M-score: An Empirically Derived Software Modularity Metric

Ernst Pisch
Drexel University
Philadelphia, USA
epp26@drexel.edu

Yuanfang Cai
Drexel University
Philadelphia, USA
yc349@drexel.edu

Rick Kazman
University of Hawaii
Honolulu, USA
kazman@hawaii.edu

Jason Lefever
Drexel University
Philadelphia, USA
jtl86@drexel.edu

Hongzhou Fang
Drexel University
Philadelphia, USA
hf92@drexel.edu

Abstract

Background: Software practitioners need reliable metrics to monitor software evolution, compare projects, and understand modularity variations. This is crucial for assessing architectural improvement or decay. Existing popular metrics offer little help, especially in systems with implicitly connected but seemingly isolated files.
Aim: Our objective is to explore why and how state-of-the-art modularity measures fail to serve as effective *metrics* and to devise a new metric that more accurately captures complexity changes, and is less distorted by sizes or isolated files.
Methods: We analyzed metric scores for 1,220 releases across 37 projects to identify the root causes of their shortcomings. This led to the creation of M-score, a new software modularity metric that combines the strengths of existing metrics while addressing their flaws. M-score rewards small, independent modules, penalizes increased coupling, and treats isolated modules and files consistently.
Results: Our evaluation revealed that M-score outperformed other modularity metrics in terms of stability, particularly with respect to isolated files, because it captures coupling density and module independence. It also correlated well with maintenance effort, as indicated by historical maintainability measures, meaning that the higher the M-score, the more likely maintenance tasks can be accomplished independently and in parallel.
Conclusions: Our research identifies the shortcomings of current metrics in accurately depicting software complexity and proposes M-score, a new metric with superior stability and better reflection of complexity and maintenance effort, making it a promising metric for software architectural assessments, comparison, and monitoring.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

Software Metrics, Software Modularity, Software Evolution

ACM Reference Format:
Ernst Pisch, Yuanfang Cai, Rick Kazman, Jason Lefever, and Hongzhou Fang. 2024. M-score: An Empirically Derived Software Modularity Metric. In *Proceedings of the 18th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24), October 24–25, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3674805.3686697

ESEM '24, October 24–25, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-1047-6/24/10
https://doi.org/10.1145/3674805.3686697

1 Introduction

Software practitioners and managers need reliable architecture metrics to monitor software evolution, compare projects, and capture architecture decay or improvement [15, 21, 35]. Over the past few decades, researchers have proposed many metrics. The most well-known metrics focus on the complexity and quality of functions or entities within a file, such as McCabe's Cyclomatic Complexity [26] and Chidamber & Kemerer's metrics [9]. These metrics have been widely studied mainly for their ability to predict and locate defects [18, 22, 28, 43], and have been integrated into commercial tools like SonarQube [39]. However, these metrics are not designed to answer questions like "*Is the software architecture decaying?*" or "*Which project is better modularized than others*?" To the best of our knowledge, MacCormack et al.'s Propagation Cost (PC) [25] and Mo et al.'s Decoupling Level (DL) [29] are the only architecture level metrics that attempted to answer these two questions. They have been applied to multiple industry case studies and evaluated for their predictive and diagnostic abilities [24, 25, 30, 32, 46].

During our attempts to apply these metrics to real-world projects, we observed that the applicability and efficiency of PC and DL was limited to projects with no isolated components, or to projects of similar sizes. In particular, PC aims to reflect dependency density change but it is sensitive to project size—larger projects always have smaller PC scores—making it unsuitable for comparing projects with different sizes. DL, on the other hand, only reflects the degree of module independence but does not account for dependency density, and its score can be unstable with the presence of isolated files or components that have no syntactic relationship with other files in the project. There are cases when adding just one or two isolated files to a project can cause the DL score to change by more than 20%, which causes confusion among developers.

In this paper, we first report our empirical study on the prevalence of seemingly isolated files in monolithic systems and how and why they can impact modularity scores. The results of this empirical study guided the creation of our new software modularity metric: M-score, which integrates the best aspects of PC and DL while addressing their shortcomings. Specifically, similar to DL, M-score first clusters dependency relation among files into a hierarchical structure and aggregates the modularity score for each module of each layer. Different from DL, it also accounts for the complexity of these modules. Similar to PC, it calculates dependency density but measures densities between and within modules separately. M-score treats modules at various layers equally, so that adding isolated files will not affect the measure of other parts of the system. As a result, M-score rewards more independent modules and rewards low coupling both within and between modules: the more independent modules are there, and the less coupled between and within modules, the higher the M-score.

We evaluated M-score first by measuring its stability with that of DL in the presence of isolated files. To assess its ability to compare different projects, we calculated how M-score correlates to DL when measuring projects without isolated files, where DL demonstrated good performance in comparing different projects [30, 32, 46]. To assess if M-score can faithfully reflect maintenance effort, we assessed how it correlates to maintainability measures extracted from revision histories.

Our results show that the variation of M-score with and without isolated files is much smaller than that of DL, and its variation is always smaller than the variation of sizes, meaning that M-score is much more stable, less affected by sizes and isolated files, and hence better suitable for multi-release monitoring. Moreover, M-score highly correlates to DL when assessing projects without isolated files, meaning M-score can also be used to compare the modularity of different projects. Finally, M-score maintains a significant correlation to project maintainability measures extracted from historical data, meaning that it retains the important properties of a maintainability metric but with better stability. Thus M-score has the potential to serve as a stable modularity metric, not only to monitor system-wide architectural changes but also to reflect local complexity variations.

Our research contributions are twofold. First, the isolation file empirical study (Section 2) reveals the prevalence of, the reasoning behind, and the impact of isolated files on existing modularity scores, following Storey et al.'s framework [40]. Second, we contribute *M-score* (defined in Section 3), a novel modularity metric that integrates the best aspects of existing metrics while addressing their shortcomings. We also present a thorough evaluation of this new metric (Section 4), demonstrating its predictive value.

2 Empirical Investigation of Modularity Metrics

Our research is motivated by the practical need to monitor the evolution of a project during its development. In addition, many real-world projects want to compare their design quality with other projects in their domain as a gauge of quality. Our goal is for modularity degradation or improvement to be reflected by the modularity scores. We have studied the following metrics, which, to the best of our knowledge, are the state-of-the-art: (1) MacCormack et al.'s *Propagation Cost* (PC) that measures how tightly coupled a system is [25]. It is defined as follows: given the dependency graph of a system, it first calculates its transitive closure to account for all direct and indirect dependencies until no more dependencies can be added. PC is calculated as the number of total indirect and direct dependencies divided by the total number of possible dependencies, that is, the square of the total number of files. The lower the PC score, the less coupled the system. (2) Mo et al.'s *Decouplig Level* (DL) [29], an option-based metric [5, 41] that first clusters a dependency graph into a hierarchical structure and rewards small and independent modules, while penalizes dependencies among modules that reduce module independence.

PC is sensitive to size by definition—the PC score is inversely proportional to the square of file count: the more files there are, the better the PC score, making it meaningless to compare projects of different sizes. For the rest of this paper, we therefore focus on a comparison with DL. Although DL is less sensitive to system size, there are cases where the DL scores are very sensitive to the presence of isolated files. Even adding a single isolated file can increase DL by more than 20% in some industrial projects. In practice, however, even for monolithic systems, having isolated files is inevitable, due to framework usage, for example.

To analyze the prevalence, root causes, and significance of this problem, we decided to revisit the projects used to evaluate DL reported in the prior work [29], where DL was evaluated after removing all of the isolated files in these subjects. In our study, we calculate the modularity scores with and without isolated files, respectively, to study how isolated files may distort the metric. Specifically, we aim to answer the following research questions:

RQ1: *How prevalent are isolated files in traditional, non-distributed projects?* If the majority of monolithic projects have isolated files for valid reasons, it implies that a modularity metric should be stable with the presence of isolated files. That is, adding or removing a small number of isolated files should not drastically change the reported measure.

