Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 907–910

ISBN 978-83-60810-22-4

Semi-Automatic Component Upgrade with RefactoringNG

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

***Abstract*—Software components evolve and this evolution often leads to changes in their interfaces. Upgrade to a new version of component then involves changes in client code that are nowadays usually done manually. We deal with the problem of automatic update of client code when the client upgrades to a new version of component. We describe a new flexible refactoring tool for the Java programming language that performs refactorings described by refactoring rules. Each refactoring rule consists of two abstract syntax trees: pattern and rewrite. The tool searches for the pattern tree in client-source-code abstract syntax trees and replaces each occurrence with the rewrite tree. The client-source-code abstract syntax trees are built and fully attributed by the Java compiler. Thus, the tool has complete syntactic and semantic information. Semantic analysis and flexibility in refactoring definitions make the tool superior to most competitors.**

I. Introduction

EVOLUTION of components often leads to changes in their interfaces (API – application programming interface). If a new version of API is not source compatible with the old version, usually both old and new versions of API are maintained in parallel, so that clients that compiled against the old version can compile against the new version as well. For example, if method `enable()` in version 1.0 evolves to method `setEnabled(boolean b)` in version 2.0, both methods will probably be present in version 2.0 and method `enable()` will be marked deprecated. Upgrade from version 1.0 to version 2.0 then involves changes in client code. These changes are nowadays usually done manually which is tedious and error-prone.

In this paper, we describe a new refactoring tool for the Java programming language that enables automatic update of client code so that it compiles against a new API. Upgrade to a new component is not fully automatic because the tool expects refactoring rules that are assumed to be written by component author. Although authoring rules is not easy, the time spent with them may pay off because once we have the rules, we can upgrade thousands of clients in very straightforward way.

This work has been supported by research program MSM6840770014.

The rest of the paper is structured as follows: section II introduces informally the rule language, section III describes API changes and corresponding refactoring rules, section IV discusses shortcomings of the tool, section V compares the tool with competitors, and section VI concludes.

II. Rule language

In this section, we introduce the rule language in which we describe source code transformation. Rather than stating the exact syntax, we introduce the language informally on examples.

A refactoring rule defines transformation of one abstract syntax tree (AST) to another AST. Each has the following form: ***Pattern -> Rewrite***. ***Pattern*** is an AST in original source code and ***Rewrite*** is an AST which the original AST 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 following structure:
Tree Attributes Content (attributes and content are optional). Trees are named as ASTs in Oracle Java Compiler [2,3] and attributes are named as their properties. 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***, the attributes that are not specified match to any value in source code. 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.

Content is a comma-separated list of children of the given tree node enclosed in `{` and `}`. For example,

```
Binary [kind: PLUS] {
```

```
    Literal [kind: INT_LITERAL],
    Literal [kind: INT_LITERAL]
  }
```

is addition of two int literals. Children of a given tree must be of appropriate types and all of them must be specified if the tree has any content. For example, `Binary` always must have two children (operands) if it has any content and either of them must be expression. If content of `Binary` is missing, the operands may have any value in source code. For example, `Binary [kind: MINUS]` means any subtraction.

In place where a tree is expected, any subclass of that tree may be used. For example, operands of `Binary` may be any subclasses of `Expression`:

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

The tree hierarchy is the same as in Oracle Java compiler [2]. Tree attributes may have one or more values. If an attribute has more values, they 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 is instrumental to 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 using `#`. For example, `b#kind` refers to the `kind` attribute of `b`. The attribute reference can be used in ***Rewrite*** as attribute value.

Lists use the same syntax as generic lists in Java. List<T> is a list of elements that are assignable to the T type. A list can be used either at the highest level or as part of another tree.

III. API Changes

In this section, we show how RefactoringNG can help with adapting client code to a new API. We identified API changes caused by refactoring that are suitable for RefactoringNG. These changes are: Rename field, Rename method, Move field, Move method, Add method argument, Delete method argument, Reorder method arguments, Add type argument, Reorder type arguments, Delete type argument, Change instance method to static, Change static method to instance, Add annotation element, Delete annotation element, Rename annotation element, Rename annotation type, Delete annotation type, Nest top level type, and Unnest nested type. For all of them we found refactoring rules that update client code so that it compiles against the updated API. Below we discuss three of these rules.

