# 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

***Abstract*—In computer programming and software design, *refactoring* is the process of restructuring existing code by changing the factoring without changing its external behavior. Refactoring is intended to improve the design, structure, and implementation of the software, while preserving its functionality. In simple words, refactoring is a *behavior preserving* code transformation technique, for example- rename, move, extract etc. Refactoring relies on two important factors - *precondition checks and code changes*. In this paper, we explain about the rename refactoring and the preconditions in detail for each type of rename refactorings.**

## 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]

The idea behind refactoring were introduced over decades ago, but scripting of refactorings is poorly supported by current IDEs tools. Today's IDEs lack fine-grained primitive refactorings including restrictive/imprecise preconditions and inconsistent code transformations. There are further more challenges for executing refactoring as it is error-prone, time consuming and laborious work. Scripting refactoring with current IDEs are too slow because precondition checks and AST operation takes largest time which makes scripting refactorings impractical. Therefore, we need tools that automate refactorings scripts with precise preconditions, high speed and reliability and which we may reuse frequently.

## 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 and each of them have different preconditions.

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

## 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 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 Java 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. **Rename Refactoring Class A to B**

Furthermore, the same error 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.

**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. 2, 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. When we use same name for outer class and inner class, for example, from Fig. 2 (a) to Fig. 3, the Java compiler produces an error *"class M is already defined*

```
package p;

public class A {

  class M {}

  class N {}
}
```

(a) Before

```
package p;

public class A {

  class N {}

  class N {}
}
```

(b) After

Fig. 2. **Rename Refactoring Class M to N**

*in package p"* for Fig. 3 (a) and *"class A is already defined in package p"* for Fig. 3 (b).

```
package p;

public class M {

  class M {}

  class N {}
}
```

(a) After Rename Refactoring on Outer Class A to M

```
package p;

public class A {

  class A {}

  class N {}
}
```

(b) After Rename Refactoring on Inner Class M to A

Fig. 3. **Nested Class after Rename Class Refactoring**

Therefore, checking whether a class with the same name already exists in a package should be the first precondition for 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. 4(a), we see that class B cannot be renamed to A as mentioned in section III-A. Furthermore, in Fig. 4(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 Java 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. 4. **Rename Refactoring Class B to C**

From the Fig. 5, any class name and imported class name cannot be duplicate in the same java file. If they have the same class name, Java compiler produces an error *"C is already defined in this compilation unit"*.

If any Java file includes imported class from another package and the target class is defined in a different Java file, then in that case we can apply RcR on class to the imported class name.

```
//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. 5. **Rename Refactoring Class B to C**

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

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

If there are more than one Java file within a package, we have to pre-check that the new name of the target class 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 Rename Refactoring Class A to C

```
//A.java

public class A {}

class C {}


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

(c) After Rename Refactoring Class B to C

Fig. 6. **Rename Refactoring Class to existing Java file name**

For example, from Fig. 6(a), when we apply RcR from *A to C* where C.java is an empty file as shown in Fig. 6(b), 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.

Also the same concept is applicable to other classes in the same package. When we apply RcR from *B to C* as shown in Fig. 6(c), the Java compiler produces same error *"Compilation unit 'C.java' already exists"*.

## IV. 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.*

If two or more methods have same method signature with below conditions, those methods will be called as duplicate methods.

- Same method name
- Same number of parameters
- Same parameter types in order

From Fig. 7, when we apply RmR from m to n, the Java compiler produces an error "*method n() is already defined in class A*". This is because another duplicate method already exists in the class. Hence, we have to check the target method name can not be the same as the existing method.

```
class A {

  void m(int a) {}

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

(a) Before

```
class A {

  void n(int a) {}

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

(b) After

Fig. 7. **Rename Refactoring Method m to n**

To further define the duplicate methods, some of the use-cases are as follows:

*1) Duplicate Methods with convertible Input parameters:* From Fig. 8, method m1 has input parameter type as 'int' and m2 has input parameter type as 'double'. When we call method m2 with int data type as the parameter, int automatically and implicitly type-casted by Java compiler into double and outputs 'bye' as shown in Fig. 8(a).

When we apply RmR from m2 to m1 and call method m1 by passing 'int' as input parameter then the output is 'hello' as shown in Fig. 8(b). Here, it is clearly evident that after RmR, the original output has changed even with pre-checking condition for duplicate method. This implies that the definition for duplicate method is not enough to rely on and we need to revise duplicate methods to prevent these ambiguity.

Below are primitive type coercions (i.e., implicit type conversions). Java allows to convert primitive types without losing information about a numeric value.

