An Exploratory Study on Extract Method *Floss-Refactoring*

Jaziel S. Moreira
Federal University of Campina Grande
Campina Grande, Paraiba, Brazil
jaziel@copin.ufcg.edu.br

Everton L. G. Alves
Federal University of Campina Grande
Campina Grande, Paraiba, Brazil
everton@computacao.ufcg.edu.br

Wilkerson L. Andrade
Federal University of Campina Grande
Campina Grande, Paraiba, Brazil
wilkerson@computacao.ufcg.edu.br

ABSTRACT

As a software evolves its code requires constant updating. In this sense, refactoring edits aim at improving structural aspects of a code without changing its external behavior. However, studies show that developers tend to combine in a single commit refactorings and behavior-changing edits (extra edits) – *floss-refactoring*. Floss-refactorings can be error-prone and require careful handling. However, little has been done to understand how refactorings and extra edits relate in practice. In this work, we propose a strategy for extracting floss-refactoring data. Moreover, we mine code repositories of 16 open-source projects and analyse commits with floss refactoring related to *Extract Method*. Our results show that developers often combine *Extract Method* with inner method extra edits (e.g., statement insert), with an expected increase of 8-16% of extra edits by each *Extract Method*. Moreover, some statements are more likely to be changed depending on the extra edit performed.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; *Maintaining software*; Software version control;

KEYWORDS

Refactoring, Extract Method, Floss Refactoring, Empirical Study

ACM Reference Format:
Jaziel S. Moreira, Everton L. G. Alves, and Wilkerson L. Andrade. 2020. An Exploratory Study on Extract Method *Floss-Refactoring*. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341105.3373893

1 INTRODUCTION

Refactoring is the act of applying behavior-preserving edits for avoiding current code issues and help software maintenance [6]. It is often related to improving aspects such as design quality, code readability, and reducing bad smells. Studies have shown that refactoring can even reduce maintenance costs [10].

Nearly 30% of all edits are refactoring related [21]. However, despite its good intentions, refactoring edits are known to be error-prone. Several studies mention a strong correlation between the timing and location of refactorings and bug fixes [2, 11]. A field study with Microsoft developers [12] showed that 77% of the survey participants perceive refactoring as a risk of introducing subtle bugs and functionality regression. This risk can be even higher, since 90% of refactorings are manually performed [15]. Nevertheless, even refactoring tools may lead to unsafe refactoring [22].

Even simple refactorings, such as an *Extract Method* can be tricky. An *Extract Method* refactoring encompasses the extraction of a code fragment into a separate method. It is known to be one of the most widely performed refactoring since it can be used for fixing a variety of code bad smells [19]. However, subtle faults can be introduced when an *Extract Method* is intended. For instance, the newly created method may add/break override/overload contract. Bavota et al. [2] found that the *Extract Method* refactoring often induces bug fixes.

On top of all, studies have shown that, in practice, developers often interweave refactoring and non-refactoring edits - *floss-refactoring* [14, 19]. By combining behavior changing edits alongside refactorings (e.g., an *Extract Method* and new features), a developer might increase the likelihood of introducing faults. For instance, Silva et al. [19] stated that developers often perform *Extract Method*s for avoiding code replication. After refactoring, the newly created method is then called in several places. However, suppose a developer, aiming at attending a requirement from one of the callers, in the same commit, introduces a new condition checking in the newly added method (floss-refactoring). If this extra edit is performed without the proper impact analysis, it may negatively impact the rest of the callers.

Refactoring-aware code revision is an emerging topic that has gained notoriety due to its practical benefits [1, 3, 8]. Due to its complexity, and possible impact, developers should review their edits considering the refactorings performed. The goal is to both detect possible faults, and/or confirm their intentions. This need is even more evident on floss-refactoring, since different concerns may be mixed in a single commit. Alves et al. [1] discussed two types of refactoring anomalies that require revision: *missing* and *extra edits*. The latter are the edits that go beyond a pure refactoring transformation (floss-refactoring). Therefore, even intentional extra edits require special attention.

However, little has been done on characterizing extra edits on floss-refactoring in practice. We believe that, by better understanding how developers often relate refactoring to extra edits, advances can be done to assist refactoring related activities, such as refactoring fault detection, refactoring revision, and to guide developers on when systematic floss-refactoring. For instance, Kim et al. [11] pointed out that a support for floss-refactoring is needed, due to

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373893

frequent errors. The current tool support in this sense is quite limited by focusing only on refactoring detection (e.g., [18, 20, 23]), or on highlighting extra edit between two versions of a code (e.g., [1, 8]).

In this paper, we present a strategy for extracting floss-refactoring data and run an exploratory study designed for characterizing *Extract Method* floss-refactoring. We mined repositories of 16 open-source projects and analyzed commits that combine refactorings and non-refactoring edits. Our main findings show that, in a single commit, developers often combine *Extract Method* with inner method extra edits (e.g., statement insert), with an expected increase of 8-16% of extra edits by each *Extract Method*. Moreover, some statements are more likely to be changed depending on the extra edit performed, such as *Method Invocation*s and *If Statement*s. Our study does not focus on whether such extra edits induce faults. However, our findings can help the development of new solutions/-tools for refactoring review, validation, and/or systematic analysis. Moreover, such information can also help other researches. Empirical investigations tend to use random code changes to simulate real refactoring faults [1]. By discovering how developers combine refactorings with other edits in the real world, researchers can then properly emulate floss-refactoring faults.

2 MOTIVATING EXAMPLE

To exemplify the issues related to *Extract Method* floss-refactoring, consider the code in Figure 1[1] from the BroadleafCommerce project repository[2][3]. BroadleafCommerce is an e-commerce framework designed for facilitating the development of enterprise and commerce-driven websites. Figure 1 shows the *persistCopyObjectTree* method, which is responsible for persisting the *copy* object provided as parameter.

In a single commit, a series of edits were performed. Code insertions are marked with '+', deletion with '–', extra edits are underlined. As we can see, an *Extract Method* edit was performed: the *If Statement*, along with its body (Figure 1(a) - lines 8-10), were extracted to the *persistPart* method (Figure 1(b) - lines 16-22). However, in the same commit, a series of other extra edits were included: a new persistence strategy was introduced, the method signature and some condition expressions were updated.

By analysing the *persistCopyObjectTree* method, we can see that there is a big difference in how the system stores the object. Previously, the method saves the object itself (Figure 1(a) - line 6), in its new version it stores only the hash code (Figure 1(b) - line 6). As a consequence, the verification of the method was changed to check the hash code (Figure 1(b) line 3). Even though those extra edits might be intentional, they may lead to subtle faults and are worth double-checking. For instance, the developer could have forgotten to update some verification dealing with the *library* throughout the code. This fault would not throw an exception since the *contains* method compares instances of *Object*. Therefore, an unchanged verification may pass unnoticed. Moreover, the cache verification at the end of the source method (Figure 1(a) - line 11) was deleted. A

```
1  persistCopyObjectTree(Object copy, Set library,
2  Context context){
3    if (library.contains(copy)) {
4      return;
5    }
6    library.add(copy);
7    ...
8    if (!service.sessionContains(copy)) {
9      [if body]
10   }
11   context.checkLevel1Cache();
12 }
```

(a) original code

```
1  persistCopyObjectTree(Object copy, Set<Integer>
2  library, Context context){
3      if (library.contains(identityHashCode(copy)))  {
4        return; }
5    library.add(identityHashCode(copy));
6      ...
7  -   if (!service.sessionContains(copy)) {
8  -     [if body]
9  +   if (copy.getAnnotation() == null) {
10 +     persistPart(copy, context);
11     }
12 -   context.checkLevel1Cache();
13 }
14
15 persistPart(final Object copy, Context context) {
16     if (!service.sessionContains(copy) &&
17 +       !service.idAssigned(copy))  {
18 +     runOperationByIdentifierThread(
19         [if body]);
20     }
21 }
```

(b) Extract Method refactoring with extra edits

Figure 1: Example of floss refactoring.

refactoring-aware revision would require the developer to confirm the intention of these extra edits and carefully analyze their impact.

The extracted method also includes several extra edits. The condition expression was updated to verify the object's ID (Figure 1(b) - line 18). With this edit, some of the objects that previously would be persisted by this method may not pass the new verification. Moreover, the extracted statements now run in a particular thread (Figure 1(b) - line 19). This extra edit alone, if not well treated, might open the system to a set of unforeseen faults related to concurrent programming.

In summary, the presented commit was clearly a Extract Method floss-refactoring. Several extra edits were applied along with the *Extract Method* (i.e., statement insert, statement delete, statement update, and condition expression change), which increased the likelihood of fault introduction. By analyzing the relationship between refactoring and extra edits, we intend to help researchers/developers to better understand how developers perform floss-refactoring in the real world, improving code review activities, and, therefore, provide new solutions to minimize its error-proneness.

3 FLOSS-REFACTORING EXTRACTION STRATEGY

To work with real-work data, we decide to mine software repositories. For that, we propose an automatic approach for extracting floss-refactoring. Our approach comprises three steps and is summarized in Figure 2:

For each project *P*:

[1]This code has been adapted for didactic purpose
[2]https://github.com/BroadleafCommerce/BroadleafCommerce
[3]Commit: a270461a25509c9b6bfc2396e416ef4f58d9a4ce

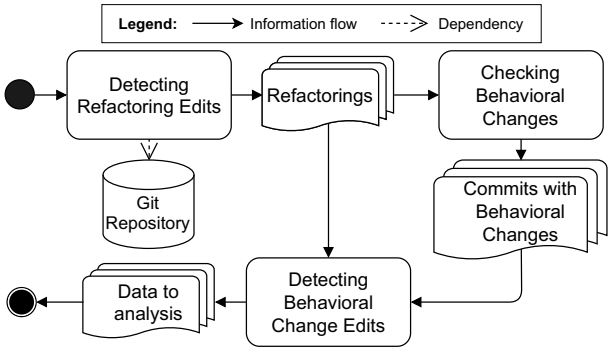

Figure 2: Data Extraction Process

(1) From P's repository, get SC_r, where $SC_r = sc1, sc2, ..., scn$ and sci is a pair of sequential commits that includes at least one refactoring edit;
(2) From SC_r, find SC_e, where SC_e is a subset of SC_r containing only non-behavior-preserving pairs of commits, e.g., commits that include extra edits;
(3) For each sci in SC_e, decompose the change edits and analyze which edits are part of the refactoring performed in the commit, and which edits are extra.