RQ2: *Why are these files isolated?* If a non-trivial portion of these monolithic projects has isolated files, we need to understand the underlying reasons of why they appear to be disconnected from the rest of the system. If these are just unused files or dead code, we could remove them; if these files are isolated for legitimate reasons, this implies that we need a better metric.

RQ3: *How and why do isolated files affect modularity scores?* The answer to this question requires a detailed investigation of these metrics and identifying the main problems, which will provide guidance for the design of a new metric.

To investigate these questions, we employed a data research strategy [40] using previously studied and published projects [29]. Next we introduce these subjects, our study methodology, the results answering these questions, and how these analyses guide the development of our new modularity metric.

2.1 Subjects

To compare against existing metrics, we analyzed 37 projects studied by Mo et al. [29]. These projects were selected from OpenHub[1], with different programming languages, sizes, and domains. Moreover, the evolution histories of these projects were well-managed—most committed code is linked to an issue ID—so that we can distinguish bug-fixing commits from other changes. These projects all have a non-trivial evolution history with 4 to 118 releases. In total, we studied 1220 releases of these projects. The first few columns in Table 1 present the demographic data of these projects, including their names ($Projects$), the number of releases (# $ofrels$), and the average file counts across all releases of each project (#$Files$).

2.2 Methodology

Now we elaborate on how we identified isolated files, examine their reasons for being isolated, and calculate their DL scores with and without isolated files.

1) To identify isolated files, we first extracted dependencies from source files using a static analysis tool called Understand[2]. This tool extracts static relations such as function calls and inheritance

[1]https://www.openhub.net/
[2]urlhttps://scitools.com/

Table 1: Subjects, Isolated Files and Modularity Scores

Projects	# of	File Averages			Category		Average DL (%)			Average M-score (%)			Standard Deviation		
	rels	#iso	#Files	iso%	ess	other	all	no_iso	Δ	all	no_iso	Δ	Iso%	Δ(*DL*)	Δ(M-score)
activemq	27	33.9	2672.3	1.27	X	X	83.76	74.05	9.71	75.88	75.57	0.31	0.002	0.030	0.001
amarok	67	13.3	1205.7	1.10		X	65.01	56.21	8.80	52.80	51.80	1.00	0.021	0.014	0.010
aurora	4	3.8	353.3	1.06		X	91.64	90.72	0.92	96.72	96.67	0.05	0.003	0.012	0.000
avro	26	4.5	290.0	1.54		X	79.22	78.12	1.09	86.39	86.17	0.21	0.011	0.008	0.002
camel	63	95.6	6457.6	1.48	X	X	88.06	82.91	5.15	86.27	86.04	0.22	0.002	0.005	0.001
cassandra	95	9.2	740.5	1.24	X	X	40.92	36.60	4.32	52.83	52.13	0.69	0.006	0.012	0.003
cxf	69	204.9	4311.0	4.75	X	X	94.71	90.21	4.50	85.85	85.14	0.71	0.002	0.005	0.001
derby	24	43.6	2463.4	1.77	X	X	69.57	62.50	7.07	56.20	55.41	0.79	0.001	0.021	0.001
flink	6	19.8	1864.2	1.06		X	83.03	80.88	2.15	81.99	81.79	0.20	0.001	0.010	0.000
flume	9	4.4	492.7	0.90	X	X	92.26	91.98	0.28	92.59	92.52	0.07	0.001	0.002	0.000
gedit	51	0.6	199.8	0.29		X	51.70	50.31	1.39	57.20	57.09	0.12	0.004	0.021	0.002
geronimo	21	154.1	2553.1	6.04	X	X	95.73	89.46	6.27	84.94	83.97	0.96	0.011	0.006	0.002
gimp	38	10.7	2715.7	0.39		X	50.57	43.26	7.31	50.45	50.26	0.20	0.001	0.005	0.000
gnome-atk	118	0.03	54.5	0.06		X	40.58	40.35	0.23	64.11	64.08	0.03	0.004	0.012	0.001
hadoop	80	15.0	1348.7	1.11	X	X	73.70	68.81	4.90	71.72	71.35	0.37	0.005	0.019	0.002
hibernate-orm	55	162.8	6052.1	2.69	X	X	73.61	67.47	6.13	66.65	65.76	0.89	0.009	0.005	0.003
hive	18	32.8	1870.6	1.75	X	X	59.40	55.19	4.21	57.97	57.17	0.80	0.016	0.023	0.003
jdt-core	15	1082	2212.8	48.90		X	93.31	72.13	21.18	80.92	65.84	15.08	0.149	0.050	0.046
log4j	21	7.4	256.2	2.90		X	80.90	77.74	3.16	82.24	81.63	0.60	0.006	0.017	0.001
mahout	9	24.8	1074	2.31		X	95.56	94.41	1.15	94.64	94.48	0.15	0.013	0.008	0.001
nant	8	18.9	317.6	5.94		X	82.92	72.30	10.63	80.81	79.44	1.38	0.018	0.021	0.002
nutch	22	10.6	374.2	2.84	X	X	88.89	85.79	3.10	89.75	89.50	0.25	0.033	0.020	0.003
ode	10	45	1029.9	4.37	X	X	78.77	73.01	5.76	66.91	65.40	1.50	0.007	0.015	0.002
oozie	10	8.9	719.9	1.24	X	X	65.76	64.07	1.69	77.67	77.38	0.29	0.001	0.007	0.001
openjpa	18	39.9	2887.5	1.38		X	71.96	64.49	7.47	66.71	66.34	0.37	0.006	0.031	0.002
pdfbox	19	5.2	765.6	0.67		X	51.10	45.97	5.13	52.08	51.75	0.33	0.000	0.008	0.000
pig	18	34.1	1320.7	2.58	X	X	63.63	55.40	8.23	66.95	66.17	0.78	0.023	0.017	0.007
spring-frm	42	361.3	5037.8	7.17	X	X	94.74	92.65	2.08	93.54	93.02	0.51	0.006	0.004	0.001
storm	7	13.1	563.3	2.33	X	X	82.40	76.25	6.14	78.48	77.94	0.54	0.003	0.051	0.001
tajo	7	23	1136.3	2.02		X	72.60	67.40	5.20	69.65	68.97	0.68	0.008	0.009	0.002
thrift	11	22.6	264.1	8.57	X	X	94.57	90.40	4.18	88.24	86.92	1.32	0.013	0.007	0.004
tika	16	9.1	323.2	2.82		X	85.11	82.92	2.18	84.16	83.77	0.39	0.017	0.009	0.003
tomcat	28	31.6	881.8	3.59		X	75.57	73.61	1.96	79.05	78.52	0.53	0.015	0.014	0.004
wicket	60	12.4	2472.3	0.50		X	76.20	71.35	4.85	78.73	78.63	0.11	0.001	0.005	0.000
wireshark	60	106.8	2582.8	4.14	X	X	43.13	23.46	19.67	36.99	34.24	2.76	0.005	0.023	0.004
xerces2-j	42	8.7	623.0	1.39		X	70.87	65.95	4.92	68.87	68.41	0.46	0.006	0.019	0.002
zookeeper	26	4.7	311.7	1.49		X	78.43	70.94	7.49	72.20	71.79	0.41	0.004	0.041	0.001
Total	1220	2679.1	60799.7		18	37									
Average	33.0	72.4	1643.2	3.67			75.13	69.71	5.42	73.76	72.79	0.97	0.012	0.016	0.003
Median	22.0	18.9	1074.0	1.75			78.43	72.13	4.90	77.67	75.57	0.46	0.006	0.012	0.002

among source files. For example, if File A *calls* File B, we consider that File A *depends* on File B, and the dependency type is function *call*. The *fan-out* of a file is the total number of files it depends on, and the *fan-in* of a file is the number of its dependents. If a file has no dependents and does not depend on any other files—that is, both of its *fan-in* and *fan-out* are zero—then this file is *isolated*. The 3rd and 5th column in Table 1 presents the average number of isolated files (*#iso*), and the average percentage of isolated files (*iso%*) across all releases of each project. This table shows that all projects have at least one release that contains isolated files.

