Mining Application-Specific Coding Patterns for Software Maintenance

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

ABSTRACT

A crosscutting concern is often implemented based on a coding pattern, or a particular sequence of method calls and control statements. We have applied a sequential pattern mining algorithm to capture coding patterns in Java programs. We have manually investigated the resultant patterns that involve both crosscutting concerns and implementation idioms. This paper discusses the detail of our pattern mining algorithm and reports detected crosscutting concerns.

1. INTRODUCTION

To develop a 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, 8, 10].

While Aspect-Oriented Programming (AOP) [13] and some object-oriented design patterns such as Template Method [8, 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. Iterator pattern, and some duplicated code fragments that has variants of the original fragment (e.g. a code fragment tangled with other functions).

To enable developers to understand and manage idiomatic code fragments, we have applied a sequential pattern mining algorithm to extract coding patterns for implementing a particular kind of concerns. We have developed a tool named Fung that translates a method in a Java program into a sequence of method calls and control-flow elements, and applies `PrefixSpan`, a sequential pattern mining algorithm proposed in [22].

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 [12, 16], a sequential pattern instance may involve disconnected method calls.

LATE Linking Aspect Technology and Evolution Workshop, April 1st, Brussels, Belgium


```
for (Iterator it=list.iterator();          Collection.iterator(
      it.hasNext(); ) {                    Iterator.hasNext()
  Item item = (Item)it.next();             LOOP
  if (item.isActive()) {                     Iterator.next()
    item.deactivate();                       Item.isActive()
  }                                          IF
}                                              Item.deactivate(
                                           END-IF
                                         END-LOOP
```

Figure 1: A sequence extracted from source code

We have applied our pattern mining to six Java programs: jEdit, JHotDraw, Azureus, Apache Tomcat, ANTLR and SableCC. We have found several crosscutting concerns that are hard to modularize using AspectJ because of their heterogeneous implementation. Our pattern mining supports developers to investigate such crosscutting concerns.

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

2. SEQUENTIAL PATTERN MINING

Sequential pattern mining extracts frequent subsequences from a sequence database [2]. We use PrefixSpan algorithm [22] to a sequence database extracted from a Java program. According to the limited space, we omit the detail of the algorithm.

We have developed a pattern mining tool Fung. Fung first translates the source code of a program into a sequence database and applies PrefixSpan algorithm. Then Fung filters and classifies the extracted patterns into pattern groups.

We defined a set of rules to translate the source code of a Java method into a sequence that comprises method call elements and control elements to capture patterns comprising method calls and control statements such as `if` and `for`. Figure 1 is an example of a sequence extracted from a source code fragment.

Method call element. A method call is translated into a call element.

To handle crosscutting implementation and dynamic binding, we simply ignore the class names in method calls. For example, String.equals and List.equals are not distinguished; a method call element "equals" is generated for each call.

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 textual sequence (left to right) in the source code.

IF/ELSE/END-IF element. An `if` statement is translated into a series of IF, ELSE and END-IF elements. 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.

LOOP/END-LOOP element. A `for` or `while` statement is translated into a pair of LOOP and END-LOOP elements. A method call in the predicate of the loop is translated into a pair of method call elements inserted before the LOOP element and the END-LOOP element according to control-flow of the loop statement.

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.

Our rules described above generate a sequence for each method in Java source code. PrefixSpan takes as input a sequence database and two threshold parameters: pattern length len and support count s. PrefixSpan outputs patterns that satisfy the following characteristics:

- A pattern is a sequence of method call elements and control elements.
- A pattern comprises at least len elements. For example, Figure 2 shows an Undo pattern comprising four method call elements: createUndoActivity, setUndoActivity, getUndoActivity and setAffectedFigures. This pattern is filtered out if len is greater than 4.
- A pattern has at least s instances. We use the term *instance* of a pattern to represent a concrete code fragment corresponding to the pattern. For example, Figure 2 shows three Undo pattern instances. If the number of instances is less than s, the pattern is filtered out.
- An instance of a pattern is defined as a list of tokens in the source code; each token corresponding to a pattern element.
- An instance 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 [a, b, c, d] implies four sub-patterns comprising 3-elements: [a, b, c], [a, b, d], [a, c, d] and [b, c, d]. 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, Fung supports 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 user of Fung can specify another threshold if necessary.

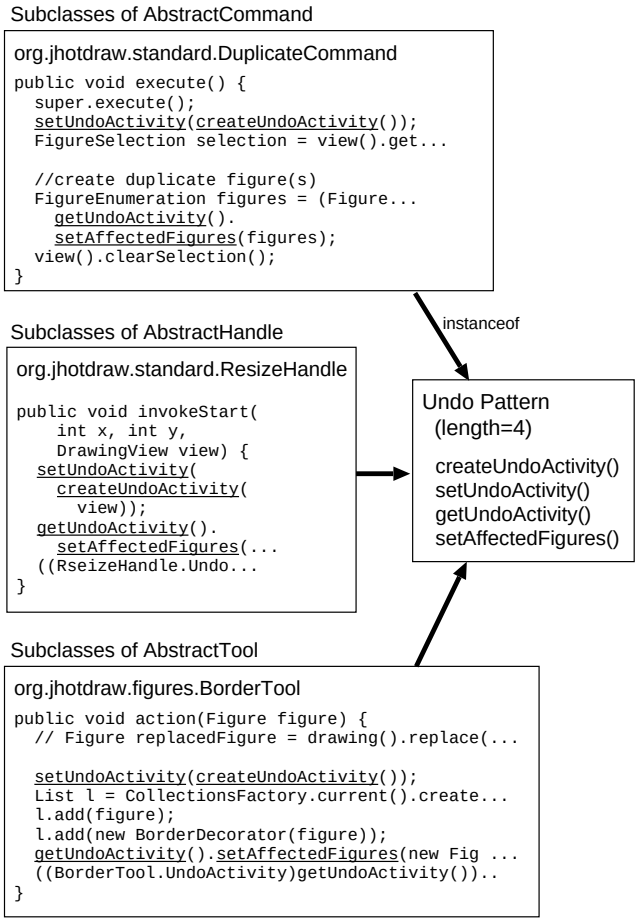

Figure 2: Undo pattern in JHotDraw 5.4b1

- Since a control statement is always transformed to a pair of the beginning and the end of a block (e.g., a pair of IF and END-IF elements), we filtered out patterns including a control element but excluding its peer 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: if two patterns p_1 and p_2 overlaps with each other, the two patterns are included in the same group. Therefore, a pattern and its sub-patterns are always included in the same group.

We extract *summary tokens* to indicate the summary of a pattern group. We split (tokenize) the method names in a pattern group with capital letters, and select the top three frequent tokens as the summary tokens of the group. For example, the pattern group including only the undo pattern in Figure 2 has the summary tokens [undo, activity, set]. This summary enables us to understand what to do in the patterns.

The resultant pattern groups are listed in Fung Pattern Viewer. Fung also exports the resultant patterns in an XML format.

3. CODING PATTERNS IN JAVA SOFTWARE

We have applied our pattern mining to the software listed in Table 1. We extracted patterns with parameters $len = 4$ and $s = 10$; a pattern comprises four or more elements and a pattern has at least

Table 1: Target Software

Name	Version	Size(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

Table 2: Patterns in JHotDraw 7.0.9

Pattern Group	Sup	Len	Type
Link action objects to menu objects (tokens: add, get, action)	29	16	Impl
Create menu objects (tokens: add, separator, contains)	28	7	Impl
Compare a string with string literals (tokens: equals)	24	4	Impl
Read entries in `for` loop (tokens: get, set, entry)	24	6	Impl
Create a rectangle from width and height (tokens: get, height, width)	21	5	Impl

ten instances in the program. In this paper, we show the top five of frequent patterns for each software according to the limited space. Each table consists of four columns: *Pattern Group*, *Sup*, *Len* and *Type*. *Pattern Group* represents the function of a pattern group (what to do). The "tokens" of a group indicate the frequent tokens in the method call elements included in the pattern group. *Sup* is the number of instances of the most frequent pattern in the group. *Len* is the number of elements of the longest pattern in the group. *Type* is a category we have manually assigned. `Impl` indicates that the pattern is an implementation idiom using only library classes and `App` indicates that the pattern contributes to a particular function in the program, respectively. We assigned the type `Impl` to a pattern if the pattern represents a general purpose code fragment so that we can reuse its instance to another Java program without any knowledge on the program including the pattern. Otherwise, we assigned the type `App`.

