CS 4530 & CS 5500
Software Engineering
Lesson 12.4: Measuring Engineering Productivity
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
Team Productivity: 9 Ways to Improve Developers Productivity

7 killers of software development productivity and how they impact value

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. Ballard, Dr. H. J. Hems

Editors: Peter Naur and Brian Randell

January 1969
A Large Scale Study of Programming Languages and Code Quality in GitHub

Baishachi Ray, Daryl Posnett, Vladimir Filkov, Premkumar Devanbu
Department of Computer Science, University of California, Davis, CA 95616, USA

ABSTRACT

What is the eff 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 (778 projects, 65 Million SLOC, 29,000 authors, 1.5 million commits) on languages to answer this question. This massive large sample size allows us to use a mixed methods approach, combining multiple regression models with visualization and new analytics, to study the eff of language features such as static vs. dynamic typing, strong vs. weak typing on software quality. By integrating findings from different sources and languages, we find that language features have a substantial eff on software quality. Most notably, it does appear that strong typing is more costly 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 effs arising from language design are overwhelmingly dominated by the process factors such as project size, team size, and commit size. However, we believe that the question of whether these modest effs might be quite possibly due to other, more intangible factors, e.g., the performance of certain personality types for functional, static and strongly typed languages.

Categories and Subject Descriptors

D.3.3 [PROGRAMMING LANGUAGES]: Language [Language Constructs and Facilities]

General Terms

Measurement, Experimentation, Languages

Keywords

programming language, type system, big fix, code quality, empirical research, regression analysis, software dolat

Introduction

1. INTRODUCTION

A variety of debates exist during discussions whether a given programming language is “the right tool for the job”. While some of these debates may appear to be targeted at the obvious features, 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 other that a bit of time spent using a more static typing type errors arise from users, conservative-static type checking algorithms are easier to implement, e.g., it’s better to rely on strong static typing to catch errors as and when they arise. These debates, however, have largely been left of the empirical evidence; 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 and language design, language properties, and usage features 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 [1], the number of developers [56], and team maturity [15].

These factors are known to have a strong influence on software quality, and instead, such process factors can effectively predict defect locations [12].

One approach is to iron out just the effect of language properties, even if the face of such daunting confounders, 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 settings) [14,21,19]. While these are controlled experiments, the results do not generalize to the real world setting.

GitHub contains many projects in multiple languages. These projects represent a great array across size, age, and maturity of developers. Each project repository provides a historical record from which we can extract data including the contribution history, project size, authorship, and defect repair. We use this data to determine the effs of language features on defect occurrence using a variety of tools. Our approach is best described as mixed methods, i.e., triangulation [13] approach. A quantitative-multiple regression study is further examined using mixed methods: text analysis, code analysis, and interview analysis. 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

EMERY D. BERGER, University of Massachusetts Amherst and Microsoft Research
CELESTE HOLLENBRCK, Northeastern University
PETR MAL, 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 720 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 of the present article, first attempts to conduct an experiment on the replication of the original study. The replication is only partially successful, due to missing code and issues with the distribution of languages. The second part of this work focuses on their main claim: the association between bugs and languages, and perform a complete, independent reanalysis of the data and statistical modeling steps undertaken by Roy et al. in 2014. This reanalysis uncovers a number of serious flaws that induce the number of languages and an association with defects down from 11 to 7 only. Moreover, the predicted effect sizes are exceedingly small. These results thus undermine the conclusions of the original study.

To be sure, the present work is by no means conclusive. The association between bugs and languages is still a topic of considerable interest for the software engineering community. The corrected problem 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. Caution is not expressed by the data at hand, of course. In our opinion, even after the fact, the methodological flaws we uncovered the many unaccounted sources of bias remain to be held for a meaningful comparison of bug rates across languages.


Additional Key Words and Phrases: Programming Languages on Code Quality

ACM Reference format


https://doi.org/10.1145/3354633

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

ACM Transactions on Programming Languages and Systems, Vol. 41, No. 4, Article 21. Publication date: October 2019.

Permission is made digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and this copy bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2015 Association for Computing Machinery.

ACM Transactions on Programming Languages and Systems, Vol. 37, No. 5, Article 30. Publication date: September 2015.
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
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

Maintainability →

- Correctability → Faults count
- Testability → Degree of testing
- Expandability → Effort

Change counts

- Closure time
- Isolate/fix time
- Fault rate

- Statement coverage
- Test plan completeness

- Resource prediction
- Effort expenditure

- Change effort
- Change size
- Change rate
Engineering Productivity: A Broad Goal

QUANTS components

- **Quality** of the code (Is it tested? Is it maintainable?)
- **Attention** from engineers (Does the process distract engineers?)
- **Intellectual complexity** (How does the complexity of the process relate to the complexity of the task?)
- **Tempo and velocity** (How quickly can engineers accomplish their tasks?)
- **Satisfaction** (How happy are engineers?)
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”
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
This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit [http://creativecommons.org/licenses/by-sa/4.0/](http://creativecommons.org/licenses/by-sa/4.0/)
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.