What is the Best Software Metric?

Computer Measurement Laboratory
info@cmlab.biz

People often want to know what the best software metric is. That is kinda like asking what is the best metric for choosing a spouse. In other words, if you could know only one thing about your future husband/wife what would that be? That is a very improbable question. People are very complex and they have many attributes. The choice of a spouse based solely on height, for example, would not be a very good idea. Measuring software based on a single attribute is equally inappropriate. There is, then, no such thing as a ***best*** software metric.

Software measurement is not for novices. Learning to measure software requires substantial training and understanding. There are two real pitfalls in the software measurement business. The first of these is the software measurement tools. These tools produce data and not information. The data that that they do produce are ill defined at best. In that there are no software measurement standards, each such tool measures the same ostensible attribute in a different way. The second pitfall in the metrics business is the availability of statistical analysis tools. In the hands of a statistical novice, these tools easily produce non-informative or erroneous information. It is rather like giving a two-year-old child a loaded 45 caliber semi-automatic pistol to play with. Their use on software measurement data should be confined to experts in statistics.

Measurement Dimensionality

There are many different program attributes. Programs may be characterized by their size or length. Programs may be characterized by the interrelationship among their program modules. Programs may be characterized by the data structures that they contain. They may also be characterized in the flow of control within each program module.

There are also many different program metrics that can measure aspects of each of these attributes. Some of these metrics are highly correlated with each other. As an example, we have the Lines of Code measure of program size. We can also measure a program in terms of its statement count. These two metrics are not measuring exactly the same thing. In the C programming language, it is possible to have many statements on one line of code. It is also possible for a single statement to span several lines of code. Therefore, these two metrics are not measuring exactly the same size attribute.

Some ostensible program metrics don't measure any program attributes. An example of such a metric is comment count. The number of comments in a program module is directly proportional to the lack of design documents for that module. A good design document for a program should completely obviate the need for comments within the program itself. Therefore, if we were to measure comments in a program module we would be measuring aspects of the software development process or the developer and not the program itself.

A good measurement program will seek to identify as many meaningful program attributes as possible and then identify tools that measure these attributes.

Criterion Measures

Measuring software attributes is in and of itself a very meaningless process. We really don't care how many lines of code a program has or what the coupling metrics are for its program modules. It is pointless to know these things. What we really want to know about a program are attributes of software quality, for example. We would like to know how many faults there are in a program module. We would like to know how many faults we just introduced into the module based on the changes that we just made to the module. Unfortunately, the very things that we would like to know about a program and its modules, Nature will not disclose to us.

These unknowable attributes of software we will call criterion measures. Before we consider measuring any other program attributes we must first established suitable criterion measures. We could, for example, chose software faults as a criterion measure. In which case we will develop a very precise standard for measuring and recording these faults. If we have done a good job of this standards formulation, we will be able to build a software tool that will extract these fault data from our configuration management system.

We can then use these criterion measures from historical program development to build predictive models from our source code metrics. In this manner, we can learn from the past. If we build really good predictive models for software, we can measure a new software system and have a real good idea where the problems in the code are likely to be. We will also have a pretty good estimate on how many faults, for example, there might be in a new system.

Measurement Granularity

A program is made up of many different program modules. A program is the sum of its parts or modules. When the program changes during the build, some modules change, some modules are deleted and some modules are added. We are primarily interested in the changes to a program at its lowest level. Therefore all measurements of a program will be taken at the module level. The term, module, is not without ambiguity. For the C programming language, for example, a program module will be defined as a function module. This definition will not necessarily be applicable to all programming languages. The important thing is that the notion of a module must be unambiguously defined and written down as a part of a measurement standard.

In some programming languages, such as C, program modules are grouped together in files. This is typically done because we wish to use global defines or global variables that will be common to all program modules. This is a very bad programming practice but it is also ubiquitous. Therefore we will need to address it. If, for example, there is a variable declaration for a global variable, then this statement must be enumerated as part of each program module in the file. Similarly the Lines of Code metric for each program module must include the total Lines of Code count for all of the statements at the global level in the file that contains the program module.

