

Quality Metrics

Contents

1	Examples of Quality Metrics	1
1.1	The Scope	1
1.2	Halstead's measure E/Software Science	1
1.3	Gilb's Portability Measure	3
1.4	Steiwer's Readability Formula	4
2	Other Methods of Quality Assessment	6
2.1	Quality Tests	6
2.2	Quality Checklists	7

1 Examples of Quality Metrics

1.1 The Scope

Using a limited subset of qualities, the following list of metrics show what has been developed.

These are some example metrics.

Halstead's Software Science Measure Steiwers Readability Formula and Gilb's portability measure.

They are all quite different, in purpose and method - the descriptions all come from a joint NCC/GMD study on Software Quality.

1.2 Halstead's measure E/Software Science

There have been many attempts to define and measure the complexity of computer programs, one of the earliest and probably the best known was devised by MH Halstead in the 1970's. He called his theories "Software Science" and although recently being the subject of some criticism, it was innovative when introduced, as it was the first time that some one had tried to measure programming, and area which previously had been accorded the status of a "craft" initially, and therefore considered not qualifiable. Its purpose was to measure the complexity and quality of programs, but it was later applied to estimating program reliability, and programming effort.

It is based, simply, on a count of program operators, and operands. This includes plain arithmetic operators like, + - * /, logical operators like greater than and equal to, and KEYWORDS, like "perform" in COBOL, plus the delimiters for logical blocks eg full stops. The operands are any constants and variables used.

In this example piece of COBOL Code the metrics are shown at the bottom.

para-1.

READ data-file INTO data record

INVALID KEY

MOVE error-message to display-area

NOT INVALID KEY

IF Record-key = I

MOVE "record type =1" to record-type-display

ELSE

MOVE "incorrect record type" to record-type-diaplay

END-IF

END-READ

PERFORM record-type-print.

Program Analysis			
Operators		Operands	
Move	3	data-file	1
Read	1	data-record	1
Invalid key	2	error-record	1
perform	1	display-area	1
=	1	record-key	1
If	1	"record type"	1
Else	1		
End-If	1	record-type-display	2
End-Read	1	"Incorrect record type"	1
.	2		
Total	14	Total	9
n1	10	n2	8
N1	14	N2	9

These values are used to calculate N - the length of a program by

$$N = N1 + N2$$

and n (The program vocabulary, ie the number of different operators and operands)

$$= n1 + n2$$

The larger the value of N, the greater the difficulty in understanding the program and the greater the effort required to maintain it. It is superior to counting the number of lines of code in that it is restricted to lines of instructions, as opposed to say, lines of comment.

His theory was used later by ELSHOFF, who used a different formula variation to produce:

$$N = n1 \log_2 n1 + n2 \log_2 n2$$

For an even more accurate gauge of difficulty -using this new metric and comparing it to N he found that the two related most closely in well structured programs. Thus, the comparison became a simple indicator of structuredness.

N2 has also been used to indicate program complexity; in that in its simplest form a program with one input and one output has the n2 value 2; increasing the number of inputs or outputs

increases the value of n_2 , which turn results in increased programming effort. In fact WOODFIELD, in an experiment entailing the conversion of ALGOL programs to FORTRAN, and then expanding the number of outputs by 1, observed that a 25% increase in n_2 doubled the programming effort required.

Halstead went on to approach program size using he number of times elements are used against the range of elements from which the choice is made. This resulted in

$$\text{VOLUME, } V = N \log_2 n$$

He assessed the difficulty in writing code as linked with the number of decisions to be made, the operators used and the way in which variables are used.

Thus:

$$\text{Difficulty } D = \frac{n_1}{2} \times \frac{N_2}{n_2} =$$

$$\frac{\text{Vocabulary}}{2} \times \text{average number of times each operand is used}$$

Combining both: Effort = a product of Difficulty and volume

$$E = D * V$$

1.3 Gilb's Portability Measure

Defined by him as "the ease of conversion of a system from one environment to another; the relative conversion cost for a given conversion method or algorithm".

