Week 11
Quality metrics

"Man is the measure of all things, but it matters greatly what he measures"

-- Stanley Mott[1]

Introduction

Software engineering is about bringing a systematic, predictable process to the production of software that meets the needs of the people that pay us to build it. If we are going to do that, we are going to need to measure many different aspects of our software and that process so that we can make sensible decisions on what to do next. We would like to:

- *Predict* aspects of the product we deliver before we have completed that part.
- *Assess* the product or process to make decisions about what to do next.

The topic of software metrics is a somewhat fraught one; lots of different metrics have been proposed; fewer have proved themselves useful over time. In this chapter, we will discuss a bit of the theory of metrics and discuss metrics that relate specifically to software quality.

Types of quality metrics

From chapter 1, we can recall that quality, as defined in ISO/IEC 9126[2], had a number of different aspects:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Metrics can be collected, more or less easily, relating to all of these. However, in many cases, there is often a trade-off between the validity of the metric and the feasibility of collection. For instance, in an ongoing project, maintainability can be assessed by keeping statistics on the effort (in terms of programmer time, or clock time) required to fix bugs, or add features. But when we are choosing the architecture of the system, or some aspect of the system, we cannot collect those statistics from software that doesn't even exist yet! In such circumstances, we can look for *proxy metrics* - things that we can measure that correlate to the thing that we would like to be able to measure but cannot. We will discuss a number of such proxy metrics.

---

[1] "Stanley Mott" is a character in James A Michener's fascinating novel, *Space*, which fictionalizes the early US space program. Well worth a read.

[2] https://www.iso.org/standard/22749.html

In the context of this unit, we are particularly interested in testing, and metrics about the "goodness" of test suites. There are specific metrics relating to these, some of which relate to the chapter on white box testing, others that we have not mentioned.

Testing metrics

The most commonly collected metrics about testing are coverage metrics, as discussed in previous chapters. Tools to collect basic coverage metrics are widely available in most languages. We have shown the use of *coverage.py*, a coverage module for Python, which can compute statement and branch coverage. Most coverage tools work in a similar way; you instrument your program to collect the coverage data, you run your test suite, you get a report.

Test coverage tools are available for most languages, though the variety of coverage metrics available in any particular tool will vary. Tools that support data flow coverage testing were popular for a time but have become quite rare; tools that support MC/DC coverage are available for the kinds of languages in which safety-critical automotive and aerospace code is written in (in a nutshell, C and C++).

As we have noted, there is a reasonable correlation between relatively simple coverage metrics such as statement and branch coverage, and the effectiveness of test suites. A smaller but still significant correlation remains, even if you factor out test suite size[3]. That is, if you have two test suites with 50 tests in them, the one with higher coverage is likely to detect more faults.

Code coverage is quick to compute, and a useful indicator of test suite quality, although there is no reliable industry-wide model that can tell you *"x% coverage is sufficient to give you reliability"*. As such, simply specifying a coverage level does not allow you to make guarantees about reliability.

If you are tracing your tests back to requirements and/or design specifications, you can also do coverage for black-box testing. Remember that a test with high code coverage still might neglect to test for functionality that *should* be there but is not. Generally, specification coverage *should* be somewhere between *very high* to 100%. However, it is easy to tick a box saying a requirement is tested while only doing a single cursory test; it is no magic bullet.

Mutation analysis

In the white-box testing chapter, we discussed how coverage metrics were evaluated - one way of evaluation is done through mutation analysis. Mutation analysis involves automatically seeding a change into your source code, running the modified code with the test suite, and seeing whether the test suite reports a fault. You repeat this lots of times with different changes to different parts of the code, and eventually get a *mutation score* from 0 to 1.

[3] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE 2014). ACM, New York, NY, USA, 435-445. NB: this paper's results do not actually match its title, in my not so humble opinion as the resident testing expert.

While not perfect, mutation scores are a better indication of whether your test suite is any good than just code coverage. Unfortunately, mutation analysis tools are not widely used. Running a test suite once to collect coverage data tends to be not too expensive but generating thousands of mutants and running the test suite on each of them tends to be problematic. But if the software under test is relatively small and the test suite needs to be evaluated rigorously, it is very useful additional information to have.

There are a number of mutation analysis tools for Java and C/C++, and one for Python 3 called MutPy[4].

Design and code metrics

A second set of interesting proxy metrics can be collected by statically analysing source code (that is, reading it, parsing it, and counting things), or in some cases, design artifacts such as UML class diagrams.

These proxy metrics are potentially useful indicators of a number of things, not just the likelihood of functional correctness. Design and code quality metrics are potentially useful proxies for the maintainability and portability of software.

Code size

One obvious way that you can measure a codebase is to determine how big it is. All other things being equal, for instance, the size of a codebase is going to correlate to the effort required to build it, testing effort required to ensure acceptable quality, and so on.

The first way to measure this is to simply count the lines of code (LOC) in your program. LOC has a very long historical basis as a measure of the size of the software system. It is very easy to measure; as a first attempt, you take your codebase and count the number of newline characters in the source files[5].

However, this simple approach is confounded by several factors - comment lines may be useful, but they tend to be much quicker to write than code and do not add functionality, for a start. Furthermore, the use of blank lines varies a lot between programmers and between projects.

So, unsurprisingly, people have written LOC counters that can provide more detailed counts, tabulating comments, and blank lines separately.

These refinements are useful, but they still don't take into account that code can be trivially split over more or fewer lines depending on the coding style of the programmer concerned; as you can imagine, if programmers are assessed on the number of lines of code produced, among other things it would likely result in a tendency to split functionality over as many lines of code as possible!

People have developed "rules of thumb" about the size of methods - generally, larger methods are much more error-prone than small methods, though preferred thresholds vary.

---

[4] https://pypi.org/project/MutPy/

[5] Exercise - time yourself to write a piece of code to do this in a) Java, and b) Python. Consider which is better suited to ad-hoc scripting!

Halstead's software science metrics

An attempt to provide a set of more accurate metrics, which could be used to predict the difficulty, effort required, and likely correctness of code, was devised by Maurice Halstead. It was widely publicized in 1977 in his ambitious book *Elements of Software Science*[6].

Halstead tried to abstract away the details of programming languages, and based his metrics on the following properties:

η_1 = number of distinct operators.
η_2= number of distinct operands.
N_1= total number of operators.
N_2= total number of operands.

But what the hell is an operator, and operand? That depends on the language, and whatever interpretation your particular metrics calculation tool uses. One C++ metrics tool, CppDepend[7], uses the following definition:

- **operators:** arithmetic ('+'), equality/inequality ('<'), assignment('+='), shift ('>>'), logical ('&&'), and unary ('*') operators. Reserved words for specifying control points ('while') and control infrastructure ('else'), type ('double'), and storage ('extern'). Function calls, array references, etc.
- **operands:** identifiers, literals, labels, and function names. Each literal is treated as a distinct operand.

By contrast, PyMetrics, a tool for Python, uses a rather different definition. Taking a "tokenized" Python script (turning the text into a list of tokens), it splits everything up according to the type of the token - I have added additional explanation in italics, the rest is taken directly from the comments in Pymetrics [5]:

- operators are tokens of type OP *(an arithmetic or logical operator)*, INDENT *(an indentation in the script)*, DEDENT *(a deindention)*, or NEWLINE since these serve the same purpose as braces and semicolon in C/C++, etc.
- operands are not operators or whitespace or comments (this means operands include keywords).

As you can see, a while statement in Python analysed using Pymetrics would be taken as an operand; a while statement in C/C++ analysed by CppDepend would be taken as an operator. Yay, consistency.

Anyway, given these (somewhat less than concrete) definitions, you can calculate the following derived metrics from the number of operators and operands in a piece of code, be it a method, class, module, package, or system:

---

[6] Maurice H. Halstead, Elements of Software Science: Elsevier North-Holland, Inc., 1977. ISBN 0-444-00205-7
[7] http://www.cppdepend.com/Metrics.aspx

Program vocabulary: $\eta = \eta_1 + \eta_2$

Program length: $N = N_1 + N_2$

Calculated program length: $\widehat{N} = \eta_1 \, log_2 \, \eta_1 + \eta_2 \, log_2 \, \eta_2$

Volume: $V = N \, log_2 \, \eta$

Difficulty: $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$

Effort: $E = DV$

Halstead went on to claim that the time required to implement can be calculated using the following formula:

$$T = \frac{E}{18} \text{seconds.}$$

Furthermore, the delivered bugs in the implementation would be:

$$B = \frac{E^{\frac{2}{3}}}{3000}$$

We will come back to the details of these claims later, but for now, I suggest take a sceptic cat's sound advice:

