Assessing the Refactoring of Brain Methods

SANTIAGO VIDAL, ISISTAN-CONICET, Argentina
IÑAKI BERRA and SANTIAGO ZULLIANI, UNICEN University, Argentina
CLAUDIA MARCOS, ISISTAN-CIC, Argentina
J. ANDRÉS DÍAZ PACE, ISISTAN-CONICET, Argentina

Code smells are a popular mechanism for identifying structural design problems in software systems. Several tools have emerged to support the detection of code smells and propose some refactorings. However, existing tools do not guarantee that a smell will be automatically fixed by means of refactorings. This article presents Bandago, an automated approach to fix a specific type of code smell called Brain Method. A Brain Method centralizes the intelligence of a class and manifests itself as a long and complex method that is difficult to understand and maintain by developers. For each Brain Method, Bandago recommends several refactoring solutions to remove the smell using a search strategy based on simulated annealing. Our approach has been evaluated with several open-source Java applications, and the results show that Bandago can automatically fix more than 60% of Brain Methods. Furthermore, we conducted a survey with 35 industrial developers that showed evidence about the usefulness of the refactorings proposed by Bandago. Also, we compared the performance of the Bandago against that of a third-party refactoring tool.

CCS Concepts: • Software and its engineering → Software evolution; Maintaining software; Software architectures;

Additional Key Words and Phrases: Code smells, refactoring, Brain Method, long method

ACM Reference Format:

1 INTRODUCTION
The evolution and maintainability of software systems is usually threatened by shortsighted design decisions or by poor-quality source code. These aspects are generally described as code smells. A code smell is a symptom in the source code that helps to identify a design problem [7]. In this way, code smells allow developers to detect fragments of code that should be re-structured, in order to improve the quality of the system. Some symptoms indicative of code smells include: duplicated code, very large methods or classes, or long lists of parameters, among others. In particular, a very common smell is the so-called Brain Method (BM). This smell refers to a large and complex
method that centralizes the intelligence of the class [15]. Thus, one of the main problems of a BM is that, when a change affects the method, the developer is likely to spend considerable time in understanding the source code logic. For this reason, it is argued that BMs decrease the modifiability and understandability of the class in which it is implemented, making it difficult to modify its source code to add new functionality or fix bugs. This kind of methods has been categorized by developers as very severe for system maintenance [33].

A technique commonly used to fix code smells is refactoring [7]. By refactoring the code, the internal system structure can be improved but the system behavior is not altered. In the case of a BM, a refactoring solution involves identifying fragments of the method that deal with a given responsibility and extract them to a new method. Fixing code smells through refactoring can help to improve system quality. Unfortunately, code refactoring is not a practice that developers perform as frequently as expected [39]. The lack of refactoring is generally associated to pressures for delivering customer-visible features, so developers often do not have enough time to fix key code smells [22]. Also, fixing smells is prone to introduce errors in the code, if not performed with due care [28]. Moreover, selecting the appropriate refactorings to fix a given smell can be a hard and task-consuming task for novice developers or developers being unfamiliar with the application code.

Several approaches have been developed in order to automatically identify code smells [15, 24, 42] and prioritize the most critical ones [16, 18, 42, 48]. However, few approaches have the capability to automatically fix code smells [19, 20]. For example, in the case of BM, most tools [35, 41, 51] only propose extractions of code fragments but do not guarantee that the smell will be effectively fixed after applying the refactoring.

In this work, we propose a tool approach called Bandago that assists developers to fix instances of BMs in an automated fashion. The approach has been implemented as an Eclipse plugin, based on the JSpIRIT\(^1\) tool for analyzing code smells [47]. Specifically, Bandago is a recommender system [10, 34] that takes a given BM and proposes to the developer different solutions that fix the smell. The solutions are based on a search-based strategy that repeatedly applies the Extract Method refactoring. A novel aspect of our approach is that it generates solutions using a simulated annealing algorithm [14] that we adapted to work with code refactorings. Furthermore, our main contribution is that Bandago guarantees that those solutions effectively fix the BM smells. Although others have reported on refactoring strategies based on simulated annealing [23, 30, 32], to the best of our knowledge, this is the first work devoted to refactoring of long and complex methods.

We evaluated our approach by conducting a case-study using 10 Java open-source applications. The results showed that Bandago can automatically remove more than 60% of the BMs, thus improving the maintainability of applications. We also performed a qualitative study with 35 industrial developers that confirmed that solutions proposed by Bandago can improve the legibility and reduce the complexity of the source code. In addition, we submitted pull requests of a subset of refactorings suggested by Bandago to the GitHub repositories of 6 of the analyzed applications, and obtained several insights about users’ acceptance of automated refactoring solutions. Moreover, we compared the performance of Bandago with that of an existing refactoring tool (JDeodorant\(^2\)) with positive results. At last, we found that, when applying the refactoring suggestions for Bandago for BMs, other kinds of code smells are also fixed.

The rest of the article is structured as follows. Section 2 introduces the BM smell and formulates its refactoring (based on Extract Method) as a search problem. Section 3 analyzes related work. Section 4 describes the details of the Bandago approach. Section 5 presents a series of experiments.

---

1https://sites.google.com/site/santiagoavidal/projects/jspirit
2https://users.encs.concordia.ca/ nikolaos/jdeodorant/
to evaluate our approach, discussing results and lessons learned. Finally, Section 6 presents the conclusions and outlines future work.

2 BACKGROUND

During the development process, some bad coding practices are common, such as the creation of duplicated code, violations in the encapsulation of a class, or having very complex and long methods or classes, among others. These are symptoms in the source code, called code smells, which help to identify a design problem [7]. Code smells are known to degrade the quality a software system, impacting on the legibility, maintainability, and evolution of the system. Different catalogs of code smells have been presented, such as those of Fowler [7] and Lanza and Marinescu [15]. Both catalogs propose code smells related to methods with too many lines of code.

2.1 Brain Method

In the case of Fowler’s catalog, it presents the Long Method smell, which focuses on the “semantic distance” between what the method does and its name. This catalog suggests that every time a developer implements a block of code that she needs to comment, she should take an action to extract it into a new method. In the case of Lanza & Marinescu catalog, it presents the BM smell for describing methods with a large number of lines of code, high complexity, and low cohesion. Because of its length and complexity, a BM is difficult to understand, debug, test and maintain [33]. In addition, this makes the method very difficult to be re-used. We can say that the long method and BM smells are equivalent. In this article, we focus on the BM because Lanza & Marinescu define heuristic rules for detecting each smell. Specifically, the strategy for identifying BMs in an automated fashion is determined by the following rule [15]:

\[
\text{BrainMethod} = (\text{LOC} > \text{HIGH}/2) \text{ and } (\text{CYCLO} \geq \text{HIGH}) \text{ and } (\text{MAXNESTING} \geq \text{SEVERAL}) \text{ and } (\text{NOAV} > \text{MANY})
\]  

(1)

where LOC is the number of lines of code of the method, CYCLO its cyclomatic complexity, MAXNESTING the maximum nesting level of control structures within the method, NOAV the number of accessed variables by the method, and HIGH, SEVERAL and MANY are predefined threshold values (59, 4, 3, and 7, respectively). These thresholds have been previously used in the iPlasma tool proposed by Lanza & Marinescu [15]. However, the thresholds should be adapted...
Fig. 2. Source code of the updateDiagram Brain Method

to each system taking into account characteristics such as the programming language or the application domain. For example, JSpIRIT allows the developer to set the thresholds using a wizard (Figure 1).

For example, let us consider the code of Overview.updateDiagram³, one of the BMs analyzed in SportsTracker, one of our case studies (Figure 2). This method draws an overview diagram according to the current selection of the user. However, let us note that the method is too large and complex, and it ends up centralizing most of the functionality of the class. When computing the metrics for identifying this smell, their values for updateDiagram are: LOC = 109 (no blank or comment lines), CYCLO = 14, MAXNESTING = 3, and NOAV = 122. Thus, according to Equation 1, we can mark updateDiagram as a BM. Moreover, this method has the largest values of LOC, CYCLO and NOAV of its class. Regarding the values of the other methods of the class, in average

³The complete source code of the class can be found in https://goo.gl/2nyBVi
the values of LOC, CYCLO, MAXNESTING, and NOAV are 23.3 (std. dev. 22.9), 4.4 (std. dev. 4.3), 1.1 (std. dev. 1.3), and 26.6 (std. dev. 25.3). Thus, this method not only is a brain method but also it is the largest and most complex method of its class.

We should also mention that the refactoring of some BMs might not be urgent or desirable [48]. For example, the refactoring of BMs in a class with no changes since its initial implementation (and not expected to be modified in the future) may have low priority when compared to a BM in a class that received modifications in a number of significant revisions. Another example are long and complex methods that implement specific tasks, such as a parser or a state machine. This kind of smells are not necessarily a design problem. In previous works, we analyzed the problem of prioritizing code smells based on multiple criteria [45–48]. A recent study [39] has also discussed reasons for not refactoring complex classes, and those reasons also apply to BMs. However, determining whether a given BM instance should be refactored is out of the scope of the this work.

2.2 Refactoring

Refactoring is the process of changing a software system in such a way it does not alter the external behavior of its code, yet improves its internal structure [7]. Refactoring is useful to make maintenance tasks easier by changing a system so that it will be more extensible and modifiable. The steps for refactoring a piece of code are the following [21]:

1. Identify the code fragments where the system should be refactored.
2. Determine which refactorings should be applied to the code fragments.
3. Check whether the candidate refactoring preserves the external behavior of the code.
4. Apply the refactoring itself.
5. Assess the effects of the refactoring on quality characteristics of the software system (e.g., complexity, maintainability) or the process (e.g., productivity, cost, effort).
6. Maintain consistency between the refactored code and related software artifacts (e.g., documentation, architecture, requirements, tests).

Step 1 is usually performed by looking at code smells in the code. Then, in step 2, the developer should analyze the problem (i.e., a particular code smell) to determine the refactoring (or group of refactorings) that should be applied to improve the code structure. When the developer selects the refactorings, she needs to check first that application behavior is preserved (step 3). This property can be guaranteed by some refactoring tools, such as the one provided with the Eclipse IDE4. However, stronger checks would require performing regression testing both before and after the refactoring. If a refactoring tool is used, the refactoring (step 4) is usually applied automatically. If no tool is used, then the step is not trivial because it usually requires technical skills from the developer. Once the refactoring is applied, the developer can assess its effects (step 5). For example, she can look at different software metrics to measure attributes of coupling, complexity, cohesion or other aspects of the system, before and after the refactoring. Finally, in step 6 the developer should update the documentation, tests and other related artifacts.

A catalog of smells usually proposes, for each smell, a refactoring or a group of refactorings that can help to solve the problem. In the case of the BM, Fowler presents a refactoring called Extract Method claiming that this refactoring is sufficient for fixing the smell [7]. Lanza & Marinescu also suggest the application of the same refactoring as a possible fix for BMs [15]. Extract Method consists of extracting a fragment of code from a method into a new method. In a BM, the intention behind this refactoring is to extract fragments of code that implement a piece of functionality that distinguishes itself from the rest of the BM. Coming back to the example of BM in Figure 2, part of

---

4https://www.eclipse.org
the functionality of updateDiagram sets a time series according to the chart type (lines 13-25). Thus, that method fragment can be taken out to a new method using the Extract Method refactoring.

