




May 30th, 8:50 AM

A Proposal for Incorporating Programming Blunder as Important Evidence in Abstraction-Filtration-Comparison Test

P. Vinod Bhattathiripad

Cyber Forensic Consultant, Polpaya Mana, Thiruthiyad, vinodpolpaya@gmail.com

Follow this and additional works at: <https://commons.erau.edu/adfsl>

 Part of the [Computer Engineering Commons](#), [Computer Law Commons](#), [Electrical and Computer Engineering Commons](#), [Forensic Science and Technology Commons](#), and the [Information Security Commons](#)

Scholarly Commons Citation

Bhattathiripad, P. Vinod, "A Proposal for Incorporating Programming Blunder as Important Evidence in Abstraction-Filtration-Comparison Test" (2012). *Annual ADFSL Conference on Digital Forensics, Security and Law*. 5.

<https://commons.erau.edu/adfsl/2012/wednesday/5>

This Peer Reviewed Paper is brought to you for free and open access by the Conferences at Scholarly Commons. It has been accepted for inclusion in Annual ADFSL Conference on Digital Forensics, Security and Law by an authorized administrator of Scholarly Commons. For more information, please contact commons@erau.edu.

EMBRY-RIDDLE
Aeronautical University™
SCHOLARLY COMMONS

(c)ADFSL



A PROPOSAL FOR INCORPORATING PROGRAMMING BLUNDER AS IMPORTANT EVIDENCE IN ABSTRACTION-FILTRATION-COMPARISON TEST¹

P. Vinod Bhattathiripad
Cyber Forensic Consultant
Polpaya Mana, Thiruthiyad
Calicut-673004
Kerala, India

Telephone: +91-495-2720522, +91-94470-60066 (m)
E-mail: vinodpolpaya@gmail.com; vinodpolpaya@yahoo.co.in

ABSTRACT

This paper investigates an unexplored concept in Cyber Forensics, namely, a Programming Blunder. Programming Blunder is identified as a variable or a code segment or a field in a database table, which is hardly used or executed in the context of the application or the user's functionality. Blunder genes can be found in many parts of any program. It is the contention of this paper that this phenomenon of blunders needs to be studied systematically from its very genetic origins to their surface realizations in contrast to bugs and flaws, especially in view of their importance in software copyright infringement forensics. Some suggestions as to their applicability and functional importance for cyber forensics are also given including the vital need and a way to incorporate programming blunders into Abstraction-Filtration-Comparison test, the official software copyright infringement investigation procedure of US judiciary

Keywords: Bug, error, blunder, genes, software piracy, software copyright, software copyright infringement, software piracy forensics, AFC, idea-expression dichotomy

1. INTRODUCTION

A programming flaw occasionally survives in well tested and implemented software. It can surface in the form of a variable or a code segment or a field in a database table, which is hardly used or executed in the context of the application or the user's functionality. Such a flaw in design can be called a Programming Blunder² (Bhattathiripad and Baboo, 2009, 2011). The term programming blunder has already been casually used (in many publications, for instance, in (McConnell, S., 1996)) to denote bad practices in programming.

The phenomenon of blunder needs to be specifically contrasted with a programming error, as unlike an error, the blunder is most unlikely to cause problems during the execution. Ideally, all blunders (like all errors) in software should be and are routinely removed at the various quality control stages of the software development. Even if it (unfortunately) makes through all quality control stages, there is again a slim chance for it be detected and removed at the implementation stage. Even so, occasionally, a few programming blunders may survive all these stages of software development and may finally appear unattended (or unnoticed) in the implemented software. Despite their apparent status as

¹ This paper is an enhanced form of the paper "**Software Piracy Forensics: Programming Blunder as an important evidence**" that was accepted as a short paper (but not presented) in the Third ICST International Conference on Digital Forensics and Cyber Crime, Dublin, Ireland, 26 - 28 October, 2011. Also, this paper contains points extracted from the author's Ph D thesis "Judiciary-friendly software piracy forensics with POSAR".

² This type of programming flaw has been christened as "Programming Blunder" because the very existence of it (in a well-tested and implemented program) is a mark of blunder-like weaknesses of the respective programmer / quality engineer. The naming is done from the forensic perspective of such programming flaws.

harmless vestiges of inattentive vetting, these blunders do provide an important service to the cyber forensic expert. They can form an important basis for providing evidence in case of an allegation and ensuing investigation of copyright infringement of such software. It is this increased cyber forensic importance (despite their being less important in areas such as software engineering and software testing) that underscore the need to specifically understand and study them, not just in the cyber forensic perspective but right from their definitional aspects.

