Towards Maintenance Support for Idiom-based Code Using Sequential Pattern Mining

Tatsuya Miyake, Takashi Ishio, Koji Taniguchi, Katsuro Inoue

Graduate School of Information Science and Technology
Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531 Japan
{t-miyake, ishio, kou-tngt, inoue}@ist.osaka-u.ac.jp

Abstract

Developers often use an idiom to implement a concern. When a fault is found in an idiom, developers have to find all source code fragments derived from the original. While code-clone detection tools can detect copy-and-pasted code, such tools cannot detect code fragments modified after pasted. We are investigating a sequential pattern mining approach to capture idiom-based code that spread across modules. This position paper shows our approach and the result of a preliminary case study.

1. Motivation: Idiom-based Coding

To develop a large scale software, developers use idioms to implement a particular kind of concerns that are not modularized in the software [9]. They obtain idioms from the source code of their software, the coding standard of their team and other available resources. This idiom-based coding leads many instances of an idiom crosscutting modules; a fault in an idiom is also copied and spreads across modules. When developers found a fault in an idiom, developers have to inspect all instances of the idiom to fix the problem [2, 3, 5].

Inspecting all instances of an idiom is time consuming since developers derive various code for each programming context from an original idiom. Although code clone detection tools such as CCFinder can find all copy-and-pasted code and some variants that are very similar to the original code [7], code clone tools cannot cover all derived code fragments that are no longer code clones. Developers may use other keyword search tools but developers are hard to assure that their inspection is completed.

Developers need a tool to find idiom-based code that are similar to one another and contribute to one specific concern. This problem is in the boundary area between code clone analysis and aspect mining since both code clone analysis and aspect mining techniques detect homogeneous code crosscutting modules to support software maintenance [4, 8].

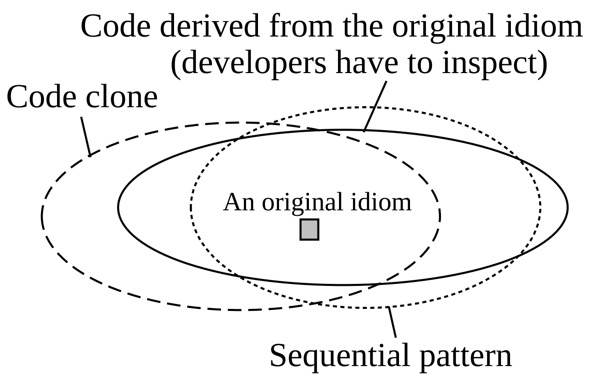
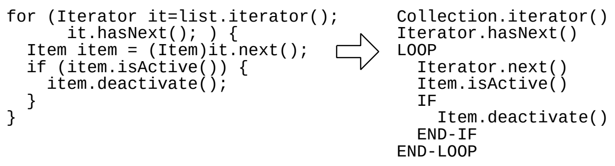

Code derived from the original idiom (developers have to inspect)
Code clone
An original idiom
Sequential pattern

Figure 1. Idiom-based code and code clone

We are planning to investigate the combination of code clone analysis, sequential pattern mining and other techniques to find all instances of an idiom of interest to developers as shown in Figure 1. As a first step of the research, we are investigating whether or not a sequential pattern mining approach detect frequent coding patterns based on idioms.

We have applied PrefixSpan [10], which is a sequential pattern mining algorithm, to JHotDraw as a preliminary case study. The result shows that the sequential pattern mining extracted several idiom-based patterns that implement crosscutting concerns such as undo functionality in JHotDraw.

We will combine our sequential pattern mining approach with code clone analysis to support developers to inspect all



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

```
Collection.iterator()
Iterator.hasNext()
LOOP
  Iterator.next()
  Item.isActive()
  IF
    Item.deactivate()
  END-IF
END-LOOP
```

Figure 2. A sequence extracted from source code

instances of an idiom. Our approach is related to Fluid AOP [6]; we focus on finding and managing crosscutting code rather than directly modularize crosscutting code as an aspect.

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 JHotDraw. Section 4 summarizes our current state and future directions.