After refactoring a BM, a more legible source code is expected. At the same time, fixing a BM can help to fix other kinds of smells that affect class complexity and coupling, such as Brain Class, Intensive Coupling, Feature Envy or Disperse Coupling [15].

Some refactoring tools (like the one provided by the Eclipse IDE) are available for performing the Extract Method refactoring on Java projects. However, in spite of the automation provided by such tools, determining the way in which a BM should be refactored is not always straightforward. To successfully fix a BM, different conditions must be observed, namely:

- The code fragment to be extracted cannot contain assignments to variables that are later used in the method flow. This is a restriction of some languages, like Java, in which a method cannot return more than a single variable.
- The code fragment to be extracted cannot contain sentences of type return, break, or continue.
- It must be ensured that the BM is completely removed after applying one or more times the Extract Method refactoring.

In addition to the technical challenges imposed by the conditions above, another challenge is that the developer has to assess the pros and cons of refactoring a given smell, before effectively fixing it. This analysis can rely on the impact that the refactoring has on different software metrics (of the sub-system under consideration). For instance, in a BM, she might look at the same complexity and coupling metrics used to detect it. Since most refactoring tools do not support this kind of assessment, a practical strategy consists of executing the refactoring, so that the developer is able to analyze its effects on a concrete basis. If the effects are not as expected, the developer should undo the changes introduced by the refactoring. This process of refactoring the code, assessing the improvements (if any), and undoing the changes if the results are not satisfactory is often time-consuming for the developer. The problem gets worse when several candidate refactorings need to be evaluated by the developer.

2.3 Refactoring Brain Method as a Search Problem

The automated refactoring of a software program can be seen as a combinatorial optimization problem [10, 13], in which an algorithm explores alternative transformations for a given input program and there is an objective function for assessing the quality of the program solutions resulting from the transformations. In this kind of problems, three elements need to be defined, namely: (i) the solution representation, (ii) the fitness (or objective) function, and (iii) the available transformations. Then, a variety of search techniques can be applied, including simulated annealing [14], but also other techniques such as genetic algorithms, hill climbing, or particle swarm optimization. In the following, we present the problem formulation for the refactoring of BMs using Extract Method as the main transformation. We follow a direct modeling approach [10, 11], in which each state of the search space is a software program block and the refactorings correspond to links between pairs of states.

Let a state be a pair $s_i = \{M^k_x, < m_1, m_2, ..., m_k >\}$, in which $M_x$ is a method defined in some class $X$ and is also an instance of BM according to the rule of Equation 1. If $M_x$ is a BM, we can apply one or more Extract Method refactorings to the method. An initial state is represented by $s_0 = \{M^0_x, \emptyset\}$, because no method extractions have been yet applied. Let $M^k_x$ be the resulting method after performing a sequence of $k$ extractions of code fragments from $M_x$. These successive extractions lead to new methods (within class $X$) denoted by the ordered set $< m_1, m_2, ..., m_k >$. Figure 3 schematizes what the search graph looks like with state $s_0 = \{updateDiagram\}$, as presented in
Figure 2. State $s_1 = \{\text{updateDiagram}^1_x, < \text{extracted}_0\textunderscore\text{updateDiagram} > \}$ models the application of an Extract Method to an if block of $s_0$, as illustrated in Figure 10. Similarly, other Extract Method refactorings are applicable to $s_0$, which generate the states $s_0$ to $s_9$ in Figure 2. A candidate sequence of refactorings that solves the initial BM is given by states $s_0$, $s_9$, $s_{10}$ and $s_{11}$ (see Figure 10).

A state $s_i$ is said to be feasible if it corresponds to a compilable program (i.e., a valid program instance). The usual conditions for applying refactorings also need to be ensured. In particular, we require all method extractions $< m_1, m_2, \ldots, m_k >$ to be feasible with respect to $M^k_x$, and also require that the functional behavior of $M^{i}_x$ is equivalent to that of $\{M^k_x, < m_1, m_2, \ldots, m_k > \}$. Usually, checking that a given transformation is feasible is accomplished via static program analysis techniques. Along this line, the transformations in our work are based on the Extract Method refactoring. Nonetheless, different instances of this refactoring are possible between a pair of consecutive states $s_i$ and $s_j$. More formally, an extract method is a transformation $em : \{M^k_x,$ $<
\[ m_1, m_2, ..., m_k \rightarrow \{ M_{k+1}^k, < m_1, m_2, ..., m_k, m_{k+1} > \}. \] We only allow transformations in which both the source and target states are feasible\(^5\).

Each state \( s_i \) is assessed with a fitness function. Our fitness function \( f_{BM} \) (to be minimized) essentially counts the number of BMs in a program state using Equation 1 for \( M_k^k \) and for every \( m_j \) in \( < m_1, m_2, ..., m_k > \). If \( f_{BM}(s_i) = 0 \) then \( s_i \) is a final or satisfying state (green circles in Figure 3), because the initial BM has been removed. From the developer’s perspective, final states are candidate solutions for the BM problem. A state \( s_i \) in which \( f_{BM}(s_i) > 0 \) means that some of the conditions of Equation 1 are met, and \( s_i \) is called a not-satisfying state (yellow circles in Figure 3). Departing from an initial state given as input, in which \( f_{BM}(s_0) = 1 \), a search algorithm traverses (and evaluates) several states until reaching a satisfying state. Many times, the application of a single transformation directly produces a satisfying state. In other scenarios, the first transformation can lead to a not-satisfying state, which after applying another transformation will become a satisfying state.

An empirical observation, based on our experiments with BMs and Extract Method refactorings, is that the search space is often a dense graph but with a relatively small diameter. By density, we mean that each vertex of the graph is connected to many other vertices (neighbors). The diameter refers to the greatest distance between any pair of vertices, or alternatively, the greatest length of any of the shortest paths between vertices. In our case, we are interested in the shortest paths between \( s_0 \) and any final state, as indicated by the arrows in Figure 3. For instance, in the search graphs of project SportsTracker (23K LOC), we found an average density of 16 and an average diameter of 2-3 when considering satisfying states. Furthermore, we noticed a high chance of finding a satisfying solution (for a BM) with one single refactoring, then a slightly smaller chance of finding a satisfying solution with a sequence of two refactorings, then a much smaller chance of having a satisfying solution with three transformations, and so forth. This means that many Extract Method-based solutions for a given BM are actually direct neighbors in the graph, and the number of satisfying solutions after more refactorings are being applied seems to decrease. This kind of search space makes it suitable for global optimization techniques such as simulated annealing [13, 14], which aim at finding (approximate) global optima rather than precise local optima. Note that our optimal solutions are approximate, because the absence of BMs is assessed with a metrics-based approach (Equation 1).

### 3 RELATED WORK

A number of approaches have been proposed to implement the Extract Method refactoring [7, 8, 25, 27]. These approaches usually check whether the statements to be extracted preserve the behavior, but it is up to the developer to determine the specific statements that should be extracted to fix the smell. For example, the Eclipse IDE includes a tool [8] that automates more than 25 refactorings (including Extract Method). This tool analyzes whether the extraction will modify the system behavior, and if so, it prevents the developer from applying the refactoring. Additionally, the tool provides an API for invoking the refactorings programmatically, so as to extend the tool or invoke it from other tools.

There are several approaches that recommend refactorings of complex methods using the Extract Method refactoring. These approaches do not only apply the refactoring, but also suggest to the developer candidate statements that should be extracted. For example, Tsantalis and Chatzigeorgiou [41] presents an approach to identify refactoring opportunities in long methods that is materialized

---

\(^5\)This limitation is a bit restrictive for the search space, and it comes from our reliance on the Eclipse IDE for the refactoring operations. While a transformation is being applied (by Eclipse) a program might go through intermediate states that may be not valid, but such states are not visible from Bandago.

ACM Transactions on Software Engineering and Methodology, Vol. 9, No. 4, Article 39. Publication date: March 2010.
in a tool called JDeodorant. The approach uses program slicing \cite{3} to detect blocks of source code that could be extracted using the *Extract Method* refactoring. Specifically, the Tsantalis & Chatzigeorgiou approach uses two strategies, called *complete computation slices* and *object state slices*, for detecting code fragments that process a given variable. Once JDeodorant detects a refactoring opportunity, the developer must select those refactorings that she wants to apply. Since the approach proposes individual refactoring opportunities, it does not guarantee that the smell will be fixed after applying a suggestion of JDeodorant (or even all the refactorings suggested by the tool for the smell). That is, the approach produces not-satisfying solutions.

Yang et al. \cite{51} proposes an approach called AutoMeD that identifies and extracts blocks of code in long methods. The approach identifies statements that could be extracted without changing the behavior of the source code, and extracts the statements using the *Extract Method* refactoring. The selection of statements to be extracted is based on finding blocks that are separated by blank lines. It is assumed that those blocks realize a specific functionality that should be extracted as a single method. After detecting the extractable blocks, AutoMeD ranks them based on the coupling and number of lines of the new method. At last, the developer must decide whether to apply the refactoring. Similar to the work of Tsantalis & Chatzigeorgiou, this approach reduces a complex method but it does not guarantee that the refactoring is a satisfying solution (i.e. the smell is not fixed). That is, the refactoring proposed could be a not-satisfying solution. Another observation is that the prototype implementing the approach is not available for downloading.

Silva et al. \cite{35} look for opportunities of extracting fragments of a method using the *Extract Method* refactoring. This approach does not refactor the code smell but rather proposes code extractions for reducing the length of a long method. That is, the approach suggests not-satisfying solutions. The approach only suggests candidate blocks to be extracted, and those blocks should meet three conditions. First, the statements composing the block must be consecutive (i.e. with no blank spaces between them). Second, the extraction of the block must not change the system behavior. To ensure this condition, the approach uses the refactoring API provided by Eclipse \cite{8}. Third, the block must have at least three lines of code. The eligible blocks are suggested to the developer by ranking them based on the coupling of the new method with respect to the original one.

Yamashita and Moonen \cite{50} performed an empirical investigation about the occurrences of 12 code smells, and how their interactions could relate to maintenance problems. The authors looked at patterns of co-located code smells using Principal Component Analysis (PCA), and their findings revealed that maintenance issues are related to the presence of several code smells in the same class. The authors suggest that certain smells should be prioritized for refactoring. However, the automation of the refactorings is not analyzed.

O’Keeffe and Cinnéide \cite{32} conducted an empirical study about the automation of refactorings using three techniques: simulated annealing, genetic algorithms, and multiple ascent hill climbing. However, this study does not analyze BMs nor use the *Extract Method* refactoring. Instead, it focuses its refactorings on improving the cohesion and coupling of classes. Thus, the refactorings used by O’Keeffe and Cinnéide do not cover the internals of a method.

The work presented by Meananeatra et al. \cite{20} is, to the best of our knowledge, the only approach that automates the refactoring of a complex method. Specifically, it identifies and refactors the Long Method smell \cite{7}. The approach uses five of Fowler’s refactorings to fix long method instances (including *Extract Method*). In order to analyze the “goodness” of the refactorings, the approach employs software metrics to measure the cohesion, complexity, and lines of code of the methods. While this work presents a small experiment to demonstrate the applicability of the approach, no tool seems to be implemented to support the approach.
The BM smell is usually fixed by splitting the method in two, or more, smaller and more cohesive methods than the original method. To this end, the Extract Method refactoring is generally used to extract fragments of code into new methods. However, selecting the code fragments to be extracted can be challenging for several reasons [26], namely: it is difficult to identify sentences being cohesively related, there can be dependencies of the selected fragment with the rest of the code, or the method behavior must be preserved, among others. For these reasons, we argue that developers should be assisted by automated search techniques for fixing BM instances. Furthermore, we believe that enabling a developer to analyze a set of alternative refactoring solutions (before she makes the final decision) is a key tool feature. In this context, we propose an approach called Bandago that removes BMs by extracting code fragments (into new methods) while considering aspects of legibility and extensibility of the resulting code. Bandago is implemented on top of JSpIRIT, an Eclipse plug-in for identifying and prioritizing code smells in Java. JSpIRIT supports the identification of BMs (among other smells) following the metrics-based detection strategy of Equation 1 [15].

