A Framework for Effective Software Monitoring in Project Management

[1]Wemembu, Uchenna Raphael, [2]Okonta, Okechukwu Emmanuel, [3]Ojugo, Arnold Adimabua
[4]Okonta, Imah Love

.

[1]Department of Mathematics Federal College of Education. (Tech) Asaba.
[2] Department of Computer Science Federal College of Education. (Tech) Asaba
[3] Department of Math/Computer Science, Fed University Petroleum Resources Effurun
[4] Department of Technical Drawing Federal College of Education (Tech) Asaba.

Abstract

Developed software for project management rely heavily on collecting metrics to provide the progress feedback necessary to allow control of the project. However, interpretation of this data is very difficult and sometimes cumbersome. This paper addressed the need of a software implementation progress model that is needed to help interpret the accumulated data. Certain criteria are set for design of a proposed implementation progress model. Some findings from the studied projects from other researchers suggest the model is consistent with the observed behaviour. In addition to quantitative validity, the model is shown to provide meaningful interpretation of collected metric data by embedding certain quality function.

Key words: Project Management, Feedback, project control, metrics, process model, quantitative validity

---

Introduction

Developing software for effective project management in modern times rely heavily on periodically collected software metric data and this in turn used to provide management with feedback about the project, the process used and the stages in development. Metric data is most commonly used in the area of quality assessment and assurance. Well-defined metrics usually provide report on quality attributes such as anticipated number of faults remaining. But other areas, such as implementation progress and system throughput, hardly utilize feedback from Metrics data. Although, the total lines of source code, could be used to report on implementation progress but such measures have not been leveraged as strongly as quality assessment and assurance have been used.

Metric data is widely regarded as a valuable management feedback tool, yet it is generally not used to monitor implementation progress. An implementation progress model is presented and shown to identify project phase boundaries, express the rate of implementation during each phase, and allow objective comparisons between projects. This paper provides a framework to help interpret periodically collected implementation data. This work develops a model for interpreting implementation progress. The proposed progress monitoring model uses existing implementation artefact metrics, tries to match our understanding of the implementation progress, and allows project evaluation based on estimation of parameter used. Well-defined and proven metrics exist for many areas of software

engineering and development including especially quality assessment and assurance. Implementation progress has no such established metrics.

The lack of proven implementation progress metrics has been a barrier to any attempt to effectively monitor project development. Nevertheless, this shortcoming is not insurmountable. Size metrics are abundant and deriving a progress metric from a size metric can be accomplished by working out the difference in consecutive size measurements. A far larger barrier than the lack of a metric is the lack of a proven implementation progress *model*. Periodically collected data is rich in detailed information but is not in itself meaningful. A model provides a specific interpretation of the data and allows meaning to be extracted. An implementation progress monitoring model will allow periodically collected implementation data to be effectively interpreted.

Several existing metrics measure size-related attributes. While these size-related metrics may have been originally developed to support quality assessment and assurance, they can be used to monitor progress in terms of size. Project size is important because it is usually used to estimate the resources needed and therefore assess the project's status with regard to the schedule for completion [6]; [1]; [16]. This known fact about management need for feedback about implementation progress should demand its use because its absence may be as result not introducing metrics to support implementation progress monitoring feedback before now.

Basic Concept

We design models normally to bridge the gap between real sampled data and expected outcomes. In real terms models act as predictors to set expectations over the next few data samples but in a small scale. Take for instance a defect model, this may indicate the number of faults to be found in a released project and the degree to which the actual number of faults discovered differs from the predicted value can actually be an indication of unexpected circumstances within the project. This means that, an unusual low value may be pointing to the fact that fewer code changes were made than expected, or that less testing was carried out than expected. This also means that Management has been forewarned; an investigation can be made and then appropriate response taken. This feedback, though small scale provides valuable and timely feedback to the management within the scope of the executing project.

In addition to this small scale feedback, models provide feedback on a larger scale, where the feedback focuses more on the overall picture portrayed by the data. This is regularly required by management, with or without a formal model. Without a formal model management team must rely on guess work. In contrast to this unusual approach, a formal model can be adopted to establish a rigorous evaluation. A formal model establishes the critical parameters within the system. Using a formal model, projects may be evaluated or compared in terms of the model parameters. Model parameters allow evaluation and comparisons to be based on defendable data rather than guess work and hearsay. Additionally, given estimated values for the parameters, the model can make predictions about the outcome. This relationship between parameters (input) and predictions (output) codifies a causal effect believed to be true within all designed system.

