SPLAB
SOFTWARE PRACTICES LABORATORY

Subject: Studying Refactoring supported by RefactoringMiner Detection Tool

Date: Q3 2018

We applied *RefactoringMiner*[1] [1] on toy examples with the purpose of studying the effects of multiple refactorings in one commit. Accordingly, our code base included some programs inspired by [2, 3, 4], and ones extracted from experiments using *SaferRefactor* and *JDolly* [5]. In this report, we present the findings of our study [2].

Two executions were carried out considering the same code base at different moments - using versions of *RefactoringMiner*, available on November 7th and December 17th. The majority of refactorings, performed in our toy, were detected by *RefactoringMiner*.

In our particular scenario, we applied refactorings to programs containing empty classes/packages, inner classes, classes with/without references, classes inside the same/different files, and methods with/without setters and getters. And, especially, we considered multiple refactorings together in the same commit.

In the next sections, we explain some code fragments that express specific cases, in which the refactorings have not been detected by *RefactoringMiner*. Finally, we present some questions related to our findings.

1 Studying multiple refactorings

To study the effect of multiple refactorings applied in the same commit[3], we considered the following types (in 27 refactorings):

- *Change Package (Rename)*
- *Extract Interface/Superclass*
- *Extract Method*
- *Extract and Move Method*
- *Inline Method*
- *Move Attribute/Method/Class*
- *Move and Rename Class*

[1] https://github.com/tsantalis/RefactoringMiner

[2] Available at https://github.com/flaviacoelho/refactorings-toy and https://github.com/flaviacoelho/refactorings-examples

[3] https://github.com/flaviacoelho/refactorings-toy/commit/32909580e1404456b1d48303220202c012e7b3c6

- *Pull Up Attribute/Method*
- *Push Down Attribute/Method*
- *Rename Attribute/Method/Variable/Parameter*

In this context, *RefactoringMiner* has detected most carried out refactorings, except for fragments shown in Listings 1-3 (False Negative), and 4-6 (False Positive).

In Listing 1, we show a *Change Package (Rename)* performed at an empty package. In this case, the refactoring was not detected.

before refactoring

```
1 package4.package4c;
```

after refactoring

```
1 package4.package4b;
```

Listing 1: [FN] *Change Package (Rename)*

We can see, from Listing 2, that *Rename Attribute* and *Move Attribute* were both applied in the same commit. However, only the latter was detected by RefactoringMiner.

before refactoring

```
1 public class A {
2     //Rename Atribute
3     String w;
4 }
```

after refactoring

```
1 public class A {
2     //Move Attribute
3     String a;
4     //Rename Atribute
5     String x;
6 }
```

Listing 2: [FN] *Rename Attribute*

In Listing 3, `m1EM()` was extracted from `main()` in class A. At the same time, the method `mInlinedMethod()` was removed because we intended to apply *Inline Method* refactoring in `m2A()`. Thus, in this case, an *Inline Method* was really applied, as presented in Listing 4. But, *RefactoringMiner* detected this transformation as a *Rename Method*, due the accomplishment of the operations in the same commit.

According to Listing 5, a *Move and Rename Attribute* was detected as *Move Attribute*.

before refactoring

```
public class A {
    public static void main(String ... args) {
        //Extract Method... (from main())
        System.out.print("This fragment is used to ");
        System.out.println("Extract Method refactoring!");
    }
}
```

after refactoring

```
public class A {
    public static void main(String ... args) {
        m1EM();
    }
    public static void m1EM() {
        //Extract Method... (from main())
        System.out.print("This fragment is used to ");
        System.out.println("Extract Method refactoring!");
    }
}
```

Listing 3: [FN] *Extract Method*

In Listing 6, a *Rename Variable/Parameter* refactoring was detected as true positive when we executed *RefactoringMiner* downloaded on November 7th, but as false positive with the version available on December 17th.

2 Studying single refactorings

After applying multiple refactorings in a single commit, we incrementally applied, for the same code base used before, all types of refactoring under investigation (as listed in Section 1). The *RefactoringMiner* was executed after each increment.

As a result, the false negative, related to *Change Package (Rename)*[4] and *Rename Attribute*[5] refactorings (Listing 1-2), was maintained. *RefactoringMiner* has not detected the *Move and Rename Attribute*[6] refactoring (Listing 5), thus it was considered a false negative.

On the other hand, the *Extract Method* (Listing 3) and *Inline Method* (Listing 4) refactorings were successfully detected by *RefactoringMiner*.

Concerning Listing 6, the previous result remained, namely, the *Rename Variable/Parameter* refactoring was detected as true positive when we executed *RefactoringMiner* downloaded on November 7th, but as false positive with the version available on December 17th.

Moreover, we executed *RefactoringMiner* on programs, with a few short refactorings, independently of the code base. In this context, just a *Inline Method*[7] refactoring, shown in Listing 7, was not detected.

