Journal of Software Engineering and Applications, 2022, 15, 150-164
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Software Analytics 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. (2022) Software Analytics of Open Source Business Software. *Journal of Software Engineering and Applications*, **15**, 150-164.
https://doi.org/10.4236/jsea.2022.155008

Received: March 30, 2022
Accepted: May 28, 2022
Published: May 31, 2022

Abstract

This paper applies software analytics to open source code. Open-source software gives both individuals and businesses the flexibility to work with different parts of available code to modify it or incorporate it into their own project. The open source software market is growing. Major companies such as AWS, Facebook, Google, IBM, Microsoft, Netflix, SAP, Cisco, Intel, and Tesla have joined the open source software community. In this study, a sample of 40 open source applications was selected. Traditional McCabe software metrics including cyclomatic and essential complexities were examined. An analytical comparison of this set of metrics and derived metrics for high risk software was utilized as a basis for addressing risk management in the adoption and integration decisions of open source software. From this comparison, refinements were added, and contemporary concepts of design and data metrics derived from cyclomatic complexity were integrated into a classification scheme for software quality. It was found that 84% of the sample open source applications were classified as moderate low risk or low risk indicating that open source software exhibits low risk characteristics. The 40 open source applications were the base data for the model resulting in a technique which is applicable to any open source code regardless of functionality, language, or size.

Keywords

Cyclomatic Complexity, Essential Complexity, Module Design Complexity, Design Complexity, Integration Complexity, Local Data Complexity, Public Global Data Complexity, Parameter Data Complexity, Risk, Application Risk Score, Application Risk Classification

1. Introduction

Since 1983, software has undergone an evolution. The establishment of open source

code has reduced the importance of custom software development in building code bases, and progressive companies have realized that open source code is a valuable business strategy to maintain profitable operation and managed growth. Consequently, open source development has evolved to complement changing application portfolio growth, and application profiles have, as their by-product, hybrid applications composed of open source code.

Additionally, software development traits have been instrumental in the evolution of applications and the growth of open source code. Software developers have changed psychologically, sociologically, and economically. Software developers' interest, tastes, and attitudes have resulted in new development patterns. Software developers are less proprietary than they once were, and collaborating and sharing within the software development community grow more popular each year. To meet the emerging reliance and flexibility characteristics of today's global world, companies are growingly reliant on software developed outside the confines of their business walls.

1.1. The Problem

With these developments, the importance of open source software grows, and the question whether to adopt or integrate it is often challenging to answer. Managers and software developers try to justify a decision based on terms of reduced cost, expedited implementation time, or natural evolution of feature development within an open source development community. The objective of this study included three elements. The first was to assess the quality of open source software and to analyze the quality measures utilizing McCabe's family of software metrics. The second component was to develop a risk classification algorithm integrating multiple quality metrics. Finally, the last element was to apply the quality classification to open source code in order to aid managers and software developers when making adoption and integration decisions regarding open source code.

1.2. The Research Methodology

Business analytics (BA) continues its growth in the business profession. It is the methods, techniques, and data that are used by an organization to measure performance and develop actionable decisions [1]. Business analytics are made up of statistical methods that can be applied to a specific project, process, or product. Within the business analytics discipline, the domains are the study of three important areas: descriptive analysis, predictive analysis, and prescriptive analysis. Software analytics is a special domain within business analytics. In a special issue of *IEEE Software* in 2013, the domain of software analysis was explored. To better frame software analytics, Menzies and Zimmerman defined software analytics as "analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions" [2]. The authors cited

"early global models and software analytics" including McCabe's cyclomatic complexity [2].

To accomplish the stated objectives, a sample of 40 open source applications was downloaded. Traditional McCabe software metrics including cyclomatic and essential complexities were examined. An analytical comparison of this set of metrics and derived metrics for high risk software was utilized as a basis for addressing risk management in the adoption and integration decisions of open source software. From this comparison, refinements were added, and contemporary concepts of design and data metrics derived from cyclomatic complexity were integrated into a classification scheme for software quality. The 40 open source applications were the base data for the model resulting in a technique which is applicable to any open source code regardless of functionality, language, or size. Finally, the validity of the classification algorithm was illustrated by applying it to the open source code sample. The classification scheme was evaluated to examine if the risk classification provides information pertaining to the software quality.

