SOFTWARE ANALYSIS BY CODE CLONES IN OPEN SOURCE SOFTWARE

SHINJI UCHIDA
Kinki University Technical College
Kumano, MIE 519-4395, Japan

TOSHIHIRO KAMIYA
Japan Science and Technology Corp.
Kawaguchi, Saitama 332-0012, Japan

AKITO MONDEN
Nara Institute of Science and Technology
Ikoma, Nara 630-0101, Japan

KEN-ICHI MATSUMOTO
Nara Institute of Science and Technology
Ikoma, Nara 630-0101, Japan

NAOKI OHSUGI
Nara Institute of Science and Technology
Ikoma, Nara 630-0101, Japan

HIDEO KUDO
Osaka Seikei University
Osaka, 533-0007, Japan

ABSTRACT

The code clone (a duplicated code section in the source files of software) is one of the factors that complicate software maintenance. However, few empirical studies have analyzed the status quo of code clones. In this paper we conduct an extensive analysis of code clones using 125 packages of open source software written in C language, and suggest guidelines for the allowable production of code clones. Our results showed 11.3% as the average CRate (clone inclusion rate). For software packages that did not include automatically generated code, the CRate was 9.7%, the in-module CRate 8.2%, and the inter-module CRate 1.3%. These rates can be used as criteria for the allowable amount of code clone production. This paper also presents our findings on factors of code clone production, the influence of code clone production on maintainability, and removal methods.

Keywords: Software Maintenance, Software Measurement, Software Metrics

INTRODUCTION

Code clones contained in software programs significantly reduce the maintainability of the software (7, 11, 29). Code clones are exact or nearly exact duplicate lines of code within the source code. These clones are typically created by copying and pasting source codes, e.g., a programmer may copy and paste a code fragment when a similar functionality of the fragment is required elsewhere in the program. If, however, a copy of a duplicated code section is revised, an update of all the other copies will be required, and this may raise the maintenance cost (9). Moreover, if one of the copies is overlooked, a fault will remain in the copy, and this may degrade the reliability of the system (29). These code clones may annoy program comprehension tasks in maintenance (34) as well as in reengineering (6) and reverse engineering activities (33). Therefore, code clones should not be created without careful consideration (7, 9, 19). To support the maintenance of software containing code clones, various methods have been proposed to efficiently detect code clones in large software (2-7, 21-28). Also, several methods for removing code clones have been proposed (7, 12). However, few empirical studies have been conducted to clarify the status quo of code clones, e.g. the amount of clones contained in well-maintained systems, the reasons why clones are produced, etc. Thus, programmers and maintenance engineers do not know how to cope with code clones. More specifically,

- The creation of large-scale software without code clones is almost impossible. However, it is not clear what types of code clones are allowable and what types are not.
- The amount of code clones allowed is not clear even when a certain amount of code clones is allowed.
- The automatic removal of code clones by macros can reduce the readability of the software, thus making maintenance even more difficult. Further, what type of clones should be removed and what type should not be removed is not clear although this can vary depending on the type of removal method.

To resolve these issues, we analyzed code clones in a number of open source software programs written in C to find out more about the creation and removal of code clones. We focused on open source software for the following reasons:

- Numerous source codes can be obtained, making statistical analysis possible (we selected C language programs for the same reason).
- There are many software programs available running on different types of platforms and covering a wide range of domains.
- Open source software is developed on the assumption that the source codes are made public. Therefore, many of these programs are maintained by a large number of people, and the programs are thus written in such a way as to make them easy to maintain. Therefore, we assume that open source software should contain relatively few code clones, which we hope will give us a guideline for the allowable amount of code clones.

For our analysis, we used CCFinder (21, 22, 23), proposed by Kamiya et al., to measure the code clones and analyze them statistically. We also used CloneWarrior (8), a code clone identification tool developed by our research group, to identify

and classify the code clones and to analyze the causes for their generation. Based on our results, we then considered whether or not the code clones should be eliminated as well as the possible elimination methods.

CODE CLONE MEASUREMENT METHODS AND DEFINITIONS

Code Clone Measurement Methods

Recently, various kinds of clone detection techniques and tools have been proposed (2-7, 21-28). Since clones usually occur when a code fragment is copied and partly modified, detecting a code fragment that is exactly identical to another fragment is not sufficient. Thus, existing tools also detect a fragment that is nearly identical to others.

In this paper we used the token-based code clone detection tool CCFinder, developed by Kamiya et al. (21, 22, 23). CCFinder compares source codes based on the syntax rules of the programming language. Therefore, even when the lines of codes differ in terms of whitespace, comments, indentation, or variable names, duplicated code sections can be detected as code clones despite these differences. CCFinder has industrial strength, and is applicable to a million-line size system within affordable computation time and memory usage (29). The basic procedure for detecting code clones is outlined below:

(Step 1) Lexical analysis
All the source files are divided into tokens based on the lexical rules of the programming language. Whitespace and comments are removed.

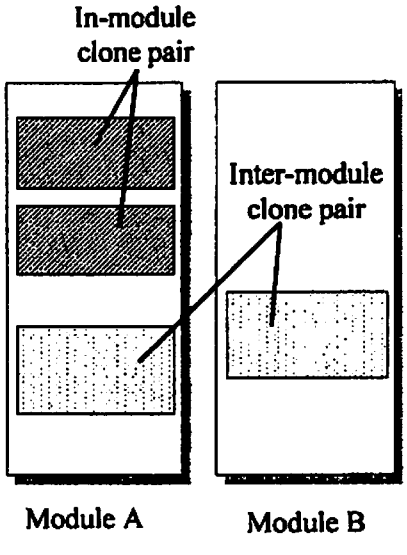
(Step 2) Transformation
Tokens representing types, variables, or constants are replaced by the same respective tokens. This replacement makes identifying a code clone as a pair of code lines in which only the variable name differs possible.

