Analysis of Coding Patterns over Software Versions

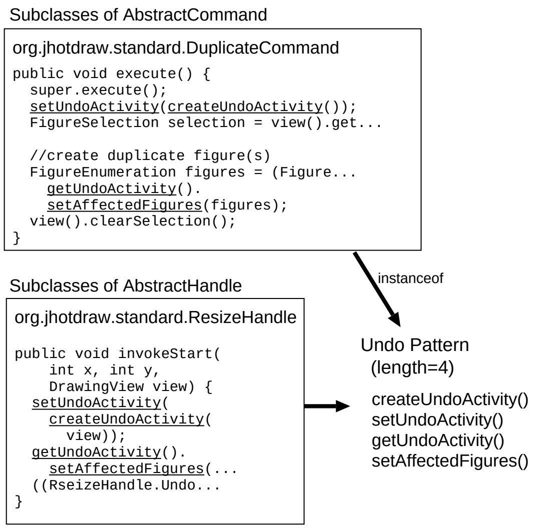
Hironori Date, Takashi Ishio, Makoto Matsushita, Katsuro Inoue

A coding pattern is a sequence of method calls and control structures, which appears repeatedly in source code. In this paper, we have extracted coding patterns of each version of ten Java programs, and then explored the number of versions in which the coding patterns appear. This paper reports the characteristics of coding patterns over versions. While learning from coding patterns is expected to help developers to perform appropriate modifications and enhancements for the software, many coding patterns are unstable as similar to the result of clone genealogy research.

1 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 software [4]. 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 [5] [7] [9].

Our research group developed a coding pattern mining tool named Fung, and in the previous research we mined coding patterns from several applications [4]. Fig. 1 shows an example of coding pattern extracted from JHotDraw. From two class definitions, we obtain a coding pattern for "Undo" ⟨createUndoActivity(), setUndoActivity(), getUndoActivity(), setAffectedFigures()⟩, where its length is four and the number of instances is two. This means that the sequence of four method


Subclasses of AbstractCommand
org.jhotdraw.standard.DuplicateCommand
Subclasses of AbstractHandle
org.jhotdraw.standard.ResizeHandle
instanceof
Undo Pattern
(length=4)
createUndoActivity()
setUndoActivity()
getUndoActivity()
setAffectedFigures()

```
public void execute() {
  super.execute();
  setUndoActivity(createUndoActivity());
  FigureSelection selection = view().get...

  //create duplicate figure(s)
  FigureEnumeration figures = (Figure...
    getUndoActivity().
    setAffectedFigures(figures);
  view().clearSelection();
}
```

```
public void invokeStart(
    int x, int y,
    DrawingView view) {
  setUndoActivity(
    createUndoActivity(
      view));
  getUndoActivity().
    setAffectedFigures(...
  ((RseizeHandle.Undo...
}
```

Fig. 1 Undo pattern in JHotDraw 5.4b1 [4]

バージョンを超えて存在するコーディングパターンの調査
伊達浩典, 石尾隆, 松下誠, 井上克郎, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.
コンピュータソフトウェア, Vol.32, No.1 (2015), pp.220–226.
[研究論文 (レター)] 2014 年 6 月 20 日受付.

Table 1 Target programs and extracted patterns

Program	#Version (Version Range)	LOC Range	#Pattern	#Stable Pattern (%)	#Common Pattern	#Stable / #Common Pattern (%)	PCC
CAROL[†1]	12 (1.0.1 to 2.0.5)	7,546 to 25,944	6,425	112 (1.7%)	146	76.7%	0.641
Cewolf[†2]	14 (1.0 to 1.1.12)	8,485 to 14,891	2,622	155 (5.9%)	157	98.7%	0.988
dnsjava[†3]	51 (0.1 to 2.0.1)	5,084 to 33,330	17,284	108 (0.6%)	287	37.6%	0.883
Jackcess[†4]	32 (1.0 to 1.2.8)	4,483 to 29,016	7,576	192 (2.5%)	291	66.0%	0.995
JmDNS[†5]	20 (0.2 to 3.4.1)	3,408 to 17,252	8,625	55 (0.6%)	93	59.1%	0.734
Joda-Time[†6]	19 (0.9 to 2.1)	40,311 to 138,710	6,663	524 (7.9%)	815	64.3%	0.984
NatTable[†7]	20 (alpha0.2 to 2.3.2)	5,520 to 42,377	6,762	66 (1.0%)	152	43.4%	0.900
OntoCAT[†8]	19 (0.9.4 to 0.9.9.1)	6,226 to 13,605	3,348	567 (16.9%)	593	95.6%	0.967
OVal[†9]	19 (0.1 to 1.80)	3,249 to 25,235	6,275	57 (0.9%)	117	48.7%	0.670
transmorph[†10]	13 (1.0.0 to 3.1.1)	6,612 to 19,090	3,609	187 (5.2%)	256	73.1%	0.940

†1 CAROL, http://carol.ow2.org/. †2 Cewolf, http://cewolf.sourceforge.net/new/index.html. †3 dnsjava, http://www.dnsjava.org/. †4 Jackcess, http://jackcess.sourceforge.net/. †5 JmDNS, http://sourceforge.net/projects/jmdns/. †6 Joda-Time, http://www.joda.org/joda-time/. †7 NatTable, http://sourceforge.net/projects/nattable/. †8 OntoCAT, http://www.ontocat.org/. †9 OVal, http://oval.sourceforge.net/. †10 transmorph, https://github.com/cchabanois/transmorph.

calls appears in those two methods.

While existing work [1] [8] [11] used patterns extracted from source code as reusable code, some patterns may be involved only in a particular version of source code. If a pattern appears in multiple versions, it is likely more reusable; in addition, the knowledge about such 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 useful 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 studies [2] [6]. 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 analyzed ten Java applications listed in Table 1. 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 such method calls is not modified across all versions, the pattern is recognized as a *stable pattern*.

This paper is a revised version of our previous work [3]. The main differences from the previous work are summarized as follows:

- Refined the definition of the number of versions where a pattern appears.
- Increased the number of target applications in the experiment from 2 to 10.
- Analyzed from the viewpoint of transition of the number of patterns between software versions.

2 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, static initializer or instance 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 in a method are obtained by normalization rules. Several normalization rules are shown in Table 2, while the complete list of the rules including exception handling and synchronization constructs is available in [3]. A part be-

Table 2 Examples of normalization rules for control statements

Source	Sequence
for (<init>; <cond>; <inc>) <body>	⟨<init>, <cond>, LOOP, <body>, <inc>, <cond>, END-LOOP⟩
while (<cond>) <body>	⟨<cond>, LOOP, <body>, <cond>, END-LOOP⟩
do <body> while (<cond>)	⟨LOOP, <body>, <cond>, END-LOOP⟩
if (<cond>) <then> else <else>	⟨<cond>, IF, <then>, ELSE, <else>, END-IF⟩

tween "<" and ">" in source code is replaced with a converted sequence of the part.

In the mining step, we use a sequential pattern mining algorithm PrefixSpan [10]. Sequential pattern mining extracts frequent subsequences from a set of sequences. Our tool 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.

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

3 Counting Versions of Coding Patterns

We have mined for patterns from each single version individually, and then we have searched identical patterns appeared 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 may temporarily disappear from version 2, but appear in version 3 again.

We define a function $NV(p)$ which returns the number of versions for a pattern p.

$$NV(p) = \left|\left\{v_i \middle| {}^{\exists}p_k \in P(v_i) : p \sqsubseteq p_k\right\}\right|$$

where $P(v_i)$ is a function that returns all patterns extracted from a version v_i. $p \sqsubseteq p_k$ means that p is a subsequence of p_k. For example, $\langle a, c, d\rangle$ is a subsequence of $\langle a, b, c, d, e\rangle$.

If a pattern p_1 is found in the version 1, 2, and 3, then $NV(p_1) = 3$. If another pattern p_2 is found in the version 1 and 3 but not in 2, then $NV(p_2) = 2$.

In the previous research [3], we used $NV_{old}(p) = |\{v_i | p_k \in P(v_i)\}|$ described as *life-span*, but we have relaxed this condition to include more related patterns in our analysis.

4 Analysis

4.1 Approach

We have analyzed ten Java open source programs, CAROL, Cewolf, dnsjava, Jackcess, JmDNS, Joda-Time, NatTable, OntoCAT, OVal, and transmorph. Table 1 shows their versions we have used in the experiments and the software size (Lines Of Code) of the minimum and maximum size in the versions.

At first, we count the number of patterns extracted from each application and count the number of versions each of the patterns exists.

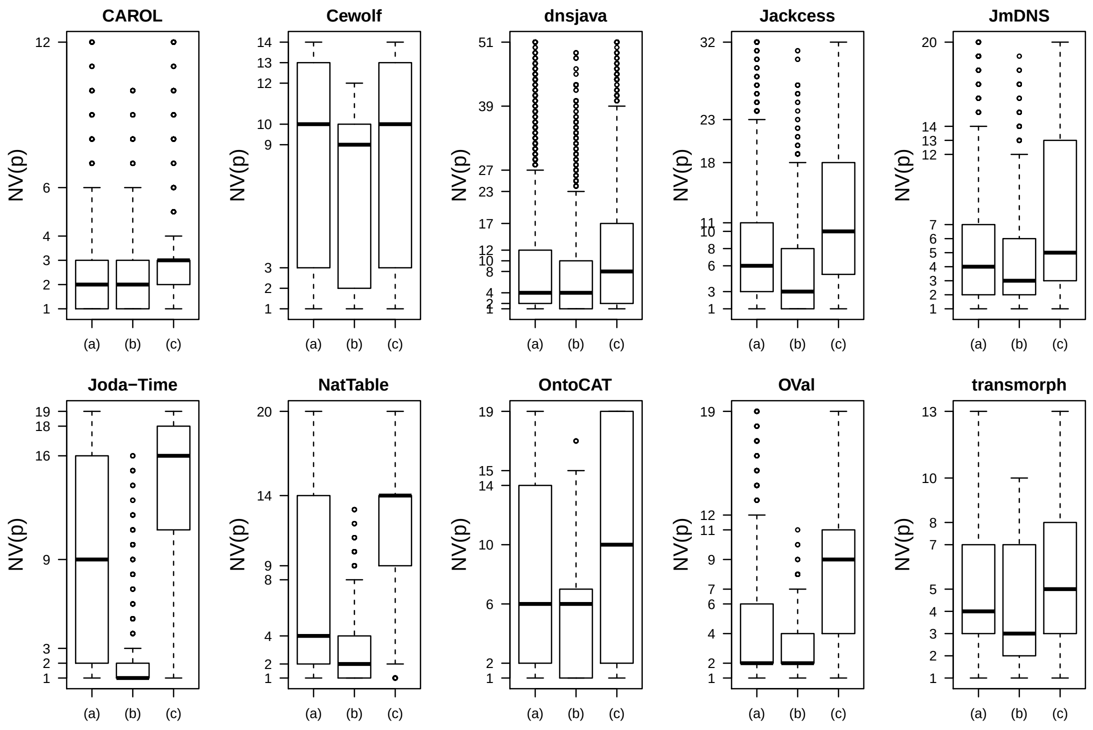
Then, since developers modify the latest version of source code in most cases, we are interested in whether a pattern survives to the latest version. Thus, we classify patterns into two categories, the patterns which appear in the latest version and ones which do not, and evaluate the difference of $NV(p)$ between patterns in these categories.

At last, we introduce some interesting patterns found in this analysis.

4.2 Stability of Patterns

The total number of patterns and the number of stable patterns (patterns appearing in all versions) are listed in Table 1.

We investigated more than ten versions of each application, and extracted approximately from 2,600 to 17,000 patterns. The distributions of $NV(p)$ of all patterns are plotted as metric (a) in Fig. 2. As $NV(p)$ of most patterns indicates very small value, patterns tend to be unstable and frag-



(a) $NV(p)$ for all patterns, (b) $NV(p)$ for patterns not appearing in the latest version, and (c) $NV(p)$ for patterns appearing in the latest version

Fig. 2 Distribution of $NV(p)$

ile. According to Table 1, on the other hand, there are from 55 to 567 patterns which appear in all versions. However, these stable patterns account for only a small fraction of all patterns.

Our results on coding patterns are consistent with the research of code clone genealogies [6]. 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.

4. 3 Patterns in Latest Version

We plotted $NV(p)$ of patterns not appearing in the latest version as metric (b), and $NV(p)$ of patterns appearing in the latest version as metric (c) in Fig. 2. (a) is combined result of (b) and (c).

By the definition, $(b) = (c) - 1$ is expected. Actual differences can be seen as the difference of Fig. 2 (b) and (c), and they are generally greater than 1. This means that the patterns not appearing in the latest versions would be rather fragile in the version history. In other words, the patterns appearing in the latest version are more stable than the others.

4. 4 Changes of the Number of Patterns

Fig. 3 shows that the number of patterns which appear (positive direction) and disappear (negative direction) at a version v, from the immediately previous version $v-1$. It also shows transitions of the number of patterns extracted from a version and the Lines of Code (LOC). Because of the limitation of the space we only show the result of JmDNS here. The other nine applications show similar results.

Focusing on a single version, as the similar numbers of patterns appear and disappear, a coding pattern tends to be replaced with another pattern when code has been modified. This observation supports the unstability of coding patterns.

Fig. 3 also shows that LOC and the number of patterns increase gradually through the progress of development. Pearson product-moment correlation

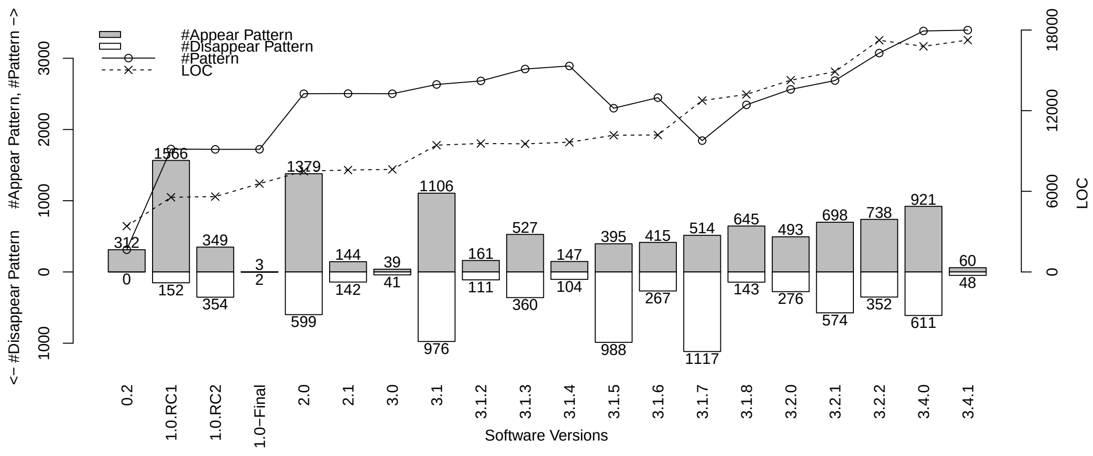


Fig. 3 Changes of the number of patterns (JmDNS)

coefficient (PCC) between LOC and the number of patterns is indicated in last column of Table 1. As the range of PCC is from 0.670 to 0.995 and the values of eight applications are greater than 0.7, there is a strong positive correlations between LOC and the number of patterns. This indicates that new code introduces new patterns.

