A context-dependent cost effectiveness indicator for software development

Hakan Erdogmus
NRC Institute for Information Technology
Ottawa, Canada
Hakan.Erdogmus@nrc.gc.ca

Abstract

Product quality, development productivity, and staffing needs are main cost drivers in software development. In particular the tension between productivity and quality makes it difficult to gauge the relative cost-effectiveness of a software development using separate quality and productivity measures. The paper proposes a one-stop cost-effectiveness indicator that resolves the tension between these two cost drivers, by combining productivity, quality, and staffing needs through an economic criterion. Context-specific instances of the indicator can be used as key performance measures to rank comparable software projects or as dependent variables to evaluate development techniques whose effectiveness hinge on the tradeoff between their relative productivity and quality benefits. The use of the indicator is demonstrated by applying it to the findings of two empirical studies.

1. Introduction

Tradeoffs between development productivity and product quality make it hard to assess the cost-effectiveness of software development, both across software development projects and across development techniques and practices that these projects employ. Previous research suggests substantial variability in quality and productivity [1], as well variability in the tension between them [2]. A high-level performance indicator can reconcile this tension. This paper develops such an indicator that relies on and aggregates software development's main cost drivers [3] -- team productivity, staffing needs, and external product quality -- into a single coherent quantity. The indicator, called *breakeven multiple*, allows comparison among projects and development techniques that share a common context and based on their relative cost-effectiveness. The indicator incorporates productivity through productivit's impact on development costs and product quality through quality's impact on defect removal or rework costs [4]. The isolation of rework effort from development effort is central to the indicator.

The paper illustrates the derivation of breakeven multiple using the Goal-Question-Metric paradigm [5], identifies its measurement needs, discusses its main characteristics, and demonstrates its application in empirical studies of software engineering.

The measurement of productivity and quality has been an elusive and controversial endeavor. The main problem centers on the selection of appropriate, meaningful, and portable measures; and several different approaches exist. Breakeven multiple does not specify the exact measures to use, but rather it assumes that the context determines them. Thus it can be specialized to meet the particular needs of a given context.

The indicator focuses only on external quality, or more specifically, the degree at which a deployed product is free of issues identifiable by its customers. This aspect of external quality is often captured by a category of product metrics collectively referred to as *defect rate* or *density* [6]. Measuring external quality involves properly counting issues, whose definition may vary from context to context. Florac [7]and Kan [8] provide comprehensive guidelines for measuring external quality.

All approaches to gauging productivity of a production process must determine the production output's volume. An output measure is also necessary to determine external quality expressed in terms of a defect density metric. In software development since the concept of output is not tangible, proxies are necessary. Output proxies attempt to measure either the *size* of the software code artifacts produced (in *lines of code*, and variations thereof) or the amount of *functionality* delivered (in *function points* or variations thereof). Both size-based proxies and functionality-based output proxies are commonly used (often together [9]), and both have been widely criticized for their shortcomings [10-13]. Empirically based schemes have also been proposed for conversion from functionality-based to size-based measures [14]. Kitchenham and Mendez [15] provide a survey of

software productivity measures with a discussion of underlying output measures.

Economic metrics for software development have existed since the late nineties. Erdogmus [16] developed a cost-benefit model based on net present value to compare two software projects. Muller and Padberg [17] adapted this model for evaluating extreme programming projects. Erdogmus and Williams [18] later combined net present value with breakeven analysis to derive a cost-effectiveness metric to analyze the economic feasibility of pair programming. Padberg and Muller [19] used a similar approach in their own analysis of the same practice. Wagner [4] recently proposed an economic efficiency model for quality that aggregates costs and benefits of quality activities into a return-on-investment metric.

The work presented here builds on the metric defined by in [18] for comparing two practices. It both generalizes and simplifies this metric, making it suitable for use in multiple contexts as a high-level performance indicator and allowing more robust, multi-way comparisons. Using the same philosophy as Wagner to account for cost of quality, it recasts the metric's derivation in Goal-Question-Metric [5] terms, leading to explicit identification of its building blocks and facilitating its customization and adaptation.

2. Basic Concepts

A *project* is work undertaken by a team. A project's ultimate output is a finished software product with no known issues that require resolution. A project comprises development and rework activities.

Development refers to all work that leads to the initial external release of parts or whole of a usable, but not necessarily perfect, product. Development results in a product that may contain issues requiring resolution. We refer to the output of development as the project's *nominal output*.

Rework refers to all work that resolves any issues in the nominal output. Rework transforms a released product into a finished product free of externally discovered issues. We refer to the output of rework as the project's *real output*.

Product quality, or simply *quality*, refers to absence of issues in a project's nominal output. Therefore we focus on *external* quality rather than *internal* quality. Think of an issue as a defect, or undesirable property or behavior, that incurs some latent cost, or prevent the benefits of a product from being realized as intended. Issues are discovered post-development by external parties and require resolution. Rework represents the cost of resolution, and thus cost of quality.

