# ClonEvol: Visualizing Software Evolution with Code Clones

Avdo Hanjalić
Department of Computing Science
University of Groningen, the Netherlands
E-mail: a.hanjalic@student.rug.nl

***Abstract*—We present ClonEvol, a visual analysis tool that assists in obtaining insight into the state and the evolution of a C/C++/Java source code base on project, file and scope level. ClonEvol combines information obtained from the software versioning system and contents of files that change between versions; The tool operates as tool-chain of Subversion (SVN), Doxygen (applied as static analyzer) and Simian as code duplication detector. The consolidated information is presented to the user in an interactive visual manner. The focus of the presented tool lies on scalability (in time and space) concerning data acquisition, data processing and visualization, and ease of use. The visualization is approached by using a (mirrored) radial tree to show the file and scope structures, complemented with hierarchically bundled edges that show clone relations. We demonstrate the use of ClonEvol on a real world code base.**

***Index Terms*—Software Visualization; Software Evolution Analysis; Code Clones**

## I. Introduction

Usage of software versioning and revision control systems such as SVN, GIT, Mercurial are common practice nowadays. In the corporate world these are known as Software configuration management (SCM) systems. These systems offer a vast amount of information that can help to understand (the evolution of) a source code base. The amount, location and span of code clones are a reliable measure when assessing quality of (the source code of) a software project. Tools, such as *Code Flows* [9] and the *Solid* toolset* [7], can be used for mining of facts from an SCM, such as file and project-level evolution, (high-level) code structure and code duplicates. However, not many tools exist that combine SCM versioning information with code clones for the analysis of clone evolution.

We present the tool ClonEvol, that assists in obtaining insight into the state and evolution of a C/C++/Java code base on project, file and scope level. This is achieved by combining information obtained from the software versioning system and contents of files that have changed between versions. More precisely, the tool combines the version change-logs with static analysis (of file contents) and clone detection. The consolidated information is presented to the user in a visual and interactive manner.

The focus of the presented tool lies on *scalability* (in time and space) concerning data acquisition, data processing and visualization, *genericity* and ease of use. *Scalability* is achieved by limiting data acquisition and fact extraction to differences between code base versions. ClonEvol in theory supports all languages that are supported by both Doxygen and Simian, which assures *genericity*.

The visualization is achieved with a mirrored radial tree to show the file and scope structures, complemented with hierarchically bundled edges that indicate the clone relations. Users can scroll through time to search for events of interest, which are highlighted by the following three color-maps:

- The *structure* color-map shows object types in the code base (files, classes, functions) and the existing clones. It is used as first overview to help the user understand the visualization of the project.
- The *difference* color-map can be used to visualize raw changes in the files and scopes, performed in a range of consecutive code base versions. Implications of the changes, such as added, removed and persisting code clones are visualized. Moreover, code drifts, splits and merges, being indicators of code refactoring, are emphasized.
- The *activity* color-map emphasizes frequently changed files and the related clones, for the purpose of identifying tightly coupled code and/or stubborn clones that form a sore spot for maintenance.

The color-maps can be used in this order to obtain insight into the dependencies and evolution of an unknown code base, or in inverse order to better understand the impact of a certain effort on a known code base.

Section II describes the design of the tool-chain and related data structures. Section III illustrates our tool with results from a real-world C/C++ code base. The paper ends with a short discussion of the results and remaining work in Section IV.

## II. Architecture

Code Flows is a method to visualize the evolution of source code, geared to the understanding of fine and mid-level scale changes across several versions [9]. We adopt here a similar approach, but take several design decisions to make our tool scale to real-world code bases that contain thousands of versions and source-code files. The data mining part of ClonEvol is a tool-chain based on open source applications Doxygen [11] and Simian [4]. Fig. 1 depicts the high-level process.

The visualization pipeline is complemented by a *shared repository* (DataStore), where the output of each "pipe" is

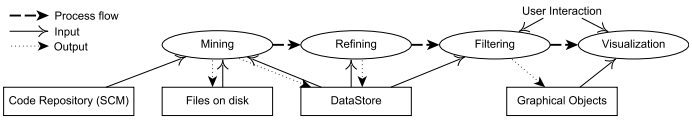


Fig. 1: Visualization Pipeline

stored rather than passed forward. Each *pipe* requires all of the mined data, therefore it would not make sense to forward it each time. This design follows the *pipes and filters* and *shared repository* architectural patterns [2]. We next detail the DataStore, data mining and data refinement processes, and filtering and visualization components of our tool.

