Metrics & Mining Software Repositories

Advanced Software Engineering
Spring 2023

Agenda

Today: Lecture - Metrics and mining software repositories (case studies)

Why should we measure metrics, and mine software repositories for them?

What metrics can we measure?

How can we trust conclusions from these metrics?

Thursday: Discussion - Methodology for MSR and a study of jupyter notebooks

Why Measure Stuff in SE?

- Do we fund a project?
- Are we done testing?
- Is our code fast enough, or secure enough?
- Is our code maintainable?
- What features should we focus on improving?
- Who do we give a bonus to?

Big Question: How do we measure?

codingsans.com

_ BLOG / MANAGEMENT

01

Team Productivity: 9 Ways to Improve Developers Productivity

by Tamás Török
/ January 23, 2018

#Management

Display a menu

Cookie Policy

intuitusadvisory.com

Insights |

7 killers of software development productivity and how they impact value

Display a menu

RING

Report on a conference sponsored by the
NATO SCIENCE COMMITTEE
Garmisch, Germany, 7th to 11th October 1968

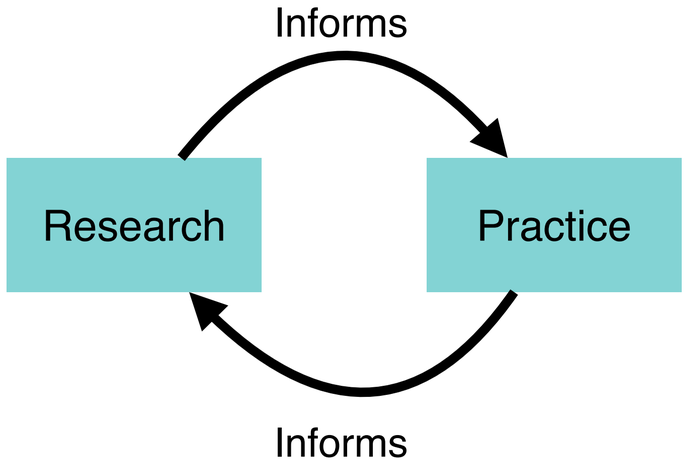
Chairman: Professor Dr. F. L. Bauer
Co-chairmen: Professor L. Bolliet, Dr. H. J. Helms

Editors: Peter Naur and Brian Randell

January 1969

Empirical Software Engineering Research

- Conduct research to clearly understand state of the art
- Develop new interventions (tools, processes, etc)
- Evaluate interventions in context
- Utilize diverse empirical methods
  - Surveys, user studies, **analyze existing artifacts**



Brainstorm: What kinds of metrics can we collect from teams/codebases?

(Bullets are notes from class discussion)

- Frequency of commits - how active
  - Content of commit messages (specific words... swears)
- LoC - per-file, per-language, etc
- Presence of bugs (reported by users, fixed by developers)
- Overall code quality (linters)
- Test coverage - broadly, test results (could include performance and other indicators)
- Specific kinds of changes (change API, change version)
- Error handling (review code and determine how well handled they are)
- Collaboration - who wrote how many LoC, who improved others' code (who introduced most bugs?)
- Number of revisions included in a single code review/pull request/change list
- Readability (expert review, or semi-automated?)
- Security (presence of known vulnerabilities)
- NON-code artifacts: instructions (README, datasets, other artifacts)
- Number of dependencies
- Downstream dependencies: how much this package is used

Brainstorm: What are the risks to collecting and using metrics?

(Bullets are notes from class discussion)

- Frequency of commits - how active
  - Content of commit messages (specific words... swears)
- LoC - per-file, per-language, etc
- Presence of bugs (reported by users, fixed by developers)
- Overall code quality (linters)
- Test coverage - broadly, test results (could include performance and other indicators)
- Specific kinds of changes (change API, change version)
- Error handling (review code and determine how well handled they are)
- Collaboration - who wrote how many LoC, who improved others' code (who introduced most bugs?)
- Number of revisions included in a single code review/pull request/change list
- Readability (expert review, or semi-automated?)
- Security (presence of known vulnerabilities)
- NON-code artifacts: instructions (README, datasets, other artifacts)
- Number of dependencies
- Downstream dependencies: how much this package is used

- General: these metrics are likely proxies for the actual goal you are trying to measure
- Co-occurrence, co-variance of metrics
- Might have set the wrong goal
- Metrics might be the wrong or could be computed wrong (leading to downstream problems)
- Risk of putting too much pressure on developers and having an unexpected influence on the overall system (see also goodhart's law)

What metrics can you mine?

- Version control: commits (who, what, when, why?)
- Bug tracker: issues (maybe introducing commit, fixing commit)
- Static code analysis (e.g. LoC, cyclomatic complexity, presence of APIs, type definitions, etc)
- Continuous integration log analysis (passing/failing tests, etc)
- Dynamic code analysis (e.g. build it and run tests)
- Survey feedback

We have seen two MSR-style projects already

2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)

How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub

Shurui Zhou
Carnegie Mellon University, USA

Bogdan Vasilescu
Carnegie Mellon University, USA

Christian Kästner
Carnegie Mellon University, USA

ABSTRACT

The notion of forking has changed with the rise of distributed version control systems and social coding environments, like GitHub. Traditionally forking refers to splitting off an independent development branch (which we call *hard forks*); research on hard forks, conducted mostly in pre-GitHub days showed that hard forks were often seen critical as they may fragment a community. Today, in social coding environments, open-source developers are encouraged to fork a project in order to contribute to the community (which we call *social forks*), which may have also influenced perceptions and practices around hard forks. To revisit hard forks, we identify, study, and classify 15,306 hard forks on GitHub and interview 18 owners of hard forks or forked repositories. We find that, among others, hard forks often evolve out of social forks rather than being planned deliberately and that perception about hard forks have indeed changed dramatically, seeing them often as a positive non-competitive alternative to the original project.

ACM Reference Format:
Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380412

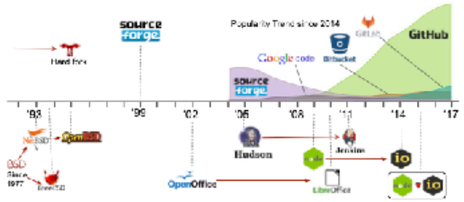


Figure 1: Timeline of some popular open-source forking events; popularity approximated with *Google Trends*.

1 INTRODUCTION

forks in the traditional sense. As in our prior work [53], we distinguish between **social forks**, referring to creating a public copy of a repository on a social coding site like GitHub, often with the goal of contributing to the original project, and **hard forks**, referring to the traditional notion of splitting off a new development branch.