2. OBJECTIVES OF THIS PAPER

Spafford and Weeber (1992) have already anticipated the importance of (blunder-like) execution paths as cyber forensic evidence in “providing clues to the author (of the suspect / alleged program)” and this anticipation is the point of departure for of this study. The emergent concept of programming blunders (in this paper) is a natural outcome of a specific study of all such execution paths in the context of software piracy³. The objectives of this paper can be set thus: (1) to thoroughly investigate the phenomenon of blunders in detail and by doing so attempt to denotatively concretize and define the term “programming blunder”; (2) to discretely identify the study of programming blunders as different from other software bugs and (3) to discuss the cyber forensic importance of programming blunders in the investigation of an allegedly pirated (copyright infringed) software.

3. DEFINING THE TERM PROGRAMMING BLUNDER

The term programming blunder has already been introduced and identified (but not properly defined) in some previous publications (Bhattathiripad and Baboo, 2009; 2011). Additionally, without using the term “Programming Blunder”, Spafford and Weeber (1992) have already mentioned certain execution paths (as said above) of a code that “cannot be executed”.

A common factor found when analyzing student programs and also when analyzing some malicious code is the presence of code that cannot be executed. The code is present either as a feature that was never fully enabled, or is present as code that was present for debugging and not removed. This is different from code that is present but not executed because of an error in a logic condition—it is code that is fully functional, but never referenced by any execution path. The manner in which it is elided leaves the code intact, and may provide some clue to the manner in which the program was developed. (Spafford and Weeber, 1992)

By taking a cue from Spafford and Weeber, can one define programming blunder as “any execution path in the program that need not and so will not be executed during the lifetime of the program on any execution platform”? Possibly not, because, such a definition has an inherent limitation in that it considers only inoperative statements (non-executed path) in the program. It overlooks and excludes those operative statements (executed paths) which are very much still present there but are not necessary for the successful functioning of the program. That means, it excludes those statements which may have been executed at some stage of the program but are not necessary for producing the final result. In other words, it does not consider those operative statements which are incoherently, redundantly and/or dysfunctionally appearing in the text of the program and/or which may have been executed at some stage but are hardly used in the user’s functionality (or to arrive at the final results). So, programming blunders turn out to be a lot more than what Spafford and Weeber had suggested.

Like Spafford and Weeber (1992), several other researchers also have already mentioned programming flaws of this genre (without using the term programming blunder) and studied their importance in software testing, author identification and other software forensic areas. For instance,

³ In this article, the term ‘piracy’ refers to the piracy of a copyrighted software.

through a recent comprehensive paper⁴ (Hayes and Offutt, 2010), Jane Huffman Hayes and Jeff Offutt examine (among other things) whether *lint*⁵ (a static analyzer that detects poor programming practices such as variables that are defined but not used) for a program can be used to identify the author or a small set of potential authors of the program. So, the notion of a programming blunder may not entirely be new. Nevertheless, none of the previous publications (where the concept of programming blunder was used in the research related to software testing, author identification and other software forensic areas) have tried to seriously explore the concept in some detail in an effort to denotationally concretize / crystallize the term programming blunder, and so differentiate it from other programming bugs and finally, study its forensic importance. This is the reason for setting the primary objective of this paper, viz. to thoroughly investigate the phenomenon of blunders in detail and by doing so attempt to concretize and define the term “programming blunder”.

Even though the existing definitions of “programming blunders” subsume execution paths of a code that “cannot be executed” (Spafford and Weeber, 1992) and variables that are defined but not used (Hayes and Offutt, 2010), a more cautious definition employed in this study is:

A programming blunder found (in well tested, implemented and allegedly pirated software) can be defined as a variable in a program or a program code segment or a field in a database table which is hardly executed in the context of the application and/or is unnecessary for the user’s functionality (Bhattathiripad and Baboo, 2009).

This definition subsumes not only the execution paths of a code that “cannot be executed” and variables that are defined but not used but also unnecessary non-execution paths (like comment lines and blocked execution paths).

A blunder in a program gains significance during the piracy forensics (or copyright infringement forensics) of the program (see below).

4. GENETIC ORIGIN OF PROGRAMMING BLUNDERS

A proper investigation of blunder, like that of any organism, should ideally start with a look into its genetic origin. Blunder genes⁶ (or genes of programming blunders) are those elements in the program that can often form the basis for (the existence or surfacing of) a programming blunder. Blunder genes can be traceable to many parts of the program like a variable, class, object, procedure, function, or field (in a database table structure). A blunder gene is developmentally (and perhaps philologically) different from a blunder just as an embryo can be from the baby. While every blunder gene has significance in software engineering research, a blunder has additional forensic significance. What the programmer leaves in the program is a blunder gene and this blunder gene can develop into and surface as a blunder during the piracy forensic analysis (or copyright infringement forensic analysis).

What elements in the program can then form the genetic basis of a blunder? The simple answer is that any item in (or a segment of) a program which is unnecessary or redundant to customer requirements can form the genetic basis for a programming blunder. Such items can, however, surface in a program in three different ways. In other words, programming blunders can be categorized in three ways according to their genetic differences.

1. Any defined but-unused item (or a code segment) in a program.

⁴ A note of gratitude to the reviewers of the Third ICST International Conference on Digital Forensics and Cyber Crime, for drawing my attention to this paper.

⁵ The UNIX utility *lint* that is commonly used to bring out flaws like programming blunders as compiler warnings.

⁶ My sincere gratitude to Dr. P. B. Nayar, Lincoln University, UK, for his valuable suggestions

2. Any defined item (in the program) which is further used for data-entry and calculation but never used for the user's functionality of a program.
3. Any blocked operative statement in a program.

Primarily, any defined but unused variable, class, object, procedure, function, or field (in a database table structure) can appear as a programming blunder. Hence, any such defined, concrete, tangible item (or a blunder gene) in the body of a program (or an external routine or data base as part of the software) which was subsequently found unnecessary or irrelevant for the operation of the program / software can evolve or materialize as a programming blunder during a forensic analysis of the program. Thus, a programming blunder may be an item (or a segment of a program) that is well defined at the beginning of an execution path in a program but is not part of the remaining stages of the execution path (example: Processing stages, Reporting stages etc.) in the program. For instance, the integer variable 'a' in the C-program given in Table-1 is a programming blunder as this variable has not been used anywhere else in the program. This variable has no relevance in the operation (for producing the intended output) of the program.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int a=0, b=2, c=0;
    C=b*b;
    printf("The result is %d", c);
    getch()
}
```

Table 1. A defined but unused variable 'a' in a C-program

Secondly, any defined item (or the blunder gene) at the beginning of an execution path in a program which is further used for data-entry but never used in the remaining stages of the execution path in the program, can also appear as a programming blunder during a forensic analysis of the program. Thus, the integer variable 'a' in the C-program given in table 2 surfaces as a programming blunder as this variable has been well defined and used for data entry but not used anywhere else in the program.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int a=0;
    scanf("%d", &a); /* reading the value of a*/
    printf("Hello, World");
    getch();
}
```

Table 2. Unnecessary declaration and input statements in a C-program

Thirdly, a blocked operative statement (or a remarked operative programming statement), which is practically an inoperative element of the program, can appear as a programming blunder. Thus, the remark statement (or the blunder gene) `/* int a; */` in the C-program given in table 3 can turn out to be a programming blunder during a forensic analysis of the program as this statement need not be there in the first place and does not justify its being there at all for long, unattended (unlike the other programming remark `/* This program prints Hello, World */` which has a purpose in the program).

```
#include <stdio.h>
#include <conio.h>
main()
{
  /* This program prints "Hello, World" */
  /*int a=0;*/
  printf("Hello, World");
  getch();
}
```

Table 3. A program in C-language

All the above suggest that, any defined variable, class, object, procedure, function, or field (in a database table structure) in a program which has no relevance in the final output (user's functionality) of the (well-tested and implemented) program can manifest itself as a blunder during the copyright infringement forensic analysis of the program.

5. COMMONALITIES AND SIMILARITIES AMONG PROGRAMMING BLUNDERS

Irrespective of their genetic origin, all programming blunders do share some features, properties, attributes and characteristics. A programming blunder

- 1) is a concrete, tangible item in (or segment of) a program and not a process.
- 2) can be an execution path which got past the quality control stage, undetected.
- 3) does not affect the syntax or sometimes even the semantics of a program which makes it hard to detect.
- 4) is not necessary to execute the program.
- 5) is not necessary for user's functionality.
- 6) does not justify its being there at all.
- 7) is a matter related to the design pattern and programming pattern of the software.

6. ETIOLOGY OF PROGRAMMING BLUNDERS

The etiology of programming blunders can be discussed along three different weaknesses of the programmer / quality engineer. Firstly, his/her inability to completely remove from the program those elements that do not help meet customer requirements can be a cause for a blunder. Secondly, his/her inattention to completely remove those statements that have been abandoned as a result of the programmer's afterthought can also be a cause for a blunder. Thirdly, his/her inattention to identify and remove items that do not contribute to either strong coupling between modules or strong cohesion within any module of the program can also be a cause for a blunder. These three different weaknesses of the programmer / quality engineer can thus be reasons for programming blunders.

7. PROGRAMMING BLUNDERS JUXTAPOSED WITH SOFTWARE BUGS

The next objective of this paper is to identify the study of programming blunders as different and discrete from that of other software bugs. Quite naturally, even experts in the software engineering might need some convincing as to why programming blunders need or demand a distinct status as against software bugs. There definitely does exist a need to be convincing because, the above mentioned genetic origins, manifestations, features, properties, attributes, characteristics and etiology of blunders may prima facie be identified with those of software bugs as well. Therefore, what makes a programming blunder deserve a special consideration different from other software bugs?

A *software bug* is the common term used to describe an error, flaw, mistake, failure, or fault in a

computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways (IEEE610.12-1990, 1990). An *error* is “the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition” (IEEE610.12-1990, 1990, p31). Other related terms in the computer science context are fault, failure and mistake. A *fault* is “an incorrect step, process, or data definition in a computer program” (IEEE610.12-1990, 1990, p31). A *failure* is “the [incorrect] result of a fault” (IEEE610.12-1990, 1990, p31) and *mistake* is “a human action that produces an incorrect result” (IEEE610.12-1990, 1990, p31). Most bugs arise from mistakes, errors, or faults made by people or flaws and failures in either a program's source code or its design, and a few are caused by compilers producing incorrect code⁷. A **programming blunder** (as defined at the beginning of the article) does not resemble a bug either in *raison d'être* or function (see above). In other words, a software bug is different from a programming blunder and this difference (which is significantly relevant for the forensic expert) may look simple, but is by no means simplistic.

8. PROGRAMMING BLUNDERS AND THE IDEA-EXPRESSION DICHOTOMY

The idea-expression dichotomy (Newman, 1999) provides an excellent theoretical perspective to look at and explain blunders. Any genuine idea which is properly expressed in the definition stage but improperly (or not at all) expressed in the remaining stages (in a program) in a manner that does not adversely affect the syntax (or sometimes even the semantics) of the program can become a programming blunder. So, the integer variable ‘a’ in the C-program given in Table-2, for example, when looked at in the idea-expression perspective, is a programming blunder. So, from the perspective of the idea-expression dichotomy, programming blunder is a partly-made⁸ functional expression of an idea. This clearly opens the door to linking blunders directly to copyright infringements of any program because the idea-expression perspective is the basis of formulation of software copyright laws in several countries (Hollaar, 2002; Newman, 1999).

Copyright laws of several countries (especially the US copyright laws) say that if there is only one (exclusive) way of effectively expressing an idea, this idea and its expression tend to “merge” (Walker, 1996, p83) and in such instances an idea is not protectable through copyright (Hollaar, 2002). However, if the same idea can be realized through more than one expression, all such different realizations are protected by copyright laws. Interestingly this means that the copyright of a program is directly related to the concept of the merger between idea and expression and that when there is no merger, the copyright of a program can extend to the blunders contained therein as well.

9. FORENSIC IMPORTANCE OF PROGRAMMING BLUNDERS

Despite their apparent functionally inactive and thus innocuous nature in a program, blunders, when copyrighted, can be of great value / assistance to the cyber forensic expert. They provide evidence of software copyright infringement and a discussion of this evidence is one of the prime objectives of this article. On the forensic importance of programming blunders, Spafford and Weeber (1992) have noted that:

Furthermore, it (a programming blunder) may contain references to variables and code that was not included in working parts of the final program — possibly providing clues to the author and to other sources of code used in this program.

⁷ http://en.wikipedia.org/wiki/Software_bug visited on 6th Feb, 2011

⁸By partly-made functional expression, what is meant or intended is an element which is defined, implemented but left unused or inoperative in the remaining stages.

