CS 4530 & CS 5500
Software Engineering

Lesson 12.4: Measuring Engineering Productivity

Jonathan Bell, John Boyland, Mitch Wand
Khoury College of Computer Sciences

Learning Objectives for this Lesson

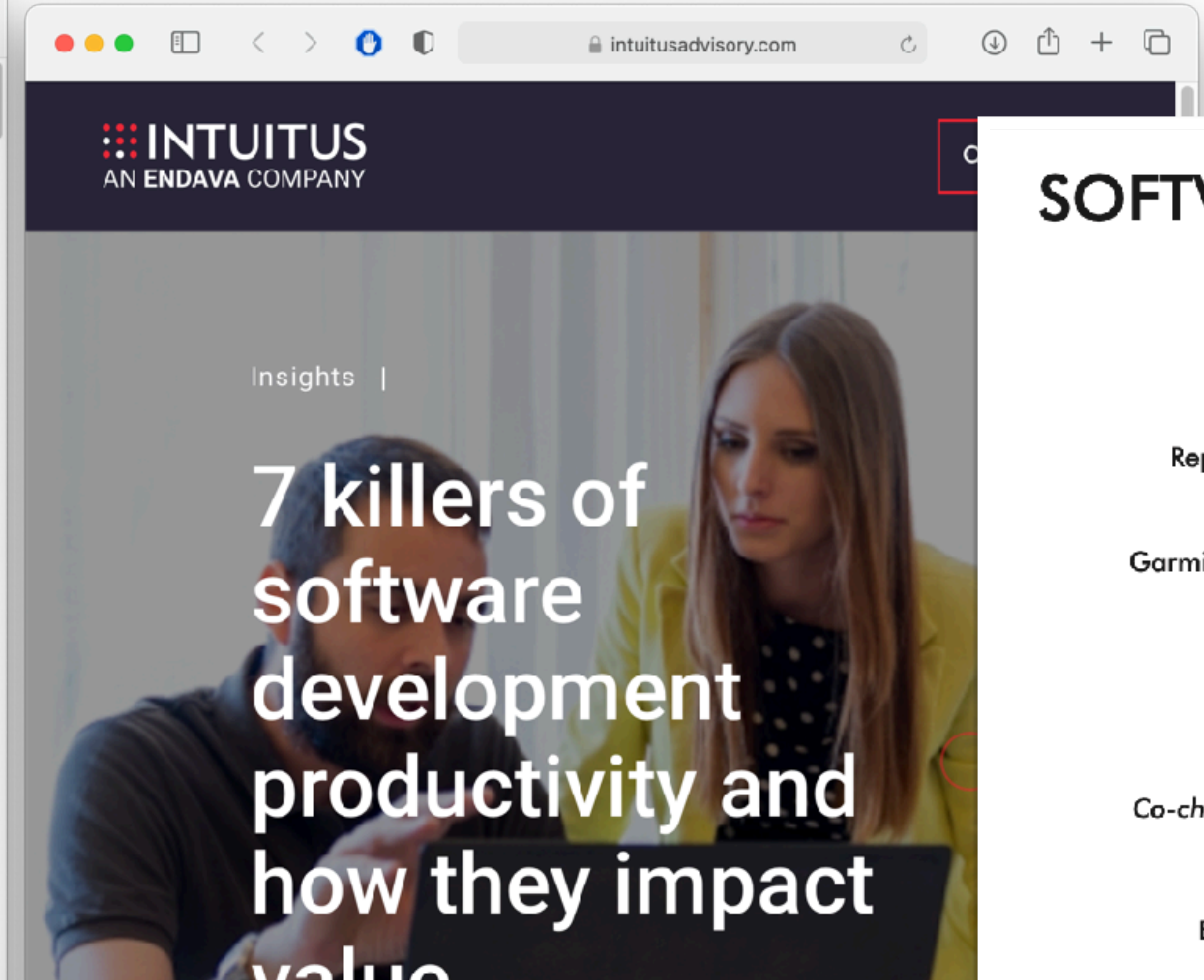
By the end of this lesson, you should be able to…

