Tools and Techniques for Analyzing Product and Process Data

Diomidis Spinellis
Department Management Science and Technology
Athens University of Economics and Business
Greece
email: dds@aueb.gr

Abstract

The analysis of data from software products and their development process is tempting, but often non-trivial. A flexible, extensible, scalable, and efficient way for performing this analysis is through the use of line-oriented textual data streams, which are the lowest useful common denominator for many software analysis tasks. Under this technique Unix tool-chest programs are combined into a pipeline that forms the pattern: fetching, selecting, processing, and summarizing. Product artefacts that can be handled in this way include source code (using heuristics, lexical analysis, or full-blown parsing and semantic analysis) as well as compiled code, which spans assembly code, machine code, byte code, and libraries. On the process front, data that can be analyzed include configuration management metadata, time series snapshots, and checked out repositories. The resulting data can then be visualized as graphs, diagrams, charts, and maps.

Keywords: repository mining, source code analysis, binary code analysis, visualization, Unix toolkit

1. Introduction

The analysis of data from software products and their development process [21] is tempting, but often non-trivial. It is tempting, because the software development

Diomidis Spinellis. Tools and techniques for analyzing product and process data. In Tim Menzies, Christian Bird, and Thomas Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 161–212. Morgan-Kaufmann, 2015. DOI: 10.1016/B978-0-12-411519-4.00007-0

process generates ample data that we should be able use in order to optimize it. Unfortunately, it is also difficult, because many tasks cannot be readily accomplished, despite the development of many platforms that collect software data and allow its analysis. Examples of such platforms and related tools include Hackystat [26], Hipikat [7], Kenyon [3], Evolizer [12], Sourcerer [35, 43], Tesseract [50], Moose [42], Churrasco [8], and Alitheia Core [16, 18]. The difficulty of using these platforms stems from a number of reasons. First, the software artifact or the type of analysis that is required may not be covered by an existing platform. Then, the disparate data sources, which are the norm when organizations build their development process in an organic way, may be difficult to combine using the analysis facilities provided by integrated development environments or platforms. Furthermore, the analysis to be performed may be highly specialized, involving an organization's domain-specific scenario or a novel research question. Finally, owing to the ease with which software process records data, the volume of the data to be examined can be enormous, making it difficult to scale some existing tools to handle real-world data.

Line-oriented textual data streams are the lowest useful common denominator for many software analysis tasks. Often, it is effective to combine Unix tool-chest programs [60] into a pipeline that forms the following pattern: fetching, selecting, processing, and summarizing. We examine this approach in Section 2, which builds on a previously published column [56].

In other cases scripting languages, such as Python, Ruby, and Perl, can also be remarkably effective. For research purposes, also worth looking at are platforms and projects that gather and analyze data from open source software repositories. These include FLOSSmole [24], Flossmetrics [22], Sourcerer, Alitheia Core, and GHTorrent [15], with FLOSSmole seeing the widest adoption outside the research team that produced it [41].

Many useful source code analysis tasks do not require implementing a full lexical analysis and parser, but can be performed using simple heuristics implemented through regular expressions [10]. In other cases, piggy-backing the analysis onto existing compiler front-ends can be just as effective (Section 3). Another useful technique involves processing compiled code. Object file symbols and Java byte code are two particularly rich sources of accurate data—see Section 4.

The best viewpoint into the software development process comes through data obtained from its configuration management (version control) system. This provides information regarding developers, progress, productivity, working hours, teamwork, and many other attributes. Three powerful analysis methods involve the processing of snapshots in a time series, revision logs, and the so-called blame listings. We examine these and more in Section 5.

Numerous tools can aid the exploratory data analysis and visualization. Notable ones include the GraphViz graph drawing tools [13], the GMT toolset for map drawing [68], the *gnuplot* plotting program, the R Project for statistical computing [47], and Python's diverse visualization libraries. Automating the publication of analyzed results can increase the researcher's productivity and aid the reproducibility of the results. We will see some examples of visualization tools in Section 6, which also loosely based on a previously published column [56].

2. A rational analysis pipeline

Line-oriented textual data streams are the lowest useful common denominator for a lot of data that passes through our hands. Such streams can represent program source code, feature requests, version control history, file lists, symbol tables, archive contents, error messages, profiling data, and so on. For many routine, everyday tasks, we might be tempted to process the data using a Swiss army knife scripting language, such as Perl, Python, or Ruby. However, to do that we often need to write a small, self-contained program and save it into a file. By that point we may have lost interest in the task, and end-up doing the work manually, if at all. Often, a more effective approach is to combine programs of the Unix toolchest into a short and sweet pipeline that we can run from our shell's command prompt. With the modern shell command-line editing facilities we can build our command bit by bit, until it molds into exactly the form that suits us. Nowadays, the original Unix tools are available on many different systems, such as GNU/Linux, Mac OS X, and Microsoft Windows (through Cygwin),[1] so there is no reason why we should not add this approach to our arsenal. Documentation for these tools is always available, either through the *man* command, or, often, by invoking the command with the `-help` option.

Many analysis one-liners that we will build around the Unix tools follow a pattern whose parts we will examine in the following sections. It that goes roughly like this: fetching (Section 2.1), selecting (Section 2.2), processing (Section 2.3), and summarizing (Section 2.4). We will also need to apply some plumbing to join these parts into a whole (Section 2.5).

2.1. Getting the data

In many cases our data will be text (for instance source code) that we can directly feed to the standard input of a tool. If this is not the case, we need to adapt our data. If we are dealing with object files or executable programs, we will have to use a tool such as *nm* (Unix), *dumpbin* (Windows), or *javap* (Java) to dig into them. We examine these approaches in sections 4.2 and 4.4. If we are working with files grouped into an archive, then a command such as *tar*, *jar*, or *ar* will list the archive's contents. In addition, the *ldd* command will print shared library dependencies associated with Unix executables, object files, and libraries. If our data comes from a (potentially large) collection of files stored on locally accessible storage *find* can locate those that interest us. Here is how we would use the *find* command to list the (header) files residing in the directory `/usr/include`.[2]

find /usr/include −**type** f

```
/usr/include/pty.h
/usr/include/time.h
/usr/include/printf.h
/usr/include/arpa/nameser_compat.h
/usr/include/arpa/telnet.h
```

[1] `http://www.cygwin.com/`

[2] The text in the Roman font denotes the commands we would write at the Unix shell command-line prompt, (often ending in `$`), while the text in typewriter font is part of the command's output.

```
/usr/include/arpa/inet.h
[...]
```

On the other hand, to get our data over the web, we can use *wget* or *curl* (see Section 5.1). We can also use *dd* (and the special file `/dev/zero`), *yes* or *jot* to generate artificial data, perhaps for running a quick test or a benchmark. Finally, if we want to process a compiler's list of error messages, we will want to redirect its standard error to its standard output; the incantation `2>&1` will do this trick.

There are many other cases we have not covered here: relational databases, version control systems (see Section 5), mailing lists, issue management systems, telemetry data, and so on. A software system's issue management system (bugs) database can provide insights regarding a product's maturity and the adequacy of the resources devoted to it. Issues are often coupled with software changes allowing even more detailed analyses to be performed. A dynamic view of a system's operation can be obtained by analyzing software telemetry data. This can include precise user interaction metrics, crash dump reports [29], and server logs. Always keep in mind that we are unlikely to be the first one who needs the application's data converted into a textual format; therefore someone has probably already written a tool for that job. For example, the *Outwit* tool suite [51][3] can convert into a text stream data coming from the Windows clipboard, an ODBC source, the event log, or the Windows registry.

2.2. Selection

Given the generality of the textual data format, in most cases we will have on our hands more data than what we require. We might want to process only some parts of each row, or only a subset of the rows. To select a specific column from a line consisting of elements separated by white space or another field delimiter, we can use *awk* with a single `print $n` command. If our fields are of fixed width, then we can separate them using *cut*. And, if our lines are not neatly separated into fields, we can often write a regular expression for a *sed* substitute command to isolate the element we want.

The workhorse for obtaining a subset of the rows is *grep*. We can specify a regular expression to get only the rows that match it, and add the `--invert-match`[4] flag to filter out rows we do not want to process.

Here is how we could use *grep* to list lines in the FreeBSD kernel source code file `vfs_subr.c` containing the `XXX` sequence, which is commonly used to flag questionable code. The first part of the fetch-selection pipeline uses *curl* to fetch the corresponding file from the FreeBSD repository. The backslash at the end of the line indicates that the line is continued on the one below, containing the URL where the file resides. The `|` (pipeline) symbol specifies that the output of *curl* (the `vfs_subr.c` file's contents) will be sent for further processing to the command following it, which is *grep*.

curl −−silent \

[3] `http://www.spinellis.gr/sw/outwit`

[4] In the interest of readability, the example use the GNU non-standard long form of the command flags.

https :// svnweb.freebsd.org/base/**head**/sys/kern/vfs_subr.c?view=**co** |
grep XXX

```
 * XXX desiredvnodes is historical cruft and should not exist.
     * XXX We could save a lock/unlock if this was only
   * Wait for I/O to complete.  XXX needs cleaning up.  The vnode can
      if (bp->b_bufobj != bo) {       /* XXX: necessary ? */
    * XXX Since there are no node locks for NFS, I
  vp = bp->b_vp;          /* XXX */
  vp = (*bo)->__bo_vnode; /* XXX */
               /* XXX audit: privilege used */
/* XXX - correct order? */
[...]
```

We can use the *grep* flags `--files-with-matches` and `--files-without-match` to obtain only the names of files that contain (or do not contain) a specific pattern. We can *fgrep* with the `--file` flag if the elements we are looking for are fixed strings stored in a file (perhaps generated in a previous processing step). If our selection criteria are more complex, we can often express them in an *awk* pattern expression. Many times we will find ourselves combining a number of these approaches to obtain the result that we want. For example, we might use *grep* to get the lines that interest us, `grep -invert-match` to filter-out some noise from our sample, and finally *awk* to select a specific field from each line.

Many examples in this chapter use *awk* for some of their processing steps. In general, *awk* works by applying a recipe we give it as an argument on each line of its input. This recipe consists of patterns and actions; actions without a pattern apply to all input lines, and a pattern without an action will print the corresponding line. A pattern can be a `/`-delimited regular expression or an arbitrary boolean expression. An action consists of commands enclosed in braces. Lines are automatically split into space-separated fields. (The `-F` option can be used to specify arbitrary field delimiters.) These fields are then available as variables named `$`*n*. For example, the following shell command will print the names of header files included by the C files in the current directory.

awk '/#include/ { print $2}' *.c

2.3. Processing

Data processing frequently involves sorting our lines on a specific field. The *sort* command supports tens of options for specifying the sort keys, their type, and the output order. Having our results sorted we then often want to count how many instances of each element we have. The *uniq* command with the `-count` option, will do the job here; often we will post-process the result with another instance of *sort*, this time with the `--numeric` flag specifying a numerical order, to find out which elements appear most frequently. In other cases we might want to compare results between different runs. We can use *diff* if the two runs generate results that should be the similar (perhaps we are comparing two versions of the file), or *comm* if we want to compare two sorted lists. Through *comm* we can perform set intersection and difference operations. To piece together results coming from unrelated processing steps based on a key, we can

first sort them and then apply the *join* command on the two lists. We can handle more complex tasks using, again *awk*.

Here are one-by-one the steps of how to build a pipeline that generates a list of header files ordered by the number of times they are included. First, use *grep* to obtain a list of include directives.

grep −−no−filename '^#include' ∗.c

```
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/exec.h>
#include <sys/imgact.h>
#include <sys/imgact_aout.h>
#include <sys/kernel.h>
#include <sys/lock.h>
[...]
```

Then, use *awk* to get from each line the included file name, which is the second field. While at it, we can replace the original *grep* with an *awk* selection pattern.

awk '/^#include/ { print $2}' ∗.c

```
<sys/cdefs.h>
<sys/param.h>
<sys/exec.h>
<sys/imgact.h>
<sys/imgact_aout.h>
<sys/kernel.h>
[...]
```

Our next step is to sort the file names, in order to bring the same ones together, so that the can be counted with *uniq*.

awk '/^#include/ { print $2}' ∗.c |
sort

```
"clock_if.h"
"cpufreq_if.h"
"linker_if.h"
"linker_if.h"
"linker_if.h"
"opt_adaptive_lockmgrs.h"
"opt_adaptive_mutexes.h"
"opt_alq.h"
[...]
```

We then use *uniq* to count same consecutive lines (file names).

awk '/^#include/ { print $2}' ∗.c |
sort |
uniq −−count

```
   1 "clock_if.h"
   1 "cpufreq_if.h"
   3 "linker_if.h"
   1 "opt_adaptive_lockmgrs.h"
   1 "opt_adaptive_mutexes.h"
   1 "opt_alq.h"
```

```
   1 "opt_bus.h"
  30 "opt_compat.h"
   1 "opt_config.h"
  34 "opt_ddb.h"
[...]
```

The final step involves sorting the output again, this time in reverse numerical order, in order to obtain a list of header file names in a descending order according to the number of times they occurred in the source code.

```
awk '/^#include / { print $2}' *.c |
sort |
uniq --count |
sort --reverse --numeric
 162 <sys/cdefs.h>
 161 <sys/param.h>
 157 <sys/systm.h>
 137 <sys/kernel.h>
 116 <sys/proc.h>
 114 <sys/lock.h>
 106 <sys/mutex.h>
  94 <sys/sysctl.h>
[...]
```

2.4. Summarizing

In many cases the processed data is too voluminous to be of use. For example, we might not care which symbols are defined with the wrong visibility in a program, but we might want to know how many there are. Surprisingly, many problems involve simply counting the output of the processing step using the humble *wc* (word count) command and its `--lines` flag. Here is again a preceding example, this time counting the number of lines containing the characters `XXX`.

```
curl --silent \
https://svnweb.freebsd.org/base/head/sys/kern/vfs_subr.c?view=co |
grep XXX |
wc --lines
   20
```

If we want to know the top or bottom 10 elements of our result list, then we can pass our list through *head* or *tail*. To format a long list of words into a more manageable block that we can then paste in a document, we can use *fmt* (perhaps run after a *sed* substitution command tacks a comma after each element). Also, for debugging purposes we might initially pipe the result of intermediate stages through *more* or *less*, to examine it in detail. As usual, we can use *awk* when these approaches do not suit us; a typical task involves summing-up a specific field with a command like `sum += $3`. In other cases, we might use *awk*'s associative arrays to sum diverse elements.

2.5. Plumbing

All the wonderful building blocks we have described are useless without some way to glue them together. For this we will use the Bourne shell's facilities. First and foremost comes the pipeline (`|`), which allows us to send the output of one processing step as input to the next one, as we saw in the preceding examples. We can also redirect output into a file; this is done by ending our command with the *>file-name* construct. In other cases we might want to execute the same command with many different arguments. For this we will pass the arguments as input to *xargs*. A typical pattern involves obtaining a list of files using *find*, and processing them using *xargs*. Commands that can only handle a single argument can be run by *xargs* if we specify the `--max-args=1` flag. If our processing is more complex, we can always pipe the arguments into a `while read` loop. (Amazingly, the Bourne shell allows us to pipe data into and from all its control structures). When everything else fails, we can use a couple of intermediate files to juggle our data.

Note that by default the Unix shell will use spaces to separate command line arguments. This can cause problems when we process file names that contain spaces in them. Avoid this by enclosing variables that represent a file name in double quotes, as in the following (contrived) example that will count the number of lines in the Java files residing in the directories under `org/eclipse`.

```
find org/eclipse -type f -name \*.java -print |
while read f
do
  cat "$f"        # File name in quotes to protect spaces
done | wc --lines
```

When using *find* with *xargs*, which is more efficient than the loop in the preceding example, we can avoid the problem of embedded spaces by using the respective arguments `-print0` and `--null`. These direct the two commands to have file names separated with a null character, instead of a space. Thus, the preceding example would be written as

```
find org/eclipse -type f -name \*.java -print0 |
xargs --null cat |
wc --lines
```

3. Source code analysis

Source code can be analyzed with various levels of accuracy, precision, and detail. The analysis spectrum spans heuristics, lexical analysis, parsing, semantic analysis, and static analysis.

3.1. Heuristics

Heuristics allow us to easily obtain rough-and-ready metrics from the code. The main advantage of heuristics in source code analysis is that they are easy, often trivial, to implement. They can therefore often be used as a quick way to test a hypothesis. In

most cases the use of heuristics entails the use of regular expressions and corresponding tools, such as the family of the Unix *grep* programs, to obtain measures that are useful, but not 100% accurate. For instance, the following command will display the number of top-level classes defined in a set of Java files.

grep −−count ^class ∗.**java**

The heuristic employed here is based on the assumption that the word `class` appears in the beginning of a line if and only if it is used to define a top-level class. Similarly, the number of subclasses defined in a set of files can be found through the following command.

fgrep −−count −−word−regexp extends ∗.**java**

Again, the preceding command assumes that the word `extends` is only used to refer to a subclass's base class. For example, the count can be set off by the word `extends` appearing in strings or comments. Finally, if the files were located in various folders in a directory tree, we could use the *grep*'s `--recursive` flag, instructing it to traverse the directory tree starting from the current directory (denoted by a dot). An invocation of *awk* can then be used to sum the counts (the second field of the colon-separated line.

grep −−recursive −−count ^class . |
awk −F: '{s += $2} END {print s}'

The preceding command assumes that the keyword `class` always appears at the beginning of a line, and that no other files that might contain lines beginning with the word `class` are located within the directory hierarchy.

3.2. Lexical analysis

When more accuracy is required than what a heuristic can provide, or when the analysis cannot be easily expressed through a heuristic, then flow-blown lexical analysis has to be performed. This allows us to identify reserved words, identifiers, constants, string literals, and rudimentary properties of the code structure. The options here include expressing the lexical analyzer as a state machine or creating code with a lexical analyzer generator.

3.2.1. State machines

A hand-crafted state machine can be used for recognizing strings and comments, taking into account a language's escaping rules. As an example, the following states can be used for recognizing various elements of C++ source code.

```
enum e_cfile_state {
        s_normal,
        s_saw_slash,            // After a / character
        s_saw_str_backslash,    // After a \ character in a string
        s_saw_chr_backslash,    // After a \ character in a character
        s_cpp_comment,          // Inside C++ comment
        s_block_comment,        // Inside C block comment
        s_block_star,           // Found a * in a block comment
```

```
        s_string,               // Inside a string
        s_char,                 // Inside a character
};
```

Given the preceding definition, a state machine that processes single characters counting the number of characters appearing within character strings can be expressed as follows.

```
static void
process(char c)
{
    static enum e_cfile_state cstate = s_normal;

    switch (cstate) {
    case s_normal:
        if (c == '/')
            cstate = s_saw_slash;
        else if (c == '\'')
            cstate = s_char;
        else if (c == '"') {
            cstate = s_string;
            n_string++;
        }
        break;
    case s_char:
        if (c == '\'')
            cstate = s_normal;
        else if (c == '\\')
            cstate = s_saw_chr_backslash;
        break;
    case s_string:
        if (c == '"')
            cstate = s_normal;
        else if (c == '\\')
            cstate = s_saw_str_backslash;
        break;
    case s_saw_chr_backslash:
        cstate = s_char;
        break;
    case s_saw_str_backslash:
        cstate = s_string;
        break;
    case s_saw_slash:           // After a / character
        if (c == '/')
            cstate = s_cpp_comment;
        else if (c == '*')
            cstate = s_block_comment;
```

```
            else
                cstate = s_normal;
            break;
        case s_cpp_comment:     // Inside a C++ comment
            if (c == '\n')
                cstate = s_normal;
            break;
        case s_block_comment:       // Inside a C block comment
            if (c == '*')
                cstate = s_block_star;
            break;
        case s_block_star:          // Found a * in a block comment
            if (c == '/')
                cstate = s_normal;
            else if (c != '*')
                cstate = s_block_comment;
            break;
        }
}
```

Given that the preceding code has a precise picture of what type of lexical elements it processes, it can be easily extended to count more complex elements, such as the number or nesting level of blocks.

The driver for the `process` function could be a simple filter-style program that will report the number of strings contained in the code provided in its standard input.

```
#include <stdio.h>

static int n_string;
static void process(char c);

int
main(int argc, char *argv[])
{
    int c;

    while ((c = getchar()) != EOF)
        process(c);
    printf("%d\n", n_string);
    return 0;
}
```

Running the driver with its source code as input, will report 1 (one string found) on its standard output.

```
count <count.c
1
```

3.2.2. Lexical analyzer generator

For heavy lifting a lexical analyzer generator [34], such as *lex* or its modern open-source incarnation *flex*, can be used to identify efficiently and accurately all of a language's tokens. The following code excerpt can be fed to the *lex* generator to create a self-standing program that will count the number of times each C lexical token has appeared in its standard input.

```
LET [a-zA-Z_]
DIG [0-9]

%{
int n_auto, n_break, n_case, n_char, n_const;
int n_volatile , n_while,  n_identifier ;
// [...]

%}

%%
"auto"                  { n_auto++; }
"break"                 { n_break++; }
"case"                  { n_case++; }
"char"                  { n_char++; }
"const"                 { n_const++; }
// [...]
"while"                 { n_while; }

{LET}({LET}|{DIG})*     {  n_identifier ++; }

">>="                   {  n_right_shift_assign ++; }
"<<="                   {  n_left_shift_assign ++; }
// [...]
">>"                    {  n_right_shift ++; }
"<<"                    {  n_left_shift ++; }
// [...]
"<="                    { n_less_than++; }
">="                    { n_greater_than++; }
"=="                    { n_compare++; }
// [...]
"="                     { n_assign++; }
.                       { /* ignore other characters */ }

%%

yywrap() { return(1); }

main()
```

```
{
        while (yylex())
                ;
        printf("auto %d\n", n_auto);
        printf("break %d\n", n_break);
        // [...]
}
```

The lexical analyzer specification begins with the definition of regular expressions for C letters (`LET`) and digits (`DIG`). Then come the C definitions of the counter variables, which are enclosed in the `%{ %}` block. The analyzer's main body, which starts after the `%%` line, consists of regular expressions on the left hand side, followed by C code in braces on the right hand side. The C code is executed when the program's input matches the corresponding regular expression. The lexical analysis specification can be easily modified to handle other languages and types of input. Note that because the specified regular expressions are matched in the order specified, longer elements and more specific elements must be specified before the corresponding shorter or more general ones. This can be clearly seen in the presented example in the handling of identifiers and operators.

The C code after the second `%%` line contains a loop to iterate over all input tokens, and statements to print the collected figures. This allows the generated code to be compiled and run as a single program. The program assumes that the code it reads has already been preprocessed to handle preprocessing commands and comments. This can be easily done by passing the source code through the C preprocessor, *cpp*.

3.3. Parsing and semantic analysis

Parsing and semantic analysis [1] is required when we want to extract more sophisticated measures from the code, involving, for instance, the scoping of identifiers, the handling of exceptions, and class hierarchies. Most modern languages are large complex beasts, and therefore this form of processing is not for the faint-hearted. The effort involved can easily require writing tens of thousands lines of code. Therefore, if this level of analysis is required it is best to adapt the code of an existing compiler. Compilers for most languages are available as open source software, and can therefore can be modified to perform the requisite analysis.

An interesting case is the LLVM platform [33] and in particular its Clang font-end, which can be used as a library to parse and analyze C-like languages, such as C, C++, and Objective C. For instance, we can build an analyzer that will print a C program's global variable declarations in about 100 lines of C++ code.[5]

3.4. Third party tools

A final option to analyze source code is to piggy-back third party tools that analyze code. Here are some tools and ideas on how to use them.

[5] `https://github.com/loarabia/Clang-tutorial/blob/master/CItutorial6.cpp`

Function Metrics

Number of elements: 229

Metric	Total	Min	Max	Avg
Number of characters	92771	13	3686	405.114
Number of comment characters	7114	0	482	31.0655
Number of space characters	20762	2	1062	90.6638
Number of line comments	0	0	0	0
Number of block comments	295	0	14	1.28821
Number of lines	4247	1	185	18.5459
Maximum number of characters in a line	10519	9	107	45.9345
Number of character strings	344	0	32	1.50218
Number of unprocessed lines	0	0	0	0
Number of C preprocessor directives	0	0	0	0
Number of processed C preprocessor conditionals (ifdef, if, elif)	0	0	0	0
Number of defined C preprocessor function-like macros	0	0	0	0
Number of defined C preprocessor object-like macros	0	0	0	0
Number of preprocessed tokens	28102	4	1084	122.716
Number of compiled tokens	33064	0	1930	144.384
Number of statements or declarations	3214	0	153	14.0349
Number of operators	4509	0	176	19.69
Number of unique operators	1153	0	19	5.03493
Number of numeric constants	988	0	64	4.31441
Number of character literals	369	0	67	1.61135
Number of if statements	599	0	32	2.61572
Number of else clauses	179	0	21	0.781659
Number of switch statements	25	0	2	0.10917
Number of case labels	189	0	28	0.825328
Number of default labels	21	0	2	0.0917031
Number of break statements	111	0	16	0.484716
Number of for statements	105	0	9	0.458515
Number of while statements	28	0	4	0.122271
Number of do statements	8	0	2	0.0349345
Number of continue statements	9	0	3	0.0393013
Number of goto statements	9	0	2	0.0393013
Number of return statements	263	0	13	1.14847
Number of project-scope identifiers	1887	0	98	8.24017
Number of file-scope (static) identifiers	476	0	47	2.0786
Number of macro identifiers	1081	0	102	4.72052
Total number of object and object-like identifiers	7321	0	270	31.9694
Number of unique project-scope identifiers	938	0	36	4.09607
Number of unique file-scope (static) identifiers	318	0	27	1.38865
Number of unique macro identifiers	676	0	34	2.95197
Number of unique object and object-like identifiers	2444	0	68	10.6725
Number of global namespace occupants at function's top	166333	0	1070	726.345
Number of parameters	330	0	6	1.44105
Maximum level of statement nesting	283	0	7	1.23581
Number of goto labels	7	0	2	0.0305677
Fan-in (number of calling functions)	719	0	61	3.13974
Fan-out (number of called functions)	881	0	34	3.84716
Cyclomatic complexity (control statements)	994	1	36	4.34061
Extended cyclomatic complexity (includes branching operators)	1196	1	42	5.22271
Maximum cyclomatic complexity (includes branching operators and all switch branches)	1360	1	69	5.93886
Structure complexity (Henry and Kafura)	440199	0	238144	1922.27
Halstead complexity	72126.2	0	3416.45	314.961
Information flow metric (Henry and Selig)	2.51256e+006	0	1.5552e+006	10971.9

Figure 1: CScout-derived function metrics for the *awk* source code.

The *CScout* program is a source code analyzer and refactoring browser for collections of C programs [59]. It can process workspaces of multiple projects (think of a project as a collection of C source files that are linked together) mapping the complexity introduced by the C preprocessor back into the original C source code files. CScout takes advantage of modern hardware (fast processors and large memory capacities) to analyze C source code beyond the level of detail and accuracy provided by current compilers, linkers, and other source code analyzers. The analysis CScout performs takes into account the identifier scopes introduced by the C preprocessor and the C language proper scopes and namespaces. After the source code analysis CScout can process sophisticated queries on identifiers, files, and functions, locate unused or wrongly-scoped identifiers, and compute many metrics related to files, functions, and identifiers. Figure 1 illustrates the metrics collected for functions.

CCFinderX[6] is a tool that detects duplicated code fragments in source code written in many modern programming languages. The tool is a redesign of CCFinder [28], which has been used for research published in tens of research papers. The command line version of the tool will print its results as a text file.

The output file format of *CCFinderX* is simple, but not trivial. Its first section lists for each file that has been analyzed, its numerical identifier, its path, and the number of tokens it contains. Here is an excerpt corresponding to the Linux kernel.

```
source_files {
...
19 arch/i386/kernel/bootflag.c 314
20 arch/i386/kernel/cpuid.c 841
21 arch/i386/kernel/i8237.c 147
22 arch/i386/kernel/microcode.c 1673
23 arch/i386/kernel/msr.c 1287
24 arch/i386/kernel/quirks.c 154
25 arch/i386/kernel/topology.c 133
26 arch/i386/mm/hugetlbpage.c 1084
27 arch/i386/oprofile/backtrace.c 310
28 arch/i386/oprofile/init.c 67
...
}
```

Then follows a list of detected code clones. Each line contains the clone's identifier, followed by a pair of source code clone specifications. Each one has the file identifier, the beginning token, and the end token of the cloned code.

In the following example we see code cloned within the same file (`microcode.c` — clone 7329), as well as code cloned between different files (e.g. `cpuid.c` and `msr.c` — clone 6981).

```
clone_pairs {
...
6981 20.785-840 23.1231-1286
10632 20.625-690 934.1488-1553
7329 22.660-725 22.884-949
...
}
```

[6] `http://www.ccfinder.net/`

The generated file can be further analyzed to derive additional measures. As an example, the following Perl program when given as input the base name of a CCFinderX result file, will print the percentage of cloned tokens in it. A high percentage of cloning can often lead to higher maintenance costs, because fixes and enhancements need to be carefully duplicated in multiple places.

```
open(IN, "ccfx.exe P $ARGV[0].ccfxd|") || die;
while (<IN>) {
  chop;
  if (/^ source_files / .. /^\}/) {
    # Initialize file map as non-cloned tokens
    ($id, $name, $tok) = split;
    $file[$id][$tok - 1] = 0 if ($tok > 0);
    $nfile++;
  } elsif (/^ clone_pairs / .. /^\}/) {
    # Assign clone details to corresponding files
    ($id, $c1, $c2) = split;
    mapfile($c1);
    mapfile($c1);
  }
}

# Given a detected clone, mark the corresponding tokens
# in the file map as cloned
sub mapfile {
  my($clone) = @_;
  ($fid, $start, $end) = ($clone =~ m/^(\d+)\.(\d+)-(\d+)$/);
  for ($i = $start; $i <= $end; $i++) {
    $file[$fid][$i] = 1;
  }
}

# Sum up the number of tokens and clones
for ($fid = 0; $fid <= $#file; $fid++) {
  for ($tokid = 0; $tokid <= $#{$file[$fid]}; $tokid++) {
    $ntok++;
    $nclone += $file[$fid][$tokid];
  }
}

print "$ARGV[0] nfiles=$nfile ntok=$ntok nclone=$nclone ",
    $nclone / $ntok * 100, "\n";
```

General-purpose tools can often be just as helpful as the specialized ones we have seen. If we want to perform some processing on comments in C, C++, or Objective-C code, then the GCC version of the C preprocessor can help us. Consider a case where we want to count the number of comment characters in a source code file. Preprocess-

ing the file with the `-fpreprocessed` flag will remove the comments, but won't do any other expansion. Thus subtracting the number of characters in the file with the comments removed from the original number will give us the number of comment characters. The following *bash* code excerpt will print the number of comment characters in `file.c`.

expr $(**wc** −−chars <prog.c) − $(**cpp** −fpreprocessed prog.c | **wc** −−chars)

We can also pass the `-H` flag to the C preprocessor in order to obtain a list of included header files. This output can, for instance, then be used to map code reuse patterns. Here is some representative output. (The dots at the beginning of each line indicate nested include levels, and can be used to study a project's module layering.)

```
. /usr/include/stdlib.h
.. /usr/include/machine/ieeefp.h
.. /usr/include/_ansi.h
... /usr/include/newlib.h
... /usr/include/sys/config.h
.... /usr/include/machine/ieeefp.h
.... /usr/include/sys/features.h
.... /usr/include/cygwin/config.h
.. /usr/lib/gcc/i686-pc-cygwin/4.8.2/include/stddef.h
.. /usr/include/sys/reent.h
... /usr/include/_ansi.h
... /usr/lib/gcc/i686-pc-cygwin/4.8.2/include/stddef.h
... /usr/include/sys/_types.h
.... /usr/include/machine/_types.h
..... /usr/include/machine/_default_types.h
.... /usr/include/sys/lock.h
.... /usr/lib/gcc/i686-pc-cygwin/4.8.2/include/stddef.h
```

Another useful family of general-purpose tools that we can repurpose for source code analysis are documentation generators, such as *Doxygen* and *Javadoc*. These parse source code and documentation comments to create code reference documents. The simplest way to use these tools is to analyze the resultant HTML text. The text's structure is simpler than the corresponding code, and, in addition, it may contain data that would be difficult to get from the original code. The trick in this case is to look at the generated HTML code (right-click – This Frame – View Source in many browsers) to determine the exact pattern to search for. For example, the following shell code will go through the Java development kit HTML documentation to count the number of methods that are declared to implement some interface (7752 methods in total).

grep −−recursive −−count 'Specified by:' . |
awk −F: '{s += $2} END {print s}'

If the generated documentation does not contain the information we want, then we can extend *Javadoc* through custom so-called *doclets*. These have a method that is called after *Javadoc* has processed the source code. The method gets as an argument a document tree of the code's element, which can then be easily processed to extract and print the results we want. As an example the *UMLGraph* system uses this approach to create UML diagrams out of Java code [53].

Regarding the analysis of the code's adherence to some style conventions, a useful approach is to apply a source code formatter, such as *indent*, on the source code, and then compare the original source code with the formatter's output. The number of differences found is an indication of how closely the code follows the code style conventions: a large number of differences indicates a poor adherence to the style conventions. A problem of this approach is that for some languages, such as C and C++, there are many acceptable style conventions. In these cases, either the code style tool has to be configured according to the documented code conventions, or the code's conventions have to be deduced from the actual code [58].

The following shell script, will deduce the code's conventions by perturbing the (far too many) settings passed to *indent*, and keeping each time the setting that minimizes the number of lines that do not match the specified style. After it is executed with `FILES` set to a (hopefully representative) set of files on which to operate, it will set the variable `INDENT_OPT` to the *indent* options that match the code's style more closely.

```
# Return number of style violations when running indent on
# $FILES with $INDENT_OPT options
style_violations ()
{
        for f in $FILES
        do
                indent -st $INDENT_OPT $1 $f |
                diff $f -
        done |
        grep '^<' |
        wc --lines
}

VIOLATIONS=`style_violations`

# Determine values for numerical options
for TRY_OPT in i ts bli c cbi cd ci cli cp d di ip l lc pi
do
        BEST=$VIOLATIONS
        # Find best value for $TRY_OPT
        for n in 0 1 2 3 4 5 6 7 8
        do
                NEW=`style_violations -$TRY_OPT$n`
                if [ $NEW -lt $BEST ]
                then
                        BNUM=$n
                        BEST=$NEW
                fi
        done
        if [ $BEST -lt $VIOLATIONS ]
        then
```

```
                INDENT_OPT="$INDENT_OPT -$TRY_OPT$BNUM"
                VIOLATIONS=$BEST
        fi
done

# Determine Boolean options
for TRY_OPT in bad bap bbb bbo bc bl bls br brs bs cdb cdw ce cs bfda \
    bfde fc1 fca hnl lp lps nbad nbap nbbo nbc nbfda ncdb ncdw nce \
    ncs nfc1 nfca nhnl nip nlp npcs nprs npsl nsaf nsai nsaw nsc nsob \
    nss nut pcs prs psl saf sai saw sc sob ss ut
do
        NEW=`style_violations -$TRY_OPT`
        if [ $NEW -lt $VIOLATIONS ]
        then
                INDENT_OPT="$INDENT_OPT -$TRY_OPT"
                VIOLATIONS=$NEW
        fi
done
```

Running *indent* on the Windows Research Kernel without any options results in 389,547 violations found among 583,407 lines. After determining the appropriate *indent* options with the preceding script (`-i4 -ts0 -bli0 -c0 -cd0 -di0 -bad -bbb -br -brs -bfda -bfde -nbbo -ncs`) the number of lines found to be violating the style conventions shrinks to 118,173. This type of analysis can pinpoint, for example, developers and teams that require additional mentoring or training to help them adhere to an organization's code style standards. An increase of these figures over time can be an indicator of stress in an organization's development processes.

4. Compiled code analysis

Analyzing the artifacts of compilation (assembly language code, object files, and libraries) has the obvious advantage that the compiler performs all the heavy lifting required for the analysis. Thus, the analysis can be efficiently performed and its results will accurately match the actual semantics of the language. In addition, this analysis can be performed on proprietary systems, repositories of binary code, such as those of the Maven ecosystem [39], and also on mixed code bases where an application's source code is shipped together with library binaries. The following sections list tools and corresponding examples.

4.1. Assembly language

Most compilers provide a switch that directs them to produce assembly language source code, rather than binary object code. The corresponding flag for most Unix compilers is `-S`. Assembly language files can be easily processed using text tools, such as *grep*, *sed*, and *awk*. As an example, we will see a script that counts the code's basic blocks.

A *basic block* is a portion of code with exactly one entry and exit point. It can be valuable to analyze code in terms of basic blocks, for it allows to measure things like code complexity and test coverage requirements.

We can obtain information regarding the basic blocks of GCC-compiled code by passing to the compiler the `--coverage` flag in conjunction with the `-s` flag to produce assembly language output. The generated code at the entry or exit of a basic block looks like the following excerpt (without the comments).

```
movl ___gcov0. stat_files +56, %eax    ; Load low part of 64−bit value
movl ___gcov0. stat_files +60, %edx    ; Load hight part of 64−bit value
addl $1, %eax                          ; Increment low part
adcl $0, %edx                          ; Add carry to high part
movl %eax, ___gcov0. stat_files +56    ; Store low part
movl %edx, ___gcov0. stat_files +60    ; Store high part
```

From the above code it is easy to see that the counter associated with each basic block occupies 8 data bytes. The compiler stores the counters in common data blocks allocated on a per-function basis, like the ones in the following example obtained by analyzing a small C program.[7]

```
.lcomm ___gcov0.execute_schedule,400,32
.lcomm ___gcov0.prunefile,48,32
.lcomm ___gcov0.bytime,8,8
.lcomm ___gcov0.print_schedule,24,8
.lcomm ___gcov0.create_schedule,184,32
.lcomm ___gcov0.parse_dates,80,32
.lcomm ___gcov0. stat_files ,72,32
.lcomm ___gcov0.xstrdup,16,8
.lcomm ___gcov0.xmalloc,24,8
.lcomm ___gcov0.D,8,8
.lcomm ___gcov0.main,816,32
.lcomm ___gcov0.error_pmsg,40,32
.lcomm ___gcov0.error_msg,24,8
.lcomm ___gcov0.usage,24,8
```

The three arguments to each `lcomm` pseudo-op are the block's name, its size, and alignment. By dividing the size by 8 we can obtain the number of basic block boundaries associated with each function. Thus we can process a set of assembly language files produced by compiling code with the `--coverage` option, and then use the following script to obtain a list of functions ordered by the number of basic blocks boundaries embedded in them.

```
# Compile code with coverage analysis
gcc −S −−coverage −o /dev/stdout file .c |

# Print the blocks where coverage data is stored
```

[7] `http://www.spinellis.gr/sw/unix/fileprune/`

```
sed --quiet ' /^\. lcomm ___gcov0/s /[.,]/ /gp' |

# Print name and size of each block
awk '{ print $3, $4 / 8}' |

# Order by ascending block size
sort --key=2 --numeric
```

Here is an example of the script's output for the preceding program.

```
D 1
bytime 1
xstrdup 2
error_msg 3
print_schedule 3
usage 3
xmalloc 3
error_pmsg 5
prunefile 6
stat_files 9
parse_dates 10
create_schedule 23
execute_schedule 50
main 102
```

The number of basic blocks in each function can be used to assess code structure, modularity, and (together with other metrics) locate potential trouble spots.

4.2. Machine Code

On Unix systems we can analyze object files containing machine code with the *nm* program.[8] This displays a list of defined and undefined symbols in each object file passed as an argument [52, pp. 363–364]. Defined symbols are preceded by their address, and all symbols are preceded by the type of their linkage. The most interesting symbol types found in user-space programs in a hosted environment are the following.

B Large uninitialized data; typically arrays

C Uninitialized "common" data; typically variables of basic types

D Initialized data

R Read-only symbols (constants and strings)

T Code (known as text)

U Undefined (imported) symbol (function or variable)

Lowercase letter types (e.g. "`d`" or "`t`") correspond to symbols that are defined locally in a given file (with a `static` declaration in C/C++ programs).

Here is as an example the output of running *nm* on a C "hello, world" program.

[8] A program with similar functionality, named *dumpbin*, is also distributed with Microsoft's Visual Studio.

```
00000000 T main
         U printf
```

As a first example of this technique consider the task of finding all symbols that a (presumably) large C program should have declared as `static`. Appropriate `static` declarations, minimize name space pollution, increase modularity, and can prevent bugs that might be difficult to locate. Therefore, the number of elements in such a list could be a metric of a project's maintainability.

```
# List of all undefined (imported) symbols
nm *.o | awk '$1 == "U" { print $2}' >imports

# List of all defined globally exported symbols
nm *.o | awk 'NF == 3 && $2 ~ /[A-Z]/ {print $3}' | sort >exports

# List of all symbols that were globally exported but not imported
# (-2: don't show only imports, -3: don't show common symbols)
comm -2 -3 exports imports
```

Our second example derives identifier metrics according to their type. The script we will see can analyze systems whose object files reside in a directory hierarchy, and therefore uses *find* to locate all object files. After listing the defined symbols with *nm*, it uses *awk* to tally in an associative array (map) the count and total length of the identifiers of each identifier category. This allows it in the end to print the average length and count for each category.

```
# Find object files
find . -name \*.o |

# List symbols
xargs nm |

awk '
  # Tally data for each symbol type
  NF == 3 {
    len[$2] += length($3)
    count[$2]++
  }
  END {
    # Print average length for each type
    for (t in len)
      printf "%s %4.1f %8d\n", t, len[t] / count[t], count[t]
  }' |

# Order by symbol type
sort --ignore-case
```

Running the preceding script on a compiled kernel of FreeBSD produces the following results.

```
A  6.0        5
b 13.2     2306
B 16.7     1229
C 10.9     2574
D 19.1     1649
d 23.0    11575
R 13.2      240
r 40.9     8244
T 17.5    12567
t 17.9    15475
V 17.0        1
```

From these results we can see that in each category there are more local (static) symbols (identified by a lowercase letter) than global ones (shown with the corresponding uppercase letter), and that global functions (T) are far more (12567) than global variables (D: 1649) and arrays (B: 1229). This allows us to reason regarding the encapsulation mechanisms used in the specific system.

Binary code analysis can go into significantly more depth than the static analysis techniques we have discussed in other sections. The code sequences can be analyzed in considerably more detail, while dynamic analysis methods can be used to obtain information from running programs. Tool support can help in both regards; a recently published survey of available tools [37] provides a good starting point.

4.3. Dealing with name mangling

Some of the techniques covered in this section work well with code written in older languages, such as C and Fortran. However, if we try them on code written in relatively newer ones, such as Ada, C++, and Objective-C, we will get gibberish, as illustrated in the following example.

```
_ZL19visit_include_files6FileidMS_KFRKSt3mapIS_10IncDetails
St4lessIS_ESaISt4pairIKS_S1_EEEvEMS1_KFbvEi 53
_ZL9call_pathP12GraphDisplayP4CallS2_b 91
_ZL11cgraph_pageP12GraphDisplay 96
_ZL12version_infob 140
```

The reason for this is *name mangling*: a method C++ compilers use to reconcile linkers that were designed for simpler languages with the requirement of C++ for type-correct linking across separately compiled files. To achieve this feat the compiler adorns each externally visible identifier with characters that specify its precise type.

We can undo this mangling by passing the resulting text through the *c++filt* tool, which ships with GNU binutils. This will decode each identifier according to the corresponding rules, and provide the full language-dependent type associated with each identifier. For instance, the demangled C++ identifiers of the preceding example are the following.

```
visit_include_files(Fileid, std::map<Fileid, IncDetails,
std::less<Fileid>, std::allocator<std::pair<Fileid const,
IncDetails> > > const& (Fileid::*)() const, bool
(IncDetails::*)() const, int) 53
call_path(GraphDisplay*, Call*, Call*, bool) 91
cgraph_page(GraphDisplay*) 96
version_info(bool) 140
```

4.4. Byte Code

Programs and libraries associated with the JVM environment are compiled into portable byte code. This can be readily analyzed to avoid the complexity of analyzing source code. The *javap* program, which comes with the Java development kit, gets as an argument the name of a class, and, by default, prints its public, protected, and package-visible members. For instance, this is the *javap* output for a "hello, world" Java program.

```
Compiled from "Test.java"
class Hello {
  Hello();
  public static void main(java.lang.String[]);
}
```

Here is how we can use the output of *javap* to extract some basic code metrics of Java programs (in this case the *ClassGraph* class).

```
# Number of fields and methods in the ClassGraph class
javap org.umlgraph.doclet.ClassGraph |
grep '^  ' |
wc --lines

# List of public methods of a given class
javap -public org.umlgraph.doclet.ClassGraph |
sed --quiet '
  # Remove arguments of each method
  s/(.*/(/
  # Remove visibility and return type of each method; print its name
  s/^  .*  \([^(]*\)(/\1/ p'
```

The *javap* program can also disassemble the Java byte codes contained in each class file. This allows us to perform even more sophisticated processing. The following script prints virtual method invocations, ordered by the number of times each class invokes a method.

```
# Disassemble Java byte code for all class files under the current directory
javap -c **/*.class |

# Print (class method) pairs
awk '
  # Set class name
  /^[^ ].* class / {
    # Isolate from the line the class name
    # It is parenthesized in the RE so we can refer to it as \1
    class = gensub("^.* class ([^ ]*) .*", "\\1", "g")
  }
  # Print class and method name
  /: invokevirtual / {
    print class, $6
```

```
}' |

# Order same invocations together
sort |

# Count number of same invocations
uniq --count |

# Order results by number of same invocations
sort --numeric-sort --reverse
```

Running the above script on the UMLGraph's compiled method, allows us to determinate that the `org.umlgraph.doclet.Options` class calls 158 times the method `String.equals`. Measures like these can be used to build a dependency graph, which can then expose refactoring opportunities.

If the output of *javap* is too low-level for our purpose, another alternative for processing Java byte code is the *FindBugs* program [23]. This allows the development of plugins that are invoked when specific code patterns are encountered. For instance, a simple plugin can detect instances of `BigDecimal` types that are created from a `double` value,[9] while a more complex one can locate arbitrary errors in method arguments that can be determined at the time of the analysis [62].

4.5. Dynamic linking

Modern systems have their executable programs link dynamically at runtime to the libraries they require in order to run. This simplifies software updates, reduces the size on disk of each executable program, and allows the running programs to share each library's code in memory [57, p. 281]. Obtaining a list of the dynamic libraries a program requires, allows us to extract information regarding software dependencies and reuse.

On Unix systems the *ldd* program will provide a list of the libraries an executable program (or an other library) requires in order to run. Here is an example of running the *ldd* command on `ldd /bin/ls` on a Linux system.

```
ldd /bin/ls
  linux-gate.so.1 =>  (0xb7799000)
  libselinux.so.1 => /lib/i386-linux-gnu/libselinux.so.1 (0xb776d000)
  librt.so.1 => /lib/i386-linux-gnu/i686/cmov/librt.so.1 (0xb7764000)
  libacl.so.1 => /lib/i386-linux-gnu/libacl.so.1 (0xb7759000)
  libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb75f5000)
  libdl.so.2 => /lib/i386-linux-gnu/i686/cmov/libdl.so.2 (0xb75f1000)
  /lib/ld-linux.so.2 (0xb779a000)
  libpthread.so.0 => /lib/i386-linux-gnu/i686/cmov/libpthread.so.0 (0xb75d8000)
  libattr.so.1 => /lib/i386-linux-gnu/libattr.so.1 (0xb75d2000)
```

As usual, we can then process this output with additional tools to generate higher-level results. As an example, the following pipeline will list the libraries required by all

[9] `http://code.google.com/p/findbugs/wiki/DetectorPluginTutorial`

programs in the `/usr/bin` directory, ordered by the number of programs that depend on them. The pipeline's output can be used to study the dependencies between modules and software reuse.

```
# List dynamic library dependencies, ignoring errors
ldd /usr/bin/* 2>/dev/null |

# Print library name
awk '/=>/{ print $3}' |

# Bring names together
sort |

# Count same names
uniq --count |

# Order by number of occurrences
sort --reverse --numeric-sort
```

These are the first ten lines of the preceding pipeline's output on a FreeBSD system,

```
392 /lib/libc.so.7
 38 /lib/libz.so.5
 38 /lib/libm.so.5
 35 /lib/libncurses.so.8
 30 /lib/libutil.so.8
 30 /lib/libcrypto.so.6
 29 /lib/libcrypt.so.5
 22 /usr/lib/libstdc++.so.6
 22 /usr/lib/libbz2.so.4
 22 /lib/libmd.so.5
```

and these on a Linux system.

```
587 /lib/i386-linux-gnu/i686/cmov/libc.so.6
208 /lib/i386-linux-gnu/i686/cmov/libdl.so.2
148 /lib/i386-linux-gnu/i686/cmov/libm.so.6
147 /lib/i386-linux-gnu/libz.so.1
118 /lib/i386-linux-gnu/i686/cmov/libpthread.so.0
 75 /lib/i386-linux-gnu/libtinfo.so.5
 71 /lib/i386-linux-gnu/i686/cmov/librt.so.1
 41 /lib/i386-linux-gnu/libselinux.so.1
 38 /lib/i386-linux-gnu/libgcc_s.so.1
 38 /lib/i386-linux-gnu/i686/cmov/libresolv.so.2
```

4.6. Libraries

Library files contain compiled object files packed together so that they can be easily shipped and used as a unit. On Unix systems *nm* can be used to see the symbols defined and referenced by each library member (see Section 4.2), while the *ar* program can be used to list the files contained in the library. As an example, the following pipeline will list the C library's files ordered by their size.

```
# Print a verbose table for file libc.a
ar tvf libc.a |

# Order numerically by size (the third field)
sort --reverse --key=3 --numeric-sort
```

Here are the first few lines of the pipeline's output on a Linux system.

```
rw-r--r-- 2308942397/2397 981944 Dec 18 01:16 2013 regex.o
rw-r--r-- 2308942397/2397 331712 Dec 18 01:16 2013 malloc.o
rw-r--r-- 2308942397/2397 277648 Dec 18 01:16 2013 getaddrinfo.o
rw-r--r-- 2308942397/2397 222592 Dec 18 01:16 2013 strcasestr-nonascii.o
rw-r--r-- 2308942397/2397 204552 Dec 18 01:16 2013 fnmatch.o
rw-r--r-- 2308942397/2397 196848 Dec 18 01:16 2013 vfwprintf.o
```

Such a listing can provide insights on a library's modularity, and modules that could be refactored into units of a more appropriate size.

The corresponding program for Java archives in *jar*. Given the name of a Java `.jar` file it will list the class files contained in it. The results can then be used by other programs for further processing.

Consider the task of calculating the Chidamber and Kemerer metrics [6] for a set of classes. These metrics comprise for a class the following following values:

WMC : Weighted methods per class

DIT : Depth of inheritance tree

NOC : Number of children

CBO : Coupling between object classes

RFC : Response for a class

LCOM : Lack of cohesion in methods

The metrics can be used to assess the design of an object-oriented system and to improve the corresponding process.

The following example will calculate the metrics for the classes contained in the `ant.jar` file, and print the results ordered by the weighted methods per class. For the metrics calculation it uses the *ckjm* program [54],[10] which expects as its input pairs of files and class names.

```
# Print table of files contained in ant.jar
jar tf ant.jar |

# Add "ant.jar " to the beginning of lines ending with .class
# and print them, passing the (filename class) list to ckjm
# c metrics
```

[10] `http://www.spinellis.gr/sw/ckjm/`

```
sed --quiet '\. class$/s/^/ant.jar /p' |

# Run ckjm, calculating the metrics for the filename class
# pairs read from its standard input
java -jar ckjm-1.9.jar 2>/dev/null |

# Order the results numerically by the second field
# (Weighted methods per class)
sort --reverse --key=2 --numeric-sort
```

Here are, as usual, the first few lines of the pipeline's output.

```
org.apache.tools.ant.Project 127 1 0 33 299 7533 368 110
org.apache.tools.ant.taskdefs.Javadoc 109 0 0 35 284 5342 8 83
org.apache.tools.ant.taskdefs.Javac 88 0 1 19 168 3534 14 75
org.apache.tools.ant.taskdefs.Zip 78 0 1 44 269 2471 5 36
org.apache.tools.ant.DirectoryScanner 70 1 2 15 169 2029 43 34
org.apache.tools.ant.util.FileUtils 67 1 0 13 172 2181 151 65
org.apache.tools.ant.types.AbstractFileSet 66 0 3 31 137 1527 9 63
```

The preceding metrics of object-oriented code, can be further processed to flag classes with values that merit further analysis and justification [57, pp. 341–342], [49], and to locate opportunities for refactoring. In this case, some of the preceding classes, which comprise more than 50 classes each, may need refactoring, because they violate a rule of a thumb stating that elements consisting of more than 30 subelements are likely to be problematic [36, p. 31].

5. Analysis of configuration management data

Analysis of data obtained from a configuration management system [55, 61], such as *Git* [38], *Subversion* [46], or CVS [19], can provide valuable information regarding software evolution [11, 66], the engagement of developers [5], defect prediction [63, 9], distributed development [4, 14], and many other topics [27]. There are two types of data that can be obtained from a configuration management system.

Metadata are the details associated with each commit: the developer, the date and time, the commit message, the software branch, and the commit's scope. In addition, the commit message, apart from free text, often contains other structured elements, such as references to bugs in the corresponding database, developer user names, and other commits.

Snapshots of a project's source code can be obtained from the repository, reflecting the project's state at each point of time where a commit was made. The source code associated with each snapshot can be further analyzed using the techniques we saw in Section 3.

5.1. Obtaining repository data

Before a repository's data can be analyzed, it is usually preferable to obtain a local copy of the repository's data [40]. Although some repositories allow remote access to clients, this access is typically provided to serve the needs of software developers, i.e. an initial download of the project's source code, followed by regular, but not high-frequency, synchronization requests and commits. In contrast, a repository's analysis might involve many expensive operations, like thousands of checkouts at successive time points, which can stress the repository server's network bandwidth, CPU resources, and administrators. Operations on a local repository copy will not tax the remote server, and will also speed up significantly the operations run on it.

The techniques used for obtaining repository data depend on the repository type and the number of repositories that are to be mirrored. Repositories of distributed version control systems [44] offer commands that can readily create a copy of a complete repository from a remote server, namely `bzr branch` for *Bazaar* [20], `git clone` for *Git*, and `hg clone` for *Mercurial* [45]. Projects using the *Subversion* or CVS system for version control, can sometimes be mirrored using the *rsync* or the *svnsync* (for *Subversion*) command and associated protocol. The following are examples of commands used to mirror a diverse set of repositories.

```
# Create a copy of the GNU cpio Git repository
git clone git:// git.savannah.gnu.org/cpio.git

# Create a copy of the GNU Chess Subversion repository using rsync
rsync -avHS rsync://svn.savannah.gnu.org/svn/chess/ chess.repo/

# Create a copy of the Asterisk Bazaar repository
bzr branch lp:asterisk

# Create a copy of the Mercurial C Python repository
hg clone http://hg.python.org/cpython
```

Using *svnsync* is more involved; here is an example of the commands used for mirroring the *Subversion* repository of the JBOSS application server.

```
svnadmin create jboss-as
svnsync init file:///`pwd`/jbossas https://svn.jboss.org/repos/jboss-as
cd jboss-as
echo '#!/bin/sh' > hooks/pre-revprop-change
chmod +x hooks/pre-revprop-change
cd ..
svnsync init file:///`pwd`/jboss-as http://anonsvn.jboss.org/repos/jbossas
svnsync sync file:///`pwd`/jboss-as
```

Alternatively, if a friendly administrator has shell-level access to the site hosting the subversion repository, the repository can be dumped into a file and restored back from it using the commands `svnadmin dump` and `svnadmin load`.

When multiple repositories are hosted on the same server, their mirroring can be automated by generating cloning commands. These can often be easily created by

screen-scrapping the web page that lists the available repositories. As an example consider the *Git* repositories hosted on `git.gnome.org`.

Each project is listed with an HTML line like the following.

```
<tr><td class='sublevel-repo'><a title='archive/gcalctool'
href='/browse/archive/gcalctool/'>archive/gcalctool</a>
</td><td><a href='/browse/archive/gcalctool/'>Desktop calculator</a>
</td><td></td><td><span class='age-months'>11 months</span></td></tr>
```

The name of the corresponding *Git* repository can be readily extracted from the URL. In addition, because the projects are listed in multiple web pages, a loop is required to iterate over them, specifying the project list's offset for each page. The following short script, will download all repositories using the techniques we saw.

```
# List the URLs containing the projects
perl -e 'for ($i = 0; $i < 1500; $i += 650) {
  print "https://git.gnome.org/browse/?ofs=$i\n"}' |

# For each URL
while read url
do
  # Read the web page
  curl "$url" |

  # Extract the Git repository URL
  sed --quiet '/sublevel-repo/s|.* href='\''/browse/\([^'\'']*\)'\'' .*|\
    git://git.gnome.org/\1|p' |

  # Run git clone for the specific repository
  xargs --max-args=1 git clone
done
```

Over the recent years GitHub has evolved to be an amazingly large repository of open source projects. Although GitHub offers an API to access the corresponding project data (see Figure 2), it does not offer a comprehensive catalog of the data stored in it [17]. Thankfully, large swathes of the data can be obtained as database dump files in the form of a torrent [15].

5.2. Analyzing metadata

Metadata analysis can easily be performed by running the version control system command that outputs the revision log. This can then be analyzed using the process outlined in Section 2.

As examples of metadata, consider the following *Git* log entry from the Linux kernel

```
commit fa389e220254c69ffae0d403eac4146171062d08
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sun Mar 9 19:41:57 2014 -0700

    Linux 3.14-rc6
```

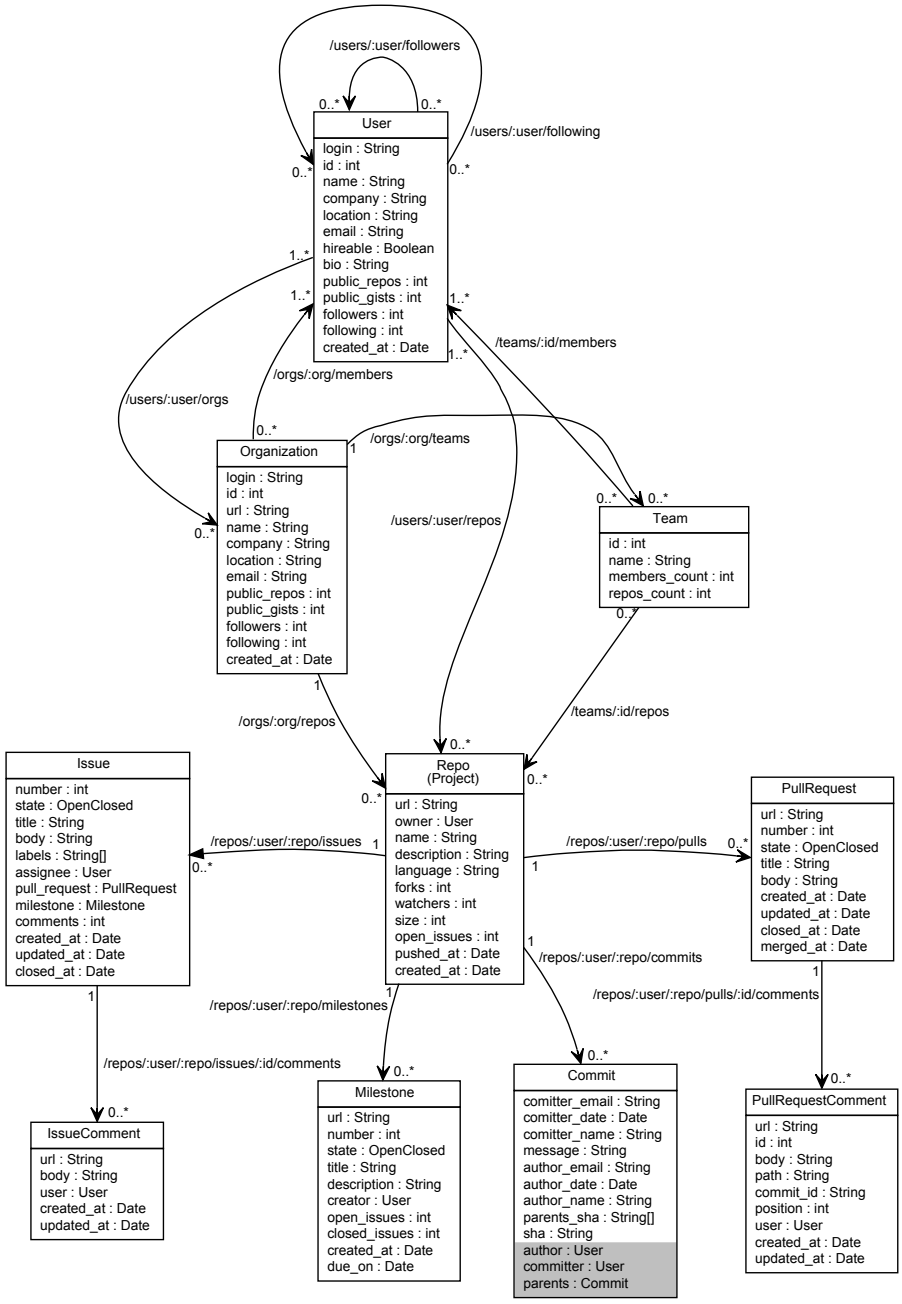

Figure 2: Schema of the data available through GitHub

and an older CVS log entry from FreeBSD's *sed* program.

```
revision 1.28
date: 2005/08/04 10:05:11;  author: dds;  state: Exp;  lines: +8 -2
Bug fix: a numeric flag specification in the substitute command would
cause the next substitute flag to be ignored.
While working at it, detect and report overflows.

Reported by:    Jingsong Liu
MFC after:      1 week
```

The following example illustrates how we can obtain the time of the first commit from various types of repositories.

```
# Git
git rev-list --date-order --reverse --pretty=format:'%ci' master |
sed --quiet 2p

# Bazaar
bzr log | grep timestamp: | tail -1

# Mercurial
hg log | grep 'date:' | tail -1

# CVS
cvs log -SN |
sed --quiet 's/^date:  \(.....\).*/\1/  p' |
sort --unique |
head -1
```

Some version control systems, such as *Git*, allow us to specify the format of the resulting log output. This makes it easy to isolate and process specific items. The following sequence will print the author names of the ten most prolific contributors associated with a *Git* repository, ordered by the number of commits they made.

```
# Print author names
git log --format='%an' |

# Order them by author
sort |

# Count number of commits for each author
uniq --count |

# Order them by number of commits
sort --numeric-sort --reverse |

# Print top ten
head -10
```

The result of running the preceding script on the last ten years of the Linux kernel is the following.

```
 20131 Linus Torvalds
  8445 David S. Miller
  7692 Andrew Morton
  5156 Greg Kroah-Hartman
  5116 Mark Brown
  4723 Russell King
  4584 Takashi Iwai
  4385 Al Viro
  4220 Ingo Molnar
  3276 Tejun Heo
```

Such lists can be used to gain insights on the division of labour within teams and developer productivity.

Aggregate results can be readily calculated using *awk*'s associative arrays. The following example shows the lines contributed by each developer in a CVS repository.

```
# Print the log
cvs log -SN |

# Isolate the author and line count
sed -n '/^date:/ s/[+;]//g' |

# Tally lines per author
awk '{devlines[$5] += $9}
        END {for (i in devlines) print i, devlines[i]}' |

# Order entries by descending number of lines
sort --key=2 --numeric-sort --reverse
```

The first ten lines from the output of the preceding command run on the FreeBSD kernel are the following.

```
gallatin 956758
mjacob 853190
sam 749313
jchandra 311499
jmallett 289413
peter 257575
rwatson 239382
jhb 236634
jimharris 227669
vkashyap 220986
```

5.3. Analyzing time series snapshots

Creating a series of source code snapshots from a repository requires us

- to perform accurate data calculations,
- to represent the dates in a format that can be unambiguously parsed by the repository's version control system, and

- to check out the corresponding version of the software.

Accurate date calculations on time intervals can be performed by expressing dates in seconds from the start of an epoch (1970-01-01 on Unix systems). The Unix *date* command allows the conversion of an arbitrary start date into seconds since Epoch. Unfortunately, the way this is done differs between various Unix-like systems. The following Unix shell function expects as its first argument a date expressed in ISO-8601 basic date format (`YYYYMMDD`). It will print the date as an integer representing seconds since Epoch.

```
iso_b_to_epoch()
{
  case `uname` in
  FreeBSD)
    date -j "$1"0000.00 '+%s' ;;
  Darwin)
    # Convert date to "mmdd0000yyyy.00" (time is 00:00:00)
    MDHMY=`echo $1 | sed 's       /\(....\)\(..\)\(..\)/\2\30000\1.00/       '`
    date -j "$MDHMY" '+%s'
    ;;
  CYGWIN*)
    date -d "$1" '+%s' ;;
  Linux)
    date -d "$1" '+%s' ;;
  *)
    echo "Unknown operating system type" 1>&2
    exit 1
  esac
}
```

The reverse conversion, from Epoch seconds to the ISO-8601 extended format (`YYYY-MM-DD`), which most version control systems can parse unambiguously, again depends on the operating system flavour. The following Unix shell function expects as its first argument a date expressed as seconds since Epoch. It will print the corresponding date in ISO format.

```
epoch_to_iso_e()
{
  case `uname` in
  Darwin)
    date -r $1 '+%Y-%m-%d' ;;
  FreeBSD)
    date -r $1 '+%Y-%m-%d' ;;
  CYGWIN*)
    date -d @$1 '+%Y-%m-%d' ;;
  Linux)
    date -d @$1 '+%Y-%m-%d' ;;
  *)
```

```
    echo "Unknown operating system type" 1>&2
    exit 1
  esac
}
```

As we would expect, the code to checkout a snapshot of the code for a given date depends on the version control system in use. The following Unix shell function expects as its first argument an ISO-8601 extended format date. In addition, it expects that the variable `$REPO` is set to one of the known repository types, and that it is executed within a directory where code from that repository has already been checked out. It will update the directory's contents with a snapshot of the project stored in the repository for the specified date. In the case of a *Bazaar* repository, the resultant snapshot will be stored in `/tmp/bzr-checkout`.

```
date_checkout ()
{
  case "$REPO" in
  bzr)
    rm -rf /tmp/bzr-checkout
    bzr export -r date:"$1" /tmp/bzr-checkout
    ;;
  cvs)
    cvs update -D "$1"
    ;;
  git)
    BRANCH=`git config --get-regexp branch.*remote |
      sed -n 's/^branch .//; s /\. remote origin // p' `
    HASH=`git rev-list -n 1 --before="$1" $BRANCH`
    git checkout $HASH
    ;;
  hg)
    hg update -d "$1"
    ;;
  rcs)
    # Remove files under version control
    ls RCS | sed 's /, v$// ' | xargs rm -f
    # Checkout files at specified date
    co -f -d"$1" RCS/*
    ;;
  svn)
    svn update -r "{$1}"
    if [ -d trunk ]
    then
      DIR=trunk
    else
      DIR=.
    fi
```

```
    ;;
  *)
    echo "Unknown repository type: $REPO" 1>&2
    exit 1
    ;;
  esac
}
```

Given the building blocks we saw, a loop to perform some processing on repository snapshots over successive ten day periods starting from, say, 2005-01-01, can be written as follows.

```
# Start date (2005-01-1) in seconds since Epoch
START=`iso_b_to_epoch 20050101`

# End date in seconds since Epoch
END=`date '+%s'`

# Time increment (10 days) in seconds
INCR=`expr 10 * 24 * 60 * 60`

DATE=$START
while [ $DATE -lt $END ]
do
        date_checkout $DATE
        # Process the snapshot
        DATE=`expr $DATE + $INCR`
done
```

5.4. Analyzing a checked out repository

Given a directory containing a project checked-out from a version control repository, we can analyze it using the techniques listed in Section 3. Care must be taken to avoid processing the data files associated with the version control system. A regular expression that can match these files, in order to exclude them, can be set as follows, according to the repository type.

```
case "$REPO" in
bzr)
  # Files are checked out in a new directory; nothing to exclude
  EXCLUDE=///
  ;;
cvs) EXCLUDE='/CVS/' ;;
git) EXCLUDE=.git ;;
hg) EXCLUDE='/.hg/' ;;
rcs) EXCLUDE='/RCS/' ;;
svn) EXCLUDE='/.svn/' ;;
esac
```

Another prerequisite for the analysis is identifying the source code files to analyze. Files associated with specific programming languages can be readily identified by their extension. For instance, C files end in `.c`; C++ files typically end in `.cpp`, `.C`, `.cc`, or `.cxx`; while Java files end in `.java`. Therefore, the following command

```
find . -type f -name \*.java
```

will output all the Java source code files residing in the current directory tree.

