Comparing Four Approaches for Technical Debt Identification

Nico Zazworka[1], Antonio Vetro'[1,2], Clemente Izurieta[3], Sunny Wong[4], Yuanfang Cai[5], Carolyn Seaman[1,6], Forrest Shull[1]

[1]Fraunhofer CESE
College Park, MD, USA
nzazworka@fc-md.umd.edu
fshull@fc-md.umd.edu

[2]Automatics and Informatics Dept.
Politecnico di Torino
Torino, Italy
antonio.vetro@polito.it

[3]Dept. of Computer Science
Montana State University
Bozeman, MT, USA
clemente.izurieta
@cs.montana.edu

[4]Siemens Healthcare
Malvern, PA, USA
sunny.wong@siemens.com

[5]Dept. of Computer Science
Drexel University
Philadelphia, PA, USA
yfcai@cs.drexel.edu

[6]Dept. of Information Systems
UMBC
Baltimore, MD, USA
cseaman@umbc.edu

ABSTRACT

Background: Software systems accumulate *technical debt* (TD) when short-term goals in software development are traded for long term goals (e.g., quick-and-dirty implementation to reach a release date vs. a well-refactored implementation that supports the long term health of the project). Some forms of TD accumulate over time in the form of source code that is difficult to work with and exhibits a variety of anomalies. A number of source code analysis techniques and tools have been proposed to potentially identify the code-level debt accumulated in a system. What has not yet been studied is if using multiple tools to detect TD can lead to benefits, i.e. if different tools will flag the same or different source code components. Further, these techniques also lack investigation into the symptoms of TD "interest" that they lead to. To address this latter question, we also investigated whether TD, as identified by the source code analysis techniques, correlates with interest payments in the form of increased defect- and change-proneness.

Aims: Comparing the results of different TD identification approaches to understand their commonalities and differences and to evaluate their relationship to indicators of future TD "interest".

Method: We selected four different TD identification techniques (code smells, automatic static analysis (ASA) issues, grime buildup, and modularity violations) and applied them to 13 versions of the Apache Hadoop open source software project. We collected and aggregated statistical measures to investigate whether the different techniques identified TD indicators in the same or different classes and whether those classes in turn exhibited high interest (in the form of a large number of defects and higher change proneness).

Results: The outputs of the four approaches have very little overlap and are therefore pointing to different problems in the source code. Dispersed coupling and modularity violations were co-located in classes with higher defect proneness. We also observed a strong relationship between modularity violations and change proneness.

Conclusions: Our main contribution is an initial overview of the TD landscape, showing that different TD techniques are loosely coupled and therefore indicate problems in different locations of the source code. Moreover, our proxy interest indicators (change- and defect-proneness) correlate with only a small subset of TD indicators.

Categories and Subject Descriptors

K.6.3 [**Software Management**]: Software Maintenance.

General Terms

Management, Measurement, Design, Economics, Experimentation

Keywords

Technical debt, software maintenance, software quality, source code analysis, modularity violations, grime, code smells, ASA.

1. INTRODUCTION

As Cunningham described [1], in a software development project, rushing implementation to meet pending deadlines is "*like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation…*"

Nowadays "technical debt" (TD) has become a well-known metaphor indicating the possibly of significant economic consequences for such quick-and-dirty implementations. Accumulated technical debt may cause severe maintenance difficulty or even project cancellation. To prevent TD from accumulating, or to decide when, where and how to pay off such debt, the first step is to make it explicit, i.e. to identify TD. One important class of TD is manifested by problematic implementations in code. Many types of such code-based TD can be potentially detected automatically using static program analysis tools that find anomalies of various kinds in the source code.

There are a number of tools designed for this purpose. Some tools are designed to detect design problems such as code smells [15] or modularity violations [9], some are designed to discover design pattern degradations [7], and some are intended to spot potential defects. From a tool user's point of view, the relevant questions are: which tool(s) should be used to inform the existence of TD under what circumstances?, and, Is it sufficient to

use one of the tools, or can the usage of multiple tools lead to benefits in finding more TD?

Plus, not all problematic code detected by these tools is worth being fixed. Some detected source code problems are not likely to cause future maintenance problems or affect the overall quality of the system. In terms of the TD metaphor, the *TD principal* (i.e. the cost of fixing the debt) may be higher than the *TD interest* being paid on the debt (i.e. the probable future cost of not fixing it). Thus, another question is: Which tools reveal TD that is likely to incur interest?

TD interest is inherently difficult to estimate or measure. Given the data that we had available in this study, we chose to use two proxies for expected interest (hereafter referred to as "interest indicators"): defect- and change-proneness. These proxies are concrete manifestations of problematic code and are related to future maintenance cost, and therefore are useful, independent indicators of likely interest payments. The proxy measures are well established and have been previously used in the assessment of maintenance problems [10][11][41][42]. However, they are not the only, and possibly not the best, indicators, since they do not capture other forms of TD interest, such as increasing effort to make changes. However, defect- and change-proneness are factors of concern to practitioners, and thus it is relevant to determine if they correlate with the TD indicators generated by the four approaches studied here. Thus, a lack of relationship between our selected interest indicators and a TD indicator cannot clearly be interpreted to mean that the associated TD identification approach is ineffective, but that instead it may identify TD that exhibits other forms of interest.

It should be noted that our aim is not to predict either defects or change-proneness in future instances of the code base, as has been done by many other researchers (e.g., [26][27][28][29][30]). Rather, we are calculating, for a given version of the system, which classes are already exhibiting change- or defect-proneness, and using these indicators as proxies of a construct (TD interest) that is more difficult to measure.

So as a first attempt to compare and contrast different TD identification techniques, we conducted an empirical study to answer the following research questions:

RQ1: Considering a set of four representative TD detection techniques (resulting in a set of 25 "TD indicators"), do they report the same set of modules as problematic? If not, how much overlap is there?

RQ2: To what extent do any of the techniques for detecting TD in code happen to point to classes with high interest indicators (defect-proneness or change-proneness)?

We first compare the results of applying the different TD identification techniques to 13 versions of the Apache Hadoop open source software project.

Secondly, we investigate whether and how likely the problems detected by these different techniques are related to the two interest indicators we have chosen. This study is a first attempt to map out a "TD landscape" that illustrates the overlaps, gaps, and synergies between a variety of code analysis techniques with respect to their use in identifying TD.

Both research questions are answered through the computation and combination of the pairwise relationship between TD indicators (RQ1) and TD indicators vs. interest indicators (RQ2). The aim is to understand which TD and interest indicators point to the same locations (i.e. Java classes).

We use four different statistical association measures: Pearson correlation, conditional probability, chance agreement and Cohen's Kappa. Each of the selected four measures assesses association from different perspectives that complement each other (see Section 4.3.1 for further details).

The TD indicators and interest indicators are computed for each class and each of the 13 selected versions of the Hadoop software: hence a further aggregation is needed to combine measures from all versions and have a unique association measure for every possible combination of TD indicators (RQ1) or TD indicator vs interest indicator (RQ2).

The results in this paper have the potential to improve our understanding about how existing source code analysis approaches can be tailored towards identifying TD. The long-term vision of this work is to create practical approaches that software developers can use to make TD visible, to help assess the principal (i.e. value) and interest (i.e. long-term cost) of the debt, and to assist managers in making educated decisions on strategies for debt retirement.

