

# Defects Density

YEGOR BUGAYENKO

Lecture #18 out of 24

80 minutes

The slidedeck was presented by the author in this [YouTube Video](#)

All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as web sites. Copyright belongs to their respected authors.



1. Defects are not just problems—they are mostly opportunities.



VICTOR R. BASILI

“Scatter plots for executable lines per module versus the number of errors found in the module were graphed. It was difficult to see any trend within these plots, so the number of errors/1000 executable lines within a module size was calculated.”

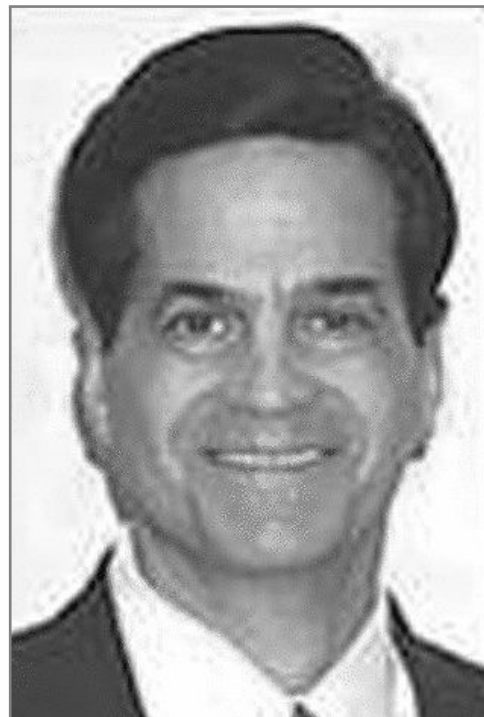
— Victor R. Basili and Barry T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42–52, 1984.  
[doi:10.1145/69605.2085](https://doi.org/10.1145/69605.2085)

TABLE IX. Complexity and Error Rate for Errored Modules

Module Size	Average Cyclomatic Complexity	Errors/1000 Executable Lines
50	6.2	65.0
100	19.6	33.3
150	27.5	24.6
200	56.7	13.4
>200	77.5	9.7

“One surprising result was that module size did not account for error proneness. In fact, it was quite the contrary—the larger the module, the less error prone it was. This was true even though the larger modules were more complex.”

Source: Victor R. Basili and Barry T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1): 42–52, 1984. doi:[10.1145/69605.2085](https://doi.org/10.1145/69605.2085)



MICHAEL FAGAN


“Feedback of results from inspections must be counted for the programmer’s use and benefit: they should not under any circumstances be used for programmer performance appraisal.”

— Michael Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 38(3):258–287, 1999. doi:[10.1147/sj.382.0258](https://doi.org/10.1147/sj.382.0258)

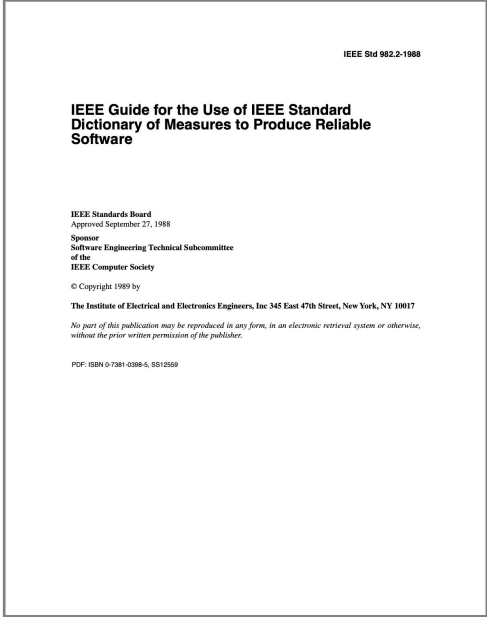
Figure 8 Example of most error-prone modules based on  $I_1$  and  $I_2$

<i>Module name</i>	<i>Number of errors</i>	<i>Lines of code</i>	<i>Error density, Errors/K. Loc</i>
Echo	4	128	31
Zulu	10	323	31
Foxtrot	3	71	28
Alpha	7	264	27 ← Average
Lima	2	106	19 Error
Delta	3	195	15 Rate
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
	67		

Source: Michael Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 38(3):258–287, 1999. doi:[10.1147/sj.382.0258](https://doi.org/10.1147/sj.382.0258)



2. Any imperfection is a defect that should be reported and accounted for.



“A defect is a product anomaly. Examples include such things as 1) omissions and imperfections found during early life cycle phases and 2) faults contained in software sufficiently mature for test or operation.”

