# RefactoringNG: a Flexible Java Refactoring Tool

Zdeněk Troníček
Faculty of Information Technology
Czech Technical University in Prague
Czech Republic
tronicek@fit.cvut.cz

## ABSTRACT
The Java programming language and the Java API evolve and this evolution certainly will continue in future. Upgrade to a new version of programming language or API is nowadays usually done manually. We describe a new flexible refactoring tool for the Java programming language that can upgrade the code almost automatically. The tool performs refactoring rules described in the special language based on the abstract syntax trees. Each rule consists of two abstract syntax trees: the pattern and the rewrite. First, we search for the pattern and then replace each pattern occurrence with the rewrite. Searching and replacement is performed on the abstract syntax trees that are built and fully attributed by the Java compiler. Complete syntactic and semantic information about the source code and flexibility in refactoring rules give the tool competitive advantage over most similar tools.

## Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

## General Terms
Languages, Design, Management

## Keywords
Java, API evolution, refactoring, software evolution.

## 1. INTRODUCTION
The Java programming language is probably the most popular programming language on our planet. It was introduced in 1996 and since then it evolved from a simple language with C syntax to a mature programming language with advanced features such as parametric polymorphism. New language features have different motivations: improvement of readability and simplification (autoboxing, enhanced for loop, varargs, static import, strings in switch, try-with-resources, diamond, underscore in integral literals, multi-catch), type safety (generics, enums), language compile-time safety (annotations), etc. All these changes were designed with regards to backward compatibility. Backward compatibility allows combining new language constructs with old sources. This facilitates adopting new language features but also leads to projects that mix several language versions. Using several language versions in a project worsens code readability and maintainability and therefore it is almost always better to upgrade the legacy code to new language features.

Simultaneously with language evolution, the Java API evolves as well. When a method in the Java API is replaced by a new one, the old method is usually marked deprecated and both methods are being maintained in parallel. For example, the `enable()` method, which was replaced by `setEnable(boolean b)` in JDK 1.1 more than ten years ago, is still present in JDK 7. This approach has two main drawbacks: first, API designers must maintain several APIs in parallel which is tedious. Second, it inhibits API evolution because API designers are restricted by the deprecated methods. It would help if we were able to upgrade code to a new API version automatically. Unfortunately, nowadays such upgrades are usually done manually.

In this paper we describe the main idea behind a new refactoring tool called RefactoringNG that enables automatic upgrade of Java source code. RefactoringNG does not have any fixed set of refactorings but offers a special rule language that can be used to define them. The rule language describes transformations of the abstract syntax trees. The tool exploits the Java compiler and the Compiler Tree API [2,11] to get access to the compiler abstract syntax trees. When the abstract syntax trees are built, RefactoringNG transforms them as described by refactoring rules and converts the transformed trees back to source code. The abstract syntax trees are always fully attributed. This means that in refactoring rules we can use information from semantic analysis. For example, each method call is resolved and for each identifier we know its type. Using the attributed abstract syntax trees enables more precise code transformations than if only the plain abstract syntax trees were used.

The rest of the paper is structured as follows: section 2 describes the rule language, section 3 is devoted to implementation, section 4 contains comparison with competitors, and section 5 outlines the future work. We describe the implementation for the Java programming language; however, the idea behind the tool can be implemented for other languages as well.

## 2. RULE LANGUAGE
In this section, we introduce the rule language that is used to describe refactorings.
A refactoring rule describes transformation of one abstract syntax tree (AST) to another AST. Each rule consists of two trees:


*SAC'12*, March 25-29, 2012, Riva del Garda, Italy.


This work has been supported by research program MSM6840770014.

**Pattern** and **Rewrite**. *Pattern* is the AST in original source code and *Rewrite* is the AST that will replace the original AST when the rule is applied. For example, the tree that describes assignment `x=42` is as follows:

```
Assignment {
   Identifier [name: "x"],
   Literal [kind: INT_LITERAL, value: 42]
}
```

Each tree has a name (e.g. `Assignment`) and may have attributes in brackets ([ and ]) and content in braces ({ and }). For example, `Identifier` may have the `name` attribute. Attributes specify the tree properties and the content specifies tree children. For example, the `value` attribute at `Literal` is the literal value and the content of `Assignment` specifies the left-hand and right-hand side of assignment.