2. RELATED WORK

Past research efforts into TD have focused on building techniques that independently identify TD through source code analysis. For instance, Gat and Heintz [3] identified TD in a customer system using both dynamic (i.e., unit testing and code coverage) and static (computing rule conformance, code complexity, duplication of code, design properties) program analysis techniques.

Nugroho et al. [4] also performed static analysis to identify TD. They first calculated lines of code, code duplication, McCabe's cyclomatic complexity, parameter counts, and dependency counts. After that, they assigned these metrics to risk categories to quantify the amount of interest owed in the form of estimated maintainability cost.

A CAST report [5] also presented the usage of static analysis as a way to identify technical debt. The proposed approach examines the density of static analysis issues on security, performance, robustness, and changeability of the code. The authors built a pricing model assuming that only a percentage of the issues are actually being fixed.

The Sonar tool (http://www.sonarsource.org/) is an open source application that has gained in popularity. It also uses static measurements against various source code metrics and attributes to assess the level of TD in a code base.

The approaches discussed thus far calculate TD holistically, i.e. they yield an overall assessment of the total TD in a system, but do not point to specific problematic parts of the code base, or to specific remedies applicable to those parts. Another approach to TD identification, that attempts to yield more actionable information, is to use source code analysis to identify potentially problematic parts of the code, and to use the results of that analysis to suggest specific changes to be made to that code. Examples of such approaches that have been partly evaluated to be valid TD indicators are code smells [6], grime build up [7] [8] and modularity violations [9]. We discuss these techniques in more detail in Section 4.

This works further evolves the study of these analysis techniques by investigating the amount of similarity between them. If it turned out that many, or even all, of the TD indicators point to the same code, one could propose to choose only one of the tools when searching for TD. Alternatively, if each of the techniques selects a unique subset of problems, the usage of

multiple tools can be recommended, where a particular project must make intentional decisions about which types of TD they are most interested in, and choose tools accordingly. The relationships among different TD identification approaches have not previously been addressed in the literature.

3. GOALS AND RESEARCH QUESTIONS

The objective of our research is twofold: the first goal is to compare the similarities and differences between four code analysis techniques in terms of TD identification. We are interested in understanding the degree of convergence and divergence of these techniques and their associations. The second goal is to understand how the problematic code identified by these four techniques relates to our chosen proxies for TD interest, defect and change-proneness. As explained in Section 1 these proxies are well-established and feasible to compute given the available data, but do not constitute a complete assessment of all TD types. We define the goals of our research according to the Goal Question Metric framework [16].

Goal 1: Characterizing the similarities and differences in the problematic classes reported by these four different TD detection approaches, in the context of an open source software project.

Goal 2: Comparing these four TD detection approaches in terms of their correlation to one subset of interest indicators, namely change-proneness and defect-proneness, in the context of an open source project.

We deduced from the above goals the following research questions:

(RQ1) Considering a set of four representative TD detection techniques (resulting in a set of 25 "TD indicators"), do they report the same set of modules as problematic? If not, how much overlap is there?

(RQ2) To what extent do any of the techniques for detecting TD in code happen to point to classes with high interest indicators (defect-proneness or change-proneness)?

4. Case Study

The application studied is Apache Hadoop (http://hadoop.apache.org). Hadoop is a software library for the distributed processing of data across numerous computer nodes, based on the map-reduce processing model. It provides two key services: reliable data storage using the Hadoop Distributed File System (HDFS) and high-performance parallel data processing using a technique called MapReduce. Data are spread and replicated differently among all the nodes of the cluster, while operations are split so that each node works on its own piece of data and then sends results into a unified whole.

We selected Hadoop because it is a mature project (it has been released 59 times starting from 2 April 2006). We focused our analysis on the Java core packages of the system (java/org.apache.hadoop.*), which includes the common utilities that support the other Hadoop subprojects and provides access to the file systems supported by Hadoop. We focused the analysis from release 0.2.0 to release 0.14.0 (the latest release, at the time this paper was written, is 1.0.3). The system initially had 10.5k NCSS (non- commented source statements) and 126 Java classes, and grew to 37k NCSS and 373 Java classes by release 0.14.0.

4.1 TD Identification Techniques Selected

We selected four main techniques for identifying technical debt in source code: modularity violations, grime buildup, code smells, and automatic static analysis (ASA). These approaches have all been studied in previous work by the collaborating authors and institutions [9, 10, 12, 14], and we chose to select these techniques as a proof of concept because the authors are most experienced with installing, using and interpreting the results of these four approaches. This work can be extended in the future to incorporate comparisons to other techniques. This section introduces the basic concepts of our selected techniques, and reports on our and other related past work.

TABLE I. INDICATORS USED IN OUR ANALYSIS

Technical Debt Indicators	Modularity Violations	[1]Presence of Modularity Violation *CLIO*			[0,1]
	Grime	[2]Presence of Grime			[0,1]
		[3]Absence of Design Pattern			[0,1]
	Code Smells *CodeVizard*		Class Level Code Smells	[4]God Class	[0,1]
				[5]Brain Class	[0,1]
				[6]Refused Parent Bequest	[0,1]
				[7]Tradition Breaker	[0,1]
				[8]Feature Envy	[0,1]
				[9]Data Class	[0,1]
			Method Level Code Smells	[10]Brain Method	[0..N]
				[11]Intensive Coupling	[0..N]
				[12]Dispersed Coupling	[0..N]
				[13]Shotgun Surgery	[0..N]
	ASA Issues *FindBugs*		By Priority	[14]High	[0..N]
				[15]Medium	[0..N]
				[16]Low	[0..N]
			By Category	[17]Bad Practice	[0..N]
				[18]Correctness	[0..N]
				[19]Experimental	[0..N]
				[20]I18N (internationalization)	[0..N]
				[21]Malicious Code	[0..N]
				[22]Multi Thread (MT) Correctness	[0..N]
				[23]Performance	[0..N]
				[24]Security	[0..N]
				[25]Style	[0..N]
Other Metrics	Size	[26]Number of Methods *Eclipse Metrics Plugin*			[0..N]
Interest Indicators	Defect Proneness	[27]Number of bug fixes affecting this version			[0..N]
		[28]Number of bug fixes fixed in this version			[0..N]
		[29]Number of bug fixes counting between affected and fixed in this version			[0..N]
	Change Proneness	[30]Change Likelihood			[0.0…1.0]

Modularity Violations (tool: CLIO). In large software systems, modules represent subsystems that are typically designed to evolve independently. During software evolution, components that evolve together though belonging to distinct modules represent a discrepancy. This discrepancy may be caused by side effects of a quick and dirty implementation, or requirements may have changed such that the original designed architecture could not easily adapt. When such discrepancies exist, the software can deviate from its designed modular structure, which is called a *modularity violation.* Wong et al. [9] have demonstrated the feasibility and utility of this approach. In their experiment using Hadoop, they identified 231 modularity violations from 490 modification requests, of which 152 (65%) violations were conservatively confirmed by the fact that they were either indeed

addressed in later versions, or were recognized as problems in the developers' subsequent comments.

