# A Genetic Algorithm for Automated Refactoring of Component-Based Software

Salim Kebir
Ecole Nationale Superieure d'Informatique
BP 68 M Oued-Smar
Algiers, Algeria
s_kebir@esi.dz

Isabelle Borne
IRISA
Université de Bretagne-Sud
Vannes, France
isabelle.borne@irisa.fr

Djamel Meslati
Laboratoire d'Ingénierie des Systèmes Complexes (LISCO)
Université Badji Mokhtar
Annaba, Algeria
djamel_meslati@yahoo.fr

## ABSTRACT

Nowadays a software undergoes modifications done by different people to quickly fulfill new requirements, but its underlying design is not adjusted properly after each update. This leads to the emergence of bad smells. Refactoring provides *a de facto* behavior-preserving means to eliminate these anomalies. However, manually determining and performing useful refactorings is known as an NP-Complete problem as stated by Harman et al. [9] 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. However the review of the litterature 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 a catalogue of component-relevant refactoring as well as detections rules for component-relevant bad smells. Then we rely on these two ingredients to propose a genetic algorithm for automated refactoring of component-based software systems.

## Keywords

Genetic Algorithm, Refactoring, Component-Based Software Engineering, Bad Smells

## 1. INTRODUCTION

Lehman's first and second laws on software evolution state that a software system written to reflect some real-world activity needs to be adapted or else it becomes less useful and as it evolves, its complexity increases unless effort is performed to reduce it [15]. This translates into the emergence of bad smells [7], also called design defects or code anomalies. As a consequence, software becomes hard and too costly to maintain.

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. In order to overcome this problem in object-oriented software systems, refactoring provides behavior-preserving means to eliminate bad smells and improve the design of a software [7]. However, manually determining and performing useful refactorings is a tough challenge [23]. Furthermore, the questions are: which useful refactorings should be performed ? where should they be performed ? in what order ? how to perform them properly ? and how to assess their positive impact on the software design in the long term ?

In order to tackle these questions, it has been proposed to view automated refactoring of object-oriented as a search-problem where an automated system can discover useful refactorings [9][19]. This can be achieved by searching for a sequence of usefull refactorings that improve the overall quality of the system.

The review of the litterature shows that automated refactoring of component-based software has not been investigated yet. This is because when considering component-based software systems, and especially those developed on top of component models most of whom rely on object technology [14][4], object-oriented refactorings seem to be inadequate since object-oriented bad smells are insufficient to express bad situations at component level due to the additional level of abstraction introduced by components and interfaces. Recently a catalogue of component-relevant bad smells has been proposed by Garcia et. al. [8] and extended by Macia et. al. [16] 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 a catalogue of component-relevant refactoring as well as detections rules for component-relevant bad smells. 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 : Section 2 gives background of related works. In Section 3, we present a detailed description of the problem. Section 4 describes our approach in detail, with focus on the bad smells detection rules, the proposed refactorings and the genetic algorithm we use. Sec-

.

BICT 2015, December 03-05, New York City, United States

DOI 10.4108/eai.3-12-2015.2262353

tion 5 contains a discussion of the experimental study that we performed. Finally, Section 6 concludes the paper and presents future perspectives.

# 2. RELATED WORKS

## 2.1 Bad Smells Detection

Fowler [7] was the first to introduce bad smells and to associate them with refactorings, although he defines bad smells informally, maintaining that only human intuition may guide a decision on whether some refactorings are necessary or not [6]. However many bad smell detection techniques have been proposed in the literature. To the best of our knowledge, all of them rely on rules based on properties captured through software metrics to detect bad smells [13], and the only difference between them lies in the process of formulation of these rules. Furthermore, these techniques can be categorized according to their level of automation and the number of detected bad smells.

## 2.2 Search-Based Refactoring

According to [10], there is two kinds of search-based refactoring approaches : *direct* and *indirect* approaches. In the *direct* approaches, the design of the source code itself is improved. That is to say, the search space contains different variants of the source code, and the exploration of this latter consists of performing fine refactoring operations to move from one individual to another. Examples of such approaches include [20][18]. In the *indirect* approaches, the design of the source code is indirectly improved through the *search* of a sequence of refactoring. In such approaches, the final result is the sequence of refactorings that best enhances the source code. Such indirect approaches has been proposed in [23][21][5]. Although many works have been proposed for search-based refactoring, none of them has considered component-based systems.

# 3. PROBLEM DESCRIPTION

According to two recent surveys, the first one conducted by Crnkovic et al. [4] and the second one by Lau et. al. [14], almost all major component models are based on object-oriented programming languages and use their underlying mechanisms (e.g. packages, classes, interfaces, methods) and additional resources (e.g. OSGi Bundles, EJB XML Configuration) to reify components and interfaces. Also, many of identifying software components from object-oriented source code approaches are based on clustering algorithms that act on an object-oriented system to produce high intra-cohesive and low inter-coupling set of classes [3]. In other words, components are 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).

We address the automated refactoring of component-based software. The considered problem is about searching for the best sequence of refactorings which improves the source code of a component-based software written on top of a known component model. In concrete, the solution to this problem consists in the detection and elimination of bad smells by operating refactoring operations at the component level, such that to improve the overall quality of the source code.

The entries of this problem are :

- **The source code :** The source code of a software is the most reliable and accurate source of informations 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. We have defined in a previous work [12] a mapping model between object-oriented concepts (i.e. classes, interfaces, packages, methods, method calls) and component based software engineering ones (i.e. components, interfaces, services). In figure 1 we propose an adaptation of this model that acts as a source code model.

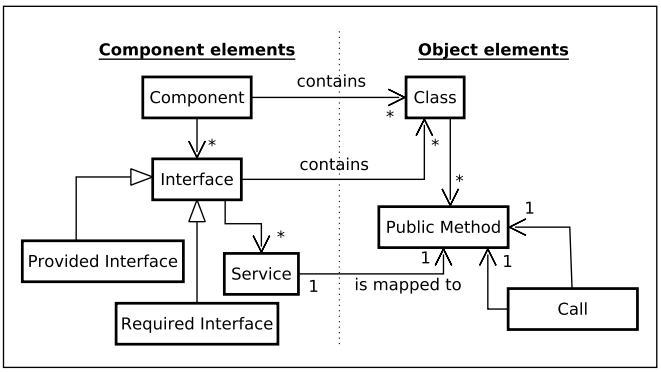


**Figure 1: Mapping model**

- **A set of component-relevant bad smells :** Recently, Garcia et. al. [8] 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. [16] extended this catalogue. Although the two authors define bad smells informally, maintaining that only human intuition may guide a decision on whether some refactorings are necessary or not [6]. Many bad smell detection techniques have been proposed in the literature. To the best of our knowledge, all of them rely on formulating detection rules based on properties captured through software metrics to detect bad smells [13], and the only difference between them lies in the process of formulation of these rules.

- **A set of component-relevant refactorings :** In general, refactorings are often associated with a set of bad smells [7] by analogy to medical diagnostic-treatments. Nevertheless, in the context of component-based softwares, object-oriented refactoring seem not

to be adequate to refactor component-based software because of the two following reasons :

- *Subject of transformation is not the same :* Object-oriented refactoring focuses on classes, methods, attributes and hierarchies where components consists of one or more classes and interfaces consists of classes interacting with the outside of the component.
- *Object-oriented bad smells are inadequate :* The main reason to refactor is to get rid of bad smells. Currently, object-oriented refactorings are associated with a set of object-oriented bad smells. Therefore, considering such bad smells is not adequate on component-based softwares since they are insufficient to reflect bad situations at component and interface level.

# 4. SOLUTION APPROACH

In our approach, automated refactoring is implemented using a genetic algorithm. For this, we decompose our approach in three steps : (i) extraction of relevant informations from source code and artifacts to construct the source code model, (ii) formulation of a detection rule for each bad smell by studying its textual definition, and (iii) explore the solution space using a genetic algorithm. Figure 2 depicts these three steps. Next, we will see in detail each step.

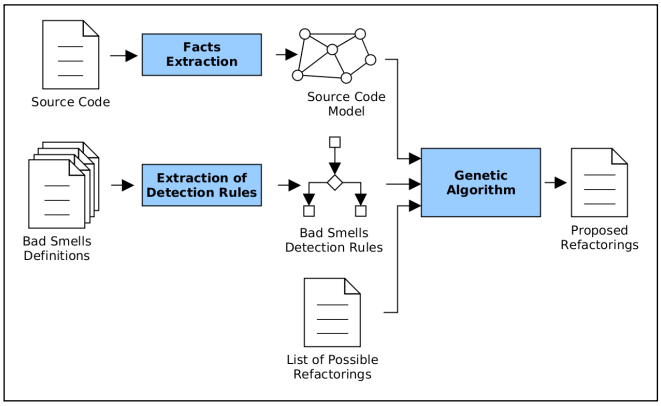


**Figure 2: Overall view of our approach**

## 4.1 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 mapping model established in section 3. In order to perform the extraction of these informations , 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.

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

## 4.2 Formulation of Component-Relevant Bad Smells Detection Rules

In this section we will revisit component-relevant bad smells. Moreover, we propose to detect each bad smells 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 our mapping model.

### 4.2.1 Ambiguous Interface

Components suffering from this bad smell offer only a *single*, general entry-point. Such interface are referred to as ambiguous [8]. Moreover it *dispatches* requests to internal services not belonging to any interface [8]. An ambiguous interface reduces analyzability and understandability since a user must look into the implementation of the component to know about the services it offers.

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

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

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 $c$.

### 4.2.2 Connector Envy

Components with *Connector Envy* encompass extensive interaction-related functionality between two or more other components [8]. This bad smell reduces reusability insofar the component can not be reused elsewhere.

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 high. Furthermore its overall cohesion should be low since it does not have a proper responsibility. Consequently we define the following rule to assess how much a component suffer from connector envy :

$$CE(C) = \frac{1}{2} \cdot \left( \frac{\sum\limits_{i \in C}^{j \notin C} (|SOC(i,j) \cup SOC(j,i)|)}{\sum\limits_{i \in C}^{\forall k} (|SOC(i,k) \cup SOC(k,i)|)} + (1 - LCC(C)) \right)$$

where :

- $LCC(c)$ denotes the cohesion of the classes belonging to the component $c$ according to the *Loose Class Cohesion* metric proposed in [2].

#### 4.2.3 Scattered Parasitic Functionality

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 [8]. A system suffering of this bad smell violates the separation of concerns principle in two ways : firstly, a concern is scattered among a set of elements, secondly, a component is responsible of realizing more than one concern at the same time.

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 more the overall cohesion is high, the more a functionality is scattered among this set of components. In the other hand, the more the cohesion of each component is high, 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 :

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

#### 4.2.4 Component Concern Overload

Components with *concern overload* are responsible for realizing two or more unrelated architectural concerns [8]. This runs counter the separation of concerns principle since an element of a system is responsible of two or more concerns.

This bad smell can be easily detected by measuring the cohesion of the component. The more this measure is low, 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 :

$$CCO(C) = 1 - LCC(C)$$

#### 4.2.5 Overused Interface

Also called *Fat Interfaces* [22], these are interfaces whose clients invoke different subsets of their services [16]. This violates the interface segregation principle [17] since clients depend on services that they do never invoke.

This bad smell can be detected by measuring for each client of a given interface, the number of services invoked together. The more this number is high, the less the interface is overused. Thus we propose in a similar manner to [22] 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 :

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

where :

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

### 4.3 Component-Relevant Refactoring

In this section, we propose a set of component-relevant refactorings to get rid of the previously described bad smells. Similarly to Fowler's approach [7], for each refactoring, we describe its significant properties using the following template.

**Table 1: Refactoring template**

| |
|---|
| The **name** and **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 **context** summarizes the situation in which the refactoring is needed. That is, it explain *when* performing the refactoring. |
| The **motivation** explains *why* the refactoring should be done by assessing the benefits brought to quality attributes. |
| The **mechanics** describes *how* to perform the refactoring. |

#### 4.3.1 Pull Interface

- **Summary:** Create a new provided interface for a component.
- **Context:** In a component suffering from *ambiguous interface*, there may be classes that offer services but are not defined as provided interfaces.
- **Motivation:** By applying these refactoring, analyzability and understandability of the component are increased since a user is no longer required to look into the implementation of the component to know about the services it offers.
- **Mechanics:** Use the underlying component model mechanisms to turn a class into a provided interface.

#### 4.3.2 Push Component

- **Summary:** Integrate a component into another.
- **Context:** 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. Further more a component with *connector envy* only delegates calls from a component to another and should be integrated to one or the other.
- **Motivation:** By applying this refactoring, a concern is no longer scattered among a set of elements. Thus, separation of concerns principle is met.
- **Mechanics:** Move all the classes present in a component into another one. And delete the old component.

#### 4.3.3 Extract Component

- **Summary:** Extract a new component from an existing one.
- **Context:** 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.

- **Motivation:** By applying this refactoring, reusability is increased as well as cohesion.
- **Mechanics:** Determine a subset of classes from the set of classes belonging to a given component to obtain a new component.

#### 4.3.4 Extract Interface

- **Summary:** Extract a new interface from an existing one.
- **Context:** An interface suffering from *Interface Overload* may be caused by a *God Class* [7].
- **Motivation:** After extracting a sub-interface from an overused interface, clients do not longer depend on services that they do never invoke. This fulfill the interface segregation principle [17].
- **Mechanics:** Determine a subset of the set of methods belonging to a given interface to obtain a new inteface.

## 4.4 Genetic Algorithm

Genetic Algorithms (GAs) [11] are evolutionary algorithms inspired from the Darwinian theory of natural evolution. They simulate the evolution of species emphasizing the law of survival of the nearly-best to solve optimization problems. Thus, these algorithms start from a set of initial individuals (i.e. solutions), and to use naturally inspired evolution mechanisms to derive new and possibly better solutions which gives the best approximation of the optimum for the problem under investigation. To this end, GAs rely on three key ingredients [1] : (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.

Basically, GA proceeds using the previous elements, as follows (Figure 3): first it randomly generates an initial population, then it performs crossovers and mutations on the fittest elements of this population until the chosen number of generation is reached.

$GA(nbOfGenerations : Integer) : Population$
**Begin**
$i \leftarrow 0$;
$p = initialPopulation()$;
**while** $i < nbOfGenerations$ **do**
  $p' = SELECT(p)$;
  $CROSSOVER(p')$;
  $MUTATE(p')$;
  $p \leftarrow p'$;
  $i \leftarrow i + 1$;
**end while**
**return** $p$;
**End**

**Figure 3: Genetic Algorithm**

In order to implement 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.

#### 4.4.1 Individuals

In our approach, individuals are composed of two elements:

- *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 current source code model in accordance to the one described in section 3. The phenotype is obtained by performing the sequence of refactorings to the initial source code model in the order that is given in the genotype.

Figure 4 depicts the individual representation. This representation carries the following key benefits. First, our use of a source code model as a phenotype to represent the component-basd software design enables efficient computation of bad smells detection rules. Second, we give the possibility to the genotype to contain *invalid* refactorings. By *invalid refactoring*, we mean that it is not able to be performed on the initial source code model and lead to incoherent situations by breaking the necessary preconditions for the following ones (e.g. push component refactoring can prevent other refactorings concerning the same component since the component does not exist anymore). When it is the case, we simply consider the following ones as invalid. However, we do not exclude them from the genotype since they may be valid in the next generation. This allows to explore the solution space more efficiently.

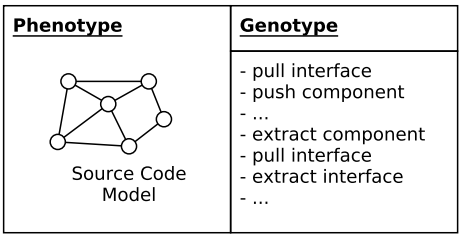


**Figure 4: Individual Representation**

#### 4.4.2 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.

The motivation behind using bad smells detection rules as part of our fitness function is that they are conjectured in the literature to impact the overall quality of systems. Also since refactorings are associated with a set of bad smells [7], this gives us the opportunity to see how much our ones are effective to get rid of component-relevant bad smells by analysing the resulting source code model at a given step of our genetic algorithm.

### 4.4.3 Change operators

In each iteration, the GA starts by selecting chromosomes. This selection is based on the fitness value of individuals. Then the offspring is generated by applying crossover on each pair to generate two new chromosomes. After that, the mutation is applied to each chromosome in the current population with a givn probability. We give in the following how we have implemented each of these three operators.

