Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs

Takashi Ishio, Hironori Date, Tatsuya Miyake, Katsuro Inoue
Osaka University
1-3 Machikaneyama, Toyonaka, Osaka, 560-8531, Japan
{ishio, h-date, t-miyake, inoue}@ist.osaka-u.ac.jp

Abstract

A coding pattern is a frequent sequence of method calls and control statements to implement a particular behavior. Coding patterns include copy-and-pasted code, crosscutting concerns and implementation idioms. Duplicated code fragments and crosscutting concerns that spread across modules are problematic in software maintenance. In this paper, we propose a sequential pattern mining approach to capture coding patterns in Java programs. We have defined a set of rules to translate Java source code into a sequence database for pattern mining, and applied PrefixSpan algorithm to the sequence database. As a case study, we have applied our tool to six open-source programs and manually investigated the resultant patterns. We report coding patterns that are candidates of aspects and several logging patterns that are well-known crosscutting concerns but hard to modularize.

1. Introduction

To develop large scale software, developers use idiomatic coding patterns to implement a particular kind of concerns that are not modularized in the software [19]. Developers obtain coding patterns from the source code of their software, the coding standard of their team and other available resources. Such idiomatic code fragments that spread across modules are problematic in software maintenance. When developers modified an instance of an idiomatic code fragment, developers should inspect and modify all other instances of the idiom to keep the code fragments consistent [3, 4, 7, 10].

While Aspect-Oriented Programming (AOP) [14] and some object-oriented design patterns such as Template Method [7, 9] are effective to refactor such an idiom to a modular unit, many idiomatic code fragments are still involved in software. This is because developers are not interested in modularizing well-known implementation idioms, e.g. a loop using an Iterator, and some duplicated code fragments that are tangled with other functions.

To enable developers to understand and manage idiomatic code fragments, we have applied PrefixSpan, or a sequential pattern mining algorithm [22], to extract coding patterns for implementing a particular kind of concerns. We have defined a set of rules to translate source code into a sequence database for PrefixSpan, and implemented our approach as a tool named Fung. Our sequential pattern mining extracts frequent subsequences of method calls and control statements in a program. Our sequential pattern mining is similar to code clone-based aspect mining approach [6]. While code clone detection techniques extract a consecutive sequence of statements or a connected subgraph of a dependence graph [13, 17], a sequential pattern instance may involve disconnected method calls.

We have applied our tool Fung to six Java programs: JHotDraw, jEdit, Azureus, Apache Tomcat, ANTLR and SableCC. We found several common coding patterns in programs. Some patterns can be refactored using AspectJ, but some other patterns are hard to modularize as aspects because of their heterogeneous implementation. Our pattern mining supports may help developers to perform refactoring or document the coding patterns for future software maintenance tasks.

The structure of the paper is following. In Section 2, we describe about coding patterns and `PrefixSpan` algorithm. Section 3 describes our sequential pattern mining approach for a Java program. Section 4 shows the result of case study on six Java programs. Section 5 discusses the characteristics of the coding patterns that our approach extracted. In Section 6, we describe related work. Section 7 summarizes our current state and the future directions.

2. Background

In this paper, we propose a pattern mining approach to find *coding patterns* including crosscutting concerns. A *coding pattern* denotes a frequent sequence of method calls and control elements to implement a concern but not modularized in a program. For example, a pair of `hasNext` and