— IEEE Standards Board. IEEE Std 982.2-1988: Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, 1989



$$\begin{aligned} I &= 7 \\ KSLOD &= 8 \end{aligned}$$

Then,

$$\sum_{i=1}^7 D_i = 78 \text{ (total defects found)}$$

$$DD = \frac{78}{8} = 9.8 \text{ (estimated defect density)}$$

Source: IEEE Standards Board. IEEE Std 982.2-1988:  
Guide for the Use of IEEE Standard Dictionary of  
Measures to Produce Reliable Software, 1989

“This measure has a degree of indeterminism. For example, a low value may indicate either a good process and a good product or it may indicate a bad process. If the value is low compared to similar past projects, the inspection process should be examined. If the inspection process is found to be adequate, it should then be concluded that the development process has resulted in a relatively defect-free product.”

Measures (Experience)	Product Measures						Process Measures		
	Errors, Faults, Failures	Mean Time to Failure; Failure Rate	Reliability Growth & Projection	Remaining Product Faults	Completeness & Consistency	Complexity	Management Control	Coverage	Risk, Benefit, Cost Evaluation
1. Fault density (2)	X								
2. Defect density (3)	X								
3. Cumulative failure profile (1)	X								
4. Fault-days number (0)	X						X		
5. Functional or modular test coverage (1)					X			X	X
6. Cause and effect graphing (2)					X			X	
7. Requirements traceability (3)	X				X			X	
8. Defect indices (1)	X						X		
9. Error distribution(s) (1)							X		
10. Software maturity index (1)			X						X
11. Man hours per major defect detected (2)							X		X
12. Number of conflicting requirements (2)	X				X			X	
13. Number of entries/exists per module (1)					X	X			
14. Software science measures (3)				X		X			
15. Graph-theoretic complexity for architecture (1)					X	X			
16. Cyclomatic complexity (3)					X	X			
17. Minimal unit test case determination (2)					X	X			
18. Run reliability (2)			X						
19. Design structure (1)						X			
20. Mean time to discover the next <i>K</i> faults (3)									X
21. Software purity level (1)			X						
22. Estimated number of faults remaining (seeding) (2)				X					
23. Requirements compliance (1)	X				X			X	
24. Test coverage (2)					X			X	
25. Data or information flow complexity (1)						X			
26. Reliability growth function (2)			X						
27. Residual fault count (1)				X					
28. Failure analysis using elapsed time (3)			X	X					
29. Testing sufficiency (0)			X					X	
30. Mean-time-to-failure (3)		X	X						
31. Failure rate (3)		X							
32. Software documentation & source listings (2)					X				
33. RELY - (Required Software Reliability) (1)								X	X
34. Software release readiness (0)									X
35. Completeness (2)					X				
36. Test accuracy (1)				X	X			X	
37. System performance reliability (2)			X						
38. Independent process reliability (0)			X						
39. Combined HW/SW system operational availability (0)			X						

Table 4.1-1 — Measure Classification Matrix

Source: IEEE Standards Board. IEEE Std 982.2-1988: Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, 1989

## 39 Measures for Reliable Software

- |  |   |   |
|--|---|---|
| 1. Fault Density                           | 14. Software Science Measures               | 27. Residual Fault Count                  |
| 2. Defect Density                          | 15. Graph-Theoretic Complexity for Arch.    | 28. Failure Analysis Using Elapsed Time   |
| 3. Cumulative Failure Profile              | 16. Cyclomatic Complexity                   | 29. Testing Sufficiency                   |
| 4. Fault-Days Number                       | 17. Minimal Unit Test Case Determination    | 30. Mean Time to Failure                  |
| 5. Functional or Modular Test Coverage     | 18. Run Reliability                         | 31. Failure Rate                          |
| 6. Cause and Effect Graphing               | 19. Design Structure                        | 32. Software Docmnt and Source Listings   |
| 7. Requirements Traceability               | 20. Mean Time to Discover the Next K Faults | 33. RELY-Required Software Reliability    |
| 8. Defect Indices                          | 21. Software Purity Level                   | 34. Software Release Readiness            |
| 9. Error Distribution(s)                   | 22. Estimated Num. of Faults Remaining      | 35. Completeness                          |
| 10. Software Maturity Index                | 23. Requirements Compliance                 | 36. Test Accuracy                         |
| 11. Manhours per Major Defect Detected     | 24. Test Coverage                           | 37. System Performance Reliability        |
| 12. Number of Conflicting Requirements     | 25. Data or Information Flow Complexity     | 38. Independent Process Reliability       |
| 13. Number of Entries and Exits per Module | 26. Reliability Growth Function             | 39. Combined H&S Operational Availability |

Source: IEEE Standards Board. IEEE Std 982.2-1988: Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, 1989



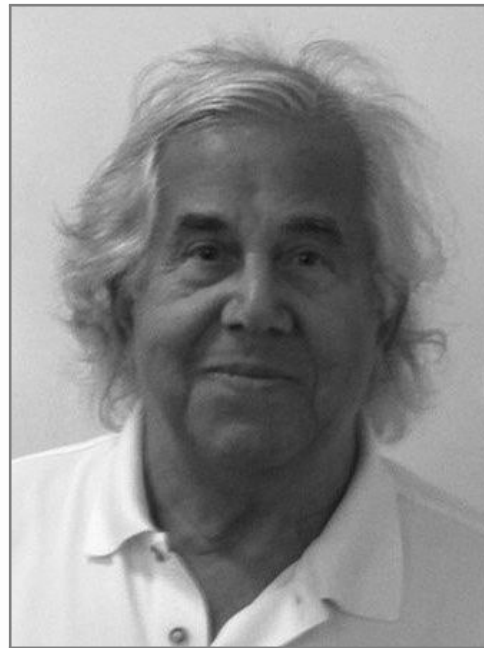
3. Defect Density (DD), as a metric, can be predicted and even planned for.



HARLAN D. MILLS

“While our experience in applying statistical quality-control techniques to software development is limited, initial experience indicates that five fixes per thousand lines of code can be tolerated without invalidating the application of statistics to estimate MTTF. This failure rate is low compared to normal development practices, where 20 to 60 fixes per thousand lines of code is not atypical.”

— Richard H. Cobb and Harlan D. Mills. Engineering Software Under Statistical Quality Control. *IEEE Software*, 7(6):45–54, 1990. doi:[10.1109/52.60601](https://doi.org/10.1109/52.60601)



JOSEPH SHERIF

“The analysis showed a significantly higher density of defects during requirements inspections. It was also observed, that the defect densities found decreased exponentially as the more products approached the coding phase.”

— John C. Kelly, Joseph S. Sherif, and Jonathan Hops. An Analysis of Defect Densities Found During Software Inspections. *Journal of Systems and Software*, 17(2):111–117, 1992. doi:[10.1016/0164-1212\(92\)90089-3](https://doi.org/10.1016/0164-1212(92)90089-3)



LIONEL C. BRIAND

“Five out of the six object-oriented metrics presented by Chidamber and Kemerer [1994] appear to be useful to predict class fault-proneness during the high- and low-level design phases of the life-cycle.”

— Victor R. Basili, Lionel C. Briand, and Walcélío L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996. doi:[10.1109/32.544352](https://doi.org/10.1109/32.544352)



NORMAN FENTON

“Our critical review of state-of-the-art of models for predicting software defects has shown that many methodological and theoretical mistakes have been made... We recommend holistic models for software defect prediction, using Bayesian Belief Networks, as alternative approaches to the single-issue models used at present.”

— Norman E. Fenton and Martin Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.  
doi:[10.1109/32.815326](https://doi.org/10.1109/32.815326)



TABLE 4  
DEFECTS DENSITY (F/KLOC) vs. MTTF

F/KLOC	MTTF
> 30	1 min
20–30	4-5 min
5–10	1 hr
2–5	several hours
1–2	24 hr
0.5–1	1 month

“This means we should be very wary of attempts to equate fault densities with failure rates, as proposed for example by Jones [1996]. Although highly attractive in principle, such a model does not stand up to empirical validation.”

Source: Norman E. Fenton and Martin Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. doi:[10.1109/32.815326](https://doi.org/10.1109/32.815326)

TABLE 1  
DEFECTS PER LIFE-CYCLE PHASE PREDICTION  
USING TESTING METRICS

Defect Origins	Defects per Function Point
Requirements	1.00
Design	1.25
Coding	1.75
Documentation	0.60
Bad fixes	0.40
Total	5.00

“We already see defect density defined in terms of defects per function point, and empirical studies are emerging that seem likely to be the basis for predictive models. For example, Jones [1991] reports the following bench-marking study, reportedly based on large amounts of data from different commercial sources.”

Source: Norman E. Fenton and Martin Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. doi:[10.1109/32.815326](https://doi.org/10.1109/32.815326)



STEVE MCCONNELL

“Industry average experience is about 1-25 errors per 1000 lines of code for delivered software. Cases that have one-tenth as many errors as this are rare; cases that have 10 times more tend not to be reported. (They probably aren’t ever completed!) Microsoft experiences about 10–20 defects per 1000 lines of code during in-house testing and 0.5 defects per 1000 lines of code in released product.”

— Steve McConnell. *Code Complete*. Pearson Education, 2004.  
doi:[10.5555/1096143](https://doi.org/10.5555/1096143)



4. Defect density depends on many factors and metrics.



PARASTOO MOHAGHEGHI

“The analysis showed that reused components have lower defect-density than non-reused ones. Reused components have more defects with highest severity than the total distribution, but less defects after delivery.”

— Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An Empirical Study of Software Reuse vs. Defect-Density and Stability. In *Proceedings of the 26th International Conference on Software Engineering*, pages 282–291. IEEE, 2004. doi:[10.1109/icse.2004.1317450](https://doi.org/10.1109/icse.2004.1317450)



NACHIAPPAN NAGAPPAN

“A case study performed on Windows Server 2003 indicates the validity of the relative code churn measures as early indicators of system defect density. Our code churn metric suite is able to discriminate between fault and not fault-prone binaries with an accuracy of 89%.”

— Nachiappan Nagappan and Thomas Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, 2005b. doi:[10.1145/1062455.1062514](https://doi.org/10.1145/1062455.1062514)



THOMAS BALL

“Our results show that the static analysis defect density is correlated at statistically significant levels to the pre-release defect density determined by various testing activities. Further, the static analysis defect density can be used to predict the pre-release defect density with a high degree of sensitivity.”

— Nachiappan Nagappan and Thomas Ball. Static Analysis Tools as Early Indicators of Pre-Release Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 580–586, 2005a.  
doi:[10.1145/1062455.1062558](https://doi.org/10.1145/1062455.1062558)



A GÜNEŞ KORU

“We studied four large-scale object-oriented products, Mozilla, Cn3d, JBoss, and Eclipse. We observed that defect proneness increased as class size increased, but at a slower rate; smaller classes were proportionally more problematic than larger classes.”

— A. Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2008. doi:[10.1109/tse.2008.90](https://doi.org/10.1109/tse.2008.90)





KAZUHIRO YAMASHITA

“Although we found some support for findings in recent literature that smaller files have higher defects density, we found further evidence that very large or complex files have lower defect densities and in some cases even lower defect proneness. Our findings have immediate practical implications: the redistribution of Java code into smaller and less complex files may be counterproductive.”

— Kazuhiro Yamashita, Changyun Huang, Meiyappan Nagappan, Yasutaka Kamei, Audris Mockus, Ahmed E. Hassan, and Naoyasu Ubayashi. Thresholds for Size and Complexity Metrics: A Case Study From the Perspective of Defect Density. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, pages 191–201. IEEE, 2016.  
doi:[10.1109/qrs.2016.31](https://doi.org/10.1109/qrs.2016.31)

## 100+ Metrics that Predict Faults

- |   |  |                                     |  |  |
|---|--|-------------------------------------|--|--|
| 1. <b>AHF</b> Attribute Hiding Factor         | 11. <b>DAM</b> Data Access Metric                | 20. <b>TCC</b> Tight class cohesion | 38. <b>OMMEC</b>                               | 51. <b>NOC</b> Number of children                            |
| 2. <b>AIF</b> Attribute Inheritance Factor    | 12. <b>DCC</b> Direct Class Coupling             | 21. <b>ACAIC</b>                    | 39. <b>OMMIC</b>                               | 52. <b>NTM</b> Number of trivial methods                     |
| 3. <b>COF</b> Coupling Factor                 | 13. <b>DSC</b> Design size in classes            | 22. <b>ACMIC</b>                    | 40. <b>ATTRIB</b> Attributes                   | 53. <b>RFC</b> Response for a class                          |
| 4. <b>MHF</b> Method Hiding Factor            | 14. <b>MFA</b> Measure of Functional Abstraction | 23. <b>AMMIC</b>                    | 41. <b>DELS</b> Deletes                        | 54. <b>WMC</b> Weighted methods per class                    |
| 5. <b>MIF</b> Method Interface Factor         | 15. <b>MOA</b> Measure of Aggregation            | 24. <b>Coh</b> A variation on LCOM5 | 42. <b>EVNT</b> Events                         | 55. <b>AMC</b> Average method complexity                     |
| 6. <b>POF</b> Polymorphism Factor             | 16. <b>NOH</b> Number of hierarchies             | 25. <b>DCAEC</b>                    | 43. <b>READS</b> Reads                         | 56. <b>Past</b> faults Number of past faults                 |
| 7. <b>SCC</b> Similarity-based Class Cohesion | 17. <b>NOM</b> Number of Methods                 | 26. <b>DCMEC</b>                    | 44. <b>RWD</b> Read/write/deletes              | 57. <b>Changes</b> Number of times a module has been changed |
| 8. <b>ANA</b> Average Number of Ancestors     | 18. <b>NOP</b> Number of polymorphic methods     | 27. <b>DMMEC</b>                    | 45. <b>STATES</b> States                       | 58. <b>Age</b> Age of a module                               |
| 9. <b>CAM</b> Cohesion Among Methods          | 19. <b>LCC</b> Loose class cohesion              | 28. <b>FCAEC</b>                    | 46. <b>WRITES</b> Writes                       | 59. <b>Changeset</b> Number of modules changed               |
| 10. <b>CIS</b> Class Interface Size           |  | 29. <b>FCMEC</b>                    | 47. <b>CBO</b> Coupling between object classes | 60. $N_1$ Total number of operators                          |
|   |  | 30. <b>FMMEC</b>                    | 48. <b>DIT</b> Depth of inheritance tree       |  |
|   |  | 31. <b>IFCAIC</b>                   | 49. <b>LCOM</b> Lack of cohesion in methods    |  |
|   |  | 32. <b>IFCMIC</b>                   | 50. <b>LCOM2</b> Lack of cohesion in methods   |  |
|   |  | 33. <b>IFMMIC</b>                   |  |  |
|   |  | 34. <b>OCAEC</b>                    |  |  |
|   |  | 35. <b>OCAIC</b>                    |  |  |
|   |  | 36. <b>OCMEC</b>                    |  |  |
|   |  | 37. <b>OCMIC</b>                    |  |  |

61. $N_2$ Total number of operands	70. <b>ICH</b> Information-flow-based cohesion	passing	86. <b>NMI</b> Number of methods inherited	94. <b>SIX</b> Specialization index
62. $g_1$ Number of unique operators	71. <b>ICP</b> Information-flow-based coupling	77. <b>NAC</b> Number of ancestor	87. <b>NMO</b> Number of methods overridden	95. <b>C3</b> Conceptual cohesion of Classes
63. $g_2$ Number of unique operands	72. <b>IH-ICP</b> Information-flow-based inheritance coupling	78. <b>NDC</b> Number of descendent	88. <b>NOA</b> Number of attributes	96. <b>McCabe</b> Cyclomatic Complexity
64. <b>AID</b> Average inheritance depth of a class	73. <b>NIH-ICP</b> Information-flow-based non-inheritance coupling	79. <b>NLM</b> Number of local methods	89. <b>NOAM</b> Number of added methods	97. <b>Delta</b> Code delta
65. <b>LCOM1</b> Lack of cohesion in methods	74. <b>CMC</b> Class method complexity	80. <b>DAC</b> Data abstraction coupling	90. <b>NOO</b> Number of operations	98. <b>Churn</b> Code churn
66. <b>LCOM5</b> Lack of cohesion in methods	75. <b>CTA</b> Coupling through abstract data type	81. <b>DAC1</b> Data abstraction coupling	91. <b>NOOM</b> Number of overridden methods	99. <b>Devs</b> Number of developers
67. <b>Co</b> Connectivity	76. <b>CTM</b> Coupling through message	82. <b>MPC</b> Message passing coupling	92. <b>NOP</b> Number of parents	100. <b>CLD</b> Class-to-leaf depth
68. <b>LCOM3</b> Lack of cohesion in methods		83. <b>NCM</b> Number of class methods	93. <b>NPAVG</b> Average number of parameters per method	101. <b>NOA</b> Number of ancestors
69. <b>LCOM4</b> Lack of cohesion in methods		84. <b>NIM</b> Number of instance methods		102. <b>NOD</b> Number of descendants
		85. <b>NMA</b> Number of methods added		103. <b>LOC</b> Lines of Code

Source: Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software Fault Prediction Metrics: A Systematic Literature Review. *Information and Software Technology*, 55(8):1397–1418, 2013.

doi:[10.1016/j.infsof.2013.02.009](https://doi.org/10.1016/j.infsof.2013.02.009)



XIAO YU

“The problem of predicting the precise number of defects via regression algorithms is far from being solved.”

— Xiao Yu, Jacky Keung, Yan Xiao, Shuo Feng, Fuyang Li, and Heng Dai.  
Predicting the Precise Number of Software Defects: Are We There yet?  
*Information and Software Technology*, 146(1):106847, 2022.  
doi:[10.1016/j.infsof.2022.106847](https://doi.org/10.1016/j.infsof.2022.106847)