Similar Research Done Before Now

Some work done before with reference to software metrics on the development process has been directed at tasks before and after implementation. Much work has been done in the pre-implementation stages to improve effort prediction and estimation [1]; [22] [16]; [5]; [36]. Metrics

have also been used to evaluate architectural design before implementation. Significant work has been done in the post-implementation stages to predict failure rates, both for a project as a whole as well as for individual modules [15]; [11]; [37]; [9]; [34].

However, substantially less work has been published regarding the use of metrics for assessment, monitoring, or control of the implementation stage itself. DeMarco confirms this when he asserts "you can't control what you can't measure" [6]. In fact, every researcher now that is proposing, defending, or merely discussing a metric agrees the reason behind metrics and measuring is to gain some degree of understanding and control over the complex process of software development [10]. Recent studies have focused attention on *how* to use the vast array of data generated by existing measures.

Some published works emphasizing how to use the potentially enormous data available can be coarsely divided into three groups. Many works assert metrics should be used to assist in monitoring, evaluation and control of projects during the implementation phase [6]; [3]; [4]; [23]; [37]; [19]. Other researchers recommend specifically that time-series metrics data be used to monitor and evaluate projects [6]; [28]; [34]. The last group that consist of several researchers emphasize the idea gained from causal models over correlative models [32]; [10]; [36]. They suggest models which provide an inherent causes-relationship are more valuable than simpler correlation models.

Model Design

The main purpose of designing a model is to provide a documented method of interpreting a set of data. In most cases the interpretation is usually obscured by the sheer quantity and detail of the enormous data available and Information can only be revealed when these data are interpreted in a particular way. The interpretation results can then be used to evaluate past performance, assess the current situation, and make predictions about future performance.

The interpretation can also be used to compare multiple data sets. Results from the same model, applied to several data sets, allow the data sets to be easily compared in terms of the model. The model provides a systematic method for comparing projects. One type of model interprets series data by attempting to fit collected data to a family of curves. The single curve which best fits the data is used to describe the data in terms of the model. The specific values used to generate the best fitting curve are considered parameters of the model. Parameters of a model reveal one or more dimensions of the collected data. In this case, parameters can be considered an output of the model. Collected data is the input and results summarizing that data are produced. Parameters can also be used as input resulting in expected sample data. When used in this way, models make predictions based on estimated parameters. In either case, the expected progress as defined by the model is given by the model curve.

The model equation and a specific set of parameters define the model curve.

A valuable model is one that produces a clear and concise interpretation of the data. Part of this interpretation is in the form of the specific values for the model parameters. For example, consider two models, one uses only two parameters while the other has eight parameters. Even though the eight-parameter model may predict the data "twice" as well, it may not be the better model if its parameters have no particular meaning or are hard to estimate. Models should have as few parameters as possible while still modelling the data with sufficient accuracy. Fewer parameters means the model is easier to understand.

Part of understanding a model is understanding the relationships between the

parameters. Parameters are related by the effects each has on the others. Knowing the trade-offs between parameters is necessary to understand a model. This is easier if the model contains fewer parameters. In addition to relatively few parameters, individual model parameters should be understandable. Understandable parameters produce simple results with meaning.

On the other hand, meaningless parameters do not help to simplify or interpret the data since they must again be interpreted. Parameter meaning is even more important when the model is used as a predictor for new projects. In this case, model parameters must be estimated before any data has been collected. If the individual parameters are well understood better estimates for each will be made. Better estimates will produce better predictions. Related to individual parameter meaning is the parameter unit. Model parameters should be expressed in well-known units, rather than new or arbitrary ones. Parameters with direct interpretations allow the model results to be easily understood and used. Well-known units are also much easier to estimate. Again, this allows for better predictions. Model parameters should be few in number, directly interpretable, and measured in existing units. These properties give the model parameters the most meaning and thus give the model the most "clarifying power".

Implementing Model's monitoring.

Generally, Implementation progress is not a new concept and so in addition to basic model requirements, an implementation progress model must be compatible with existing models. An informal progress model already exists; it can be seen in project vocabulary and assumptions. For instance, this informal model is commonly used to answer certain project status queries, such as:

What is the expected completion date based on the current pace?
What was the size of the total effort for that project?
What fraction of the total effort is currently done?
What fraction of the total effort will be done by a certain date?