The first activity consists of filtering commits that include refactoring edits. For that, we use the RefDiff tool [20]. RefDiff is a tool that detects refactorings between two versions of a Java program. It supports 13 of the most common refactoring types [14]. Regarding RefDiff accuracy, we refer to its reported precision (100%) and recall (88%) [20].

Although there are other tools for detecting refactorings (e.g., [18, 23]), we chose RefDiff due to its high precision and because it works directly on software repositories. The RMiner [23] tool also works on software repositories and claims to have higher accuracy. However, by the time of its release, our study was in a later stage. Nevertheless, we ran a manual validation using randomly selected samples of our dataset. This analysis concluded that, in the context of our study, both tools (RefDiff and RMiner) provided similar accuracy.

Then, we filter the commits with behavior changes, i.e., pairs of commits that are not pure-refactoring. For that, we use the SafeRefactor tool [22], which checks for safe refactoring edits. It analyzes two versions of a program and, by generating and running sets of unit tests, it indicates whether the program's behavior remain unchanged. In the context of our study, only pairs of commits that *failed* in SafeRefactor's analysis were used, since they mean refactoring edits combined with behavior-changing ones. On the other hand, *pass* outputs were discarded.

In the third step, we analyze the extra edit types from commits that combine refactorings with behavior changes. For that, we use the ChangeDistiller tool [5]. ChangeDistiller is a tool that compares two versions of an Abstract Syntax Tree (AST) and classifies the source code edits performed between them. Although other tools perform a similar activity (e.g. [4]), we opt for using ChangeDistiller due to its fine granularity and feedback. Moreover, other change-based empirical studies also use it [1, 11, 23].

```
1  EMFilter (m1:original Method, m2: extracted Method){
2     l1 = changes in m1
3     l2 = staments from m2
4     changes = {}
5     foreach s in l1{
6         if(l2.contains(s)){
7             l1.remove(s)
8             l2.remove(s)} }
9     updates = {}
10    foreach s1 in l1{
11        foreach s2 in l2{
12            if (similarity(s1,s2) >= 0.6){
13                l1.remove(s1)
14                l2.remove(s2)
15                update = newUpdate(s1,s2)
16                updates.add(Distiller.classify(update)) } } }
17    inserts = {}
18    foreach s in l2
19        inserts.add(Distiller.classify(s))
20    changes.addAll(updates)
21    changes.addAll(l1)
22    changes.addAll(inserts)
23    foreach s in changes{
24        if (isCaller(s) OR isSimpleReturn(s))
25                changes.remove(s) }
26 }
```

Listing 1: Extract method filter algorithm

Among ChangeDistiller edit type classification, we can list: *Statement Insert*, *Statement Delete* and *Statement Update*, which correspond to the introduction, removal and update of statements from a method's body, respectively. A *Statement Parent Change* represents a statement that was moved to a different code structure, while a *Condition Expression Change* represents that a condition from loop/control that was updated. As for outer-method edits we can list: *Additional Object State*, *Additional Functionality* and *Removed Functionality*. The first corresponds to the introduction of a new field to a class, while the other two corresponds to the introduction and removal of a method, respectively.

ChangeDistiller receives two versions of a given class (v1 and v2) and lists all edits performed between them. However, since our study focuses on extra edits, we needed a way to filter ChangeDistiller's output list in order to remove edits that were part of refactorings. For instance, when a *Extract Method* is performed, ChangeDistiller classifies the extracted method as an *Additional Functionality*, which is not an extra edit. Therefore, we extended ChangeDistiller to include a module that analyzes both ASTs and ChangeDistiller's output list and, based on the refactorings returned by RefDiff, filter it and returns the edits that are not part of refactoring changes. For filtering extra edits, we first check whether each edit returned by ChangeDistiller is related to any refactoring returned by RefDiff. When a refactoring related edit is found, a specific treatment is applied based on the refactoring type it is related to.

We defined filtering algorithms for each refactoring type. Listing 1 shows the pseudo-code of the algorithm used for filtering *Extract Method*-related edits. For each *Extract Method* edit found by RefDiff, we run EMFilter to select the extra edits applied in the extracted and source method. The changes classified as part of the refactoring (not an extra edit) are excluded from the edit list, remaining only extra edits.

First, the algorithm tries to match identical statements between the statements excluded from the source method and the ones

present in the extracted method (lines 2 - 8). The matches found in the first step are considered part of a refactoring, and, therefore, excluded from the edit list. Then, it identifies the statements that were updated during the refactoring process by verifying its similarity (lines 9 - 16). We used the N-Gram similarity function with a threshold of 60%, which is the same applied by ChangedDistiller. The selected statements are then removed from their previous lists and added as an update to a different list. The remaining unmatched statements are considered extra edits. To determine the edit type of the remaining statements from the extracted method, we also used the ChangeDistiller's classifier (lines 17 - 19). As for the remaining changes from the source method, they were added to the final edit list, as well as the updated statements and the new statements from the extracted method (lines 20 - 22). Finally, we verify if any of the final edits reference any refactored entity. When an edit refers to a refactored entity, we consider the edit as part of the refactoring, since update references is a step in the refactoring process (lines 23 - 26).

