# On the Impact of Refactoring Operations on Code Quality Metrics

Oscar Chaparro[1], Gabriele Bavota[2], Andrian Marcus[1], Massimiliano Di Penta[2]
[1]University of Texas at Dallas, Richardson, TX 75080, USA
[2]University of Sannio, Via Traiano, 82100 Benevento, Italy
ojchaparroa@utdallas.edu, gbavota@unisannio.it, amarcus@utdallas.edu, dipenta@unisannio.it

***Abstract*—Refactorings are behavior-preserving source code transformations. While tool support exists for (semi)automatically identifying refactoring solutions, applying or not a recommended refactoring is usually up to the software developers, who have to assess the impact that the transformation will have on their system. Evaluating the pros (e.g., the bad smell removal) and cons (e.g., side effects of the change) of a refactoring is far from trivial. We present RIPE (Refactoring Impact PrEdiction), a technique that estimates the impact of refactoring operations on source code quality metrics. RIPE supports 12 refactoring operations and 11 metrics and it can be used together with any refactoring recommendation tool. RIPE was used to estimate the impact on 8,103 metric values, for 504 refactorings from 15 open source systems. 38% of the estimates are correct, whereas the median deviation of the estimates from the actual values is 5% (with a 31% average).**

***Index Terms*—Refactoring Impact, Code Quality**

## I. Introduction

Refactoring is "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [1]. While refactoring can remove bad smells, it can also have important side effects (e.g., the removal of a design pattern wanted in the system) and introduce bugs in the code [2]. Assessing the pros and cons of a refactoring operation, before applying it, is very challenging for developers, since some refactorings (e.g., an Extract Class) may require a very high number of code transformations that may be difficult to mentally visualize. This is even more challenging when we consider that different code properties, measured by metrics, conflict with one another (e.g., Coupling vs. Cohesion) and developers must pursue a trade-off [3].

We introduce in this paper RIPE (**R**efactoring **I**mpact **Pr**E**diction), an approach that predicts the impact of 12 refactoring operations on 11 code metrics. RIPE implements 89 impact prediction functions that show developers the variation of code metrics before the application of a refactoring. RIPE is meant to help developers making design decisions and choosing between different refactoring alternatives. For example, suppose class $A$ is a God Class and the developer has to decide between several options of splitting class $A$ via the Extract Class refactoring. Suppose class $A$ can be split into classes $B$ and $C$, or into classes $C$ and $D$, and after using RIPE, the developer would conclude option $B - C$ is not the best option because class $B$ would have a very low cohesion. The developer would use RIPE to evaluate these options before applying the refactoring - rather than after - since it takes time and effort to implement them.

This paper focuses on evaluating the prediction functions on atomic refactorings, while in the long run, RIPE will be able to predict the impact of composite refactorings (i.e., sequences of related refactorings). We empirically evaluated RIPE on a set of 504 atomic refactorings performed in 15 open source systems to estimate a total of 8,103 metric values. RIPE precisely estimated the refactorings' impact on metrics in 38% of cases, while the median deviation from the actual value was 5% (31% on average).

## II. Related Work

Soetens *et al.* [4] study the circumstances when the Cyclomatic Complexity metric increases, decreases, or remains the same, by defining three estimation formulas, one per each of the 3 refactorings studied. RIPE is different because (i) its functions predict the changes at class level, which allows more granular prediction; (ii) the earlier work does not aim at providing an accurate impact estimation of refactoring on the metric; and (iii) their formulas are limited to three refactoring operations and a single code metric, whereas RIPE includes 12 refactoring operations and 11 metrics.

Du Bois *et al.* [5], [6] introduced impact tables, which specify *a priori* knowledge on the effect of a set of 3 refactorings on a set of 5 coupling and cohesion metrics. RIPE accounts for a more extensive set of refactorings and metrics, and defines explicit functions for all the classes directly involved on each refactoring, not only the source class, as the earlier work.

Piveta *et al.* [7] focused on assessing the impact of Aspect Oriented Programming (AOP) refactorings on AOP metrics, through mathematical functions, which aim at guiding developers in choosing the correct refactoring pattern to implement, without performing the change. Unlike Piveta *et al.*, our focus is on Object-Oriented code and we support a much larger set of refactorings and metrics.

