Preconditions on Rename Refactorings

Soumya Mudiyappa
West Chester University
fs926226@wcupa.edu

Swetchha Shukla
West Chester University
ss928947@wcupa.edu

Yung-Chen Cheng
West Chester University
yc917559@wcupa.edu

I. Introduction

Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing". However the cumulative effect of each of these transformations is quite significant. By doing them in small steps we reduce the risk of introducing errors. We also avoid having the system broken while carrying out the restructuring - which allows us to gradually refactor a system over an extended period of time. [1]

In simple words, refactoring is a behavior preserving code transformation technique, for example- rename, move, extract etc. Refactoring rely on two important factors - precondition checks and code changes. In this paper, we brief about the rename refactoring and explain the preconditions for each type of rename refactorings.

II. Rename Refactorings

Rename Refactoring changes the name of identifiers in a program without changing the program's behavior. There are five types of Rename Refactoring in Java:

1) Rename Class Declarations
2) Rename Package Declarations
3) Rename Method Declarations
4) Rename Local Variables
5) Rename Field Variables

III. Preconditions of Rename Class Refactoring

Rename Class Refactoring (RcR) changes the name of the class and all references to that class to the new name without changing its behavior. There are certain preconditions required for RcR.

1) The target class cannot be duplicate with any existing class within same package after rename.
2) The target class cannot be duplicate with any imported class from different package after rename.
3) The target class file name cannot be duplicate with any existing java file name within same package after rename.

A. The target class cannot be duplicate with any existing class within same package after rename.

When we rename a class with any existing class name, the Java compiler produces error message about duplicate class name. The classes will be conflicted if we run RcR on the target class using the name of an existing class in the same package. So, we can not have duplicate class names in the same package.

For example, if we run RcR on the class name from A to B as in Fig. 1, then the compiler produces an error message: *"duplicate class: p.B"*.

```
package p;

class A{
}

class B{
}

class C{
}
```

(a) Before

```
package p;

class B{
}

class B{
}

class C{
}
```

(b) After

Fig. 1. **Example of RcR from A to B**

Furthermore, the same situation occurs in nested classes. The examples below show that for RcR we can not use the same name either as inner or as outer class for nested classes like Fig. 2.

```
package p;

public class A{

  class M{
  }

  class N{
  }
}
```

Fig. 2. **Nested Class before RcR**

Example 1: In order to apply RcR for inner class, we should pre-check that we do not use the same name as any of other inner class name. As shown in Fig. 3, when we run RcR on the inner class from M to N, the Java compiler produces an error *"class A.N is already defined in class A"*.

Example 2: In order to apply RcR for outer class, we should pre-check that we do not use the same name as any of the inner class name and vice-versa. As shown in Fig. 4, when we use same name for outer class and inner class, the Java compiler produces an error *"class M is already defined in package p"* for Fig. 4(a) and *"class A is already defined in package p"* for Fig. 4(b).

Therefore, checking whether a class with the same name already exists in a package should be the first precondition for RcR.

```
package p;

public class A{

  class N{
  }

  class N{
  }
}
```

Fig. 3. **Example 1 for Nested Class after RcR from M to N**

```
package p;

public class M{

  class M{
  }

  class N{
  }
}
```

(a) After RcR on Outer Class A to M

```
package p;

public class A{

  class A{
  }

  class N{
  }
}
```

(b) After RcR on Inner Class M to A

Fig. 4. **Example 2 of Nested Class after RcR**

B. The target class cannot be duplicate with any imported class from different package after rename.

If a class is imported from different package, we have to pre-check that the new name of the target class is not duplicate with the imported class after rename refactoring.

