2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Understanding Architectural Complexity, Maintenance Burden, and Developer Sentiment —a Large-Scale Study

Yuanfang Cai
Drexel University, USA
Email: yuanfang.cai@drexel.edu

Lanting He, Jun Qian, Yony Kochinski, Nan Zhang, Ciera Jaspan, Antonio Bianco
Google LLC, USA
Emails: {lantinghe, juqian, yonyk, nanzh, ciera, abianco}@google.com

***Abstract*—Intuitively, the more complex a software system is, the harder it is to maintain. Statistically, it is not clear which complexity metrics correlate with maintenance effort; in fact, it is not even clear how to objectively measure *maintenance burden*, so that developers' sentiment and intuition can be supported by numbers. Without effective complexity and maintenance metrics, it remains difficult to objectively monitor maintenance, control complexity, or justify refactoring. In this paper, we report a large-scale study of 1252 projects written in C++ and Java from Google LLC. We collected three categories of metrics: (1) architectural complexity, measured using propagation cost (PC), decoupling level (DL), and structural anti-patterns; (2) maintenance activity, measured using the number of changes, lines of code (LOC) written, and active coding time (ACT) spent on feature-addition vs. bug-fixing, and (3) developer sentiment on complexity and productivity, collected from 7200 survey responses. We statistically analyzed the correlations among these metrics and obtained significant evidence of the following findings: 1) the more complex the architecture is (higher propagation cost, more instances of anti-patterns), the more LOC is spent on bug-fixing, rather than adding new features; 2) developers who commit more changes for features, spend more lines of code on features, or spend more time on features also feel that they are less hindered by technical debt and complexity. To the best of our knowledge, this is the first large-scale empirical study establishing the statistical correlation among architectural complexity, maintenance activity, and developer sentiment. The implication is that, instead of solely relying upon developer sentiment and intuition to detect degraded structure or increased burden to evolve, it is possible to objectively and continuously measure and monitor architectural complexity and maintenance difficulty, increasing feature delivery efficiency by reducing architectural complexity and anti-patterns.**

Software Design, Software Metrics, Software Maintenance

I. Introduction

Intuitively, the more complex a software system is, the harder it is to maintain. However, it is not clear which complexity metrics correlate with maintenance burden, especially developers' ability to add new features. In the past five decades, numerous metrics have been proposed to measure code complexity [1]–[3], from lines of code (LOC), fan-in, fan-out, to McCabe's Cyclomatic Complexity [3] and Chidamber & Kemerer's coupling-cohesion metrics suite [4]. These metrics have been used for various software analyses, such as defect prediction and localization [5]–[9], but there is no empirical evidence on how well these metrics are correlated with developers' ability to deliver features or their sentiment. Moreover, these well-known metrics measure *individual files*, rather than the *architectural complexity* rooted in the dependencies among files and components.

MacCormack et al. proposed a new metric, *propagation cost* (PC) [10], to measure how tightly source files are coupled, calculated using both direct and indirect dependencies. This metric has been successfully used to compare different projects of similar sizes. Mo et al. proposed *decoupling level* (DL) [11], another metric assessing how files are decoupled into independent modules based on options theory [12]. Mo et al. also proposed a suite of architecture anti-patterns [13], [14] that capture file and component structures that violate design principles, including Clique, Unhealthy Inheritance, and Package Cycle.

While prior studies have shown that these anti-patterns are linked to high maintenance costs [13]–[17], in daily software development practice, it remains unclear when teams should pause to refactor and reduce complexity. Moreover, it is uncertain whether removing these anti-patterns or improving these modularity scores directly enhances a team's ability to deliver features, a critical productivity metric for software organizations. In this study, we measure the effort required for developers to add new features compared to fixing bugs and examine whether these productivity metrics correlate with architecture complexity and developer sentiment on complexity-related technical debt. If such correlations exist, organizations should systematically and continuously collect this data, rather than relying solely on developers' intuition, to identify when a decline in feature-adding capability signals the need for refactoring, and guide refactoring strategies to reduce complexity, ultimately enhancing productivity and reducing maintenance costs.

Research on open-source projects often proximate productivity and quality using data mined from the revision histories of the subject projects, including: *change frequency*—the number of changes in a unit time range; *bug frequency*—the number of bugs closed in a unit time range; *change churn*—the average number of LOC spent on changes; and *bug churn*—the average number of LOC spent on bug-fixing. Using these measures, researchers have revealed that files involved in anti-patterns—defined as dependency structures that violate design principles, are more error-prone and change-prone [11], [13], [18]. These four measures, however, do not directly measure development *effort*, such as time and LOC spent on bug fixing versus developing new features.

Delivering features quickly is critical to most software companies, but change frequency (or change throughput) alone does not represent a team's capability to add features: change throughput could be high due to intensive bug fixing or migrations. A more productive team should be able to spend relatively more effort on accommodating requirement changes/adding new features [19]. There is no research investigating how architectural complexity influences feature-addition capabilities. Without such a measure, the only way to distinguish between codebases that enable high velocity from those that do not is to ask developers directly, a costly option that is difficult to use for monitoring and to justify refactorings.

In this paper, we report on a large scale study to answer the following two research questions:

Q1: *Can developers' effort of adding features vs. fixing bugs reflect their sentiment of complexity in their codebase?*

Q2: *Does their feature vs. bug activity ratio correlate with architectural complexity?*

The first question explores whether these recorded maintenance activities are consistent with developers' sentiment about being hindered by complexity; if they are, we can use these measures instead of relying on periodic sentiment survey data to assess complexity, which is a lagging indicator and expensive to collect. Our second objective is to assess whether architectural complexity metrics are correlated with maintenance burden as recorded in the revision history; if so, it indicates that those architectural complexity metrics can be used to indicate opportunities for refactoring toward more efficient feature delivery.

To answer these questions, we collect the following three categories of metrics from 1252 projects and 7,200 developers in Google LLC:

1) *Developer sentiment* about the extent to which complexity is hindering the developer's productivity, measured through survey responses (Section III-A);
2) *Maintenance activity*, measured using the number of change lists (#CL), lines of code spent (#LOC), and active coding time (ACT) spent on feature-addition vs. bug-fixing (Section III-B);
3) *Architectural complexity*, measured using propagation cost (PC), decoupling level (DL), and three structural anti-patterns calculated using a tool called DV8 [20] (Section III-C).

Our statistical analysis (Section IV-E) reveals significant evidence that:

- Developers who spent more effort on features compared to bugs also feel that they are less hindered by technical debt and complexity.
- Systems with higher architectural complexity, less modularity, or more instances of design anti-patterns, especially cycles among files and improper usage of inheritance, are correlated with more development effort on bug-fixing rather than feature-addition.

To the best of our knowledge, this is the first large-scale study establishing the statistical correlation among architectural complexity metrics, maintenance activity metrics, and developers' sentiment reflected in survey responses.

The implication is that, instead of relying upon developer sentiment and intuition, it is possible to objectively and continuously measure, monitor, and hence control/improve, both the architectural complexity, and the maintenance burden, especially the ability to add new features. The detected anti-patterns can be used to pinpoint specific files that need refactoring, and provide guidance on how to reduce architectural complexity, improve feature productivity, which in turn, increase developers' satisfaction.