A proposed framework implementation model should serve the same purpose as the informal model. The model must help provide answers to questions about implementation speed and progress of current and future projects. The informal progress model captures another key attribute of implementation progress. The informal model acknowledges that project speed is not constant throughout a project: because sometimes projects "speed up" and "slow down". There must always be the ability or desire to constantly determine implementation velocity. As noted by McConnell [28] this velocity increases at the beginning and decreases near the end. A formal implementation progress model should be informed by this experience and capture the canonical variations in velocity during implementation.

In a nut shell, the desired attributes of a formal implementation progress model include relatively few parameters, understandable parameters, well-known parameter units, consistency with informal progress model, and the ability to answer management questions involving size and velocity.

Evaluation and Control

DeMarco presents a development process relying on steadily improving estimates to provide feedback and control during all phases of software development [6]. Metrics collected at each stage of development provide raw data for creating an improved estimate for the next stage. Metrics from the current project as well as previous projects are used to make inform decisions. What was focus throughout his work is the need for continuing feedback

by continuously improving estimates while allowing the development effort to be properly directed. He made a compelling case supporting estimates reviews, process metrics, and cost models in order to make quality improvement. DeMarco identifies and recommends appropriate metrics for each stage of development.

In discussing appropriate metrics for the implementation stage DeMarco points to process metrics such as *compilation rate* but did not explore them. The primary implementation measure is *code weight*, which is defined as a product of two dimensions: size and complexity. DeMarco defines code size as information content within a program. He recommends using Halstead's volume metric [12] to find size. Several alternatives for measuring complexity are presented, but McCabe's cyclomatic complexity measure [27] was recommended. Using these two dimensions as parameters, an algorithm is presented for computing *implementation weight.* Historical data from similar projects and environments is also used to provide scaling factors. According to DeMarco, the primary motivation for computing implementation weight is to improve future project estimates. However, he also calls it a "project predictor", as it was deplored to predict the final size of the project accurately. According to this novel system presented by DeMarco, the measure should be taken once near the middle of implementation. However, progress model been proposed by this paper may provide a better estimate of implementation size. Since the proposed model considers the complete project history, not simply a single point in time, because this is less prone to errors.

Boehm considered a broader approach to development feedback than simply focusing on improved estimates. He introduces a software development methodology whose principal aim is risk management [4]. His spiral model of software development relies on risk evaluation as the impetus for each unit of work, whether the work unit is a prototype, design document, or code. Risk management implies the ability to control what is being managed. This agrees with DeMarco's argument that our need to measure the software development process stems from our desire to control the process [6]. Boehm's methodology assumes feedback metrics exist to inform the risk evaluation process, but does not dictate specific measures or measurement processes.

Addressing the selection of appropriate metrics for quality control, Solingen and Berghout defined the Goal/Question/Metric Method (GQM) of improving software quality [37]. Goal/Question/Metric Method integrates metrics into the development process in order to answer questions about quality raised by corporate goals. Their methodology relies on the ability to follow the connection between corporate goals and specific metrics, in both directions. Measurements are defined by goals and the results interpreted in terms of those goals. In the area of quality control, well developed process models exist to help define and interpret metrics. However, implementation models in general and implementation progress in particular, have not been well developed.

Kirsopp addresses the need to capture development models and enough data to evaluate them. He strongly argues that the software development process needs measurements for feedback and that the integration must be close, detailed and appropriate [19]. Organizations must support metrics outside of a single project in order to validate the process, validate the results, and collect historic data. All three of these are required to assist future project estimates. An *experience factory* provides a repository for captured experiences and models, allowing reuse ‘within an organization. Kirsopp cites the “Tailoring a Metric Environment” Project [3] as a working example of an experience factory.

Lott provides an alternative approach; instead of suggesting or analyzing metrics, he studied several available and proposed software engineering environments [23]. Many of these environments include integrated tools for collecting numerous metrics about various development artefacts created. Lott suggests collected data can be used to guide development and to call attention to atypical patterns worthy of investigation. In this regard, he assumes time-series data will be collected and evaluated. Inherent in this idea is the development of a canonical pattern or typical shape for a particular metric. Unfortunately, neither Lott nor the systems studied define how to select or interpret the automatically collected measurements.

Time-Series Shape Metrics

Estimation is core to every software metrics consideration but in addition to that, DeMarco briefly notes that process metrics, such as compilation rate, can be used to identify project dysfunction and impending problems [6]. Periodic sampling of a metric allows the value measured to be graphed against time. For some measures, such as compilation rate, which means that all well planned projects, may all have similar shapes when viewed as time-series data. If this is the case, then projects not properly planned can be detected if or when they deviate from the canonical shape. Actually, DeMarco suggested that compilation rates that continue at a steady rate without showing any decline may be an indication of a poor work from development team. While this particular evaluation may not apply to all development environments, the idea of a well planned project canonical shape can be applied to all environments.