In Fig. 5(a), we see that class B is not duplicate with class A and we can apply RcR on class B to any other name except 'A' as mentioned in section III-A. However, in Fig. 5(b), when we apply RcR from *B to C*, Java compiler produces an error *"C is already defined in this compilation unit"*. This is because the compiler cannot distinguish between the *imported class C* of package 'p' and the *existing class C* of package 'q'.

```
package q;
import p.C;

class A{
}

class B{
}
```

(a) Before

```
package q;
import p.C;

class A{
}

class C{
}
```

(b) After

Fig. 5. **RcR from B to C**

This precondition also holds good for renaming a child class. Suppose a Java file has a class and its children classes as shown in Fig. 6. We see that if parent class A imports a class C from package 'p' and if we apply RcR on the child class from B to C or D to C, Java compiler produces an error *"C is already defined in this compilation unit"*. This precondition is applicable for all ancestor class and we have to trace back and check if any of the parent class is importing a class with same name before renaming the child class within the same java file.

```
//A.java

package q;
import p.C;

public class A{
}

class B extends A{
}

class D extends B{
}
```

(a) Before

```
//A.java

package q;
import p.C;

public class A{
}

class C extends A{
}

class D extends B{
}
```

(b) After

Fig. 6. **RcR for child class B to C**

If a parent class imports a class from different package and if the child class is defined in a separate java file, then in that case we can apply RcR on child class to the imported class name.

Therefore, it is essential to pre-check that the target class should not have duplicate name with any of imported class after RcR.

C. The target class file name cannot be duplicate with any existing java file name within same package after rename.

If classes are defined in separate java files, we have to pre-check that the new name of the target class declared in its class file name is not duplicate with any existing Java file name within the same package.

```
//A.java

public class A{
}

class B{
}


//C.java
//empty file
```

(a) Before

```
//C.java

public class C{
}

class B{
}


//C.java
//empty file
```

(b) After

Fig. 7. **RcR from A to C if A is declared in file A.java**

For example, from Fig. 7, when we apply RcR from *A to C* where C.java is an empty file, the Java compiler produces an error *"Compilation unit 'C.java' already exists"*. This is because after RcR the Java file also gets renamed from *A.java to C.java* and therefore creates a conflict for duplicate file name as *C.java* already exists in the same package.

However, we can apply RcR from *B to C* as class B is not declared within its class file name.

IV. Preconditions of Rename Package Refactoring

Rename Package Refactoring (RpR) changes the name of the package and all references to that package to the new name without changing its behavior. There is one precondition required for RpR.

For Rename Package Refactoring (RpR), we can not use duplicate package name within the same source folder. However, we can run RpR with same package name from different Java folders.

For example, we assume that there are two packages in one source folder like Fig. 8, we can not run RpR on package p to q since package q is already exist in the same source folder.

```
package p;

class A{
}

class B{
}
```

(a) package p

```
package q;

class M{
}

class N{
}
```

(b) package q

Fig. 8. **Example of RpR**

V. Preconditions of Rename Method Refactoring

Rename Method Refactoring (RmR) changes the name of the method and all references to that method to the new name without changing its functionality in the program.

There are three types of Rename Method Refactorings:

1) Rename Static Method Declarations.
2) Rename Non-static Method Declarations.
3) Rename Constructor Method Declarations.

There are certain preconditions required for RmR which are applicable to each type of RmR.

1) The target method cannot be a duplicate of an existing method after rename.
2) A duplicate method in a child class cannot have different return type.
3) A duplicate method in a child class cannot reduce visibility.

A. The target method cannot be duplicate of an existing method after rename.

From Fig. 9, we have method *m* and method *n* with same input parameters but different method names. If we do RmR for m to n, the compiler will show the error: *method n() is already defined in class A*. The behavior of the code does not remain the same after doing RmR. Hence, we have to check the target method name can not be the same as the existing method.

If we use the same name of an existing method for RmR like Fig. 9, the compiler will show the error: *method n() is already defined in class A*.

To understand the above precondition, we should know what exactly is duplicate method.

By Definition of duplicate method, there are following properties: 1) The method should have same name. 2) Same Parameter types in order 3) Same number of function parameters

To derive the definition of Duplicate methods, there are certain use-cases to be considered. Some of them are as follows:

```
class A {

  void m(int a) {
  }

  void n(int b) {
  }
}
```

(a) Before RmR

```
class A {

  void n(int a) {
  }

  void n(int b) {
  }
}
```

(b) After RmR

Fig. 9. **Example of RmR**

1) .: Duplicate Methods with convertible Input parameters / Argument ambiguity.

From Fig. 10,we have 2 methods m1 and m2 , m1 has input parameter ‘int’ and m2 has input parameter ‘double’, if we call m2 with integer as the input, m2 internally typecasts the passed integer into double and displays the output bye as shown in Fig. 10(a). If we RmR m2 to m1 and then pass integer as input then the output will be ”hello” as shown in Fig. 10(b).

