API Evolution with RefactoringNG

Zdeněk Troníček
Department of Software Engineering
FIT CTU in Prague
Czech Republic

***Abstract*—When frameworks and libraries evolve, they often change interfaces. Upgrade to a new version of framework or library leads to changing code manually. We describe a new refactoring tool for the Java programming language that enables automatic migration to a new framework or library, provided that library authors delivered the refactoring rules.**

Keywords-Java, refactoring, API evolution, NetBeans

I. INTRODUCTION

RefactoringNG is a flexible and powerful Java refactoring tool, implemented as NetBeans module. Refactoring rules are described as transformations of source abstract syntax trees to destination abstract syntax trees. For example, the rule that rewrites x = x + 1 to x++ is as follows:

```
Assignment {
   Identifier [name: "x"],
   Binary [kind: PLUS] {
      Identifier [name: "x"],
      Literal [kind: INT_LITERAL, value: 1]
   }
} ->
Unary [kind: POSTFIX_INCREMENT] {
   Identifier [name: "x"]
}
```

The names and structure of abstract syntax trees are the same as in Sun Java compiler. The tool uses Compiler Tree API (com.sun.*) [2], the formal language model in the JDK API (javax.lang.model.*) [3], and the NetBeans infrastructure [4]. In section II we introduce informally the RefactoringNG tool and in section III we show how it can be used to boost migration to a new API.

II. REFACTORINGNG

RefactoringNG is a general refactoring tool. The refactoring performed is defined by a set of refactoring rules. Each rule describes a transformation of one abstract syntax tree (AST) to another AST. The rule has the following form:

```
Pattern -> Rewrite
```

Pattern is an AST in original source code and *Rewrite* is an AST which the original tree will be rewritten to. For example, the rule that rewrites p = null to p = 0 is as follows:

```
Assignment {
   Identifier [name: "p"],
   Literal [kind: NULL_LITERAL]
} ->
Assignment {
   Identifier [name: "p"],
   Literal [kind: INT_LITERAL, value: 0]
}
```

Pattern and *Rewrite* have the same structure:

Tree attributes$_{opt}$ *content*$_{opt}$

(attributes and content are optional).

Trees have the same name as ASTs and attributes have the same name as properties in Sun Java compiler. Attributes are enclosed in [and] and are comma-separated. They specify additional information about the tree. For example, in

```
Literal [kind: NULL_LITERAL]
```

the kind attribute says that literal is the null literal. In *Pattern*, if the attribute is not specified, it may have any value. For example,

```
Identifier
```

means any identifier and

```
Literal [kind: INT_LITERAL]
```

means any int literal. In *Rewrite*, the tree must be described completely so that a new tree can be built. For example, each Identifier in *Rewrite* must have the name attribute specified.

The content of the tree is a comma-separated list of children enclosed in { and }. For example,

```
Binary [kind: PLUS] {
   Literal [kind: INT_LITERAL],
   Literal [kind: INT_LITERAL]
}
```

978-0-7695-4303-1/10 $26.00 © 2010 IEEE
DOI 10.1109/WCSE.2010.107

is addition of two int literals. Children of a given tree must have appropriate types and all of them must be specified if the tree has any content. For example, if Binary has any content, it must have two children (operands) and each of them must be Expression or any subclass. If the content of Binary is missing, the real operands may have any value. That is,

```
Binary [kind: MINUS]
```

means any subtraction. The same tree can also be described as follows:

```
Binary [kind: MINUS] {
  Expression,
  Expression
}
```

As no attribute of Expression is specified, Expression stands for any expression. In place where a tree is expected, any subclass of this tree may be used. For example, an operand of Binary may be any subclass of Expression:

```
Binary [kind: MULTIPLY] {
  Identifier,
  Literal [kind: INT_LITERAL, value: 0]
}
```

The tree hierarchy is the same as in Sun Java compiler.

Some attributes may have more values. Then, the values are separated by |. For example,

```
Binary [kind: PLUS | MINUS]
```

is either addition or subtraction.

Each tree in *Pattern* may have the id attribute. The value of this attribute must be unique in a given rule and serves for referring to the tree from *Rewrite*. For example,

```
Assignment {
  Identifier [id: p],
  Literal [kind: NULL_LITERAL]
} ->
Assignment {
   Identifier [ref: p],
   Literal [kind: INT_LITERAL, value: 0]
}
```

rewrites p = null to p = 0 where p stands for any identifier.

References to attributes are written by #. For example, b#kind refers to the kind attribute of b. The reference to attribute can be used in *Rewrite* as attribute value. For example, the rule that swaps the arguments in the division or remainder operations is as follows:

```
Binary [id: b, kind: DIVIDE | REMAINDER] {
  Identifier [id: x],
  Identifier [id: y]
} ->
Binary [kind: b#kind] {
  Identifier [ref: y],
  Identifier [ref: x]
}
```

(This rule rewrites x / y to y / x and x % y to y % x, where x and y stand for any identifiers.)