Once that a developer had analyzed a BM and that she decided that it must be refactored, she can use Bandago to automatically fix it. Bandago follows an iterative cycle of work, which takes as input the source code of a BM and produces as output alternative solutions (refactorings) for the smell. Then, the developer can choose any of those solutions, based on information given by Bandago and on her own experience. Internally, Bandago performs a heuristic search using a simulated annealing algorithm [36]. The approach involves five main activities (Figure 4), as follows.

1. Obtaining statements of a method: given a BM instance, this activity provides a set of statements that can be extracted from the method. The statements are filtered and grouped according to its kind (e.g., if statement, while statement, etc).

2. Obtaining a candidate statement: the statements just obtained are ranked with the goal of choosing a concrete statement to be extracted. The ranking is based on different criteria, called operators, such as length or complexity of the statement.

3. Extracting candidate statement: the chosen statement is extracted into a new method via an Extract Method refactoring. In order to generate several solutions for a BM, Bandago applies the refactoring “virtually” [40, 43]. A virtual refactoring means that it is accomplished in a program model that resides in memory, instead of applying it to the (Java) source code. Once a solution is generated and assessed, the (in-memory) changes introduced by the refactoring are discarded. A refactoring is only applied to the source code when the developer picks a concrete solution proposed by Bandago.

4. Checking for existence of BMs: after applying a refactoring, Bandago checks that both the affected method and the new method (generated by the extracted fragment) are not BMs anymore. To do so, Bandago applies the rules of Equation 1 for determining if the current solution is satisfying. In case any method is still a BM, additional refactorings need to be explored (feedback arrow to activity 1).

5. Assessing generated solution: when a solution for a BM is found, a set of metrics is computed on that solution in order to assess its goodness. These metrics help the developer to choose a “good” solution from the proposed solutions.

After performing activity 5, the approach iterates n times, from activity 1 to 5, in order to return a predefined set of solutions to the developer.

In the following sub-sections, we provide details of each activity of the approach.
4.1 Simulated Annealing

Simulated annealing is a search meta-heuristic algorithm based on the heating and cooling of materials in metallurgy [14]. Simulated annealing has been used to solve different optimization problems with multiple local optima [1, 14], and in particular, it has been used to search for refactorings as well [13, 23, 31]. In this context, a local optimum is the best solution found by the algorithm, but it is not necessarily the best solution to the problem, which is called a global optimum. Simulated annealing follows an iterative method that starts by considering a state $I$. In each iteration, the algorithm generates a neighbor state $J$ and evaluates whether $J$ is better than $I$. When $J$ is better than $I$, the algorithm changes its current state to $J$. When $J$ is worse than $I$, the algorithm probabilistically decides if it should change the current state for $J$. This probability is based on an evaluation function of each state, $I$ and $J$, and a variable called temperature that controls the number of iterations. As the simulation progresses, the temperature gradually decreases according to a cooling factor. By means of these variables (temperature and cooling factor) the algorithm decreases the probability of changing to a worse state, as iterations are completed. The algorithm finishes its execution when it finds a solution or when the number of iterations exceeds a pre-determined threshold (so as to avoid an infinite cycle when there is no solution).

Based on the problem formulation of Section 2.3, we use simulated annealing to search for refactorings for BM because it allows us to explore a range of different solutions to fix a BM using only one transformation. Moreover, the algorithm can (probabilistically) accept not-satisfying solutions with the goal of finding further solutions via a (small) sequence of transformations.
this end, we adapted the basic simulated annealing algorithm to deal with BM instances and Extract Method refactorings, as described in Listing 1. The pseudocode derives from the different activities of our approach (see Figure 4). Note that the fitness function of our algorithm (also referred to as energy function in simulated annealing) looks for solutions in which the number of BMs is minimized (i.e., there is zero BMs in the current state). After obtaining the source code of a BM and setting up the initial temperature, the algorithm gets the statements that can be extracted (activity 1). In terms of simulated annealing, these statements are seen as “neighbors” of the initial state. The statements are filtered to choose a candidate statement (activity 2). Then, the statement is extracted (at the model level) and the algorithm evaluates whether the resulting code minimizes the fitness function (activities 3 and 4). If the result of the extraction is not satisfying, the temperature...
function determines if the solution should be accepted (feedback arrow to activity 1) and drives a new round of search departing from that state. After a fixed number of iterations, a particular solution is returned. Finally, once a number of satisfying solutions has been collected, the algorithm ends and returns those solutions (activity 5).

4.2 Activity 1: Obtaining Statements of a Method

Once a BM has been identified by JSpIRIT, the developer resorts to Bandago to fix the smell. For example, Figure 5 shows the JSpIRIT GUI that lists all the smells detected. Using this GUI, the developer can access Bandago and target, for instance, the BM updateDiagram (presented in Section 2) for analyzing refactoring proposals. Then, Bandago gets the source code of the BM and its constituting statements.

The process of obtaining the statements scans all the lines of code of a method and categorizes them. For example, the kind of statements can be assignment, for, if, expression, among others\(^7\). Figure 6 shows a fragment of the BM updateDiagram. A statement can be composed by other kind of statements. For example, the for statement is composed of statements assignment, expression, and if. After categorizing the statements, they are filtered using one of the following criteria:

- **IfStatementFilter**: it only returns if statements.
- **WhileStatementFilter**: it only returns while statements.
- **ForStatementFilter**: it only returns for statements.

\(^7\)The types of statements considered were the following: Assert, Break, Continue, Do, Empty, Block, Expression, For, If, Labeled, Return, Switch, Synchronized, Throw, Try, Constructor_invocation, Switch_case, Type_Declaration, Variable_declaration, and While.

---

**Fig. 5.** JSpIRIT view: identification of code smells

**Fig. 6.** For statement of updateDiagram BM
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Prioritizes the statements with the highest number of lines of code</td>
</tr>
<tr>
<td>NESTING</td>
<td>Prioritizes the statements with the highest level of nesting</td>
</tr>
<tr>
<td>CYCLO</td>
<td>Prioritizes the statements with the highest cyclomatic complexity</td>
</tr>
<tr>
<td>NOAV</td>
<td>Prioritizes the statements with the highest number of accessed variables</td>
</tr>
<tr>
<td>Random</td>
<td>Randomly picks one of the previous 4 operators</td>
</tr>
</tbody>
</table>

Table 1. Operators for statements

- **TryCatchStatementFilter**: it only returns try-catch statements.
- **SwitchCaseStatementFilter**: it only returns switch-case statements.
- **GroupStatementFilter**: it returns a group of statements grouped by comments or line breaks. The rationale behind this filter is to detect a common developers’ practice that involves grouping code fragments that are semantically related [7].
- **AllStatementFilter**: it returns all the statements (no filtering is performed).

In order to avoid getting too many statements, the developer can configure the minimum number of lines of code of the statements to be filtered. Additionally, Bandago can be configured by the developer to use only a specific filter or a combination of them. In the latter case, the simulated annealing algorithm will randomly select the particular filter to be used at each iteration.

For example, Figure 2 shows all the possible blocks identified by Bandago for extraction when the **AllStatementFilter** is applied with a minimum of 3 lines of code. However, if this minimum is set to 5, then the blocks between lines 4-7, 46-49, 52-54, 105-108, 114-117, 130-132, and 136-138 will be discarded.

In general, the output of this activity will be all the filtered statements of a given method. Depending on the kind of filtering, the output statements might contain all the blocks identified or just a subset of them. For example, in the case of using **IfStatementFilter** only the if statements (with a predefined number of lines) will be identified.

### 4.3 Activity 2: Obtaining a Candidate Statement

The goal of this activity is to rank the statements that were filtered in the previous activity. To do so, five kinds of operators are defined (Table 1). Each operator ranks statements that can be extracted without altering the behavior of the method. We selected these particular operators because they are based on the same metrics used by Lanza & Marinescu [15] to identify a BM (see Section 5.6 for an empirical analysis of the operators). In this way, we aim at reducing the value of at least one of those metrics in the method, with the hope of directing the search towards fixing the smell. In other words, the operators try to generate partially-satisfying solutions. The random operator was chosen to generate solutions that are not driven by a specific metric, but rather derive from any of the other four operators equally. Overall, our intention is to provide solutions that progressively improve (i.e. reduce) all the metrics that define a BM.

For example, let us consider the scenario shown in Figure 7. In this case, three statements of method `updateDiagram` have been filtered using **GroupStatementFilter** and then ranked using the **LOC** operator. Since **Group Statement 1** has more lines of code than the other statements, it is ranked in the first position. If we take into account the whole source code of this BM (Figure 2) with an **AllStatementFilter**, the if statement (lines 94-126) will be ranked first when using the **LOC** operator. However, the if-else statement (lines 14-25) will be alternatively ranked first if the CYCLO, NOAV or NESTING operator is used.
In order to check that the candidate statement can be extracted without altering the method behavior, Bandago relies on the refactoring API provided by Eclipse [8]. The API implements program slicing [4] to assure that the execution flow of a method does not change after extracting a part to a new method. Since this analysis is complex in terms of CPU processing, Bandago only performs the check on candidate statements (i.e., statements ranked first in activity 2), and ignores all the remaining statements found in a method (activity 1). If a candidate statement cannot be extracted safely, it is discarded and Bandago selects the next statement in the ranking. For example, some blocks containing `return` or `break` statements cannot be extracted. As shown in Figure 8, the if block could not be extracted because the block could not be removed from the method using `Extract Method`. Another example is the switch-case statement of Figure 2 (lines 56-80). Although this block is effectively identified, we are unable to extract it using `Extract Method` because the block modifies local variables (`dateFormatTooltip`, `dateFormatAxis`, and `dateTickUnit`) being used in the subsequent code. Moreover, the blocks of statements that could be extracted from the code in Figure 2 are all but the groups of lines 4-7, 52-54, and the mentioned switch-case statement. Although Bandago currently attempts to fix BMs by applying solely `Extract Method`, other kinds of refactorings are also possible, and this aspect will be analyzed in future work.

In summary, the output of this activity will be a candidate statement that can be extracted using the `Extract Method` refactoring without changing the method behavior.