1.3. Organization of the Paper

The previous section was a general background statement to present the topic area to the reader, and it included the objective and the methodology of the research. In the next section, there is a review of open source software growth. Then, the primary research is presented. The sample open source applications are analyzed, and the classification scheme is defined before it is applied to the sample applications. Last, the final section contains suggestions of further research and conclusions.

2. Open Source Software

In 1983, Richard Stallman, a programmer from MIT, came up with the idea of "free software movement." The main motive behind this thought was to provide freedom to the programmers to study and modify source code according to their needs. The open source movement is one that supports the use of open source licenses for some or all the software. Today, businesses, managers, and software developers who support the open source software philosophy contribute to the open source community by voluntarily writing and exchanging programming code for software development. The term "open source" requires that no one can discriminate against any group or individual in obtaining or editing the code of an open source application. This approach to software development allows anyone to obtain and modify the open source code. These modifications are distributed back to the developers within the open source community of people who are working with the software. Now, numerous businesses offer codebases with open source that span business functions including enterprise software, retail and e-commerce, financial services, mobile apps, and marketing [3].

Open source software is used everywhere. Through its use, companies are

likely to achieve the benefits of open source software. Some of the key benefits of open source software and how enterprise companies can help to realize those benefits are as follows [4] [5] [6] [7] [8]:

- Reduces costs
  - Total cost of ownership (TCO)
  - Development costs
  - Support costs
  - Per user costs
  - License fees
- Shortens negotiations
- Implements interoperability using open standards and APIs
- Embraces suitability and security
- Addresses a company's needs
- Avoids vendor lock-in
- Supports rapid prototyping
- Facilitates redistribution to other organizations
- Promotes flexibility and stability
- Embraces security and reliability
- Gains community support
- Reduces development time and expense
- Generates reliable code
- Yields maintainable code
- Accesses third-party improvements

Free and open source software (FOSS) has found its way into most companies. Companies look to open source software to promote growth, achieve a level of flexibility, and attain a high level of reliability. Its use spans different sectors of IT services including [9]:

- Server software products such as Apache and Samba 4
- Development using PHP, Rails, and Perl
- Security including Linux core functionality
- Workflow using Pentaho, Collabtive, and SugarCRM
- Collaboration through the Cyn.in community edition, Zimbra Open Source Edition, and Kolab
- Big data using SUSE
- Cloud computing with an ownCloud platform
- Multimedia products such as Audacity and OpenShot
- e-commerce via PrestaShop

Increasingly, modern software development projects rely on open source software including a heavy reliance on products for building enterprise software. This trend results in vulnerability to the failure of open source products and responsibility to support these products and their ecosystems [10]. Open source software is an important tool for helping business develop software rapidly and effectively.

3. Mccabe Metrics

Tom J. McCabe authored and published cyclomatic, essential, and actual complexities in in *IEEE Transactions on Software Engineering*. In 2009, his article was chosen as one of the retrospective 23 highest impact papers in computer science by ACM-SIGSOFT. In the software engineering discipline, cyclomatic complexity, v(g), is recognized as a software quality attribute. It is also associated with McCabe Structured Testing Methodology. A v(g) may range from 1 to + ∞. For unit level testing using McCabe Structured Testing Methodology, a basis set of paths equal to a testable module's v(g) is the minimum unit level of testing [11]. Research has shown that when v(g) exceeds 10, the reliability of the testable unit decreases exponentially. Essential complexity, ev(g), is a measurement of a software module's coding structure. When the coding structure violates structured programing constructs, ev(g) grows from 1 to v(g) depending on how many decision predicates violate the single entry, single exit property of structured programming. In the McCabe automated testing tool, McCabe IQ, a threshold of ev(g) ≤ 3 is set [12]. A higher ev(g) indicates that the maintainability of the software decreased [13].

Since the introduction of cyclomatic and essential complexities, the McCabe family of metrics was expanded. **Table 1** contains a summary of the entire McCabe metric set and each metrics quality implications [14]. McCabe design metrics focus on software integration. Module design, design, and integration complexities measure the low level, design volume, and end-to-end integration requirements using McCabe Structured Testing Methodology. When software requires higher levels of integration effort, it is considered more difficult to qualify and risker [15].

