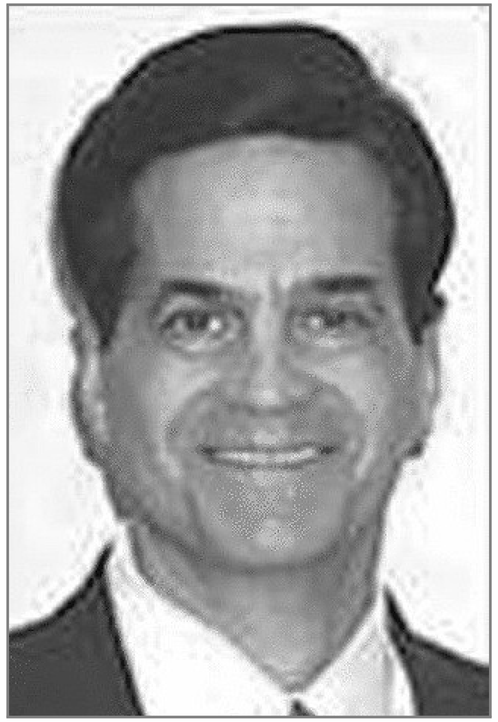
Defects Density

Yegor Bugayenko

Lecture #18 out of 24
80 minutes



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

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

Module name	*Number of errors*	*Lines of code*	*Error density, Errors/K. Loc*
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		

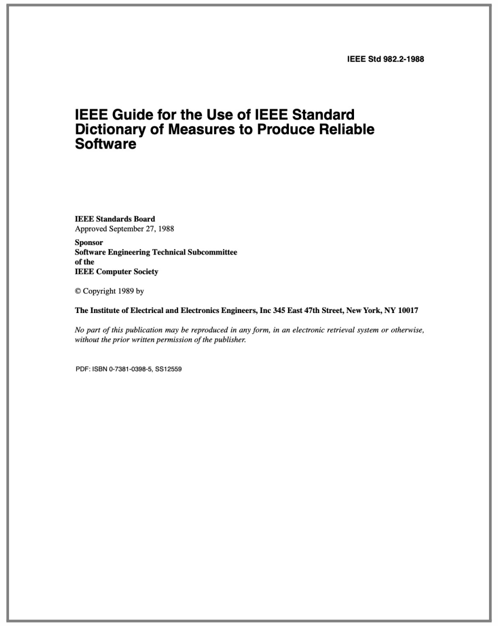
Source: Michael Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 38(3):258–287, 1999. doi:10.1147/sj.382.0258

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



IEEE Std 982.2-1988

IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software

IEEE Standards Board
Approved September 27, 1988

Sponsor
Software Engineering Technical Subcommittee
of the
IEEE Computer Society

© Copyright 1989 by

The Institute of Electrical and Electronics Engineers, Inc 345 East 47th Street, New York, NY 10017

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

PDF: ISBN 0-7381-0398-5, SS12559

“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

$$I = 7$$
$$KSLOD = 8$$

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."

Table 4.1-1 — Measure Classification Matrix

	Product Measures						Process Measures		
Measures (Experience)	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			
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 *K* 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						

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

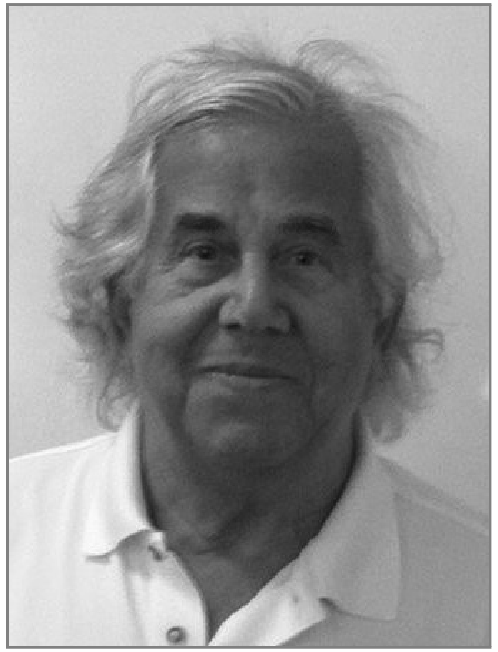
1. Fault Density
2. Defect Density
3. Cumulative Failure Profile
4. Fault-Days Number
5. Functional or Modular Test Coverage
6. Cause and Effect Graphing
7. Requirements Traceability
8. Defect Indices
9. Error Distribution(s)
10. Software Maturity Index
11. Manhours per Major Defect Detected
12. Number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

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



Joseph Sherif

“The analysis showed a significantly higher density of defects during requirements inspections. It was also observed, that the defect densities found decreased exponentialy as the mork 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

Victor R. Basili

"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 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

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

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."

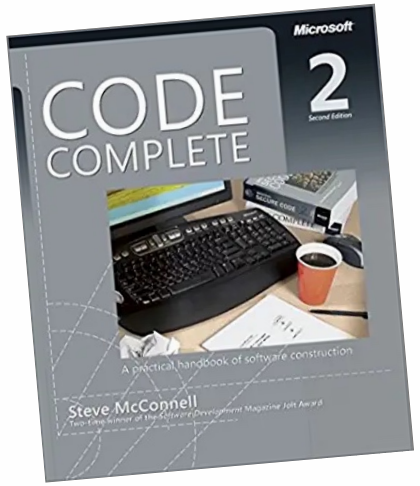
Source: Norman E. Fenton and Martin Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. doi:10.1109/32.815326

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



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

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

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

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

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

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."

