A Framework for Evaluating Software Development Methods

Virgil H. Cook, Jr. Thomas H. Hartrum James W. Howatt Duard S. Woffinden

Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

Abstract

We devise a framework for gathering data from software projects so we can compare different development methods and tools, to determine those that enhance developer productivity and product quality. We define criteria for selecting metrics to characterize the software development process. We then review both quality and productivity candidate metric classes. We describe a prototype for gathering data and analyzing productivity across projects, and discuss future directions based on experience with the prototype.

1 Introduction

Selecting from among the plethora of software development techniques available today represents a puzzling problem as selecting a candidate to run for president. We hear testimonials on how well certain techniques worked on the speaker’s projects, but we wonder how well they will perform in our development environment. Currently, the only way to answer our question is to look at a method’s track record or trust claims made by proponents. Although empirical studies on development projects appear often in the literature, they are so diverse that we cannot be sure their results will apply to our problems. What we really need is an objective way to select the development techniques and tools that, given our specific problem and constraints, will best satisfy our goals.

At the Air Force Institute of Technology (AFIT) we are devising a way to gather data on development projects so we can help our developers select methods and tools based on analyses of their use on projects done in-house. Specifically, we hope to answer two questions:

(1) Which development technique is best suited for the problem that I have to solve?
(2) Of the many tools available that support the chosen technique, which is the best, given the project’s constraints?

Our original goal was to gather enough data at AFIT to be able to answer these questions for any software project. However, we realized that this goal was too ambitious. Our software engineering courses use idealized, small-scale projects, in that we have only 10 to 20 weeks to complete them and the product is generally only 1000 to 5000 lines of code. Further, software products completed in support of thesis research isn’t true software engineering; only one person works on the project, eliminating the most important aspect of any project, the communication between project members.

Therefore, we narrowed our goal to that of answering the questions above for the types of software projects we do at AFIT. We hope our research results will provide insight that can be extended to “industrial-strength” software development projects.

In the next two sections we discuss the types of measures we need to quantify software project and product characteristics. In Section 4, we describe a prototype tool for gathering, storing and analyzing such metrics. The last section describes how we plan to expand on the prototype to fully characterize and analyze development efforts.

2 Productivity Measures

Why should we be concerned with productivity in an environment that certainly is not concerned with making money? To our students, time is more precious than money. Tools that help the student use time more productively result in time that can be used for other endeavors. Therefore, we want to be able to direct our students to tools that will make them most productive.

Jones [7, 8] includes time as a major component of his speed measure—project characteristics measurable per unit time, like documentation pages per month or lines of code per programmer-month, or even functions implemented per month.

He also defines two additional measures of productivity: cost and yield. Cost as its name implies, reflects the monetary value of the product produced or the expense of producing the product. Yield is a count units of product, such as pages per docu-
ment, or pages per function, or even lines of code per function, as well as errors per test case and corrections per module.

At AFIT, we rarely concern ourselves with the dollar cost of producing projects and theses. However, both speed and yield are important in a program that sets strict limits on the time our developers have to complete their work. Therefore, we want to be able to effectively measure both of these properties.

We currently have students working on theses to select the metrics to be used as productivity quantifiers. The Function Points measure [1] was chosen to reflect both speed and yield for a prototype productivity system developed by Cook [3]. Justification for this measure is given with a description of the prototype in Section 4.

3 Quality Measures

While we want to increase the productivity of our students, we also want their products to be of good quality. Many of the thesis products are software development tools themselves, to be used either by future AFIT students or by our sponsors in DoD. Therefore, we want to be able to quantify the "goodness" of our students' products, so we can identify those methods and tools that enhance software quality.

Of the many contributors to program quality/complexity, the following six are most described.

Control flow. Because of the emphasis on "structured programming" over the last twenty years, control flow complexity seems to get the most attention of all the properties. Proper characterization of control flow must include more than just path counts and nesting measures. We plan to implement the control flow characterization provided in [6] which provides a set of measures that fully reflects control flow properties.

Data flow/data dependency. After control flow measures, data flow has probably received the most attention. Understanding how assignments to one variable are affected by values of other variables is important in program maintenance. Bieman [2] describes a model of program data dependency from which such measures can be defined and computed.

Modularity. Whether we use top-down decomposition, object-oriented design, or any of the many other design techniques, the modularity of the resulting program or system affects both its maintainability and quality. System-level measures such as Henry and Kafura's Information Flow Metric [5] and fan-in/fan-out types of measures for structure charts are possible methods of reflecting the degree of modularity of a software design.

Data structures. Although most attention is focused on executable program code, the way in which abstract data types' data structures are implemented also affect the understandability, and therefore the maintainability, of the program. Measures such as those proposed by Tsai [12] should reflect of the complexity contributed by data structures.

