Introduction

There are various means of measuring the software engineering process. Recently many academic advancements have been made in the area of software metrics, following the popularisation of the subject. However, for a multitude of reasons many of these academic developments have occurred in isolation of the software industry, due to a disparity of interests and motivations. This report will look at the problems associated with some of the common metrics in use in the private industry and proffer a model for solution. From Lines Of Code metrics to Bayesian Belief Nets, there are varying degrees of complexity to consider, as well as the importance of understanding the underlying factors for why the software industry persists in applying dated metrics when we stand as proponents of the next shiny big deal that will definitely, definitely make you lots of lovely money this time.

Metrics and measurements

The goal of measuring the software engineering process is deceptive, as it is not truly programmers we are measuring, but their productivity, which can be defined as the ratio of output to inputs used in a production process. In software engineering, the input can be reduced to the money paid to the programmer without too many problems (depressing isn't it?), but defining the output is a trickier task. Measuring sheer volume of code produced is obviously insufficient, so plainly we are interested in quality of code produced as well (for completeness, there are other factors to consider such as the effect the programmer has on the output of his team etc. Matters which may not be so easily quantifiable, but for the sake of this report we will focus on the problem of measuring the code produced), and so the issue of measuring a programmer quickly centres on the issue of measuring code.

Software metrics has recently been incorporated into the mainstream of Computer Science, and there are now roughly as many books on the subject of software metrics as there are on the general topic of software engineering. This has led to an explosion in the area of academic development in software metrics. Keyword, however, being ‘academic’. The software industry has not been so quick to adopt the fruits of academic labours, and not for reasons of dull-headed bloody-mindedness either.

Much of the research performed has been inherently irrelevant to the needs of the industry. Academia has focused on metrics which can only be applied to small programs, while all industrial motivations for using software metrics are to apply them on large system, and many models rely on data which cannot be computed in reality. Aside from issues of scope, academic metrics tend to focus on properties that are of little practical importance. The two sectors of development in software appear to be largely irrelevant to each other.

Lack of industrial enthusiasm for modern software metrics can also be put down to poor incentive to adopt. It not at all apparent that taking on software metrics is a good idea from a financial point of view. There are few stories of long-term pay-back, which is really what adopters of such systems will be motivated by, and operating with a software metric adds an overhead of between 4% and 8%. It is effective as a management tool, but only with the active involvement and commitment of technical staff in development and testing, making it a costly-venture, difficult to implement, and the first corner to be cut when deadlines come rattling at the door.

The metrics that have been adopted in the software industry are largely ineffectual, having been known to be invalid twenty years ago. Many collect defect data that does not differentiate between defects found in operation (failures) and defects found in development (faults).

Before we begin looking at specific metrics, it may be a good idea to identify common measurements used to extract information about programs, programmers, and thereby the software engineering process. One of the most basic and common measurements is Lines Of Code (LOC), as the name suggests, this simply tracks the number of lines of code the programmer has written. The term KLOC is used to represent a thousand Lines Of Code. These measurements both track quantity, while quality is often determined by defects per KLOC, the computation of which is rather more complex (if desired to be). These are surrogate measures for effort, functionality, and efficiency.

Keeping in mind the state of the software metrics in the private industry, we can judge which metrics have been a relative success under the criteria of being widely used across a section of the software industry:

- The LOC metric: though many convincing arguments of this metric being a very poor ‘size’ measure oppose its use, it is still used as a normalising measure for software quality (defects per KLOC), as a means on deciding productivity (LOC/programmer/month), and as a means of providing (exceedingly) crude estimates of cost/effort. Despite obvious deficiencies, this metric is easy to compute and visualise, resulting in its widespread use and success.
- Metrics relating to defect count: almost all industrial metrics programs attempt this to some degree. Often not recognised as software metrics programs as they are regarded as simply applying good programming practice and configuration management.
- McCabe’s Cyclomatic Number: faces many criticisms but remains widely popular. It is easily computed by static analysis and commonly used for quality control. It involves measuring complexity by the number of linear paths through a program’s source code. It was said that a Hewlett and Packard, modules with a McCabe’s Cyclomatic Number of greater than sixteen would have to be redesigned.
- Function points: very difficult to properly implement and unnecessarily complex if used for cost/effort estimation. Still experiences common use in the financial IT sector.

So ends the rather unimpressive list, however, we will see later that it is the models and application that have been fundamentally flawed rather than the metrics.

