University of Amsterdam

Software Evolution - Clone Detection

CLource: Visualising the Evolution of Code Clones

December 16, 2019

Students:
Lars van Hijfte
11291680

Rick Watertor
11250550

Lecturer:
Riemer van Rozen

Course:
Software Evolution

Contents

1 Introduction **2**

2 Code Clone Detection **2**
2.1 Detecting Clones . . . 2
2.1.1 Method Based . . . 3
2.1.2 Normalize Blocks . . . 4
2.1.3 Strip Identifiers . . . 4
2.1.4 Retrieve Statements . . . 4
2.1.5 Resolve Overlaps . . . 5
2.1.6 Statistics . . . 5
2.2 Threats to Validity . . . 6

3 Visualisation **6**
3.1 Motivation . . . 6
3.2 Git Usage . . . 7
3.3 Report . . . 7
3.4 Gource . . . 7
3.4.1 Implementation notes . . . 8

4 Results **8**

5 Conclusion **10**

1 Introduction

When developing for large software systems, it can happen that a developer reuses the same pieces of code by implementing it twice in the system. These similar pieces of code are called *code clones*. Clones have a range of positive and negative effects: they can improve evolvability, and its understandability [1]. It can also have negative effects, however: clones evolve inconsistently, they are indicative of lower maintainability, and code tends to couple around clones [2].

Depending on the definition and implementation of the detection, estimates are that, for large systems, between 7% and 23% of the lines of code are clones [3].

To help remove these clones, there are already many visualisation tools made. Some examples include: Bakers program [4], CCFinder [5], and CP-miner [6]. However, these tools only show the current state of a software system.

We introduce our tool, called CLource, to not only visualise the current state, but also the history of a software system. This creates a better understanding of when, how and by whom clones were created. Furthermore, it may help in creating a better understanding of how clones develop during the evolution of a software system.

2 Code Clone Detection

Clone detection has been done a lot in many different ways. However, there arose four broad types of when code is considered a clone. These are the following [7]:

- Type 1: Identical code fragments.
- Type 2: Token-based identical code fragments. Here identifiers and literals are removed.
- Type 3: Syntactically identical code fragments. Might have a slight change at statement level.
- Type 4: Semantic similarity. Code will have the same result but might come to the solution differently.

To detect these type of clones there are (for each one) also a lot of different approaches [8]. These can be grouped into text-based, token-based, tree-based, metrics-based, and graph-based approaches. Our implementation uses the tree-based approach using an *Abstract Syntax Tree* (AST). This enables the tool to easily modify parts of the code, which is used for type 2 code detection. Furthermore, the AST inherently does not include any whitespace and comments which should not be used in clone detection.

Our implementation can detect clone types 1 and 2. This section will describe how the detection of the clones is done, and after that discuss the threats to validity with our tool.

2.1 Detecting Clones

Our implementation tries to find all code clones using the AST of a software system. An overview of how this is done can be found in Figure 1. Extra explanation for each non-trivial part is stated below.

First, the AST is transformed to the type of clones that should be detected, and optionally normalized blocks. After that, it uses the sliding window technique on all executables[1] to group sections of identical code together, coupled with their source-code location. After this is done, it has a list of all sections (i.e., slices of '6' executables long) and their respective locations. When a section has multiple locations, it is considered duplicate code. However,

[1] Using the definition provided by Oracle: `https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/reflect/Executable.html`

if the clones are longer than the window used, it would register each new line as a different code 'class'. For this reason, all overlapping sections are merged. Each step will be explained in more detail in the following subsections.

```
// Get the whole codebase of a software system as abstract syntax tree
AST := getAST(softwareSystem)
// if user wants to ignore the blocks (these are by default normalized)
if (ignore_blocks) {
    // add blocks to bodies of statements that do not yet have a block
    AST := addBlocks(AST)
    // remove blocks that are not direct bodies of other statements
    AST := removeBlocks(AST)
}
if (type2) {
    // strip prefix, types, and literals
    AST := stripIdentifiers(AST)
}
map[Section, list[Location]] allSections := {}
if (methodBased) {
    trees := allMethods(AST)
} else {
    trees := allClasses(AST)
}
for (tree in trees){
    // Get all statements in chronological order
    statements := getStatements(tree)
    // Go over all statements in a window of default 6
    for (section in getSections(statements, 6)) {
        if (section in allSections) {
            // Add location as another location to section
            allSections := updateSection(section, allSections)
        } else {
            // Add location to section
            allSections := createSection(section, allSections)
        }
    }
}
// Get all sections that have multiple locations
clones := multipleLocations(allSections)

// If some sections overlap other sections, merge them together
clones := resolveOverlaps(clones)
```

