

Measuring Contribution to a Software Group

Jonathan Francis Roscoe <jjr6@aber.ac.uk>
Department of Computer Science, Aberystwyth University

Abstract

It's not uncommon for animosity, frustration and a lack of motivation to arise in projects as a result of perceived differing work efforts. This paper proposes a framework for assessing team member contributions to coding, testing and documentation for software engineering projects. The intent is to reveal that the worth of contributions are not always as obvious as they may seem, and some seemingly significant contributions are in actuality considerably less substantial.

I. INTRODUCTION

"Software metrics" refers to the use of quantitative measures to facilitate project planning and management. There is both criticism[13] and support[6] for the use of metrics with analysis yielding information on the financial costs of development, the productivity of developers and system maintainability. Astutely applied, metrics can be used to make project predictions.

Most criticisms of quantitative methods are based on the quality of metrics and techniques used, which are often seen as inconsistent and unreliable[16]. This paper examines the entities of a typical software development project and considers how metrics might be effectively utilised. The objective in designing the framework is to assess contribution to code, tests and documentation, answering two crucial questions: "who is contributing what?" and "how valuable is the contribution to the project?".

These questions bring to light a common issue in software development - it is hard to judge the worth and effort of a developer. The implementation of an elegant algorithm may only cover a few short lines yet take many hours of design and study before it can be accomplished, yet a stub class with a series of "getters and setters" may consume many hundreds of lines yet be designed and written in a much shorter amount of time. These problems persist for the creation of tests and documentation as well. At a glance the layperson (e.g. management) may infer greater effort from longer code, when this is not necessarily true. Long code may in fact be detrimental to a program, and some metrics are designed to ascertain code maintainability and similar properties using such attributes[4].

II. MODELS

For each entity in an observed system, attributes must be identified and measured. The rules by which these measurements are made are referred to as a model.

Fenton and Pfleeger[8] describe an entity as an object, process or event in the real world (e.g. a person, source code, a document), every entity has features or attributes. The way in which we measure and represent attributes must retain their real-world value and relationships with other attributes in order to be reliably used. Attributes may be measured with nominal (categorical/one-to-one), ordinal (monotonic, increasing - ordered transformation), interval (adjacent -linear transformation), absolute or ratio (multiplication transformation) scales. Different admissible transformations exist for these scales, affecting the way they can be assigned a new homomorphism (map to a scale), this affects the way different scales may relate to one another.

Entities may have either internal or external attributes, where external attributes are measured in relation to other entities. Below is an outline of the entities that the framework recognises and their pertinent attributes.

A. Developer

As the goal is to assess individual contributions, a sensible first entity is the developer. The term 'developer' is used loosely in this paper to refer to all relevant members of the team contributing to the software project. It is apt to say that contribution is a matter of productivity - the ability for a programmer to develop substantial code in a time frame. Productivity is an external attribute that relies on the programmer's internal attributes as well as the attributes of their work entities.

Attribute	Measurement Scale	Description/Rule
Time	Absolute	<i>Internal.</i> A measurement of the team member's allotted working hours.
Productivity	Ordinal (e.g. unproductive/productive/highly productive)	<i>External.</i> This relies on the results of other measurements, for example, the time average number of faults found per hour of testing and lies on a curve relative to other developers.

B. Code

The rate of code development is sometimes naively thought to indicate programmer performance, however one amount of code may be significantly more complex than another. Halstead's complexity and volume measures[1] are an early yet rigid set of measures formed by combining basic measures of code that provide a more sophisticated analysis.

Halstead's original measurements are included here to illustrate how they relate to one another:

u_1 : the number of unique operators

u_2 : the number of unique operands

n_1 : total number of operators

n_2 : total number of operands

Length: $N = n_1 + n_2$

Vocabulary: $u = u_1 + u_2$

Volume: $V = N \times \log_2 u$

V^* = volume of smallest implementation

Difficulty¹: $D = 1 \div (V^* \div V)$

An unfortunate shortcoming of such measures is that in the end, they are only basic observations of code, with no accounting for background research, debugging and experiment by the programmer. Even an expert programmer could end up spending the majority of their time familiarising themselves with an unfamiliar code base before any real progress can be made, which makes it difficult to link programmer time to code volume. There are also more basic issues such as a lack of consensus on the interpretation of "volume" and "difficulty", and the fact that the methods Halstead used to combine basic measures are unsubstantiated.

