Clone Detection vs. Pattern Mining: The Battle

Céline Deknop and Kim Mens
UCLouvain
Louvain-la-Neuve, Belgium
{kim.mens, celine.deknop}@uclouvain.be

Simon Baars and Ana Oprescu
University of Amsterdam
Amsterdam, The Netherlands
a.m.oprescu@uva.nl and simon.mailadres@gmail.com

Johan Fabry
Raincode Labs
Brussels, Belgium
johan@raincode.com

Abstract

In this paper we compare two approaches to discover recurrent fragments in source code: clone detection and frequent subtree mining. We apply both approaches to a medium-sized Java case and compare qualitatively and quantitatively their results in terms of what types of code fragments are detected, as well as their size, relevance, coverage, and level of detail. We conclude that both approaches are complementary, while existing overlap may be used for cross-validation of the approaches.

Index terms— clone detection, pattern mining, frequent subtree mining, code clones, type 3 clones, duplicate code.

1 The Battle

Recurrent code fragments are often considered as symptoms of bad design [1]. They create implicit dependencies, thus increasing maintenance efforts or causing bugs in evolving software. Changing one occurrence of such a duplicated fragment may require other occurrences to be changed as well [2]. Also, duplicated code has been shown to add up to 25% of total system volume [3], which entails more code to be maintained. Unfortunately, due to the "tyranny of the dominant decomposition" [4], redundant crosscutting code fragments cannot always be avoided. Nevertheless, it remains essential to discover them, in order to better maintain, understand or evolve the software.

Several techniques have been proposed to detect recurrent code fragments. This paper compares two such approaches: *clone detection* and *pattern mining*. Clone detection entails that fragments of code are compared to each other line by line or statement by statement, in order to find similar fragments, often with certain groups of tokens being excluded from the match. Pattern mining, in particular frequent subtree mining, is the problem of finding all subtrees occurring frequently in other trees, in our case abstract syntax trees (ASTs), with a support that is above a given threshold.

Since clone detection is more lightweight than pattern mining, a relevant question is whether pattern mining is worth the additional effort, by providing alternative, richer or larger patterns, or whether it is largely redundant with respect to clone detection. To perform this comparison, we conduct a case study where we compare the approaches from two angles: to what extent do mined patterns correspond to detected clones, and how well do clones match to mined patterns?

For each direction we investigate what code fragments are found by one approach that are not found by the other. For those fragments found by both approaches we compare their coverage, size, and level of

In: D. Di Nucci, C. De Roover (eds.): Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop, Brussels, Belgium, 28-11-2019, published at http://ceur-ws.org

detail. This analysis allows us to make interesting observations regarding the similarity and differences of both approaches, and to draw conclusions regarding the strengths of either approach.

2 The Arena

To compare both approaches, we apply them to a medium-sized Java project: JHotDraw [5]. We selected JHotDraw version 7.5.1, which is part of the *Qualitas Corpus* [6], a curated collection of open-source Java software systems meant to be used for empirical studies of code artefacts [7]. JHotDraw is a two-dimensional graphics framework for structured drawing editors. It consists of 428 files and is known to make good use of design patterns [8]. Previous studies have used earlier versions of JHotDraw to look for recurrent code regularities [9, 10]. All this makes us confident that it is an interesting case on which to conduct our comparison.

3 The fighters

We now explain both approaches, their corresponding tools and used configuration, and provide some raw data and statistics on their results when applying them to JHotDraw.

3.1 Clone Detection

Code cloning is an active field of study: many detection techniques and tools were proposed [11]. Different clone types allow a different granularity of variance between cloned fragments [12].
Type 1 clones allow variance in structure and whitespace only.
Type 2 clones allow variance in identifier names.
Type 3 clones allow complete statements to vary.
We identify two useful concepts regarding code clones [12]:
Clone instance: A single cloned fragment.
Clone class: A set of similar clone instances, referred to hereafter as a clone.
We detect clones using our CloneRefactor tool[1]. This tool supports several clone type definitions, but for this study we only considered type 3 clones. We used the following detection settings for our comparison:
Minimum number of lines cloned: 3
Minimum clone class size: 5

With these settings we detected 136 clone classes each having 9.2 instances on average. Each of these instances span 6.8 lines on average. This makes up for a total of 8.559 out of 39.403 lines cloned (i.e., 21.7% of the system is cloned). About half of these clones are found in method bodies, other clones were found in constructors or exceed the boundaries of a single method. Detecting these results in the analysed JHotDraw system took 38.7 seconds on a Macbook Air (this is relatively fast compared to pattern mining).

