Technical Report

Studying the *RMiner* Refactoring Detection Tool

Q2 2018 – Q3 2018

Abstract

We replicate the experimental evaluation of *RMiner* - a refactoring detection tool which operates on version control commits [1]. In addition, we run *RMiner* for some codes inspired by [2, 3, 4] and codes extracted from experiments using *SaferRefactor* and *JDolly* [5]. In this report, we present the replication findings and suggestions regarding bugs identified during our work.

1 Replication Results

In this section, we present the replication of the experiment described in [1]. The experiment evaluates the usefulness of *RMiner* by answering the research questions:

- RQ1 - What is the accuracy of *RMiner* and how does it compare to the previous state-of-the-art?
- RQ2 - What is the execution time of *RMiner* and how does it compare to the previous state-of-the-art?

In the replication, we evaluate the accuracy and running time of *RMiner* and *RefDiff* [6] for a sample[1] of 37 commits randomly selected from the original dataset (oracle). Table 1 describes the instrumentation applied to the replication.

Type	**Description**
Code/Documentation	*RMiner* [2] and *RefDiff* [3]
System configuration	*Intel core* i7-8550U CPU@2.70 GHz, 16 GB DDR3, 2TB SSD, Linux Ubuntu 16.04
Development	Java 1.8.0_162, IDE *Eclipse* Oxygen.3a *Release* (4.7.3a), R 3.4.4 e *RStudio* 1.1.423
Measurement	Manual inspection for couting TPs (True Positives), FPs (False Positives) and FNs (False Negatives), and the System.nanoTime() Java method for collecting the running time.

Table 1: Instrumentation

[1] All replication results for this sample are available on `github.com/flaviacoelho/replication-RMiner`

Table 2 shows the type of refactorings detected by *RMiner* and *RefDiff* for the sample, which contains 425 refactorings.

Refactoring type
type
Extract Superclass, *Move Class*, *Rename Class*
method
Extract Method, *Inline Method*, *Pull Up Method* *Rename Method*
field
Pull Up Field, *Push Down Field*, *Move Field*

Table 2: Types of refactorings detected by *RMiner* and *RefDiff* for the sample

1.1 Findings

Figures 1-3 describe the number of TPs, FPs and FNs by type of refactoring detected for the sample. As Figure 1 shows, the majority of applied refactorings are Extract Method and Move Class. On mean, the number of TPs is 12.6 and 12.4 to *RefDiff* and *RMiner*, respectively.

Figure 1 - Number of True Positive x Refactoring Type x Detection Tool

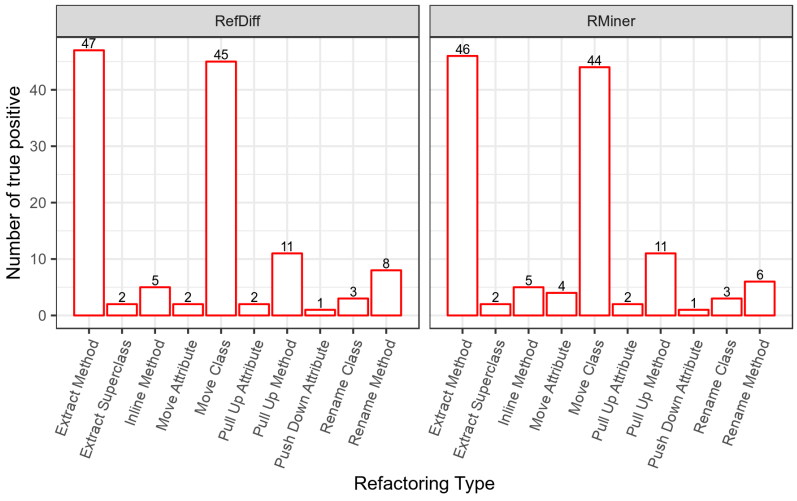


Figure 2 shows the distribution of FPs by refactoring type. There are outliers in the detection of Pull Up Attribute and Pull Up Method by *RMiner*. On mean, the number of FPs is 0.3 to *RefDiff* and 15.3 to *RMiner*.