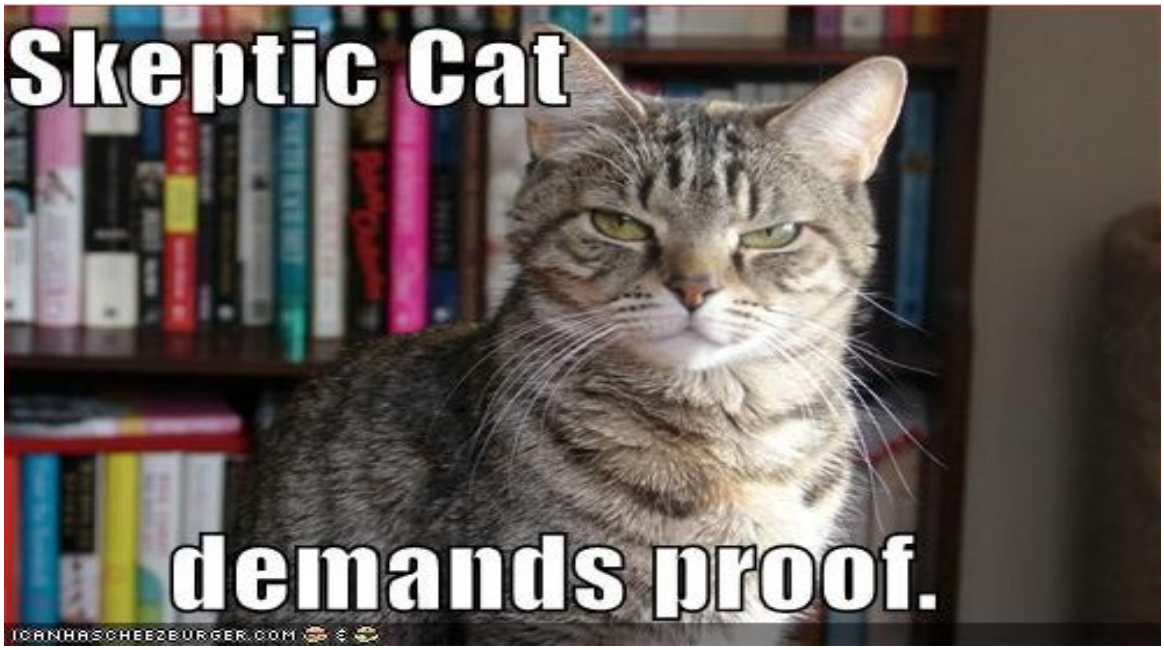

Figure 1: A sceptic cat

McCabe's complexity metric

McCabe cyclomatic complexity metric, which is a way to estimate how "complex" a piece of code is. Remember that we compute the "approximation" of the cyclomatic complexity using the formula:

$$M = \tau + 1,$$

where τ =number of decisions.

There are lots of tables relating *M* to the maintainability and testability of code, with the short rule of thumb being that a method with *M* over 10 to 15 is getting dicey, and anything over about 30 is untestable and unmaintainable.

Cyclomatic complexity can be calculated by using control flow graphs or with respect to functions, modules, methods, or classes within a software program. While using CFGs we can computer McCabe as follows:

$$V(G) = E - N + 2$$

Where E is the number of edges in CFG, and N is the number of nodes in CFG.

Microsoft's maintainability index

Microsoft's Visual Studio 2013 takes this one step further and derives a metric from Halstead's Program Volume and McCabe's Cyclomatic Complexity called the Maintainability Index. According to Visual Studio's 2013 documentation [6], the maintainability index is an:

> *".. index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. Colour coded ratings can be used to quickly identify trouble spots in your code. A green rating is between 20 and 100 and indicates that the code has good maintainability. A yellow rating is between 10 and 19 and indicates that the code is moderately maintainable. A red rating is a rating between 0 and 9 and indicates low maintainability."*[8]

Visual studio can be configured to produce warnings or even errors if methods have low enough maintainability indexes.

Microsoft's formula is based on a formula devised by Paul Oman and Jack Hagemeister[9] in the early 1990s. They asked the maintainers of several projects at Hewlett-Packard to rate the quality and maintainability of their code using a questionnaire, and essentially added up the responses to compute a subjective "maintainability rating" for each project. They then computed a large number of code metrics for each project and tried to find a formula that predicted the human-provided maintainability score based on the code metrics. They came up with the following formula:

---

[8] https://msdn.microsoft.com/en-us/library/bb385914.aspx

[9] Paul Oman, Jack Hagemeister, Construction and testing of polynomials predicting software maintainability, Journal of Systems and Software, Volume 24, Issue 3, March 1994, Pages 251-266

$$MI = 171 - 3.42\ ln\ (avgE) - 0.23\ avgV(G) - 16.2\ ln\ (avgLOC) + 0.99(avgCL)$$

Where *avgE* is the average Halstead effort for each module, *avgV(G)* is the average McCabe complexity for each module, *avgLOC* is the average lines of code, and *avgCL* is the average number of comment lines per module.

Note here that they were not trying to predict the maintainability of individual modules within each system; they were trying to predict the maintainability of the system as a whole.

Microsoft's maintainability index is, by contrast, being used to determine whether an individual class is maintainable enough. They also modified the formula[10], not considering comments and rebasing the index from 0 to 100:

$$MI = max(0, \frac{100 \cdot (171 - 5.2\ ln\ (V) - 0.23(V(G)) - 16.2\ ln\ (LOC))}{171})$$

where V is the Halstead Volume, V(G) is McCabe Complexity, and LOC is lines of code.

OO design metrics

The proxy metrics that we have looked at so far look at code at a micro-level. They do not really appear to look at how pieces of a large system interact with each other. If you paid attention in Software Engineering units such as FIT1010 etc, you will know that the more interaction, and the more complex the interaction, between classes is, the harder code is to write and debug. It also should be obvious that interactions between classes make testing more difficult (for one thing, it is more mocking for unit testing, or more tracing faults down through many different classes).

Chidamber and Kemerer, in a ridiculously widely cited 1994 paper[11], proposed a number of metrics for object-oriented designs. Some of these metrics attempt to look at the bigger picture of how methods and classes interact with each other. While there are other metrics that have been devised for assessing object-oriented software systems, they are, in the main, quite similar to Chidamber and Kemerer's.

Weighted Methods Per Class - WMC

The first metric they proposed was Weighted Methods Per Class (WMC). For a class with methods M_1, M_2, up to M_n, each of which has "complexity" c_1, c_2, c_n, the weighted methods per class is simply the sum of the complexities of the methods:

$$WMC = \sum_{i=1}^{n} c_i$$

[10] http://blogs.msdn.com/b/codeanalysis/archive/2007/11/20/maintainability-index-range-and-meaning.aspx
[11] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng. 20, 6 (June 1994), 476-493. DOI=http://dx.doi.org/10.1109/32.295895

Chidamber and Kemerer deliberately did not define the complexity of a method - there are a number of ways we've already discussed that might be appropriate for this.

In any case, their theory was that this would predict the time and effort required to develop and maintain the class. It would also have some impact on reusability, because classes with large numbers of methods were more likely to be application specific.

Note that if the complexity metric you use requires code to measure, then you cannot calculate WMC without code!

Depth of Inheritance Tree - DIT

The second metric was Depth of Inheritance Tree (DIT). Let us assume you have got a class diagram that looks like this:

The DIT for a class is the number of transitions from a root node (class that has no superclass) to that class. So, the DIT for Object is 0, and the DIT for VerticalLayout is 3.

If the class uses multiple inheritance, the DIT for the class is the *maximum* length path from a root node to the class.

A class with a higher DIT number was thought to make designs more complex (possibly bad, because more complex designs are harder to implement and test correctly) because a class at the bottom of the hierarchy inherits more and more properties, but a high DIT also implies more reuse (which is considered good).

