Journal of Software Engineering and Applications, 2023, 16, 144-153
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Software Metric Analysis of Open-Source Business Software

Charles W. Butler

Department of Computer Information Systems, Colorado State University, Fort Collins, USA
Email: charles.butler@colostate.edu

How to cite this paper: Butler, C.W. (2023) Software Metric Analysis of Open-Source Business Software. *Journal of Software Engineering and Applications*, **16**, 144-153.
https://doi.org/10.4236/jsea.2023.165008

Received: April 23, 2023
Accepted: May 28, 2023
Published: May 31, 2023

Abstract

Over the past decade, open-source software use has grown. Today, many companies including Google, Microsoft, Meta, RedHat, MongoDB, and Apache are major participants of open-source contributions. With the increased use of open-source software or integration of open-source software into custom-developed software, the quality of this software component increases in importance. This study examined a sample of open-source applications from GitHub. Static software analytics were conducted, and each application was classified for its risk level. In the analyzed applications, it was found that 90% of the applications were classified as low risk or moderate low risk indicating a high level of quality for open-source applications.

Keywords

Open-Source Software, Software Quality, Software Risks, Cyclomatic Complexity, Essential Complexity, Module Design Complexity, Design Complexity, Integration Complexity, Local Data Complexity, Public Global Data Complexity, Parameter Data Complexity, Risk Score, Risk Classification

1. Introduction and Objective

If software is open-source, its source code is freely available to its users. Its users have the ability to take this source code, modify it, and distribute their own versions of the program. The users also have the ability to distribute as many copies of the original program as they want. Anyone can use the software for any purpose. There are no licensing fees or other restrictions on the software. The average percentage of open-source software in codebases of proprietary applications has grown from 36% to 57% [1]. A large number of applications now contain more open-source code than proprietary code [2]. Since open-source is growing in popularity, the quality characteristics of the software are critical to companies.

According to a security firm, Synopsys, at least 84% of codebases have at least one open-source vulnerability, and 48% have high-risk vulnerabilities, which have known exploits or are classified as allowing remote code execution [3].

There are a number of key challenges of open-source software including controlling long-term maintenance, managing evolution costs, and ensuring acceptable levels of quality. This research was an exploratory study of open-source software quality. The study objective sought to answer the question, "How risky is open-source software?" The use of open-source software does not pose risks that are fundamentally different from the risks presented by the use of proprietary or self-developed software. However, the acquisition and use of open-source software necessitates implementation of unique risk management practices.

2. Methodology

In order to study open-source software, forty open-source applications (See **Table 5**) were downloaded from GitHub. The business nature of these applications is enterprise resource planning, accounting, scheduling, and gaming. The sample contained applications written in Java, C++, and C. The forty applications contain over four million lines of code. The applications were parsed, and static analysis using McCabe IQ was conducted. McCabe software security, quality, testing, release, and configuration management solutions and McCabe IQ have been used to analyze the security, quality, and testing of business-critical software [4]. For each application:

1) A profile was built with software metrics for the application, risk component, outlier component, and extreme complex component.

2) Software analytics were applied to generate descriptive statistical modeling.

3) A risk scorecard using supervised induction was used for data mining to identify a risk classification for each application.

Table 1 contains the family of McCabe software metrics that were used to conduct risk analysis. McCabe's cyclomatic complexity, v(g), is a traditional software metric associated with software quality, risk, and testability. For the risk component, a software module with a $v(g) > 10$ is considered too complex and unreliable [5]. Ev(g) is another traditional software metric associated with software quality. It measures the structuredness of software code which is how the module's logic conforms to single entry, single exit constructs of structure programming. In McCabe IQ, an $ev(g) > 3$ is considered too complex and less maintainable [5]. **Table 1** also describes two additional sets of McCabe metrics. There are three design metrics, S_0, S_1, and iv(g). Design complexity, S_0, measures the size or magnitude of a design. A larger design is considered complex and implies high levels of integration in the design. Those integration levels are measured by integration complexity, S_1 and module design complexity, iv(g). Module design complexity measures low-level integration which is the integration test requirement between a superordinate module and its called immediate subordinates. Integration complexity measures the test requirement of a

Table 1. McCabe software metrics.