Another adaptation was needed. Since ChangeDistiller runs over two versions of a single class, it was necessary to map the classes to its respective pair across the system versions. Thus, to address renamed and moved classes, we mapped the previous and new signatures based on the *Rename Class* and *Move Class* refactorings found by RefDiff. Details about our strategy and its implementation can be found in our website[4].

Besides extracting floss-refactoring, we believe this approach can also be used to separate spotted commits during review activities, to improve versioning control by, for instance, warning developers on such commits, and to help researchers conduct empirical research in online repositories.

4 STUDY DESIGN

By using our data extraction process, we ran an empirical study to help us understand how developers perform Extract Method floss-refactoring. This section presents details about the experimental design, its goals, research questions, and procedure.

4.1 Objective

We designed an empirical study for analyzing how developers combine *Extract Method* refactorings with extra edits. For that, we mined commits from GIT software repositories searching for association patterns between refactoring and extra edits types. To guide our investigation, we established the following research questions (RQ):
- **RQ1:** Which extra edits are more likely to appear alongside *Extract Method* refactorings?
- **RQ2:** How the number of extra edits increases based on the frequency of the *Extract Method* refactoring?
- **RQ3:** How Extract Method Floss-Refactorings Are Performed?

4.2 Subject Selection

We mined 16 Java projects and extracted the extra edits data using the strategy presented in Section 3. Table 1 lists the projects used, and summarizes its characteristics. As we can see, the number of extra edits is much higher than refactorings. Thus, we could say that refactorings appeared to facilitate or supplement extra edits, which corroborates Silva et al.'s conclusions [19]. Figure 3(a) shows the frequency of each refactoring type considering all analyzed commits. As we can see, *Extract Method* refactorings were among the most common. Moreover, the refactoring types found in the projects used in our study go along with Murphy-Hill et al.'s findings regarding common refactorings [14].

Table 1: Projects (LOC - Lines of Code, TC - Total Commits, SC - Selected Commits, REF - Refactorings, EE - Extra Edits)

Project	LoC	TC	SC	REF	EE
codefollower/Lealone	100789	1255	175	2249	8601
nutzam/nutz	92671	5883	59	153	2798
AsyncHttpClient/async-http-client	30812	3935	45	480	2264
BroadleafCommerce/BroadleafCommerce	185869	15545	32	129	3093
vkostyukov/la4j	13480	857	30	149	2132
thymeleaf/thymeleaf	40910	1645	29	174	3612
mrniko/redisson	101069	3743	26	129	2386
Athou/commafeed	9037	2521	23	52	441
dropwizard/metrics	19098	2375	23	101	691
alibaba/druid	296482	5824	16	81	1712
graphhopper/graphhopper	60847	3431	16	82	1141
Graylog2/graylog2-server	147138	14139	15	55	719
clojure/clojure	40728	3167	14	32	590
bennidi/mbassador	5520	334	14	88	833
apache/incubator-dubbo	101610	2491	12	68	284
opentripplanner/OpenTripPlanner	92171	8999	10	165	3016
Total	1338231	76144	539	4187	34313

As for the extra edits, we considered the eight extra edit types with the highest number of occurrences. Moreover, the number of commits used goes according to Peduzzi [17] recommendations of sample sizes. We believe that low frequency extra edits should be relate to sporadic edits, not edition patterns. Figure 3(c) shows the extra edits frequency throughout the selected commits. The description of the selected extra edit types can be found in Section 3. Our investigation focuses on the extra edits performed along with *Extract Method* refactorings. Although the *Extract Method* ranked as second on frequency, it appeared in more commits, as shown in Figure 3(b). Our dataset is also available on our website.

4.3 Analysis Method

To analyze how developers combine *Extract Methods* with extra edits, we built *regression models* that relate the *Extract Method* refactoring to each extra edit. It is important to highlight although a regression model is often used as a prediction artifact, it can also be used as a describing tool for evidencing how the variables (dependent and independent) relate. Other works use regression models to describe the relationship between two or more variables (e.g., [13, 16], where the coefficients of the regression model represent the impact of the independent variables on the dependent variables.

Since we want to describe the extra edits based on the refactorings operations, our independent variable is the *Extract Method*, while the *extra edit* is the response (or dependent) variable. Before building regression models, we first run a normality test that attested that the collected data does not follow a normal distribution. Then, we investigated what regression method best fits our data, based on its distribution. According to Hilbe [9], the most common distributions for counting data are Poisson and Negative Binomial. Poisson could not be used since it assumes a data where mean and variance are the same, and our data variance is at least 12 times greater than the mean. On the other hand, the Negative Binomial estimates both parameters independently, allowing the distribution