The McCabe family of metrics has a set of data metrics, ldv(g), pgdv(g), and pdv(g). When considering design qualities such as encapsulation, separation of concerns, and data hiding, data plays a critical role. When software is positively encapsulated, exhibits positive data hiding, and behaves with positive separation of concerns, it uses high levels of local data and low levels of public data. While parameter data extends data use into other software units, it is more positive than global data because its use is explicitly coded and limited by sharing between specific software units. For quality inferences, the following summarize McCabe metrics:

- Cyclomatic complexity: an increase in v(g) > 10 is negative and software units with higher v(g) are considered risker.
- Essential complexity: an increase of ev(g) > 3 is negative and software units with higher v(g) are considered risker.
- Design complexity: an increase in S_0 is problematic and larger software solutions are considered risker.
- Integration complexity: an increase in S_1 is problematic and software with higher end-to-end integration requirements is considered riskier.
- Module design complexity: an increase in iv(g) is problematic and software with higher low-level integration requirements is considered riskier.

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	$S_0 = \sum iv$	$1 \le S_0 \le +\infty$
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, v(g)	The number of decision predicatesin 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, ev(g)	How well structured is the code; how easily is the code modularized (decomposed); how easily is the codemaintained	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 optimal, McCabe IQ defaults to ev ≤ 3; higher ev is riskier
Module design complexity	iv, iv(g)	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, ldv(g), ($sdv_{local\ data}$)	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, pgdv(g), ($sdv_{global\ data}$)	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, pdv(g), ($sdv_{parameter\ data}$)	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

- Local data complexity: an increase in ldv(g) is positive and software with higher local data use is considered less risky because it exhibits better encapsulation, better separation of concerns, and better data hiding
- Public global data complexity: an increase in pgdv(g) is negative and software with higher pgdv is considered riskier because it exhibits poorer encapsulated, poorer separate of concerns, and poorer data hiding.
- Parameter data complexity: an increase in pdv(g) is positive and software with higher parameter data complexity is considered less risky because it exhibits better encapsulation, better separation of concerns, and better data

hiding.

4. Findings

Table 2 contains a summary of selected McCabe metrics for the open source sample used in this study. Forty applications were parsed using McCabe IQ. For each application, the lines of code (LOC) were tabulated. Design complexity, the number of modules (n), μ_v, u_{ev}, and μ_{iv} were also calculated for each application. This data is in the "Application" section of the table. The "Risk (v > 10)" section of the table contains the same selected McCabe metrics shown in the "Application" section but only for modules whose v > 10. To examine the magnitude of the Δv, the percentage change of risk μ_v from application μ_v was calculated.

4.1. Application Metrics

The forty applications account for over 4 million lines of code written in Java, C++, and C. There are almost 360,000 modules in the sample code. The application μ_v, u_{ev}, and μ_{iv} range from 1.1 to 6.0, 1.0 to 3.2, and 1.1 to 4.3, respectively. The weighted grand means for μ_v, u_{ev}, and μ_{iv} are 2.3, 1.4, and 2.1, respectively. There is one metric, u_{ev} for Git, that falls in a negative range (ev > 3). μ_v ranges from 1.1 to 6.0, and μ_{ev} from 1.0 to 3.2. At the application level, the applications exhibit low risk as measured by McCabe metrics.

4.2. Risk Metrics

The high risk modules for the forty applications account for almost 8500 modules. The μ_v, u_{ev}, and μ_{iv} for risk modules (v > 10) range from 11.0 to 46.8, 1.0 to 17.7, and 1.1 to 23.0, respectively. The weighted grand means for risk module μ_v, u_{ev}, and μ_{iv} are 21.4, 9.3, and 16.9, respectively. Only one metric, u_{ev} for weekly_planner, is in a low risk range (ev ≤ 3). The grand mean of μ_v for risk modules increased by 824% from 2.3 to 21.4. The magnitude of the percentage increase in μ_v ranges from 243% to 1555% indicating a measurable shift in riskiness as μ_v climbs above 10. Further, as μ_v increases above 10, there is a corresponding increase in higher u_{ev}. As μ_v increases above 10, the modules' structuredness and maintainability grows riskier, as u_{ev} grew from 1.4 to 9.3. As a collective group, the sample applications' modules exhibit risk. The "level of risk" is not defined.