(Step 3) Match detection
From all the substrings of the transformed token sequence, a pair of identical substrings is detected as a clone pair.

Code Clone Categorization

A pair of code lines that are the same (or similar) is called a code clone pair (or simply a "clone"). Figure 1 shows a diagram of code clone pairs. As shown in the figure, there are two types of code clone pairs that differ in their effects on the degree of coupling between modules. Hereafter, a "module" means a source file, but in essence it can mean a cohesive unit of software, e.g. package or a library.

FIGURE 1
Classification of Code Clone



(1) In-module clone pairs
When the two lines of code comprising a code clone pair are in the same module, they are called an in-module clone pair. The effect on the degree of coupling between modules is considered to be slight.

(2) Inter-module clone pairs
When the two lines of code comprising a code clone pair are in different modules, they are called an inter-module clone pair. Because there are pieces of code performing the same processing in two modules, such pairs are assumed (believed) to strengthen the degree of coupling between modules.

In general, making revisions and changes is difficult if two modules are strongly linked, so inter-module clone pairs can significantly reduce the maintainability of the software. Note that for the purposes of this paper, clone pairs having overlapping code lines (self-crossing clone pairs) were not detected as code clones.

Code Clone Metrics

The following three metrics were employed for this paper.

- Clone inclusion rate (CRate)
The number of tokens included in code clone lines, divided by the number of tokens in the entire program, multiplied by 100.
- In-module clone inclusion rate (inMCRate)
The number of tokens in an in-module clone line, divided by the total number of tokens in the program, multiplied by 100.

- Inter-module clone inclusion rate (interMCRate)
  The number of tokens in an inter-module clone line, divided by the total number of tokens in the program, multiplied by 100.

Visualization Tool for Code Clone Analysis

In order to analyze code clones detected by CCFinder, we developed a code clone visualization tool called *CloneWarrior* (8). Figure 2 shows the architecture of CloneWarrior, which consists of a Graphical User Interface (GUI), GUI controllers (File list manager, Code clone list manager and Source code editor) and Code clone storage. The user of CloneWarrior first specifies a criterion for code clone detection, i.e., the least length of a clone fragment.[1] Next, the user specifies a set of source files that need to be analyzed. After all the clones are detected from the given source files and stored in the code clone storage, the user can analyze the clones through three views (the file list view, the code clone list view, and the source code editor). Figure 3 shows a GUI of CloneWarrior. In Figure 3, (a) and (b) show file lists of target source code. CCfinder detects code clones between source files in (a) and in (b). In this paper we specified the same file list in (a) and (b). Detected code clone pairs are shown in the code clone list view (d). Source codes of clones are shown in the source code editor (c).

When the user clicks one of the files in (a) or in (b), a list of code clones included in the clicked file are shown in (d), and when the user clicks one of the clones in (d), source lines of the clicked clone are highlighted in (c). Therefore, the user can easily recognize the location of clones without paying attention to token criteria.

CODE CLONE ANALYSIS

Analyzed Software

We analyzed 115 software packages selected at random from the open source software products in the GNU Project (15), and ten individual software programs considered by the open source community to be examples of successful programs (1) (Linux kernel, emacs, apache, and others) for a total of 125 programs. (The names of the software can be found in the appendix.) All of the programs were written in C. These open source software programs (called "free software") meet nine specific conditions that include not only open sources, but also freedom of distribution and the possibility of distributing modified versions of the source code (1).

The scale of the 125 programs we analyzed (in terms of LOC: Lines of Code) was on average 55,000 LOC, with the smallest containing 478 LOC, and the largest about 2,670,000 LOC.

Analysis Procedure

The procedure we followed in our analysis is as follows.

(1) First, we detected the code clones in the 125 software products using CCFinder and performed a code clone metrics measurement. In order to avoid identifying accidentally matching code lines (that were not the result of copying and pasting) as code clones, we defined the code clones for detection to be code line pairs having 50 or more matching tokens. Code pairs with fewer than 50 token matches were ignored. We determined this value of (50 tokens) based on past research. Ducasse et al. (9) used ten LOC and Baker (2) and Monden et al. (29) used 30 LOC as the number of minimum clone length. We considered using their mean value of 20 LOC, which is about 50 tokens in C language, but in the software we used for our analysis, 50 tokens were equivalent to 21 LOC on average.

(2) For software with many code clones, we examined the modules (files) having a high clone inclusion rate, using CloneWarrior to identify the types and causes of the code clones. We also considered methods of removing the code clones.

(3) Next, we performed statistical analysis of the code clone causes identified in (2) from the 125 software programs. We also considered standard values for the code clone inclusion rates.

Overview of the Software Measurement Results

Table 1 shows an overview of the results of the code clone measurement. On the left columns for the number of modules, the number of lines, the clone inclusion rate, the in-module clone inclusion rate, and the inter-module clone inclusion rate are provided. In each column is the average, median, maximum, minimum, and standard deviation values.

As shown in Table 1, although a program with the extremely high clone inclusion rate of 61.2% exists, 26 programs with no clones at all also exist. The average clone inclusion rate was 11.3%. The number of in-module clones was greater than the number of inter-module clones---3.4 times greater in terms of the clone inclusion rate.

From these results, we learned that there are large differences in the clone inclusion rate depending on the software, and that even open source software, which assumes the publication of the source code, sometimes contains many clones.