[1]Available on GitHub: `https://github.com/SimonBaars/CloneRefactor`

3.2 Pattern mining

Our pattern mining tool is based on an extension of the existing FREQT tree mining algorithm [13]. As input it takes an abstract syntax tree (AST) representation of the source code, meaning that a mined pattern is an AST fragment that occurs frequently in the codebase. One of FREQT's limitations is that it tends to find too many patterns to be practical, and that many of them remain quite small. In order to be useful for mining code fragments in large codebases, we adapted the original FREQT algorithm with some dedicated constraints and with an additional step to try to grow the patterns found as large as possible [14]. More specifically, we use maximal frequent subtree mining to ensure that a condensed representation of large patterns is found, and we add the following additional constraints to the mining process:
C0 minimum support: for the experiment presented here, we used a value of 5;
C1 maximum size of the pattern: 4;
C2 minimum size of the pattern: 2;
C3 limit the set of labels allowed to occur in the root of patterns: Type Declaration and Blocks;
C4 forbid some labels to occur in patterns: Javadoc, annotations;
C5 limit the number of siblings in a pattern that can have the same label: 10;
C6 all leaf nodes in a pattern must have a label that can occur as a leaf node in the AST;
C7 discovered patterns may not miss any mandatory labels.

The threshold values of the different constraints above were determined experimentally by applying the tool on several Java cases and manually analysing the quality of the patterns mined. Since mining is quite computationally intensive, in order for our mining algorithm to finish within reasonable memory and time bounds, we have to split the codebase on which we work into separate folds, and run the miner on one fold at a time.[2] For JHotDraw, we created 4 folds, and found 156 patterns in total for the configuration above. The size of the AST fragments of the mined patterns varied between 15 and 207 nodes, with an average size of 36. The mining took 33 minutes on a middle grade tower PC. We noticed, not surprisingly,

[2]This does imply that some patterns that occur across folds may not be found, if their frequency within a single fold does not surpass the minimum support threshold. The less and larger the folds, the less this problem occurs.

that the bigger the pattern size, the more interesting it tends to be.

4 The Fight

We now describe how we compared both approaches and then report on the results of our comparison.

4.1 Methodology

We compared the results of both approaches through (1) *manual comparison*: the pattern mining team exhaustively went over all clones to look for possible matching patterns and (2) *automated comparison*: the clone detection team formalized an automated method to compare clones and patterns.

4.1.1 Manual comparison

Per clone class we manually investigated [3]

- the overlap and coverage (in #classes, #LOC, #occurrences/ class) when (at least) one pattern matches it;
- if there was a matching pattern, we also looked for similar patterns occurring multiple times;
- the relevance/usefulness/richness of the pattern for a software developer (this is a subjective criterion), in terms of a rating -/0/+.

4.1.2 Automated comparison

Using a script[4], we find for each pattern the clone class that is the most similar and vice versa. These results help analysing to what extent one approach is redundant over the other: if a large percentage of clones closely resembles most patterns (or vice versa) this would be the case.

In the automated comparison we first determine the locations of all 156 patterns and 136 clones. These locations consist of the file and range of each instance in a pattern/clone. For clones, this range is simply the begin and end line of a code fragment that exists elsewhere. To simplify the comparison, we chose to compare on a line-level and not take into account the begin and end column of ranges. For patterns, each separate AST node that belongs to the pattern (highlighted in orange in Fig. 3) has its own range (line and columns). For simplicity, we consider the range of a pattern instance to be the begin line of the first AST node in the pattern and the end line that of the last AST node in the pattern.

To compare the patterns and clones at these locations we define a similarity metric for clones and patterns. For each pattern, we search for the clone with the highest similarity and vice versa by comparing each clone instance of each clone class to each pattern instance of each pattern. We then use the following formula to determine the similarity percentage between a clone instance and pattern instance:

$$similarity(p_i, c_i) = \frac{2 * match(p_i, c_i)}{size(p_i) + size(c_i)} \quad (1)$$

where p_i is a pattern instance, c_i is a clone instance, *match* is the function that yields the number of matching lines between two instances and *size* is the function that yields the number of lines in a code fragment. If a pattern and a clone do not intersect, the result of *match* will be 0, meaning no similarity at all. Given a clone instance, we compute the instance similarity by seeking the pattern instance that maximizes the *similarity* function (so we find the pattern instance most similar to a clone instance):