Figure 1: Pseudocode of the implemented algorithm for duplicate code detection

2.1.1 Method Based

From Figure 1 it can be seen that our implementation can be both method based and class based. By default, this is method based as clones inside a method are more likely to be actual clones. Especially in Java, class based clone detection can generate a lot of clones unfairly. Say two classes have three simple getters next to each other, especially in type 2, this would always be considered a duplicate as it is a section of 6 consecutive lines of duplicate code (ignoring the method names and field names). For this reason, our implementation only looks at the bodies of methods. However, as this deviates from the definition, we also added the possibility to turn this off and look at the full class. Both approaches are fully covered

in our testing suite, and should therefore be correct.

2.1.2 Normalize Blocks

Although moving to the AST already removes a lot of code styling and normalizes most of the layout, it does not normalize blocks. Because of this, the code shown in Figure 2 would not be considered clones. This could indeed be seen as not identical, however, our implementation still provides the user to choose to also normalize these blocks. It does this by first adding blocks to all bodies of statements that do not already have one, and then removing blocks that do not add any functionality or have any semantic value.

```
1   // Without blocks
2   if (number == 0)
3       return number;
4   if (number == 1)
5       return number;
6   else
7       return fibonacci(number - 1) + fibonacci(number - 2);
8
9   // With blocks
10  if (number == 0) {
11      return number;
12  } if (number == 1) {
13      return number;
14  } else {
15      {   // Excessive block
16          return fibonacci(number - 1) + fibonacci(number - 2);
17      }
18  }
```

Figure 2: Example of where normalizing blocks would influence the results

2.1.3 Strip Identifiers

As stated in the beginning of Section 2, Type 2 clone detection is defined as "token-based identical code fragments, with identfiers and literals removed". This means that when we comparing sections of code (our windows of 6 long), all identifiers and literals need to be dismissed. Due to this, our implementation strips these before the search for clones. As identifiers are stored as strings in the AST, it removes all strings. This also results in all literal strings and numbers being set to an empty literal as those are also stored as strings in the AST. Number prefixes (- and +, before a literal number) are removed, as we consider these part of the literal number. Furthermore, all type names, method calls, method declarations, class names, and other identifiers, are also removed, as we consider these part of the identifiers.

2.1.4 Retrieve Statements

In order to use a sliding window technique, it needs to retrieve a list of all consecutive statements in a method. It does this by traversing over all the statements in the AST depth-first and adding each statement to a list.

To preserve the indentation level of each statement (to, for example, distinguish statements within a for loop or outside a for loop), it also keeps track of the relative indentation level compared to the next statement. All statements have by default an indentation level of 0. If a statement is at the end of a block, the statement's indentation level is decremented

```
// code                    list representation (statement : indent level)
for(int i = 0; i < 5; i++) { // \for() : 1
    System.out.println(i); // \expressionStatement() : 0
    System.out.println(0); // \expressionStatement() : -1
}
```

Figure 3: Visual representation of the mapping of source code to our statement representation.

by one. For statements that have something in its body, it sets the indentation level to 1. An example can be found in Figure 3.

Lastly, blocks and empty statements are not added to the list. In our opinion, these should not be considered an individual statement. This is because they at function only as structure statements, and do not add any instructions on the CPU level.

2.1.5 Resolve Overlaps

After all the clones with a length of the window size are found, it tries to resolve any overlap between them. If a clone is longer than the window size, it would detect and mark every window size long consecutive code within the clone as a clone. This is not user-friendly, as this would mean that one line of code would be redundantly marked as a clone many times (and consecutively show up in a visualization as multiple problems to be resolved). To prevent this, we applied a resolution strategy to merge overlapping clones.

To understand our strategy, the input format of this strategy will need some discussion. As input, the algorithm receives a list of lists of `loc` types in Rascal (these types map back to the original source code location). This data type can be seen as a list of all 'duplication classes', with a list of locations that each describe where an instance of the clone class is present.

First of all, the merging algorithm resolves self-overlap. Consider the scenario of a single line repeated more than the window size. With our algorithm, two or more clones would be found: from 0 to `windowSize` and from 1 to `windowSize+1`, and so on. The self-overlapping resolution strategy checks if within a duplication class directly adjacent windows overlap, and if so, merges them into one window. This results in one larger clone, containing the duplicate lines. We perform this resolution strategy until fixed-point, to ensure no more simplification steps can be taken.