In Java we have type conversion of a smaller type to a larger type size automatically.

byte < short < char < int < float < double

When the duplicate functions has the larger type size and smaller type is passed then it automatically gets converted to larger type size.The revised rule should be that Duplicate functions can have the different datatype but the property of Widening or Automatic Type Conversion has to be taken care of.

```
class A {

  void m1(int i) {
    System.out.println("hello");
  }

  void m2(double i) {
     System.out.println("bye");
  }

  public static void main(String[] args){
     A a = new A();
     a.m2(100); // outputs bye
  }
}
```

(a) Before RmR

```
class A {

  void m1(int i) {
    System.out.println("hello");
  }

  void m1(double i) {
     System.out.println("bye");
  }

  public static void main(String[] args){
     A a = new A();
     a.m1(100); // outputs hello
  }
}
```

(b) After RmR

Fig. 10. **Duplicate Methods with convertible input parameter**

2) .: Duplicate Methods with input parameters of same parent Class

From Fig. 11, we have 3 classes , Class A, Class B and Class C. Class C has methods m1 and m2 with input parameters of type Class A and B. If we call new C().m1(new B()), it displays the output as "hello". After RmR m1 to m2 and calling new C().m2(new B()) will display the output as bye. In both the methods we are calling input parameter of type class which is associated with same parent class A and the behavior of the output is different. The revised rule should be that duplicate functions can have the parameters of type 'class' but it should not be associated with the same parent class.

```
public class A {
}

class B extends A {
}

class C {

   void m1(A a) {
      System.out.println("hello");
   }

   void m2(B b) {
      System.out.println("Bye");
   }
}
public class Main {

   public static void main(String[] args) {

      new C().m1(new B()); //outputs hello

   }
}
```

(a) Before RmR

```
public class A {
}

class B extends A {
}

class C {

   void m2(A a) {
      System.out.println("hello");
   }

   void m2(B b) {
      System.out.println("Bye");
   }
}
public class Main {

   public static void main(String[] args) {

      new C().m2(new B()); //outputs bye

   }
}
```

(b) After RmR

Fig. 11. **Duplicate Methods with input parameters of same parent Class**

B. A duplicate method in a child class cannot have different return type.

Java does not allow methods with the same signature to have different return types.

In Fig. 12 (a), A.m1() and B.m2() are two non-duplicate methods which can be rename refactored. However, in Fig. 12 (b), after doing RmR from B.m2() to B.m1(), we get compile-error because the duplicate method m1 have different return types which is not allowed. Thus, we need to check this precondition before doing RmR to avoid compile errors.

```
class A {

  void m1() {
    System.out.print ("Hello");
  }
}
class B extends A{

  int m2() {
    System.out.print ("Bye");
  }
}
```

(a) Before RmR

```
cclass A {

  void m1() {
    System.out.print ("Hello");
  }
}
class B extends A{

  int m1() {
    System.out.print ("Bye");
  }
}
```

(b) After RmR

Fig. 12. **Duplicate methods with different return types**

C. A duplicate method in a child class cannot reduce visibility.

The use of access modifiers helps with visible duplicate methods, however, there is a proper hierarchy for access modifiers to be followed for duplicate methods.

public > protected > package-public > private

As shown in 13 (b), the package-public visibility can not be reduced to private for duplicate methods m1(). We also need to check this precondition before doing RmR to the duplicate methods.

```
class A {

  void m1() {
    System.out.print ("Hello");
  }
}
class B extends A{

  private void m2() {
    System.out.print ("Bye");
  }
}
```

(a) Before RmR

```
cclass A {

  void m1() {
    System.out.print ("Hello");
  }
}
class B extends A{

  private void m1() {
    System.out.print ("Bye");
  }
}
```

(b) After RmR

Fig. 13. **Duplicate methods with reduced visibility**

VI. Preconditions of Rename Local Variable Refactoring

A local variable is a variable declared inside a method and it is only accessible inside the method that declared it

Rename Local Variable Refactoring (RvR) changes the name of the local variable and all references to that variable to the new name without changing its behavior. There are certain preconditions required for RvR.