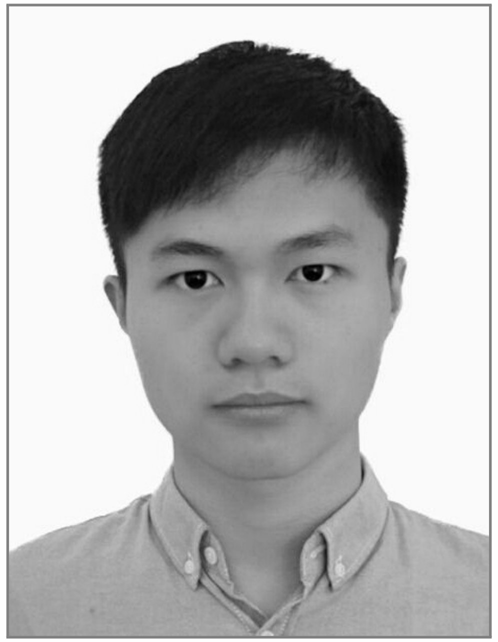
— Kazuhiro Yamashita, Changyun Huang, Meiyappan Nagappan, Yasutaka Kamei, Audris Mockus, Ahmed E. Hassan, and Naoyasu Ubayashi. Thresholds for Size and Complexity Metrics: A Case Study From the Perspective of Defect Density. In *Proceedings of the International Conference on Software Quality, Reliability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

100+ Metrics that Predict Faults

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

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



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*, 146:106847, 2022. doi: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[a]	Performance measures
Ostrand [18] 2005	ISS/12	Negative Binomial Regression (NBR)	PofB
Janes [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, ARE
Afzal [21] 2008	ISS/3	Genetic Programming (GP)	Pred(*l*), 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 [26] 2016	PROMISE/18	DTR	AAE, ARE, Pred(*l*)
Rathore [27] 2016	Eclipse/3	(Bagging/Boosting/Random subspace/Rotation Forest/Stacking)+(LR/Multilayer Perceptron Regression (MPR)/DTR)	AAE, ARE
Rathore [28] 2017	Firefox/3	NBR, ZIPR, MPR, GP, DTR, LR	AAE, ARE, Pred(*l*), Completeness
Rathore [29] 2017	PROMISE/11	Linear Regression based Combination Rule (LRCR), Gradient Boosting based Combination Rule (GRCR), MPR, GP, LR, NBR, ZIPR	AAE, ARE, Pred(l), Completeness
Rathore [30] 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(*l*), Completeness
Yu [31] 2017	PROMISE/22	(SMOTER/RUS/AdaBoost.R2)+(DTR/BRR/LR), SmoteNDBoost, RusNDBoost	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 [32] 2018	PROMISE/31	BRR, DTR, GBR, LR, NNR, MPR, and SVR	FPA
Rathore [33] 2019	PROMISE and Eclipse/19	A dynamic selection algorithm (DynSelection), LR, MPR, DTR, GP, NBR, ZIPR	AAE, ARE, Pred(*l*), Precision, Recall, F-measure
Chen [34] 2019	PROMISE/24	(SMOTER/SMOTUNED/AdaBoost.R2)+(DTR/BRR/LR)	FPA, Kendall
Huang [35] 2019	PROMISE/30	Multi-Project Regression (MPR), LR, NNR, SVR, DTR, BRR, GBR	AAE, ARE
Nevendra [36] 2019	PROMISE/15	AdaBoost.R2+(Extra Tree Regression (ETR)/Random Forest Regression (RFR)/Extreme Gradient Boosting Regression (EGBR)/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^2
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(*l*),
Tong [38] 2021	PROMISE/27	Subspace Hybrid Sampling Ensemble (SHSE), SmoteR, SmoteRDE, DynSelection, SmoteNDBoost, RusNDBoost	FPA, Kendall, RMSE

[a] (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*, 146:106847, 2022. doi: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."

My Own Statistics (2 Feb 2024)

Github Repository	Stack	KLoC	Issues	I/KLoC
zerocracy/farm	Java	58	2343	40.4
objectionary/eo	Java	49	2837	57.9
yegor256/cactoos	Java	34	1707	50.2
yegor256/takes	Java	27	1227	45.4
zold-io/zold	Ruby	12	810	67.5
yegor256/tacit	CSS	1	227	227.0

All repositories are open source.

References

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.

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.

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.

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.

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.

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.

Capers Jones. *Applied Software Measurement*. McGraw-Hill, 1991. doi:10.5555/109758.

Capers Jones. The Pragmatics of Software Process Improvements. *Technical Council on Software Engineering*, 14(2), 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.

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.

Steve McConnell. *Code Complete*. Pearson Education, 2004. doi: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.

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.

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.

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.

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.

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*, 146:106847, 2022. doi:10.1016/j.infsof.2022.106847.