The distribution of FNs is presented in Figure 3. On mean, 0.9 of refactorings are FNs in *RefDiff* detection and 1.0, in *RMiner*.

Figures 4-5 describe the precision and recall obtained by refactoring type. The precision of *RMiner* and *RefDiff* ranges from 88.9% to 100%, and from 33% to 100%, respectively. For the sample, on mean, the *RefDiff* precision is 98.5% and the *RMiner* precision is 81.9%.

Figure 2 - Number of False Positive x Refactoring Type x Detection Tool

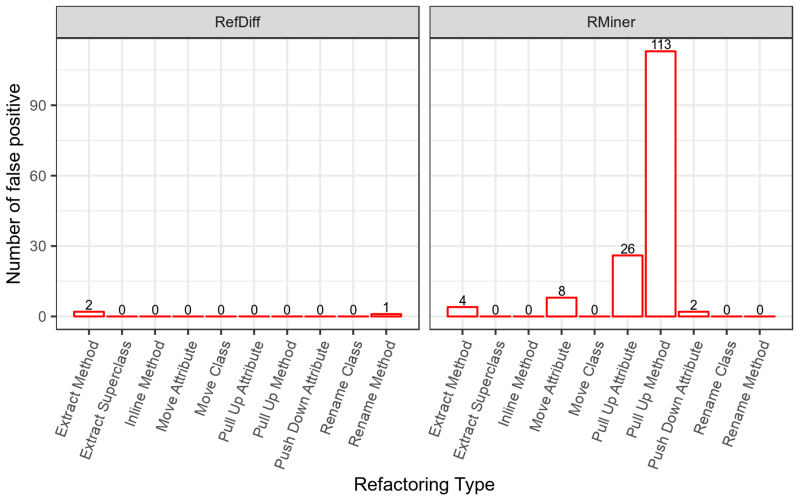


Figure 3 - Number of False Negative x Refactoring Type x Detection Tool

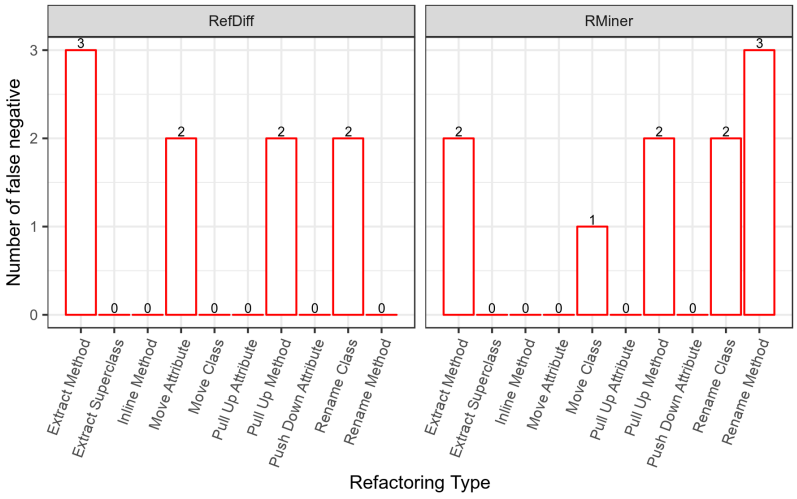
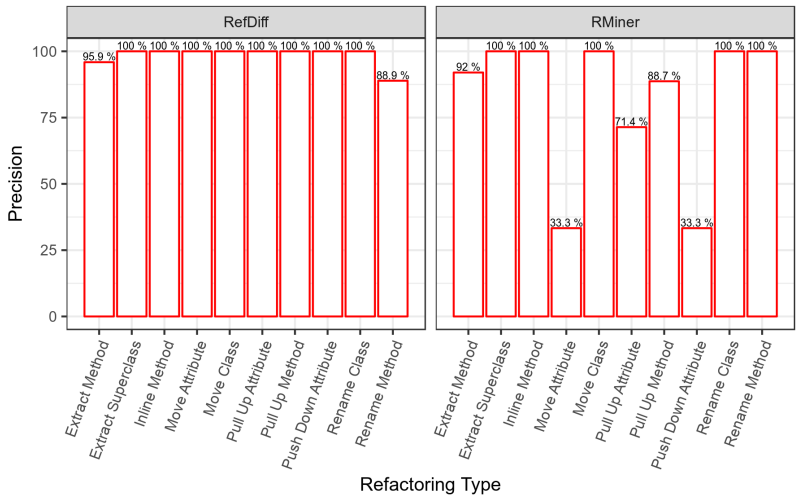


