A programming flaw (or a flounder) 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 Blunder (Baboo and Bhattathiripad, 2009) and thus, it is a design pattern / programming pattern related issue (see section 5.4). The programming blunder needs to be specifically contrasted with a software bug, as the latter, unlike the former, is bound to cause problems during execution. Ideally, all such flaws in programming 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 status as 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 of plagiarism of such software. To be precise, they can form an important basis for providing design pattern / programming pattern related evidence (see chapter 5) during software piracy forensics.

No existing software piracy forensic tool (whether manual or automated) considers programming blunder as important evidence. However, Spafford and Weeber have already anticipated the importance of (blunder-like) execution paths as cyber forensic evidence in “providing clues to the author” (Spafford and Weeber, 1992) (of the suspect /
alleged program). The concept of programming blunders (in this chapter) is a natural outcome of a specific study of such execution paths in the context of software piracy. The objectives of this chapter are set thus: (1) to thoroughly investigate the phenomenon of blunders in detail and by doing so attempt to 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 software.

6.1 Definition of programming blunder

The term programming blunder has already been introduced and identified by this researcher (but not properly defined) in some previous publications (Baboo and Bhattathiripad, 2009, Baboo and Bhattathiripad, 2011). Additionally, without using the term “Programming Blunder”, Spafford and Weeber (1992) have already mentioned about 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)

Thus, according to Spafford and Weeber, a programming blunder is 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 (Spafford and Weeber, 1992). Such a definition, however, has an inherent limitation in that it considers only inoperative statements (non-executed path) in the program. It does not consider those operative statements (executed paths) which 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 that are incoherently, redundantly and/or dysfunctionally appearing
in the text of the program and may have been executed at some stage but are hardly used in the user’s functionality (or to arrive at the final results). A more cautious definition will be employed in this study, namely:

A programming blunder found (in well tested and implemented software) can be defined as a variable or a 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.

### 6.2 Blunder genes in a program

A proper investigation of blunder, like that of any organism, should ideally start with a look into its genetic origin. Blunder genes (or genes of programming blunders) are those elements in the program that can often form the basis for (the existence 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).

What elements in the program can form the genetic basis of a blunder? Generally, 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 surface in a program in three different ways. In other words, programming blunders can be categorized in three ways according to their genetic differences as follows.

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

b) 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.

c) 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 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 be a programming blunder. Thus, a
A programming blunder may be an item (or a segment of a program) that is well defined in the program but is not part of the remaining stages (example: Processing stages, Reporting stages etc.) in the program. For instance, the integer variable ‘a’ in the C-program given in Table-6.1 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.

```c
#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 6.1. A defined but unused variable ‘a’ in a C-program

Secondly, any defined item (in the program) which is further used for data-entry but never used in the remaining stages of the program, can also appear as a programming blunder. Thus, the integer variable ‘a’ in the C-program given in table 6.2 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.

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

Table 6.2. 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 /* int a; */ in the C-program given in table 6.3 can turn out to be a programming blunder 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).

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

Table 6.3. 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 program can appear as a blunder in a program.

6.3 Features of a programming blunder

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

a) is a concrete, tangible item in (or segment of) a program and not a process.

b) is an execution path which got past the quality control stage, undetected.

c) does not affect the syntax or sometimes even the semantics of a program which makes it hard to detect.
d) is not necessary to execute the program.

e) is not necessary for user’s functionality.

f) does not justify its being there at all.

g) is a matter related to the design pattern and programming pattern of the software.

6.4 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 be reasons for programming blunders.

6.5 Programming blunders versus 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. This is 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. Bugs and blunders have something more in common in that neither should ideally be there in a program and should have been detected and edited out prior to the implementation of the program. However, they differ in one big way in that while bugs interfere with the proper functioning of a program, blunders do not. So, this confusion needs to be discussed and clarified before pursuing blunders further.
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 code\(^{56}\). A **programming blunder** (as defined at the beginning of the chapter) 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.

6.6 **Programming blunders as a functional expression of an idea**

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-6.2, 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-made\(^ {57}\) functional expression of an idea. This clearly opens the door to linking blunders directly to copyright

---


\(^{57}\) By 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.
infringements of any program because the idea-expression perspective is the basis of formulation of software copyright laws in several countries (Hollar, 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 (Hollar, 2002, p108). 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.

### 6.7 Programming blunders as indication of piracy

Despite their apparent immaterial nature, 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 chapter. 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 (Spafford and Weeber, 1992).

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, 2006), SCAP (Frantzeskou, et al., 2007) 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.
Programming blunders may be dismissed as insignificant or incidental or predictable elements in a program and so, cyber experts are likely to disregard them as not too reliable (Baboo and Bhattathiripad, 2009). However, most programming blunders can provide direct evidence to establish piracy and others can provide probable, corroborative or supporting evidence. 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. Anyway, blunders are forensically important. The repetition of programming blunders\(^{58}\) (even if these programming blunders are ‘generic’ by nature) in both original\(^{59}\) and the pirated\(^{60}\) 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 original\(^{61}\) as well as the pirated\(^{62}\) 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 table 6.4).

| 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 |

---

58 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.

59 Throughout this chapter, original means the version of the software that the complainant submits to the law enforcement agency for software piracy forensics. This chapter 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.

60 Throughout this chapter, pirated means the allegedly pirated software.

---

58, 59, 60, 61, 62
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.

Table 6.4: Secondary or inconclusive programming blunder genes

### 6.8 AFC and Programming blunders

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 Chapter 3 for more details on AFC) 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, 2006, p86]. The functionally inactive parts (or those items are irrelevant for user’s functionality) like programming blunders will not be considered for abstraction. In such case of unfortunate inconsideration, the programming blunders will not be available for final comparison and so, the final results of AFC cannot be fully relied upon (or taken as the only evidence) by the expert to establish software copyright infringement. 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 reporting his/her the final findings and inferences to the court. In this report, what the judge expects from the cyber forensic 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 brief\(^{63}\) (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:

\(^{63}\) Amicus Brief of Computer Scientists, Harbor Solutions v. Applied Systems No. 97-7197 (2nd Circuit, 1998) at 8-9 (citations omitted)
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 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 makes the relevance of the programming blunders crucial 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, 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).

The ultimate decisions like whether these pieces of evidence (concerning programming blunders) (1) are useful or not; (2) are direct or secondary evidence; and (3) are generic (and hence, non-copyrightable) by nature or not, are 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.

6.9 Conclusion

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. Programming blunders can greatly influence the expert to give support to his findings, thus indirectly influencing 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. The new protocol, explained in Chapter 8, has been designed with this in mind.