Medwell Online
Asian Journal of Information Technology 7 (11): 484-488, 2008
ISSN: 1682-3915
© Medwell Journals, 2008

Modeling Software Reuse in Traditional Productivity Model

E. Nwelih and I.F. Amadin
Department of Computer Science, Faculty of Physical Sciences,
University of Benin, Benin, Nigeria

Abstract: Productivity is expressed as a ratio of output-input and helps to identify underutilized resources. Software reuse can be used to increase productivity. In this research, the traditional software productivity model is remodeled to include reuse and the impact of reuse on productivity is quantified.

Key words: Productivity, model, reuse, traditional, software, remodeled

INTRODUCTION

Software reuse is one of the most promising approaches for increasing productivity. By reusing existing software or components, one can avoid downstream costs of maintaining additional code and increase the overall quality of the software product and productivity.

To analyze and model productivity, the resource attributes such as personnel must be considered. The most commonly used model for productivity measurement expresses productivity as a fraction of the process output influenced by the personnel divided by personnel effort or cost during the process i.e., productivity is viewed as a resource attribute and captured as an indirect measure of a product attribute (Fenton and Pyleeger, 1997; Pressman, 2001).

According to the Japanese notions of spoilage, productivity is viewed as a measure of how much effort is expended on fixing things that could have been put right before delivery. Some software engineers compare the cost of fault prevention with the cost of fault detection and correction (Fenton *et al*., 2001). Many software-engineering methods proposed and developed in the last 25 years provide rules, tools and heuristics for producing software products. Almost invariably, these methods give structure to the products; it is claimed that this structure makes them easier to understand, analyze and test. The structure involves 2 aspects of development:

- The development process, since certain products need to be produced at certain stages
- The products themselves, since they must conform to certain structural principal

In particular, product structure is usually characterized by levels of internal attributes such as modularity, coupling, or cohesiveness. Brooks (1975) and Yourdon (1979) assumed that good internal structure leads to good external quality.

This study is concerned with remodeling of the traditional software productivity model to include reuse there by quantifying the impact of reuse benefit on productivity.

Productivity can be defined as a complex attribute of software and people. It is measured indirectly as a composition of measurable attributes, such as size and quality.

Software reuse is the use of existing software artifacts or knowledge to create new software.

Productivity is typically calculated as size of the system divided by cost spent to develop it, for some measure of size and cost. Keeping the size of a system constant, increasing productivity will result in a reduction in cost. This concept of reuse is important when size is provided as input to effort, cost and productivity models. The reuse of software (including requirements, designs, delimitation, test data, scripts as well as code) improves our productivity and quality, allowing us to concentrate on new problems rather than continuing to solve old ones again. However, traditional notions of productivity, measured as size of output divided by effort expended, must be adjusted to accommodate these reuses. We must include in size measurement, some method of counting the reused products counting reused code is not as simple as it sounds. It is difficult to define formally what we mean by reused code. We some times reuse whole programs without modification (a module, function, or procedure). And we often modify that unit to some extent. However we account for reused code, we want to distinguish a

Corresponding Author: E. Nwelih, Department of Computer Science, Faculty of Physical Sciences, University of Benin, Benin, Nigeria

module with one modified line from a module with 100 modified lines. Thus, we consider the notion of the extent of reuse measured on an ordinal scale by NASA/ GODDARDS Software Engineering Laboratory, in Software Productivity Consortium (1995). As follows:

Reused verbatim: The code in the unit was reused without any changes.

Slightly modified: Fewer than 25% of the limes of code in the unit were modified.

Extensively modified: A 25% or more of the lines of code were modified.

New: None of the code comes from a previously constructed unit.

The classification can be simplified and reduced to 2 levels, New (level 3 and 4) or Reused (level 1 and 2). A typical FORTRAN Project at the Software Engineering Laboratory include 20% reused line of code, while a typical Ada project include 30% reused line codes.

The inclusion of previously constructed code in new software will certainly increase the value of the productivity equation, but unless the code is executed, we have no real increase in productivity. Thus, productivity as a measure defined by the productivity equation fails to satisfy the repetition condition for measurement.

We can demonstrate this failure as follows: when measuring programmer productivity, entities and individuals producing specific programs. Suppose that P is the entity "Emma producing program A", that it takes Emma 100 days to complete A and that A is 5000 lines of code. According to the productivity equation, Productivity (P) = 50LOC per day suppose that Emma adds another copy of program A to the original one, in such a way that the second copy is never executed. This new program, A, has 1000 LOC but its functionally equivalent to the old one. Moreover, since the original version of A was already tested, the incremental development time is nil, so that the total time required to create A' is essentially equal to the time required to create A. Intuitively, we know that Emma's productivity has not changed. However, let P' be the entity "Emma producing program A". The productivity equation tells us that: Productivity (P') = 100 LOC per day.

The significant increase in productivity (according to the measure) is a clear violation of the representation condition, telling us that the productivity equation does not define a true measure, in the sense of measurement theory (Fenton *et al.*, 2001; Pressman, 2001).

