# Improved Tool Support for the Investigation of Duplication in Software

Cory Kapser and Michael W. Godfrey
Software Architecture Group (SWAG)
School of Computer Science, University of Waterloo
{cjkapser, migod}@uwaterloo.ca

## Abstract

*Code duplication is a well documented problem in software systems. There has been considerable research into techniques for detecting duplication in software, and there are several effective tools to perform this task. However, a common problem with such tools is that the result set returned can be too large to handle without complementory tool support. The goal of this paper is to describe the criteria for a complete tool that is designed to aid in the comprehension of cloning within a software system. Furhermore, we present a prototype of such a tool and demonstrate the value of its features through a case study on the Apache httpd web server. For example, in our study we found that a single subsystem comprising only 17% of the system code contained 38.8% of the clones.*

## 1. Introduction

Code duplication, or code cloning, is generally believed to be common in large industrial systems [5, 7, 9, 15, 20, 21, 23]. Management of code cloning and the various problems associated with it is important for the successful evolution of software systems. One such problem is the copying of assumptions that may or may not be documented: when the environment changes to violate these assumptions it may not be clear where in the code changes need to be made. For example, suppose there is an assumption of the existence of a previously initialized global array that has been copied. If the array is now to be initialized within a specific function, bugs will be introduced into the system unless appropriate constraint checking and initialization is introduced in each copy of the code. To ensure the successful evolution of large software systems, such problems must be addressed. The effort to determine the required changes may be daunting in large industrial systems.

Maintenance of clones in software is therefore important. One problem with many clone detection methods is that they may return a large set of suspected clones, but provide little or no additional information about them to aid the user in their interpretation. This makes clone management cumbersome at best and intractable in general. Viewing and classifying or grading thousands of clones manually is time consuming and impractical, but necessary if one hopes to manage clones successfully, by either removing or documenting important points of duplication. For example, in our Apache httpd case study, a naive use of the clone detection tool CCFinder resulted in 13062 clone pairs. In this study, we set the minimum clone size to be 30 tokens. The tool we propose tries to aid in the process of clone filtering and classification by using automatic classification and filtering. The classification will help users direct their efforts towards types of clones they believe to be most beneficial.

While automatic classification of clones may indeed aid the user in identifying clones that are relevant to their task, it is not enough. Clone categories may have hundreds of clones, and these clones relay little information about the architectural relationships these clones create within the system. Mechanisms for navigation and exploration must be in place. Such mechanisms should include multiple views of the clones in the system, an architectural perspective for example; the ability to query the clone set, for similar or related clones; and the ability to filter the clone set as the user desires.

The goal of this work is to describe the criteria for a tool that is general enough to be used in most tasks related to understanding cloning in a software system, yet complete enough to remain useful in each of these tasks. We describe a tool we have developed to meet these criteria and perform a case study to demonstrate the value of meeting these criteria.

This work makes several contributions to the area of clone detection research. It lays the ground work for the requirements of an effective clone navigation and visualization tool. It also provides a prototype of a tool that would embody the features we believe are required to effectively understand and ultimately manage clones in software. It also provides further information about how cloning occurs in software systems by means of a case study.

The paper is organized as follows: Section 2 describes the necessary criteria for tools that enable the exploration and comprehension of clones in software; Section 3 describes a prototype of such a tool; Section 4 describes a case study we did using the tool; Section 5 describes related work in this area of research; Section 6 concludes the paper.

## 2. A Tool To Explore Clones

In this section we briefly outline our general criteria for a tool used to navigate and understand cloning in a software system. We then describe in more detail the features needed to meet these criteria. This set of criteria and features is derived from much manual work by the authors in attempting to understanding cloning in software systems, starting with our studies of the Linux kernel file-system subsystem and the database server Postgresql [16, 17]. Some features were also taken from suggestions by students in a senior level graduate course who used the tool Gemini [25] and CLICS, the tool described in this paper, to perform an analysis of clones within the Linux kernel source code.

### 2.1. Criteria

The core challenge to the maintenance and management of cloning in software systems is comprehending the actual types of clones and the dependencies they create in the software system. To complete such a task, duplication must be detected and then cloning must be evaluated throughout the system at a variety of different levels of abstraction. Methods to find the duplication in the source code has been a topic of research for some time now and there is a large variety of methods to detect clones [5, 6, 7, 9, 12, 15, 21, 22, 23]. However, clone detection tools can return very large result sets and viewing every possible clone is infeasible. To address this problem, tools and processes need to be developed to help guide the software maintainer toward the information they require to complete the task.

We consider that any tool designed to help navigate and understand cloning in a software system should provide:

- facilities to evaluate overall cloning activity.
- mechanisms to guide users toward clones that will be most effectively used in their task.
- methods for filtering and refining the analysis of the clones.

Each of these criteria is described in more detail below.

