Software Code Quality Measurement: Implications from Metric Distributions

Si Yuan Jin[1,2], **Ziyuan Li**[3,1,*], **Bichao Chen**[1,*], **Bing Zhu**[1,*], **and Yong Xia**[1,*]

Presenter: **Bruce Si Yuan Jin**
[1] HSBC Laboratory, Guangzhou, China
[2] School of Business and Management, Hong Kong University of Science and Technology, Hong Kong, China
[3] School of Physics, Sun Yat-sen University, Guangzhou, China
Acknowledgment: Y. Xia is partly supported by the "Pioneering Innovator" award from the Guangzhou Tianhe District government. Z. Li is partly supported by the Guangdong Basic and Applied Basic Research Foundation (2021A1515012039). We would like to acknowledge useful discussions and support from our colleagues at the HSBC Lab.

December 17, 2023

Background

- The initial definition on code quality is the collective features of software that meet given needs (Fitzpatrick, 1996).
- **Precise code quality measurement** can improve software products, increase user satisfaction, and save costs of IT systems (Kekre et al., 1995), which influences the success and adoption of software (Levine and Toffel, 2010).

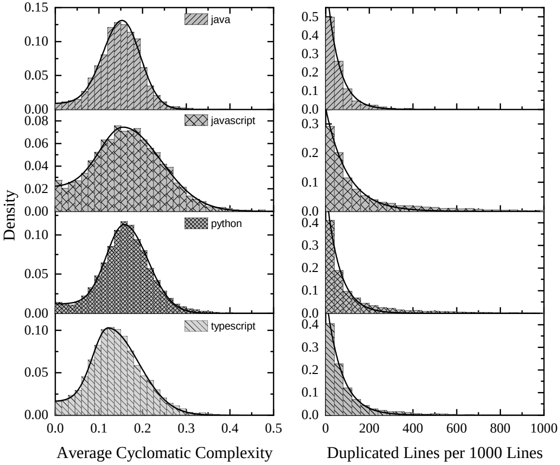
Foreground

- Code quality has dimensions (Polites et al., 2012). ISO/IEC 25010 standard (Klima et al., 2022): **maintainability** (Motogna et al., 2023), **readability** (González-Prieto et al., 2023), and **functionality** (Shen et al., 2020).
- Code quality is a **multi-dimensional** construct (Edwards, 2001):

Construct	Definition	Dimensions	Definition
Code Quality	How well-written the code is, including maintainability, reliability, and functionality. (Lee et al., 2009)	Maintainability	The code is easy to understand, enhance, or correct. (Deligiannis et al., 2003)
		Reliability	The code is user-friendly and stable. (Lee et al., 2009)
		Functionality	The code has useful functions. (Lee et al., 2009)

Foreground

- 20 distinct metric measurements from literature.
- **Two types**: Monotonic metrics and non-monotonic metrics.



Research Question

- **No uniform solution**: The literature lacks methodologies for evaluating code quality metrics, especially for non-monotonic metrics.
- **Research Question**: How to propose a consistent method to evaluate them?

Theoretical Relevance

- Code quality's metric measurements (Bianchi et al., 2012): the size of components (Stamelos et al., 2002), code complexity (McCabe, 1976; Shin et al., 2010).
- However, existing metric identifications have focused on **monotonic areas** rather than **non-monotonic metrics**.
- Our paper considers both and proposes a uniform solution for them.

Pracitical Relevance

- Many firms utilize and contribute to OSS (Mehra et al., 2011) and developers reuse OSS to lower their search cost (Haefliger et al., 2008), which requires high code quality.
- Performance evaluations for software:
  - Inappropriate performance measurements – a major cause of IT systems failing (Kekre et al., 1995; Fitoussi and Gurbaxani, 2012).
  - As new technologies and techniques emerge (Jin and Xia, 2022), more precise measurements of software quality and fit are needed.
- We study OSS because of their high code quality (Ljungberg, 2000; Von Krogh and Von Hippel, 2006).

Metric Identification

We normalized metrics based on the number of functions, lines, and classes.