If we can then find a `loc` object in a different duplication class that is directly adjacent or overlapping with our current class, we can merge the two classes into one class. However, this merging can only happen when there is overlap for *all* entries in a duplication class. A clone can only be duplicated with a same-sized code (disregarding type 3 and 4), and therefore the overlap should be present for all. Once it has found such a match, it combines the original two duplication classes into one bigger duplication class.

After all original classes are checked, it repeats the merge algorithm until a fixed-point is reached, indicating that no further merges are possible. After this, it has a list of all clones with their respective locations, simplified to the most comprehensive clones possible (and therefore clones longer than 6 lines can be found).

2.1.6 Statistics

Finally, to present some statistics and information about the clones, it determines the number of (raw) lines duplicated, percentage of duplicated lines, and the number of clone classes found.

To determine the number of raw lines duplicated, it takes all the locations of each clone, finds their starting line and ending line, and creates a set of all pairs "uri" and "line". This combination results in a per-file unique set of duplicated lines, and if any other duplication class includes the same lines in the same file, they are automatically filtered (due to the set property). Therefore we reach an accurate count of raw lines that are duplicated. That is, not stripped of comments, whitespace, or otherwise altered source code.

Alongside the duplication fragments, their source files and line numbers are embedded in the output files that are used for the visualisation.

2.2 Threats to Validity

The first threat to validity our implementation contains, has to do with the final closing bracket of the body of statements. If the code shown in Figure 4 would be a clone, this implementation would not count the closing bracket as a line of that clone. This is because the closing bracket is not considered as a statement on its own, and thus is not counted (if it falls after the final statement). The impact of this should be minimal, as it does not prevent duplications from being found. It does have an impact on the resulting number of raw clone lines found, but this is at most one line per clone instance.

```
1    int i = 1;
2    if (i < 5)
3        i += 10;
4    int j = 10;
5    for (int k; k < 10; k += i) {
6        j += k;
7    }
```

Figure 4: Clone with a closing bracket at the end

The second thread comes with type 2 clones. The implementation for type 2 strips away all strings. This is useful for stripping away all literals. However, we do not know when Rascal uses strings to also represent other stuff. One example we found was that when assigning, for example: `i = 3`, the sign is also stored as a string. This means that `i = 3` would be considered the same as `i += 3`. This should not be the case. However, due to time constraints, we could not detect all cases, and exclude those when stripping away strings. From our relatively comprehensive testing suite, we only found this as part of one clone instance. In real-life code this phenomenon may be more prevalent, but we do not have the data to back this up, or to deny this.

3 Visualisation

The visualisation of CLourse is done in two steps. First, it creates a Git repository containing all the clones, whereafter it uses Gource to visualise this repository. This section will explain why this is done, and further describe what is visualised.

3.1 Motivation

Software systems can become very complex. Therefore, when there is a lot of code duplication in a project, simply emitting the number of clones (and other metrics) is not enough: it is hard to find where and why code is duplicated. A programmer has to know how to resolve these problems, and therefore should be equipped with the proper tools to battle the issues in their code base.

This is where the field of Software Visualization can help. The field of Software Visualization mainly concerns itself with displaying complex software systems, but the same concepts can be applied to display complex software systems with metrics embedded into them.

For example, Noack's visual graph clustering model [9] can visually help the human viewer infer properties about the system that is being visualized: such as finding well-distributed nodes, and finding well-separated clusters.

Visualizations based on software history are further supported by the work of Gračanin et al. [10], who discuss in a section the advances in version history visualization. They note that manual inspection of metrics and changes on a software system are labor-intensive, and that visualization can help here.

The field of visualization is also supported by non-scientifically developed tools, such as such as Gource [2], which help visualize a software system over time, but its github reputation shows that it is a well-known tool for vizualizing version control repositories.

3.2 Git Usage

To create a better insight into the history of a software system, CLource works with Git[3]. This means that the visualisation only works for Git projects. However, as of writing 71% of opensource projects found on OpenHub are written using Git [4]. This, plus the growing trend in Git usage, caused us to choose Git, as it seems to be the most widely adopted tool for controlling versions in software systems. Additionally, additional launcher scripts can be created to support different types of Version Control Software.