Where development stops and rework starts depends on the context and sometimes is a matter of interpretation. It is reasonable to assume that a project does not intentionally release output that is known to contain issues to its customers. Therefore quality-assurance related lifecycle activities that precede the external release of a product are considered by the above definitions as part of development, and not rework.

Schedule is the duration of an activity, measured in calendar time. *Unit schedule* is the time unit used to measure schedule. *Effort* is the labor cost of an activity, measured in person-time, where time is expressed as a multiple of unit schedule. Effort-schedule conversion depends on salary-adjusted staff load. For example, if a project's average salary-adjusted staff load over a given period is 2.5 times a base salary and the unit schedule is 1 day, then for that period, 1 project day is equivalent to 2.5 person-days of effort.

A project need not release its product all at once, nor have a pre-determined schedule. In a variety of situations, development and rework often proceed concurrently through an intertwined pipeline. In such as situations, it is more realistic to assume that a project produces nominal output continually in small increments, or even continue to perpetuity with no definite end point. Central to an economic model that incorporates product quality as a factor is the ability to separate effort allocated to each kind of activity, and not the activities' specific ordering or degree of overlap.

A pair of projects P_i and P_j are *comparable* if there exists a common output measure such that it is possible to measure the nominal output of P_i and P_j in terms the common output measure and the measure is interpreted in the same manner and on the same scale for both projects. Projects in general are comparable when they have similar contexts, in other words, similar organizational, technical, and domain, and complexity characteristics.

If under the same development effort the nominal output of project P_i is greater than that of a comparable project P_j, P_i has higher *nominal productivity* than P_j.

3. GQM model

The end goal is to conceive a simple, generic cost-effectiveness indicator M for comparable projects such that M can be customized according to a given context. Specific instantiations of M will not be universally meaningful, but will be meaningful only within an identifiable context and under certain assumptions: if these conditions hold and if $M(P_i) > M(P_j)$, we should

be able to objectively claim that project P_i is more *cost effective*, or economically more efficient, than project P_j.

Since *M* is intended to be meaningful only internally within the context for which it is conceived, all of its instantiations will necessarily be *context-dependent*.

The indicator *M* should be interpretable on a ratio scale for sound ranking of projects.

The Goal-Question-Metric (GQM) paradigm [5] allows a top-down identification of a set of cost drivers that need to be measured to achieve the goal. The cost drivers are then mapped into a set of concrete derived measures that quantify them. The concrete measures are subsequently aggregated into an indicator, this time in a bottom-up fashion.

In GQM terms, the goal can be cast as:

To analyze
the cost effectiveness of a project
for the purpose of
ranking it relative to another project
with respect to
project's cost structure
from the viewpoint of
a manager
in the context of
a set of comparable projects.

Figure 1 illustrates the GQM model together with a set of concrete measures at the bottom-most level. The two questions (**Q**) relate to the two major components of a software development project's cost structure: development cost and rework cost. The abstract metrics at the next level (**M**) derive from these two questions.

Development cost captures the nominal cost of production, and can be determined if information about two cost drivers is available:

- *Nominal productivity*: How fast is the project team able to produce and release a nominal product to an external customer?
- *Staffing*: How is the project staffed through its lifetime?

Rework cost on the other hand captures the cost of quality, and can be determined if information about three cost drivers is available:

- *Product quality*: What is the quality of the nominal product?
- *Rework productivity:* How fast is the project team able to rework the nominal product to resolve all outstanding issues for an acceptable finished product?
- *Staffing*: Again, how is the project staffed through its lifetime?

Therefore, the GQM model identifies three general areas of measurement: *staffing*, (*team) productivity, and (product) quality.*

The bottom-most level in Figure 1 consists of a set of concrete derived measures, with a one-to-one correspondence to the cost drivers from the metrics level (**M**) of the GQM model immediately above. The "+" and "–" signs indicate respectively whether the cost driver at the GQM model's metric level is directly or inversely related to the corresponding derived measure below it.