[4]https://github.com/moreiraJS/SAC2020

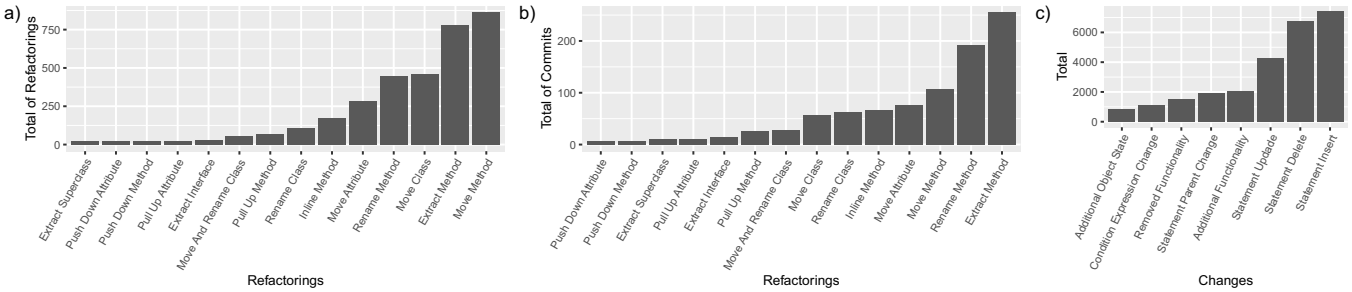

Figure 3: Refactoring frequency

to deal with overdispersion (variance greater than the mean) and underdispersion (variance lower than the mean) [9].

Another common problem when dealing with counting data is the great number of zero counts. Considering that our dataset with 539 observations, the number of zero counts per extra edit type is very high. According to Hilbe [9], Hurdle models are typically used to model the zero count. Therefore, for our analysis, we used *Hurdle Negative Binomial* models.

A Hurdle Negative Binomial model is divided into two parts, its Logit and Count models. The first is a regular logistic regression, which relates dependent variables with independent variables in order to predict the probability of the dependent variable being different from zero [0;1], i.e., its presence or absence. The Count model predicts the frequency of the dependent variable based on the independent variable starting from one ($[1, \infty]$). Equation 1 shows the general structure of the models we built, where *ExtraEdit* is the predicted frequency of an extra edit[5], *EM* is the number of *Extract Method* in a given commit, and β_0 and β_1 are the found statistical coefficients.

$$ExtraEdit = \frac{e^{\beta_0+\beta_1 EM}}{1 + e^{\beta_0+\beta_1 EM}} \quad (1)$$

We analyzed whether the independent variables generate significant impact on the dependent variables. To better understand our results, we analyzed the relation between variables using *Odds Ratios* (OR), which is given by e^C, where C represents the coefficient of the independent variable in each model [9]. The Odds Ratios indicate the increase, or decrease, in the likelihood of the dependent variable for a unit increase of the independent variable. For instance, if the Logit model relates the *Extract Method* refactoring with a *Statement Insert* extra edit with an OR of 1.1, it means that for each *Extract Method* refactoring, there are 10% higher chances of appearing a *Statement Insert* edit in the same commit. Considering a Count Model in the same scenario with an OR of 1.2, it means a 20% increase in the number of *Statement Inserts* for each *Extract Method*.

5 RESULTS AND DISCUSSIONS

Table 2 shows the Odds Ratio (OR) of the selected extra edits for the *Extract Method* refactoring. The significance of the ORs is represented by the number of " * " after each value, where " *** " represents a significance level higher than 99.9%, while " ** " represents a significance level higher than 99%, and " * " a significance level higher than 95%. Below, we discuss the results considering each model, independently.

5.1 RQ1 - Which extra edits are more likely to appear alongside a given refactoring edit?

To answer our first research question, we focused on the Logit model ORs presented in Table 2. The values in the Logit column represent the change in the likelihood of a respective extra edit appear in a commit (Section 4.1 - RQ1). Interestingly, all ORs values were higher than 1, which means that there is an increase in the likelihood of these extra edits appearing in the same commit when a developer performs an *Extract Method* refactoring. For instance, the OR for *Additional Functionality* edit was 1.264771, which means that there are 26% more chances of an extra new method to appear alongside an *Extract Method*. This finding suggests that developers often combine *Extract Method* edits with feature introductions and/or updates, which corroborates with Palomba et al. [16].

The introduction of new methods can be error-prone, once it can override or be overridden by other methods, making the system behavior differ from what is expected. In this context, a focused code review is recommended, where the developer would give it a second look to ensure that the new method does not impact on the refactoring, and/or does not override, or is overridden, by any other method.

The *Additional Object State* edit presented 12% increase in its likelihood. This edit increases the number of possible states of an object, which can directly impact comparison functions. Comparison functions takes into account specific aspects of an object, or the whole object, to return its result. However, by adding a new object state, the comparison function might return a different result from the expected. For instance, if the software uses a comparison

Table 2: ORs values from our Logistic models.

Extra edit	Logit	Count
Statement Insert	1.395968 ***	1.102821 ***
Statement Delete	1.447590 ***	1.111233 ***
Statement Update	1.392035 ***	1.083520 **
Additional Functionality	1.264771 ***	1.104078 **
Statement Parent Change	1.073166 *	1.160105 *
Removed Functionality	1.078408 *	1.038553
Condition Expression Change	1.109404 **	1.087638 **
Additional Object State	1.124247 **	1.059852

[5]The logit model varies from [0;1], while the count model ranges from [1; ∞]

function that compares the whole object, it might be necessary to develop a specific function to compare the other aspects except the new one. As for the *Remove Functionality* edit, its OR presented approximately 8% increase.