II. Background

The study is based on the rich experiences of technical debt management and code quality measurement within Google LLC, and state-of-the-art research results.

Since 2018, Google LLC conducts quarterly anonymous survey to collect developers' opinions on technical debt (TD), and to what extent this TD have impacted their productivity. Each quarter, the survey is distributed to a rotating set of developers, and each developer gets invited to take the survey every 9 months. While this survey can provide high level indicators of technical debt, the sampling method (and using a survey at all) means that the survey results cannot be used for continuous monitoring of TD on smaller teams. While Google LLC has evaluated many per-file metrics, such as cyclomatic complexity and cognitive complexity, their internal studies have found that these do not correspond with developer perceptions of technical debt or code quality and they are therefore not heavily used. Google LLC has not evaluated metrics of the complexity across source files.

Academic research has proposed new complexity metrics and technical debt detection methods that could be used in a more "always on" fashion to allow regular monitoring of technical debt. In particular, propagation cost (PC) [10] and decoupling level (DL) [11] are two state-of-the-art metrics measuring the architectural complexity and modularity formed by file dependencies. As the counterpart of the widely studied code smells [21]–[24] (e.g., cloned code, God class, spaghetti code) that focus on individual files, a suite of design anti-patterns (e.g., cliques, unhealthy inheritance) based on dependency structures were proposed [13], [18] to detect problematic dependency structures among files. These file-dependency based measures, PC, DL, and anti-patterns, supported by the DV8 tool [20], have been applied and validated using a large number of open-source and industrial projects [11], [14]–[17], [19], [25], [26].

Prior research reveals that, if a system has low DL/high PC scores, it can be hard to maintain [10], [11], [14]. Files associated with more anti-patterns are more error-prone and change-prone, measured using change frequency, bug frequency, change churn, and bug churn [11], [13], [14], [26]–

[28]. A recent longitudinal case study revealed that after reducing anti-patterns through refactoring, the maintainability score improved, and the LOC/days needed to fix bugs and make changes significantly reduced [15].

Using change frequency/churn and bug frequency/churn alone, however, cannot fully capture the severity of maintenance burden: increased change frequency/churn could be the result of frequent bug fixing, a consequence of degraded quality, rather than improved maintainability. Moreover, change frequency and churn do not directly measure the most critical aspect of productivity: adding new features, nor do they reflect an important aspect of "*effort*": coding time spent on maintenance activities. Given this background, in this study, we merge Google LLC's internal research with latest academic advances to investigate effective and objective complexity and maintenance burden measures that are consistent with developers' perceptions.

III. Three Categories of Metrics

In this section, we introduce the three types of metrics collected from three independent data sources: developers' sentiment collected from periodic surveys, maintenance activities extracted from revision history in the form of activity logs, and architectural complexity obtained from static analysis.

A. Developer Sentiment

To collect developers' sentiment toward code quality, complexity, and technical debt, we leverage the responses collected from an existing developer satisfaction survey. This survey has been run quarterly within the company for several years. A rotating one third of developers take the survey each quarter, so that every developer gets the survey every 9 months. The response rate each quarter is about 33%. The survey has many questions, and each participant only needs to answer some of them. For this study, we focus on the following question, to which all participants are required to answer:

"*In the last three months, how much has technical debt or overly complicated code inside your project hindered your productivity?*". Among all the survey questions designed by Google experts, this is the only one that directly addresses technical debt and complexity, making it the most relevant to our study. Respondents can choose from "*Extremely hindered*", "*Very much hindered*", "*Moderately hindered*", "*Slightly hindered*", or "*Not at all hindered*". The responses are re-coded on a scale from 1 to 5, where a lower score indicates that the developer was more hindered by technical debt or code complexity. The question is specifically worded to ask how much a developer is *hindered* by technical debt, rather than asking whether the developer has *encountered* technical debt, as most developers encounter technical debt regularly (and indeed, a certain amount of incurred debt can be a prudent decision [21]).

B. Maintenance Activity

To measure "maintenance activity", the unique concept we took is to measure the proportion of *feature development* effort relative to the total spent on *feature development and bug fixing*. The idea is to capture how much "overhead" is being taken up with fixing bugs, an indicator of maintenance costs.

Within Google LLC, development activities are centered around a *changelist* (CL); a set of code changes that are submitted together. Changelists may be associated with *issues* in the issue tracker, and those issues can be categorized as a *bug*, a *feature request*, or one of the other 13 types defined in the issue tracker. We found that 75% of all CLs were associated with an issue, and of those, 60% of the CLs could be labeled as either "feature request" or "bug". The remaining 40% were associated with the other 13 issue types, including "process", "vulnerability", or "milestone", these other 13 types are used inconsistently across the Google LLC though and so are removed from our analyses.

We then defined three metrics of development effort:

1) Number of changelists `(CL)`,
2) Lines of code `(LOC)`, including added, modified, and deleted lines in the CLs,
3) Active coding time (`ACT`), capturing the "*fingers on keyboard*" time spent on coding, including time spent on editing code, looking up relevant documentation, and debugging test results, among other activities. This metric is described and validated in [29].

For each of these metrics, we then calculate the ratio of effort spent on `CL/LOC/ACT` associated with only feature requests to the `CL/LOC/ACT` associated with either feature requests or bugs. This gives us a measure of "maintenance burden": a ratio of 1.0 means there is no maintenance burden, while 0.7 means developers are spending 30% of their "effort" on bug fixing, measured using `CL`, `LOC`, or `ACT`.

If a system is getting more and more complex, and the complexity starts to impact team productivity, it is common to observe that the team has to reduce their feature delivery speed and spend more time fixing bugs. Figure 1 depicts the feature-over-bug ratio measured using LOC and ACT respectively from a sample project to illustrate the increased maintenance burden: at the beginning of the project, around 2019, most coding time and LOC were spent on new features; starting from the middle of 2020, the LOC and ACT spent on bug fixing exceeded that of feature addition. The blue lines (feature LOC/ACT) keep decreasing, indicating reduced feature delivery velocity and increased maintenance burden.

Note that the changing ratio of feature-over-bug as shown in Figure 1 can be caused by other reasons than increased complexity. It is possible that a product grows to have more users and hence more bugs are reported, or the features of a project are all implemented, and the project is transforming from focusing on features to focusing on bug-fixing. In this case, the trends in Figure 1 could be the reflection of a natural life cycle of a project. In our study, we didn't conduct a longitudinal study to track the lifecycle of a product, or look for a perfect causality from architectural complexity scores to feature-over-bug ratio. Instead, we test the hypothesis that higher architectural complexity is correlated with more bug-fixing effort, across all 1252 projects. If this hypothesis is true, it means that for a project in its bug-heavy stage, it tends to have higher architectural complexity; on the other hand, if a project is still in the feature-heavy stage, high architectural complexity can become a hindering technical debt.

In this project, we measure maintenance activities from two perspectives: the activities of each individual developer, and the maintenance activity of each individual project. For each perspective, we propose three metrics:

Developer activity metrics. To measure the maintenance activity of a developer, we collected the total number of CL, LOC, and ACT committed and spent by each developer. As we are comparing these values against quarterly survey data, we calculate the total counts of each measure across the entire quarter, using the three metrics as listed in Table I.

