Design and Implementation of an Extensible and Modifiable Refactoring Tool

Katsuhisa Maruyama
Department of Computer Science
Ritsumeikan University
1-1-1 Noji-higashi Kusatsu
Shiga 525-8577, Japan
maru@cs.ritsumei.ac.jp

Shinichiro Yamamoto
Department of Information Systems
Aichi Prefectural University
1522-3 Ibaragabasama Kumabari Nagakute-cho
Aichi-gun Aichi 480-1198, Japan
yamamoto@ist.aichi-pu.ac.jp

Abstract

Refactoring is an essential and useful practice in developing and maintaining object-oriented software since it improves the design of existing code without changing its external behavior. Therefore, several refactoring tools tend to be integrated into contemporary IDEs. However, these tools represent source code as an abstract syntax tree (AST) and thus their implementations are hard to extend and modify. This paper presents Jrbx, a refactoring tool that uses a fine-grained XML representation of source code and supports stylized manipulations of the representation. Moreover, Jrbx aggressively exploits control flow graphs (CFGs) and program dependence graphs (PDGs) for both precondition checking and change creation. The use of the XML, CFG, and PDG representations makes the implementation of Jrbx more understandable and reusable, and thus facilitates tool developers creating new refactorings and modifying existing ones.

1. Introduction

Source code understanding is one of essential activities in software maintenance. Refactoring [21, 7, 19] helps maintainers (or programmers) to understand unfamiliar code written by others since it improves the readability of the code without changing its observable behavior. A refactoring tool is particularly useful for increasing comprehensibility of such code. The tool allows the maintainers to actually change the code to reflect their ideas how the code works, and lets them view different code having the same behavior as before. In other words, the maintainers can obtain experimental code to verify their understanding of the original code. If they want to see other experimental code, they can undo applied refactorings (or restore the original contents by using a copy of the original code).

An automated refactoring tool partially frees programmers from tedious checks and careful modifications of source code [9, 24]. Therefore, contemporary integrated development environments (IDEs) tend to include a refactoring tool (or browser). Almost all tools provide many primitive refactorings (e.g., RENAME or MOVE) and some of them support a bit more complex refactorings (e.g., EXTRACTMETHOD). However, each transformation of the existing tools is fixed, that is, its variety is limited.

In the absence of a full cover of every refactoring that programmers want to support, it is desirable that a refactoring tool is extensible and modifiable. However, almost all conventional tools are not interested in how tool developers create new refactorings or modify existing ones. Although pluggable mechanisms (e.g., plug-ins or wizards) have been typically successful to make it easier to add new refactorings or remove existing refactorings, these mechanisms are inadequate for the purpose of extending or modifying the functionality of provided refactorings. For example, the processor/participant architecture of Eclipse [5] allows a developer to freely register processors and participants performing a refactoring by editing XML-based settings. However, he/she can not easily reuse or change parts of the processors or participants since their implementations are not exposed as application programming interfaces (APIs). For a refactoring tool to be truly extensible and modifiable, it should be implemented so that the developers can obtain fine-grained and sufficient information about source code in easy-to-use and easy-to-extend manners. Additionally, it is crucial that they can freely manipulate actual source code or the contents of a representation reflecting it.

This paper presents Jrbx, a refactoring tool that uses a fine-grained and extensible representation of source code and supports standardized and stylized manipulations of the representation. Jrbx builds on the Sapid/XML tool platform [15] that manages source code by using the extensible markup language (XML) [29]. An XML document converted from source code involves fine-grained information resulting from both syntactic and semantic analysis. Therefore, the developers of refactoring tools can build modules for checking preconditions and creating changes by using existing XML processors and trivial wrappers (high-level APIs for accessing XML documents) provided by the platform. Moreover, as compared with an abstract syntax tree (AST) [1], the XML representation is suitable for storing information specific to each refactoring since the developers can easily define new tags and/or attributes (although the extension needs a simple consistency check for a new XML schema). In most cases, the new tags and attributes are processed by only additional modules and ignored in the remaining modules of the original implementation. That is, the required modification is minimized. The effort to learn XML elements (tags and attributes) and standard XML APIs might be the same as to learn AST elements and proprietary APIs since the structure of our use of XML is similar to that of the AST and tag names are based on programmers' view as well as srcML [14].

In addition to the introduction of the XML representation, Jrbx aggressively utilizes a control flow graph (CFG) [1] and a program dependence graph (PDG) [6] to check whether semantic conditions are satisfied and to determine which code fragments will be changed. Although ASTs contain sufficient information needed to automate various kinds of refactorings, it is worth exploiting CFGs and PDGs since they are sophisticated and understandable representations of control and data flow lurking in source code. Both queries and manipulations using such representations can be stylized. The conventional tools not using these graphs would need to further analyze ASTs (or XML documents) to obtain implicit information and thus their implementations would be complex.

The remainder of the paper is organized as follows: Section 2 describes an architecture of Jrbx. Section 3 explains its implementation in detail. Section 4 discusses several observations. Section 5 describes studies related to the implementation of refactoring tools. Finally, Section 6 concludes with a brief summary and future work.

2. Jrbx: Java Refactoring Browser with XML

Transformations in refactoring can be mainly shown in the studies by Opdyke and Fowler. Opdyke proposed 26 low-level refactorings and three high-level ones [21]. Preconditions that should be satisfied prior to each refactoring and procedures to change code are defined. Fowler proposed a catalog of 72 refactorings, consisting of names, motivations, mechanics, and examples [7].

We have developed Jrbx which supports the application of 22 refactorings mainly shown in the Fowler's catalog. The task to check preconditions and change code is automated. The refactorings classified as five for a class, five for a method, seven for a field, three for a local variable or parameter, one for a statement, and one for a lump of code. Jrbx leaves programmers the task to identify which code should be refactored and determine which refactoring should be applied.