Difficulties in metrics predicting software quality

A belief has permeated software development that ‘There is a clear intuitive basis for believing that complex programs have more faults in them than simple programs’ [successes, failures and new directions Norman E. Fenton*, Martin Neil]. This was found not only to be unfounded, but there was in fact evident supporting the converse. Almost all faults discovered in pre-release testing appear in modules which thereafter revealed almost no operational failures. Most operational failures are caused by a small number of latent faults; just because there are a high number of failures does not necessarily mean that there are a high number of bugs or that the code in general is of poor quality. If fault density is measured pre-release, as is the norm, it tells us nothing about the quality of the code at module level; a high fault density value is more likely to be the result of vigorous testing as it is to be the fault of sub-standard code. The complexity metrics are closely related to LOC, and are reasonable predictors of absolute fault numbers, but very poor predictors of fault density.

AS well as the problem of using metrics in isolation, another major weakness of the common simplistic approach to defect prediction has been a misunderstanding of cause and effect; that correlation does not equal causation. For example, few faults may have found in development, but this may have been due to weak testing, which many faults may be the result of thorough testing. These results cannot be taken as a strong indication of failures to come in operation. The association between size and fault density is not causal. It is important to augment with testing effort and operational usage, as if a module is not tested or use, faults and failures won’t be noticed.

Following this, it is apparent we need models which can handle diverse process and product evidence, genuine cause and effect relationships, uncertainty, and incomplete information, without introducing large software metric overheads.

According to Norman E. Fenton and Martin Neil, Bayesian Belief Nets (BBN) are by fat the best solution to the problem. They have received attention in the past as a solution to problems of decision support under uncertainty. Bayesian probability has been around for a long time, but building and executing models has only recently become possible due to developments in algorithms and software tools to implement them. With a Bayesian belief net it is possible to propagate consistently the impact of evidence on the probabilities of uncertain outcomes. This sounds like gibberish (to me at least), but what it means is supposing we have evidence of the number of defects found in testing. When we enter this evidence to the BBN all the probabilities in the entire net are updated. From this, the BBN's probability tables are a mixture of empirical data and subjective judgements.

The benefits of using the BBN model include:

- Explicit modelling of 'ignorance' and uncertainty in estimates, as well as cause-effect relationships
- Make explicit those assumptions that were previously hidden, thereby adding visibility and auditability to the decision-making process.
- Intuitive graphical form allows for easy understanding of complex and seemingly contradictory reasoning.
- Ability to forecast with missing data.
- Use of 'what-if?' analysis and forecasting of effect of process changes.
- Use of subjectively or objectively derived probability distributions.
- Rigorous, mathematical semantics.
- No need to do any of the complex Bayesian calculations, since tools like Hugin do this.

Ethical Concerns

There are a number of ethical concerns on the subject of software development such as consent and confidentiality. The question of how invasive the tracking of employees by companies is legally allowed to be can be raised. Is the tracking of social interactions to determine which teams are linked to each overstepping the mark? Are out of hours activities of valid interest to an employer? For instance, would an employer have a right to know if an employee was consistently coming into work without sleeping the night before, or hungover, or insufficiently fed etc. All of these factors are of some degree of importance when determining the efficiency of an employee, but clearly there is a line in the sand drawn by the employee as to which information they are happy to give and which they are not. It is clearly important for consent to be obtained while tracking employees, and also equally clear that an employee failing to give it may affect their job prospects. On the matter of confidentiality, should a company acquire data on a person, actions should be taken to preserve the anonymity of the person in question, such as storing the idea with reference to an ID number rather than the name and address, and ensuring that the data is secure.

Conclusion

Though the past and present of software metrics in the private sector is bleak, there is hope for it yet. While simplistic and dated metrics which are easy to compute and visualise, and demand little overhead cost, have thus far ruled the day as token gestures begrudgingly bought, the future of the discipline seems brighter as more ambitious models such as Bayesian Belief Nets come to the fore. Though great enough in scope and complex enough in content to offer a solution to the needs of the software industry, tools such as Hugin offer to abstract the complicated calculations from its use, allowing for smoother adoption.

Sources:

- Fenton, N. E., and Martin, N. (1999) "Software metrics: successes, failures and new directions." Journal of Systems and Software 47.2 pp. 149-157.
- https://en.wikipedia.org/wiki/Bayesian_Network

- https://en.wikipedia.org/wiki/Cyclomatic_complexity
- https://en.wikipedia.org/wiki/Programming_productivity#Factors_influencing_programming_productivity
- Ethical Issues in Empirical Studies of Software Engineering: Janice Singer and Norman G. Vinson