Document 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*.

The idea behind refactoring were introduced over decades ago, but scripting of refactorings is poorly suppported 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 is a feature that provides an easy way to change the name of identifiers for code symbols without changing its behavior and functionality. Rename can be used to change the names in comments and in strings and to change the declarations and calls of an identifier. There are five types of Rename Refactoring:

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

III. Preconditions of Rename Class Refactoring

Rename Class Declarations is a refactoring feature that changes the name of the class and all references to that class to the new name without changing its behavior and functionality. There are certain preconditions required for Rename Class Refactoring.

1) The target class cannot be duplicate with an existing class within same package after rename.
2) The target class cannot be duplicate with any imported class from another package.
3) The target sub-class cannot be duplicate with any imported class into its parent class from another package.

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

When we try to rename a class with an existing class name, the Eclipse produces syntax error: “Please choose another name”. [2] The classes will be conflicted if we rename 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, a package p contains class: A, B and C, now we want to refactor the class name A to B:

```
package p;
class A{}
class B{}
class C{}
```

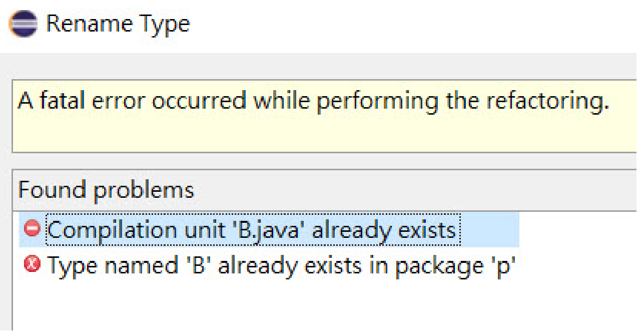
(a) Before Rename Refactoring Class A

```
package p;
class B{}
class B{}
class C{}
```

(b) After Rename Refactoring Class A

Fig. 1. Example of Rename Class Refactoring

Then the java compiler shows up the error that B.java already exists as figure 2:


Rename Type

A fatal error occurred while performing the refactoring.

Found problems

Compilation unit 'B.java' already exists

Type named 'B' already exists in package 'p'

Fig. 2. The error of using same class name for refactoring

So checking whether a class with the same name already exists in a package should be the first job we have to do for rename refactoring.

B. The target class cannot be duplicate with any imported class from another package.

If a class is imported from another package, we have to pre-check that the target class meant to be rename refactored does not match with the imported class. This is because the compiler will not be able to distinguish between the imported class and the existing class.

```
package beforevisitor;
class A{

}
class Test{

}

package aftervisitor;
import beforevisitor.Test;
class A{

}
class B{

}
```

(a) Before Rename Refactoring Class B

```
package beforevisitor;
class A{

}
class Test{

}

package aftervisitor;
import beforevisitor.Test;
class A{

}
class Test{

}
```

(b) After Rename Refactoring Class B to Test

Fig. 3. Precondition when importing a class

In Fig. 3 (a), we see that class B is not a duplicate for class A and class B can be rename refactored to any other name instead of A as mentioned in section III-A. However, in Fig. 3 (b), when we try to rename refactor the class B to class Test, we get compile error *"a compilation unit must not import and declare a type with the same name"* [2]. This is because a class named Test is being imported from another package and on renaming the target class B to Class Test will have compilation error due to duplicate names.

```
package aftervisitor;
public class Visitor {
  public void print() {
      System.out.println("Hello");
  }
}

package beforevisitor;
import aftervisitor.Visitor;
class A {
    public static void main(String[] args) {
        Visitor v = new Visitor();
        v.print();
    }
}
```

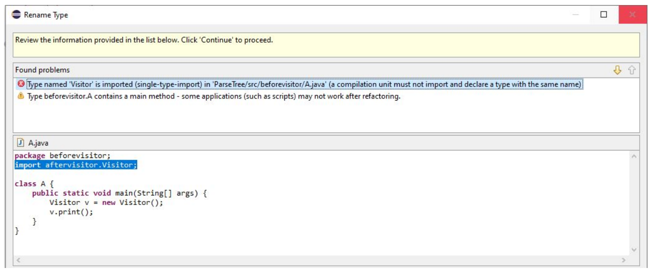
(a) Before Rename Refactoring Class A

```
package aftervisitor;
public class Visitor {
  public void print() {
      System.out.println("Hello");
  }
}

package beforevisitor;
import aftervisitor.Visitor;
class Visitor {
    public static void main(String[] args) {
        Visitor v = new Visitor();
        v.print();
    }
}
```

(b) After Rename Refactoring Class A to Visitor

Fig. 4. Class already defined in Compilation Unit



Review the information provided in the list below. Click 'Continue' to proceed.

Found problems

Type named 'Visitor' is imported (single-type-import) in 'ParseTree/src/beforevisitor/A.java' (a compilation unit must not import and declare a type with the same name)

Type beforevisitor.A contains a main method - some applications (such as scripts) may not work after refactoring.

A.java

```
package beforevisitor;
import aftervisitor.Visitor;

class A {
    public static void main(String[] args) {
        Visitor v = new Visitor();
        v.print();
    }
}
```

Fig. 5. The Compilation Unit Error

Another example where rename refactoring requires a pre-check if the target name is not already defined in the compilation unit is shown below . In Fig. 4 (a), on executing the output generated is *Hello*. However, after rename refactoring class A to Visitor as shown in Fig. 4 (b), we get compile error as shown in Fig. 5.
Therefore, it is essential to pre-check that the target class should not have duplicate name with any of imported class after rename refactoring class.

C. The target sub-class cannot be duplicate with any imported class into its parent class from another package.

If we try renaming a subclass with the same name as the class imported into its parent class from another package, compiler produces the error as ‘a compilation unit must not import and declare a type with the same name’ [2]. To rename a class or subclass for refactoring the code, some of the necessary preconditions have to be checked. This precondition can be explained by the following example.

```
package q;
import p.C;
class A{
}
class B extends A{
}
```

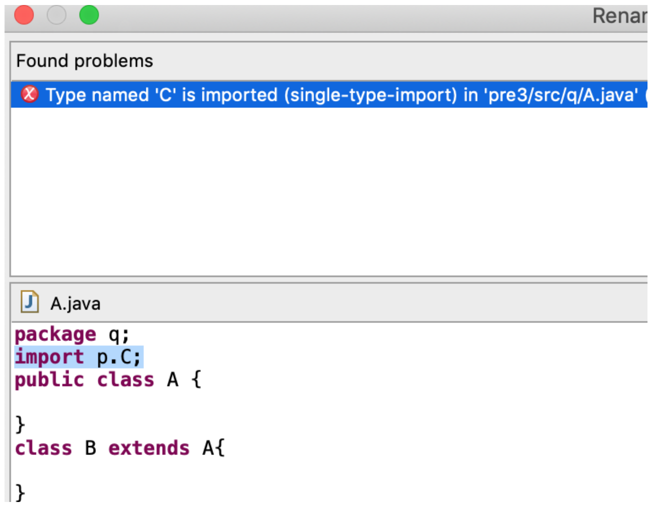
(a) Before renaming Subclass B

```
package q;
import p.C;
class A{
}
class C extends A{
}
```

(b) After Renaming Subclass B to C

Fig. 6. Precondition for Renaming Sub-class

From the above figure 6, We see that if one has imported class C from package p to the parent Class A and if we try to rename the sub-class B to C, Compiler produces the error as given in the figure 7 . As mentioned in section III-B, the same precondition also holds good for renaming a sub-class. Before renaming the sub-class, one has to check the precondition that the target sub-class cannot be duplicate with any imported class into its parent class from another package.



```
Type named 'C' is imported (single-type-import) in 'pre3/src/q/A.java'
```

```
package q;
import p.C;
public class A {

}
class B extends A{

}
```

Fig. 7. Error produced after renaming the sub-class.

IV. Code Change Rules

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019.
[2] “Eclipse Refactorings Properties,” https://git.eclipse.org.