### 4.4 Activity 3: Extracting Candidate Statement

This activity extracts the candidate statement (as determined in the previous activity) by means of a particular `Extract Method` refactoring. Bandago applies the refactoring automatically and assigns a default name to the new method. Since the applicability of the refactoring was assured in activity 2, the refactoring is always possible in this step. For example, Figure 9 shows the extraction of one of the if statements of Figure 2 (lines 14-25) into a new method called `extracted_2(TimeTableXYDataset,List)`.
private void updateDiagram () {
    updateSelectionControlsState ();
    // get selected time range and value type and its name to display
    TimeRangeType timeType = getCurrentTimeRangeType ();
    ValueType vType = getCurrentValueType ();
    String vTypeName = getCurrentValueTypeName ();
    int year = getSelectedYear();

    // create a table of all time series (graphs) and the appropriate colors
    TimeTableXYDataset dataset = new TimeTableXYDataset ();
    List<Color> lGraphColors = new ArrayList<> ();

    // setup TimeSeries in the diagram (done in different ways for all the value types)
    if (getCurrentValueType () == ValueType.SPORTSUBTYPE) {
        setupSportSubTypeDiagram(dataset, lGraphColors);
    } else if (getCurrentValueType () == ValueType.EQUIPMENT) {
        setupEquipmentDiagram(dataset, lGraphColors);
    } else if (getCurrentValueType () == ValueType.WEIGHT) {
        setupWeightDiagram(dataset, lGraphColors);
    } else {
        setupExerciseDiagram(dataset, lGraphColors);
    }
}

private void extracted_2(TimeTableXYDataset dataset, List<Color> lGraphColors) {
    if (getCurrentValueType () == ValueType.SPORTSUBTYPE) {
        setupSportSubTypeDiagram(dataset, lGraphColors);
    } else if (getCurrentValueType () == ValueType.EQUIPMENT) {
        setupEquipmentDiagram(dataset, lGraphColors);
    } else if (getCurrentValueType () == ValueType.WEIGHT) {
        setupWeightDiagram(dataset, lGraphColors);
    } else {
        setupExerciseDiagram(dataset, lGraphColors);
    }
}

Fig. 9. Example of a statement extraction

4.5 Activity 4: Checking for Existence of Brain Methods

After extracting the candidate statement, the simulated annealing algorithm generates a new state (cf. Listing 1). Then, the algorithm checks whether the new state is a satisfying solution. That is, both the extracted method and the method from which the code fragment was extracted (i.e., the original BM) are not BMs. If this is a satisfying solution, it is saved in a solutions list. If not, the current state can still be replaced by the new state based on a random probability and the temperature (cf. Listing 1). This probabilistic rule allows simulated annealing to avoid local optima and possibly reach a global optimum by exploring different regions of the search space. Since the temperature decreases with each iteration, the probability of accepting a not satisfying state as the new state diminishes over time. When the current state is still not satisfying, Bandago restarts the process in activity 1 with the affected method as input. For example, let us consider that, in the scenario of Figure 9, method extracted_2 is not a BM, but updatedDiagram() is still a BM. In this case, Bandago will try to extract another code fragment from updatedDiagram(). The iterations of the algorithm will continue until no more BMs are found. As it was stated in Section 4.1, to avoid an infinite cycle when there is no solution, the approach stops cycling by predetermining a maximum number of iterations (see Listing 1).
Figure 10 shows a solution proposed by Bandago for the `updatedDiagram()` BM (Figure 2) using the `IfStatementFilter` and the Random operator. To obtain this solution, Bandago initially selected the if sentence with the highest LOC value (Activity 2). This case corresponds to the if sentence in lines 94-126 (Figure 2) having a LOC value of 26. Since this code fragment can be refactored using Extract Method, Bandago extracted it into a new method called `extracted_0_updateDiagram`. 

ACM Transactions on Software Engineering and Methodology, Vol. 9, No. 4, Article 39. Publication date: March 2010.
Then, Bandago checked that neither extracted_0_updateDiagram nor updatedDiagram are BMs. In the case of extracted_0_updateDiagram, it is not a BM because its LOC and NESTING values are under the respective thresholds (LOC=28 and NESTING=2). However, in the case of updatedDiagram, its values reveal that it is still a BM (LOC=86, CYCLO=11, NESTING=3, and NOAV=106). For this reason, Bandago goes back to Activity 1 in which the remaining if statements are identified. The Random operator prioritizes the if sentence in lines 141-152 based on its NOAV value (23). After extracting this code fragment into extracted_1_updateDiagram, Bandago checked again for remaining BMs. The new method extracted_1_updateDiagram is not a BM because its LOC, CYCLO, and NESTING values are under the thresholds (LOC=14, CYCLO=2, and NESTING=1). Nonetheless, method updatedDiagram is still a BM (LOC=75, CYCLO=10, NESTING=3, and NOAV=83). After going back to Activity 1, Bandago identifies the remaining if statements. As part of Activity 2, the if sentence in lines 14-25 is selected, because it has the highest LOC (12), and it is extracted into extracted_2_updateDiagram (Activity 3). At last, Bandago found that neither extracted_2_updateDiagram nor updatedDiagram are a BM. For extracted_2_updateDiagram, its LOC value is under the threshold (LOC=14). Regarding updatedDiagram, it is not detected as a BM because its NESTING value (1) does not exceed the threshold. At this point, there are no more BMs derived from updatedDiagram().

4.6 Activity 5: Assessing Generated Solution(s)

Once the algorithm finds a set of solutions, this activity computes a set of metrics (at the method level), so that the developer can compare the original BM against the extracted methods proposed by each solution. Additionally, Bandago presents the average of each metric for a given solution. Specifically, Bandago calculates the four metrics used to identify a BM (Nesting level, LOC, cyclomatic complexity, and NOAV) for every method of a solution, and also informs the number of extracted methods per solution.

Figure 11 shows a Bandago GUI in which 4 solutions are presented for the updateDiagram BM. Using this view, the developer can choose the most convenient solution for her context, based on the metrics values and also on the pre-visualization of the source code after applying the refactoring. For example, the fourth solution is the one shown in Figure 10. In the case of the method extracted_0_updateDiagram, the view shows that the nesting level is 2, the number of lines of code is 28, its cyclomatic complexity is 4, and the number of accessed variables is 48.

5 EVALUATION

In this section, we report on the results of applying Bandago to 10 Java open-source applications of different sizes and with varying amounts of BMs. We selected these applications from the Qualitas Corpus [38], an on-line repository for empirical studies. Each selected application must meet the
Assessing the Refactoring of Brain Methods

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th>#Lines of code</th>
<th>#Classes</th>
<th>#Brain Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Columba</td>
<td>1.4</td>
<td>26600</td>
<td>436</td>
<td>5</td>
</tr>
<tr>
<td>JGraphT</td>
<td>0.9.0</td>
<td>14180</td>
<td>218</td>
<td>4</td>
</tr>
<tr>
<td>SportTracker</td>
<td>5.7</td>
<td>5200</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>Cayanne</td>
<td>4.0</td>
<td>45000</td>
<td>533</td>
<td>16</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>6.4.1</td>
<td>60000</td>
<td>399</td>
<td>52</td>
</tr>
<tr>
<td>Jena</td>
<td>2.12.1</td>
<td>54410</td>
<td>697</td>
<td>22</td>
</tr>
<tr>
<td>JGroups</td>
<td>3.4.8</td>
<td>76570</td>
<td>644</td>
<td>37</td>
</tr>
<tr>
<td>Quartz</td>
<td>2.1.7</td>
<td>26810</td>
<td>176</td>
<td>19</td>
</tr>
<tr>
<td>Roller</td>
<td>5.1.2</td>
<td>47460</td>
<td>452</td>
<td>43</td>
</tr>
<tr>
<td>Squirrel</td>
<td>3.6.0</td>
<td>79070</td>
<td>879</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 2. Applications used in the evaluation

following conditions: i) it must be open-source, ii) it must be written in Java, iii) it must contain at least one BM. Table 2 summarizes information of the applications used in our study, including the software versions so that the experiments can be reproduced by others. Additionally, both Bandago and JSpIRIT are available for download.

For evaluating our approach, we formulated the following research questions:

- RQ#1: What is the percentage of BMs automatically refactored by Bandago?
- RQ#2: Does the fix of BMs reduce complexity at the class level?
- RQ#3: Does Bandago fix more BMs than JDeodorant?
- RQ#4: Do developers find the solutions proposed by Bandago useful?
- RQ#5: Does the fix of BMs help to deal with other code smells?
- RQ#6: What operator (from Activity 2) produces the best results?
- RQ#7: What is the average time to generate solutions for a BM?

The focus of RQ#1 is to determine how many BMs can be fixed (in average) when using our approach based on the Extract Method refactoring. Moreover, in RQ#2 we analyze whether the complexity of the classes hosting BMs is affected, after fixing the smells. Reducing class complexity is an important factor, because complex classes decrease system modifiability and understandability. Later, we compare Bandago against JDeodorant, which is a well-known refactoring tool from the literature. In RQ#3, we analyze the two tools for the same set of BMs. In RQ#4, we conducted a study with industrial developers to analyze their perception about the complexity and legibility of the source code after applying solutions proposed by Bandago and JDeodorant. In RQ#5, we investigate the effects of fixing BMs on other code smells. Whereas in RQ#1 we analyze the performance of Bandago in general, this performance is dependent on the choice of the right operator. In RQ#6, we assess each operator separately. In addition, we measured the time spent by Bandago to find a solution for a BM, which is an indicator of usability for developers.

To collect the data for the experiments, for each application we identified its BMs using JSpIRIT and then used Bandago to generate different solutions to fix those smells. Figure 12 depicts the experimental procedure to collect the data. For each application, we ran Bandago 5 times, each time with a different operator. We configured Bandago to get 3 solutions for each BM identified by JSpIRIT. In order to simulate the situation in which a developer chooses a solution given by Bandago, we selected any of those solutions according to a metric improvement criterion (the same metrics used in the operators) and made Bandago apply the refactoring to the application.

---

8https://goo.gl/SHi2UB

ACM Transactions on Software Engineering and Methodology, Vol. 9, No. 4, Article 39. Publication date: March 2010.
We repeated this action 5 times, one for each available metric. As a result, Bandago was run 25 times per application, and obtained 25 versions of the refactored application. Note that, since our intention is to analyze the performance of Bandago, in this evaluation we fixed all BMs for which the tool offers a solution. However, in a typical scenario the developer should analyze the BMs identified and decide according to her expertise which of them should be fixed.

Regarding the parameters initialTemperature and coolDownFactor in the simulated annealing algorithm, they were empirically determined and set to 10 and 0.9 respectively. We set the minimum number of lines for an extracted method to 3. This setting follows from recommendations by Kirkpatrick [14] and Nourani [29].

5.1 RQ#1: What is the percentage of BMs automatically refactored by Bandago?

The goal behind this research question is to assess the performance of Bandago with respect to BMs. To this end, we compared the number of BMs in the original version of an application against the number of BMs in the refactored version of the same application, after executing the solutions recommended by Bandago.
As shown in Table 3, an average of 62.88% of the BMs can be automatically solved by Bandago (with a standard deviation of 16.15%). Furthermore, most of the projects (8 out of 10) had a percentage higher than 50% of BMs automatically fixed. Note that the percentages reported in the table summarizes the 25 executions of Bandago, for each application. For example, Columba only has 5 BMs, but as it was analyzed 25 times, Table 3 reports 125 BMs analyzed. The reason for this counting is that Bandago could find solutions for a BM instance with a given operator, but could not find any solution for the same instance using another operator.

Figure 13 shows the percentages of BMs refactored by Bandago by operator type. We found that similar results were obtained (for the same application), with some exceptions in Columba. The Random operator yielded the best results for every application. The small differences among operators (for a given application) are due to the order in which the statements are extracted. When a statement is extracted, it can enable (or prevent) the extraction of other statements in future refactorings. In this context, the Random operator permits to fix of a larger number of BMs than other operators, because the statements to be extracted are selected in any order.