2) To understand *why* these files are isolated, that is, why they do not appear to refer to, or to be referenced by, any other files in a project, we manually investigated the files and packages of each project, and identified two main categories of isolated files: Essential (*ess*) and Other (*other*) (the 6th and 7th columns in Table 1). Isolated files in the *ess* category contained references to external libraries and frameworks, and the project will not function without these files, whereas the *other* category contains files that can be excluded from production, such as tests, examples, and documentation. Table 1 depicts which projects contain which categories of isolated files. Although this manual process may not be completely accurate, e.g., a file categorized as being *esstial* may be obsoleted, the possible inaccuracy in these categorizations will not impact the result.

3) Finally, to understand how isolated files may distort DL, we calculated the DL scores of each release with and without isolated files, using a tool called DV8 [8], which has been used to calculate PC and DL scores as reported in multiple industrial case studies [20, 30, 32]. The DL scores were presented in the "*Average DL %*" columns of Table 1, which contains the average DL variation (Δ) calculated by subtracting the DL without isolated files (*no_iso*) from the DL with all files (*all*) and those values were averaged across all releases. Comparing the isolated file percentage (*iso%*) and DL variation (Δ) columns, it becomes clear that for many projects, a small increase in isolated files can cause drastic changes in DL, as we will discuss next. To understand *why* DL is sensitive to isolated files, we created examples to better understand the calculations for DL. Given these data and analyses, we are ready to answer these research questions.

2.3 Answers to Research Questions

(1) Answer to RQ1 (Prevalence of Isolated Files). Table 1 shows that isolated files are quite prevalent: all the 37 projects contained at least one isolated file in one of their multiple releases. While most

projects contain a relatively small percentage of isolated files across the average releases, we see that isolated files in the following 6 projects, nant, geronimo, spring-framework, thrift, and jdt-core, take more than 5% of all files. In project jdt-core, 48.90% of files are isolated. The data reveals that the presence of isolated files is extremely prevalent and should not be ignored.

(2) Answer to RQ2 (Reasons of Isolation). The "*Category*" columns in Table 1 shows that, of all the 37 projects, 18 of them have isolated but essential files. These files are isolated for various reasons: they can be external libraries, framework code, objects used in dependency injection, or classes needed for dynamical loading. Complex libraries and frameworks use interfaces, plug-ins, or dependency injection to load, and sometimes use client code within the framework or library itself. These files appear to be isolated because the source code of the external library or framework cannot be extracted by the static dependency extraction tools but should be included in modularity measurement.

All 37 projects contain isolated files that are test cases, examples, or documentation. These files are not essential to the system's functionality and can be excluded from production code. One might exclude these non-essential and isolated files in modularity measurement. However, many projects contain tests and example files that are explicitly connected with the rest of the system. It would be inconsistent if only isolated, non-essential files are excluded. Therefore, we do not attempt to exclude tests and examples within the repository in the calculations of the maintainability metric.

We also observed that some isolated files are difficult to categorize, as they do not reference external libraries nor appear to be test or example code. As these are real projects in the process of evolution, these files could be dead code, or files being developed and would be integrated with the rest of the files in the future.

Regardless of the reasons behind these isolated files, they are part of the repository with associated maintainability costs. Having these files included in a maintainability score seemed reasonable, as they may need to be understood by new developers, or used by the final product, all of which impact the maintainability of a project.

(3) Answer to RQ3 (Impacts of Isolation). Table 1 presents the average variation of DL (Δ), with (*all*) and without isolated files (*iso*). The table shows that while on average, isolated files make up about 3.67% of the projects, the average DL variation is 5.42%, meaning that the DL scores are significantly affected by the isolated files even though the complexity of the rest of the system remains the same. The distortion is more severe in some projects. For example, the DL scores of wireshark varied about 20% while only 4% of it are isolated files. As another example, in activemq, there are only 1.27% of isolated files, but its DL scores vary by 9.71%.

After recreating examples and investigating the DL calculations, we determined that the distortion of the score is because DL treats isolated files differently from other files. Since DL is based on the hierarchical structure of the dependency graph, adding a single isolated file introduces an additional layer. This might be fine, except that DL applies a size penalty only to the lowest layer. Therefore, if a system had a large module in the lowest layer, it would incur a severe size penalty. Once an isolated file is introduced, it becomes the new lowest layer and effectively removes the size penalty in that large module, because it is no longer in the bottom layer. Projects that have a large increase of DL due to isolated files are always those that have large modules in the lowest level, when isolated files are excluded.

2.4 Diagnosis of Current Metrics

The diagnosis of DL and PC provides guidance for our proposed new metric. In addition to the sensitivity issue of DL in the presence of isolated files, our investigation revealed two more gaps that were not handled by the current metrics. Unlike PC, which calculates dependency density only, DL doesn't take into account dependency density at all. As a result, even if there are big and overly complex modules in the system, as long as they are not in the bottom layer and do not have many dependents, there is no penalty for such complexity. As a result, DL ignores larger middle-layer modules and penalizes large, bottom-layer modules only. Therefore, even if the development team reduced the complexity of middle-layer modules, the DL score would not improve.

Another problem is that DL doesn't account for the number of dependencies between layers: as long as there are dependencies, either one or dozens of dependencies, DL has the same penalty. Consequently, even if a development team reduces the majority of dependencies between modules, the DL score does not change until all dependencies between modules are removed.

To summarize, our investigation highlighted a few inefficiencies in state-of-the-art modularity metrics: PC accounts for dependency density only but does not explicitly reward independent modules, and its definition determines that projects with more files have better PC scores. By contrast, DL rewards module independence, but ignores the dependency density within a module or dependency count between layers. It applies a size penalty only to the bottom layer (that contains independent modules), making DL oversensitive to isolated files. As a result, neither metric can be generally applied to compare different projects or reflect complexity variations effectively. These insights to the shortcomings in current metrics guided our creation of M-score.

3 Software Modularity Metric: M-score

In this section, we introduce the definition of our new software modularity metric: M-score, addressing the problems in existing metrics. In particular, M-score can faithfully reflect complexity variations in projects, with or without isolated files. We first introduce the definition of M-score, and then illustrate each of its factors using examples.

Definition. The M-score calculation is based on a dependency graph reverse engineering from a project, where each file in the project is a node. If File *A depends* on File *B*, there is a directed edge from node A to node B. Based on this graph, we then create a condensation graph that clusters these nodes into a hierarchical structure [45]. In this structure, files in lower layers depend on files in upper layers, but the reverse is not true. Within each layer, nodes are further decomposed into *independent modules*; that is, each module is a group of files that are connected with each other but do not depend on files within other modules of the same layer.

Fig. 1 illustrates such a hierarchical structure clustered from a dependency graph reverse-engineered from the source code of a survey management system. The system is supposed to handle

multiple types of questions and answers, different user interfaces, and different saving and logging approaches. This representation shows the layers after the dependency graph is clustered into a hierarchical structure (layers are separated by dashed lines). The clustered modules are the grey rounded rectangles that contain one or more circles. The circles represent individual files in the code base. The lines connecting the circles represent dependencies between the files. In this structure, the files in $layer_i$ can only depend on files in $layer_{i-1}$, and the modules within a layer are independent of each other.