Design Patterns and Grime Buildup. Design patterns are popular for a number of reasons, including but not limited to claims of easier maintainability and flexibility of designs, reduced number of defects and faults [17], and improved architectural designs. Software designs decay as systems, uses, and operational environments evolve, and decay can involve design patterns. Classes that participate in design pattern realizations accumulate grime – non-pattern-related code. Design pattern realizations can also rot, when changes break the structural or functional integrity of a design pattern. Both grime and rot represent forms of TD. Izurieta and Bieman [7] introduced the notion of design pattern grime and performed a pilot study of the effects of decay on one small part of an open-source system, JRefactory. They studied a small number of pattern realizations and found that coupling increased and namespace organization became more complex due to design pattern grime, but they did not find changes that "break" the pattern (design pattern rot). Izurieta and Bieman [14] also examined the effects of design pattern grime on the testability of JRefactory, a handful of patterns were examined, and they found that there are at least two potential mechanisms that can impact testability: 1) the appearance of design anti-patterns [18] and 2) the increases in relationships (associations, realizations, and dependencies) that in turn increase test requirements. They also found that the majority of grime buildup is attributable to increases in coupling.

Code Smells (tool: CodeVizard). The concept of code smells (aka bad smells) was first introduced by Fowler [6] and describes choices in object-oriented systems that do not comply with widely accepted principles of good object oriented design (e.g., information hiding, encapsulation, use of inheritance). Code smells can be roughly classified into identity, collaboration, and classification disharmonies [19]. Automatic approaches (detection strategies [20]) have been developed to identify code smells. Schumacher et al.'s research [15] focused on evaluating these automatic approaches with respect to their precision and recall, and their other work [10] [11] evaluated the relationship between code smells (e.g., god classes) and the defect and change proneness of software components. This work showed that automatic classifiers for god classes yield high recall and precision when studied in industrial environments. Further, in these environments, god classes were up to 13 times more likely to be affected by defects and up to seven times more change-prone than their non-smelly counterparts.

ASA issues (tool: FindBugs). Automatic static analysis (ASA) tools analyze source or compiled code looking for violations of recommended programming practices ("issues") that might cause faults or might degrade some dimensions of software quality (e.g., maintainability, efficiency). Some issues can be removed through refactoring to avoid future problems. Vetro' et al. [12][13] analyzed the issues detected by FindBugs [21] on two

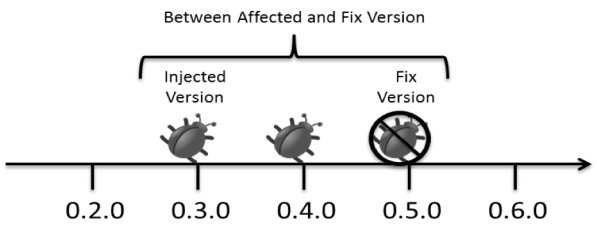

Figure 1: Three ways of computing defect proneness

pools of similar small programs (85 and 301 programs respectively), each of them developed by a different student. Their purpose was to examine which issues detected by FindBugs were related to real defects in the source code. By analyzing the changes and test failures in both studies they observed that a small percentage of detected issues were related to known defects in the code. Some of the issues identified as good/bad defect detectors by the authors in these studies were also found in similar studies with FindBugs, both in industry [22] and open source software [23]. Similar studies have also been conducted with other tools [24] [25] and the overall finding is: a small set of ASA issues is related to defects in the software, but the set depends on the context and type of the software.

4.2 Data Collection

We measured each class of every Hadoop version using the 30 class-level indicators listed in Table I. We call each of the measurements a *data point.* Of all the 30 indicators, 25 (Table I: indicators 1-25) were calculated by the four techniques described in Section 2. We included one additional size metric (Table I: metric 26) to provide a point of comparison. We also considered four software interest indicators (Table I: metrics 27-30), which reflect defect- and change-proneness, for studying our second goal.

For our analysis we considered 13 Hadoop releases. We ignored the very first one (0.1.0) since CLIO's modularity violation computation is based on the current and previous versions. Across the 13 Hadoop releases, from 0.2.0 to 0.14.0, and all 30 indicators over every class, the total size of our data set was 96,720 data points. Due to limitations in the tools used for TD identification we excluded nested classes from our analysis. To understand the threat to validity, we inspected all versions of Hadoop and found that (depending on the version) 39-45% of all classes were nested classes. We will discuss this threat in Section 6.

It should be noted that the range (i.e., possible values) of each indicator varies. As shown in Table I, TD indicators that solely express the presence of TD (e.g., the presence of a modularity violation or a code smell on class level) map to 0 (meaning no presence) or 1 (meaning the indicator is present). This is expressed by [0,1] in Table I. For indicators that can be identified multiple times in a Java class (e.g. code smells on the method level and ASA issues that can be repeatedly detected) the measure indicates how many times the indicator was identified. Table I shows this as [0..N].

We measured the presence of grime as well as the absence of design patterns. Even if we cannot be sure that the absence of design patterns is harmful, we included this information to investigate if we can find interesting relationships. Therefore classes not following design patterns received a value of "1" for the indicator. We collected issues reported by FindBugs (version 1.3.9) from the source code of each Hadoop version, considering all issues of any FindBugs category (Table I: 17-25) and priority (Table I: 14-16).

Defect proneness measurement. To link classes with defects, for a bug that was fixed and closed in a version *v*, we computed which classes were modified during the fix change (identifiable through Subversion repository by using bug links provided in commit comments, e.g. HADOOP-123). The linkage between source code anomalies and their resulting defects is potentially stretched over time. For example, as illustrated in **Error! Reference source not found.**, a bug can be found and reported in version 0.3.0, but may not be fixed until version 0.5.0.