**byte < short < int < float < double**

When the duplicate methods have the larger type size and smaller type is passed then it can be converted to larger type size. The revised definition is that the duplicate methods can have the different data type but the property of widening or automatic type conversion has to be taken care of.

```
class A {

  void m1(int i) {
    System.out.print("hi");
  }

  void m2(double j) {
     System.out.print("bye");
  }

  void main(String[] args) {

     // Output: bye
     this.m2(100);
  }
}
```

(a) Before

```
class A {

  void m1(int i) {
    System.out.print("hi");
  }

  void m1(double j) {
     System.out.print("bye");
  }

  void main(String[] args) {

     // Output: hi
     this.m1(100);
  }
}
```

(b) After

Fig. 8. **Rename Refactoring Method m2 to m1**

*2) Non-Duplicate methods Ambiguity:* From Fig. 9(a), methods $m$ and $n$ are non-duplicate methods and the output is 'hi'. However, when we apply RmR from n to m as shown in Fig. 9(b), the Java compiler produces an error *"reference to m is ambiguous as both method m(String,float) in A and method m(Object,int) in A match"*.

```
class A{

  void m(String s, float d) {
     System.out.print(s);
  }

  void n(Object o, int i) {
     System.out.print(o);
  }

  void main(String[] args) {

     // Output: hi
     this.m("hi", 1);
  }
}
```

(a) Before

```
class A{

  void m(String s, float d) {
     System.out.print(s);
  }

  void m(Object o, int i) {
     System.out.print(o);
  }

  void main(String[] args) {

     // Error
     this.m("hi", 1);
  }
}
```

(b) After

Fig. 9. **Rename Refactoring Method n to m**

Hence, it is essential to pre-check that the non-duplicate methods on RmR does not result in argument ambiguity.

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

If a duplicate method exists in both parent and child class, Java does not allow these duplicate methods to have different return types.

From Fig. 10(a), when we apply RmR from B.m2() to B.m1() as shown in Fig. 10(b), the Java compiler produces an error *" m1() in B cannot override m1() in A as return type int is not compatible with void"*. This is because the duplicate method m1 have different return types which is not allowed.

```
class A {

  void m1() {
    System.out.print("hi");
  }
}

class B extends A{

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

(a) Before

```
class A {

  void m1() {
    System.out.print("hi");
  }
}

class B extends A{

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

(b) After

Fig. 10. **Rename Refactoring Method m2 to m1**

### 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**

From Fig. 11(a), when we apply RmR from B.m2() to B.m1(), the package-public visibility gets reduced to private for duplicate methods m1() as shown in 11(b) and Java compiler produces an error *"m1() in B cannot override m1() in A as attempting to assign weaker access privileges; was package"*.

```
class A {

  void m1() {
    System.out.print("hi");
  }
}

class B extends A{

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

(a) Before

```
class A {

  void m1() {
    System.out.print("hi");
  }
}

class B extends A {

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

(b) After

Fig. 11. **Rename Refactoring Method m2 to m1**

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

## V. Preconditions of Rename Field Refactoring

Fields are the variables of a class i.e. instance variables and static variables.

Rename Field Refactoring (RfR) changes the declaration and references to the field variables to the new name without changing its behavior. 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. From Fig. 12(a), when we apply RfR from $j$ to $i$ as shown in Fig. 12(b), 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. Similarly when we apply RfR from $s$ to $i$ as shown in Fig. 12(c), Java compiler produces an error *"variable i is already defined in class A"*. So, irrespective of the data type, we cannot have duplicate names for field variables within same class.

```
class A {

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

(a) Before

```
class A {

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

(b) After Rename Refactoring Field j to i

```
class A {

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

(c) After Rename Refactoring Field s to i

Fig. 12. **Rename Refactoring 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. 13(a), the output is '0'. However, after we apply RfR from $x$ to $y$ as shown in Fig. 13(b), the output is '1'. Although the Java compiler do not produce 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.

The same concept is applied to class hierarchies. A field variable declared within a parent class will be shadowed by

```
class A{

   int x = 0;

   void m(int y) {

      //Output: 0
      System.out.print(x);
   }

   void main(String[] args) {

      this.m(1);
   }
}
```

(a) Before

```
class A{

   int y = 0;

   void m(int y) {

      //Output: 1
      System.out.print(y);
   }

   void main(String[] args) {

      this.m(1);
   }
}
```

(b) After

Fig. 13. **Rename Refactoring Field x to y**

any variable with the same name in a child class. As shown in Fig. 14(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. 14(b), the output is '1'. This Variable Shadowing has changed the original behavior of the program after RfR.