A. Rename field

Let's rename the `x` field in the `Position` class to `dx`. To update the client code, we have to replace each occurrence of `x` with `dx`. The appropriate refactoring rule is as follows:

```
MemberSelect [identifier: "x"] {
  Identifier [id: p,
    instanceof: "component.Position"]
} ->
MemberSelect [identifier: "dx"] {
  Identifier [ref: p]
}
```

B. Rename method

Let's rename the `read` method in the `Input` class to `readInt`. The rule that replaces invocations of the `read` method with invocations of the `readInt` method is as follows:

```
MethodInvocation {
  List<Tree> { },
  MemberSelect [identifier: "read"] {
    Identifier [id: p,
      instanceof: "component.Input"]
  },
  List<Expression> { }
} ->
MethodInvocation {
  List<Tree> { },
  MemberSelect [
    identifier: "readInt"] {
    Identifier [ref: p]
  },
  List<Expression> { }
}
```

The argument list (`List<Expression>`) enables us to select a method in case of overloading. For example, to address a method invocation with a single string-literal argument, we use the following argument list:

```
List<Expression> {
  Literal [kind: STRING_LITERAL]
}
```

C. Add method argument

Let's add an argument to the one-argument `sail` method in the `Ship` class. The following rule replaces invocations of the one-argument method with the two-argument one:

```
MethodInvocation {
  List<Tree> { },
  MemberSelect [identifier: "sail"] {
    Identifier [id: s,
      instanceof: "component.Ship"]
  },
  List<Expression> [id: args]
} ->
MethodInvocation {
  List<Tree> { },
  MemberSelect [identifier: "sail"] {
```

```
      Identifier [ref: s]
    },
    List<Expression> {
      ListItems [ref: args],
      Literal [kind: INT_LITERAL,
        value: 42]
    }
  }
```

When adding a method argument, we are not restricted to the last position in argument list. Using the `begin` and `end` attributes at `ListItems` we can insert a new argument at arbitrary position.

IV. Evaluation

In this section, we discuss each rule stated in previous section and describe its limits and shortcomings.

A. Rename field

Although we expect the rule is sufficient in many situations, it has two serious shortcomings. First, it renames references to method `x` as well and even though it is a bad practice to have a field and method with the same name, it is allowed in the Java programming language and thus we must count with it. Second, this rule renames only references of form `p.x` where `p` is a variable and there is no simple way how to address references of form `e.x` where `e` is an expression of the `Position` type or any subtype.

B. Rename method

There is no simple way how to address method invocations that have either a string literal or a variable as argument in a single rule. We can address either a string literal by

```
List<Expression> {
  Literal [kind: STRING_LITERAL]
}
```

or a variable by

```
List<Expression> {
  Identifier [
    instanceOf: "java.lang.String"]
}
```

but we cannot address both in one rule. So, if we want to update code in both cases, we need two rules.

Even worse situation is when the argument is an expression. There is no simple way how to address a method invocation with argument of specific type. This may cause problems if the method is overloaded:

```
public class Input {
  public int read(int i) { ... }
  public int read(String s) { ... }
}
```

If we want to rename `read(int i)` to `readInt`, there is no simple way how to address all invocations of this `read`. In addition, if the `read` method is declared with `int` and `Integer` arguments as follows:

```
public class Input {
  public int read(int i) { ... }
  public int read(Integer i) { ... }
}
```

there is no way how to distinguish between these two methods. Note that the following list of arguments matches either of them:

```
List<Expression> {
  Identifier [
    instanceOf: "java.lang.Integer"]
}
```

Overriding causes problems too. For example, given classes `Input` and `ExtInput`:

```
public class Input {
  public int read() { ... }
}
public class ExtInput extends Input {
  public int readInt() { ... }
}
```

and the rule that renames `read()` to `readInt()`, we may end up with a program that is semantically wrong because we unintentionally redirect a method call to `readInt()` in `ExtInput`.

Another problematic situation is when we call a method only by name:

```
public class ExtInput extends Input {
  public int m() {
    return read();
  }
}
```

There is no simple way how to address such method call.

C. Add method argument

The shortcoming here is that the tool does not check existence of a method with the same signature as the resulting method. If such method exists, it may happen that we unintentionally redirect the method call to this method. For example, if client declares the `ExtShip` class as follows:

```
public class ExtShip extends Ship {
  public void sail(int direction,
    int speed) { ... }
}
```

we may end up with a result that is semantically wrong.

Many of these shortcomings have a common cause: missing check whether the refactoring is valid in client context. This is why we strongly recommend inspecting the code before applying changes. For that purpose, the proposed changes are displayed in the standard NetBeans refactoring window and user may decide which of them they confirm.