Metric	Symbol	Description	Calculation	Value Range
# of methods (modules)	n	The total number of methods in the application	Count of methods in the application	$1 \le n \le \infty$
Design complexity	S_0	The size or volume of the application design	$S_0 = \Sigma iv$	$1 \le S_0 \le \infty$
Integration complexity	S_1	The size of the high level integration basis set of subtrees	$S_1 = S_0 - n + 1$	$1 \le S_1 \le \infty$
Cyclomatic complexity	v or vg	The number of decision predicates in the module; the size of a basis set of paths for unit level testing	# design predicates + 1; $v = e - n + 2$ where e is the # of edges and n is the number of nodes in a flowgraph	$1 \le v \le \infty$ $v > 10$ is considered risky; higher v is riskier
Essential complexity	ev or evg	How well structured is the code; how easily is the code modularized (decomposed); how easily is the code maintained	The v of a reduced flowgraph where only single-entry, single-exit constructs are logically eliminated	$1 \le ev \le v$; $ev = 1$ is considered good, higher ev is riskier
Module design complexity	iv or ivg	The number of decision predicates (plus 1) that significantly impact calls to subroutines; the size of a basis set of paths for low level integration testing; iv risk is based on where the module is located; a management module should have a high iv, higher iv is riskier	The v of a reduced flowgraph where design predicates that do not significantly impact calls to subroutine are logically eliminated	$1 \le iv \le v$
Local data complexity	ldv ($sdv_{\text{local data}}$)	The number of decision predicates (plus 1) that significantly impact the use of local data; the size of a basis set of paths for local data testing	The v of a reduced flowgraph where design predicates that do not significantly impact the use of local data	$0 \le ldv \le v$; lower ldv is riskier
Public global data complexity	pgdv ($sdv_{\text{global data}}$)	The number of decision predicates (plus 1) that significantly impact the use of public global data; the size of a basis set of paths for public data testing	The v of a reduced flowgraph where design predicates that do not significantly impact the use of public global data	$0 \le pgdv \le v$; higher pgdv is riskier
Parameter data complexity	pdv ($sdv_{\text{parameter data}}$)	The number of decision predicates (plus 1) that significantly impact the use of parameter data; the size of a basis set of paths for parameter data testing	The v of a reduced flowgraph where design predicates that do not significantly impact the use of parameter data	$0 \le pdv \le v$; lower pdv is riskier

superordinate module and its multi-level, subtree subordinates. When integration testing requirements increase, the risk associated with the design grows [6].

Table 1 also includes an extended set of McCabe data metrics—local, public global, and parameter data complexities. These software metrics represent extensions of cyclomatic complexity into a module's data component. Local data complexity, ldv(g), measures the use of local data. High use of local data is a positive design concept and indicates less risk for a module since the data is not shared with other modules. Public global data complexity, pgdv(g), measures the use of global data. In contrast to local data, high use of public global data is a negative design conceptsince the data is shared among numerous other modules. Parameter data complexity, pdv(g), is the third data software metric. While parameter data complexity is a type of global data use, it is less risky due to the explicit nature of its use. Parameter data is explicitly stated in a call between a superordinate module and its subordinate modules. This approach is less risky since it is known which subordinate modules use the parameter data. In general, as the magnitude of McCabe metrics increases, the quality of the software module decreases, and the risk associated with the software module increases. The exceptions to this pattern are ldv(g) and pdv(g). As the use of data is restricted to a module or its use is explicitly stated, the quality of the software module increases, and the risk associated with the software module decreases.

For the outlier component, the study utilized Deming's statistical quality control concepts. In addition to McCabe's criteria, $v > 10$, the outlier component utilized 3σ to determine outliers in the sample. These modules would be subject to Deming's plan-do-study-act [7]. In order to reduce the risk of complex modules, a software engineering plans to find high risk modules (plan), identifies them (do), studies their logic (study), and refactors (act) them to reduce the decision logic complexity and risk. The basis for the extreme complex component is McCabe's categorization of cyclomatic complexity to the Department of Homeland Security [8]:

- 1 - 10: simple procedure, little risk
- 11 - 20: more complex, moderate risk
- 21 - 50: complex, high risk
- >50: untestable code, very high risk

3. Findings

The first task in this study was to parse and conduct static metric analysis using McCabe IQ. Forty applications were downloaded from GitHub for this task. For each application, a profile was built with software metrics for the total application and the risk ($v > 10$), outlier ($v > 3\sigma$), and extreme ($v > 50$) components. Table 2 contains a profile for an application named, Git. Git is written in the C language and is 193,694 lines of executable code. It is a business scheduling application.

Review the software metrics for Git in Table 2. The column labeled Profile

Table 2. Git profile.