Figure 4 - Precision x Refactoring Type x Detection Tool



The opposite occurs in relation to recall. On mean, *RMiner* obtained 90.5% and *RefDiff*, 88.9%. In this terms, Figure 5 shows the distribution of recall for *RefDiff* (50% a 100%) and *RMiner* (60% a 100%).

Figure 5 - Recall x Refactoring Type x Detection Tool

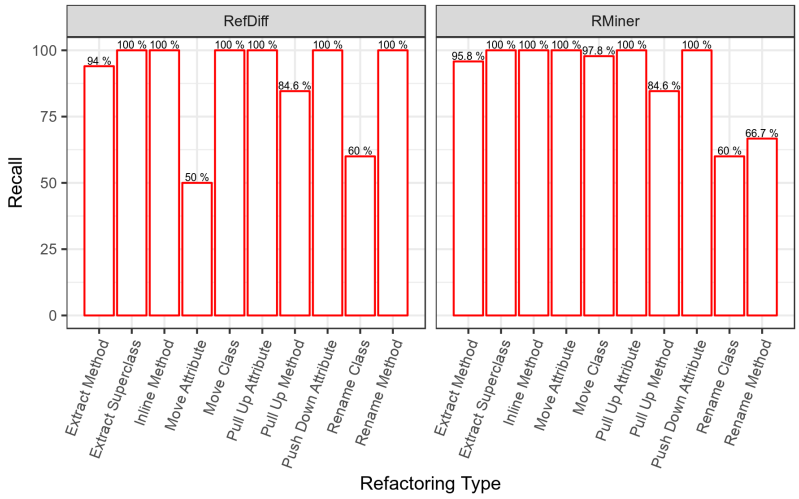


Figure 6 present the distribution for *RefDiff* running time, which ranges from 227 a 9008 ms (median value is 1522 ms).

The *RMiner* running time ranges from 0.0014 to 0.2071 ms (median value is 0,002 ms), as shown in Figure 7.

1.2 Discussion

Our sample of 37 commits may be considered no representative due to the types of detected refactorings (Table 2).

Figure 6 - RefDiff Running Time

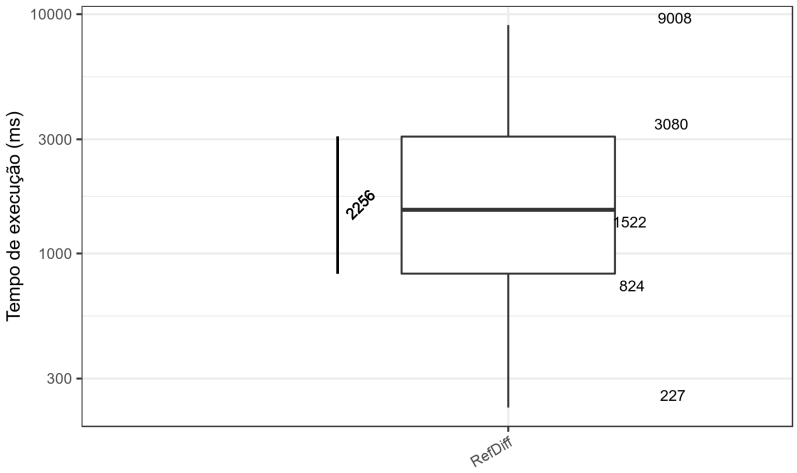
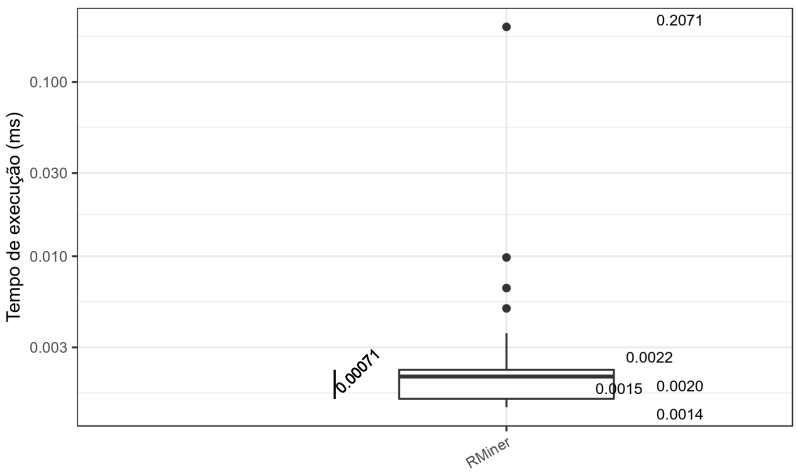