However, Halstead's work has been used to distill the fundamental attributes of code used in the framework (outlined in the below table) as they provide a seemingly sensible interpretation of key code attributes and in contrast to function point measures[14] are less prone to inaccuracies in analysis, require less specification and are more intuitive.

Attribute	Measurement Scale	Description/Rule
Length	Interval	<i>Internal.</i> A sum of the number of times operators and operands occur.
Vocabulary	Absolute	<i>Internal.</i> A sum of the number of unique operators and operands.
Volume	Absolute	<i>Internal.</i> A normalised assessment of code size. Based on vocabulary and length.
Difficulty	Ratio	<i>Internal.</i> Also known as "maintainability". An assessment of complexity based on the occurrences of unique operands and operators. Requires a known volume for a smallest implementation to calculate the program level.
Reuse	Nominal (e.g. high/medium/low)	<i>Internal.</i> By looking at other attributes we can make assumptions about code reuse. For example, the closer length are vocabulary are to one another, the less code is being reused.

For object-oriented systems there exist metrics such as "number of children" and "coupling between object classes". Whilst this does not apply to all code, it is relevant to some projects, and attributes such as coupling, cohesion and ancestry should be considered in the model if necessary. A selection of attributes for object-oriented code (in addition to the above) are:

¹ $V^* \div V$ is referred to as "program level", the presented framework has removed this measurement for simplicity.

Attribute	Measurement Scale	Description/Rule
DIT	Absolute	<i>Internal.</i> Depth of inheritance tree.
LCOM	Absolute	<i>Internal.</i> Lack of cohesion metric
WMC	Absolute	<i>Internal.</i> Weight methods per class. A sum of the measure of complexity for all methods in a class.

The attributes (amongst others) were formed by Chidamber and Kemerer[3] in a sound and refined[12] suite for object-oriented metrics. These attributes can be used to highlight code complexity, quality and imply maintainability. For example, classes with a large number of children may be of great value to the project, but may also be a point of difficult maintainability due to dependents. There exists considerable criticism for current object-oriented metrics, with suggestions that there is no suitable solution[11]. However, there are some judicious attributes that have been selected for use in the framework.

C. Testing

The testing attributes enable a basic measure of the productivity of the testing process. This is done by measuring the amount of code tested and the time. Coverage can also be used as an implication of quality, with full coverage indicating that every possible path of execution - every eventuality is tested for. The attributes used are more intuitive than the various measures for code, and are well documented and used in industry[10].

Attribute	Measurement Scale	Description/Rule
Time	Absolute	<i>Internal.</i> A measurement of the time it takes to run tests.
Coverage	Ratio	<i>Internal.</i> The number of lines tested.
Faults found	Absolute	<i>Internal.</i> A measurement of the number of faults found during testing.
Effectiveness	Ratio	<i>External.</i> A measure of how many faults testing catches and how many faults were shipped. Effectiveness $e = 1 \div (f * \div f)$ where $f*$ is number of faults discovered in total (including user reported), f is number of faults discovered during testing.

D. Documentation

Measures for documentation is arguably the least explored area of software engineering metrics. Le Vie[15] highlights some of the ineffective measurements typically found in industry. To ensure the metrics framework is as suitable to software projects as possible an external attribute, coverage, is being used. The coverage attribute is a measure of the effectiveness of the documentation for presenting information on the software product.

Attribute	Measurement Scale	Description/Rule
Length	Absolute	<i>Internal.</i> A measure of the documentation length.
Coverage	Ratio	<i>External.</i> A measure of the ratio of software features to features listed in documentation.

III. METRIC COLLECTION

For each entity, all internal measurements must be collected, after which the external measurements can be formed. In order to provide reliable analysis and reporting, it is important that the measurements are as accurate as possible; inaccuracy is the downfall of many such systems[11].