Regarding those BMs for which Bandago could not generate a solution, we observed that they often contained break and return sentences. Since Bandago only uses the Extract Method refactoring, these types of sentences inhibit the extraction of the corresponding statements. In fact, in the case of Quartz that had the lowest percentage of refactored BMs (36.84%), there was a high number of auto-generated classes including many break and return sentences.

In order to claim that the percentages of BMs refactored by Bandago are statistically significant, we analyzed them using a one sample t-test [49]. Since Bandago refactored more than 50% of the BM in 8 out of 10 projects, we expect Bandago to solve at least one of every two problems. Formally:

- $H_{10}$: Bandago fixes less than 50% of the BMs of a system.
- $H_{11}$: Bandago fixes 50% or more of the BMs of a system.

After running the test, we can reject the null hypothesis $H_{10}$ with a one tailed test with a probability of error (or significance level) $\alpha = 0.05$ (i.e. there is a 5% chance of wrongly rejecting $H_{10}$) and a
p-value of 0.02022. In this way, we can answer RQ#1 by accepting $H_1$: Bandago fixes 50% or more of the BMs of a system.

5.2 RQ#2: Does the fix of BMs reduce complexity?

In this research question, we aim at analyzing if the refactorings proposed by Bandago reduce the complexity of the refactored methods/classes. A reduction in complexity is important as it means improved legibility and modifiability of the code. In order to assess complexity aspects, we computed a predefined set of metrics per application, before and after the refactorings. In particular, we used 5 well-known software metrics [2, 12, 37] related to system complexity. Given the nature of a BM, these metrics are focused on class internals. The metrics were the following:

- **MLOC (Method Lines of Code):** it counts the number of lines of code of a method (without considering blank lines nor comments). The higher the number of lines, the more complex the method is.

- **NOM (Number of Methods):** the number of methods of a class is usually associated with its complexity. The higher the number of methods, the more complex the class is. However, for correctly interpreting this metric it should be analyzed together with MLOC, because a class with a low NOM but with high MLOC is still a complex class.

- **CYCLO (Cyclomatic Complexity):** this metric indicates the complexity of a method, based on the number of linearly independent paths of the code. The lower the cyclomatic complexity of a method, the better the modifiability and legibility of such a method.

- **NBD (Nested Block Depth):** this metric indicates the complexity of a method based on the level of nesting of a method. High levels of nesting are usually associated with poor legibility, and thus negatively impact on modifiability.

- **LCOM (Lack of Cohesion of Methods):** it measures the correlation between the methods of a class and its instance variables. A low correlation indicates a low cohesion which increases class complexity.

The metrics were computed using the Eclipse plugin Metrics9 for all classes affected by BMs (before and after refactoring the applications). Figure 14 shows the improvement of the metrics across applications for our five operators. That is, values above zero mean an improvement in the metric.

The number of methods per class (NOM) increased in 20.53% on average (with a standard deviation of 8.86%). Thus, the improvement of this metric is negative (see Figure 14). In some particular applications, NOM values increased considerably. For example, using the Random operator, the NOM of SportsTracker and CheckStyle was higher than 30% (Figure 14e). This increment was expected since the Extract Method refactoring generates new methods in the class. However, it does not necessarily mean that complexity was negatively affected because the extracted methods are smaller than the original one.

Regarding CYCLO, we obtained improvements in all applications regardless of the operator. Specifically, the cyclomatic complexity of the classes improved a 10.16% on average (with a standard deviation of 4.07%). The improvements in CYCLO are due to the fact that our refactoring of BMs distributes the method logic among the extracted methods. Also, in some cases the complexity might decrease because the calls to the extracted methods generate a lower number of execution flows in the original method.

The NBD metric slightly deteriorated in most applications. On average, the nesting level increased a 1.39% (with a standard deviation of 3.16%). The reason for this behavior is that the Extract Method refactoring does not guarantee that the nesting level will be reduced. For example, let us consider

9http://metrics.sourceforge.net
Fig. 14. Improvements in complexity by operator.
the situation exemplified in Figure 15. The original class contains methods `foo()` and `brainMethod()` with a maximum nesting level of 3 and 4, respectively. In this way, the average NBD of the class is 3.5. After refactoring `brainMethod()` by extracting the second block, the maximum nesting level of the methods are 3, 4, and 4, and the average NBD of the refactored class gets higher than that of the original class. Thus, we can say that NBD is not helpful to analyze class complexity in our case.

The cohesion (LCOM) between methods of the affected classes was barely affected by the refactorings. On average, LCOM decreased a 0.66% (with a standard deviation of 2.6%), which could mean that our strategy to refactor BMs does not improve class cohesion. We argue that the LCOM of the class is not affected (after the refactorings), since the variation is very small with respect to the original class.

Regarding the average lines of code per method (MLOC), we found that they slightly increased. Specifically, MLOC had a 1.61% increase (with a standard deviation of 0.91%) on average. This result was expected because each Extract Method introduces an overhead of lines of code given by the method headers. Also, there might be an overhead when the extracted method initializes a variable. We think that the increment of MLOC is marginal, with respect to the original class.

To explore the influence of the operators in the metrics, we conducted a multi-variate analysis using Kruskal Wallis’ ranks sum test [49]. Kruskal Wallis is a non-parametric version of ANOVA for analyzing experiments from a number of different designs, i.e., with more than one treatment. Our intuition here is that choosing a particular operator does not influence the metric results. Formally, the null and research hypothesis of the test are the following:

- \( H_{20} \): The results of a given metric are equal among the different operators.
- \( H_{21} \): The results of a given metric are different among the different operators.

We set the probability of error at \( \alpha = 0.05 \). After running the tests, we cannot reject \( H_{20} \) for any metric with a p-values 0.8369 (MLOC), 0.6955 (NOM), 0.6394 (CYCLO), 0.9079 (NBD), and 0.9899 (LCOM). This means that the differences of applying different operators are not statistically significant when measuring the metrics.
Overall, from the analysis of the five metrics obtained from the applications, we can only report gains in CYCLO. Thus, it is not possible to claim that Bandago reduces class complexity. To achieve this goal, we believe that more code smells should be taken into consideration. After refactoring a BM, the existence of other kinds of smells should be checked in order to fix them (see Section 5.5). Such an extended refactoring is out of the scope of this article, and we will address it in future work.

5.3 RQ#3: Does Bandago fix more BMs than JDeodorant?

In this section, we compare the effectiveness of Bandago against that of a third-party refactoring tool called JDeodorant [41]. We chose JDeodorant since it is a well-known tool and has been applied in previous work (e.g. [6, 9, 17, 35]). JDeodorant is not intended to fix BMs but rather to identify refactoring opportunities and then apply the Extract Method refactoring. Basically, we check the BMs solved by applying all the refactoring opportunities detected by JDeodorant, and compare its results with those of Bandago. More specifically, we look at the number of BMs fixed by both tools and analyze the number of Extract Method refactorings needed for fixing the BM instances.

The procedure for data collection was as follows. Initially, we identified the list of BMs of a given application using JSpIRIT. In the case of JDeodorant, for each BM, we applied the first refactoring opportunity proposed by the tool. After successfully applying a refactoring, we checked again (with JSpIRIT) whether the BM was actually removed from the code. If so, we continued analyzing the next BM in the list. If not, we applied the next refactoring opportunity proposed by JDeodorant. This procedure is repeated until either the BM is fixed or all refactoring proposals from JDeodorant are exhausted. In the case of Bandago, we configured it to generate one solution per BM using the RANDOM operator (since in RQ#6 we determined that it often produces good results). For each BM, we applied the refactoring solution just proposed (if any), with the assurance that the BM was removed. As output of this procedure, we obtained two search trees for the same application: one for JDeodorant and another for Bandago. The root of each tree is the original BM and the leaves correspond to refactoring solutions (similar to Figure 3). Note that the (terminal) refactoring solutions proposed by JDeodorant may or may not fix the BM. This procedure was manually applied, and it is a time-consuming task. Therefore, we only applied it on two applications, namely: SportsTracker (23K LOC) and CheckStyle (47K LOC).

For SportsTracker, JDeodorant suggested a total of 78 refactoring opportunities (i.e. 78 Extract Method refactorings) for fixing a total of 6 BMs detected (Table 4). However, only 29 of these refactoring opportunities could be successfully applied. That is, after trying to apply 49 of the refactorings proposed by JDeodorant, the resulting solution left the source code in a non-compilable state so we had to discard the solution. This situation was often caused by return or break statements that cannot be addressed by applying the Extract Method refactoring (see Section 4.3). Unlike Bandago, JDeodorant does not check the applicability of these refactorings. Moreover, after applying the 29 successful refactorings opportunities, the BMs could not be fixed. When it comes to Bandago, it applied a total of 12 Extract Method refactorings to fix 4 BMs. The remaining 2 BMs could not be solved by Bandago due to the existence of returns statements in the blocks. We observed that, while JDeodorant applies a larger number of Extract Method than Bandago, the strategies of JDeodorant to extract code blocks are generally not enough to fix BMs. Moreover, the refactorings of JDeodorant that were applicable consisted of extracting small code blocks, when compared to the blocks extracted by Bandago. Specifically, the extractions of JDeodorant had an average of 11.41 LOC (with a standard deviation of 7.96) while the extractions of Bandago had an average of 25 LOC (with a standard deviation of 16.9). For example, let us consider the BM DiagramPanel.updateDiagram shown in Figure 2. JDeodorant applied 4 Extract Method refactorings whereas Bandago only applied 2 refactorings for the same smell. However, while the extractions
of JDeodorant had 9, 14, 5, and 12 LOC, the extractions of Bandago had 45 and 21 LOC, which illustrates the difference between the two tools.

For CheckStyle, which has more BMs (52) than SportsTracker, we obtained similar trends as in SportsTracker. On one hand, JDeodorant suggested a total of 165 refactoring opportunities but only 43 of them were successfully applied. After these 43 refactorings, only one BM was fixed. On the other hand, Bandago applied a total of 155 Extract Method refactorings to fix 31 BMs. The code blocks extracted by JDeodorant were again smaller than the ones extracted by Bandago. Specifically, the extractions of JDeodorant had an average of 9.72 LOC (with a standard deviation of 19.46) while the extractions of Bandago had an average of 29.08 LOC (with a standard deviation of 21.63). As

<table>
<thead>
<tr>
<th>Brain Method</th>
<th>JDeodorant</th>
<th>Bandago</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Extractions proposed</td>
<td>#Erroneous extractions</td>
<td>Fixed?</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>TimexPwx Parser.parse WorkoutNode</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TimexPwx Parser.parse WorkoutSegments</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>TimexPwx Parser.parse WorkoutSamples</td>
<td>45</td>
<td>42</td>
</tr>
<tr>
<td>DiagramPanel. updateDiagram</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>PolarHsrRaw Parser. parseExercise</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>PolarSRaw Parser. parseExercise</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>78</td>
<td>49</td>
</tr>
</tbody>
</table>

Table 4. JDeodorant and Bandago effectiveness for SportTracker

Fig. 16. Distribution of refactoring suggestions by BM for CheckStyle.
it is shown in Figure 16, we noticed that JDeodorant only suggests refactorings for 19 out of 52 smells (regardless of whether they fix the smell) while Bandago suggests solutions for 31 out of 52. JDeodorant proposed between 1 and 4 extractions for a total of 10 smells, and 5 or more extractions in 9 occasions. The number of valid extractions for a smell proposed by Bandago were also in the range of 1 to 4 for a total of 19 smells, and 5 or more extractions for 12 smells.