## III. RIPE (Refactoring Impact PrEdiction)

RIPE includes a set of functions that predict, at class level, the impact of 12 refactorings on 11 code quality metrics. We include common refactoring operations [1], such as, the ones dealing with generalization (e.g., Pull Up Method - PUM), moving features (e.g., Move Field - MF), and composing methods (e.g., Extract Method - EM). We also consider

common code metrics that measure code properties such as Coupling (e.g., Response For a Class - RFC), Size (e.g., Lines of Code - LOC), Complexity (e.g., Cyclomatic Complexity - CYCLO), or Inheritance (e.g., Depth of Inheritance Tree - DIT). The complete list of refactoring and metrics is available online at http://www.cs.wayne.edu/~severe/ripe.

For each refactoring-metric pair we defined a prediction function for the source classes involved in the refactoring and one for the target classes. Since not every refactoring operation impacts all metrics, the prediction functions have been defined only when appropriate. The functions are heuristic-based and are defined based on Fowler's definition of each refactoring [1], on our development experience, and on the study of many examples of refactorings in literature and practice. In each case, the functions are based on the most common cases of the refactorings. The functions are defined assuming that the refactorings are performed in isolation, i.e., no additional code changes are performed, excluding those needed to preserve the system external behavior. Consequently, it is not expected 100% accuracy in the cases where additional changes occur with the refactoring.

In this section we explain in detail the functions for Replace Method with Method Object (RMMO) and Extract Class (EC), which are refactorings that deal with methods, fields, and classes. These refactorings are exemplars of 2 categories dealing with: composing methods (e.g., RMMO), and moving features between objects (e.g., EC). For each of these operations, we discuss the functions estimating their impact on Coupling Between Objects and Lack of Cohesion of Methods 5, which are metrics that capture coupling and cohesion of the code. A complete list of all the prediction functions defined and implemented in RIPE is available online at http://www.cs.wayne.edu/~severe/ripe.

### A. Code Metrics Definition

**Lack of Cohesion of Methods 5 (LCOM5)** [8] measures the level of cohesion based on the field usage by the methods of a class. It is computed with Equation 1, where $NA(c)$ is the number of attributes of class $c$, $NM(c)$ is the number of methods and $TMA(c)$ is the total number of field accesses, with $TMA(c) = \sum_i NMA(a_i)$, i.e., the sum of the number of methods using each field $a_i$ of class $c$.

$$LCOM5(c) = \frac{TMA(c)/NA(c) - NM(c)}{1 - NM(c)} \quad (1)$$

$LCOM5(c) = 0$ indicates class $c$ is cohesive and higher values indicate that class $c$ is less cohesive.

**Coupling Between Objects (CBO)** measures the coupling between classes based on class usage. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class [9]. The CBO for a class $c$ is defined as the norm of the set of classes ($c_{coup}$), which is computed with Equation 2. The range of values of this metric spans from zero (low coupling) to the total number of classes of a system (high coupling).

$$CBO(c) = |c_{coup}| \quad (2)$$

### B. Prediction Functions for Replace Method with Method Obj.

Fowler [1] defines RMMO as a way to simplify the Extract Method refactoring. The goal is to turn a method $m_k$ of a source class $c_s$ into a new class $c_t$, having the method implemented in it, with all the local variables of the method converted into fields. Then, the method in the new class $c_t$ can be easily broken into multiple methods. When method $m_k$ is using fields or other methods of the source class $c_s$, then the method in the new class $c_t$ uses those fields through an instance of the source class, which is received in the constructor of the new class. Hence, there is no need to pass the fields used in method $m_k$ to the constructor of the new class $c_t$. The prediction functions below consider this scenario and also assume that the getters and setters of the fields exist and are used in the target class.