**2.1.1. Overall System Evaluation.** As a first step in understanding cloning within a software system, regardless of the end goal, maintainers must have an understanding of the cloning from a high level of abstraction. This understanding will allow the user to evaluate the extent and the severity of the duplication in order to estimate cost and/or necessity of the task. This information will also be a starting point to guide them through the rest of their task.

Several mechanisms can be used to evaluate cloning from a high level. These mechanisms can be visualization methods, such as scatter-plots [5, 9, 15, 24, 25] for example. They can also be metric oriented, such as reporting the percent of lines cloned, average length of clone, etc. Whether the method is visualization or metric oriented, it is important that the information provided is scalable and covers a wide range of aspects.

**2.1.2. Guide and Empower the user.** The possibly large sets of clones returned by the clone detection methods make it infeasible to look at each individual clone. A tool designed to aid in the comprehension of cloning in a software system should provide a means of guiding the user to clones relevant to his/her task and reduce the data set as much as possible without loss of information.

There are several ways to direct users toward the clones they seek. Metrics can be used to query the data set [11]. Some examples of metrics that might be used are the size of the clone, the types of changes made to the clone, and types of external dependencies a code segment has. Such a method can direct users to promising refactoring opportunities. Other methods of querying the data set can also be used, such as querying based on location of the clones in the software and the type of code the clone exists in. As an example of such a query, a user might be concerned about cloning of macros originating in a particular file. Querying mechanisms provide flexible and customizable analysis, allowing users to leverage their own knowledge about the software and cloning, making the user more effective in their task.

Upon initial survey of a software system, users may not be fully aware of what types of information they want or need. Query facilities can suffer from this weakness and strong static analysis of the data set is also required. Static analysis should provide low level metrics about cloning activities in the system, as well as a method of navigating through clones in such a way that the user has some knowledge about the clones they are seeing. An example of this would be categorization of the clones as is done in [6, 17].

Cloning should be described in terms of the system architecture in some views. We believe that relating cloning and architecture can have great benefits to comprehension of cloning. Cloning is a type of implicit architectural dependency, and as such can provide information about the architecture and design of the system. This also enables users to use their own knowledge of the architecture of the system when evaluating clones.

**2.1.3. Analysis Refinement.** Due to the subjective nature of the analysis of clones, from the perspective of the user, there will always be clones that do not apply to their task, For this reason, it is important that tools supporting the comprehension of cloning provide mechanisms to remove and filter clones from the analysis.

## 3. The CLICS System

This section provides details of a sample implementation of a tool written to meet the criteria listed above. The *CLone Interpretation and Navigation System (CLICS)* is a prototype implementation of a tool that meets the criteria listed above. It is an example of the types of features that can be used to provide a high level initial view, guide users through the clone understanding process, and refine the fact base as users learn more about the cloning in the software system.

### 3.1. System Overview

CLICS was first described in [17]. While the tool could be used with any number of clone detection tools, it uses clone detection results from the tool CCFinder [15] as the dataset. CLICS uses a taxonomy of clone types to categorize clones and generate statistics about clones in a software systems. The taxonomy, described in [17], has been constructed through manual classification of clones in several software systems. The most recent version of the taxonomy can be found at [3]. Currently, CLICS supports clone analysis in C/C++ and Java.

**3.1.1. Extracting Regions from Source Code.** In our first step we use ctags, a tool for extracting indices of language objects found in the source code. We then find the end points of functions, macros, structs, unions, and enumerations. Then we join consecutive objects of the same type if they are type definitions, prototypes, or variables into regions.

Using this information we map the file to eight types of regions: consecutive type definitions, prototypes, and variables; individual macros, structs, unions, enumerations, and functions. Comments are ignored in the analysis. Subregions are also extracted for fuctions. These regions include initialization or variables, code blocks such as control flow statements, and series of function calls.

**3.1.2. Mapping Clone Pairs to Regions.** In the next step of the process, for each clone pair we map both segments of the clone to a region in a file. We consider a segment's region to be the one that contains the largest portion of its code. The tool CCFinder ends clones that are part of a function at the end of the function, so we are not concerned about clones that may map to several functions. In the case where a clone maps to several different region types, it may be better to break the clone up, but in practice we find the current method to work quite well.

If two regions have cloning between them, we say they have a *cloning relationship*. For each region with a cloning relationship we group together all the clones that form this relationship, and we call this a *Regional Group of Clones* (RGC). An example of such a group would be several clones between two functions. The concept of RGC is useful for both visualizing and filtering clones.

**3.1.3. Filtering.** CLICS begins by filtering the dataset of several common types of false positives that are detected by parameterized string matching methods. These filters work by enforcing stricter criteria for a clone match in particular areas of the code. There are currently four filters we have implemented. In all cases, the thresholds were chosen through rigorous trials.

The first filter operates on structs, union, type definitions, variables, and prototypes. If an RGC has one region of the previously mentioned types, any clone in this RGC must have a minimum of 60% of its lines match exactly. In our experience this filter eliminates a substantial number of clone pairs from the result set without removing many, if any, true positive matches.

