On the Search for Proxy Measures of Effort

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

***Abstract*—Obtaining detailed measures of effort in software projects is difficult. The vast majority of developers in open-source projects do not report their effort. And in closed-source (proprietary) software, when developers track effort, they typically do so at the level of system, sub-system, task, or feature. Furthermore this proprietary data is seldom made public. This is a problem for researchers who are interested in the relationships between software quality and effort. The raw data on which an analysis could be based is simply not available. In this paper we propose, justify, and provide empirical evidence for the adoption of three proxy measures for effort, based on data that is freely available from open-source projects. We call these measures actions (the number of patches and commits made to a file in addressing an issue), churn (the number of lines of code changed in addressing an issue), and discussions (the number of textual comments associated with a file in addressing an issue). We first motivate the use of these three measures, and then analyze 5 Apache projects to gain insight into how these measures behave, and their relationships to software complexity. We conclude by showing that these three proxy measures of effort are complementary with each other, and their variations are significantly correlated with the variation of canonical file metrics, lending support to the hypothesis that these are truly operationalizing effort.**

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

I. Introduction

Estimating the effort, cost and complexity of software development activities is of vital importance for IT management, but notoriously challenging to do well. Molukken and Jorgensen [1] showed that 60-80% of software projects run over budget by 33% on average. During the maintenance phase, software ages and the code base gets cluttered by an accumulation of changes, often referred to as *technical debt* [2]. When technical debt is not addressed, further development will be hindered. A 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 [3], [4], or to predict project development cost and effort [5], [6]. 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?*

While the costs of modularization activities such as refactoring are significant and immediate, their benefits are largely 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 of *a source code file* and the variation of its maintenance effort. A big impediment to creating such an association is that the raw data on which an analysis could be based is simply not available. The vast majority of developers in open-source projects do not report their effort. And in closed-source (proprietary) software, effort data are generally not publicly available. Open data repositories, such as Promise[1], have made significant contribution by allowing projects to donate data—including effort data—to be used by researchers. The way effort data are typically recorded, however, makes them insufficient to answer the above question.

First, effort is usually measured using units of person-days or person-months and at the level of system, sub-system, task, or features, but it is difficult to attribute these effort measures to *files*. Even for commercial systems, person-days spent on maintaining individual files are not recorded. Consequently, even if file metrics indicate modularity degradation, the penalty caused by the structural degradation of a set of files is hard to be distinguish, quantify, or link to extra budget or effort spent on these files. We need file-level effort metrics that are widely applicable to both open-source and closed-source projects.

Second, prior work has demonstrated a strong correlation between file complexity and quality. File metrics, such as complexity, coupling and cohesion, have long been used to predict maintainability [7]. Still, 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 goal is 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 spent on maintaining the code—and hence cost. As the first step, in this paper, we propose, justify, and provide empirical evidence for the adoption of three *proxy* measures for effort, based on data that is freely available from open-source projects. We hypothesize that the following proxy measures

[1] https://code.google.com/p/promisedata/

have the potential to bridge the gap between file complexity variation and associated maintenance cost calculation:

- *actions*: the number of patches and commits made to a file in addressing an issue,
- *churn*: the number of lines of code changed in addressing an issue, and
- *discussions*: the number of textual comments associated with a file in addressing an issue.

We begin by motivating and justifying the choice of these three measures. We motivate the use of actions and churn by appealing to prior art in software cost estimation. We motivate the use of discussion as a measure of effort by applying the Grounded Theory Method to open-source software. We then analyze 5 Apache projects to gain insight into how these measures behave, and to understand their relationships to software complexity. We conclude by showing that these three proxy measures of effort are complementary to each other, and that their variations are significantly correlated with the variation of file complexity, lending support to the hypothesis that these are truly operationalizing effort.

II. Measures of Effort

We have considered a number of factors in our search for appropriate measures of effort, including granularity, credentials, and rationale. Measures of effort, to be useful for our goals of making economic decisions about modularity and refactoring, need to be defined at the appropriate level of granularity. For our purposes, that is the *file* level, since source files contain modularity decisions and source files are what developers re-modularize and refactor. Software project effort, however, is traditionally measured using person-days or person-months and is allocated to "tasks" in a work-breakdown structure, which is far too coarse-grained to make file-level decisions. We propose the aforementioned three effort proxies based on the following rationale.

A. Proposed Effort Measure Suite

Churn. Perhaps the most widely and longest used measure of effort in software engineering is lines of code. While this measure of effort is not without problems, lines of code has been a common output of software effort estimation models such as Function Points, Use Case Points, and the CoCoMo family of models [8], [9] for nearly 3 decades. So we chose to include lines of code added, delete, or changed, commonly referred to as *churn*, as *one* type of measure of effort. Recently *churn* has been intensively investigated to predict software defects [10], [11].

Actions. Although code churn has the advantage that it can be clearly identified and counted, even a naive programmer knows that not all lines of code are equal, and some require far more effort to create or understand than others. When considering that lines of code represent effort, we realized that the number of changed (modified, added, deleted) lines of code between one version of a file and the next may not accurately represent the effort that went into the modification. In open-source software (OSS) projects, many *patches* are proposed by contributors, each of which may add, delete, or modify some code in a set of source files. Some of these may be accepted and others may eventually be rejected in the eventual *commit*. Simply looking at the lines of code that changed between one version of a file and another would not capture the full complexity of all of the patches, which after all represent effort. Hence, we reasoned, that counting the patches and commits may be another valid and complementary measure of effort: the more patches that are required to address a change or fix a bug, the greater the effort. We term this measure of effort *actions*.

Discussions. Finally, as pointed out above, not all lines of code are equal. Every experienced programmer has had times when they struggled over 20 lines of code, or when they created 500 lines of code with relative ease. Part of the reason for this disparity is the inherent complexity of the issue being addressed, or the programmer's understanding of the issue. In an OSS project such issues are *discussed* among the contributors and committers. Hence, we reasoned, counting the amount of discussion associated with a file might be another valid and interesting measure of effort. We call this measure of effort (not surprisingly) *discussions*, and our reasoning is based on a Grounded Theory investigation.

Grounded Theory (GT) was first conceived and described by Glaser and Strauss [12] as a systematic methodology in the social sciences to discover theory through data analysis. In GT a theory is discovered by the creation and linking of concepts from collected, coded data. From these base concepts, categories are formed as the foundation for the creation of a theory, or the formation of hypothesis. We found that the characteristics of OSS make it an easy fit for the GT methodology. Because OSS development is largely distributed, its activities are logged in textual form via issue tracking systems, commit messages and mailing lists. These textual corpuses formed the dataset for our GT investigation. We first investigated the discussions in Apache Lucene[2], and later expanded our dataset to include multiple Apache projects. The raw source for the analysis consisted of commit titles, JIRA archives, IRC discussions and mailing lists.

After applying several GT techniques, we first obtained base categories, that is, groups of similar concepts that are used to generate a theory. Subsequently we identified additional projects to analyze as a means of augmenting the data and refining the derived categories. In this way, and without premeditation, a *core category* (theory)—*Iterative Informed Consent (IIC)*—emerged. IIC identifies the behavior pattern of contributors iteratively discussing and providing solutions (tests and patches) to finally achieve consent. Once consent is achieved, voting may occur and decisions may be taken (e.g. open an issue in JIRA, submit a commit), or not (e.g. the topic may be moved to a user or dev mailing list for further discussion). Once the core category was identified, we noted that many forms of evidence supported it. IIC can be observed in many OSS communities: an iterative discussion leads to the participants gathering enough facts and opinions such that they can reason over for a final decision (a commit). Specific patches would appear as part of proposed implemented solutions, to clarify or strengthen the chances of consent. Of course, this is not *always* the case–a solution might be directly implemented by a developer with commit

[2]http://lucene.apache.org/core/

permission–but it is the norm. It is our belief, however, that the results of our GT-based investigation provide evidence for the choice of *discussions* as another effort proxy.

B. Other Effort Measure Candidates

We considered several other possible maintenance effort proxies, such as the time used to resolve an issue and the number of defects associated with a file. We chose not to include time because the time elapsed from when a ticket is open until it is closed often contains 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.

Different from other defect-prediction work [4], [10], we choose to not use the number of defects as effort proxy for two reasons. First because it is possible that not all defects are addressed, and there are duplications between bug reports. Second, when a defect is fixed, that fix is usually manifested as changes in code, patches, commits, and discussions, which will be captured by our three effort measures. On the other hand, if an issue is not resolved (there is no churn), if there are no attempts to solve it (no actions), and if it has not been discussed, then there is no evidence that effort has been spent on these defects. Counting the number of defects is thus not sufficient to capture the effort caused by code complexity.

III. Research Questions

To validate that the three proposed effort proxies, we investigate the following research questions.

Q1: Are these three effort proxies correlated with each other?
We first were interested in observing whether there was any consistent relationship among our three proxy measures. A consistent relationship would be evidence that they were really measuring the same thing. Our expectation is that these effort proxies complement each other, rather than predict each other.

Q2: Is the variation of file complexity correlated with the variation of these proxies?
The answer to this question will help us validate the proposed effort proxy by checking if they indeed reflect the effort caused by complexity. We will examine the variation of the selected metrics and their correlation to the variation of the three proxy measures on a per-file basis, using statistical models.

Q3: Does the correlation between file-level code metrics and the effort proxies differ between projects?
The answer to this question will help us understand if the types of maintenance effort that are best correlated with file complexity are mostly project-specific.

IV. Related work

As previously stated, our research goal is to find an effective way to measure maintenance effort at the file level, so that the costs of code structure degradation can be more precisely calculated. To situate our contribution we will first review prior research on cost estimation models and then examine the research on correlating file metrics to maintenance effort. We also discuss how our approach differs from prior work.

A. Research on cost estimation models

Software cost estimation has been investigated for decades. Early cost estimation models include COCOMO [9], SLIM [13], and Function Points [14]. These models took basic software properties as input, such as SLOC [9], [13] and function points [14], and estimate effort using person-months. Recent cost estimation models take into account additional project properties and potential risks, and employ various sophisticated models to predict project effort, such as linear regression [15] and Bayesian network models [16]. These models also estimate costs at the project level in units of person-hours or months. MacDonel et al's study [17] showed that the prediction made from one project's data is hard to generalize to other projects.

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.

Welker et al. [18] proposed a polynomial model 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*. Misra [19] and Zhou and Xu [20] 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.

Harrison et al. [21] 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 [22] 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.

Li and Henry [23] 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 [24] 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. [25] 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 [26] 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.

There is other research that has used similar measures—such as code churn [4], [10]—to predict defects. By contrast,

the main focus of our work is the direct measure of maintenance *effort*, of which defect fixing is only a part. If a defect is fixed, then the fix is manifested as changes in code, patches, commits, and discussions, which will be captured by our three effort measures. However, if an issue is not solved, our approach will not count it as incurring any effort.

C. Differences in our approach

Compared with earlier research, our work is different and novel in the following aspects:

First, we propose proxy measures of effort that work at the *file* level, rather than at project, sub-system, or task level as in prior research [9], [13]–[16]. Our purpose is to advance our ability to reason about maintenance penalties caused by file structure degradation, the first step toward a more accurate technical debt calculation, and the cost-and-benefit analysis needed to justify refactoring activities.

Second, prior work on cost models use person-months or staff hours as the only measures of effort. But the accuracy and availability of staff hours are hard to ensure even in commercial software projects, and impossible to obtain in open source projects. We propose a proxy measure suite with three complementary dimensions that are both obviously correlated with effort in terms of time, but also readily available in most modern projects employing issue tracking and version control systems, either open source projects or proprietary projects.

Third, as the first step towards a new way of measuring effort, we are not proposing any prediction models; this is our future work. Instead, in this paper, we justify the possibility and necessity of using multi-dimensional, complementary measures to assess the effort spent on maintaining a file.

V. Data Collection Methodology

In this section we describe our data collection methodology, project selection criteria, and metric selection criteria for investigating our three research questions.

A. The selection of subject projects

The selection of projects is important to the quality of the data, and therefore to the validity of the research. 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:

a. 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.

b. 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.

c. 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 [28], but many have argued that larger teams are better for productivity and quality in OSS.

d. Variation in project age: The age of a body of software can influence developer productivity in ways that may not be measurable by file 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.

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 you can see from the table, Derby[3], Lucene[4], PDFBox[5], Ivy[6], and FtpServer[7] are from different domains, with different team sizes and different project ages. The number of resolved issues ranges from 329 to 3058.

B. The selection of file metrics

Numerous metrics (summarized in [29], [30]) have been proposed that are purported to predict software quality and maintenance effort. To validate the three effort proxies, we select a set of metrics against 3 criteria.

a. 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.

b. 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 this 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.

c. The metric has been consistently validated 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.

We thus have selected the following metrics:

1. 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.

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

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

[3] http://db.apache.org/derby/
[4] http://lucene.apache.org/core/
[5] http://pdfbox.apache.org/
[6] http://ant.apache.org/ivy/
[7] 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

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

5. 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.

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

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

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

9. 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 [31]. We have altered their definitions slightly to make them meaningful at the file level. The C&K suite has been studied heavily, and its effectiveness of predicting software maintainability have been validated in many studies (e.g. [19], [21], [23]–[25]).

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 [32], is a coupling-based metric, and there has been promising research on the predictive power of PC on maintenance effort [32], [33].

PC is based on a *visibility matrix* [32], 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, PC 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 [34] and Yang and Tempero [35] who have found promising results.

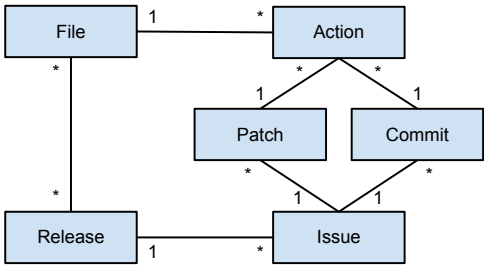

Fig. 1. The data model

We have also introduced a new variant of PC 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 PCs with path lengths of 1, 3, 5, 10, and 20 – with and without decay. When describing our results, we will use a naming convention; for example, PROP-OUT-10-N indicates outgoing PC with path length 10 and without decay, whereas PROP-IN-5-D indicates incoming PC with path length 5 and with decay. Considering all combinations, we have a total of 18 variants of PC 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. We discuss the potential bias when linking commits to issues in Section VIII.

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

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[8], which has a WSDL API that is usable for data extraction. Their version control system is Subversion[9]. We constructed a number of tools that query both systems, format the data, and insert it into our database.

Calculating file metrics We used 3 different programs to calculate file metrics. To calculate the source Lines of Code (LOC), we used the utility *SLOCCount*[10]. 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[11]. For the propagation cost based metrics, we wrote our own tool, which first generates a Dependency Structure Matrix (DSM) [32], 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.

D. Calculating effort proxies

We calculate our effort proxies as follows:

Discussions. 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 V-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.

VI. STATISTICAL ANALYSIS

The data collection methodology described in section V gives us a data set consisting of file metrics and effort proxy measures for each issue, for each release of each file. Since our objective is to analyze the relationship between the *increasing* or *decreasing* of file metrics and the *variation* of maintenance effort, we transform these numbers so that they reflect the *change* in file metrics and effort measures from one release to the next.

To this end, for each file metric or effort proxy value of each file in a release, we 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. For effort proxy measures, 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 (The group column will be explained in section VI-A). Each row of the table contains the data for one issue. 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 file 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,

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

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 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.

VII. Results

Since we have performed a Spearman analysis on a set of 81 (27 x 3) file metrics versus effort proxies, 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 organize the results based on the answers to the research questions proposed in Section III:

Q1: Are these effort proxies correlated with each other?

Aggregating data from all the projects, we obtained 9 significant correlations (rho $>$ 0.3, p $<$ 0.01, and a sample size of at least 15) between effort proxies, as shown in Table III. It shows that churn and discussion are often correlated, but churn and actions are never correlated in the aggregated data set. Table IV contains the three most significant correlations among effort proxies for each project. Of all the effort proxy pairs, we only observed one significant correlation between churn and actions in Derby, with p = 0.00114, rho = 0.43 (not shown in Table IV since it was not one of the three highest correlations). Other than that, churn and actions do not appear to be significantly correlated in most cases. On the other hand, churn is most frequently observed to be significantly correlated with discussions, which is also somewhat correlated with actions. These correlations vary between different projects. For example, in PdfBox, only one group shows significant correlation, which is between discussion and actions.

TABLE III. EFFORT - EFFORT CORRELATION FOR THE AGGREGATED DATA SET

Group	Effort	Effort	p	rho
1	churn	discussion	0.00000	0.3907
2	churn	discussion	0.00000	0.4615
3	churn	discussion	0.00000	0.4898
4	churn	discussion	0.00000	0.4668
5	churn	discussion	0.00000	0.4618
6	churn	discussion	0.00002	0.4068
7	churn	discussion	0.00101	0.3510
8	discussion	actions	0.00522	0.3627
9	discussion	actions	0.00173	0.4224

TABLE IV. EFFORT - EFFORT CORRELATION FOR INDIVIDUAL PROJECTS

Project	Grp	Effort	Effort	p	rho
Derby	2	churn	discussion	0.00000	0.51
Derby	4	churn	discussion	0.00022	0.55
Derby	4	discussion	actions	0.00001	0.59
Lucene	3	churn	discussion	0.00000	0.42
Lucene	5	churn	discussion	0.00000	0.44
Lucene	7	churn	discussion	0.00026	0.43
PDFBox	2	discussion	actions	0.00128	0.42
Ivy	2	churn	discussion	0.00000	0.45
Ivy	3	churn	discussion	0.00000	0.54
Ivy	5	churn	discussion	0.00915	0.55
FtpServer	1	churn	discussion	0.00000	0.50
FtpServer	2	churn	discussion	0.00000	0.55
FtpServer	4	churn	discussion	0.00001	0.78

The correlation of discussion with churn or actions supports the *IIC* theory: it is expected that more actions and more churn is the result of IIC, as generated by more discussions. That is, the effort spent on discussions should not be neglected. More interestingly, actions and churns are rarely correlated, which is counter-intuitive: one may think that more actions will result in more changes in LOC. But this result implies that in many cases, a single commit may change a lot of code, or more patches may be committed to just change a small amount of code. In other words, changing fewer LOC may require more effort reflected by the multiple attempts (patches). This also implies that using just one proxy measure of effort, say, churn, is not sufficient to capture the full complexity of file maintenance effort.

Q2: Is the variation of file metrics correlated with the variation of effort proxies?

Aggregating data from all the projects, we obtained 17 significant correlations between file metrics and effort proxies.

We also 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 this research question, we report significant results ($p < 0.01$) obtained from the aggregated data set and from individual project data in Table V and Table VI. In Table V, 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 proxies: actions and churn. In addition, it is interesting to note that 7 of the top 10 results show a strong correlation with coupling-based metrics: 4 variants of PC and 3 C&K metrics (Ca, CBO, and RFC).

The values in Table V 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 software structure will suffice. 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 V.

LOC, several variants of PC as well as RFC, CBO, WMC, NPM, and Ca have been shown to be significantly correlated to the actions and churn proxies. Previous work has shown that the 8 metrics from the C&K metric suite are good predictors of maintenance effort [23], [24], [36]. These effort measures appear to be valid proxies for effort, in the sense that they are correlated with complexity measures that have previously been shown to be strongly correlated with effort in other studies.

Q3: Does the correlation between file metrics and the effort proxies differ between projects?

In Table VI, 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 Q3 asked whether these correlations would differ between projects. The project-by-project results shown in both Table VI and IV suggest that this is indeed the case. While actions and churn are 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. This result is consisten with recent research results showing that predictions obtained from one project data are not generalizable to other projects [37].

Summary. Considering Table III, IV, V and VI together, we observe that although file metrics are more often correlated with churn and actions, but not discussions, churn and actions, in turn, are often correlated with discussions. This means that file metrics and discussions impact each other in a related, but indirect manner. This result implies that the three effort measures are complementary, and that each measures file maintenance effort from different but correlated perspectives.

TABLE V. METRIC - EFFORT CORRELATION FOR THE 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

TABLE VI. 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

VIII. DISCUSSION

A. Threats to Validity

As with any empirical study, our results are subject to a number of threats to validity. Below we discuss the most significant ones: the threats to validity caused by the choice of maintenance effort measures, the choice of projects to study, and the the limitations of our data extraction methods.

We employed the three types of file-level effort measures because, as stated earlier, developers in open-source projects do not log their hours. Even in industrial settings, accurate effort logs are rare. Moreover, we need maintenance effort at the *file level*, which is even rarer. We hypothesize that our 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 given our existing data-set.

In our calculation of *discussions* as a measure of effort, we did not mine the projects' mailing lists. We avoided the mailing lists because it is difficult to accurately link backwards from a mail message to a specific file/release/issue combination. But this means that we are not able to accurately assess the full body of discussions associated with a project, and hence we are potentially biasing our *discussions* measure.

As described earlier, 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 associated with a file. We chose not to include these other possible proxy measures for various reasons: the inability to distinguish work time from slack time, our inability to reliably and automatically link email discussions with files,

and the difficulty of attributing a bug fix to a specific release.

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. [38].

The effort proxies we proposed are based on the assumption that the project employs a version control and issue tracking system from which the effort proxies can be extracted. As a result, the proxies won't be applicable for projects without comparable tools. More importantly, as with any research replying upon bug fix data, our study needed to link files with issues by finding developer's commit messages indicating which issue the commit is intend to address. As Bird et al. [39] have pointed out, in many projects, only a small portion of bug fixes are actually recorded in version control systems. Since our study compares the behavior of the three proxies using the same dataset, the extent to which our results are biased due to the incompleteness of the dataset needs to be evaluated.

Another threat to validity is that we only examined open-source projects. While we expect that similar results will be obtained from industry projects, this is an open research question. And within our chosen projects, we ignored files that had more than one class defined in them. However, as we showed in section V, this only excludes 7% of the files. We have only investigated 5 projects thus far, all of which were from the Apache foundation, 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.

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 the inherent variation in change complexity adds to the data.

B. Future Work

In future work, we intend to flesh out this research framework with data from more projects and we intend to address the threats to validity outlined above. We will also focus on collecting data from closed-source projects, ideally including projects with logged hours for maintenance work. We have already identified two industrial projects and are negotiating with the companies for access to their data. Assuming we are granted access, their logged effort data will allow us to validate our proxy effort measures. We also intend to explore the use of our maintenance effort proxies in real industrial settings when efforts in terms of logged hours are *not* available. It is possible that our maintenance effort proxies will correlate differently in industrial projects than in open-source projects.

When we have a broader data set that contains maintenance cost measures from both open-source and industrial projects, we will work on constructing a polynomial model that predicts maintenance effort in terms of person-days (and hence cost) using the three effort proxies. The model will be configurable based on project characteristics and on source code metrics. We believe that the three proxies proposed in this paper make it possible to uniformly map complementary but heterogeneous manifestations of effort into one unified measure, i.e. person-days. For example, discussions recorded in a project usually are timestamped, the time spent on churn and actions can be possibly inferred by mining more detailed file revision information. In the envisioned model, the three proxies can be weighted differently in different projects. For example, the discussion effort should be weighted differently in OSS than in a commercial project where all developers are co-located.

Finally, we may be able to address the threat to validity that arose because we were not processing email discussions. Bacchelli et al. [40] have made progress 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. Our long-term goal is to be able to support project-level decisions about whether, when, and where to invest in refactoring and re-modularization activities. Currently projects do this via gut feelings and hunches; we aim to provide a sound empirical basis for such decisions.

IX. Conclusion

In this paper, we contribute three complementary file-level effort proxies: the amount of discussion, churn, and actions taken to resolve issues. These proxies are validated by the fact that their variations are significantly correlated with variations of well validated file metrics. Our study also shows that 1) different metrics are strongly correlated with different effort proxies, 2) no regular, consistent correlations among these proxies were observed, and 3) that discussions are often correlated with churn and actions. These results indicate that using only one dimension, such as churn, is not sufficient to capture the overall effort spent on maintaining a file. Instead, effort estimation should consider complementary factors, such as the effort spent on discussions and multiple patches.

Our next step is to map these three complementary measure of effort into a single effort measure, e.g, the amount of time spent on the churn, patches and discussions for a file, taking into account properties of specific projects. The effort proxies proposed in this paper have the potential to help predict future maintenance cost based on changes in source code metrics without the necessity of recording person-days of effort at the file level. 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 economically justify.

Acknowledgment

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

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] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 47–52.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] C. F. Kemerer, "An empirical validation of software cost estimation models," *Commun. ACM*, vol. 30, no. 5, pp. 416–429, May 1987.

[9] B. W. Boehm, *Software Engineering Economics*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[10] 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.

[11] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 83–92.

[12] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. New York, NY: Aldine de Gruyter, 1967.

[13] P. L. and A. Fitzsimmons, "Estimating software costs," *Datamation*, vol. 25, pp. 10–12, 1979.

[14] A. A.J. and J. Gaffney. J., "Software function, source lines of code, and development effort prediction: A software science validation," *IEEE Trans. Softw. Eng. SE*, vol. 6, pp. 639–646, 1963.

[15] M. Jørgensen, "Experience with the accuracy of software maintenance task effort prediction models," *IEEE Trans. Softw. Eng.*, vol. 21, no. 8, pp. 674–681, Aug. 1995.

[16] P. C. Pendharkar, G. H. Subramanian, and J. A. Rodger, "A probabilistic model for predicting software development effort," *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 615–624, Jul. 2005.

[17] S. G. MacDonell and M. J. Shepperd, "Comparing local and global software effort estimation models – reflections on a systematic review," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 401–409.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 343–351.

[38] 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.

[39] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130.

[40] 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.