**Impact on LCOM5.** Equations 3 and 4 are the prediction functions for the LCOM5 metric. In the case of the source class $c_s$, LCOM5 value is negatively affected by $l_k$, the number of attributes used by $m_k$. In the case of the target class $c_t$, let $r$ be the number of local variables and $p$ the number of parameters used in method $m_k$. Also, let $t$ be the number of methods in $c_t$ using the field that references the source class, in which case, $t = 1$ if the field is only referenced in the constructor, or $t = 2$ if it is referenced in both methods, the constructor and method $m_k$.

$$LCOM5_p(c_s) = \frac{(TMA_b - l_k)/NA_b - NM_b}{1 - NM_b} \quad (3)$$

$$LCOM5_p(c_t) = \frac{r - t + 2}{r + p + 1} \quad (4)$$

The target class will have the constructor and the moved method, i.e., the number of methods will be two ($NM = 2$), the number of fields will be $r+p+1$, i.e., the number of variables plus the number of parameters plus a field referencing the source class ($NA = r+p+1$), and the total field accesses will be $r+2p+t$, i.e., the variables used in the method plus the parameters used in both the constructor and the method, plus the additional field access, defined by $t$ ($TMA = r+2p+t$).

Using Equation 1, replacing the values $TMA = r+2p+t$, $NA = r+p+1$ and $NM = 2$, and simplifying the expression, we obtain Equation 4.

**Impact on CBO.** Equations 5 and 6 predict the CBO metric value for the source and target classes, respectively. Equation 5 is reused from Du Bois and Mens [6] and we add here new functions for the target classes, not only for source classes. For the source class, the CBO value is negatively affected by $l_k$, which is the number of distinct classes used by $m_k$ and not used by any other method of $c_s$, and $d_k = 1$, if the source class is not coupled with the target class, otherwise, $d_k = 0$. As for the target class $c_t$, the metric value depends on the number of classes used by method $m_k$. In Equation 6, $e_k$ is the number of distinct classes used by $m_k$, and $r_k = 1$ if there is any field or method usage of the source class from the target class, otherwise $r_k = 0$.

$$CBO_p(c_s) = CBO_b(c_s) - l_k + d_k \quad (5)$$

$$CBO_p(c_t) = e_k + r_k \quad (6)$$

### C. Prediction Functions for Extract Class

EC refactoring is performed on a source class $c_s$ when it is large and implements more than one responsibility [1]. In such cases, a set of fields $\{a_k\}$ and/or methods $\{m_h\}$ are moved to one (or more) newly created class $c_t$. EC consists of several Move Field and/or Move Methods refactorings, and the creation of a field referencing the new class in the source class. The following prediction functions consider the case in which, before moving each field, the Self-Encapsulate refactoring is performed on them. Moreover, the methods to be moved are removed from the source class and the fields in the new class are Self-Encapsulated as well.

**Impact on CBO.** Equations 7 and 8 compute the impact on CBO. In the case of the source class $c_s$, the metric is positively impacted by the new field referencing the new class and negatively impacted by $d$, the number of classes used by all the moved methods $\{m_h\}$ and not referenced in the other methods of class $c_s$. In the case of the target class $c_t$, the metric is computed as the sum of $e$, the number of classes referenced by the methods $\{m_h\}$, and $r$, which indicates whether the new class is coupled with the source class ($r = 1$) or not ($r = 0$). If the target class is coupled with the source class, then it means some method (in the target class) is referencing a field or method of the source class.

$$CBO_p(c_s) = CBO_b(c_s) + 1 - d \tag{7}$$

$$CBO_p(c_t) = e + r \tag{8}$$

**Impact on LCOM5.** Equations 9 and 10 are the prediction functions for LCOM5. Let $n_2 = |\{m_h\}|$ be the number of methods to be moved. For the source class, the prediction function corresponds to Equation 1 with the predicted value of each (sub)expression. Since $n_1$ fields are removed and one additional field is created, the predicted number of attributes is defined as $NA_p = NA_b - n_1 + 1$. The predicted number of methods is affected by $n_2$ methods removed and $2n_1$ added methods, thus, $NM_p = NM_b - n_2 + 2n_1$. The predicted total method accesses is defined as $TMA_p = TMA_b - \sum_k T_k - \sum_h Q_h + 2n_1 + t$, where $T_k$ is the number of methods of $c_s$ accessing the field to move $a_k$, and $Q_k$ is number of fields of $c_s$ accessed by method $m_k$. $2n_1$ methods (the getters and setters of each field) will be accessing the field referencing the target class and $t$ is the number of methods of class $c_s$ referencing any field or method to be moved.

