A New Perspective on Predicting Maintenance Costs

Florian Uunk*, Rick Kazman†, Yuanfang Cai‡, Noah Black‡, Carlos Andrade§ Giuseppe Valetto‡, Lu Xiao‡ and Fetsje Bijma*
* *VU University Amsterdam*
† *Software Engineering Institute, Carnegie-Mellon University, and University of Hawaii*
‡ *Drexel University*
§ *Federal University of Bahia*

***Abstract*—In this paper, we present a new approach to correlating file metrics to maintenance effort. We examine the correlations between *variations* in file metrics and *variations* in the maintenance effort spent on these files over multiple releases. Because effort data is seldom accurately collected, and is never collected for open source projects, we have employed three novel, broadened and more holistic measures of file-level *maintenance effort*: the number of lines of code changed to resolve tasks (*churn*), the amount of discussion that tasks generated (*discussions*), and the number of atomic changes made to a file to resolve a task (*actions*). From the data extracted from multiple Apache projects, we found that a small subset of file metrics were significantly correlated to our effort measures, especially to code churn and actions. The best correlations vary from project to project, suggesting that maintenance effort measurements should be project-specific.**

***Keywords*-metrics; maintenance effort; architectural complexity; coupling;**

I. Introduction

Estimating the duration, effort, cost and complexity of software development activities is of vital importance for IT management, but notoriously challenging to do well. Molukken and Jorgensen's review [1] showed that 60-80% of all software projects run over budget by 33% on average. More than half of the total development effort in software projects is spent on the maintenance phase [2]. During the maintenance phase, software tends to age [3] and the code base gets cluttered by an accumulation of changes, often referred to as *technical debt*[1]. When technical debt is not addressed, further development will be hindered. One major way to address technical debt is code (re-)modularization through refactoring.

If decision-makers (e.g. project managers) do not have good insight into the benefits of refactoring, it is difficult to know *when* to refactor, if at all. Numerous prediction models have been proposed recently to identify components that are error-prone [4]–[6], or to predict project development cost and effort [7], [8]. But this existing work does not directly support a project's decision–makers in answering the following question: *when is it worthwhile to refactor the software to reduce the complexity and make it better modularized?*

[1]http://www.martinfowler.com/bliki/TechnicalDebt.html

While the costs of modularization activities such as refactoring are significant and immediate, their benefits are largely invisible, intangible and long-term. It has been known for decades that modularity decay can cause substantial problems in projects, such as reduced ability to provide new functionality and fix bugs, operational failures, and, in the extreme, canceled projects. But there is no established quantitative association between modularity variation and maintenance effort variation. That is to say, there is no way for a decision-maker to know, with confidence, if a project's modularity gets worse (or better) how much more (or less) it will cost to maintain and extend. Without such a foundation, it is difficult to predict the costs of the technical debt incurred from a deterioration in a project's modularity. And it is equally difficult for decision-makers to justify the potential cost-savings from a proposed refactoring activity.

Our research aims to provide an empirical foundation upon which sound refactoring decisions may be based, by relating variation in the complexity of code to variation in effort—and hence cost. As a first step in that direction, we test the following primary hypothesis:

> *There is a statistically significant correlation between software complexity variation over successive releases of software project files, and the variation of effort required to maintain those files.*

If this hypothesis is true, then it becomes possible to understand how the maintenance effort for a code unit, such as a file, varies with changes in one or more source file metrics that characterize its complexity and modularization, which we can easily capture and track. That, in turn, makes it possible to predict the future maintenance costs of that file. This information can then be used to make economics-driven decisions about software maintenance, including refactoring. To test this hypothesis, we need to measure how structural relations in a file change over time on one hand, and how maintenance effort spent on that file changes on the other.

Our first step in addressing this hypothesis is to select a suite of suitable file metrics. Numerous source code metrics have been proposed and studied, but not all of them have a proven correlation with maintenance costs. We conducted a literature survey and identified a set of metrics that have validated relationships to maintainability, with solid theoretical and empirical bases.

The second step is to select a suitable measurement of *effort* at the file level. Although effort at the project or task level is usually measured using person-months, there is no widely agreed-upon effort measurement at the file level. Maintenance effort spent on a file can be measured in multiple ways. Just measuring changes in the number of lines of code–the most obvious measure–may be misleading. For example, if the change is inherently difficult, even adding just a few lines of code may require a great deal of effort, in the form of intensive discussions within the project team, or multiple trial-and-error rounds leading to several revisions.

We thus employ a new, and we believe, more holistic approach to measuring effort, from three dimensions: the number of lines of code changed to resolve tasks (*churn*), the amount of discussion that tasks generated (*discussions*), and the number of atomic changes made to a file to resolve a change request (*actions*).

Employing our selected file metrics and new effort measures, we have developed a new empirical approach to correlating file metrics variation to the variation of maintenance effort on a file. Using these techniques, we investigate the following research questions to test the primary hypothesis:

Q1: Which file-level code metrics are significantly correlated to which types of maintenance effort?
The answer to this question will help us understand which code metrics best track the increase or decrease of maintenance effort. We will examine the selected metrics and their correlation to the maintenance effort measures on a per-file basis, using statistical models.

Q2: Does the correlation between file-level code metrics and maintenance effort differ between projects?
The answer to this question will help us understand which metrics to use to make predictions for a variety of different projects, or if the types of maintenance effort that are best correlated with file metrics are mostly project-specific.