We extract common patterns between the oldest version and the latest version in the target programs, and present the number of the common patterns in Table 1. As these common patterns include the stable patterns, we also show the percentage of the stable patterns in the common patterns between the oldest version and the latest version. This result tells us that we can effectively extract stable patterns by investigating only two (oldest and latest) versions.

4.5 Examples of Stable Patterns

This section reports the characteristic examples of patterns identified through this experiment. We selected the patterns appearing in all versions and investigate them to be worth introducing.

Pattern 1: ***idiom*** This pattern is extracted from multiple applications. The sequence is as follows: ⟨iterator(), hasNext(), LOOP, next(), hasNext(), END-LOOP⟩. This pattern might be less important because it is well-known.

Pattern 2: ***debugging*** This pattern ⟨isDebug Enabled(), IF, debug(java.lang.String), END-IF⟩ is extracted from Jackcess. This pattern changes the behavior of the program if it is executed under the debug mode. This kind of pattern should be kept for the consistent operation of the software.

Pattern 3: ***try-catch*** This pattern ⟨TRY, CATCH, printStackTrace(), END-TRY⟩ is from JmDNS, related to the exception handling in Java applications. This is a kind of *idiom* that outputs the state of stack memory to notify the exceptional situation to developers.

Pattern 4: ***multi-threading*** This pattern also appears in all versions of JmDNS and consists of three elements, ⟨SYNCHRONIZED, get(java.lang.Object), END-SYNCHRONIZED⟩. This pattern shows exclusive control of the access to a collection object. As bugs related to multi-thread are difficult to find, we should investigate carefully the related code of this kind of patterns, especially violating the rules.

Pattern 5: ***instantiation*** Joda-Time gives us a pattern of ⟨getChronology(), org.joda.time.Date Time.<init>(long, org.joda.time.Chronology)⟩. This is an application specific pattern to create a Date Time object from a Chronology object. This type of patterns would be useful for newcomers to the development community.

Pattern 6: ***library*** NatTable uses SWT library, and has its regular usage like, ⟨java.lang.Runnable. <init>(), asyncExec()⟩. If we collect this kind of patterns for a specific library from various applications, we may create a practical manual with real code.

5 Threat to Validity

Since we have used method names to identify identical patterns across versions, we cannot track patterns which include the call of a renamed method. As the change of a method name may imply the change of the meaning of the method, we did not treat patterns as the same ones before and after the change of the method name.

6 Conclusion

In this paper, we investigated the stability of coding patterns across versions. We defined a function $NV(p)$ of coding pattern as the number of versions including a pattern p, and investigated more than ten versions of ten target applications.

Contrary to our expectation, many patterns disappear in a few versions. Only 0.6% to 16.9% of all patterns are extracted from all versions of application. Thus, most of the patterns have short life and are unstable and fragile. Comparing to patterns which are not extracted from the latest versions of application, patterns which are extracted from the latest versions tend to have long life. The result indicated that stable coding patterns would be extracted from the oldest and the latest versions. At the beginning of software development, source code is sometimes modified drastically and patterns would be also changed by the modification. Moreover, fewer common patterns are extracted between different major versions. If developers limit the version range on analysis, the resultant patterns might be suitable for their interest.

In the future work, we would like to evaluate the effectiveness of code completion using stable coding patterns.

Acknowledgment This work is supported by KAKENHI (S) No.25220003 and (B) No.26280021.

References

[1] Acharya, M., Xie, T., Pei, J. and Xu, J.: Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications, in *Proceedings of the 6th Joint Meeting of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE'07*), 2007, pp. 25–34.

[2] Bettenburg, N., Shang, W., Ibrahim, W., Adams, B., Zou, Y. and Hassan, A. E.: An Empirical Study on Inconsistent Changes to Code Clones at Release Level, in *Proceedings of the 16th Working Conference on Reverse Engineering* (*WCRE'09*), 2009, pp. 85–94.