There is a 40% increase on the likelihood of a developer perform a *Statement Insert* in a commit every time an *Extract Method* refactoring is performed. For the *Statement Delete* extra edit, we found the highest OR value (an increase of 45%). For the *Statement Update*, the increase is 39%, while it is nearly 11% for the *Condition Expression Change* edit, and 7% for the *Statement Parent Change* edit. In practice, it means that developers, when applying *Extract Method* edits, tend to change both the refactored and non-refactoring-related methods. These extra edits are worth revision, since often lead to behavior changes. The impact caused by these changes can be even greater when occurring in the extracted method, once the change made might seem harmless in the method, but have direct impact on its caller.

Listing 2 presents a scenario from the *Druid* project[6]. In a single commit, code fragments from 26 different methods were extracted with *Extract Method* edits. The condition on lines 2-3 was extracted from 24 methods, while lines 4-8 were extracted from other two methods. Finally, Line 9 is a new condition added (extra edit). This edit might impact a great number of methods, therefore, we believe all those extra edits deserve proper revision.

Method extracted from a set of methods to reduce code duplicity

```
1  public static boolean checkParameterize(SQLObject x) {
2    if (Boolean.TRUE.equals(...))) {
3      return false; }
4    SQLObject parent = x.getParent();
5    if (parent instanceof SQLDataType ...
6  +      || parent instanceof SQLAssignItem) {
7      return false;}
8      return true; }
```

Listing 2: Method extracted for reducing code duplicity.

5.2 RQ2 - How the number of extra edits changes based on the refactorings performed by developers?

To answer the second research question, we focused on the Odds Ratio values for the Count part of the Hurdle Negative Binomial models reported in Table 2. The Count model relates the frequency of extra edits to the number of refactorings performed in a commit (Section 4.1 - RQ2). That is, the increase in the amount of the extra edits for each *Extract Method* refactoring operation.

The first difference we can notice between the count and logit models is that the *Removed Functionality* and the *Additional Object State* edits with the *Extract Method* refactoring were statistically significant for the Logit model but not for the Count model. This means that the first appearance of these extra edits are related to the presence of the *Extract Method* refactoring (Logit model), but their frequency are not. Another interesting case is the *Statement Parent Change*, which was the only extra edit that got a greater OR value for the Count model. For each occurrence of an *Extract Method* refactoring, it is expected an increase of 16% in the amount of *Statement Parent Change* edits, which was the highest OR for the Count models. The *Statement Delete* extra edit presented an 11% increase in its amount, while, for the *Statement Insert*, the increase was 10%.

The *Additional Functionality* edit also presented 10% increase in its frequency for each *Extract Method* performed. The *Condition Expression Change* and the *Statement Update* presented the smallest significant OR, approximately 9% increase. These results could support, not only refactoring-driven tools, but also tools that emulate floss-refactoring (e.g., mutation tools), since we provide a characterization on the common types of extra edits and their frequency, based on real-world scenarios.

5.3 RQ3 - How Extract Method Floss-Refactorings Are Performed?

In order to get a more thorough analysis, we also investigated which entities were often modified by inner-method extra edits. By inner-method edits we call the edits that are performed directly on refactored entities. For instance, in an *Extract Method* we considered inner-method edits any extra edit in the original method and/or on the newly created one. Considering our whole dataset (34.313 extra edits) 4.598 (13,4%) were inner-method.

Considering that the *Statement Insert* edit had a 10% increase for each *Extract Method* occurrence, we analyzed the type of those statements (e.g., a *Variable Declaration Statement*, an *If Statement*, a *Method Invocation*) when performed inside refactored entities. With this in mind, for each refactoring and extra edit type, we considered commits with both, refactorings and extra edits from the types under analysis. Moreover, in this analysis, we considered just the five most frequent entity types which, interestingly, were the same for all inner method edits: (*Method Invocation*, *If Statement*, *Variable Declaration Statement*, *Assignment* and *Return Statement*). It is important to notice that in cases where a statement is composed by multiple types, the entity is classified based on the highest type level. For instance, the statement in Figure 1(a) - line 3 is classified as an *If Statement*, even though it is also composed by a *Method Invocation*.

To analyze the relationship between the refactoring and the entity types, again, we built a new set of Hurdle Negative Binomial models, but none of the Logit models achieved statistical significance. We believe this happened because, once we filtered the data by the extra edit type, the second process assumed by the Hurdle assumption became irrelevant. This filter is enough to change data distribution because, by selecting the commits with both specific edits, we reduced the number of zeros. Therefore, we used simple Negative Binomial regression. Table 3 shows the OR values for each entity and its respective edit type. The significant values (considering a significance level of $\alpha = 0.5\%$) are highlighted with " * ", following the same pattern of Table 2.

