Investigation of Coding Patterns over Version History

Hironori Date, Takashi Ishio, Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka 565-0871, Japan
Email: {h-date, ishio, inoue}@ist.osaka-u.ac.jp

***Abstract*—A coding pattern is a sequence of method calls and control structures, which appears repeatedly in the source code. In this paper, we have extracted coding patterns of each version of two Java applications, and then explored the life-span of all of the coding patterns across those versions. This paper reports the characteristics of coding patterns of various life-spans. While learning from coding patterns enables us to perform appropriate modifications and enhancements for the software, many coding patterns are unstable as similar to the result of clone genealogy research.**

***Keywords*-Repository Mining; Coding Pattern; Sequential Pattern Mining; Java;**

I. Introduction

A coding pattern is a frequent sequence of method calls and control statements to implement a particular kind of concerns that are not modularized in the software [1]. Coding patterns include API usage patterns and application-specific behavior patterns. For example, a method call `hasNext` followed by a method call `next` is a typical usage of an `Iterator` object in Java. In addition to many instances of such API usage patterns, a large-scale application often includes its own coding patterns. For example, Apache Tomcat 6.0.14 has a logging feature for debugging. The feature is implemented by 304 pairs of `isDebugEnabled` and `debug` method calls. Azureus 3.0.2.2 is a multi-threaded program; it includes 151 methods using `AEMonitor` class to synchronize multi-threaded execution. A text editor jEdit 4.3 often calls `isEditable` with an `if` statement so that the text editor can prevent users from modifying a read-only file. Since coding patterns reflect implicit rules in a program, knowledge of patterns helps developers understand source code, and detect potential defects in the program [2], [3], [4].

Our research group developed a Coding Pattern Mining Tool named Fung, and in the previous research [1] we mined coding patterns from several applications. Figure 1 shows an example of coding pattern extracted from JHotDraw. From two class definitions, we obtain a coding pattern for "Undo" with length 4 and support (instance) 2, ⟨createUndoActivity(), setUndoActivity(), getUndoActivity(), setAffectedFigures()⟩. This means that the four method calls appear in those two classes in such order.

While existing work [5], [6], [7] used patterns extracted from source code as useful source code fragments, some

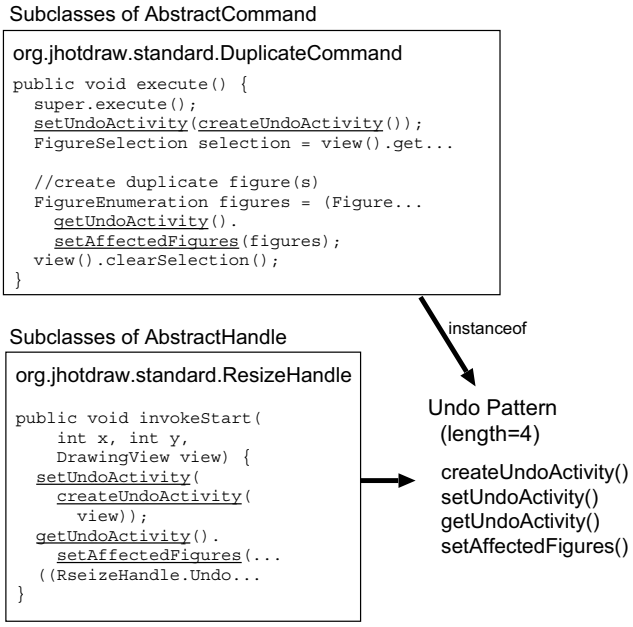

Figure 1. Undo pattern in JHotDraw 5.4b1 [1]

patterns may be involved only in a particular version of source code. If a pattern appears in multiple versions, the stable pattern is likely more reusable; in addition, the knowledge about such stable patterns may be effective for source code reading tasks. However, a long pattern of method calls always implies many shorter patterns of method calls. It is difficult to manually select likely stable patterns from the similar patterns.

In this research, we have investigated how many versions of an application include the same pattern, as similar to Clone Genealogy [8], [9]. Our pattern mining tool uses PrefixSpan, a sequential pattern mining algorithm [10]. Each coding pattern is a sequence of method calls and control elements such as `if`, `while` and `try-catch`. A pattern survives until the sequential order of method calls and control elements are modified.

We have applied our pattern mining to each version of two applications, dnsjava and JmDNS. We have chosen these middle-size applications so that we can extract all possible

patterns which have at least two instances and comprise at least two elements. In other words, if two methods include the same two method calls in the same order, we recognize the method calls as one of the shortest patterns. If the pair of method calls are not modified across versions, the pattern is recognized as a stable pattern.

We have analyzed 51 versions of dnsjava and 20 versions of JmDNS. Our results show that many patterns disappear in a few versions. Although we have extracted 25,909 patterns in source code, only 35 patterns are found in the all versions of an application. While the generalizability of our investigation is limited since we could not detect renamed methods, the result indicated that coding patterns should be extracted from a number of latest versions.

II. Coding Pattern Mining

The mining process of coding pattern we use here comprises two steps: *normalization step* and *mining step*. The normalization step translates each Java method, constructor or initializer in a program into a single sequence of call elements and control elements.

A method call is translated into a method call element with the method name and argument list. A constructor call is also translated into a constructor call element with the package name, class name and argument list.

The control elements are taken by the normalization rules shown in Table I. Some of these rules come from our previous work [1], but we have extended them by including additional control elements, `try-catch-finally` and `synchronized`, to enrich coding patterns.

In the mining step, we use a sequential pattern mining algorithm named PrefixSpan[10]. Sequential pattern mining extracts frequent subsequences from a set of sequences. Fung extracts only closed patterns; in other words, Fung filters out redundant shorter subpatterns whose instances are completely covered by the instances of a longer pattern.

III. Research Question

Our research question for the investigation of the life-span of the coding patterns is as follows.

RQ: *Are the coding patterns generally stable over the version history?*

In our previous paper, we have investigated the coding patterns in a single version of a software system [1]. In this work, we will trace the coding patterns through multiple versions of the software system. We would like to know if the coding patterns are fragile in the sense that a pattern found in a version can be easily disappear in the later versions. Or the patterns are fairly stable in the later versions.

If a pattern might be fragile, we would think that the pattern is temporary one, so we should not reuse the pattern in other systems. If it is stable, the pattern would be an important one to the later reuse for efficient and reliable coding.

Table I
Normalization rules of control elements

Source	for (<init>; <cond>; <inc>) <body>
Sequence	<init>, <cond>, LOOP, <body>, <inc>, <cond>, END-LOOP
Source	for (: <init>) <body>
Sequence	<init>, LOOP, <body>, END-LOOP
Source	while (<cond>) <body>
Sequence	<cond>, LOOP, <body>, <cond>, END-LOOP
Source	do <body> while (<body>)
Sequence	LOOP, <body>, <cond>, END-LOOP
Source	if (<cond>) <then> else <else>
Sequence	<cond>, IF, <then>, ELSE, <else>, END-IF
Source	<cond> ? <then> : <else>
Sequence	<cond>, IF, <then>, ELSE, <else>, END-IF
Source	try <try> catch ( ) <catch> ... finally <finally>
Sequence	TRY, <try>, CATCH, <catch>, ..., FINALLY, <finally>, END-TRY
Source	synchronized(<exp>) <body>
Sequence	<exp>, SYNCHRONIZED, <body>, END-SYNCHRONIZED
Source	synchronized return-type method-name(args) <body>
Sequence	SYNCHRONIZED, <body>, END-SYNCHRONIZED
Source	throw <exp>
Sequence	<exp>, THROW

Table II
Target programs

Application	Version Range	#Version	#Pattern
dnsjava	0.1 to 2.0.1	51	17,284
JmDNS	0.2 to 3.4.1	20	8,625

IV. Overview of Experiments

To answer the research question, we have extracted coding patterns over multiple versions. We have mined for patterns from each single version individually, and then we have searched identical patterns in multiple versions. Two coding patterns are judged as identical if all the elements of the patterns are identical. It is necessary to check not only consecutive two versions but all pairs of arbitrary two versions, since the identical patterns may be found at non-consecutive two versions. For example, a pattern extracted in version 1 can temporarily disappear from version 2, and appear in version 3 again.