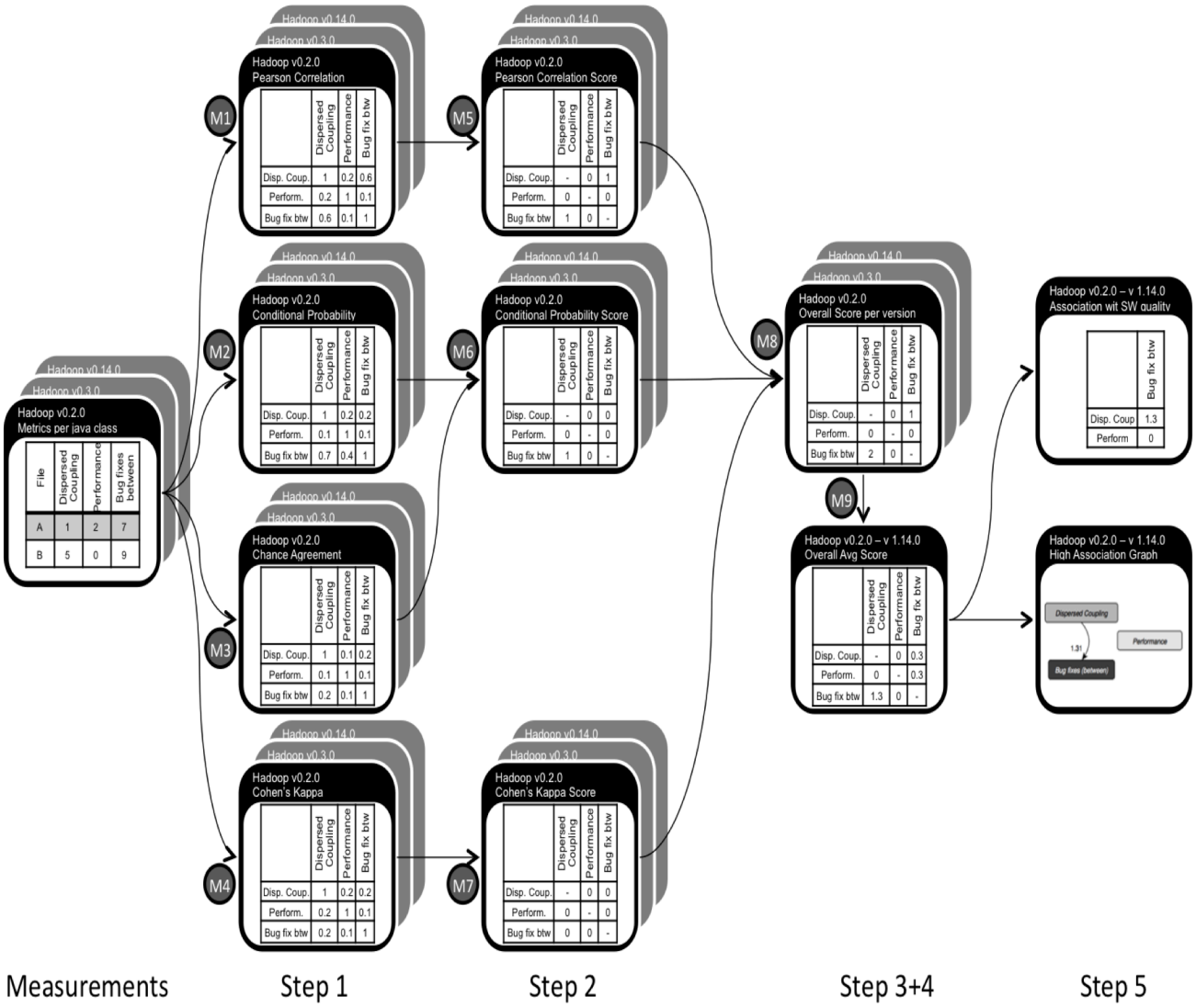

Figure 2: Five-step analysis methodology

We thus measure the defect proneness of a class *c* in version *v* using the following three different ways respectively:

1. The number of times class *c* is involved in fixing bugs that were *injected* in version *v*, that is, the version where the bugs were found and reported.
2. The number of times class *c* is involved in fixing bugs that were *resolved* in version *v*.
3. The number of times class *c* is involved in fixing bugs that were *alive* in version *v*, that is, the bugs were reported before or in version *v*, and were resolved after or in version *v*.[1]

Change proneness measurement. Following the work of Schumacher et al. [15], we measure the change proneness of class *c* in version *v* as the number of repository changes affecting class *c* divided by the total number of changes in the repository during the class' lifetime (e.g., creation to deletion date).

[1] This approach is necessary since versions can overlap (in time) in the SVN repository and single revisions cannot always be clearly assigned to a single version.

Size measurement. We chose the Number of Methods in each class as a measure of size. In Section 6 we discuss possible threats posed by this choice.

4.3 Analysis Methodology

In order to investigate the two research questions proposed in Section 3, that is, the overlap between the results generated by these TD detection techniques (TD indicators), and their correlation with defect- and change- proneness (interest indicators), we designed a 5-step methodology that reduces the complex set of indicator values on the large, multidimensional dataset into a graph. The methodology is illustrated in Figure 2:

Step 1: Compute a set of association measures to examine how each TD indicator and each interest indicator are related to each other.

Step 2: Apply statistical and significance functions to filter only highly associated pairs of indicators.

Step 3: Combine the set of three significant association measures into one measure.

Step 4: Combine measures from each of the 13 Hadoop versions to one set of aggregate measures.

Step 5: Build visualization and data tables to provide insight into the most strongly associated indicators.

To answer the first research question regarding the overlap between the results reported by these techniques, we examine the association measures between their respective TD indicators. To answer the second research question regarding interest indicators, we order the TD indicators (and the one size measure) by their level of association with the four interest indicators introduced in Section 4.2.

4.3.1 Step1: Compute Statistical Association Measures

We identified different statistical techniques to quantify the relationship between pairs of TD indicators in each version of Hadoop. Because there are many choices for association measurement (e.g. Pearson correlation, conditional probability), we performed a sensitivity analysis to investigate how the results generated from these statistic models differ from each other. This analysis showed that different statistical analysis techniques result in different answers: from the top 50 list of the most highly associated pairs of indicators and metrics generated by each of the statistical analysis techniques, only three pairs were common. Therefore we concluded that using only one single measure of association would be inadequate because different statistic models assess association from different perspectives that complement each other. We thus apply 4 different association models to assess the relation between the 30 X 30 pairs of indicators. The different association measures will be combined in one unique measure in Step 3.

In order to illustrate our methodology, we use a pair of TD indicators, Dispersed Coupling (one type of code smell) and Performance issues (reported by FindBugs), as a running example. The four association techniques are described below:

1. The **Pearson correlation** between the number of occurrences of Dispersed Coupling and the number of occurrences of Performance issues across all Java classes of one Hadoop version (Figure 2: Step 1, M1). Pearson correlation is widely used in the literature on defect prediction models [26], including the usage of ASA issues as an early indicator of defects [27], and in maintainability prediction [28].

2. The **conditional probability** of a Java class having at least one occurrence of Dispersed Coupling given that the same class has at least one occurrence of Performance issue, and vice versa: P(Dispersed Coupling | Performance), P(Performance | Dispersed Coupling) (Figure 2: Step 1, M2). The conditional Probability has been used previously for software defects and maintainability predictions [29] [30].

3. The **chance agreement**, which is the probability that a Java class holds an occurrence of Dispersed Coupling and Performance issue at the same time by chance. This probability is computed as: P(Performance)*P(Dispersed Coupling)+P(No Performance)*P(No Dispersed Coupling) (Figure 2: Step 1, M3). Chance agreement is at the basis of Cohen's Kappa computation [31].

4. **Cohen's Kappa**, which is an inter-rater agreement between Dispersed Coupling and Performance issues, and indicates the strength of agreement and disagreement of two raters (e.g., TD indicators). Cohen's Kappa is appropriate for testing whether agreement exceeds chance levels for binary and nominal ratings. Therefore, Dispersed Coupling and Performance issues are in agreement if both occur at least once in the same Java class, or if both of them are not present. In any other case, they are in disagreement (Figure 2: Step 1, M4). Cohen's Kappa has been used for the assessment of defect classification schemes [32] and in software process assessment [33]. Used in conjunction with other statistical measures, it prevents the misinterpretation of results possibly affected by prevalence and bias.

The computation of chance agreement, conditional probability and Cohen's Kappa required data transformation for some measures: all metrics ranges [0..N] were reduced to [0,1], where "1" indicates that at least one occurrence of the TD indicator was found in the class and "0" otherwise. For example, for the Number of bug fixes (fixed) in version *v*, we assign "1" if the Java class was part of at least one defect fix during the analyzed version. The partial loss of measurement resolution is further discussed in Section 6.