Dimension	Metric	Definition
Maintainability	Cyclomatic Complexity	Number of independent paths through code.
	File Complexity	Cyclomatic complexity averaged by files.
	Cognitive Complexity	Combination of cyclomatic complexity and human assessment.
	Code Smells	Number of code smell issues.
	Coupling Between Objects	Number of classes that are coupled to a particular class.
	Fan-in	Number of input dependencies a class has.
	Fan-out	Number of output dependencies a class has.
	Depth Inheritance Tree	Number of "fathers" a class has.
	Number of Children	Number of immediate subclasses that a particular class has.
	Lack of Cohesion of Methods	Degree to which class methods are coupled.
	Tight Class Cohesion	Ratio of the number of pairs of directly related methods in a class to the maximum number of possible methods in the class.
	Loose Class Cohesion	Ratio of the number of directly or indirectly related method pairs in a class to the maximum number of possible method pairs.
Reliability	Total Violations	Number of issues including all severity levels.
	Critical Violations	Number of issues of the critical severity.
	Info Violations	Number of issues of the info severity.
Functionality	Line to Cover	Lines to be covered by unit tests.
	Comment Lines	Number of comment lines.
	Duplicated Blocks	Number of duplicated blocks of line.
	Duplicated Files	Number of files involved in duplicated blocks.
	Duplicated Lines	Number of lines involved in duplicated blocks.

Distribution-based Evaluation

- Distribution fitting of each metric in high-star OSS repositories.
- Score them according to their locations in the distributions.
- **Overall score**: weights to individual scores. for the overall score for repository k:

$$Q_k^{overall} = \sum_i \omega_i \cdot Q_{i,k}^{metric}, \text{subject to:} \sum_i \omega_i = 1. \quad (1)$$

- The weights ω_i can be obtained from calculating the importance of scores to a code quality reflective measurement, such as the number of GitHub stars Medappa and Srivastava (2019).

Monotonic Metrics

We fit an exponential distribution to the monotonic-metric data.

$$f_1(x; c, \lambda) = \begin{cases} 0 & \text{if } x \leq c \\ \lambda \exp\left[-\lambda(x - c)\right] & \text{if } x > c \end{cases} \tag{2}$$

where λ and c are the fitting parameters. The corresponding score function based on the CDF of Eq. (2) reads as

$$M_1(x; c, \lambda) = 100 \times \begin{cases} 1 & \text{if } x \leq c \\ \exp\left[-\lambda(x - c)\right] & \text{if } x > c \end{cases} \tag{3}$$

Non-monotonic Metrics

The non-monotonic metrics follow an asymmetric Gaussian distribution, the PDF of which reads as

$$f_2(x;\mu,\sigma_1,\sigma_2) = \begin{cases} \frac{1}{\sqrt{2\pi}}\frac{2}{\sigma_1+\sigma_2}\exp\left(-\frac{(x-\mu)^2}{2\sigma_1^2}\right) & \text{if } 0 \leq x < \mu \\ \frac{1}{\sqrt{2\pi}}\frac{2}{\sigma_1+\sigma_2}\exp\left(-\frac{(x-\mu)^2}{2\sigma_2^2}\right) & \text{if } x \geq \mu \end{cases} \tag{4}$$

where $\mu, c, \sigma_1, \sigma_2$ represent the peak position, peak height on the right, and peak widths on each side, respectively. The corresponding score function is

$$M_2(x,\mu,\sigma_1,\sigma_2) = 100 \times \begin{cases} 1-\mathrm{erf}\left(\frac{x-\mu}{\sigma_1\sqrt{2}}\right) & \text{if } 0 \leq x < \mu \\ 1-\mathrm{erf}\left(\frac{x-\mu}{\sigma_2\sqrt{2}}\right) & \text{if } x \geq \mu \end{cases} \tag{5}$$

where the score falls into the range of $0 \sim 100$, peaks at μ, and decays according to the Z-score of the Gaussian function on each side.

Data

- The top $\sim 20,000$ repositories for each programming language.
- **Exclusion**: non-engineering repositories, such as a guide for Java interviews in JavaGuide.
- **Metrics Data:** We used code scanners to obtain metrics. Scripting language (Python, Javascript, TypeScript) repositories can be directly imported, while non-scripting Java repositories need to be compiled first.
- Therefore, we only chose repositories with GitHub releases for compilation, which led to **36,460** repositories and over **600 million** lines of code.

Distribution Fitting - Monotonic Metrics

- **Higher-tolerance**: The threshold parameter *c* for 'Code Smells' in Java approximates 1.
- **Low-sensitive Metrics:** $\lambda \lesssim 1$ is observed for metrics such as 'File Complexity', 'Depth Inheritance Tree', 'Number of Children', 'Duplicated Blocks', and 'Duplicated Files'.
- **High-sensitive Metrics:** Total violation, Code Smells.

Metric	Java$(\boldsymbol{c}, \boldsymbol{\lambda})$	JavaScript$(\boldsymbol{c}, \boldsymbol{\lambda})$	Python$(\boldsymbol{c}, \boldsymbol{\lambda})$	TypeScript$(\boldsymbol{c}, \boldsymbol{\lambda})$
File Complexity	(0,0.485)	(0,0.884)	(0,0.917)	(0,0.492)
Code Smells	(1.123,50.731)	(0.036,60.260)	(0.004,37.177)	(0.017,16.530)
Depth Inheritance Tree	(1.003,0.502)	/	/	/
Number of Children	(0.002,0.137)	/	/	/
Lack of Cohesion of Methods	(0.053,80.004)	/	/	/
Total Violations	(1.160,54.376)	(0.054,63.313)	(0.004,387.551177)	(0.021,18.168)
Critical Violations	(0.019,9.872)	(0.020,48.811)	(0.007,9.443)	(0.005,5.497)
Info Violations	(0.019,1.934)	(0.001,1.436)	(0.002,1.401)	(0.003,1.535)
Line to Cover	(0,0.000)	(0,0.000)	(0,0.000)	(0,0.000)
Duplicated Blocks	(0,0.015)	(0.001,0.021)	(0,0.010)	(0,0.021)
Duplicated Files	(0.003,0.135)	(0.001,0.203)	(0,0.222)	(0,0.116)
Duplicated Lines	(0.439,63.284)	(0.145,163.258)	(0.081,124.342)	(0.085, 102.796)

Distribution Fitting - Non-monotonic Metrics

- ”Comment lines” for Javascript and Typescript are almost monotonic ($\mu = 0$), potentially because they are generally easy to understand.
- **Asymmetric Sensitivity:** For ’Comment lines’ (Python), the sensitivity is large (small) before (after) the central point.

Metric	Java($\boldsymbol{\mu}$,$\boldsymbol{\sigma_1}$,$\boldsymbol{\sigma_2}$)	JavaScript($\boldsymbol{\mu}$,$\boldsymbol{\sigma_1}$,$\boldsymbol{\sigma_2}$)	Python($\boldsymbol{\mu}$,$\boldsymbol{\sigma_1}$,$\boldsymbol{\sigma_2}$)	TypeScript($\boldsymbol{\mu}$,$\boldsymbol{\sigma_1}$,$\boldsymbol{\sigma_2}$)
Cyclomatic Complexity	(155.228,50.947,40.902)	(166.692,88.415,78.289)	(162.321,53.497,52.789)	(127.273,51.616,66.733)
Cognitive Complexity	(50.870,40.120,75.664)	(33.238,32.586,121.541)	(170.042,33.546,0.000)	(29.619,22.964,81.617)
Comment Lines	(15.841,11.451,137.269)	(0.007,6.575,96.312)	(91.730,64.805,148.192)	(0.002,9.300,72.443)
Fan-in	(1.101,0.463,1.217)	/	/	/
Fan-out	(5.181,2.043,4.639)	/	/	/
Loose Class Cohesion	(0.329,0.149,0.176)	/	/	/
Tight Class Cohesion	(0.228,0.100,0.128)	/	/	/
Coupling Between Objects	(7.055,2.580,5.086)	/	/	/

Importance Weights - Reliability

- **Reliability**: 'Total Violations' contributes mostly to Java, while the 'Critical Violations' is the most important for the other three languages.
- **Priority**: mitigating all violations for Java repositories; mitigating critical violations for other three languages.

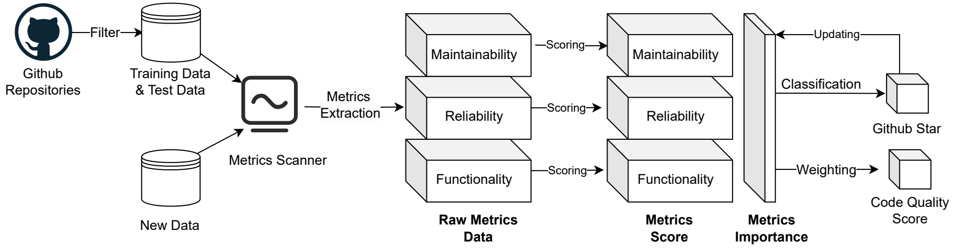
Dimension	**Metric**	**Importance**			
		Java	**JavaScript**	**Python**	**TypeScript**
Reliability	Total Violations	0.474 (0.056)	0.288 (0.070)	0.293 (0.068)	0.228 (0.065)
	Critical Violations	0.272 (0.032)	0.420 (0.102)	0.410 (0.095)	0.414(0.118)
	Info Violations	0.254 (0.030)	0.292 (0.071)	0.297 (0.069)	0.358 (0.102)
	Sum	1 (0.118)	1 (0.243)	1 (0.232)	1 (0.285)

Importance Weights - Functionality

- **Functionality**: the 'Comment Lines' metric score explains the most for Java adoption, implying its influential role in adopting Java OSS repositories.
- Java might be less intuitive to understand, thereby making code comments essential for understanding Java codes.

Dimension	Metric	Importance			
		Java	JavaScript	Python	TypeScript
Functionality	Line to Cover	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)
	Comment Lines	0.454 (0.059)	0.317 (0.103)	0.370 (0.107)	0.318 (0.112)
	Duplicated Blocks	0.162 (0.021)	0.286 (0.093)	0.197 (0.057)	0.148 (0.052)
	Duplicated Files	0.190 (0.025)	0.120 (0.039)	0.166 (0.048)	0.179 (0.063)
	Duplicated Lines	0.194 (0.025)	0.277 (0.090)	0.267 (0.077)	0.355 (0.125)
	Sum	1 (0.130)	1 (0.325)	1 (0.289)	1 (0.352)

Model Workflow Summary and Explainability



Language	Java	JavaScript	Python	TypeScript
Accuracy	0.947	0.826	0.808	0.817
Precision	0.971	0.838	0.831	0.834
Recall	0.917	0.803	0.771	0.784
F1	0.943	0.820	0.800	0.808
AUC_ROC	0.946	0.826	0.815	0.817
R2	0.787	0.274	0.186	0.247

Conclusion

- Code quality with three dimensions: maintainability, reliability, and functionality.
- We evaluate metrics based on their distributions.
- **Contribution**: Our study advances the understanding of code quality and contributes to better quality control standards and practices, ultimately supporting the OSS success.
- **Limitations**:
  - Not yet systematically validated the effectiveness of our method.
  - Parameters of fitted distributions are sensitive to data distribution, making it necessary to incorporate more data for determining them.

Comments welcome!

References

Bianchi, A. J., Kang, S. M., and Stewart, D. (2012). The Organizational Selection of Status Characteristics: Status Evaluations in an Open Source Community. *Organization Science*, 23(2):341–354.

Deligiannis, I., Shepperd, M., Roumeliotis, M., and Stamelos, I. (2003). An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139.

Edwards, J. R. (2001). Multidimensional constructs in organizational behavior research: An integrative analytical framework. *Organizational research methods*, 4(2):144–192.

Fitoussi, D. and Gurbaxani, V. (2012). It outsourcing contracts and performance measurement. *Information Systems Research*, 23(1):129–143.

Fitzpatrick, R. (1996). Software quality: definitions and strategic issues. *Report*, 1.

González-Prieto, Á., Perez, J., Diaz, J., and López-Fernández, D. (2023). Reliability in software engineering qualitative research through inter-coder agreement. *Journal of Systems and Software*, 202:111707.

Haefliger, S., Von Krogh, G., and Spaeth, S. (2008). Code reuse in open source software. *Management science*, 54(1):180–193.

Jin, S. Y. and Xia, Y. (2022). Cev framework: A central bank digital currency evaluation and verification framework with a focus on consensus algorithms and operating architectures. *IEEE Access*, 10:63698–63714.

Kekre, S., Krishnan, M. S., and Srinivasan, K. (1995). Drivers of customer satisfaction for software products: implications for design and service support. *Management science*, 41(9):1456–1470.

Klima, M., Bures, M., Frajtak, K., Rechtberger, V., Trnka, M., Bellekens, X., Cerny, T., and Ahmed, B. S. (2022). Selected code-quality characteristics and metrics for internet of things systems. *IEEE Access*, 10:46144–46161.

Lee, S.-Y. T., Kim, H.-W., and Gupta, S. (2009). Measuring open source software success. *Omega*, 37(2):426–438.

Levine, D. I. and Toffel, M. W. (2010). Quality management and job quality: How the iso 9001 standard for quality management systems affects employees and employers. *Management Science*, 56(6):978–996.

Ljungberg, J. (2000). Open source movements as a model for organising. *European Journal of Information Systems*, 9(4):208–216.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, 0(4):308–320.

Medappa, P. K. and Srivastava, S. C. (2019). Does superposition influence the success of floss projects? an examination of open-source software development by organizations and individuals. *Information Systems Research*, 30(3):764–786.

Mehra, A., Dewan, R., and Freimer, M. (2011). Firms as incubators of open-source software. *Information Systems Research*, 22(1):22–38.

Motogna, S., Vescan, A., and Şerban, C. (2023). Empirical investigation in embedded systems: Quality attributes in general, maintainability in particular. *Journal of Systems and Software*, 201:111678.

Polites, G. L., Roberts, N., and Thatcher, J. (2012). Conceptualizing models using multidimensional constructs: a review and guidelines for their use. *European Journal of Information Systems*, 21:22–48.

Shen, Q., Wu, S., Zou, Y., Zhu, Z., and Xie, B. (2020). From api to nli: A new interface for library reuse. *Journal of Systems and Software*, 169:110728.

Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. (2010). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE*