Supporting Refactoring Activities Using Histories of Program Modification

Shinpei HAYASHI\(^{(a)}\), Nonmember, Motoshi SAEKI\(^{(a)}\), and Masahito KURIHARA\(^{(b)}\), Members

SUMMARY Refactoring is one of the promising techniques for improving program design by means of program transformation with preserving behavior, and is widely applied in practice. However, it is difficult for engineers to identify how and where to refactor programs, because proper knowledge and skills of a high order are required of them. In this paper, we propose the technique to instruct how and where to refactor a program by using a sequence of its modifications. We consider that the histories of program modifications reflect developers’ intentions, and focusing on them allows us to provide suitable refactoring guides. Our technique can be automated by storing the correspondence of modification patterns to suitable refactoring operations. By implementing an automated supporting tool, we show its feasibility. The tool is implemented as a plug-in for Eclipse IDE. It selects refactoring operations by matching between a sequence of program modifications and modification patterns.

key words: refactoring, software development environment, modification history, pattern matching

1. Introduction

Refactoring\(^{(1)}\),\(^{(2)}\) is one of the promising techniques for improving program design by means of program transformation with preserving behavior, and is widely taken into practice. It can be applied even to the program that has not been completed, so we can incrementally improve its quality while it is being developed.

However, it is difficult for engineers to identify how and where to refactor programs, because proper knowledge and skills of a high order are required of them. Thus computer-aided refactoring techniques are useful. To automate refactoring activities, some research has been carried out, which can be classified into two categories:

- to construct the tools that automate the execution of pre-defined refactoring procedures (e.g. refactoring browser\(^{(3)}\)),
- to construct the tools that automatically perform refactoring activities and free the programmers from troublesome activities that are manually performed in order to identify where and how to refactor a program.

In this paper, we concentrate on the latter approach. The problems of this approach are the techniques to identify the following:

- where: identifying flaws of a program to be refactored, so-called Bad Smells\(^{(2)}\),\(^{(4)}\),
- which: selecting suitable refactoring operations from the Refactoring Catalog\(^{(2)}\),
- when: notifying programmers when they should refactor the target program in their design and programming process.

To detect bad smells, the techniques of static code analysis are frequently used, for example, searching for pre-defined code patterns\(^{(5)}\),\(^{(6)}\) in a program, selecting refactoring rules using measures via some metrics\(^{(7)}\),\(^{(8)}\), and so on.

As software development progresses, the revision of the program becomes more difficult from the viewpoint of cost consumption and of revision efforts. Therefore, refactoring should be at the early stages of software development processes\(^{(9)}\). Consequently, a mechanism that informs a developer of a refactoring opportunity would be effective in software development processes. Furthermore, the mechanism is embedded into a program editing tool, and automatically and explicitly provides the information on suitable refactoring to the developer during his/her development activities. By achieving this mechanism, the developer can obtain the support of refactoring activities naturally in the sense that he/she need not pay attention to which refactoring should be done where and when.

The essential point to achieve the above mechanism is the way how to get the information on suitable refactoring while a developer is working on a program without interfering with his/her activities. For example, suppose that we adopt static analysis technique of a source code to identify how to refactor a program appropriately. The static analysis technique analyzes all parts of the program and detects the parts to be refactored from the whole program. If the developer is not working on the detected parts but another part of the program, i.e. if he/she is not currently involved in the detected parts, sudden instruction of refactoring to currently irrelevant parts may cause confusion. Thus the focus of the developer is on the part of the program that he/she is currently working on, e.g. inputting and editing just now. Refactoring should be instructed just on the part of the program that he/she is currently working on to make it more efficient. Moreover the static analysis for whole of the program is time-consuming and it is not suitable to execute the
analysis in real-time while the developer is working.

To achieve the above aim, we focus on the editing operations that a developer performs in his/her activities of program modifications. Suppose that the developer repeats copy-and-paste operations of a certain part of the program, e.g. A. As a result, the code clone of A appears in several parts of the modified program, and it shows a sign of code duplication, one of bad smells. At this time, it is preferable that the developer is instructed to refactor the occurrences of the clone A into one so that the clone occurrences are eliminated. The timing, i.e. immediately after the developer performs a sequence of copy-and-paste operations of A, is significant for this refactoring, because his/her current focus is on A. We abstract patterns of program modifications as shown in the above example such as copy-and-paste operations on a part, and make them correspond to refactoring operations. This correspondence can be considered as knowledge of refactoring.

In this paper, we concentrate on the history of program modification which is actually executed by developers. Our system automates the identification of which refactoring is suitable and of where to apply it, by using a sequence of the program modifications executed by developers. Using the positional information of executed modifications, the retrieval space of bad smells could be reduced. Moreover, developers’ intention can be conjectured by finding typical modification patterns from histories. In Extreme Programming, Beck et al. propose Pair Programming practice [10]. In this practice, the observer supports the driver. We aim at a construction of Artificial Observer, that is the implementation of automated observer actor.

We can describe what is the most important difference between the traditional static analysis techniques and our technique. That is, static analysis techniques, such as code clone analyses [11]–[13], are often used in offline but our technique is in online. In the offline approach, all or selected fragments of code are analyzed in order to find opportunities of all or selected refactorings when the developer wants to. On the other hand, in the online approach, the information about refactoring opportunities is provided while the developer is inputting or editing. In the latter approach, a quick analysis technique is required. By using the static analysis techniques, we can analyze code fragments in detail via the offline approach, but cannot via the online approach because of its high time cost of the analysis. Our technique uses the information about modification histories, so the opportunities of refactorings are suggested quickly. Thus we can apply the technique online. These approaches are independent of each other, so developers can use both approaches if necessary.

The rest of this paper is organized as follows. Section 2 shows the essential points of program modifications for supporting refactoring, and illustrates simple examples and a technique to do it automatically. Section 3 includes formal definitions of the concepts necessary for our technique. In Sect.4, we discuss our tool that realizes our technique and its implementation strategies. Furthermore we explain the case study to assess the feasibility of our approach. In Sect. 5, we discuss the scalability and the effectiveness of our technique. Finally, we give our conclusions and suggest future work in Sect. 6.

2. Instructing Refactoring Using Program Modification

In this section, we illustrate two examples to show that histories of program modifications are useful for supporting refactoring activities. Then, based on the consideration of our examples, we propose how to instruct suitable refactoring operations for a developer.

2.1 Using the Information about Code Duplication

Code clone is defined as code fragments similar to an original code fragment, or duplicated code fragments. In order to decrease software management costs, code clones should be collected together into one occurrence, and this is realized by clone analysis. Some code clones occur when a developer duplicates a code fragment to several places in a program by using copy-and-paste editing operation. So we can instruct a refactoring to eliminate the occurrences of code clones by monitoring developer’s duplication operations such as carrying out copy-and-paste command. Figure 1 shows an example of instructing a refactoring which extracts a super class and forms Template Method pattern [14], in order to eliminate code clones of class C. A developer first duplicates a class C, and as a result, the program has the code clone C_dup of C. After that, he/she modifies an internal method M in C_dup.

Since the program has an occurrence of a code clone of C, the refactoring to eliminate it can be recommended. More concretely, the refactoring operation called Form Template Method can be used to do it. We extract a super class C_super and re-organize a template method M and hook method Mpo, in order to represent the difference of the
method $M$ in $C_{dup}$ from $C$. Thus we can have a kind of rule to instruct a refactoring as follows:

If the program modification includes “duplicate $C$ to $C_{dup}$” and “add $S$ in $M$ of $C_{dup}$”, then the possibility of adding a template method with a super class $C$ containing a template method $M$ should be considered.

If the developer modifies the fragments of code in class $C$ during the period of time from the execution of the duplication to the addition, the refactoring that forms template method may not fit the program. So we validate the conditions of class $C$ and $C_{dup}$ after the checking of modifications because it is difficult to detect this case by using the modification features only.

Although the above rule is informally described, the way how to formalize it will be mentioned in the next section. The “if” part is used for matching to the histories of actual modifications, and thus it is formally defined with a regular expression with variables. The validation of code fragments is formally defined as the satisfaction of a constraint.

2.2 Using the Changes of Complexity Measures

Some of software metrics such as WMC of C&K metrics [15] expresses the complexity of a component of the program, and it can be applied to decide when and where refactoring should be carried out. For example, suppose that a developer finds that a class $A$ in the program is too large while a class $B$ is too small when he/she calculates the measures denoting the size of a code fragment. He/she can carry out the refactorings for decomposing the class $A$ into several smaller components and for inlining the class $B$ or the methods included in $B$. As shown above, by using the techniques of software metrics, especially complexity metrics, we can decide some refactorings to be carried out according to the value of metrics.

However, since the values of such measures represent a current snapshot of the program, they cannot reflect developer’s intention on whether he/she will increase or decrease the complexity of the component. Obviously, just after starting coding a component, its complexity value is low but this fact does not necessary follow that inline refactoring should be carried out to it. That is to say, we may not instruct suitable refactoring based on a developer’s intention, by using just a value of software metrics. It means that we should focus not on the value of software but on the changes of the metrics to extract the developer’s intention. If he/she intends to build up a component in a program, he/she is adding some statements to the component and its complexity value is increasing. In this situation, when the complexity value comes near the threshold, i.e. the component comes to be too large, we can recommend the refactoring for decomposing it. On the other hand, if he/she performs consecutive deletions of statements in a component and its complexity value becomes too low, we can instruct inline refactoring operations. We can informally summarize this rule as follows:

If the program modification includes consecutive additions to (deletions in) a component, the decomposition (inline) refactoring can be applied.

3. Modeling Process for Instructing Refactoring

3.1 Overview

Our system is used in interactive program development environment, as mentioned above, in order to collect program modifications done by a developer in real-time. That is to say, it always monitors editing commands issued by the developer and extracts program modifications. In our technique, these modifications are the target of pattern matching. To realize the pattern matching formally, we represent the modifications with a string following a certain syntactic rule, i.e. a language, and we call this language Edit Script. The approach that associates the editing commands with a language is often carried out [16], [17]. By using this approach, we also represent modification patterns as a similar language. The user can also extend the system by creating new custom patterns.

Figure 2 shows an overview of our system. The extracted modifications are sent to the kernel of our system which is depicted on the left side of the figure. Our system searches suitable refactorings whenever a modification is received. Whether a refactoring should be instructed or not is checked by using the modification pattern sequences and the constraints which are stored in Refactoring DB. If the history of actual program modifications includes the part that matches to a modification pattern sequence and satisfies a constraint, some of recommended refactoring operations, which are associated with the modification pattern sequence and the constraint in Refactoring DB in advance, are selected out and instructed. A modification pattern sequence is described with a regular expression with variables as mentioned in the previous section, and matching succeeds when an instance of the pattern produced by substituting appropriate values to its variables corresponds to a part of

![Fig. 2 An overview of the proposal program developing model.](image-url)
modification histories. After a successful pattern match, our system checks the satisfaction of the associated constraint. A constraint is a kind of static code analyses. By using the information from pattern matching, the analyzing context of a constraint is decided.

Whenever new program modification is done, our system evaluates modification pattern sequences and checks constraints. If the patterns match the modification history and constraints are satisfied, our system instructs the refactoring operation in a while. In this sense, our system can control the timing of instructions using modification patterns and constraints stored in Refactoring DB. Additionally, we restrict that the most recent modification (the last element of the modification history) must be significant to this pattern matching. If the patterns match the modification history and constraints are satisfied, our system instructs the refactoring operation in a while. In this sense, our system can control the timing of instructions using modification patterns.

Figure 3 shows an example of a composed modification. In this case, \( \mu_2, \mu_3 \) are child modifications of a composed modification \( \mu_1 \) in terms of an input program \( p_0 \). Composed modifications can be considered in the form of tree structure consisting of descendant modifications.

For example, consider the above duplication modification \( \mu \), this modification is associated with the duplication of a class which is identified by ‘#1’, e.g. A. In this case, this modification has the following child modifications: addition of A, duplication of all the code fragments which are included in A, e.g. methods, fields, and so on. Then, the relation of these modifications are represented following using an composition of the modifications:

\[
\text{duplication of } A \rightarrow \text{addition of } A \circ \text{duplication of all the code fragments of } A.
\]

To represent the semantic concept of a modification that is an operator which modifies a program, we define the following transition function.

**Definition 4** (Transition Function). Let \( \delta : P \times M_b \rightarrow P \) be a Basic Transition Function. Transition function \( \delta : P \times M \rightarrow P \) is defined as follows:

\[
\delta(p, m) = \begin{cases} 
    \mu_1 \circ \mu_2 \circ \mu_3, & \text{if } \mu_1 \circ \mu_2 \circ \mu_3 \text{ is a valid transition function}, \\
    p, & \text{otherwise}.
\end{cases}
\]
Next, we define a modification history. In this paper, a modification history is represented as the sequence of modifications.

**Definition 5 (Modification Sequence).** We call sequenced modifications \( \text{Modification Sequence} \). We use the symbol ‘\( \cdot \)’ as the separator of modifications: \( \mu_1; \mu_2; \ldots; \mu_n \). \( \vec{\mu} \) denotes a modification sequence, and \( |\vec{\mu}| \) denotes the length of the sequence \( \vec{\mu} \).

A concatenation of modification sequences (e.g. \( \vec{\mu}; \vec{\mu}' \)), and a concatenation of a modification sequence and a modification (e.g. \( \vec{\mu}; \mu; \mu; \vec{\mu} \)) are also represented as a flat modification sequence.

We represent a software development process as a pair \( (p, \vec{\mu}) \), where \( p \) is the current program and \( \vec{\mu} \) is a history of program modifications. The program \( p \) consists of a sequence of history elements \( p_0, p_1, \ldots, p_n \), where \( p_n \) denotes the latest state of the program. Thus any history elements can be represented as follows:

\[
\delta(p, \mu) \overset{\text{def}}{=} \begin{cases} 
\delta(\delta(p, \mu_1), \mu_2) & \text{if } \mu \overset{\mu_1}{\rightarrow} \mu_1 \circ \mu_2, \\
\delta(p, \mu) & \text{otherwise.}
\end{cases}
\]

For example, consider the following modification and modification pattern:

\[
\mu = \text{duplicate}(#)\cdot Package/\#2:\text{Class}, \#1:\text{Package}/\#3:\text{Class}, \\
\pi = \text{duplicate}(#)\cdot Package/\#6:\text{Class}, \#1:\text{Package}/\#3:\text{Class}.
\]

In this case, ‘\( \#s \)’ and ‘\( \#d \)’ are variables. To use the following substitution table, we can unify the modification and the pattern:

\[
\sigma = \{(\#s, \#2), (\#d, \#3)\}.
\]

In order to define a pattern matching, we prepare to define the inclusion relationship between modifications.

**Definition 9 (Inclusion between Modifications).** \( \mu' \subseteq \mu \) denotes that the modification \( \mu' \) is included in the modification \( \mu \), if and only if the following condition is satisfied:

1. \( \mu' = \mu \), or
2. \( \mu \rightarrow \mu_1 \circ \mu_2 \land (\mu' \subseteq \mu_1 \lor \mu' \subseteq \mu_2) \).

The inclusion between modifications holds if and only if the included modification is a sub tree of the target modification. In the case of Fig. 3, all modifications from \( \mu_1 \) to \( \mu_7 \) are a sub tree of the target modification, so \( \mu_i \subseteq \mu_1 \) (1 \( \leq \) i \( \leq \) 7) holds.

This inclusion relationship between modifications can be extended to modification sequences in the similar way as follows.

**Definition 10 (Inclusion between a Modification and a Modification Sequence).** \( \vec{\mu} \subseteq \mu \) denotes that the modification sequence \( \vec{\mu} \) is included in the modification \( \mu \), if and only if the following condition is satisfied:

1. \( \vec{\mu} = \emptyset \) (\( |\vec{\mu}| = 0 \)), or
2. \( \vec{\mu} = \mu_1 \) (\( |\vec{\mu}| = 1 \)) \( \land \mu_1 \subseteq \mu \), or
3. \( \exists \text{ division } \vec{\mu} = \vec{\mu}_1; \vec{\mu}_2 \) (\( |\vec{\mu}| \geq 0 \)) \( \land \mu_1 \rightarrow \mu_1 \circ \mu_2 \land \vec{\mu}_1 \subseteq \mu_1 \).

**Definition 11 (Inclusion between Modification Sequences).** \( \vec{\mu}' \subseteq \vec{\mu} \) denotes that the modification sequence \( \vec{\mu}' \) is included in the modification sequence \( \vec{\mu} \), if and only if the following condition is satisfied:

1. \( \exists \text{ modification } \mu \text{ in } \vec{\mu} \text{ s.t. } \mu' \subseteq \mu \), or
2. \( \exists \text{ division } \vec{\mu}' = \vec{\mu}_1; \vec{\mu}_2; \vec{\mu}_3 \) (\( |\vec{\mu}| > 0 \), \( |\vec{\mu}_1| > 0 \), \( |\vec{\mu}_2| \geq 0 \)) \( \land \vec{\mu}_1 \subseteq \vec{\mu}_1 \).

By using this definition of inclusion, we define the pattern matching between modifications and modification patterns.

**Definition 12 (Pattern Matching).** \( \vec{\sigma} \preceq \vec{\mu} \) denotes that the modification pattern sequence \( \vec{\sigma} = \sigma_1; \ldots; \sigma_m \) matches to the modification sequence \( \vec{\mu} \) by the substitution table \( \sigma (\in S) \), and is defined as follows:

\[
\vec{\sigma} \preceq \vec{\mu} \overset{\text{def}}{=} \text{sub}(\sigma_1, \sigma); \cdots; \text{sub}(\sigma_m, \sigma) \subseteq \vec{\mu}.
\]
Figure 4 shows an overview of a pattern matching. Pattern matching is satisfied when the modification pattern sequence, which is in the bottom part of the figure, is transformed into a modification sequence by substitution $\sigma$, and the sequence is included in the other modification sequences, which is in the top part of the figure.

Given two modification pattern sequences, we can calculate the substitution which one matches to the other, if it exists. For example, the algorithm which finds pattern matching and corresponding substitution is shown in Fig. 5. This algorithm finds mapping between modifications and patterns by using depth-first-searching and backtracking.

After patterns match modification histories, the current program is checked. This checking is modeled as a satisfaction of Constraint.

**Definition 13 (Constraint).** A function $C : P \times S \rightarrow \{0, 1\}$ is called Constraint.

When the current program and the substitution table as context information are inputted to a constraint, it returns the satisfaction of it. Whenever a modification sequence is executed, our system finds suitable refactorings from the knowledge base, i.e. Refactoring DB. An element of DB is a tuple $(\bar{\pi}, C, \pi)$:

- $\bar{\pi} = \pi_1; \cdots; \pi_m$ is a modification pattern sequence,
- $C$ is a constraint,
- $\pi$ is a modification pattern which denotes a refactoring operation.

Now, let $(p, \bar{\mu})$ be a process. Our system instructs a developer a refactoring sub($\pi$, $\sigma$) if the following conditions are satisfied:

1. $\bar{\pi} \subseteq \bar{\mu}$,
2. sub($\pi_m$, $\sigma$) $\subseteq \mu_n$,
3. $C(p, \sigma) = 1$.

Condition 1 represents a pattern matching, which is described in step 1 of Fig. 2. As mentioned above, the last element of the modification history must be significant to pattern matching process. Then we check it in condition 2. If the last item in the modification pattern sequence matches to the latest modification, then we consider the assumption is satisfied. Actually, condition 1 and 2 are checked at once for an effective pruning of searching space. Condition 3 represents the satisfaction of the constraint, which is described in step 2 of Fig. 2.

**4. Supporting Tool**

4.1 Overview

Figure 6 shows a snapshot of the screen in the proposed system. We have implemented the above system as a plug-in for Eclipse integrated development environment (IDE) [18]. Using Eclipse IDE, we obtain the following benefits:

- **Wide availability.** Most of Java developers use Eclipse IDE, so they can use our system easily.
- **Quick parsing.** Eclipse IDE parses Java source code automatically and immediately after the developer has edited it.
Refactoring browser is provided. After instructing the candidates of refactoring, the user can select a refactoring out of them, and actually refactor the program by using the browser.

Our plug-in can extract modifications from a difference on Abstract Syntax Trees (ASTs). After a program modification is extracted, it is added to the history, and then pattern matching process is executed. If suitable refactoring is found, it is presented in “Suggested Refactoring” view.

We illustrate the procedure how to refactor a program step by step.

### 4.2 Extracting Program Modifications

In order to detect program modifications from a Java source code, we extract the difference between two ASTs. Our tool picks up the latest version of AST from Eclipse IDE whenever the IDE invokes an event that modified the AST. Then, our tool gets the difference of ASTs by comparing the latest version of the AST with the previous. In general, it takes a great deal of cost to calculate a difference of tree structures [19], [20].

Our tool records all of the positional information about the added/removed characters via Eclipse Java editor. It is clear that all of added/removed nodes are related to the recorded characters. Then, we can omit to search nodes which are not related to the recorded characters. As a result, the searching space is reduced drastically. Additionally, our tool traps each copy-and-paste event generated by a user, and records it because the modification caused by this event leads to the duplication of code.

We translate the extracted program modifications into the form of Def. 2 of Sect. 3.2, i.e. Edit Scripts. At present, we provide four common modifications $M$ shown in Table 1. By using a combination of these modifications, our tool can almost completely represent complex edit behavior. However, our tool also allows users to define new modifications, e.g. automatic code generation, or furthermore some refactoring operations. In Table 1, most commands have Path arguments, which specify the node of the AST. Path is defined as a sequence of steps connected with "/", and step is a pair of the unique ID and the type name of the node. IDs are consistently assigned according to the identification of the AST nodes. We show some examples of Path as follows:

- #1:Package/#2:Class/#3:Method
- #1:Package/#4:Class/#5:Method/#6:Block/#8:If

Some examples of modifications are shown in Sect. 4.4.

### 4.3 Refactoring DB

Modification patterns stored in Refactoring DB include the variables whose types are shown in Table 2, and several variables have some limitations where values should be substituted following given rules. That is, a set of available substitutions $S$ is implicitly defined.

In this subsection, we mention a simple case study where we modify a sample Java program and observe which refactorings are really instructed. Table 3 shows three of refactorings stored in Refactoring DB which we use in this study.

All of modification patterns are described as a pattern of Edit Scripts, and all of constraints are defined as methods of Java programs. To describe a new pattern and constraint method, the user can add original possibilities of refactorings.

### 4.4 Case Study

Figure 7 shows an initial input program at the top, and a sequence of the modified programs numbered with 1, 2 and 3 in order. Modified statements are underlined in Fig. 7. Executed modifications and instructed refactorings are as follows:

1. Replacement from integer 2 to the method invocation f.f. As a result, Introduce Explain Variables refactoring is instructed. Replacement was represented as a pair of a deletion and an addition.
2. Duplication of class C to C2. First, by using a copy-and-paste, the code fragment of class C is duplicated. Secondly, the name of the duplicated class is modified to C2. As a result, Extract Super Class refactoring is instructed. Additionally, Introduce Explanation Variables refactoring is also instructed because of an add modification which is contained in duplicate of class C.
3. Addition of if statements. As a result, Form Template Method refactoring is instructed.

Figure 8 shows the result of pattern matching when
Table 3  Suggested refactorings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Extract Super Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>π</td>
<td>duplicate(<strong>/$src:Class,</strong>/$dst:Class)**</td>
</tr>
<tr>
<td></td>
<td>It represents a class duplication from $src$ to $dst$.</td>
</tr>
<tr>
<td>C</td>
<td>Anytime 1.</td>
</tr>
<tr>
<td>π</td>
<td>extract_superclass($src,$dst):</td>
</tr>
<tr>
<td></td>
<td>1. Duplicate $src$ to Superclass.</td>
</tr>
<tr>
<td></td>
<td>2. Make $src$ and $dst$ inherit Superclass.</td>
</tr>
<tr>
<td></td>
<td>3. Delete implementations of $src$ and $dst$.</td>
</tr>
</tbody>
</table>

Name  Introduce Explain Variables

| π | duplicate(**/$parent:Call/$child:Call/**)** |
|   | It represents a method invocation $parent$ which has other method invocation $child$ as an argument. |
| C | desc($child$) ≥ 3 ∧ desc($parent$)/desc($child$) > 1.3, where desc() is one of complexity metrics that calculates the number of descendant nodes. |
| π | introduce_explain_variable($child$): |
|   | 1. Add new local variable $L$ to the block which has $parent$. |
|   | 2. Move method invocation $child$ as the assignment of $L$. |
|   | 3. Add variable reference of $L$ to where $child$ was. |

Name  Form Template Method

| π | duplicate(**/$src:Class,**/$dst:Class**); |
|   | add(**/$dst:Class/$m:Method/*:Block/$s:**)** |
|   | It represents an addition of some statement to the method in $dst$, following the class duplication from $src$ to $dst$. |
| C | The contents of related classes are the same except for the addition (add) modification. |
| π | form_template_method($src,$dst,$m,$s,$n): |
|   | 1. Extract super class as Superclass from $src$ and $dst$. |
|   | 2. Adding blank method to Superclass as Mpo. |
|   | 3. Substitute the statement $s$ in method $m$ in Superclass to method invocation of Mpo. |
|   | 4. Implement $s$ in $m$ in $dst$ as Mpo. |

Form Template Method refactoring is instructed. Left hand side of the figure describes a modification pattern sequence of Form Template Method refactoring. On the other hand, right hand side of the figure describes a sequence of modifications which are actually executed in the case study, i.e. modification history. Now, pattern matching was satisfied when modification prefixed with “(3)” was executed. Underlines with the numbers in Fig.8 represent a mapping of pattern matching. The variable in the pattern sequence which is underlined with the number $n$ leads the ID, steps, or type name of nodes which is underlined with the same number $n$ in modification history.

By applying suggested refactorings, we get the source code shown in Fig.7-Refactored. By looking at the result of refactorings, we can see that most of instructed refactorings are effective. Additionally, the execution time of the instructing phase is actually small. So, we believe that the reduction of searching space using pattern matching information is meaningful. However, this case study deals with a tiny program. We should experiment with more source code or a large scale. Furthermore, the limit of our technique.

Initial:

```java
class C {
  int foo(int t) {  int a = f(1, 2, 3);
    int b = f(4, 5, f(6, 7, t));
    return a - b;
  }
}
```  

(1):

```java
class C {
  int foo(int t) {
    int a = f(1, F.f(6, 7, t), 3);
    int b = f(4, 5, F.f(6, 7, t));
    return a - b;
  }
}
```  

(2):

```java
class C {
  int foo(int t) {
    int a = F.f(1, F.f(6, 7, t), 3);
    int b = F.f(4, 5, F.f(6, 7, t));
    return a - b;
  }
}
```  

(3):

```java
class C {
  int foo(int t) {
    int a = F.f(1, F.f(6, 7, t), 3);
    int b = F.f(4, 5, F.f(6, 7, t));
    if(a > 8) a = F.f(1, 2, a);
    return a - b;
  }
}
```  

Refactored:

```java
class C_super {
  int foo(int t) {
    int a = F.f(6, 7, t);
    int b = F.f(4, 5, F.f(6, 7, t));
    a = mpo(a);
    return a - b;
  }
}
```  

```java
class C {
  int mpo(int a) {
    return a;
  }
}
```  

```java
class C extends C_super {
  int mpo(int a) {
    if(a > 8) a = F.f(1, 2, a);
    return a;
  }
}
```  

Fig. 7 An overview of the case study.
should also be examined.

5. Discussion

5.1 Use of Static Analysis Techniques

We consider that our tool will be used more effectively to collaborate with other useful techniques. Our tool can adapt to some traditional static analysis techniques. As mentioned above, these techniques have an ability of analyzing possibilities of refactorings in detail. Thus, by using these techniques, we can make our technique more accurate.

In our technique, whether a refactoring operation should be instructed or not is determined by the matching of modification patterns and the satisfaction of constraints. And constraints are designed as Java programs in our implementation. Thus, we can combine our technique and the static analysis techniques by embedding the implementation of them as constraints.

5.2 Detection of Modifications

In our implementation, the detection of some modifications are considered as a problem. We discuss how to detect these modifications.

duplication Now, duplication is involved with the trapping of the copying command and the pasting command in a text editor, which can only be detected when the two commands are executed immediately one after another. Thus the implementation cannot detect the duplication of code if the copied code fragments are modified before the pasting. In this way, variation of editing command sequence may cause a serious problem of scalability, in particular, under the situation of the development by a team.

undo/redo We specially consider the commands of undo and redo operations. When the developer executes the redo operation, he/she has the intention to restore the program to the state before the undo operation was executed. To avoid an insignificant instruction of refactorings, our tool should be silent when the undo/redo operations are executed. When the undo operation is executed, related modifications should be removed from the modification histories.

5.3 Development by a Team

Our technique can apply to development by a team by using version management systems. By using some version management systems such as CVS, each developers virtually have an independent environment, the Sandbox. In this environment, our tool works well.

On the other hand, in order to deal with the modifications which are executed by each developers as a whole, we have to merge all of the modification histories executed by each developers. To solve this problem, we must consider the variation of editing command sequence.

6. Conclusion and Future Work

In this paper, we concentrate on program modifications by a developer. We propose the system which instructs with suitable refactorings, applying to a sequence of program modifications. Finally we show the system's feasibility by implementing an automated supporting tool as a plug-in of Eclipse IDE.

We consider two types of the characteristic modifications: duplication of codes and change of complexity measures. By using them, our system instructs Form Template Method refactoring or Introduce Explain Variables refactoring. Using the same approaches, our system can instruct following refactorings: Inline Class, Inline Method, Move Method, Move Field, and Extract Method.

We should consider following problems as future work:

- To discover modifications and modification patterns which illustrate the feasibility of historical analysis, and clarify the guide to describe modification patterns.
- To discover modifications and modification patterns. We consider a new technique, which represents modification histories not as Edit Scripts but as versioned tree structures.
- To collaborate existing refactoring browsers [3], [18] in order to check pre-conditions of refactorings more precisely.

Acknowledgments

The authors would like to thank Mr. Takao Yamazaki and
Mr. Rodion Moiseev for their valuable comments to the earlier version of this paper.

References


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


Shinpei Hayashi received a BS in information engineering from Hokkaido University in 2004. He is currently a master course student in computer science of Tokyo Institute of Technology. His research interests include software reuse, software development environment, software architecture and artificial intelligence.

Motoshi Saeki received a BS in electrical & electronic engineering, and a MS and a Ph.D. in computer science from Tokyo Institute of Technology, in 1978, 1980 and 1983 respectively. He is currently a professor of computer science at Tokyo Institute of Technology. His research interests include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).

Masahito Kurihara received a BS in electrical engineering, and a MS and a Ph.D. in information engineering from Hokkaido University, in 1978, 1980 and 1986 respectively. He is currently a professor of computer science at Hokkaido University. His research interests include mathematical logic and automated reasoning in computer science and artificial intelligence.