On the other hand, if we wish to process all source code files (for instance to count the source code size in terms of lines) we must exclude binary files, such as those containing images, sound, and compiled third-party libraries. This can be done by running the Unix *file* command on each project file. By convention the output of *file* will contain the word `text` only for text files (in our case source code and documentation).

Putting all the above together, here is a pipeline that measures the lines in a repository snapshot checked out in the current directory.

```
# Print names of files
find . -type f |

# Remove from list version control data files
fgrep --invert-match "$EXCLUDE" |

# Print each file's type
file --files-from - |

# Print only the names of text files
sed --quiet 's/:  .* text .*// p' |

# Terminate records with \0, instead of newline
tr \\n \\0 |

# Catenate the contents of all files together
xargs --null cat |

# Count the number of lines
wc --lines
```

5.5. Combining files with metadata

The version control system can also be used to help us analyze a project's files. An invaluable feature is the annotation (also known as "blame") command offered by many version control systems. This will display a source code file, listing with each line the last commit associated with it, the committer, and the corresponding date.

```
d62bd540 (linus1            1991-11-11  1) /*
d62bd540 (linus1            1991-11-11  2)  *  linux/kernel/sys.c
d62bd540 (linus1            1991-11-11  3)  *
cf1bbb91 (linus1            1992-08-01  4)  *  Copyright (C) 1991 Linus Torvalds
d62bd540 (linus1            1991-11-11  5)  */
```

```
d62bd540 (linus1            1991-11-11  6)
9984de1a (Paul Gortmaker    2011-05-23  7) #include <linux/export.h>
23d9e975 (linus1            1998-08-27  8) #include <linux/mm.h>
cf1bbb91 (linus1            1992-08-01  9) #include <linux/utsname.h>
8a219a69 (linus1            1993-09-19 10) #include <linux/mman.h>
d61281d1 (linus1            1997-03-10 11) #include <linux/reboot.h>
e674e1c0 (linus1            1997-08-11 12) #include <linux/prctl.h>
ac3a7bac (linus1            2000-01-04 13) #include <linux/highuid.h>
9a47365b (Dave Jones        2002-02-08 14) #include <linux/fs.h>
74da1ff7 (Paul Gortmaker    2011-05-26 15) #include <linux/kmod.h>
cdd6c482 (Ingo Molnar       2009-09-21 16) #include <linux/perf_event.h>
3e88c553 (Daniel Walker     2007-05-10 17) #include <linux/resource.h>
dc009d92 (Eric W. Biederman 2005-06-25 18) #include <linux/kernel.h>
e1f514af (Ingo Molnar       2002-09-30 19) #include <linux/workqueue.h>
c59ede7b (Randy.Dunlap      2006-01-11 20) #include <linux/capability.h>
```

Given such a list we can easily cut out specific columns with the Unix *cut* command, and analyze version control metadata at the level of source code lines rather than complete files. For example, the following command will list the top contributors in the file `cgroup.c` of the Linux kernel at its current state.

```
# Annotated listing of the file
git blame kernel/cgroup.c |

# Cut-out the author name
cut --characters=11-30 |

# Order by author name
sort |

# Count consecutive author name occurences
uniq --count |

# Order occurences by their number
sort --reverse --numeric |

# Show top contributors
head
```

This is the command's output.

```
2425 Tejun Heo
1501 Paul Menage
 496 Li Zefan
 387 Ben Blum
 136 Cliff Wickman
 106 Aristeu Rozanski
  60 Daniel Lezcano
  52 Mandeep Singh Baines
  42 Balbir Singh
  29 Al Viro
```

Similarly we could find how many lines of the file stem from each year.

```
# Annotated listing of the file
git blame kernel/cgroup.c |

# Cut-out the commit's year
cut --characters=11-30 |

# Order by year
sort --key=2
```

This is the output we get by the preceding command.

```
1061 2007
 398 2008
 551 2009
 238 2010
 306 2011
 599 2012
2243 2013
  37 2014
```

5.6. Assembling repositories

Projects with a long history provide an interesting source data. However, the data are seldom stored neatly in a single repository. More often than not there are snapshots from the beginning of a project's lifetime, then one or more frozen dumps of version control systems that are no longer used, and, finally, the live version control system. Fortunately, *Git* and other modern version control systems, offer mechanisms to assemble a project's history retroactively, by piecing together various parts.

With *Git*'s *graft* feature, multiple *Git* repositories can be pieced together into a whole. This is for instance the method Yoann Padioleau used to create a *Git* repository of Linux's history, covering the period 1992–2010.[11] The last repository in the series is the currently active Linux repository. Therefore, with a single `git pull` command, the archive can be easily brought up to date. The annotated Linux file in Section 5.5 stems from a repository assembled in this way. A similar repository[12] covers 40 years of Unix development history.

If the repositories are not in *Git* format, then, given *Git*'s flexibility, the most expedient way to combine them into *Git* format, and use the methods we saw in this section for analyzing modern repositories.

Snapshots of a project can be imported into a *Git* repository with the correct date and committer (which must be derived from external sources, such as timestamps), using a Unix shell function like the following. This expects as its first argument the directory where the snapshot's files are located, and as its second argument an ISO-8601 basic date format (`YYYYMMDD`) date associated with the snapshot. When run within a directory where a *Git* repository has been checked out, it will add the files to the repository with a commit dated as specified. The code utilizes the `iso_b_to_epoch` func-

[11] `https://archive.org/details/git-history-of-linux`
[12] `https://github.com/dspinellis/unix-history-repo`

tion we saw in Section 5.3. To specify the code's author the `git commit --author` flag can be added to the code.

```
snapshot_add()
{
  rm --recursive --force *
  cp --recursive $1 .
  git add *
  GIT_AUTHOR_DATE="'iso_b_to_epoch $2' +0000" \
  GIT_COMMITTER_DATE="'iso_b_to_epoch $2' +0000" \
  git commit --all --message="Snapshot from $1"
  git tag --annotate "snap-$2" -m "Add tag for snapshot $2"
}
```

Importing into *Git* data stored in other repositories is relatively easy, thanks to existing libraries and tools that can be used for this purpose. Of particular interest is the (badly misnamed) *cvs2svn* program,[13] which can convert RCS [67] and CVS repositories into *Git* ones. In addition, the Perl VCS-SCCS library[14] contains an example program that can convert legacy SCCS [48] repository data into *Git* format.

Finally, if a program that can perform the conversion cannot be found, a small script can be written to print revision data in *Git*'s *fast import* format. This data can then be fed into the `git fast-import` command, to import it into *Git*. The data format is a textual stream, consisting of commands, like `blob`, `commit`, `tag`, and `done`, which are complemented by associated data. According to the command's documentation and the personal experience of this study's author, an import program can be written in a scripting language within a day.

6. Data visualization

Given the volume and complexity of the data derived from the analysis methods we examined, it is easier to automate the process of diagram creation, rather than using spreadsheets or GUI-based graphics editors to draw them. The text-based tools we will see in this section do not beat the speed of firing up a drawing editor to jot a few lines or a spreadsheet to create a chart from a list of numbers. However, investing time to learn them, allows us to be orders-of-magnitude more efficient in repeatedly diagramming big data sets, performing tasks no one would dream of attempting in the GUI world.

6.1. Graphs

Perhaps the most impressive tool of those we will examine is *dot*. Part of the Graphviz suite [13], originally developed by AT&T, it lets us describe hierarchical relations between elements using a simple declarative language. For instance, with the following input *dot* will generate the diagram shown in Figure 3.

[13] `http://cvs2svn.tigris.org/cvs2svn.html`
[14] `http://search.cpan.org/dist/VCS-SCCS/`

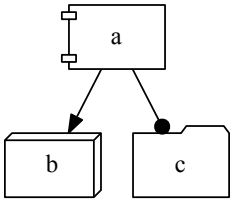

Figure 3: A simple diagram made with *dot*

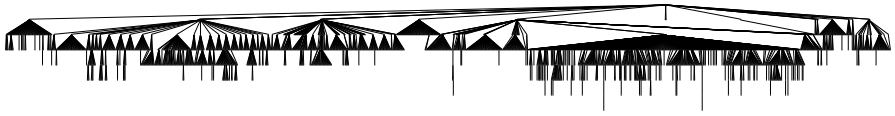

Figure 4: The Linux directory tree

```
digraph {
  a [shape="component"];
  b [shape="box3d"];
  c [shape="folder "];
  a -> b;
  a -> c [arrowhead="dot"];
}
```

Dot offers a wide choice of node shapes, arrows, and options for controlling the graph's layout. It can handle graphs with thousands of nodes. This study's author has used it to display class hierarchies, database schemas, directory trees, package dependency diagrams, mechanical gear connections, and even genealogical trees. Its input language is simple (mostly graph edges and nodes), and it is trivial to generate a diagram from a script of just a few lines.

For example, the following Perl script will create a diagram of a directory tree. When run on the Linux source code tree, it will generate the Figure 4 [58]. This shows a relatively shallow and balanced tree organization, which could be a mark of an organization that maintains an equilibrium between the changes brought by organic growth and the order achieved through regular refactorings. An architect reviewer might also question the few deep and narrow tree branches appearing in the diagram.

```
open(IN, "find $ARGV[0] -type d -printl");

while (<IN>) {
```

```
  chop;
  @paths = split (/\//, $_);
  undef $opath;
  undef $path;
  for $p (@paths) {
    $path .= "/$p";
    $name = $path;
    # Make name a legal node label
    $name =~ s/[^a-zA-Z0-9]/_/g;
    $node{$name} = $p;
    $edge{"$opath->$name;"} = 1 if ($opath);
    $opath = $name;
  }
}

print 'digraph G {
  nodesep=0.00001;
  node [height=.001,width=0.000001,shape=box,fontname="",fontsize=8];
  edge [arrowhead=none,arrowtail=none];
';

for $i (sort keys %node) {
  print "\t$i [label=\"\"];\n";
}
for $i (sort keys %edge) {
  print "\t$i\n";
}
print "}\n";
```

It is also easy to create diagrams from the version control system's metadata. The following Unix shell script will create a diagram of relationships between Linux authors and committers. The result of processing the first 3000 lines of the Linux kernel *Git* log can be seen in Figure 5.

```
(
  # Specify left-to right ordering
  echo 'digraph { rankdir=LR;'

  # Obtain git log
  git log --pretty=fuller |

  # Limit to first 3000 lines
  head -3000 |

  # Remove email
  sed 's/<.*// ' |
```