- Apply a goal/signal/metrics framework in software engineering as a feedback loop to improve processes

McNamara Fallacy

Reminder (See Lesson 12.2)

- Measure whatever can be easily measured
- Disregard that which cannot be measured easily
- Presume that which cannot be measured easily is not important
- Presume that which cannot be measured easily does not exist



SOFTWARE ENGINEERING

Report on a conference sponsored by the

NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968

Chairman: Professor Dr. F. L. Bauer

Co-chairmen: Professor L. Bolliet, Dr. H. J. Helms

Editors: Peter Naur and Brian Randell

January 1969

https://codingsans.com/blog/team-productivity-improve-developers-productivity

https://intuitusadvisory.com/insights/7-killers-of-software-development-productivity-and-how-they-impact-value

A Large Scale Study of Programming Languages and Code Quality in Github

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, Premkumar Devanbu
{bairay@, dpposnett@, filkov@cs., devanbu@cs.}ucdavis.edu
Department of Computer Science, University of California, Davis, CA, 95616, USA

ABSTRACT

What is the effect of programming languages on software quality? This question has been a topic of much debate for a very long time. In this study, we gather a very large data set from GitHub (728 projects, 63 Million SLOC, 29,000 authors, 1.5 million commits, in 17 languages) in an attempt to shed some empirical light on this question. This reasonably large sample size allows us to use a mixed-methods approach, combining multiple regression modeling with visualization and text analytics, to study the effect of language features such as static *v.s.* dynamic typing, strong *v.s.* weak typing on software quality. By triangulating findings from different methods, and controlling for confounding effects such as team size, project size, and project history, we report that language design does have a significant, but modest effect on software quality. Most notably, it does appear that strong typing is modestly better than weak typing, and among functional languages, static typing is also somewhat better than dynamic typing. We also find that functional languages are somewhat better than procedural languages. It is worth noting that these modest effects arising from language design are overwhelmingly dominated by the process factors such as project size, team size, and commit size. However, we hasten to caution the reader that even these modest effects might quite possibly be due to other, intangible process factors, *e.g.*, the preference of certain personality types for functional, static and strongly typed languages.

Categories and Subject Descriptors

D.3.3 [**PROGRAMMING LANGUAGES**]: [Language Constructs and Features]

General Terms

Measurement, Experimentation, Languages

Keywords

programming language, type system, bug fix, code quality, empirical research, regression analysis, software domain

1. INTRODUCTION

A variety of debates ensue during discussions whether a given programming language is "the right tool for the job". While some of these debates may appear to be tinged with an almost religious fervor, most people would agree that a programming language can impact not only the coding process, but also the properties of the resulting artifact.

Advocates of strong static typing argue that type inference will catch software bugs early. Advocates of dynamic typing may argue that rather than spend a lot of time correcting annoying static type errors arising from sound, conservative static type checking algorithms in compilers, it's better to rely on strong dynamic typing to catch errors as and when they arise. These debates, however, have largely been of the armchair variety; usually the evidence offered in support of one position or the other tends to be anecdotal.

Empirical evidence for the existence of associations between code quality programming language choice, language properties, and usage domains, could help developers make more informed choices.

Given the number of other factors that influence software engineering outcomes, obtaining such evidence, however, is a challenging task. Considering software quality, for example, there are a number of well-known influential factors, including source code size [11], the number of developers [36, 6], and age/maturity [16]. These factors are known to have a strong influence on software quality, and indeed, such process factors can effectively predict defect localities [32].

One approach to teasing out just the effect of language properties, even in the face of such daunting confounds, is to do a *controlled experiment*. Some recent works have conducted experiments in controlled settings with tasks of limited scope, with students, using languages with static or dynamic typing (based on experimental treatment setting) [14, 22, 19]. While type of controlled study is *"El Camino Real"* to solid empirical evidence,another opportunity has recently arisen, thanks to the large number of open source projects collected in software forges such as GitHub.