An overall architecture of Jrbx is shown in Figure 1. Jrbx consists of three components: the Sapid/XML tool platform [15] (or Sapid/XML in short), CFG/PDG libraries, and Refactor. Sapid/XML manages fine-grained information about Java source code by using XML, called XSDML (extensible software document markup language) [15]. The CFG/PDG libraries construct a CFG and a PDG for each method in source code. They are separated from Refactor so as to make them easier to reuse since such libraries are considered common to various software tools. Refactor uses these two components and transforms target source code into refactored one. In this section, we explain the details of each of the three components.

2.1. Sapid/XML Tool Platform

Sapid/XML originated from Sapid (sophisticated APIs for CASE tool development) [25] that is the tool platform based on fine-grained software repository of C or Java programs. Sapid/XML provides APIs for converting source code into an XSDML document and retrieving an XSDML document corresponding to source code of interest.

The syntactic parser generates a fundamental XSDML document representing Java source code as 20 non-terminal and 7 terminal elements. It adds directly these elements into original source code without eliminating its characters (see Figure 3). For example, the non-terminal element `<Class>`, `<Method>`, `<Field>`, `<Param>`, `<Local>`, `<Stmt>`, `<Type>` or `<Expr>` delimits a class declaration, a method declaration, a field declaration, a formal parameter, a local variable declaration, a statement, a type, or an expression, respectively. All tokens (identifiers, literals, keywords, comments, operators or separators, white spaces, and new lines) remain in the textual contents of the elements `<ident>`, `<literal>`, `<kw>`, `<comment>`, `<op>`, `<sp>`, and `<nl>`, respectively. Attributes are used for expressing additional properties (modifiers, accessibility settings, fully-qualified names, sorts of statements or expressions, and links of code fragments). The whole or portion of the original code can be recovered by only removing tags from its XSDML document.

The semantic analyzer inserts two kinds of link (reference) information: method invocation and field access. The link is either a local or global link. The local link is represented by both the attributes `id` and `defid`. The `defid` indicates the call or access to the element the `id` value of which equals to the `defid` value. The global link across several documents is enhanced by adding the attribute `ref` which indicates a class defining the called method or the accessed field. The process of determining which method

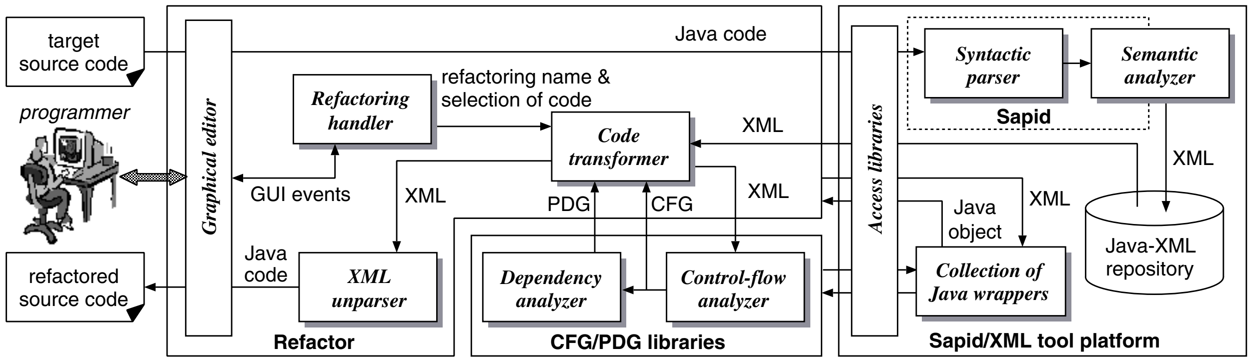


Figure 1. Architecture of Jrbx.

would be called and which field would be accessed is based on the declarative (apparent) type of a related object.

All XSDML documents are stored in the repository and retrieved through access libraries. They can be examined and manipulated by using various XML technologies such as the document object model (DOM) [28], the simple API for XML (SAX) [16], or the extensible stylesheet language (XSL) and XSL transformations (XSLT) [30]. In addition to these low-level APIs, Sapid/XML provides 15 wrappers that are collections of high-level Java APIs. For example, the wrapper JavaClass, JavaMethod, or JavaCall corresponds to the XSDML element `<Class>`, `<Method>`, or `<Expr sort="MethodCall">`, respectively.

2.2. CFG/PDG libraries

The information about CFGs and PDGs can be obtained through our prepared Java classes or respective XML representations. The CFG consists of a set of nodes corresponding to statements and directed edges between the nodes. A statement is either an assignment or a condition predicate, which is marked with a statement tag `<Stmt>` or an expression tag `<Expr>` related to the method call. Every edge represents immediately control flow between two statements, which is either a true-control edge, a false-control edge, or a fall-through edge [3]. An example of the XML representation for a node and an edge of a CFG (of source code in Figure 5) is as follows:

```
<node no="6" id="s826277891">
  <use-var id="s805306372" name="i"/>..</node>
<edge src="6" dst="3"
      sort="TrueCtrlFlow" loopback="yes"/>
```

The attribute `id` indicates the corresponding XSDML element. The attribute `src` or `dst` denotes the value of the attribute `no` of a source or destination node, respectively. The attribute `sort` denotes the kind of flow edges. The `loopback="yes"` means the back-edge for a loop.

The PDG consists of a set of nodes as well as those of the CFG and directed dependence edges between the nodes. The control dependence edge represents a control condition on which the execution of a statement depends. The data dependence edge represents the reachability of data between statements (def-use chain for a variable or a parameter). The def-use chain is classified as either a loop-carried or a loop-independent [12]. An example of the XML representation for an edge of a PDG is as follows:

```
<edge src="6" dst="6" sort="DefUseDep">
  <var id="s805306372" name="i" lc="3"/></edge>
```

The attribute `sort` denotes the kind of dependence edges. The attribute `lc` in the element `var` indicates a loop-node carrying the edge enclosing `var`.

The current version of the CFG/PDG libraries cannot deal with control flow involving exceptions. To alleviate this limitation, a path edge [10] which indicates control flow for exception handling will be embedded. Moreover, it analyzes the inside of the specified method since the dependency analysis of the whole program is too expensive. It assumes that every parameter would affect the return value of the invocation. The expensive cost of construction and synchronization of the graphs might be solved by on-demand data flow analysis based on the virtual control flow [20].