Children of a given tree must be of appropriate types and either all of them or none must be specified. For example, `Binary` must have either no or two children (operands) of the `Expression` type.

Some attributes may have multiple values. In such case the values are separated by |. For example,

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

is either addition or subtraction.

The rule is written in the form **Pattern -> Rewrite**. For example, the rule that rewrites 1+2 to 3 is as follows:

```
Binary [kind: PLUS] {
   Literal [kind: INT_LITERAL, value: 1],
   Literal [kind: INT_LITERAL, value: 2]
} ->
Literal [kind: INT_LITERAL, value: 3]
```

Each tree in *Pattern* may have the `id` attribute that enables references to this 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 is any identifier.

References to attributes are written using `#`. For example, `b#kind` refers to the `kind` attribute of `b`. The reference to an attribute can be used in *Rewrite* as attribute value. For example,

```
Binary [id: b, kind: PLUS | MULTIPLY] {
   Identifier [id: x],
   Literal [id: c, kind: INT_LITERAL]
} ->
Binary [kind: b#kind] {
   Literal [ref: c],
   Identifier [ref: x]
}
```

rewrites x + c to c + x and x * c to c * x, where x is any identifier and c is any integer literal.

The special value `null` means that the tree is not present. For example,

```
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]
}
```

adds an initializer to the variable declaration.

`NoneOf` enables to state that the tree may be anything except the given trees. For example,

```
Class [elementKind: CLASS] {
   Modifiers,
   List<TypeParameter>,
   NoneOf<Tree> [nullable: true] {
      Identifier [elementKind: CLASS,
         qualifiedName: "java.lang.Number"]
   },
   List<Tree>,
   List<Tree>
}
```

describes any class that does not extend `java.lang.Number`. The `nullable` attribute at `NoneOf` says that the tree may be missing (i.e. it may be null). In this case it means that the superclass does not have to be specified.

## 2.1 Trees and Attributes

RefactoringNG supports all the ASTs of the Oracle/Sun Java compiler in JDK 1.6 (see Table 1 and [15]).

**Table 1. The ASTs supported in RefactoringNG.**

| | |
|---|---|
| Annotation | ArrayAccess |
| ArrayType | Assert |
| Assignment | Binary |
| Block | Break |
| Case | Catch |
| Class | CompilationUnit |
| CompoundAssignment | ConditionalExpression |
| Continue | DoWhileLoop |
| EmptyStatement | EnhancedForLoop |
| Erroneous | Expression |
| ExpressionStatement | ForLoop |
| Identifier | If |
| Import | InstanceOf |
| LabeledStatement | Literal |
| MemberSelect | Method |
| MethodInvocation | Modifiers |
| NewArray | NewClass |
| ParameterizedType | Parenthesized |
| PrimitiveType | Return |
| Statement | Switch |
| Synchronized | Throw |
| Tree | Try |
| TypeCast | TypeParameter |
| Unary | Variable |

| | |
|---|---|
| `WhileLoop` | `Wildcard` |

In addition, there are a few artificial nodes that either correspond to AST properties or were added to facilitate defining the refactoring rules (see Table 2).

**Table 2. The artificial trees in RefactoringNG.**

| |
|---|
| `List` – a list of trees |
| `ListItem` – a list item |
| `ListItems` – list items |
| `Modifier` – a Java modifier, e.g. public or volatile |
| `NoneOf` – a tree that must not be present |
| `Set` – a set of modifiers |

Lists use the same syntax as generic lists in the Java programming language. List<T> is the list of elements of the T type. For example, `List<Expression>` is the list of expressions. A list can be used either at the highest level or as part of a tree. For example,

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

describes a block with one empty statement.

The `minSize` attribute specifies the minimum number of elements. For example,

```
Try {
   Block {
      List<Statement> [minSize: 1]
   },
   List<Catch> {
      Catch {
         Variable [
           elementKind: EXCEPTION_PARAMETER,
           name: "e"] {
            Modifiers,
            Identifier [
               elementKind: CLASS,
               qualifiedName:
                  "java.lang.Exception"],
            null
         },
         Block {
            List<Statement> { }
         }
      }
   },
   null
}
```

describes a try block with at least one statement and empty catch block that catches `java.lang.Exception`.