We define the *life-span* of a pattern as the number of versions where we find the identical pattern. For example, if a pattern is found in the version 1, 2 and 3, then its life-span is 3. If a pattern is found in the version 1 and 3 but not in 2, then its life-span is 2.

Fung takes two parameters: the minimum length of a pattern and the minimum number of occurrences (instances) of a pattern. We have extracted patterns which comprise at least 2 method calls and appear in at least 2 methods. We have chosen these values so that we can extract all possible patterns. If we extract only patterns which have at least 10 instances, we cannot distinguish a pattern which still have 9 instances (but not reported by Fung) and a completely deleted pattern.

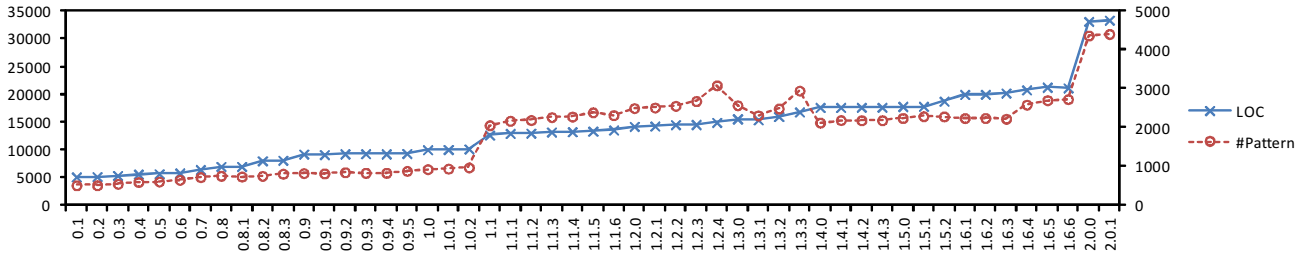

Figure 2. LOC and the number of patterns of dnsjava

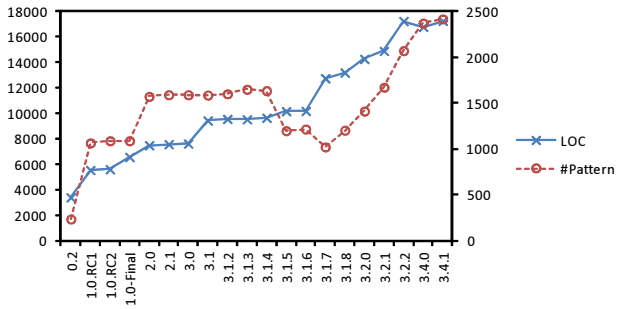

Figure 3. LOC and the number of patterns of JmDNS

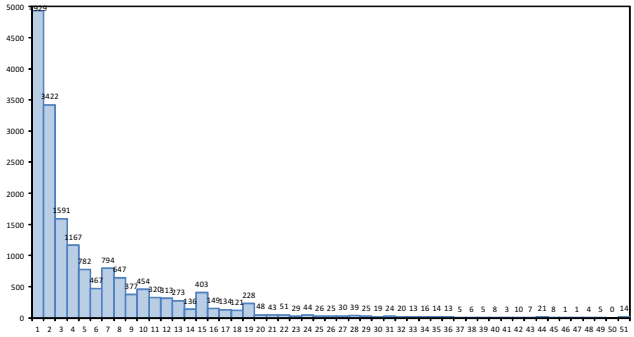

Figure 4. Life-span of patterns in dnsjava

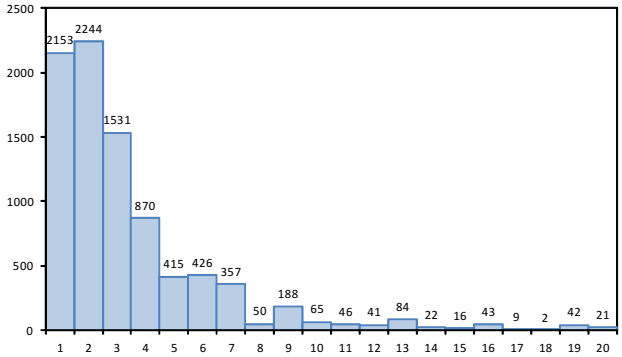

Figure 5. Life-span of patterns in JmDNS

We have analyzed two open source systems, dnsjava[1] and JmDNS[2]. Table II shows the target versions we have used in the experiments. Also it shows the number of patterns found by Fung.