For change likelihood, where the metric ranges from 0.0 to 1.0 (floating point numbers), we investigated its empirical distribution on each version and selected a threshold value to be used in the data transformation. This threshold was computed so as to guarantee that, on average, only the top 25% of classes with high change likelihood obtained a value of "1" in the data transformation, "0" otherwise. The computed threshold was 0.01, indicating that a Java class that is changed more often than in 1 out of 100 cases is considered *change prone*.

The same was done for the size metric, Number of Methods. We use a threshold of 11 to guarantee that only the top 25% of classes were considered *large*. The computation of the four statistical analysis techniques produced four respective matrices shown in Figure 2: M1, M2, M3 and M4. Each of the statistical measures offers a different perspective on the association between pairs of metrics.

The Pearson correlation indicates whether two indicators increase/decrease together following a linear pattern. The conditional probability is useful in understanding the direction of the association, since it indicates whether an indicator is present in a class given that another indicator is present. The chance agreement, instead, is the probability that two indicators are either indicating or not indicating a problem in the same class randomly; in the following section we will see how we use this measure with the conditional probability. Finally, the Cohen's Kappa is useful because the agreement takes into account not only when two indicators are present in the same class, but also when they are not present simultaneously.

4.3.2 Step 2: Apply Significance Functions

After applying the different techniques in Step 1, the goal was to filter the more significant associations from the less significant ones. Therefore we applied different significance functions Ω. The significance functions map the associations between each pair of indicators to [0,1], expressing that the pairs are strongly associated ("1") or not associated strongly enough ("0"). The following three formulae define the functions applied to produce matrices M5, M6 and M7 for each of the 13 versions in Figure 2:

- Significant Pearson Correlations:

$$\Omega(M1_{i,j}) = \begin{cases} 1, & if\ (M1_{i,j}) \geq T1 \wedge p\text{-}val \leq 0.05 \wedge (i \neq j) \\ 0, & otherwise \end{cases}$$

- Significant Conditional Probability: using M2 (Cond. Probability) and M3 (Chance agreement):

$$\Omega(M2_{i,j}, M3_{i,j}) = \begin{cases} 1, & if\ (M2_{i,j}) > (M3_{i,j}) \wedge (M2_{i,j}) \geq T2 \wedge (i \neq j) \\ 0, & otherwise \end{cases}$$

- Significant Cohen's Kappa:

$$\Omega(M4_{i,j}) = \begin{cases} 1, & if\ (M4_{i,j}) > T3 \wedge (i \neq j) \\ 0, & otherwise \end{cases}$$

The goal of the significance function Ω is to discern significant relations from insignificant ones. Parameters **T1**, **T2** and **T3** are three specific thresholds of the respective significance functions that we chose based on the association strength levels found in the literature:

T1 (Correlation) = 0.60. We found two main correlation strength classifications, Cohen [34] and Evan [35]. We adopt Evan's strong definition because it is stricter than Cohen's definition.

T2 (Conditional probability) = 0.60. To our knowledge, existing literature does not provide a commonly accepted and generally applicable threshold for conditional probability. Therefore, we calculated the distribution of conditional probabilities in the different versions and we selected the threshold 0.60, which on average filtered out 80% of all data. Moreover, since this criterion is merged with the criterion conditional probability > chance agreement, we consider such threshold high enough to discriminate significant data.

T3 (Cohen's Kappa) = 0.60. Many tables of Kappa's strength of agreement can be found in the literature, the most relevant of which are [36] [37] [31]. We adopt a threshold value of 0.60. Thus, a constraint > 0.60 corresponds to a "good"/"substantial" agreement in all the proposed ranks.

4.3.3 Step 3: Combine Statistical Association Measures

The next step is an aggregation of the association measures.. For each cell of a matrix (a pair of indicators), we compute the sum over all three matrices M5, M6, and M7. The resulting matrix (displayed as M8 in Figure 2) contains the significance score for each pair and version that ranges from 0 (not significant in any of the three methods) to 3 (significant in all three methods).

4.3.4 Step 4: Combine Versions

The fourth and final computation step of the process is the aggregation of the significance score over versions. For every cell in Matrix M8, we compute the mean over the 13 versions of Hadoop, resulting in a single matrix (M9 in Figure 2).

4.3.5 Step 5: Visualize Most Significant Outcomes

In the final step, we visualize the pairs of TD indicators with most significant associations as a graph (Figure **3**) and a list of TD indicators most related to interest indicators (Table II).

5. RESULTS

We made the following observations from the result. First, the value of TD indicators increases together with the size of Hadoop. The sum of all TD indicators increases in the evolution from release 0.2.0 to release 0.14.0. FindBugs issues ranged from 307 to 486 but the average number of issues per class does not expose the same monotone increasing trend, and the range of the average number of issues per class is [1.30-1.76].

Code smells in the last release (352) are more than twofold the number of code smells in the first release (143), and the average of code smells per class is [0.78-1.01]. Modularity violations have the sharpest increase: they were 8 in the first analyzed release and 37 in the last one (reaching a maximum of 38 in v. 0.13.0). The average number of Modularity violations per class ranged from 0.04 to 0.11. Moreover, we detected in each version two realizations of Singleton design pattern, two realizations of State pattern and six of Abstract factory. However, none of the classes that are participating in these design patterns was affected by grime.

We do not observe a trend in any of the interest indicators. The sum of Bug fixes collected with the defect proneness strategy "inject" ranges from a minimum of 16 (v 0.8.0) to a maximum of 102 (v 0.3.0), while the range of the average number per class is

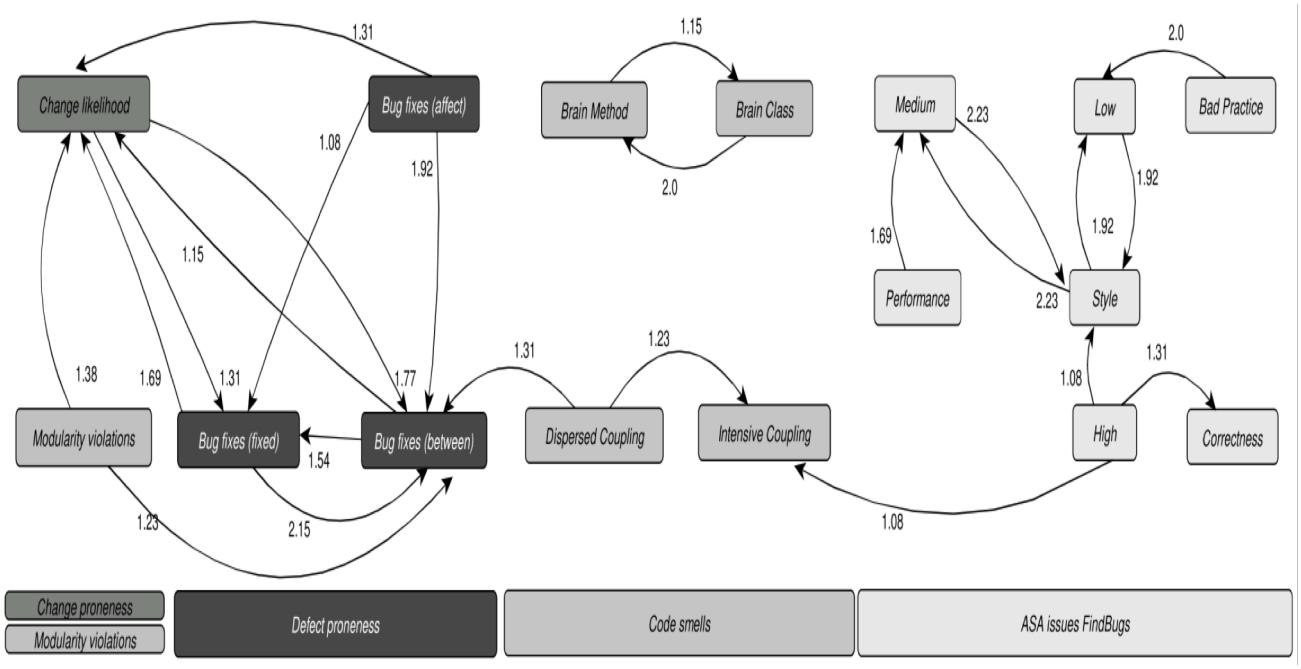

Figure **3**: Graph of top ranked pairs (average score > 1)