We also filter clones occurring on statements that are "simple function calls". Regions of code that are "simple function calls" are sequences of code of the form function_name(token [, token]*). The criteria for a match is 70% of the function names in either region must be similar. We consider two function names to be similar if the edit distance, as computed by the Levenshtein Distance algorithm, is less than half the length of the shortest of the two functions being compared.

We found that clones within switch statements are often false positives. To filter clones in these areas, we require that 50% of the lines of code in these areas match. Clones in very simple *if-then-else* blocks are also filtered in this way.

Clones whose two segments of code overlap by more than 30% of their length are also removed. Although the filtering mechanisms are quite simple, they are quite effective. Using these filters we are able to reduce the dataset by approximately 60%.

**3.1.4. Sort Clones into Taxonomy and display results.** Once we have a set of initially filtered clones, we can sort them into the taxonomy. Then the user is presented with a GUI containing views of clones using trees of hierarchical classification, clone classes, and system architecture views. The taxonomy is only briefly described in this paper.

### 3.2. The Clone Taxonomy

Here we provide a very brief overview of the clone taxonomy. For a more detailed description, see [3, 17]. The taxonomy is a hierarchical classification of the clones using attributes such as location and functionality. Its goal is to reflect the types of clones that occur in software systems, and is based on manual inspection of detected clones in several software systems.

A compressed version of the taxonomy can be seen in Table 2. There we can see the hierarchical structure of the taxonomy. First, clones are divided by how close the two code segments of the clone are within the system. Clones at this level are either *Same File*, *Same Directory - Different File*, or *nth Cousin Clones* where *n* is the distance measured by the closest common node in the file-system of the software system. For example, *Same Directory, Different File* clones are synonomous to 1st cousin clones, clones occurring in two different directories that have the same parent directory would be *2nd Cousin Clones* and so on.

Clones are then partitioned by the type of region they are found in. Clones between two functions are called *Function to Function Clones*. All other clones are referred to here as *Non-function Clones* for brevity.

*Function to Function* clones are subdivided as functions that are nearly the same *Function Clones*; functions that are very similar, *Partial Function Clones*; functions where a large portion of one is cloned into another, *Cloned Function Bodies*; and small segments of code are shared between two functions, *Clone Blocks*. *Clone Blocks* is further subdivided in the taxonomy but due to space limitations not described here.

### 3.3. Meeting the Criteria For a Clone Navigation Tool.

The following section describes the features we implemented to meet the criteria we described for a clone comprehension tool. It is a proof of concept implementation.

**3.3.1. Overall System Evaluation.** To provide a general system overview, we compute a series of metrics encompassing several aspects of the system: system size, percentage of system cloning, and frequency of clone types. The metrics detailing system size include the number of files and LOC.

Metrics describing percentage of system cloning include percentage of lines that have a clone, percentage of methods containing a clone, and percentage of files containing a clone. Metrics describing the percentage of lines with clones and the percentage of methods with clones are useful indications of the degree of cloning in the software system. The percentage of files containing clones is a useful metric when determining the clone density in the system.

To describe the frequency of different types of clones in a software system, we list the the number of occurrences of each clone type in the taxonomy, similar to Table 2. These metrics can provide indications as to the types of problems that may be occurring in the system and may indicate the degree of difficulty of the investigation and management.

**3.3.2. Guide and Empower the user.** CLICS uses several mechanisms to enable the user to perform an in-depth analysis of clones in the system. These mechanisms include visualization of clone relations of subsystems using a hierarchical containment graph, metrics for entities at all levels of architectural abstraction, clone navigation through the taxonomy, clone navigation through the subsystem tree, and query facilities.

To visualize the cloning activity we use *LSEdit*, part of the architecture recovery toolkit, *SWAGKit*. *LSEdit* is a graph visualization tool that is designed for the exploration of software "landscapes". Landscapes are graphs representing software architectures and their dependencies. The nodes of the graphs are software artifacts such as a subsystem, file, or method. The edges of the graph are dependencies between two software artifacts. For the purpose of clone investigation, the edges are clone relationships. Entities of the graph can be hierarchically contained, allowing varying levels of abstraction during analysis. *LSEdit* is a full featured graph exploration tool with a rich set of features for navigation and display of architectures.

We visualize the clones in this way because we believe it is more scalable than scatter-plots used in [5, 9, 25]. Users can see the dependency between subsystems easily via cardinalities and arrow heads indicating magnitude of dependency. They can also see unexpected cloning relationships quickly, such as the dependency between *server* and *include* in Figure 1. These two clones were caused by cloning of four function prototypes and one cloned file, discussed below.

Cloning can be explored through two other mechanisms. The first is via the clone taxonomy described earlier. This view uses a *clone type navigation tree* similar to those found in several file management programs. This structure is well suited for this purpose because of the taxonomy's hierarchical nature. Users can view clones in each category, and remove any clones they believe to be false positives. This method of navigation is useful when performing initial analysis of clones in the system. Clones themselves are displayed using highlighted text. Clones residing in the same region as the selected clone are also highlight to show the user the degree of cloning in that region. From this view, and all other views of the source code, the user can query for clones related to the currently select clone and source code.