A. Developer

Depending on the working atmosphere, a timesheet or contractual obligation may be used provide a measurement of time. The external measure of productivity would require the measures of all other entities with which the developer is involved with (code/testing/documentation) to be formed. Length, coverage and complexity measures of other entities can be used to determine how productive the developer is in respect to those entities

of the project(a matter of dividing the relevant measures over time). It would be expected that on a well balanced team, the productivity of all developers falls within a bell curve, with many developers being equally productive, with a small number of outliers.

B. Code

In a practical implementation, static code analysis can be used for the collection of these attributes, there exist readily available CMT or Complexity Measurement Tools for code². Static analysis is a common part of software development, from very basic tools such as lint, to complex debuggers, they are a tried and tested means of analysing code. Popular revision control systems such as subversion and git can even be configured to execute scripts for analysis on commit triggers. Continuous integration systems are a common means of performing round the clock code analysis and are tried and testing for executing test suites. An appropriately developed static analysis complexity measurement tool would be able to collect all of the code attributes outlined.

The framework uses no true external attributes for code, typical external measures such as maintainability are based on complexity which are gained from Halstead's internal measures.

C. Testing

Many well-managed software projects involve some form of tracking system for reporting and managing code faults. Dynamic analysis tools³ can be used to determine the code coverage of test suites, as previously mentioned, systems exist to facilitate this, as well as collect information on time and faults found. However, in order to collect the total number of faults found (for the external measure of effectiveness), bug reports must be relied upon and will most likely require manual checking.

D. Documentation

Static analysis can also be applied to text, to form word and page counts. For the external coverage attribute, the only feasible solution would be a manual checklist of functionality against documented areas.

IV. FRAMEWORK OUTLINE

The proposed framework has five distinct phases. Phases 1-4 involve the collection of measurements, whilst the final phase concerns their interpretation. The phases are shown in figure 1.

The collection of metrics for testing and documentation can be done in parallel, but must be performed after code measurements, that is, after the coding period of the development cycle. The process is designed to facilitate collection of dependent measurements in an efficient way.

Once the phases of metric collection have been completed, analysis can be performed to answer the questions outlined previously.

V. ANALYSIS

With all measures collected the most difficult task of making sense out of the data. A tool for the framework will need to be developed.

²Testwell CMT++ - <http://www.testwell.fi/cmtdesc.html>, JavaCMT - http://www.verifysoft.com/en_cmtx.html