Clones in Software with High Clone Inclusion Rates

Table 2 shows the code clone measurement results for the ten software programs with the highest clone inclusion rates. In this section, we discuss the typical clones included in these software programs and investigate their characteristics.

(1) Clones caused by platform-dependent code

The software with the highest clone inclusion rate was superopt. The main clones were due to code that was dependent on the platform (CPU). This software outputs assembly code for a given arithmetic expression, and it is ported for a wide variety of CPUs. An investigation of the clone lines revealed that, as shown in Figure 4 (a), the assembly code output processing for each operator was written in switch-case statements. Further, as shown in Figure 4 (b), where the processing code depends on each CPU instruction set but the instruction sets were similar to each other, clones were formed. However, the CPU-dependent processing code and the independent code were clearly separated, so that the code had a high readability. There is the risk that readability would be reduced if an attempt were made to share the processes dependent on the CPU in order to remove clones. All the clones appeared in a very regular manner, and are

[1]Note: Its default value is 50 tokens. The average number of tokens per LOC (lines of code) was 2.4 in our analysis of C source codes, so the user can estimate roughly the number of tokens from LOC. We believe that the number if tokens is a better criterion than the LOC because the LOC is much too sensitive to a programming style, to comments, and to white spaces, etc.

contained within a module; therefore, modification could be made relatively simply by using the find/replace feature of a text editor. These clones can be considered allowable clones. In the case of superopt, the separation of the non-dependent code and the independent code is successfully achieved, thus making the code comprehensible even though it contains clones. However, in other cases (i.e., when such separation is well not archived), removing clones by sharing the non-dependent codes would be desirable

FIGURE 2
Architecture of Clone Warrior

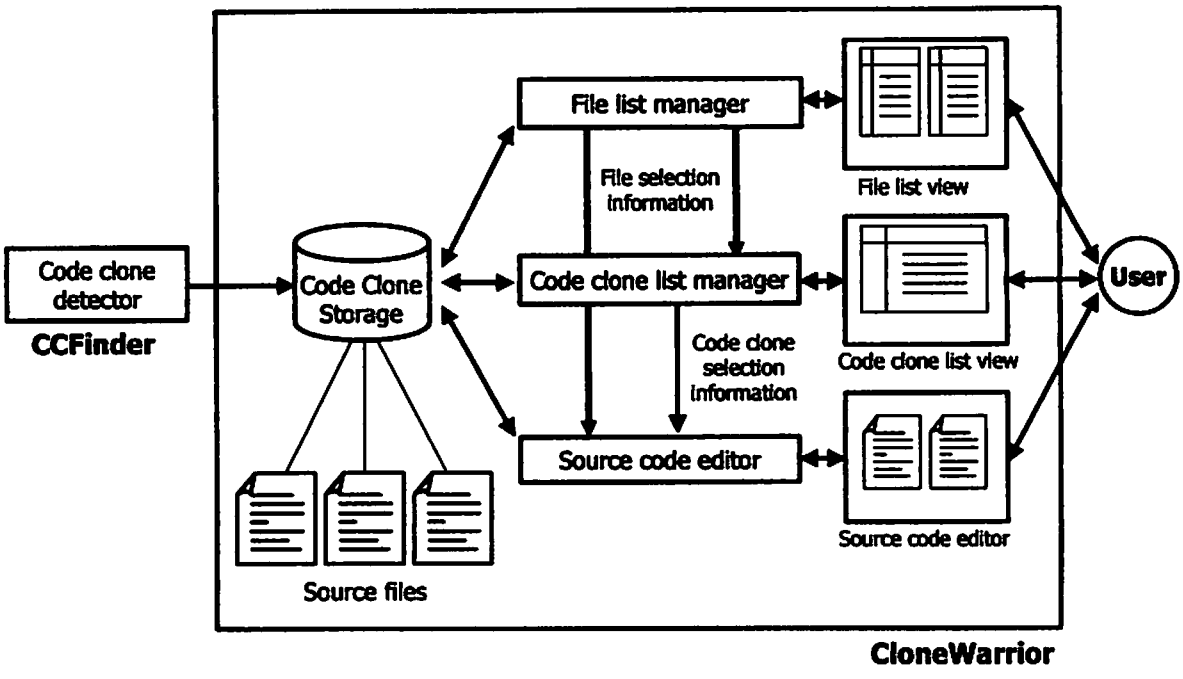


TABLE 1
Overview of Results of Code Clone Measurement

	# of Modules	LOC	CRate	inMCRate	interMCRate
Average	86	55,675	11.3%	9.1%	3.0%
Median	22	11,297	8.7%	5.9%	0.8%
Max	3,755	2,678,939	61.2%	61.2%	37.6%
Min	1	478	0.0%	0.0%	0.0%
Std.dev.	340	242,695	11.464	10.107	5.657

TABLE 2
The Top 10 Highest Clone Rate Software

Program	# of Modules	LOC	CRate	inMCRate	interMCRate
superopt-2.5	1	3,080	61.2%	61.2%	0.0%
chemtool-1.5	12	27,254	51.6%	26.5%	37.6%
gwcc-0.9.8	19	9,332	47.0%	47.0%	0.0%
bucob-0.1.2	43	44,828	43.7%	9.9%	35.5%
xcdroast-0.98	17	36,193	42.2%	39.4%	15.5%
rubrica-0.9.1	57	30,542	35.3%	34.1%	10.7%
gaby-2.0.2	106	60,652	33.3%	27.4%	15.9%
galculator-0.8	8	2,999	32.5%	32.5%	0.0%
pitchtune-0.0.3	10	4,373	31.9%	31.9%	0.0%
hme-1.3.1	17	7,211	30.9%	26.6%	7.4%