In another breathe, he recommended reporting test progress as a graph showing measurements against time. Time-series graphs make it clear how test progress has been proceeding and how its trends change over time. In general, comparing the current project with similar historic projects using graphs can highlight abnormal trends which may be an indicator of trouble. Given DeMarco's emphasis on continuous monitoring and improvement, it is surprising he does not suggest using implementation artefact metrics, such as size or complexity, to monitor implementation progress.

Schneidewind used time-series metrics to create a method for evaluating process stability [34]. Schneidewind emphasizes that metric trends are a significant indicator of the underlying process and monitoring the trends can provide feedback about the process. To quantify these trends, he introduces two new classes of indirect metrics. A *change metric* is computed using differences in consecutive values of a traditional metric. This metric can be viewed as the derivative of the primary metric. The other class of indirect metric introduced is the *shape metric.* A shape metric is derived from the curve of the time-series metric data when graphed against time. For example, one shape metric suggested is the time at which the failure rate is highest. Lower values for this metric may indicate process stability, while higher values may indicate instability in the development process. A strong case is presented for the use of time-series data, and indirect metrics derived from it, in the context of process stability. Monitoring progress during the development stage using change and shape metrics is an obvious extension of this study.

McConnell understands typical "code growth" on a project to contain three distinct phases [28]. In the first phase, architectural development and detailed design generate very little code. The second phase provides staged deliveries and includes detailed design, coding, and unit testing. During this phase code growth is very high. During final release, the third phase, code growth slows to a crawl. McConnell shows a graph depicting a typical code growth pattern for a well-managed project. He indicates the phase

transitions occur at approximately 25% and 85% of the total development time, but acknowledges that this varies to some degree. His main point is periodic monitoring of code size is a valuable feedback tool for managers. No details are given about the specific metric(s) involved or the process used to collect the data. The proposed progress model clarifies how metrics are used and provides a specific interpretation of the three phases documented by McConnell.

Process Models

Powell expanded the frontier of the role of software measurement to explicitly include not only prediction and control but also assessment and understanding [32]. He propounded arguments for assessment similar to those presented by Boehm and DeMarco for prediction and control. Regardless of the motive, measurements are always based on assumptions about the process in which the measurement is taken. Powell states "it is impossible to talk about measurements without implying some form of [process] model" [32]. Before measurements can be taken, and before metrics can be determined, a model of the development environment must be chosen.

Turski presents a model for understanding the observed rate of software growth as a function of time [36]. Using the number of modules as the dependent variable and uniform inter release intervals as the independent variable, he shows size correlates strongly with the third root of time {size = \/time). While defendable on the bases of Lehman's Laws of Software Evolution [21], Turski uses a simple mental model to understand the same relationship. He suggests envisioning a system as a sphere with "surface" modules being easy to modify while "interior" modules are much harder to modify. With this model in mind, it is easy to see that the proportion of easy modules to hard modules tends toward zero as the project (sphere) grows with time.

Turski believes that simple and manageable models provide powerful insights into understanding the forces at work in software development. In particular, models which exhibit causal relationships rather than simple statistical correlations provide not only better interpretation but improved understanding of the process.

Framework Design

Project managers and software designers have developed an actual framework from intuition about what should occur during a software development process. And as a matter of fact framework implementation progress model should be consistent with this acquired experience. A condensed version of this collective wisdom is presented by McConnell [28]. He uses *code growth* as a measure of progress and provides a nominal code growth pattern as well *as* a range of normal variations for well-run projects. An appropriate progress model should reflect the basic shape of accepted norms such as those presented by McConnell.

Another constraint on designing an appropriate framework implementation progress model is its interpretive power. Interpretation of metric data relies on some understanding of the underlying process and how it works. Take for instances, changes in the rate of progress in an otherwise stable environment may indicate the project has transitioned to a new phase. This assumes the rate of progress is dependent on the project state. This process of drawing meaning from data, such as when a phase ends, is interpretation and of course an implementation progress model must approximate actual project data collected.

Solution Model:

Static Analysis

Halstead's Software Physics or Software Science

n1 = no. of distinct operators in program
n2 = no. of distinct operands in program
N1 = total number of operator occurrences
N2 = total number of operand occurrences
Program Length: N = N1 + N2
Program volume: $V = N \log_2 (n1 + n2)$
(represents the volume of information (in bits) necessary to specify a program.)
Specification abstraction level: L = (2 * n2) / (n1 * N2)
Program Effort: $E = (n1 + N2 * (N1 + N2) * \log_2 (n1 + n2)) / (2 * n2)$
(interpreted as number of mental discrimination required to implement the program.)