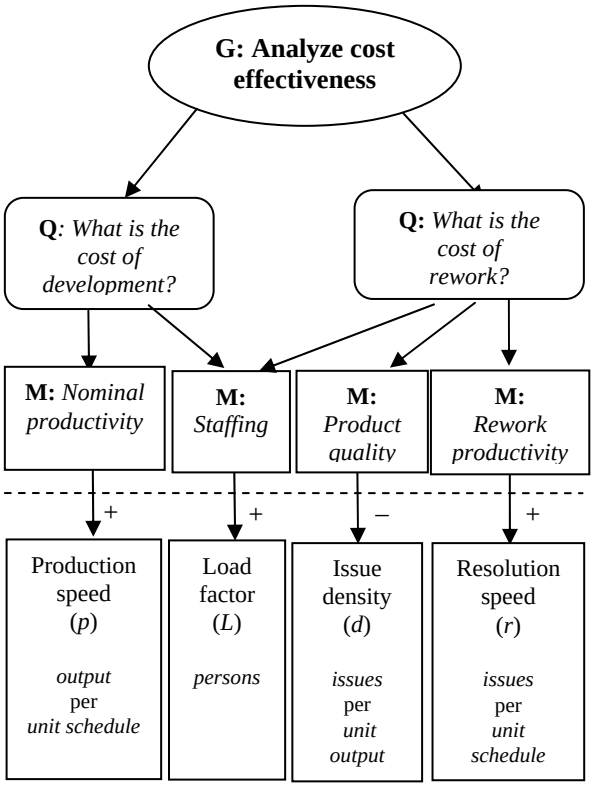

Figure 1. A GQM model of cost effectiveness.

3.1. Base measures

The concrete measures that map to the metrics level of the GQM model are not elementary in terms of being directly observable. They are derived from a set of base measures: *nominal output, development schedule, rework schedule, issue count, and staffing profile.* Each of these base measures needs to be tracked and recorded.

The top two levels of Figure 2 illustrate the dependence of the concrete derived measures from the bottom-most level of Figure 1 on the base measures.

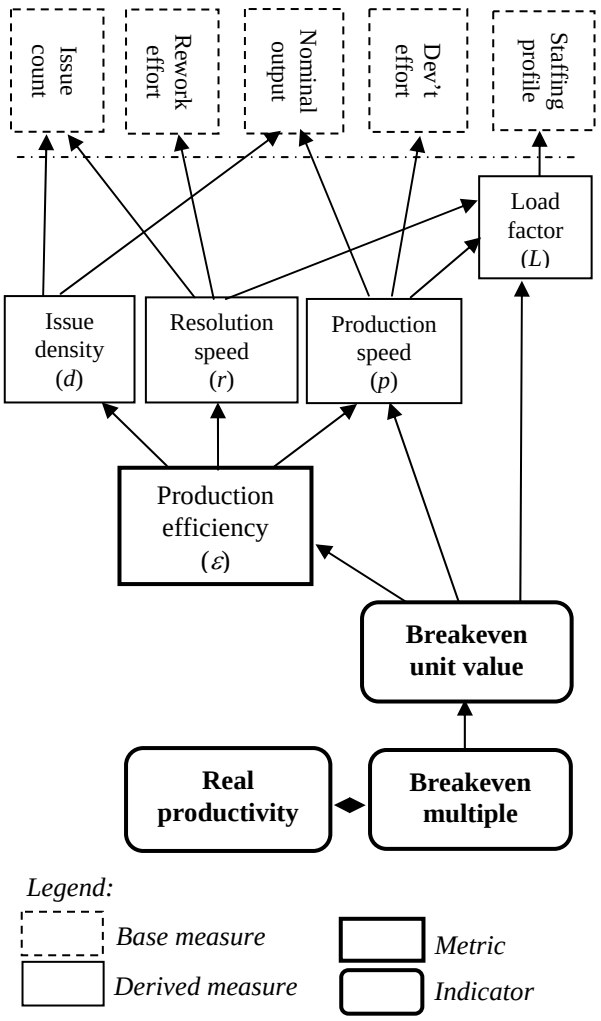

Figure 2. Hierarchy of cost-effectives measures.

Development schedule and rework effort are straightforward. They represent total effort dedicated to development and rework activities respectively, measured in person-time.

Nominal output refers to the output of development, before rework. Across all comparable projects, nominal output should be counted such that for values that are sufficiently large to yield averaging out effects, nominal output is interpretable on a ratio scale.

Issue count refers to the number of issues discovered by an external party in a released nominal product. Within the same project, issues should be counted in an as uniform manner as possible such that issue counts that are sufficiently large will produce averaging out effects, making them interpretable on a ratio scale. Otherwise adjustments may be necessary to normalize issues of differing complexity according to their estimated resolution effort.

Staffing profile is the salary loading of project as a function of schedule. It allows calculation of the project's total labor cost for any given period. Salary loading at a given period is specified as a multiple of a base compensation level or base salary.

The ratio interpretations are desirable for nominal output and issue count at realistic and sufficiently large scales to derive sound quantities that admit standard arithmetic operations. Otherwise any derived quantity would not allow meaningful comparison among projects. The appropriate choice of a specific measure and granularity level depends on the context.

3.3. Load factor

Load factor (L) quantifies the average staffing of a project in terms of a multiple of a base salary. Load factor is derived from the project's staffing profile.

The unit of load factor is persons, where a person equals the base salary and k times the base salary equals k persons.

Figure 3 shows the staffing profile of two projects with a constant salary-adjusted staffing load of l_1 and l_2, respectively. For these simple cases, the projects' load factors respectively equal their constant salary loads: $L_1 = l_1$ and $L_2 = l_2$.