Project activity metrics. A *project* is represented by one or more file paths containing all the files involved in a project. For a project, we are similarly interested in the LOC, ACT, and CL spent on it in a given time range. For each project, we first collected the weekly totals of CL, LOC, and ACT spent on features and bugs, respectively. We then calculated the ratio of these metrics spent on features versus the total spent on both features and bugs. Finally, we used the median of these weekly metrics over the past six months to represent the project's maintenance activity. For example, "*In project Foo, the* `med_feature_bug_loc_ratio` *is .26*" means that the median weekly percentage of LOC spent on adding new features is 26% over all the LOC spent on adding features or fixing bugs. This gives us the 3 metrics shown in Table II. Unlike metrics commonly used in prior studies—such as bug frequency, change frequency, bug churn, and change churn—the feature-to-bug ratio is less influenced by the number of developers or commits. Instead, it directly reflects the portion of effort that can be spent on features.

C. Architectural Complexity

As we briefly introduced in Section II, in this study, we adopt two state-of-the-art metrics and a suite of structural anti-patterns to assess architectural complexity formed by dependency among files. These metrics and anti-patterns are all based on a dependency model called the *design structure matrix* (DSM) [12], [30]–[32].

Figure 2 and 3 depict four DSMs[1] that are used to illustrate these concepts. In a DSM, the columns and rows are labeled with the same set of files or packages of the same order, and a mark in row i, column j indicates that the item on row i depends on the item on column j. In Figure 2, the columns and rows are labeled with package names. The numbers on the first row, first column, and along the diagonal are indexes of these 15 packages. The numbers in the cell indicate the number of dependencies among the files within each package. For example, the number "86" in row 4, column 3 indicates that there are 86 dependencies from the files in the $clients$ folder to the files in the sdk folder.

[1]All the DSMs displayed in this paper are generated from open source projects for illustration purposes, not from proprietary Google LLC projects.

In Figure 3, the three DSMs are labeled with file names. Consider Figure 3c as an example: this DSM presents the dependency structure among these 9 files. Row 2, column 1 is labeled with "1", meaning that $TopicName.java$ has one dependency on $ServiceUnitID.java$. The blocks along the diagonal are file group clusters: Figure 3c depicts 3 file groups, formed by File 1, File 2-4, and File 5-9 respectively.

Definitions. Using the DSM model, architectural metrics and anti-patterns are defined as follows:

(1) *Propagation Cost (PC),* measuring to what extent files are coupled with each other [10]. To calculate PC, we first represent the dependencies among files using a DSM, and then calculate its maximum transitive closure to account for all the direct and indirect dependencies. The PC is calculated as the total number of direct and indirect dependencies divided by the total number of cells in the DSM. The larger the PC, the more coupled the files are.

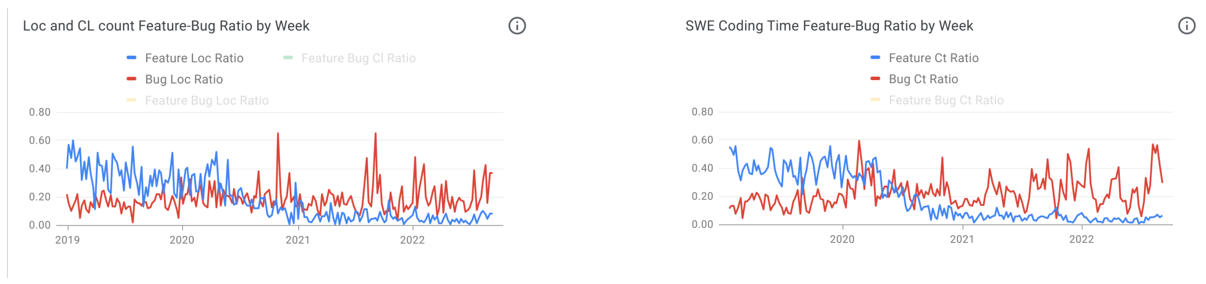
(2) *Decoupling Level (DL),* measuring to what extent file modules can evolve independently [11]. To calculate DL, we first reorder a DSM into a hierarchical structure [33], in which each layer contains a number of mutually-independent modules. Using this hierarchical structure, DL is calculated based on the following rationales: the more modules are truly independent, the higher the DL; the larger a module, and smaller the DL; the more dependencies between layers, the lower the DL. These new metrics have been applied and validated in multiple case studies [14], [15], [17].

(3) *Structural Anti-patterns.* Mo et al. proposed 6 anti-patterns [13], [18], defined based on the violations of design principles. Three of these anti-patterns, *Unstable Interface*, *Crossing*, and *Modularity Violation* are defined using both structural dependencies and evolutionary dependencies [13], [18], that is, the co-change frequency between two files. We do not consider these history-based anti-patterns in this study, otherwise these data will not be independent from the development activity data source. In addition, different projects have different number of developers, different level of activeness, and it is hard to form a fair comparison.

Here we only consider anti-patterns formed by structural dependencies only, including:

Cliques: files that form a strongly connected component. That is, changing one of these files may impact other files within the same clique. It is widely accepted that cyclical dependencies should be avoided [23], [34]–[38]. Figure 3a depicts an example of Clique, in which changes to any of these 14 files may propagate to all the other files through direct or indirect dependencies.

Package Cycles: pairs of folders that depend on each other. Ideally namespaces/packages/folders should form a hierarchical structure to ease extension and contraction [36], [38]. Such a hierarchical structure can be ordered into a lower-triangle form in a DSM, that is, no dependency above the diagonal as depicted in Figure 2. Cycles among packages make it impossible to form such a hierarchical structure. Figure 3b depicts a package cycles between the package containing files 1-3 and the package containing files 4-14. These marked



(a) Feature vs. Bug LOC ratio

(b) Feature vs. Bug ACT ratio

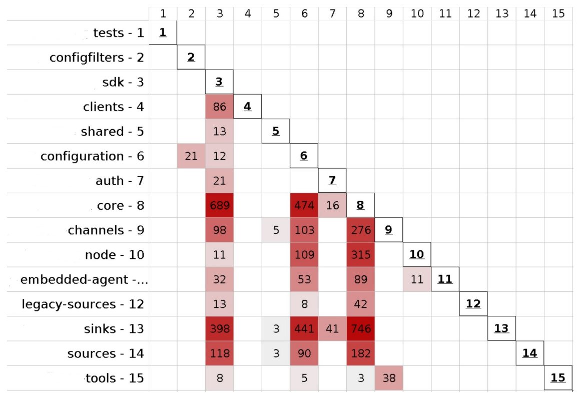
Fig. 1: Maintenance Activity Trend of a Sample Project

TABLE I: Developer Feature-Bug Activity Metrics

Name	**Definition** (Time Unit: Quarter)
`dev_feature_bug_cl_count_ratio`	Total FEATURE CLs / (Total FEATURE CLs + Total BUG CLs)
`dev_feature_bug_loc_ratio`	Total FEATURE LOC / (Total FEATURE LOC + Total BUG LOC)
`dev_feature_bug_coding_time_ratio`	Total FEATURE ACT / (Total FEATURE ACT + Total BUG ACT)