Our tool creates a complimentary Git repository with similar commits and the same file-directory structure as the respective software system. Only the contents of each file is different. For each commit in the history of the software system each file contains the clones present at that commit together with the other files that the clone can be found in (i.e., each clone instance has a reference to the other locations that share the same clone class). The commit message is set to the respective commit id[5] together with the statistics, as described in Section 2.1.6, and the original message of the commit. The author of this 'fake commit' is set to the same author as the original commit, to properly visualize the actors on the repository in the visualization.

3.3 Report

During the search for clones per commit, the same statistics that are posted in the commit message are also reported back to the user via the terminal. These statistics include: the amount of duplicate code, number of clones, biggest clone, number of clone classes, and the biggest clone class. Furthermore it shows three example clones which are not inserted into the commit message itself. These example clones are randomly determined, by picking up to three random indices in the clone pool and presenting them as plain text.

3.4 Gource

After the Git repository is created, all different types of Git tools can be used to analyse the clones. Eazybi[6], for example, can be used to view the changes over time. Or Git-stats[7] can be used to see which contributors created the most amount of clones. We specifically

[2] `https://github.com/acaudwell/Gource`
[3] `https://git-scm.com/`
[4] `https://www.openhub.net/repositories/compare`
[5] In git, this is a commit hash.
[6] `https://eazybi.com/integrations/git`
[7] `https://github.com/IonicaBizau/git-stats`

looked into Gource[8]. This is a film based demonstration tool for visualising the history of software systems [11]. It shows the file-directory structure changes over time.

This in itself would already give some indication on when which author made clones. However, we extended on this by improving the colours used, and showing more information per file, and allowing the user to view the statistics from the commit message. Lastly, we added the possibility, for anyone running the tool, to open vim[9] directly where the clones are in the project. These modifications can be found at `https://github.com/Larspolo/Gource`.

3.4.1 Implementation notes

Gource by itself uses the 'git log' to speed up their program. It reads in all previous commits and files by using the git log command, and from there it can display all files easily. We, however, needed the content of the files, as that is where we wrote the information on the sizes of the clones, the locations of the clones, the references to the other instances of a clone class, and other meta-information.

Since Gource traverses a git history, we also needed to traverse the git history physically in the file-structure. Everytime a file is clicked, the target file is 'checked-out' in git, to show the file at the time of the commit. Similarly, every time a file is hovered over, information about that file is retrieved as part of the commit that is present at the time in Gource. Finally, every time a change is made (as part of the commit), the difference in duplications has to be determined. For this we load in the new file from the new commit, and read its contents to determine the new duplication amount. We then update the file colour accordingly. These actions can slightly impede on performance (and greatly increases startup times), but the final tool is effective, and seems to perform well enough to be user-friendly.

4 Results

This section contains screenshots of CLource. For these screenshots CodeArena[10] was used as the project where the clone detection was run on.

[8] `https://github.com/acaudwell/Gource`
[9] `https://www.vim.org/`
[10] `https://github.com/SimonBaars/CodeArena/`

