WHITE PAPER

Which Software Quality Metrics Matter?

Introduction

As in all scientific and engineering disciplines, a quantitative measure of the software is often required. These metrics are used in numerous applications such as planning and performance optimization and especially as a measure of software quality.

However, there is much discussion over which, if any, metrics are actually of value. In this paper, we look at some of the arguments concerning the necessity of software metrics and whether their use actually improves software quality.

Software Metrics

Generally, software metrics and software quality are closely associated, the former being seen as a measure of the latter. However, no single metric can give a definitive measure of the quality of software and it is difficult to select the set of metrics that give the quality coverage required.

Zuse[1] suggests that some of the obstacles in the selection of such a set include:

1. No best measure.
2. One number cannot express software quality.
3. The properties of the metrics used should be well known.
4. One metric is not sufficient for a whole software system which will consist of many properties.
5. Software quality or maintainability index, produced from a combination of metrics has to be carefully considered. It may make technological sense, but not be meaningful in a qualitative sense.
6. The software environment for every company is different and may require particular metrics for its specific needs.
7. There is no correct initial set or indeed final set of metrics – it is dependent on the needs of the company.

In any organization, it is necessary to have a set of metrics that are well understood, clearly defined, and unambiguous.

ISO9000-3 points out that “there are no universally accepted measures of quality”. However, it does provide some guidance as to what software metrics are appropriate to each phase of the software lifecycle.

For example:

In the requirements phase, suitable metrics may be:

- Number of users interviewed per job function and category.
- Number of errors traced back to the Requirements phase.
- Number of ‘shalls’ (the software shall….).

While for planning, software size and resource usage are more relevant. In the design phase, software size is again useful.

In the coding phase, there are multiple metrics available, which will be discussed later.

Suitable metrics in the testing phase are:

- Number of test cases run
- Number of test failures

And finally in the maintenance phase, the most widely used metric is Number of Defects found after Release, but other common metrics are:

- Number of Change Requests.
- Time to Identify and Correct Defects.
- Defect Density.

It is important to have a reason for collecting software metrics, they are not an end in themselves. So the initial step should be to define a measurable goal and then attribute the metric that measures progress towards that goal: Number of defects found at each stage in the development cycle.

Each software metric quantifies a characteristic of the software, and each characteristic should be capable of yielding a number of measureable properties. So when selecting metrics it should be ensured that they are quantifiable and measure one or more of the characteristics that are to be examined.

For example, if the goal is to ship defect free software, metrics would be needed to measure:

- Identification of the stage from which the defect originates.
- Number of open defect reports.

Metric selection should be practical, realistic and pragmatic, taking into consideration the process in place. Additionally, the cost of gathering the metrics versus the benefit gained from them should be a factor in the selection. Metrics can be produced at all stages of the software development cycle. Their use early in the cycle; for example in the requirements and design phases; can avert future quality issues.

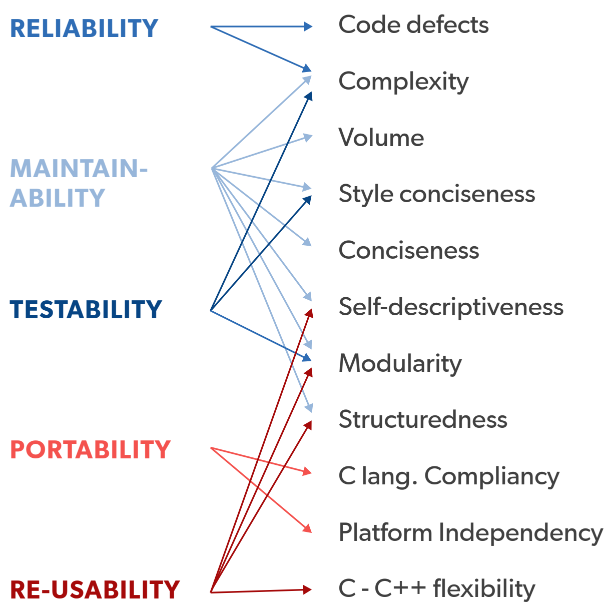
The coding phase is probably the area where there are the most metrics available and this is where the remainder of the paper will focus. The appropriate metrics can depend on the viewpoint taken – for example a manager's view may be different from that of the developer.