³e.g. Clover for Java or PHPUnit for PHP

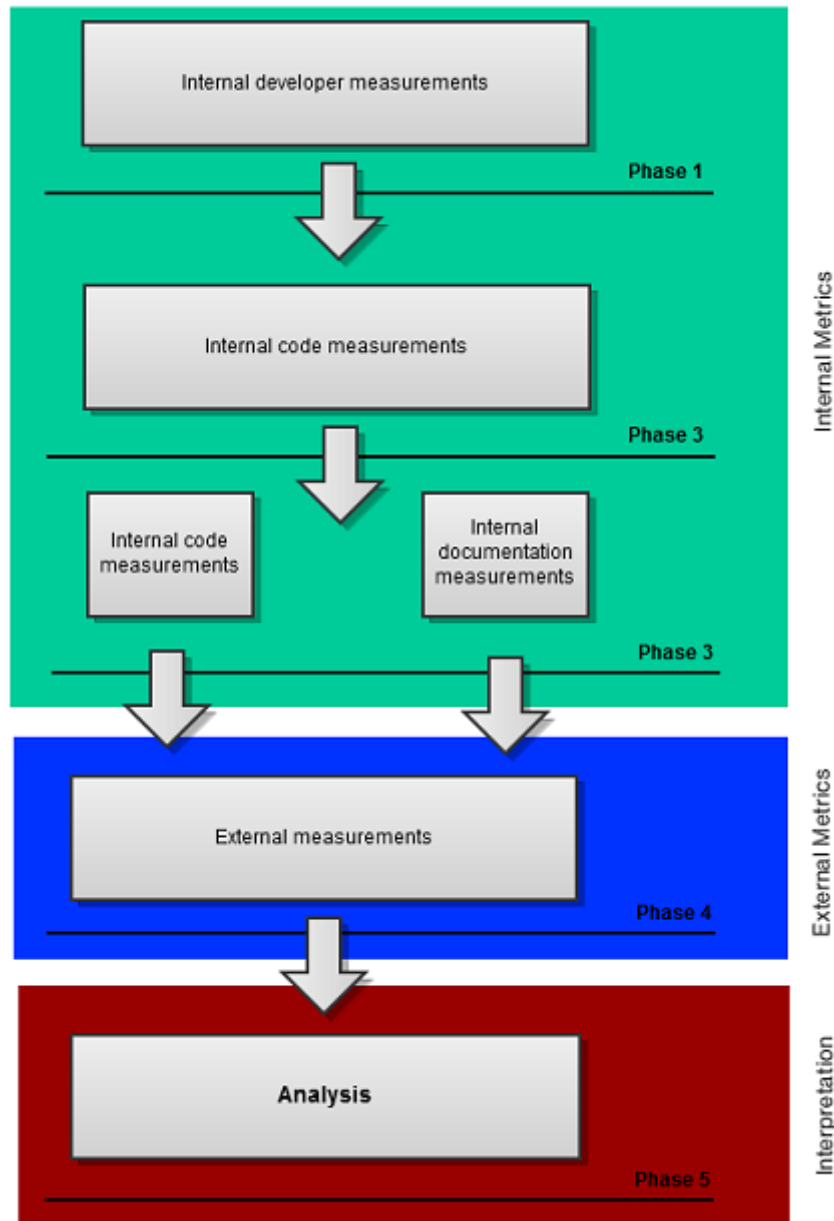


Figure 1. Diagram demonstrating the progression through phases of the framework. *Phase 1: Team member (developer) measurements.* *Phase 2: Internal code measurements.* *Phase 3: Subsequent internal measurements.* *Phase 4: External attribute measurements.* *Phase 5: Analysis/Report.*

A. Who is contributing what to the project?

As previously mentioned, we can ascertain which developer entities are contributing using the ordinal entity of productivity.

$$\text{productivity}_i = t_i \div ((cl_i + cd_i) + tc_i + dc_i)$$

Where

- i** : team member
- t** : developer time
- cl** : code length
- cd** : code difficulty (It would be wise to add WMC to this component for OO systems)
- tc** : test coverage
- dc** : documentation coverage

Groups of team members (ie. just testers or just documenters) could be assessed amongst themselves by excluding the irrelevant entity attributes. Of course, this does not account for those who interact with entities as only a part of their work (time would also need adjusting).

The result of applying this formula for each team member on a project would be a set of scores that show the productivity of team members, these could easily be plotted on a chart charting productivity along the x axis.

B. How valuable is this contribution to the project as a whole?

We can determine the worth of developer contributions using the effectiveness and reuse measures outlined.

For code, we use the reciprocal of reuse over length:

$$\text{code_contribution}_i = 1 \div (\text{cr}_i \div \text{cl}_i)$$

For testing the effectiveness of effectiveness over coverage:

$$\text{test_contribution}_i = 1 \div (\text{te}_i \div \text{tc}_i)$$

For documentation we use coverage over length:

$$\text{test_contribution}_i = 1 \div (\text{dc}_i \div \text{dl}_i)$$

Where

- i : team member
- cl : code length
- cr : code reuse
- te : testing effectiveness
- tc : testing coverage
- dc : documentation coverage
- dl : documentation length

C. Project Prediction

The remaining measures outlined by the framework can be used for the prediction of maintainability, in particular, the measures for code such as the object-oriented cohesion and inheritance rules. These rules provide a means to assess difficulties in future development of a codebase. Combining cost measurements may also be useful for planning

VI. EVALUATION

A series of entities and attributes have been identified as reasonable properties for the measuring team member contribution to software projects. Decisions have been made based on previous, authoritative work as referenced. It is difficult to ascertain for certain the viability of the proposal ideally the objectives and requirements would be more thoroughly investigated prior to implementation and would most likely need some degree of customisation for a project depending on the technical characteristics and development methodology. Empirical surveys would need to be conducted to determine if the analysis of the metrics correlate with the feelings and interpretation of the team members and management.