2. Sequential Pattern Mining

Sequential pattern mining extracts frequent subsequences from a sequence database [1]. We applied PrefixSpan [10] to a sequence database extracted from Java software. We translated the source code of a Java method into a sequence that comprises method call, IF and LOOP elements since an idiom is a small code fragments including method calls and several control blocks. Figure 2 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 dynamic binding, an element has a reference to a class that declares the method in the class hierarchy. For example, a method call to String.equals(Object) and List.equals(Object) are not distinguished; a method call element Object.equals(Object) is generated for each call.

IF/ELSE/END-IF element An `if` statement is translated into a pair of IF and END-IF elements. If the predicate of the statement calls a method, a method call element corresponding the method call is inserted before the IF element.

LOOP/END-LOOP element `for` and `while` statements are translated into a pair of a LOOP and an END-LOOP elements. A method call in a predicate of the loop is translated into a method call element inserted before the LOOP element. We focus on the syntactic structure of a loop instead of precise control-flow information. We ignore `break`, `continue` and `return` statements in a loop.

Concern	Support	Class
Iteration (1)	54	31
Undo (1)	14	14
Undo (2)	12	12
selection of figures	10	10
selection of figures	9	9
Iteration (2)	8	8
Updating views	6	6
Handling mouse input	6	6
Manipulating polygons	6	1
Drawing image	6	1

Table 1. Patterns extracted from JHotDraw

PrefixSpan extracts frequent subsequences from the database for a Java program. Several extracted patterns form a group that shares the common set of methods. For example, a pattern "Collection.iterator, LOOP, Iterator.hasNext, Iterator.next, END-LOOP" implies shorter patterns such as "LOOP, Iterator.hasNext, Iterator.next, END-LOOP" and "Iterator.hasNext, Iterator.next". We aggregate these patterns to a single pattern group.

3. Sequential Patterns in JHotDraw

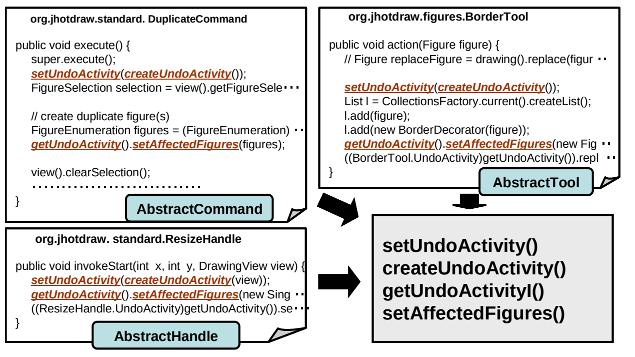
To evaluate whether sequential patterns can capture idiom-based code or not, we have conducted a preliminary case study. We applied the sequential pattern mining method described in the previous section to JHotDraw, that is a drawing application and well studied in AOP community. JHotDraw comprises 2,900 methods. Its total size is 18,000 lines of code.

We extracted method call patterns that involve at least 4 elements and have at least 4 instances in the program. As a result, we extracted 38 method call patterns.

We have manually investigated what concerns the extracted patterns implement. Table 1 shows the top 10 frequent patterns extracted from JHotDraw. A row represents a pattern. The first column `Concern` is the name of the concern that the pattern implements. The second column `Support` indicates the number of instances of the pattern. The third column `Class` shows the number of classes that involve one or more instances of the pattern in their methods.