V. Result of Experiments

A. Life-Span and Pattern Length

Figure 2 and Figure 3 show LOC (Lines Of Code) and the number of patterns extracted in each version of dnsjava and JmDNS. We can easily recognize that LOC grows over the versions, and that the number of pattern mostly grows along the LOC growth with minor decreases. We have investigated the correlation between LOC and the number of patterns. As a result, the correlation coefficients become 0.912 on dnsjava and 0.721 on JmDNS. Thus there is a strong positive correlation between LOC and the number of patterns.

Figure 4 presents the frequency distribution of the life-span in dnsjava, and Figure 5 in JmDNS. The life-span of patterns mined in dnsjava range from 1 to 51, and that in JmDNS range from 1 to 20. The most frequent life-span is 1 in dnsjava and 2 in JmDNS.

Figure 6 indicates that the frequency of pattern length and life-span in dnsjava, and Figure 7 indicates that in JmDNS. The range of pattern length is 2 to 45 in dnsjava, and 2 to 79 in JmDNS. In dnsjava, the peak 425 appears at 2 in life-span and 2 in pattern length. In JmDNS, the peak 204 appears at 2 in life-span and 6 in pattern length. In JmDNS, compared to dnsjava, there exist patterns including many elements. In both cases, there are a few long-lived patterns including many elements.

B. Examples of Patterns

You can see the complete list of the patterns which appear in all versions of dnsjava in Table III and JmDNS in Table IV. There are only 14 patterns out of total 17,284 patterns in dnsjava, and 21 patterns out of total 8,625 patterns in JmDNS. As the number of instances of a pattern differs depending on software versions, we show the minimum and maximum values, and the values in the first and the last

[1] dnsjava, http://sourceforge.net/projects/dnsjava/
[2] JmDNS, http://sourceforge.net/projects/jmdns/

version. Characteristics of these patterns with the longest life-span are as follows.

- Most patterns are related to the usage of Java library.
- Pattern length tends to be short.

VI. Observations

We answer to the following research question.