Hard forks have been discussed controversially throughout the history of free and open-source software. On the one hand, free and open-source licenses codified the right to create hard forks, which was seen as essential for guaranteeing flexibility and fostering disruptive innovations [15, 30, 32] and useful for encouraging a

2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)

How to Not Get Rich: An Empirical Study of Donations in Open Source

Cassandra Overney,† Jens Meinicke,‡ Christian Kästner,‡ Bogdan Vasilescu‡
†Olin College, USA ‡Carnegie Mellon University, USA

ABSTRACT

Open source is ubiquitous and many projects act as critical infrastructure, yet funding and sustaining the whole ecosystem is challenging. While there are many different funding models for open source and concerted efforts through foundations, donation platforms like *PayPal*, *Patreon*, and *OpenCollective* are popular and low-bar platforms to raise funds for open-source development. With a mixed-method study, we investigate the emerging and largely unexplored phenomenon of donations in open source. Specifically, we quantify how commonly open-source projects ask for donations, statistically model characteristics of projects that ask for and receive donations, analyze for what the requested funds are needed and used, and assess whether the received donations achieve the intended outcomes. We find 25,885 projects asking for donations on GitHub, often to support engineering activities; however, we also find no clear evidence that donations influence the activity level of a project. In fact, we find that donations are used in a multitude of ways, raising new research questions about effective funding.

ACM Reference Format:
Cassandra Overney, Jens Meinicke, Christian Kästner, Bogdan Vasilescu. 2020. How to Not Get Rich: An Empirical Study of Donations in Open Source. In *Proceedings of 42nd International Conference on Software Engineering, Seoul,*

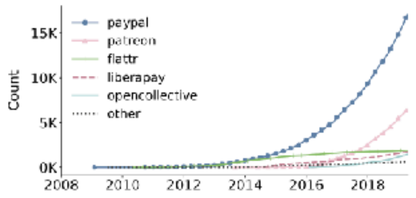


Figure 1: Adoption of donation platforms over time on GitHub (number of new non-fork repositories per month).

relevant. With increasing popularity, demands for maintenance and support work typically rise, including large numbers of support requests, feature requests, and reported issues. When open-source infrastructure is insufficiently maintained or even abandoned by their developers, this can raise significant costs and risks for users of such infrastructure, who might need to work around known bugs or make significant changes to find alternatives. How to supply all

Case Studies - Motivations and Metrics

- An academic question: Can we help developers write code that has less defects?
  - By structuring their code differently?
  - By focusing their testing efforts?
  - By choosing languages or frameworks?
- A practical question: Can we make developers more productive?
  - By changing a code review process?

Software Metric: McCabe Cyclomatic Complexity

Rationale: If we can measure complexity, then we can detect and avoid it

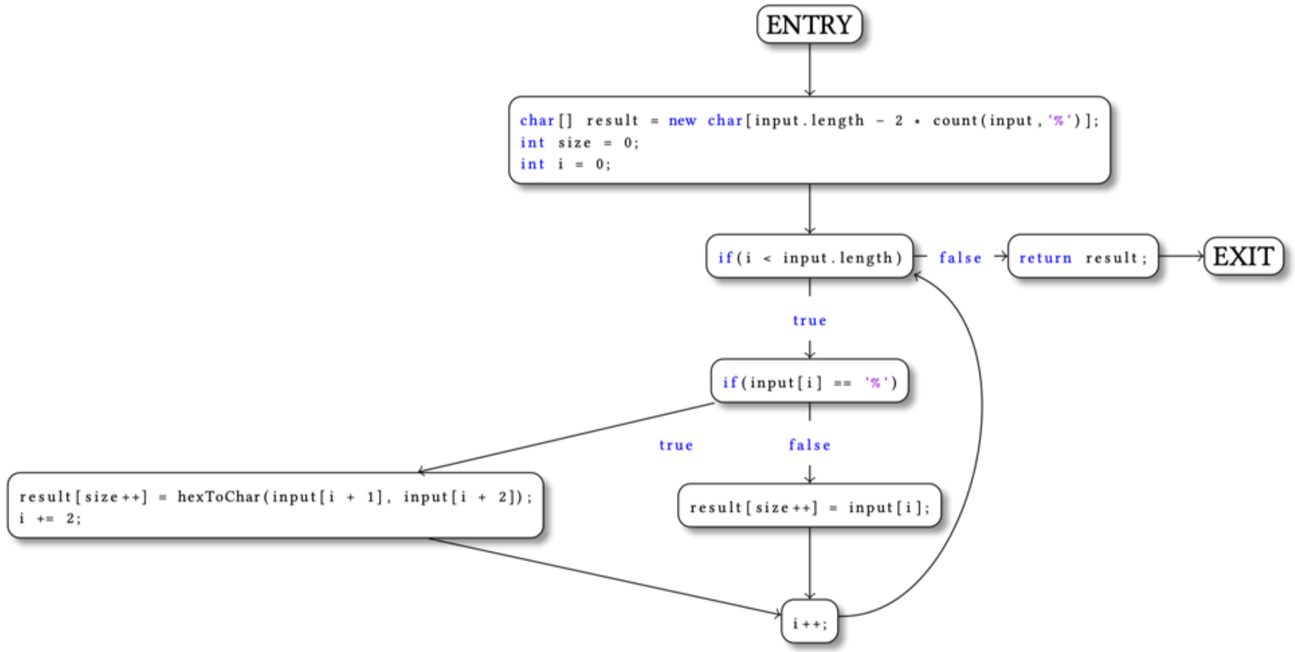
```
public static char[] percentDecode(char[] input) {
  char[] result = new char[input.length - 2 * count(input, '%')];
  int size = 0;
  for(int i = 0; i < input.length; i++) {
    if(input[i] == '%') {
      result[size++] = hexToChar(input[i + 1], input[i + 2]);
      i += 2;
    } else {
      result[size++] = input[i];
    }
  }
  return result;
}
```

Input: “Hello%20World”

Output: “Hello World”

Software Metric: McCabe Cyclomatic Complexity

Is this code complex to understand?



$$M = E - N + 2P$$

$$M = 10 - 9 + 2 \cdot 1$$

$$M = 3$$

Is this good?

Software Metric: McCabe Cyclomatic Complexity

Risk: Correlation != Causation

A critique of cyclomatic complexity as a software metric

by Martin Shepperd

McCabe's cyclomatic complexity metric is

1 Introduction

Table 1 Empirical validations of cyclomatic complexity

Researchers	LOC	Errors		Programming effort	Bug location	Program recall	Design effort
		density	*absolute*				
Basili (Ref. 38)	r^2=0.94	*r* is –ve					
Basili (Ref. 39)				*R*=0.48	*R*=0.21		
Bowen (Ref. 40)			r^2=0.47		*r*=–0.09		
Curtis (Ref. 41)	r^2=0.41,0.81,0.79					*r*=–0.35*	
Curtis (Ref. 42)	r^2=0.81,0.66				r^2=0.4,0.42		
Davis (Ref. 43)						*r* is –ve, +ve	
Feuer (Ref. 44)	r^2=0.90***						
Gaffney (Ref. 45)				r^2=0.60			
Henry (Ref. 46)	r^2=0.84***		r^2=0.92****				
Kitchenham (Ref. 47)	r^2=0.86,0.88		r^2=0.46,0.49,0.21****				
Paige (Ref. 48)	r^2=0.90						
Schneiderman (Ref. 49)	r^2=0.61*****		r^2=0.32*****				
Shen (Ref. 50)			r^2=0.78***				
Sheppard (Ref. 51)	r^2=0.79			r^2=0.38		*r*=0.35	
Sunohara (Ref. 52)				r^2=0.4,0.38			r^2=0.72,0.7
Wang (Ref. 53)	r^2=0.62			r^2=0.59			
Woodfield (Ref. 54)				r^2=0.26,*R*=0.39			
Woodward (Ref. 22)	r^2=0.90						

r^2 = Pearson moment *R* = Rank Spearman

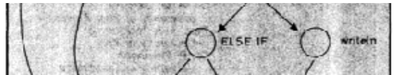
* *r* was 'improved' when modified for potentially 'aberrant' results

** correlated with *N* (i.e. Halstead's token count)

*** a simple decision count (i.e. *v*(*G*)–1)

**** indirect error count (i.e. version count), or program change count

***** using log-log transformations



theoretical and empirical grounds. Therefore cyclomatic complexity is of very limited utility.

2 The cyclomatic complexity metric

2016 IEEE International Conference on Software Quality, Reliability and Security

Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density

Kazuhiro Yamashita*, Changyun Huang*, Meiyappan Nagappan†, Yasutaka Kamei*,
Audris Mockus§, Ahmed E. Hassan†, and Naoyasu Ubayashi*
* Kyushu University, Japan; {yamashita, huang}@posl.ait.kyushu-u.ac.jp, {kamei, ubayashi}@ait.kyushu-u.ac.jp
† Queen's University, Canada; ahmed@cs.queensu.ca
‡ Rochester Institute of Technology, USA; mei@se.rit.edu
§ University of Tennessee-Knoxville, USA; audris@utk.edu

***Abstract*—Practical guidelines on what code has better quality are in great demand. For example, it is reasonable to expect the most complex code to be buggy. Structuring code into reasonably sized files and classes also appears to be prudent. Many attempts to determine (or declare) risk thresholds for various code metrics have been made. In this paper we want to examine the applicability of such thresholds. Hence, we replicate a recently published technique for calculating metric thresholds to determine high-risk files based on code size (LOC and number of methods), and complexity (cyclomatic complexity and module interface coupling) using a very large set of open and closed source projects written primarily in Java. We relate the threshold-derived risk to (a) the probability that a file would have a defect, and (b) the defect density of the files in the high-risk group. We find that the probability of a file having a defect is higher in the very high-risk group with a few exceptions. This is particularly pronounced when using size thresholds. Surprisingly, the defect density was uniformly lower in the very high-risk group of files. Our results suggest that, as expected, less code is associated with fewer defects. However, the same amount of code in large and complex files was associated with fewer defects than when located in smaller and less complex files. Hence we conclude that risk thresholds for size and complexity metrics have to be used with caution if at all. Our findings have immediate practical implications: the redistribution of Java code into smaller and less complex files may be counterproductive.**

***Index Terms*—Software metrics; Thresholds; Defect models;**

I. Introduction

There has been a considerable amount of research in the field of software defect prediction, especially in the last decade. Over 100 papers have been published just from 2000 - 2011 in this area of Empirical Software Engineering (ESE) research alone [16, 27]. The primary goal of such research is to be able to provide guidelines to practitioners on what kind

metrics themselves [11, 14], error models [6, 10, 26], cluster techniques [22, 34], and metric distributions [31, 32].

Almost all of these approaches treat extreme values of file metrics as detrimental. Defect prediction approaches that use a linear statistical model, conclude that an extreme value of the metrics implies poor quality of code, by virtue of choosing such a model. For example, 'larger files will have more defects' is a common refrain in ESE research [15]. Alternatively the "Goldilocks Principle" suggests that extreme values of a metric are a sign of poor quality code [17, 18]. Fenton and Neil provide a more comprehensive list of research studies with respect to metrics based defect prediction [12]. They find that the literature has contradictory evidence regarding the relationship between software defects and software metrics like size and complexity. It is, thus, unclear if metric thresholds should be used to identify source code files that are at high risk.

Consequently, we aim to observe *if a consistent relationship between metric thresholds and software quality* is present in OSS and industrial projects. Therefore we conduct a case study on three OSS and four industrial projects. The metrics that we choose to evaluate are size based (Total LOC in a file, and Module interface size in a file), and complexity based (cyclomatic complexity and module inward coupling). We evaluate software quality using two criteria - (a) defect proneness (probability of a file having a defect), and (b) defect density (the number of defects/LOC).

We replicate the most recently published state-of-the-art technique (proposed by Alves *et al.* [3]) to determine the thresholds for these metrics, and found them to be very close to ones reported earlier. In order to use this approach, we need a set of projects to calculate the thresholds from. In

Risk: McNamara Fallacy

- 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

Categorizing Methodological Risks: Threats to Validity

- Construct validity: Are we measuring the right thing? Does the treatment actually correspond to the cause/effect that we are observing?
- Internal validity: Are there other factors in our experiment that might have also had an impact on the effect?
- External validity: Do these results generalize to other contexts and environments?

Avoiding Defects: Another Take

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems (Yuan et al, OSDI 2014)

- Research question: Where are bugs in distributed systems?
- Methodology: randomly sample 198 real world failures from HDFS, Hadoop, Base, Cassandra, Redis
  - Only select priority “Blocker”, “Critical” or “Major” in the past 4 years, rejecting issues where reporter and assignee are the same
  - Manually investigate every single error: failure report, discussion, error logs, source code, patches
  - MUCH more effort than just measuring statistics

https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

Studying the reproducibility of these failures

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems (Yuan et al, OSDI 2014)

How many inputs are needed to reproduce the failures?

Num. of events	%	
1	23%	single event
2	50%	multiple events: 77%
3	17%	
4	5%	
> 4	5%	

Table 3: Minimum number of input events required to trigger the failures.

Input event type	%
Starting a service	58%
File/database write from client	32%
Unreachable node (network error, crash, etc.)	24%
Configuration change	23%
Adding a node to the running system	15%
File/database read from client	13%
Node restart (intentional)	9%
Data corruption	3%
Other	4%

Table 4: Input events that led to failures. The % column reports the percentage of failure where the input event is required to trigger the failure. Most failures require multiple preceding events, so the sum of the "%" column is greater than 100%.

Are failures deterministic?

Software	num. of deterministic failures
Cassandra	76% (31/41)
HBase	71% (29/41)
HDFS	76% (31/41)
MapReduce	63% (24/38)
Redis	79% (30/38)
Total	74% (147/198)

Table 6: Number of failures that are deterministic.

Source of non-determinism	number
Timing btw. input event & internal exe. event	27 (53%)
Multi-thread atomicity violation	13 (25%)
Multi-thread deadlock	3 (6%)
Multi-thread lock contention (performance)	4 (8%)
Other	4 (8%)
Total	51 (100%)

Table 7: Break-down of the non-deterministic failures. The "other" category is caused by nondeterministic behaviors from the OS and third party libraries.

https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

Studying the cascading nature of these failures

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems (Yuan et al, OSDI 2014)

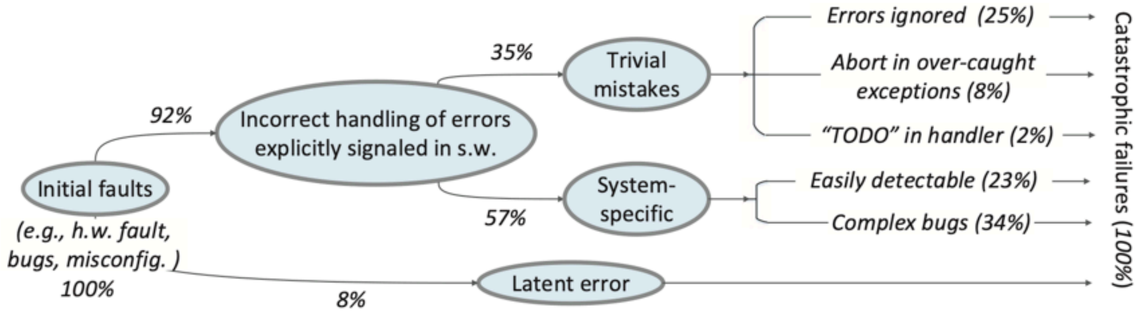


Figure 5: Break-down of all catastrophic failures by their error handling.

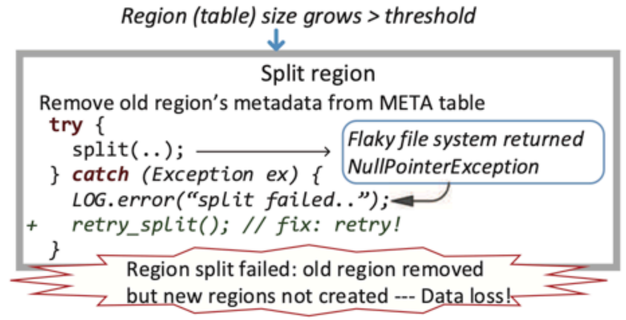


Figure 7: A data loss in HBase where the error handling was simply empty except for a logging statement. The fix was to retry in the exception handler.

```
    try {
      namenode.registerDatanode();
+   } catch (RemoteException e) {
+     // retry.
    } catch (Throwable t) {
      System.exit(-1);
    }
```

RemoteExcepion is thrown due to glitch in namenode

Only intended for IncorrectVersionException

Figure 8: Entire HDFS cluster brought down by an over-catch.

https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

Taking Action: Detecting bad error handlers

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems (Yuan et al, OSDI 2014)

- Criteria:
  - Catch block is empty or just contains a print
  - Catch block contains TODO or fix
  - Catch block for “Exception” or “Throwable” that calls System.exit()
- Actionable/useful?

System	Handler blocks	Bug					Bad practice					False pos.
		total / confirmed		ignore / abort / todo			total / confirmed		ignore / abort / todo			
Cassandra	4,365	2	2	2	-	-	2	2	2	-	-	9
Cloudstack	6,786	27	24	25	-	-	185	21	182	1	2	20
HDFS	2,652	24	9	23	-	1	32	5	32	-	-	16
HBase	4,995	16	16	11	3	2	43	6	35	5	3	20
Hive	9,948	25	15	23	-	2	54	14	52	-	2	8
Tomcat	5,257	7	4	6	1	-	23	3	17	4	2	30
Spark	396	2	2	-	-	2	1	1	1	-	-	2
YARN/MR2	1,069	13	8	6	-	7	15	3	10	4	1	1
Zookeeper	1,277	5	5	5	-	-	24	3	23	-	1	9
Total	36,745	**121**	**85**	101	4	16	**379**	**58**	354	14	11	115

Table 9: Results of applying Aspirator to 9 distributed systems. If a case belongs to multiple categories (e.g., an empty handler may also contain a “TODO” comment), we count it only once as an ignored exception. The “Handler blocks” column shows the number of exception handling blocks that Aspirator discovered and analyzed. “-” indicates Aspirator reported 0 warning.

https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

Studying Defect Density

“A Large Scale Study of Programming Languages and Code Quality in Github” (Ray et al FSE 14)

- Research question: What is the effect of programming languages on software quality?
- Methodology: Download 729 projects written in 17 language, look for correlations in languages/defects
  - Categorize languages by hand as “functional”, “strongly typed”, etc
  - Identify bug-fixing commits (includes keyword ‘error’, ‘bug’, ‘fix’, ‘issue’, ‘mistake’, ‘incorrect’, ‘fault’, ‘detect’, and ‘flaw’)
  - Build a classifier to label bugs as performance, security, etc.
  - Examine correlations between languages, language properties, bugs, bug classes

Studying Defect Density

“A Large Scale Study of Programming Languages and Code Quality in Github” (Ray et al FSE 14)

Are functional languages less buggy than others?

Table 7: **Functional languages have a smaller relationship to defects than other language classes where as procedural languages are either greater than average or similar to the average. Language classes are coded with weighted effects coding so each language is compared to the grand mean. *AIC=10419, Deviance=1132, Num. obs.=1067***

Defective Commits		
(Intercept)	−2.13	(0.10)***
log commits	0.96	(0.01)***
log age	0.07	(0.01)***
log size	0.05	(0.01)***
log devs	0.07	(0.01)***
Functional-Static-Strong-Managed	−0.25	(0.04)***
Functional-Dynamic-Strong-Managed	−0.17	(0.04)***
Proc-Static-Strong-Managed	−0.06	(0.03)*
Script-Dynamic-Strong-Managed	0.001	(0.03)
Script-Dynamic-Weak-Managed	0.04	(0.02)*
Proc-Static-Weak-Unmanaged	0.14	(0.02)***

$^{***}p < 0.001, ^{**}p < 0.01, ^{*}p < 0.05$

Result 2: *There is a small but significant relationship between language class and defects. Functional languages have a smaller relationship to defects than either procedural or scripting languages.*

Are some defect categories more common per-language?

Table 8: **While the impact of language on defects varies across defect category, language has a greater impact on specific categories than it does on defects in general. For all models above the deviance explained by language type has $p < 3.031e-07$.**

	Memory		Concurrency		Security		Failure	
(Intercept)	−7.45	(0.45)***	−8.19	(0.75)***	−7.65	(0.56)***	−6.41	(0.40)***
log commits	2.32	(0.12)***	2.60	(0.21)***	2.04	(0.15)***	2.03	(0.11)***
log age	0.31	(0.15)*	0.37	(0.24)	0.74	(0.18)***	0.14	(0.13)
log size	0.00	(0.09)	−0.23	(0.16)	−0.07	(0.11)	0.30	(0.08)***
log devs	0.18	(0.10)	0.25	(0.17)	0.46	(0.14)***	−0.03	(0.10)
C	1.81	(0.11)***	0.58	(0.22)**	0.51	(0.17)**	0.67	(0.12)***
C++	1.14	(0.10)***	1.18	(0.18)***	0.49	(0.15)**	1.15	(0.11)***
C#	−0.05	(0.17)	0.93	(0.24)***	−0.30	(0.22)	−0.02	(0.16)
Objective-C	1.47	(0.15)***	0.52	(0.29)	−0.04	(0.23)	1.16	(0.15)***
Go	0.00	(0.25)	1.73	(0.30)***	0.55	(0.27)*	−0.38	(0.24)
Java	0.59	(0.14)***	0.97	(0.22)***	0.14	(0.18)	0.32	(0.13)*
CoffeeScript	−0.36	(0.23)	−1.64	(0.55)**	−0.26	(0.26)	0.00	(0.19)
JavaScript	−0.11	(0.10)	−0.12	(0.17)	−0.02	(0.12)	−0.16	(0.09)
TypeScript	−1.32	(0.40)**	−2.15	(0.98)*	−1.34	(0.41)**	−0.34	(0.07)***
Ruby	−1.12	(0.20)***	−0.82	(0.30)**	−0.24	(0.20)	−0.34	(0.16)*
Php	−0.63	(0.17)***	−1.64	(0.35)***	0.27	(0.19)	−0.61	(0.17)***
Python	−0.44	(0.14)**	−0.21	(0.23)	0.25	(0.16)	−0.07	(0.13)
Perl	0.41	(0.36)	−0.86	(0.86)	−0.16	(0.43)	−0.40	(0.40)
Scala	−0.41	(0.18)*	0.73	(0.25)**	−0.16	(0.22)	−0.91	(0.19)***
Clojure	−1.16	(0.27)***	0.10	(0.30)	−0.69	(0.26)**	−0.53	(0.19)**
Erlang	−0.53	(0.23)*	0.76	(0.29)**	0.73	(0.22)***	0.65	(0.17)***
Haskell	−0.22	(0.20)	−0.17	(0.32)	−0.31	(0.26)	−0.38	(0.19)
AIC		3033.55		2234.29		3411.57		4188.71
BIC		3143.90		2344.63		3521.92		4299.06
Log Likelihood		-1494.77		-1095.14		-1683.79		-2072.36
Deviance		905.38		666.41		921.50		1070.60
Num. obs.		1114		1114		1114		1114
Residual Deviance (NULL)		8508.6		3426.5		3372.5		6071.5
Language Type Deviance		587.2		157.03		61.40		303.8

$^{***}p < 0.001, ^{**}p < 0.01, ^{*}p < 0.05$

Result 4: *Defect types are strongly associated with languages; Some defect type like memory error, concurrency errors also depend on language primitives. Language matters more for specific categories than it does for defects overall.*

Threats to Validity

As stated in Section 5 of "A Large Scale Study of Programming Languages and Code Quality in Github" (Ray et al FSE 14)

- Why not use a bug database? "We wanted to capture the issues that developers continuously face in an ongoing development process, not just the reported bugs"
- Labeling projects by domain might be biased, but another author double-checked
- Categorization might be biased, but hand-sampled 180 random fixes and found overall precision 84%, recall 84%
- Categorizing languages (e.g. "functional", "procedural") could be subjective
- Outside factors might impact the incidence of bugs-per-language

Brainstorm: Threats to Validity

“A Large Scale Study of Programming Languages and Code Quality in Github” (Ray et al FSE 14)

- Research question: What is the effect of programming languages on software quality?
- Methodology: Download 729 projects written in 17 language, look for correlations in languages/defects
  - Categorize languages by hand as “functional”, “strongly typed”, etc
  - Identify bug-fixing commits (includes keyword ‘error’, ‘bug’, ‘fix’, ‘issue’, ‘mistake’, ‘incorrect’, ‘fault’, ‘detect’, and ‘flaw’)
  - Build a classifier to label bugs as performance, security, etc.
  - Examine correlations between languages, language properties, bugs, bug classes

There were more threats to validity.

- Bug categories are inconsistently labeled
- Not all commits were included, some were duplicates
- Detecting “fix” commits is very inaccurate
- Other potential threats: project age, bug rate, developers; only consider open source software
- Most important: is this even a question we should try to answer?

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

How to determine what to study?

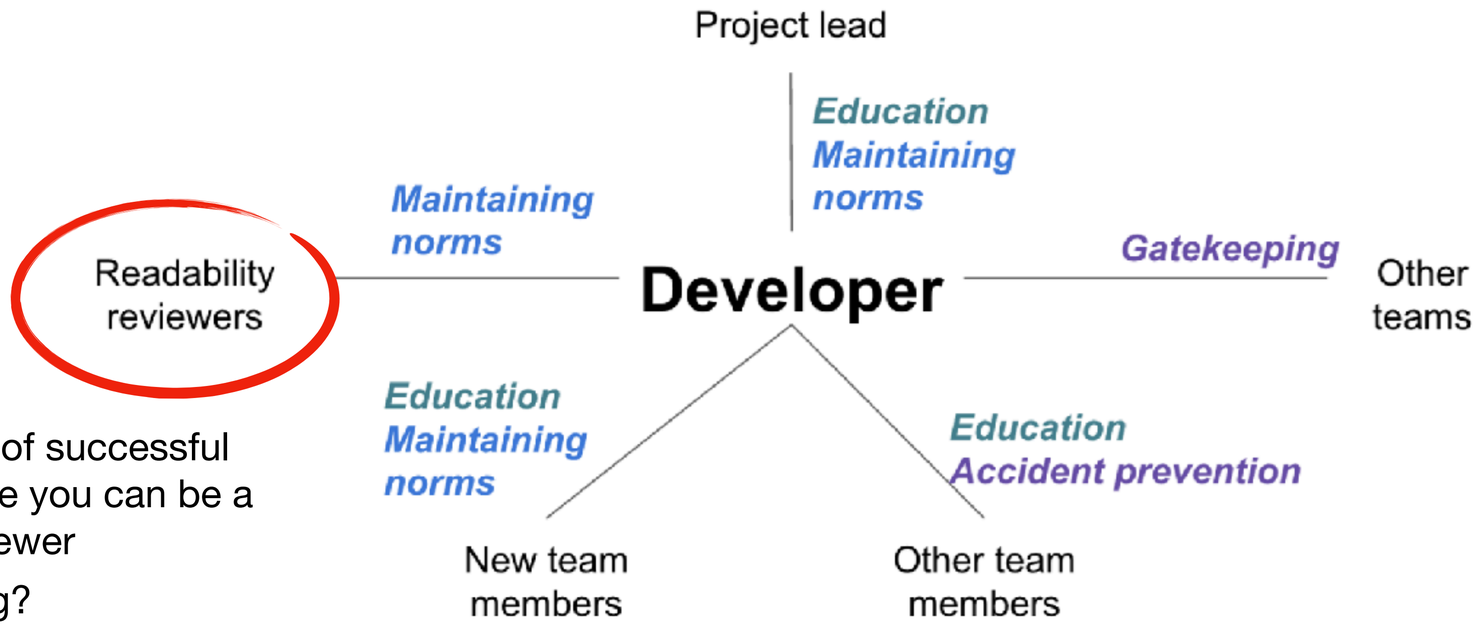
Consider these questions from the experts at Google's SE productivity team

- What result are you expecting to find, and why?
- If the data supports the expected result, what action will be taken?
- If the data supports a negative result, will appropriate action be taken?
- Who is going to decide to take action on the result, and when would they do it?

[Software Engineering @ Google Ch 7, C. Jaspan]

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, C. Jaspan]

Engineering Productivity: A Broad Goal

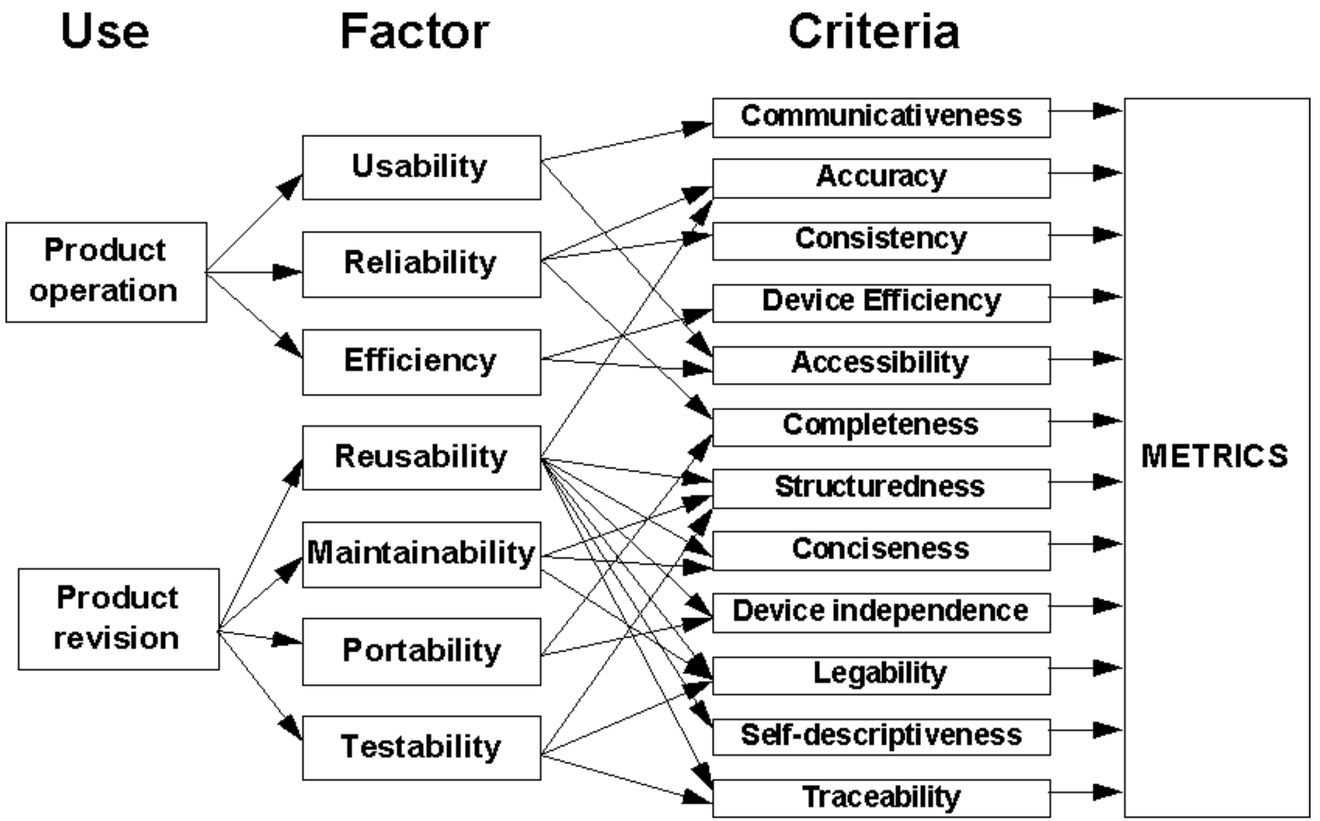
QUANTS components

- **Qu**ality of the code (Is it tested? Is it maintainable?)
- **A**ttention from engineers (Does the process distract engineers?)
- I**n**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, C. Jaspan]

From Quality Goals to Metrics

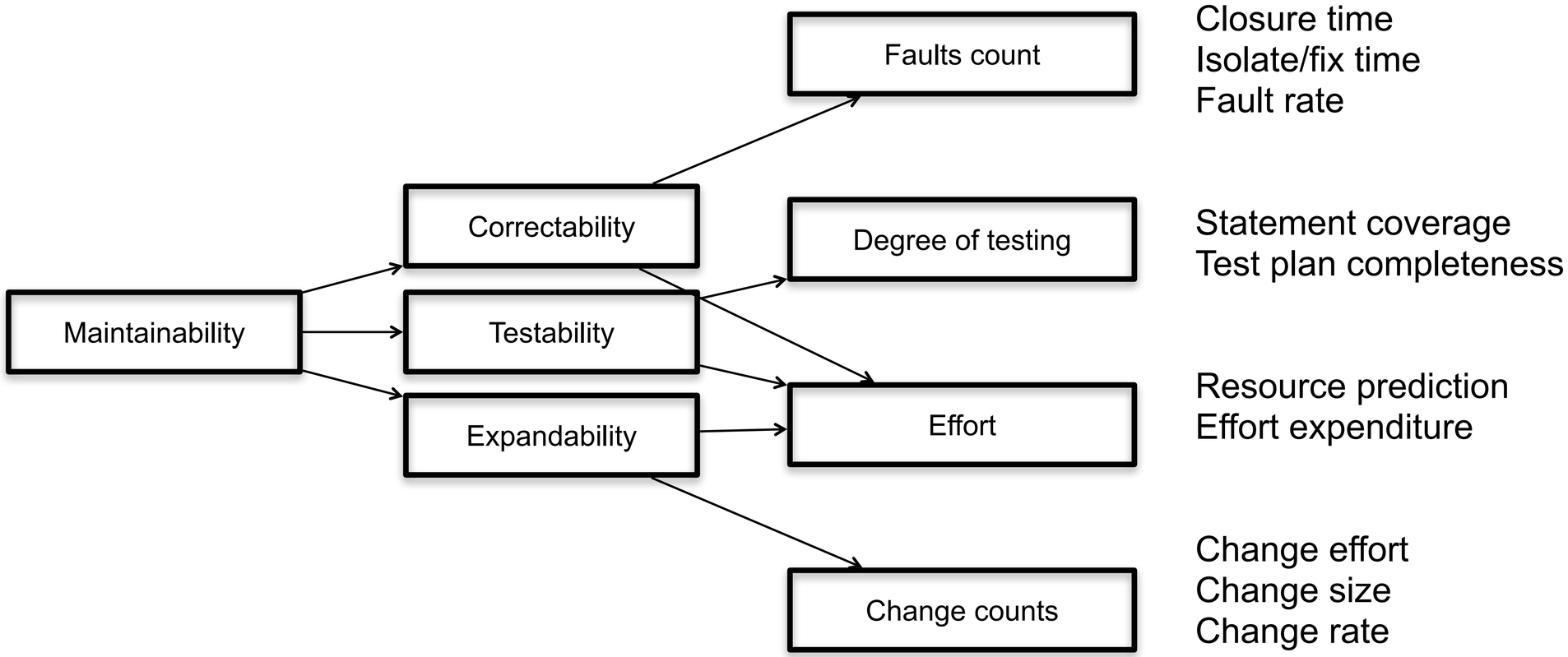
McCall Quality Model



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

From Quality Goals to Metrics

McCall Quality Model



From Goals to Signals and Metrics

Readability Review

- Goal: “Engineers are more productive as a result of the readability process.” [Partial list]
  - Signal: “Engineers who have been granted readability judge themselves to be more productive than engineers who have not been granted readability.”
    - Metric: “Quarterly Survey: Proportion of engineers reporting that they’re highly productive”
  - Signal: “Changelists (CLs) written by engineers who have been granted readability are faster to review than CLs written by engineers who have not been granted readability.”
    - Metric: “Logs data: Median review time for CLs from authors with readability and without readability”
  - Signal: “CLs written by engineers who have been granted readability are easier to shepherd through code review than CLs written by engineers who have not been granted readability.”
    - Metric: “Logs data: Median shepherding time for CLs from authors with readability and without readability”

[Software Engineering @ Google Ch 7, C. Jaspan]

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, C. Jaspan]

Readability Review: The Conclusion

- Engineers who had readability:
  - Felt satisfied
  - Felt that they learned from the process
  - Had their code reviewed faster
- Survey data identified pain points in the process that were folded into the process
- Readability continues.

[Software Engineering @ Google Ch 7, C. Jaspan]

Code Review on GitHub

“First Come First Served: The Impact of File Position on Code Review” (Fregnan et al ESEC/FSE 2022)

- Research question: “Does the order in which a file is shown in a review bias the outcome of the review process?
- Methodology:
  - Study 138 open source projects on GitHub to identify correlation between order and number of comments
  - Controlled experiment, participants review code with seeded defects, vary the ordering

Code Review on GitHub

“First Come First Served: The Impact of File Position on Code Review” (Fregnan et al ESEC/FSE 2022)

Does file position influence number of comments?

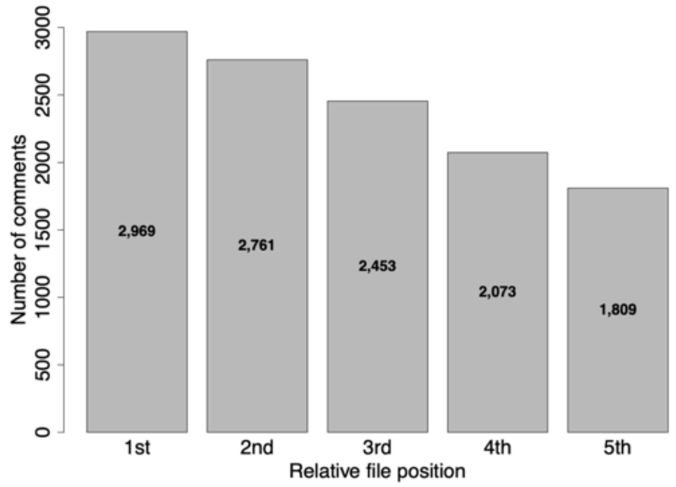


Figure 4: Distribution of comments in a PR with five files.

Does file position influence number of comments?

Table 5: Participants who found/not found each bug divided by treatment: Corner Case defect (CC) first or Missing Break defect (MB) first.

	Corner Case		Missing Break	
	Found	Not found	Found	Not found
CC_f-MB_l	34	22	24	32
MB_f-CC_l	18	32	26	24
	p-value: **0.011**		p-value: 0.346	
	phi-coefficient: 0.247		phi-coefficient: 0.091	
	odds ratio: 2.75		odds ratio: 1.44	

Does file position influence review time?

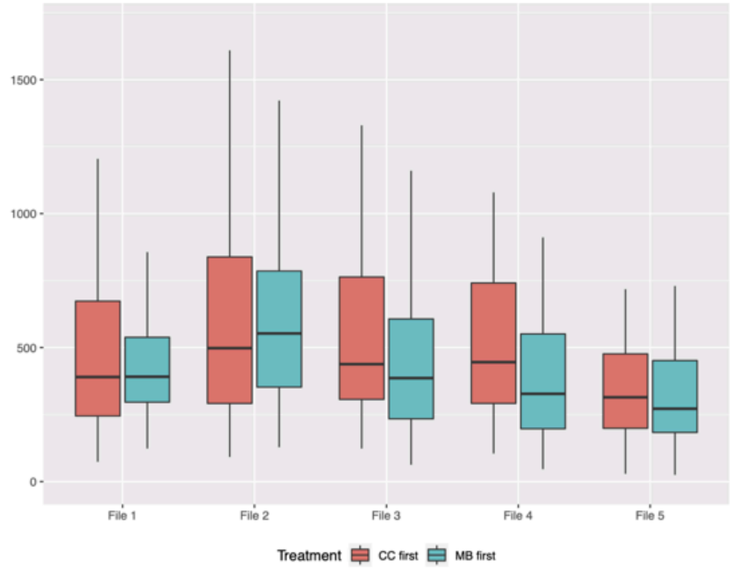


Figure 6: Time (in seconds) participants visualized each file. To improve the clarity, we limited the size of the Y axis.

On Productivity Incentives

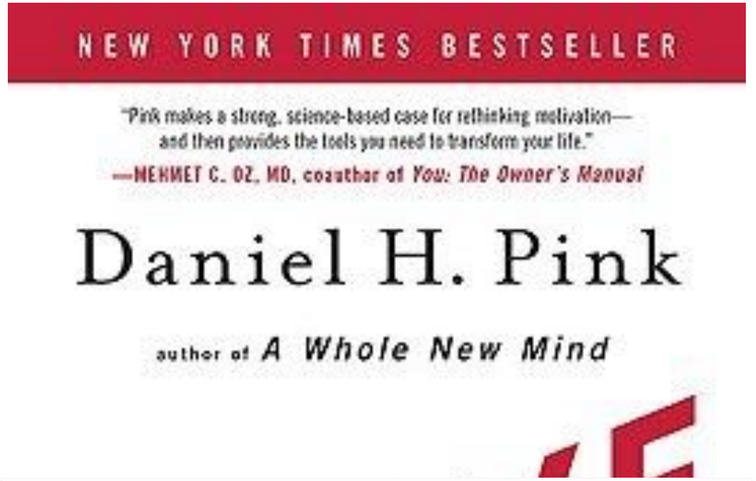
Goodhart's Law: When a measure becomes a target, it ceases to be a good measure

Productivity Metrics

Intrinsic & Extrinsic Motivations

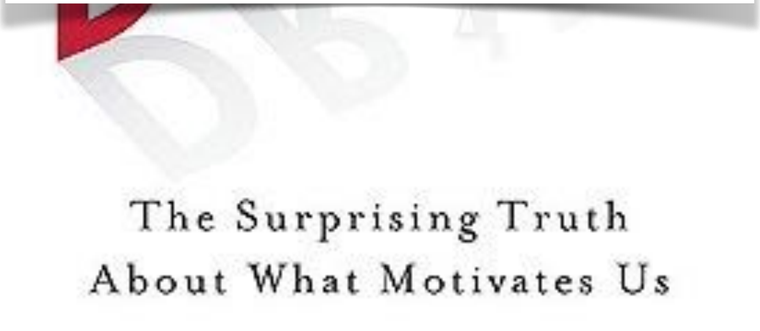
Extrinsic rewards:

Can extinguish intrinsic motivation
Can diminish performance
Can crush creativity
Can crowd out good behavior
Can encourage cheating, shortcuts, and unethical behavior
Can become addictive
Can foster short-term thinking



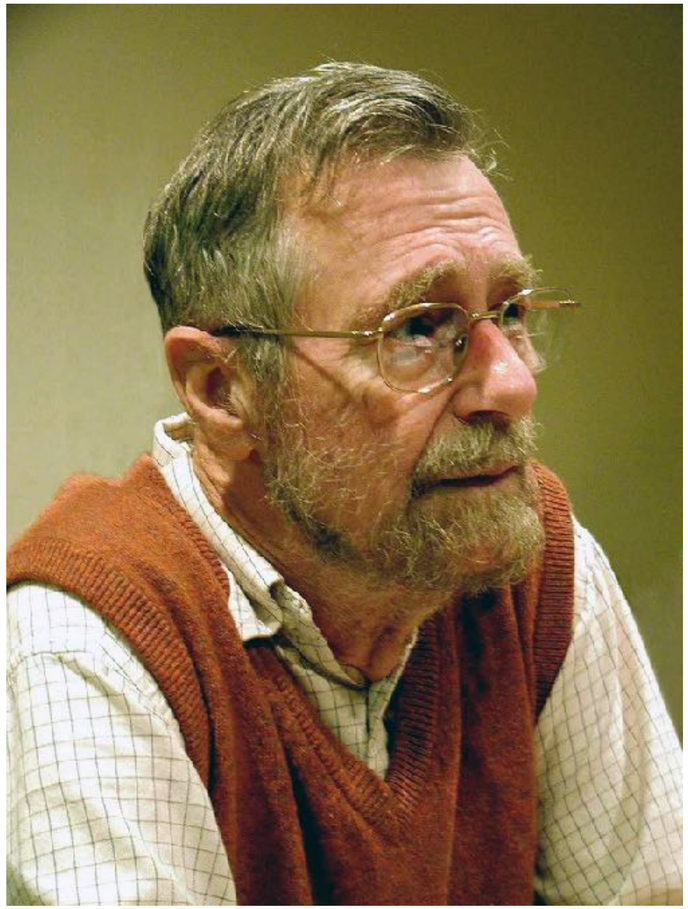
Focus on encouraging:

Autonomy
Mastery
Purpose



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