Application Profile Git (C)								
Item #	Static Metrics	Profile	Risk (v > 10)	Risk %	Outlier [v > 14]	Outlier %	Extreme v > 50	Extreme v > 50%
	Number of modules	7108	639	9%	123	1.7%	43	0.60%
	Design Complexity, S_0	30,624	11,006	36%	4235	13.8%	2201	7.19%
	Integration Complexity, S_1	23,517	10,368	44%	4113	17.5%	2159	9.18%
	Cyclomatic complexity (v):							
	Total cyclomatic complexity	42,345	16,349	39%	6345	15%	3194	7.54%
1	Average cyclomatic complexity	2.2	25.6	1138%	51.6	2295%	74.3	3305%
	Standard deviation—cyclomatic complexity	4.0	16.5	414%	21.9	551%	23.0	578%
	Maximum—cyclomatic complexity	159	159		159		159	
	Essential complexity (ev):							
	Total essential complexity	22,396	8278	37%	3000	13%	1471	6.57%
	Average essential complexity	3.2	13.0	411%	24.4	774%	34.2	1086%
2	Standard deviation—essential complexity	4.9	10.5	215%	16.4	337%	20.7	426%
	Maximum—essential complexy	134	134		134		134	
	Maintenance coefficient (ev/v)	0.53	0.51		0.47		0.46	
	Module design complexity (iv):							
	Total module design complexity	30,624	11,006	36%	4235	14%	2201	7.19%
	Average module design complexity	4.3	17.2	400%	34.4	799%	51.2	1188%
3	Standard deviation—module design complexity	6.0	12.9	214%	19.9	331%	24.2	402%
	Maximum—module design complexity	142	142		142		142	
	Management coefficient (iv/v)	0.72	0.67		0.67		0.69	
	Local data complexity (ldv):							
	Total local data complexity	28,827	6921	24%	2209	8%	989	3%
	Average local data complexity	1.2	10.8	915%	18.0	1517%	23.0	1942%
4	Standard deviation—local data complexity	3.5	12.7	359%	21.3	601%	25.1	707%
	Maximum—local data complexity	139	121		121		94	
	ldv density (ldv/v)	0.68	0.42		0.35		0.31	
	Global data complexity (pgdv):							
	Total global data complexity	8281	1976	24%	644	8%	285	3%
	Average global data complexity	1.2	3.1	265%	5.2	449%	6.6	569%
5	Standard deviation—global data complexity	3.0	6.1	207%	9.2	311%	10.1	341%
	Maximum—global data complexity	52	52		52		52	
	pgdv density (pgdv/v)	0.20	0.12		0.10		0.09	

Continued

	Parameter data complexity (pdv):							
	Total parameter data complexity	25,192	4413	18%	1137	5%	612	2.43%
	Average parameter data complexity	3.5	6.9	195%	9.2	261%	14.2	402%
6	Standard deviation—parameter data complexity	4.7	7.9	170%	13.0	279%	18.6	399%
	Maximum—parameter data complexity	113	63		63		63	
	pdv density (pdv/v)	0.59	0.27		0.18		0.19	

(accented in green) represents the total application; the column labeled Risk (accented in orange) represents the portion of modules that are high risk with *v*(*g*) > 10; the column labeled Outlier (accented in blue) represents the portion of modules that are considered statistical outliers with an *average ev*(*g*) more than 3σ above the *average v*(*g*) for the application. There are 639 modules, or 9% of the application, in the risk profile (*v*(*g*) > 10). There are 123 modules in the outlier profile, or 1.7% of the application. An examination of other McCabe metrics reveals how the risk of the Git application is assessed. The *average v*(*g*) for the application is 2.2 which is below the threshold hold of poor-quality software, *v*(*g*) > 10. However, when examining the risk and outlier profiles, the *average v*(*g*) increases to 25.6 and 51.6, respectively. These components of Git exhibit low quality and high risk for testability. When examining the *ev*(*g*) measurements, Git's application profile shows a low magnitude of 3.2. When examining the risk and outlier profiles, the *average ev*(*g*) increases to 13.0 and 24.4, respectively. These measurements of coding quality reveal low quality and high risk. A similar pattern for the remaining McCabe is observed. Module design, local data, public global data, and parameter data significantly increase in magnitude (low quality) in the risk and outlier profiles.

Regarding this pattern, the question to be explored is "How can this pattern be used to represent quality and a risk score for an application?" In order to do so, an applied classification algorithm or supervised induction was used for data mining to build a risk scorecard for each application's risk component (v > 10). Table 3 contains risk classification criteria applied to the sample applications. Risk classification utilizes a business analytics descriptive approach. The intent is to know what is happening with an application and understand underlying trends and causes of quality level and risks.