GitHub contains many projects in multiple languages. These projects vary a great deal across size, age, and number of developers. Each project repository provides a historical record from which we extract project data including the contribution history, project size, authorship, and defect repair. We use this data to determine the effects of language features on defect occurrence using a variety of tools. Our approach is best described as mixed-methods, or triangulation [10] approach. A quantitative (multiple regression) study is further examined using mixed methods: text analysis, clustering, and visualization. The observations from the mixed methods largely confirm the findings of the quantitative study.

On the Impact of Programming Languages on Code Quality: A Reproduction Study

21

EMERY D. BERGER, University of Massachusetts Amherst and Microsoft Research
CELESTE HOLLENBECK, Northeastern University
PETR MAJ, Czech Technical University in Prague
OLGA VITEK, Northeastern University
JAN VITEK, Northeastern University and Czech Technical University in Prague

In a 2014 article, Ray, Posnett, Devanbu, and Filkov claimed to have uncovered a statistically significant association between 11 programming languages and software defects in 729 projects hosted on GitHub. Specifically, their work answered four research questions relating to software defects and programming languages. With data and code provided by the authors, the present article first attempts to conduct an experimental repetition of the original study. The repetition is only partially successful, due to missing code and issues with the classification of languages. The second part of this work focuses on their main claim, the association between bugs and languages, and performs a complete, independent reanalysis of the data and of the statistical modeling steps undertaken by Ray et al. in 2014. This reanalysis uncovers a number of serious flaws that reduce the number of languages with an association with defects down from 11 to only 4. Moreover, the practical effect size is exceedingly small. These results thus undermine the conclusions of the original study. Correcting the record is important, as many subsequent works have cited the 2014 article and have asserted, without evidence, a causal link between the choice of programming language for a given task and the number of software defects. Causation is not supported by the data at hand; and, in our opinion, even after fixing the methodological flaws we uncovered, too many unaccounted sources of bias remain to hope for a meaningful comparison of bug rates across languages.

CCS Concepts: • **General and reference → Empirical studies;** • **Software and its engineering → Software testing and debugging;**

Additional Key Words and Phrases: Programming Languages on Code Quality

ACM Reference format:
Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the Impact of Programming Languages on Code Quality: A Reproduction Study. *ACM Trans. Program. Lang. Syst.* 41, 4, Article 21 (October 2019), 24 pages.
https://doi.org/10.1145/3340571

This work received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (awards 1518844, 1544542, and 1617892), and the Czech Ministry of Education, Youth and Sports (grant agreement CZ.02.1.010.00.015_0030000421).

Authors' addresses: E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek, Khoury College of Computer Sciences, Northeastern University, 440 Huntington Ave, Boston, MA 02115; emails: emery.berger@gmail.com, majpetr@fit.cvut.cz, celeste.hollenbeck@gmail.com, o.vitek@northeastern.edu, vitekj@me.com.