The second form of navigation of clones is done through

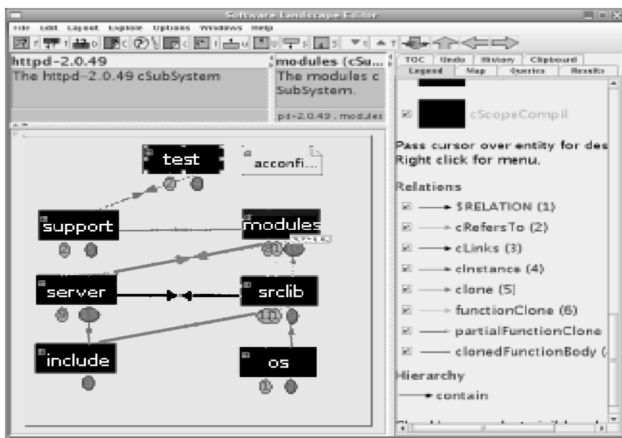


**Figure 1. Architectural View**

*system navigation tree*. This is structured to represent the subsystem containment hierarchy of the software. As users select a node on the tree, they are given various sources of information about the entities contained within it. This information includes cloning metrics, subsystems in a cloning relationship with the entity, and the file source code. Table 3 shows the metrics of the *mpm* subsystem of the *Apache httpd* server. For each software artifact within the selected entity, the following metrics are shown:

- number of clones involving only entities within the referenced entity
- number of clones with one segment within the referenced entity, and one segment somewhere else in the software system.
- percentage of lines of code of the software system contained within the given entity.
- percentage of total clones that are involved with referenced entity.

These metrics give the user information to quickly see cloning "hotspots". A "hotspot" would be an artifact (a subsystem, file, method, etc) containing a substantially larger portion of clones than other entities near it. For example, in Table 3 we see that the *server* subsystem contains 38.8% of the overall clones in the system, but only has 17% of the lines.

Currently only limited query support is implemented in CLICS. CLICS supports querying clones based on location, based on clones relations to code segments, and clones of a particular size. Queries of clones based on location include clones strictly within a given entity, clones going from one entity to another, and clones that have at least one of its code segments in the entity. All query results are displayed in a categorized clone navigation tree as described above.

Querying for clones related to a region of code includes queries for clones that are directly related to the code, and queries for clones that are related transitively. Directly related clones are clones where one of the segments of a clone pair is within that region. Queries for transitively related clones return the result of the transitive closure of the clone relation. Such queries often uncover clone relations that are missed clones, missed because the code segments differ enough to not be detected as actual clones. Each of these clone queries can be modified to restrict the query to a particular line of code, restricted to the same clone type as the current clone selected, and generalized to all clones involving that region.

### 3.4. Analysis Refinement

Refinement facilities in CLICS are currently restricted to the manual removal of clones and removal/addition of files from the analysis. Users can remove individual clones, whole RGCs, and clone classes. They can also select files to be excluded from the analysis. More advanced filtering mechanisms are currently being developed.

## 4. Case Study

In this section we will describe the case study we performed on the Apache httpd web server [1]. Based on NCSA httpd 1.3, the first release of Apache was made in April 1995. Since then, it has become tremendously popular. As of March 2005, it has been found that more than 68% of web sites are served by Apache [2].

Through a case study, we show the value of many of these features at different levels of the analysis. We also show the value of viewing cloning as a dependence relationship in a software architecture. We show the value of metrics as a tool for initial inspection of the cloning, and also as a guide to find hotspots in the system and why they occur.

We chose Apache as our case study because it is of non trivial size, and has several interesting characteristics in its architecture that we wished to investigate. The first characteristic is that it runs on several different platforms: BeOS, Netware, OS/2, Unix, and Windows. In particular, there is a Multi-Processing Module (MPM) for each of these platforms. It also has several experimental MPMs released with the source. These modules were expected to have a high degree of cloning amongst them as they were implementing very similar functionality.

Apache is a medium sized software system. Our study focused on Apache version 2.0.49, which consists of 709 *.c* and *.h* files. There are 261,219 LOC. It has a plugin style architecture, with a core responsible for process management and low level data transmission and receiving. Modules "hook" into the core by defining points in the data processing chain where they can fill a particular need [1, 10]. The

architectural representation we use to represent *Apache* was derived directly from the directory structure of the source code.

### 4.1. The Data Set

Using CCFinder we detected clones in the source code. CCFinder allows the user to set the minimum size of a clone that will be detected. We chose 30 tokens to be the minimum size of detected clones.

Using the filters we described earlier, a significant portion of clones were removed from the data set. Table 1 illustrates the usefulness of our filters. We were able to remove 8081 clones, 61.8% of the total, with high confidence that very few true positives were removed. We assessed the quality of the filtering through manual analysis of the filtered clones. After removing a single subsystem that was clearly causing false positives we were able to remove an additional 1405 clones from the data set.