### A. DataStore

The DataStore contains revisions $R$, that consist of four components; Each revision is a set $R_n = \{F, S, CC, SC\}$, where $F$ is the FileTree of modified, added and deleted files, $S$ is the ScopeTree containing scopes $s \in f \in F$, $CC$ contains sets of related code blocks so that $\{(f \in F, line_{start}, line_{end})\} \in cc \in CC$ and last $SC$ contains scope-clone relations so that $(s_a, s_b) \in SC, a \neq b$. $F$, $S$ and $CC$ are built up during the mining procedure and $SC$ is generated by the refinement procedure. The purpose of each of the DataStore components is as follows:

- **FileTree** $F$: The final visualization must be file-centric. To be able to produce it, a hierarchy of (relevant) files is required. Moreover, the (changed) files must be acquired before the contained scopes can be mined.
- **ScopeTree** $S$: Scopes are used to provide fine-grained information on code changes, below the scope of an entire file; The ScopeTree is a unification of abstract syntax trees (AST) from $F_n$ and therefore contains more information than the separate syntax trees.
- **Code-clones** $CC$: The raw code clone relations are necessary to relate similar scopes.
- **Scope-clones** $SC$: Changes in similarity relations between scopes form the data that is ultimately to be visualized.

The mapping of elements from $F$ to $S$ is one to many and the reverse mapping $S$ to $F$ is one to one; A file can contain multiple scopes but each scope has one main file where its skeleton is implemented and its sub-scopes are defined (forward declared). In order to unite $F$ and $S$, it seems obvious to pick one of the hierarchies as master and embed the other. However, the chosen approach is to keep both trees and interconnect the leaves in a *Compound Graph*. We do this by creating a graph from $F$ and $S$, where containment relations between the elements are explicitly modeled. This allows us to visualize the data from both a file and scope point of view.

### B. Data Mining

Unlike most other project-level analysis tools, ClonEvol only requires the modified files $F_n$ to be acquired for $R_n$ and $R_{n-1}$ to produce the desired information. The SCM provides meta-information in the form of change-logs, that contain records of files modified from revision $R_{n-1}$ to $R_n$. Files that were not changed cannot be subject to code drifts, as a drift implies that a certain piece of code was moved and therefore the file must differ between revisions $R_n$ and $R_{n-1}$. The change-log is used to build $F_n$ and then acquire each $f \in F_n$ to continue with the next step.

To model the changes in a source code base, scope information must be extracted from the source files, a job performed by static analyzers. Doxygen is a code documentation tool that supports C, C++, Java programming languages among many others [11]. It extracts structural data down to class attribute level and can be used as a lightweight, zero-configuration, static analyzer.

Next, by means of similarity analysis we extract code clones within the same revision (*intra-clones*) and between subsequent revisions (*inter-clones*). Roy *et al.* have performed a comparison of code clone detection tools and techniques and give an overview of several aspects including performance, accuracy, flexibility and license type [8]. Simian [4] is an open source similarity analyzer that can compare virtually any text based file and indicate which blocks of code are clones. This property nicely complements Doxygen's static analysis features, and allows our tool to be easily extended to other languages. The mining process is depicted in Fig. 2.

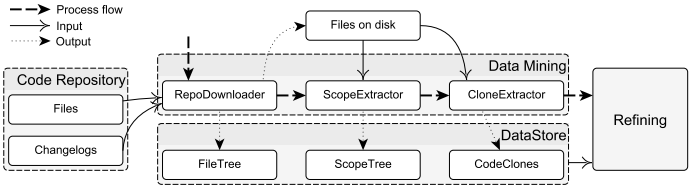


Fig. 2: Data mining procedure

### C. Data Refining

Precision of the raw data is increased by combining information in the components of DataStore, to ultimately find *changes* in similarity relations. Initially, the operation performed on a scope (modified, added, deleted) is inherited from the file that contains the scope. For $s \in S_n$, the scope operations are refined by comparing their existence in $R_n$ and $R_{n-1}$: $s \in S_n \backslash S_{n-1}$ are marked as added, $s \in S_{n-1} \backslash S_n$ are added to $S_n$ and marked as deleted. This approach indeed works, as $f \in F_n$ have also been acquired for $R_{n-1}$. The generation of scope-level clones is performed by matching the line numbers of code-clones $cc \in CC$ with the elements of $S_n$ and $S_{n-1}$.