The `maxSize` attribute specifies the maximum number of elements. For example,

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

is a list of two or three expressions. Value `*` of the `maxSize` attribute means unbounded 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> { }
      }
   }
}
```

`ListItem` in a `List` refers to a list item of another `List`. For example, the following rule will replace *if (<expression>) { statement ... }* with *if (<expression>) { statement }*:

```
If {
   Expression [id: cond],
   Block {
      List<Statement> [id: then, minSize: 1]
   },
   null
} ->
If {
   Expression [ref: cond],
   Block {
      List<Statement> {
         ListItem [ref: then, pos: 0]
      }
   },
   null
}
```

`ListItems` in a `List` refers to items of another `List`. For example, the following rule adds value 3 to a list of integer values 1 and 2:

```
List<Expression> [id: args] {
   Literal [kind: INT_LITERAL, value: 1],
   Literal [kind: INT_LITERAL, value: 2]
} ->
List<Expression> {
   ListItems [ref: args],
   Literal [kind: INT_LITERAL, value: 3]
}
```

The `begin` and `end` attributes at `ListItems` select a range of elements from the original list. For example, the following rule omits the first argument of each invocation of the `addAll` method:

```
MethodInvocation {
   List<Tree> { },
   Identifier [name: "addAll"],
   List<Expression> [id: args]
} ->
MethodInvocation {
   List<Tree> { },
   Identifier [name: "addAll"],
   List<Expression> {
      ListItems [ref: args, begin: 1]
   }
}
```

Since the `end` attribute is not specified here, its default value is used. The default value of the `end` attribute is the index of the last element.

The `exclude` attribute of `ListItems` specifies which elements are to be omitted from the original list. For example, the following rule removes the catch block for `IOException` from the list of catch blocks:

```
List<Catch> [id: catches,
   minSize: 2, maxSize: *] {
   Catch [id: c, pos: *] {
      Variable {
         Modifiers,
         Identifier [elementKind: CLASS,
            qualifiedName:
               "java.io.IOException"],
         null
      },
      Block
   }
} ->
List<Catch> {
   ListItems [ref: catches, exclude: c]
}
```

The `pos` attribute at `Catch` here means the position in the list. Its value can be either a number (position) or `*` (any position).

`Modifier` and `Set` specify the class, method, or variable modifiers. For example, declaration of `serialVersionUID` is described by the following tree:

```
Variable [name: "serialVersionUID"] {
   Modifiers {
      List<Annotation> { },
      Set<Modifier> {
         PRIVATE STATIC FINAL
      }
   },
   PrimitiveType [primitiveTypeKind: LONG],
   Literal [kind: LONG_LITERAL, value: 1]
}
```

The tree attributes in RefactoringNG (see Table 3) either correspond to the AST attributes (`identifier`, `kind`, `label`, `name`, `primitiveTypeKind`, `simpleName`, `static`, `value`), or refer to values that come from semantic analysis (`elementKind`, `instanceof`, `nestingKind`, `qualifiedName`), or are artificial (`begin`, `end`, `exclude`, `id`, `maxSize`, `minSize`, `nullable`, `pos`, `ref`, `size`).

**Table 3. The tree attributes in RefactoringNG.**

| |
|---|
| `begin` – the low endpoint of the sublist (at `ListItems`) |
| `elementKind` – the kind of the element, e.g. `CLASS` or `ENUM` (at `Class`, `Identifier`, `Method`, `Tree`, and `Variable`) |
| `end` – the high endpoint of the sublist (at `ListItems`) |
| `exclude` – the elements that are to be omitted (at `ListItems`) |
| `id` – the tree unique identifier (at any tree) |
| `identifier` – the member that is to be selected (at `MemberSelect`) |
| `instanceof` – the type of identifier (at `Identifier`) |
| `kind` – the tree kind (at `Binary`, `CompoundAssignment`, `Expression`, `Literal`, `Statement`, `Tree`, `Unary`, and `Wildcard`) |
| `label` – the label (at `Break`, `Continue`, and `LabeledStatement`) |
| `maxSize` – the maximum number of elements (at `List`) |
| `minSize` – the minimum number of elements (at `List`) |
| `name` – the simple (unqualified) name (at `Identifier`, `Method`, `TypeParameter`, and `Variable`) |
| `nestingKind` – the nesting kind of the class, e.g. `ANONYMOUS` or `MEMBER` (at `Class`) |
| `nullable` – the obligation of the tree presence (at any tree except `List`, `ListItem`, `ListItems`, and `Set`) |
| `pos` – the position of tree in the list (any tree except `ListItems`) |
| `primitiveTypeKind` – the primitive type, e.g. `boolean` or `byte` (at `PrimitiveType`) |
| `qualifiedName` – the fully qualified name (at `Class` and `Identifier`) |
| `ref` – the reference to another tree (at any tree) |
| `simpleName` – the simple (unqualified) name (at `Class`) |
| `size` – the number of elements (at `List`) |
| `static` – the block or import kind (at `Block` and `Import`) |
| `value` – the literal value (at `Literal`) |

`Identifier` at `MemberSelect` specifies which member is to be selected. For example, invocation `s.hashCode()` is described as follows:

```
MethodInvocation {
   List<Tree> { },
   MemberSelect [identifier: "hashCode"] {
      Identifier [name: "s"]
   },
   List<Expression> { }
}
```

The `instanceof` attribute specifies the type which the identifier must be assignable to. For example, the following tree selects member `enable` on any identifier that is of type `java.awt.Component` or any subtype:

```
MemberSelect [identifier: "enable"] {
   Identifier [
      instanceof: "java.awt.Component"]
}
```

The `label` attribute specifies the label of the `break` statement, `continue` statement, or labeled statement. For example, `break loop;` is described as

```
Break [label: "loop"]
```

and `continue` with no label is described as

```
Continue [label: null]
```

The `nestingKind` attribute specifies whether the class is anonymous, local, member, or top-level. For example,

```
Class [elementKind: CLASS,
   nestingKind: MEMBER]
```

addresses member classes.

The `qualifiedName` attribute specifies the fully qualified name. For example, a class that does not have any superclass specified and implements a single interface `Runnable` is described as follows:

```
Class [elementKind: CLASS] {
   Modifiers,
   List<TypeParameter>,
   null,
```

```
    List<Tree> {
        Identifier [elementKind: INTERFACE,
          qualifiedName: "java.lang.Runnable"]
    },
    List<Tree>
}
```

# 3. IMPLEMENTATION

In this section, we briefly describe the RefactoringNG implementation. RefactoringNG is implemented as NetBeans [13] module and is tightly coupled with the Oracle/Sun Java compiler. The implementation consists of three parts: the refactoring engine, the rule editor, and the integration with the NetBeans infrastructure. The refactoring engine (i) preprocesses the refactoring rules, (ii) searches for the pattern trees, and (iii) rewrites the pattern trees to the rewrite trees. Each rule is preprocessed into two trees: the pattern tree and the rewrite tree. The pattern tree is used when searching for the pattern and the rewrite tree is used when rewriting a tree. Since the pattern trees and the rewrite trees are similar, we use the same classes to represent them. These classes are structurally analogous to the ASTs. For example, `BinaryPattern` has attributes `leftOperand` and `rightOperand` and `IfPattern` has attributes `condition`, `thenStatement`, and `elseStatement`. Each pattern is a subclass of the `Pattern` class that contains these methods:

```
boolean matches( TreePath parent, Tree node,
Resolver res )

Tree rewrite( TreeMaker make, Resolver res )