Since Spafford and Weeber (1992), a variety of experiments (for instance, Krsul (1994)) have been performed on authorship analysis of source codes and copyright infringement establishment of software. Also, at least half a dozen techniques and procedures (for instance, AFC (Hollaar, 2002), SCAP (Frantzeskou, 2007) etc.) have been put forward for establishing authorship and copyright infringement of software. However, none of them have taken cue from the above note of Spafford and Weeber and considered programming blunders as evidence in the court.

Not all blunders are “substantial” in the copyright infringement forensic analysis. Blunders which can provide direct (substantial) evidence to establish piracy (and thus, to establish copyright infringement) or can provide probable, corroborative or supporting evidence are forensically important. The repetition of programming blunders (even if these programming blunders are ‘generic’ by nature) in both original⁹ and the pirated¹⁰ in identical contexts would be a serious indication of piracy. If, for instance, a variable with a universally uncommon name ‘PXRN_CODE_AUCQ CHAR[6]’ is defined in identical places of the identical procedures in the original¹¹ as well as the pirated¹² software, but not used anywhere else (see the three categories of blunders, above), that is an instance of programming blunder and such programming blunders attract/deserve forensic importance. The forensic importance of blunders arises from the obvious fact that it is highly unlikely that two programmers will blunder exactly in the same way, thus elevating the similarity into possible evidence of piracy (See also Appendix-1).

While trying to establish a variable name or a code segment as a programming blunder, the expert needs to confirm that it is (i) absent elsewhere in the original, (ii) present in the allegedly pirated exactly in the same context as it was found in the original and (iii) absent elsewhere in the allegedly pirated (Bhattathiripad & Baboo, 2010).

In the absence of other direct evidence of piracy, programming blunders can form the only basis for the expert to convince the judiciary about the possibility of piracy.

⁹ Throughout this article, original means the version of the software that the complainant submits to the law enforcement agency for software piracy forensics. This article presupposes that the law enforcement agency has satisfactorily verified the legal aspects of the documentary evidence of copyright produced by the complainant and is convinced that the complainant is the copyright holder of this version of the alleged software.

¹⁰ Throughout this article, pirated means the allegedly pirated software

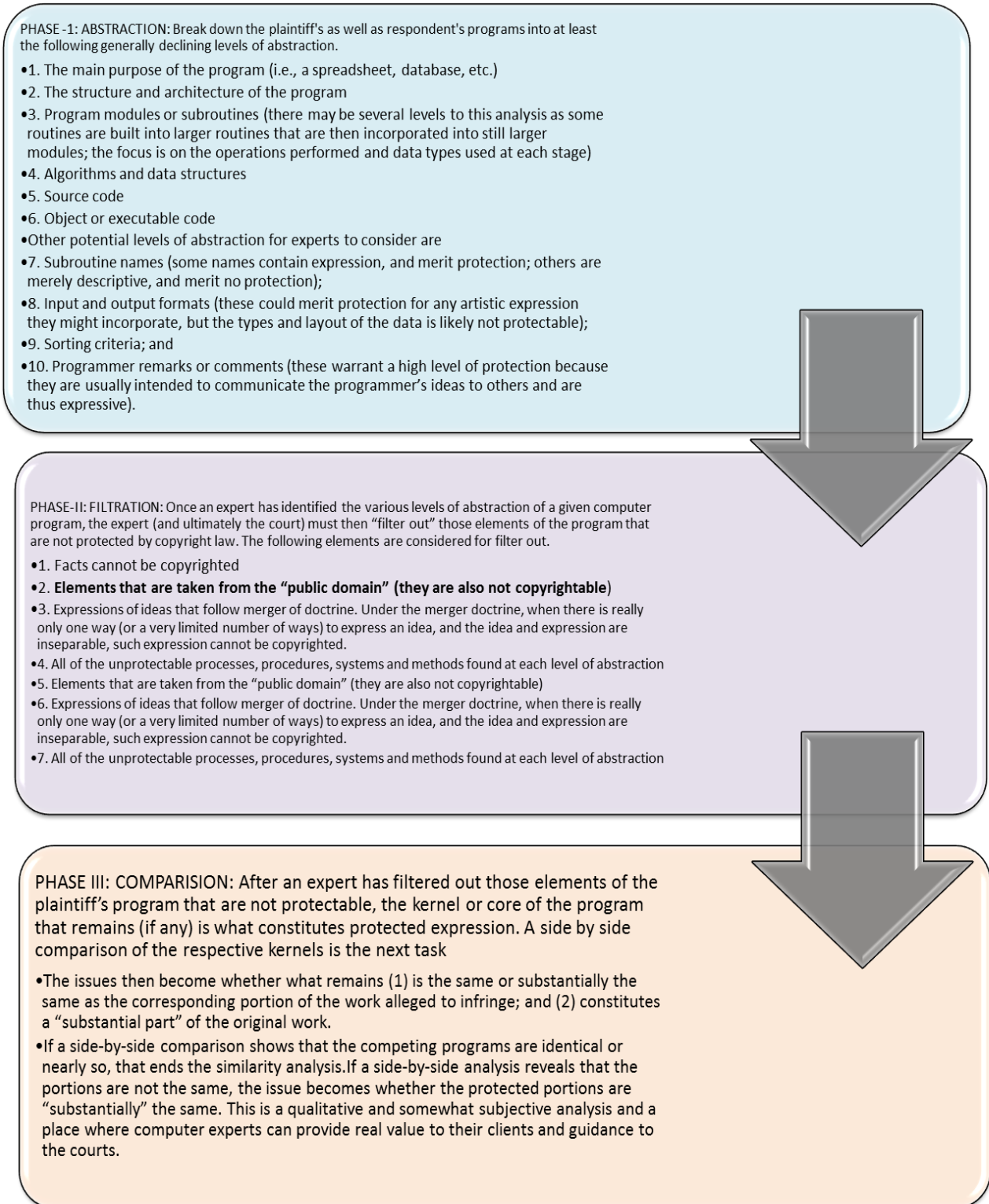


Fig 1: The software copyright infringement forensic procedure of AFC¹³

¹³ Abstraction-Filtration-Comparison Analysis Guidelines for Expert Witnesses
<http://web20.nixonpeabody.com/np20/np20wiki/Wiki%20Pages/Abstraction-Filtration-Comparison%20Analysis%20Guidelines%20for%20Expert%20Witnesses.aspx>

10. PROGRAMMING BLUNDERS AND THE AFC TEST

As things stand, it appears that forensic procedural protocols of software copyright infringement have not fully recognized the importance of programming blunders. Nor have there been many attempts to ensure their inclusion in the forensic expert's repertoire. Programming blunders are very unlikely to be detected, for instance, by the AFC test (see Appendix-3 and fig 1) which is the only judiciary-accepted procedure for establishing software copyright infringement in the US. They are not detected or even considered because during the abstraction of the program, only the functionally active parts (of the program) will be considered for abstraction and used for further investigation (Hollaar, 2002, p86). The functionally inactive parts (or those items are irrelevant for user's functionality) like programming blunders will not be considered for abstraction¹⁴. Moreover, abstraction may remove many individual lines of code from consideration and so blunder genes as well as blunders may also get automatically removed during the abstraction stage of AFC. In such case of unfortunate non-consideration, the programming blunders will not be available for final comparison and this unavailability may make the final results of AFC, incomplete and thus, unreliable. This is one of the fallibilities of AFC (Bhattathiripad and Baboo, 2010) and can probably be a reason for the reluctance to use the AFC test in the copyright infringement forensics of modern software (however, US judiciary continues to use AFC test for software copyright infringement forensics¹⁵). Hence, this paper proposes that, along with the AFC results, the evidence concerning the programming blunders, if any, should also be identified and gathered separately by the expert by comparing the original with the pirated, before applying the AFC test and reporting his/her the final findings and inferences to the court¹⁶. This paper also proposes that the AFC test should be re-designed so as to ensure that possible evidence like programming blunders are available for final comparison.

Before concluding, a note on what a judge expects from an AFC expert, would add value to the special attention and consideration given to programming blunders. In any software comparison report, what the judge would expect from the AFC expert is a set of details that help the court in arriving at a decision on the copyrightable aspects of programming blunders. Jon O. Newman (1999), one of the judges on the panel (in the 2nd circuit of U. S. judiciary) that received an amicus brief¹⁷ (concerning a software copyright infringement case) indicates what he needs from expert witnesses in a trial (or in the amicus brief) on an appeal involving software copyright infringement:

These professionals would be well advised not to tell me simply that the source code is or is not protectable expression. Their opinion are relevant, but, as with all opinions, what renders them persuasive is not the vehemence of their assertion and not even the credentials of those asserting them; it is the cogency and persuasive force of the reasons they give for their respective positions. These reasons had better relate to the specifics of the computer field. For example, as *Altai* (United States Court of Appeals, 1992) indicates, even with its overly structured mode of analysis, it will be very

¹⁴ Because AFC test does allow for 'literal' similarities between two pieces of code, there is a general belief that AFC can make available "literal" pieces like programming blunders (for example, variables that are defined but unused and execution paths that "cannot be executed") for final comparison. But in practice, AFC does not either abstract these "literal" pieces of evidence or filter them out in the filtration stage and either way, these programming blunders like "literal" pieces are unavailable for final comparison.

¹⁵ See United States District Court of Massachusetts (2010), Memorandum and Order, Civil Action number 07-12157 PBS, Real View LLC. Vs. 20-20 Technologies, p.2

¹⁶ The ultimate decisions like whether these pieces of evidence (concerning programming blunders) (1) are useful or not; (2) form direct or secondary evidence; and (3) are generic (and hence, non-copyrightable) by nature or not, come under judicial matters and are subject to the provisions of the prevailing Evidence Act of the respective country. However, cyber forensic report can remain suggestive on these three aspects and also on the (conclusive, corroborative, and supportive) nature of the programming blunders found in the original and the pirated.

¹⁷ Amicus Brief of Computer Scientists, *Harbor Solutions v. Applied Systems* No. 97-7197 (2nd Circuit, 1998) at 8-9 (citations omitted)

important for me to know whether the essential function being performed by the copyrighted program is a function that can be accomplished with a variety of source codes, which will strengthen the case for protection, or, on the other hand, is a function, capable of execution with very few variations in source code, or, variations of such triviality as to be disregarded, in which event protection will be unlikely. For me, this mode of analysis is essentially what in other contexts we call the merger doctrine – the expression is said to have merged with the idea because the idea can be expressed in such limited ways that protection of the plaintiff's expression unduly risks protecting the idea itself. (Newman, 1999)

So, what is expected in the case of programming blunders also is a set of details on the merger aspects of the ideas and expressions contained in any programming blunder (This is because, as said earlier,).

In conclusion, the following facts emerge: it seems reasonable to state that any variable or a code segment or a field in a database table, which is hardly used or executed in the context of the application or the user's functionality can form the basis of a programming blunder. The copyright of the software can often extend to the blunders contained therein. Also, any programming blunder that goes against the merger doctrine is copyrightable and repetition of it in another program can be a serious indication of copyright infringement. As a result, programming blunders can greatly influence the expert to give supportive evidence to his findings, thus indirectly contributing to the judge's decision. Because of this, identification and reporting of programming blunders need to be a part of the copyright infringement investigation. Hence, procedures like the AFC test needs to be re-designed so as to ensure that possible evidence like programming blunders are not filtered out.

11. REFERENCES

Bhattathiripad., P. V., and Baboo (2009), S. S., Software Piracy Forensics: Exploiting Nonautomated and Judiciary-Friendly Technique', *Journal of Digital Forensic Practice*, 2:4, pages 175 — 182

Bhattathiripad., P. V., and Baboo (2011), S. S., Software Piracy Forensics: Impact and Implications of post-piracy modifications, *Conference on Digital Forensics, Security & Law*, Richmond, USA

Bhattathiripad., P. V., and Baboo, S. S. (2010), Software Piracy Forensics: The need for further developing AFC, 2nd International ICST Conference on Digital Forensics & Cyber Crime, 4-6 October 2010, Abu Dhabi, UAE

European Software Analysis Laboratory (2007), The "SIMILE Workshop": Automating the detection of counterfeit software, available at www.esalab.com

Frantzeskou, G., Stamatatos, E., Gritzalis, S., Chaski, C. E., and Howald, B. S. (2007), Identifying Authorship by Byte-Level N-Grams: The Source Code Author Profile (SCAP) Method, *International Journal of Digital Evidence*, 6, 1

Hayes, J. H., and Offutt, J. (2010), Recognizing authors: an examination of the consistent programmer hypothesis, *Software Testing, Verification & Reliability - STVR* , vol. 20, no. 4, pp. 329-356

Hollaar L. A. (2002), *Legal Protection Of Digital Information*, BNA Books

IEEE610.12-1990 definition (1990), *IEEE Standard Glossary of Software Engineering*

Krsul, I., (1994), Identifying the author of a program, Technical Report, CSD-TR-94-030, Purdue University, available at <http://isis.poly.edu/kulesh/forensics/docs/krsul96authorship.pdf> , visited on 14th April, 2011

McConnell, S. (1996), Who cares about software construction?, *Software*, IEEE, Volume 13, Issue 1, Jan 1996, p.127-128