TABLE II: Project Feature-Bug Activity Metrics

Name	**Definition** (Time Span: 6 Months)
`med_feature_bug_cls_ratio`	Median (Weekly FEATURE CLs / (Weekly FEATURE CLs + Weekly BUG CLs))
`med_feature_bug_loc_ratio`	Median (Weekly FEATURE LOC / (Weekly FEATURE LOC + Weekly BUG LOC))
`med_feature_bug_act_ratio`	Median (Weekly FEATURE ACT / (Weekly FEATURE ACT + Weekly BUG ACT))



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
tests - 1	1														
configfilters - 2		2													
sdk - 3			3												
clients - 4			86	4											
shared - 5			13		5										
configuration - 6		21	12			6									
auth - 7			21				7								
core - 8			689			474	16	8							
channels - 9			98		5	103		276	9						
node - 10			11			109		315		10					
embedded-agent -...			32			53		89		11	11				
legacy-sources - 12			13			8		42				12			
sinks - 13			398		3	441	41	746					13		
sources - 14			118		3	90		182						14	
tools - 15			8			5		3	38						15

Fig. 2: Hierarchical Structure of a Sample Project

symmetric cells, such as cell (r3,c10) and cell (r10,c3) indicate that changes to one package may propagate to the other.

Unhealthy Inheritance: here we detect two cases that violate Liskov Substitution Principle [39]: either a parent class depends on one or more of its subclasses, or a client of an inheritance hierarchy uses both the parent class and one or more of its subclasses. In both cases, the parent class does not serve as an interface that decouples clients from subclasses, and does not support polymorphism or run-time substitution. Figure 3c depicts a groups of files that violate both rules: the parent class $ServiceUnitId.java$ depends on two of its subclasses $TopicName.java$ and $NamespaceName.java$. In the meantime, the clients of this inheritance family, files 5 to 9, all depend on the parent class as well as one or more of its subclasses.

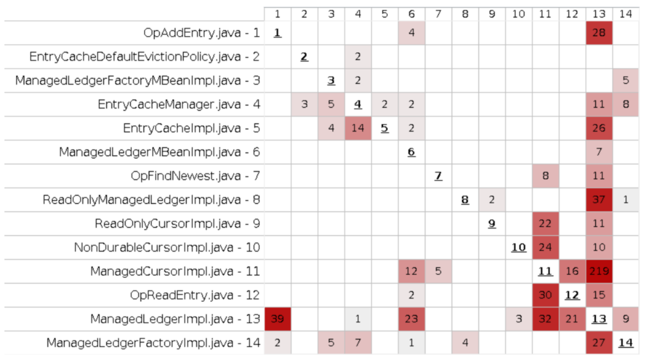
Architectural Complexity Metric Suite. In addition to PC and DL that measure the overall modularity structure, we also derive a number of architectural complexity metrics based on anti-patterns because prior work has confirmed that these anti-patterns often reflect real technical debt [14]–[16]. For example, if a system has more cliques, it is more likely to be more complex and incur higher maintenance burdens.

TABLE III: The Influence of Size

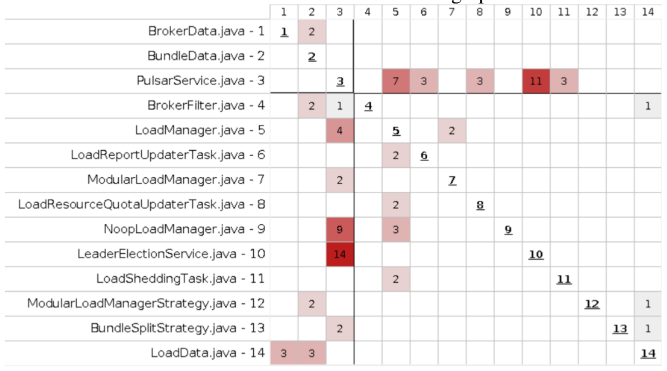
	Cpp		
	`#Clique`	`#UnhInh`	`#PkgClc`
`#Files`	0.3222	0.6703	0.7677
	Java		
	`#Clique`	`#UnhInh`	`#PkgClc`
`#Files`	0.5558	0.6525	0.6160

Note that the size of a project, measured using the number of files (`#File`), or the total LOC count (`#LOC`), not only reflect an important aspect of architectural complexity, but also have significant impact on anti-patterns: it is intuitive that the larger the projects, i.e., the more files are there, and the more anti-patterns can be found. To assess the impact of size, we summarize the correlations between file counts and the number of detected anti-pattern instances in Table III for C++ and Java projects respectively. All these correlations have significant p-values less than 0.01. The results revealed that project sizes have significant impact on the number of anti-pattern instances.

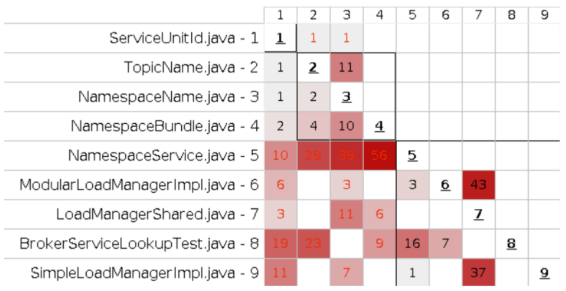
To offset the impact of size, in our study, we measure the *density* of each type of structural anti-patterns, using both `#Files` and `#LOC` as the denominator respectively. For example, to assess the density of Cliques, we measure both `#Clique/#File` and `#Clique/#LOC`. Table IV summarizes all the 10 architectural complexity metrics we use in this


OpAddEntry.java - 1
EntryCacheDefaultEvictionPolicy.java - 2
ManagedLedgerFactoryMBeanImpl.java - 3
EntryCacheManager.java - 4
EntryCacheImpl.java - 5
ManagedLedgerMBeanImpl.java - 6
OpFindNewest.java - 7
ReadOnlyManagedLedgerImpl.java - 8
ReadOnlyCursorImpl.java - 9
NonDurableCursorImpl.java - 10
ManagedCursorImpl.java - 11
OpReadEntry.java - 12
ManagedLedgerImpl.java - 13
ManagedLedgerFactoryImpl.java - 14

(a) Clique
All 14 files form a connected graph


BrokerData.java - 1
BundleData.java - 2
PulsarService.java - 3
BrokerFilter.java - 4
LoadManager.java - 5
LoadReportUpdaterTask.java - 6
ModularLoadManager.java - 7
LoadResourceQuotaUpdaterTask.java - 8
NoopLoadManager.java - 9
LeaderElectionService.java - 10
LoadSheddingTask.java - 11
ModularLoadManagerStrategy.java - 12
BundleSplitStrategy.java - 13
LoadData.java - 14

(b) Package Cycle
Files in both packages depend on each other


ServiceUnitId.java - 1
TopicName.java - 2
NamespaceName.java - 3
NamespaceBundle.java - 4
NamespaceService.java - 5
ModularLoadManagerImpl.java - 6
LoadManagerShared.java - 7
BrokerServiceLookupTest.java - 8
SimpleLoadManagerImpl.java - 9

(c) Unhealthy Inheritance
(1) Parent class $ServiceUnitId$ depends on its two subclasses;
(2) The clients of this inheritance family all depend on
both the parent class and the subclasses

Fig. 3: Design Structure Matrices and Anti-patterns.
Note: all these DSMs are from open source projects for illustration purposes.

study. Using this metric suite, we plan to assess which ones can better reflect maintenance burden.