FIGURE 3
Screenshot of Clone Warrior

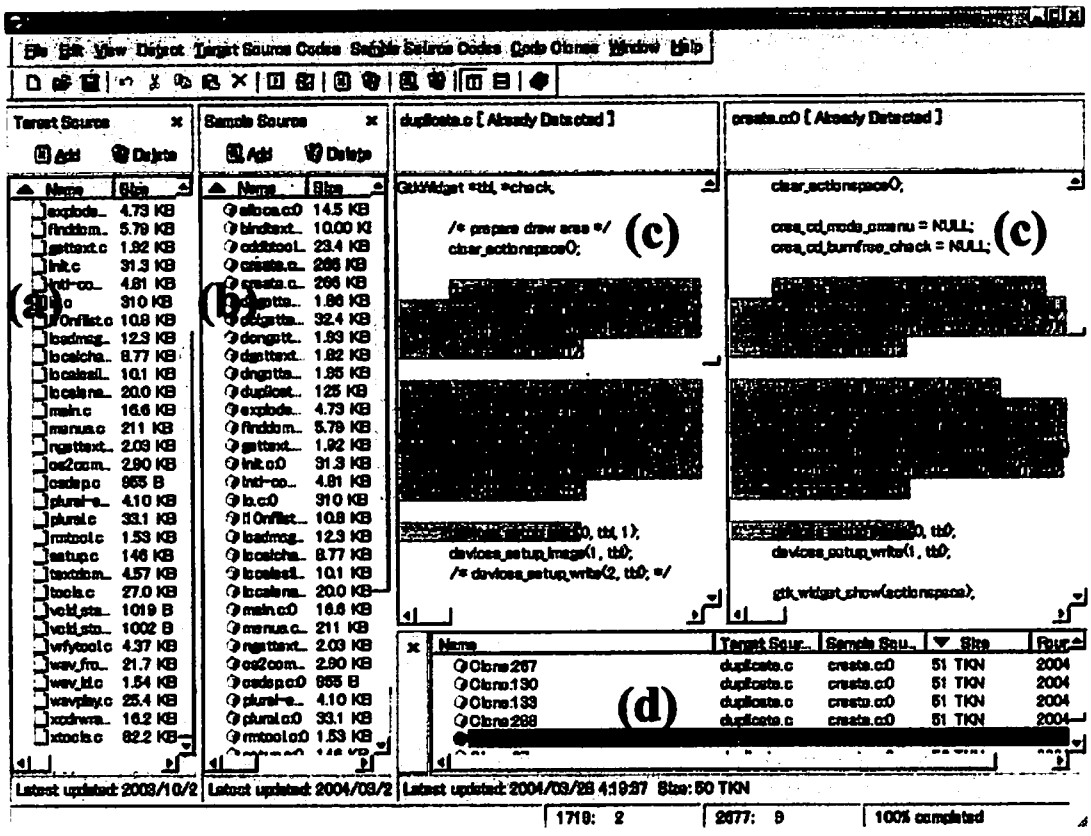


FIGURE 4
Example of Code Clone in Superopt

```
case LSHIFTR_CO:
  printf("shrl      %s,%s",NAME(s2),NAME(d));break;
case ASHIFTR_CO:
  printf("sarl      %s,%s",NAME(s2),NAME(d));break;
case SHIFTL_CO:
  printf("shll      %s,%s",NAME(s2),NAME(d));break;
case ROTATEL_CO:
  printf("roll      %s,%s",NAME(s2),NAME(d));break;
```

(a) Code clone found in switch case statement

```
#elif ALPHA
    ....
#elif HPPA
    ....
#elif SH
    ....
```

(b) CPU dependent code

(2) Clones caused by a program translator

The clones in Chemtool, the product with the second-highest code clone inclusion rate, were generated by the program translator. For some of the modules in this software, a Pascal-to-C translator (31) was used to automatically convert the source code from Pascal into C. The general-use macro definitions and function definitions became in-module and inter-module clones. If these modules are not counted, the in-module clone inclusion rate for Chemtool is 5.6%, and the inter-module clone inclusion rate is 0.1%, both lower than the average. The translated source code is also provided in Pascal, so developers are not likely to be troubled by the clones included in the C-language program.

(3) Clones caused by GUI builder tools

The products GWCC (GNOME Workstation Command Center), galculator (scientific calculator) and pitchtune (musical instrument tuner) contain codes generated by the GUI (Graphical User Interface) builder tool GLADE (14), and they have formed in-module clones. The code generated by GLADE is not intended to be manually manipulated (it is outside of the maintenance scope), so the clones do not reduce maintainability. Further, when the codes that were generated by GLADE are removed from these three software products, the in-module clone inclusion rates fall to 2.5%, 0%, and 0%, respectively.

(4) Clones generated by lexical/syntax analyzers

The software Bucob (a COBOL compiler) has a high inter-module clone inclusion rate (35.5%). However, our investigation revealed that 9 modules were lexical analysis modules generated by flex (11). An additional 3 modules were syntax analysis modules generated by GNU Bison (16). These modules are not intended to be manually manipulated, so they do not reduce maintainability.

(5) Clones related to the generation of GUI parts

The software products xcdroast (CD writing software), rubrica (address book), and gaby (database) are all intended for use with X-Window. In these products, many code clones were found in the processing related to the GUI. Each of the software products includes a GUI using the GUI toolkit GTK+ (18), but does not include any code generated by GLADE or similar products. The products were manually coded. Figure 5 shows a typical example of a code clone pair found in these software products. Each clone line generates a different GUI item and draws it on the screen. Furthermore, these codes appear to have been generated by copy and paste. These software products have many GUI items, so there are many other similar clone lines, causing both the in-module and inter-module clone inclusion rates to rise.