[0.06-0.53]. Version 0.8.0 has no Bug fixes (fixed) and version 0.7.0 is the version with the largest number (590). The average of all classes per version is in the range [0-2.63]. The ranges of Bug fixes (between) are [162-675] and [0.55-3.01], respectively for their total by version and average of all classes by version. Finally, the Change likelihood range is [0.006-0.017] per class.

Figure **3** shows the resulting directional graph of the TD pairs and interest indicators that were on average significant in more than one statistical measure (overall mean score in Matrix M9 > 1). The nodes of the graph show the indicators. The color (or shade) of the node indicates which TD indicators were derived from the same TD detection technique. The directional edges are further labeled by their association strength (ranging from 1 to 3). And lastly, the direction indicates the conditional properties inherited from the conditional probability metric. For example, Modularity violations and Bug fixes are associated, meaning that if a class has a Modularity violation, then it is also likely that such a class will have Bug fixes. However, the reverse statement (i.e. classes containing Bug fixes are not as likely to have Modularity violations at the same time) is not necessarily true, and does not show in the graph.

Figure **3** represents the strongest associations found in our analysis. The graph contains 24 relationships (edges), one of them between TD indicators (nodes) belonging to different techniques (colors or shades), three of them between TD indicators and interest indicators or size, and the remaining 20 are among TD indicators detected by the same technique or among the interest indicators.

As for correlations between TD indicators and interest indicators, Dispersed Coupling points to classes that are more defect prone. Modularity Violations do not strongly co-occur with code smells or ASA issues but are likely to point to defect and change prone classes.

Seven out of the twelve ASA/FindBugs issue types appear in the graph. The strongest associations (average score ≥ 2) are between Style and Low and Bad Practice and Medium. We also observe an association between a FindBugs issue (High) and a Code Smell (Intensive Coupling). Four out of ten code smells show up in the graph. Brain Class and Brain Method code smells are related in both directions, as well as Dispersed and Intensive Coupling (but only one direction).

Lastly, defect prone classes tend to be also change prone, and vice-versa. The size metric Number of Methods does not shoot up in the graph indicating that neither the TD indicators nor the interest indicators are very strongly associated with size.

5.1 RQ 1: Which techniques tend to report problems in the same sets of classes?

The results shown in Figure **3** lead to our first finding in response to RQ1: Different TD techniques point to different classes and therefore to different problems.

The only arc in Figure 3 that relates two different types of TD identification approaches is Intensive Coupling and FindBugs High priority issues. A method exhibits intensive coupling if it "calls too many methods from a few unrelated classes" [19]. FindBugs High priority issues are those issues thought to have higher probability to detect serious problems in the code. The direction of the association indicates that classes with many High priority issues have methods affected by Intensive Coupling. A possible explanation for this relation is that both of these indicators point, more than any others, to generally poorly designed code.

Looking at the associations inside the boundaries of the techniques, we observe a characteristic of all FindBugs issues in the graph: significant relationships are only revealed between priority and type categories, which are not independent indicators and are constructed by the FindBugs authors[2]. Therefore this relationship is not a surprising result. A follow up analysis revealed that 81-87% (depending on version) of all classes contain FindBugs issues of only one single category.

Shifting the focus of the results analysis to the code smells group, we observe three associations between particular code smells: Brain Class → Brain Method (2.0), Brain Method → Brain Class (1.15) and Dispersed Coupling → Intensive Coupling (1.23). The first relationship is stronger in the direction → Brain Method and it indicates that Brain classes are more prone to contain Brain methods. This observation can be explained by the way Brain Class code smells are detected using Marinescu's detection strategy [19] [20]: the Brain Class detection requires that the inspected class contains at least one Brain Method. Therefore the conditional probability as defined in Section IV of P(Brain Method | Brain Class) is always 1.0. The second relationship between code smells is Dispersed Coupling → Intensive Coupling (1.20). While the latter indicates that a class has methods that invoke *many* functions of a *few* other classes, the former shows classes having methods invoking functions of many other classes. Their association demonstrates that classes in Hadoop having the former of the coupling smells also have the latter smell, which intensifies the problem of coupling. No other relationship within different code smells was found in this analysis.

We also observe that modularity violations are not strongly related to any other indicator. This confirms and validates one of the findings reported in Wong et al [9], who found that 40% of modularity violations in Hadoop are not defined as code smells and are not detectable using existing approaches.

To conclude, the 4 TD detection approaches (modularity violation, code smells, grime, and ASA issues) have only very little overlap and are therefore pointing to different problems. Within the broad approaches, relations are stronger (as one would expect). However the data also shows that many code smells and some ASA issue types are not inter-related (i.e. the ones not showing in Figure 3) indicating that even at a lower level indicators point to different problem classes.

5.2 RQ2: Which TD indicators correlate with the interest indicators defect- and change-proneness?

Turning to RQ2, our major finding concerning defect-proneness is summarized as follows: **The dispersed coupling code smell and modularity violations are located in the classes that are more defect-prone.**

For each TD indicator (Column 1), Table II reports the average score obtained in matrix M9 for the association between the indicator and defect proneness (columns 2-4) and change proneness (column 5). TD indicators are listed in the same order as Table I, but those with average score less than or equal to 0.3 in all associations with interest indicators are not shown.

[2] Each bug pattern is assigned a priority and category by the FindBugs authors. Some categories are biased towards single priorities: e.g., *correctness* is considered more often to be of *high* priority.

Figure **3** shows that no single FindBugs indicator has a very strong relationship (>1) with Bug fixes. However, when investigating less correlated indicators we find the strongest FindBugs indicator to be Multithread Correctness having a borderline value of 1.0 (Table II). This category is very specific but ties very well into the studied application; Hadoop has to deal with both distributed data storage and computations. Previous work [12] [13] [38] reported that only a small percentage of FindBugs issues are actually related to bug fixes. This is supported by our results and a follow up analysis: Multithread Correctness issues make up only 5.3% of the total of FindBugs issues found in Hadoop.

Another strong relationship with Bug fixes (in two approaches, between and fixed) involves the Dispersed Coupling code smell. In a related work [39], Dispersed Coupling was highly correlated with bug fixes only when the prevalence of this smell increased during the evolution of the software. We observe a border value (1.0) also for one other code smell: the God Class indicator has an average score of 1.0. Zazworka et al. [11] reported in their previous work that in an industrial system god classes contained up to 13 times more defects than non-god classes.