<R, D> R accept( Visitor<R, D> v, D d )
```

The `matches` method compares the pattern to the AST node passed in as argument and returns the result of comparison. The comparison is deep: besides attributes, the children are compared as well. The `Resolver` object passed in as the third argument is used to resolve the values of references from *Rewrite* to *Pattern*. These references cannot be resolved until the pattern is found because they may refer to the pattern attributes that are different at each pattern occurrence.

The `rewrite` method returns a new abstract syntax tree that is used as replacement of the *Pattern* occurrence. The `accept` method implements the visitor design pattern. The visitor checks context constraints on the refactoring rule. For example, for each reference it checks that there is a corresponding id attribute and that the referrer and referee are compatible.

The project contains context-aware rule editor with formatting and syntax highlighting (see Figure 1).

The rule is performed by selecting the item from context menu. The menu item is available on a Java file, on a package, and on a project. When the rule is performed on a package or a project, the rule is performed on each Java file in the package or the project, respectively.

To facilitate authoring new rules, the tool contains generator that for given Java source code generates AST description in the RefactoringNG language. The generator is available in context menu and when activated, it opens a new editor window with AST description. The text in editor window may serve as base for a new rule.

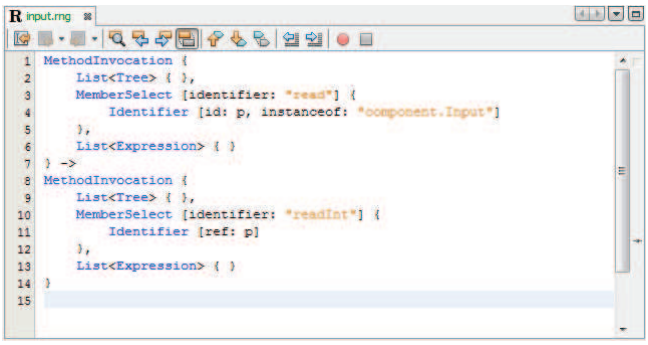


**Figure 1. The context-aware editor of refactoring rules.**

The tool is integrated with the NetBeans refactoring infrastructure and displays changes in the standard NetBeans refactoring window (see Figure 2) before they are applied to source code.

# 4. RELATED WORKS

Refactoring is well described in literature. For example, Fowler, Beck, Brant, Opdyke, and Roberts [7] present many refactorings in detail. Apart from the theoretical interest, refactoring is also highly practical and every Java IDE has some support for refactorings. The support in IDEs may range from simple refactorings such as Rename class or variable to complex ones such as Encapsulate field or Introduce method.

The API evolution was investigated by Dig and Johnson [5]. They examined five frameworks and found out that more than 80% of API breaking changes was caused by refactorings.

Concerning the projects similar to RefactoringNG, the NetBeans IDE [13] contains Jackpot [10] that uses a simple language for description of code transformations. In comparison with the language in RefactoringNG, Jackpot's language is more intuitive but less powerful. For example, Jackpot does not have analogy of the `nestingKind` attribute in RefactoringNG. So, for example, one cannot distinguish between a local and a top-level class in Jackpot.

IntelliJ [9] has 'Structural search and replace' that is similar to RefactoringNG. We specify here two code fragments: one for searching and one for replacement. These fragments are specified in almost pure Java which is undoubtedly advantage over RefactoringNG. On the other hand, RefactoringNG has some features (e.g. attributes `elementKind` and `nestingKind`) that do not have analogy in IntelliJ. Another plus of RefactoringNG is batch processing.

Eclipse [6] supports automatic migration of library JARs. The author of library may capture and store performed refactorings and the client developer may then apply these refactorings to client. Only the refactorings that are implemented in Eclipse are supported.

In comparison to structure-based transformers, like TXL [3], that parse the source code and transform it according to some rules, RefactoringNG offers more information on Java language symbols. In RefactoringNG, complete syntactic and semantic analysis is performed prior to transformation and so we have more information than if only the ASTs were built. For example, given an identifier, we know whether it is class, interface, or enum. This information is available even for types that are declared outside of

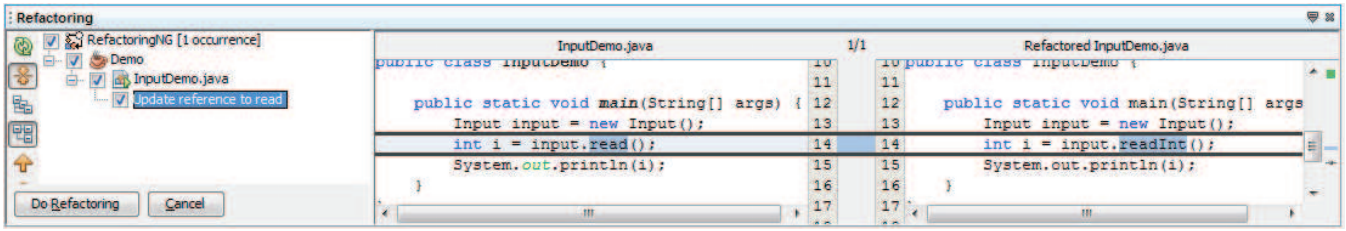


**Figure 2. Confirmation of changes in the NetBeans refactoring window.**

the project that is being processed. On the other hand, systems like TXL enable multiple rewriting.