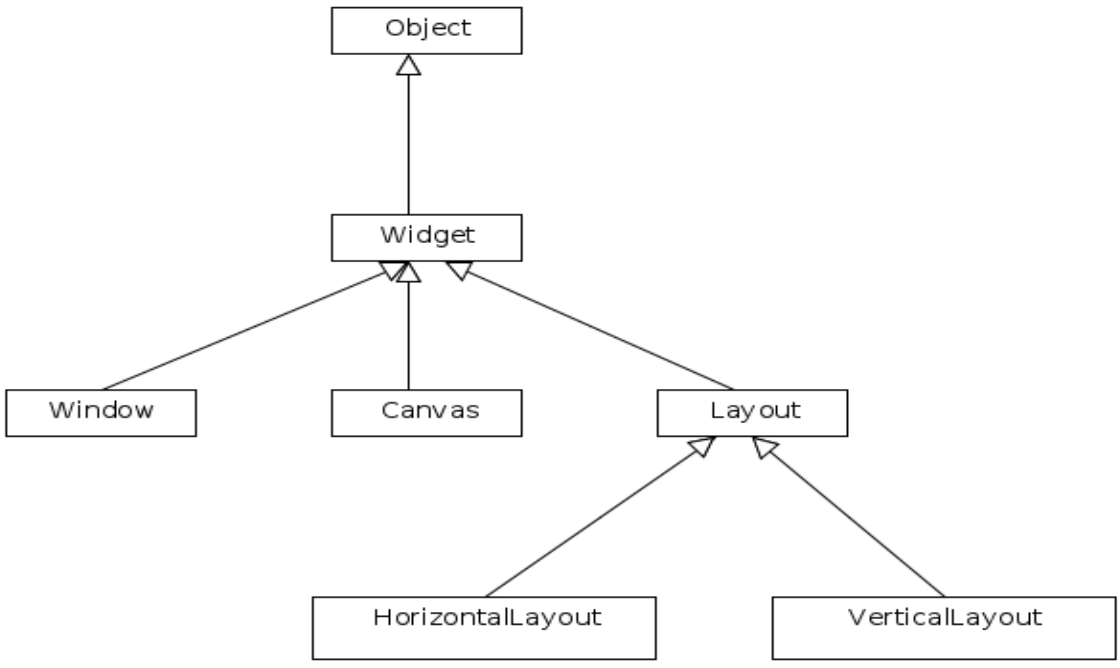

Figure 2: An example of inheritance diagram.

Number of Children - NOC

The third proposed metric for classes was the number of classes inherit from it - the "Number of Children" (NOC). In the class diagram above, Widget has three children, Object one, and leaf nodes like VerticalLayout have zero.

Chidamber and Kemerer thought that a high NOC was indicative of reuse (which is good) but could also indicate an inappropriate abstraction (bad). Furthermore, classes with high numbers of children were thought to require more testing, because it has a high influence over large parts of the design and is likely to be heavily used.

Coupling Between Objects - CBO

The fourth metric, Coupling Between Objects (CBO), is defined for an individual class as the number of other classes it interacts with directly. This includes both method invocations or access to public data fields (if your classes have such monstrosities - public data members are THE BAD THING, at least in languages like Java).

As you should remember, high coupling is bad, because it makes it harder to design, build, test, debug, extend and reuse classes. So, if the CBO metric is a reasonable measure of coupling, a high CBO number for classes is bad.

You do not need a full implementation to calculate CBO - you can do so from class diagrams if they're comprehensive enough. A class with a high CBO has lots of association arrows between it and other classes.

Response For a Class - RFC

The fifth metric, Response For Class (RFC), is defined for a given class C, as the size of the following set: *all the methods in C, and all the methods invoked directly by methods in C.* Chidamber and Kemerer's definition is not transitive - you do not need to worry about the methods invoked in the methods invoked by C, and so on!

In their view, a high RFC meant that the class was more complex and would be more complex to test and debug as the response to a message might involve a large number of methods.

Lack of Cohesion in Methods - LCOM

If you remember coupling, you will remember its twin cohesion, both of which were originally defined by Larry Constantine in the early 1970s[12]. Cohesion is the closeness of the relationship between the elements of the same module. Cohesion is a subjective concept; in Constantine's view, the best type of cohesion, functional cohesion, occurs when a module has a single, clear, unambiguous purpose. However, Chidamber and Kemerer thought the following was a reasonable proxy metric for cohesion:

For a class with methods $\{M_1, M_2, \ldots, M_n\}$, each method M_j has a set $\{I_j\}$ of instance variables it operates on (reads and/or writes). For each *pair* of methods in the class M_a and M_b, determine if any of the instance variables they operate on are the same (That is, I_a and I_b contain at least one shared element). If they share no instance variables add 1 to P, otherwise add 1 to Q

[12] Stevens, W.P.; Myers, G.J.; Constantine, L.L., "Structured design," IBM Systems Journal , vol.13, no.2, pp.115,139, 1974

LCOM for the class is then defined as follows:

LCOM = P - Q, if P>Q, 0 otherwise.

The higher the LCOM metric for a class - that is, the more pairs of methods that don't have any overlap on the instance variables they modify - the less cohesive the class is believed to be. Low cohesion as measured by this metric was believed by Chidamber and Kemerer to lead to code that is less easy to test, debug, maintain, and reuse.

Code/design metrics and quality

The point of collecting code and design metrics is to be able to make decisions about design quality. So, a key question is whether they actually tell us anything to support these decisions.

Halstead metrics, in the main, have not been treated terribly kindly by history and empirical evidence; indeed, the main reason I mention them is their continued use in the maintainability index.

In practice, the relationships between program length, program volume, and lines of code are linear and very strong, as shown in multiple studies[13][14], suggesting the amount of extra information gained from using Halstead metrics is quite limited.