McCabe's Cyclomatic Complexity
Hypothesis: Difficulty of understanding a program is largely determined by complexity of control flow graph.

- Cyclomatic number V of a connected graph G is the number of linearly independent paths in the graph or number of regions in a planar graph.

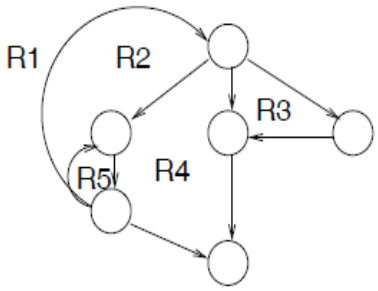


Fig. 1: Planar Graph

- Claimed to be a measure of testing difficulty and reliability of modules.
- McCabe recommends maximum V(G) of 10.

Static Analysis (Problems)

- Does not change as program changes.
- High correlation with program size.
- No real intuitive reason for many of metrics.
- Ignores many factors: e.g., computing environment, application area, particular algorithms implemented, characteristics of users, and ability of programmers.
- Very easy to get around. Programmers may introduce more obscure complexity in order to minimize properties measured by particular complexity metric.
- Size is best predictor of inherent faults remaining at start of program test.

Bug Counting Using Dynamic Measurement
Estimate number remaining from number found.

1) Failure count models 2) Error seeding models Assumptions:

- Seeded faults equivalent to inherent faults in difficulty of detection.
- A direct relationship between characteristics and number of exposed and undiscovered faults.
- Unreliability of system will be directly proportional to number of faults that remain.
- A constant rate of fault detection.

What does an estimate of remaining errors mean?

- Interested in performance of program, not in how many bugs it contains.
- Most requirements written in terms of operational reliability, not number of bugs. Alternative is to estimate failure rates or future inter -failure times.

Estimating Failure Rates
Input-Domain Models:

- Estimate program reliability using test cases sampled from input domain.
- Partition input domain into equivalence classes, each of which usually associated with a program path.
- Estimate conditional probability that program correct for all possible inputs given it is correct for a specified set of inputs.

- Assumes outcome of test case given information about behaviour for other points close to test point.

Reliability Growth Models

Software Reliability: The probability that a program will perform its specified function for a stated time under specified conditions.

- Execute program until "failure" occurs, the underlying error found and removed (in zero time), and resume execution.
- Use a probability distribution function for the inter failure time (assumed to be a random variable) to predict future times to failure.
- Examining the nature of the sequence of elapsed times from one failure to the next.
- Assumes occurrence of software failures is a stochastic process.

Software Uncertainty

Assumption: The mechanism that selects successive inputs during execution is unpredictable (random). O is the image set of I_F under the mapping p

