Not All That Glitters Is Gold

Nils Göde

Universität Bremen
nils@informatik.uni-bremen.de

Abstract

Type-2 clones are duplicated passages of source code that have identical sequences of tokens (whitespace and comments are ignored) except for the textual representation of identifiers and literals—*parameters*. Commonly, type-2 clones are detected by simply abstracting from the parameters' textual representation. This normalization, however, frequently results in similar but otherwise unrelated code clones. We propose an alternative way to detect type-2 clones and compare the clones detected by our new approach to those detected by a traditional technique. Our results indicate that normalization of parameters is often inappropriate and should be applied with care.

1 Introduction

Code clones threaten the maintainability of a system as they increase the effort for change propagation and bear the risk of unwanted inconsistencies. The task of clone detection is to identify redundancy—clones—within the source code of a system. Clones are categorized according to the similarity of their token sequences. *Type-1* clones have identical token sequences disregarding comments and whitespace. *Type-2* clones are like type-1 clones but allow the textual representation of identifiers and literals (parameters) to be different. The token sequences of *type-3* clones may furthermore contain gaps—that is, some tokens are not found in all clones. For a general overview of clone research, please refer to [3, 4].

An important aspect of clone detection is that not all similar code fragments are relevant for software maintenance. Some code fragments just happen to be structurally similar although they are otherwise unrelated. We have observed such situations to occur frequently for type-2 clones that are detected by simply ignoring the textual representation of identifiers. In this paper, we present an alternative to detect type-2 clones avoiding frequent types of irrelevant clones. We compare the clones detected by our new approach to those detected by a traditional technique.

2 Type-2 Clone Detection

Type-2 clones are traditionally detected by normalizing identifiers and literals in the source code prior to the detection itself. Figure 1 shows an example of a type-2 clone as it might be reported using this approach. Even Baker's approach [1] that ensures a consistent renaming of identifiers would report this clone. From a maintenance perspective, however, this clone is rather irrelevant since the two code fragments are conceptually unrelated.

```
1  parse("language", l);
2  parse("input", i);
3  parse("length", len);
```

(a) First fragment

```
1  debug("#N:", nodes);
2  debug("#E:", edges);
3  debug("Time:", time);
```

(b) Second fragment

Figure 1: Irrelevant type-2 clone.

Our alternative way of detecting type-2 clones is based on the observation that the irrelevant clones detected by traditional techniques have a common property: They lack a longer sequence of identical tokens. Our improved approach consists of two steps. First we detect identical (type-1) clones of a given length l. Second, we merge two or more neighboring clones if the gaps between them are small enough. For details on this approach, please refer to [2]. The advantage of this two-step process is that all type-2 clones are guaranteed to have an identical token sequence of at least length l. This prevents detection of irrelevant clones as shown in Figure 1.

3 Study Setup

For our comparison, we chose the source code of JabRef[1] from January 1st, 2009. We excluded source files which were irrelevant to our analysis (for example, generated code). Then we ran simple type-2 detection (ignoring the textual representation of parameters) obtaining the set of type-2 clones C_S and our new approach giving us the type-2 clone set C_N. For both approaches, we used 50 tokens as minimum clone length. For our new technique, we set the minimum length of identical subsequences $l = 5$ tokens. To compare both approaches, we analyzed the differences between the clones detected by the simple approach C_S and the clones detected by our new technique C_N.

[1] http://jabref.sourceforge.net

```
1  int piv = 0;
2  for (Iterator<BibtexEntry> i =
3       set.iterator();
4       i.hasNext();) {
5    BibtexEntry entry = i.next();
6    idArray[piv] = entry.getId();
```

(a) First fragment

```
1  int i = 0;
2  for (Iterator<File> iterator =
3       files.iterator();
4       iterator.hasNext();) {
5    File file = iterator.next();
6    fileNames[i] = file.getAbsolutePath();
```

(b) Second fragment

Figure 2: Type-2 clone in iterations.

```
1  inputPanel.add( fieldScroller ) ;
2  con.fill = GridBagConstraints.HORIZONTAL ;
3  con.weighty = 0;
4  con.gridwidth = 2 ;
5  gbl.setConstraints( radioPanel, con ) ;
6  inputPanel.add( radioPanel ) ;
```

(a) First fragment

```
1  wordPan.add(removeWord);
2  con.anchor = GridBagConstraints.WEST;
3  con.gridx = 0;
4  con.gridy = 0;
5  gbl.setConstraints(fieldNameField, con);
6  fieldNamePan.add(fieldNameField);
```

(b) Second fragment

Figure 3: Type-2 clone in GUI setup.

4 Results

The first thing to note is that $C_N \subset C_S$, because every clone detected by our new approach is also detected by the simple technique which is less restrictive. Hence, we concentrated on the clones that are no longer detected using our new approach ($C_S \setminus C_N$). In total, the number of clones was reduced by 48%.

We manually investigated and categorized each such clone according to its structure. In total, we identified 11 categories. Note that a clone may have characteristics of more than one of the following categories. We have categorized each clone according to its predominant characteristic. Hence, our judgment is a potential threat to the validity. The percentage for each category indicates how many clones belong to it.

1. Sequences of **method calls** (34.2%). Figure 1 provides an example.
2. Sequences of `new`**-instantiations** (22.3%).
3. **Other** clones without predominant characteristic (8.9%).
4. Sequences of variable **declarations** (7.8%).
5. Sequences of **assignments** (7.8%).
6. Sequences of statements used to configure **GUI** elements (for example, a layout manager) (6.3%). An example is given in Figure 3.
7. **Iteration** loops and associated statements (5.6%). Figure 2 shows an example.
8. Definitions of **anonymous classes** (3.3%).
9. Sequences of `if`**-statements** (2.2%).
10. Declarations of **arrays** (0.7%).
11. Concatenation of **string literals** (0.7%).

In summary, we found that for roughly 90% of all inspected clones, there is no reasonable abstraction. Most of them are of sequential nature like the clone shown in Figure 1. Again, our subjective judgment is a potential threat to the validity.

The remaining 10% are *false negatives* that should have been detected. Most of these clones were close to the minimum length and had parameters at the front and back. Our new approach would detect only the identical middle sequence, which is in itself shorter than the minimum clone length. Decreasing the minimum clone length, however, would again increase the number of clones that are detected but irrelevant. Further research could be directed at finding parameters that provide the best trade-off.

5 Discussion

The results suggest that our new approach avoids many irrelevant clones. Another alternative would be applying simple type-2 detection first and then excluding type-2 clones based on the frequency of parameters. However, we have experimented with different measures and found that they cannot reliably be used to identify irrelevant clones. This is due to a number of relevant type-2 clones containing even more parameters than irrelevant type-2 clones.

6 Conclusion

Our conclusion is that our new approach of detecting type-2 clones is a reasonable alternative to the simple technique. It increases precision, as it avoids many clones that are irrelevant and for which there is no common abstraction. We conclude that naïve normalization of parameters is often inappropriate and should be applied with care.

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering*. IEEE, 1995.

[2] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *International Conference on Software Engineering*, 2011. Accepted for puplication.

[3] R. Koschke. Survey of research on software clones. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, 2007.

[4] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queens University at Kingston, Ontario, Canada, 2007.