Table 1 shows how to calculate the salary-adjusted staffing load for a given period from the salaries of the team members active full-time in that period. The base salary is \$100K for both projects. The base salary is part of the context and cannot vary for the projects to be comparable. Project 1 has 3 active members on the project at that period with salaries \$80K, \$100K and \$120K, resulting in individual salary loads of 0.8, 1, and 1.2 respectively. Similarly Project 2 has 4 active members with individual salary loads of 0.8, 1.2, 1.2, and 1.5. The project's salary load for the period is therefore 3.0 persons for Project 1 and 4.7 persons for Project 2.

Part-time activity requires only a straightforward adjustment proportional to percentage time on project when calculating the individual salary loads. For example, if the fourth member of Project 2 were on the project 50% of the time, the project's salary load would be $0.8 + 1.2 + 1.2 + 1.5(.5) = 3.95$.

Figure 4 depicts the calculation of load factor under variable salary load for two projects with different salary profiles and schedules. When the salary load varies in time, a project's load factor L is given by the average staff load over time: it equals the area under the staffing profile A divided by the total schedule T.

Table 1. Calculation of salary loads for a given period .

	Project 1 salaries (3 members)	Project 2 salaries (4 members)
Base salary = $100K	$80K	$80K
	$100K	$120K
	$120K	$120K
		$150K
Total salary	$300K	$470K
Salary load	k_1 = 3.0 persons	k_2 = 4.7 persons

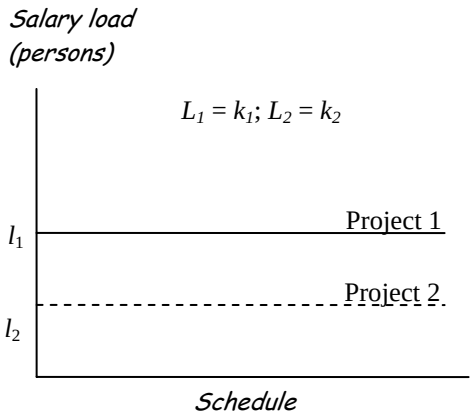

Figure 3. Load factor under constant staffing profile.

3.2. Production speed

Production speed (*p*) quantifies nominal productivity. Capturing the development component of team productivity, production speed is the average delivery speed of nominal output by the project as a whole.

Production speed is computed from development effort, nominal output, and load factor:

$$p = \frac{\text{Nominal output}}{\text{Development effort}} \times L \qquad \text{(output/schedule)}$$

3.3. Issue density

Issue density (*d*) quantifies product quality. Capturing the level of rework that the released nominal product requires, issue density is the average issue count of unit of nominal output:

$$d = \frac{\text{Issue count}}{\text{Nominal ouput}} \qquad \text{(issues/output)}$$

The smaller the issue density, the higher the product quality.

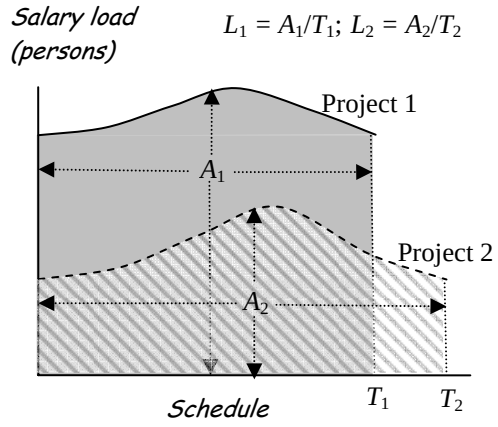

Figure 4. Load factor under variable staffing profile.

3.4. Resolution speed

Resolution speed (*r*) quantifies rework productivity. Capturing the rework component of team productivity and the unit cost of reduced quality simultaneously, resolution speed is the average rate at which the project as a whole resolves issues in the nominal product:

$$r = \frac{\text{Issue count}}{\text{Rework effort}} \times L \qquad \text{(issues/schedule)}$$

4. Derivation of cost effectiveness indicator

With the concrete measures corresponding to the cost drivers at the metrics level of the GQM model and their underlying base measures in place, it is now possible to proceed in a bottom-up fashion to arrive at the targeted cost effectiveness indicator, as advocated by GQM. The portion of Figure 2 below the dashed line illustrates the three-step strategy used to aggregate the derived measures first into an efficiency metric and ultimately into a high-level indicator.

4.1 Production efficiency

To begin, we combine three of the cost drivers -- nominal productivity, product quality, and rework productivity -- into a simple metric that quantifies the schedule efficiency of production. Let's call this measure *production efficiency* (or *efficiency* for short) and denote it by ε.

Assume the project's target is to produce a nominal output of *U* units. A project having a production speed of *p* output units per unit schedule, an issue density of

d issues per unit output, a resolution speed of *r* issues per unit schedule, and a load factor of *L* persons requires a development effort of *UL*/*p* and a rework effort of *ULd*/*r* person-schedule units.