D. Metrics Summary

To summarize, in this study, we assess developer sentiment, maintenance activity, and architectural complexity, using four sets of metrics:
M1: developer sentiment score,
M2: developer activity metrics (Table I),
M3: project activity metrics (Table II), and
M4: architectural complexity metrics (Table IV).

TABLE IV: Architectural Complexity Metrics Suite

Name	Definition
`#File`	The total number of files in a project.
`#LOC`	The total number of lines of code in a project.
`PC`	The propagation cost score calculated from the project DSM.
`DL`	The decoupling level score calculated from the project DSM.
`#Clique/#File`	Clique density per file.
`#Clique/#LOC`	Clique density per line of code.
`#PkgClc/#File`	Package Cycle density per file.
`#PkgClc/#LOC`	Package Cycle density per line of code.
`#UnhInh/#File`	Unhealthy Inheritance density per file.
`#UnhInh/#LOC`	Unhealthy Inheritance density per line of code.

Our objective is to assess the relation between developers' activities (**M2**) and their sentiment (**M1**) to answer Q1, and between architectural complexity (**M4**) and project activity (**M3**) to answer Q2. As we will elaborate in the next section, (**M1**) was extracted from developer surveys, and (**M4**) was calculated from source code snapshots. Both (**M2**) and (**M3**) are extracted from the revision history of Google projects, but from different time periods to make fair comparisons with developer sentiment and architectural complexity respectively.

IV. Methodology

Given the definition of these three categories of metrics, in this section, we elaborate the methodology of our study: project selection, data collection, and statistical analyses that answer the two research questions proposed in Section I.

Here we elaborate how we collected data for the four sets of metrics, **M1**, **M2**, **M3**, and **M4** respectively.

A. M1: Developer Sentiment Data Collection

The developer sentiment data were extracted from employee survey collected quarterly from all developers of Google LLC. We started with the data for all respondents from the prior 12 quarters. However, as we plan to do a panel analysis, we need to have data from developers who took the survey at least twice in this time period; this gave us 7200 developers who responded at least twice. Of the responses collected each quarter, about 55% indicated "not at all hindered".

B. M2: Developer Activity Data Collection

For each developer and quarter that we have survey data for, we collected their maintenance activity data from the prior 12 quarters as listed in Table I. This data is used to conduct a panel analysis [40] to investigate if and how developers' sentiment changes with the maintenance activities they perform.

C. M3: Project Activity Data Collection

To answer Q2, we need data at the project level. Our goal was to select non-trivial and active projects from the code repository. Therefore, we collected all projects which met the following criteria:

1) The project should have least 500 Java files or 500 C++ files. We selected these languages since they are the ones for which we can calculate the dependency metrics.
2) The project should have had changes submitted by at least 20 developers in the prior 12 months.

These criteria ensure that these projects are non-trivial, long-lasting, and being actively developed or maintained by a team with considerable number of developers. We obtained 1252 projects as the subjects for our study: 642 Java projects and 610 C++ projects.

To assess the relation between architectural complexity and maintenance activity, we measure the architectural complexity of the latest version for each of the 1252 projects (see the next subsection), and collected project activity metrics, as listed in Table II, spanning 6 months prior to the snapshot was scanned, and use their median values as the final project activity scores. For example, if we collected Project_A's architectural complexity metrics in April 2022, then we collect and aggregate its activity data from Nov 2021 to April 2022.

D. M4: Architectural Complexity Data Collection

For each project, we collected the 10 architectural complexity metrics as listed in Table IV, and the 3 activity metrics as listed in Table II. These total 30 pairs of variables are used to conduct architectural complexity vs. project activity correlation analysis. For each pair of variables, e.g., `PC` vs. `med_feature_bug_loc_ratio`, we calculate their correlation from all 1252 projects, including 642 Java projects and 610 C++ projects. As we mentioned above, the architectural complexity metrics are extracted from the latest version of each of the 1252 projects. Ideally, we would like to collect longitudinal data for each project over a longer period of time, as we did for the developers' sentiment data. However, given the way projects are managed within Google, there is no straightforward method to extract snapshots in history like we can do for projects managed in Github. But it is possible to periodically collect architectural complexity data using our workflow, which is our future work.

Different from the activity data that were collected by querying corresponding data tables, all the architectural complexity metrics are calculated from each project's codebase, using a workflow with the following steps:

1. Extract dependencies from source code. Google uses Kythe[2] to index all the dependencies among all files, and our first step is to write scripts to extract dependency information from Kythe, and export it into a standard JSON format. Given the complexity of language syntax, we only have scripts to extract dependencies from Java and C++ projects.

2. Using these JSON-formatted dependency files as input, we use a third-party component, DV8 [14], [20], to calculate PC, DL, and anti-patterns.

3. Using the output of DV8, we created another script to aggregate and calculate the 10 metrics as listed in Table IV, which will be used in our statistical analysis.

[2]https://kythe.io

In summary, to understand how developers' sentiment change with their maintenance activities, we collected longitudinal data (M1 and M2) from the past 12 quarters; to understand if a project with higher architectural complexity also has higher maintenance burden, we collected a snapshot from each of the 1252 project, calculated its architectural complexity (M4), and collected 6-month project maintenance activity data from the activity logs up to the time when the snapshot was extracted (M3).

E. Statistical Analyses

Given the different natures of the three categories of data, we need to employ two different types of statistical analyses:

1) *Panel analysis:* to discover the causal relation between development activities and developer sentiment using longitudinal data;
2) *Regression analysis:* to discover the relation between architectural complexity and maintenance activities using snapshot data.

1) Developer Sentiment and Maintenance Activity: Panel analysis is a statistical method suitable to analyze two-dimensional (typically cross sectional and longitudinal) panel data, that is, data collected over time and over the same individuals. A regression is run over these two dimensions [40]. This method is most suitable to analyze per-developer sentiment and per-developer activity data collected for the past 12 quarters, assessing for an individual developer, whether his/her sentiment changes with their development activities.

Let Y_{it} denote the response of a developer, SWE_i, at time t, X_{it} an activity score, and Z_{it} a set of control variables. Here we use a differential model [41] to eliminate the factors that may have an impact in the developers responses, but do not change over time, such as the developer's highest degree. That is $Z_{it} = Z_i$ for all time t. In this setting, the model we use in this study is as follows:

$$\Delta Y_{it} = \alpha_0 + \alpha_X * \Delta X_{it} + \gamma_i + \epsilon_{it} \quad (1)$$

Here $\Delta Y_{it} = Y_{it} - Y_{i(t-1)}$ is the change in the responses of a developer between consecutive surveys, and $\Delta X_{it} = X_{it} - X_{i(t-1)}$ is the change in an activity measure. Note that a per-developer random effect γ_i is still necessary after differencing, because if a developer has responded to the survey 3 or more times, this developer will still contribute two or more potentially correlated data points to the analysis. ϵ_{it} is a developer and time-point specific residual. Using this model, we can interpret the coefficient for ΔX_{it} as the rate of change of the expected response increment, per unit change in the activity increment. We used the model above to analyze the relation between ΔY_{it}, the variation of a developer's sentiment, and the three developer-based activity measures as listed in Table I.

