Sound and Extensible Renaming for Java

Max Schäfer, Torbjörn Ekman, Oege de Moor

Daniel Gąsienica
Software Engineering Seminar
May 12, 2009

What Is Refactoring?

"To rewrite existing source code in order to improve its readability, reusability or structure without affecting its meaning or behavior." –Wiktionary

Renaming.

Example: Rename Variable

```
class A {
   int x;
   A(int y) {
      x = y;
   }
}
```

```
class A {
   int x;
   A(int newX) {
      x = newX;
   }
}
```

Example: Rename Variable

```
class A {
   int x;
   A(int y) {
      x = y;
   }
}
```

```
class A {
   int x;
   A(int x) {
      x = x;
   }
}
```

Example: Rename Variable

```
class A {
   int x;
   A(int y) {
      x = y;
   }
}
```

```
class A {
   int x;
   A(int x) {
      this.x* = x;
   }
}
```

*also ((A)this).x or A.this.x

Problems

1. Tools refuse to perform certain refactorings, even though we know they could be done with some minor modifications to the code.

2. Tools perform refactorings that leave you with code that either does not compile or worse, code that suddenly has bugs.

The Real Problems

Preconditions can't be too strong either as some renamings are possible with minor qualifications, e.g. *this* qualifier.

1. Too Strong Preconditions

2. Too Weak Preconditions

Programs that won't compile after refactoring.
Programs where names refer to different declarations after renaming than before the transformation.
The authors of the paper found several bugs in common IDEs such as Eclipse, NetBeans, IntelliJ and JBuilder.

Example: Too Weak Preconditions

```
class A {
    public static void main(String[] args) {
        final int y = 23;
        new Thread() {
            int x = 42;

            public void run() {
                System.out.println(y);
            }
        }.start();
    }
}
```

```
class A {
    public static void main(String[] args) {
        final int x = 23;
        new Thread() {
            int x = 42;

            public void run() {
                System.out.println(x);
            }
        }.start();
    }
}
```

Correctness Criterion

Described in William Opdyke's PhD thesis named “Refactoring Object-oriented Frameworks”

Preserving Behavior

vs

Preserving Entity/Name Bindings

Implementation Strategy

Pragmatism: Behavior Preservation impossible to achieve under dynamic class loading and reflection, anyway.

Correctness Invariant

Only names are affected by the refactoring

and

Each name refers to the same declared entity, before and after the transformation.

Strategy

Creating Symbolic Names by Inverting Lookup Functions*

* based on the JastAdd Extensible Java Compiler (JastAddJ)

Theory

$$lookup_p : access \rightarrow decl$$

$$access_p : decl \rightarrow access$$

$$lookup_p(access_p(d)) = d$$

p : program location
d : declaration

Implementation: Variable Lookup

```
eq Block.getStmt(int i)
   .lookupVariable(String name)
{
   // find local declarations
   Variable v = localVariable(name);
   if(v != null) return v;
   // otherwise delegate to enclosing context
   return lookupVariable(name);
}
```

Implementation: Variable Access Without Qualifiers

```
eq Block.getStmt(int i)
   .accessVariable(Variable v)
{
   Access acc = accessLocal(v);
   if(acc != null) return acc;
   return accessVariable(v);
}
```

Example: Oops

```
class A {
    int x;
    void m() {
        int x;
        •
    }
}
```

```
eq Block.getStmt(int i)
    .accessVariable(Variable v)
{
    Access acc = accessLocal(v);
    if(acc != null) return acc;
    return accessVariable(v);
}
```

Implementation: Fixing the Inversion

```
class A {
    int x;
    void m() {
        int x;
        •
    }
}
```

```
eq Block.getStmt(int i)
    .accessVariable(Variable v)
{
    Access acc = accessLocal(v);
    if(acc != null) return acc;
    acc = accessVariable(v);
    // check for shadowing in block
    if(localVariable(acc) != null)
        return null;  // abort
    return acc;
}
```

Adding Qualifiers

```
class A {
    int x6;
}
class B extends A {
    int x5;
}
class C extends B {
    int x4;

    class D extends F {
        int x1;
        •
    }
}
class E {
    int x3;
}
class F extends E {
    int x2;
}
```

Field name	Source	Bend	Safely qualified access
x1	D	D	this.x1
x2	F	D	super.x2
x3	E	D	((E)this).x3
x4	C	C	C.this.x4
x5	B	C	C.super.x5
x6	A	C	((A)C.this).x6

Adding Qualifiers

```
class A {
    int x6;
}
class B extends A {
    int x5;
}
class C extends B {
    int x4;

    class D extends F {
        int x1;
        •
    }
}
class E {
    int x3;
}
class F extends E {
    int x2;
}
```

Field name	Source	Bend	Safely qualified access
x1	D	D	this.x1
x2	F	D	super.x2
x3	E	D	((E)this).x3
x4	C	C	C.this.x4
x5	B	C	C.super.x5
x6	A	C	((A)C.this).x6

Adding Qualifiers

```
class A {
    int x6;
}
class B extends A {
    int x5;
}
class C extends B {
    int x4;

    class D extends F {
        int x1;
        •
    }
}
class E {
    int x3;
}
class F extends E {
    int x2;
}
```

Field name	Source	Bend	Safely qualified access
x1	D	D	this.x1
x2	F	D	super.x2
x3	E	D	((E)this).x3
x4	C	C	C.this.x4
x5	B	C	C.super.x5
x6	A	C	((A)C.this).x6

Adding Qualifiers: Generating Accesses

```
class A {
    int x6;
}
class B extends A {
    int x5;
}
class C extends B {
    int x4;

    class D extends F {
        int x1;
        •
    }
}
class E {
    int x3;
}
class F extends E {
    int x2;
}
```

```
Access toAccess() {
    VarAccess va = new VarAccess(target.getID());
    if(needsQualifier) {
        if(bend == enclosingType()) {
            if(source == bend)
                return new Dot(new ThisAccess(), va);
            else if(source == bend.getSuper().type())
                return new Dot(new SuperAccess(), va);
        }
        return null;
    } else {
        return va;
    }
}
```

Adding Qualifiers: Generating Accesses

```
class A {
    int x6;
}
class B extends A {
    int x5;
}
class C extends B {
    int x4;

    class D extends F {
        int x1;
        •
    }
}
class E {
    int x3;
}
class F extends E {
    int x2;
}
```

```
Access toAccess() {
    VarAccess va = new VarAccess(target.getID());
    if(needsQualifier) {
        if(bend == enclosingType()) {
            if(source == bend)
                return new Dot(new ThisAccess(), va);
            else if(source == bend.getSuper().type())
                return new Dot(new SuperAccess(), va);
        }
        return null;
    } else {
        return va;
    }
}
```

Adding Qualifiers: Generating Accesses

```
class A {
    int x6;
}
class B extends A {
    int x5;
}
class C extends B {
    int x4;

    class D extends F {
        int x1;
        •
    }
}
class E {
    int x3;
}
class F extends E {
    int x2;
}
```

```
Access toAccess() {
    VarAccess va = new VarAccess(target.getID());
    if(needsQualifier) {
        if(bend == enclosingType()) {
            if(source == bend)
                return new Dot(new ThisAccess(), va);
            else if(source == bend.getSuper().type())
                return new Dot(new SuperAccess(), va);
        }
        return null;
    } else {
        return va;
    }
}
```

Determining Endangered Declarations

Scenario

Renaming entity `x` to `y`.

Strategy*

Sweep entire program for simple names `x` and `y` and consider them **endangered**.

*Yes, it turns out the naïve approach works rather well.

Results

Correctness

Custom test suite (several hundred tests) including tests from **Eclipse Refactoring Test Suite** (~50 tests)
10% contained shadowing/hiding.
Inter-type declarations (AOP) not handled by other tools.

Code Size

~1/3 of Eclipse Refactoring Engine

Performance

Benchmark: Jigsaw webserver (~100K LOC Java 1.4) code base
Locating endangered accesses: ~0.3s
Total time: 1.4 – 3.3s

Conclusion

+ Sound

Sound when each lookup rule in the name analysis has a corresponding inversion rule.

+ Flexible

Since level of qualification in names introduced during renaming can be specified by the developer.

+ Modular

Language constructs can be specified separately.

+ Extensible

Extensible since concept can be applied to new language features and other programming models such as AOP.

~ Formal Verification

~ Automation

Questions?

BACKUP

Adding Qualifiers: Moving Access

```
class A {
    int x6;
}
class B extends A {
    int x5;
}
class C extends B {
    int x4;

    class D extends F {
        int x1;
        •
    }
}
class E {
    int x3;
}
class F extends E {
    int x2;
}
```

```
// returning from parent node
public VarAccessInfo moveInto(ClassDecl td)
{
    if(td.memberField(target.getID())!=null)
        needsQualifier = true;
    return this;
}
// returning from parent type
public VarAccessInfo moveDownTo(ClassDecl td)
{
    if(td.localVariable(target.getID())!=null)
        needsQualifier = true;
    return this;
}
```

Adding Qualifiers: Moving Access

```
class A {
    int x6;
}
class B extends A {
    int x5;
}
class C extends B {
    int x4;

    class D extends F {
        int x1;
        •
    }
}
class E {
    int x3;
}
class F extends E {
    int x2;
}
```

```
// returning from parent node
public VarAccessInfo moveInto(ClassDecl td)
{
    if(td.memberField(target.getID())!=null)
        needsQualifier = true;
    return this;
}
// returning from parent type
public VarAccessInfo moveDownTo(ClassDecl td)
{
    if(td.localVariable(target.getID())!=null)
        needsQualifier = true;
    return this;
}
```

Example: Merging Accesses

```
class A {
    int x;
}

class B extends A {
    int y;
}

class C {
    int m(B b) {
        return b.x;
    }
}
```

Scenario

Renaming `x` to `y`.

Computed Suggestion

`super.y`

Incorrectly Merged

`b.super.y`

Correctly Merged

`((A)b).y`

Access Merging: Rewrite Rules

$$q \oplus n \quad \rightarrow \quad q.n$$

$$q \oplus \texttt{this}.n \quad \rightarrow \quad q.n$$

$$q \oplus \texttt{super}.n \quad \rightarrow \quad ((A)q).n$$

where A is the superclass of q's type

Example: Static Imports

```
import static java.lang.Math.*;
    class Indiana {
        static double myPI = 3.2;
        static double CircleArea(double r) {
            return PI*r*r;
        }
}
```

Example: Static Imports

```
import static java.lang.Math.*;
    class Indiana {
        static double PI = 3.2;
        static double CircleArea(double r) {
            return Math.PI*r*r;
        }
}
```

Implementation

```
syn Variable Block.localVariable(String name)
{
    // iterate over contained statements
    for(Stmt s : getStmts())
        if(s.isVariable(name))
            return (Variable)s;
    return null;
}
```

Implementation

```
syn Access Block.accessLocal(Variable v)
{
    // iterate over contained statements
    for(Stmt s : getStmts())
        if(s == v) // and search for a particular variable
            return new VarAccess(v.getID());
    return null;
}
```