As far as the "Time Required" and "Bugs" metrics, the time taken to implement code varies by an order of magnitude, for the same programming task, as demonstrated in many studies over decades[15]. Halstead's metrics also completely ignore many of the factors that contribute to the variability in programming time. Not least, the size and complexity of the external APIs that a piece of code must interact with, which greatly complicates and slows down programming.

What's more surprising is that McCabe complexity is also extremely tightly correlated with lines of code as close or even more closely related (depending on which study you look at) to lines of code than Halstead volume. There is a strong correlation, but not as strong, between McCabe complexity and Halstead volume[16].

As such, if McCabe complexity measures complexity, it seems that complexity grows linearly with code size as measured by LOC.

---

[13] Yahya Tashtoush, Mohammed Al-Maolegi, Bassam Arkok, The Correlation among Software Complexity Metrics with Case Study, arxiv.org preprint, http://arxiv.org/pdf/1408.4523.pdf

[14] van der Meulen, M.J.P.; Revilla, M.A., "Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs," Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on, vol., no., pp.203,208, 5-9 Nov. 2007

[15] Bill Curtis, Substantiating Programmer Variability, Proceedings of the IEEE, vol 69, no 7, July 1981

[16] Graylin Jay, Joanne E Hale, Randy K. Smith, David Hale, Nicholas A Kraft, and Charles Ward, Cyclomatic Complexity and Lines of Code: Journal of Software Engineering and Applications, 2009, 2: 137-143

Fault prediction

A key quality property that would be nice to be able to predict is functional correctness. Not only do you want your code to be functionally correct, but if you have modules that are predicted to be buggy, they are the ones that you will want to test most intensively!

When you have a bunch of measurements of your code, and you want to predict something about it, what academics tend to do is feed all those measurements into some kind of model generator using machine learning techniques (or to use the latest buzzword, “Big Data” - it's so big, it has capital letters in its name). In a nutshell, if you're trying to build a software fault prediction model, you take some training examples (systems for which you know both the number of faults, as well as the values of all the metrics you’re using), feed that into your model in order to "train" it. Then you use that trained model to predict whether other code has faults.

Lots of people have tried to do this using lots of different metrics, including:

- Static source code measures like the ones we have been discussing.
- Software process metrics such as the number of bugs already found in modules.
- Personnel-based information such as who has been working on the modules.

These are interesting *both* in that it would ultimately be nice to be able to apply fault prediction models industrially, and they can give some hints as to what to look for when you're applying metrics on your own.

In a review of nearly 200 studies of fault prediction models, Hall et al.[17] made some interesting general observations about the usefulness of these static metrics:

> *"...On the other hand, models using only static code metrics (typically complexity based) perform relatively poorly. Model performance does not seem to be improved by combining these metrics with OO metrics. Models seem to perform better using only OO metrics rather than only source code metrics. However, models using only LOC seem to perform just as well as those using only OO metrics and better than those models only using source code metrics."*

That is, if you want to go looking for modules with lots of bugs in them, the best place to look are the ones with the most lines of code. Beyond that, we do not have good general guidelines.

Maintainability

From a quality standpoint, functional correctness is not the only thing we care about; we also care about the ability to maintain and extend the code we write.

You will recall that the maintainability index was specifically designed for this purpose. So - does it predict maintainability?

---

[17] Hall, T.; Beecham, S.; Bowes, D.; Gray, D.; Counsell, S., "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," Software Engineering, IEEE Transactions on , vol.38, no.6, pp.1276,1304, Nov.-Dec. 2012

Its proponents at HP certainly thought so. In one article, they describe applying the maintainability index to other pieces of code lying around HP; they found that high maintainability indices matched code that maintainers approved of, whereas low maintainability ratings was related to hard-to-maintain code[18]. Furthermore, they found this relationship was a useful rule of thumb for individual modules, not just whole systems made up of many modules:

> *The figure shows that 364 files, or roughly 50 percent of the system, fall below the quality-cutoff index, strongly suggesting that this system is difficult to modify and maintain. Prior to our analysis, the HP maintenance engineers had stated that the system was very difficult to maintain and modify. Further analysis proved of - 91. Table 3. A polynomial comparison of two systems corroborated an informal evaluation by engineers. A that change-prone and defect-prone subsystem components (files) could be targeted using the ranked order of the maintainability indices.*

That is interesting. However, there are two obvious issues:

- Do developer's *perceptions* of maintainability relate to actual quantifiable maintenance properties - in a business context, does it relate to time/cost to maintain?
- Does a metric originally devised based on C code decades ago still apply to code written in OO languages such as Java, Python and so on?