RQ: *Are the coding patterns generally stable over the version history?*

A: *No, the coding patterns are NOT generally stable over*

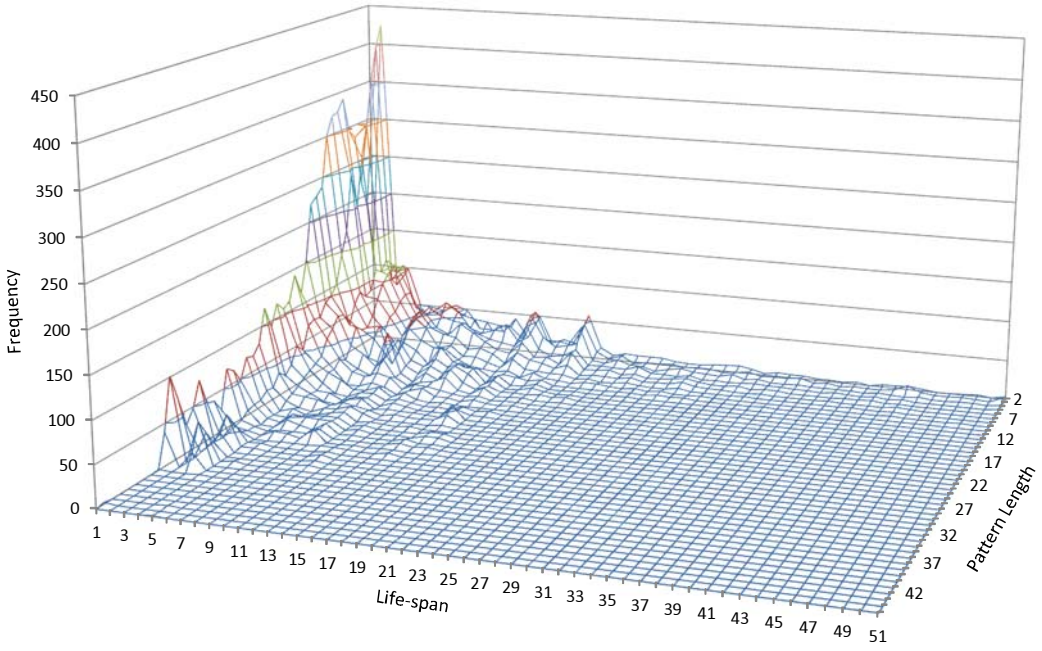

Figure 6. Pattern length and life-span in dnsjava

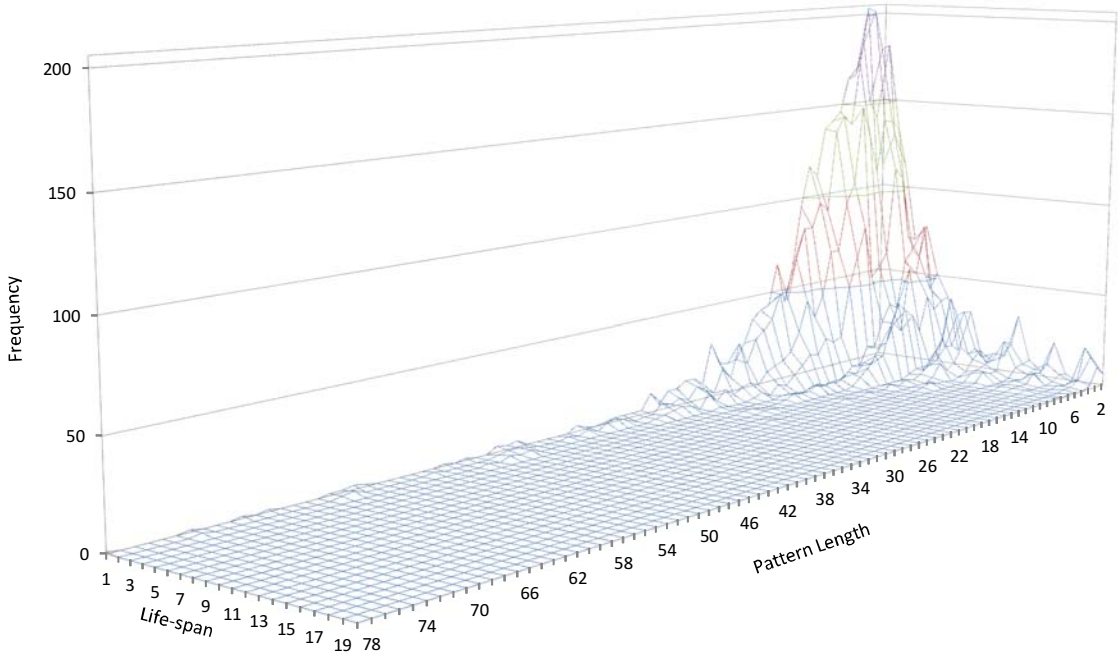

Figure 7. Pattern length and life-span in JmDNS

Table III
PATTERNS WHICH APPEARS IN ALL VERSIONS OF DNSJAVA

Len.	#Instance Min.	Max.	Ver. 0.1	Ver. 2.0.1	Pattern
2	2	6	2	5	⟨getHeader(), getRcode()⟩
2	3	8	3	7	⟨getHeader(), getName()⟩
2	5	7	5	5	⟨java.io.InputStreamReader.<init>(java.io.InputStream), java.io.BufferedReader.<init>(java.io.Reader)⟩
3	3	11	3	11	⟨LOOP, equals(java.lang.Object), END-LOOP⟩
3	3	23	3	23	⟨LOOP, get(int), END-LOOP⟩
3	3	4	3	3	⟨LOOP, getCount(int), END-LOOP⟩
3	2	4	3	4	⟨LOOP, startsWith(java.lang.String), END-LOOP⟩
3	5	18	5	7	⟨hasMoreTokens(), nextToken(), hasMoreTokens()⟩
3	7	13	7	12	⟨LOOP, charAt(int), END-LOOP⟩
3	6	11	8	9	⟨LOOP, length(), END-LOOP⟩
4	5	12	5	7	⟨LOOP, nextToken(), hasMoreTokens(), END-LOOP⟩
4	4	8	7	6	⟨length(), LOOP, length(), END-LOOP⟩
5	4	10	4	6	⟨hasMoreTokens(), LOOP, nextToken(), hasMoreTokens(), END-LOOP⟩
5	3	6	5	5	⟨length(), LOOP, charAt(int), length(), END-LOOP⟩

Table IV
PATTERNS WHICH APPEARS IN ALL VERSIONS OF JMDNS

Len.	#Instance Min.	Max.	Ver. 0.2	Ver. 3.4.1	Pattern
2	2	2	2	2	⟨charAt(int), writeByte(int)⟩
2	2	2	2	2	⟨parseInt(java.lang.String), registerService(javax.jmdns.ServiceInfo)⟩
2	2	5	2	3	⟨toLowerCase(), remove(java.lang.Object)⟩
2	2	3	3	2	⟨substring(int, int), parseInt(java.lang.String)⟩
2	3	15	3	12	⟨toLowerCase(), get(java.lang.Object)⟩
2	3	4	4	3	⟨getAddress(), getPort()⟩
3	2	33	2	33	⟨TRY, close(), END-TRY⟩
3	2	14	2	14	⟨LOOP, get(java.lang.Object), END-LOOP⟩
3	2	5	2	5	⟨LOOP, put(K, V), END-LOOP⟩
3	2	12	2	9	⟨IF, equals(java.lang.Object), END-IF⟩
3	2	13	2	13	⟨IF, get(java.lang.Object), END-IF⟩
3	2	2	2	2	⟨isQuery(), getAddress(), getPort()⟩