$$instanceSim(p, c_i) = max(\{similarity(p_i, c_i) \mid p_i \in p\}) \quad (2)$$

where p is a set of pattern instances. Next, we calculate the similarity at the clone class level by summing the instance similarities of all instances in a clone class:

$$classSim(p, c) = sum(\left\{ \frac{instanceSim(p, c_i)}{size(p) + size(c)} * size(c_i) \;\middle|\; c_i \in c \right\}) \quad (3)$$

where c is a clone class. We compute which of the C=136 clone classes is **most similar** to a pattern p as the maximum similarity across all clone classes:

$$collectionSim(p, C) = max(\{classSim(p, c) \mid c \in C\}) \quad (4)$$

Finally, we collect per pattern the most similar clone class:

$$overallSim(P, C) = \{collectionSim(p, C) \mid p \in P\} \quad (5)$$

where P is the set of all 156 identified patterns. Using this method, we can find which pattern is most similar to a clone and vice versa (by using the respective symmetric relations).

4.2 Results of the manual analysis

4.2.1 Clone instances with no matching pattern

The first thing the pattern team noticed when exploring the clones was that out of 90 clones for which there was no matching pattern, 42 occurred in less

[3] You can find the full manual analysis here: `https://github.com/CelineDknp/CloneVSPatternAnalysis/blob/master/CloneVSPatternAnalysis.txt`

[4] Source code available at `https://github.com/SimonBaars/CloneRefactor/tree/master/src/main/java/com/simonbaars/clonerefactor/scripts/intimals`

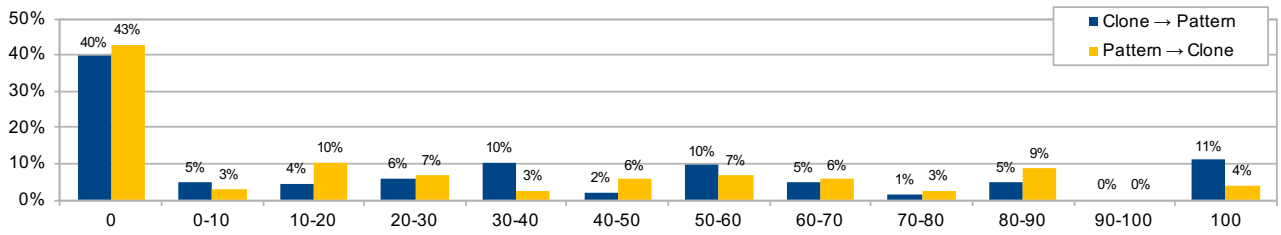

Figure 1: Percentage of clones/patterns that have a certain percentage of instances intersecting with each other. The x-axis shows the percentage of clones or patterns. The y-axis shows the percentage of instances that intersect with each other.

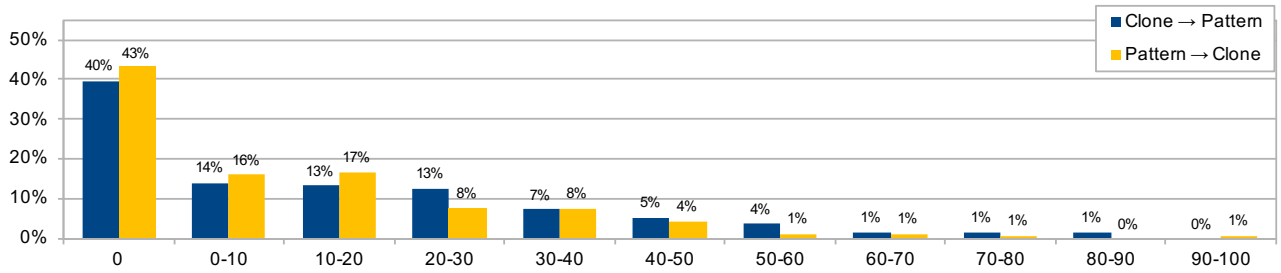

Figure 2: Distribution of similarity percentages for all clones and patterns. The x-axis shows the percentage of clones or patterns. The y-axis shows the similarity as defined in Sec. 4.1.2.

than five different code files. For example, the following code snippet occurs exactly 6 times in the `JMDIDesktopPane` file, but was not found as a pattern by the miner.

```
try {
    ((JInternalFrame)allFrames[i])
                                .setMaximum(false);
} catch (PropertyVetoException e) {
    e.printStackTrace();
}
```

We quickly realised that this behavior is caused by the C0 constraint (minimum support), which considers patterns occurring in at least five different *code files* rather than having five different occurrences. This is a result of the way the miner was configured to have a manageable amount of patterns, and concluded that it might not have been the best fit for this comparison. For future comparisons with code clone detection tools that often find multiple clone instances in a given file, we will need to change the configuration to avoid this issue.

A second reason why some clones are not matched by patterns, results from the fact that for the mining process to finish correctly, we need to divide the code base into smaller folds. But if there is a clone that spans over 5 files but that are not all in the same fold, the pattern may still be discarded because it would not have a high enough support value for either fold. We found 13 clones that did not match a pattern for this reason. (The mining team knew this limitation of their approach and is investigating how to improve their algorithm to work on larger data sets to avoid this problem.) The following code snippet is an example of such a code clone, repeated exactly 9 times in 6 different files, of which 1 file in fold 1, 2 files in fold 2 and 3 files in fold 4. So it was not discovered as a pattern in either of those folds.

```
protected void generateLookupTables() {
    radials = new float[w * h];
    angulars = new float[w * h];
    alphas = new int[w * h];
    float radius = getRadius();
    float blend = (radius + 2f) / radius - 1f;
    //clone ends here
```

The 35 other clones for which there was no matching pattern are either too small to fit the constraints of the miner, or cannot really be considered as a pattern. For example, some clones consist of the end of one function along with the start of another one, like in the following snippet.

```
//Start of function not in clone
 isClampRGB = b;
}

public boolean isClampRGBValues() {
    return isClampRGB;
//End of function not in clone
```

4.2.2 Matching clone instances and patterns

When analysing clone instances that did match mined patterns, the pattern mining team observed that we tend to have two types of matches.

A first type is where the pattern is broader, so that it includes multiple clones. For example, Fig. 3 shows pattern 46 of fold 4, which actually matches 4 different clone classes, one for each `case` block.

```
93      @Override
94      public void keyPressed(KeyEvent evt) {
95          RoundRectangleFigure owner = (RoundRectangleFigure) getOwner();
96          Point2D.Double oldArc = new Point2D.Double(owner.getArcWidth(),
    owner.getArcHeight());
97          Point2D.Double newArc = new Point2D.Double(owner.getArcWidth(),
    owner.getArcHeight());
98          switch (evt.getKeyCode()) {
99              case KeyEvent.VK_UP:
100                 if (newArc.y > 0) {
101                     newArc.y = Math.max(0, newArc.y - 1);
102                 }
103                 evt.consume();
104                 break;
105             case KeyEvent.VK_DOWN:
106                 newArc.y += 1;
107                 evt.consume();
108                 break;
109             case KeyEvent.VK_LEFT:
110                 if (newArc.x > 0) {
111                     newArc.x = Math.max(0, newArc.x - 1);
112                 }
113                 evt.consume();
114                 break;
115             case KeyEvent.VK_RIGHT:
116                 newArc.x += 1;
117                 evt.consume();
118                 break;
119         }
```

Figure 3: Pattern 46 of fold 4

A second type of match is when we have multiple patterns for a single clone class. In such cases, the clone usually corresponds to a specific part of multiple patterns that are similar, or to related patterns across folds. For example, the following code snippet occurs in our patterns 60, 78, 111 and 115 of fold 4.

```
@Override
public void transform(AffineTransform tx) {
    Point2D.Double anchor = getStartPoint();
    Point2D.Double lead = getEndPoint();
    setBounds(
            (Point2D.Double) tx.transform(anchor,
                anchor),
            (Point2D.Double) tx.transform(lead,
                lead));
}
```

Finally, using our rating of how relevant a developer considers a pattern, we observed that out of a total of 46 clones that where matched by a pattern, 41 were considered interesting, leading us to conclude that many of the results found by both methods can be considered as useful to developers.

4.3 Results of the automated analysis

Using the automated analysis method described in Section 4.1.2 we collected interesting statistical data on the similarity of the clones and patterns found by both tools.

When looking into the percentage of clones and patterns that intersect each other (see Fig. 1), we observe that 40% of clones do not match any patterns.[5] For 11% of the clones, all clone instances intersect with the pattern instances. 43% of the patterns do not intersect with any clones. For 4% of the patterns, all pattern instances intersect with the clone instances.

When manually inspecting the 11% of clone classes of which all instances intersect with patterns, we find that most of the clone instances in the clone classes are contained within a pattern. Patterns are often larger because they allow more variance in a fragment of code. Because of that, patterns often capture additional information surrounding a clone. This is why there is a large difference between clone to pattern and pattern to clone at 100% intersecting instances (see Fig. 1).