$$LCOM5_p(c_s) = \frac{TMA_p/NA_p - NM_p}{1 - NM_p} \tag{9}$$

As for the target class $c_t$, the predicted number of attributes is $NA_p = n_1$ and the predicted number of methods is $NM_p = 2n_1 + n_2$, i.e., the setter a getter of each field $a_k$, and the number of methods to be moved. Finally, the predicted total field accesses is $TMA_p = 2n_1 + \sum_h V_h$, i.e., each field is used by its getter and setter, and for each method $m_h$, $V_h$ is the number of fields in $\{a_k\}$ accessed by the method.

$$LCOM5_p(c_t) = \frac{(2n_1 + \sum_h V_h)/(n_1) - (2n_1 + n_2)}{1 - (2n_1 + n_2)} \tag{10}$$

## IV. Empirical Evaluation

The *purpose* of the empirical study we conducted is to evaluate RIPE's accuracy in estimating the impact of refactoring operations on code metrics. The *context* consists of 15 software systems (the list of systems and refactorings is available online at http://www.cs.wayne.edu/~severe/ripe). The study aims at answering the following research question:

**RQ**: *What is RIPE's accuracy in estimating the impact of refactoring operations on code metrics?*

To answer the **RQ** we need an oracle against which to compare RIPE, specifically: (i) a set of refactorings for each of the 12 refactoring operations supported by RIPE; and (ii) the actual difference between pre- and post- refactoring of the 11 code metrics predicted by RIPE.

### A. Refactoring Operations Collection

We collected a set of *manually seeded* refactorings operations (referred to as *seeded refactorings*) and another set of *existing* ones (referred to as *existing refactorings*) on the object systems. The purpose of the *seeded refactorings* is to have a uniform distribution of refactoring instances across the different kinds of refactorings, while the purpose of the *existing ones* is to validate the proposed approach on data that is based on everyday typical changes.

**Seeded Refactorings Collection.** We asked two Ph.D. students to manually perform five to ten refactorings for each of the 12 refactoring operations on a version of two software systems (i.e., ArgoUML and aTunes) and they performed a total of 173 refactorings (60 on aTunes and 113 on ArgoUML). Both students have a good knowledge of refactoring and bad smells, gained during their industry experience and graduate courses. We did not provide the students with specific places in the code where to implement the refactorings, but instead asked them to randomly identify these locations such that the following two constraints are fulfilled: *(i)* a class *c* of the system must not be involved in more than one refactoring; *(ii)* all changes needed to ensure the preservation of the external behavior must be performed.

**Existing Refactorings Collection.** We also collected existing refactorings from the remaining 13 systems, creating the *existing refactorings* data set. We built a tool using a code analyzer developed in the MARKOS European project (http://www.markosproject.eu), which is able to identify instances of the 12 refactorings supported by RIPE. Given the log information extracted from the versioning system, the code analyzer parses the source code after each commit and extracts a set of facts about the changes performed by the developer in a specific commit. Since our tool is based on heuristics and it is not 100% precise, we manually validated the refactoring returned by it on the systems on which it was executed.

### B. Evaluation Procedure

We adopted the following process to assess RIPE's accuracy. For each refactoring $r_i$, we measured the 11 code metrics supported by RIPE before and after its application (our oracle). Then, we used RIPE to predict the post-refactoring value of