Newman J. O. (1999), New Lyrics For An Old Melody, The Idea/Expression Dichotomy In The Computer Age, 17, *Cardozo Arts & Ent. Law Journal*, p.691

Raysman R. and Brown P. (2006), Copyright Infringement of computer software and the Altai test, New York Law Journal, Volume 235, No. 89

Spafford, E. H., and Weeber, S. A. (1992), Software forensics: Can we track the code back to its authors?, Purdue Technical Report CSD-TR 92-010, SERC Technical Report SERC-TR 110-P, Department of Computer Sciences, Purdue University

United States Court of Appeals (1992), Second Circuit, Computer Associates International, Inc. v. Altai, Inc., 982 F.2d 693; 1992 U.S. App. LEXIS 33369; 119 A.L.R. Fed. 741; 92 Cal. Daily, Op. Service 10213, January 9, 1992, Argued, December 17, 1992, Filed

United States District Court of Massachusetts (2010), Memorandum and Order, Civil Action number 07-12157 PBS, Real View LLC. Vs. 20-20 Technologies, p.2

Walker, J. (1996), "Protectable 'Nuggets': Drawing the Line Between Idea and Expression in computer Program Copyright Protection", 44, Journal of the Copyright Society of USA, Vol 44, Issue 79

APPENDIX-1: SECONDARY OR INCONCLUSIVE PROGRAMMING BLUNDER GENES

Most genes of programming blunders can be conclusively identified. But, there are certain elements in a program that may not have yet surfaced as blunders but are potentially prone to surfacing later as programming blunders. Conversely, all ideas which are successfully expressed but are superfluous to customer requirements also may not surface as a blunder because such expressions (or a code segment) may affect the semantics of the program and thus end up as an error at some point of time during the life time of the program (if not during its pre-implementation testing stage). Any such code segment is basically a gene of an error and not of a blunder. However, any such code segment in *a time-tested program* may eventually form the basis of a blunder because such an item or a code segment does not justify its being there at all *for long* unattended. But, such blunders are very difficult to be conclusively identified and used.

APPENDIX-2: BUG REPEATED V. BLUNDER REPEATED

A bug repeated (in a pirated program) may be noticed during the functioning of the pirated. Unlike a bug, a programming blunder repeated (in a pirated program, though noticeable) would remain unnoticed during the functioning of the pirated. In the absence of a thorough quality control round by the pirated (which is very likely), these programming blunders would remain intact in the pirated and may turn into evidence of piracy.

APPENDIX-3: A NOTE ON ABSTRACTION-FILTRATION-COMPARISON TEST

AFC (Abstraction-Filtration-Comparison) test was primarily developed by Randall Davis of the Massachusetts Institute of Technology for evaluating copyright infringement claims involving computer software and used in the 1992 *Computer Associates vs. Altai* case, in the court of appeal of the 2nd federal circuit in the United States (Walker, 1996). Since 1992, AFC has been recognized as a legal precedent for evaluating copyright infringement claims involving computer software in several appeal courts in the United States, including the fourth, tenth, eleventh and federal circuit courts of appeals (European Software Analysis Laboratory, 2007; Raysman & Brown, 2006; United States District Court of Massachusetts, 2010). AFC is basically a manual comparison approach. The theoretical framework of AFC not only makes possible the determination of “literal” similarities between two pieces of code, but it also takes into account their functionality to identify “substantial” similarities (Walker, 1996). In the AFC test, both the pirated as well as the original software are put through three stages, namely, abstraction, filtration and comparison. While other approaches (and the automated tools based on these approaches) generally compare two software packages literally, without considering globally common elements in the software, AFC, as the name indicates, first abstracts the original as well as the allegedly pirated, then filters out the globally common elements in them to zero in on two sets of comparable elements and finally compares these two sets to bring out similarities or “nuggets” (Walker, 1996).

On the copyright aspects of the software, the AFC-test specifies three categories (more aptly, levels) of exclusions, called doctrines (Walker, 1996). Firstly, if there is only one way of effectively expressing an idea (a function), this idea and its expression tend to “merge”. Since the idea itself would not be protectable, the expression of this idea would also escape from the field of the copyright protection. Secondly, there is the “scènes a faire” doctrine which excludes from the field of protection, any code that has been made necessary by the technical environment or some external rules imposed on the programmer. Thirdly, there are those elements that are in the public domain. These three categories of elements in the software are filtered out of the original as well as the allegedly pirated before arriving at the two set of comparable elements, mentioned above.