3	3	3	3	3	⟨LOOP, charAt(int), END-LOOP⟩
3	3	4	3	4	⟨LOOP, writeByte(int), END-LOOP⟩
3	3	12	4	5	⟨SYNCHRONIZED, get(java.lang.Object), END-SYNCHRONIZED⟩
3	2	11	4	10	⟨IF, length(), END-IF⟩
3	4	9	4	7	⟨IF, put(K, V), END-IF⟩
4	2	2	2	2	⟨SYNCHRONIZED, getProperties(), get(java.lang.Object), END-SYNCHRONIZED⟩
4	2	2	2	2	⟨LOOP, length(), startsWith(java.lang.String), END-LOOP⟩
7	2	2	2	2	⟨IF, write(int), ELSE, write(int), ELSE, write(int), END-IF⟩
9	2	2	2	2	⟨LOOP, IF, write(int), ELSE, IF, write(int), END-IF, END-IF, END-LOOP⟩

the version history.

As described in Section V, there are few coding patterns with long life-span. On the other hand, coding patterns with short life-span account for a large part of all patterns.

According to Figure 4 and Figure 5, most patterns live very short in the history, and there are few patterns appearing in all versions. The total of the patterns living through less than 15 versions is 90% of all dnsjava's patterns, and the median life-span is 3. Similarly, the total of the patterns living through less than 8 versions covers 90% of all in JmDNS patterns, and the median is 2.

Our results on coding patterns are consistent with the result of code clone genealogy research [8]. Many code clones also disappear in a few versions, and code clones including method calls imply coding patterns. Some disappeared coding patterns are affected by code cloning activity of developers.

VII. THREATS TO VALIDITY

A. Experimental Objects

We chose 2 as the threshold of the minimum occurrence of a pattern so that we can extract all possible patterns in pattern mining process. As pattern mining under this severe condition consumes huge time, we cannot adopt large size of applications as targets of experiments. An effective solution to this limitation is the reimplementation of Fung with more efficient sequential pattern mining algorithm.