| Ref. | Metr. | RFC | | CBO | | DAC | | MPC | | LOC | | NOM | | CYCLO | | LCOM2 | | LCOM5 | | NOC | | DIT | | TOTAL | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* | *Seed.* | *Exist.* |
| **EM** | *Accur* | 7% | x | - | - | - | - | - | - | 73% | x | 93% | x | 93% | x | 80% | x | 80% | x | - | - | - | - | 71% | x |
| | *Med* | 3% | x | - | - | - | - | - | - | 0% | x | 0% | x | 0% | x | 0% | x | 0% | x | - | - | - | - | 0% | x |
| | *Avg* | 3% | x | - | - | - | - | - | - | 0% | x | 2% | x | 0% | x | 3% | x | 2% | x | - | - | - | - | 2% | x |
| **IM** | *Accur* | 0% | x | - | - | - | - | - | - | 60% | x | 100% | x | 67% | x | 87% | x | 87% | x | - | - | - | - | 67% | x |
| | *Med* | 2% | x | - | - | - | - | - | - | 0% | x | 0% | x | 0% | x | 0% | x | 0% | x | - | - | - | - | 0% | x |
| | *Avg* | 4% | x | - | - | - | - | - | - | 3% | x | 0% | x | 3% | x | 0% | x | 0% | x | - | - | - | - | 2% | x |
| **RMMO** | *Accur* | 50% | x | 20% | x | 53% | x | 80% | x | 23% | x | 80% | x | 70% | x | 33% | x | 33% | x | 100% | x | - | - | 54% | x |
| | *Med* | 1% | x | 33% | x | 0% | x | 0% | x | 6% | x | 0% | x | 0% | x | 14% | x | 21% | x | 0% | x | - | - | 0% | x |
| | *Avg* | 54% | x | 50% | x | 196% | x | 94% | x | 47% | x | 17% | x | 35% | x | 23% | x | 31% | x | 0% | x | - | - | 55% | x |
| **PUF** | *Accur* | - | - | 88% | 25% | 29% | 75% | - | - | 50% | 13% | - | - | - | - | 94% | 25% | 94% | 25% | - | - | - | - | 71% | 33% |
| | *Med* | - | - | 0% | 5% | 14% | 0% | - | - | 1% | 4% | - | - | - | - | 0% | 27% | 0% | 55% | - | - | - | - | 0% | 4% |
| | *Avg* | - | - | 2% | 26% | 28% | 25% | - | - | 1% | 24% | - | - | - | - | 0% | 38% | 0% | 52% | - | - | - | - | 6% | 33% |
| **PUM** | *Accur* | 30% | 2% | 80% | 83% | - | - | 73% | 79% | 43% | 5% | 50% | 2% | 50% | 7% | 60% | 52% | 67% | 38% | - | - | - | - | 57% | 34% |
| | *Med* | 4% | 12% | 0% | 0% | - | - | 0% | 0% | 1% | 16% | 2% | 13% | 1% | 12% | 0% | 0% | 0% | 1% | - | - | - | - | 0% | 6% |
| | *Avg* | 18% | 20% | 6% | 5% | - | - | 3% | 7% | 9% | 25% | 28% | 27% | 13% | 24% | 12% | 12% | 11% | 11% | - | - | - | - | 13% | 16% |
| **PDF** | *Accur* | 53% | 0% | 19% | 0% | 94% | 33% | - | - | 57% | 0% | 89% | 0% | 87% | 0% | 89% | 83% | 89% | 83% | - | - | - | - | 72% | 25% |
| | *Med* | 0% | 10% | 9% | 11% | 0% | 3% | - | - | 0% | 7% | 0% | 9% | 0% | 9% | 0% | 0% | 0% | 0% | - | - | - | - | 0% | 7% |
| | *Avg* | 9% | 19% | 12% | 10% | 2% | 3% | - | - | 2% | 8% | 1% | 22% | 0% | 14% | 1% | 1% | 1% | 1% | - | - | - | - | 3% | 10% |
| **PDM** | *Accur* | 60% | 0% | 82% | 0% | - | - | 77% | 0% | 84% | 0% | 84% | 0% | 86% | 0% | 91% | 45% | 91% | 45% | - | - | - | - | 82% | 11% |