Two possible ways to remove or to reduce code clones from GUI related code exist. One way uses the inheritance mechanism of GTK+ and the other, a wrapper function to combine code clones. Generally, the former technique (using inheritance) is referred to as differential programming or programming by modifications. Typically, we define the common properties in the superclass, and then we only need to append or to redefine the differences in the subclass. If the inheritance is used appropriately, we will gain both code reduction and improved maintainability (12). In the case of Figure 5, one possible way to use the inheritance is to develop a subclass of GtkTable and to write its initialization function containing "gtk_table_set_row_spacings(GTK_TABLE(tbl), 10)" and "gtk_table_set_col_spacings(GTK_TABLE(tbl),10)" so that these two statements are removed from the original code. However, writing a subclass itself requires a certain amount of extra code; thus, this process sometimes increases the total amount of code. Moreover, improper use of inheritance may cause an ill-structured design, which makes maintenance even more difficult.[2]

On the other hand, using a wrapper function is a coding-level technique rather than a design-level technique. In this technique, we combine a recurring pattern (i.e. a code clone) into a new function called a wrapper function. Then, all the recurring patterns in the source code are replaced with a call to the wrapper function so that the patterns disappear and the total amount of code is reduced. In the case of Figure 5, we could write the wrapper function "GtkWidget *my_gtk_table_create (guint rows, guint columns, GtkWidget *packedTo)" containing gtk_table_new, gtk_table_set, gtk_box_pack_start, and gtk_widget_show. However, we sometimes need more global (i.e. design level) improvement than local (coding level) improvement. In such a case, the developer should consider design-level improvement, which includes the use of inheritance.

(6) Clones caused by copy-and-past programming

The software HME (Height Map Editor) contains many functions that are similar to each other, and we found many code clones that seemed to have been generated using copy and paste. For example, the clone fragment show in Figure 6a was contained in two functions. This clone fragment could be easily removed by combining the clone lines into a new (wrapper) function. In contrast, the clone fragment shown in Figure 6b is contained in three functions, but it would not necessarily be an improvement to combine them into a wrapper function. These clone lines contain many local variables as well as assignment statements for the local variables. This indicates that simply combining clone fragments into a wrapper function would likely degrade the maintainability. In this program, "start_x," "end_x," "start_y," and "end_y" are local variables, and values are assigned to these variables. Therefore, if we simply build a new function to combine code clones, we need to pass all these local variables to the function by pointers. This is not an elegant solution because passing many variables by pointers will strengthen the coupling between functions and thus may degrade maintainability. In addition, another concern is that there are too many global variables in this program ("selection_x_1," "selection_x_2," "selection_y_1," and "selection_y_2"), and they are not combined as a struct data type. A more elegant solution would be to construct a struct data type including these global variables as well as local variables, to build a wrapper function as shown in Figure 7. In Figure 7, a new data type "struct Selection" was defined and its instance became an argument for the function "combined." We assume that "selection" is a local variable. However, this function works even if it is a global variable. (Note that since we are not developers of HME, better solutions might be developed.)

Analysis of Presence or Absence of Generated Code

The analysis described in the previous section revealed that there are many open source software products that contain automatically generated code using case tools, and the generated code is one cause of clone generation.

As shown in Table 3, out of 125 software products, 33 were found to contain automatically generated code. These products had higher average in-module and inter-module clone inclusions rates than the products that did not contain generated code (a significant difference with a 5% level of significance).

An investigation of each generated code revealed that the following code generation tools were used: GUI builder tools

[2]Note: For the proper use of inheritance, please refer to Fowler's book (12).

[GLADE (14), SWIG (35), and fdesign (10)], lexical analysis/ syntax analysis program generation tools [flex (11) and GNU Bison (16)], tools for generating code required for communication [Orbit id (30) and RPCGen (320], a text-processing tool [gettext (13)], a source code translator [Pascal-to-C translator (31)], and a hash function generator [gperf (17)]. Researchers and developers who analyze code clones need to be aware of the presence of code generated by tools such as these.

FIGURE 5
Example of Code Clone Pair in scdroast

```
tbl = gtk_table_new(3,8,TRUE);
gtk_table_set_row_spacings(GTK_TABLE(tbl),10);
gtk_table_set_col_spacings(GTK_TABLE(tbl),10);
gtk_box_pack_start(GTK_BOX(vbox),tbl,FALSE,FALSE,
10);
gtk_widget_show(tbl);
l1 = rightjust_gtk_label_new(text(103));
gtk_table_attach_defaults(GTK_TABLE(tbl),l1,0,2,0,1);
gtk_widget_show(l1)
```

Clone pair

```
tbl = gtk_table_new(2,16,TRUE);
gtk_table_set_row_spacings(GTK_TABLE(tbl),10);
gtk_table_set_col_spacings(GTK_TABLE(tbl),10);
gtk_box_pack_start(GTK_BOX(actionspace),tbl,FALSE,
FALSE,10);
gtk_widget_show(tbl);
l1 = rightjust_gtk_label_new(text(219));
gtk_table_attach_defaults(GTK_TABLE(tbl),l1,0,6,0,1);
gtk_widget_show(l1);
```

FIGURE 6
Example of Clone Fragments in HME

```
sum = *(terrain_height + y * WIDTH + x);
sum += *(terrain_height + ((y - 1) * WIDTH) + x - 1);
sum += *(terrain_height + ((y - 1) * WIDTH) + x);
sum += *(terrain_height + ((y - 1) * WIDTH) + x + 1);
sum += *(terrain_height + (y * WIDTH) + x - 1);
sum += *(terrain_height + (y * WIDTH) + x + 1);
sum += *(terrain_height + ((y + 1) * WIDTH) + x - 1);
sum += *(terrain_height + ((y + 1) * WIDTH) + x);
sum += *(terrain_height + ((y + 1) * WIDTH) + x + 1);
if(sum%9>4)sum = (sum / 9)+1;
else sum=sum/9;
*(terrain_height + y * WIDTH + x) = sum;
```