Fig. 2 shows how many clones and patterns are similar to each other. The same percentages of clones/-patterns that do not intersect can be found here as 0% similar. However, here we see the categories gradually decreasing towards 100% (no clone/pattern combination is actually 100% similar). By far, most clone/-pattern combinations are less than 50% similar: 92% of clones and 96% of patterns. This leaves only 8% of clones and 4% of patterns with a similarity higher than 50%.

5 The outcome

Before presenting our analysis of these results, we emphasize that this experiment was only an initial comparison between the two approaches. Further validation on other case studies, with improved settings, and by researchers other than the original tool creators, are required. Nevertheless, the results obtained already allowed us to reach some interesting insights.

5.1 Manual inspection

In cases where both approaches found similar code fragments, the overlap was not always complete. Sometimes one approach (often code cloning) found more fragments than the other, or (typically pattern mining) found larger or richer code fragments. Combining both approaches to complement each other's results would be an interesting research direction.

[5] This percentage would very likely become significantly larger if we would resolve the issue we encountered with the C0 constraint; requiring the patterns to occur in different code files.

5.2 Automated comparison

The results of the automated comparison show that, although most clones share some relation with some patterns, there are only very few clones where this relation is particularly strong. Often, patterns are found in completely different parts than where clones are found, or they briefly intersect instead of matching completely.

This shows that clone detection does, for the largest part, find different results than pattern mining. This indicates that both approaches are not redundant over one another, instead complementing each other.

6 Conclusion

As recurrent code fragments are often considered symptoms of bad design, several detection techniques have been proposed. Comparative studies across emerging techniques could shed further light into the trade-offs between time complexity and quality of results. We set out to compare two such approaches: *clone detection* and *pattern mining*, since clone detection seems more lightweight and a relevant question is whether pattern mining is worth the additional effort. Our automated comparison method involves several levels of data aggregation and is symmetric with respect to the comparison direction: patterns to clone classes and vice versa.

Our findings indicate that the two approaches are rather complementary. About half of the clones/patterns share no relation with each other. In the cases that clone detection and pattern mining are not 100% similar but do intersect, often the pattern is larger than the clone. This is because patterns allow for more structural variance in recurrent fragments than clone detection. The main reason that sometimes clones are larger than patterns is that patterns seem constrained to a single subtree, whereas clones can span several methods and even exceed class boundaries if they are all similar.

Acknowledgments

Part of this work was conducted in the context of an industry-university research project, funded by the Belgian Innoviris TeamUp project INTiMALS (2017-TEAM-UP-7).

References

[1] Martin Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley, second edition, 2018.

[2] Jan-Peter Ostberg and Stefan Wagner. On automatically collectable metrics for software maintainability evaluation. In *Proceedings of the 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, pages 32–37. IEEE Computer Society, 2014.

[3] Magiel Bruntink, Arie Van Deursen, Remco Van Engelen, and Tom Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[4] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119. IEEE, 1999.

[5] Erich Gamma and Thomas Eggenschwiler. Jhotdraw, 2004.

[6] Ewan Tempero. Qualitas Corpus, 2013. [http://qualitascorpus.com/; accessed 31-October-2019].

[7] Jolita Savolskyte. Review of the jhotdraw framework, 2004. Technical University Hamburg-Harburg.

[8] Henrik Bærbak Christensen. Frameworks: Putting design patterns into perspective. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '04, pages 142–145. ACM, 2004.

[9] Andy Kellens, Kim Mens, and Paolo Tonella. A survey of automated code-level aspect mining techniques. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development IV*, pages 143–162. Springer, 2007.

[10] Angela Lozano, Andy Kellens, Kim Mens, and Gabriela Arevalo. Mining source code for structural regularities. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, pages 22–31. IEEE Computer Society, 2010.

[11] Jeffrey Svajlenko and Chanchal K Roy. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 321–330. IEEE, 2014.

[12] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[13] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Sakamoto, and Arikawa Setsuo. Efficient substructure discovery from large semi-structured data. *IEICE Transactions on Information and Systems*, 04 2002.

[14] Hoang Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, Coen De Roover, Johan Fabry, and Vadim Zaytsev. Mining patterns in source code using tree mining algorithms. In Petra Kralj Novak, Tomislav Šmuc, and Sašo Džeroski, editors, *Discovery Science*, pages 471–480, Cham, 2019. Springer International Publishing.