Table 2 shows the top five 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 2) is already refactored. All the five patterns are implementation idioms for typical operations in JHotDraw rather than application-specific concerns. This table does not mean that JHotDraw 7.0.9 has no crosscutting concerns; a pair of `willChange` and `changed` methods form a (less frequent) pattern to fire events before and after figures are manipulated.

Table 3 shows the patterns in jEdit. The top pattern group calls `openNodeScope` and `closeNodeScope` at the beginning and the end of methods in various classes used by `bsh.Parser`. This seems a typical crosscutting concern that may be refactored using the before and after advices in AspectJ. The third pattern is an interesting pattern because the method `beep` is called from various locations where jEdit executes some ext editing function to a text buffer. While capturing method calls to `beep` is easy, this pattern is difficult to directly refactor to an aspect because capturing all text editing functions using AspectJ pointcut designators is difficult.

Table 4 shows the patterns in Azureus. Azureus uses a pair of `enter` and `exit` methods to synchronize a packet buffer. Logging is also a crosscutting concern spread across the modules. Although a textual search can easily capture the code, the logging concern is difficult to modularize since Azureus records various messages such as "ping ok", "ping failed" and "add store ok", 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`.

Table 3: Patterns in jEdit 4.3pre10

Pattern Group	Sup	Len	Type
Open and close "scope" before and after visiting a node, respectively (tokens: node, scope, close)	55	13	App
A `for` loop using `size` and `get` methods (tokens: get, size, property)	35	4	Impl
Beep if buffer is not editable (tokens: beep, get, toolkit)	34	6	App
Read and update a Hashtable (tokens: put, get, set)	29	5	Impl
Create GUI components from properties (tokens: add, get, property)	28	12	Impl

Table 4: Patterns in Azureus 3.0.2.2

Pattern Group	Sup	Len	Type
A loop for an array of objects (tokens: get, size, add)	320	10	Impl
Enter and exit a monitor before and after a loop, respectively (tokens: next, iterator, enter)	151	10	App
A loop using an Iterator (tokens: next, has, iterator)	151	9	Impl
Print stack trace if error (tokens: print, stack, trace)	140	7	Impl
Logging if enabled (tokens: log, is, enabled)	119	13	App

Table 5 shows the patterns in Apache Tomcat. Logging is the largest pattern group in this experiment; the most frequent pattern has 304 instances and there are 442 variant patterns in the group (6192 instances in total). This logging code is also hard to refactor because there are various messages for each location where Tomcat executes an important action. Executing a function in the privileged mode is also a difficult crosscutting concern to refactor since the concern requires appropriate implementation for each function as shown in Figure 3.

Table 6 shows the patterns in ANTLR. The top two patterns are unit testing patterns 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. ANTLR also includes several coding patterns to process the nodes of an abstract syntax tree.

Table 7 shows the patterns extracted from SableCC. All the patterns extracted from SableCC are to process a tree or a list. All patterns are not crosscutting concerns and short enough to understand what the patterns do.