2.3. Refactor

Refactor is a core component of Jrbx, which restructures existing code without altering its behavior. It consists of four sub-components: a code transformer, a refactoring handler, an XML unparser, and a graphical editor.

The code transformer controls several modules to check preconditions for each refactoring and to rewrite the contents of an XSDML document converted from source code. It receives a refactoring name and a selection of code given by the programmer, and assigns suitable modules to do the specified refactoring. The modules exploit not only XML documents but also Java wrappers, CFGs, and PDGs. Some of the modules might request the programmer to input additional information through the refactoring handler. More details on the implementation of the code transformer and how it manipulates code will be explained in Section 3.

The refactoring handler determines a refactoring the programmer wants to apply based on GUI events arising from his/her instructions on the editor (i.e., the selected code and chosen menu item). Additionally, it records refactorings done in the past and information about them so that the programmer can undo undesired changes.

The XML unparser recovers the refactored code by removing every tag and leaving behind the textual contents of elements. It also informs the graphical editor which texts in the original and refactored code should be highlighted.

The graphical editor provides the capabilities to file and edit source code as well as a standard editor. The programmer can select code, specify a refactoring, input additional information, and preview refactored code through it.

3. Implementation

Jrbx is capable of refactoring programs written in Java 1.4 or earlier. Its implementation contains about 94,000 NCNB (non-comment, non-blank lines of code) of Sapid/XML written in Java, C, and C++, and 10,187 NCNB of Refactor and 3,163 NCNB of the CFG/PDG libraries written in only Java. This section describes the basic design of Jrbx and then explains the use of XML, CFGs, and PDGs in the code transformer.

3.1. Basic Design

The basic design of the code transformer is shown in Figure 2. All implementations of the supported refactorings are classified into six groups (class, method, field, variable, statement, and miscellanea[1]) based on a code fragment selected by a programmer. Settings and checks common to each group are made by a superclass (e.g., MethodRefactoring) of the classes in the group. Moreover, the common algorithm for applying respective refactorings is unified in the template method execute in the abstract class Refactoring by using the template method pattern [8]. The methods for tasks of each refactoring, setUp, precondition, transform, and additionalTransformation, are redefined in its subclasses. The classification and the use of this template method help the developers to find appropriate classes to be modified or grasp the rough behavior without examining the detailed implementation of each refactoring.

Moreover, a class changing code (e.g., MoveMethodTransformer) is separated from a class checking preconditions (e.g., MoveMethod), as shown in Figure 2. There are two reasons for this design. One is derived from the fact that the task of a refactoring is mainly divided into two phases—precondition checking and change creation—and most defects in the implementation of its automation respectively arise from an insufficient check or an incorrect change. This separation facilitates independent tests and helps the developer to detect errors from the actual faults. The other reason is that this separation allows the developer to reuse classes for code changes and combine them when building new refactorings. In fact, the class RenameField uses the class RenameMethodTransformer to renaming accessors of the renamed field, and EncapsulateField uses SelfEncapsulateFieldTransformer to encapsulate a field existing in the class on which it is defined.

If the developers want to modify an existing refactoring, they would create a new subclass inheriting several features from a class of the refactoring, and then replace the old name registered in the refactoring menu with the name of the new class. For example, to add a new precondition to the MOVEMETHOD refactoring, they will create SpecialMoveMethod derived from MoveMethod, redefine the method preconditions, and register the name of the created class. In case of adding a new refactoring, the developers have to create two new classes (e.g., InlineMethod and InlineMethodTransformer derived from MethodRefactoring and RefactoringTransformer, respectively), and then register the name of the former class. An object corresponding to the registered name is instantiated by using Java reflection when each refactoring is applied.

3.2. Using XML to Manipulate Code

The code transformer of Jrbx performs precondition checking and change creation of source code by manipulating the DOM tree of an XSDML document converted from the code. For precondition checking, the developers can use DOM APIs, the wrappers provided by Sapid/XML, and six utility classes (e.g., QueryProject or QueryMethod) provided by Jrbx. For example, the following Java code[2] using DOM checks whether the specified method is native.

```
if (elem.getAttribute("native").equals("yes"))
```

The `elem` denotes a DOM element corresponding to the XSDML element `<Method>`. The code using the wrapper JavaMethod is as follows.

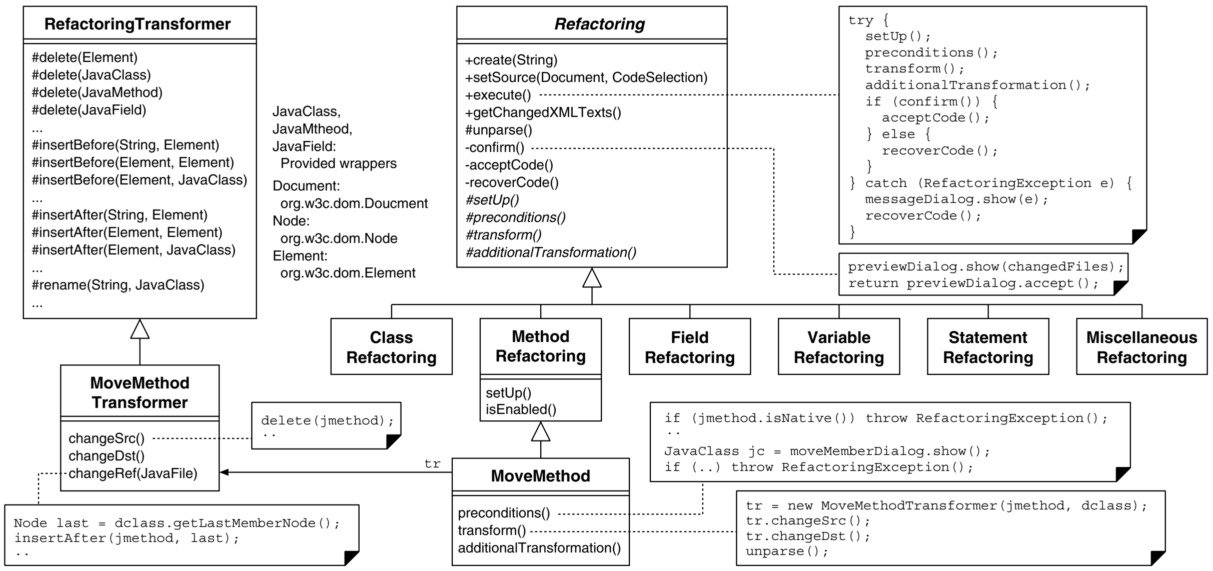
```
JavaMethod jmethod = new JavaMethod(elem);
if (jmethod.isNative())
```