There is less information in the software engineering literature on these questions than I would like - lots of studies, but lots of them compare proxy metrics with other proxy metrics. However, one study that *does* apply to relatively modern code in modern languages, and correlates to actual effort required to make changes to code, was performed by Sjoberg, Anda, and Mockus at the University of Oslo[19]. They compared four web-based information systems, which were commercially implemented based on the *same* specification by four different companies. They calculated several maintainability metrics for each of these four systems, including maintainability index, variants of the OO design metrics we described earlier, and two we haven't discussed - the number of two different types of "code smell" per thousand lines of code.

They found that *none* of the maintainability metrics were good predictors of the actual amount of time required to make a change to the code:

---

[18] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. 1994. Using Metrics to Evaluate Software System Maintainability. Computer 27, 8 (August 1994), 44-49.

[19] Dag I.K. Sjøberg, Bente Anda, and Audris Mockus. 2012. Questioning software maintenance metrics: a comparative case study. In Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM '12). ACM, New York, NY, USA, 107-110.

Table 1. Maintenance metrics and a maintenance study applied to the four systems
Legend: green indicates the best system and red the worst one

Category	Metrics	System A	System B	System C	System D
Size	Number of Java files	63	168	29	119
	Java lines of code (LOC)	8205	26679	4983	9960
Maintainability Index (MI)	LOC, # of comments, cyclomatic complexity, Halstead's volume	113	117	114	120
Structural measures (SM)	Coupling (OMMIC)	7.7	5.3	8.6	4.7
	Cohesion (TCC)	0.26	0.17	0.20	0.11
	Size of classes (WMC1)	6.9	7.8	11.4	4.9
	Depth of inheritance tree (DIT)	0.46	0.75	0	0.83
Code smells	Feature Envy (# per KLOC of code)	4.51	1.27	3.41	2.51
	God Class (# per KLOC of code)	0.12	0.19	0.60	0.20
Study of actual maintainability	Average effort (hours)	18	33	13	23
	Predictor of quality (avg. # changes)	148	125	76	124

Figure 3: Results of a study by Sjoberg, Anda, and Mockus. Results show a lack of a relationship between the maintainability measured in effort (hours) and a number of common maintainability metrics, for four systems with identical specs by different authors.

The results were all over the place. In fact, based on this small study, the only metric that seems like a useful predictor of maintenance effort is … lines of code! The smallest system was considerably easier to maintain than the larger ones.

As they observe, this is hardly definitive, given it's one study of relatively small code bases. They state:

> *It is possible that metrics apart from size may play a role in reducing maintenance effort in large projects where it takes a long time (> 3 years) for developers to become fluent [19], but we see no evidence that they matter in our context.*

More broadly, in a systematic review of maintainability prediction models[20], Riaz, Mendes and Tempero were not overly impressed with the effectiveness of the models developed to date. While the individual studies they examined indicated that a broad range of metrics were somehow connected with maintainability (they found that the most successful predictors related to "size, complexity and coupling") they concluded that the results were not convincing enough for broad applicability:

> *The results of the SR suggest that there are no obvious choices for maintainability prediction models. Of the models proposed in the literature, few are supported with accuracy measures, few use any form of cross-validation, and few have evidence for external validity.*

Rules of thumb and BYO models

So where does that leave us? If you ask an experienced software engineer to examine an artifact - source code, design documents, even requirements - most are able to give an assessment of whether it's "up to snuff". But we're still a long, long way from being able to feed a set of artifacts into a

[20] Riaz, Mehwish; Mendes, E.; Tempero, E., "A systematic review of software maintainability prediction and metrics," Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, vol., no., pp.367,377, 15-16 Oct. 2009 doi: 10.1109/ESEM.2009.5314233

computer program you can simply download from the internet and draw strong conclusions on whether they will result in a high-quality piece of software.

My own opinion is that part of the problem is that we have passed on at least some of the easy lessons about writing good-quality software already. The kinds of blatant design mistakes that a computer can identify do not get made very often in industrial systems. For instance, the maintainability indices of the four systems shown in figure 3 were around 120, the equivalent of about 70 on Microsoft's scale. That's way, way above the 20 threshold that Microsoft uses as a "red flag".

For what it is worth, very anomalous metrics - that "red flag" on the maintainability index - are probably useful indicators of potential quality problems, including correctness and maintainability. Observe that what this is likely to come down to is the presence of overly large methods! Similarly, indications of overly close coupling between classes are likely indicative of a design problem and is something you can probably spot at the design stage.