(a) Easily removable clone fragment

```
if (!terrain_height)return;
change_cursor(cursor_wait);
copy_to_undo_buffer();
if(selection_x_1<selection_x_2){
        start_x=selection_x_1;
        end_x=selection_x_2;
}else{
        start_x=selection_x_2;
        end_x=selection_x_1;
}
if(selection_y_1<selection_y_2){
        start_y=selection_y_1;
        end_y=selection_y_2;
}else{
        start_y=selection_y_2;
        end_y=selection_y_1;
}
```

(b) Not easily removable clone fragment

FIGURE 7
An Example of Clone Elimination

```
struct Selection{
 int x1, y1, x2, y2;
 int startx, endx, starty, endy;
};

int combined (struct Selection *selection){
{
 if (!terrain_height) return FALSE;
 change_cursor(cursor_wait);
 copy_to_undo_buffer();
 if(selection-> x1 < selection-> x2){
                selection->startx = selection->x1;
                selection->endx = selection->x2;
 }else{
                selection->startx = selection->x2;
                selection->endx = selection->x1;
 }
 if(selection->y1 < selection->y2){
                selection->starty = selection->y1;
                selection->endy = selection->y2;
 }else{
                selection->starty = selection->y2;
                selection->endy = selection->y1;
 }
 return TRUE;
}
```

TABLE 3
Relation between Generated Code and Clone Rate

Code Generation	# of Programs	CRate	inMCRate	interMCRate
Yes	33	15.8%	11.6%	5.5%
No	92	9.7%	8.2%	2.1%

Analysis of the Presence or Absence of a GUI

The analysis described in a previous section revealed that the codes for creating GUI objects are considered as a cause of clones in GUI software.

As shown in Table 4, 56 of the 125 software products had GUIs. Compared to products without GUIs (command line interface products), the products with GUIs showed on average higher in-module and inter-module clone inclusion rates (a significant difference with a 5% level of significance).

TABLE 4
Relation between GUI and Clone

GUI	# of Programs	CRate	inMCRate	interMCRate
Yes	56	14.9%	12.4%	3.9%
No	69	8.3%	6.5%	2.2%

Standard Values for Code Clone Inclusion Rates

We classified software into four categories, depending on the presence of generated code and GUIs, and calculated the average in-module and inter-module clone rates for each category (Table 5). We expect that these values might be helpful as a guide to software developers. For example, in the development of software without generated code and without a GUI, if the in-module clone inclusion rate is about 8.2% and the inter-module clone inclusion rate is about 1.3%, the amount of clones in the software is average for open source software in general. On the other hand, if the clone inclusion rates are higher than these average values, the software may be difficult to maintain, and it may be necessary to remove some clones.

RELATED WORK

This section gives a brief survey of existing papers on code clone analyses, especially analyses on industrial systems (i.e. not open source). Table 6 shows the result of our survey. Although

the CRate (clone inclusion rate) depends on the clone detection tool being used and on the minimum clone length, which is a criterion for clone detection, all the industry products appearing in existing papers had a CRate greater than 11.3%, which is the average CRate in our case of 125 open source software products. From this result, we expect that many of open source software products will contain relatively few code clones, which we hoped should give us a guideline for the allowable amount of code clones.

TABLE 5
Average Clone Inclusion Rate of Four Kinds of Software

(a) Clone Inclusion Rate

		Code Generation	
		Yes	No
GUI	Yes	16.9%	13.8%
	No	13.4%	7.2%

(b) In-module Clone Inclusion Rate

		Code Generation	
		Yes	No
GUI	Yes	13.6%	11.7%
	No	7.9%	6.1%

(c) Inter-module Clone Inclusion Rate

		Code Generation	
		Yes	No
GUI	Yes	4.9%	3.4%
	No	6.6%	1.3%

CONCLUSIONS

In this paper, we examined 125 open source software products written in C to analyze the code clones contained in the software. Our main findings are outlined below.

- A great variation in the clone inclusion rates existed, with the average being 11.3%, the minimum 0%, and the maximum 61.2%. On average, the in-module clone inclusion rate was greater than the inter-module clone inclusion rate by 3.4 times.
- Software that has been ported for multiple platforms and which includes codes dependent on each platform tend to have many clones. In software of this type, clones among platform-dependent codes are allowable, but clones among non-dependent codes are not necessarily allowable.
- Codes that are generated by a program generator tend to become clones, but as long as the program generator and the input for it are provided, these clones will not necessarily reduce maintainability.
- In software containing GUIs, codes related to the generation of GUI parts can easily become clones, so attention to the design phase is required. Where GTK+ is used, the GUI object inheritance feature can be used to reduce clones.
- The part of clones generated by copy and paste can be eliminated by combining them into a new function called a wrapper function. However, if the clones involve a number of assignment statements for local variables, it is not desirable to simply combine the clones into a wrapper function. In such cases, a more broad-ranging improvement of the design is required.
- For software that included neither generated code nor a GUI, the average in-module clone inclusion rate was 8.2%, and the average inter-module clone inclusion rate was 1.3%. These results can serve as a quantitative guideline for software developers.