After this analysis, we empirically observed that Bandago fixed 60.34% of the BMs analyzed while JDeodorant only fixed 1.72% of them. As we mentioned before, the performance of JDeodorant in the two applications under analysis was poor mainly because the kind of extractions supported by JDeodorant were not enough to fix most BMs.

5.4 RQ#4: Do developers find the solutions proposed by Bandago useful?

Since the metric improvements (Section 5.2) do not necessarily indicate that the solutions suggested by Bandago are useful to developers, we conducted a qualitative study with industrial developers. As part of this study, we also analyzed the developers’ opinion regarding the refactorings proposed by Bandago and JDeodorant. To this end, we invited industrial developers to complete an on-line questionnaire about BMs in the application JGroups. We choose JGroups because it is one of the applications with the largest number of BMs and also because it is well documented. The invitations were sent via e-mail and Java development groups of Linkedin\( ^{10} \). The participants received the questionnaire via Google forms\( ^{11} \), which provided: detailed instructions to do the experiment, a background survey, and a number of tasks (BMs and their refactorings) to be performed by each participant. A total of 35 developers participated in the study, most of them (91%) had 3 or more years of Java programming experience.

Initially, each participant was asked to answer questions about their background in programming and refactoring. Also, each developer was given a short introduction to the JGroups application and its architecture. At last, we divided developers into two groups: one for Bandago (19 subjects) and another for JDeodorant (16 subjects). In the first group, each participant was asked to perform 10 tasks. A task involved a comparison between 2 versions of a particular BM taken from JGroups. The first version was the original method, while the second version corresponded to a refactoring of that method as proposed by Bandago (or by JDeodorant). Also, we included a short explanation of the goal of each original method to help participants understand the context of the method. From a total of 37 BMs detected in JGroups, we selected a sample of 10 BMs in order to not overwhelm the participants with the tasks. When choosing this sample of BMs, we focused on methods that could represent a problem and avoided, for instance, auto-generated methods. Regarding JDeodorant, the tool only proposed refactorings for 7 of the 10 BMs initially selected but in 2 cases all the refactorings suggested produced compilation problems. For this reason, each participant of the JDeodorant group was asked to perform 5 tasks (instead of 10 tasks, as in the Bandago Group).

Because of their internal working, the refactorings offered by the two tools for the same BM were different. For each BM, we applied all the refactorings proposed by JDeodorant (those that did not lead to compilation errors). It should be noticed that none of the 5 BMs were fixed by JDeodorant after applying all its suggested refactorings.

For both groups, the BM instances were presented to each participant in a random order. To avoid disclosing the goal of our experiment (which could affect the results), we did not reveal to the participants that the original methods were BMs nor that the refactored versions were generated with Bandago (or with JDeodorant). Additionally, we made available the complete source code of

\( ^{10} \)https://www.linkedin.com/groups/70526, https://www.linkedin.com/groups/50472

\( ^{11} \)The tasks and questions for Bandago and JDeodorant can be found at https://goo.gl/forms/XCB65N0Qa0hSTf9M2 and https://goo.gl/forms/hPimkcZISpa7bLth2, respectively.

ACM Transactions on Software Engineering and Methodology, Vol. 9, No. 4, Article 39. Publication date: March 2010.
the JGroups project to the participants. At the end of each task, we asked each participant to fill in a post-task questionnaire, using a Likert scale ranging from 1 to 5. This post-task questionnaire had the following questions:

1. Please, rate, in your opinion, what the possibility would be that the "original" method causes a maintenance problem? (1=unlikely; 5=very likely)
2. How useful do you consider the refactoring that was applied in the method regarding the readability of the code? (1=unhelpful; 5=very useful)
3. How do you perceive the complexity of the source code resulting from the refactored version when compared to the original version? (1=less complex; 5=more complex)
4. Would you have refactored the method differently? If yes, please indicate what reasoning you would have followed to do so

Participants were allowed to spent as much time as they need to complete the tasks. Also, we allowed participants to leave optional comments about the refactoring applied. Next, we analyze the results of the four questions (above) asked to participants. Please note that question 4 was an open-ended question in order to get feedback from the developers.

### 5.4.1 Perception of maintenance problems

To analyze if developers perceived complexity in the analyzed methods, we asked each participant whether she considered that the method can cause a maintenance problem. This question can be seen as a sanity check about the method representing a code smell. Figure 17a and Figure 18a plot the answers of the developers (1=unlikely; 5=very likely). With the exception of tasks 1 and 6, all medians are above 3, which means that developers estimated that is likely that the methods will produce a maintenance problem. Although the box-plots give descriptive insights, no conclusion can be made so far on whether there is a significant difference on how developers perceive the possibility of a maintenance problem. In order to run a statistical test, we tested the data (developers’ responses) for normality using the Shapiro-Wilks test and concluded that the data deviates from normality (p-value=1.521e-14). Thus, we used the Wilcoxon test to check the difference between developers on the perception of maintenance problems. In this context, we define the following null hypothesis for each task:

- **H3 − 10** The median of the answers is less or equal than 3

---

12The survey containing the tasks and questions can be found in https://goo.gl/forms/XCB65N0Qa0hSTf9M2
13All the results can be found at https://goo.gl/fYWjra
(a) Possibility that the "original" method causes a maintenance problem

(b) Usefulness of the refactoring regarding the readability

(c) Complexity of the refactored source code

Fig. 17. Distribution of the answers (Likert scale) by task for Bandago group.
(a) Assessment of the “original” method as a possible maintenance problem

(b) Usefulness of the refactoring regarding its readability

(c) Complexity of the refactored source code

Fig. 18. Distribution of the answers (Likert scale) by task for JDeodorant group.
The alternative hypothesis ($H_3 - 1$) is that the median of the answers is greater than 3. Table 5 describes the p-values obtained for each task (note that tasks 2, 3, 5, 7, and 9 contain data from both groups). We tested the hypotheses at a 5% ($\alpha$) significance level, and used an adjusted alpha value for the comparisons, based on the Bonferroni correction\(^\text{14}\). This is a multiple-comparison correction when several tests are performed simultaneously, and it implies that the alpha value needs to be lowered to account for the number of comparisons. A conservative approach is to use an alpha value divided by the number of comparisons. Since we compare each $BM$ for maintenance, usefulness and complexity, we evaluate them with respect to 1/3 of the 5% significance level, i.e., $\alpha = 0.01666$. With these values, $H_3 - 1_0$ is rejected for all the tasks (with the exception of tasks 1 and 10) with a one-tailed test with an adjusted probability of error (or significance level) $\alpha = 0.01666$. That is, in most cases, the participants agreed that the methods selected can possibly produce a maintainability problem. In the case of task 1, many developers perceived that the possibility of the method causing a modifiability problem would be low (20 developers, out of 35, answered between 1 and 3 in the Likert scale). Based on the developers’ comments, we believe that this perception stems from the fact that some developers thought the method should not be refactored because it is a kind of parser. Similarly, in task 10, 9 developers, out of 19, answered between 1 and 3 in the Likert scale. In this case, the perception stems from the fact that the method mainly initializes the variables of the class.

### 5.4.2 Usefulness of the refactoring

For question 2, we measured how useful developers find the proposed refactorings for each method. For the Bandago group, we found that from 190 answers, 125 (65.8%) were 4 and 5 (1=unhelpful; 5=very useful). However, for the JDeodorant group, only 35% (28 out of 80) of the answers were 4 or 5. Figure 17b and Figure 18b plot the answers of developers. In order to test whether a statistically significant difference exists on how developers perceive the usefulness of the refactorings, we ran a statistical test. Initially, we found that the data is not normally distributed across tasks for both groups using the Shapiro-Wilk test (p-value=1.104e-12 and 1.612e-05). Then, we used the Wilcoxon test to check for any difference among developers’ answers. In this context, we define the following null hypothesis for each task:

- $H_3 - 2_0$ The median of the answers is less or equal than 3

\[^{14}\text{http://mathworld.wolfram.com/BonferroniCorrection.html}\]

<table>
<thead>
<tr>
<th>Task</th>
<th>Bandago</th>
<th></th>
<th></th>
<th>JDeodorant</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Median</td>
<td>p-value</td>
<td>Median</td>
<td>p-value</td>
<td>Median</td>
<td>p-value</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0.003778</td>
<td>2</td>
<td>0.0003608</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0.0004323</td>
<td>2</td>
<td>6.778e-05</td>
<td>2.5</td>
<td>0.8902</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.1747</td>
<td>2</td>
<td>0.001969</td>
<td>3.5</td>
<td>0.1538</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0.1809</td>
<td>2</td>
<td>0.004521</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0.344</td>
<td>1</td>
<td>0.0003499</td>
<td>3</td>
<td>0.8477</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0.009833</td>
<td>2</td>
<td>0.0008573</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>0.0004521</td>
<td>2</td>
<td>0.01025</td>
<td>4</td>
<td>0.02381</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0.01987</td>
<td>2</td>
<td>0.004714</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>0.00109</td>
<td>2</td>
<td>0.01263</td>
<td>2.5</td>
<td>0.9946</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0.03648</td>
<td>2</td>
<td>0.0001615</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6. Statistical tests: usefulness and complexity.
The alternative hypothesis ($H_3 - 2_1$) is that the median of the answers is greater than 3. Table 6 shows the p-values obtained for each task. We applied a Bonferroni correction to 1/3 of the 5% significance level to ensure a low risk of type-I error. For the Bandago group, $H_3 - 2_0$ can be rejected in 5 of the tasks (hypotheses of tasks 3, 4, 5, 8, and 10 could not be rejected) with a one-tailed test with an adjusted probability of error $\alpha = 0.01666$. These results indicate that participants found the refactorings (computed with Bandago) useful in half of the tasks.

Regarding the comments provided by the developers about the refactorings, some developers indicated that they would complement the refactoring by extracting more code blocks from the original method (these comments were mostly made in tasks 3, 4, and 5, whose hypotheses could not be rejected). This was the case, for example, of a switch-case statement in which Bandago only extracted the source code of a case statement into a new method. This situation happens because Bandago stops extracting methods after finding that the $BM$ not longer exists. However, this kind of situations is subject of future work.

For the JDeodorant group, and looking at the values in Table 6, $H_3 - 2_0$ cannot be rejected for any task with a one-tailed test with an adjusted probability of error $\alpha = 0.01666$. Thus, developers did not found useful the refactorings suggested by JDeodorant. If we only compare the 5 tasks shared by both groups, 3 hypotheses of the Bandago group can be rejected against no hypothesis of the JDeodorant group.

