Bugs

Computer Measurement Laboratory
info@cmlab.biz

Perhaps the most useless and misinformative term in use in the software development world is the term, ***bug***. It means some many different things to so many different people. But, what is worse, there is no standard definition for the term. It could be a term that refers to a failure event. "A bug just killed my program." It could refer to a defect in the code. "There is a bug in the code." It could refer to a heroic design fault. "The first Ariane 5 spacecraft had a bug in the software."

Sometimes to fix a bug, hundreds of lines of code change. Sometimes to fix a bug, a single character in a single line of code will change. This seems to imply that bugs come in different sizes. Sometimes they are the result of programming problems. Sometimes the programs are just fine but the design was faulty.

It makes little or no sense to try and track these bugs in bug tracking tools. We simply do not know what a bug is. Therefore it makes no sense to "track" them. Let us, then, turn our attention to a more precise terminology for software defects and then to a means of measuring these events.

Unfortunately there has been no particular definition of just precisely what a software defect or fault is, a problem that we intend to ameliorate. In the face of this difficulty it has rather hard to develop meaningful associative models between faults and metrics. In other words, a fault is a physical characteristic of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. People making errors in their tasks introduce faults into a system. These errors may be errors of commission or errors of omission. There are, of course, differing etiologies for each fault. Some faults are attributable to errors in the specification of requirements. Some faults are directly attributable to error committed in the design process. Finally, there are faults that are introduced directly in to the source code.

Software faults simply cannot be measured. If we had a tool that could peruse a source code base and identify faults, the progress that could be made in software development would be astonishing. Sadly, we will only be able to recognize them when they cause problems or through intense scrutiny of individual source code modules.

There are two major subdivisions of faults in our fault taxonomy. There are faults of *commission* and faults of *omission*. Faults of commission involve deliberate, albeit unwitting, implementation of a behavior that is not part of the specification or design. Faults of omission involve lapses wherein a behavior specified in the design was not implemented. It is important to make these distinctions especially so the inspection protocol can be used as a checklist for specific faults that have been found in the past.

In order to count faults, there must be a well-defined method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our static source code measurements. In a careful examination of software faults over the years, we have observed that the overwhelming number of faults that are recorded as code faults are really design faults. Some software faults are really faults in the specification. The design implements the specification and the code implements the design. We must be very careful to distinguish among these fault categories.

There may be faults in the specification. The specification may not meet the customer's needs. If this problem first manifests itself in the code, it still is not a code fault. It is a fault in the program specification or a *specification fault*. The software design may not implement the software requirements specification. Again, these design problems tend to be made manifest during software testing. Any such design faults must be identified correctly as *design faults*. In a small proportion of faults, the problem is actually a code problem. In these isolated cases, the problem should be reported a *code fault*.

We observed an example of this type of problem recently in a project on a large embedded software system. The program in question was supposed to interrogate a status register on a particular hardware subsystem for a particular bit setting. The code repeatedly misread this bit. This was reported as a software problem. What really happened was that the hardware engineers had implemented a hardware modification that shifted the position of the status bit in the status register. They had failed to notify the software developers of this material change in the hardware specification. The software system did exactly what it was supposed to do. It is just that this no longer met the hardware requirements. Yet the problem remains on record to this date as a software fault.

It is clear, then, that the etiology of the fault must be determined. It is the subject of this discussion to identify and enumerate faults that occur in source code. We could also leverage this discussion to the measurement of faults in design documents and also in requirements specification documents. Unfortunately, most programs have neither of these documents so we will, for the time being, simply focus on how to measure faults in source code. We ought to be able to do this mechanically. That is, it should be possible to develop a tool that could count the faults for us. Further, some program changes to fix faults are substantially larger than are others. We would like our fault count to reflect that fact. If we have accidentally mistyped a relational operator like '<' instead of '>' , this is very different from having messed up an entire predicate clause from an `if` statement. The actual changes made to a code module are tracked for us in configuration control systems such as `rcs` or `sccs` as code deltas. All we must learn to do is to classify the code deltas that we make as to the origin of the fix. In other words, each change to each module should reflect a specific code fault fix, a design problem, or a specification problem. If we manifestly change any code module, give it a good overhaul, and fail to record each fault as we repaired it, we will pay the price in losing the ability to resolve faults for measurement purposes.