ISO 9001 suggests the following for measures as a good starting point for the measurement of code quality:

Characteristic	Measure
Structure/Architecture	Logic Complexity Size
Maintainability	Correlation of Complexity/Size
Reusability	Correlation of Complexity/Size
Internal Documentation	Comment Percentage
External Documentation	Readability Index

These measures are not a definitive list and can be changed or added to – perhaps it is better to use data coupling and functional cohesion rather than correlation of complexity/size. What is important is to place software metrics in the context of continuous improvement. In order to achieve this, a well defined quality model is needed with the desired characteristics and the metrics needed to measure them.

Below is an example of a Quality Model with the software metrics which a related to each characteristic.



No. Major problems
Static Path Count
Cyclomatic Complexity
Nesting of Control Structures
Meyers Interval
No. Maint Lines of code
No. Stylistic warnings
No. Maintenance warnings
Halstead's index
No. of unused variables
Comment density
Program Difficulty
Estimated function coupling
No. external var's declared
No. Function calls
No. Backward jumps
No. Goto's
Knot Count
No. Obsolescene warnings
No. ANSI warnings
No. syntax errors and ANSI constraints
No. Portability warnings
Mobility
No. C++ warnings

Well Known Metrics

So let us take a closer look at some of the more well known metrics used and consider their value. It is strange to note that the most used and discussed metrics are LOC (lines of code) and the Measures of McCabe[2].

These were first defined over 40 years ago, and the question of whether the Measure of McCabe is a good metric or a bad one is still much discussed. Which is why there are many papers that cover this in great detail (for example Myers[3], Gill & Kemerer[4], Goodman[5]).

However, the fact remains that LOC is used as the basis for other metrics.

LOC has been used for many years to evaluate the size of a software system or the effort to write code. It is still widely used as it is easy to evaluate and can be useful if applied consistently. Its basic assumption is that the larger the software system, the more difficult it will be to understand and maintain.

Due to its simplicity, it has been the subject of harsh criticism[6] – mainly because it does not really measure the systems functions and features. There is no industry standard for counting lines of code, so how is a line of code defined? Is it a physical line or a logical line? Is a comment a line of code? A blank line? And what about C/C++ macros?

This section of code has 1 line of code, but 2 logical lines and a comment

```
For (i=0; i<10; i++) printf(“hello”);
/* print hello 10 times */
```

To make any use of LOC, a line of code has to be precisely defined.

As suggested above, it can be useful to determine the number of comments in code to help determine the readability and therefore the maintainability of code. However, once defined within an organization, trend information for LOC can be gathered from various projects over a period of years and used in future planning.

LOC is a size metric, another simple alternative could be number of functions.

A more complex alternative is Function point analysis as defined by A.J. Albrecht[7]. This estimates the size of a software product by using the weighted sums of five different factors relating to user requirements.

These factors are:

- Inputs
- Outputs
- Logic Files (or master files)
- Inquiries
- Interfaces

Functions points are counted by first tallying the number of each type of factor. These totals are then adjusted by applying complexity measures to each type of function point. The sum of all the adjusted function points becomes the total adjusted function point count.

This method is widely used across the world and is a good example for a pragmatically created software metric, However, although the validity of this metric has been shown by experiment, there are many unclear factors in the method. The complexity factors are arbitrary, subjective and based on the developer’s judgment and there is no empirical evidence to indicate which values should be used.

However, as with LOC, if an organization defines and consistently applies its own standards, useful trend information can be gathered and comparisons made between project and products. Function points are also an early indicator of rising/excessive complexity in the software lifecycle yielding more timely corrective action.

A separate group of software metrics are control flow metrics. These are based on the premise that the more complicated the control flow, the most complex the program. The most common of these is cyclomatic complexity which is the count of the number of linearly independent paths through the source code.

For example, if the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, as there is only a single path through the code.

Similarly, if the code had a single IF statement containing a single condition there would be two paths through the code, one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE, resulting in a complexity of 2.

Cyclomatic complexity can be used in two ways:

1. To limit the complexity of code.
2. To determine the number of test cases necessary to thoroughly test it.

However, this metric is a great source of controversy regarding its usefulness in finding defects. It is claimed by Les Hatton[8] that cyclomatic complexity has the same prediction ability as lines of code.

McCabe suggested that 10 is a good limiting factor for cyclomatic complexity and this number is certainly the most widely used. However, this simple value is not sufficient and it is more useful to consider the risk associate with the value. Below are the standard values of cyclomatic complexity.