| | *Med* | 0% | 7% | 0% | 11% | - | - | 0% | 6% | 0% | 5% | 0% | 8% | 0% | 8% | 0% | 1% | 0% | 1% | - | - | - | - | 0% | 6% |
| | *Avg* | 6% | 7% | 5% | 11% | - | - | 6% | 7% | 4% | 8% | 4% | 10% | 3% | 9% | 1% | 1% | 0% | 1% | - | - | - | - | 4% | 7% |
| **RDI** | *Accur* | - | - | - | - | 100% | x | - | - | 55% | x | - | - | - | - | 45% | x | 45% | x | 91% | x | 100% | x | 73% | x |
| | *Med* | - | - | - | - | 0% | x | - | - | 0% | x | - | - | - | - | 2% | x | 2% | x | 0% | x | 0% | x | 0% | x |
| | *Avg* | - | - | - | - | 0% | x | - | - | 2% | x | - | - | - | - | 13% | x | 13% | x | 5% | x | 0% | x | 5% | x |
| **RID** | *Accur* | - | - | - | - | 100% | x | - | - | 67% | x | - | - | - | - | 58% | x | 50% | x | 100% | x | 100% | x | 79% | x |
| | *Med* | - | - | - | - | 0% | x | - | - | 0% | x | - | - | - | - | 0% | x | 3% | x | 0% | x | 0% | x | 0% | x |
| | *Avg* | - | - | - | - | 0% | x | - | - | 11% | x | - | - | - | - | 8% | x | 14% | x | 0% | x | 0% | x | 6% | x |
| **EC** | *Accur* | 17% | 1% | 63% | 10% | 68% | 29% | 37% | 9% | 23% | 4% | 27% | 14% | 27% | 6% | 23% | 18% | 20% | 17% | - | - | - | - | 34% | 12% |
| | *Med* | 18% | 40% | 0% | 27% | 0% | 50% | 8% | 47% | 11% | 35% | 45% | 20% | 14% | 44% | 14% | 30% | 15% | 28% | - | - | - | - | 11% | 35% |
| | *Avg* | 35% | 91% | 20% | 69% | 11% | 52% | 22% | 127% | 27% | 124% | 79% | 68% | 56% | 78% | 19% | 68% | 24% | 78% | - | - | - | - | 33% | 85% |
| **MF** | *Accur* | 48% | 3% | 38% | 41% | 97% | 16% | 79% | 21% | 83% | 5% | 97% | 9% | 97% | 3% | 97% | 12% | 97% | 12% | - | - | - | - | 81% | 14% |
| | *Med* | 2% | 19% | 7% | 8% | 0% | 27% | 0% | 24% | 0% | 18% | 0% | 26% | 0% | 19% | 0% | 9% | 0% | 6% | - | - | - | - | 0% | 17% |
| | *Avg* | 3% | 46% | 11% | 29% | 1% | 67% | 2% | 85% | 0% | 41% | 0% | 42% | 0% | 39% | 0% | 31% | 0% | 28% | - | - | - | - | 2% | 46% |
| **MM** | *Accur* | 50% | 11% | 53% | 41% | - | - | 67% | 27% | 57% | 8% | 90% | 11% | 77% | 11% | 90% | 48% | 87% | 49% | - | - | - | - | 71% | 26% |
| | *Med* | 1% | 19% | 0% | 10% | - | - | 0% | 19% | 0% | 30% | 0% | 25% | 0% | 24% | 0% | 1% | 0% | 1% | - | - | - | - | 0% | 15% |
| | *Avg* | 4% | 36% | 7% | 29% | - | - | 4% | 57% | 2% | 38% | 1% | 36% | 9% | 40% | 1% | 18% | 1% | 18% | - | - | - | - | 3% | 34% |
| **TOTAL** | *Accur* | 43% | 7% | 57% | 38% | 75% | 23% | 70% | 26% | 58% | 6% | 79% | 10% | 75% | 9% | 75% | 38% | 74% | 37% | 97% | x | 100% | x | 68% | 22% |
| | *Med* | 2% | 19% | 0% | 10% | 0% | 27% | 0% | 17% | 0% | 25% | 0% | 22% | 0% | 24% | 0% | 2% | 0% | 2% | 0% | x | 0% | x | 0% | 14% |
| | *Avg* | 14% | 43% | 12% | 32% | 35% | 59% | 18% | 66% | 9% | 47% | 13% | 39% | 12% | 42% | 6% | 26% | 7% | 26% | 2% | x | 0% | x | 12% | 41% |