Table V presents the results from the panel analysis, showing that the developers feel less hindered by complexity and technical debt, if they spent more active coding time (ACT), LOC, and CLs on features. The coefficients are small but the correlation is significant with p-values < 0.05. The developers sentiment could be affected by many factors, and the data reveal strong evidence that different types of development activity they perform do have an impact on their sentiment.

Answer to Q1: Yes, developers report feeling less hindered by complexity and technical debt if they could spend more time adding features, rather than fixing bugs.

2) Architectural Complexity and Maintenance Activity: In this section, we introduce the process and results of our complexity-activity analysis, answering the second research question proposed in Section I.

Regression analysis. To discover the relationship between architectural complexity and maintenance activity, we experimented with quantile regression [42] analysis between 10 (architectural complexity) x 3 (maintenance activity) pairs of metrics extracted from the 1252 projects.

Quantile regression estimates the conditional median of the response variable. We chose quantile regression over ordinary least square regression because it works better with outliers, and we have found projects with much higher or lower than normal architectural complexity scores.

For quantile regression, the null hypothesis is that here's no relationship between maintenance activity and architectural complexity metrics. The p-value tests the null hypothesis that the coefficient is equal to 0. A low p-value indicates there's strong evidence against the null hypothesis, meaning the effect sizes are statistically significant. The results are considered to be significant if p-values are less than 0.05.

Since we are conducting multiple experiments for complexity-activity correlation analysis, we also report *adjusted p-values* that are calculated using Benjamini and Hochberg's multiple hypothesis test correction (FDR_BH) [43]. A small adjusted *p-value* indicate a "strong evidence", a much higher standard than a typical *p-value* alone, with most conservative assumptions. We introduce coding language as a control variable for this regression as the architectural complexity scores of C++ and Java reveal very different characteristics.

Results. Table VI lists the analysis results between all architectural complexity and maintenance activity metric pairs, ranked by their p values. *Activity* measures are expressed as ratios ranging from 0 to 1, and we applied min-max normalization to the *Complexity* measures to scale them to the same range. A coefficient with an absolute value greater than 1 indicates a significant impact of *Complexity*, as a one-unit change in *Complexity* is expected to result in a change of more than one unit in *Activity*. The analysis reveals that there are five significantly correlated metrics pairs (the top 5 rows where $Adj.\ p-value$ < 0.05) :

(1) The density of improper inheritance usage (`UnhInh/File` and `UnhInh /LOC`) are negatively correlated with feature-over-bug ratios, measured using LOC;

(2) The density of cycles (`Clique/LOC`) is also negatively correlated with feature-over-bug ratios, measured using LOC;

(3) PC is negatively correlated with feature-over-bug ratio measured using both LOC and active coding time (ACT).

The implication is that the more coupled the system is (higher PC, more cycles among files), and the higher density of of improper inheritance, the more lines of code has to be spent on bug-fixing, rather than feature-addition. For example, the 3rd row in Table VI reveals that one unit increase in `#Clique/LOC` leads to a -1.7% decrease in feature_loc_ratio.

Note that of all the three types of anti-patterns, Package Cycle does not seem to have significant impact on any of the maintenance activity, which is expected because cycles among packages may not be caused by cycles among files. We also didn't find significant correlation between architectural complexity and CL-based maintenance metrics, which is also expected because the number of CLs are mostly determined by the activeness and the number of users of a project. There is no significant relation between ACT-based metrics and anti-pattern based metrics either, which is not surprising because there are many factors that may impact a developer's coding time. To our surprise, there is no significant correlation between DL and any maintenance activity metrics. As we will discuss in Section VI, this could be due to the widely used "installer" framework within Google.

To verify whether these anti-patterns are real issues from the developers' perspective, we conducted a qualitative analysis of multiple projects, interviewed over a dozen teams at Google, and confirmed that most anti-patterns reported by DV8 represent actual technical debt. These findings will be detailed in a separate paper.

Answer to Q2: Objective architectural complexity metrics have significant correlations with maintenance effort: the more coupled, more file-level cycles, and more problematic inheritance a system has, the less portion of LOC is spent on feature-addition.

V. Implication and Impact

Our study is the first that establishes the relations among architectural complexity, maintenance activity, and developer sentiment metrics, with the following implications:

(1) First of all, the development team should continuously collect these triangulated metrics to determine "*when*" the feature-adding capability is declining (see Figure 1) with increased complexity and anti-patterns. If the team observes both degraded effort on features (increased effort on bugs) and increased architectural complexity, it indicates the need to reduce architectural complexity by removing these anti-patterns to save maintenance costs. Complex code does not grow overnight. Instead, as Cunningham [44] pointed out, complexity and technical debt emerge and accumulate when shortcuts, compromises, or suboptimal design decisions were made to expedite the delivery of features. To mitigate the problem, a designer should be able to recognize when design debt occurs, for example, by detecting the emergence

TABLE V: Maintenance Activity and Developer Sentiment

Activity	Coefficient	Intercept	Std. Err.	p-value
dev_feature_bug_loc_ratio	0.081	0.044	0.025	0.001
dev_feature_bug_cl_count_ratio	0.113	0.044	0.033	0.001
dev_feature_bug_coding_time_ratio	0.067	0.044	0.028	0.015

of anti-patterns. More importantly, the designer should be able to recognize the proper solution to remove these anti-patterns, for example, by applying proper abstractions or design patterns [45]. In other words, to prevent complexity and technical debt from accumulating, the designers should be knowledgeable on design principles [39], [46], [47], skilled at abstraction, capable of applying these principles in coding, and capable of rectifying their violations.

(2) The architectural complexity metrics and anti-patterns can be used to provide guidance on when, where, and how to reduce architectural complexity. Simply stating "*this system is too complex*" does not lead to concrete actions. According to Table VI, anti-patterns caused by cycles among files and improper usage of inheritance present most impact on maintenance burden, indicating that it is possible to reduce architectural complexity by removing cycles and employing more effective abstraction. Our DV8 workflow also outputs each instance of each type of anti-patterns, pinpointing the specific files involved in these anti-patterns. Using this information, the team can take concrete actions to reduce overall architectural complexity by addressing specific anti-patterns. Researchers, educators, and students can leverage these insights to explore solutions for anti-patterns that directly enhance feature addition, bridging the gap between academic design principles and the productivity metrics most relevant to practitioners.

VI. Threats to Validity

In this section, we discuss the external, internal, and construct threats to the validity of our study [48].

External Threats to Validity. In this study, we only examined projects written in Java and C/C++. Therefore, it is possible that the results could be different for projects written in other programming languages. Since we only studied projects within Google LLC, it is also possible that other industrial projects may have issues that require more refinement of our metrics.