Table 1 The literature overview of the studies for predicting the numbers of defect.			
Study	Corpus/Number	Regression algorithms <sup>a</sup>	Performance measures
Ostrand [18] 2005	ISS/12	Negative Binomial Regression (NBR)	PvB
Jones [19] 2006	ISS/5	Poisson Regression (PR), NBR, Zero-Inflated Negative Binomial Regression (ZINBR)	Alberg diagrams
Gao [20] 2007	ISS/1	PR, Zero-Inflated Poisson Regression (ZIPR), NBR, ZINBR, Hurdle Poisson Regression (HPR)	AAE, AKE
Albal [21] 2008	ISS/3	Genetic Programming (GP)	Pred(I), MMRE, Spearman
Yu [22] 2012	PROMISE/5	NBR	Accuracy, Precision, Recall
Wang [15] 2012	Bugzilla and Jira/6	BugStates	Absolute Error (AE), Mean Absolute Error (MAE)
Rathore [23] 2015	PROMISE/10	Neural Network Regression (NNR), Genetic Programming (GP)	ARE, Recall, Completeness
Rathore [24] 2015	PROMISE/10	GP	ARE, Recall, Completeness
Chen [25] 2015	PROMISE/26	Linear Regression (LR), Bayesian Ridge Regression (BRR), Support Vector Regression (SVR), Nearest Neighbors Regression (NNR), Decision Tree Regression (DTR), Gradient Boosting Regression (GBR)	Precision, RMSE
Rathore [16] 2016	PROMISE/18	DTR	AAE, ARE, Pred(I)
Rathore [27] 2016	Eclipse/3	(Bagging/Boosting/Random subspace/Rotation Forest/Stacking)+(LR/Multilayer Perceptron Regression (MPR)/DTR)	AAE, ARE
Rathore [18] 2017	Firefox/3	NBR, ZIPR, MPR, GP, DTR, LR	AAE, ARE, Pred(I), Completeness
Rathore [19] 2017	PROMISE/11	Linear Regression based Combination Rule (LRCR), Gradient Boosting based Combination Rule (GBCR), MPR, GP, LR, NBR, ZIPR	AAE, ARE, Pred(I), Completeness
Rathore [10] 2017	PROMISE and Eclipse/17	Error Rate based Weighted Average (ERWA) combination rule, Linear Regression based Weighted Average (LRWA) combination rule, Decision Tree Forest based (DTF) ensemble method, Gradient Boosting Regression (GBR) based ensemble method, LR, MPR, DTR, GP, NBR, ZIPR	AAE, ARE, Pred(I), Completeness
Yu [51] 2017	PROMISE/22	(SMOTER/RUS/AdaBoost.R2)+(DTR/BRR/LR), SmoteNDBoost, RuNDBoost	FPA, Kendall
Zhang [14] 2018	Firefox/7	Sample entropy-Support Vector Regression (SSVR), Auto-Regressive Integrated Moving Average (ARIMA) model, X12-ARIMA model, NNR	Magnitude of Relative Error (MRE), MMRE
Wu [12] 2018	PROMISE/31	BRR, DTR, GBR, LR, NNR, MPR, and SVR	FPA
Rathore [13] 2019	PROMISE and Eclipse/19	A dynamic selection algorithm (DynSelection), LR, MPR, DTR, GP, NBR, ZIPR	AAE, ARE, Pred(I), Precision, Recall, F-measure
Chen [34] 2019	PROMISE/24	(SMOTER/SMOTUNED/Adaboost.R2)+(DTR/BRR/LR)	FPA, Kendall
Huang [33] 2019	PROMISE/20	Multi-Project Regression (MPR), LR, NNR, SVR, DTR, BRR, GBR	AAE, ARE
Nevedin [36] 2019	PROMISE/15	Adaboost.R2+(Extra Tree Regression (ETR)/Random Forest Regression (RFR)/Extreme Gradient Boosting Regression (XGBoost)/GBR)	MAE, MRE
Qiao [17] 2020	PROMISE and ISS/2	Deep Learning Neural Network (DPNN), SVR, DTR, Fuzzy Support Vector Regression (FSVR), RFR	Mean Squared Error (MSE), R <sup>2</sup>
Bal [37] 2020	PROMISE/26	Weighted Regularization Extreme Learning Machine (WR-ELM), Weighted Extreme Learning Machine (WELM), ELM, SmoteR+(ELM/SVR/NNR)	AAE, ARE, Pred(I),
Tong [38] 2021	PROMISE/27	Subspace Hybrid Sampling Ensemble (SHSE), SmoteR, SmoteRDE, DynSelection, SmoteNDBoost, RuNDBoost	FPA, Kendall, RMSE

<sup>a</sup>(Bagging/Boosting/Random subspace/Rotation Forest/Stacking)+(LR/MPR/DTR) represents that the five ensemble learning methods (Bagging, Boosting, Random subspace, Rotation Forest, and Stacking) use LR, MPR, and DTR as the base learners. It is the same below.

Source: Xiao Yu, Jacky Keung, Yan Xiao, Shuo Feng, Fuyang Li, and Heng Dai. Predicting the Precise Number of Software Defects: Are We There yet? *Information and Software Technology*, 146(1):106847, 2022. doi:[10.1016/j.infsof.2022.106847](https://doi.org/10.1016/j.infsof.2022.106847)

“Software testers want to not only know which software modules should be inspected first, but also evaluate the reliability and maintenance effort of each module. Therefore, they can first employ the historical data to construct a Defect Number Prediction (DNP) model, then use the two trained models to predict the defective-proneness or the number of defects.”



5. On average, modern projects have 10-50 defects per KLoC.

A few open source projects (12 Feb 2025)

Project	Stack	KLoC	Issues	I/KLoC
<u>tensorflow</u>	C++	3,600	40,200	11.2
<u>flutter</u>	Dart	2,100	103,000	49.0
<u>rust</u>	Rust	2,100	56,500	26.9
<u>spark</u>	Java	1,400	53,750	38.4
<u>kafka</u>	Java	980	19,800	20.2
<u>angular</u>	TypeScript	800	28,700	35.9
<u>guava</u>	Java	346	3,600	10.4

All repositories are open source.



## My own statistics (2 Feb 2024)

Github Repository	Stack	KLoC	Issues	I/KLoC
<a href="#">objectionary/eo</a>	Java	49	2,837	57.9
<a href="#">yegor256/cactoos</a>	Java	34	1,707	50.2
<a href="#">yegor256/takes</a>	Java	27	1,227	45.4
<a href="#">zold-io/zold</a>	Ruby	12	810	67.5
<a href="#">yegor256/tacit</a>	CSS	1	227	227.0

All repositories are open source.

# Bibliography

- Victor R. Basili and Barry T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42–52, 1984. doi:[10.1145/69605.2085](https://doi.org/10.1145/69605.2085).
- Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996. doi:[10.1109/32.544352](https://doi.org/10.1109/32.544352).
- IEEE Standards Board. IEEE Std 982.2-1988: Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, 1989.
- Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. doi:[10.1109/32.295895](https://doi.org/10.1109/32.295895).
- Richard H. Cobb and Harlan D. Mills. Engineering Software Under Statistical Quality Control. *IEEE Software*, 7(6): 45–54, 1990. doi:[10.1109/52.60601](https://doi.org/10.1109/52.60601).
- Michael Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 38(3):258–287, 1999. doi:[10.1147/sj.382.0258](https://doi.org/10.1147/sj.382.0258).
- Norman E. Fenton and Martin Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. doi:[10.1109/32.815326](https://doi.org/10.1109/32.815326).
- Capers Jones. *Applied Software Measurement*. McGraw-Hill, 1991. doi:[10.5555/109758](https://doi.org/10.5555/109758).
- Capers Jones. The Pragmatics of Software Process Improvements, 1996.
- John C. Kelly, Joseph S. Sherif, and Jonathan Hops. An Analysis of Defect Densities Found During Software Inspections. *Journal of Systems and Software*, 17(2): 111–117, 1992. doi:[10.1016/0164-1212\(92\)90089-3](https://doi.org/10.1016/0164-1212(92)90089-3).
- A. Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2008. doi:[10.1109/tse.2008.90](https://doi.org/10.1109/tse.2008.90).
- Steve McConnell. *Code Complete*. Pearson Education, 2004. doi:[10.5555/1096143](https://doi.org/10.5555/1096143).
- Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An Empirical Study of Software Reuse vs. Defect-Density and Stability. In *Proceedings of the 26th International Conference on Software Engineering*, pages 282–291. IEEE, 2004. doi:[10.1109/icse.2004.1317450](https://doi.org/10.1109/icse.2004.1317450).
- Nachiappan Nagappan and Thomas Ball. Static Analysis Tools as Early Indicators of Pre-Release Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 580–586, 2005a. doi:[10.1145/1062455.1062558](https://doi.org/10.1145/1062455.1062558).
- Nachiappan Nagappan and Thomas Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, 2005b. doi:[10.1145/1062455.1062514](https://doi.org/10.1145/1062455.1062514).
- Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software Fault Prediction Metrics: A Systematic Literature Review. *Information and Software Technology*, 55(8):1397–1418, 2013. doi:[10.1016/j.infsof.2013.02.009](https://doi.org/10.1016/j.infsof.2013.02.009).
- Kazuhiro Yamashita, Changyun Huang, Meiyappan Nagappan, Yasutaka Kamei, Audris Mockus, Ahmed E. Hassan, and Naoyasu Ubayashi. Thresholds for Size and Complexity Metrics: A Case Study From the Perspective of Defect Density. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, pages 191–201. IEEE, 2016. doi:[10.1109/qrs.2016.31](https://doi.org/10.1109/qrs.2016.31).
- Xiao Yu, Jacky Keung, Yan Xiao, Shuo Feng, Fuyang Li, and Heng Dai. Predicting the Precise Number of Software Defects: Are We There yet? *Information and Software Technology*, 146(1):106847, 2022. doi:[10.1016/j.infsof.2022.106847](https://doi.org/10.1016/j.infsof.2022.106847).