The special value 'null' says that the tree is missing. For example, the following rule adds an initializer to variable declarations:

```
Variable [id: v] {
  Modifiers [id: m],
  PrimitiveType [ primitiveTypeKind: INT],
  null
} ->
Variable [name: v#name] {
  Modifiers [ref: m],
  PrimitiveType [ primitiveTypeKind: INT],
  Literal [kind: INT_LITERAL, value: 42]
}
```

III. LISTS

Lists use the same syntax as generic lists in Java. List<T> is a list of elements which are of type T or any subclass of T. For example,

```
List<Expression>
```

is a list of expressions. A list can be used as part of another tree or it can be used at the highest level. For example, the rule that rewrites empty block to the block with empty statement is as follows:

```
Block {
  List<Statement> { }
} ->
Block {
  List<Statement> {
    EmptyStatement
  }
}
```

And the rule that rewrites a list of string literals to another list of string literals is as follows:

```
List<Expression> {
  Literal [kind: STRING_LITERAL, value: "London"],
  Literal [kind: STRING_LITERAL, value: "Paris"]
} ->
List<Expression> {
  Literal [kind: STRING_LITERAL, value: "Prague"]
}
```

The size attribute specifies the number of elements. For example, the rule that rewrites a list of any two literals to the list of single char element is as follows:

```
List<Literal> [size: 2]
->
List<Literal> {
  Literal [kind: CHAR_LITERAL, value: '@']
}
```

The minSize and maxSize attributes specify the range for the number of elements. For example,

```
List<Expression> [minSize: 2, maxSize: 3]
```

is a list that contains two or three expressions. Value * of the maxSize attribute means any size. For example, the rule that rewrites a list of two or more catches to the list of a single catch is as follows:

```
List<Catch> [minSize: 2, maxSize: *]
->
List<Catch> {
  Catch {
    Variable [name: "e"] {
      Modifiers {
        List<Annotation> { },
        Set<Modifier> { }
      },
      Identifier [name: "Exception"],
      null
    },
    Block {
      List<Statement> { }
    }
  }
}
```

IV. NONEOF

NoneOf says that the tree can be anything except the stated trees. For example, the following rule rewrites assignment only if the variable is neither x nor y:

```
Assignment {
  NoneOf<Expression> [id: i] {
    Identifier [name: "x"],
    Identifier [name: "y"]
  },
  Literal [kind: DOUBLE_LITERAL, value: 3.14]
} ->
Assignment {
  Expression [ref: i],
  MemberSelect [identifier: "PI"] {
    Identifier [name: "Math"]
  }
}
```

V. API EVOLUTION

In this section, we will go through several common API changes [5] and show how RefactoringNG can facilitate adoption of the new API. We assume an application that compiles against some API (component, library, or framework). For several structural API changes we show how the library authors can help with adoption of the new API if they provide the refactoring rules for RefactoringNG. Once we have refactoring rules, upgrade to the new API is quick and painless.

A. Moved Method

Let's have the dock method in the Ship class:

```
public class Ship {
  public void dock() { ... }
  ...
}
```

As part of API evolution, we will move the method to the Harbour class and make it static:

```
public class Harbour {
  public static void dock(Ship s) {
    ...
  }
  ...
}
```

So, the code that calls the dock method on instance of Ship must be refactored to call to the dock method on the Harbour class. For example,

```
s.dock();
```

will be replaced with

```
Harbour.dock(s);
```

assuming that s is a variable of type Ship or any subclass.

In RefactoringNG, this can be accomplished by the following rule:

```
MethodInvocation {
  List<Tree> { },
  MemberSelect [identifier: "dock"] {
    Identifier [id: s, instanceof: "navy.Ship"]
  },
  List<Expression> { }
} ->
MethodInvocation {
  List<Tree> { },
  MemberSelect [identifier: "dock"] {
    Identifier [name: "Harbour"]
  },
  List<Expression> {
    Identifier [ref: s]
```

```
  }
}
```

B. Moved Field

Let's assume the DEFAULT_SHIP_CAPACITY field in the Harbour class:

```
public class Harbour {
  public static final int
    DEFAULT_SHIP_CAPACITY = 100;
  ...
}
```

In the process of evolution we will move the field to the Ship class and rename it:

```
public class Ship {
  public static final int
    DEFAULT_CAPACITY = 100;
  ...
}
```