1) The renamed local variable cannot be duplicate with any of the existing local variable in a method or block or constructor.
2) The renamed local variable must not result in variable shadowing.

A. The renamed local variable cannot be duplicate with any of the existing local variable in a method.

From Fig. 14 and Fig. 15, we see two examples of duplicate local variable. In Example 1 when we apply RvR for local variable 'x' to 'num' , the java compiler produces the error as "variable num is already defined in method m1(int)". Similarly we cannot run RvR on 'y' to 'num' or 'x' to 'y' . Renaming local variables to existing local variable in a method creates a conflict of duplicity in local variable.

Therefore, it is essential to pre-check that the target name of local variable should not have duplicate name with any of the existing local variable in a method after RvR.

B. The renamed local variable must not result in variable shadowing.

Shadowing refers to the concept of using two variables with the same name within scopes that overlap. When we do that, the variable with the higher-level scope is hidden because the variable with lower-level scope overrides it. This results in the higher-level variable being "shadowed".

```
public class A {

    void m1(int num) {
       int x = 1;
       int y = 2
    }
}
```

(a) Before

```
public class B {

    void m1(int num) {
        int num = 1;
        int y = 2;
    }
}
```

(b) After

Fig. 14. **RvR from x to num**

```
public class A {

    void m1(int num) {
       int x = 1;
       int y = 2
    }
}
```

(a) Before

```
public class B {

    void m1(int num) {
        int y = 1;
        int y = 2;
    }
}
```

(b) After

Fig. 15. **RvR from x to y**

Suppose a local variable has the same name as one of the field variable(instance variable), the local variable shadows the field variable inside the method block. For Example, from Fig. 16, the class B has field variable 'i' with value 2 and a method (display()). In a method there is local variable 'j' with value 1. After RvR from 'j' to 'i' , when we access the variable in the method, the local variable value will be printed shadowing the field variable.

```
public class B {

    public int i = 2;

    public void display() {
      int j = 1;
      System.out.println(j);
    }

    public static void main(String args[]) {

      B b = new B();
      b.display();      //Outputs 1
    }
}
```

(a) Before

```
public class B {

    public int i = 2;

    public void display() {
      int i = 1;
      System.out.println(i);
    }

    public static void main(String args[]) {

      B b = new B();
      b.display();      //Outputs 1
    }
}
```

(b) After

Fig. 16. **RvR from j to i results in variable shadowing**

Similarly variable shadowing occurs when the same variables are defined in parent and child classes. For Example, from Fig. 17, the field variable in parent class B has variable ‘i’ with value 1 and child class C has a variable ‘k’ with value 3, local variable in method ‘m1’ of class B has the variable name as ‘j’ with value 2. Aft RvR from j to i and k to i ,when we try to access the variable ‘i’ in methods through Class B and C, the respective local variable will be printed shadowing the field variable of its parent class.

```java
public class B {

    int i = 1;

    public void m1() {
      int j = 2;
      System.out.println(j);
    }
}

class C extends B {

    int k = 3;
}

public class Main {

    public static void main(String args[]) {

       B b = new B();
       b.m1();  // outputs 2
       C c = new C();
       c.m1();  // outputs 2
    }
}
```

(b) Before

```java
public class B {

    int i = 1;

    public void m1() {
      int i = 2;
      System.out.println(i);
    }
}

class C extends B {

    int i = 3;
}

public class Main {

    public static void main(String args[]) {

       B b = new B();
       b.m1();  // outputs 2
       C c = new C();
       c.m1();  // outputs 2
    }
}
```

(b) After

Fig. 17. **RvR from j to i and k to i results in variable shadowing**

Therefore, it is essential to pre-check that local variable must not result in variable shadowing after RvR.

VII. Preconditions of Rename Field Refactoring

Rename Field Refactoring (RfR) changes the declaration and references to the field variables to the new name without changing its behavior. Fields are the variables of a class i.e. instance variables and static variables. There are certain preconditions required for RfR.

1) The target name of field variable cannot be duplicate with any existing field within same class after rename.
2) The target name of field variable cannot be duplicate with any local variable within same method after rename.
3) The duplicate field variable in a child class cannot reduce visibility.