Quartiles and interquartile range help identify spread with a subset of data. The quartile is a quarter of the number of data points given with a data set. Quartiles are determined by first scoring the data and then splitting the sorted data into four disjoint smaller data sets. In Table 3, risk classification criteria are assigned for quartiles, representing low risk, moderate low risk, moderate high risk and high-risk classifications. The first attribute used for risk classification is $\%n_{risk}$. This attribute is the percentage of the application that falls into the risk range (v > 10). The higher the percentage, the greater portion of the application falls within the risk domain, and the poorer the quality and higher the risk. The

Table 3. Risk classification criteria.

Metric	Quartile 1	Quartile 2	Quartile 3	Quartile 4
	Low Risk	Moderate Low Risk	Moderate High Risk	High Risk
$\%n_{risk}$	$\%n_{risk} \leq 2.5\%$	$2.5\% < \%n_{risk} \leq 5.0\%$	$5.0\% < \%n_{risk} \leq 7.5\%$	$\%n_{risk} > 7.5\%$
μ_v	$\mu_v \leq 10$	$10 < \mu_v \leq 20$	$20 < \mu_v \leq 30$	$\mu_v > 30$
$\%S_0$ $\%S_1$ ev density iv density (1-ldv) density pgdv density (1-pdv) density	0 < metric ≤ 25%	25% < metric ≤ 50%	50% < metric ≤ 75%	75% < metric ≤ 100%
Source code risk score	0 < risk score ≤ 17.5	17.5 < risk score ≤ 25.0	25.0 < risk score ≤ 32.5	32.5 < risk score ≤ 40

second attribute (the second row in the table) used for risk classification is μ_{vrisk}. Higher v(g) values are associated with poorer testability quality and higher risk. μ_{vrisk} is divided into equal groups of 10 units each with the last grouping being open ended. The third row in the table specifies how the remaining McCabe metrics map to quality and risk. Transformations of S_0, S_1, ev(g), iv(g), ldv(g), pgdv(g), and pdv(g) are defined so that equal quartile separation is achieved for groups between 0 and 100%. In general, the values for these attributes imply the following:

- $\%S_0$: When $\%S_0$ increases, quality decreases and risk increases.
- $\%S_1$: When $\%S_1$ increases, quality decreases and risk increases.
- ev density: When ev/v increases, quality decreases and risk increases.
- iv density: When iv/v increases, quality decreases and risk increases.
- ldv density: When ldv/v increases, quality increases and risk decreases.
- pgdv density: When pgdv/v increases, quality decreases and risk increases.
- pdv: When pdv/v increases, quality increases and risk decreases.

The final table row defines the risk scorecard quantity for risk classification using nine attributes. Table 4 illustrates the application of the risk classification algorithm applied to Git.

Note that in the table, density ldv and density pdv are subtracted from one. This transformation is applied so that the magnitude of these attributes aligns with the other attributes. By transforming density ldv and density pdv, the higher the value, the poor the quality and the higher the risk. When any of the attributes increase, the quality of the application declines. Also, note that avg v is weighed twice. Weighting v reinforces the importance of the published research significance of v > 10 being associated with low quality and testability of software.

Table 4. Risk classification example.

Git Risk Classification			
	Metric	Risk Value	Quartile
1	% #modules	9%	1
2	$\%S_0$	36%	2
3	$\%S_1$	44%	2
4	avg v (weighted double)	25.59	6
5	density ev	0.51	3
6	density iv	0.67	3
7	(1-density ldv)	0.58	3
8	density pgdv	0.12	1
9	(1-density pdv)	0.41	2
static risk score		23	
static risk classification		moderate low risk	

4. Analysis

The above risk classification algorithm was applied to the sample applications. Table 5 contains the results. Four of the applications are classified as moderate high risk. Twenty-eight applications are classified as moderate low risk, and eight applications are classified as low risk. No applications are classified as high risk. Of the eight applications classified as low risk, seven did not contain a "risk profile" which means there are no modules with a v > 10 in these applications. If an application has no "risk profile", it follows that it does not have "Outlier" and "v > 50" profiles. These applications are not included in Table 5.

Table 5. Open-Source application risk classification.