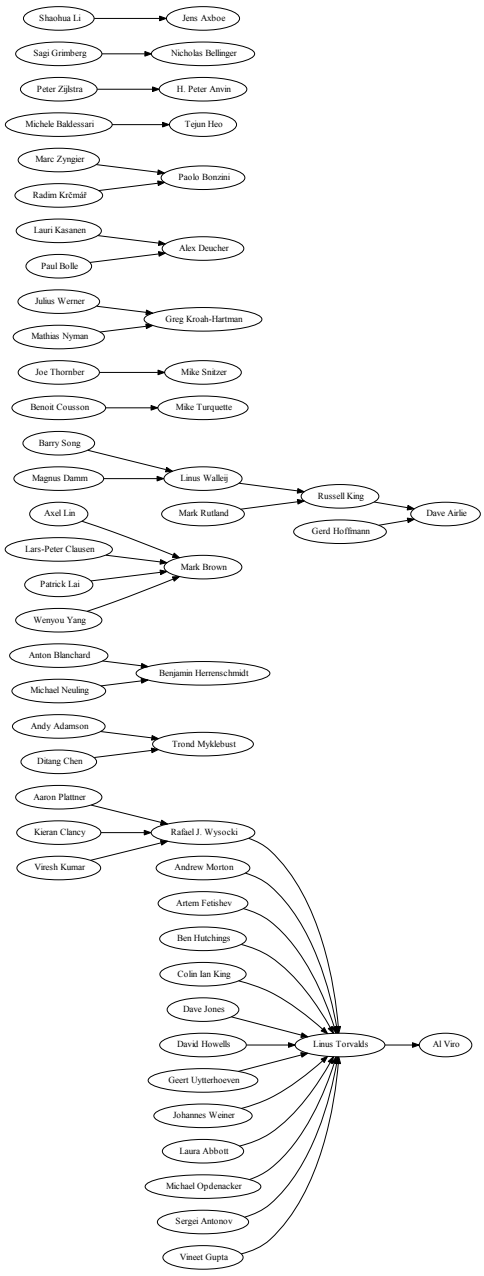

Figure 5: Relationships between Linux authors and committers

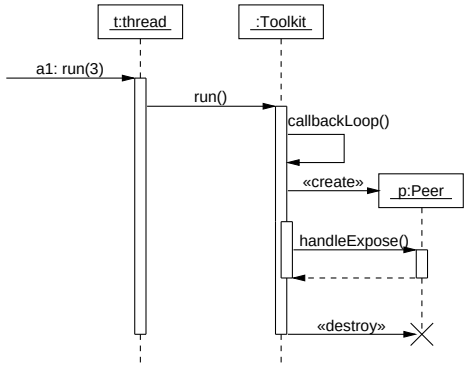

Figure 6: A diagram made with *pic*

```
  # Print author-committer pairs
  awk '
    # Store author and committer
    /^Author:/ { $1 = ""; a = $0}
    /^Commit:/{ $1 = ""; c = $0}
    /^CommitDate:/{
      if (a && c && a != c)
        print "\"" a "\" -> \"" c "\";"
    }' |

    # Eliminate duplicates
    sort -u
    # Close brace
    echo '}'
)
```

Three cousins of *dot*, also parts of GraphViz, are *neato*, for drawing undirected graphs, and *twopi* and *circo*, for drawing radial and circular layout graphs. All use an input language similar to dot's. They are less useful for visualizing software systems, but in some cases they come in handy. For instance, this study's author has used *neato* has to draw the relationships between software quality attributes, links between Wikipedia nodes, and collaboration patterns between software developers.

6.2. Declarative Diagrams

A slightly less declarative, but no less versatile, family of tools are those that target text-based typesetting systems: TikZ [64], which is based on TEX [32], and *pic* [2], which is based on *troff* [30]. The *pic* program was originally developed at AT&T's Bell Labs as part of the Unix document preparation tools [31], but these days it is more likely to appear in its GNU *groff* reincarnation. *Pic*'s language gives us commands such as `box`, `circle`, `line`, and `arrow`. Unlike the GraphViz tools, *pic* will not lay out the diagram for us, but it makes up for its lack of intelligence by letting us create macros

and supporting loops and conditionals. This lets us define our own complex shapes (for our project's specialized notation) and then invoke them with a simple command. In effect, we are creating our own domain-specific drawing language. As an example, the following *pic* code, in conjunction with macros defined as part of the *UMLGraph* system [53], will result in Figure 6.

```
.PS
  copy "sequence.pic";

  # Define the objects
  pobject(E,"External Messages");
  object(T,"t:thread");
  object(O,":Toolkit");
  pobject(P);

  step();

  # Message sequences
  message(E,T,"a1: run(3)");
  active(T);
  message(T,O,"run()");
  active(O);
  message(O,O,"callbackLoop()");
  cmessage(O,P,"p:Peer","");
  active(O);
  message(O,P,"handleExpose()");
  active(P);
  rmessage(P,O,"");
  inactive(P);
  inactive(O);
  dmessage(O,P);
  inactive(T);
  inactive(O);

  step();

  complete(T);
  complete(O);
.PE
```

6.3. Charts

When dealing with numbers, two useful systems for generating charts are *gnuplot*[15] [25] and the *R Project* [65]. *Gnuplot* is a command-line driven graphing utility,

[15] `http://www.gnuplot.info/`

whereas R is a vastly larger and more general system for performing statistical analysis, which also happens to have a very powerful plotting library.

Gnuplot can plot data and functions in a wide variety of 2D and 3D styles, using lines, points, boxes, contours, vector fields, surfaces, and error bars. We specify what our chart will look like with commands like `plot with points` and `set xlabel`. To plot varying data (for instance, to track the number of new and corrected bugs in a project), we typically create a canned sequence of commands that will read the data from an external file our code generates.

As an example of using *gnuplot* to draw a chart from software-generated data, consider the task of plotting a program's stack depth [57, p. 270]. The stack depth at the point of each function's entry point can be obtained by compiling the program's code with profiling enabled (by passing the `-pg` flag to GCC), and using the following custom profiling code to write the stack depth at the point of each call into file pointed by the file descriptor `fd`.

```
_MCOUNT_DECL(frompc, selfpc) /* _mcount; may be static , inline , etc */
    u_long frompc, selfpc ;
{
    struct gmonparam *p;
    void *stack = &frompc;

    p = &_gmonparam;
    if (p->state != GMON_PROF_ON)
        return;
    p->state = GMON_PROF_BUSY;
    frompc -= p->lowpc;
    if (frompc > p->textsize)
        goto done;
    write(fd, &stack, sizeof(stack));
done:
    p->state = GMON_PROF_ON;
    return;
overflow:
    p->state = GMON_PROF_ERROR;
    return;
}
```

MCOUNT

Then, a small script, like the following one written in Perl, can read the file and create the corresponding *gnuplot* file, which will then create a chart similar to the one seen in Figure 7. The information gathered from such a figure can be used to judge the size and variation of the program's stack size, and therefore allow the tuning of stack allocation in memory-restricted embedded systems.

```
print OUT qq{
set ytics 500
```

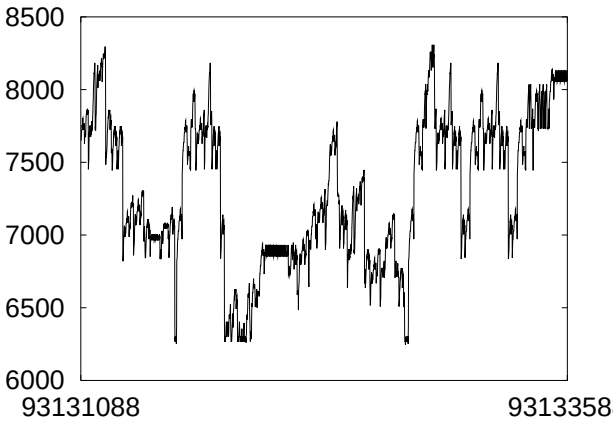

Figure 7: Stack depth measurements plotted using *gnuplot*

```
set format x "%.0f"
set terminal postscript eps enhanced "Helvetica" 30
set output 'stack.eps'
plot [] [] "-" using 1:2 notitle with lines
};
for (my $i = 0; $i < $nwords; $i++) {
  read(IN, $b, 4);
  my ($x) = unpack('L', $b);
  $x = $stack_top - $x;
  print OUT "$i $x\n";
}
print OUT "e\n";
```

More sophisticated diagrams can be plotted with R and the *ggplot2* library [69].

6.4. Maps

The last domain that we will covere involves geographical data. Consider data like the location of a project's contributors, or the places where a particular software is used. To place the corresponding numbers on the map, one option is the Generic Mapping Tools (GMT) [68].[16] We use these by plumbing together 33 tools that manipulate data and plot coastlines, grids, histograms, lines, and text using a wide range of map substrates and projections. Although these tools are not as easy to use as the others we have covered, they create high-quality output and offer extreme flexibility in a demanding domain.

[16] `http://gmt.soest.hawaii.edu/`

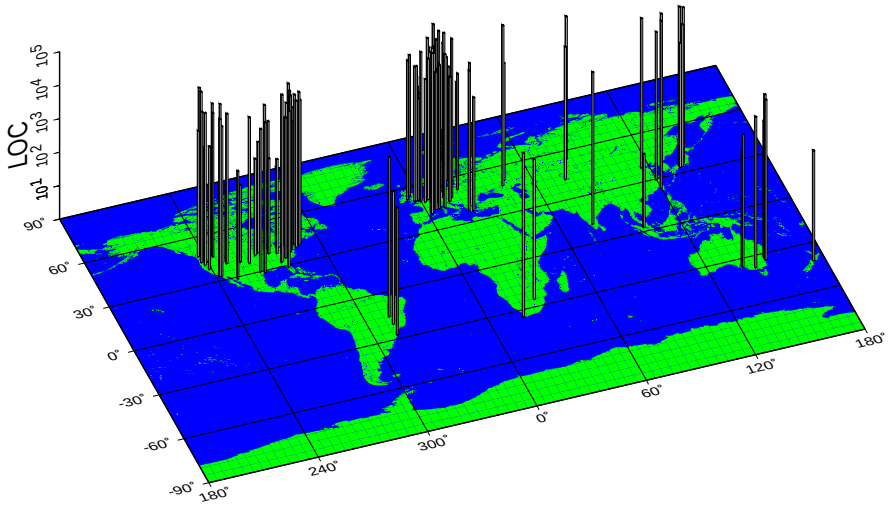

Figure 8: Contributions by FreeBSD developers around the world

As an example, consider the map depicting the contributions of FreeBSD developers around the world (Figure 8), showing that development is mainly concentrated in Europe, North America, and Japan. The map was generated using the following script, which ends with two GMT commands. As this is the last script of this work, it brings together many of the techniques we have examined, integrating process and product data with their visualization.

```
# 1. List developer locations
# Remove comments, etc. from the FreeBSD contributor location file
sed '/^#/d;/^$/d;s/,/"/;s/,/"/;s/^#//;s/[ ]*// g' \
  /usr/ports/astro/xearth/files/freebsd.committers.markers |

# Print latitude, longitude, developer-id
awk 'BEGIN{FS="\x22"}{print $1, $2, $4}' |

# Split developers living in the same city
perl -na -e 'for $d (split (",", $F[2])) { print "$d $F[0] $F[1]\n"}' |

# Remove empty lines
sed '/^/d' |

# Sort (for joining)
sort >dev-loc

# 2. Calculate lines per developer
```

```
# Find files in the FreeBSD checked-out repository
find . -type f |

# Remove version control files
grep --invert-match CVS |

# Create a log
xargs cvs -d /home/ncvs log -SN 2>/dev/null |

# Total lines each developer has contributed
awk '
  /^date/{ lines[$5 " " hour] += $9}
  END {
    for (i in lines)
      print i, lines[i]}
  ' |
# Remove ;
sed 's /;// g' |

# Sort (for joining)
sort >dev-lines


# 3. Plot the map
# Join developer lines with their locations
join dev-lines dev-loc |

# Round location positions to integer degrees
sed 's /\.[0-9]*// g' |

# Total lines for each location
awk '
  { lines[$4 " " $2] += $2}
  END {
    for (i in lines)
      print i, lines[i]
  }' |

# Draw the map
{
  # Draw the coastlines
  pscoast -R-180/180/-90/90 -JX8i/5id -Dc -G0 -E200/40 \
    -K W0.25p/255/255/255 -G0/255/0 -S0/0/255 -Di -P
  # Plot the data
  psxyz -P -R-180/180/-90/90/1/100000 -JX -JZ2.5il \
    -So0.02ib1 -G140 -W0.5p -O -E200/40 -B60g60/30g30/a1p:LOC:WSneZ
```

} >map.eps

Another alternative involves generating KML, the Google Earth XML-based file format, which we can then readily display through Google Earth and Maps. The limited display options we get are offset by the ease of creating KML files and the resulting display's interactivity.

If none of the tools we have seen fits our purpose, we can dive into lower-level graphics languages such as PostScript and SVG (Scalable Vector Graphics). This approach has been used to annotate program code [52] and to illustrate memory fragmentation [57, p. 251]. Finally, we can always use *ImageMagick*[17] to automate an image's low-level manipulation.

The tools described in this section offer a bewildering variety of output formats. Nevertheless, the choice is easy. If we are striving for professional-looking output, we must create vector-based formats such as PostScript, PDF, and SVG; we should choose the format our software best supports. The resulting diagrams will use nice-looking fonts and appear crisp, no matter how much we magnify them. On the other hand, bitmap formats, such as PNG, can be easier to display in a presentation, memo, or Web page. Often the best way to get a professional-looking bitmap image is to first generate it in vector form and then rasterize it through Ghostscript or a PDF viewer. Finally, if we want to polish a diagram for a one-off job, the clever route is to generate SVG and manipulate it using the *Inkscape*[18] vector-graphics editor.

7. Concluding Remarks

The software product and process analysis methods we have examined in this work offer a number of advantages.

Flexibility and Extensibility The scripts we have seen can be easily modified and adapted to suit a variety of needs. New tools can be easily added to our collection. These can be existing tools, or tools developed to suit our own unique needs.

Scalability The underlying tools have few if any inherent limits. Arbitrary amounts of data can flow through pipelines, allowing the processing of gigantic amounts of data. In our group we have used these approaches to process many hundreds of gigabytes of data.

Efficiency The workhorses of many pipelines, *git*, *sort*, and *grep*, have been engineered to be as efficient as possible. Other tools, such as *join*, *uniq*, and *comm*, are designed to run in linear time. When the tools run together in pipelines, the load is automatically divided among multiple processor cores.