Hewlett-Packard, a market leader in electronics is also a pioneer in software measurement. In 1983, in response to the recommendations of a task force that focused on software productivity and quality, the company set up teams to address short-term productivity improvement through tools as well as long-term productivity improvements through the use of a common development environment. Hewlett-Packard created a software engineering laboratory, which in turn created a software metrics council. The metrics council, representing the major divisions of the company, proposed a set of metrics standards that addressed 5 issues. The 5 issues are:

Size: Low big is the software that is produced?

People, time and cost: How much does it cost to produce a given piece of software and associated documentation?

Defects: How many errors are in the software?

Difficulty: How complex is the software to be produced and how sever are the constraints on the project?

Communications: How much effort is spent in communicating with other entities?

The Metrics recommended addressing the issue:

Size: Non-comment source statements.

People, time and cost: Payroll month.

Defects: Problem or error in the software.

Difficulty: A number between 35 and 165, with 165 being the most difficult, the number is determined by ensures to a questionnaire.

Communications: Number of interfaces that a lab team project has.

Over time, the defect measurements have expanded to include number of defect severity, duplicate reports for the same defect and efficiency of testing defect removal. Also, user satisfaction has been monitored by looking at user defect reports and the number of user requests for enhancement (Fenton *et al.*, 2001; Pressman, 2001). This helped Hewlett-Packard address the corporate goals put forth by president Young (1986). Fenton *et al.* (2001) has reported a significant measurement effort that addresses the effectiveness of inspections at Hewlett-Packard describes a quantitative investigation of

reuse in the corporation. Gaffney and Durek (1989), reports the effects of reuse on software productivity at 2 divisions of Hewlett-packard. The data were collected during a formal process called a reuse assessment. By implementing reuse, the manufacturing productivity division exhibited a 51% defect reduction and a 57% increase in productivity. Similarly, the Sam Diego Technical Graphics division reduced its defects by 24% and increased productivity by 40% it also reduced time to market by 42%.

MATERIALS AND METHODS

Examine software reuse model as relates to productivity: Gaffney and Durek (1989) propose productivity model for software reuse, which is:

$$P = 1/c = 1/((b-1)R + 1)$$

where:

- p = Productivity
- c = Cost of software development for a given product relative to all new code (for which c = 1)
- R = The proportion of reused code in the product
- b = The cost relative to that for all new code. b = 1 for incorporating the reuse code into the new product

Using reusable software generally results in higher overall development productivity; however, the cost of building reusable components must be recovered through many reuses.

Favaro (1991) utilized the model from Fenton *et al.* (2001) to analyze the economics of reuse. Note at this point that Fenton *et al.* (2001) model is the same as that of Gaffney and Durek (1989). The relevant variables and formulas are:

$$Rc = (b + (E/N) - 1)R + 1$$
$$Rp = 1/Rc$$
$$No = E/(1-b)$$

where:

- R = Percent of code contributed by reusable components
- b = Integration cost of reusable component as opposed to development cost
- Rc = Relative cost of overall development effort
- Rp = Relative productivity
- E = Relative cost of making a component reusable
- No = Payoff threshold value (number of reuses needed to recover all component development cost)

Favavro's research team estimated the quantities R and b for an Ada-based development project. They found it difficult to estimate R because it was unclear whether to measure source code or relative size of the load modules. The parameter b was even more difficult to estimate because it was unclear whether cost should be measured as the amount of real-time necessary to install the component in the application and whether the cost of learning should be included.

Poulin (1995) present a set of metrics used by IBM to estimate the effort saved by reuse. Although the measures used are similar to the productivity model of Gaffney and Durek (1989). Poulin named the metrics from a business perspective and they provide a finer breakdown for some calculations. For example cost is broken down into development cost and maintenance cost.

$$\text{Product reuse percent} = \frac{\text{RSI}}{(\text{RSI}+\text{SSI})} \times 100\%$$

$$\text{Development cost avoidance} = \text{RSI} \times 0.8 \times (\text{new code cost})$$

$$\text{Service cost avoidance} = \text{RSI} \times (\text{error rate})$$

$$\text{Reuse cost avoidance} = \text{Development cost avoidance} + \text{Service cost avoidance}$$

$$\text{Reuse value added} = \frac{(\text{SSI}+\text{RSI}+\text{SIRBO})}{\text{SSI}}$$

$$\text{Additional development cost} = (\text{relative cost of reuse}^{-1}) \times \text{code written for reuse by other} \times \text{new code cost}$$

where:

- SSI = Shipped Source Instructions
- CSI = Changed Source Instructions
- RSI = Reused Source Instructions
- SIRBO = Source Instructions Reused by others
- Cost per LOC = Software development cost
- Error rate = Software development error rate
- Cost per error = Software error repair cost

RESULTS AND DISCUSSION

Proposed software productivity model that includes reuse: The following are the existing model description of the current model as it obtains productivity measure.

$$\text{Productivity} = \frac{\text{LOC produced}}{\text{Person months of effort}}$$

Productivity can also be measured as some form of: amount of output/effort input (Fenton and Pyleeger,1997).

The general notion is an economic one, where businesses or markets are judged by comparing what goes in with what comes out.