[4]https://github.com/flaviacoelho/refactorings-toy/commit/34feffaed800be28f1a132081974d80d62e599b3
[5]https://github.com/flaviacoelho/refactorings-toy/commit/55553c1b8e3696db2df6532d636a209e6aa4a2bb
[6]https://github.com/flaviacoelho/refactorings-toy/commit/c357cafb3e1ee65e9bd844cb09a229b7943f73f2
[7]https://github.com/flaviacoelho/refactorings-toy/commit/14d1ad4a9b4695976a3db45f43aed7a6c78ea211

before refactoring

```
1  public class A {
2      public static void main(String ... args) {
3          m2A();
4      }
5      public static void m2A() {
6          //Inline Method...
7          System.out.print("This fragment is used to ");
8          mInlinedMethod();
9      }
10     public static void mInlinedMethod() {
11         System.out.println("Inline Method refactoring!");
12     }
13 }
```

after refactoring

```
1  public class A {
2      public static void main(String ... args) {
3          m2A();
4      }
5      public static void m2A() {
6          //Inline Method...
7          System.out.print("This fragment is used to ");
8          System.out.println("Inline Method refactoring!");
9      }
10 }
```

Listing 4: [FP] *Inline Method* detected as *Rename Method*

3 Studying behaviour-preserving transformations

Listings 8-9 illustrate a sample of programs extracted from experiments conducted by Soares et al. [5]. These experiments evaluate approaches for identifying behaviour-preserving transformations activities on software repositories.

Listing 8 shows that pushing down a method to a class in another package may shadow a class declaration leading to a behavioural change[8]. For this reason before the transformation, method m() yields 1, but after that, it produces 0.

Applying the *Pull Up Attribute* refactoring to a field f moves all fields f of the subclasses to the superclass. If one of the fields is initialised with a different value, the behaviour of the program may change[9]. In Listing 9, the method test returns 10 instead of 20. Therefore, the transformation changes the program behaviour.

Therefore, *RefactoringMiner* detected the transformations as *Push Down Method* and *Pull Up Attribute* refactorings, respectively, however, in both cases it happened a behaviour change.

[8]https://github.com/flaviacoelho/refactorings-examples/commit/505c6fa16eba14d29e2c6f65fb422d9725862602
[9]https://github.com/flaviacoelho/refactorings-examples/commit/9a9cd138b9f268c2f0749a2556dd114e57278bfe

before refactoring

```
public class R {

}
public class S {
    //Move and Rename Attribute...
    String s;
}
```

after refactoring

```
public class R {
    //Move and Rename Attribute...
    String r;
}
public class S {

}
```

Listing 5: [FP] *Move and Rename Attribute* detected as *Move Attribute*

4 Questions

In order to collaborate with further improvements in *RefactoringMiner*, we would like to raise the following topics for discussion:

- When we performed a *Change Package (Rename)*, Listing 1, at an empty package, the refactoring was not detected (in both experiments – with multiples and single refactorings). Would it be because the package is empty?
- We can see, from Listing 2, that *Rename Attribute* and *Move Attribute* were applied at the same class, but just the *Move Attribute* refactoring was detected by *RefactoringMiner*. Would it be because both refactorings were performed at the same commit?
- In the single refactorings implementation, the *Rename Attribute* (Listing 2) was also not detected. Would you see a reason for that?
- In our multiple refactorings implementation, we believe that the abstraction technique [1] could have limited the detection of both refactorings (Listings 3 and 4), because they happened on statements that wrap expressions. We believe in that because both refactorings were successfully detected by *RefactoringMiner* in our single (incremental) refactorings implementation. The abstraction technique could also be the reason for the false negative in Listing 7, if we would follow the same thought. Would our understanding be wrong?
- We do not understand why the *Move and Rename Attribute* was detected as a *Move Attribute* (Listing 5), either in our multiple refactorings implementation or a single one. Would you see some reason for this?
- As for Listing 6, this is only a warning regarding the *RefactoringMinger* version available on December 17th;

before refactoring

```
1 public class H extends C{
2     //Rename Variable/Parameter...
3     public void mH(String v) {
4         String r = v + "Rename Variable/Parameter
              refactorings!";
5         System.out.println(r);
6     }
7 }
```

after refactoring

```
1 public class H extends C{
2     //Rename Variable/Parameter...
3     public void mH(String in) {
4         String out = in + "Rename Variable/Parameter
              refactorings!";
5         System.out.println(out);
6     }
7 }
```

Listing 6: [TP] *Rename Variable/Parameter* on November 7th and [FN] on December 17th

- We believe that the abstraction technique can limit a potential validation of the packages, in the case shown in Listing 8. Further, we suppose that the result obtained from the code, in Listing 9, is due to the abstraction not considering the attributes' values. Would our understanding be wrong?

References

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

before refactoring

```
public int getZeroValue() {
    return isZero() ? 1 : 0;
}
public boolean isZero() {
    return(tmp == 0);
}
```

after refactoring

```
public int getZeroValue() {
    return (tmp == 0) ? 1 : 0;
}
```

Listing 7: [FN] *Inline Method*

before transformation

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

Listing 8: [FP] *Push Down 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;
  }
}
```

Listing 9: [FP] *Pull Up Atrribute*