Concerning a tool support to facilitate rule definition, RefactoringNG contains generator that for a given source code generates AST in RefactoringNG syntax. The output can be used as a base for a rule definition. The rules can be prepared in context-aware editor.

V. Related works

Code refactoring is an area that is described extensively in literature. For example, Fowler, Beck, Brant, Opdyke, and

Roberts [1] describe many refactorings in detail. Refactoring is used in many programming languages and programmers usually spend some time doing refactoring every day, either manually or by refactoring tools. For that purpose, every Java IDE offers some kind of refactoring, such as rename or encapsulate, in their menu.

The API evolution has already been investigated too. Dig and Johnson [5] conducted a study of API changes of five frameworks. In all the cases, more than 80% of the API breaking changes were identified as refactorings.

Concerning the projects similar to RefactoringNG, we found the Jackpot project [8] that is part of NetBeans IDE [4]. Jackpot has a simple language for description of code transformation. The language is more intuitive but less expressive than the language in RefactoringNG. For example, in Jackpot, you cannot distinguish between a local variable and a field. In RefactoringNG, we can use the `elementKind` attribute for this.

IntelliJ [10] offers 'Structural search and replace'. You declare here two code fragments: one for searching and one as replacement. In some sense it is similar to RefactoringNG. As for differences, use of Structural search and replace is easier because the code fragments are described in almost pure Java. It also has a few features (e.g. regular expressions) that are not implemented in RefactoringNG. On the other hand, it lacks some RefactoringNG's features (e.g. attributes `elementKind` and `nestingKind`) and does not have batch processing.

As far as we know, no tool for automatic component upgrade is commonly used in Eclipse [11].

The problem of automatic upgrade to a new version of API has been investigated by several researchers. Chow and Notkin [6] describe an approach in which a library author annotates changed library functions with rules. These rules are then used to generate tools that can update client code automatically.

Henkel and Divan [7] describe the CatchUp! tool. The main idea behind the tool is to record and replay refactoring actions. The tool captures refactoring actions when developer evolves API and enables to replay them later. Replaying is used for updating client code. The tool supports only a few low-level refactorings that all can be done in RefactoringNG.

Balaban, Tip, and Fuhrer [12] present a framework for automatic migration between library classes. The framework is implemented as Eclipse plugin and uses a special language for specifying migrations. As for functionality, the framework provides only a subset of RefactoringNG.

Tansey and Tilewich [13] present a tool that infers transformation rules from two versions of a class, one before and one after upgrading. These rules are then used by transformation engine to refactor the application source code. It provides automatic inference of transformation rules and the rule language is more intuitive and readable but less powerful than the language in RefactoringNG.

Nguyen *et al.* [14] present a tool that learns how to adapt the client code to a new API. The tool identifies the API library changes and compares client codes before and after library migration. The comparison serves to identify adaptation patterns that are subsequently applied to other clients. The main limitation of this tool is that it requires a set of source codes that already migrated to a new API.

VI. Conclusion

Evolution of software components often leads to changes in their API. When a new version of API is released, usually both old and new APIs are maintained in parallel so that clients that compile against the old API are not broken when they replace the old component with the new one. This approach to API evolution has two shortcomings: (i) maintaining several versions of API is tedious and (ii) it inhibits API evolution because API designers are restricted by the requirement for backward compatibility. Since the old API must be maintained until all clients migrate to the new API, it is desirable to speed the migration up. In this paper, we described the tool that facilitate code migration to a new API. Although several similar tools exist, none of them is widely used and code changes are usually done manually. This is quite surprising because several researchers already showed that many of these adaptations can be done automatically.

Acknowledgment

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

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 IDE.* http://www.netbeans.org.
[5] D. Dig and R. Johnson. 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] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. *International Conference on Software Maintenance*, pp. 359–368, 1996.
[7] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. *International Conference on Software Engineering*, pp. 274–283, 2005.
[8] *Jackpot project.* http://wiki.netbeans.org/Jackpot.
[9] *RefactoringNG project.* http://kenai.com/projects/refactoringng.
[10] IntelliJ IDE. http://www.jetbrains.com/idea.
[11] Eclipse IDE. http://www.eclipse.org.
[12] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. *OOPSLA,* pp. 265–279, 2005.
[13] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. *OOPSLA*, pp. 295–312, 2008.
[14] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to API usage adaptation. *OOPSLA,* pp. 302–321, 2010.