By definition, the modules in the lowest level are *truly independent modules* [5] that do not have external dependencies and are able to be changed or replaced without needing to modify any other modules. Dependencies between modules are only across layers. We define the following terms and definitions:

- n: number of layers within the hierarchical structure
- m: number of modules within each layer
- j: the jth module, where j = 1..m
- $\#Files(M_j)$: number of files contained within the jth module
- N: number of files contained within the code base
- $\#LowerLayerDeps(M_j)$: number of files within lower layers that depend on module M_j
- $\#LowerLayerFiles$: number of files in all layers lower than the current level, L_i
- $InternalDeps(M_j)$: number of dependencies internal to the module M_j.

$$\text{M-score} = \sum_{L_i=1}^{n} \sum_{M_j=1}^{m} \text{SF}(M_j) * \text{CLDDF}(M_j) * \text{IMCF}(M_j) \quad (1)$$

$$\text{SF}(M_j) = \frac{\#Files(M_j)}{N} * \text{SP}(M_j) \quad (2)$$

$$\text{CLDDF}(M_j) = 1 - \text{CLDDP}(M_j) \quad (3)$$

$$\text{IMCF}(M_j) = 1 - \text{IMCP}(M_j) \quad (4)$$

$$\text{SP}(M_j) = \begin{cases} \frac{1}{log_5(\#Files(M_j))} & \text{if } \#Files(M_j) >= 5 \\ 1, & \text{if } \#Files(M_j) < 5 \end{cases} \quad (5)$$

$$\text{CLDDP}(M_j) = \begin{cases} \frac{\#LowerLayerDeps(M_j)}{\#LowerLayerFiles * \#Files(M_j)} & \text{if } L_i \neq n \\ 0, & \text{if } L_i = n \end{cases} \quad (6)$$

$$\text{IMCP}(M_j) = \begin{cases} \frac{\#InternalDeps(M_j)}{(\#Files(M_j)-1)*(\#Files(M_j)} & \text{if } \#Files(M_j) >= 5 \\ 0, & \text{if } \#Files(M_j) < 5 \end{cases} \quad (7)$$

For each module of each layer, we calculate the following terms:

- Size Factor (SF) (2)
- Cross Layer Dependency Density Factor (CLDDF) (3)
- Inner Module Complexity Factor (IMCF) (4)

Each of these factors has an associated penalty that accounts for the lack of modularity from different aspects.

- Size Penalty (SP) (5) is applied to each module to encourage smaller modules throughout the code base.
- Cross Layer Dependency Density Penalty (CLDDP) (6) accounts for the external dependencies of a module and rewards fewer coupling between layers.
- Inner Module Connected Penalty (IMCP) (7) accounts for the density of the internal dependencies within a module.

Based on these definitions, as shown in formulas (1) to (7), the score of each module is the combination of all three factors, and the M-score of the overall design is the summation of all modules of all layers (1). Large, complex modules will be penalized both for their size and complexity. Accordingly, the more small, independent modules there are, the higher the M-score; the fewer dependencies between layers, and the less coupling within a module, the higher the score. Next, we elaborate on each of these factors using illustrative examples.

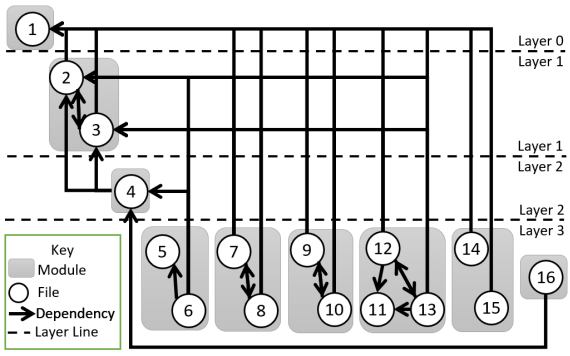

Figure 1: Illustration: A Hierarchical Structure

3.1 Size Penalty

Following the work of Mo et al. [29], we apply a size penalty to modules that contain more than 5 files. The threshold of 5 is based on Gobet and Clarkson's cognitive complexity [16] study: one can easily process approximately 5 "chunks" of information at a time. Miller [27]'s famous paper proposed a law of human cognition and information processing, stating that people can process no more than seven units of information, plus or minus two, at a given time. Based on these theories, we assume that a module with five or fewer files can be processed together by developers without extra effort to understand or maintain the module.

As shown in Equation (5), M-score penalizes all modules with more than 5 files in all layers of the hierarchical structure. This differs from DL, which only applies the penalty to the modules in the lowest layer. This means that when isolated files are added and form a new lowest layer—as in Fig. 2 where Fig. 2B adds one isolated file, $i1$, and forms a new Layer 2—we still apply the size penalty to the large module with $N - 1$ files in Layer 1, even though it is now moved to an upper layer. The M-score will increase slightly as adding independent modules should increase the M-score, but the increase should be small, reflecting just a minor structure change.

To illustrate how the M-score varies when adding an isolated file, we calculate it for Fig. 2A and Fig. 2B, along with the delta. Assume $N = 6$, then M-score$_A$ is 66.67%, M-score$_B$ is 71.43%: M-score increased 4.76 with 17% more files. When N grows large, e.g.,

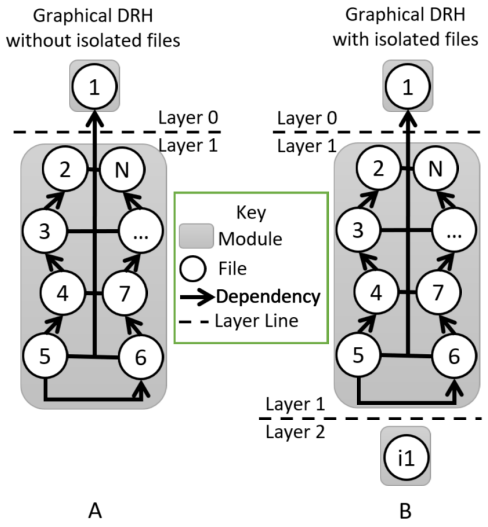

Figure 2: Adding an isolated file from A to B

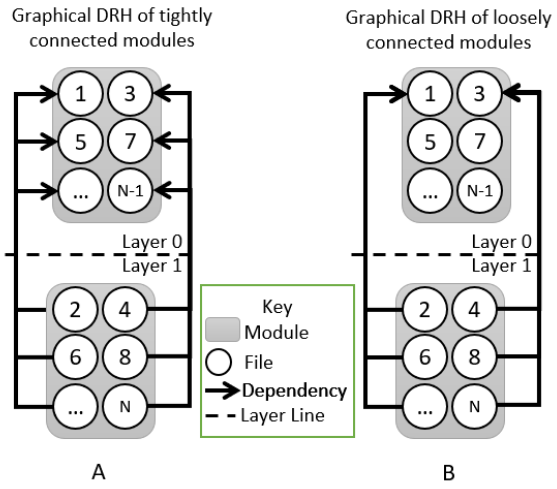

Figure 3: Reducing cross-layer dependencies from A to B

$N = 10,000$, then M-score$_A$ is 17.47% , M-score$_B$ is 17.48% and the change in M-score from A to B is 0.01%.

In general, as N gets extremely large, the *Size Penalty* becomes the largest penalty. The impact associated with adding an isolated file in a large system approaches zero, as adding a single isolated file with no connections or dependencies to other files should have less and less of an impact on the score. As such, adding an isolated file only adds a small percentage to the M-score, and the *Size Penalty* applied to a large module cannot be removed by simply adding one isolated file, which was the main issue of DL.

3.2 Cross Layer Dependency Density Penalty

The *Cross Layer Dependency Density Penalty* (*CLDDP*) of a module corresponds to the density of the files that depend on it compared to the total possible number of connections between those files. For any given module, the maximum possible dependencies are the number of lower-layer files multiplied by the number of files in the module. The penalty will increase when more dependencies are added and decrease when dependencies are removed.

We now calculate the M-scores for the graphs in Fig. 3A and Fig. 3B to demonstrate how the score will change when the dependencies crossing the two layers changed from 6 (Fig. 3A) to 2 (Fig. 3B). When N=8, two modules where each module has 4 files (hence no size penalty), the M-score$_A$ is 75.00%, M-score$_B$ is 87.50%, that is, M-score increases 12.50%, reflecting the reduced coupling. When N is large, e.g., $N = 12,000$, then M-score$_A$ is 13.88%, M-score$_B$ is 18.50%, increasing 4.62%, with the Cross Layer Dependency Density Penalty $CLDDP_B = 0.000167$, directly proportional to the amount of decoupling performed between layers.