TABLE 6
Overview of Results in Previous Research

Paper	Software	Language	Loc	Clone Tool	Minimum Clone Length	CRate
Baker (2)	AT&T System	C	200,000	Dup	30LOC	23.0%
					15LOC	38.0%
Baxter (6)	Process-control	C	400,000	CloneDR	n.s.	12.7%
Ducasse et al. (9)	DatabaseServer	Smalltalk	245,000	DupLOC	10LOC	36.4%
Ducasse et al. (9)	Payroll system	COBOL	40,000	DupLOC	10LOC	59.3%
Monden et al. (29)	Mainframe MIS	COBOL/S	1,000,000	CCFinder	30LOC	15.4 %
This paper	125 open source	C	-	CCFinder	50 Tokens (21LOC)	11.3%

The software we examined for this paper is not exclusive so other researchers may use the same source codes to conduct follow-up tests, and thereby improve the accuracy of our analysis and the reliability of our results.

In the future, we intend to perform a deeper analysis on a large number of software products, and to perform similar analysis for software in languages other than C.

ACKNOWLEDGEMENT

We wish to thank Dr. Shuuji Morisaki of Internet Initiative Japan Inc. and Dr. Masatake Yamato of Red Hat, K.K. for their enthusiastic debate and advice. This study was supported by the Industrial Technology Research Grant program from the New Energy and Industrial Technology Development Organization(NEDO) of Japan.

REFERENCES

1. Aoyama, M. "Perspective on Open Source Software," **IPSJ Magazine**, 43:12, December 2002, pp. 1319-1324.
2. Baker, B.S. "A Program for Identifying Duplicated Code," **Proc. 24th Symposium on the Interface Computing Science and Statistics**, Texas, Maerch 18-21, 1992, pp. 49-

57.
3. Baker, B.S. "On Finding Duplication and Near-duplication in Large Software System," **Proc. 2nd IEEE Working Conf. on Reverse Eng.**, Toronto, Ontario, July 14-16, 1995, pp. 86-95.
4. Balazinska, M., E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis. "Measuring Clone Based . Reengineering Opportunities," **Proc. 6th IEEE Intl. Symposium on Software Metrics**, Boca Raton, FL, Nov. 4-6, 1999, pp. 292-303.
5. Balazinska, M., E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis. "Partial Redesign of Java Software Systems Based on Clone Analysis," **Proc. 6th IEEE Working Conf. on Reverse Eng.**, Atlanta, GA, oct. 6-8, 1999, pp. 326-336.
6. Banerjee, D. and C. Simpson. "A Formal Methodology for Subject Area Identification in IEF™ Re-engineering Projects," **J. of Computer Information Systems**, 34:4, 1994, pp. 37-39.
7. Baxter, I.D., A. YUahin, L. Loura, M. Sant'Anna, and L. Bier. "Clone Detection Using Abstract Syntax Trees," **Proc. IEEE Intl. Conf. on Software Maintenance**, Bethesa, MD, Nov. 16-20, 1998, pp. 368-377.
8. CloneWarrior homepage: http://se.aist-nara.ac.jp/clonewarrior/
9. Ducasse, S., M. Rieger, and S. Demeyer. "A Language Independent Approach for Detecting Duplicated Code," **Proc. IEEE Intl. Conf. on Software Maintenance**, Oxford, UK, Aug. 30-Sept. 3, 1999, pp. 109-118.
10. fdesign(XForms) homepage: http://world.std.com/~xforms/
11. flex homepage: http://www.gnu.org/software/flex/
12. Fowler, M. **Refactoring: Improving the Design of Existing Code.** Addison Wesley, 1999.
13. Gettext homepage: http://www.gnu.org/software/gettext/
14. Glade homepage: http://glade.gnome.org/
15. GNU Project and FSF: http://www.gnu.org/
16. GNU Bison homepage: http://www.gnu.org/software/bison/bison.html
17. gperf homepage: http://www.gnu.org/software/gperf/
18. GTK+ homepage: http://www.gtk.org/
19. Imai, T., Y. Kataoka, and T. Fukaya. "Evaluating Software Maintenance Cost Using Functional Reduncancy Metrics," **Proc. 26th Intl. Computer Software and Applications Conference**, Oxford, England. Aug. 26-29, 2002, pp. 299-306.
20. Inoue, K., T. Kamiya, and S. Kusumoto. "Code-Clone Detection Methods," **Computer Software**, 18:5, 2001, pp. 47-54.
21. Kamiya, T. and Y. Ueda. "CCFinder/Gemini Web Site," http://sel.ics.es.osaka-u.ac.jp/cdtools/index.html.en
22. Kamiya, T., F. Ohata, K. Kondou, S. Kusumoto, and K. Inoue. "Maintenance Support Tools for Java Programs: CCFinder and JAAT," **Proc. 23rd Intl. Conf. on Software Eng.**, Toronto, Canada, May 12-19, 2001, pp. 837-838.
23. Kamiya, T., S. Kusumoto, and K. Inoue. "CCFinder: A Multi-linguistic Token-based Code Clone Detection System for Large Scale Source Code," **IEEE Trans. Software Engineering**, 28:7, 2002, pp. 654-670.
24. Johnson, J.H. "Identifying Redundancy in Source Code Using Fingerprints," **Proc. IBM Centre for Advanced Studies Conference**, Toronto, Ontario, October 24-28, 1993, pp. 171-183.
25. Johnson, J.H. "Substring Matching for Clone Detection and Change Tracking," **Proc. IEEE Intl. Conf. on Software Maintenance**, Victoria, British Columbia, Sep. 19-23, 1994, pp. 120-126.
26. Kontogiannis, K.A., R. Mori, E. DeMerlo, M. Galler, and M. Bernstein. "Pattern Matching Techniques for Clone Detection and Concept Detection," **J. of Automated Software Eng.** 3, 1996, pp. 770-1108.
27. Laguë, B., E.M. Merlo, J. Mayrand, and J. Hudepohl. "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," **Proc. IEEE Intl. Conf. on Software Maintenance**, Bari, Italy, Sep. 29-Oct. 3, 1997, pp. 314-321.
28. Mayland, J., C. Leblanc, and E.M. Merlo. "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," **Proc. IEEE Intl. Conf. on Software Maintenance**, Monterey, CA, Nov. 4-8, 1996, pp. 244-253.
29. Monden, A., D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. "Software Quality Analysis by Code Clones in Industrial Legacy Software," **Proc. 8th IEEE Intl. Software Metrics Symposium**, Ottawa, Canada, June 4-7, 2002, pp. 87-94.
30. Orbit homepage: http://orbit-resource.sourceforge.net/
31. Pascal to C translator homepage: http://www.mpsinc.com/pas2c.html
32. rpcgen (NetKit) distribution: http://freshmeat.net/projects/netkit/
33. Saiedian, H., M. Zand, and J. Welborn. "On Challenges of Reverse Engineering for Large Software Systems," **J. Computer Information Systems**, 33:1, 1992, pp. 37-40.
34. Sharpe, S. "Unifying Theories of Program Comprehension," **J. Computer Information Systems**, 38:1, 1997, pp. 86-93.
35. SWIG homepage: http://www.swig.org