While testing these hypotheses is of interest to most software projects, few industrial projects are willing to contribute the data needed to answer the question with authority. We have therefore selected five Apache open source projects containing between 8 and 18 releases as our experimental subjects. For each file of each project, we calculated its structural properties using the selected file metrics, and computed the delta of each metric between subsequent releases. We also extracted maintenance effort for each file, and their deltas between releases. After that, we analyzed the correlation between pairs of code metrics deltas and effort measure deltas. We are interested to know whether *changes* in code metrics are strongly correlated with changes in effort measures.

Our results demonstrated that there are strong and significant correlations between a small subset of the selected file metrics and two of the three maintenance effort measures: the number of actions and the number of code churns. This result is confirmed across all the five subject projects. Furthermore we observed that, when each project is considered separately, the most significantly correlated metrics-effort pairs differ, and in certain projects, the amount of discussion also shows strong and significant correlations. We also observed that the significance of the correlation between metrics and effort increases in later releases. These results provide a positive answer to the hypothesis, showing that it is possible to quantitatively predict maintenance effort changes from code-level file metrics, and that the best measures and metrics may be project-specific.

II. Related work

As previously stated, our research aims to advance the way maintenance effort is measured, and the way source code metrics and maintenance effort are correlated. To situate our contribution we will first review prior research on code metrics and then examine the research on correlating these metrics to maintenance effort. Finally, we discuss how our approach differs from prior work.

A. Research on metrics

Published source code metrics can be broadly divided into 5 categories, based on what they measure: size, complexity, coupling, cohesion, and inheritance. We will give a brief description of each category, along with some of the most influential publications on source code metrics.

1) Size: Size is the most straightforward metric for source code. The number of lines of code (LOC) is the most obvious and simplest way of measuring size. But it has its drawbacks. For example, it is always possible to write the same functionality with fewer (or more) lines of code, while maintaining similar complexity. To address this problem, several other metrics have been proposed.

2) Complexity: Measures of the complexity of a source file are postulated to affect modifiability and maintainability: lower complexity is better. Examples of complexity metrics are Halstead Volume [9]—based on operator and operand counts, and McCabe Complexity [10]—based on the number of possible paths in program control graph.

3) Coupling: Coupling describes the number of connections a file or class has to other files or other classes. The assumption is that lower coupling is better. Briand et al. proposed a set of metrics that measure different possible versions of class-to-class coupling [11]. Another coupling metric is Propagation Cost, which was first introduced by MacCormack et al. in 2006 [12].

4) Cohesion: Cohesion measures how strongly the responsibilities within a code unit are related. The rationale behind measuring cohesion is the belief that code units, such as source files or classes, should focus on just one thing, and that doing so will improve maintainability.

5) Inheritance: Inheritance-based metrics only apply to object-oriented code. Less complex inheritance hierarchies are expected to be easier to understand and maintain.

Chidamber and Kemerer [13] (henceforth C&K) developed the best known metrics suite aimed at measuring object-oriented source code, including metrics for coupling, cohesion and inheritance.

B. Research on metrics and maintenance effort

A number of papers have attempted to correlate source code metrics to maintenance effort. However, there is no generally agreed-upon method to predict maintenance effort at the level of a source code *file*. We now describe a number of the approaches that have been attempted.

1) Comparing against expert judgement: Welker et al. [14] proposed a polynomial that uses complexity based metrics to predict maintenance effort. The weights for each of these metrics are automatically fitted, so the polynomial matches data of expert judgement in 8 systems. They presented this polynomial as the *Maintainability Index*.

2) Comparing against Maintainability Index: Misra [15] and Zhou and Xu [16] compared a list of complexity and inheritance metrics against the Maintainability Index at the system level. Both papers found significant correlations to both inheritance and complexity.

3) Comparing in controlled experiments: Harrison et al. [17] compared metrics against both expert judgement and maintenance measurements obtained in a controlled experiment. They found that both complexity and cohesion correlated with their maintenance measures, and that complexity correlated with the expert judgement of a system.

Arisholm [18] looked at 10 changes made to an industrial system and logged hours spent on each task. He found no correlation between source code metrics and effort, possibly due to the small size of the data set.

4) Comparing against change: Li and Henry [19] linked a set of metrics to the total volume of changes to classes in two different projects. They found significant correlations for complexity, coupling, cohesion and inheritance metrics. Binkley and Schach [20] looked at change volume of an industrial system. They positively correlated this to coupling and complexity metrics and to one inheritance metric. Ware et al. [21] looked at the number of changes and the number of lines changed for files in a commercial application. They found significant correlations for complexity and coupling measures.

In a slightly different vein, Anbalagam and Vouk [22] found a significant correlation between the number of participants in a bug report and the time taken to complete it. While this is not an effort measure, it is certainly related to the organizational dimension of a software project and the corresponding effort overhead.

5) Comparing over releases: Demeyer and Ducasse [23] attempted to identify problem areas in the source code of a project and check if those problem areas were refactored in later releases. They did not find such a pattern, but they noted that the project they studied was in substantially good shape, so there might not have been a major need for refactoring. Alshayeb and Li [24] attempted to correlate a polynomial, consisting of complexity, coupling, and inheritance metrics, to maintenance effort in iterative projects. They measured lines of code added, deleted and changed, first between releases of a project, then between changes within a release. They found that their constructed polynomial was reasonably good at predicting effort between changes, but not as good at predicting effort between releases.

C. Differences in our approach

Our research is different from earlier research as follows: First, instead of simply comparing file-based code metrics to maintenance effort, we compare an increase or decrease in file metrics to an increase or decrease in maintenance effort. This is intended to give us a more accurate insight into the effect of source code variations across the lifetime of a project. If there is a clear correlation between a change in a metric value and a change in a maintainability measure, a project manager will be able to use this knowledge to make informed decisions about maintenance and refactoring opportunities. Although D'Ambros et al. [25] also employed variations of source code metrics, they used them to predict defects. Our research is the first to correlate variations of file-level code metrics to maintenance effort.

Second, instead of measuring maintenance effort merely using the number of changes in lines of code (churn), we add two new measurements: the number of *actions*—atomic changes to resolve an issue—and the amount of *discussion* among developers for an issue. This gives us a more holistic, multi-dimensional view of maintenance effort, assuming that a complex change will require more discussion and more changes to files (since some of the initial changes will not be correct and will necessitate subsequent rounds of changes).

Third, only Alshayeb and Li [24] have also looked at the relation between metrics and maintenance effort over multiple releases, but instead of comparing variations in metrics over different releases to variations in maintenance effort, they created a formula to predict maintenance effort from source code metrics, and tested the formula over various releases and changes on the project. They examined only 13 changes in total, whereas we are looking at thousands of changes. While other researchers have leveraged code churn to predict project effort [7] and defect density [4], we are the first to investigate the correlation between file metrics and code churn, which is a manifestation of effort at the file level. We will discuss more about our choice of effort measures, including threats to their validity, in Section VI.

III. Methodology

In this section we describe the subject projects that we have studied, the set of measures we chose to collect from these projects, the rationale behind these choices, and our data collection methods.

A. The subject projects

The selection of projects is important to the quality of the data, and therefore to the validity of the research. Now we describe the criteria we used to select the projects for this research, and the motivation behind those criteria.

1) Criteria: To ensure the generality of our research, we attempted to obtain a diverse set of projects. We specifically looked for heterogeneity along the following dimensions:

Variation in domain of software: Uses of software can be categorized into various application domains. We tried to find projects from distinct domains to ensure that our research results would apply generically.

Variation in source code sizes: Even though Dolado [27] has shown that development productivity does not vary significantly across project sizes, maintaining a large-scale software project is still, in practice, different from maintaining a small scale software project.

Variation in team size: There has been considerable research on the effects of team size on development speed. Brooks argues that smaller teams tend to have greater productivity per person [26], but many have argued that larger teams are better for productivity and quality in Open Source Software.

Variation in project age: The age of a body of software can influence developer productivity in ways that may not be measurable by source code metrics. For example, the technology chosen (language of implementation, operating system, development libraries, etc.) can cease to be supported, and key developers can leave the project, resulting in a knowledge loss.

We have, however, restricted our attention to projects written in Java, so that we could repeat the same metrics extraction process for our entire set of projects. Furthermore, we only selected projects that use a version control system and a bug tracking system, as our maintenance data is derived from these systems. The projects selected all contain source code and maintenance data for at least 8 releases, and all have more than 300 resolved issues in their bug tracking systems.

2) Selected projects: We have summarized the project characteristics, the first and last release for which we have extracted data, and the number of resolved or closed issues that we were able to extract in table I. As we can see from the table, Derby[2], Lucene[3], PDFBox[4], Ivy[5], and FtpServer[6] are from different domains, with different team sizes and different project ages. The number of resolved issues ranges from 329 to 3058.

B. The selected metrics

In this section, we discuss the metrics that were calculated for each file of each project.

1) Criteria: As described in section II, numerous metrics (summarized in [28], [29]) have been proposed that are purported to predict software quality and maintenance effort. Unfortunately, testing all these metrics for their power in predicting maintenance effort was infeasible. To select a smaller target set of metrics to analyze, we applied 3 criteria.

The metric is widely applicable: Since we are restricting our research to projects written in Java, the metrics will have to be applicable at least to this language.

The metric is defined at the file level: The unit of analysis in our research is the source file, so the metric has to be interpretable at the file level. We do, however, employ metrics that are defined at the class level. To account for this discrepancy we only consider files that contain a single class. This constraint only eliminates around 7% of the total files from our candidate data set.

The metric has been consistently proven in previous research: To keep the scope of the research manageable, we chose only metrics that have been consistently shown to be correlated with maintenance effort in previous studies.

2) Selected metrics: We have selected the following metrics:

Source Lines of Code (LOC): The total lines of code in the file. The idea behind this metric is that, all other things being equal, larger files are harder to maintain.

Weighted Method Complexity (WMC): The sum of the complexities of the methods in a class.

Response For a Class (RFC): Total number of methods that may be invoked as a result of a invoking any method in a class.

Coupling Between Objects (CBO): The number of other classes that the class in this file is connected to.

Lack of Cohesion Of Methods (LCOM): The number of method pairs in the class in this file that do not share the usage of a single attribute of the class.

Depth in Tree (DIT): The number of classes that are a superclass of the class in this file.

Number of Children (NOC): The number of classes that have the class in this file as a superclass.

Afferent Couplings (Ca): A measure of how many other classes use the specific class in this file.

[2]http://db.apache.org/derby/
[3]http://lucene.apache.org/core/
[4]http://pdfbox.apache.org/
[5]http://ant.apache.org/ivy/
[6]http://mina.apache.org/ftpserver/

Table I
SELECTED PROJECTS

Project	Releases	Resolved Issues	Contributers	First Release	Last Release	Domain
Derby	18	3058	458	2005/08 (10.1.1.0)	2011/10 (10.8.2.2)	Relational Database
Lucene	18	2444	652	2006/03 (1.9.1)	2010/12 (3.0.3)	Distributed search
PDFBox	8	699	387	2010/02 (1.0.0)	2011/07 (1.6.0)	PDF document manipulation tool
Ivy	12	758	408	2006/11 (1.4.1)	2010/10 (2.2.0)	Transitive relation dependency manager
FtpServer	10	329	112	2007/02 (1.0.0-M1)	2011/07 (1.0.6)	Java-based FTP server

Number of Public Methods (NPM): The number of methods in a class that are declared as public.

WMC, RFC, CBO and LCOM, DIT and NOC have all been described by C&K in 1994 [13]. We have altered their definitions slightly to make them meaningful at the file level, as described in section III-B1. The C&K suite has been studied heavily, and its metrics have been validated in many studies (e.g. [15], [17], [19]–[21]).

In addition to the extended set of C&K metrics, we have selected one more metric—Propagation Cost, which aims to capture *architectural* complexity.

Propagation cost (PC): PC, first introduced by MacCormack et al. in 2006 [12], is a coupling-based metric, and there has been some promising research on the predictive power of PC on maintenance effort [12], [30].

PC is based on a *visibility matrix* [12], which is a binary matrix where a project's files are the rows and columns, and dependencies between the files are the values. These dependency values are determined using a path length L, which allows a file A to be dependent on a file B through a dependency chain of length L. For example, a path of length of 1 denotes traditional coupling, since only direct dependencies are represented in the matrix. The propagation cost is computed as the sum of all dependencies in the visibility matrix, divided by the total possible dependencies.

However, propagation cost calculated that way is defined at the project level. Instead, we compute *incoming propagation cost* at the file level, by taking the sum of the incoming dependencies for a file, divided by the total possible dependencies; concretely, this means that we take the sum of the column in the visibility matrix that represents the file, and divide this value by the length of the column. Analogously, to calculate *outgoing propagation cost* per file, we take the sum of the row in the visibility matrix that represents the file, and divide that by the length of the row. This approach is a slight variation on the work of Ferneley [31] and Yang and Tempero [32] who have found promising results.

We have also introduced a new variant of the propagation cost metric that employs a *decay rate*. With this decay rate, we reduce the strength of indirect dependencies by a factor for each additional step in the dependency chain. In the present study, we applied a decay rate factor of 0.1.

In this research, we have investigated both incoming and outgoing propagation costs with path lengths of 1, 3, 5, 10, and 20 – with and without decay – to see which variant of the propagation cost metric has the most predictive power. When describing our results, we will use a naming convention; for example, PROP-OUT-10-N indicates outgoing propagation cost with path length 10 and without decay, whereas PROP-IN-5-D indicates incoming propagation cost with path length 5 and with decay.

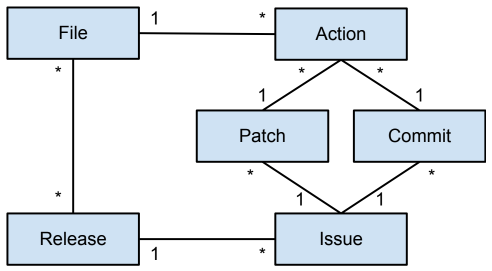

Figure 1. The data model

Considering all combinations, we have a total of 18 propagation cost metrics (not 20, since when path length is 1 the decaying and non-decaying version of the metrics are the same).

The 18 propagation cost metrics, plus LOC and the 8 C&K metrics, give us a grand total of 27 metrics that we calculate for each file of each release of each project.

C. The data collection method

The data model used in our data extraction and analysis procedure is illustrated in Figure 1.

For each project, we study a number of releases, each of which have a set of files. Each project also has a list of issues, which are extracted from the project's bug- or issue-tracking software. Issues consist of both bug reports and change requests. Developers can submit patches to suggest a solution to an issue. Patches consist of a list of actions, which are changes to files that were made to resolve the issue. For each action, we measure code churn as the number of lines of code added and removed, where changed lines count as both added and removed. Each action corresponds to a change done in 1 file for 1 issue, but issues are often resolved using multiple actions. The patches that finally get accepted (i.e., that resolve the issue) are called commits.

Developers can also associate comments with issues. These comments are used for communication between developers.

1) Extracting the data: We populated our data model by extracting data from the bug tracking system and version control repositories for each project. Since our 5 projects were all maintained by the Apache foundation, the same technologies were used. The bug tracking system in use is Jira[7], which has a WSDL API that is usable for data extraction. Their version control system is Subversion[8]. We constructed a number of tools that query both systems for the data, format it, and insert it into our database based on our data model.

2) Compiling code metrics: We used 3 different programs to calculate file metrics. To calculate the source Lines of Code (LOC), we used the utility *SLOCCount*[9]. This gives us the number of lines of code in the file. For the extended set of C&K metrics, we used *ckjm* 1.9[10]. For the propagation cost based metrics, we wrote our own tool, which first generates a Dependency Structure Matrix (DSM) [12], from which we can count the number of incoming and outgoing dependencies per file over various path lengths.

For each release, we took a snapshot of the code base from the version control system of the project. After extracting the selected metrics per file per release from those snapshots, we stored them in our database for analysis.

3) Measuring maintenance effort: Since we are looking at open source projects, the developers did not log hours for their maintenance work. To approximate maintenance effort, we scrutinized our data set to see which *proxy measures* for maintenance effort we could find. We settled on and collected 3 file-based proxy measures.