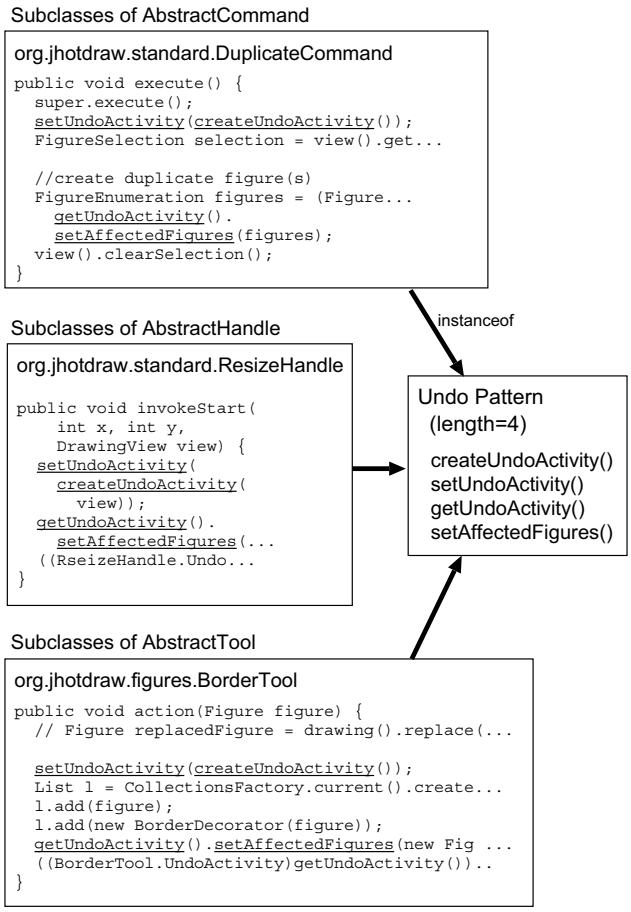

Figure 1. Undo pattern in JHotDraw 5.4b1

`next` method of `Iterator` interface always implements a loop with a `for` statement or a `while` statement. Another example pattern is an Undo implementation in JHotDraw 5.4b1 as shown in Figure 1. The Iterator pattern is a well-known implementation idiom in Java but the undo pattern is a crosscutting concern in JHotDraw 5.4b1. Documenting the undo pattern can help a developer to understand how the undo functionality is implemented.

To write a code fragment based on an existing pattern, developers often copy-and-paste a code fragment [15]. Such duplicated code fragments known as code clones [3, 4] are regarded as aspect candidates [6]. However, most of code clone detection tools cannot detect code fragments modified after copy-and-pasted. For example, CCFinder, an efficient code clone detection tool, detects consecutive sequences of tokens [13]. Therefore, if a new statement is inserted to a copy-and-pasted code fragment, the modified code fragment is no longer a code clone of the original one. Our pattern mining covers such code fragments.

2.1. Aspect Mining

Aspect mining is a research area to identify crosscutting concerns that are not modularized in the source code. Aspect mining techniques employ some heuristic functions to detect typical implementation of crosscutting concerns in object-oriented programs. Bruntink tried to cover crosscutting concern code by code clones [6]. Marin proposed fan-in analysis to extract methods that are frequently used in a program such as logging [21]. Breu's history-based aspect mining focuses on extracting co-located methods in a program from its software repository [5]. Krinke proposed a control-flow graph mining approach to detect methods that should be called at the beginning/end of some method or before/after the method call [18].

Our sequential pattern mining approach employ PrefixSpan algorithm proposed in [22]. Different points from the previous aspect mining approaches are listed below.

- A coding pattern involves control elements such as IF and LOOP. This enables developers to understand method call patterns with its associated control-flow.
- A coding pattern is an *ordered* list of elements. We can detect code fragments that are tangled with other code as a pattern instance until the sequential order of method calls has been modified.
- We defined normalization rules to handle variants of a pattern. For example, rewriting a `for` statement with a `while` statement does not affect our analysis.

2.2. Sequential Pattern Mining

Sequential pattern mining extracts frequent subsequences from a sequence database [2]. A sequence in a sequence database is an ordered list of elements. PrefixSpan takes as input a sequence database S and the minimum support threshold min_sup, and extracts a set of sequential patterns [22]. The algorithm first finds length-1 sequential patterns that are frequent elements from the database S. For example, length-1 patterns in an example database $S = \{\langle abcd\rangle, \langle aeade\rangle, \langle dafb\rangle$ and $\langle acd\rangle\}$ are following:

$\langle a\rangle : 4, \langle b\rangle : 2, \langle c\rangle : 2, \langle d\rangle : 3$

A pattern in the form of $\langle pattern\rangle : support$ represents the pattern and its associated support count. In this example, we used $min\_sup = 2$ that filters out $\langle e\rangle$ and $\langle f\rangle$ because each element is involved in only one sequence.

Then, PrefixSpan constructs length-$(k+1)$ patterns from length-k patterns. The algorithm collects the sequences containing a pattern, and extracts a *projected database*; each sequence in the projected database is prefixed with the first occurrence of the pattern. For example, $\langle a\rangle$-projected database contains four sequences: $\langle bcd\rangle$, $\langle eade\rangle$, $\langle fb\rangle$ and

Table 1. PrefixSpan algorithm repeatedly creates projected databases to extract patterns. The first row indicates the original sequence database.

$prefix$	$prefix$-Projected Database	Patterns
	$\langle abcd\rangle, \langle aeade\rangle,$ $\langle dafb\rangle, \langle acd\rangle$	$\langle a\rangle : 4, \langle b\rangle : 2,$ $\langle c\rangle : 2, \langle d\rangle : 3$
$\langle a\rangle$	$\langle bcd\rangle, \langle eade\rangle, \langle fb\rangle, \langle cd\rangle$	$\langle ab\rangle : 2, \langle ac\rangle : 2,$ $\langle ad\rangle : 3$
$\langle b\rangle$	$\langle cd\rangle$	
$\langle c\rangle$	$\langle d\rangle, \langle d\rangle$	$\langle cd\rangle : 2$
$\langle d\rangle$	$\langle e\rangle, \langle afb\rangle$	
$\langle ab\rangle$	$\langle cd\rangle$	
$\langle ac\rangle$	$\langle d\rangle, \langle d\rangle$	$\langle acd\rangle : 2$
$\langle ad\rangle$	ϕ	
$\langle cd\rangle$	ϕ	
$\langle acd\rangle$	ϕ	

$\langle cd\rangle$. Frequent elements in a projected database represent length-$(k+1)$ patterns. The elements b, c and d in $\langle a\rangle$-projected database results in their corresponding length-2 patterns: $\langle ab\rangle : 2$, $\langle ac\rangle : 2$ and $\langle ad\rangle : 3$. Similarly, all projected databases are examined as shown in Table 1. Each row shows a length-k pattern ($prefix$), its projected database and new length-$(k+1)$ patterns. ϕ indicates an empty projected database.