As we can see, most ORs were significant. For the *Statement Parent Change*, the only entity with significant OR was the *Return Statement*, with 9% increase in its frequency for each *Extract Method* performed in a commit. Among the entities in *Statement Insert*, the *Method Invocation* and the *Assignment* presented the highest OR, having approximately 8% more occurrences for each refactoring. The *Variable Declaration Statement* got the second highest OR, with 7%. The *If Statement* appeared 4% more often for each refactoring,

[6]https://github.com/alibaba/druid

Table 3: Relation Between Refactorings and Entity Types

Extract Method				
Entity Type	Statement Insert	Statement Delete	Statement Update	Statement Parent Change
Method Invocation	1.075090 ***	1.139508 ***	1.102857 ***	1.028549
If Statement	1.043572 **	1.060844 **	1.055093 ***	1.030626
Variable Decl. Statement	1.068004 ***	1.085530 ***	1.051662 ***	1.030330
Assignment	1.076447 **	1.077544 **	1.103971 **	0.989705
Return Statement	1.061480 ***	1.028390	1.054431	1.090631 **

while the *Return Statement* entity presented an increase on its frequency of 6%.

For the *Statement Delete*, the *Method Invocation* entity is the most likely to be deleted, it occurred approximately 14% more when an *Extract Method* is performed, which was the highest OR from the entity types. By analyzing the *Statement Update*[7], we found an increase of 10% for the *Method Invocation* and the *Assignment*, while it was approximately 5% for the *If Statement* and the *Variable Declaration Statement*.

In practice, these results could be useful for static analysis tools, who would have a starting point during analysis. Most of these edits could introduce unexpected behavior in the system, specially the *If Statement*, whose impact could easily pass unnoticed, since it could change the system behavior without throwing any exception.

6 THREATS TO VALIDITY

Construct Validity: Our approach relies on three different tools (RefDiff, SafeRefactor, and ChangeDistiller) and their accuracy may directly affect its output. All tools have high accuracy levels [5, 20, 22]. Moreover, we ran sampling tests for validating the results in the context of our investigation. Our ChangeDistiller extension was developed based on its analysis method and Fowler's guidelines for each refactoring type [6].

Regarding SafeRefactor, since it uses generated test suites to check a possible pure-refactoring edit, its analysis might not detect all behavior changes. However, since our study works only with non-behavior-preserving pairs of commits, we used only versions that SafeRefactor presented failing test cases. Therefore, possible limitation on SafeRefactor analysis does not impact our study, since failing test cases guarantee behavior changes.

Internal Validity: One may argue that SafeRefactor's capability of detecting behavior changes may be influenced by its test generation time limit. However, Soares et al. [21] found that when considering generations beyond 120 seconds, the suites do not present considerable coverage increases. Our study used this threshold.

Conclusion Validity: To assess the relationship between the refactorings and extra edits, we relied on Hurdle Negative Binomial models, which require a high number of observations. However, we used sample sizes greater than the ones recommended by Peduzzi [17] for logistic regressions.

External Validity: Our study was restricted to Java-based projects hosted on GitHub. Thus, we cannot generalize our findings beyond the projects used. However, we worked with a great number of projects and commits from different sizes and contexts.

[7]If the type of entity changes during the *Statement Update*, the type reported for the analysis is the type before the edit is applied

7 RELATED WORK

To assess the benefits and challenges from refactorings, Kim et al. [12] conducted a field study at Microsoft. They found that frequently refactored modules are less likely to have post-release defects. Moreover, the participants emphasized the need for a proper tool support for minimizing the costs and risks related to refactoring.

Tsantalis et al. [23] proposed an approach and tool for detecting refactoring between two versions of a system based on AST matching. They compared their tool against RefDiff [20] (the tool we used in our study) and achieved better accuracy. By the time we run our study Tsantalis et al.'s tool had not been released yet, and we were not able to analyse its applicability. However, we ran a sample testing in our dataset for checking the impact of the use of RefDiff, which proved to be an effective refactoring-detection option in the context of our study.

Murphy-Hill et al. [14] conducted a study on four data sets with more than 140,000 refactorings and 3,400 version commits. They found that developers often apply refactorings alongside other edits. Moreover, developers not only apply refactoring edits alongside extra edits but, according to Silva et al. [19], they are mainly driven by requirements changes.

Kim et al. [11] investigated API-level refactorings in three large open source projects to understand its role on software evolution. They found that the number of bug fixes increases after API-level refactorings, either because refactoring edits introduced new bugs, or they helped developers to identify and fix previous bugs. Due to frequent floss-refactoring mistakes observed, the authors pointed out the need for tools to support the safe application of refactoring and non-refactoring edits together.

Coelho et al. [3] performed a systematic literature mapping on refactoring-awareness during code review. The authors emphasise the lack of appropriate characterisation studies, accurate support to change set with multiple refactorings types simultaneously, as well as the need for studies on the effectiveness of refactoring-aware solutions for code review. In this sense, Ge et al. [7] propose a refactoring-aware code review tool that can separate refactorings and non-refactoring edits. On the other hand, Alves et al. [1] propose a tool that applies static analysis for inspecting manual refactorings and highlights edits that go further than what an automatic tool consider as pure-refactoring.