Discussion: This is the amount of discussion that occurred in resolving an issue. The assumption is that a more complex change is likely to generate more discussion. Concretely, we measure the number of comments that have been made in the bug tracking system for an issue, and we associate the numbers to the files modified to resolve that issue.

Change in lines of code (Churn): Churn is the total number of lines of code that were changed in the file to resolve an issue. If a file is changed multiple times for the same issue, we see if the file changes overlap to make sure we don't count the same changes multiple times.

Actions: We also measure the number of actions, as described in section III-C, that were performed to resolve an issue. Concretely, this counts the total number of patches and commits that were needed to resolve an issue affecting a file. The notion here is that the more complex the file, the more likely it is that it would require a large number of actions if something has to be changed in it.

[7]http://www.atlassian.com/software/jira/
[8]http://subversion.tigris.org/
[9]http://www.dwheeler.com/sloccount/
[10]http://www.spinellis.gr/sw/ckjm/

Table II
Example Data Extracted from Project Files

	File name	Releases	LOC	actions	Group	Rel. LOC	Rel. actions
1	client.java	1.0	100	40	-	-	-
2	client.java	1.0	100	60	-	-	-
3	client.java	1.1	140	75	1	1.4	1.5
4	client.java	1.2	70	60	2	0.5	0.8
5	library.java	1.1	60	75	-	-	-
6	library.java	2.0	72	150	1	1.2	2.0
7	server.java	1.0	200	40	-	-	-
8	server.java	1.2	240	52	1	1.2	1.3

IV. Statistical analysis

The methodology described in section III gives us a data set consisting of code metrics and maintenance effort measures for each release of each file. Since we want to analyze the relationship between increasing or decreasing code metrics and maintenance effort, we transform these numbers so that they reflect the *change* in that value from the previous release. To this end we take the metric value of each file in a release, and divide it by the equivalent value in the previous release. If the resulting quotient is less than 1, it means the value has decreased; if it is greater than 1, it has increased; if it equals 1, the value remains unchanged. We do an analogous calculation for effort values: since the same file can be involved in multiple issues for the same release, we divide the effort measure for each issue in the new release by the *average* effort measure per issue in the previous release. These derived data points each represent the change in a metrics or effort measure between two consecutive releases of a given source file.

The calculation of relative values is exemplified in Table II (please ignore the group column for now, which will be explained in section IV-A). This table shows how the relative values for one file metric, *LOC*, and one effort measure, *actions*, are calculated. The relative LOC value of "client.java" in row 3 (1.4) is the quotient of its current LOC value (140) and its LOC value in the previous release (100). We calculated relative effort values in a slightly different way. This is because our effort measures are calculated on a per-issue basis. For example, if "client.java" was changed in release 1.0 to address two separate issues, it takes two separate rows (Row 1 and 2 in Table II).

To calculate the relative *actions* value of "client.java" in release 1.1 in row 3 (1.5), where "client.java" was involved in two issues in the previous release, we divide its current value (75) by the *average* of all previous *actions* values ((40 + 60) / 2 = 50). The file "library.java" in row 5 and 6 has no action data associated with it for release 1.0 or 1.2, so the entry for release 1.1 is used as the first data point, and the entry for release 2.0 as the second data point. We calculated relative values in this manner for each file with all 27 x 3 = 81 metric-effort pairs.

A. Spearman analysis

Each line in Table II is considered as a *data point*. We thus calculated the data points for each individual file of each project as explained above, to understand the correlations between code metric variation and effort measure variation. Since the effort values do not follow a normal distribution, we have used the Spearman rank correlation test.

However, Spearman assumes independent measurements in the data set subject to the test. Since our data reflects deltas of the same measurements over time for each file, we cannot assume such independence, and we must not include data points pertaining to the various releases of the same file in the same data set.

We thus segment our data points into *groups* (see Table II) according to the following procedure: we first skip all data points which come from the first release we have on record for a file, because without a *previous* release, we cannot calculate meaningful relative values. After that, all data points that belong to the second release of each file go in group 1, all data points that belong to the third release of a file go in group 2, etc. This means that data from different releases of the same file never appear in the same group. We can then apply the Spearman correlation test separately to each group.

Once more, Table II shows an example of how the segmentation of data into groups is done. Row 1 and 2 belong to the first release we have information for file "client.java". Because we use values that are relative to the previous release, we can not use these values in our analysis. Row 3 belongs to the second release of "client.java", so it goes into group 1. Row 4 belongs to the third release of "client.java" that we have information for, so it goes into group 2.

Row 5 contains the first release of "library.java" that we have information for, thus it has no relative values. Please note that release 1.1, which the data point for row 5 belongs to, does not have to be the first release in which library.java existed in the project; rather release 1.1 is the first release where we have maintenance effort data for "library.java". Row 6 belongs to the second release of "library.java" that we have information for, so it goes into group 1.

Row 7 is the first release of "server.java" that we have information for, so we can not add it to a group. Row 8 is the second release of "server.java" that we have information for, so we add it to group 1. Please note that for row 8 (release 1.2), the relative values are not relative to a data point from the previous release (1.1), but from two releases before (1.0). This is because there is no effort data recorded for server.java in release 1.1.