The algorithm terminates when no new sequential pattern is found in a pass. PrefixSpan with $min\_sup = 2$ finds five sequential patterns from the example database: $\langle ab\rangle : 2, \langle ac\rangle : 2, \langle ad\rangle : 3, \langle cd\rangle : 2$ and $\langle acd\rangle : 2$.

3. Coding Pattern Mining for Java

We propose an application of PrefixSpan to detect coding patterns in Java programs. Our approach comprises three steps: normalization of source code, pattern mining and classification of the resultant patterns.

The normalization step translates each Java method in a program to a sequence that comprises method call elements and control elements. Figure 2 is an example of a sequence extracted from a source code fragment. Our normalization rules, partly shown in Figure 3, generate a sequence of the following elements:

Method call element. A method call in a Java method is translated into a method call element. A call element is a method signature without its class name. We ignore a class name to handle dynamic binding. We also ignore variables containing receiver objects since different variable names are used for each context. For example, a method call it.hasNext() is translated into a method call element hasNext(): boolean. In Figure 2, return types are omitted to save space.

If two or more methods are called in an expression, the corresponding method call elements are sequentially ordered by its evaluation order according to the Java specification. The tied (or undefined) methods are sorted by their textual order (left to right) in the source code.

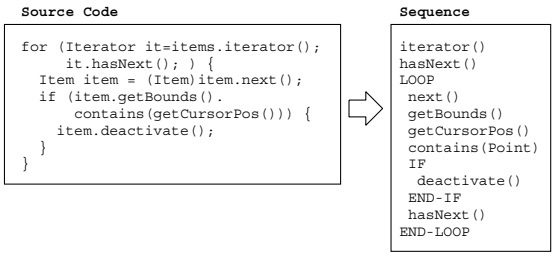

Figure 2. A sequence extracted from source code

```
Statement: if (<cond>) <then> else <else>;
Sequence:  <cond>, IF, <then>, ELSE,
           <else>, END-IF

Statement: for (<init>; <cond>; <inc>) <body>;
Sequence:  <init>, <cond>, LOOP, <body>,
           <inc>, <cond>, END-LOOP

Statement: while (<cond>) <body>;
Sequence:  <cond>, LOOP, <body>, <cond>, END-LOOP
```

Figure 3. Normalization of control statements.

IF/ELSE/END-IF element. An `if` statement is translated into a series of IF, ELSE and END-IF elements. The top pair of a statement and its corresponding sequence in Figure 3 shows the normalization rule. If the predicate of the statement calls a method, the corresponding method call element is inserted before the IF element since the predicate is evaluated before the `if` statement selects control-flow. Elements corresponding to statements controlled by the `if` statement are placed between an IF element and its corresponding END-IF element.

LOOP/END-LOOP element. A `for` or `while` statement is translated into a pair of a LOOP element and an END-LOOP element. A method call in the predicate of the loop is translated into a pair of method call ele-

ments inserted before the LOOP element and the END-LOOP element according to control-flow of the loop statement. Figure 3 shows the rules for loop statements. If a developer rewrites a `for` loop to a `while` loop, both loops are translated into this same sequence.

In the current implementation, we ignore `break`, `continue` and `return` statements in a loop since we focus on the syntactic structure of a loop instead of precise control-flow information.

We apply the above rules to generate sequences for each Java method. In the pattern mining step, PrefixSpan takes as input the normalized sequences, a threshold support count min_sup, and a filtering parameter min_len that excludes patterns whose length is shorter than min_len. We added the min_len parameter since PrefixSpan extracts too many short patterns to investigate. Patterns extracted by our approach satisfy the following characteristics.

- A pattern is a sequence of method call elements and control elements.
- A pattern comprises at least min_len elements. For example, Figure 1 shows an Undo pattern comprising four method call elements: createUndoActivity, setUndoActivity, getUndoActivity and setAffectedFigures.
- A pattern has at least min_sup instances. We use the term *instance* of a pattern to represent a concrete code fragment corresponding to the pattern. An instance of a pattern is a list of tokens in the source code; each token corresponding to a pattern element. For example, Figure 1 involves three Undo pattern instances indicated by underlines.
- An instance of a pattern may interleave with other code fragments.
- A pattern implies its sub-patterns (shorter patterns) that have at least the same number of instances. For example, a pattern $\langle abcd \rangle$ implies four sub-patterns comprising 3-elements: $\langle abc \rangle$, $\langle abd \rangle$, $\langle acd \rangle$ and $\langle bcd \rangle$. If the number of instances of a sub-pattern is the same as its super pattern, the sub-pattern is filtered out. This property also implies that a method call may be involved in two or more patterns.