[3] Date, H., Ishio, T. and Inoue, K.: Investigation of Coding Patterns over Version History, in *Proceedings of the 4th International Workshop on Empirical Software Engineering in Practice* (*IWESEP'12*), 2012, pp. 40–45.

[4] Ishio, T., Date, H., Miyake, T. and Inoue, K.: Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs, in *Proceedings of the 15th Working Conference on Reverse Engineering* (*WCRE'08*), 2008, pp. 123–132.

[5] Kagdi, H., Collard, M. L. and Maletic, J. I.: An Approach to Mining Call-Usage Patterns with Syntactic Context, in *Proceedings of the 22nd International Conference on Automated Software Engineering* (*ASE'07*), 2007, pp. 457–460.

[6] Kim, M., Sazawal, V., Notkin, D. and Murphy, G. C.: An Empirical Study of Code Clone Genealogies, in *Proceedings of the 5th Joint Meeting of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE'05*), 2005, pp. 187–196.

[7] Li, Z. and Zhou, Y.: PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code, in *Proceedings of the 5th Joint Meeting of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE'05*), 2005, pp. 306–315.

[8] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Tamrawi, A., Nguyen, H. V., Al-Kofahi, J. and Nguyen, T. N.: Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion, in *Proceedings of the 34th International Conference on Software Engineering* (*ICSE'12*), 2012, pp. 69–79.

[9] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M. and Nguyen, T. N.: Graph-based Mining of Multiple Object Usage Patterns, in *Proceedings of the 7th Joint Meeting of the 12th European Software Engineering Conference and the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE'09*), 2009, pp. 383–392.

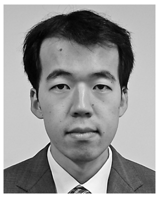
[10] Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.: PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth, in *Proceedings of the 17th International Conference on Data Engineering* (*ICDE'01*), 2001, pp. 215–224.

[11] Thummalapenta, S. and Xie, T.: PARSEWeb: A Programmer Assistant for Reusing Open Source

Code on the Web, in *Proceedings of the 22nd International Conference on Automated Software Engineering* (*ASE'07*), 2007, pp. 204–213.

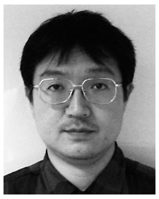
伊達浩典

2007 年関西大学総合情報学部卒業．2009 年大阪大学大学院情報科学研究科博士前期課程修了．2014 年同大学大学院情報科学研究科博士後期課程退学．プログラム解析，ソフトウェアパターンに関する研究に従事．情報処理学会会員．



石 尾 隆

2003 年大阪大学大学院基礎工学研究科博士前期課程修了．2006 年同大学情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員 (PD)．2007 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助教．博士 (情報科学)．プログラム解析，プログラム理解に関する研究に従事．日本ソフトウェア科学会，情報処理学会，電子情報通信学会，ACM，IEEE 各会員．



松 下 誠

1993 年大阪大学基礎工学部情報工学科卒業．1998 年同大学大学院博士後期課程修了．同年同大学基礎工学部情報工学科助手．2002 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助手．2005 年同専攻助教授．2007 年同専攻准教授．博士 (工学)．ソフトウェア開発環境，リポジトリマイニングの研究に従事．情報処理学会，日本ソフトウェア科学会，ACM 各会員．

井上克郎

1956 年生．1979 年大阪大学基礎工学部情報工学科卒業．1984 年同大学大学院博士課程了．同年同大学基礎工学部助手．1984 年–1986 年ハワイ大学マノア校情報工学科助教授．1989 年大阪大学基礎工学部講師．1991 年同助教授．1995 年同教授．2002 年大阪大学大学院情報科学研究科教授．2012 年大阪大学大学院情報科学研究科・研究科長．工学博士．ソフトウェア工学，特に，ソフトウェア開発手法，プログラム解析，再利用技術の研究に従事．