The project's total schedule is thus *UL*/*p* + *ULd*/*r*. The project's average production efficiency, ε, is a unitless metric given by the ratio of development effort to the total effort:

$$\varepsilon = (UL/p)/(UL/p + ULd/r) = r/(r + pd)$$

Thus ε is a function of *r*, *p*, and *d* only. Since its range is [0, 1], it can be expressed as a percentage. For example, if ε = .6, or 60%, then 60% of the project's effort is dedicated to development and 40% to rework.

4.2. Breakeven unit value (BUV)

Let *T* be a project's development schedule with a nominal output of *U* units. If *V* denotes the hypothetical value earned by a single unit of nominal output, then for each unit schedule the project on average earns a value of $Uv/T = (pT\varepsilon)V/T = Vp\varepsilon$.

Now suppose *S* is the base salary (in $ per person-schedule) of an average developer. If the project has a load factor of *L* persons, it incurs for each unit of schedule a cost of *SL*.

Then the average net value earned by project per unit schedule is:

$$\text{Net Value} = Vp\varepsilon - LS$$

When this net value is positive, the project turns a profit. When it is zero the project breaks even. Otherwise it operates in a deficit. We are interested in the minimum level of the parameter *V*, the hypothetical unit value, that allows the project to break even. Solving the equation Net Value = 0 for *V* yields this *breakeven unit value*, or BUV:

$$\textbf{BUV} = \min\{\ V \mid \text{Net Value} = 0\ \} = LS/p\varepsilon \quad (\$/\text{output})$$

BUV is a cost-effectiveness indicator that combines productivity and quality. Its smaller levels indicate higher profitability potential. A project with a smaller BUV breaks even more easily then a comparable project with a higher BUV.

4.3. Breakeven multiple (BM)

For easy interpretation, it is better for higher levels of a performance indicator to correspond to higher levels of the concept it is intended to capture. Breakeven unit value does the opposite: it is essentially a measure of ineffectiveness rather than of effectiveness, in the same sense issue density is essentially a measure of lack of quality rather than of quality.

In addition, breakeven unit value's dependence on the base salary *S* is unnecessary: *S* is invariant within the same context and the load factor parameter *L* already incorporates any deviation in labor costs with respect to *S*. Normalizing the base salary *S* with respect to BUV results in a more compact indicator with a more intuitive interpretation, called the *breakeven multiple*, or BM:

$$\textbf{BM} = S/\textbf{BUV} = p\varepsilon/L \quad (\text{output/person-schedule})$$

The breakeven multiple expresses the base salary *S* in terms of a multiple of BUV, but it does not depend on *S*. The base salary is BM times as big as the BUV. Since *S* is invariant, if a project's BM increases, the project requires a lower unit value to break even, and the project's cost-effectiveness and profitability increase.

5. Characteristics of breakeven multiple

5.1. Alternative interpretation

Remarkably the breakeven multiple's unit, output per person-schedule, looks like the effort equivalent of nominal productivity. Indeed BM has an alternative interpretation as a productivity indicator.

Let T be the total schedule (including rework) of a project with a BM of $p\varepsilon/L$. We can rewrite the BM as $p\varepsilon T/TL$. On the one hand, the numerator $p\varepsilon T$ denotes the real (as opposed to nominal) output of the project, that is output with no outstanding issues to resolve. The denominator *TL* on the other hand denotes the total effort. We can think of the real output divided by total effort as the project's average combined productivity, or *real productivity*:

Hence real productivity and breakeven multiple coincide, establishing equivalence between cost-effectiveness and real productivity.

5.2. Preservation of net present value

An important property of the breakeven multiple is its soundness with respect to standard financial theory. Investment alternatives are often compared based on the sum of their discounted cash flows, or *Net Present Value* (NPV) [16], rather than based simply on the absolute sum of benefits net of costs. Cash flows represent benefits when they are positive and costs when they are negative. Discounting allows accounting for time value of money by weighting cash flows that

occur further out in the future less than those that occur closer to the present time.

Remarkable, when a project accumulates both costs and benefits incrementally and continuously, its breakeven multiple derived from discounted cash flows remains unchanged.

Let c be a (positive) continuously compounded discount rate per unit schedule. Recall that the net value per unit scheudle is $Vp\varepsilon - LS$. Then the project's net cash flow for a small interval dt from time t to time t + dt equals $(Vp\varepsilon - LS)$dt. Discounting this amount t periods to the present time yields a present value of $(Vp\varepsilon - LS)e^{-ct}dt$. If the project's total schedule is T, then the project's net present value equals:

$$\mathbf{NPV} = \sum_{i = 0..T/dt} (Vp\varepsilon - LS)\, e^{-ci}dt$$

When the small interval dt approaches to 0, the NPV reduces to the definite integral:

$$\mathbf{NPV} = \int_{t = 0..T} (Vp\varepsilon - LS)\, e^{-ct}dt = (Vp\varepsilon - LS)(1 - e^{-cT})/c$$

For positive T, $1 - e^{-ct}$ is positive. Hence solving NPV = 0 for V results in the same expression for the breakeven unit value as before, where **BUV** = $LS/p\varepsilon$, and ultimately the same expression for the breakeven multiple.