Our approach focuses on mining coding patterns related to method calls. Therefore, we are not interested in patterns that comprise only control statements. To filter out such patterns, we use two filtering rules as follows.

- If more than 70% elements of a pattern are control elements, the pattern is filtered out. We have defined the threshold value based on our preliminary experiment; a developer can specify another threshold if necessary.
- We filtered out patterns including a control element but excluding its peer element since a control statement is always transformed to a pair of the beginning and the end of a code block (e.g., a pair of an IF element and an END-IF element),

After filtering, we classify the patterns into groups since our sequential pattern mining extracts a large number of patterns that are similar to one another. We are using a simple rule for grouping: two patterns are included in the same group if the patterns p_1 and p_2 overlap with each other, i.e., an instance of a pattern shares an element with an instance of the other pattern. This rule categorizes a pattern and its sub-patterns into the same group.

Finally, we sort pattern groups by their support count. The support count of a pattern group is the same as the most frequent pattern in the group.

We have implemented the whole process described in this section as a tool named Fung. The tool takes as input a Java program, a pattern mining parameter min_sup and a filtering parameter min_len. Fung's GUI shows a list of patterns with source code and class hierarchies. Selecting a pattern in a pattern list highlights instances in source code. Fung also exports the resultant patterns in an XML format.

Table 2. Target Software

Name	Version	LOC	#Pattern	#Group
JHotDraw	7.0.9	15104	747	37
jEdit	4.3pre10	17024	137	33
Azureus	3.0.2.2	85248	4682	128
Tomcat	6.0.14	33568	1415	85
ANTLR	3.0.1	3616	352	29
SableCC	3.2	6336	62	18

4. Case Study

We have applied our pattern mining tool Fung to six Java programs: JHotDraw, jEdit, Azureus, Apache Tomcat, ANTLR and SableCC. The programs are chosen from three different domains: GUI applications (JHotDraw and jEdit), network systems (Azureus and Apache Tomcat) and parser generators (ANTLR and SableCC). Their version, size and the number of detected patterns are listed in Table 2. We extracted patterns with parameters $min\_len = 4$ and $min\_sup = 10$; a pattern comprises four or more elements and a pattern has at least ten instances in a program. We excluded pattern groups that comprise only method calls to JDK classes since JDK-only patterns such as a loop with Iterator represent a general purpose code fragment.

We have investigated the top five frequent pattern groups for each program. We have manually inspected source code

```
public class CompleteWord extends ... {

  public static void completeWord(View view) {
    JEditTextArea textArea = view.getTextArea();
    Buffer buffer = view.getBuffer();
    int caretLine = textArea.getCaretLine();
    int caret = textArea.getCaretPosition();

    if(!buffer.isEditable()) {
      textArea.getToolkit().beep();
      return;
    }
    :
  }
}

public class TextArea extends JComponent {
  public void backspaceWord(
                    boolean eatWhitespace) {
    if(!buffer.isEditable()) {
      getToolkit().beep();
      return;
    }
    :
  }
}
```

Figure 4. The length-4 pattern ⟨isEditable / IF / beep / END-IF⟩ in jEdit (9S in Table 4) prevents a user from editing a read-only buffer.

of the most frequent pattern and the longest pattern in each of pattern groups. We summarized the result as tables with four columns: *ID*, *Sup*, *Len* and *Elements*. A pattern *ID* comprises a number indicating a pattern group that the pattern belongs to, and letters 'S' and 'L' indicating the type of the pattern. 'S' represents the pattern is the most frequent (Supported) in the group, and 'L' represents the Longest pattern in the group, respectively. For example, a pattern ID "2L" indicates that the pattern is the longest pattern in the 2nd frequent pattern group. *Sup* is the number of methods involving an instance of a pattern. *Len* is the number of elements of a pattern. The column *Elements* shows the elements of a pattern. We describe the patterns found in the target programs in the rest of this section.

Table 3 shows frequent patterns in JHotDraw 7.0.9. JHotDraw 7.0.9 is well modularized, e.g., the undo coding pattern in JHotDraw 5.4b1 (Figure 1) is already refactored. The most frequent pattern 1S is a small null-check pattern as follows:

```
if (getAction() != null) getAction().XXX();
```

A pair of `willChange` and `changed` forms a pattern 8S to fire events before and after figures are manipulated. This pattern may be modularized with an aspect.