3.3 Inner-Module Complexity Penalty

Lastly, the *Inner Module Complexity Factor* (*IMCF*) addresses the density of the internal connections of a module: a more complex module incurs a larger penalty. We now calculate the M-scores of the graphs in Fig. 4A and Fig. 4B, where Fig. 4B is refactored

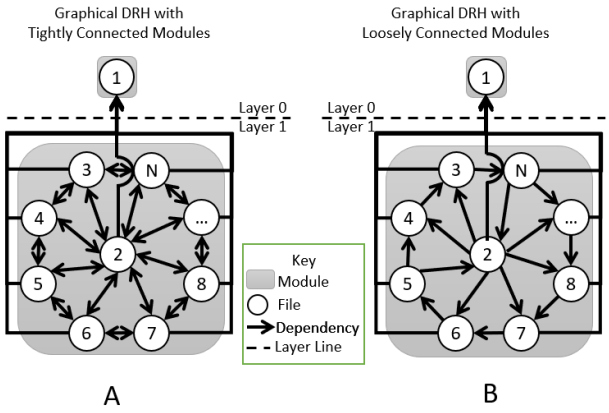

Figure 4: Reducing inner-module coupling from A to B

from Fig. 4A by changing all bi-directional dependencies among files to unilateral dependencies. In both graphs, there are N files, partitioned into two layers, one module in each layer, where the larger, lower-level module contains $N - 1$ files. Fig. 4A's lower-level module contains $4(N - 2)$ internal connections, and Fig. 4B's lower-level module contains $2(N - 2)$ internal connections.

Considering N=10, where the code base has 10 files, M-score$_A$ is 36.62%, M-score$_B$ is 51.27%, that is, the score increases 14.65% due to the reduced internal coupling. When N grows large, e.g., $N = 10,000$, then M-score$_A$ is 17.466%, M-score$_B$ is 17.469%, increasing 0.003%. This increase is so small because the main penalty comes from the large size of the module. The overall possible inner connections is: $N * (N - 1)$ therefore $\frac{4(N-2)}{N(N-1)}$ is still relatively small amount. It is also likely that while addressing inner-module complexity, the consendation graphs will change, which will create smaller modules and help increase the score.

Table 2: Statistics of DL and M-score

	DL_{all}	DL_{no_iso}	M-score$_{all}$	M-score$_{no_iso}$
Average	68.50%	63.09%	69.15%	68.39%
Median	73.10%	66.95%	68.97%	68.26%
Max	99.73%	95.31%	97.56%	97.14%
Min	34.63%	20.14%	31.44%	30.69%
Stdev	19.37%	20.03%	15.20%	15.40%
20th Pt	45.10%	42.13%	53.77%	53.19%
40th Pt	66.66%	60.17%	65.98%	64.78%
60th Pt	76.38%	71.22%	74.79%	74.11%
80th Pt	89.08%	84.01%	85.24%	84.43%

3.4 M-score Summary

In summary, the M-score definition combined the best aspects of both PC and DL, and addressed their shortcomings. Similar to DL, M-score rewards small and independent modules: the more such modules, the higher the score. Both DL and M-score follow Baldwin and Clark's options theory [5]: independent modules create option values through independent improvement, and the more high-value modules are there, the higher the M-score for the overall system. As shown in Formula (1), M-score, similar to DL, is the summation of the scores of all the modules of all layers. Unlike DL where a size penalty is only applied to the bottom layer of the hierarchy, the M-score *Size penalty* (SP), defined in Formula (5), is applied to all modules in all layers.

Similar to PC, M-score accounts for the density among files. Unlike PC, M-score, is not sensitive to size. Moreover, while PC calculates the dependency density of all files, M-score calculates dependency density within and across modules, respectively, and rewards fewer dependencies between layers and lower coupling within a module. As a result, reduced or increased complexity at different levels can be reflected in the score. The definition of *Cross–Layer Dependency Density Penalty* (CLDDP) (Formula (6)), and *Inner Module Connected Penalty* (IMCP) (Formula (7)) reflect these density penalties that make M-score different from DL and PC.

In summary, M-score, rooted in options theory, accounts for module independence, module size, inner-module complexity, and inter-module coupling simultaneously. To calculate the M-score for a project, we first use Understand to extract file dependencies, which are used by DV8 to generate the hierarchical clustering needed to calculate M-score. Using the output of DV8 as input, we created a script to calculate the M-score.

4 Evaluation

The objective of our evaluation is to empirically assess if M-score can serve as a reliable "metric" to (1) reflect software modularity variation, with or without isolated files, and (2) reflect maintenance effort as evidenced in the project's revision history. We use revision history to determine if a higher M-score indicates that features and bugs can be handled in a modularized way, such that different issues are handled by different developers in parallel. We compared M-score with DL, as DL is the state-of-the-art modularity metric to the best of our knowledge. Our goal was to empirically evaluate M-score, using the same set of subjects used to evaluate DL, to form a fair comparison. Specifically, we investigate the following evaluation questions:

EQ1: *Is M-score able to handle isolated files better than DL?* We answer this question by comparing the variation of M-scores with and without isolated files, respectively, against the variation of DL values. If M-score differences are smaller, then it performs better in monitoring the evolution of a software system. We determined previously that isolated files will happen during software evolution, and a useful metric should not be distorted when isolated files are added temporarily or permanently in the process of evolution.

EQ2: *Can M-score be reliably used to compare different projects and releases in terms of modularity differences?* Since prior works [29, 30, 32] have demonstrated that DL can be an effective measure to compare different projects and releases when there are no isolated files, we calculated the correlation of M-score and DL using projects without isolated files. If they are strongly correlated, it implies that M-score is also a reliable metric for project comparison.

EQ3: *Does M-score significantly correlate with maintenance measures derived from the revision history?* Following the work of DL [29], we investigate the correlation between M-score values with a suite of maintainability measures to assess how well development tasks can be parallelized.

We used the same subjects as reported in Section 2 to conduct our evaluation.

4.1 M-score Stability with Isolated Files

For all the releases of each of the 37 projects, we calculated the M-score and DL scores, with and without isolated files, and presented their summary statistics in Table 2. From this table, we can see that isolated files have a significant impact on DL: the median DL score reduces from 73.10% to 66.95% once isolated files are removed. This volatility makes it hard for DL to serve as a modularity benchmark since, as we analyzed in Section 2, isolated files are inevitable during software evolution. By contrast, the M-score statistics are much more stable: all the statistics differ less than 1%. The implication is that isolated files have a minor impact on M-score, as it mainly reflects the complexity, not the size, of a system. The benchmark formed by M-score can thus provide reliable references for projects. For example, we can say that if a project achieves a M-score of 68%, it reaches the median of the benchmark.

Table 1 contains more detailed data to reflect the stability difference between M-score and DL:

- The *Average DL* columns contain (*all*), the average DL score across all releases with all files; (*no_iso*), the average DL with only non-isolated files; and Δ, the difference between the two across all releases of the project.
- The *Average M-score* columns are defined similarly but present M-score instead of DL scores.
- The *Standard Deviation* columns present the standard deviations for the percentage of isolated files (*iso%*), the variations of DL scores (ΔDL), and the variation of M-scores, (Δ *M-score*) across all releases.

Table 1 provides a detailed stability comparison for each project. For example, the "*activemq*" row shows that when the file count changes 1.27% because of isolated files, the average DL change is 9.71%, while the M-score only changes 0.31%, which is expected

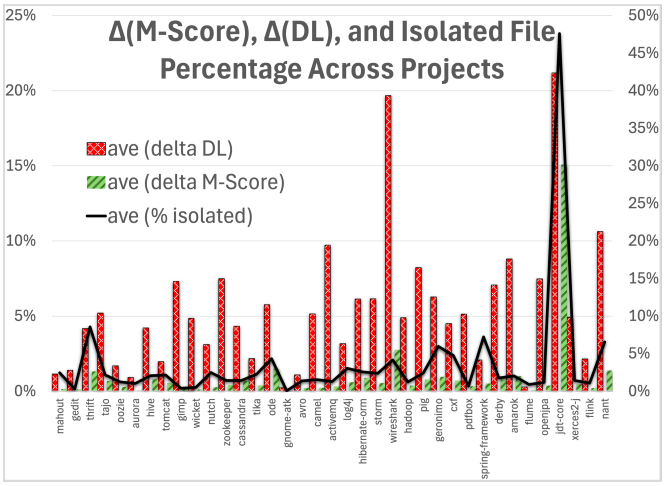

Figure 5: Variation of DL, M-score, and file count

Table 3: DL and M-score Correlations: all p-values < 1e-309

Metrics	DL_{all}	DL_{no_iso}	M-score$_{all}$	M-score$_{no_iso}$
DL_{all}	-			
DL_{no_iso}	97.34%	-		
M-score$_{all}$	85.04%	89.86%	-	
M-score$_{no_iso}$	82.85%	89.14%	99.31%	-

Table 4: Spearman correlation to maintenance measures

History Metrics	rho	p-value
Code Commit Overlap Ratio (CCOR)	-0.44	5.90e-03
Bug Commit Overlap Ratio (BCOR)	-0.52	9.61e-04
Code Commit Fileset Overlap Ratio (CCFOR)	-0.48	2.80e-03
Bug Commit Fileset Overlap Ratio (BCFOR)	-0.53	7.56e-04
Code Pairwise Committer Overlap (CPCO)	-0.51	1.31e-03
Bug Pairwise Committer Overlap (BPCO)	-0.52	9.98e-04

(and much more stable). For all projects, the DL variation is *always* higher than that of the M-score. The chart in Fig. 5 shows that, compared with file count variation, DL varies drastically, while M-score is more stable and lower than file count variations, showing that M-score is less affected by file isolation or size variation.