0164-0925/2019/10-ART21 $15.00
https://doi.org/10.1145/3340571

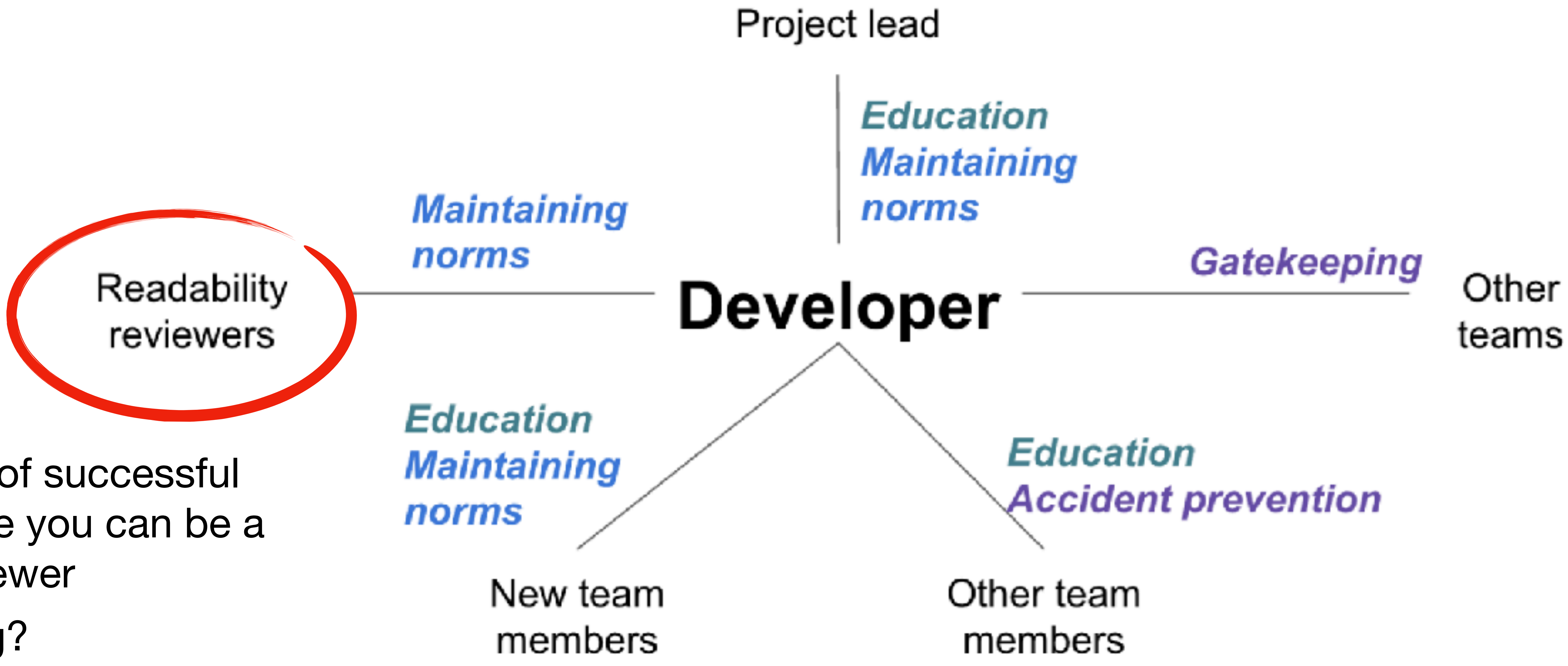
Metrics and Productivity

Applying metrics, sanely

- Consider multiple quantitative *and* qualitative metrics
- Use metrics to evaluate performance *in aggregate*, and *not* for an individual's performance review

Measuring and Improving Engineering Productivity

Example: Code Review Processes



You need to have 100's of successful changes integrated before you can be a readability reviewer

Is this hazing?

Do linters replace this?

"Modern Code Review: A Case Study at Google", Sadowski et al, ICSE 2018

How do we measure process efficiency?