Application				Risk (v > 10)					
Application	Language	#Mod	Risk%	μ_v	$\Delta\mu_v$	μ_{ev}	μ_{iv}	Risk Score	Risk Classification
UK_Player	C	6	6%	46.8	858%	17.2	19.7	30	moderate high
Tmux	C	164	15%	23.5	523%	10.5	20.4	28	moderate high
Ofbiz	Java	943	7%	23.5	523%	10.5	20.4	26	moderate high
solitaire	C++	4	6%	22.0	573%	9.0	20.0	26	moderate high
Scorpio	Java	1344	5%	22.7	605%	10.1	19.5	25	moderate low
Blueseer	Java	666	10%	16.0	295%	4.9	12.6	25	moderate low
Adempiere	Java	1109	2%	26.5	1078%	9.1	17.3	24	moderate low
Idempiere	Java	941	2%	22.8	879%	10.4	19.5	24	moderate low
Schedulis	Java	118	3%	17.1	590%	11.0	13.6	24	moderate low
Nacos_Develop	Java	105	2%	15.9	654%	6.8	13.6	24	moderate low

Continued

Gaussian_YOLOv3	C	36	9%	18.5	249%	5.8	13.6	24	moderate low
Darknet	C	36	9%	18.3	245%	5.7	13.4	24	moderate low
phone_info	C++	4	1%	24.3	1005%	7.8	16.8	24	moderate low
sudokumaster	C++	2	2%	11.5	356%	9.0	6.0	24	moderate low
Wumpus	Java	6	7%	17.5	442%	6.2	10.5	24	moderate low
Git	C	639	9%	25.6	330%	13.0	17.2	23	moderate low
Dbeaver	Java	512	2%	17.4	729%	8.3	13.9	23	moderate low
Axelor	Java	171	3%	16.4	447%	6.9	14.0	23	moderate low
Mes	Java	63	1%	14.7	717%	7.4	13.1	23	moderate low
RedDragonERP	Java	15	0.3%	23.0	1555%	14.9	21.4	23	moderate low
Redisson	Java	41	0.5%	18.2	1155%	11.0	13.6	23	moderate low
OpenRefine	Java	195	4%	17.9	367%	11.5	14.3	23	moderate low
Portfolio	Java	171	2%	16.3	826%	6.9	14.0	22	moderate low
Scr_Cpy	C	3	2%	29.7	963%	17.7	23.0	21	moderate low
Eladmin	Java	3	1%	17.0	750%	8.0	16.7	21	moderate low
JMX_Exporter	Java	4	3%	22.8	832%	8.8	19.3	21	moderate low
Java.Battleship	Java	8	12%	14.5	334%	3.5	5.0	21	moderate low
Libevent	C	111	7%	17.5	357%	9.3	11.6	20	moderate low
Spring_All	Java	1	0.1%	11.0	787%	10.0	10.0	20	moderate low
space_blok	C++	3	1%	12.7	545%	5.7	4.7	20	moderate low
weekly_planner	C++	1	1%	11.0	450%	1.0	1.0	20	moderate low
Metafresh	Java	1033	1%	17.8	947%	8.0	14.6	19	moderate low
maps_samples	C++	9	1%	13.4	570%	5.6	7.4	12	low

5. Conclusions

This exploratory study sought to assess the quality of open-source software. Forty applications were downloaded, and static software metric analysis was conducted on each application. Using McCabe software metrics, a risk scorecard algorithm was applied to assign a risk classification to each software application. In summary, the risk classification distribution for the sample is as follows:

- High risk: 0%
- Moderate high risk: 10%
- Moderate low risk: 70%
- Low risk: 20%

In conclusion, the study showed that open-source software exhibits a high degree of quality as 90% of the applications are "moderate low risk or low risk" classified. The implication is that the use of open-source software in commercial business software is a compelling high-quality approach for commercial development and deployment of applications.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Gehman, C. (2019) How to Use Open Source Code in Proprietary Software. https://www.perforce.com/blog/vcs/using-open-source-code-in-proprietary-software

[2] Zorz, Z. (2018) The Percentage of Open-Source Code in Proprietary Apps Is Rising. *Slashdot*. https://news.slashdot.org/story/18/05/22/1727216/the-percentage-of-open-source-code-in-proprietary-apps-is-rising

[3] McKay, T. (2023) Open-Source Vulnerabilities Wide Spread in Codebases, Report Finds, IT Brew. https://www.itbrew.com/stories/2023/03/20/open-source-vulnerabilities-widespread-in-codebases-report-finds

[4] McCabe Software (2023) http://mccabe.com/

[5] McCabe, T.J. (1976) A Complexity Measure. *IEEE Transaction on Software Engineering*, **SE-2**, 308-320. https://doi.org/10.1109/TSE.1976.233837

[6] McCabe, T.J. and Butler, C.W. (1989) Design Complexity Measurement and Testing. *Communications of the ACM*, **32**, 1415-1425. https://doi.org/10.1145/76380.76382

[7] Henshall, A. (2020) How to Use the Deming Cycle for Continuous Quality Improvement. https://www.process.st/deming-cycle/

[8] Wikipedia (2023) Cyclomatic Complexity. https://en.wikipedia.org/wiki/Cyclomatic_complexity