Some may counter that the lack of a graphical user interface for using these analysis methods results in a steep learning curve, which hinders their use. This however can be

[17] `http://www.imagemagick.org/`
[18] `http://www.inkscape.org/`

mitigated in two ways. First, the use of each command can be easily learned by referring to its online manual page available through the *man* command, or by invoking the command with the `--help` argument. In addition, the creation of analysis scripts can be simplified by configuring, learning, and utilizing the shell's command-line editing and completion mechanisms.

Once the tools and techniques we examined are mastered, it is hard to find an alternative where one can be similarly productive.

References

References

[1] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D., 2007. Compilers: Principles, Techniques, & Tools. Pearson/Addison Wesley.

[2] Bentley, J. L., Aug. 1986. Little languages. Communications of the ACM 29 (8), 711–721.

[3] Bevan, J., Whitehead, Jr., E. J., Kim, S., Godfrey, M., 2005. Facilitating software evolution research with Kenyon. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. ACM, New York, NY, USA, pp. 177–186.

[4] Bird, C., Nagappan, N., June 2012. Who? where? what? examining distributed development in two large open source projects. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. pp. 237–246.

[5] Capiluppi, A., Serebrenik, A., Youssef, A., June 2012. Developing an h-index for oss developers. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. pp. 251–254.

[6] Chidamber, S. R., Kemerer, C. F., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6), 476–493.

[7] Cubranic, D., Murphy, G., Singer, J., Booth, K., June 2005. Hipikat: a project memory for software development. Software Engineering, IEEE Transactions on 31 (6), 446–465.

[8] D'Ambros, M., Lanza, M., apr 2010. Distributed and collaborative software evolution analysis with Churrasco. Science of Computer Programming 75 (4), 276–287.

[9] Eyolfson, J., Tan, L., Lam, P., 2011. Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th Working Conference on Mining Software Repositories. MSR '11. ACM, New York, NY, USA, pp. 153–162.
URL `http://doi.acm.org/10.1145/1985441.1985464`

[10] Friedl, J. E., 2006. Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools, 3rd Edition. O'Reilly Media, Sebastopol, CA.

[11] Gala-Pérez, S., Robles, G., González-Barahona, J. M., Herraiz, I., 2013. Intensive metrics for the study of the evolution of open source projects: Case studies from apache software foundation projects. In: Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13. IEEE Press, Piscataway, NJ, USA, pp. 159–168.
URL http://dl.acm.org/citation.cfm?id=2487085.2487119

[12] Gall, H., Fluri, B., Pinzger, M. ., Jan-Feb 2009. Change analysis with Evolizer and ChangeDistiller. IEEE Software 26 (1), 26 – 33.

[13] Gansner, E. R., North, S. C., 2000. An open graph visualization system and its applications to software engineering. Software: Practice & Experience 30 (11), 1203–1233.

[14] Giaglis, G. M., Spinellis, D., Nov. 2012. Division of effort, productivity, quality, and relationships in FLOSS virtual teams: Evidence from the FreeBSD project. Journal of Universal Computer Science 18 (19), 2625–2645.
URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2012-JUCS-GSD/html/GS12b.html

[15] Gousios, G., 2013. The ghtorrent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13. IEEE Press, Piscataway, NJ, USA, pp. 233–236.

[16] Gousios, G., Spinellis, D., May 2009. A platform for software engineering research. In: Godfrey, M. W., Whitehead, J. (Eds.), MSR '09: Proceedings of the 6th Working Conference on Mining Software Repositories. IEEE, pp. 31–40.
URL http://www.dmst.aueb.gr/dds/pubs/conf/2009-MSR-Alitheia/html/GS09b.html

[17] Gousios, G., Spinellis, D., Jun. 2012. GHTorrent: Github's data from a firehose. In: Lanza, M., Penta, M. D., Xie, T. (Eds.), 9th IEEE Working Conference on Mining Software Repositories (MSR). IEEE, pp. 12–21.
URL http://www.dmst.aueb.gr/dds/pubs/conf/2012-MSR-GitHub/html/github-mirror.html

[18] Gousios, G., Spinellis, D., 2014. Conducting quantitative software engineering studies with Alitheia Core. Empirical Software Engineering 19 (4), 885–925.

[19] Grune, D., 1986. Concurrent versions system, a method for independent cooperation. Report IR-114, Vrije University, Amsterdam, NL.

[20] Gyerik, J., 2013. Bazaar Version Control. Packt Publishing Ltd, Birmingham, UK, iSBN 978-1849513562.

[21] Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R., Godfrey, M. W., 2013. The MSR cookbook: Mining a decade of research. In: Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13. IEEE Press, Piscataway, NJ, USA, pp. 343–352.

[22] Herraiz, I., Izquierdo-Cortazar, D., Rivas-Hernandez, F., González-Barahona, J., Robles, G., Dueñas Dominguez, S., Garcia-Campos, C., Gato, J., Tovar, L., march 2009. Flossmetrics: Free/libre/open source software metrics. In: CSMR '09: 13th European Conference on Software Maintenance and Reengineering. pp. 281–284.

[23] Hovemeyer, D., Pugh, W., Dec. 2004. Finding bugs is easy. ACM SIGPLAN Notices 39 (12), 92–106, oOPSLA 2004 Onward! Track.

[24] Howison, J., Conklin, M., Crowston, K., 2006. Flossmole: A collaborative repository for FLOSS research data and analyses. International Journal of Information Technology and Web Engineering 1 (3), 17–26.

[25] Janert, P. K., 2009. Gnuplot in Action: Understanding Data with Graphs. Manning Publications.

[26] Johnson, P., Kou, H., Paulding, M., Zhang, Q., Kagawa, A., Yamashita, T., July-Aug. 2005. Improving software development management through software project telemetry. Software, IEEE 22 (4), 76–85.

[27] Kagdi, H., Collard, M. L., Maletic, J. I., Mar. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance and Evolution: Research and Practice 19 (2), 77–131.

[28] Kamiya, T., Kusumoto, S., Inoue, K., Jul. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28 (7), 654–670.

[29] Kechagia, M., Spinellis, D., 2014. Undocumented and unchecked: Exceptions that spell trouble. In: MSR '14: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 312–315.

[30] Kernighan, B., Lesk, M., Ossanna, J. J., July-August 1978. UNIX time-sharing system: Document preparation. Bell System Technical Journal 56 (6), 2115–2135.

[31] Kernighan, B. W., July/August 1989. The UNIX system document preparation tools: A retrospective. AT&T Technical Journal 68 (4), 5–20.

[32] Knuth, D. E., 1986. TeX: The Program. Addison-Wesley, Reading, MA.

[33] Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004: International Symposium on Code Generation and Optimization. IEEE, pp. 75–86.

[34] Lesk, M. E., Oct. 1975. Lex—a lexical analyzer generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ.

[35] Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P., 2009. Sourcerer: mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery 18, 300–336, 10.1007/s10618-008-0118-x.

[36] Lippert, M., Roock, S., 2006. Refactoring in Large Software Projects. John Wiley & Sons, Chichester, England Hoboken, NJ.

[37] Liu, K., Tan, H. B. K., Chen, X., August 2013. Binary code analysis. Computer 46 (8), 60–68.

[38] Loeliger, J., McCullough, M., 2012. Version Control with Git: Powerful tools and techniques for collaborative software development. O'Reilly Media, Inc., Sebastopol, CA, iSBN 978-1449316389.

[39] Mitropoulos, D., Karakoidas, V., Louridas, P., Gousios, G., Spinellis, D., 2014. The bug catalog of the Maven ecosystem. In: MSR '14: Proceedings of the 2014 International Working Conference on Mining Software Repositories. ACM, pp. 372–365.

[40] Mockus, A., 2009. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories. MSR '09. IEEE Computer Society, Washington, DC, USA, pp. 11–20.

[41] Mulazzani, F., Rossi, B., Russo, B., Steff, M., Oct 2011. Building knowledge in open source software research in six years of conferences. In: Hissam, S., Russo, B., de Mendonça Neto, M., Kon, F. (Eds.), Proceedings of the 7th International Conference on Open Source Systems. IFIP, Springer, Salvador, Brazil, pp. 123–141.

[42] Nierstrasz, O., Ducasse, S., Gĭrba, T., 2005. The story of Moose: an agile reengineering environment. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. ESEC/FSE-13. ACM, New York, NY, USA, pp. 1–10.

[43] Ossher, J., Bajracharya, S., Linstead, E., Baldi, P., Lopes, C., 2009. SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects. In: Proceedings of the International Workshop on Mining Software Repositories. IEEE Computer Society, Vancouver, Canada, pp. 183–186.

[44] O'Sullivan, B., Sep. 2009. Making sense of revision-control systems. Communications of the ACM 52 (9), 56–62.

[45] O'Sullivan, B., 2009. Mercurial: The definitive guide. O'Reilly Media, Inc., Sebastopol, CA, iSBN 978-0596800673.

[46] Pilato, C. M., Collins-Sussman, B., Fitzpatrick, B. W., 2009. Version control with Subversion. O'Reilly Media, Inc., Sebastopol, CA, iSBN 978-0-596-51033-6.

[47] R Core Team, 2012. R: A language and environment for statistical computing.

[48] Rochkind, M. J., 1975. The source code control system. IEEE Transactions on Software Engineering SE-1 (4), 255–265.

[49] Rosenberg, L. H., Stapko, R., Gallo, A., Nov. 1999. Applying object-oriented metrics. In: Sixth International Symposium on Software Metrics—Measurement for Object-Oriented Software Projects Workshop. Presentation available online http://www.software.org/metrics99/rosenberg.ppt (January 2006).
URL http://www.software.org/metrics99/rosenberg.ppt

[50] Sarma, A., Maccherone, L., Wagstrom, P., Herbsleb, J., 2009. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In: Proceedings of the 31st International Conference on Software Engineering. ICSE '09. IEEE Computer Society, Washington, DC, USA, pp. 23–33.

[51] Spinellis, D., Jun. 2000. Outwit: Unix tool-based programming meets the Windows world. In: Small, C. (Ed.), USENIX 2000 Technical Conference Proceedings. USENIX Association, Berkeley, CA, pp. 149–158.
URL http://www.dmst.aueb.gr/dds/pubs/conf/2000-Usenix-outwit/html/utool.html

[52] Spinellis, D., 2003. Code Reading: The Open Source Perspective. Addison-Wesley, Boston, MA.
URL http://www.spinellis.gr/codereading

[53] Spinellis, D., March/April 2003. On the declarative specification of models. IEEE Software 20 (2), 94–96.
URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2003-IEEESW-umlgraph/html/article.html

[54] Spinellis, D., July/August 2005. Tool writing: A forgotten art? IEEE Software 22 (4), 9–11.
URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2005-IEEESW-TotT/html/v22n4.html

[55] Spinellis, D., September/October 2005. Version control systems. IEEE Software 22 (5), 108–109.
URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2005-IEEESW-TotT/html/v22n5.html

[56] Spinellis, D., November/December 2005. Working with Unix tools. IEEE Software 22 (6), 9–11.
URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2005-IEEESW-TotT/html/v22n6.html

[57] Spinellis, D., 2006. Code Quality: The Open Source Perspective. Addison-Wesley, Boston, MA.
URL http://www.spinellis.gr/codequality

[58] Spinellis, D., May 2008. A tale of four kernels. In: Schäfer, W., Dwyer, M. B., Gruhn, V. (Eds.), ICSE '08: Proceedings of the 30th International Conference on Software Engineering. Association for Computing Machinery, New York, pp. 381–390.
URL http://www.dmst.aueb.gr/dds/pubs/conf/2008-ICSE-4kernel/html/Spi08b.html

[59] Spinellis, D., Apr. 2010. CScout: A refactoring browser for C. Science of Computer Programming 75 (4), 216–231.
URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2010-SCP-CScout/html/cscout.html

[60] Spinellis, D., 2010. The Unix tools are your friends. In: Henney, K. (Ed.), 97 Things Every Programmer Should Know. O'Reilly, Sebastopol, CA, pp. 176–177.
URL http://programmer.97things.oreilly.com/wiki/index.php/The_Unix_Tools_Are_Your_Friends

[61] Spinellis, D., May/June 2012. Git. IEEE Software 29 (3), 100–101.
URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2005-IEEESW-TotT/html/v29n3.html

[62] Spinellis, D., Louridas, P., Jul. 2007. A framework for the static verification of API calls. Journal of Systems and Software 80 (7), 1156–1168.
URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2007-JSS-api-verify/html/SL07b.html

[63] Steff, M., Russo, B., June 2012. Co-evolution of logical couplings and commits for defect estimation. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. pp. 213–216.

[64] Tantau, T., 2013. Graph drawing in TikZ. In: Proceedings of the 20th International Conference on Graph Drawing. GD'12. Springer-Verlag, Berlin, Heidelberg, pp. 517–528.

[65] The R Development Core Team, 2010. R: A language and environment for statistical computing. R Foundation for Statistical Computing, 2nd Edition.
URL http://www.lsw.uni-heidelberg.de/users/christlieb/teaching/UKStaSS10/R-refman.pdf

[66] Thomas, S. W., Adams, B., Hassan, A. E., Blostein, D., 2011. Modeling the evolution of topics in source code histories. In: Proceedings of the 8th Working Conference on Mining Software Repositories. MSR '11. ACM, New York, NY, USA, pp. 173–182.
URL http://doi.acm.org/10.1145/1985441.1985467

[67] Tichy, W. F., Sep. 1982. Design, implementation, and evaluation of a revision control system. In: Proceedings of the 6th International Conference on Software Engineering. ICSE '82. IEEE, pp. 58–67.

[68] Wessel, P., Smith, W. H. F., 1991. Free software helps map and display data. EOS Transactions American Geophysical Union 72 (41), 441, 445–446.

[69] Wickham, H., 2009. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag.