Cloning by Accident:

An Empirical Study of Source Code Cloning Across Software Systems

Raihan Al-Ekram, Cory Kapser, Richard Holt, Michael Godfrey
Software Architecture Group (SWAG),
School of Computer Science, University of Waterloo
200 University Avenue West, Waterloo, Ontario, N2L 3G1
Email: {rekram, cjkapser, holt, migod}@swag.uwaterloo.ca
Phone: (519) 888-4567 x 2988, Fax: (519) 885-1208

Abstract

One of the key goals of open source development is the sharing of knowledge, experience, and solutions that pertain to a software system and its problem domain. Source code cloning is one way in which expertise can be reused across systems; cloning is known to have been used in several open source projects, such as the SCSI drivers of the Linux kernel [16] . In this paper, we discuss two case studies in which we performed clone detection and analysis on several open source systems within the same domain: we examined nine text editors written in C, and eight X-Windows window managers written in C and C++. To our surprise, we found little evidence of "true" cloning activity, but we did notice a significant number of "accidental" clones --- that is, code fragments that are similar due to the precise protocols they must use when interacting with a given API or set of libraries. We further discuss the nature of "true" versus "accidental" clones, as well as the details of our case studies.

Key Words: Source Code Cloning, Cross Cloning, Code Resuse and Open Source.

1 Introduction

Open source software (OSS) is becoming an increasingly important part of day-to-day computing. Universities are beginning to use OSS projects such as Linux and FreeBSD as an operating system in both the teaching and research environments. We see evidence of industries acknowledgement of the importance of open source software as large projects such as OpenSolaris, Mozilla, and Eclipse are released for public consumption.

OSS projects are as diverse as they are plentiful. As of May 17, 2005 there were 100,172 projects registered on SourceForge, a common portal for OSS projects. These projects range from simple text editors to database engines. As in closed source software projects (CSS), there tends to be several competing products in any given domain. Unlike CSS projects, OSS projects can take advantage of the knowledge and source of products within their domain. To many, such as the people of the GNU Projects, source code is considered knowledge that should be shared with mankind [12] . Copying of and incorporating code or parts of code is an acceptable practice in the OSS community.

This poses an interesting question: How much knowledge is reused from similar products in a single domain? While this question is somewhat difficult to answer through empirical study, we can get some idea about the nature of the answer by studying a simpler question. In this paper the question we investigate is: How much and what kind of code is duplicated from one software project to a new software in a given domain? In this question, we are concerned not with projects that formed by branching an existing project such as Mozilla and Firefox. Nor are we interested in projects that are directly based on another project, such as PDF document viewers kpdf and xpdf. Certainly these are examples o knowledge transfer, but not the type we are interested in. We would like to see the degree of knowledge transfer that occurs when an entirely new project begins. In order to answer the question, we use clone detection techniques to identify code duplication across software systems and use the measure of cloning as an indirect measure of reuse of knowledge from one software system in another.

Source code cloning can be defined as the act of intentional duplication of code fragments with possible customization [8] . Studies have shown cloning to be a common

phenomenon that generally occurs in large software systems [1] [21] [4] [2] [8] . A large software system typically consists of 10-15% of code duplication [1] . There are two typical scenarios that lead to code cloning within a software system. The first is the ad-hoc reuse of some existing code in the program that implements a similar functionality. Instead of forming a proper abstraction of the similar code a programmer may simply make a copy of it in order to save effort and time. Sometimes it can be difficult to identify a proper abstraction due to the presence of crosscutting concerns [14] . The second scenario leading to code cloning is the planned duplication of code. For example, software developers may clone code to avoid the formation of an abstraction due to strict performance requirement. This type of cloning is desirable. Another such desirable cloning is the code duplication between two architectural entities in order to maintain architectural clarity. Code fragments may also be accidentally identical. Technically these are not clones since they were not intentionally copied from each other. But the clone detection tools may identify them as clones since they look similar. We name such accidental identical code fragments *Clones by Accident*.

Cloning across software systems, on the other hand, can result either because of reuse of existing code from existing software implementing similar functionality or by accident. In this research we perform a detailed study on cross system. We select two problem domains and a number of software for each of the domains. For each group of software we perform clone detection across all the software in that domain. In our analysis of these systems, we wish to determine the amount of code cloning across the software systems. We would also like to analyze the types of clones that occur between the systems, the amount of code reuse from available software when developing a new software, the contribution of the problem domain towards cloning, the factors that contribute most to cross system cloning and what amount of cloning appears by accident.

The rest of this paper is organized as follows: Section 2 describes the approach taken to study the code duplication and hence knowledge reuse across software systems. Section 3 presents two case studies in two different problem domains: Text Editor and Window Manager. Each case study contains an analysis of cloning among the representative software in that domain and tries to answer the questions we are interested in. Section 4 gives a critical discussion on the results of the case studies. Section 5

discusses some related work in the area of clone detection and analysis. Finally Section 6 concludes the paper presenting the findings from our study.

2 Study Approach

We selected two problem domains to perform our study: Text Editors and Window Managers. We then selected several software projects in each domain to perform the cloning analysis. All of the selected software is open source in order to facilitate the reproducibility of the results.

Our approach to analyzing the clones between the software systems began with detecting the clones. There are numerous clone detection algorithms available [1] [13] [8] [19] [9] [21] [18] [10] [11] . We chose to use a tool that implements a parameterized string matching algorithm [1] [21] called *CCFinder* [21] . We chose to use *CCFinder* because of it has high recall [20] ; it detects most clones. Also, we expect that most copied code will have undergone change in naming to match naming conventions of the new project, which suggests a parameterized matching technique for clone detection is needed.

The post detection processing and analysis was performed using the *Clone Interpretation and Categorization Systems (CLICS).* One weakness of parameterized string matching techniques is that they can have low precision [20] . *CLICS* has several built in filters that remove a large number of the false positives from the data set, typically resulting in a 60% reduction in the number of code clone pairs [2] [3] . In addition to clone filters, *CLICS* provides strong clone navigation and data set refinement facilities, which were essential when finding and assessing commonalities between systems.

2.1 Clone Detection

CCFinder uses a parameterized string matching technique on the token stream of the input code. *CCFinder* currently works on C/C++, COBOL, EMACS Lisp, FORTRAN, Java and plain text source files. Because of the parameterized string matching technique *CCFinder* gives a high recall and low precision on the result [5] .

In our study, for each domain we are only concerned with clones between software systems, not clones within each software system. Using *CCFinder,* we detect clones across software systems only. The minimum number of tokens in a code fragment to be considered as clone is set to be 30. This size has been determined through testing and was found to be the best in terms of the precision/recall tradeoff.

2.2 Clone Analysis

In order to efficiently analyze the clones we use the *Clone Interpretation and Classification System (CLICS).* To improve the precision of the clone data set returned *by CCFinder, CLICS* filters out many false positives. These filters enforce stricter criteria for a clone match on segments of code that has simple structure [3] . The filters remove approximately 60% of the clone pairs in the original data set while removing no false positives or very few.

In these case studies we are interested in where the cloning is taking place, between what systems, and to what degree. Because of the nature of these questions, the features of *CLICS* that were of interest to us in were the clone navigation facilities, particularly those related to software architecture. *CLICS* also provides methods to query for clones related to particular entities in a software system, and entity being a subsystem, file, function or line of code. It also allows the user to query for clones related to a particular clone being viewed. These navigation features and query features were vital when investigating the cloning in these case studies.

3 Case Study

This section presents the results of two case studies we performed when investigating knowledge reuse through code cloning. In each case study we describe the study subjects. General clone statistics and a detailed analysis describing the cloning between the software systems is also given.

In the studies below we discuss cloning in terms of *clone pairs* and *clone groups*. A clone pair is two segments of code that have been found to match using some clone detection method. A clone group is a set of clones that occur in the same logical region of

code. Regions are defined as consecutive *prototypes* and *definitions* and individual *structs, unions, enumerations, macros* and *functions*. It is important to note that a high number of clone pairs between two systems do not necessarily mean a lot of cloned code. Often clone pairs can be overlapping, making it necessary to consider both the number of clone pairs that occur between two systems, and the number of clone groups. Often a high number of clone pairs and a low number of clone groups can be an indication of code segments with simple structure being matched in several different alignments. In our case studies we will present both metrics as we discuss the relations between the systems.

3.1 Text Editors

Our first case study is on the text editor domain. In this domain we have selected most commonly used text editors as guinea pigs for analysis: *Emacs*, *Gedit*, *Glimmer*, *Nano*, *Nedit*, *Pico, Vi, Vim* and *Xenon*. Table 1 lists the basic information about them.

Table 1: Text Editor Guinea Pigs

Text Editor	**Release**	**Files**	**LOC**	**Lang**	**Platform**	**Description**
Emacs	21.3	495	346,152	C	Console	GNU text editor
Gedit	2.8.2	121	46,877	C	X11	Gnome text editor
Glimmer	1.2.1	124	44,339	C	X11	X-based code editor
Nano	1.2.4	12	13,573	C	Console	Pico clone
Nedit	5.5	136	123,675	C	X11	Motif text editor
Pico	4.63	72	28,259	C	Console	Pine text editor
Vi	050325	65	34,190	C	Console	Traditional vi editor
Vim	6.3	94	259,116	C	Console	Vi clone
Xenon	0.6.6	78	16,940	C	X11	X-based text editor
Total		**1,197**	**913,121**			

The size of the source code of each editor ranges from some 13K to over 250K lines of code. All of them are written in C. Some of them use the console mode, whereas others use X11 graphical mode. The combined size of all the editors is 913K lines of code spanned in 1,197 files.

3.1.1 Cloning Statistics

Facts about the cloning in the text editors are gathered after applying the *CCFinder* and *CLICS* tools on the source code. Table 2 shows the cloning statistics of the software in the text editor domain.

Out of a total of 1,197 files in the 9 text editors 98 files contain one or more clones in it, which is over 8% of the total files. Out of 913,121 lines of code 4,482 lines contribute to cloning, which is a little below 0.5% of total LOC. There are 11,519 (=23,038/2) clone pairs and 373 (=746/2) clone groups across the text editors. Each clone has an average size of 7 lines of code.

Table 2: Cloning Stats for the Text Editors

Text Editor	Files			LOC			Clone	
	Total	with Clones	%	Total	with Clones	%	Pairs	Groups
Emacs	495	18	3.64	346,152	1,277	0.369	9,307	131
Gedit	121	19	15.7	46,877	470	1.002	517	161
Glimmer	124	24	19.35	44,339	1,165	2.627	561	183
Nano	12	3	25.00	13,573	95	0.670	101	7
Nedit	136	15	11.02	123,675	1,036	0.837	10,519	145
Pico	72	0	0.00	28,259	0	0.000	0	0
Vi	65	1	1.54	34,190	10	0.029	4	1
Vim	94	15	15.95	259,116	405	0.156	2,027	116
Xenon	78	1	1.28	16,940	24	0.142	2	2
Total	**1,197**	**98**	**8.19**	**913,121**	**4,482**	**0.491**	**23,038**	**746**

The editor *Nedit* contains the most amounts of clone pairs, over 90% of the total. Followed by *Emacs* containing 80% of total clone pairs. *Emacs* also contains the highest number of source lines that contributes to cloning, 1,277 LOC. *Glimmer* has the highest clone density. Over 2.6% of LOC of *Glimmer* are clones. Followed by *Gedit* with over 1% LOC.

3.1.2 Detailed Analysis

If we take closer look at the cloning we find that a total of 9,837 out of 10,519 clone pairs in *Nedit* comes from only seven out of 136 files in the source code. Only three functions of 109 in the *search.c* file contribute to over half of them. In *Emacs* 9,160 clone pairs out of 9,307 comes from only three files out of 495. Only one function out of 55 in the *lwlib-Xm.c* file contributes to 6,419 of them. In *Vim* a single file *gui_motif.c* out of 93 contributes to 1,936 out of 2,027 clone pairs in the system. Interestingly, almost all the clone pairs that originate in the files mentioned above of one system also terminate in the files mentioned above of another system. So most of the clones originating from the three files of *Emacs* form a pair with either the seven files in *Nedit* or the one file in *Vim*; forming the largest class of clones. We investigate the source code of these clone classes and determin that the purpose of these functions are setting up and creating widgets and dialog boxes for graphical user interaction. The reason there number of clone pairs compared to the number of files is that there are a lot of overlapping clones. This is due to the simple structure of the code involved.

Table 3 depicts fragments of code with four clone pairs between the *make_dialog* function of *lwlib-Xm.c* file of *Emacs* and *CreateFindDlog* function of *search.c* file of *Nedit*. Table 4 lists the files and functions involved in the clone class that accounts for over 10,000 of the total clone pairs. These clones clearly were not copied across different text editors; they are similar because they perform similar tasks in different systems. Therefore, the cloning across different editors in this clone class is by accident.

A second significant clone class contains clones between *mdi-routines.c* file of *Glimmer* and *gedit-mdi-child.c* file of *Gedit*. Although the class consists of only 127 clone pairs and 25 clone groups in it, it is significant because of the code in the clones. The codes in the two files deal with signaling among the MDI children when editing

multiple files in the editor. Table 5 depicts code fragments with a clone pair for the clone class given in Table 6. The clones in this class are specific to the text editor problem domain, since they both implement the support for editing multiple documents in the editors.

Table 7 shows the third interesting clone class, comprises of two files of the same name *regex.c* in two different text editors *Glimmer* and *Emacs* with 4,938 and 6.055 LOC respectively. There are a total 54 clone pairs 20 clone groups between the files. Investigating the source code reveals that *regex.c* is an extended regular expression matching and search library that is used by both the editors. But the version in *Emacs* is newer and contains additional helper macros and functions with the older macros and functions almost unchanged. Between them, over 800 LOC are cloned. This is clearly a case of code reuse between the two editors.

Table 3: Example Clones of Class 1

Emacs: lwlib-Xm.c/make_dialog
1093: XtSetArg(al[ac], XmNnumColumns, left_buttons + right_buttons + 1); ac++;
1094: XtSetArg(al[ac], XmNmarginWidth, 0); ac++;
1095: XtSetArg(al[ac], XmNmarginHeight, 0); ac++;
1096: XtSetArg(al[ac], XmNspacing, 13); ac++;
1097: XtSetArg(al[ac], XmNadjustLast, False); ac++;
...
1259: XtSetArg(al[ac], XmNleftAttachment, XmATTACH_WIDGET); ac++;
1260: XtSetArg(al[ac], XmNleftOffset, 13); ac++;
1261: XtSetArg(al[ac], XmNleftWidget, icon); ac++;
1262: XtSetArg(al[ac], XmNrightAttachment, XmATTACH_FORM); ac++;
1263: XtSetArg(al[ac], XmNrightOffset, 13); ac++;

Nedit: search.c/CreateFindDlog
1219: XtSetArg(args[argcnt], XmNrightAttachment, XmATTACH_FORM); argcnt++;
1220: XtSetArg(args[argcnt], XmNtopWidget, label1); argcnt++;
1221: XtSetArg(args[argcnt], XmNleftOffset, 6); argcnt++;
1222: XtSetArg(args[argcnt], XmNrightOffset, 6); argcnt++;
1223: XtSetArg(args[argcnt], XmNmaxLength, SEARCHMAX); argcnt++;
...
1351: XtSetArg(args[argcnt], XmNlabelString, st1=MKSTRING("Cancel")); argcnt++;
1352: XtSetArg(args[argcnt], XmNtopAttachment, XmATTACH_FORM); argcnt++;
1353: XtSetArg(args[argcnt], XmNbottomAttachment, XmATTACH_NONE); argcnt++;
1354: XtSetArg(args[argcnt], XmNleftAttachment, XmATTACH_NONE); argcnt++;
1355: XtSetArg(args[argcnt], XmNrightAttachment, XmATTACH_POSITION); argcnt++;

Table 4: Clone Class 1 – GUI Specific Clones

Text Editor	File	Clone Pairs	Function	Purpose
Emacs	xfns.c	421	x_window	Setup X Widgets
	lwlib-Xaw.c	2,320	make_dialog Xaw_create_scrollbar	Lucid Widget Library
	lwlib-Xm.c	6,419	make_dialog	
Nedit	search.c	5,380	CreateFindDlog CreateReplaceDlog CreateReplaceMultiFileDlog	Create Dialog Boxes
	smartIndent.c	677	EditCommonSmartIndentMacro EditSmartIndentMacros	
	userCmds.c	533	EditShellMenu EditMacroOrBGMenu	
	highlightData.c	496	EditHighlightPatterns EditHighlightStyles	
	shell.c	355	CreateOutputDialog	
	fontsel.c	1,628	FontSel	
	printUtils.c	768	CreateForm	
Vim	gui_motif.c	1,936	find_replace_dialog_create gui_mch_add_menu_item gui_mch_dialog gui_x11_create_widgets	Motif Support

A very important observation is that there is no cloning what so ever between *pico* and *nano* and between *vi* and *vim*. This was somewhat surprising because *nano* is a clone of *pico* and some implementation details would be expected to be cloned. Similarly, *vim* was developed as an improvement to *vi* but copied most of the behavior and features of *vim*.

Table 5: Example Clones of Class 2

Glimmer: mdi-routines.c

```
67:  gtk_signal_connect (GTK_OBJECT (new_file->text), "undo_changed",
68:                       GTK_SIGNAL_FUNC (undo_changed), new_file);
69:  gtk_signal_connect (GTK_OBJECT (new_file->text), "move_to_column",
70:                       GTK_SIGNAL_FUNC (file_pos_changed), 0);
71:  gtk_signal_connect (GTK_OBJECT (new_file->text), "focus_in_event",
72:                       GTK_SIGNAL_FUNC (editor_window_focused), new_file);
73:  gtk_widget_set_style (new_file->text, extext_style);
```

Gedit: gedit-mdi-child.c

```
322: g_signal_connect (G_OBJECT (child->document), "name_changed",
323:         G_CALLBACK (gedit_mdi_child_document_state_changed_handler),
324:         child);
325: g_signal_connect (G_OBJECT (child->document), "readonly_changed",
326:         G_CALLBACK (gedit_mdi_child_document_readonly_changed_handler),
327:         child);
328: g_signal_connect (G_OBJECT (child->document), "modified_changed",
329:         G_CALLBACK (gedit_mdi_child_document_state_changed_handler),
330:         child);
331: g_signal_connect (G_OBJECT (child->document), "can_undo",
```

Table 6: Clone Class 2 – Domain Specific Clones

Text Editor	File	Clone			
		Pairs	Groups	LOC	Purpose
Glimmer	mdi-routines.c	79	15	11	MDI Signaling
Gedit	Gedit-mdi-child.c	48	10	27	

Table 7: Clone Class 3 – Code Reuse

Text Editor	File	LOC	Clone			Purpose
			Pairs	Groups	LOC	
Glimmer	regex.c	4,938	54	20	846	Extended regular expression matching and search library
Emacs		6,055			864	

3.1.3 Summary

The presented cloning statistics and analysis indicates that almost all the clones in text editor domain are related to user interface, e.g. setting-up environments and widgets for creating dialog boxes. These clones are not due to intentional copying of code, but rather resulted accidentally while implementing similar functionality. The problem domain has very little contribution in cloning, but the front-end nature of the domain contributes most of the clones. We also observe a little reuse of code between the systems.

3.2 Window Managers for the X Window System

In this case study we analyze several window managers for the X Window System. The X Window System has been developed with the ideology of achieving functionality through the co-operation of several specialized programs [17] . This is a common paradigm in Unix systems. In X, most things the user interacts with are windows, and it is the responsibility of the window manager to manage them. In managing, they provide the "look and feel" of the windows, and provide a means by which the user may interact with them.

Each of these window managers are essentially fulfilling the same base requirements, but how they go about doing this can be quite different. Additionally, the range of features each window manager implements is quite broad. *WM2* is a very basic window manager that just simple manages the windows and nothing else, while FVWM2 is quite rich in features such as management icons and virtual desktops, and provideds 3D style menus and windows. We expected to find cloning at points of interaction with the X Window System and some simple window management tasks such as moving and resizing.

3.2.1 Cloning Statistics

In total, there were 1,031 files in this case study, 23 files had cloning in them, which is 2.2% of the total files. When considering the number of lines that are part of at least one clone, we see an even sparser picture of cloning. Of the 491,655 LOC in the study, only 1,402 LOC were involved in a clone, less than 0.3%. From this data alone it is pretty clear that in this case study a minimal amount of knowledge has been reused in the form

of code cloning across software systems. The total number of clones found is 1,716 pairs in 116 groups. After some manual filtering to remove obvious false positives the actual number of clone pairs came down to 858, grouped into 58 region groups. A large number of clones in a relatively small number of groups is generally a sign of a lot of overlapping clones. This was certainly true in this case study. The cause of the many overlapping clones was a series of code segments that were reasonably simple in structure that could be aligned as clones in many different ways.

Table 8 shows the amount of cloning occurring in each subject of the case study. Immediately, we can see that there is very little cloning that happens between the software systems. From this table we can see that a considerable amount of cloning seems to occur in a relatively few number of regions, as noted above. In this case study, some regions contained as many as 55 clone pairs covering as few 60 lines of code. When considering the data in

In Table 9 we can say that the number of actual clone pairs with a segment in a given system is closer to the number of clone groups. As a general rule, when analyzing cloning, it is important to consider both the number of clone pairs and the number of clones groups reported. In the next section we will see exactly what these clones were.

Table 8: Window Manager Guinea Pigs

Window Manager	**Release**	**Files**	**LOC**	**Lang**	**Description**
WindowMaker	0.91.0	258	139,964	C	GNUstep WM
BlackBox	0.70.0	73	24,268	C++	A light WM
Fvwm2	2.4.19	270	133,334	C	Feeble Virtual WM
Icewm	1.2.21pre	200	52,689	C++	A simple WM
Metacity	2.4.55	90	58,532	C	Gnome WM
Olvwm	4.5	93	56,218	C	Open Look Virtual WM
sawfish	1.3	32	21,860	C	An extensible WM
Wm2	4	15	4,790	C	A small WM
Total		**1,031**	**491,655**		

Table 9: Cloning Stats for the Window Managers

Window Manager	Files			LOC			Clone	
	Total	with Clones	%	Total	with Clones	%	Pairs	Groups
WindowMaker	258	6	2.32	139,964	573	0.410	312	31
BlackBox	73	0	0.00	24,268	0	0.000	0	0
Fvwm2	270	6	2.22	133,334	411	0.308	292	24
Icewm	200	3	1.50	52,689	33	0.063	276	18
Metacity	90	3	3.33	58,532	255	0.435	203	16
Olvwm	93	2	2.15	56,218	103	0.183	442	10
sawfish	32	2	6.25	21,860	18	0.082	97	9
Wm2	15	1	6.66	4,790	9	0.188	94	8
Total	**1,031**	**23**	**2.23**	**491,655**	**1,402**	**0.285**	**1,716**	**116**

3.2.2 Detailed Analysis

Looking closer at the clones we see several interesting things. The first was the cause for most of the clone pairs. During the setup phase of the window manager, identifiers for X properties need to be assigned or determined. These identifiers are called atoms and will be later used to set or adjust properties in the X Window System. Table 10 presents a typical instance of such code.

Table 10: Example Clones in Setup Phase

```
_XA_XdndPosition = XInternAtom(dpy, "XdndPosition", False);
_XA_XdndStatus = XInternAtom(dpy, "XdndStatus", False);
_XA_XdndActionCopy = XInternAtom(dpy, "XdndActionCopy", False);
_XA_XdndSelection = XInternAtom(dpy, "XdndSelection", False);
_XA_XdndFinished = XInternAtom(dpy, "XdndFinished", False);
```

The type of clone associated with this type of code in the window managers would be considered a template clone by Kim et al. [15] . In these types of clones, the structure of the code remains the same, but parameters such as parameter names tend to be changed. This type of clone is certainly an accidental clone as defined in this paper. These clones were in every window manager, occurring in nine different functions, all

related to the initialization of the window manager or in the case of two functions the initialization of test data. Of the 858 clone pairs found, 592 were part of these clones.

In this case study, we found several points of knowledge reuse through code cloning. The first was between *FVWM* and *WindowMaker*. In this case both *FVWM* and *WindowMaker* contained a copy of a file called *alloca.c*. The file is 511 and 498 lines of code in the two systems. There are minor changes between the two files and 298 lines of code were found to be clones. This is a public domain implementation of a memory allocation library designed to reclaim allocated memory after procedures exit. Another set of clones that appear to be copies between two systems are the flipping animation routines in *WindowMaker* and *FVWM* used for animating resizing and iconifying windows. These functions are nearly the same, with only minor changes.

Cloning happened between *Metacity* and *WindowMaker*. In this case, the file *gradient.c* in *Metaciy* is largely copied from the *wrlib/gradient.c* of *WindowMaker*. These files are 905 and 588 lines of code respectively. Only 189 lines were detected to be cloned from *wrlib/gradient.c*, but further inspection reveals that the cloning is more than that. The code is difficult to detect as a clone because it has been modified to fit into its new environment. Interestingly enough, this copying was documented as being such. In this example and the example of the animation clone above we see cloning related to the problem domain.

3.2.3 Summary

In this case study, we saw more examples of accidental cloning where code was very similar due to the nature of the task and library that had to be used. We also saw several interesting examples of knowledge reuse through code copying, but in the end we saw very little of it.

4 Discussion

In the above case studies we discovered an unexpected result. We found very little code cloning between software systems, even when they are highly related, as *nano* and *pico*, and *vim* and *vi* are. This result seems counter intuitive at first, but has several very likely reasons. First, developers would need to be familiar with existing code in order to know

where to clone from. The investment of time required to sufficiently gain this knowledge is likely too daunting to appear as any benefit to a developer. Second, the copied code may be difficult to detect. Even in the case of the copying of *gradient.c* the source code was modified significantly. In the author's own words, it was "gtk-enised".

In both case studies, we observed several accidental clones, code that is similar only because of the nature of the problem being solved, not because the code was copied and pasted. These clones tended to occur when dealing with APIs such as GUI toolkits and libraries. Such code is similar because of the constraints on the usage of the libraries and the protocol required when interacting with them. While this code is not technically a clone, it could be considered a point where abstractions could be made to simplify the use of the API. In the example of the window managers, perhaps a function could be created that took a list of strings and returned a list of atoms rather then using a separate call for each one. In general, if the same ordering of several function calls must be done to use a feature of an API, such as create a button or window, perhaps a more abstract function can be created to perform this common set of events.

From the lack of code cloning observed, we can see that in these case studies code cloning is likely not used as a form of knowledge reuse. But there is a possible weakness to this study and the conclusion must be considered with prudence. Because we are using code clone detection techniques to find code that may have been copied to reuse knowledge we will only find copies that have not been change substantially. This situation is certainly imaginable, as we have seen in the case of the cloning of the *gradiant.c* file. Keeping this in mind, parameterized matching clone detection tools, *CCFinder* in particular, can be reasonably resilient to change in code, as long as the overall structure is not changed too much. Because of this fact, we are reasonably confident that our results are accurate and will be true for most software systems.

The knowledge reuse through code cloning that we did find does appear to be related to problems in the domain of the software systems. This is a predictable result, and does provide some weight that OSS is a way to share knowledge. Perhaps as methods of documenting and analyze software systems improve and become more wide spread we will see more instances of knowledge reuse through code cloning.

The results of these studies can also be used as a form of validation for case studies on cloning within a software system. Given the results in this paper, and also those reported by Kamiya et al. [21] it is clear that cloning behavior within software systems is significant. It is typically reported that 10 – 15% of code is cloned within a software system. If these numbers were mostly attributed to noise or inaccuracy of the clone detection methods, then we would see similar numbers across software systems. The fact that we do not gives further validation that cloning within a software system is significant.

5 Related Work

Several case studies have been done investigating code duplication within a single software system [1] [8] [2] [3] [4] [13] [19] [21] [6] [7] . All of these studies have consistently shown that cloning within a software system is of non-trivial size. Typical values of code duplication are in the range of 10 – 15% of source code. Ducasse analyzed a system written in COBOL that had as much as 50% of duplication [19] .

While there have been several studies investigating the types of cloning that occurs within a software system, very little work has addressed the issue of copying code across software systems as a software development strategy. Kamiya et al. [21] presents an analysis of cloning across the source code of three different operating systems: Linux, FreeBSD and NetBSD. Their analysis showed that there is about 20% cloning between FreeBSD and NetBSD, whereas this amount is less than 1% between Linux and FreeBSD or NetBSD. FreeBSD and NetBSD originate from the same BSD OS and hence they share a certain amount of common code. Linux, on the other hand, originated and grew independently. Our study supports the finding that separate projects have very little code cloning between them. Most of the clones that were found in the text editor and window manager case studies were accidental clones, clones that appear not through explicit copy and paste, but as a side effect of using similar libraries and interfaces.

6 Conclusion

One motivation behind open source software is the sharing of knowledge. In this paper we investigate how knowledge reuse might be done through code cloning. Using clone detection methods, we performed case studies in two software domains in search of code reuse. Our results were somewhat surprising. We expected to see cloning between related software systems, but we found very little; most of the cloning we did find was accidental clones rather than real reuse of code.