To check complex preconditions, the utility methods are in general used. For example, the following code tests if all methods called by the moved method can be accessed from the class to which the method is moved.

```
for (Iterator it = jmethod.getMethodCallNodes()
     .iterator(); it.hasNext(); ) {
  JavaCall ja = new JavaCall((Node)it.next());
  JavaMethod jm = ja.getCalledJavaMethod();
  if (!QueryMethod.isAccessible(dst, jm)) ..
```

[1]EXTRACTMETHOD is categorized in the miscellaneous refactoring since the selection contains multiple code fragments.

[2]All constant values are actually defined in the class JXConstants.



```
try {
  setUp();
  preconditions();
  transform();
  additionalTransformation();
  if (confirm()) {
    acceptCode();
  } else {
    recoverCode();
  }
} catch (RefactoringException e) {
  messageDialog.show(e);
  recoverCode();
}
```

```
previewDialog.show(changedFiles);
return previewDialog.accept();
```

```
delete(jmethod);
..
```

```
if (jmethod.isNative()) throw RefactoringException();
..
JavaClass jc = moveMemberDialog.show();
if (..) throw RefactoringException();
```

```
Node last = dclass.getLastMemberNode();
insertAfter(jmethod, last);
..
```

```
tr = new MoveMethodTransformer(jmethod, dclass);
tr.changeSrc();
tr.changeDst();
unparse();
```

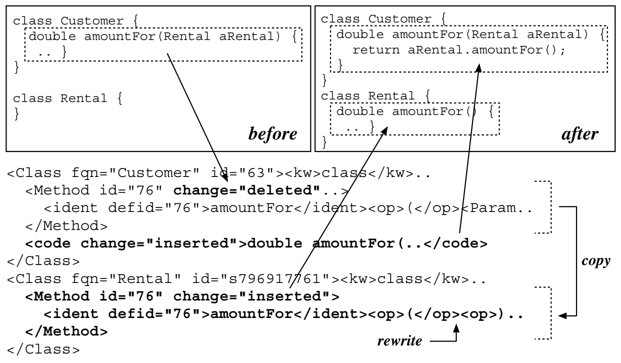
Figure 2. Basic design of the code transformer of Jrbx (part of a class diagram).

The `QueryMethod.isAccessible` is a utility method examining if the class `dst` is accessible to the method `jm`. The implementation of this utility method is also simple since references to the specified method can be easily found by using the attributes `ref` and `defid` of XSDML.

The code change is performed by rewriting of the DOM tree of an XSDML document. For this, the class RefactoringTransformer (see Figure 2) provides several primitive manipulations. For example, the method delete is used for deleting an element (portion of code) from the code, and the method insertBefore or insertAfter is used for inserting a new element before or after a specified element, respectively. The method rename replaces an old name with a new name by calling both the methods delete and insertAfter. For an element corresponding to the specified code fragment, the code transformer basically takes only four actions by calling the methods in RefactoringTransformer. It (1) adds the attribute `change`, the value of which is either `"deleted"`, `"inserted"`, or `"reference"`, (2) inserts the element `code` containing a new text, (3) appends a clone of the portion of the DOM tree to the original tree, and (4) rewrites the textual content of an element.

Figure 3 shows XML manipulation in moving a method. The attribute `change="deleted"` was added to the element `<Method>` of the moved method and its clone with `change="inserted"` was appended to the element `<Class>` corresponding to the target class as its child. The part of clone was rewritten according to the target class. Moreover, a new element `<code>` containing the code for a delegating method was inserted after the moved element.

To recover the original code, the XML unparser collects both elements delimited by the element with `change` `="deleted"` and those not delimited by the element with `change="inserted"`, and then removes every tag and leaves behind the textual contents of the collected elements. In case of the refactored code, it collects both elements delimited by the element with `change="inserted"` and those not delimited by the element with `change="deleted"`. The attribute `change` `="reference"` is used for highlighting notable code.



```
class Customer {
  double amountFor(Rental aRental) {
   .. }
}

class Rental {
}
```

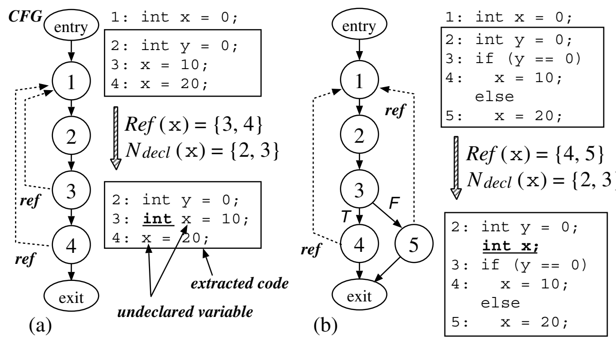
```
class Customer {
  double amountFor(Rental aRental) {
    return aRental.amountFor();
  }
}
class Rental {
  double amountFor() {
   .. }
}
```

```
<Class fqn="Customer" id="63"><kw>class</kw>..
  <Method id="76" change="deleted"..>
    <ident defid="76">amountFor</ident><op>(</op><Param..
  </Method>
  <code change="inserted">double amountFor(..</code>
</Class>
<Class fqn="Rental" id="s796917761"><kw>class</kw>..
  <Method id="76" change="inserted">
    <ident defid="76">amountFor</ident><op>(</op><op>)..
  </Method>
</Class>
```

Figure 3. Manipulation in moving a method.