Cyclomatic Complexity	Risk
1-10	Low
11-20	Moderate
21-50	High
51+	Very High (May be untestable)

In his 1997 book[9], Fenton shows that size metrics like LOC are closely correlated to complexity metrics. Therefore both are reasonable predictors of the absolute number of faults but are very poor predictors of fault density.

However, used earlier in the software lifecycle (e.g. planning) it is easier to estimate so that it is possible to plan for the quality challenges that may be met in the future.

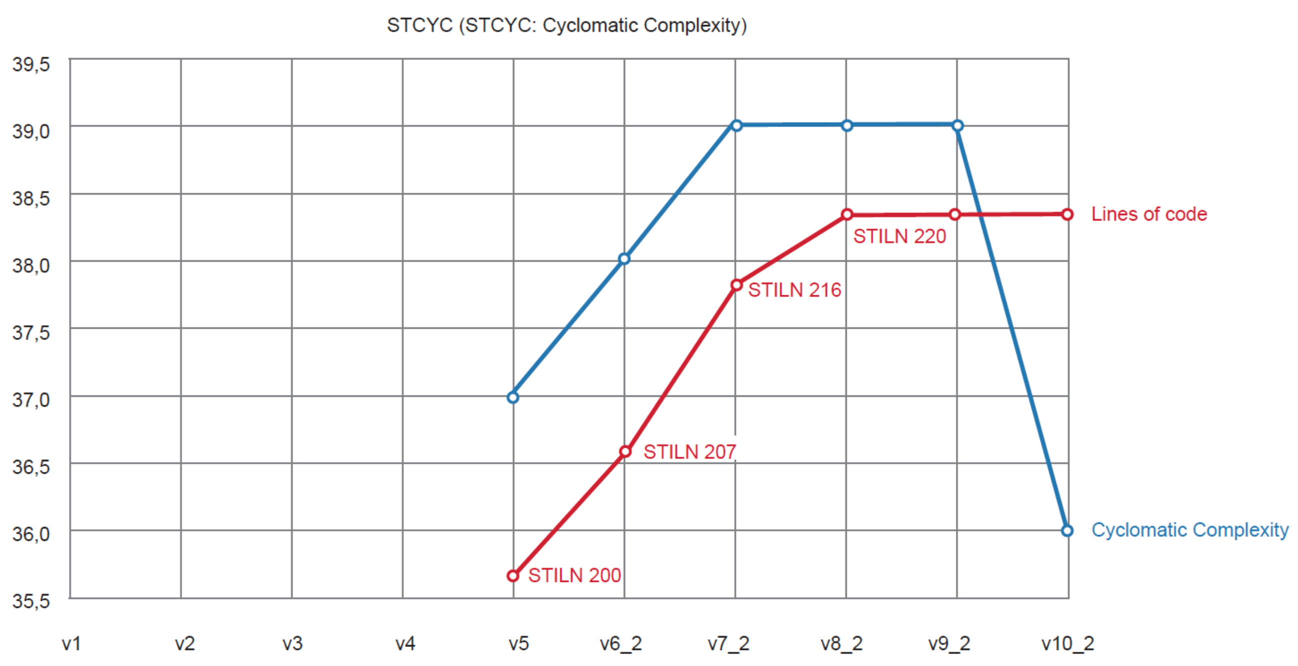
Trending

Following the trend of metrics through software development is a useful way of monitoring the quality of software produced. To obtain the most value from software metrics, it is necessary to be able to observe trends over subsequent versions. Thus it is important that they are gathered in a consistent manner and the results are reproducible.

Here a quality capture tool; such as Helix QAC and Klocwork, is beneficial as capture tools have a comprehensive metric analysis capability for successive versions of code project. The metrics from previous versions are stored to allow trends to be easily observed in graphical format.

The following diagram shows how, with what appears to be a major change in code, the cyclomatic complexity doesn't increase greatly. Therefore if the cyclomatic complexity does suddenly increase, this should be treated as a danger signal. Trending of metric values over a sequence of project history is possible, at both a project top-level and at a granular function/file/class level ("drill-down trending").

Metric Trending Graph for Cyclomatic Complexity and LOC vs Release Version



Cyclomatic Complexity and LOC for function 'retrieve_tree'

The following diagram shows how, with what appears to be a major change in code, the cyclomatic complexity doesn't increase greatly. Therefore if the cyclomatic complexity does suddenly increase, this should be treated as a danger signal. Trending of metric values over a sequence of project history is possible, at both a project top-level and at a granular function/file/class level ("drill-down trending").