5.3. Advantages

The breakeven multiple is one-stop indicator that aggregates productivity, quality, and staffing into a single, simple measure. It makes it possible to compare projects with opposite productivity and quality characteristics (low-quality, high-productivity vs. high-quality, low-productivity), thus reconciling the underlying trade-offs.

Through alternative derivations, breakeven multiple captures simultaneously cost-effectiveness and real productivity, both of which admit intuitive interpretations. It is also sound with respect to standard financial theory under the assumption of continuous incremental delivery.

Breakeven multiple requires five base measures to be collected about a project. It can be instantiated (or customized) for a given context by appropriately choosing the underlying base measure. It allows easy comparison and ranking of projects that share a common context.

5.4. Limitations

Breakeven multiple is not a portable indicator; it is context-dependent. The most serious limitation is the dependence of its unit on the unit of the particular output measure used. Thus projects having different output measures are not comparable by this indicator.

Important requirements apply to the selection of base measures to guarantee a sound indicator interpretable on a ratio scale. The base measures of nominal output and issue count should be interpretable on a ratio scale for realistically large ranges. Satisfaction of these requirements is neither easy nor obvious. Software does not admit a universal and uniform measure of output. Thus either size measures such as lines of code (low-level) and function points (high-level) or requirements-oriented measures like use-cases and extreme-programming-style stories are adopted as proxies. However, no proxy is perfect and each has its own its own advantages and disadvantages [13].

Neither limitation is particular to breakeven multiple. Finding portable, meaningful, sound measures of size, functionality, productivity and quality has been a central, but elusive endeavor of software measurement.

5.5. Uses

The breakeven multiple has two intended uses:

1) As a high-level, one-stop performance indicator inside a portfolio of comparable projects.
2) As a one-stop dependent variable in empirical studies of software development techniques, practices, or processes.

Adapting the indicator for use in empirical studies requires replacing the concept of a project by that of a technique (process or practice), interpreting the base measures as independent variables, and a slight adjustment to the GQM goal statement as follows:

> *To analyze* the cost effectiveness of technique A
> *for the purpose of* comparing it to technique B
> *with respect to* the quality of artifacts produced by groups employing the techniques, the productivity of the groups, and their staffing requirements *from the viewpoint of* the researcher *in the context of* study X.

The limitations discussed in Section 5.4 are much less severe when the indicator is used in experimental contexts where study design can control or alleviated them.

6. Application examples

The following examples demonstrate the breakeven multiple's application in two experimental contexts to

evaluate the cost-effectiveness of software development practices.

6.1. Solo vs. pair programming

The first example is a variation of the analysis of pair programming conducted by Erdogmus and Williams [18]. It focuses on measure selection, dealing with incomplete information, and the use of the load. The analysis compares the practice of pair programming with solo programming, augmenting the results of an empirical study with benchmark statistics obtained from the general literature.

The empirical study in question compared the average productivity and quality of programmer pairs (working collaboratively on a common task using a certain protocol and sharing a common workstation) and solo programmers (working alone and independently on the same task). The study measured nominal productivity using a readily adoptable production speed measure: lines of code (LOCs) per hour. Product quality measure was percentage of tests passing from a standard automated acceptance test suite, and need adaptation. The study did not measure rework productivity.

Base measures. The key is to treat the experimental groups representing the two development practices as two projects, say *Solo* and *Pair*, belonging to a common context. Since nominal productivity was measured in LOC/hour, the common output measure is lines of LOC. The unit schedule equals an hour.

To develop a complete model, we must first find a proxy for rework productivity for both practices. Then depending on the measure of rework productivity, we can convert the study's original quality measure to a quality measure compatible with breakeven multiple.

Rework productivity

Erdogmus and Williams use industry benchmark data reported in the literature to complete the model. To represent rework productivity, they substitute the Capers Jones' benchmark figure of 0.0303 defects/hour as the resolution speed of the practice *Solo*, where a defect is defined as a post-deployment fault. The definition of a defect is consistent with the definition of an issue in Sections 2 and 3. We reuse the same strategy here, equating an issue with a defect.

To determine the rework productivity of the practice *Pair*, Erdogmus and Williams adjust the resolution speed chosen for the practice *Solo* by the observed average nominal productivity gain of the practice *Pair*. This choice was based on the assumption that programmer pairs would achieve comparable productivity speed-ups during both development and rework activities. Here, we take a more conservative approach and adopt the same issue resolution speed for both practices.

Product quality. The particular choice of the rework productivity measure determines the context's quality measure as defects/LOC. This is the unit of issue density. Thus we must convert "percentage acceptance tests passing" to this measure, a straightforward task if we use a linear mapping.

According to the empirical study, *Solo* programmers on average passed 75% of the specified acceptance tests, whereas this average increased to 90% for programmer pairs. If a 25% failure rate is equivalent to a benchmark issue density of 0.00585 defects/LOC, then the 10% failure rate would be equivalent to $0.00585 \times 10/25 = 0.00234$ defects/LOC. These two figures are adopted as the issue density of *Solo* and *Pair* respectively.

Load factor. Pair programming is illustrative of staffing requirements' impact on cost-effectiveness. The staff load of a solo programmer is one person. That of a pair programmer is two persons. Assuming that all programmers are compensated on the same salary scale, then we can straightforwardly set the load factors of the practice *Solo* and *Pair* to 1 and 2, respectively.

Breakeven multiple

Table 2 summarizes the cost-effectiveness analysis for this study. The computed breakeven multiples suggest a slight economic advantage for pair programmers.

Table 2. Analysis of pair programming.

	Unit	*Solo*	*Pair*
Load factor (L)	persons	1	2
Production speed (p)	LOC/hour	25	43.5
Issue density (d)	defects/LOC	0.00585	0.00234
Resolution speed (r)	defects/hour	0.0303	0.0303
Production efficiency (ε)	(none)	17%	23%
Breakeven multiple (BM)	**LOC/ person-hour**	**4.29**	**4.98**

6.2. Test-driven development

As a second example, consider test-driven development (TDD), a coding technique in which development tasks are driven by unit tests written before production code. An empirical study by Erdogmus, Morisio, and Torchiano [20] evaluated the effects of writing unit tests before production code (test-first) relative writing units tests after production code (test-last), in both cases using incremental development approach. The example demonstrates the

use of sensitivity analysis to deal with incomplete information.

The study measured the average nominal productivity and product quality of two groups performing a programming task involving the incremental implementation of a set of requirements. Subjects in one group employed a test-first strategy in the spirit of TDD, while subjects in the other group employed the more traditional test-last variation. The requirements were specified as micro-stories, in the style of extreme programming, but at a very fine level of granularity. The evaluators measured quality in a similar manner to the pair programming study, but on a per story basis: they determined first the percentage of successful acceptance tests for each completed story and then they averaged the quality scores over all completed stories. The acceptance tests evaluated the programs' conformance to the specified requirements. This study did not measure rework productivity either.

Base measures. Again think of the two experimental groups representing the two techniques as two projects, say *Test-First* and *Test-Last*, sharing a common context. The nominal output measure of the context is number of completed user stories. The measure of nominal productivity is stories per hour, which is readily adoptable as the unit of production speed. Unit schedule is again an hour.

A failing acceptance test is equivalent to an issue. Normalizing number of such failures with respect to the nominal output yields an issue density measure with failures per story as its unit.

The load factor for this context is constant since the two techniques were executed by solo programmers. Thus $L = 1$ person.

Product quality. The only necessary adjustment here is to translate the average percentage of successful acceptance tests (a relative measure) to failures per story (the absolute target measure). In the study, a story was associated with an average of 17 acceptance sets. We can take advantage of this 1-to-17 ratio when converting the original relative quality measure to an absolute issue density. For example an average story quality of a 85% results in a corresponding issue density of $(1 - .85) \times 17 = 2.55$ failures/story.

Rework productivity. The output and quality units force the measure rework productivity to be measured in failures per hour. Since rework productivity is unspecified, we must again find a proxy for resolution speed. Alas, the context's specificity doesn't make borrowing from published benchmarks viable this time.

Instead, we resort to sensitivity analysis. First, note the a main finding of the study: *Test-First* exhibited a statistically significant nominal productivity speed-up of 28% relative to *Test-Last*.

Breakeven multiple. The strategy to analyzing the behavior of breakeven multiple begins with fixing the production speed of *Test-Last*, and estimating the resolution speed of *Test-First* by applying the observed 28% nominal productivity speed-up. Subsequently, vary *Test-Last*'s resolution speed, estimate the *Test-First*'s resolution speed, compute the corresponding production efficiencies, and finally plot the breakeven multiples against the resulting production efficiency pairs. The chart in Figure 4 shows this analysis, where the x-axis is labeled by the resulting production efficiency pairs in the form "$\mathcal{E}_{\text{Test-Last}} \mid \mathcal{E}_{\text{Test-First}}$".