Length. Although the size of a software product is a very poor indicator of its quality, one can certainly expect to expend more resources developing and debugging a million-line system than a thousand-line program. The most popular measure for the size of a program has been the count of its lines of source code. However, Levitin [9] argues that a token count such as provided by Software Science's length measure [4] is a much better reflection of size.

Readability. This property is perhaps the least easily quantified. It includes characteristics such as indentation, spacing and variable naming for programs and graphical versus textual representations of specifications and designs. These characteristics are more difficult to quantify than the previous five properties, and remain quite subjective. (Developers of pretty printers and other source code formatters incorporate readability principles in their design, so many of these issues may be taken care of that way.)

Code is not the only product we want to measure. Since code quality can be affected by specification and design quality, we plan to measure properties of these, too. Several thesis students are looking into metrics that capture most of the properties described above, so they can be incorporated into the development tools to provide feedback about the quality of the software as it is being developed, instead of after it has been finished.

4 Prototype Evaluation Tool

Cook's stated goal in developing the prototype was to develop a framework within which to measure and analyze software development productivity...to determine which software engineering methodologies are effective and which are not. [3, p. 43]

To achieve this, Cook determined which aspects of the development process to quantify, designed a method for storing this data, and developed a way to compare productivity data on specified properties across projects.

Before deciding which properties of the development process to measure, Cook first defined guidelines for evaluating the measures to be used. His choice of productivity and quality measures was limited because he had no tools to compute them automatically. He had to rely on data gathered from manual computations by students who developed the measured software. This placed four restrictions on the metrics he chose:

1. The metrics had to be well-defined, so when different students computed the same metric for the same product, their results would be the same.
2. The measures had to be easy to compute. Student motivation for computing accurate metrics would drop drastically if they were faced with time-consuming, difficult computations.
3. The measures had to be widely applicable. To compare differing methods and programming languages, results from one application have to be comparable to those of another; otherwise, comparisons between the methods would not be meaningful.
4. Finally, since he did not have the time to develop and validate his own measures, he wanted to select measures presented in the open literature for which there was evidence that the measures accurately reflected the properties being measured.
Additionally, two desirable criteria were identified:

1. The measures should be intuitively appealing. Although a measure may correlate well with some property, unless it seems reasonable, it may have the same impact as the "price of rice in China".

2. The measures should be applicable in all phases of the software development life cycle. The quality and timeliness of requirements specifications and design documents are often more important than that of software.

Using the restrictions to evaluate candidate measures, Cook selected Albrecht's Function Point measure [1]. This measure is computed by first determining the occurrences of five system properties, weighting the occurrence counts, and summing the weighted values. The properties are (1) external inputs, (2) external outputs, (3) logical internal files, (4) external interface files, and (5) external inquiries.

The sum is supposed to be further adjusted by ±25 per cent depending on the system's complexity. (See Figure 2 in [11, p. 3] for a description of computing the complexity factor). However, Cook felt that the complexity adjustment computation was too subjective to be uniformly applied by different students. He chose, instead, a measure that is more rigorously defined—McCabe's Cyclomatic Complexity [10]. This measure satisfies all the requirements except being applicable across the entire life cycle, since it is computed solely from source code. However, the need for an objective measure overshadowed this limitation.

Cook wanted to be able to include information on projects for which he may not have the source documents to compute the function points measure. Since most projects define size by lines of code (LOC), he chose to record converted LOC for a project. Cook uses a conversion factor defined by Jones [8] that normalizes LOC for each of the major programming languages to equivalent lines of assembly language. This allows comparisons across projects using different implementation languages.

Cook also needed to characterize the productivity of project participants. For obvious reasons he chose as one measure the amount of time each student spent on the project. However, the time spent by two students, even using the same tool, may differ depending on their experience. Thus, Cook included a factor in the collected data to reflect a student's experience with the languages and tools and environments he is using in his project.

Cook defined his database to hold all this data using a commercially available database package for microcomputers. He added an interface to a statistics package so analyses by project factors could be done easily.

He gathered only enough data to be able to demonstrate the capabilities of his database and analysis programs. We are currently planning to gather data on most class and thesis projects so we have enough data to make meaningful comparisons.

5 Conclusions

We have taken the initial steps in instrumenting the software development process at AFIT. We used Cook's thesis results as the springboard to selecting and validating metrics and choosing methods of characterizing productivity factors. When current thesis research is finished in December, we will have tools to compute measures and to gather other productivity data automatically, replacing the limited manual computations used for the prototype. These additional capabilities will allow us to more accurately capture those properties of the software development process that both enhance and deter productivity and quality. Once we have validated our characterization of the process, we will begin using different methods and tools to evaluate their usefulness in our environment.

While we realize that the results we gather for the student projects at AFIT may not apply to "industrial strength" development projects, they should provide insight to identifying the tools and methods to use for given applications in given environments.

References