5.4.3 Complexity of the refactored version. Finally, developers were asked about their perception of the complexity of the source code when compared to the original version of the method. For the Bandago group, we found that they perceived a reduction in the complexity. Specifically, from 190 answers, we obtained 142 (74.7%) answers being 1 or 2 (where 1=less complex; 5=more complex). However, in the JDeodorant group, only 37.5% (30 out of 80) of the answers were 1 or 2. Figure 17c and Figure 18c plot the developers’ answers, showing that, in the case of Bandago, all the medians are 2 (with the exception of task 5 that is 1), while in the case of JDeodorant the medians are 2.5 and 3. In order to test whether a statistically significant difference exists on how a developer perceives the complexity reduction (in each task), we ran a statistical test. Initially, we found for both groups that the data is not normally distributed across tasks using the Shapiro-Wilks test (p-value=4.582e-13 and 5.292e-08 respectively). Then, we used the Wilcoxon test to check for any difference among developers’ answers. In this context, we define the following null hypothesis for each task:

- $H_3 - 3_0$ The median of the answers is greater or equal than 3

The alternative hypothesis ($H_3 - 3_1$) says that the median of the answers is less than 3. Table 6 shows the p-values obtained for each task. As in the previous cases, we applied a Bonferroni correction to 1/3 of the 5% significance level. Regarding the Bandago group, from these values, $H_3 - 3_0$ can be rejected for all the tasks with a one-tailed test with an adjusted probability of error $\alpha = 0.01666$. That is, developers perceived a strong reduction of the method complexity after the solutions suggested by Bandago were applied. In contrast, in the JDeodorant group, $H_3 - 2_0$ can only be rejected for tasks 3 and 7 (40% of the tasks). That is, developers only noticed a complexity reduction in two of the $BM$s refactored by JDeodorant.

Overall, taking into account the answers to the three questions, we can answer RQ#4 positively by saying that developers found that the solutions proposed by Bandago reduce the complexity of $BM$s.

5.4.4 Analysis of developers’ feedback. To complement the study presented before, we analyzed the responses provided by the participants for question 4 in a qualitative way. Approximately 65% of the participants from the Bandago group provided feedback about their tasks, while 40% of the
participants did it in the JDeodorant group. The main observation for both groups was that, when confronted with the refactoring of a BM, developers are not only concerned with removing the smell per se, but they also want to improve the readability and maintainability of the methods. Thus, although developers accepted many of the refactorings proposed by Bandago or JDeodorant, they often felt that “more refactoring was necessary”. In addition, developers expected that an automated refactoring follows “regularities” with respect to the methods being extracted from the smelly method. For example, if a BM consists of two (or more) for blocks in a sequence, a developer usually expects all the for blocks to be refactored into corresponding methods, even when a tool identifies that extracting only one of the for blocks would fix the smell. This aspect was even more evident with JDeodorant, as the tool was not able to remove any BM completely, but developers did not seem to worry about this. As another example, if a BM entails a switch-case statement, a developer is likely to have each case block refactored into a method. The arguments behind this reasoning were a better readability of the code or a cleaner division of responsibilities in support of system evolution. Some participants even suggested the application of design patterns (e.g., Command, or State) or the re-structuring of the class hierarchy, which were not supported neither by Bandago nor JDeodorant. Another suggestion related to the separation of responsibilities was that of reuse of extracted methods, in a few tasks in which the extraction of several methods led to code duplication. This kind of analysis is not currently supported in Bandago, but it could be incorporated as future work. We interpret all these human considerations as a way for developers to justify their efforts in making (or assessing) a refactoring of working code. At last, some participants highlighted the need of following some coding conventions or good programming rules when performing the refactorings, such as: setters and getters in method naming conventions, avoidance of methods with side-effects, usage of curly brackets, etc. We believe these code quality features can be easily supported with existing tools, such as Autorefactor or Spartan, which can work in tandem with Bandago.

Regarding the specific refactorings proposed by Bandago, they seemed satisfactory to the participants. In particular, developers agreed with refactorings in which the method extraction was based on the source code comments. Nonetheless, the participants reported that those refactorings involving “nested” method extractions (i.e., extracting a given code fragment from the main method, and subsequently extracting a smaller fragment from the previous fragment) were more difficult to reason about than a set of refactorings performed in several parts of the main method (without nesting). In the refactorings proposed by JDeodorant, there were less method extractions than in Bandago (for the same tasks) and the extractions were fine-grained, in the sense that they affected smaller code portions. As a result, developers had trouble to follow the “logic” behind a JDeodorant refactoring, and made this noticeable in their feedback, asking for more effective refactorings. Although these observations seemed to indicate that Bandago (with its own limitations) proposed more appealing refactorings than JDeodorant, the observations are anecdotal and should not be taken as a systematic qualitative study. A caveat here is that all participants in both the Bandago and JDeodorant groups were not the original developers of the projects under evaluation.

In order to probe the original developers of the projects about Bandago, we submitted a series of pull requests with refactorings to those projects in GitHub. A subset of seven projects were hosted in GitHub, namely: JGroups, SportsTracker, Cayenne, CheckStyle, Jena, Quartz, and Roller. We looked at the current version of each project, and made one pull request per project. Each pull request contained two BMs refactored with Bandago, along with a short explanation of the

\[\text{http://autorefactor.org} \]
\[\text{https://www.spartan.org.il} \]
\[\text{http://github.com} \]
\[\text{All the results can be found at https://goo.gl/fYWjra} \]
Table 7. Code smells fixed by Bandago

<table>
<thead>
<tr>
<th>Code Smell</th>
<th>#Smells in original version</th>
<th>#Smells in refactored version</th>
<th>% of smells fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain Method</td>
<td>5400</td>
<td>3262</td>
<td>62.88</td>
</tr>
<tr>
<td>Brain Class</td>
<td>1350</td>
<td>620</td>
<td>54.07</td>
</tr>
<tr>
<td>Dispersed Coupling</td>
<td>37225</td>
<td>38220</td>
<td>-2.67</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>57850</td>
<td>60854</td>
<td>-5.19</td>
</tr>
<tr>
<td>God Class</td>
<td>8000</td>
<td>8109</td>
<td>-1.36</td>
</tr>
<tr>
<td>Intensive Coupling</td>
<td>9600</td>
<td>9815</td>
<td>-2.24</td>
</tr>
</tbody>
</table>

5.5 RQ#5: Does the fix of BMs help to deal with other code smells?

In general, BMs appear connected to other code smells in the system. In this section, we analyze whether additional smells are affected by the refactoring of BMs. To answer RQ#5, we compared the list of code smells found by JSpIRIT, before and after refactoring each application with Bandago. The smells considered as directly related to BM [15] were the following: Brain Class, Dispersed Coupling, Feature Envy, God Class, and Intensive Coupling. We conjecture that when Bandago successfully applies a Extract Method refactoring for a BM, it might also fix some other smells in the process.

As reported in Section 5.1, Bandago refactored an average of 62.88% of the BMs. This refactoring exercise also had a positive influence on fixing other code smell, as shown in Table 7 (Note that results in the table summarize the 25 executions of Bandago, for each application). Overall, an interesting fact was that 54.07% of the Brain Class smells were indirectly solved. On the contrary, a 2.67%, 5.19%, 1.36%, and 2.24% of new instances for Dispersed Coupling, Feature Envy, God Class, and Intensive Coupling were detected, respectively, after refactoring the BMs. Next, we analyze in detail each of the smells according to the operator used (Figure 19).

The high rate of success in fixing Brain Classes can be explained by the fact that a Brain Class is defined as a very complex class with at least one BM. Regarding the kind of operator, we could not find significant differences in the results for this smell.

As for Dispersed Coupling, we found that 2.67% of Dispersed Coupling smells were created, with a standard deviation of 0.82% among the different operators (Figure 19). After analyzing the
Assessing the Refactoring of Brain Methods

Fig. 19. Code smells fixed by operator.
source code, we observed that many of the BMs originally detected were also instances of Dispersed Coupling. In this way, the refactoring of the BMs naturally eliminated some Dispersed Couplings, but most of them were unfortunately moved to the extracted methods. This mainly happened for the LOC (Figure 19a) and NESTING (Figure 19c) operators, in which the values of new Dispersed Couplings were 5.25% and 5.52%, respectively. In the case of LOC, when blocks of several lines of code are extracted, it is more likely to extract a set of statements with several calls to methods of different classes. In the case of NESTING, the chances of extracting a Dispersed Coupling are higher because a nesting level above 1 is one of the rules to detect the smell [15].

The case of Feature Envy is similar to that of Dispersed Coupling, as the smells already existed in the BM and were later moved to an extracted method. The number of Feature Envy smells increased in 5.19%, with a standard deviation of 1.2% among the operators. The existence of Feature Envy stems from the fact that a BM accesses to variables declared in other classes. These accesses cannot be reduced by applying the Extract Method refactoring. Thus, a number of Feature Envy instances can be generated when refactoring BMs. NOAV is the operator that led to the largest number of Feature Envy instances (8.43%). This is because NOAV extracts blocks of code that maximize the number of variables involved in a set of sentences, increasing the chances for these variables to be declared in other classes. For Intensive Coupling, the trend is similar to the previous ones. The code smells originally existed in the BM and, after the refactoring, they were moved to an extracted method. The refactorings applied by Bandago generated an average 2.24% of new Intensive Couplings, with a marginal difference depending on the operator being used (standard deviation of 0.33%).

At last, the God Class is the only smell that did not exist in the original applications but appeared after applying the refactorings. On average, Bandago generated 1.36% of new God Classes, with a standard deviation of 0.85% among the operators. This low percentage is attributed to particular cases, in which the thresholds of the metric-based detection rules are borderline. In fact, after analyzing the source code of the new God Class instances, we found that previously to the refactoring, many instances were actually Brain Classes. This situation happens because the class cohesion was reduced, and the detection rules of God Class and Brain Class are similar.

The above analysis allowed us to empirically observe that Bandago fixed indirectly a 40% of Brain Classes when it refactors BMs. Also, the results show that, despite small increments in some kinds of smells, the emergence of new smells is related to smells that already existed in the original BM. Thus, the design of refactoring strategies for other kinds of code smells should be analyzed in future work.

5.6 RQ#6: What operator (from Activity 2) produces the best results?

Since Bandago can be configured to work with different operators (Activity 2), we need to study the results of each operator. Specifically, we are interested in analyzing the number of BMs solved by an operator and the improvements in the source code achieved by the refactorings proposed by each operator. Our initial hypothesis is that there exists an operator that performs better than the rest, that is:

- $H_{50}$: All the operators fix the same amount of code smells.
- $H_{51}$: There is an operator that fixes more code smells than the rest.

Figure 20 shows the percentage of code smells fixed, on average, by each operator. In those cases where percentages are negative it means that new instances of the smell were detected after applying the refactorings. As we explained in the previous section, a usual case is that the smells already existed in the BM and were later moved to an extracted method. In general, the best results were obtained with the RANDOM operator. This happens because RANDOM composes its solutions
by means of extractions generated using the remaining four operators. More Brain Classes are fixed using RANDOM, since Brain Method and Brain Class are smells closely related. Moreover, the RANDOM operator created solutions with the lowest number of Disperse Coupling and Intensive Coupling smells. Regarding Feature Envy and God Class, while the RANDOM operator does not guarantee a low number of new smells, the percentages of improvement are similar to those of other operators. For example, for Feature Envy, the lowest number of new smells is generated by the NESTING operator (5.8%) while the greatest number is obtained by NOAV (8.42%).

Figure 21 shows the average metric improvement after applying the refactoring solutions generated with each operator. We used the same set of metrics explained in Section 5.2. For example, after analyzing the results of NOM, we can see that the RANDOM operator, which fixes the highest number of BMs (Figure 20), is also the operator that creates more new methods. This result was expected since solving many BMs requires more extractions to be performed. Also, the NOAV operator generates a similar amount of new methods as RANDOM. This happens because NOAV...
prioritizes smaller code fragments (to be extracted) than the other operators. Thus, NOAV needs on average more extractions to generate a solution than the other operators.