Annotation of intra and inter-clones is needed to be able to distinguish movements of code between revisions from already existing clones. It concerns a trivial procedure, as the two types can be separated by comparing the revision number of the scope-clone's source and target; An *intra-clone* has both its source and target in $R_i$, while an *inter-clone* has a source in $R_{n-1}$ and target in $R_n$.

Drifts are scopes that have changed from $R_n$ and $R_{n-1}$ but still have similar contents. They are inter-clones, that have no related intra-clones. The amount of sources and/or targets is not defined, however either the source or target must be unique; Any case with multiple sources and targets will involve intra-clones. To illustrate that this is indeed the case, try to

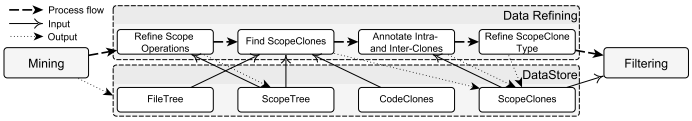


Fig. 3: Data refinement procedure

answer the following question: If a scope $a_{n-1} = b_n = c_n$, has $a_{n-1}$ drifted to $b_n$ or $c_n$? The remaining cases can be distinguished as *drift* (one source to one target), *split* (one source to many targets) and *merge* (multiple sources to one target).

Although intra-revision clones cannot be drifts, information in terms of changes in time can be harvested from them. Moreover, they are crucial for filtering out meaningless inter-clones. The refinement process is depicted in Fig. 3.

### D. Filtering & Visualization

For visualization a radial tree view (cf. Fig. 4) is chosen as it can preserve the space needed for visualization. Moreover, edges can be drawn without occluding the nodes.

The nodes of the radial tree represent the file-scope hierarchy and the edges show clone relations. In order to maintain stability of the visualization, the visual elements are mapped from the union of all revisions' files, scopes and clones. Only elements that exist in the user selected revision range are color-mapped. A node can be expanded as root of the visualization, to allow investigation of fine-grained details, e.g. clones between functions; Fig. 4 shows a sub-directory of FileZilla code base with *all* its child nodes.

To prevent occlusion when visualizing a large code base, nodes can be filtered by type. The full code base of FileZilla is shown in Fig. 5, where the level of detail is limited to directories, files and classes. Clone relations of hidden child nodes are aggregated and drawn as edges between (visible) parent nodes. When aggregation of child relations results in self-clones, these are shown as glyphs. Relations between nodes that are not in the innermost ring, e.g. files that are obstructed by classes, are indicated as 'hidden' clones.

Hierarchical Edge Bundling (HEB) [5] is used to group edges that connect scopes with similar parent scopes and/or files. HEB is combined with Catmull-Rom spline interpolation [10] to draw of smooth edges between cloned scopes.

The mirrored radial tree is accompanied by the *structure* (cf. Fig. 5), *difference* and *activity* color-maps depicted on top and bottom of Fig. 6 respectively. The first visualization was achieved by using libgraphicstreeview [1], which is based on Qt [3] and already provides the basics needed for interaction.

## III. Results

We next demonstrate our tool on the analysis of clone evolution in FileZilla client [6] (430 files, 5165 revisions, average of 3.6 changed files per revision). Fig. 4 shows a snapshot from our tool's interface. Block 1 is used to provide the repository URL and revisions of interest. Block 2 is used to filter and modify the rendering. Block 3 is used to select a range of revisions and scroll through them. The radial tree

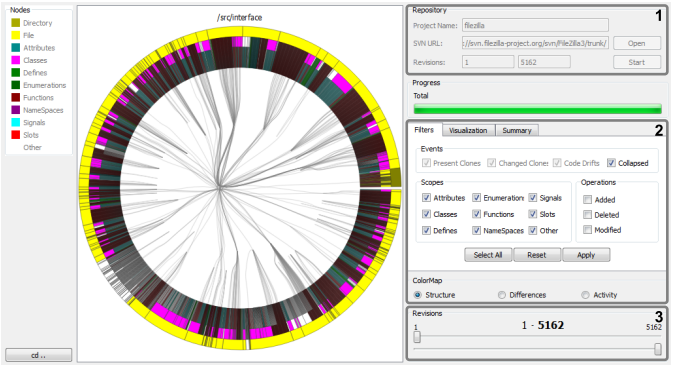


Fig. 4: Screenshot of ClonEvol application

provides interactivity to open files and folders for investigation of lower-level changes.

The *structure* color-map in Fig. 5 shows the structure and state of FileZilla code base at revision 5,165. We see that the majority of code is written in an object oriented language. After inspection of the directories and files, by hovering over or double-clicking on them in the tool, the files containing classes appear to be written in C++. The smaller directories that do not contain classes turn out to be written in C. The most clones can be found in `src/interface`, which is not surprising as the programming of interface components involves much repetition. Furthermore, the latter directory shows high cohesion with `src/engine`.

In Fig. 6 a time lapse is shown of the evolution of FileZilla code base. The *difference* color-map is an aggregation of changes that were made between the selected start and end revision; It can be interpreted as the union of the separate change-sets. We see that most clone changes and code drifts occurred in `src/interface`. Most refactoring was performed between revision 3,000 and 4,000, where clone deletions and code drifts are most prominent. The project seems to have reached a stable state around revision 4,000 as the clone activity was minimal ever since.

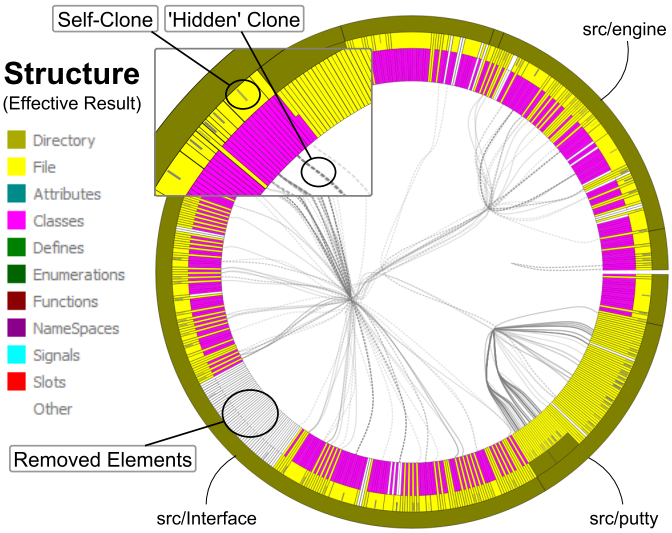


Fig. 5: Structure of FileZilla trunk/src at revision 5,165

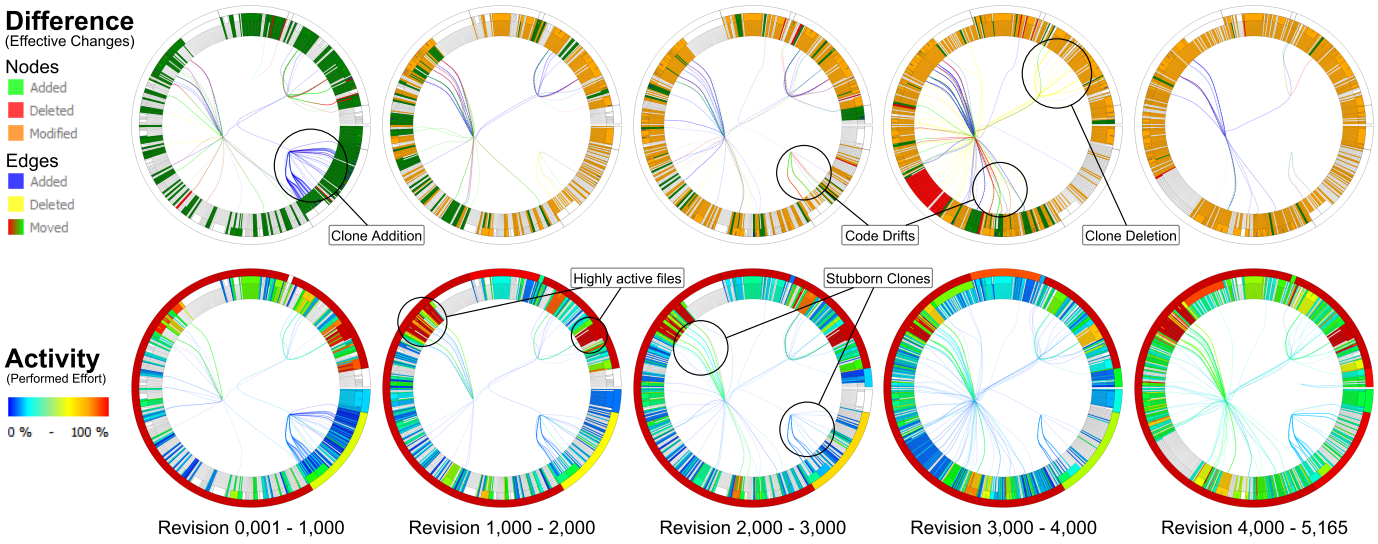


Fig. 6: Evolution of FileZilla trunk/src from revision 1 to 5,165

The *activity* color-map shows the files, scopes and clones that were most prominent in the change-sets. In each snap-shot most changes occurred in `src/interface` and `src/engine`. Three files appear to be very important for the project as they repeatedly show high activity. In these files the same clones re-appear in many change-sets, as illustrated by clone activity, from which we can conclude that these are 'stubborn' clones.

## IV. Discussion

We have shown the first results of ClonEvol, a tool-chain for visualizing the evolution of a software project in a large amount of versions. As well as virtually all source code analysis tools, in this demonstration the data was displayed in a file-oriented fashion. A feature yet unfinished is scope-oriented view on the contents, which builds the tree starting from the scopes instead of files/directories. This view is expected to be of additional value, in particular for software (re-)designers; They are often not interested in the complexity of code but rather in that of 'logical' components.

Time-wise, approximately 11:45 hours were needed to mine 5,160 versions of the FileZilla repository, containing 14,619 changed files, 10,741,843 lines-of-code in total. Hereof respectively 4:00, 5:30 and 2:15 hours elapsed during file acquisition, scope extraction and clone extraction. Used hardware included an Intel Core i7 2600k CPU @ 4GHz and a Samsung 840 Pro SSD. The total amount of scope elements was 665,232, which required about 580MB of RAM. User configuration was needed only to supply an URL of the repository and set the revisions to be mined. This illustrates that the tool is indeed easy to use and scalable computationally and data-wise.

The main performance bottleneck is the scope extraction step, particularly when C/C++ projects are imported; To be able to process the contents of implementation files (.c and .cpp), Doxygen requires the forward declarations of classes (typically located in header files). The issue is currently resolved by acquiring all headers for the first revision of interest. Hence, complexity of the scope mining procedure is $O(h * |R|)$, where $h$ is the total amount of headers in the repository's (sub)directory and $|R|$ the amount of revisions to analyze. For comparison: the code clone mining complexity is $O(\sum_{i=1}^{|R|}(|F_i|)) = O(|R|)$, as the average amount of (modified) source files was 3.6 for 5,165 versions of FileZilla and 3.4 for 10,000 versions of TortoiseSVN.

ClonEvol is subject of the author's MSc. thesis, which will be made available November 2013. More information can be found at: http://www.cs.rug.nl/svcg/SoftVis/ClonEvol.

## Acknowledgment

I would like to thank my supervisor, Prof. Dr. Alexandru C. Telea for his guidance during my work on ClonEvol.

## References

[1] S. Andrade, "libgraphicstreeview," 2012. [Online]. Available: http://liveblue.wordpress.com/2012/05/
[2] P. Avgeriou and U. Zdun, "Architectural patterns revisited - a pattern language," 2005.
[3] Digia. (2013) Qt project. [Online]. Available: http://qt-project.org/
[4] S. Harris. (2011) Simian: Similarity analyser. [Online]. Available: http://www.harukizaemon.com/simian/
[5] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 741–748, 2006.
[6] T. Kosse. (2013) Filezilla - the free ftp solution. [Online]. Available: http://filezilla-project.org/
[7] D. Reniers, L. Voinea, O. Ersoy, and A. Telea, "The solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product," *Science of Computer Programming*, 2012.
[8] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.
[9] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," in *Computer Graphics Forum*, vol. 27, no. 3. Wiley Online Library, 2008, pp. 831–838.
[10] C. Twigg, "Catmull-rom splines," *Computer*, vol. 41, no. 6, pp. 4–6, 2003.
[11] D. van Heesch. (2013) Doxygen: Source code documentation generator tool. [Online]. Available: http://www.doxygen.org/