The last indicator strongly related to Bug fixes (between) is Modularity violations, which are located in the same classes where the more bug fixes are found (but not in all of them).

Although defect prediction is not a goal of this work, it is useful to look at measures of precision and recall to further describe the relationships we've found. We used the two TD indicators most strongly related to bug fixes and the two border value indicators to predict, in each version, classes with at least one bug fix (strategy between). We observe in Table III high precision and low recall values. Each of the four indicators points out a small subset of defect-prone classes very well. When using all four indicators together recall can be raised to 0.33 by trading off some precision.

TABLE II: ASSOCIATION OF TD INDICATORS WITH INTEREST INDICATORS

	TD Indicator	Bug fixes (between)	Bug fixes (inject)	Bug fixes (fixed)	Change likeli-hood
Mod Viol	*Modularity violations*	**1.23**	0.23	0.54	**1.38**
Code Smells	*God Class*	**1.00**	0.23	0.77	0.85
	Brain Class	0.62	0.23	0.46	0.62
	Tradition Breaker	0.69	0.31	0.38	0.69
	Feature Envy	0.54	0.15	0.31	0.15
	Brain Method	0.77	0.23	0.54	0.46
	Intensive Coupling	0.54	0.00	0.08	0.15
	Dispersed Coupling	**1.31**	0.23	0.54	0.92
	Shotgun Surgery	0.31	0.00	0.08	0.08
FindBugs issues	*High*	0.92	0.08	0.46	0.62
	MT Correctness	**1.00**	0.08	0.46	0.69
	Correctness	0.92	0.15	0.46	0.62
	Performance	0.31	0.00	0.08	0.08
	Style	0.31	0.00	0.08	0.38
Size	*Number of Methods*	0.62	0.00	0.08	0.08

TABLE III: PRECISION AND RECALL WHEN PREDICTING DEFECTS PRONE CLASSES

	Multithread Correctness	Dispersed Coupling	God Class	Modularity Violations	All 4
Avg. prec.	0.78	0.81	0.96	0.85	0.77
Avg. recall	0.13	0.12	0.11	0.21	0.33

The second part of RQ2 was concerned with change proneness. The following summarizes this result. **Modularity violations point to change prone classes.**

Change-prone classes might indicate maintenance problems (e.g., classes that have to be changed unusually often are candidates for refactoring). As outlined in Section IV we labeled on average the top 25% most frequently changed classes as "change-prone."

The results indicate that Modularity Violations are strongly related to change likelihood. Table II shows that the highest average scores are Modularity violations (1.38), Dispersed coupling (0.92) and God Classes (0.85). This fits expectations since all of the three approaches claim to identify maintenance problems. Modularity violations and Dispersed coupling point to classes that have collaboration disharmonies. The God class code smell identifies classes that implement multiple responsibilities and should be refactored (e.g. split up into multiple classes).

Further, the relation between defect and change proneness shows that these issues are interconnected. Explanations for the phenomena can be that maintenance problems lead to less correct code, or that many quick-and-dirty bug fixes lead to less maintainable code.

Finally, we point out that a large set of TD indicators (i.e. 9 out of 25) do not show significant associations with defect or change proneness. This proportion suggests that these indicators either point to different classes of quality issues (e.g. FindBugs type Performance) or to none at all. Therefore these results can be further used to tailor TD indicators towards quality attributes of interest. If one is most interested in defect and change proneness issues in Hadoop (or similar software) we suggest analyzing for dispersed coupling and modularity violations.

6. THREATS TO VALIDITY

We list the threats to the validity and generalizability of the study following the structure proposed by Wohlin et al. [40], who identify four categories: construct, internal, conclusion and external threats.

A first conclusion threat concerns the impact of thresholds T1, T2 and T3 applied on Step 2 (Section 4.3.2) on the results. We documented the choice of thresholds based on values discussed and adopted in the literature, and we adopted higher values to decrease the level of uncertainty. The collection and aggregation of different statistical measures also lowers the risk associated with this threat.

A further statistical point of discussion is the loss of measurement resolution in data transformation from ranges [0..N]

to the range [0,1]. The transformation was required to compute Cohen's Kappa and the conditional probability. To limit this threat we investigated distributions carefully to find reasonable thresholds (e.g. for Number of Methods we decide on a threshold of 11 for a top 25% cutoff). Moreover, the choice of at least one occurrence as criterion for transformation was driven by a preliminary analysis of distribution that revealed that TD and interest indicators were equal to zero on average in 90% of the classes.

Another conclusion threat is derived by the decision to not normalize measures by size. Our choice was based on past experiences [10] and from the analysis of the results of the current study. Figure 4 shows a correlation plot between size and number of defect fixes for each file over all versions. The 3rd order polynomial trend line shows that the correlation is not simply linear, e.g. a class twice as large is not twice as defect-prone. This analysis suggests that a linear normalization by number of methods is not required.

Moreover, the choice of this size metric rather than other size metrics is also a threat (construct). We examined whether the Total Number of Methods correlates with other size metrics in the different Hadoop versions. We obtained almost perfect correlations with Total Number of Statements, Total Number of Lines of Code (that include comments) and Total Number of Files: 0.9958, 0.9950 and 0.9711 respectively. In addition to that, we checked, for each version, the correlation between the Number of Methods and two other metrics, i.e. the Number of Lines of Code and the Number of Statements in Java classes. We also obtained at this granularity very high correlation, respectively 0.8975 and 0.8780. We conclude that using Number of Methods as measure of class size is equivalent to lines of code and did not affect results.

A further construct threat is the selection of outer classes, ignoring nested classes. At this point in time, the tools used did not allow the collection of all metrics for nested classes. Therefore the validity scope of our results is limited to outer classes only. Future work will be devoted to include nested classes in the analysis.

We believe that the findings of this work apply only in the context analyzed (external threat). They may apply in similar applications, but we are not aware of any other published results that can be compared to ours. However, although results cannot be generalized, they contribute to begin composing the TD landscape.

7. CONCLUSION

The main findings of this study are:

- Different TD techniques point to different classes and therefore to different problems. There are very few overlaps among the results reported by these techniques.
- Dispersed coupling, god classes, modularity violations and multithread correctness issues are located in classes with higher defect-proneness.
  Modularity violations are strongly associated with change proneness.

Our results indicate that the issues raised by the different code analysis techniques are in different software classes. Moreover, only a subset of the problematic issue types identified by these techniques is shown to be more defective or change prone. This is consistent with the result of earlier work where these techniques were applied independently ([10] [11] [12][13]).

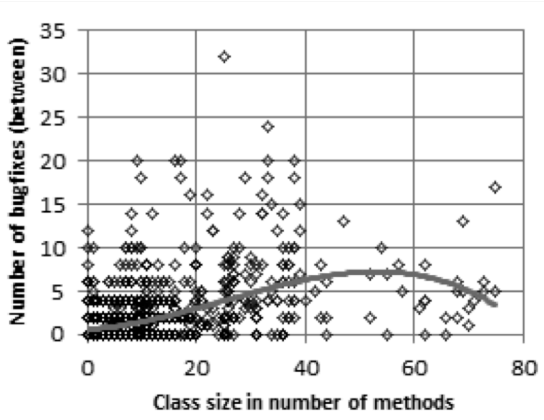

Figure 4: Correlation plot for size vs. defect proneness

These findings contribute to building an initial picture of the TD landscape, where TD techniques are loosely overlapping and only a subset of them is strongly associated to software components' defect and change proneness.