In our analysis we exclude issues that did not affect source files. For a file to show up in at least one group, it must be

Table III
METRIC - EFFORT CORRELATION FOR AGGREGATED DATA SET

Group	Metric Type	Effort	P	Rho
3	CKJM - WMC	actions	0.00045	0.3910
4	CKJM - RFC	actions	0.00106	0.5158
4	raw LOC	actions	0.00167	0.4425
4	CKJM - Ca	actions	0.00180	0.3436
5	PROP-OUT-20-N	churn	0.00293	0.6378
5	PROP-OUT-5-N	churn	0.00899	0.5654
5	PROP-OUT-10-N	churn	0.00294	0.6378
6	CKJM - CBO	churn	0.00228	0.6694
6	PROP-OUT-1-N	actions	0.00008	0.7586
6	CKJM - NPM	actions	0.00006	0.7154

changed in more than one release. As a result, 2888 of the total 9733 issues from all 5 projects were used to generate usable data points.

We ended up having 9 groups from this aggregated data set, each having 87,602, 47,028, 21,686, 9,614, 5,804, 3,167, 1,388, 374, and 348 data points respectively. Since groups with higher numerical IDs will have an increasingly smaller population of data points (that is, fewer files have that many rounds of changes over releases), which is detrimental to the accuracy and reliability of the statistical analysis, we decided to only use groups 1 to 7.

In summary, each data point represents how one type of maintenance effort and one type of file metrics of a file vary over two successive releases where the file was changed. Investigating all the groups will reveal how these two aspects change together over multiple releases.

V. Results

Since we have performed a Spearman analysis on a set of 81 (27 x 3) code metric type versus maintenance effort type pairs, over 9 different groupings of files, we have run a total of 729 Spearman tests and therefore obtained 729 p and rho values. Due to the large number of tests, we allow a maximum p-value of 0.01 to ensure the significance of the results. We thus obtained 17 significant correlations from the aggregated data set. In addition, we performed the same Spearman tests on each project. The five projects have 9 (Derby), 3 (FtpServer), 7 (Ivy), 8 (Lucene) and 4 (PdfBox) groups, yielding over 2500 additional data points. To answer the two research questions proposed in Section I, we report significant results ($p < 0.01$) obtained from the aggregated data set and from individual project data in Table III and Table IV respectively.

Our research question Q1 asked which metrics are significantly correlated to which types of maintenance effort. To answer this question, in Table III, we list all significant results ($p < 0.01$) with rho value of at least 0.3, obtained from a sample size of at least 15. All of the top results show a strong correlation with just two effort types: actions and churn. In addition, it is interesting to note that 7 of the top 10 results show a strong correlation with coupling-based

Table IV
METRIC - EFFORT CORRELATION FOR INDIVIDUAL PROJECTS

Project	Grp	Metric Type	Effort	P	Rho
Derby	6	CKJM-NPM	actions	0.00007	0.81
Derby	3	CKJM-WMC	actions	0.00222	0.55
Derby	1	PROP-IN-20-N	discussion	0.00001	0.30
Lucene	1	PROP-IN-5-N	actions	0.00010	0.38
Lucene	1	PROP-IN-1-N	actions	0.00022	0.38
Lucene	1	CKJM-Ca	actions	0.00010	0.35
PDFBox	2	CKJM-LCOM	discussion	0.00062	0.58
PDFBox	2	PROP-IN-1-N	churn	0.00208	0.50
PDFBox	1	CKJM-RFC	actions	0.00841	0.48
Ivy	3	CKJM-RFC	churn	0.00031	0.71
Ivy	3	PROP-OUT-20-N	churn	0.00157	0.46
Ivy	3	PROP-OUT-10-N	churn	0.00157	0.46
FtpServer	1	PROP-IN-5-N	actions	0.00741	0.50
FtpServer	1	PROP-IN-20-N	actions	0.00910	0.49
FtpServer	1	PROP-IN-10-N	actions	0.00910	0.49

metrics: 4 variants of PC and 3 C&K metrics (Ca, CBO, and RFC).

The values in Table III are ordered in terms of their group. It is encouraging that the rho value tends to increase as the group number increases for two reasons. First, a good predictive model should improve as you accumulate more data. Second, the accuracy of a predictive measure of maintainability, if it is a good measure, should increase over time as maintenance concerns and technical debt accumulate in a project. The idea is that: in the early stages of a project there tends to be a low level of technical debt and so virtually any structure of the software will not be overly problematic. As technical debt accumulates, complexity concerns tend to overwhelm a programmer's cognitive abilities, and maintenance costs go up. This is exactly what we can see from the results presented in Table III.

In Table IV, we list the top 3 most significant results (p $<$ 0.01) with rho value of at least 0.3 and sample size of at least 15, for each individual project. Our research question Q2 asked whether these correlations would differ between different projects. The project-by-project results shown in Table IV suggest that this is indeed the case.

While actions and churn show up as important measures in all 5 projects—highly correlated with the C&K and PC metrics—discussion is highly correlated with PC in Derby and with LCOM (Lack of Cohesion Of Methods) in PDFBox. Similar to the results obtained from the aggregated data set, coupling-based metrics are strongly correlated with effort in 12 out of 15 cases.