Palomba et al. [16] performed an exploratory study on the relationship between refactoring and other change categories[8]. They classified each commit to the categories based on the commits' logs and found that refactorings that focus on improving code maintainability and comprehensibility (e.g., *Add Parameter, Consolidate Duplicate Conditional Fragments*) are more likely to appear

[8]Fault Repairing, Feature Introduction, and General Maintenance.

alongside fault repairing modifications. Refactorings that focus on improving code cohesion (e.g., *Add Parameter, Extract Method*) are more likely to appear alongside feature introduction modifications. However, the authors considered only commits categories, ignoring all code edits applied by the developers in their commits.

The works previously listed are essential to the field and provide substantial contributions. However, none of them focus on investigating the relationship between the refactorings and the extra on the edit-level. Our empirical study focused on providing a more in-depth understanding on the relationship of edits in floss-refactoring and aims at helping researchers to understand how developers perform code evolution in practice, and consequently guide efforts for developing novel strategies for a more safer and systematic code edition.

8 CONCLUSION

As far as we understand, little has been done on understanding how floss-refactoring is performed in practice. Results in this sense can help the assessment of refactoring code review, by guiding what edits should be analyzed first, and to help other empirical investigations, by providing a deeper understanding on floss-refactoring.

We ran an empirical study that investigated how refactoring and non-refactorings changes relate. We used a series of state-of-the-art tools for mining repositories and collecting non pure-refactoring commits. Then, we decomposed those edits and built regression models to visualize the relationship between *Extract Method*s and a series of extra edits. Our results showed that the introduction of new methods is more common when an *Extract Method* is performed, besides the extracted one, as well as different types of inner method extra edits. A deeper analysis, showed that some statements are more likely to be changed depending on the extra edit performed.

As future work, we intend to extend our analysis with a larger variety of systems, as well as analyze other refactorings and extra edit types. Moreover, we plan to investigate the impact that each extra edit fault may generate during software evolution. Finally, for helping new research works on this field, we intend to work on a mutation tool for automatically introduce faults related to extra edits in a given system.

9 ACKNOWLEDGEMENTS

This research was partially supported by a cooperation between UFCG and Ingenico do Brasil LTDA, stimulated by the Brazilian Informatics Law n. 8.248, 1991. Second and third authors are supported by National Council for Scientific and Technological Development (CNPq)/Brazil (processes 429250/2018-5 and 315057/2018-1). First author was supported by UFCG/CNPq.

REFERENCES

[1] E. L. G. Alves, M. Song, T. Massoni, P. D. L. Machado, and M. Kim. 2018. Refactoring Inspection Support for Manual Refactoring Edits. *IEEE Transactions on Software Engineering* 44, 4 (April 2018), 365–383. https://doi.org/10.1109/TSE.2017.2679742

[2] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM '12)*. IEEE Computer Society, Washington, DC, USA, 104–113. https://doi.org/10.1109/SCAM.2012.20

[3] FlÃąvia Coelho, Tiago Massoni, and Everton Alves. 2019. Refactoring-Aware Code Review: A Systematic Mapping Study. In *Proceedings of 3rd International Workshop on Refactoring*.

[4] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[5] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (Nov 2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

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

[7] X. Ge, Q. L. DuBose, and E. Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *2012 34th International Conference on Software Engineering (ICSE)*. 211–221. https://doi.org/10.1109/ICSE.2012.6227192

[8] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill. 2017. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 71–79. https://doi.org/10.1109/VLHCC.2017.8103453

[9] Joseph M. Hilbe. 2014. *Modeling Count Data.* Cambridge University Press. https://doi.org/10.1017/CBO9781139236065

[10] M. Kaya, S. Conley, Z. S. Othman, and A. Varol. 2018. Effective software refactoring process. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*. 1–6. https://doi.org/10.1109/ISDFS.2018.8355350

[11] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An Empirical Investigation into the Role of API-level Refactorings During Software Evolution. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/1985793.1985815

[12] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

[13] Nikolaos Mittas, Makrina Viola Kosti, Vasiliki Argyropoulou, and Lefteris Angelis. 2010. Modeling the Relationship Between Software Effort and Size Using Deming Regression. In *Proc. of the 6th International Conference on Predictive Models in Software Engineering (PROMISE '10)*. ACM, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/1868328.1868339

[14] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 287–297. https://doi.org/10.1109/ICSE.2009.5070529

[15] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 552–576. https://doi.org/10.1007/978-3-642-39038-8_23

[16] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An Exploratory Study on the Relationship Between Changes and Refactoring. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 176–185. https://doi.org/10.1109/ICPC.2017.38

[17] Peter Peduzzi, John Concato, Elizabeth Kemper, Theodore R Holford, and Alvan R Feinstein. 1996. A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology* 49, 12 (1996), 1373–1379.

[18] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. 2010. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609577

[19] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[20] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *Proc. of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, 269–279. https://doi.org/10.1109/MSR.2017.14

[21] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni. 2011. Analyzing Refactorings on Software Repositories. In *2011 25th Brazilian Symposium on Software Engineering*. 164–173. https://doi.org/10.1109/SBES.2011.21

[22] G. Soares, R. Gheyi, D. Serey, and T. Massoni. 2010. Making Program Refactoring Safer. *IEEE Software* 27, 4 (July 2010), 52–57. https://doi.org/10.1109/MS.2010.63

[23] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proc. of the 40th Int. Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206