The metrics CYCLO and MLOC followed a behavior similar to that of NOM. For CYCLO, the operators RANDOM and NOAV had the best results because the more extractions are performed, the more likely the cyclomatic complexity is reduced. In the case of MLOC, the more extractions are performed, the higher the overhead of the extractions are. Regarding NBD, it obtained best results after applying the RANDOM operator. Along this line, we can say that, by generating a solution using the different strategies of the RANDOM operator, the average nesting level of the methods of a class is barely affected (0.88%). The LCOM metric does not seem to be affected by the number of methods extracted by a solution. As analyzed in Section 5.2, we argue that the method cohesion of a refactored class was slightly affected by the refactorings.

Although the values reveal the performance of each operator, it is important to test the data to verify any statistically significant difference between operators. We focused our tests in both the fixing of BMs and the metric improvements just discussed. First, in the context of \( H_{50} \), we test if there is an operator that solves more BMs than another. After using a Kruskal-Wallis non-parametric test with a probability of error (or significance level) \( \alpha = 0.05 \) (i.e. there is a 5% chance of wrongly accepted \( H_{50} \)), we cannot reject \( H_{50} \) with a \( p\)-value=0.9182. That is to say, while the RANDOM operator consistently fixes more BMs than other operators (Figure 19), there is not enough evidence to claim that there is a significant difference about the number of BMs fixed by each operator. Second, we test if there is an operator that has better results (for any metric) than another operator. We again used Kruskal-Wallis with \( \alpha = 0.05 \). After conducting the tests, we cannot reject \( H_{50} \) for any of the operators with \( p\)-values 0.6955, 0.6394, 0.9079, 0.9899, and 0.8369 for NOM, CYCLO, NBD, LCOM, and MLOC, respectively. Thus, there is no statistically significant difference in the improvements made by the operator, according to the different metrics.

Overall, we can answer to RQ#6 by saying that the operators produce statistically similar results. However, we can notice a small difference in favor of the RANDOM operator.

5.7 RQ#7: What is the average time to generate solutions for a BM?

A practical aspect of Bandago is the computation time needed for recommending a refactoring for a BM. To address this question, we followed the same procedure as in research questions 1-4. Since the computation time may vary depending on the operators used in the simulated annealing algorithm, we measured the execution time needed per operator for generating each of the 3 solutions for a smell (when possible). Occasionally, Bandago might abort the search for a solution, if it cannot find a candidate statement to be extracted. In such cases, it is not always possible to obtain 3 solutions, so we just measured the time until the search is stopped. The experiment was conducted with a laptop MacBook Air, CPU 1.3 GHz Intel Core i5, 4Gb 1660 MHz DDR3 of memory. We hypothesize that there exists an operator that finds solutions in less time than the rest, with the following hypotheses:

- \( H_{70} \): All the operators spend the same time to find a solution for a smell.
- \( H_{71} \): There is an operator that spends less time to find a solution than the rest.

Figure 22 shows the distribution of the time to compute a solution for a BM instance, according to individual operators. Most solutions using the operators LOC, CYCLO, NESTING and NOAV were in the range of 50-70 seconds. Moreover, the medians using these operators were 61.4, 60.45, 59.8, and 63.65 seconds respectively. We observed that the RANDOM operator achieved the best results, as it was able to find solutions in a range of 25-35 seconds (average 30.294 seconds) and with a median of 32.5 seconds. As noted in Section 5.6, we believed that RANDOM finds a wide range of solutions because it combines the strategies of the other four operators.
In order to check whether a statistically significant difference exists regarding the time spent by each operator, we used the Kruskal-Wallis non-parametric test with a probability of error $\alpha = 0.05$. After running the test, we can reject $H_{0}$ with a p-value=2.2e-16. That is to say, there is enough evidence to claim that there is a difference in the time spent by each operator. Complementary, we conducted a post-hoc test to determine which pairs of operators have significant time differences when generating solutions. The post-hoc test revealed that RANDOM is the fastest operator because it has significant differences with LOC (p-value=9.8e-14), CYCLO (p-value=2.7e-12), NESTING (p-value=2.7e-11), and NOAV (p-value=4.6e-11).

We can answer to RQ#7 by saying that the average time to generate a solution to a BM with the fastest operator (RANDOM) is around 30 seconds. We claim that this time is acceptable for a normal user, and it does not compromise the usability of Bandago.

5.8 Threats to validity

Our study involved a number of threats to validity, which are discussed next.

Conclusion validity. This threat concerns the statistical analysis of the results. The main concern is that the study was made over 10 applications. Analyzing only a limited number of applications could reduce the ability to draw correct results. To minimize this threat we selected applications from different domains and of different sizes. This small number of applications is noticeable in the case of RQ#3, in which the evaluation should be performed using more applications in future work. Another threat is the possibility of having false positive smells introduced by the metric-based detection rules of JSpIRIT. We mitigated this threat by applying the same metric thresholds proposed
by Lanza and Marinescu [15], which had been also used in previous work [48]. Another threat for the validity of the conclusion is that we did not consider the execution of application tests before and after the Brain Method refactorings in our experiments. We relied on the Extract Method API to guarantee that system behavior is preserved (the API also guarantees that no compilation errors are created). However, this property could not hold in odd cases for some refactored classes. In such a situation, performing regression testing is a recommended practice.

**Internal validity.** This threat concerns causes that can affect the independent variable of the experiment without the researcher’s knowledge. An internal threat is related to the variability that the results could have, due to the intrinsic randomness of the simulated annealing algorithm. That is, different executions of the algorithm on a given BM could produce different solutions. We think that this threat was mitigated by the total number of BMs (5400) refactored during the study (see Section 5). Some applications used in the experiments (CheckStyle, Jena, Quartz, Roller, and Squirrel) had in average 18% of auto-generated classes with BMs (the rest of the applications did not involve BMs in auto-generated classes), which are not refactoring targets for developers in practice. Nonetheless, we considered these auto-generated classes as exemplars to exercise Bandago refactoring capabilities, and thus, we believe they should not affect the results of the experiment. That is, we are not analyzing the origin of the methods but rather if Bandago can extract fragments of their source code to fix the BM and reduce its complexity. Another threat is the kind of operators chosen for our approach. Certainly, using other kinds of operators could lead to different results. Also, the set of metrics for measuring complexity (RQ#2) could bias the results, as other alternative metrics could have been selected instead.

**Construct validity.** It is concerned with the design of the experiment and the behavior of the subjects. A main concern is that our study did not assess the quality of the refactorings made by Bandago (except for the metrics analyzed in RQ#2). Moreover, we did not consider the semantic cohesion of the extracted methods. However, we used some criteria to select the statements to be extracted (Activity 1), such as GroupStatementFilter, that could help to improve the semantic relationship of the extracted methods. This kind of validation is out of the scope of this article and it could be subject of future work. Another mitigation for this threat was the inclusion of an experiment with industrial developers. By means of this experiment, we assessed the quality of the refactorings made by Bandago regarding the reduction of the source code complexity. Although these developers were not the original ones of the JGroups project, we required them to have a moderate (or higher) experience in Java programming, and also provided them with a design context of the tasks to be performed in the questionnaire. Unfortunately, conducting an extensive study with the original project developers is not always feasible.

**External validity.** It is concerned with having a subject that is not representative of the population. The main threat is that some applications analyzed are of small size and have a limited amount of BMs. We mitigated this threat by including some applications of larger size (and thus, with BMs) within our study. Another threat is the fact that we emulate the developer’s behavior by choosing one of the solutions (proposed by the tool) based on metrics. In practice, a real developer could consider other factors (besides the metrics) in order to choose a suitable refactoring solution. Our qualitative analysis of the developers’ answers, although preliminary, shed light on this direction. Furthermore, we were able to perform a small inquiry via pull requests to a subset of the projects, and obtained reasonable results for some Bandago refactorings. At last, another threat is the selection of the 10 BMs of JGroups in Section 5.4, since another set of BMs could lead to different results.
6 CONCLUSIONS

This article presents an approach called Bandago that automates the refactoring of BMs. For each BM, Bandago proposes several solutions that fix the smell using a novel strategy based on a simulated annealing algorithm. In addition to typical benefits of refactoring tools [28, 44], such as lower error rates and less time required, we posit that assistive approaches, like Bandago, bring added value to developers. First, Bandago is lightweight but still goal-driven, in the sense that it does not attempt to make a big code change, it rather focuses on fixing the smell at hand (i.e., the developers intention). Second, it contributes to awareness, as the candidate refactorings presented to the developer show quality metrics about the effects of those refactorings.

In order to assess the benefits of our approach, we conducted an experiment driven by 6 research questions. We empirically demonstrated that Bandago is practical and effective for fixing BMs. In our set of applications, Bandago was able to automatically fix 60% of the BMs (RQ#1), with computation times of ~30 seconds for finding solutions using the RANDOM operator (RQ#7). We also found that the solutions of Bandago can somehow contribute to modifiability aspects of the refactored projects, although the evidence was not conclusive (RQ#2). However, an online survey with 35 industrial developers showed that the Bandago solutions were useful to them, and furthermore, that the refactorings (for a particular project) decreased the complexity of methods (RQ#4). Also, we obtained interesting feedback from 6 pull requests submitted to the GitHub repositories of the analyzed projects. Additionally, there was an indirect effect of fixing BMs with Bandago towards Brain Classes, which were also fixed when applying Extract Method refactoring (RQ#5). We observed that the RANDOM operator seems to fix more BMs than the other operators, although more analysis is needed to confirm this claim (RQ#6). At last, we compared the outputs of Bandago against those of JDeodorant, and our preliminary results (with two applications) indicate that Bandago can fix ~60% more BMs than JDeodorant (RQ#3).

Although Bandago is effective in assisting the developer to refactor BMs, the approach has still some limitations. First, the approach was solely conceived to work on Java applications. However, we think that its constructive principles can be used to fix BMs in other object-oriented languages, such as C++ or C#. Second, although Bandago performed reasonably well, the search for valid refactoring transformations is currently dependent on the behavior-preserving checks of the Eclipse IDE, which might be a major performance contributor to the search process. Third, the current prototype only uses the Extract Method refactoring to deal with BMs. While we proved that BMs can be solved with this refactoring, other kinds of refactorings can also help to fix this type of smell. As future work, we plan to explore strategies based on other types of refactorings (e.g., Feature Envy, or Intensive Coupling), such as introduce parameter object or preserve whole object [7]. We might also need to investigate other operators. We conjecture that implementing these strategies can lead us to refactor a larger number of BMs, but also to produce better solutions regarding modifiability aspects. Moreover, we need to further extend our qualitative analysis of Bandago focusing on the acceptability of the proposed refactorings by developers in real projects. Also, we plan to incorporate into Bandago search strategy some objectives or preferences related to the readability and reusability of the extracted methods. Along this line, it is worthwhile to extend Bandago with refactoring capabilities for other kind of smells and reuse the infrastructure already provided by JSpIRIT. An alternative in this direction is to investigate tools such as RefactorIT19, which implements other kinds of refactorings than Eclipse. Furthermore, we envision that the approach can be used in conjunction with smell prioritization tools, such as JSpIRIT [48] or JCodeodor [5], which can point out to developers which problems should be analyzed first and then (if feasible)

19https://sourceforge.net/projects/refactorit/

ACM Transactions on Software Engineering and Methodology, Vol. 9, No. 4, Article 39. Publication date: March 2010.
refactored. This way, the research work can move towards the idea of “goal-directed refactoring recommendations” [39] in order to support developers more effectively.

REFERENCES


Assessing the Refactoring of Brain Methods 39:43


Received February 2007; revised March 2009; accepted June 2009