This provides two insights into the nature of software and code duplication. First, it appears that code reuse through code duplication is significantly rare, likely because people clone from code they know about, and it is likely most developers are not familiar with the source code of other projects. Second, given the stark differences between the degree of duplicated code found within a software system and the degree of duplicated code across software systems, we can conclude that code duplication within a software system is in fact a significant phenomenon.

We found cloning that is related to knowledge reuse does have some relation to the problem domain, for example the animation of resizing windows. However, we found very few examples of this type of code reuse. We hypothesize that one major reason for this is the difficulty of knowing where to copy code from. In order for OSS software to achieve the goal of sharing knowledge, it is important that this problem be addressed. Key issues associated with this problem are efficient techniques for reverse engineering and software comprehension, both important in locating and understanding code that can be reused in a new project.

Future work in this area includes a more complete investigation of duplication and knowledge reuse in the open source community. A larger study needs to be done to show how applicable these results are across all software domains. Additionally, code reuse is only one form of knowledge reuse. We intend to investigate what other kinds of expertise sharing exist, and how they affect open source development.

References

[1] B. S. Baker. “On Finding Duplication and Near-Duplication in Large Software Systems”. *Proceedings of the Working Conference on Reverse Engineering*, 1995.

[2] C. Kapser and M. W. Godfrey. “Aiding Comprehension of Cloning Through Categorization”. *Proceedings of the International Workshop on Principles of Software Evolution*, 2004.

[3] C. Kapser and M. W. Godfrey. “Improved Tool Support for the Investigation of Duplication in Software”. Submitted for review, 2005.

[4] C. Kapser and M. W. Godfrey. “Toward a Taxonomy of Clones in Source Code: A Case Study”. *Proceedings of the International Workshop on Evolution of Large Scale Industrial Software Applications*, 2003.

[5] F. Van Rysselbergh and S. Demeye. “Evaluating Clone Detection Techniques”. *Proceedings of the International Workshop on Evolution of Large Scale Industrial Software Applications*, 2003.

[6] G. Antoniol, U. Villano, E.Merlo and M. D. Penta. “Analyzing Cloning Evolution in the Linux Kernel”. *Information and Software Technology 44(13)*, 2002.

[7] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. “Identifying Clones in the Linux Kernel. *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, 2001.

[8] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier. “Clone Detection Using Abstract Syntax Trees”. *Proceedings of the International Conference on Software Maintenance*, 1998.

[9] J. H. Johnson. “Substring Matching for Clone Detection and Change Tracking”. *Proceedings of the International Conference on Software Maintenance*, 1994.

[10] J. Krinke. “Identifying Similar Code with Program Dependence Graphs”. *Proceedings of the Working Conference on Reverse Engineering*, 2001.

[11] J. Mayrand, C. Leblanc, and E. Merlo. “Experiment on the Automatic Detection of Function Clones in a Software System using metrics”. *Proceedings of the International Conference on Software Maintenance*, 1996.

[12] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, Kouichi Kishida, and Y. Ye. "Evolution Patterns of Open-Source Software Systems and Communities". *Proceedings of the International Workshop on Principles of Software Evolution*, 2002.

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

[14] M. Bruntink, A. van Deursen and T. Tourwe. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. Proceedings of the International Conference on Software Maintenance, 2004.

[15] M. Kim, L. Bergman, T. Lau, and D. Notkin. "An Ethnographic Study of Copy and Paste Programming Practices in OOPL. *Proceedings of the International Symposium of Empirical Software Engineering*, 2004.

[16] M. W. Godfrey and Q. Tu. "Evolution in Open Source Software: A Case Study". Proceedings of the International Conference on Software Maintenance, 2000

[17] Matt Chapman. "Window Managers for X". http://xwinman.org. Last visited May 2005.

[18] R. Komondoor and S. Horwitz. "Using Slicing to Identify Duplication in Source Code". *Lecture Notes in Computer Science*, 2126:40–50, 2001.

[19] S. Ducasse, M. Rieger and S. Demeyer. "A Language Independent Approach for Detecting Duplicated Code". *Proceedings of the International Conference on Software Maintenance*, 1999.

[20] Stefan Bellon. "Vergleich von Techniken zur Erkennung duplizierten Quellcodes". Diplomarbeit, Universität Stuttgart, September 2002.

[21] T. Kamiya, S. Kusumoto and K. Inoue. "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code". *IEEE Transactions on Software Engineering*, July 2002.