The resources which are measured initially may be manpower or time or machine resources, but the aim is to convert whichever of these is chosen into economic units.

The formula is stated as

$$PS = 1 - (ET/ER)$$

Where:

- PS is the portability measure;
- ET is the resources estimated to move the system to its new environment;
- ER is the resources required to create the system for its resident environment.

For example

The system has cost 100,000 money units to create in the current environment, say IBM 370 under DOS, and the estimated maximum cost (discounted in current money value) for moving it to say a VAX is 10,000 money units.

$PS = 1 - (10,000/100,000) = .9000$
therefore portability is 90%

Confirming that portability comes in degrees, is measurable and can be used as system design objective it is useful for evaluating tasks like conversion.

The implication of the measure is that since it is a linear function, it is assumed that for a program of 100,000 statements the conversion effort will be 100 times that for a 1000 statement program.

The following is an example clause which Gilb used in a contract to ensure this quality in a payroll package which was expected to run on three different computers; using the metric, to ensure that actual cost limits were set up.

The producer of the software is motivated by the measure to design and construct the software to a predetermined level. If they fail they have to pay a financial and practical penalty. Note that performance and reliability are specified as constant quality parameters so that the producer does not simplify the conversion task at the expense of other system qualities.

The concept can be fruitfully applied to many sub-systems besides logical programs, for example human procedures, and hardware. The criterion should be "can portability be influenced by alternative design decisions in the subsystem?" If the answer is "yes" then the alternatives should be considered, with it in mind.

Example - the purchase of office micros. The advantages of high portability being:

The ability to transfer to future environments with ease;
The ability to be spread to multiple environments;
The ability to function in back-up environments.
A very different measure is

1.4 Steiwer's Readability Formula

This was developed by Laure Steiwer at the Institute Pedagogique in Luxembourg in the mid-seventies as a measure of readability of German text documents.

It advocates something called an "estimated mean doze value" to measure the comprehensibility of prose text. The term "doze" is derived from closure, a Gestalt psychology term referring loosely to the human tendency to complete mentally a familiar but unfinished pattern. Prose based doze tests are constructed by replacing a percentage of words in a text with short blank lines. Subjects attempt to reconstruct the original text by filling in the blank lines. It is assumed that the subject's comprehension of the original text is an

increasing function of the number of words the subject replaces correctly. Using the "doze" value 60 different prose texts were analysed, and 38 different variables were produced which described different aspects of text difficulty.

This set was reduced to 25 indicators, which in turn were correlated to produce 3 readability formulae:

- 1 an exact readability formula - to produce all 25 variables included;
- 2 a computer formula - which only used variables allowing automatic text analysis;
- 3 a shortcut formula - using 4 variables which can be calculated manually.

The validity coefficients for each range from

0.87 (shortcut 0 to 0.91 (exact))

For Example

The shortcut formula reads as follows:

$$V^* = 235.95993 - (\text{VAR2} * 73.02100) - (\text{VAR1} * 12.56438) - (\text{VAR} * 50.03293)$$

Where:

V^* = estimated mean doze value (reading ease)

$\text{VAR1} = \log [(AWOR/ASAT) + 1.0]$

$\text{VAR2} = \log [(ABUC/AWOR) + 1.0]$

$\text{VAR3} = AUWO/AWOR$

(log = natural log)

AWOR = no. of words

ASAT = no. of sentences

ABUC = no. of letters

AUWO = no. of different words

High doze values mean good comprehensibility.

There are criticisms of measures like this:-the main one being that they are too dependent on a syntactic analysis, and ignore the logical organisation and semantic structure of the text, others are that there is no theoretical foundation linking cognitive information processing with the variables and text comprehensibility.