VI. Discussion

In this section, we discuss the results, threats to validity and future work. Based on the analysis results presented in the previous sections, we can now answer the research questions proposed in Section I:

1) *Which file-level code metrics are significantly correlated to which types of maintenance effort?*
LOC, several variants of PC as well as RFC, CBO, WMC, NPM, and Ca have been shown to be significantly correlated to the maintenance effort measures of actions and churn. Previous work has shown that the 8 metrics from the C&K metric suite are good predictors of maintenance effort [19], [20], [33]. Our work supports these results and furthermore provides empirical evidence that PC metrics are also good predictors of effort.
2) *Does the correlation between file-level code metrics and maintenance effort differ between projects?*
We found meaningful differences in the correlations between metrics and measures in the different projects that we studied. This is not surprising, since projects have very different characteristics (e.g. size, age, domain, inherent complexity) as shown in Table I, as well as very different styles of leadership and cultures. For example, discussions are highly correlated with complexity metrics in just 2 of the 5 projects we studied: Derby and Lucene. This emphasizes the need for a large and varied corpus of data on which to base a predictive model of maintenance, since the most effective metrics and measures may be project-specific.

A. *Threats to validity*

Now we discuss possible threats to validity caused by the choice of maintenance effort measures, the limitations of data extraction, and the choice of projects to study.

We employed the three types of file-level effort measures because developers in open source projects do not log their hours for maintenance work. Even in industrial settings, accurate effort logs are hard to obtain. Moreover, we need maintenance effort at the *file level*, which is even harder to obtain, if not impossible. We hypothesize that the three measures (churns, actions, and discussions) can adequately approximate the maintenance effort spent on a file. This is, however, a hypothesis that is impossible to directly test within our existing research framework.

We considered several other possible maintenance effort proxies, such as the time used to resolve an issue, the number of discussions by email or the number of defects. We chose not to include them for the following reasons: The time elapsed from a ticket was open until it was closed can easily contain slack time that does not reflect the actual effort. A ticket may, for example, remain open for a long time because of its low priority rather than its inherent difficulty. It is possible that discussions of an issue can happen in developers' mailing list. However, there is no reliable and automatic way to link email discussions with the files involved in the maintenance activities being discussed. Bacchelli et al. [34] have made progress recently on the linkage between email contents and source code. As part of our further work, we would like to further strengthen our results by including discussions in emails if such tools are available and reliable.

There are many other works that use similar measures, such as the number of changes [4], [6], to predict defects. By contrast, the main focus of our work is the direct measure of maintenance *effort*, in which defect fixing is only a part. If a defect is fixed, then the fix is usually manifested as changes in code, patches, commits, and discussions, which will be captured by our three effort measures. These effort measures can also capture the effort spent on other kinds of maintenance activities. However, if an issue is not resolved, our approach won't count it as incurring any effort.

We have also assumed that all bugs/change requests that were resolved between one release and the next were attributable to the latest release. However, it is possible that a few of the bugs or change requests actually applied to an older release that is still being maintained. These bugs or change requests are described as *backports* by Bachmann et al. [35]. We also ignored files that had more than one class defined in them. However, as we showed in section III-B1, only 7% of the files are excluded for this reason. Another threat to validity is that we only examined open source projects. While we expect that similar results will be obtained in industry projects, we cannot guarantee it.

Furthermore, we have only investigated 5 projects and we only investigated a subset of the possible metrics that we could have considered. For example, we could have considered PC metrics with different path lengths and different decay factors. A larger study employing more projects and more metric types would improve the validity of our conclusions.

B. Future work

In future work, we intend to flesh out this research framework with data from more projects. We will also focus on collecting data from commercial (non open source) projects, including projects with logged hours for maintenance work. These logged hours will also allow us to verify our current assumptions about our proxy maintenance measures. We will also explore more maintenance effort proxies in real industrial settings when efforts in terms of logged hours are not available. It is possible that our maintenance effort proxy measures will correlate differently in industrial settings.

As mentioned above, there are many extraneous project factors such as the inherent difficulty of bugs and issues, and the inherent skill of the developers that add noise to the data. Further study that controls for some of these noise factors is thus called for. For example, we are already seeing some promising results by categorizing change requests into groups of small, medium, and large requests. This way, we can see what the influence of source code metrics is on the different sizes of changes, and eliminate the noise that this variation in change size adds to the data.

We are also interested in further research on the correlation of source code metrics to fault rates, using our new framework, specifically to look at long-term reduction in fault rates after refactoring, and the possible introduction of faults during refactoring effort.

Another research topic could be on different variations of the PC metric. An option is for example to test a set of other decay factors than 0.10, and compare the results against non-decaying PC.

Finally, when we have a balanced data set that contains more maintenance cost measures and both open source and industrial projects, we will work on constructing a polynomial model that predicts maintenance cost based on predicted values of source code metrics.

VII. Conclusion

In this paper, we have introduced a new set of proxy measures of maintenance effort, including the amount of discussion, churn, and actions taken to resolve issues. We have also introduced a new way of examining the relationships between file-level code metrics and maintenance effort (and hence cost) by investigating whether and how variations in such metrics correlate with variations in maintenance effort. From this study, we have identified a set of file-level source code metrics that are strongly correlated with our proposed measures of maintenance effort. Some of these metrics are well-known and already empirically well-justified. Others–principally the variants of propagation cost–have not been broadly and empirically verified until now. We have also shown that the best measures of a project's complexity may be project-specific, which underlines the need to have a large corpus of projects and project characteristics, so that the most appropriate metrics and measures can be selected for estimation purposes.

The framework proposed in this paper has the potential to help predict future maintenance cost based on changes in source code metrics. The maintenance effort measures proposed support our long term vision of explicitly estimating the value of costly maintenance activities, such as refactoring, that are currently hard to quantitatively justify.

Acknowledgment

This work was supported in part by the National Science Foundation under grants CCF-0916891, CCF-1065189 and CCF-1116980.

Part of the work has been carried out as a Master thesis of the VU University Amsterdam, in the context of the Double Degree GSEEM Program on Global Software Engineering.

References

[1] K. Molokken and M. Jorgensen, “A review of software surveys on software effort estimation,” in *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, Sept. 2003, pp. 223 – 230.

[2] B. P. Lientz, “Issues in software maintenance,” *ACM Comput. Surv.*, vol. 15, pp. 271–278, Sept. 1983.

[3] D. Parnas, “Software aging,” in *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, May 1994, pp. 279 –287.

[4] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE ’05. New York, NY, USA: ACM, 2005, pp. 284–292.

[5] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, “Predicting faults from cached history,” in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE ’07, Washington, DC, USA, 2007, pp. 489–498.

[6] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09, Washington, DC, USA, 2009, pp. 78–88.

[7] A. Mockus, D. M. Weiss, and P. Zhang, “Understanding and predicting effort in software projects,” in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE ’03, Washington, DC, USA, 2003, pp. 274–284.

[8] M. Jorgensen and M. Shepperd, “A systematic review of software development cost estimation studies,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 33–53, Jan. 2007.

[9] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[10] T. McCabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308 – 320, Dec. 1976.

[11] L. Briand, P. Devanbu, and W. Melo, “An investigation into coupling measures for c++,” in *Software Engineering, 1997., Proceedings of the 1997 (19th) International Conference on*, May 1997, pp. 412 –421.

[12] A. MacCormack, J. Rusnak, and C. Y. Baldwin, “Exploring the structure of complex software designs: An empirical study of open source and proprietary code,” *Manage. Sci.*, vol. 52, pp. 1015–1030, July 2006.

[13] S. Chidamber and C. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476 –493, Jun 1994.

[14] K. D. Welker, P. W. Oman, and G. G. Atkinson, “Development and application of an automated source code maintainability index,” *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 3, pp. 127–159, 1997.

[15] S. C. Misra, “Modeling design/coding factors that drive maintainability of software systems,” *Software Quality Control*, vol. 13, no. 3, pp. 297–320, Sep. 2005.

[16] Y. Zhou and B. Xu, “Predicting the maintainability of open source software using design metrics,” *Wuhan University Journal of Natural Sciences*, vol. 13, pp. 14–20, 2008.

[17] R. Harrison, S. J. Counsell, and R. V. Nithi, “An investigation into the applicability and validity of object-oriented design metrics,” *Empirical Softw. Engg.*, vol. 3, no. 3, pp. 255–273, Sep. 1998.

[18] E. Arisholm, “Empirical assessment of the impact of structural properties on the changeability of object-oriented software,” *Information and Software Technology*, vol. 48, no. 11, pp. 1046 – 1055, 2006.

[19] W. Li and S. Henry, “Object-oriented metrics that predict maintainability,” *Journal of Systems and Software*, vol. 23, no. 2, pp. 111 – 122, 1993.

[20] A. B. Binkley and S. R. Schach, “Inheritance-based metrics for predicting maintenance effort: An empirical study,” Computer Science Department, Vanderbilt University, Tech. Rep. TR 9705, 1997.

[21] M. P. Ware, F. G. Wilkie, and M. Shapcott, “The application of product measures in directing software maintenance activity,” *J. Softw. Maint. Evol.*, vol. 19, no. 2, pp. 133–154, Mar. 2007.

[22] P. Anbalagan and M. Vouk, “On predicting the time taken to correct bug reports in open source projects,” in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, Sept. 2009, pp. 523 –526.

[23] S. Demeyer and S. Ducasse, “Metrics, do they really help,” in *Proc. Langages et Modèles à Objets LMO (LMO’99)*, Paris, France, 1999, pp. 69–82.

[24] M. Alshayeb and W. Li, “An empirical validation of object-oriented metrics in two different iterative software processes,” *IEEE Trans. Softw. Eng.*, vol. 29, no. 11, pp. 1043 – 1049, Nov. 2003.

[25] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, May 2010, pp. 31 –41.

[26] J. Brooks, F.P., *The Mythical Man-Month, Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.

[27] J. Dolado, “On the problem of the software cost function,” *Information and Software Technology*, vol. 43, no. 1, pp. 61 – 72, 2001.

[28] K. Z. Michalis Xenos, D. Stavrinoudis and D. Christodoulakis, “Object-oriented metrics - a survey,” in *Proceedings of the FESMA 2000, Federation of European Software Measurement Associations*, Madrid, Spain, 2000.

[29] M. Riaz, E. Mendes, and E. Tempero, “A systematic review of software maintainability prediction and metrics,” in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, Oct. 2009, pp. 367 –377.

[30] J. Carriere, R. Kazman, and I. Ozkaya, “A cost-benefit framework for making architectural decisions in a business context,” in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 2, May 2010, pp. 149 – 157.

[31] E. H. Ferneley, “Design metrics as an aid to software maintenance: An empirical study,” *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 1, pp. 55–72, 1999.

[32] H. Yang and E. Tempero, “Measuring the strength of indirect coupling,” in *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, April 2007, pp. 319 –328.

[33] V. Basili, L. Briand, and W. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751 –761, Oct 1996.

[34] A. Bacchelli, T. Dal Sasso, M. D’Ambros, and M. Lanza, “Content classification of development emails,” in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012, Piscataway, NJ, USA, 2012, pp. 375–385.

[35] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, “The missing links: bugs and bug-fix commits,” in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE ’10. New York, NY, USA: ACM, 2010, pp. 97–106.