We will base our recognition and enumeration of software faults on the grammar of the language of the software system. Specifically, faults are to be found in statements, executable and non-

executable. In the C programming language we will consider the following structures to be executable statements.

```
<executable_statement> ::= <labeled_statement> |
<expression> |
<selection_statement> |
<iteration_statement> |
<jump_statement>
```

In very simple terms, these structures will cause our executable statements metric count to change. If any of the tokens change that comprise the statement then each of the change tokens will represent a contribution to a fault count.
Within the framework of non-executable statements there is

```
<declaration> ::= <declaration_specifiers> ;
| <declaration_specifiers> <init_declarator_list> ';'
```

We will find faults ***within*** these statements. The granularity of measurement for faults will be in terms of tokens that have changed. Thus if I typed the statement in C

```
a = b + c * d;
```

but I had meant to type

```
a = b + c / d;
```

then there is but one token that I had got wrong. In this example, there are eight tokens in each statement. There is one token that has changed. There is one fault. This circumstance is very different when wholesale changes are made to the statement. Consider that this statement

```
a = b + c * d;
```

was changed to

```
a = b + (c * x) + sin(z);
```

We are going to assume, for the moment that the second statement is a correct implementation of the design and that the first was not. This is clearly a coding error. (Generally when changes of this magnitude occur they are design problems.) In this case there are 8 tokens in the first statement and 15 tokens in the second statement. This is a fairly substantial change in the code. Our fault recording methodology should reflect the degree of the change. This is not an unreasonable or implausible notion. If we are driving our car and the car ceases to run, we will seek to identify the problem or have a mechanic do so for us. The mechanic will perform the necessary diagnostics to isolate the problem. The fan belt may have failed. That is a single problem and a simple one. The fan belt may have failed because the bearing on the idler pulley failed. We expect that the mechanic will isolate *all* of the problems and itemize the failed items on our bill. How much information would we have if the mechanic simply reported that the engine broke? Most of us would feel that we would like to know just exactly what pieces of the engine had failed and were subsequently replaced. We expect this level of granularity in reporting engine problems. We should expect the same level of granularity of reporting on code fixes.

The important consideration with this fault measurement strategy is that there must be some indication as to the amount of code that has changed in resolving a problem in the code. We have regularly witnessed changes to tens or even hundreds of lines of code recorded as a single "bug" or fault. The only really good index of the degree of the change is the number of tokens

that have changed to ameliorate the original problem. To simplify and disambiguate further discussion consider the following definitions.

> ***Definition:*** A fault is an invalid token or bag of tokens in the source code that will cause a failure when the compiled code that implements the source code token is executed.
>
> ***Definition:*** A failure is the departure of a program from its specified functionalities.
>
> ***Definition:*** A defect is an apparent anomaly in the program source code.

Each line of text in each version of the program can be seen as a bag of tokens. That is, there may be multiple tokens of the same kind on each line of the text. When a software developer changes a line of code in response to the detection of a fault, either through normal inspection, code review processes, or as a result of a failure event in a program module, the tokens on that line will change. New tokens may be added. Invalid tokens may be removed. The sequence of tokens may be changed. Enumeration of faults under this definition is simple, straightforward. Most important of all, this process can be automated. Measurement of faults can be performed very precisely, which will eliminate the errors of observation introduced by existing ad hoc fault reporting schemes.

An example would be useful to show this fault measurement process. Consider the following line of C code.

(1) `a = b + c;`

There are six tokens on this line of code. They are B_1 = {<a>, <=>, <b>, <+>, <c>} where B_1 is the bag representing this token sequence. Now let us suppose that we desired, in fact, required that the difference between b and c be computed to wit:

(2) `a = b - c;`

There will again be six tokens in the new line of code. This will be the bag

B_2 = {<a>, <=>, <b>, <->, <c>}.

The bag difference is

$B_1 - B_2$ = {<+>, <->}.

The cardinality of B_1 and B_2 is the same. There are two tokens in the difference. Clearly, one token has changed from one version of the module to another. There is one fault.

Now let us suppose that the new problem introduced by the code in statement (2) is that the order of the operations is incorrect. It should read

(3) `a = c - b;`

The new bag for this new line of code will be

B_3 = {<a>, <=>, <c>, <->, <b>}.

The bag difference between (2) and (3) is

$B_2 - B_3$ = { }.

The cardinality of B_2 and B_3 is the same. This is a clear indication that the tokens are the same but the sequence has been changed. There is one fault representing the incorrect sequencing of tokens in the source code.

Now let us suppose that we are converging on the correct solution however our calculations are off by 1. The new line of code will look like this.

(4) `a = 1 + c - b;`

This will yield a new bag

B_3 = {<a>, <=>, <1>, <+>, <c>, <->, <b>}.

The bag difference between (3) and (4) is

$B_3 - B_4$ = { <1>, <+> }.

The cardinality of B_3 is six and the cardinality of B_4 is eight. Clearly there are two new tokens. By definition, there are two new faults.

It is possible that a change will span multiple lines of code. All of the tokens in all of the changed lines so spanned will be included in one bag. This will allow us to determine just how many tokens have changed in the one sequence.

The source code control system should be used as a vehicle for managing and monitoring the changes to code that are attributable to faults and to design modifications and enhancements. Changes to the code modules should be discrete. That is, multiple faults should not by fixed by one version of the code module. Each version of the module represents should represent exactly one enhancement or one defect.

We will take a simple example and trace the evolution of a source code program through three successive revisions through the UNIX `rcs` program. The sample program is as follows:

```
1  int Sum(int upper)
2  {
3    int sum = 0;
4    int index = 0;
5
6    label:
7       if(index < upper)
8          {
9             index++;
10            sum = sum + index;
11            goto label;
12         }
13   return sum;
14 }
```

The program above represents version 1.1 of the program. Successive updates to this will be 1.2, 1.3, etc. The `rcs` system will keep track of the version number, the date and time of the update, and the author of the `rcs` activity. An abridged version of the `rcs` module structure to record these data is shown in the table below.

Table 1. `rcs` Header Information

1.4 date 2005.02.01.22.17.38; author John Doe; next 1.3;
1.3 date 2005.01.22.22.01.31; author John Doe; next 1.2;

```
1.2
date  2005.01.20.21.54.08;     author Sam Lee;
next  1.1;
```

```
1.1
date  2005.01.15.21.49.29;     author Mary Roe;
next  ;
```

The odd part of `rcs` is that the most recent version, in this case 1.4, is kept at the top of the list and the list is numbered chronologically backwards in time. Each version keeps a pointer to the next version in the table.

The actual changes to the source code at each version are shown in Table 2. The `rcs` program will always keep the most recent version in the file. This is shown in the table entry beginning with, in this case, version 1.4. The second entry in the record for version 1.4 is an entry beginning with the word log and delimited by @'s. This is the log comment introduced by the developer. In our proposed model this log entry would begin with the word, `fault,` if the version increment were attributable to a fault fix or the word, `change,` if it were attributable to a change in design or requirements. The initial log entry, version 1.1, is for neither a change nor a fault fix but is the title of the program.

Table 2. `rcs` Text Information

```
1.4
log
@fault: fixed relational operator
@
text
@int Sum(int upper)
{
  int sum = 0;
  int index = 0;
  label:
     if(index > upper)
        {
           index++;
           sum = sum + index;
           goto label;
         }
  update (index);
  return sum;
@
```

```
1.3
log
@fault: inserted call to update function
@
text
@d7 1
a7 1
     if(index <= upper)
```

```
@
1.2
log
@fault: found a problem with a relational operator
@
text
@d13 1
@
1.1
log
@Initial revision
@
text
@d7 1
a7 1
     if(index < upper)
@
```

Following the `log` entry is the `text` entry. In the case of `rcs`, the topmost `text` entry is the most recent version of the program. Each of the subsequent table entries show the changes that must be made to the most recent program to change it to a previous version. All changes are made, in `rcs`, by adding or deleting whole lines. Thus, to return to version 1.3 from version 1.4, the text part of record 1.3 tells us to go to line 7 (relative to 1) of the program and delete one line. That is what the line `d7 1` tells us. The next text line says that we must add one line, `a7 1`, again at line 7. The text that must be added is on the following line. Thus, version 1.3 will look like this:

```
1  int Sum(int upper)
2  {
3    int sum = 0;
4    int index = 0;
5
6    label:
7       if(index <= upper)
8          {
9             index++;
10            sum = sum + index;
11            goto label;
12         }
13   update (index);
14   return sum;
15 }
```

Line number 7 has been changed on version 1.3. Let

B_2 = {<if>, <(>, <index>, <<=>, <upper>, <)>}.

represent this bag of tokens. On version 1.4 the bag of tokens is

B_1 = {<if>, <(>, <index>, <>>, <upper>, <)>}.

The bag difference is

$B_2 - B_1$ = {<<=>, <>>}.

The cardinality of B_2 is 6 and the cardinality of B_1 is 6. The cardinality of the bag difference is two. Therefore, one token has changed and we will record one fault.

To return to version 1.2 from version 1.3 we see that we must delete line 13. All of the tokens on this line were placed there in remediation of a fault. The bag representing this line of tokens is

B_3 = {<update>, <(>, <index>, <)>, <;> }.

There are five tokens on this line. There was no former version of this line in version 1.2. Therefore all of the tokens on this line were put into the program to fix a defect in the program. We will then record 5 faults for this fix.

Finally, to return to the initial version, 1.1, of the program we must delete line 7 and add a new line represented by the bag

B_4 = {<if>, <(>, <index>, <<>, <upper>, <)>}.

This is similar to the transition between versions 1.3 and 1.4. Only one token has changed. We' will record one fault for this module version.

Now that we have a standard notion of what a software fault is and how to enumerate it, we will now foreswear the use of the term bug forever more.