Figure 7 - RMiner Running Time



The *RMiner* precision (81.9%) obtained is less than the RefDiff precision (98.5%). These values are different of the results in [1]. We believe that the high number of FPs during the detection of Move Attribute and Push Down Attribute refactorings is the reason to the inferior precision of *RMiner*. For instance, 113 refactorings (with the same description) was counted by *RMiner* (Figure 2).

In terms of recall, the findings (90.5% and 88.6% to *RMiner* and *RefDiff*, respectively) confirm the results obtained in [1].

We also applied the Wilcoxon signed rank test on the paired samples of the running time for each commit, which too rejected the null hypothesis "*RefDiff* execution time is smaller than that of *RMiner*" with a p-value <2.2e-16.

2 Additional Findings

In this section, we present some examples of no detected instances and detected transformations (no refactorings) for the refactorings types that follow.

The codes shown in Table 3-6 are available on `github.com/flaviacoelho/refactorings-toy`.

before refactoring

```
public int getZeroValueOfWeeklySalary() {
    return isZeroWeeklySalary() ? 1 : 0;
}
public boolean isZeroWeeklySalary() {
    return(weeklySalary == 0);
}
```

after refactoring

```
public int getZeroValueOfWeeklySalary() {
    return (weeklySalary == 0) ? 1 : 0;
}
```

Table 3: [FN] Inline Method

Tables 7-13 illustrate some codes extracted from experiments conducted by Soares et al. [5]. These experiments evaluate approaches for identifying behavior-preserving transformations activities on software repositories. The codes are available on `github.com/flaviacoelho/refactorings-examples`.

References

[1] Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. *Accurate and efficient refactoring detection in commit history.* In Proceed-

before refactoring

```java
public class Invoice implements Payable{
    private final String partNumber;
    private final String partDescription;
    private int quantity;
    private double pricePerItem;
    private String message;
    private ClientType typeOfClient;

    //omitted code
}
```

after refactoring

```java
public class ClientType {

    private String name;
    private String id;
    private String type;
    private Invoice invoice;
    private double bonus;
    private String message; //Move attribute from Invoice

    //omitted code
}
```

Table 4: [FN] Move Attribute

before refactoring

```
public class Person {
    //omitted code
}

class SpecialPerson{ //empty class
}
```

after refactoring

```
public class Person {
    //omitted code
}

class OnePerson{ //SpecialPerson renamed to OnePerson
}
```

Table 5: [FN] Rename Class

ings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 2018.

[2] Deitel, P. and Deitel, H. *Java: How to Program*. Tenth Edition. Pearson, 2014.

[3] Fowler, M. *Refactoring: Improving the Design of Existing Programs*. AddisonWesley, 1999.

[4] Eilertsen, A. M. *Making Software Refactorings Safer*. Master Thesis – Department of Informatics – University of Bergen. Bergen, 2016.