Using Halstead's measures means there is no ambiguity over "lines of code" since length is measured using the occurrence of operands and operators; blank and commented lines are irrelevant. Comments, however, can be invaluable in code and it is typical for developers to sprinkle their code as they go along. One significant problem is that the framework does not account for time spent that doesn't directly contribute to code, for example research or design. One way to overcome this would be to have developers keep track of the amount of time spent programming specifically, then compare the time values for each programmer.

Measures for testing and documentation are similar in nature, the selection of these attributes will hopefully provide a robust basis for the framework, but some significant development is needed. It would be beneficial to better link the code and testing entities as testing may often interact with the coding process (in the way of bug fixes).

It is reasonable to say that the findings of any software metric framework should be taken with a pinch of salt. There is disagreement on the practicality and usefulness of metrics in the software industry, and a lack of consensus in vocabulary, the understanding of volume, effort and similar terms can vary significantly. It is necessary that the development process be clearly defined and measurements meticulously collected[7]. Sometimes assumptions must be made with unclear internal attributes[5] making reliable analysis difficult. However, methodologies have been successfully applied to industrial projects[9]. The crux of the matter is that software metrics is an area with much disagreement and vying opinion.

I believe a sensible argument to conclude with is that the best metric is client/manager satisfaction. Measurements of various project characteristics may be made, but their analysis doesn't necessarily have any credence. Some schools of thought[18] suggest effective teamwork arises out of mixed-ability groups with a variety of contribution (a lack of tangible contribution to the development does not deny a lack of contribution to the group)[2]. Software metrics are most likely best applied in rigid inch-pebble development scenarios where control over measurements is assured.

REFERENCES

- [1] C T Bailey and W L Dingee. A software study using halstead metrics. *SIGMETRICS Perform. Eval. Rev.*, 10:189–197, January 1981.
- [2] G Beranek, W Zuser, and T Grechenig. Functional group roles in software engineering teams. In *Proceedings of the 2005 workshop on Human and social factors of software engineering*, HSSE '05, pages 1–7, New York, NY, USA, 2005. ACM.
- [3] S R Chidamber and C F Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.
- [4] D Coleman, D. Ash, B Lowther, and P Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.
- [5] N Fenton. Software measurement: a necessary scientific basis. *Software Engineering, IEEE Transactions on*, 20(3):199–206, March 1994.
- [6] N Fenton and M Neil. Software metrics: successes, failures, and new directions. *Journal of Systems and Software*, 1999.
- [7] N Fenton and M Neil. Software metrics: roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 357–370, New York, NY, USA, 2000. ACM.
- [8] N Fenton and S L Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.
- [9] T Hall and N Fenton. Implementing effective software metrics programs. *IEEE Softw.*, 14:55–65, March 1997.
- [10] M L Hutcheson. *Software Testing Fundamentals: Methods and Metrics*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.
- [11] Churcher N I and Shepperd M J. Comments on 'a metrics suite for object oriented design'. *IEEE Transactions on Software Engineering*, 21:263–265, 1995.
- [12] El Emam K, Melo W, and Machado J C. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001.
- [13] C Kaner and W P Bond. Software Engineering Metrics: What Do They Measure and How Do We Know? *10th International Software Metrics Symposium (METRICS 2004)*.
- [14] B Kitchenham. Counterpoint: The problem with function points. *IEEE Softw.*, 14:29–, March 1997.
- [15] D Le Vie Jr. Documentation metrics: What do you really want to measure? *INTERCOM Magazine*, Vol. 47, No. 10, December 2000.
- [16] R Lincke, J Lundberg, and W Löwe. Comparing software metrics tools. pages 131–142, 2008.
- [17] T J McCabe and C W Butler. Design complexity measurement and testing. *Commun. ACM*, 32:1415–1425, December 1989.
- [18] N A Stanton and J S Prichard. Testing belbin's team role theory of effective groups. *Journal of Management Development*, Vol. 18 Iss: 8, pp.652 - 665.