3.3. Using CFGs and PDGs

CFGs and PDGs are useful for automating various refactorings and making a refactoring tool more understandable. This is because several complex refactorings require deep and implicit information about control and data flow in source code and these graphs are common representations to capture this information. Nevertheless, most tools confine such information in the task of individual refactorings and are not designed to use them explicitly. Jrbx utilizes



```
1: int x = 0;
2: int y = 0;
3: x = 10;
4: x = 20;
```

```
2: int y = 0;
3: int x = 10;
4: x = 20;
```

```
1: int x = 0;
2: int y = 0;
3: if (y == 0)
4:    x = 10;
   else
5:    x = 20;
```

```
2: int y = 0;
   int x;
3: if (y == 0)
4:    x = 10;
   else
5:    x = 20;
```

Figure 4. Source codes and their CFGs.

CFGs and PDGs in automation of three refactorings, EXTRACTMETHOD, SPLITTEMPORARYVARIABLE, and REPLACECONDITIONALWITHPOLYMORPHISM. Due to space limitation, we partially explain the use of CFGs and PDGs in EXTRACTMETHOD and SPLITVARIABLE.

3.3.1 EXTRACTMETHOD

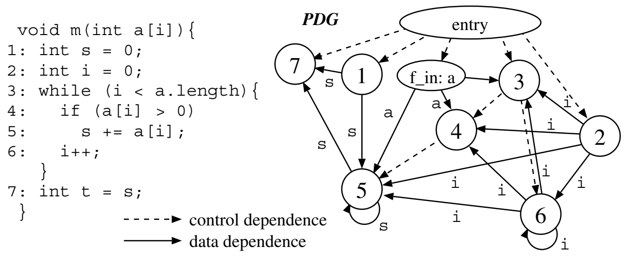
The EXTRACTMETHOD refactoring turns part of the original code into a new method. Here G_M, G_X, or G_R is a CFG of the original method, the new extracted method including the selected code, or the method consists of remaining code, respectively. The node set of a graph G (a CFG or PDG) for a method is denoted by $N(G)$. In this refactoring, CFGs and PDGs are mainly used in respective two ways.

CFG-1: Every execution flow must end in a **return** statement if the extracted method returns the value to the calling method. Jrbx traverses control flow on the CFG of the extracted method and collects all flows from a node except **return** to a node not included in the method. If both flows from a **return** node and a non-**return** node are detected, the selection is considered to be invalid.

CFG-2: Jrbx determines a suitable position where a variable in the extracted or remaining method should be declared, by calculating the intersection of all reachable paths from the entry node of G to nodes referring to the variable.

$$N_{decl}(v) = \bigcap_{q \in Ref(v)} \{\, p \in N(G) \mid p \overset{*}{\mapsto}_f q \,\},$$

where G is either G_X or G_R, and v is a variable which is declared outside G. $Ref(v)$ is a set of nodes in G referring to v, and $p \mapsto_f q$ denotes a control flow from a node p to a node q except a back edge for loop. The relation $\overset{*}{\mapsto}_f$ means the reflexive and transitive closure of the relation $\mapsto_f$. If $N_{decl}(v)$ contains one or more nodes, the layout-based lowest node $d \in N_{decl}(v)$ is selected. The variable v is declared at d if d permits declaring v, otherwise a new declaration of v is inserted before d. An empty set of $N_{decl}(v)$ means that v has no need to be declared in the method for G.



```
 void m(int a[]){
1: int s = 0;
2: int i = 0;
3: while (i < a.length){
4:   if (a[i] > 0)
5:      s += a[i];
6:   i++;
   }
7: int t = s;
 }
```

Figure 5. Source code and its PDG.

In Figure 4(a), the variable x would be undeclared in the extracted code. Accordingly, the declaration (the underlined "int") which is the type of x was added at the node 3 because it is the layout-based lowest node of $N_{decl}(\mathtt{x})$. In Figure 4(b), the new declaration (the underlined "int x") was inserted before the node 3 because it (**if**-statement) is the layout-based lowest node of $N_{decl}(\mathtt{x})$ but does not permit declaring a variable. Theoretically, the behavior of refactored code is preserved by inserting all declarations into either the beginning of the extracted method or immediately after the invocation to the extracted method in remaining method. However, it is undesirable that a variable declaration is far from its reference.

PDG-1: All statements in a selection of code must be enclosed by a single statement (or block) which exists immediately outside the selection since extracting statements not satisfying this condition give rise to non-executable code. Jrbx uses control dependence of PDGs for this validation. Consider the following set:

$$N_{dom} = \{\, p \in N(G) \mid p \rightarrow_c q \wedge p \notin N(G_X) \ \wedge\ q \in N(G_X) \ \wedge p \neq \textbf{do} \,\},$$

where $p \rightarrow_c q$ denotes a control dependence from a node p to a node q. N_{dom} is a set of the outside nodes each of which is not included in the selection but its destination of the control dependence is included in the selection. The selection is valid only if a unique outside node is found (i.e., the size of N_{dom} is one) under the condition that all tokens of nodes dominated by the nodes in the selection were judged to be included in the selection. The reason to except a control dependence with respect to a statement enclosed in a **do**-block is that such statement is usually dominated by both a **do**-statement and its parent statement. Jrbx removes more internal one of the detected two control dependences.

For the code in Figure 5, the selection $\{2, 3, 4, 5, 6\}$ or $\{4, 5\}$ is valid because the N_{dom} is $\{entry\}$ or $\{3\}$, respectively. On the other hand, the selection $\{2, 3\}$ is invalid because nodes 4, 5, and 6 are dominated by the node 3 but tokens of these nodes are not included in the selection. The selection $\{5, 6\}$ is also invalid because N_{dom} is $\{4, 3\}$.

PDG-2: Jrbx also uses PDGs for detecting a local variable the value of which either passes into the extracted method as a parameter or goes back to the calling method as a return. V_{in} is a set of variables each of which data dependence existing from a node not included in the extracted method to a node included in it.

$$V_{in} = \{\, u \in V(G_X) \mid p \to_d^u q \wedge p \notin N(G_X) \wedge q \in N(G_X) \,\},$$

where $p \to_d^u q$ denotes a data dependence from a node p to a node q due to a variable u. $V(G_X)$ is a set of all local-scope variables (local variables and parameters) appearing in G_X. A variable in V_{in} will become a parameter. In contrast, V_{out} is used for detecting candidates for a return variable.

$$V_{out} = \{\, u \in V(G_X) \mid p \in N(G_X) \wedge [\, p \to_d^u q \wedge q \notin N(G_X) \vee p \to_{d(l)}^u q \wedge l \notin N(G_X) \,] \,\},$$

where $p \to_{d(l)}^u q$ denotes a loop-carried data dependence from a node p to a node q carried by a loop node l. If V_{out} is empty, no return variable is needed. If V_{out} has only one variable, it will become a return variable. If V_{out} contains two or more variables, the selection is invalid since the extracted method can not return the values of plural variables.

For example, consider the selection $\{2, 3, 4, 5, 6\}$ in Figure 5. The variables `a` and `s` are determined to be parameters because the values of `a` and `s` come from the nodes `f_in:a` and 1, respectively. Moreover, `s` is determined to be a return variable because the value of `s` goes to the node 7. As another example, the selection $\{4, 5, 6\}$ is invalid because both `s` and `i` are candidates for a return variable.

3.3.2 SPLITVARIABLE

The SPLITVARIABLE refactoring makes a separate temporary variable for each responsibility. All references to be separated along with the variable are determined by using a PDG G_M of the original method M.

Jrbx repeats the following two steps until the number of separated nodes (N_{sep} described below) does not increase.

Step 1: It collects nodes that might affect the value of a specified variable u at a node q by backward traversing data-dependence edges on G_M. N_{aff} is a set of the affecting nodes determined as follows.

$$N_{aff}(u, q) = \{\, p \in N(G_M) \mid p \overset{*}{\to}_d^u q \,\},$$

where $p \overset{*}{\to}_d^u q$ denotes the reflexive and transitive closure of a data dependence $p \to_d^u q$.

Step 2: It gathers nodes in which the nodes collected at the step 1 might affect the references of u by traversing forward data-dependence edges on G_M. N_{sep} is a set of the affected nodes, which will be separated.

$$N_{sep}(u, q) = \{\, r \in N(G_M) \mid p \in N_{aff}(u, q) \wedge p \overset{*}{\to}_d^u r \,\}.$$

If all nodes defining the value of the variable u are included in N_{sep}, it is not split because such splitting leads to the same result as the RENAMEVARIABLE refactoring is applied. Jrbx changes every variable name appearing in N_{sep} to a new name. Then it determines a suitable position where the separated variable should be declared in the same way of the procedure **CFG-2** of EXTRACTMETHOD. Strictly, this implementation slightly differs from the mechanics of the SPLITTEMPORARYVARIABLE refactoring proposed by Fowler [7]. Jrbx separates a temporary variable that might be assigned several times while the original makes a separate variable for each assignment.

4. Discussion

This section discusses several observations regarding the benefits and performance of Jrbx.

4.1. Noteworthy Code in Implementation

Each refactoring often requires information specific to itself. For example, the EXTRACTMETHOD refactoring permits selecting usual local declarations but forbids selecting local declarations in initializers ("ForInit" parts) of **for**-statement. That is, these declarations must be distinguished. For this, Jrbx introduced the attribute `sort="For"`. It analyzes XSDML elements of local declarations and their parent elements by using DOM APIs, and then adds such attribute to elements enclosed in "ForInit". This procedure was written in the method setUp of the class ExtractMethod. The up-front analysis and manipulation separates the task for collecting information about source code from that for manipulating the code. Accordingly, Jrbx simplifies the implementation of precondition checking or change creation.

As another example, XSDML (or XML) documents free the developers to query and manipulate source code. Consider the following Java code that collects types used in a code fragment of interest (e.g., a class, method, field, statement, or expression).

```
NodeList nl = elem.getElementsByTagName("Type");
for (int i = 0; i < nl.getLength(); i++) {
  Element e = (Element)nl.item(i);
  JavaType jt = new JavaType(e); ..
```

Any XSDML element can be specified as `elem` in the code. That is, this code is available for every XML element and thus it is not needed to prepare respective APIs. In case that the developers want to create a new refactoring that collects used types in the range other than that of the existing refactoring, they can reuse this code without changing it. Moreover, this property has an advantage in order to understand existing refactorings without reading various kinds of code.

Next consider the following code that validates a selection by using PDGs in EXTRACTMETHOD.

```
GraphCompSet dom = new GraphCompSet();
Iterator it = pdg.getEdges().iterator();
while (it.hasNext()) {
  Dependence edge = (Dependence)it.next();
  if (edge.isCD()) {
```

```
PDGNode src = (PDGNode)edge.getSrcNode();
PDGNode dst = (PDGNode)edge.getDstNode();
if (!src.getCFGNode().isDoSt() &&
    !PDGNodes.contains(src) &&
    PDGNodes.contains(dst)) {
  dom.add(src); ..
```

The pdg is an object of the class PDG provided by Jrbx, which denotes the PDG of the original method. As space is limited, we will not explain details of the code. Nevertheless it is reasonable to suppose that this code is very similar to the logical formula described in **PDG-1** of Section 3.3. The implementation conforming to a formal procedure of a refactoring facilitates the developers understanding, modifying, and testing it.

4.2. The Size of Code in Implementation

To demonstrate the advantages of Jrbx, several experiments with Jrbx and other refactoring tools will be made, and their extensibility and modifiability must be carefully measured, but it is hard to do this. Hence, we estimated how much is the burden of reading code to understand and change existing refactorings by counting lines of code included in the implementation of Jrbx.

Table 1 shows how many NCNB (non-comment, non-blank) lines of code have been written for each refactoring group of Refactor. The "Common" indicates the number of lines of code common to each group (e.g., MethodRefactoring in Figure 2) and the "Utility" denotes the number of lines of code common to every group (e.g., RefactoringTransformer in Figure 2 or the utility class QueryMethod). The values in Table 1 do not include the number related to code for dialog windows.