Table 5: Patterns in Apache Tomcat 6.0.14

Pattern Group	Sup	Len	Type
Logging if debugging mode (tokens: debug, is, enabled)	304	24	App
Compare a string with string literals (tokens: equals, substring, println)	77	23	Impl
A `for` loop using Iterator (tokens: next, has, iterator)	64	7	Impl
Create a string from objects (tokens: append, length)	60	4	Impl
Execute a function in privileged mode if protection is enabled (tokens: protection, is, enabled)	46	4	App

```
public void getServletContextName() {
  if (SecurityUtil.isPackageProtectionEnabled()) {
    doPrivileged("getServletContextName", null);
  } else {
    return contet.getServletContetName();
  }
}

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 3: Privileged execution pattern in Apache Tomcat

4. DISCUSSION

Our technique does cover all code related to a concern if the concern is implemented using only one idiom. Our technique identifies sequential patterns that comprise a sequence of method calls and control statements. Sequential patterns involve idioms interleaving with other code fragments while sequential patterns may exclude some patterns that are not strictly ordered [1, 16].

Our approach easily identifies all code fragments related to a concern if the concern is implemented using the one idiom, e.g., the undo pattern in JHotDraw 5.4b1. In the case of the undo implementation, various method calls are inserted to the basic idiom to achieve undo actions for each command while the basic idiom is not changed.

Our approach detects implementation idioms as false positives. We regard well-known patterns, e.g. a `for` loop using Iterator, as false positives. We are investigating a way to automatically distinguish the well-known patterns from other patterns. Filtering the JDK classes might be a simple but effective approach since we have found few functional patterns using the JDK library.

Some of crosscutting concerns that are difficult to modularize may be false positives for developers who would like to modularize them using AspectJ. However, many industrial developers working without aspects are interested in these information for their software maintenance because developers who modify an idiomatic code fragment have to investigate the change impact and apply the same change to the other instances of a pattern if necessary. We are planning to generate documentation for such patterns from the result of our pattern mining. A promising tool is a set of predicates provided by SoQueT [20].

Table 6: Patterns in ANTLR 3.0.1

Pattern Group	Sup	Len	Type
Create code generators for unit testing (tokens: set, tool, code)	107	8	App
Parse the result for unit testing (tokens: equals, assert, error)	69	5	Impl
Loop for a list of objects (tokens: get, size)	48	4	Impl
Parse AST and report an error if necessary (tokens: LT, match, report)	38	11	App
Visit tree nodes for textual output (tokens: match, get, text)	29	8	App

Table 7: Patterns in SableCC 3.2

Pattern Group	Sup	Len	Type
Apply a function to an array of objects (tokens: apply, to, array)	72	7	Impl
Apply a function to a tree of objects (tokens: apply)	42	9	Impl
Read textual data and ID from nodes (tokens: get, id, text)	41	8	Impl
Add a node to a list (tokens: add)	33	4	Impl
Register the name of node to a map (tokens: get, text, put)	27	6	Impl

We have analyzed various programs in different domains. The programs upon which we validated our technique are suitable as a part of a common benchmark, because the target programs are selected from different domains and maintained by different development groups. JBoss, Eclipse, Apache Ant, WALA and Soot are candidates for our analysis in the future work. We are also interested in analyzing how the patterns are evolved in a development process.

5. RELATED WORK

Aspect mining techniques [5, 6, 17, 18] employ some heuristic functions to detect typical implementation of crosscutting concerns and apply refactoring to aspect candidates. We adopted a sequential pattern mining technique, which is not applied in aspect mining area yet. Although idiom-based code has variations [7], sequential pattern mining can detect idioms interleaving with other code fragments. Our approach may detect a partially ordered API usage [1] as several distinct sequential patterns. Instead, our approach can recognize the control structure such as IF and LOOP in patterns.

Some of crosscutting concerns we have detected are also detected by the fan-in analysis [18]. The difference is that our approach detects control structure in addition to method calls. Our approach also detects several patterns including co-located methods, e.g., open/close and enter/exit. Breu's history-based aspect mining focuses on extracting such co-located methods in a program from its software repository [5].