The output (LOC produced) is the final product: The input (person month of effort) is the number of person month used to specify, design code and test the software.

$$\text{Productivity} = \text{Size/effort size related measure}$$

(Fenton and Pyleeger, 1997)

$$\text{Productivity} = \frac{\text{No. of functional points implanted}}{\text{Person-months}}$$

Function-related measure (Abrecht and Gaffney, 1983; Pressman, 2001).

The process of using these models without the consideration of reuse is a problem. In Nigeria for instance, with special references to IT industry, this traditional method involved with management assessment of programmer productivity process contain certain draws back on the system. So based on all the issues raised above, we now decided to model productivity to include reuse as shown in Fig. 1.

Reuse: Assuming that 1% of the objects will be reused from previous projects.

$$\text{New object points} = \frac{(\text{Object points}) \times (100\text{-r})}{100}$$

Length: Total length (LOC) NCLOC + ClOC Density of comments in a program = CLOC/LOC (Fenton *et al.*, 2001).

Functionality: Unadjusted function point count (UFC)

$$\text{UFC} = \sum_{i=1}^{(15)} \left((\text{No. of items of varietyi}) \times \text{weight}\right)$$

Technical Complexity Factor (TCF) = 0.65+0.01 (sum (Fi)).

The factor varies from 0.65 (if each Fi is set to 0) to 1.35 (if each Fi is se to 5) (Fenton *et al.*, 2001) the final function point calculation is:

$$\text{FP} = \text{UFC} \times \text{TCF}$$

Complexity: Big O notation (n,n^2 nlogn, nlogn^2) where a and b are constant mode s is size measured in thousands of delivered sources instructions (KDSI).

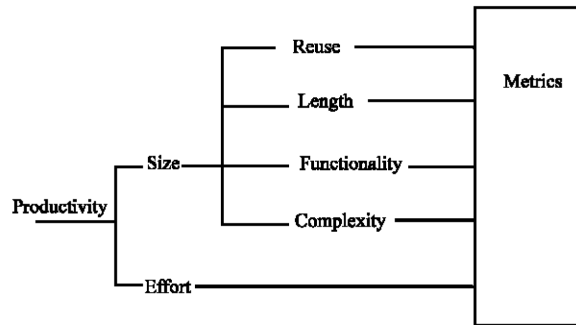

Fig. 1: Productivity model including reuse

Note: LOC = Lines of code. CLOC = commented line of code (Line of code is used not as a measure of length but as a measure of effort, utility, or functionality).

$$\text{Productivity} = \sum_{i=1}^{n} \frac{(r_i + f_i + l_i + c_i)}{\sum_i}$$

where:

r_I = Reuse
F_i = Functionality
L_i = Length
Σ_i = Effort
C_i = Complexity

CONCLUSION

In this research, we looked at reuse as an important aspect of productivity, when size is provided as input to effort, cost and productivity model. Appling reuse, software productivity cost appears to be quite large. With the reuse model stated in this study, productivity measure becomes effective.

REFERENCES

Abrecht, A. and J. Gaffney, 1983. Software function, source lines of code and development effort prediction: A software science valiation. IEEE. Trans. Soft. Eng., Se-9 (6): 639-648. www.informatik.uni-trier.de/~ley/db/indices/a-tree/a/Albrecht:Allanj=html-4k.

Brooks, F., 1975. The Relationship Between Software Development Team Size and Software Development. ACM, 52 (1): pp: 141-144. iablog.sybase.com/paulley/2009/01/fred-brooks-is-still-right/-32k.

Favaro, J., 1991. What Price Reusability? A Case Study. Ada Lett. (spring), pp: 115-124. Frakes.CS.vt.edu/6704SO6.htm-12k.

Fenton, N.E. and S.L. Pyleeger, 1997. Software metrics a rigorous and practical Approach. Chapman and Hall, London. www2.umassed.edu/SWPI/processBiblogr-aphy/bib-measurement.html-11k.

Fenton, N.E., M. Neil and D. Marquez, 2001. A New Bayesian Network Approach to Relibility Modelling. www.agenarisk.com/resources/technology-articles/MMR_A%20new%20BN%20approach%20to%20Relibility%20mod.

Gaffney, J. and T. Durek, 1989. Software Reuse-Key to enhance Productivity: Some quantitative models. Inf. Sofware Technol., 31 (5): 258-267. Archive.nyu.edu/bitstream/2451/14190/3/1s-97-32.pdf.txt-89k.

Poulin, J.S., 1995. Populating Software Repositories: Incentives and Domain-Specific Software. J. Syst. Software, Elsevier Science, New York, 30 (3): 187-199. http://citeseer.ist.psu.edu/poulin94populating.html.

Pressman, 2001. Software Engineering a Practical Approach. Loadingvault.com/search.php?q=software+engineering+roger+pressman+solution+manual-28k.

Yourdon, E.N., 1979. Classics in Software Engineering. Yourdon Press. portal.acm.org/citation.cfm?id=1010904.1010910.

Young, R.A., 1986. Simulation of Human Retinal Function with the Gaussian Derivative Model. CVPR, pp: 564-569. Line:ee.hawil.edu/~treed/ispg/DGT_demo/ref.html-3k.