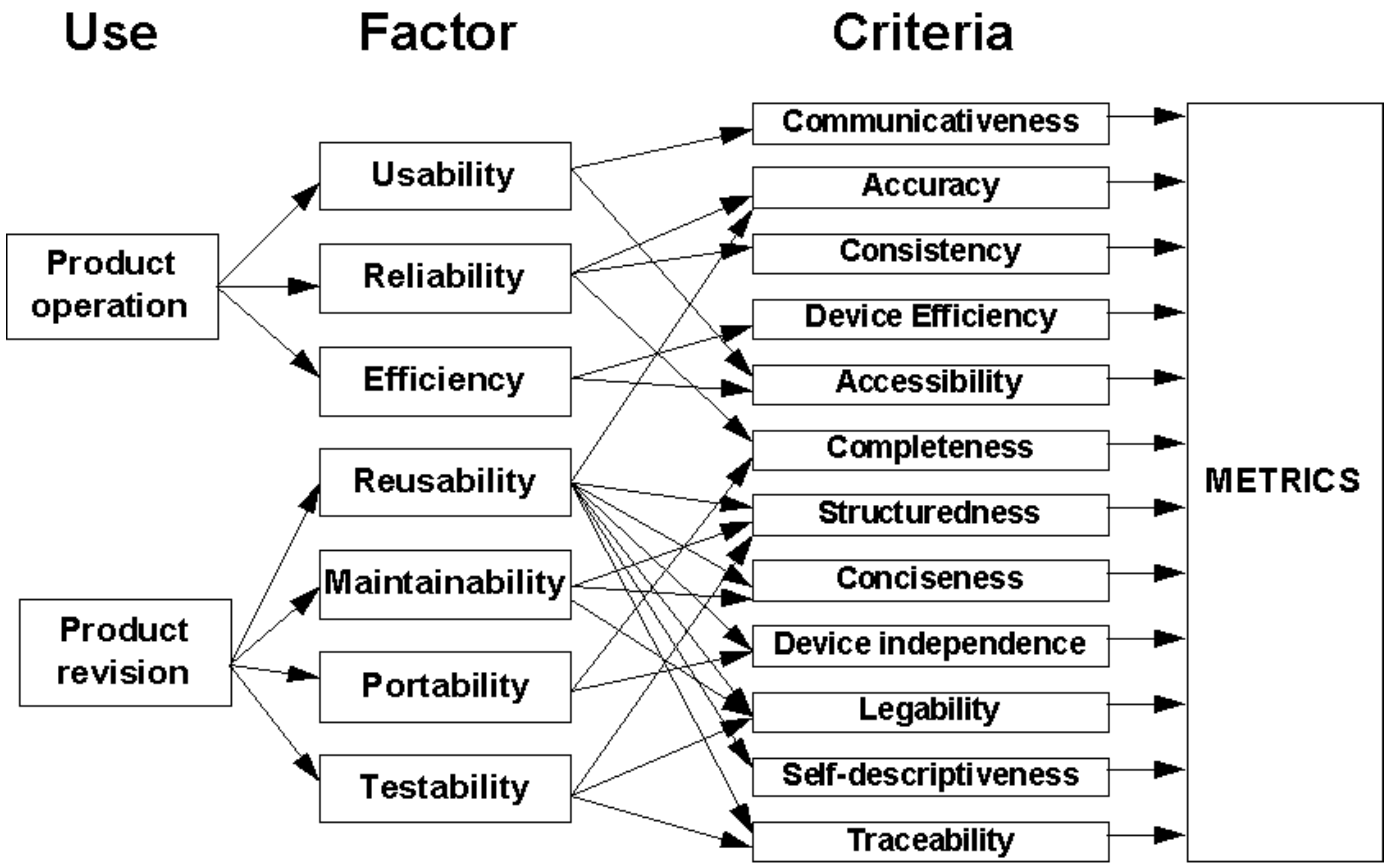
Goal/Signal/Metric framework

- Goal: desired end result
- Signal: How we’re likely to know if we’ve achieved the end result, may not be measurable
- Metric: A proxy for a signal, which can actually be measured

[Software Engineering @ Google Ch 7]

From Quality Goals to Metrics

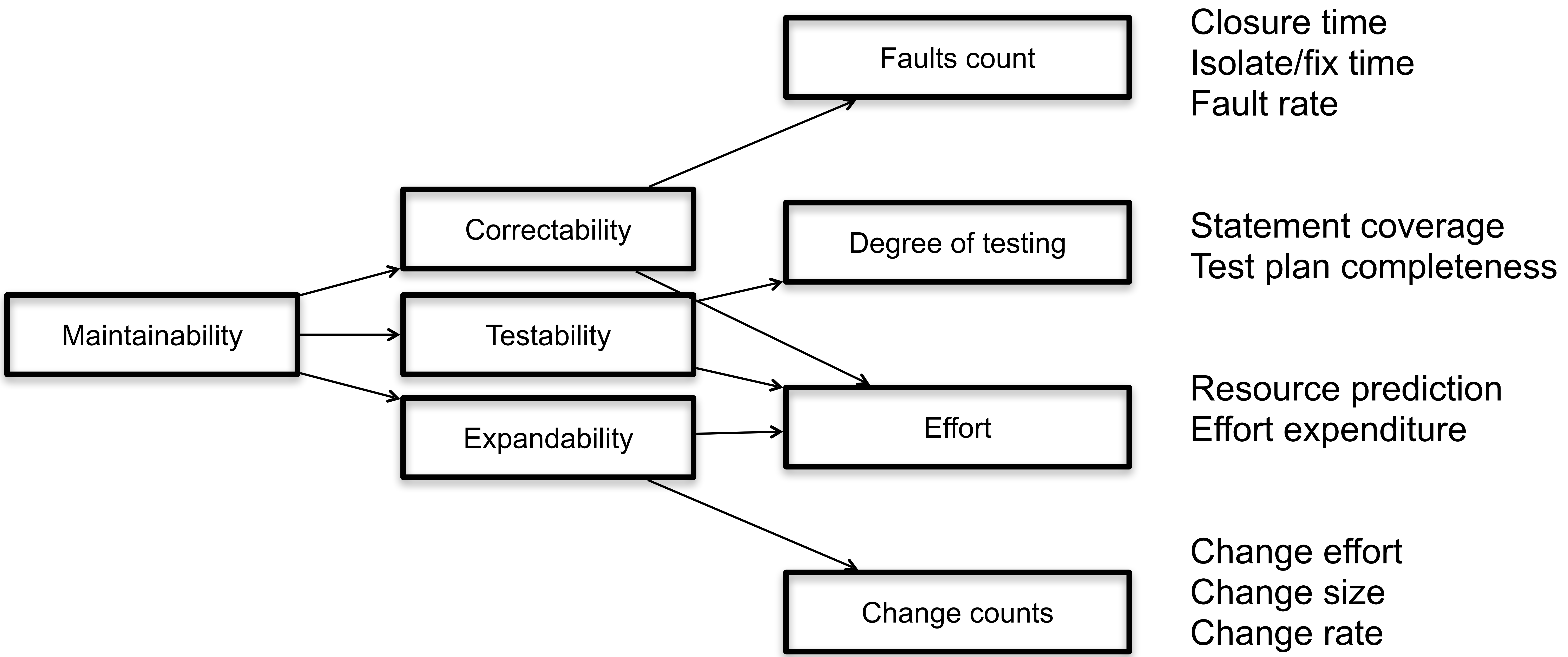
McCall Quality Model



“A Framework for the Measurement of Software Quality”, Cavano & McCall

From Quality Goals to Metrics

McCall Quality Model



Engineering Productivity: A Broad Goal

QUANTS components

- **Qu**ality of the code (Is it tested? Is it maintainable?)
- **A**ttention from engineers (Does the process distract engineers?)
- **In**tellectual complexity (How does the complexity of the process relate to the complexity of the task?)
- **T**empo and velocity (How quickly can engineers accomplish their tasks?)
- **S**atisfaction (How happy are engineers?)

[Software Engineering @ Google Ch 7]

From Goals to Signals and Metrics

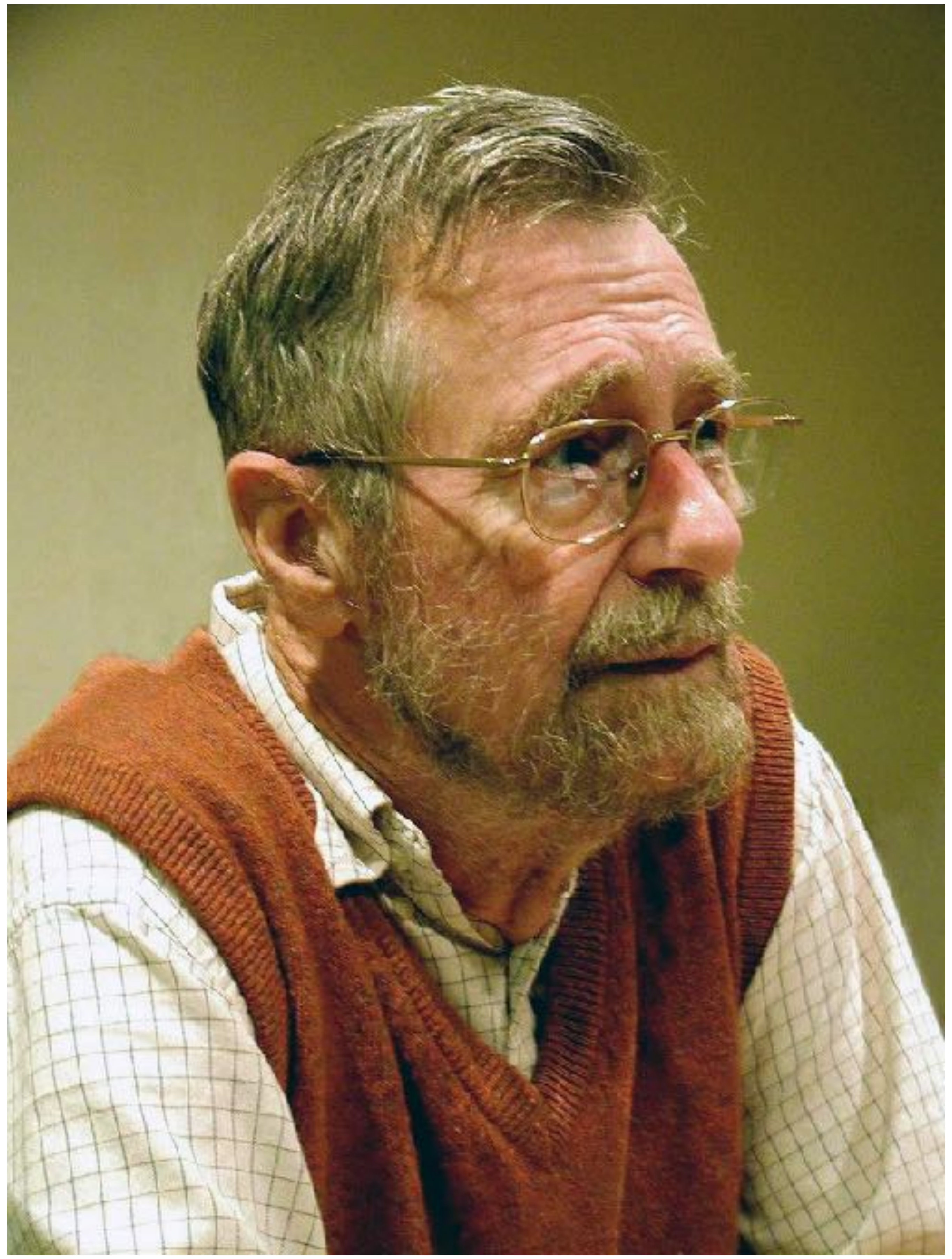
Readability Review

- Goal: “Engineers write higher-quality code as a result of the readability process.”
  - Signal: “Engineers who have been granted readability judge their code to be of higher quality than engineers who have not been granted readability.”
    - Metric: “Quarterly Survey: Proportion of engineers who report being satisfied with the quality of their own code”
  - Signal: “The readability process has a positive impact on code quality.”
    - Metric: “Readability Survey: Proportion of engineers reporting that readability reviews have no impact or negative impact on code quality”
    - Metric: “Readability Survey: Proportion of engineers reporting that participating in the readability process has improved code quality for their team”

[Software Engineering @ Google Ch 7]

A closing word on productivity

“On the cruelty of really teaching computing science”



From there it is only a small step to measuring ‘programmer productivity’ in terms of ‘number of lines of code produced per month.’ This is a very costly measuring unit because it encourages the writing of insipid code, but today I am less interested in how foolish a unit it is from even a pure business point of view. My point today is that, if we wish to count lines of code, we should not regard them as ‘lines produced’ but as ‘lines spent’: the current conventional wisdom is so foolish as to book that count on the wrong side of the ledger.

- Edsger W. Dijkstra