Since idiomatic code fragments are not explicitly modularized, developers often copy-and-paste a code block [14]. Such copy-and-pasted code fragments are known as code clones [3, 4, 10, 12, 16]. 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 [12]. 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.

As some long-lived code clones are *unfactorable* [15], some patterns we have extracted are unfactorable. The pattern information can be used in the documentation approach of SoQueT [20], FluidAOP-based code maintenance [11] and simultaneous modification [10] for maintenance of the patterns.

6. SUMMARY

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 both crosscutting concerns and implementation idioms. In the future work, we will investigate a way to automatically categorize the detected patterns, and to generate documentation for developers to understand coding patterns.

7. 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 Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp.25-34, 2007.

[2] Agrawal, R. and Srikant, R.: Mining Sequential Patterns, In Proc. of the International Conference on Data Engineering (ICDE), pp.3-14, 1995.

[3] Baker, B. S.: A Program for Identifying Duplicated Code. Computing Science and Statistics, Vol.6, pp.49-57, 1992.

[4] Baxter, I., Yahin, A., Moura, L., Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees. In Proc. of the International Conference on Software Maintenance (ICSM), pp.368-377, 1998.

[5] Breu, S. and Zimmermann, T.: Mining Aspects from Version History. In Proc. of the International Conference on Automated Software Engineering (ASE), pp.221-230, 2006.

[6] Bruntink, M., van Deursen, A., van Engelen, R. and Tourwe, T.: On the Use of Clone Detection for Identifying Crosscutting Concern Code. IEEE Transactions on Software Engineering, Vol.31, No.10, pp.804-818, 2005.

[7] Bruntink, M., van Deursen, A., D'Hondt, M. and Tourwé, T.: Simple crosscutting concerns are not so simple - analysing variability in large-scale idioms-based implementations. In Proc. of the International Conference on Aspect-Oriented Software Development (AOSD), pp.199-211, 2007.

[8] Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley, 1999.

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

[10] Higo, Y., Ueda, Y., Kusumoto, S. and Inoue, K.: Simultaneous Modification Support based on Code Clone Analysis. In Proc. of the Asia-Pacific Software Engineering Conference (APSEC), pp.262-269, 2007.

[11] Hon, T. and Kiczales, G.: Fluid AOP Join Point Models. In Proc. of the Asian Workshop on Aspect-Oriented Software Development (AOAsia), pp.14-17, 2006.

[12] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering, Vol.28, No.7, pp.654-670, 2002.

[13] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming. In Proc. of the European Conference on Object-Oriented Programming (ECOOP), pp.220-242, 1997.

[14] Kim, M. Bergman, L., Lau, T. and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In Proc. of the International Symposium on Empirical Software Engineering (ISESE), pp.83-92, 2004.

[15] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. C.: An Empirical Study of Code Clone Genealogies. In Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp.187-196, 2005.

[16] J. Krinke: Identifying Similar Code with Program Dependence Graphs. In Proc. of the Working Conference on Reverse Engineering (WCRE), pp.301-309, 2001.

[17] Krinke, J.: Mining Control Flow Graphs for Crosscutting Concerns. In Proc. of the Working Conference on Reverse Engineering (WCRE), pp.334-342, 2006.

[18] Marin, M., van Deursen, A. and Moonen, L.: Identifying Aspects using Fan-in Analysis. In Proc. of the Working Conference on Reverse Engineering (WCRE), pp.132-141, 2004.

[19] Marin, M.: Reasoning about Assessing and Improving the Seed Quality of a Generative Aspect Mining Technique. In Proc. of the International Linking Aspect Technology and Evolution Workshop (LATE), `http://aosd.net/workshops/late/2006/`, 2006.

[20] Marin, M., Moonen, L. and van Deursen, A.: SoQueT: Query-Based Documentation of Crosscutting Concerns. In Proc. of the International Conference on Software Engineering (ICSE), pp.758-761, 2007.

[21] Marin, M., Moonen, L. and van Deursen, A.: An Integrated Crosscutting Concern Migration Strategy and its Application to JHOTDRAW. In Proc. of the International Working Conference on Source Code Analysis and Manipulation (SCAM), pp.101-110, 2007.

[22] Pei, J., Han. J, Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.: PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. In Proc. of the International Conference on Data Engineering (ICDE), pp.215-224, 2001.

[23] Roy, C. K., Uddin, M. G., Roy, B. and Dean, T. R.: Evaluating Aspect Mining Techniques: A Case Study. In Proc. of the International Conference on Program Comprehension (ICPC), pp.167-176, 2007.