3.1. Patterns of a Crosscutting Concern

We found 22 patterns related to crosscutting concerns in 38 extracted patterns. We recognized two features in these patterns.



```
org.jhotdraw.standard. DuplicateCommand

public void execute() {
    super.execute();
    setUndoActivity(createUndoActivity());
    FigureSelection selection = view().getFigureSele···

    // create duplicate figure(s)
    FigureEnumeration figures = (FigureEnumeration) ··
    getUndoActivity().setAffectedFigures(figures);

    view().clearSelection();
    ·····························
}
```

```
org.jhotdraw.figures.BorderTool

public void action(Figure figure) {
    // Figure replaceFigure = drawing().replace(figur ··

    setUndoActivity(createUndoActivity());
    List l = CollectionsFactory.current().createList();
    l.add(figure);
    l.add(new BorderDecorator(figure));
    getUndoActivity().setAffectedFigures(new Fig ··
    ((BorderTool.UndoActivity)getUndoActivity()).repl ··
}
```

```
org.jhotdraw. standard.ResizeHandle

public void invokeStart(int  x, int  y, DrawingView view) {
    setUndoActivity(createUndoActivity(view));
    getUndoActivity().setAffectedFigures(new Sing ··
    ((ResizeHandle.UndoActivity)getUndoActivity()).se···
}
```

Figure 3. Undo patterns

- A particular token frequently appears in the elements of a pattern. For example, a pattern for undo functionality has the method call elements getUndoManager() and pushUndo() that have a common token undo in their name.
- A common token also frequently appears in the name of methods that involve a pattern. For example, a pattern used to implement mouse handling concern appears in methods such as mouseUp and mouseDown that have the same prefix mouse.

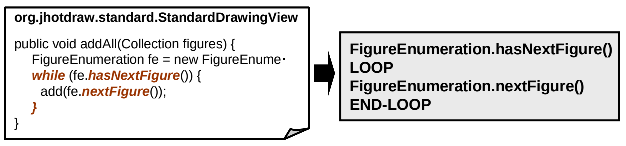
These features enables us to easily understand concerns implemented by idioms. These patterns are important information to maintain software. Figure 3 shows three undo implementation patterns in different classes: 11 subclasses of AbstractCommand, 6 subclasses of AbstractHandle and 9 subclasses of AbstractTool. Our approach successfully captured these patterns using the same idiom although the implementation detail for each class are different one another.

3.2. Patterns of Implementation Idioms

The rest of 16 patterns in 38 extracted patterns are implementation idioms, or general coding patterns that are not interesting in aspect mining research. Figure 4 shows a loop pattern that has 57 instances in JHotDraw. These patterns are less important than application-specific patterns but still useful information for developers inspecting idioms.

4. Summary and Future Direction

Sequential pattern mining found idiom-based implementation of crosscutting concerns. Although we need further case studies on other software systems, the result seems promising.



```
org.jhotdraw.standard.StandardDrawingView

public void addAll(Collection figures) {
    FigureEnumeration fe = new FigureEnume·
    while (fe.hasNextFigure()) {
      add(fe.nextFigure());
    }
}
```

FigureEnumeration.hasNextFigure()
LOOP
FigureEnumeration.nextFigure()
END-LOOP

Figure 4. A loop iteration idiom

The next step of our research is combining this approach and code clone analysis to detect all code fragments derived from an original idiom. This work will extend the experiment covering a concern using code clones [4] with the result of sequential pattern mining. We will investigate the trade-off between recall and precision since we would like to enable developers to assure that their code inspection is adequate to fix all defects caused by a fault in an idiom.

Another research direction is an aspect mining based on sequential pattern mining. Sequential patterns can capture an idiom that is crosscutting modules and interleaving with other code fragments. Distinguishing a crosscutting concern implementation from implementation idioms is also interesting.

References

[1] Agrawal, R. and Srikant, R.: Mining Sequential Patterns, Proceedings of ICDE 1995, pp.3-14.

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

[3] Baxter, I., Yahin, A., Moura, L., Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees. Proceedings of ICSM 1998, pp.368-377,

[4] 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.

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

[6] Hon, T. and Kiczales, G.: Fluid AOP Join Point Models. Proceedings of AOAsia 2006, pp.14-17.

[7] 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.

[8] Krinke, J.: Mining Control Flow Graphs for Crosscutting Concerns. Proceedings of WCRE 2006, pp.334-342.

[9] Marin, M.: Reasoning about Assessing and Improving the Seed Quality of a Generative Aspect Mining Technique. Proceedings of LATE 2006.

[10] Pei, J., Han. J, Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.: PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth. Proceedings of ICDE 2001, pp.215-224.