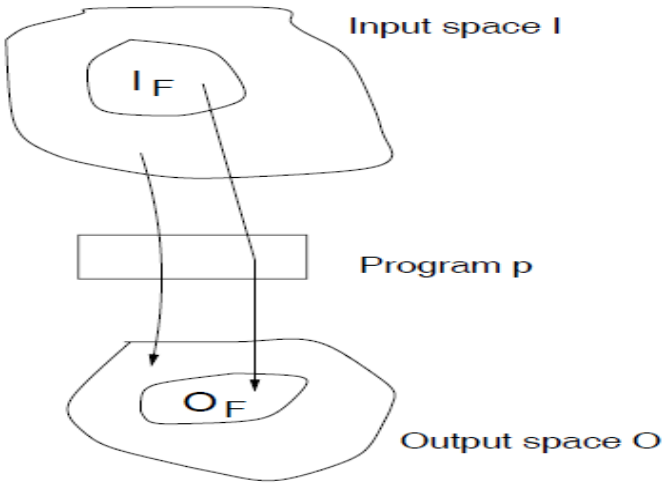


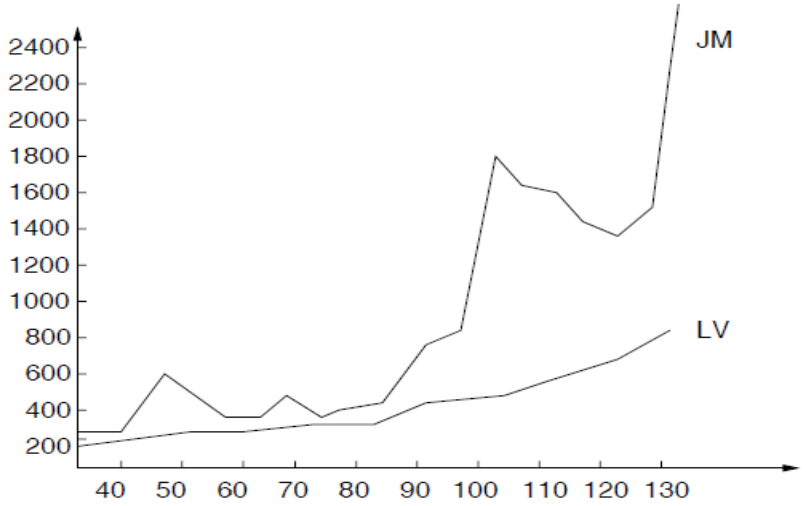
Fig. 2: Software Uncertainty illustrated

Table 1: Sample Inter Time Data

3	30	113	81	115	9	2	91	112	15
138	50	77	24	108	88	670	120	26	114
325	55	242	68	422	180	10	1146	600	15
36	4	0	8	227	65	176	58	457	300
97	263	452	255	197	193	6	79	816	1351
148	21	233	134	357	193	236	31	369	748
0	232	330	365	1222	543	10	16	529	379
44	129	810	200	300	529	281	160	828	1011
445	296	1755	1064	1783	860	983	707	33	868
724	2323	2930	1461	843	12	261	1800	865	1435
30	143	109	0	3110	1247	943	700	875	245
729	1897	447	386	446	122	990	948	1082	22
75	482	5509	100	10	1071	371	790	6150	3321
1045	648	5485	1160	1864	4116				

Applying the Models

Different models can give varying results for the same data; there is no way to know a priori which model will provide the best results in a given situation



.

Fig. 3: Models Applied

The nature of the software engineering process is too poorly understood to provide a basis for selecting a particular model.

Software Design Metrics

Number of parameters

- Tries to capture coupling between modules.
- Understanding modules with large number of parameters will require more time and effort (assumption).
- Modifying modules with large number of parameters likely to have side effects on other modules.

Number of modules

Number of modules called (estimating complexity of maintenance).

Fan-in: number of modules that call a particular module.

Fan-out: how many other modules it calls.

- High fan-in means many modules depend on this module.
- High fan-out means module depends on many other modules. Makes understanding harder and maintenance more time-consuming.

Data Bindings

Triplet (p,x,q) where p and q are modules and X is variable within scope of both p and q.

Potential data binding:

- X declared in both, but does not check to see if accessed.
- Reflects possibility that p and q might communicate through the shared variable.

Used data binding:

- A potential data binding where p and q use X.
- Harder to compute than potential data binding and requires more information about internal logic of module.

Actual data binding:

- Used data binding where p assigns value to x and q references it.
- Hardest to compute but indicates information flow from p to q.

Cohesion metric

Construct flow graph for module.

- Each vertex is an executable statement.
- For each node, record variables referenced in statement.
  Determine how many independent paths of the module go through the different statements.
- If a module has high cohesion, most of variables will be used by statements in most paths.

- Highest cohesion is when all the independent paths use all the variables in the module.

Robert Cecil Martin Popularized Software Package Metrics

Robert Cecil Martin is one of the creators of agile software development methodologies and extreme programming. Created metrics used for evaluating object oriented software packages which was also meant to be used with in an extreme programming framework. The advantage is that the metric calculation is relatively transparent.

As long as the criteria are important, developers can build software that follows these constraints and get better metrics about their code various aspects of software packages can be measured such as [4]

This metric shows how the package balances between abstractness and stability. A package will do well in this metric by being either mostly abstract or stable, or completely concrete and instable. Package dependency cycles: packages in the packages hierarchy that depend on each other. (Dependency cycles)

Embedding Quality

When fault prediction is incorporated in Robert Cecil Martin models it will allow organizations to predict defects in code before software has been released.

There is debate around aggregating the models or modelling only some of the defects more accurately. Part of the debate stems from the need for more research in software decomposition and how to decompose software for better quality systems. .Models are used to explain aspects of a software system numerically in at a statistically significant level. Research that is trying to address the human judgment side of metrics processing is also being addressed

Breaking down the defects that software is measured for will give a better view of the particular type of defect you are interested in and with the design Frameworks it is easy to understand metrics and making sure that we are using them correctly. Though the metrics analysis techniques, and the usefulness of data is not fool proof. Software metrics are statistical predictions and estimations, and not just a number. The numbers have three dimensions [2] error, bias, and variance or scatter. A human typically ignores these dimensions for simplicity and with the loss of information comes over optimism and over-confidence.