The problem of automatic upgrade to a new version of API has been examined by several researchers. For example, Henkel and Diwan [8] describe the CatchUp! tool that records refactoring actions when API evolves and then replays these actions in order to upgrade the code that uses this API. Functionally, the tool is a subset of RefactoringNG.

Boshernitsan and Graham [1] describe a tool for interactive transformation of Java program. They designed a transformation language that combines textual and graphical elements and implemented an Eclipse plugin that serves as editor of transformation rules and source code transformer. Unfortunately, the paper does not contain any description of the transformation language and so any comparison is difficult to perform.

Kapur, Cossette, and Walker [12] investigated the problem of dangling references that may appear in source code when a programmer replaces old library on classpath with a new one. They describe a tool, called Trident, that helps programmers to refactor these dangling references.

Dagenais and Robillard [4] present a recommendation system, called SemDiff, that analyzes framework sources in order to find how the framework adapted to its own changes. This knowledge is then used to propose adaptive changes to client programs.

Nita and Notking [14] investigated the problem of adapting programs to alternative APIs. They let programmers specify changes that when applied migrate a program from using one API to using another API. These changes are specified as pairs of Java methods.

## 5. CONCLUSION

Although upgrade to a new language version or new API can be done automatically in many cases, no upgrading tool is widely used and the upgrade is usually done manually which is tedious and error prone. In this paper, we described a refactoring tool for the Java programming language that can help with such upgrades. The tool is called RefactoringNG and is based on pattern matching and AST rewriting.

RefactoringNG is a general and flexible tool that enables to define how the ASTs will be rewritten. However, for code refactoring this is sometimes not enough because AST rewriting does not take into account the context. For example, if we are renaming a variable, to validate that the refactoring is valid, we have to check that no variable with the given name exists in the same scope. Unfortunately, RefactoringNG does not do any context validation currently and checking whether the refactoring is valid or not must be done visually by the user. This deficiency is certainly a direction for future work.

## 6. ACKNOWLEDGMENTS

Denis Stepanov deserves thanks for his contribution to RefactoringNG. He attached the project to the NetBeans window infrastructure and implemented the rule editor.

## 7. REFERENCES

[1] Boshernitsan, M., and Graham, S. L. Interactive transformation of Java programs in Eclipse. *International Conference on Software Engineering*, pp. 791–794, 2006.

[2] *Compiler Tree API.* http://download.oracle.com/javase/6/docs/jdk/api/javac/tree/

[3] Cordy, J. R., Halpern-Hamu, C. D., and Promislow, E. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, Volume 16, Issue 1, pp. 97–107, 1991.

[4] Dagenais, B., and Robillard, M. P. Recommending adaptive changes for framework evolution. *International Conference on Software Engineering,* pp. 481–490, 2008.

[5] Dig, D., and Johnson, R. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 18, Issue 2, pp. 83–107, 2006.

[6] *Eclipse IDE.* http://www.eclipse.org

[7] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the design of existing code.* Addison-Wesley, 1999.

[8] Henkel, J., and Diwan, A. CatchUp!: capturing and replaying refactorings to support API evolution. *International Conference on Software Engineering*, pp. 274–283, 2005.

[9] *IntelliJ IDE.* http://www.jetbrains.com/idea

[10] *Jackpot project.* http://wiki.netbeans.org/Jackpot

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

[12] Kapur, P., Cossette, B., and Walker, J. R. Refactoring references for library migration. *OOPSLA,* pp. 726–738, 2010.

[13] *NetBeans IDE.* http://www.netbeans.org

[14] Nita, M. and Notking, D. Using twinning to adapt programs to alternative APIs. *International Conference on Software Engineering*, pp. 205–214, 2010.

[15] *RefactoringNG project.* http://kenai.com/projects/refactoringng