The last rows of Table 1 show that the M-score variation caused by isolated files is significantly less than that of DL variation: On average, the isolated files increase total file count by 3.67%, which causes an average increase of 5.42% in DL, but only a 0.97% increase in M-score. Similarly for the median case, the isolated file percentage is 1.75%, which causes a median DL increase of 4.90%, but only a 0.46% increase in M-score.

To statistically assess if the standard deviation of M-score variation is significantly smaller than that of DL, we first tested the average standard deviations of both scores across the releases for each of the 37 projects, as listed in the "*Standard Deviation*" columns in Table 1, using the Anderson-Darling test [4]. The normality test determined that neither standard deviation columns in Table 1 are normally distributed: the p-value for $\Delta(DL)$ and Δ(M-score) are $1.6791E-4$ and 0 respectively. As neither fits a normal distribution, we used the non-parametric 1-tailed Mann-Whitney U Test and determined that M-score standard deviations are smaller than that of DL, as indicated by the small p-value: of 2.27E-11.

Answer to EQ1: Compared with DL, M-score is better at handling isolated files within a project. While the exact impact of adding isolated files and the number of isolated files varies depending on the structure of the code base, in all cases, the percent increase of M-score is strictly less than that of DL. The standard deviation of M-score with isolated files is strictly and significantly smaller than that of DL, as presented in Table 1.

4.2 M-score for Monitoring and Comparison

Since DL has been used to effectively compare and monitor both open-source and industrial projects without isolated files [29, 30, 32, 46], to ensure that M-score can be used for similar purposes, we calculated the Spearman correlation coefficient and p-value to determine the correlation between M-score and DL across the 1220 releases. When comparing M-score and DL, which can vary greatly in the presence of isolated files, we calculate the correlation between each pair of the following metrics:

- DL_{all}: DL score including isolated files
- DL_{no_iso}: DL score excluding isolated files
- $M\text{-}score_{all}$: M-score value including isolated files
- $M\text{-}score_{no\_iso}$: M-score value excluding isolated files

Their pair-wise correlation is presented in Table 3. All the metric pairs are highly correlated, and these correlations are all significant with *p-value < 1E-309*, which makes us confident that using any of these metrics to monitor project evolution in the long run should provide similar results. The table also shows that M-score with and without isolated files are almost perfectly correlated with a coefficient of 99.31%, which is another demonstration that M-score is stable in the presence of isolated files. Both M-score$_{all}$ and M-score$_{no_iso}$ are highly correlated with DL_{no_iso}, with a coefficient greater than 89%. The implication is that M-score can also be used to indicate architecture variation, similar to a real "*metric*".

Answer to EQ2: The high correlation numbers and low p-values make it clear that M-score can also be used to monitor architecture variation and compare different projects, but will be more stable in the presence of isolated files or components.

4.3 M-score and Maintenance Metrics

To measure how well maintenance tasks are accomplished in parallel, prior work [29] established three maintenance metrics: Commit Overlap Ratio (COR), Commit Fileset Overlap Ratio (CFOR), and Pairwise Committer Overlap (PCO). These metrics use a project's revision and issue-tracking histories to determine how much parallel development and maintenance *could* have occurred in each project, based on the assumption that if the project could have been developed and maintained in parallel, it is likely that the project is decoupled. Their hypothesis was that for projects with higher modularity scores, their code base should be better modularized, and thus, a developer's maintenance tasks should be better paralleled.

Table 5: M-score vs. Maintenance Measures

Project	Code COR	Bug COR	Code CFOR	Bug CFOR	Code PCO	Bug PCO	Average M-Score
activemq	4.62	2.88	2.68	1.72	0.53	0.50	75.88%
amarok	13.76	4.01	4.15	1.94	2.05	0.90	52.80%
aurora	6.09	1.96	2.39	1.76	0.36	0.20	96.72%
avro	4.94	2.85	1.49	1.22	0.21	0.10	86.39%
camel	5.72	1.96	2.30	1.29	0.31	0.15	86.27%
cassandra	14.18	5.07	4.19	2.24	1.48	0.88	52.83%
cxf	6.93	2.22	2.09	1.35	0.50	0.23	85.27%
derby	11.20	2.97	4.85	1.70	1.93	0.44	57.48%
flink	4.45	1.37	2.82	1.23	0.57	0.14	77.61%
flume	3.63	2.15	1.76	1.50	0.55	0.56	91.40%
gedit	16.74	3.82	5.82	2.55	0.79	1.28	57.96%
geronimo	3.81	2.12	2.11	1.44	0.80	0.42	83.42%
gimp	16.93	4.35	5.43	2.05	1.23	0.38	51.44%
gnome-atk	18.52	5.67	6.84	3.36	0.46	1.47	63.95%
hadoop	5.26	2.21	2.56	1.83	1.28	1.13	71.60%
hibernate-orm	3.34	1.66	2.49	1.35	0.38	0.25	66.83%
hive	6.22	3.09	3.18	1.93	1.71	1.13	58.54%
jdt-core	8.77	4.92	2.68	2.30	1.73	1.26	82.59%
log4j	7.85	2.98	2.66	1.26	0.83	0.27	79.77%
mahout	4.81	1.56	2.85	1.37	0.86	0.31	93.23%
nant	10.56	2.12	3.17	1.06	0.91	0.02	83.19%
nutch	5.81	2.26	3.26	1.67	1.26	0.53	89.16%
ode	4.00	2.03	2.15	1.58	0.74	0.44	68.93%
oozie	5.96	3.48	3.50	2.47	1.72	1.34	77.37%
openjpa	5.05	1.76	2.74	1.56	1.37	1.01	66.82%
pdfbox	5.67	2.40	2.03	1.45	0.66	0.45	53.13%
pig	6.54	2.48	2.48	1.79	0.95	0.66	65.86%
spring-framework	5.34	2.19	2.94	1.41	0.31	0.17	92.97%
storm	4.14	1.30	2.10	1.27	0.65	0.13	80.47%
tajo	3.44	2.54	1.71	1.31	0.44	0.28	71.45%
thrift	3.63	2.29	2.04	1.48	0.12	0.10	86.11%
tika	7.75	2.60	2.56	1.69	1.08	0.53	84.49%
tomcat	9.11	3.15	2.44	1.42	0.62	0.20	79.62%
wicket	4.97	2.22	3.03	1.62	0.84	0.58	78.69%
wireshark	21.38	4.22	6.62	2.72	2.65	1.04	38.43%
xerces2-j	8.42	2.47	3.04	1.50	0.86	0.31	67.24%
zookeeper	5.48	2.57	1.95	1.94	0.96	1.06	71.78%
P-value	3E-7	8E-4	2E-6	1E-3	2E-2	7E-4	2E-1
Normally Distributed?	No	No	No	No	No	No	Yes

To evaluate M-score, we obtained and generated similar inputs from the repositories of our subjects to determine if and how well M-score correlates with COR, CFOR, and PCO. Each of which can be decomposed into bug and all commit metrics: BCOR (Bug COR), BCFOR (Bug CFOR), BPCO (Bug PCO), CCOR (Commit COR), CCFOR (Commit CFOR), and CPCO (Commit PCO). Table 5 contains the maintenance metrics broken down by project.

The maintenance metrics were not normally distributed as indicated by the Anderson-Darling test p-value shown in Table 5. Accordingly, we decided to use Spearman's rank correlation instead. Table 4 presents these correlations, which shows that all these metrics are significantly and negatively correlated with the maintenance measures with very small p-values.

Answer to EQ3: Yes, M-score is significantly correlated with maintenance metrics: the higher the M-score, the more likely that maintenance tasks can be accomplished independently.

4.4 Evaluation Summary

In summary, we have shown that M-score addresses the limitations of DL, better reflects inter-layer and inner module complexity, and is more stable in the presence of isolated files. In addition, as a metric, it has the same key properties of reliably reflecting architectural-level modularity variations and maintenance efforts.

5 Threats to Validity

In this section, we discuss external, internal, and construct threats to the validity of our study, following [7].

External Threats to Validity. The projects we examined are mostly Apache projects written in Java, C, C++, and C#. Therefore, it is possible that M-score may not be applicable to other projects in other programming languages. It is also possible that other open-source projects in other communities or industrial projects may introduce issues that could necessitate refinement of our metric.

Internal Threats to Validity. The primary internal threat to validity is associated with the tool used to extract dependencies between files. In this study, we used Understand [2] to extract dependencies. It is possible that if we use another static analysis tool, such as Depends [1], the results could be different. In our comparative evaluation with DL, we minimized the risk of this bias by analyzing the same projects and using the same tools as used in prior work [29].

Construct Threats to Validity. In this study, we used the maintenance measures, COR, CFOR, and PCO proposed by Mo et al. [29] as the ground truth, which may not reflect real maintenance effort or experts' perceptions, presenting a restricted generalizability threat. In practice, real history data has an ebb and flow with new features being added and bugs being swatted, all complicated by deadlines, inexperienced developers, turnover, and many other issues that could make the data noisy. As such history data may not perfectly represent the amount of decoupling within a code base.

6 Discussion

Here we discuss the implications and future work of our research.

Implications. We believe that M-score takes a step further towards our overall objective of seeking an *operable metric*. That is, using such a metric, the designer should be able to faithfully assess if the quality of their design is getting better or worse as the system evolves. If the system is refactored, locally or globally, the metric should be able to tell the extent to which the design has improved. If a developer takes a shortcut and introduces unexpected dependencies, the metric should be able to reflect the presence of modularity degradation immediately so that severe architecture decay can be detected and prevented early.

Having a reliable modularity metric is the first step towards an industrial benchmark comprising modularity scores collected from a large number of projects, so that any project can compare its modularity level with similar projects, i.e., projects using the same language, in the same domain, or from the same corporation.

Future Work. We envision a number of extensions and future work of our research.

Investigate Dependency Extraction Tools. In this study, we only used Understand [2] to extract dependencies. A future research effort could be to compare and contrast the features, capabilities, and limitations of different dependency extraction tools to better understand what should be the most accurate and efficient way to extract dependency data. Possible questions to investigate are: what are the differences between dependency extraction tools? How do they detect third-party software used in the projects, and

how do they handle different language mechanics or frameworks? How the tools handle these issues could result in different sets of dependencies. For example, is an isolated file actually disconnected from any dependencies, or could the tool not detect it?

Incorporate Dependency Type and Weight into Metrics. M-score considers all types of dependencies to be equal, whether that dependency is *Call*, *Extend*, *Import*, or *Implement*. One could argue that these different dependencies represent a different amount of coupling; for example, *Extend* and *Implement* are likely stronger dependencies than *Contain* or *Import*. The current M-score doesn't account for dependency strength between two files either: two files calling each other multiple times have the same dependency count as those only call each other once. Incorporating dependency types and weights may lead to a more accurate modularity metric.

Case Study using Actual Maintainance Effort. Our study uses revision history data to approximate maintainability. In the future, we would evaluate M-score using actual maintenance effort, measured using coding time or LOC, collected from real industrial projects. Another option is to use a respondent empirical strategy where we solicit opinions from domain experts, asking them to identify architecture improvements (and degradations) and then investigate if those correlate to an increase (or decrease) in M-score.

7 Related Work

Determining how to measure software to assess maintainability and prevent architecture decay has been an indispensable step for software quality control and improvement [15, 21, 35], and for identifying refactoring opportunities during software evolution. Software metrics have been studied for decades. McCabe Cyclomatic complexity [26] and Halstead metrics [17] are among the oldest and well-known ones to measure program complexity. C&K metrics [9], LK Metrics [23], and MOOD Metrics [12] were proposed to measure object-oriented programs. These metrics are derived mainly from inner file relations and have been used for bug prediction and localization [18, 22, 28, 38, 43], but not for comparing and monitoring architecture decay or modularity variation.

Researchers have also explored the effectiveness of using combined metrics to aid software maintenance and evolution. Oman and Hagemeister [33] proposed a Maintainability Index, a composite model for predicting maintainability integrating multiple metrics, including lines of code and cyclomatic complexity per module. Bijlsma et al. [6] and Heitlager et al.'s [19] combined metrics for software maintainability assessment. Commercial tools such as SonarQube [39] and Infusion [12] also provide a single "index" to monitor software quality.

More recently, Papamichail and Symeonidis [34] presented a framework to identify non-maintainable components by training Support Vector Machines classifiers based on multiple metrics, such as complexity, cohesion, coupling, and inheritance properties. Czuibula et al. [11] proposed a new aggregated coupling capturing both the structural and the conceptual coupling. The problem, however, is that measures derived from one set of projects cannot be used for other projects [31, 42].

In addition to object or class level metrics, researchers have also proposed metrics for packages [3], components [14], and subsystems [37] to predict maintainability, testability, and run-time communication structure complexity between distinct components and their relationships to quality factors including maintainability. Given the popularity of microservices, metrics have also been proposed to measure microservice systems. Similar to monolithic systems, coupling is still considered an important indicator that negatively impacts service maintainability. Independence among service components is important to allow the system to change and evolve. [10, 13, 35, 44, 47].

In this paper, we focus on monolithic systems and attempt to find independent modules, following the essence of design rule theory [5, 41]: independent modules create option values, and the more such modules there are, the higher the value of the overall system. To the best of our knowledge, the only two option-based metrics that have been used to compare different projects or to monitor software evolution continuously are PC and DL. DL is the only modularity metric that has been successfully applied to hundreds of projects, both open source and industrial [30, 32, 46]; it has been used to monitor, evaluate, and compare software projects and their evolution. However, when we tried to apply DL to industrial projects, the developers complained about the instability of DL in the presence of isolated files. Additionally, they were confused about why two DL scores were provided by DV8 (an "all" score and a "non-isolated" score) and how to interpret them. These issues motivated the creation of M-score.

8 Conclusion

In this paper, we proposed a software modularity metric called M-score that leverages the advantages of Decoupling Level (DL) and Propagation Cost (PC), and addresses their problems, especially their stability problems in the presence of isolated files or components. M-score offers four major contributions: (1) aggregating modularity scores for each module so that a system with more independent modules will have a better M-score; (2) accounting for dependency density between modules to reward low coupling among modules; (3) accounting for inner-module dependency density to reward low-coupling within each module; and (4) applying a size penalty uniformly to all modules so that isolated files will not have a drastic effect on the overall score. As a result, a system with low coupling both within and among modules and with more small, independent modules will have a higher M-score.

After analyzing 1220 releases of 37 projects, we have demonstrated that: M-score is significantly more stable than DL, significantly correlated with maintenance difficulty, and preserves the most important properties of being a "*metric*". That is, it can be used to compare different projects or monitor the evolution of a single project. The implication is that M-score has the potential to serve as an effective and reliable modularity metric to monitor the evolution of a single project, and also to serve as the foundation of an industrial modularity benchmark.

9 Data Availability

We provide all the data in tables, charts, and figures as well as release dependency files in a replication package at [36].