Research is being done to use meta cognition experiment results in application to software metrics interpretation and use in project planning and reflection. [2]. Researchers Carolyn Mair and Martin Sheppard want to include the perspective on how people actually employ software metrics results currently so they can understand how to make those metrics better. Kaner and Bond argue that metrics in software engineering are not yet properly used. In part this is because the validity of the metrics that are used is not emphasized.[10] .This could be due to many reasons, one being that software engineering is traditionally a empirical field. If the software is working the default action is typically to leave it alone.

Also, one of the most common form of software metrics is automated and user testing. This happens to be very expensive and one of the first things to get cut from the development lifecycle. So if a company or organization is going to cut testing, it seems likely to us that they would also not use metrics for the same reasons.

Conclusions

Few metrics have been demonstrated to be predictable or related to product or process attributes so interpreting implementation progress measurements is difficult. A simple model is needed to provide a framework to help interpret the

data. We have developed a piecewise approximation based on a three-phase model of linear implementation velocity. The model corresponds well to our intuition of how project progress occurs. It identifies project phase boundaries as well as the velocity of implementation during each phase. Furthermore, the progress model allows objective comparisons of project velocity between projects and easily supports estimating.

The progress model fits the available sample data much better than a linear model. With only one additional degree-of-freedom, the model produces fits with approximately two-thirds less error than a linear fit. When compared with a polynomial fit, the progress model performs at least as well as a polynomial model which has one additional degree-of-freedom.

Limitations and Recommendation

The progress model presented here only considers non-maintenance Implementation. Projects with clear delivery dates, after which continuing development is not planned, fall into this category. Projects in maintenance or under continuous development may not exhibit phases similar to projects with firm end dates. Any model is only as good as the data on which it is based. Errors were discovered in both dimensions of the sample data. Spurious data entries were occasionally introduced due to the check-in process used. Similarly, project billing information could have helped improve the quality of the time data collected.

There is "a strong tendency for professionals to display over-optimism and over-confidence"

Arguments that simplistic measurements may cause more harm then good, ie data that is shown because it is easy to gather and display [5]

There are arguments about the effects that software metrics have on the developers' productivity and well being

This research paper provides a sound basis for further study in this area. Application of the progress model to continuous development projects should be investigated.

Again there should be further study to take advantage of the stability of the model for making predictions. Estimating project parameters such as final size, delivery date, development pace, etc. during implementation should be investigated. Similarly, the effect of project properties, such as number of engineers, experience level, domain familiarity, length of project, etc., on the model parameters should also be studied..

.

References

[1] Albrecht, A. J. and J. John E. Gaffney (1983, Nov). Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering* 5(6), 639-648.

[2] Andersson, T., K. Enholm, and A. Torn. RMC a length-independent measure of software complexity. Reports on computer science and mathematics, Abo Akademi University.

[3] Basili, V. R. and H. D. Rombach (1988). The TAME project: Towards improve mentoriented software environments. *IEEE Transactions on Software Engineering 14 {6)*, 758-773.

[4] Boehm, B. W. (1988). Spiral model of software development and enhancement. *IEEE Computer 21 {3)*, 61-72.

[5] Boraso, M., C. Montangero, and H. Sedehi (1996). Software cost estimation: an experimental study of model performances. Technical Report TR-96-22, Universita di pisa, departmento di informatatica.
[6] DeMarco, T. (1982). *Controlling Software Projects Management, Measurement and Estimation.* Inglewood Cliffs, NJ: Yourdon Press.
[7] El-Eman, K. (2000, June). A methodology for validating software product metrics. Technical Report NRC/ERB-1076 44142, National Research Council Canada, Institute for Information Technology.
[8] Fenton, N. E. (1991). *Software Metrics: A Rigorous Approach.* London: Chapman and Hall.
[9] Fenton, N. E. and M. Neil (1999). A critique of software defect prediction models. *Software Engineering 25 {5)*, 675-689.
[10] Fenton N. E. and M. Neil (2000, June). Software metrics: roadmap. In *Proceedings of the conference on The future of Software Engineering,* pp. 357-370. ACM Press.
[11] Goel, A. L. and K. Okumoto (1979, August). Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE Transaction on Reliability RE-28{3),* 206-210.
[12] Halstead, M. H. (1977). *Elements of Software Science.* New York, NY: Elsevier Scientific.
[13] Henry, S. M. and D. G. Kafura (1981, Sep). Software structure metric based on information flow. *IEEE Transactions on Software Engineering* 7(5), 510-518.
[14] Huffman, D. A. (1952, Sep). A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Electrical and Radio Engineers 40(9),* 1098-1101.
[15] Jelinski, Z. and P. Moranda (1971). Software reliability research. In W. Freiberger (Ed.), *Statistical Computer Performance Evaluation,* pp. 465-484. Providence, RH: Academic Press.
[16] Jorgensen, M. (1995). Experience with the accuracy of software maintenance task effort prediction models. *IEEE Transactions on Software Engineering 21 {8),* 674-681.
[17] Kafura, D. and J. Canning (1985, Aug). A validation of software metrics using many metrics and two resources. In *Proceedings of the 8th International Conference on Software Engineering,* pp. 378-385.
[18] Kaner, C., Bond, W. (2004) Software Engineering Metrics: What Do They Measure and How Do we Know? *10th International Software Metrics Symposium*
[19] Kirsopp, C. (2001, Apr). Measurement and the software development process. In 1*2th European Software Control and Metrics Conference,* pp. 165-173.
[20] Lake, A. and C. R. Cook (1994, apr). Use of factor analysis to develop oop software complexity metrics. In *Proceedings Sixth Annual Oregon Workshop on Software Metrics.*
[21] Lehman, M. M., J. F. Ramil, R D. Wernick, and D. E. Perry (1997). Metrics and laws of software evolution-the nineties view. In *Proceedings of 4th International Symposium on Software Metrics,* pp. 20.
[22] Lind, R. K. and K. Vairavan (1989, may). An experiemental investigation of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering 15{h),* 649-653.
[23] Lott, C. M. (1993, Oct). Process and measurement support in sees. *ACM SIGSOFT Software Engineering Notes 18{-i),* 83-93.

[24] Mair, C., Shepperd, M. (2011) Human Judgement and Software Metrics: Vision for the Future. *WETSoM'11* (May 24, 2011), Waikiki, Honolulu, HI USA.
[25] Marasco, J. (2002, August). Tracking software development projects. *Dr. Dobb's Journal.*
[26] Martin, R. D. and V. Yohai (2001). Data mining for unusual movements in temporal data. In *KDD Workshop on Temporal Data Mining.*
[27] McCabe, T. J. (1976, Dec). A complexity measure. *IEEE Transactions on Software Engineering* S(4), 308-320.
[28] McConnell, S. (1998). *Software Project Survival Guide.* Redmond, WA: Microsoft Press.
[29] Misirli, A., Caglayan, B., Miransky, A., Bener, A., Ruffolo, N. (2011). Different Strokes for Different Folks: A Case Study on Software Metris for Different Defect Categories.
[30] Nagappan, N., Ball, T. (2007) Using Software Dependencies and Churn Metrics to Predict Field
Failures: An Empirical Case Study. *IEEE First International Symposium on Empirical Software Engineering Measurement.*
[31] Park, R. E. (1992). Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, PA.
[32] Powell, A. L. (1998). A literature review on the quantification of software change.technical report YCS-98-305, University of York, Department of Computer Science.
[33] Robert Cecil Martin (2002) .http://www.sdtimes.com/link/34157
[34] Schneidewind, N. F. (1999, Nov). Measuring and evaluating maintenance process using reliability, risk, and test metrics. *IEEE Transactions on Software Engineering 25 {6)*, 761-781.
[35] Strelzoff, A. and L. Petzold (2003). Revision recognition for scientific computing: theory and application. In *18th Annual Software Engineering and Knowledge Engineering Conference*, pp. 46-52.
[36] Turski, W. M. (2002, August). The reference model for smooth growth of software systems revisited. *IEEE Transactions on Software Engineering 28{8)*, 814-815.
[37] van Solingen, R. and E. Berghout (1999). *The Goal/Question/Metric Method.* London: McGraw-Hifl Publishing Company.
[38] Vasilescu, B., Serebrenik, A., van den Brand, M. (2011). By No Means: A Study on Aggregating Software Metrics.
[39] WETSoM'11 (May 24, 2011), Waikiki, Honolulu, HI USA. Agile Software Development: Principles, Patterns and Practices. Pearson Education.
[40] Woodings, T. L. and G. A. Bundell (2001, April). A framework for software project metrics. In *Proceedings of the ESCOM 2001*

.