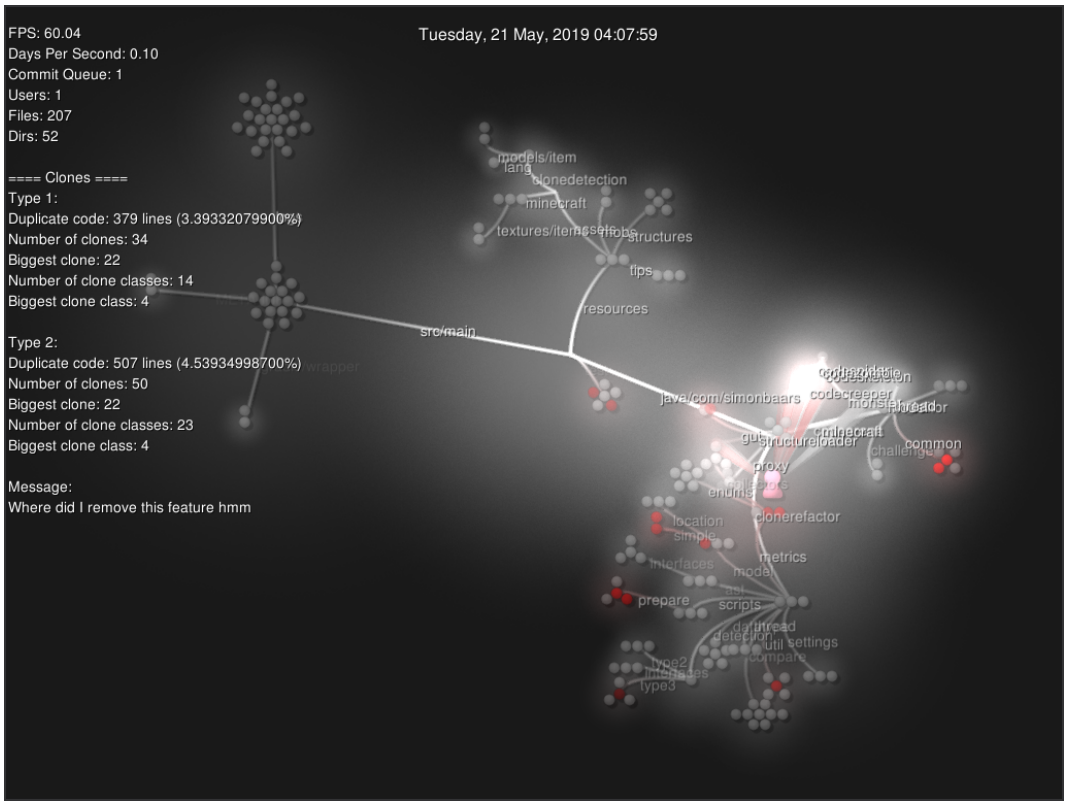

Figure 5: Each node is a file. Edges are how the files are connected in the file structure. Grey nodes (also files) do not include any clones, while red nodes do contain clones. The brighter the red, the more clones it contains. Furthermore, on the left are the statistics.

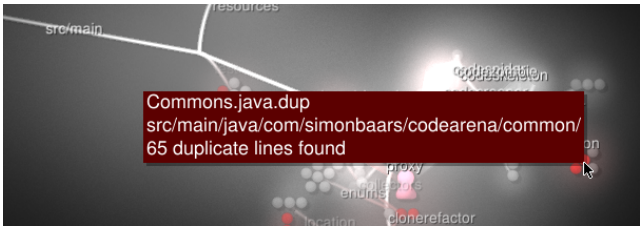

Figure 6: When hovering over the files, it shows the filename together with the path and the amount of duplicate lines.

Figure 7: When clicking on a node, a terminal opens the first detected clone within that file. It does this by opening vim with tabs for each of the files that the clone is found in. Additional clone information is opened in a separate terminal tab.

5 Conclusion

CLource is a tool that can be used to visually indicate not only the current Type 1 and Type 2 clones that exist in a project, but can also give insightful information in the evolution of said software system. It does this by both creating a Git repository where the history of all the clones is tracked. On this git repository, various git visualization tools can be used. We specifically adapted and modified Gource to visualise the repository in an interactive and time-based manner.

Future research

Although this tool works and is able to visualise the evolution of clones within a software system, there are also still some limitations. This tool currently works with only Type 1 and 2 clone detection. This does not limit the visualisation, but it could be useful to the end-user to also detect type 3 clones within CLource. Furthermore, the tool can still be optimized. Currently, a separate script is running for each commit individually, and therefore computes the clones for the entire program every commit. This can be optimized by only using the actual changes in the commit history. Lastly, we still noted some minor threats to validity that need to be fixed to have an optimally working tool.

References

[1] Cory J Kapser and Michael W Godfrey. ""Cloning considered harmful" considered harmful: patterns of cloning in software". In: *Empirical Software Engineering* 13.6 (2008), p. 645.

[2] Elmar Juergens et al. "Do code clones matter?" In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 485–495.

[3] Chanchal K Roy, James R Cordy, and Rainer Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". In: *Science of computer programming* 74.7 (2009), pp. 470–495.

[4] Brenda S Baker. "A program for identifying duplicated code". In: *Computing Science and Statistics* (1993), pp. 49–49.

[5] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670.

[6] Zhenmin Li et al. "CP-Miner: Finding copy-paste and related bugs in large-scale software code". In: *IEEE Transactions on software Engineering* 32.3 (2006), pp. 176–192.

[7] Vaibhav Saini et al. "Oreo: Detection of clones in the twilight zone". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, pp. 354–365.

[8] Chanchal K Roy and James R Cordy. "Scenario-based comparison of clone detection techniques". In: *2008 16th IEEE International Conference on Program Comprehension*. IEEE. 2008, pp. 153–162.

[9] Andreas Noack. "An energy model for visual graph clustering". In: *International symposium on graph drawing*. Springer. 2003, pp. 425–436.

[10] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. "Software visualization". In: *Innovations in Systems and Software Engineering* 1.2 (2005), pp. 221–230.

[11] Andrew H Caudwell. "Gource: visualizing software version control history". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2010, pp. 73–74.