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 Kythe[3] to extract dependencies. It is possible that if we use another static analysis tool, such as Understand[4], the results could be different, especially in complicated cases where sophisticated syntax are used. Different tools extract and export these dependencies slightly differently. We mitigated this threat by presenting the extracted DSMs to the developers from a few sample projects, and didn't observed missed or misreported dependencies. Analyzing the differences among these dependency tools is our future work.

[3]https://kythe.io

[4]https://scitools.com/

Another threat to our study of sentiment is that it is solely based on survey responses. It is possible that developers who are unhappy about their projects are not willing to respond to the survey, or developers can be too busy and ignore these survey requests. Using panel analysis to assess how sentiment varies with development activities for each developer mitigates the threat to some extent.

Although we obtained statistically significant results between architectural complexity and feature-over-bug activity metrics, when it comes to a specific project, it is possible that CLs are not properly labeled by some developers. On the other hand, we hope these results will encourage more teams to adopt more rigorous practice to label their CLs carefully.

We also noticed that a large number of Java projects uses a dependency injection framework, which allows a class to *install* other modules. As a result, these "*connector*" classes depend on many other classes through the *install* relation, which could form a big module that significantly reduce its DL score, but not on PC or anti-patterns. Assessing the impact of such framework is our future work.

Construct Threats to Validity. In this study, we statistically analyzed the relations between two pairs of measures: architectural complexity vs. maintenance activity, and developer sentiment vs. maintenance activity. Given the limitation of the available data sources, we were not able to directly assess the correlation between architectural complexity and developer sentiment. Ideally, we would monitor the variation of PC, DL, and anti-patterns within a project for an extended period of time, and conduct panel analysis between architectural complexity and sentiment directly.

Moreover, it is possible that some legacy projects with much longer life span that have not employed the best practice may have more anti-patterns. It is also possible that these legacy projects do not need to add a lot of new features any more, hence the lower feature-over-bug ratio. In this study, we tried to mitigate this threat by selecting projects that have evolved for at least one year (long-lasting), with more than 500 files (no-trivial), and with commits by at least 20 developers (active), so that these projects are unlikely to be either brand new or retiring. Adding the project age as a control variable could be an interesting future work.

To assess developers' sentiment, we recoded a Likert scale to a numeric interval scale. This conversion assumes that the distances between the survey response are equal (e.g. the change in hindrance for someone to switch their rating from "Moderately hindered" to "Slightly hindered" is the same as switching from "Slightly hindered" to "Not at all hindered"), which may not be true. A better model choice would be to fit an ordinal mixed effects regression to the panel data, which

TABLE VI: Architectural Complexity and Maintenance Activity

Activity	Complexity	Coefficient	p-value	Adjusted p-value
med_feature_bug_loc_ratio	#UnhInh/#File	-0.397	0.001	0.016
med_feature_bug_loc_ratio	#UnhInh/#LOC	-0.685	0.002	0.021
med_feature_bug_loc_ratio	#Clique/#LOC	-1.699	0.003	0.021
med_feature_bug_loc_ratio	PC	-1.345	0.003	0.021
med_feature_bug_act_ratio	PC	-1.027	0.008	0.045
med_feature_bug_file_loc_ratio	#Clique/#File	-1.567	0.046	0.231
med_feature_bug_act_ratio	#Clique/#LOC	-0.985	0.060	0.257
med_feature_bug_act_ratio	#Clique/#File	-1.233	0.076	0.284
med_feature_bug_act_ratio	#UnhInh/#File	-0.152	0.139	0.465
med_feature_bug_act_ratio	#UnhInh/#LOC	-0.523	0.184	0.553
med_feature_bug_act_ratio	DL	0.077	0.409	1.000
med_feature_bug_act_ratio	#PkgClc/#File	-0.332	0.427	1.000
med_feature_bug_file_loc_ratio	DL	-0.069	0.515	1.000
med_feature_bug_file_loc_ratio	#PkgClc/#File	0.313	0.528	1.000
med_feature_bug_act_ratio	Files	-0.197	0.609	1.000
med_feature_bug_file_loc_ratio	#PkgClc/#LOC	0.148	0.646	1.000
med_feature_bug_act_ratio	LOC	-0.159	0.654	1.000
med_feature_bug_file_loc_ratio	Files	0.175	0.698	1.000
med_feature_bug_act_ratio	#PkgClc/#LOC	1.011	0.807	1.000
med_feature_bug_file_loc_ratio	LOC	-0.098	0.811	1.000
med_feature_bug_cls_ratio	PC	0.000	1.000	1.000
med_feature_bug_cls_ratio	Files	0.000	1.000	1.000
med_feature_bug_cls_ratio	LOC	0.000	1.000	1.000
med_feature_bug_cls_ratio	#PkgClc/#LOC	0.000	1.000	1.000
med_feature_bug_cls_ratio	#UnhInh/#LOC	0.000	1.000	1.000
med_feature_bug_cls_ratio	#UnhInh/#File	0.000	1.000	1.000
med_feature_bug_cls_ratio	#Clique/#LOC	0.000	1.000	1.000
med_feature_bug_cls_ratio	#PkgClc/#File	0.000	1.000	1.000
med_feature_bug_cls_ratio	#Clique/#Files	0.000	1.000	1.000
med_feature_bug_cls_ratio	DL	0.000	1.000	1.000

could be our future work.

VII. Related Work

In the past decade, technical debt analysis and its impact on software productivity and quality have been widely studied [26], [44], [49]. In these prior studies, technical debt is mostly approximated as code smells or source files with poor quality scores, which can be detected by various research and commercial tools, as summarized by Fontana et al. [50], Fernandes et al. [51], Thanis et al. [52], and Avgeriou et al. [53]. The most common proxies of technical debt include God Class, Cloned code, and Feature Envy. A recent comparative study [54] reveals that even for commercial tools, the accuracy of these tools and their capability of detecting true debt are mostly questionable. There is also no direct evidence that these file-level code smells have significant impact on productivity.

The impact of technical debt on productivity is mostly estimated. Curtis et al. [55] presented a model to estimate the cost of technical debt from source code static relations. Nord et al. [56] proposed a formula to estimate technical debt impact on long-term product evolution. Martini and Bosch [57] proposed AnaConDebt, a TD management method that aims to help practitioners to make decisions on refactoring architectural debt items. Carriere et al. [58] also proposed a cost-benefit model to estimate the benefits of reducing the level of coupling in an e-commerce architecture. Still these works do not directly link to maintenance activities objectively collected from development logs.

Similar to our work, Kazman et al. [16] used anti-patterns to pinpoint design debt, and estimate the return on investment of potential refactoring activities. Mo et al. [14] reported that anti-pattern analysis helped practitioners to make decisions on if and how to refactor. Recently Nayebi et al. [15] also reported a case study using anti-patterns, DL, and PC to justify refactoring, and demonstrated productivity and quality improvement after refactoring using activity logs.

These anecdotal evidences motivated our large-scale study reported in this paper. None of the prior work establishes statistical evidence on to what extent DL, PC and anti-patterns impact maintenance effort, measured using LOC, CL, and active coding time, especially the effort spent to adding features, the most important aspect of productivity. This study is also the first that establishes statistical relation between maintenance effort and developer sentiment.

VIII. Conclusion

In this paper, we report a large-scale study of 1252 projects written in C++ or Java from Google LLC, aiming to understand the relation between architectural complexity, maintenance activity, and developer sentiment. We have obtained significant evidence of the following findings:

(1) Developers who felt that they are less hindered by technical debt and complexity were able to spent more effort on feature-addition rather than bug-fixing;

(2) The more complex a system is (higher propagation costs and higher density of anti-patterns), the more LOC is spent on bug-fixing, rather than adding new features.

The implication is that, it is possible to monitor the teams' ability to add features vs. fixing bugs using development activity logs, observing the variation of maintenance difficulties objectively and continuously, instead of solely relying upon developer surveys collected periodically. In order to reduce maintenance burden and increase developers' satisfaction, it is important to control the architectural complexity caused by file dependencies and reduce anti-patterns, so that both design quality and feature-addition ability can be improved in a measurable way. The three categories of measures can be used together to help the team make informed decisions on when should refactor, how to refactor, and assess the impact of refactoring activities.

IX. Acknowledgements

This work was supported in part by the National Science Foundation of the US under grants CCF-2232720, CCF-2213764, and TI-2236824.

References

[1] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA, USA: PWS Publishing Co., 1998.

[2] F. B. e Abreu, "The mood metrics set," in *Proc. ECOOP'95 Workshop on Metrics*, 1995.

[3] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[4] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transaction on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[5] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, Aug 1994.

[6] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[7] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[8] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th International Conference on Software Engineering*, 2006, pp. 452–461.

[9] R. Mahouachi, M. Kessentini, and M. Ó. Cinnéide, "Search-based refactoring detection using software metrics variation," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 126–140.

[10] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, Jul. 2006.

[11] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: A new metric for architectural maintenance complexity," in *Proc. 38rd International Conference on Software Engineering*, 2016.

[12] C. Y. Baldwin and K. B. Clark, *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

[13] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[14] R. Mo, W. Snipes, Y. Cai, S. Ramaswamy, R. Kazman, and M. Naedele, "Experiences applying automated architecture analysis tool suites," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 779–789. [Online]. Available: https://doi.org/10.1145/3238147.3240467

[15] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and F. Chew, "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)*, May 2019, pp. 171–180.

[16] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *Proc. 37th International Conference on Software Engineering*, May 2015.

[17] W. Wu, Y. Cai, R. Kazman, R. Mo, Z. Liu, R. Chen, Y. Ge, W. Liu, and J. Zhang, "Software architecture measurement—experiences from a multinational company," in *Proceedings of the 12th European Conference on Software Architecture*, 2018, pp. 303–319.

[18] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of recurring high-maintenance architecture issues," in *Proc. 12th Working IEEE/IFIP International Conference on Software Architecture*, 2015.

[19] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis," in *Proc. 35rd International Conference on Software Engineering*, May 2013, pp. 891–900.

[20] Y. Cai and R. Kazman, "DV8: Automated architecture analysis tool suites," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019, pp. 53–54.

[21] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[22] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 242–251.

[23] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[24] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[25] R. Mo, Y. Cai, R. Kazman, and Q. Feng, "Assessing an architecture's ability to support feature evolution," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 297–307. [Online]. Available: https://doi.org/10.1145/3196321.3196346

[26] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debts," in *Proc. 38rd International Conference on Software Engineering*, 2016.

[27] Q. Feng, Y. Cai, R. Kazman, D. Cui, T. Liu, and H. Fang, "Active hotspot: An issue-oriented model to monitor software evolution and degradation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 986–997.

[28] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight," in *Proc. 36th International Conference on Software Engineering*, 2014.

[29] C. Jaspan, M. Jorde, C. Egelman, C. Green, B. Holtz, E. Smith, M. Hodges, A. Knight, L. Kammer, J. Dicker, C. Sadowski, J. Lin, L. Cheng, M. Canning, and E. Murphy-Hill, "Enabling the study of software development behavior with cross-tool logs," *IEEE Software*, vol. 37, no. 6, pp. 44–51, 2020.

[30] D. V. Steward, "The design structure system: A method for managing the design of complex systems," *IEEE Transactions on Engineering Management*, vol. 28, no. 3, pp. 71–84, 1981.

[31] A. MacCormack, J. Rusnak, and C. Baldwin, "The impact of component modularity on design evolution: Evidence from the software industry," Harvard Business School, Working Paper 08-038, Dec. 2007.

[32] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "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*, Sep. 2001, pp. 99–108.

[33] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2009, pp. 197–208.

[34] J. Lakos, *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[35] H. Melton and E. Tempero, "An empirical study of cycles among classes in java," *Empirical Softw. Engg.*, vol. 12, no. 4, p. 389–415, aug 2007. [Online]. Available: https://doi.org/10.1007/s10664-006-9033-1

[36] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *Proc. 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2005, pp. 167–176.

[37] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.

[38] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Transactions on Software Engineering*, vol. 5, no. 2, pp. 128–138, Mar. 1979.

[39] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, 1994.

[40] *Analysis of Panels and Limited Dependent Variable Models*. Cambridge University Press, 1999.

[41] G. Arminger, "Linear stochastic differential equation models for panel data with unobserved variables," *Sociological Methodology*, vol. 16, pp. 187–212, 1986. [Online]. Available: http://www.jstor.org/stable/270923

[42] R. Koenker, *Quantile Regression*, ser. Econometric Society Monographs. Cambridge University Press, 2005. [Online]. Available: http://www.amazon.de/Quantile-Regression-Econometric-Society-Monographs/dp/0521608279/ref=sr_1_1?ie=UTF8&qid=1312553603&sr=8-1

[43] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995. [Online]. Available: http://www.jstor.org/stable/2346101

[44] W. Cunningham, "The WyCash portfolio management system," in *Addendum to Proc. 7th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 1992, pp. 29–30.

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[46] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002.

[47] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–8, Dec. 1972.

[48] M. H. M. O. B. R. C. Wohlin, P. Runeson and A. Wesslen, *Experimentation in Software Engineering*, 2012.

[49] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing four approaches for technical debt identification," *Software Quality Journal*, pp. 1–24, 2013.

[50] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, pp. 5:1–38, Aug. 2012.

[51] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '16. New York, NY, USA: Association for Computing Machinery, 2016.

[52] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, p. 7, 12 2017.

[53] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, N. Saarimaki, D. D. Sas, S. S. de Toledo, and A. A. Tsintzira, "An overview and comparison of technical debt measurement tools," *IEEE Software*, pp. 0–0, 2020.

[54] J. Lefever, Y. Cai, H. Cervantes, R. Kazman, and H. Fang, "On the lack of consensus among technical debt detection tools," *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 121–130, 2021.

[55] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the principal of an application's technical debt," vol. 29, no. 6, 2012, pp. 34–42.

[56] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *SConference on Software Architecture and European Conference on Software Architecture*, 2012, pp. 91–100.

[57] A. Martini and J. Bosch, "An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt," in *International Conference on Software Engineering Companion*, 2016, pp. 31–40.

[58] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," 2010, pp. 149–157.