So, we need to replace all occurrences of Harbour.DEFAULT_SHIP_CAPACITY with Ship.DEFAULT_CAPACITY. In RefactoringNG, this can be done by the following rule:

```
MemberSelect [
  identifier: "DEFAULT_SHIP_CAPACITY"] {
  Identifier [elementKind: CLASS,
    qualifiedName: "navy.Harbour"]
} ->
MemberSelect [identifier: "DEFAULT_CAPACITY"] {
  Identifier [name: "Ship"]
}
```

C. Deleted Method

Let's assume the Barge class with the start method which must be called at the beginning of work with the barge:

```
public class Barge extends Ship {
  public void start() { ... }
  ...
}
```

As part of API evolution, we will move the responsibility of this method to the constructor and remove it. So, we can remove all the calls of this method:

```
List<Statement> [id: st, minSize: 1,  maxSize: *] {
  ExpressionStatement [id: del,  pos: *] {
    MethodInvocation {
      List<Tree> { },
      MemberSelect [ identifier: "start"] {
        Identifier [instanceof: "navy.Barge"]
      },
      List<Expression> { }
    }
  }
} ->
List<Statement> {
  ListItems [ref: st, exclude: del]
}
```

D. Changed Argument Types

Let's assume the load method in the Ship class:

```
public class Ship {
  public void load(Cargo c) { ... }
  ...
}
```

In the process of evolution we will change the argument type from Cargo to List<Cargo>:

```
public class Ship {
  public void load(List<Cargo> c) {
    ...
  }
  ...
}
```

In RefactoringNG, we first will add import of java.util.Collections:

```
List<Import> [id: imports]
->
List<Import> {
  ListItems [ref: imports],
  Import [static: false] {
    MemberSelect [ identifier: "Collections"] {
      MemberSelect [ identifier: "util"] {
        Identifier [name: "java"]
      }
    }
  }
}
```

And then we will replace a call to the original method with the call to the new method:

```
MethodInvocation {
  List<Tree> { },
  MemberSelect [identifier: "load"] {
    Identifier [id: s, instanceof: "navy.Ship"]
  },
  List<Expression> [id: args] {
    Expression [kind: IDENTIFIER | NEW_CLASS]
  }
} ->
MethodInvocation {
```

```
  List<Tree> { },
  MemberSelect [identifier: "load"] {
    Identifier [ref: s]
  },
  List<Expression> {
    MethodInvocation {
      List<Tree> { },
      MemberSelect [ identifier: "singletonList"] {
        Identifier [ name: "Collections"]
      },
      List<Expression> [ref: args]
    }
  }
}
```

VI. Extra Argument

Let's assume the sail method in the Ship class:

```
public class Ship {
  public void sail() { ... }
  ...
}
```

As part of API evolution, we will add an argument to this method:

```
public class Ship {
  public void sail(int speed) { ... }
  ...
}
```

So, we need to replace invocations of the no-argument method with the one-argument one:

```
MethodInvocation {
  List<Tree> { },
  MemberSelect [identifier: "sail"] {
    Identifier [id: s, instanceof: "navy.Ship"]
  },
  List<Expression> { }
} ->
MethodInvocation {
  List<Tree> { },
  MemberSelect [identifier: "sail"] {
    Identifier [ref: s]
  },
  List<Expression> {
    Literal [kind: INT_LITERAL, value: 42]
  }
}
```

VII. Related Work

Refactoring is well described in literature. For example, book [1] explains many kinds of refactoring in detail. Also the problem of automatic upgrade to a new version of API has already been investigated. For example, in [6], a similar approach based on syntactic and semantic analysis is described, and in [7], the problem is addressed by capturing and replaying refactoring actions.

VIII. Conclusion

Software evolution goes along with API changes. When a new version of API is released, usually both old and new versions of API are maintained until all clients migrate to the new version. Nowadays, such migration is usually done manually by programmers. With RefactoringNG we can automate this process in so far that programmer only selects an item in NetBeans menu. This approach can be used if we are able to describe the refactoring required for upgrading as transformation of one AST to another. Although this does not cover all possible API changes, it works satisfactory for common ones and helps to get rid of tedious and uninteresting work in software development life cycle.

Acknowledgment

The RefactoringNG project is a collaborative work of Denis Stepanov and me. Denis is responsible for cooperation with the NetBeans infrastructure.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code," Addison-Wesley, 1999.

[2] JSR 199: Java Compiler API: http://www.jcp.org/en/jsr/detail?id=199.

[3] JSR 269: Pluggable Annotation Processing API. http://jcp.org/en/jsr/detail?id=269.

[4] NetBeans home site: http://www.netbeans.org.

[5] D. Dig and R. Johnson, "The Role of Refactorings in API Evolution," Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 389–398.

[6] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," Proceedings of the International Conference of Software Maintenance, 1996, pp. 359–369.

[7] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 274–283.

[8] Jackpot project: http://wiki.netbeans.org/Jackpot.

[9] RefactoringNG: http://kenai.com/projects/refactoringng.