APPENDIX

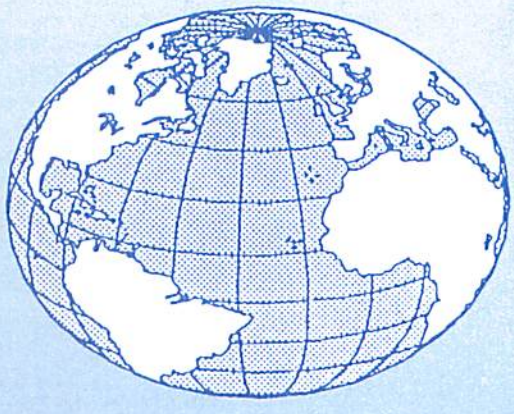
Analyzed software programs are listed below.

- 115 software programs downloaded from GNU project:
  cdcd-0.5.5, glabels-1.89.0, gnutrition-0.3, ffproxy-1.3, galculator-0.8, cdparanoia-alpha9.8, gnofin-0.8.4, abcm2ps-1.3.3, pound-1.0, vp7wkp-0.5.0, cdrtool-1.11, gnumeric-1.0.9, myleague-0.9.2, kadet-2.0.9, eukleides-0.9.2, glame-0.6.2, oleo-1.99.16, scoreboard-0.9, vnc_reflector-1.1.9, aide-0.8, lame-3.89, privoxy-3.0.0, abc2ps-1.3.3, gwcc-0.9.8, finger-1.37, normalize-0.7.4, gnubg-0.1.2, units-1.55, gnobog-0.4.2, fwlogwatch-0.9, radioactive-1.4.0, asteroids-0.2.2, bc-1.06, rws-0.9.6, gringotts-0.6.2, sound-1.9.1, dumb0.13.12, statist-1.0.1, fingerd-1.4, openssh-3.4, gnuradio-0.3, freeciv-1.12.0, dap-1.7, bucob-0.1.2, siege-2.53, grip-2.99.2, gnuboy-1.0.3, euler-1.60, ccache-1.9, tripwire-2.3.2-1, easytag-0.2.1, gnugo-3.2, spline-1.1, strace-4.4.1, zorp-1.4.6, sweep-0.1.1, xtux-20010107, gzip-1.2.4, check-0.8.3, gtk-knocker-0.6.6, xcdroast-0.98, stereograph-0.31, avdbtool-0.3, led-2.0, firestarter-0.9.0, gmmusic-1.1.11, plotutils--2.4.1, linuxtart-3.0.4, memwatch-2.67, razorback-1.0.2, gneomereadio-1.0, lib3ds-1.2.0, mtools-3.9.8, superopt-2.5, sharesecret-0.2.0, gcal-3.0.1, gmandel-1.1.0, reed-5.3, calcoo-1.3.13, ccvssh-0.9.1, Kalarm-0.7.5, danpei-2.8.6, battstat-2.0.11, gflow-0.9.13, cvsadmin-1.0.2, xdiary-1.32, filmgimp-0.6, gbonds-0.7.5, postgis-0.7.3, cvsgraph-1.3.0, gaby-2.0.2, geomview-1.8.1, gneuro-0.9.1, sunclock-3.46, expect-5.38, gnucash-1.6.6, cave-1.0, lr-1.7, hh2000-0.7, sh-utils-2.0, rubrica-0.9.1, text-color-1.0.3.1, tea-applet-1.1, hme-1.3.1, srm-1.2.4, toutdoux-1.2.7, lifelines-3.0.14, balance-2.33, gpsdrive-1.26, cvsps-1.3.3, barcode-0.98, pitchtune-0.0.3, fnord-1.5, chemtool-1.5, install-log-1.9
- 10 software programs considered by the open source community to be examples of successful programs:
  perl-5.005, emacs-21.2, csv-1.11.2, httpd-2.0.43, apache-1.3.27, samba-2.2.4, sendmail-8.12.6, tcl/tk-8.4.1, python-2.1.3, linux-kernel-2.4.9

JOURNAL OF COMPUTER INFORMATION SYSTEMS

JCIS is the official journal of the

INTERNATIONAL ASSOCIATION FOR COMPUTER INFORMATION SYSTEMS



VOLUME XLV
NUMBER 3
SPRING 2005