4.3. Comparison

An alternative view of application risk is gained by examining a cross section of applications by size. Measuring application size is an arbitrary process. For this study, size is determined by lines of code using the following groups:

- Extra small: 0 ≤ LOC ≤ 25,000
- Small: 25,000 < LOC ≤ 50,000
- Medium: 50,000 < LOC ≤ 75,000
- Large: 75,000 < LOC ≤ 100,000
- Extra large: 100,000 < LOC

Table 2. Open source application sample.

		Application						Risk (v > 10)				
Application	Language	LOC	S_0	#Mod	μ_v	μ_{ev}	μ_{iv}	#Mod	μ_v	$\Delta\mu_v$	μ_{ev}	μ_{iv}
Metafresh	Java	964,889	162,506	103,191	1.7	1.2	1.6	1033	17.8	947%	8.0	14.6
Adempiere	Java	736,289	141,417	68,742	2.3	1.4	2.1	1109	26.5	1078%	9.1	17.3
Idempiere	Java	530,663	108,732	50,829	2.3	1.4	2.1	941	22.8	879%	10.4	19.5
Scorpio	Java	328,453	70,285	24,754	3.2	1.8	2.8	1344	22.7	605%	10.1	19.5
Ofbiz	Java	223,764	44,417	13,319	3.8	2.0	3.3	943	23.5	523%	10.5	20.4
Blueseer	Java	196,509	22,848	6741	4.1	1.6	3.4	666	16.0	295%	4.9	12.6
Git	C	193,694	30,624	7108	6.0	3.2	4.3	639	25.6	330%	13.0	17.2
Dbeaver	Java	157,521	44,879	24,356	2.1	1.4	1.8	512	17.4	729%	8.3	13.9
Axelor	Java	106,385	16,351	6123	3.0	1.6	2.7	171	16.4	447%	6.9	14.0
Mes	Java	96,263	17,264	10,103	1.8	1.2	1.7	63	14.7	717%	7.4	13.1
Portfolio	Java	74,071	13,638	8228	1.8	1.2	1.7	171	16.3	826%	6.9	14.0
RedDragonERP	Java	64,403	7452	5602	1.4	1.1	1.3	15	23.0	1555%	14.9	21.4
Redisson	Java	61,448	11,152	8268	1.5	1.1	1.4	41	18.2	1155%	11.0	13.6
OpenRefine	Java	56,782	11,003	4922	3.8	2.0	2.9	195	17.9	367%	11.5	14.3
Schedulis	Java	45,735	9298	4189	2.5	1.4	2.2	118	17.1	590%	11.0	13.6
Libevent	C	42,943	4902	1675	3.8	2.0	2.9	111	17.5	357%	9.3	11.6
Nacos_Develop	Java	39,278	10,308	5395	2.1	1.3	1.9	105	15.9	654%	6.8	13.6
Tmux	C	30,595	4514	1078	3.8	2.0	3.3	164	23.5	523%	10.5	20.4
Gaussian_YOLOv3	C	11,995	1421	391	5.3	2.1	3.6	36	18.5	246%	5.8	13.6
Darknet	C	11,839	1405	385	5.3	2.1	3.7	36	18.3	243%	5.7	13.4
maps_samples	C++	8959	1193	768	2.0	1.1	1.6	9	14.3	601%	5.6	7.4
Spring _All	Java	4135	852	707	1.2	1.0	1.2	1	11.0	787%	10.0	10.0
phone_info	C++	3243	546	313	2.2	1.2	1.7	4	24.3	1005%	7.8	16.8
UK_Player	C	2783	317	109	4.9	2.4	2.9	6	46.8	858%	17.2	19.7
Scrcpy	C	2756	383	155	2.8	1.7	2.5	3	29.7	963%	17.7	23.0
Eladmin	Java	2663	528	291	2.0	1.3	1.8	3	17.0	750%	8.0	16.7
space_blok	C++	2565	375	259	2.0	1.1	1.5	3	12.7	545%	5.7	4.7
PiggyMetrics	Java	1510	228	215	1.1	1.0	1.1	0				
JMX_Exporter	Java	1387	250	118	2.4	1.5	2.1	4	22.8	832%	8.8	19.3
solitaire	C++	1231	192	67	3.3	1.8	2.7	4	22.0	573%	9.0	20.0
sudokumaster	C++	1204	190	98	2.5	1.6	1.9	2	11.5	356%	9.0	6.0
weekly_planner	C++	1168	155	99	2.0	1.1	1.6	1	11.0	450%	1.0	1.0
diner	C++	1029	150	96	2.0	1.1	1.6	0				
Wumpus	Java	938	206	91	3.2	1.6	2.3	6	17.5	442%	6.2	10.5
Java.Battleship	Java	921	130	67	3.3	1.6	1.9	8	14.5	334%	3.5	5.0
GadgetProbe	Java	733	138	79	1.8	1.4	1.8	1	24.0	1204%	17.0	20.0
protractore	C++	378	46	31	2.3	1.4	1.5	0				
Spring Analysis	Java	316	86	80	1.1	1.0	1.1	0				
Bubble_level	C++	169	23	17	1.8	1.0	1.4	0				
Weighted mean					2.3	1.4	2.1		21.4	821%	9.3	16.9
Total		4,011,607		359,059				8468				

Table 3 contains a selection of extra large, large, medium, small, and extra small applications from the sample data. Axelor has 106,385 lines of code and is classified as large because it is the closest fit for the large grouping. In this table, 21 McCabe unit, design, data, and transformation metrics are shown. The % n, % S_0, and % S_1 metrics measure the portion of modules, design complexity, and integration complexity residing in the high risk modules (v > 10). For example Metafresh, 1% of the application modules are high risk modules. This 1% contains 9% and 24% of S_0 and S_1, respectfully, a disproportionate amount of the design and integration complexities. Total ldv, pgdv, and pdv are included so that additional quality and risk attributes can be evaluated. Earlier in this paper, encapsulation, separation of concerns, and data hiding were referenced. Recall that low use of global data and high use of local and parameter data infers lower risk. In Table 3, note the density ratios of data metrics to cyclomatic complexity (ldv/v, pgdv/v, and pdv/v). This data transformation normalizes McCabe metrics across size classification. The comparison of extra large, large, medium, small, and extra small application shows that metrics do not increase due to size as measured by LOC:

Table 3. Application risk comparison (v > 10).

Metric	Extra Large			Large		Medium		Small		Extra Small	
	Metafresh	Adempiere	Git	Axelor	Mes	Portfolio	Open Refine	Libevent	Tmux	Darknet	Gaussian YOLOv3
% n	1%	3%	9%	3%	1%	1%	4%	7%	15%	9%	2%
% S_0	9%	21%	36%	15%	5%	11%	25%	26%	47%	34%	14%
% S_1	24%	39%	44%	22%	11%	26%	43%	37%	57%	44%	19%
total	18,367	35,534	16,349	2799	927	1671	3483	1940	3483	657	666
average	17.8	20.5	25.6	16.4	14.7	17.6	17.9	17.5	23.5	18.3	18.5
total	8231	15,865	8278	1184	464	478	2249	1027	1709	206	207
average	8.0	9.1	13.0	6.9	7.4	5.0	11.5	9.3	10.5	5.7	5.8
ev density (ev/v)	0.45	0.45	0.51	0.42	0.50	0.29	0.65	0.53	0.49	0.31	0.31
total	15,092	29,967	1106	2398	826	1488	2796	1293	2133	483	490
average	14.6	2.1	17.2	14.0	13.1	14.0	14.3	11.6	20.4	13.4	13.6
iv density (iv/v)	0.82	0.84	0.67	0.86	0.89	0.89	0.80	0.67	0.61	0.74	0.74
total	14,305	27,605	6921	2141	808	1135	1503	486	1041	444	448
average	13.8	16.1	10.8	12.5	12.8	12.0	7.7	4.4	19.8	12.3	12.4
ldv density (ldv/v)	0.78	0.78	0.42	0.76	0.87	0.68	0.43	0.25	0.30	0.68	0.67
total	10,450	18,286	1976	1324	621	864	1962	163	370	179	185
average	10.1	10.6	3.1	7.7	9.9	9.1	10.1	1.5	15.2	5	5.1
pgdv density (pgdv/v)	0.57	0.51	0.12	0.47	0.67	0.52	0.56	0.08	0.11	0.27	0.28
total	6152	11228	4413	1416	527	7456	957	410	880	267	274
average	6	6.3	6.3	8.3	8.4	8.0	4.9	3.7	8.5	7.4	7.6
pdv density (pdv/v)	0.33	0.32	0.27	0.51	0.57	0.48	0.27	0.21	0.25	0.41	0.41
Total LOC	964,889	736,289	736,183	106,385	96,263	74,071	56,782	42,943	30,595	11,839	11,995

- Mes, a large application, has a lower average v(g) than Tmus (23.5), a small application.
- Metafresh, a extra large application, has a high local data density ratio (0.78) while Libevent, a small application, has a low local data density ratio (0.25).
- Mes, a large application, makes much greater use of public global data (0.67) than Tmux (0.11), a small application.
- The three extra large applications, Metrafresh, Adempiere, and Git, make less use of parameter data, 0.32, 0.33, and 0.27, respectively, than the two large applications, Axelor and Mes, 0.51 and 0.57, respectively.
- Adempiere, an extra large application, averages more local data use (average 16.1) than Gaussian YOLOv3 (12.4), an extra small application.
- Git, an extra large application, averages less public global data use (3.1) than Tmux (15.2), a small application.
- Axelor, a large application, averages more parameter data use (83.) than Libevent (3.7), a small application.

Size does not dictate the negative or risk measures for the application. The measurement challenge is integrating multiple quality and risk factors into a simple, meaningful format. In the next section, a quality classification algorithm is introduced to support risk assessment for open source software.

5. Application Risk Score

During thirty years of working with McCabe & Associates clients, antidotal evidence surfaced regarding their software. Due to non-disclosure agreements (NDA), clients chose to not publicize metric analysis of their code bases. However, client software exhibited risk factors that can be applied to open source software. For example, when examining client applications, it was observed that high risk modules accounted for 0% - 7.5% of the total modules. Note that in Table 3 the proportion of high risk modules to total modules ranges from 1% to 15%. This observation is incorporated into an algorithm for an application risk score. To calculate an application risk score, 9 metrics and transformations associated with the McCabe metric family shown in Table 3 are utilized. Let's highlight these nine.

1) % n: the percentage of high risk modules to total modules.

2) % S_0: the percent of total design complexity for high risk modules to total application design complexity.

3) % S_1: the percent of total integration complexity for high risk modules to total application integration complexity.

4) μ_v: The average cyclomatic complexity for high risk modules.

5) ev density: the ratio of total essential complexity for high risk modules to total essential complexity for all application modules.

6) iv density: the ratio of total module design complexity for high risk modules to total module design complexity for all application modules.

7) ldv density: the ratio of total local data complexity for high risk modules to total local data complexity for all application modules; this ratio is subtracted

from one to place density ldv in the same order of magnitude as other ratios; high density ldv is positive; high "1 – high density ldv" is negative.

8) pgdv density: the ratio of total public global data complexity for high risk modules to total public global data complexity for all application modules.

9) pdv density: the ratio of total parameter data complexity for high risk modules to total parameter data complexity for all application modules; this ratio is subtracted from one to place density pdv in the same order of magnitude as other ratios; high density pdv is positive; high "1 – high density" pdv is negative.

A weight is added when calculating the application risk score. The applicable v(g) quartile is weighted twice. This weight is justified based upon the empirical significance of v(g) and v(g) > 10 association with high risk software. As shown in Figure 1, the application risk score is calculated by assigning each of the 9 metrics into four quartiles – low risk, moderate low risk, moderate high risk, and high risk quartiles. Then, the quartiles are combined into an overall risk score for the application.

With 10 data points (recall that μ_v is weighted twice) used to calculate the application risk score, its magnitude can be from 10.0 to 40.0. Using equal groups for this range, the application risk score classifications are as follows (See the source code risk score in Figure 1):

- 10 < low risk ≤ 17.5
- 17.5 < moderate low risk ≤ 25.0
- 25.0 < moderate high risk ≤ 32.5
- 32.5 < high risk ≤ 40.0

Table 4 illustrates the metrics associated with the risk score and risk classification for 4 sample applications. In this table, low risk, moderate low risk, and moderate high risk are shown. Metafresh and Adempiere have moderate low risk scores (20 and 24, respectively) and risk classifications; Ofbiz has a moderate high risk score (24) and moderate high risk classification; and samples_maps has a low risk score (15) and low risk classification.

	Quartile 1	Quartile 2	Quartile 3	Quartile 4
Metric	Low Risk	Moderate Low Risk	Moderate High Risk	High Risk
% n_{risk}	$n_{risk} \leq 2.5\%$	$2.5\% < n_{risk} \leq 5.0\%$	$5.0\% < n_{risk} < 7.5\%$	$n_{risk} > 7.5\%$
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	10 ≤ risk score ≤ 17.5	17.5 < risk score ≤ 25.0	25.0 < risk score ≤ 32.5	32.5 < risk score ≤ 40

Figure 1. Risk score classification.

Table 4. Risk score calculation examples.

Static Metrics	Metafresh	Quartile	Adempiere	Quartile	Ofbiz	Quartile	maps_samples	Quartile
% n_{risk}	1%	1	3%	2	7%	3	1%	1
% S_0	9%	1	21%	1	43%	2	19%	1
% S_1	24%	1	39%	2	59%	3	14%	1
2 * μ_v	17.8	4	20.5	6	23.5	6	14.3	4
ev density	0.45	2	0.45	2	0.45	2	0.39	2
iv density	0.82	4	0.84	4	0.87	4	0.52	3
(1 – ldv density)	0.22	1	0.22	1	0.84	4	0.00	1
pgdv density	0.57	3	0.51	3	0.64	3	0.00	1
(1 – pdv density)	0.67	3	0.68	3	0.36	2	0.00	1
risk score	20		24		29		15	
risk classification	Moderate Low Risk		Moderate Low Risk		Moderate High Risk		Low Risk	

Consider Metafresh. Metafresh risk score is 20 based upon the following 10 data points summarized into its risk score:

Metric	Value	Quartile
% n_{risk}	1%	1
% S_0	9%	1
% S_1	24%	1
2 * μ_v	17.8	2 × 2
ev density	8231/18,367 = 0.45	2
iv density	15,892/18,367 = 0.82	4
1 – ldv density	1 – (14,305/18,367) = 0.22	1
pgdv density	10,450/183,676 = 0.57	3
1 – pdv density	1 – (6152/18,367) = 0.67	3
Total		20

The associated risk classification for Metafresh risk score (20) is moderate low risk.

Sample Risk Classification

In Table 5, twenty-five open source applications are shown. Table 5 shows the risk scores and classifications for the twenty-five largest sampled open source applications. In the sample, applications are classified low risk, moderate low risk, and moderate high risk. No open source application in the sample was classified as high risk. When calculating the risk score and determining the risk classification, fifteen open source applications are eliminated because the number of risk modules are low (0, 1, 2, 3, or 4). Applications with few or no risk modules are considered not risky. Only 16% of the open source applications fell into the moderate high risk classification. Of the twenty-five largest open source applications, twenty-one are classified as moderate or lower risk indicating that risk assessments for sample open source applications overwhelmingly fall into low risk groupings.

Table 5. Application risk summary.

Application	Language	Size Classification	Risk Score	Risk Classification
IJPlayer	C	extra small	30	Moderate High Risk
Blueseer	Java	extra large	27	Moderate High Risk
Ofbiz	Java	extra large	26	Moderate High Risk
Scorpio	Java	extra large	26	Moderate High Risk
Git	C	extra large	25	Moderate Low Risk
Adempiere	Java	extra large	24	Moderate Low Risk
Idempiere	Java	extra large	24	Moderate Low Risk
Nacos_Develop	Java	small	24	Moderate Low Risk
Schedulis	Java	small	24	Moderate Low Risk
Tmux	C	small	24	Moderate Low Risk
Wumpus	Java	extra small	24	Moderate Low Risk
Axelor	Java	large	23	Moderate Low Risk
Darknet	C	extra small	23	Moderate Low Risk
Dbeaver	Java	extra large	23	Moderate Low Risk
Mes	Java	large	23	Moderate Low Risk
Open Refine	Java	medium	23	Moderate Low Risk
RedDragon ERP	Java	medium	23	Moderate Low Risk
Redisson	Java	medium	23	Moderate Low Risk
Portfolio	Java	medium	22	Moderate Low Risk
Java.Battleship	Java	extra small	21	Moderate Low Risk
Libevent	C	small	20	Moderate Low Risk
Metafresh	Java	extra large	20	Moderate Low Risk
Gaussian YOLOv3	C	extra small	19	Moderate Low Risk
phone_info	C++	extra small	16	Low Risk
maps_samples	C++	extra small	15	Low Risk

6. Further Research

Based upon the finding of this research, future research should examine the composition of the risk score. The 9 software metrics should be examined to determine if they are objective proxies for software quality and risk. A limitation of this study is also the sample size and sample collect technique. Additional open source applications should be collected, and the platforms for collection should extend to broader open source communities and vendor repositories. This research can be conducted in a different context. The risk assessment algorithm can be extended to testing potential risk differences based upon application type, application size, and programming language.

7. Conclusions

An examination of open source software sample showed that its quality is moderate low risk or lower. The criteria for this conducted assessment were based

upon established software metrics and design characteristics. The measurement of cyclomatic complexity is the single most important feature, since it is an accepted, empirically sound software metric in the software engineering disciple for analyzing software quality. The framework for the analysis of risk assessment contains additional proxy measurements of accepted design characteristics—encapsulation, separation of concerns, and data hiding. By utilizing quantification for these design characteristics, risk assessment is conducted for its susceptibility to objective review and quantification.

While the definition of software risk assessment is an important discipline task, it is also important that it be conducted independently and transparently. As in current software development methodologies, the independent variables for software quality are debated among practicing software engineers. Within the software quality framework, unit level, design level, and data metrics are embodied. In addition, the density transformation, one of the features of the model, is included as a measurement of software's risk potential by indicating which modules have the potential to be troublesome during future feature maintenance and expansion.

When risk classification was applied to the sample applications, it provided risk scores and classifications derived from unit, design, and data metrics. Analysis of the open source software sample code revealed that 84% of the application exhibited moderate low or low risk design architecture. Since open source code is readily attainable, the defined algorithm can be developed and refined across a wide domain of software functionality. Using McCabe metrics, additional open source applications can be analyzed with speed and accuracy. Using a weighted factor of cyclomatic complexity yielded a usable risk score and classification. This approach further integrates the analysis of software design and promotes the risk assessment process for management and software engineers.

Conflicts of Interest

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

References

[1] Sharda, R., Delen, D. and Turban, E. (2018) Business Intelligence, Analytics, and Data Science: A Managerial Perspective. Pearson, New York.

[2] Menzies, T. and Zimmerman, T. (2013) Software Analytics: So What? *IEEE Software*, **30**, 31-37. https://doi.org/10.1109/MS.2013.86

[3] (2022, January 31) IT Names: Richard Stallman—GNU. Apeiron Systems. https://apeirondb.com/en/blog/it-names-richard-stallman-gnu

[4] Murren, A. (2016, October 4) What Is Open Source Software. NIST. https://csrc.nist.gov/csrc/media/projects/supply-chain-risk-management/documents/ssca/2016-fall/tue_am_1_what_oss_is_and_is_not_andy_murren.pdf

[5] Jonas, J. (2022, March 6) What Is Open Source Software PDF? PostVines. https://postvines.com/what-is-open-source-software-pdf

[6] (2018, November 30) How to Use Open Source Software: Features, Main Software Types, and Selection Advice. Altexsoft. https://www.altexsoft.com/blog/engineering/how-to-use-open-source-software-features-main-software-types-and-selection-advice

[7] Ahlawat, P., Boynes, J., Herz, D., Schmieg, F. and Stephan, M. (2021, April 16) Why You Need an Open Source Strategy. BCG. https://www.bcg.com/publications/2021/open-source-software-strategy-benefits

[8] Raysman, R. (2011) Open-Source Software: Use and Compliance. Practical Law Publishing and Practical Law Company, Inc., New York.

[9] Wallen, J. (2015, March 5) 10 Best Uses of Open Source Software in the Business World. TechRepublic. https://www.techrepublic.com/article/10-best-uses-for-open-source-software-in-the-business-world/#:~:text=%2010%20best%20uses%20for%20open%20source%20software,is%20a%20challenging%20one%2C%20but%20there...%20More%20

[10] Lhotka, R. (2021, April 7) Responsible Use of Open Source in Enterprise Software. *Forbes*. https://www.forbes.com/sites/forbestechcouncil/2021/04/07/responsible-use-of-open-source-in-enterprise-software/?sh=6a58f4967bcf

[11] Watson, A.H. and McCabe, T.J. (1996) Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. National Institute of Standards and Technology, Gaithersburg.

[12] (2022, February 10) McCabe IQ. McCabe Software. http://www.mccabe.com/iq.htm

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

[14] (2022, February) Software Metrics Glossary. McCabe Software. http://www.mccabe.com/iq_research_metrics.htm

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