This function was introduced in version 5 of project (see below). The starting Cyclomatic Complexity value was 37:

```
if (dl_url_file_map && hash_table_contains (dl_url_file_map, url))
    {
            DEBUGP ((“Already downloaded \”%s\”, reusing it from \”%s\”.\n”,
                     url, (char *)hash_table_get (dl_url_file_map, url)));
    }
    else
    {
            int dt = 0;
            char *redirected = NULL;
            int oldrec = opt.recursive;

            opt.recursive = 0;
            status = retrieve_url (url, &file, &redirected, NULL, &dt);
            opt.recursive = oldrec;
```

In Version 6_2 (below), an additional statement has been added. This has increased the Cyclomatic Complexity to 38. (LOC also increases by 1):

```
if (dl_url_file_map && hash_table_contains (dl_url_file_map, url))
    {
            file = xstrdup (hash_table_get (dl_url_file_map, url));
            DEBUGP ((“Already downloaded \”%s\”, reusing it from \”%s\”.\n”,
                    url, file));
            if (string_set_contains (downloaded_html_set, file))
               descend = 1;
    }
    else
    {
            int dt = 0;
            char *redirected = NULL;
            int oldrec = opt.recursive;

            opt.recursive = 0;
            status = retrieve_url (url, &file, &redirected, referer, &dt);
            opt.recursive = oldrec;
```

The code snippet below shows a different section of the function “retrieve_tree”, again for version 6_2 (Cyclomatic Complexity 38):

```
struct url *start_url_parsed = url_parse (start_url, NULL);

/* Enqueue the starting URL. Use start_url_parsed->url rather than just URL so we enqueue the
canonical form of the URL. */

url_enqueue (queue, xstrdup (start_url_parsed->url), NULL, 0);
```

In Version 7_2 a further “if” statement have been added, but it should be noticed that the Cyclomatic Complexity has only increased to 39. (LOC in this case increased by 9 - depending on how a line of code is defined):

```
int up_error_code;
  struct url *start_url_parsed = url_parse (start_url, &up_error_code);
    if (!start_url_parsed)
     {
          logprintf (LOG_NOTQUIET, “%s: %s.\n”, start_url,
                     url_error (up_error_code));
          return URLERROR;
     }
    queue = url_queue_new ();
    blacklist = make_string_hash_table (0);
    /* Enqueue the starting URL. Use start_url_parsed->url rather than just
    URL so we enqueue the canonical form of the URL. */
    url_enqueue (queue, xstrdup (start_url_parsed->url), NULL, 0);
```

This is a section of the same function version 9_2:

```
    xfree_null (referer);
      xfree_null (file);
    }
/* If anything is left of the queue due to a premature exit, free it now. */
    {
      char *d1, *d2;
      int d3, d4;
      while (url_dequeue (queue,
                              (const char **)&d1, (const char **)&d2, &d3, &d4))
    {
    xfree (d1);
      xfree_null (d2);
    }
```

In version 10_2 (below), the cyclomatic complexity has dropped to 36 by the introduction of the macro FREE_MAYBE, whereas LOC has remained the same:

```
    FREE_MAYBE (referer);
      FREE_MAYBE (file);
    }
/* If anything is left of the queue due to a premature exit, free it now. */
    {
    char *d1, *d2;
    int d3, d4;
    while (url_dequeue (queue,
                              (const char **)&d1, (const char **)&d2, &d3, &d4))
    {
    xfree (d1);
      FREE_MAYBE (d2);
    }
```

Industry Standards

Industry standard rulesets such as HIS (Hersteller Initiative Software)10 and ISO/IEC 9126 make provision for the production of software metrics in order to determine the software capability and maturity of suppliers.

In environments that use these rulesets; particularly in the area of large scale embedded development; software metrics are considered as the basis for efficient project and quality management. However, even here there are differences between standards in the way that metrics are considered.

HIS provides a fundamental set of metrics to be used in the evaluation of software together with the acceptable values of those metrics. These metrics comprise two distinct sets:

1. Metrics with limits which generally measure the complexity of the code.
2. Metrics without limits that measure the change in the number of statements in code between versions to give a stability index. (These are pure measured values but must still be documented in every case.)

HIS is purely concerned with the coding phase of the software life cycle. By analysing these metrics, and ensuring that they are within the specified limits, the effort required in the phases following, particularly testing, will be reduced.

For example:

Number of GOTO Statements
This metric is very simple, but it can easily be seen that the higher the number, the more paths through the code so the more difficult the code is to test.

Number of Return Points Within a Function
Good practice dictates that this should be 1 – this improves the maintainability of the function (a function with no specific return is also acceptable).

It should be noted that to produce the stability index, information is required from previous releases regarding number of statements changed, deleted or added.

ISO/IEC 9126 provided a framework for organizations to define a quality model for a software product leaving it up to each organization to specify precisely its own model. This standard lists six quality characteristics:

1. Functionality
2. Reliability
3. Usability
4. Efficiency
5. Maintainability
6. Portability

This standard was superseded in 2011 by ISO/IEC 25010 and two additional characteristics were added:

1. Security
2. Compatibility

All of these characteristics need to be measurable. However, the standards do not give exact metrics to be used. Some metrics which may be appropriate are:

- Total Defects
- Defects at Delivery
- Mean time to Defect

Each of the metrics is divided into three main categories:

- **Internal** – Metrics which do not rely on software execution – static measures.
- **External** – Metrics applicable to running software.
- **Quality in Use** – These metrics only when the final product is used in real conditions.

Ideally, these categories progress incrementally in scope so better internal quality leads to better external quality which in turn leads to better quality in use.

WHITE PAPER
| Which Software Quality Metrics Matter?

PERFORCE

Conclusion

In conclusion, software metrics are what is made of them and the different views of them.

There is no point in producing metrics just because the process says they should be produced. If this is the case, they are a waste of time and resources.

The different metrics should be applicable to the role of the viewer; the manager's view is different to that of the developer. If a metric is not applicable to any role, it should not be produced. Metrics can be expensive to collect, report, and analyze so if no one is using a metric, producing it is a waste of time and money.

However, if the metrics are selected to measure the progress to achieve specific goals, and the data gathered is analyzed and used by the appropriate people, they are invaluable as a measure of progress and current software quality plus as an aid to improvement in the future.

Why You Use Perforce Static Analyzers to Accurately Measure Software Quality Metrics?

Perforce's static code analyzers — Helix QAC and Klocwork — have been trusted for over 30 years to deliver the most accurate and precise results to mission-critical project teams across a variety of industries.

HELIX QAC

Helix QAC is the most accurate code analyzer for C and C++ programming languages. It's certified for functional safety compliance by SGS-TÜV and in ISO 9001|TickIT plus Foundation Level.

Helix QAC is the ideal static code analyzer for rigorous compliance coding standards — such as MISRA and Autosar — required by functional safety standards such as ISO 26262.

KLOCWORK

Klocwork is the most accurate code analyzer supporting C, C++, C#, and Java programming languages. It scales to projects of any size and is very effective within a continuous integration (CI) and DevOps environment. What's more, it's unique connected desktop technology provides developers with the shortest possible analysis times from inside their IDE

Klocwork is the ideal static code analyzer for effectively implementing process improvements.

Regardless of whichever Perforce static code analyzer you may choose, either will help ensure that your code will be safe and secure, reliable, and compliant.

You can see for yourself the benefit that Helix QAC or Klocwork can have on the quality of your code by visiting perforce.com/products/sca/free-static-code-analyzer-trial.

REFERENCES

1. Zuse – The Framework of Software Management, 1997, de Gruyter.
2. McCabe – A Complexity Measure. IEEE Transactions of Software Engineering, Volume SE-2, No. 4 pp 308-320, December 1976.
3. Myers - An Extension to the Cyclomatic Measure of Program Complexity, SIGPLAN Notices, October 1977.
4. Gill & Kemerer, Cyclomatic Complexity Density and Software Maintenance Productivity, IEEE Transactions on Software Engineering, December 1991.
5. Goodman – Software Metrics: Best Practices for Successful IT Management, 2005 – Rothstein Associates Inc.
6. Basili & Hutchens – An Empirical Study of a Complexity Family. IEEE Transactions of Software Engineering, Volume 9, No. 6 pp 664-672, November 1983.
7. Albrecht - Measuring Application Development Productivity, Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California, October 14–17, IBM Corporation (1979), pp. 83–92.
8. Hatton - Keynote at TAIC-PART 2008, Windsor, UK, Sept 2008.
9. Fenton & Pfleeger - Software Metrics: A Rigorous and Practical Approach, 1997, PWS Publishing Co.
10. http://portal.automotive-his.de/images/pdf/SoftwareTest/his-sc-metriken.1.3.1_e.pdf.