Even though it might be difficult to make *general* recommendations from metrics, they can still be worth measuring and recording. Collecting metrics using automated tools from source code is an extremely cheap process. If you do that across a large project, or multiple projects in an organization *you* can look for your own patterns connecting particular "proxy metrics" with the things you ultimately want to achieve.

How? By keeping metrics as the product is in development, and seeing how this relates to what you ultimately care about, which might include:

- Number and severity of failures reported *after* release.
- Time required to fix bugs.
- Time required to add new features.
- Number of code changes required to add new features.

Over time, you might be able to see what kind of relationship there is between pre-release metrics and post-release actual behaviour and make decisions for future work based on that.

Be careful! Analysis of statistical data has a number of traps for the unwary. To give a very simple example, say you collect 50 source code metrics, and look for ones that have some relationship to - say - time required to fix bugs. The way statistics works, even if *none* of the metrics has a real predictive relationship with fixing bugs, one or two, just by random chance, will probably show as having a "statistically significant" relationship by the standards usually used in scientific publishing (and used by other statisticians as default). There are standard ways to correct for this - but you need statistical expertise or the advice from a statistics expert to avoid this kind of problem.

So, if you're trying to find this kind of a relationship, learn a bit more about statistics!

Even if there turns out to be valid relationships between certain code properties and quality, you can run into traps putting this knowledge to use. I'd like to end with some thoughts on what you should

do with *any* metric you use to assess any aspect of the software development process, with a little detour into the management theory of a century ago.

Taylorism

The idea of managing the behaviour of workers producing something by systematic measurement was (to simplify a little) popularized by Fredrick Winslow Taylor in the early twentieth century. His ideas are summarized in his book, Principles of Scientific Management[21]. Many of the principles espoused in it are echoed in software engineering today, particularly the more rigid, process-oriented types. In short, Taylor espoused the following:

- Based on empirical measurement, develop a detailed "science" for each worker's job - essentially, a manual describing in detail exactly how they should do their work as efficiently as possible.
- "Scientifically" choose the best people for the job, and train them fully in the methodology.
- Work with employees to ensure that they follow this "scientific" process precisely. This would involve incentivising them to a) follow the process, and b) rewarding them for maximising the management's desired outputs, which must be numerically measurable.
- Take all of the thinking out of the worker's job and hand it to management.

Taylor illustrates his principles with a not-so-charmingly racist and classist story about a labourer named Schmidt who was taught and incentivized to load "pig iron" into a rail car at approximately four times the average rate previously achieved, at the cost of paying him roughly 50% more.

While Taylor's carefully chosen example seems to have worked rather well, in the real world, measuring desirable outputs is not always so easy. For instance, consider this little anecdote, where the desired output, is, well, less output:

> ### *Toilet training*
>
> *Training pants have little pictures that disappear if "accidents" occur. This gave us a visible and external monitoring device. It was viewable both to us and our son. So that is the first thing we would check in the morning, and a celebration would ensue if the pictures were still there.*
>
> *As you can guess, celebrations only get you so far. So, as has been our pattern throughout all of this, we moved to more tangible rewards. He was old enough to understand a points system that would lead to rewards. So a dry night would get a point, and seven points would get you a reward--usually a book or a toy. This was sufficient motivation, and he had a clear focus: "Make sure the pictures don't go out and you get a point."*
>
> *Well we had some good nights and intermittent accidents. But then we had a full week of dry training pants. Much rejoicing ensured, including a bonus; no more*

[21] Available from Project Gutenberg or downloadable for free from the Kindle Store - not suggesting you read it all, but it is there if you're interested.

training pants. Sadly, the next night was an accident. "These things happen," you might say. But it turns out, the problem was that these things hadn't happened.

Our son had a small rubbish bin in his room. Upon inspection, we found five full training pants. Child No. 2 was getting up in the morning, noticing the pictures were gone, and getting himself a new pair of training pants! -- Joshua Gans [20].

Misusing metrics

Inappropriate metrics, and making inappropriate decisions based on them, has unfortunate outcomes for parenting toddlers.

Can this kind of thing happen in software engineering? Of course, it can. As is often the case, Dilbert, Lisa and Wally illustrate nicely:

Figure 4: Dilbert by Scott Adams, 13/11/95.

Software is an extraordinarily complex thing, with many quality properties, some easy to quantify and some not. Reducing assessments of software to the things that you can easily measure runs the risk that other important factors will be ignored. And if you direct - and possibly incentivize - a development team to concentrate on one of these easily measured indicators, you can very easily cause this problem.