A. The target name of field cannot be duplicate with any existing field within same class after rename.

In order to apply RfR we have to pre-check that the new name of the field is not duplicate with any existing field name of any data type within same class. As shown in Fig. 18(b), when we apply RfR from *‘j’ to ‘i’*, Java compiler produces an error *“variable i is already defined in class A”*. This is because of the conflict for duplicate names for field variables. As shown in Fig. 18(c), when we apply RfR from *‘s’ to ‘i’*, Java compiler produces an error *“variable i is already defined in class A”*.

```java
class A {

   int i = 0;
   int j = 2;
   String s = "hi";
}
```

(a) Before RfR

```java
class A {

   int i = 0;
   int i = 2;
   String s = "hi";
}
```

(b) After RfR from j to i

```java
class A {

   int i = 0;
   int j = 2;
   String i = "hi";
}
```

(c) After RfR from s to i

Fig. 18. **RfR on field variables**

B. The target name of field cannot be duplicate with any local variable within same method after rename.

In order to apply RfR we have to pre-check that the new name of field is not duplicate with any local variable name, if field is being used in the same block as local variable.

As shown in Fig. 19(a), the output is ‘0’. However, after we apply RfR from *‘x’ to ‘y’* as shown in Fig. 19(b), the output is ‘5’. Although the compiler have not produced any error but the behavior of code has changed since the output after RfR has changed. This is because if a field variable and a local variable have the same name, then the local variable will be accessed. This process effectively shadows the field variable and therefore known as *Variable Shadowing*. Therefore, it is essential to pre-check that the new name of field is not duplicate with any existing local variable if both are used in same block.

```
class A{

   int x = 0;

   void m(int y){
      System.out.print(x);
   }

   void main(String[] args){
      this.m(5);
   }
}
```

(a) Before

```
class A{

   int y = 0;

   void m(int y){
      System.out.print(y);
   }

   void main(String[] args){
      this.m(5);
   }
}
```

(b) After

Fig. 19. **RfR from x to y**

The same concept is applied to class hierarchies. A field variable declared within a parent class will be shadowed by any variable with the same name in a child class. As shown in Fig. 20(a), the output is ‘0’. However, after we apply RfR on field variable of a child class from j to i as shown in Fig. 20(b), the output is ‘1’. This Variable Shadowing has changed the original behavior of the program after RfR.

```
class A {

   int i = 0;
}

class B extends A {

   int j = 1;
}

public class C {

   public static void main(String[] args){

      B b = new B();
      System.out.println(b.i); // Outputs 0
   }
}
```

(a) Before

```
class A {

   int i = 0;
}

class B extends A {

   int i = 1;
}

public class C {

   public static void main(String[] args){

      B b = new B();
      System.out.println(b.i); // Outputs 1
   }
}
```

(b) After

Fig. 20. **RfR on child class field variable j to i**

Therefore, it is essential to pre-check that the duplicate field variables in parent and child class does not result in variable shadowing.

C. The duplicate field variable in a child class cannot reduce visibility.

The child class variables are able to reduce the visibility of the parent class variables. The use of access modifiers helps with visible duplicate field variables, however there is a proper hierarchy for access modifiers to be followed for duplicate variables.

public > protected > package-public > private

```
public class A {

   public int i = 0;

   public static void main(String[] args){

      B b = new B();
      System.out.print(b.i); // Outputs 0
   }
}

class B extends A {

   private int j = 1;
}
```

(a) Before RcR on child class field j

```
public class A {

   public int i = 0;

   public static void main(String[] args){

      B b = new B();
      System.out.print(b.i); // Error
   }
}

class B extends A {

   private int i = 1;
}
```

(b) After RcR on child class field j to i

Fig. 21. **Visibility reduction not allowed**

As shown in Fig. 21(a), the two field variables ‘i’ and ‘j’ are completely distinct and have completely separate visibilities in different classes. However, on applying RfR on child class variable from ‘j’ to ‘i’ as shown in Fig. 21(b), the public visibility of field variable ‘i’ gets reduced to private and Java compiler produces an error “field B.i is not visible”.

Therefore, it is essential to pre-check that the new name of the field variable does not result in visibility reduction.

REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019.