There is widespread use of this kind of measure in the development of technical documents, and even creative writing, which indicates that they may also be useful for measuring the comprehensibility of software documents. Recently there has been some research into doze testing as an alternative to multiple choice quizzes, for assessing the comprehensibility of program code. Although initial results showed that in some cases there was little correlation it appears that by careful separation of the code into "program-dependent " (requiring at least

some understanding of the purpose, functionality, and algorithm involved) and "program-independent" (can be completed with only syntactic knowledge and general reasoning) items, more reliable figures can be obtained, although it is accepted that this is only the first step in being able to use the measure universally (Hall and Zwebon IEEE S/E May 86).

2 Other Methods of Quality Assessment

2.1 Quality Tests

These are allied in many cases to quality metrics, but there are some occasions when testing procedures are unrelated to known quality metrics except in principle. An example of this is compiler validation. This is a method of software assessment originally introduced by the US Government, in an attempt to make all compilers purchased by them comply with defined language Standards. The testing is done by subjecting the computer to a series of programs, written especially to check each feature of the language as defined in the Standard. The method used is that of "black-box" testing, with the results being analysed usually in the form of a test pass or test fail.

In most cases tests include the compiler and the object programs generated by it. Sometimes the operating system and supporting run-time software is also tested.

The tests are of basically 4 types:

Conformance: checks that programs which conform to the rules of the language as laid down in the standard, are handled correctly.

Error-handling: these check that any error routines defined in the standard are executed at the correct time and in the correct way.

Where the language standard defines the circumstances adequately, some validation suites also include:

Deviance-tests: these are deliberately incorrect tests which determine that the compiler rejects all deviations from the standard.

For some languages there are areas which are not controlled by the standard and are implemented in different ways by compiler producers depending on what they perceive to be the most efficient method and the limitations of the hardware being used. Some testing procedures identify these and produce information about them.

Finally, extensions to certain compiler validation suites check and assess features not specifically included in the standard, but which are considered to be important. These are generally known as Quality tests, and cover areas such as mathematical accuracy.

Despite the comprehensive testing techniques applied in this area, a product can be checked, validated and certified, yet still have errors. There is guarantee given that the compiler does anything more than conform to the contents of the standard. Because of it way the software is being checked the qualities which are being assessed are limited particular instance and some

important ones are ignored. Apart from this the concept of testing quality in this way has still not been generally put into practice, although through its very existence, this service has provided the first steps towards a National Software Quality Centre. It is proposed that this should be developed by providing increasing levels of software certification, over a wide range of applications.

2.2 Quality Checklists

Where it is not yet possible to measure or test for quality levels within a piece of software, an alternative method of discovering the extent of quality or the qualities, is to isolate those system features which could contribute to the creation of the quality characteristic. These can be correlated to observations of the quality, to determine which ones are most influential. The list can be made more usable by converting it to a series of questions, preferably requiring a "yes" or "no" answer. The biggest disadvantage of checklists over "objective" measures is that both composition and completion relies on the qualitative judgement of a human evaluator. There are many other uses besides basic qualities; checklist can be based on the requirements of the original system specification to ensure that these are being met, for instance.

Checklists are being used more and more as a replacement for lists of actions, their advantage being that people can respond more easily to direct questions and their responses can be recorded simply. This reduces the chance of missing particular items and simple questions reduce the possibility of misunderstandings.

Checklists can be used over several stages of the system life-cycle, to produce a specific quality, like this example of part of a reliability checklist:

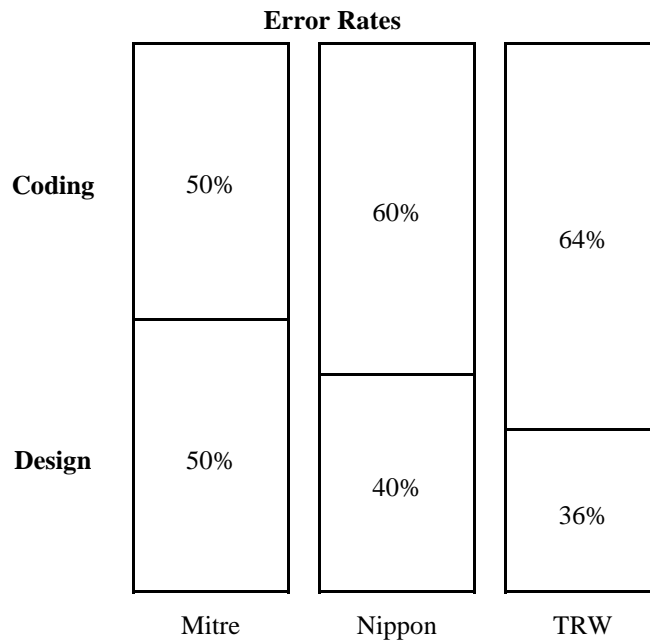
Program Reliability Checklist

1. does the program check for illegal arithmetic operations (eg dividing by zero)
2. are loop termination and index items limits tested before use?
3. are subscript limits tested before being used?
4. have error recovery and restart procedures been included?
5. has input data been validate?
6. are test results satisfactory (do actual and expected results match?)
7. do tests indicate that....?

or they can be used to check for the presence of a particular quality at the end of one development stage.

More and more emphasis is being given to ensuring that quality is being incorporated into software during its production and tested for at the end of each stage. This approach is fundamental to both structure techniques and software engineering and results in errors being detected as early as possible in the development process. This results in more reliable software, at a lower cost. The savings are produced because the earlier that errors are detected, the smaller is the cost of correcting them. Studies have also shown that errors are generally distributed between design coding in the following proportions.

IBM STUDY (IBM Knight-on Software Quality and Productivity)



If correcting design errors detected after coding means more substantial rewriting of programs (and increases the probability of introducing additional coding errors) then catching the errors at the earliest stage can only prove to be beneficial.

Checklists have been found to significantly improve the productivity of the inspection process at the end of each phase of the development cycle, as follows:

Experience has shown that modules having a high number of errors per thousand object code locations later prove to give rise to disproportionate amounts of testing, maintenance, etc. It is beneficial to identify these modules early and take corrective action as soon as possible. The following is a list of modules with an indication of the most error prone.

Error Proneness			
Module	No. of Errors	Lines of Code	Error Density Errors / K100
A	4	128	31
B	10	323	31
C	3	107	28
D	7	264	27
E	2	106	19
F	3	195	15

average error rate assumed here

Using a distribution of error types, the statistical deviations from the expected indicate which

modules are most likely to benefit from analysis and correction.

Distribution of Error Types			
	No. of Errors	%	Normal / Usual Distribution
Logic	23	35	44
Interconnection	21	31	18
Control Blocks	6	9	13
.....
.....
.....
		100%	100%

In this example the interconnection module has a higher than usual error rate and will need examination.

Each of the modules is analysed and detailed lists of the errors found during inspections are then produced. From these, lists of the typical error types are made and the ones which occur most frequently and/or cost most to repair can be identified. If the clues which reveal the presence of each of the error types can be described, they can be incorporated into a checklist. This can be used to effectively prompt people to find the most important errors. The following is an example detailed design inspection checklist for "logic" errors:

Missing

1. Are all constants defined?
2. Are all unique explicitly values tested on input parameters?
3. Are values stored after they are calculated?
4. Are defaults checked explicitly tested on input parameters?
5. If character strings are created are they complete, are all delimiters shown?
6. If a keyword has many unique values, are they all checked?
7. If a queue is being manipulated, can the execution be interrupted; if so, is queue protected by a locking structure; can queue be destroyed over an interrupt?
8. Are registers being restored on exits?
9. Are queuing/de-queuing should any value be decremented/incremented?
10. Are all keywords tested in macro?
11. Are all keywords related parameters tested in service routine?
12. Are queues being held in isolation so subsequent interrupting requestors are receiving spurious returns regarding the held queue?
13. Should any registers be saved on entry?
14. Are all increment counts properly initialized (0 or 1)?

Wrong

1. Are absolutes shown where there should be symbolics?
2. On comparison of two bytes, should all bits be compared?
3. On built data strings, should they be character or hex?
4. Are internal variables unique or confusing if concatenated?

Extra

1. Are all blocks shown in design necessary or are they extraneous?