Table I: RIPE's prediction accuracy ($Accur$), median ($Med$) and average deviation ($Avg$) for each refactoring operation, quality metric, and data set: *seeded* ($Seed.$) and *existing* ($Exist.$). The dashes "-" represent no change in the metric (thus, no prediction function defined) and the exes "x" represent no data available (only for the *existing refactorings*).

the metrics after the application of $r_i$, given the pre-refactoring values. The metrics measurement and prediction have been done with the tool we built. We assess the accuracy of RIPE by comparing the actual changes in the code metric values with the predicted ones. For each of the 11 considered code metrics, $M_k$, we compute for each refactoring operation $r_i$ the percentage deviation of RIPE's change prediction:

$$dev_{\%}(r_i, M_k) = \frac{|actual(M_k) - predicted(M_k)|}{actual(M_k)}$$

where $actual(M_k)$ is the actual change of the value of metric $M_k$ observed after applying $r_i$, while $predicted(M_k)$ is the $M_k$ change predicted by RIPE for the application of $r_i$. We report descriptive statistics of the $dev_{\%}$ achieved by RIPE for all the 12 refactoring operations and 11 metrics, as well as the percentage of metric values correctly predicted, i.e., the prediction accuracy ($Accur$). $Accur$ is the ratio between the number of metric values perfectly predicted (i.e., $dev_{\%} = 0\%$) and the total number of metric values. Also, we analyze the average ($Avg$) and median ($Med$) of $dev_{\%}$.

### C. Results and Discussion

In total, RIPE predicted 8,103 metric values, 2,903 for the *seeded refactorings* and 5,200 for the *existing* ones (Table I). As expected, the predictions for the *seeded refactorings* have higher accuracy (68%) than for the *existing refactorings* (22%). While the *seeded refactorings* have been introduced in the system in isolation, the *existing* refactoring are often accompanied by other changes performed by developers in conjunction with the refactoring, which negatively impact RIPE's predictions. However, when RIPE's prediction is not 100% accurate, it achieves a 0% median $dev_{\%}$ (12% avg) for *seeded refactorings* and 14% (41% avg) for *existing* ones.

In order to have an overview of RIPE's overall performance, we also computed the evaluation metrics on the entire data set (i.e., *seeded* plus *existing refactorings*). We obtained a 38% accuracy (i.e., 3,102 exact metric predictions), with 5% median (31% average) $dev_{\%}$. The gap between the median and average values indicates the presence of negative outliers (i.e., predictions with low accuracy and high deviation).

**Detailed Analysis.** Table I reveals that, among the *seeded refactorings*, DIT, No. of Children (NOC) and No. of Methods (NOM) are predicted with the highest accuracy: 100%, 97%, and 79%, respectively. These metrics quantify information at rather coarse-grained granularity, hence the net changes are usually small. Also, there is a little or no ambiguity in how a refactoring can impact such metrics (e.g., there is only one way to modify the inheritance relationship between classes). The predictions for Data Abstraction Coupling (DAC), CYCLO, LCOM2/5, and Message Passing Coupling (MPC) are very accurate as well (i.e., $Accur > 70\%$).

The rest of the metrics are predicted with an accuracy below 60%. For example, RFC has the lowest prediction accuracy (43%), followed by CBO (57%), and LOC (58%). Table I reveals that EC and RMMO are the refactoring operations for which the predictions are most difficult. After manually inspecting the refactorings in the target systems, we found that the refactoring impact was harder to predict because the students considered that some changes, stated in Fowler's book and assumed by the prediction functions, were not needed.

As for the *existing refactorings*, CBO and LCOM2/5 are the metrics with the highest prediction accuracy (~38%). The metrics with the lowest prediction accuracy are LOC, RFC, and CYCLO *(Accur* around 7%). Push Down Field (PDF) and Push Down Method (PDM) refactorings had a noticeable negative influence on these metrics, as well as on CBO (see Table I). A manual inspection of these refactorings revealed that there were multiple refactorings of the same type (e.g., PDFs) performed on the same classes.

In the case of *seeded refactorings*, the operations with the highest prediction accuracy are PDM, MF, and Replace Inheritance with Delegation - RID (~80%). For these refactorings there are not many implementation alternatives, so the impact on the metrics should be easy to predict. We investigated why these refactoring were not 100% accurate, by analyzing the cases with the largest prediction deviation. RIPE's analysis component is rather conservative in its current form, hence it sometimes misidentified specific program components. For example, methods being called in some class that in reality correspond to the same method were accounted as different methods. Nonetheless, the median and average deviation for these refactoring is very low (0% for the median and about 5% for the average), which indicates an excellent accuracy.

| Metric | Avg $dev_{\%}$ | Argo. Avg metric | Argo. Metr. Dev. |
|---|---|---|---|
| RFC | 25% | 23 | 6 |
| CBO | 29% | 6 | 2 |
| DAC | 142% | 2 | 3 |
| LOC | 21% | 64 | 13 |
| NOM | 63% | 7 | 4 |
| CYCLO | 48% | 16 | 8 |
| LCOM2 | 23% | 0.30887 | 0.071 |
| LCOM5 | 27% | 0.38662 | 0.104 |

Table II: Average metrics deviation for ArgoUML.