```
class A {

   int i = 0;

   void main(String[] args) {

      B b = new B();

      // Output: 0
      System.out.print(b.i);
   }
}

class B extends A {

   int j = 1;
}
```

(a) Before

```
class A {

   int i = 0;

   void main(String[] args) {

      B b = new B();

      // Output: 1
      System.out.print(b.i);
   }
}

class B extends A {

   int i = 1;
}
```

(b) After

Fig. 14. **Rename Refactoring 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**

As shown in Fig. 15(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. 15(b), the public visibility of field variable $i$ gets reduced to private and Java compiler produces an error *"field B.i is not visible"*.

```
class A {

   public int i = 0;

   void main(String[] args) {

      B b = new B();

      // Output: 0
      System.out.print(b.i);
   }
}

class B extends A {

   private int j = 1;
}
```

(a) Before

```
class A {

   public int i = 0;

   void main(String[] args) {

      B b = new B();

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

class B extends A {

   private int i = 1;
}
```

(b) After

Fig. 15. **Rename Refactoring on child class field j to i**

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

## 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.
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.*

```
public class A {

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

(a) Before

```
public class A {

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

(b) After Rename Refactoring Variable x to z

```
public class A {

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

(c) After Rename Refactoring Variable x to y

Fig. 16. **Rename Refactoring of local Variables**

From Fig. 16 , we see two examples of duplicate local variable. From Fig. 16(b), when we apply RvR for local variable $x$ to $z$ , the Java compiler produces the error as

*"variable z is already defined in method m1(int)"*. Similarly we cannot run RvR on $y$ to $z$ 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".

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. From Fig. 17, after RvR from $y$ to $x$, the output gets changed from '0' to '1'. This is because when we access the variable in the method, the local variable value is printed shadowing the field variable.

```
class A{

   int x = 0;

   A(int y) {

      // Output: 0
      System.out.print(x);
   }

   void main(String[] args) {

      A a = new A(1);
   }
}
```

(a) Before

```
class A{

   int x = 0;

   A(int x){

      // Output: 1
      System.out.print(x);
   }

   void main(String[] args) {

      A a = new A(1);
   }
}
```

(b) After

Fig. 17. **Rename Refactoring Variable y to x**

```
class A {

   int i = 1;

   void main(String[] args) {

       B b = new B();
   }
}

class B extends A{
   B() {

      int j = 2;

      // Output: 1
      System.out.print(i);
   }
}
```

(b) Before

```
class A {

   int i = 1;

   void main(String[] args) {

       B b = new B();
   }
}

class B extends A{
   B() {

      int i = 2;

      // Output: 2
      System.out.print(i);
   }
}
```

(b) After

Fig. 18. **Rename Refactoring Variable j to i**

Similarly variable shadowing occurs when the same variables are defined in parent and child classes. For Example, from Fig. 18, after RvR from $j$ to $i$, the output gets changed from '1' to '2'. This is because when we access the variable $i$ through Class B, the respective local variable is printed shadowing the field variable of its parent class.

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

## VII. 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.

The target package name cannot be duplicate with any existing 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 as shown in Fig. 19, we can not run RpR on package q to p since package p is already exist in the same source folder.

```
//A.java
package p;

class A {}

class B {}


//M.java
package q;

class M {}

class N {}
```

(a) Before

```
//A.java
package p;

class A {}

class B {}


//M.java
package p;

class M {}

class N {}
```

(b)After

Fig. 19. **Rename Refactoring Package q to p**

## VIII. Conclusions

Refactoring is the process of simplifying the design in such a way that it does not alter the external behavior of the code, yet improves its internal structure. The main goal of refactoring is to have reduced complexity, improved performance and code readability, while preserving its functionality. Refactoring relies on two important factors - *precondition checks and code changes*.

Though refactoring was introduced decades ago, there is no standard preconditions set for all the IDEs. The IDEs such as Eclipse, IntelliJ etc follow imprecise preconditions and inconsistent code transformations which fails to preserve the behavior of the code. There are many challenges for executing refactoring as it is prone to error and time-consuming. With the objective of simplifying development of all refactoring tools, refactoring functionality of different IDEs has been examined in order to find opportunities to simplify the design of the existing code.

In this paper, we have explained different sets of preconditions for each type of *Rename Refactoring* (Class, Method,

Field, Local Variable and Package) in detail with examples which gives a clear picture of the importance of refactoring.

In the world of computer programming and software design, refactoring developers are in need of tools which are capable of having automated refactoring scripts with precise preconditions, high speed and reliability. Refactoring engines are standard tools in today's IDEs, yet their reliability does not hold up to a close scrutiny, as we have shown in this paper. Therefore, it is concluded that regular updates and documentation of precondition checks will ease the implementation of Refactoring tools for all the IDEs.

## REFERENCES

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