[5] G. Soares, R. Gheyi and T. Massoni. *SaferRefactor Experiments*, 2010, `http://www.dsc.ufcg.edu.br/~spg/saferefactor/experiments.html`

[6] Silva, D. and Valente, M. T. *Refdiff: Detecting refactorings in version histories*. In Proceedings of the 14th International Conference on Mining Software Repositories,page 269–279, Buenos Aires, Argentina, 2017.

before refactoring

```
public class Person {

    //omitted code

    public void testExtractAndMoveMethod() {
        officeTelephone.setAreaCode("00");
        System.out.println(getName());
        officeTelephone.setAreaCode("88");
    }
}
```

after refactoring

```
public class TelephoneNumber {

    //omitted code

    public void testExtract(Person p) { //extracted from
        Person
        setAreaCode("00");
        System.out.println(p.getName());
        setAreaCode("88");
    }
}
```

Table 6: [FN] Extract and Move Method

before transformation

```
package a;
public class A {
    public int k() {
        return 20;
    }
}

package a;
import b.*;
public class B extends C {
    public int test() {
        return k();
    }
}

package b;
public class C {
    public int k() {
        return 10;
    }
}
```

after transformation

```
package a;
public class C { //renamed class
    public int k() {
        return 20;
    }
}

package a;
import b.*;
public class B extends C {
    public int test() {
        return k();
    }
}

package b;
public class C {
    public int k() {
        return 10;
    }
}
```

Table 7: [FP] Rename Class

before transformation

```
package p2;
import p1.*;
public class B extends A {
  protected long n(int a){
    return 0;
  }
  public long m(){
    return this.k(2);
  }
}
package p1;
public class A {
  public long k(long a){
    return 1;
  }
}
```

after transformation

```
package p2;
import p1.*;
public class B extends A {
  protected long k(int a){ //n renamed to k
    return 0;
  }
  public long m(){
    return this.k(2);
  }
}

package p1;
public class A {
  public long k(  long a){
    return 1;
  }
}
```

Table 8: [FP] Rename Method

before transformation

```
package p2;
public class B extends A {
}
package p1;
import p2.*;
public class B extends A {
  protected long k(int a){
    return 0;
  }
  public long test(){
    return m();
  }
}
package p2;
import p1.*;
public class A {
  public long k(long a){
    return 1;
  }
  public long m(){
    return new B().k(2);
  }
}
```

after transformation

```
package p1;
import p2.*;
public class B extends A {
  protected long k(int a){
    return 0;
  }
  public long test(){
    return m();
  }
public long m() {
    return new B().k(2);
  }
}
package p2;
public class B extends A {
    public long m() {
        return new B().k(2);
      }
}
package p2;
import p1.*;
public class A {
  public long k(long a){
    return 1;
  }
}
```


Table 9: [FP] Push Down Method

before transformation

```
public class A {
  public int k = 10;
}
public class B extends A {
  public int k = 20;
}
public class C extends B {
  public int test() {
    return super.k;
  }
}
```

after transformation

```
public class A {
  public int k = 10;
}
public class B extends A {
}
public class C extends B {
  public int k = 20; //pushed down
      field
  public int test() {
    return super.k;
  }
}
```

Table 10: [FP] Push Down Field

before transformation

```
public class A {
  public int k(long l) {
    return 1;
  }
  private int k(int l) {
    return 2;
  }
}
public class B extends A {
  public int m() {
    return k(2);
  }
  public int test() {
    return m();
  }
}
```

after transformation

```
public class A {
  public int k(long l) {
    return 1;
  }
  private int k(int l) {
    return 2;
  }
  public int m() { //pulled up
    return k(2);
  }
}
public class B extends A {
  public int test() {
    return m();
  }
}
```

Table 11: [FP] Pull Up Method

before transformation

```
public class A {
}
public class B extends A {
  public int k = 10;
}
public class C extends A {
  public int k = 20;
  public long test() {
    return k;
  }
}
```

after transformation

```
public class A {
  public int k = 10; //pulled up
}
public class B extends A {
}
public class C extends A {
  public long test() {
    return k;
  }
}
```

Table 12: [FP] Pull Up Field

before transformation

```
public class A {
  public B b;
  protected long k(int a) {
    return 2;
  }
}
public class B {
  private long k(long a) {
    return 1;
  }
  public long test() {
    return k(2);
  }
}
```

after transformation

```
public class A {
  public B b;
}
public class B {
  private long k(long a) {
    return 1;
  }
  public long test() {
    return k(2);
  }
  protected long k(int a) { //moved from A
    return 2;
  }
}
```

Table 13: [FP] Move Method