- *Selection :* We adopt the roulette wheel selection. All of the population selected chromosomes will form a mating pool for the crossover and mutation.
- *Crossover :* In our approach, we adopt the one-point crossover operator which conceptually operates on two genotype (sequence of refactoring in our case) at a time and generates offspring by cutting each of the two parent chromosomes into two subsets of genes. Then two new chromosomes are created by interleaving the two subsets.
- *Mutation :* Our mutation operator either replaces a randomly chosen refactoring operation by a new one or randomly inserts/deletes a new refactoring operation to the genotype. The probability of performing a mutation is chosen by the user as a parameter of the GA.

# 5. CASE STUDY

In this section we present our first evaluation of the GA. Currently only the OSGi [2] component model is considered. In OSGi, a component is known as a bundle [4]. Each bundle is defined by a JAR file containing named packages and a manifest file containing the description of the bundle interaction with the other bundles, mainly the exported packages that are meant to reify provided interfaces and in dual manner the imported packages that are meant to reify required interfaces. In this section, we present the system that we used for our experiments, and we examine and discuss the obtained results.

## 5.1 System under investigation

To perform our experiments, we use the open source Eclipse MAT (Memory Analyzed Tool)[3] which is an OSGi standalone application that supports programmers to detect memory leaks.

We choose Eclipse MAT because it is a medium-sized project which source code is freely available. Figure 5 depicts the dependencies between the system components according to the documentation that we have found by inspecting each JAR File composing the system. The core of the system is a component called *api*. It provides a set of routines, protocols, and tools for analyzing memory usage of a Java application. All of the remaining components either depends directly or indirectly on the *api* component except the *report* component which acts like a bridge between Eclipse MAT and Eclipse Business Intelligence and Reporting Tools. The components *ui*, *ui rcp* and *ui help* are responsible for providing the user interface and interactive help for the tool. The component *jruby resolver* provides support for the JRuby JVM-based language. The *parser* component provides heap parsing utilities which are used by two components: *dtfj* and *hprof*. The first one accesses the heap dump and provides diagnostic tools for analyzing Java classes and objects and their classloaders that were present in the heap. The second one relies on the heap dump to present CPU usage, heap allocation statistics, and monitor contention profiles. The *ibmdumps* component provides support for IBM Java virtual machines. The two remaining components namely *chart* and *chartui* provide feature-rich charting functionalities.

[2]OSGi Alliance : www.osgi.org

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

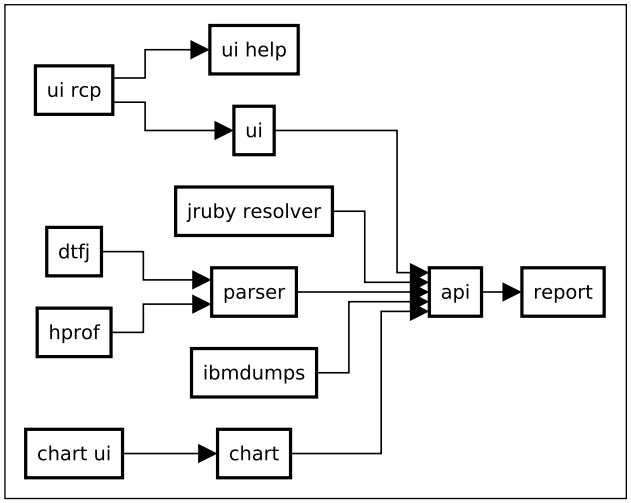


**Figure 5: Dependency diagram of Eclipse MAT**

Table 2 gives a more explanatory overview of system's components and interfaces. For each component, we give the number of provided and required interfaces. Due to space limitations we have omitted the prefix `org.eclipse.mat` from all components and interfaces name.

**Table 2: Eclipse MAT components and interfaces**

| Components | # Provided itfs. | # Required itfs. |
|---|---|---|
| api | 12 | 12 from report |
| chart.ui | 1 | 1 from chart |
| chart | 1 | 12 from api |
| dtfj | 1 | 5 from parser |
| hprof | 1 | 5 from parser |
| ibmdumps | 1 | 12 from api |
| jruby.resolver | 1 | 12 from api |
| parser | 5 | 12 from api |
| report | 12 | |
| ui.help | 1 | |
| ui.rcp | 1 | 8 from ui<br>1 from ui.help |
| ui | 8 | 12 from api |

## 5.2 Settings

First we investigated how much the system is affected by bad smells. To achieve this, for each bad smell we calculate the value of the associated detection rule on each system components and interfaces. For *AI* and *OI* bad smells, we calculated the average of detection rules of all provided interface of a given component. Regarding *SPF* bad smell, since

we can not measure it on a single component, we calculated the average of its detection rule for all possible partitions of the system components containing two or more connected componenets (i.e. we consider only partitions where there exists an indirected path that connects all the components of the partition. For example the partition $\{api, ui.help\}$ has been ommited insofar as there is no path that connects these two components). The measurement results are compiled in table 3.

**Table 3: Bad Smells in Eclipse MAT**

| Component | AI | CE | SPF | CCO | OI |
|---|---|---|---|---|---|
| api | 0.02 | 0.09 | - | **0.59** | **0.87** |
| chart.ui | 0.00 | 0.00 | - | 0.00 | 0.00 |
| chart | **0.76** | **0.42** | - | 0.01 | 0.02 |
| dtfj | 0.00 | 0.00 | - | 0.00 | 0.00 |
| hprof | 0.00 | 0.00 | - | 0.00 | 0.00 |
| ibmdumps | 0.00 | 0.00 | - | 0.00 | 0.00 |
| jruby.resolver | 0.00 | 0.00 | - | 0.00 | 0.00 |
| parser | 0.04 | 0.05 | - | **0.26** | 0.08 |
| report | 0.06 | 0.00 | | 0.02 | 0.12 |
| ui.help | **0.65** | 0.00 | - | 0.00 | 0.01 |
| ui.rcp | 0.00 | 0.00 | - | 0.01 | 0.00 |
| ui | 0.04 | **0.47** | - | **0.31** | 0.01 |
| **Total** | **1.56** | **1.03** | **3.37** | **1.20** | **1.11** |
| **Overall Fitness** | **8.27** | | | | |

By examining these results, we note that the system suffers from approximately eight bad smells. For instance:

- *api* suffers from *CCO* and *OI* respectively because, it is the core of the application and its provided interfaces are used by many clients with different purposes.
- *chart* and *ui.help* suffer from *AI* because they have only a single interface providing few services.
- *chart* and *ui* suffer from *CE* because they delegates the majority of their requests respectively to *chart.ui* and *ui.rcp*.

In order to validate our approach, we propose to answer the two following research questions:

- **RQ1: How much the proposed approach is efficient to correct component-relevant bad smells ?** To answer this question, we calculate the number of corrected bad smells over the total number of bad smells before applying the proposed refactorings. The more this value is high, the more the approach is efficient.
- **RQ2: How much the proposed approach is accurate in correcting detected component-relevant bad smells ?** To answer this question, we calculate the number of false positives and false negatives. On the one hand, false positives just means the ratio of component that have been refactored but are not affected by bad smells. On the other hand, false negatives means the ratio of components that have not been refactored but are seriously affected by bad smells. The less these two values are small, the more the approach is accurate.

## 5.3 Results and discussion

Figure 6 shows the results obtained by applying the genetic algorithm on Eclipse MAT. The line represents the minimum fitness value, for each generation.

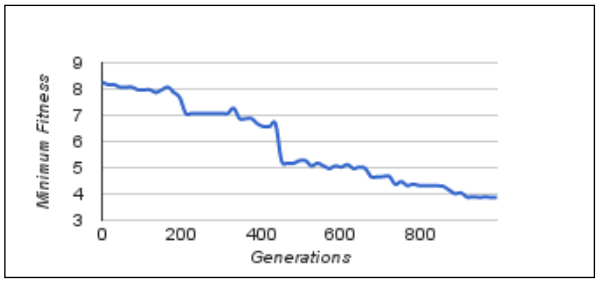


**Figure 6: Optimization results (Lower values are better)**

We used 1000 generations for a population size of 20. As expected, We notice that our approach is able to improve 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 $53\%(4.41/8.27)$.

We investigated the proposed solution to judge if the proposed refactoring are accurate and we have found that the best solution produced by the GA contains 9 components. We have proposed significant names for the newly obtained components (colored in grey in Figure 7). In order to achieve this, we manually inspected their internal source code. This task is not easy, even if it is about a medium-sized system, it was accomplished by providing a lot of effort.

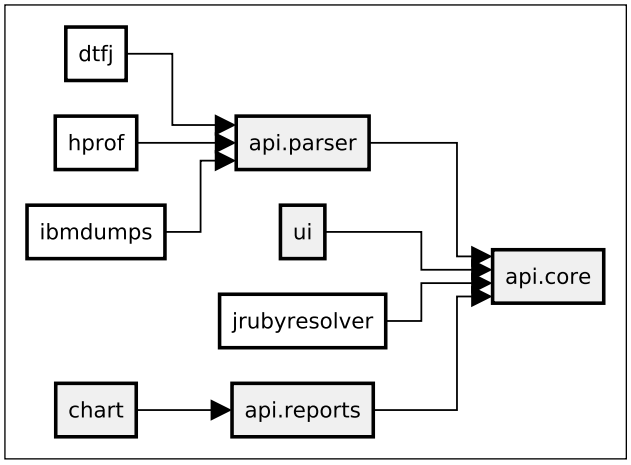


**Figure 7: Evolved System Structure**

Eight components from the original design have been refactored into 5 new components. We note that among these eight components only two ones (*ui.rcp* and *chart.ui*) are not affected by bad smells according to table 3. This gives us a *false positives* value of $16.66\%(2/12)$.

The four remaning components from the original design have stayed untouched, namely: *dtfj*, *hprof*, *ibmdumps* and *jrubyresolver*. This gives us a *false negatives* value of $0\%(0/12)$.

This can be explained by the following reasons: First, Figure 5 shows that these four components are very loosely coupled with the rest of the system. Second, according to their description in the documentation (c.f. subsection 5.1), these components focus on a single purpose and each of them provide a highly cohesive feature. Finally, according to Table 3, these components do not suffer from bad smells.

The obtained false positives and false negatives indicate that our approach is very accurate on correcting detected bad smells.

# 6. CONCLUSION

In this paper, we proposed an approach based on genetic algorithms for automated refactoring of component-based software systems. To tackle this problem, we proposed a set of detection rules for the recently proposed component-relevant bad smells and a catalogue of component-relevant refactorings. Then, we propose a genetic algorithm to find the best sequence of refactorings to perform. We validated our approach in terms of efficiency and accuracy by applying it on a medium-sized system.

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 metrics to improve the exploration of the solution space.

# 7. REFERENCES


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

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

[3] D. Birkmeier and S. Overhage. On component identification approaches–classification, state of the art, and comparison. In *Component-Based Software Engineering*, pages 1–18. Springer, 2009.

[4] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, 2011.

[5] D. Fatiregun, M. Harman, and R. M. Hierons. Evolving transformation sequences using genetic algorithms. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 65–74. IEEE, 2004.

[6] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5–1, 2012.

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

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

[9] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[10] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106–1113. ACM, 2007.

[11] J. H. Holland. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. 1975.

[12] S. Kebir, A.-D. Seriai, S. Chardigny, and A. Chaoui. Quality-centric approach for software component identification from object-oriented code. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 181–190. IEEE, 2012.

[13] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni. A cooperative parallel search-based software engineering approach for code-smells detection. 2014.

[14] K.-K. Lau and Z. Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, 2007.

[15] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[16] I. Macia, A. Garcia, C. Chavez, and A. von Staa. 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, 2013.

[17] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[18] I. H. Moghadam. Multi-level automated refactoring using design exploration. In *Search Based Software Engineering*, pages 70–75. Springer, 2011.

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

[20] M. O'Keeffe and M. Ó. Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(5):345–364, 2008.

[21] F. Qayum and R. Heckel. Local search-based refactoring as graph transformation. In *Search Based Software Engineering, 2009 1st International Symposium on*, pages 43–46. IEEE, 2009.

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

[23] O. Seng, J. Stammel, and D. Burkhart. 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, 2006.