As dnsjava and JmDNS have the word "dns" in the names of the applications, you may think these applications to be unfair (biased) experimental objects. However, each "dns" implies different kind of service; dnsjava is an implementation of DNS, while JmDNS is an implementation of multicast DNS used in local networks. The implementations are not related to each other.

The generality of our findings is limited since we have analyzed only 2 applications. Further experiments with various kinds of software make it clear whether our findings are general or not.

B. Change of Method Name

In case that a callee method name has changed, we cannot tie the renamed methods over versions. Thus, we cannot bind the before and after versions of the patterns which include the call of the renamed method. As the change of a method name should imply the change of the contents in the method body, the meaning of the related patterns may change. Therefore, there is no need to treat patterns as the same ones before and after the change of the method name.

C. Verbose Subpatterns

Our pattern tracking algorithm in this paper cannot track relationships among super/sub-patterns. Suppose that a method has a sequence of method calls ⟨A, B, C⟩ and another method has a sequence of method calls ⟨A, B, C, D⟩. Fung recognizes a pattern ⟨A, B, C⟩ in these methods. If a developer added a method call D to the former method, Fung recognizes a new pattern ⟨A, B, C, D⟩ and filters out ⟨A, B, C⟩ because the shorter pattern is covered by the new pattern. In this case, we regard the pattern ⟨A, B, C⟩ as a disappeared pattern. We did not track this kind of super/sub-patterns because a longer pattern implies a huge number of patterns. For example, the pattern ⟨A, B, C, D⟩ implies four 3-element patterns (⟨A, B, C⟩, ⟨A, B, D⟩, ⟨A, C, D⟩ and ⟨B, C, D⟩) and six 2-element patterns (⟨A, B⟩, ⟨A, C⟩, ⟨A, D⟩, ⟨B, C⟩, ⟨B, D⟩ and ⟨C, D⟩). In general, a pattern comprising N elements implies nearly 2^N subpatterns.

According to the limitation, the life-span in this paper is possibly underestimated. For more detailed investigation, we should improve the tracking algorithm to deal with super/subpatterns.

VIII. Conclusion

In this paper, We investigated the stability of coding patterns across versions. We defined a life-span of coding pattern as the number of versions where we find the identical pattern, and investigated the 51 versions of dnsjava and 20 versions of JmDNS.

As a result, many patterns disappear in a few versions. 90% of all dnsjava's patterns are found in at most 15 versions. The median life-span is 3. Similarly, 90% of all JmDNS's patterns are found in at most 8 versions. The median life-span is 2.

While the generalizability of our investigation is limited since we could not detect renamed methods and super/sub patterns, the result indicated that coding patterns should be extracted from a number of latest versions so that developers can filter out temporary patterns.

Acknowledgment

This research was supported by JSPS KAKENHI Grant Number 23680001.

References

[1] T. Ishio, H. Date, T. Miyake, and K. Inoue, "Mining coding patterns to detect crosscutting concerns in java programs," in *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008, pp. 123–132.

[2] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the Joint Meeting of the 12th European Software Engineering Conference and the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2009, pp. 383–392.

[3] H. Kagdi, M. Collard, and J. Maletic, "An approach to mining call-usage patterns with syntactic context," in *Proceedings of the 22nd International Conference on Automated Software Engineering*, 2007, pp. 457–460.

[4] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code." in *Proceedings of the Joint Meeting of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 306–315.

[5] 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*, 2007, pp. 25–34.

[6] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 69–79.

[7] S. Thummalapenta and T. Xie, "PARSEWeb: A programmer assistant for reusing open source code on the web," in *Proceedings of the 22nd International Conference on Automated Software Engineering*, 2007, pp. 204–213.

[8] 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 International Symposium on Foundations of Software Engineering*, 2005, pp. 187–196.

[9] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at release level," in *Proceedings of the 16th Working Conference on Reverse Engineering*, 2009, pp. 85–94.

[10] 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*, 2001, pp. 215–224.