References

[1] [n. d.]. Depends: Multi-language Source Code Dependency Extraction Tool. https://github.com/multilang-depends/depends.

[2] [n. d.]. Understand: Take Control of Your Code. https://scitools.com/.

[3] Saleh Almugrin, Waleed Albattah, and Austin Melton. 2016. Using indirect coupling metrics to predict package maintainability and testability. *Journal of systems and software* 121 (2016), 298–310.

[4] T. W. Anderson and D. A. Darling. 1952. Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes. *The Annals of Mathematical Statistics* 23, 2 (1952), 193 – 212. https://doi.org/10.1214/aoms/1177729437

[5] Carliss Y. Baldwin and Kim B. Clark. 2000. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press.

[6] Dennis Bijlsma, Miguel Alexandre Ferreira, Bart Luijten, and Joost Visser. 2012. Faster Issue Resolution with Higher Technical Quality of Software. *j-sqj* 20, 2 (June 2012), 265–285.

[7] M. Host M. Ohlsson B. Regnell C. Wohlin, P. Runeson and A. Wesslen. 2012. *Experimentation in Software Engineering*.

[8] Yuanfang Cai and Rick Kazman. 2019. DV8: Automated Architecture Analysis Tool Suites. In *Proceedings of the Second International Conference on Technical Debt* (Montreal, Quebec, Canada) *(TechDebt '19)*. IEEE Press, 53–54. https://doi.org/10.1109/TechDebt.2019.00015

[9] Shyam R. Chidamber and Chris F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (June 1994), 476–493.

[10] Michel-Daniel Cojocaru, Alexandru Uta, and Ana-Maria Oprescu. 2019. Attributes assessing the quality of microservices automatically decomposed from monolithic applications. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 84–93.

[11] Istvan Gergely Czibula, Gabriela Czibula, Diana-Lucia Miholca, and Zsuzsanna Onet-Marian. 2019. An aggregated coupling measure for the analysis of object-oriented software systems. *Journal of Systems and Software* 148 (2019), 1–20.

[12] Fernando Brito e Abreu. 1995. The MOOD Metrics Set. In *Proc. ECOOP'95 Workshop on Metrics*.

[13] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. 2018. Evaluation of microservice architectures: A metric and tool-based approach. In *Information Systems in the Big Data Era: CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings 30*. Springer, 74–89.

[14] Xiaoqin Fu and Haipeng Cai. 2019. Measuring interprocess communications in distributed systems. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 323–334.

[15] Daniel Galin. 2004. *Software quality assurance: from theory to implementation*. Pearson education.

[16] Fernand Gobet and Gary Clarkson. 2004. Chunks in expert memory: evidence for the magical number four ... or is it two? *Memory* 12, 6 (Nov. 2004), 732–47.

[17] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc.

[18] R. Harrison, S. J. Counsell, and R. V. Nithi. 1998. An Investigation into the Applicability and Validity of Object-Oriented Design Metrics. *Empirical Software Engineering* 3, 3 (Sept. 1998), 255–273.

[19] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A Practical Model for Measuring Maintainability. In *Proc. 6th International Conference on Quality of Information and Communications Technology*. 30–39.

[20] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andrey Shapochka. 2015. A Case Study in Locating the Architectural Roots of Technical Debt. In *Proc. 37th International Conference on Software Engineering*.

[21] Serkan Kirbas, Alper Sen, Bora Caglayan, Ayse Bener, and Rasim Mahmutogullari. 2014. The effect of evolutionary coupling on software defects: an industrial case study on a legacy system. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–7.

[22] Wei Li and Sallie Henry. 1993. Object-oriented Metrics That Predict Maintainability. *Journal of Systems and Software* 23, 2 (Nov. 1993), 111–122.

[23] Mark Lorenz and Jeff Kidd. 1994. *Object-Oriented Software Metrics*. Prentice Hall. 146 pages.

[24] Alan MacCormack, John Rusnak, and Carliss Baldwin. 2008. *Exploring the Duality between Product and Organizational Architecture: A Test of the Mirroring Hypothesis*. Working Paper 08-039. Harvard Business School. http://www.hbs.edu/research/pdf/08-039.pdf.

[25] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science* 52, 7 (July 2006), 1015–1030.

[26] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* 2, 4 (Dec. 1976), 308–320.

[27] George A. Miller. 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review* 63, 2 (1956), 81–97.

[28] Subhas Chandra Misra. 2005. Modeling Design/Coding Factors That Drive Maintainability of Software Systems. *Software Quality Control* 13, 3 (Sept. 2005), 297–320.

[29] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2016. Decoupling Level: A New Metric for Architectural Maintenance Complexity. In *Proc. 38rd International Conference on Software Engineering*.

[30] Ran Mo, Will Snipes, Yuanfang Cai, Srini Ramaswamy, Rick Kazman, and Martin Naedele. 2018. Experiences Applying Automated Architecture Analysis Tool Suites. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 779–789. https://doi.org/10.1145/3238147.3240467

[31] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proc. 28th International Conference on Software Engineering*. 452–461.

[32] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and F. Chew. 2019. A Longitudinal Study of Identifying and Paying Down Architecture Debt. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 171–180. https://doi.org/10.1109/ICSE-SEIP.2019.00026

[33] Paul Oman and Jack Hagemeister. 1994. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software* 24, 3 (1994), 251–266.

[34] Michail D Papamichail and Andreas L Symeonidis. 2020. A generic methodology for early identification of non-maintainable source code components through analysis of software releases. *Information and Software Technology* 118 (2020), 106218.

[35] Mikhail Perepletchikov, Caspar Ryan, and Keith Frampton. 2007. Cohesion metrics for predicting maintainability of service-oriented software. In *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE, 328–335.

[36] Ernst Pisch, Yuanfang Cai, Jason Lefever, Rick Kazman, and Hongzhou Fang. 2024. *M-Score: An Empirically Derived Software Modularity Metric*. https://doi.org/10.5281/zenodo.12747463

[37] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen, et al. 1991. *Object-oriented modeling and design*. Vol. 199. Prentice-hall Englewood Cliffs, NJ.

[38] Richard W. Selby and Victor R. Basili. 1991. Analyzing Error-Prone System Structure. *IEEE Transactions on Software Engineering* 17, 2 (Feb. 1991), 141–152.

[39] SA SonarSource. 2013. SonarQube. *Capturado em: http://www. sonarqube. org* (2013).

[40] Margaret-Anne Storey, Neil A. Ernst, Courtney Williams, and Eirini Kalliamvakou. 2020. The who, what, how of software engineering research: a socio-technical framework. *Empirical Softw. Engg.* 25, 5 (sep 2020), 4097–4129. https://doi.org/10.1007/s10664-020-09858-z

[41] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. 2001. The Structure and Value of Modularity in Software Design. In *Proc. Joint 8th European Conference on Software Engineering and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. 99–108.

[42] Menzies Tim, Butcher Andrew, Marcus Andrian, Zimmermann Thomas, and Cok David. 2011. Local vs. Global Models for Effort Estimation and Defect Prediction. In *Proc. 26thIEEE/ACM International Conference on Automated Software Engineering*. 343–351.

[43] Melanie P Ware, F G Wilkie, and Mary Shapcott. 2007. The Application of Product Measures in Directing Software Maintenance Activity. *Journal of Software Maintenance* 19, 2 (March 2007), 133–154.

[44] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software* 182 (2021), 111061.

[45] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. 2009. Design Rule Hierarchies and Parallelism in Software Development Tasks. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*. 197–208.

[46] Wensheng Wu, Yuanfang Cai, Rick Kazman, Ran Mo, Zhipeng Liu, Rongbiao Chen, Yingan Ge, Weicai Liu, and Junhui Zhang. 2018. *Software Architecture Measurement—Experiences from a Multinational Company: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings*. 303–319. https://doi.org/10.1007/978-3-030-00761-4_20

[47] Chenxing Zhong, He Zhang, Chao Li, Huang Huang, and Daniel Feitosa. 2023. On measuring coupling between microservices. *Journal of Systems and Software* 200 (2023), 111670. https://doi.org/10.1016/j.jss.2023.111670