Table 4 shows the patterns in jEdit. The most frequent pattern group calls `openNodeScope` and `closeNodeScope` before and after a functionality in various methods in `bsh` package. This seems a typical crosscutting concern that may be refactored as a Template Method pattern or AspectJ advices. jEdit also includes patterns 3S, 3L, 9S and 9L related to `beep` method. The patterns are to prevent a user from editing a read-only buffer as shown in Figure 4. The patterns are involved in only methods that modify a text buffer. A developer may replace the patterns with an `around` advice in AspectJ if the developer could define a pointcut to capture all text edit methods in the system. Patterns 5S and 5L create GUI components based on jEdit properties (return values of `jEdit.getProperty` method). These patterns are difficult to modularize since each instance creates independent components.

Table 5 shows coding patterns in Azureus, or a BitTorrent client. Azureus is a multi-threaded program; therefore, it frequently uses a pair of `enter` and `exit` methods of `AEMonitor` class for synchronization. The patterns 4S, 4L, 8S and 8L are exception handling patterns with `Debug.printStackTrace` method. The pattern 5S is a logging concern spread across the modules. Although a textual search can easily capture logging method calls, the logging concern is difficult to modularize since Azureus records various messages for each logging method call. We found 51 distinct messages in 55 call sites that call `DHTLog.log`, and 148 distinct messages in 200 call sites that call `Logger.log`. To modularize such logging method calls as a logging aspect, developers have to map join points to messages such as "ping ok", "ping failed" and "add store ok".

Table 6 shows the patterns in Apache Tomcat. The logging pattern 1S is the largest pattern group in the case study; the most frequent pattern has 304 instances and there are 442 variant patterns in the group. The patterns 8S and 8L are also logging patterns; 6192 logging instances in total are involved in Apache Tomcat. These logging code are also hard to modularize because there are various messages for each location where Tomcat executes an important action.

The pattern 6SL in Table 6 is to execute a function in the privileged mode if `isPackageProtectionEnabled` method returns true as shown in Figure 5. Although the structure of the pattern is the same as jEdit shown in Figure 4, this pattern requires additional coding for each pattern instance.

The patterns 12S and 12L in Table 6 are idiomatic patterns related to Managed Bean. A pair of `createMBean` and `destroyMBean` methods contains the shorter, frequent pattern. On the other hand, the longer, less frequent pattern is only contained in `createMBean` method. These patterns enable developers to understand how to use Managed Bean methods.

Table 7 shows the patterns in ANTLR. The top four

Table 3. Patterns in JHotDraw 7.0.9

ID	Sup	Len	Elements
1S	29	4	getAction / IF / getAction / END-IF
1L	11	16	getAction / add / ... (the same length-2 sequence is repeated 8 times)
4SL	19	6	entrySet / LOOP / getKey / getValue / setAttribute / END-LOOP
8S	19	4	LOOP / willChange / changed / END-LOOP
8L	11	5	LOOP / willChange / transform / changed / END-LOOP
11SL	15	4	getView / IF / getView / END-IF
12S	14	5	getAction / add / getAction / configureJCheckBoxMenuItem / add
12L	10	7	size / IF / LOOP / configureJCheckBoxMenuItem / add / END-LOOP / END-IF

Table 4. Patterns in jEdit 4.3pre10

ID	Sup	Len	Elements
1S	55	4	openNodeScope / jjtreeOpenNodeScope / closeNodeScope / jjtreeCloseNodeScope
1L	10	13	openNodeScope / jjtreeOpenNodeScope / jj_consume_token / Expression / jj_consume_token / IF / clearNodeScope / ELSE / popNode / END-IF / closeNodeScope / jjtreeCloseNodeScope
3S	34	4	IF / getToolkit / beep / END-IF
3L	10	6	isEditable / IF / getToolkit / beep / END-IF / remove
5S	28	4	getProperty / add / getProperty / add
5L	10	12	getProperty / addComponent / ... (the same length-2 sequence is repeated 6 times)
8S	25	8	jj_ntk / jj_consume_token / clearNodeScope / ELSE / popNode / END-IF / closeNodeScope / jjtreeCloseNodeScope
8L	11	10	jj_consume_token / Expression / jj_consume_token / IF / clearNodeScope / ELSE / popNode / END-IF / closeNodeScope / jjtreeCloseNodeScope
9S	25	4	isEditable / IF / beep / END-IF
9L	10	5	isEditable / IF / beep / END-IF / setCaretPosition

patterns 1S, 1L, 2S and 2L are involved in test methods working with JUnit. Although JUnit provides `setUp` and `tearDown` methods for modularizing a common procedure for test cases, ANTLR has to create parsers with various configurations for each test case. The other patterns are coding patterns to process the nodes of an abstract syntax tree.

Table 8 shows the patterns extracted from SableCC. All the patterns extracted from SableCC are to process a tree or a list. For example, the patterns 3S and 3L call `apply` method for each element in an array created by `toArray` method.

