proposed Refactoring: Extract Interface. An interface suffering from *Interface Overload* may be caused by a *God Class* (Fowler et al., 1999). The refactoring proposed here namely *Extract Interface*

consists in extracting a new interface from an existing one by creating a new interface containing a subset of methods from the set of methods belonging to a given interface.

3.3 Genetic Algorithm

Basically Search-Based approaches rely on three key ingredients (Bavota et al., 2014) : (i) *an individual representation* used to encode a solution to the problem; (ii) *a fitness function* which is a mean to assess the quality of a given individual; and (iii) *change operators* which are used to produce new neighborhood solutions starting from existing ones. In order to implement a genetic algorithm (GA) for automated refactoring of component-based software, we describe in the following each of the three above-mentioned elements and how they are articulated within the genetic algorithm.

3.3.1 Individuals Representation

In our approach, individuals are composed of two elements (Figure 3):

- **The genotype** which is an ordered variable-length sequence of refactorings including necessary parameters. When the sequence of refactorings is executed, it performs these changes and produces a modified version of the source code model.
- **The phenotype** which is the obtained source code model after performing the sequence of refactorings to the initial source code model in the order that is given in the genotype.

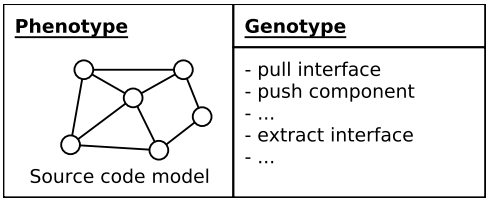

Figure 3: Individual Representation.

Our use of a source code model as a phenotype enables efficient computation of bad smells detection rules.

3.3.2 Fitness Function

In our approach, the fitness function is the sum of the five above-defined rules used to detect bad smells in all components and interfaces of the application. The fitness function is evaluated on an individual by (i) running the sequence of refactoring operations contained in its genotype and (ii) evaluating the detection rules on the resulting source code model contained in its phenotype.

3.3.3 Change Operators

In each iteration, the GA starts by (i) selecting chromosomes that will form a mating pool for crossover and mutation using the roulette wheel selection. This selection is based on the fitness value of individuals. Then (ii) the offspring is generated by applying one-point crossover on each pair to generate two new chromosomes. After that, (iii) mutation is applied to each chromosome in the current population with a user-defined probability. It either replaces a randomly chosen refactoring operation by a new one or randomly inserts/deletes a new refactoring operation to the genotype. The process continues until the choosen number of generations is reached.

4 CASE STUDY

We have experimented our approach on Eclipse MAT (Memory Analyzed Tool)[2] which is an OSGi standalone application that supports programmers to detect memory leaks. Eclipse MAT contains 12 components (OSGi Bundles). Figure 4 depicts the dependencies between Eclipse MAT components with focus on ones severly affected by bad smells (colored in grey).

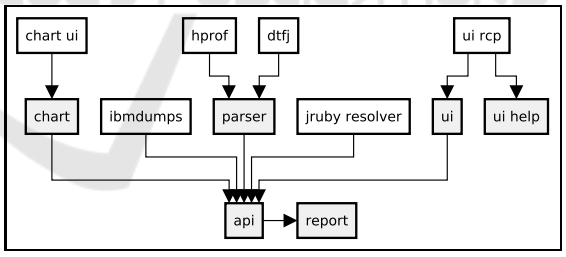

Figure 4: Dependency diagram of Eclipse MAT.

4.1 Approach Efficiency

In our genetic algorithm, we used 1000 generations for a population size of 20. The results of our evaluation are summarized in Figure 5.

We notice that our approach improves pretty good the value of the fitness function. Indeed we have found that the value of the fitness function of the best proposed solution was 3.86. This indicates that $4.41(8.27-3.86)$ of bad smells have been fixed which gives an acceptable efficiency value of $53\%(4.41/8.27)$.

[2]Eclipse Memory Analyzer Tool : www.eclipse.org/mat

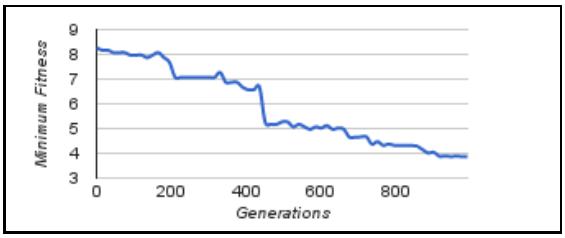

Figure 5: Fitness value evolution (Lower values are better).

4.2 Approach Accuracy

We manually investigated the obtained design to judge if the proposed refactorings are accurate and we have found that the best solution produced by the GA contains 9 components (Figure 6).

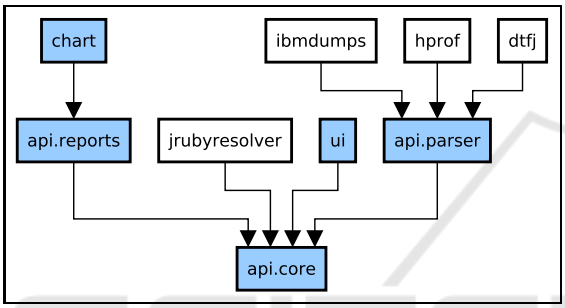

Figure 6: System design after applying refactorings.

We notice that in the original design 6 components were severly suffering from bad smells (colored in grey in Fig. 4). However, 8 components have been refactored into 5 new ones (colored in blue in Fig. 6) and the 4 remaining components stayed untouched. Among these 8 components only 2 ones were not affected by bad smells. This gives us a *false positives* value of 16.66%$(2/12)$. This low value indicates that our approach is very accurate on detecting and correcting bad smells.

5 CONCLUSION

In this paper, we have addressed automated refactoring of component-based software systems. To tackle this problem, we have proposed detection rules for the recently proposed component-relevant bad smells as well as a catalogue of component-relevant refactorings. Then, we relied on these two elements to propose a genetic algorithm to find the best sequence of refactorings to perform. We have experimented our approach on a medium-sized software and evaluated it in terms of efficiency and accuracy.

To the best of our knowledge, our approach is the first attempt to automated refactoring of component-based applications. We believe that we can further improve it in the future. In the short term, we plan to extend our extraction engine to support more component models. In the long term, we plan to use component-relevant quality metrics to improve the exploration of the solution space.

REFERENCES

Bavota, G., Di Penta, M., and Oliveto, R. (2014). Search based software maintenance: Methods and tools. In *Evolving Software Systems*, pages 103–137. Springer.

Bieman, J. M. and Kang, B.-K. (1995). Cohesion and reuse in an object-oriented system. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 259–262. ACM.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). Refactoring: Improving the design of existing programs.

Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Toward a catalogue of architectural bad smells. In *Architectures for adaptive software systems*, pages 146–162. Springer.

Macia, I., Garcia, A., Chavez, C., and von Staa, A. (2013). Enhancing the detection of code anomalies with architecture-sensitive strategies. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 177–186. IEEE.

O'Keeffe, M. and Cinnéide, M. O. (2006). Search-based software maintenance. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10–pp. IEEE.

Romano, D., Raemaekers, S., and Pinzger, M. (2014). Refactoring fat interfaces using a genetic algorithm. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 351–360. IEEE.

Seng, O., Stammel, J., and Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916. ACM.

Vale, T., Crnkovic, I., de Almeida, E. S., Neto, P. A. d. M. S., Cavalcanti, Y. C., and de Lemos Meira, S. R. (2016). Twenty-eight years of component-based software engineering. *Journal of Systems and Software*, 111:128–148.