Our analysis of $dev_{\%}$ reveals that there are cases with high accuracy, yet also high prediction deviation (e.g., NOM or DAC). While we normalized the $dev_{\%}$ values presented in the paper, we must note that the metrics have values on different scales. More than that, some measures are rather coarse and easy to interpret (e.g., NOM), whereas others are more fine grained and less intuitive (e.g., LCOM5). Hence, 10% deviation, for example, may have a slightly different meaning for different metrics. To better understand these differences we give as example the partial deviation data for one of the systems with seeded refactorings, i.e., ArgoUML. Table II shows the average prediction $dev_{\%}$ ($Avg$) of the metrics for the *seeded refactorings* (2nd column), the average metric values for ArgoUML (3rd column), as well as the metric value deviation for this system (4th column). The $Avg$ for DAC (i.e., the number of attributes having different type than the class they belong to) is the highest (142%). While that may seem like a large $dev_{\%}$, in practice, for ArgoUML, it means a deviation of only 3 attributes. A similar situation occurs for NOM, which corresponds to a deviation of only 4 methods (i.e., on average, when RIPE is not accurate, its predicted NOM values are larger/lower than the actual value by 4).

## V. Conclusions and Future Work

Our approach, named RIPE, includes a set of atomic, independent and reusable functions that predict the impact of 12 refactoring operations on 11 code quality metrics, for the classes involved in the refactoring, before this is implemented. RIPE's empirical evaluation using 504 refactorings from 15 Java open source systems showed good prediction performance. RIPE perfectly predicted 38% of 8,103 metric values, with a low median deviation from the actual metric value (i.e., 5%). Using RIPE, developers can assess refactoring opportunities in their everyday maintenance tasks, since it allows them to compare specific code metric changes caused by the refactoring operations, especially when refactorings include numerous transformations (e.g., Extract Class) and different metrics that measure conflicting properties of the code (e.g., Cohesion vs. Coupling).

As for future work, we will improve our prediction functions to consider more implementation alternatives for refactorings where RIPE's prediction accuracy is lower. We will include more metrics and refactoring operations as well as more empirical studies. Finally, we plan to move towards measuring and predicting the quality impact of sequences of refactoring operations by composing the prediction functions.

## Acknowledgments

We wish to thank the Ph.D. students that participated in the empirical study. This research was supported in part by grants from NSF (CCF-1017263 and CCF-0845706).

## References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[2] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When Does a Refactoring Induce Bugs? An Empirical Study," in *Proceeding of the International Working Conference on Source Code Analysis and Manipulation*, pp. 104–113, 2012.

[3] M. Kim, T. Zimmermann, and N. Nagappan, "An Empirical Study of Refactoring Challenges and Benefits at Microsoft," *IEEE Trans. on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2014. (to appear).

[4] Q. Soetens and S. Demeyer, "Studying the Effect of Refactorings: A Complexity Metrics Perspective," in *Seventh Int. Conf. on the Quality of Information and Communications Technology*, pp. 313–318, 2010.

[5] B. Du Bois and T. Mens, "Describing the impact of refactoring on internal program quality," in *International Workshop on Evolution of Large-scale Industrial Software Applications*, pp. 37–48, 2003.

[6] B. Du Bois, *A Study of Quality Improvements by Refactoring*. PhD thesis, Universiteit Antwerpen, 2006.

[7] E. K. Piveta, *Improving The Search For Refactoring Opportunities on Object-Oriented And Aspect-Oriented Software*. PhD thesis, Universidade Federal do Rio Grande do Sul, 2009.

[8] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)," *Object Oriented Systems*, vol. 3, pp. 143–158, 1996.

[9] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Soft. Eng.*, vol. 20, pp. 476–493, Jun 1994.