Metric Primitives

A metric primitive is a single measurement of a single program attribute. The Lines of Code metric is such an example. It is a measure of program size or length.

Each metric primitive must have a well-defined and unambiguous standard. For the Lines of Code metric, for example, we might simply count the number of <\n> characters in a program. Unfortunately, this might be misleading in that some programmers use quite a bit of white space in their programs and others might not use any. This programmer-induced source of variation can be eliminated by adding to the standard the notion that lines that only contain a <\n> will no be counted.

We will also insist that each metric in our measurement system should explain sources of variation in the chosen criterion metric(s) that is not explained by other primitive metrics. In this sense, we are insisting that our metrics have validity with respect to our chosen criterion metrics(s).

Derived Metrics

Many folks in the software business put a great deal of stock in what we will call derived metrics. These are arbitrary linear and (worse) non-linear compounds of primitive metrics. A good example of this would be Maurice Halstead's Program Vocabulary metric. This is the sum of the total number of program operators and program operands. There is no new information in this derived metric of program vocabulary. If I were to tell you that I had 25 cents in my left pants pocket and also that I had 75 cents in my right pants pocket, you would learn nothing new in my telling you that I had $1.00 in change on my person.

Some derived metrics are indeed misleading to misinformative. A very good example of this type of metric is McCabe's Cyclomatic Complexity. This metric is derived by subtracting the number of nodes in a control flow graph representation of a program module from the number of edges. The metric primitives, nodes and edges, are really good measures of control flow complexity. Unfortunately, their difference is simply a measure of program size. In other words, its is highly correlated with other measures of program size such as Lines of Code or statement count.

McCabe's metric is a very good example of what is wrong with derived metrics. It is almost uniformly true that these derived metrics have never been validated. That is, there is no empirical evidence to show that the metrics are, in fact, related to the program constructs that they purport to measure. The Department of Defense routinely evaluates programs and program modules by their Cyclomatic Complexity as a measure of control flow complexity. As a rule, the Cyclomatic Complexity for any one module is therefore not allowed to exceed 15 (truly a magic number), To get around this problem for a module whose value exceeds 15, the typical developer will split the module into two separate modules. This has the net effect of shifting the complexity of an attribute that is being measured (size) to one that is not being measured (coupling). In general, this misuse of a metric will result in the net complexity of a program to increase but, unfortunately, in attributes that are not being measured.

Some software development managers attempt to be creative quite on their own. They will create their own derived metrics. The thing that is wrong with this attempt is that metric primitives, in general, measure different software attributes. It just doesn't make sense to add these metrics. It is rather like adding a programmer's height and weight together to create a new measure of programmer volume.

A very good rule, then, for the use of derived metrics is: DON'T. Derived metrics are simply not valid at their best or misleading at their worst.

Characteristics of a Good Software Measurement Program

As we have clearly established, software metrics in and of themselves are vacuous. They only have meaning based on their relationship with some type of criterion measure. Therefore, the first step in establishing a good software measurement program is to define one or more software quality criterion measures. We might choose, for example, to measure software faults. That said, then we must establish a good unambiguous standard for enumerating these faults. This will insure that everyone in the organization will report them in exactly the same way. Even better, we could build a tool that would extract these fault data for us given a completely unambiguous definition of the notion of a software fault. Further, we will realize that there are faults attributable to problems in the requirements document, in the software design documents, or in the code itself.

Once we have established a meaningful standard and measurement methodology, we are then in a position to begin to collect measurement data for program products. Again, we will chose metric primitives that we know are closely related to our criterion measure. We will then choose a measurement tool that will produce these data for us.

A good software metric program will be built into the software build process itself. A program going through a series of builds is evolving. This means that the metrics for each program module may be changing in this process. We must be able to know what the metrics are for all modules of each program build. This means that the measurement process itself must be continuous.