5. Discussion

We have manually investigated both the frequent pattern and the longest (less frequent) pattern in each of 30 pattern groups. In our experience, the following information helps us to understand a pattern:

A list of methods in a pattern. This is the basic information indicating what the pattern is doing. Although we ignored a class name in a method signature when applying PrefixSpan, a class name is also a good clue to know what a method is doing.

A list of methods involving a pattern instance. Some pattern belongs to a particular set of methods. For example, all instances of the pattern 12L in Apache Tomcat are involved in `createMBean` methods. Similarly, the patterns 6S and 6L in Azureus belong to `refresh` methods in a `tableitems` package. A consistent method name is a good clue to understand the purpose of the pattern.

Comparison between a longer pattern and a shorter pattern in a group. A frequent pattern is shorter, thus simple and easy-to-understand. On the other hand, longer patterns include a non-obvious sequence of method calls. If a longer pattern is just a repeated pattern of the shorter pattern (e.g. the pattern 1L in Table 3), the structure of the shorter pattern is important. If a longer pattern includes method calls

Table 5. Patterns in Azureus 3.0.2.2

ID	Sup	Len	Elements
2S	151	4	enter / iterator / next / exit
2L	10	10	enter / iterator / hasNext / LOOP / next / IF / add / END-IF / END-LOOP / exit
4S	140	4	size / LOOP / printStackTrace / END-LOOP
4L	10	7	size / LOOP / get / printStackTrace / END-LOOP / size / get
5S	119	4	isEnabled / IF / log / END-IF
5L	10	13	log / isEnabled / IF / log / END-IF / isEnabled / IF / log / END-IF / isEnabled / IF / log / END-IF
6S	97	4	getDataSource / setSortValue / isValid / setText
6L	12	5	getDataSource / setSortValue / isValid / getText / setText
8S	85	4	iterator / hasNext / next / printStackTrace
8L	14	7	iterator / hasNext / next / iterator / hasNext / next / printStackTrace

Table 6. Patterns in Apache Tomcat 6.0.14

ID	Sup	Len	Elements
1S	304	4	isDebugEnabled / IF / debug / END-IF
1L	10	24	isDebugEnabled / IF / debug / END-IF / ... (the same length-4 sequence is repeated 6 times.)
6SL	46	4	isPackageProtectionEnabled / IF / doPrivileged / END-IF
7S	44	4	getString / IF / getString / END-IF
7L	10	11	log / IF / getString / println / END-IF / getString / println / getString / println / getString / println
8S	42	4	IF / debug / END-IF / debug
8L	11	6	IF / debug / END-IF / IF / debug / END-IF
11S	38	5	isInfoEnabled / IF / getString / info / END-IF
11L	11	8	getName / getString / isInfoEnabled / IF / getName / getString / info / END-IF
12S	38	7	createManagedName / findManagedBean / getDomain / IF / getDefaultDomain / END-IF / createObjectName
12L	19	13	createManagedName / findManagedBean / getDomain / IF / getDefaultDomain / END-IF / createMBean / createObjectName / isRegistered / IF / unregisterMBean / END-IF / registerMBean

in addition to the elements in a shorter pattern, the longer one is a variant of the shorter one. In this case, the additional method calls may indicate a feature of the methods that include the longer pattern.

Our pattern viewer provides the above information through a list of patterns, a class hierarchy view and a source code view. However, reading all instances of a frequent pattern is a tedious task; Fluid AOP approach might be effective to visualize the common structure of pattern instances as a unified code fragment [11].

5.1. Common Patterns

During the case study, we recognized several common patterns using different methods but implement the same sort of concerns. We recognized common patterns in different programs as follows.

A flag method to execute an additional action in multiple methods. Patterns in this category execute an additional action if a condition is satisfied. This category includes logging patterns in Azureus and Apache Tomcat. This pattern comprises at least four elements: a method call to get a Boolean value indicating the state of a program or an object, a pair of IF/END-IF elements using the Boolean value and a method call to execute an additional action in the IF block. For example, `debugEnabled()` returns a global flag. These patterns are typical crosscutting concerns, while we are hard to modularize them since logging patterns use various messages for each pattern instance.

A flag method to change the behavior of multiple methods. In this category, the current state of a program or an object changes the behavior of methods related to a specific feature. For example, a pattern "Beep if a read-only buffer is to be edited" in jEdit shown in Figure 4 prevents a method from editing a read-only text buffer. A pattern "Executing an action in privileged mode" in Apache Tomcat shown in Figure 5 changes the behavior of Facade classes. A pat-

Table 7. Patterns in ANTLR 3.0.1

ID	Sup	Len	Elements
1S	107	4	setErrorListener / newTool / setCodeGenerator / genRecognizer
1L	10	8	setErrorListener / newTool / setCodeGenerator / genRecognizer / getRecognizer / indexOf / substring / assertEquals
2S	69	4	setErrorListener / newTool / translate / assertEquals
2L	10	5	setErrorListener / newTool / translate / assertEquals / checkError
3S	38	4	LT / match / reportError / recover
3L	10	11	LA / LT / match / LT / match / LT / match / LT / match / reportError / recover
4S	29	8	match / getText / match / reportError
4L	11	8	getText / match / getText / match / getText / match / getText / reportError
5S	27	5	getCharIndex / getLine / getCharPosition / getCharIndex / emit
5L	11	16	getCharIndex / getLine / getCharPosition / match / getCharIndex / mID / getCharIndex / match / getCharIndex / template / IF / IF / getCharIndex / emit / END-IF / END-IF

Table 8. Patterns in SableCC 3.2

ID	Sup	Len	Elements
1S	110	4	pop / get / addAll / add
1L	15	9	pop / get / IF / addAll / END-IF / IF / add / END-IF / add
2SL	82	7	IF / parent / END-IF / parent / IF / parent / END-IF
3S	72	4	toArray / LOOP / apply / END-LOOP
3L	13	7	toArray / LOOP / apply / END-LOOP / LOOP / apply / END-LOOP
4SL	63	11	IF / parent / END-IF / IF / parent / IF / parent / removeChild / END-IF / parent / END-IF
5SL	42	4	apply / IF / apply / END-IF

tern changing the behavior of a single class may be refactored using Strategy pattern or Template Method pattern. If a pattern changes the behavior of various classes, the pattern might be replaced with an `around` advice in AspectJ.

A pair of a set-up step and a clean-up step. A procedure often involves its set-up and clean-up steps at the beginning and the end of the procedure. A pattern in this category comprises a pair of set-up and clean-up method call elements. For example, `Parser` class in jEdit uses a pair of `openNodeScope` and `closeNodeScope` before and after processing a node, Azureus uses a pair of `AEMonitor.enter` and `AEMonitor.exit` to serialize operations. A pair of `before` and `after` advices or Template Method are applicable to modularize this sort of patterns.

Common patterns described above are to implement consistent behavior of a system. Some patterns are *unfactorable* as some code clones are hard to remove [16]. For example, in the case of logging patterns, developers have to consider how a logging aspect generate appropriate messages indicating "what a program is doing" for each join point. If a developer writes advices for each join point, the aspect would be fragile since it strongly depends on the behavior of a base program. Capturing a concept such as "all methods to edit a text buffer" in terms of pointcut designators is also a difficult task [25].

To maintain these unfactorable code fragments, we are planning to generate documentation for patterns from the result of our pattern mining. SoQueT [20] is a promising tool to collaborate with our approach since translating a sequential pattern into *consistent behavior* and *contract enforcement* that are crosscutting concern sorts in SoQueT [20]. We are also interested in FluidAOP [11] and simultaneous modification [10] for maintenance of the patterns.

5.2. Limitations

In the case study, we have extracted frequent patterns that have at least 10 instances since frequent code fragments are likely crosscutting concerns [21]. On the other hand, some copy-and-pasted code fragments may form a long, less-frequent pattern. Therefore, investigating less-frequent (longer) patterns and compare them with code clones are our future work.

Our approach ignores the number of instances in a

```
public String[] getParameterValues(String name) {
  :
  if (SecurityUtil.isPackageProtectionEnabled()){
    ret = (String[])AccessController.
      doPrivileged(
        new GetParameterValuePrivilegedAction(
          name));
    if (ret != null) {
      ret = (String[]) ret.clone();
    }
  } else {
    ret = request.getParameterValues(name);
  }
  return ret;
}

public String getLocalizedMessage(
                    final String message) {
  if (SecurityUtil.isPackageProtectionEnabled()){
    return (String)AccessController.doPrivileged(
      new PrivilegedAction() {
        public Object run() {
          return Localizer.getMessage(message);
      } } );
  } else {
    return Localizer.getMessage(message);
  }
}
```

Figure 5. The length-4 pattern is to execute an privileged action in Apache Tomcat (6SL in Table 6). ⟨isPackageProtectionEnabled / IF / doPrivilegedAction / END-IF ⟩

method. This is because `PrefixSpan` uses the number of sequences (methods) involving a pattern to select frequent patterns. For example, if 10 instances of a pattern were involved in three methods, its support count is three; the pattern is filtered out by the algorithm. Therefore, our approach might miss a pattern whose instances are concentrated in few methods. We have accepted this limitation since we focus on delocalized code fragments that affect software maintenance tasks rather than local patterns. To find all instances of a detected pattern after the mining process, we are planning to apply an AST-based matching approach [12].

5.3. Possible Extension

We use only IF and LOOP statements as control-flow information in coding patterns. A possible extension is to add rules to normalize `synchronized` and `try`/`catch`/`finally` blocks to detect synchronization and exception handling patterns.

An interesting question is how to detect common coding patterns among programs. A key challenge to automatically detect such patterns is how to compare method signatures in different applications; each program uses its own classes and methods in general.

Another challenge is the performance of the tool. The performance of PrefixSpan depends on the number of pattern candidates in a program. The current version of Fung takes a minute to analyze JHotDraw but it takes several hours to analyze Azureus on the same PC with 1GB RAM. To conduct a large scale analysis, we are planning to implement a parallel pattern mining system with a PC-cluster since *Parallel Modified PrefixSpan*, or an extension of PrefixSpan for parallel computing, is already available [23, 24].

6. Related Work

We proposed an application of a pattern mining algorithm to detect coding patterns or frequent idiomatic code fragments that are not modularized. Aspect mining techniques [5, 6, 18, 21] employ some heuristic functions to detect typical implementation of crosscutting concerns and apply refactoring to aspect candidates. The difference is that our approach detects control structure in addition to method calls as we described in Section 2.

Since coding patterns are not explicitly modularized, developers often copy-and-paste code fragments [15]. Such copy-and-pasted code are also known as code clones [3, 4, 13, 17]. However, most of code clone detection tools cannot detect code fragments modified after copy-and-pasted. For example, CCFinder, an efficient code clone detection tool, detects consecutive sequences of tokens [13]. Therefore, if a new statement is inserted to a copy-and-pasted code fragment, the modified code fragment is no longer a code clone of the original one. Our sequential pattern mining can detect such modified code fragments until the sequential order of method calls are modified. Another clone detection tool Deckard [12] and its extension [8] can detect a certain type of interleaved code fragments, but their approach is not to detect patterns that change an abstract syntax tree and a program dependence graph, e.g. a pattern in Figure 4.

Our approach focuses on method calls related to application classes rather than API usage, but API usage mining is similar to our work. Acharya proposed an approach to capture partial-ordered API usage [1]. Our sequential pattern mining approach detects total-ordered method calls and their control-flow information. Our approach may detect a partial-ordered API usage as several distinct sequential patterns.

7. Conclusion

We have adopted a sequential pattern mining algorithm to detect coding patterns that implement crosscutting concerns. We have developed a pattern mining tool named

Fung and applied the tool to six open-source Java programs. As a result, we have detected common coding patterns that are not modularized in the programs. Some patterns can be refactored using Template Method pattern or AspectJ advices. On the other hand, some other patterns such as logging are hard to modularize. Documenting such coding patterns is a possible way for us to help developers to maintain source code with the patterns.

In the future work, we will investigate a way to generate documentation for developers to understand coding patterns that are hard to modularize. We are also planning to improve the performance of our tool and make the tool public.

Acknowledgements

This research was supported by the Microsoft IJARC CORE4 Project.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the Joint Meeting of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 25–34, 2007.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14, 1995.

[3] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 6:49–57, 1992.

[4] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the 14th International Conference on Software Maintenance*, pages 368–377, 1998.

[5] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proceedings of the 21st International Conference on Automated Software Engineering*, pages 221–230, 2006.

[6] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

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

[8] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, pages 321–330, 2008.

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

[10] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous modification support based on code clone analysis. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 262–269, 2007.

[11] T. Hon and G. Kiczales. Fluid AOP join point models. In *Proceedings of the 2nd Asian Workshop on Aspect-Oriented Software Development*, pages 14–17, 2006.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, 2007.

[13] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[15] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[16] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proceedings of the Joint Meeting of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 187–196, 2005.

[17] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309, 2001.

[18] J. Krinke. Mining control flow graphs for crosscutting concerns. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 334–342, 2006.

[19] M. Marin. Reasoning about assessing and improving the seed quality of a generative aspect mining technique. In *Proceedings of the International Linking Aspect Technology and Evolution Workshop*, 2006.

[20] M. Marin, L. Moonen, and A. van Deursen. SoQueT: Query-based documentation of crosscutting concerns. In *Proceedings of the 29th International Conference on Software Engineering*, pages 758–761, 2007.

[21] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 132–141, 2004.

[22] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the 17th International Conference on Data Engineering*, pages 215–224, 2001.

[23] T. Sutou, K. Tamura, Y. Mori, and H. Kitakami. Design and implementation of parallel modified prefixspan method. In *Proceedings of the 5th International Symposium on High Performance Computing*, volume 2858 of *Lecture Notes in Computer Science*, pages 412–422, 2003.

[24] M. Takaki, K. Tamura, T. Sutou, and H. Kitakami. New dynamic load balancing for parallel modified prefixspan. In *Proceedings of the 21st International Conference on Data Engineering Workshops*, pages 1243–1246, April 2005.

[25] L. Ye and K. D. Volder. Tool support for understanding and diagnosing pointcut expressions. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, pages 144–155, 2008.