In addition, we observed six (or four) refactorings shown in Table 2. The "Jrbx" and "Eclipse" show how many NCNB lines of code are included in Jrbx and Eclipse 3.0 implementations. These values do not include the number related to common and utility modules. Note that the less value of MOVEMETHOD in "Eclipse" indicates the number of lines of code moving only instance methods, and the value in parentheses contains the number of lines of code moving static members (not only static methods but also classes and fields). The value of MOVEMETHOD in "Jrbx" denotes the number of lines of code moving both instance and static methods (Jrbx can also move static classes and fields but its implementation is divided into different classes). The maximum and minimum values of EXTRACTMETHOD in "Eclipse" indicate the numbers of lines of code including and not including modules for flow analysis.

The "Passed" in Table 2 means the ratio of the number of passed test cases to all test cases provided by Eclipse[3]. This shows the similarity of respective refactorings. For example, Jrbx passed all 200 test cases of the Eclipse's

[3]Test cases not passed in Eclipse itself were eliminated.

Table 1. Number of NCNB lines in Jrbx.

Refactoring	# refac.	Precond.	Transform.	Common	Total
Class	5	321	197	34	552
Method	5	859	361	36	1256
Field	7	638	282	47	967
Variable	3	218	142	46	406
Statement	1	188	165	65	418
Miscellanea	1	642	311	47	1000
Total	22	2866	1458	275	4599
Average	-	130	66	13	209
Utility	-	-	-	-	2903

Table 2. Number of NCNB lines and passed test cases in each refactoring.

Refactoring	Jrbx	Eclipse	Passed
RENAMEMETHOD	275	461	200 / 200
RENAMEFIELD	207	451	50 / 54
MOVEMETHOD	630	1617 (2615)	65 / 73
EXTRACTMETHOD	953	1239 $\sim$ 2963	204 / 229
SPLITVARIABLE	215	N/A	—
REPLACECONDITIONAL	353	N/A	—

RENAMEMETHOD refactoring and thus it can be considered to provide as the same functionality as Eclipse's RENAMEMETHOD. The failure does not indicate an error in testing but arises from the difference of functionality. For example, in RENAMEFIELD, one failure was caused by the difference between respective rules of renaming accessors. Moreover, Jrbx does not rewrite Javadoc comments according to code change by refactorings. On the other hand, Eclipse does not permit a renamed field to have the same name as any local variable although those names would cause no conflict. In MOVEMETHOD, Eclipse moves an adjacent comment (not Javadoc) along with the moved methods. Moreover, Eclipse can move two methods together but Jrbx prohibits this movement. Referring to Table 2, the functionality of the top three refactorings of Jrbx and Eclipse are considered to be almost the same or similar. The ratio (204/229) with respect to EXTRACTMETHOD does not indicate accurate similarity because Jrbx can not extract a method from **try-catch** blocks or expressions and the ratio was calculated without including test cases related to such method extraction.

The results in Tables 1 and 2 would help estimating the cost of inspecting the existing refactorings or creating a new refactoring although it might not directly show the effect on the extensibility and modifiability of Jrbx. Moreover, the bottom three refactorings could be all completed with few lines of code (953, 215, and 353, respectively) considering the complex refactorings. These results are reasonable since the XML documents we used explicitly contain fully analyzed information, and information about flow and dependence is provided by the CFG/PDG libraries. As another reason, many procedures related to the XML manipulation are shared by the implementations of refactorings as described in Section 4.1. This also agrees with the fact that the number of "Utility" in Table 1 is relatively large.

4.3. Performance

The use of XML makes it easier to extend the existing representation of source code and implement various refactorings, but it sacrifices performance. It is worth noticing the results of the simple experiment with Sapid/XML. According to our experiment[4] reported in [15] (using four programs: Notepad, Stylepad, SwingSet2, and Java2D packaged in the Sun Microsystems J2SDK1.4.2), the size of a converted XSDML document is about 10 times larger than the original source file. Moreover, the processing time (the sum of conversion and simple manipulation time) for each source file is about 7 to 15 seconds. Currently, the semantic analyzer of Sapid/XML is being revised. As a result of an experiment with its latest version, the processing time is improved as about 2.4 to 5.5 seconds (Notepad: 5.5, Stylepad: 3.3, SwingSet2: 2.7, and Java2D: 2.4 seconds, respectively[5]).

These values show that Jrbx consumes much more space and time. Especially, the result of the processing time suggests that Jrbx must pay unreasonable penalties when doing refactorings. If it assumes that one source file involves relationships to ten other source files, the programmer must wait about 26.4 (2.4×11) to 60.5 (5.5×11) seconds until he/she gets factored code. The reason for taking a long time is that Jrbx requires reconverting (and synchronizing) the whole of a source file to be refactored and its related files, and the implementation of the semantic analyzer is immature. To reduce such time, it is needed to cache previously analyzed information and partially convert source files based on the cached information.

5. Related Work

The first refactoring tool is RefactoringBrowser [23] for Smalltalk and its architecture has been followed by many tools. Regarding Java, several refactoring tools including Eclipse [5], IntelliJ IDEA [13], and JRefactory [26] are available[6]. The architecture of Jrbx is strongly influenced by these tools. The conventional refactoring tools usually use ASTs as an internal representation of source code and provide well-named, sophisticated APIs to examine and manipulate the source code. In addition, several tool platforms such as the DMS Software Reengineering Toolkit [4] and RECODER [22] are capable of examining and manipulating source code by using ASTs. Jrbx differs much from these tools and platforms in respect to the use of an XML representation of source code. Of course, ASTs contain sufficient information to implement various refactorings and AST manipulation is quick. However, the AST manipulation might require modification of AST elements or addition of new APIs although the functionality of an existing refactoring is slightly changed. On the other hand, the XML manipulation of Jrbx accommodates slight changes by extending an XML schema and using standard APIs.

Jrbx is analogous to RefaX [17] since they are both flexible refactoring frameworks based on XML representations of source code and their motivations are very similar. One of the main differences between Jrbx and RefaX is the expectation of their used XML representations. Although RefaX aims to be independent of source models, target programming languages, and code manipulation technologies, these benefits are hard to obtain from only the adoption of XML. This is because implementations of complex refactorings much depend on source models and programming languages. Jrbx supports standard and stylized manipulations of source code by exploiting not only XML but also CFGs and PDGs, and provides modifiable implementations of complex refactorings (e.g., EXTRACTMETHOD).

Several XML representations of source code, such as JavaML [2], GXL [11], and srcML [14], have been proposed. The use of these representations in refactoring might be expected. This paper shows a running implementation of the refactoring tool using an XML representation and discusses its features.

The concept of introducing CFGs and PDGs into refactorings is not novel. Griswold and Notkin presented a program transformation tool exploiting these graphs and its usefulness [9]. Nevertheless, no concrete implementation in refactoring has been provided. The three refactorings of Jrbx, which are briefly mentioned in Section 3.3, demonstrate that the use of CFGs and PDGs increases the possibility of automating several refactorings and simplifies the implementation of automated refactorings.

Regarding the use of graphs, Mens, Demeyer, and Janssens suggested a graph representation of source code and formal transformations of refactorings by using graph production rules [18]. Besides the graph representation, Tip, Kiezun, and Bäumer implemented some refactorings related to generalization by using type constraints for source code [27]. The main concern of these approaches is to formalize transformations that preserve the behavior of source code and to increase the reliability of refactoring tools. On the other hand, the purpose of our study is to make refactoring tools more maintainable. Extensible and modifiable implementation is still significant under the situation that not all refactorings can be perfectly formalized or generated.

[4]The experiment was performed on a computer with a Pentium4 2.4GHz CPU and a 640MB of RAM, running RedHat Linux9 and Sun Microsystems J2RE1.4.2_01.

[5]These values ware measured on a computer with a Pentium4 3.0GHz CPU and a 1GB of RAM, running RedHat Linux9 and Sun Microsystems J2RE1.4.2_06.

[6]Many up-to-date tools can be seen in http://www.refactoring.com.

6. Conclusion

Refactoring helps maintainers to understand existing source code by making it more readable and actually showing the changed code to them. We have presented Jrbx, a tool that automates several transformations in refactoring, and described how it has been designed and implemented. Jrbx uses a fine-grained XML representation and two graph representations (CFGs and PDGs) to examine and manipulate source code. Accordingly, Jrbx makes it easier to create new refactorings or modify existing ones.

The development of Jrbx is continuing. There are three issues in its enhancement. The first issue is to improve its scalability and performance as mentioned in Section 4.3. The second one is to increase the applicability to target source code. Refactorings would be usually applied to only complete programs. However, refactoring complete parts of incomplete programs is often required in actual development or maintenance. To agree this request, the extension of the XML presentation is needed. The last and immediate issue is to cooperate with existing IDEs. Jrbx is currently a stand-alone application having a poor GUI editor and has no pretty printer to format refactored code. We are planning to integrate Jrbx into popular IDEs (e.g., Eclipse).

Jrbx and the Sapid/XML platform can be downloaded from http://www.jtool.org/.

Acknowledgments

The authors would like to thank André van der Hoek and Christopher Van der Westhuizen for their valuable suggestions on the early version of this paper and the anonymous reviewers for their helpful comments. We also thank Akinori Yonezawa, Etsuya Shibayama, and all members of the Sapid project including Kiyoshi Agusa. This work was sponsored by the Information-technology Promotion Agency (IPA), Japan.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] G. J. Badros. JavaML: A markup language for Java source code. In *Proc. Int'l WWW Conference*, 2000. http://www9.org/w9cdrom/index.html.

[3] T. Ball and S. B. Horwitz. Slicing programs with arbitrary control flow. In *Proc. Intl. Work. on Automated and Algorithmic Debugging,* LNCS 749, pages 206–222, 1993.

[4] I. D. Baxtor, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proc. ICSE'04*, pages 625–634, 2004.

[5] Eclipse.org. Eclipse. http://www.eclipse.org/.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM TOSEM*, 2(3):228–269, 1993.

[10] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proc. OOPSLA*, pages 312–326, 2001.

[11] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a standard exchange format. In *Proc. WCRE'00*, pages 162–171, 2000.

[12] S. Horwitz, T. Ball, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

[13] JetBrains. IntelliJ IDEA. http://www.jetbrains.com/idea/.

[14] J. I. Maletic, M. L. Collard, and A. Marcus. Source code files as structured documents. In *Proc. IWPC'02*, pages 289–292, 2002.

[15] K. Maruyama and S. Yamamoto. A CASE tool platform using an XML representation of Java source code. In *Proc. SCAM'04*, pages 158–167, 2004.

[16] D. Megginson. Simple API for XML (SAX). http://www.saxproject.org/.

[17] N. C. Mendonça, P. H. M. Maia, L. A. Fonseca, and R. M. C. Andrade. Refax: A refactoring framework based on XML. In *Proc. ICSM'04*, pages 147–156, 2004.

[18] T. Mens, S. Demeyer, and D. Janssens. Formalising behavior preserving program transformation. In *Proc. Graph Transformation, LNCS2505*, pages 286–301, 2002.

[19] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE TSE*, 30(2):126–139, 2004.

[20] J. D. Morgenthaler. Static analysis for a software transformation tool. Technical report, Ph.D. thesis, University of California, San Diego, 1997.

[21] W. F. Opdyke. Refactoring object-oriented frameworks. Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign, 1992.

[22] RECODER. http://recoder.sourceforge.net/.

[23] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[24] D. B. Roberts. Practical analysis for refactoring. Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign, 1999.

[25] Sapid: Sophisticated APIs for CASE tool Development. http://www.sapid.org/.

[26] C. Seguin and M. Atkinson. JRefactory. http://jrefactory.sourceforge.net/.

[27] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *Proc. OOPSLA*, pages 13–26, 2003.

[28] World Wide Web Consortium. Document object model (DOM). http://www.w3.org/DOM/.

[29] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML.

[30] World Wide Web Consortium. The extensible stylesheet language family (XSL). http://www.w3.org/Style/XSL/.