This subsystem was test suites for the *apr* subsystem. These test suites use the *C Unit Testing Framework (CuTest)*. *CuTest* uses a very simple and repetitive method of adding tests to a test suite. There were 1380 clones within this subsystem, the vast majority between code for building the test suite.

After the automatic and slight manual filtering, we began to look more deeply into the cloning of the *httpd* server. The reader should note that while the number of clone pairs in the data set was greatly reduced, there was only a minor reduction in the actual percentage of the system that contains clones, as shown in Table 1. This is because only small portions of the code generate a large percentage of the false positives.

### 4.2. Cloning - From the High Level

At first glance, Apache exhibits several differing characteristics from previous reports on other software systems. Studies suggest that cloning tends to occur within the same directory or subsystem [15, 16, 17]. In Tables 1 and 2 we see cloning is somewhat more prominent across subsystems rather than within the same subsystem. Table 1 describes the number of clones in the same file, same directory and in different directories. It illustrates the distribution of clones at various levels of filtering, including no filtering, automatic filtering as described above, removing the *apr/test* subsystem, and the final data set. Table 2 illustrates the types of clones in the system and their distribution. It shows the total number of clone pairs in each category, and the total number of RGCs with clones in these groups.

Both tables show us that a high degree of cloning has occurred across subsystems. After filtering, both manually and automatically, we see that 51% of the clones in *Apache* are crossing subsystem boundaries, shown in Table 2 as the cousin groups. This is explained by the method of behavioral duplication the *Apache* team used when porting *Apache* to the platforms it runs on today. In many cases, when platform specific code was required, such as in the *mpm* subsystem, much code appears to have been copied and then ported to the platform.

Table 2 illustrates the importance of filtering. There is a drastic difference in the number of *Same Directory Clones* from the initial automatic filtering and the subsequent manual filtering carried out. This can also be seen in Table 1. Two important points can be drawn from this observation. First, one can see the impact of false positives on the results and the importance of careful inspection of the data set before reporting results. Second, it highlights the usefulness of providing metrics for each subsystem as the users browses from the architectural perspective. Within five minutes, the authors identified the *apr/test* subsystem anomaly by noting the disproportionate percentage of clones in the subsystem when compared to its relative size.

Table 2 also tells something about the types of clones that are in the system. We see in the table that 60% of clone pairs contribute to function clones. For clones outside of the same file, 70% of clones contribute to function clones. From this set of clones, we can see that function clones outside of the same file are composed of 2.5 clones on average. This is an interesting result. It indicates that the functions that have been cloned possibly have several non trivial changes in the code. This indicates that the functions are likely going to be difficult to be refactored. We can also see from this table that there are several non-function clones. These clones are *Macro clones, Prototype clones*, and *Clones of Global Variables*.

Immediately from these results we can see the value of providing a variety of views and metrics to facilitate initial high level analysis of the system. Without metrics describing both the type and the frequency of clones throughout the system, it would be difficult to have such an in-depth analysis of the overall cloning in the software system at such an early stage in the process.

### 4.3. Inspecting the code

During our initial inspection, we were concerned with the various edges between unrelated subsystems, as pictured in Figure 1. In particular, we wanted to determine the causes of the edges, and eliminate any false positives contributing to them. The end result of that investigation led, in addition to the filtering mention previously, to the removal of several false positives, and the corresponding metrics for the taxonomy are list as “More Filtering” in Table 2. The total time spent refining the results and investigating the high level dependencies depicted in 1 was under two hours. The

| Filtering | Same File | Same Dir | 2nd Cousin | 3rd Cousin and more | Total | % of System |
|---|---|---|---|---|---|---|
| **None** | 2809 | 1464 | 1494 | 7294 | 13061 | 15.6% |
| **Automatic** | 1471 | 1334 | 883 | 770 | 4458 | 12.7% |
| **Automatic & Manual** | 1291 | 135 | 883 | 753 | 3053 | 12.1% |
| **More Manual** | 912 | 135 | 840 | 641 | 2528 | 12.0% |

**Table 1. Frequency of clones at different levels of filtering**

| | Auto. Only | | More Manual | |
|---|---|---|---|---|
| **Type** | **Num. Clone Pairs** | **Num. Region Pairs** | **Num. Clone Pairs** | **Num. Region Pairs** |
| **Same File** | **1471** | **970** | **912** | **845** |
| *Same Region* | *584* | *200* | *528* | *180* |
| *Function to Function* | *877* | *760* | *749* | *655* |
| Function Clones | 530 | 491 | 451 | 416 |
| Partial Function Clones | 67 | 54 | 42 | 38 |
| Cloned Function Body | 27 | 22 | 24 | 19 |
| Clone Blocks | 253 | 193 | 232 | 182 |
| *Non-Function Regions* | 10 | 10 | 10 | 10 |
| **Same Dir. Different File** | **1333** | **326** | **135** | **90** |
| *Function to Function* | *1332* | *324* | *134* | *89* |
| Function Clones | 1211 | 241 | 62 | 53 |
| Partial Function Clones | 65 | 34 | 38 | 9 |
| Cloned Function Body | 25 | 25 | 8 | 8 |
| Clone Blocks | 31 | 24 | 26 | 19 |
| *Non-Function Regions* | *1* | *1* | *1* | *1* |
| **2nd Cousin** | **883** | **400** | **840** | **394** |
| *Function to Function* | *877* | *394* | *834* | *388* |
| Function Clones | 658 | 240 | 592 | 237 |
| Partial Function Clones | 52 | 41 | 45 | 38 |
| Cloned Function Body | 22 | 22 | 22 | 22 |
| Clone Blocks | 175 | 102 | 175 | 91 |
| *Non-Function Regions* | *6* | *6* | *6* | *6* |
| **3rd Cousin** | **693** | **267** | **611** | **241** |
| *Function to Function* | *683* | *257* | *601* | *231* |
| Function Clones | 477 | 147 | 465 | 143 |
| Partial Function Clones | 72 | 32 | 34 | 24 |
| Cloned Function Body | 9 | 9 | 7 | 7 |
| Clone Blocks | 125 | 69 | 95 | 57 |
| *Non-Function Regions* | *10* | *10* | *10* | *10* |
| **4th Cousin** | **65** | **25** | **19** | **18** |
| *Function to Function* | *33* | *9* | *18* | *17* |
| Function Clones | 64 | 9 | 7 | 7 |
| Partial Function Clones | 10 | 3 | 2 | 2 |
| Cloned Function Body | 1 | 1 | 0 | 0 |
| Clone Blocks | 20 | 11 | 9 | 8 |
| *Non-Function Regions* | *1* | *1* | *1* | *1* |
| **5th Cousin** | **12** | **12** | **11** | **11** |
| *Function to Function* | *11* | *11* | *10* | *10* |
| Function Clones | 6 | 6 | 6 | 6 |
| Clone Blocks | 5 | 5 | 4 | 4 |
| *Non-Function Regions* | *1* | *1* | *1* | *1* |

**Table 2. Frequency of clone categories — Parametric String Match**

reader should note that the authors had some knowledge about the architecture of *Apache httpd* prior to this study. As mentioned above, the difference in the resulting data set is quite dramatic.

A surprising dependency seen in Figure 1 is the cloning between *include* and two other subsystems, *server* and *srclib*. A single file, *pcreposix.h* and several function prototypes have been copied from *include*. Such a dependency should not exist, and this is an example of bad cloning. Unnecessary effort is required to keep the header files synchronized. The authors believe that these clones should be eliminated by maintaining only one copy of this file.

Clones between the *srclib* and other subsystems were unexpected. *srclib* is primarily composed of the *Apache Portable Runtime Project (APR)*. It is shipped with the Apache web server source code, it is not directly related to web services. It is designed to provide a consistent, platform independent API to underlying platform specific functionality. Cloning between the *server* and *scrlib* involved cloning of time and date formatting, queue control, and bucket management. It was composed of eight function clones and a few other clone types, most of which were identical clones. In the cases of the function clones, the names were the same, or very close. *os* and *srclib* shared a single function clone, again an exact duplicate with only the name of the function and the parameter type were changed. *srclib* and *modules* share clones involving the management of a hash table. There are three *function clones* and a *partial function clone* in this set. While the function clones clearly have the same origin, and are nearly identical, we would not suggest refactoring in this case because they are only a few function of many that are responsible for managing the hash tables in the two subsystems. A much better approach to management would be to either refactor the code using the *modules* hash to use the one in *apr* or document the functions and maintain them in parallel. These two subsystems also share a small code segment involved with command formatting.

We were not surprised to see dependencies between *modules* and *server*. Much of the servers http processing code was moved from the *server* to the *modules* with the release of version 2.0. Because we tend to find clones in closely related files in regards to architecture, we were not surprised to see clones here. Additionally, both *modules* and *server* process the same data types. This is reflected in the types of clones we see, which primarily involve handling of the bucket brigade and translating document requests. That said, there was still relatively little cloning between the two subsystems, with a total of eight clones.

Cloning between *modules* and *test* and *support* were surprising. The *support* subsystem consists of several individual executables used for benchmarking and configuring the server. The clones we found were between the Apache Benchmark utility and the *mod_ssl* subsystem. They were simple function clones involving the setup of an ssl connection. Clones between *test* and *modules* involved converting time strings, there were only two.

| Name | Internal | External | %Clones | %Lines |
|---|---|---|---|---|
| **httpd** | | | | |
| srclib | 1165 | 22 | 40.8 | 39.8 |
| server | 1103 | 24 | 38.8 | 17.0 |
| modules | 513 | 19 | 18.3 | 40.5 |
| support | 68 | 7 | 2.6 | 3.1 |
| os | 9 | 3 | 0.4 | 1.2 |
| test | 3 | 6 | 0.3 | 0.6 |
| include | 0 | 3 | 0.1 | 1.9 |
| **server/MPM** | | | | |
| experimental | 155 | 512 | 23.0 | 2.9 |
| worker | 11 | 281 | 10.1 | 1,2 |
| prefork | 5 | 222 | 7.8 | 0.6 |
| netware | 4 | 192 | 6.8 | 0.6 |
| winnt | 18 | 166 | 6.3 | 2.2 |
| beos | 5 | 174 | 6.2 | 0.5 |
| mpmt_os2 | 1 | 118 | 4.1 | 0.4 |

**Table 3. Distribution in Subsystems**

Most of the clones we listed above, specifically function clones, involved generic high level concepts that should be implemented in a standard library, such as *srclib*. For most of the above redundant code, it would be sensible to factor out the commonalities to such a place. Overall, however, the cloning at the high level was mostly superficial, and not overly concerning.

It is important to point out several features were very important at this stage of the analysis. Visualization using LSEdit provided us with the questions we needed to ask about the high level dependencies in the software. Being able to query for only clones occurring between a set of entities was vital in the analysis. This feature gave us quick access to the clones composing a given edge in the graph making the investigation fast and efficient. Automatically categorizing the query results using the clone taxonomy described earlier made it very easy to determine the types of dependencies that bound the subsystems.

**4.3.1. Deeper Into MPM.** An example where cloning across subsystems is very high within the *Apache* web server is the *server/mpm* subsystem. As mentioned earlier, it is also a good example of what we call “hotspots”. This subsystem contains the implementation of the process management for the various platforms. In Table 3 we see that there is a very high degree of external cloning within the *mpm* subsystem. The view in LSEdit (not shown here) is one of a fully connected graph, each subsystem being related to every other subsystem. It is interesting to note that, aside from *experimental* there is relatively little cloning within each entity, but the external cloning is very high.

Looking at the categorised clones within this subsystem provides more detail about the cloning within *mpm*. From this we noted that 78.0% of the clones in *mpm* contribute to

function clones. Also, 93% of clones in this subsystem are *2nd* and *3rd* cousin clones.

There are several functions that contribute to large clone groups within this subsystem, with names similar to *setserver_limit, ap_mpm_query, set_deamons_to_start, set_max_free, set_min_spare_threads, set_signals*. The reason for this high degree of cloning is that the high level behaviour of the systems are similar. We believe that this subsystem would benefit greatly from a language that supports inheritance.

Due to space limitations, we can not describe the full analysis of the *mpm* subsystem here. We can, however, mention the features that are important when analyzing the cloning at this level and in such a complex web of dependencies. Querying for related clones, or generating the clone class, of a region of code is very important when determining the degree of cloning and the types of changes that have been made. Also, metrics were invaluable when expressing the type of cloning activity in this system.

**4.3.2. Other Sources of Heavy Cloning.** The *mpm* subsystem is not the only entity in the system where duplicate behavior through code duplication is performed. The systems *threadproc, lock* and *fileio* in *apr* also exhibit similar activity. These subsystems are also cloning “hotspots”. The subsystem *threadproc* contains 9.7% of the clones, but only 2.3% of the lines of code of the software. They display a similar distribution of clone types as *mpm*. Within *mpm* in particular, transitive queries were useful when trying to grasp the full extent of the cloning within the subsystem.

### 4.4. Summary

Upon initial inspection, *Apache* appears to deviate from previous findings that cloning tends to occur within subsystems. However, closer inspection reveals that this still tends to be true. While cloning was most often between two distinct subsystems, most subsystems sharing code were contained within the same higher level subsystem. In fact, as described in our case study, cloning across the highest level subsystems was quite rare in comparison to cloning within, shown in Table 2 as *4th* and *5th Cousin Clones*.

The *Apache* case study has raised interesting questions about cloning in multi-platform software systems. In this case study, we found that platform specific code often had a high degree of cloning. It appears that such cloning is a reasonable design strategy, in terms of flexibility and design of the software system. Activities like this provide a way to “bootstrap” the porting of platform specific code, without requiring major changes to the design of the overall system. This can be an advantage in the initial stages of development when appropriate abstraction levels and degrees of commonality between subsystems are unclear. In later stages of the program development cycle, this can still be an appropriate method of duplicating behavior in a software system. In cases of experimental additions to the system, such as *mpm/experimental*, it is reasonable to clone code because you do not want prototypes or exploratory projects infecting the currently stable and maintainable code.

We have observed that while the code of platform specific implementations can be quite similar, there are many slight variations in the duplicated code. It would be difficult to completely refactor the software at these points without breaking the understandability of the code. For this reason, it seems that cloning like this in similar systems is a sensible practice.

## 5. Related Work

Visualization of clones is commonly done using scatter-plots to present matched lines of code [5, 9, 24, 25] These scatter-plots provide the ability to select and view clones, as well as zoom in on regions of the plot. In practice, we have found scatter-plots do not scale well with medium to large software systems. The representation of the clones becomes so small that it is difficult to pick out all but the most severe cloning. Additionally, scatter-plots do not easily lend themselves well to providing the context of cloning from an architectural perspective.

Gemini [25] and Aries [11] are two tools that use CCFinder as their core clone detection mechanism. In addition to scatter-plots, Gemini also provides visualization through metrics graphs and file similarity tables. It allows users to browse clones either pair by pair, or using clone classes. Aries is a refactoring support environment for duplicated code. Aries supports refactoring using metrics based querying. Users can query for clones matching a variety of metrics and thresholds. While Aries provides the capability to refine the displayed clones using queries, these tools do not support data set refinement or views mapping clones to system architecture.

In [13], Johnson used Hass diagrams to visualize cloning relationships. In [14], he proposed the use of hyper-linked documents to navigate cloning relationships. Reiger et. al. describes five polymetric views with the focus of showing what parts of the system are connected via code cloning and what parts are cloned the most [24]. These views have been designed to educate the user about the cloning in a software system at different levels of abstraction, providing progressibly more infomation about the cloning in the software. Using metrics, architectural graph representations and the *System Navigation Tree* we also provide the first four views. Our work differs from the above works in that we aim to provide the criteria required to make a complete clone comprehension tool. Providing high level views and navigation through visualization is one part of the overall system. We

also require filtering facilities, metrics reporting, querying facilities.

There is a wide variety of clone detection techniques that have been developed. These methods range from string comparison, metrics comparison, and program graph comparison strategies [6, 5, 7, 9, 12, 15, 19, 21, 22, 23]. Clone classification schemas have been previously suggested, usually based on the degree of similarity of segments of code and also the type of differences [6, 23]. These classifications are limited to function clones only.

Recent studies have shown that cloning in software tends to occur between files that are close within the system [15, 16, 17]. Studies have also shown that in the *Linux* kernel the addition of similar subsystems was done through code reuse rather than code cloning [4, 8]. From our case study, we see that the *Apache* development team tended to clone when adding code related to a specific platform. Studies on how programmers actually create clones in code were done by Kim et. al. [18].

## 6. Conclusions

Cloning in software systems is an important maintenance challenge. It requires tool support to make analysis and management a tractable and feasible problem to solve. The purpose of this paper was to layout a set of criteria for tools designed to aid in the understanding of clones in a software system. We described the types of features that could be used to meet these criteria and then demonstrate the use of prototype of a tool designed to meet those criteria. Through a case study, we show the value of many of these features at different levels of the analysis. We also demonstrate the importance of relating cloning to architectural dependencies.

## References


[1] The apache http server project. `"http://httpd.apache.org/"`, 2005.

[2] Netcraft: Web server survey archives. `"http://news.netcraft.com/archives/web_server_survey.html"`, 2005.

[3] A taxonomy of clones in software. `"http://swag.uwaterloo.ca/~cjkapser/CLICS/taxonomy"`, 2005.

[4] G. Antoniol, U. Villano, E. Merlo, , and M. D. Penta. Analyzing cloning evolution in the linux kernel. In *Information and Software Technology 44(13)*, 2002.

[5] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 86, Washington, DC, USA, 1995. IEEE Computer Society.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the 7th. Working Conference on Reverse Engineering*, pages 98–107, 2000.

[7] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[8] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Identifying clones in the linux kernel. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 92–100. IEEE Computer Society Press, 2001.

[9] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings ICSM'99: International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.

[10] A. E. Hassan and R. C. Holt. A reference architecture for web servers. In *Proceedings of WCRE 2000: Working Conference on Reverse Engineering*, 2000.

[11] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *The 8th IASTED International Conference on Software Engineering and Applications(SEA 2004)*, pages 222–229, 2004.

[12] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintanence*, pages 120–126, 1994.

[13] J. H. Johnson. Visualizing textual redundancy in legacy source. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 32. IBM Press, 1994.

[14] J. H. Johnson. Navigating the textual redundancy web in legacy source. In *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 16. IBM Press, 1996.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering 8(7)*, pages 654–670. IEEE Computer Society Press, 2002.

[16] C. Kapser and M. W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *Evolution of Large Scale Industrial Software Architectures*, 2003.

[17] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *Proc. of 2004 International Workshop on Principles of Software Evolution (IWPSE-04)*, pages 85–94, 2004.

[18] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *The Proceedings of the International Symposium of Empirical Software Engineering*, 2004.

[19] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[20] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of Working Conference on Reverse Engineering*, pages 44–55. IEEE Computer Society Press, 1997.

[21] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Autom. Softw. Eng.*, 3(1/2):77–108, 1996.

[22] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[23] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–253. IEEE Computer Society Press, 1996.

[24] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *11th Working Conference on Reverse Engineering (WCRE'04)*, pages 100–109.

[25] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the Eighth IEEE Symposium on Software Metrics*, pages 67–76. IEEE Computer Society Press, 2002.