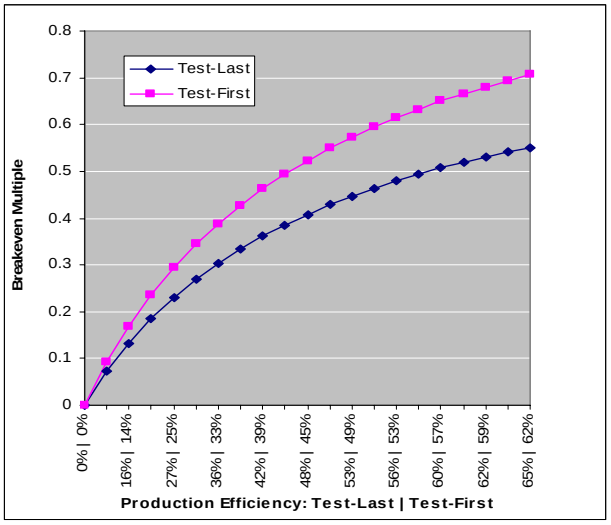

Figure 4. Breakeven multiples for the TDD study as a function of the resolution speed.

7. Discussion

The breakeven multiple is a new aggregate economic metric for software development. It reduces what would ordinarily be multi-criteria comparisons based on separate quality, productivity, and staffing measures into a single-criterion comparisons based on cost-effectiveness. It is indented for use as a high-level performance indicator for software projects and as a dependent variable in empirical studies of software development.

The main measurement requirement of this indicator is the ability to differentiate between development and rework effort. It is robust and easily customizable. Although it builds on five base measures, three suffice in a variety of simplified situations, for example in controlled, experimental settings. In particular, breakeven value is easily computed if development effort, rework effort, staffing profile, and nominal output are known. One

dimensional productivity measures, such as problem solving effort, commonly used in experimental settings can be shown to be special cases of the breakeven multiple. The breakeven multiple stresses the importance of the often-neglected capability to measure an important cost driver of software development: production efficiency, or the ratio of development effort to total effort.

Breakeven multiple has limitations. It is impossible to encapsulate all important and relevant aspects of a complex production process in a single quantity. Breakeven multiple is not an exception: it is only one viable measure among many. Nor is this indicator immune to the pitfalls of metrics misuse and common problems plaguing software measurement.

8. References

[1] K. Maxwell and P. Forselius, "Benchmarking software development productivity," *IEEE Software*, pp. 80-88, 2000.

[2] A. MacCormack, C. F. Kemerer, M. Cusumano, and B. Crandall, "Trade-offs between productivity and quality in selecting software development practices," *IEEE Software*, vol. Sep/Oct, pp. 78-85, 2003.

[3] B. W. Boehm and P. N. Papaccio, "Understanding and controlling software costs," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1462-1477, 1988.

[4] S. Wagner, "A literature survey of the quality economics of defect-detection techniques," presented at International Symposium on Empirical Software Engineering, 2006.

[5] V. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric Approach," in *Encyclopedia of Software Engineering*, vol. 1: John Wiley & Sons, 1994, pp. pp. 528-532.

[6] K. Akingbehin, "Taguchi-based metrics for software quality," presented at Fourth Annual ACIS Int'l Conference on Computer and Information Science, 2005.

[7] W. A. Florac, "Software quality measurement: a framework for counting problems and defects," Software Engineering Institute, Pittsburgh, PA, Technical Report CMU/SEI-92-TR-22, 1992.

[8] S. H. Kan, "Product Quality Metrics," in *Metrics and Models in Software Quality Engineering*, 2 ed: Addison-Wesley, 2002.

[9] L. Layman, L. Williams, and L. Cunningham, "Motivations and measurements in an agile case study," presented at Foundations of Software Engineering, 2004 Worshop on Quantitative Techniques for Software Agile process, 2004.

[10] B. Kitchenham, "The problem with function points," in *IEEE Software*, 1997.

[11] S. Furey, "Why should we use function points?," in *IEEE Software*, 1997.

[12] P. Armour, "The unmyths of project estimation," in *Communications of the ACM*, vol. 45, 2002, pp. 15-18.

[13] M. Asmild, J. C. Paradi, and A. Kulkarni, "Using data envelopment analysis in sofware development productivity measurement," *Software Process Improvement and Practice*, vol. 11, pp. 561-572, 2006.

[14] C. Jones, "Backfiring: converting lines of code to function points," *Computer*, vol. 28, pp. 87-88, 1995.

[15] B. Kitchenham and E. Mendez, "Software productivtity measurement using multiple size measures," *IEEE Transactions on Software Engineering*, vol. 30, pp. 1023-1035, 2004.

[16] H. Erdogmus, "Comparative evaluation of software development strategies based on Net Present Value," presented at First ICSE Workshop on Economics-Driven Software Engineering Research, Los Angeles, California, 1999.

[17] M. Müller and F. Padberg, "On the economic evaluation of XP projects," presented at Jouint 9th European Software Engineering Conference and 11th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering, Helsinki, Finland, 2003.

[18] H. Erdogmus and L. Williams, "The Economics of Software Development by Pair Programmers," *The Engineering Economist*, vol. 48, 2003.

[19] F. Padberg and M. Müller, "Analyzing cost and benefits of pair programming," presented at 9th International Software Metrics Symposium, 2003.

[20] H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *IEEE Transactions on Software Engineering*, vol. 31, pp. 226-237, 2005.