This initial picture will contribute to future research efforts concerned about continuously monitoring and managing TD in software projects, a larger subject that is out of the scope of this study. But this study constitutes an important step in addressing the TD problem holistically: not all TD should be considered sufficiently harmful to warrant repayment, in particular if the cost of repair (paying the principal) outweighs the interest payments in long term. Thus, not all TD is bad, and not all TD needs to be avoided. The results of this study build a stepping-stone for further trade-off analysis studies by providing insights into how tools can help to point to TD that is worth being managed.

7.1 Implications for Practice

The results indicate that, in practice, multiple TD indicators should be used instead of only one of the investigated tools. As a recommendation to practitioners, these initial results evidently show that different tools point to different problems in a code base. The use of a single tool or single indicator (e.g. a single code smells) will only in rare cases point to all important TD issues in a project. As a result, development teams need to make intentional decisions about which of the TD indicators are of most relevance to them, based on the quality goals of their project, as suggested in [43]. For example, is maintainability a priority for the team, or is the system expected to be short-lived? Is code performance important? Different answers to questions like these would lead to different choices for a TD detection strategy.

In the current state of research we cannot yet give a more complete recommendation on which indicators are best for signaling specific quality shortcomings, however, our results give some preliminary advice on which indicators to start with when looking for TD related to defects and maintenance bottlenecks, namely: Modularity Violations, God Class, Dispersed Coupling, and Multi Thread Correctness issues.

7.2 Implications for Research

Since results indicate that there might not be a project independent one-size-fits-all tool to detect TD, but rather a necessary tailoring process to the right subset of indicators required, future research should be concerned with investigating and showing connections between TD techniques, types of technical debt, effect and tailoring towards project specific software quality characteristics. Future work should also

investigate other TD indicators when they become available to broaden the landscape.

As more specific advice for future research directions, we recommend extending the interest indicators towards a broader range of software quality aspects, beyond defect and change-proneness as investigated here. Further, we recommend extending this type of quantitative study with qualitative insights, e.g. from practitioners that investigate if the studied approaches point to the most important kinds of technical debt.

8. REFERENCES

[1] W. Cunningham, "The wycash portfolio management system," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 29–30. [Online]. Available: http://-doi.acm.org/10.1145/157709.157715

[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. [Online]. Available: http://-doi.acm.org/10.1145/1882362.1882373

[3] I. Gat and J. D. Heintz, "From assessment to reduction: how cutter consortium helps rein in millions of dollars in technical debt," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 24–26. [Online]. Available: http://-doi.acm.org/10.1145/1985362.1985368

[4] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing* Technical *Debt*, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1985362.1985364

[5] CAST, "Cast worldwide application software quality study: Summary of key findings," Tech. Rep.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley Professional, Jul. 1999.

[7] C. Izurieta and J. Bieman, "How software designs decay: A pilot study of pattern evolution," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, sept. 2007, pp. 449 –451.

[8] ——, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," Springer Software Quality Journal. February 2012, http://dx.doi.org/10.1007/s11219-012-9175-x.

[9] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proc. 33th International Conference on Software Engineering*, May 2011, pp. 411–420.

[10] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceeding of the 2nd working on Managing technical debt*, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 17–23. [Online]. Available: http://doi.acm.org/-10.1145/1985362.1985366

[11] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609564

[12] A. Vetro', M. Torchiano, and M. Morisio, "Assessing the precision of findbugs by mining java projects developed at a university," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, may 2010, pp. 110 –113.

[13] A. Vetro', M. Morisio, and M. Torchiano, "An empirical validation of findbugs issues related to defects," *IET Seminar Digests*, vol. 2011, no. 1, pp. 144–153, 2011. [Online]. Available: http://link.aip.org/link/abstract/IEESEM/v2011/-i1/p144/s1

[14] C. Izurieta and J. Bieman, "Testing consequences of grime buildup in object oriented design patterns," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, april 2008, pp. 171 –179.

[15] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:10. [Online]. Available: http://-doi.acm.org/10.1145/1852786.1852797

[16] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *Software Engineering, IEEE Transactions on*, vol. SE-10, no. 6, pp. 728 –738, nov. 1984.

[17] Y.-G. Guéhéneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, ser. TOOLS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 296–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882501.884740

[18] W. J. Brown, R. C. Malveau, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, Mar. 1998. [Online]. Available: http://www.worldcat.org/isbn/0471197130

[19] M. Lanza and R. Marinescu, *Object-oriented Metrics in Practice*. Berlin: Springer-Verlag, 2006.

[20] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," *Software Maintenance, IEEE International Conference on*, vol. 0, pp. 350–359, 2004.

[21] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, pp. 92–106, December 2004. [Online]. Available: http://doi.acm.org/10.1145/-1052883.1052895

[22] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831738

[23] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 27–. [Online]. Available: http://-dx.doi.org/10.1109/MSR.2007.26

[24] C. Boogerd and L. Moonen, "Evaluating the relation between coding standard violations and faultswithin and across software versions," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, may 2009, pp. 41 –50.

[25] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/-1287624.1287633

[26] N. Nagappan, T. Ball, and A. Zeller, “Mining metrics to predict component failures,” in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 452–461. [Online]. Available: http://doi.acm.org/10.1145/-1134285.1134349

[27] N. Nagappan and T. Ball, “Static analysis tools as early indicators of pre-release defect density,” in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, may 2005.

[28] S. Muthanna, K. Kontogiannis, K. Ponnambalam, and B. Stacey, “A maintainability model for industrial software systems using design level metrics,” in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, 2000, pp. 248 –256.

[29] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, “Predicting software defects in varying development lifecycles using bayesian nets,” *Information and Software Technology*, vol. 49, no. 1, pp. 32 – 43, 2007, <ce:title>Most Cited Journal Articles in Software Engineering - 2000</ce:title>. [Online]. Available: http://-www.sciencedirect.com/science/article/pii/-S0950584906001194

[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] J. L. Fleiss, *Statistical Methods for Rates and Proportions*, 2nd ed., ser. Wiley series in probability and mathematical statistics. New York: John Wiley & Sons, 1981.

[32] K. El Emam and I. Wieczorek, “The repeatability of code defect classifications,” in *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, nov 1998, pp. 322 –333.

[33] H.-M. Park and H.-W. Jung, “Evaluating interrater agreement with intraclass correlation coefficient in spice-based software process assessment,” in *Quality Software, 2003. Proceedings. Third International Conference on*, nov. 2003, pp. 308 – 314.

[34] J. Cohen, *Statistical power analysis for the behavioral sciences : Jacob Cohen.*, 2nd ed. Lawrence Erlbaum, Jan. 1988. [Online]. Available: http://www.worldcat.org/isbn/-0805802835

[35] J. Evans, *Straightforward statistics for the behavioral sciences*. Brooks/Cole Pub. Co., 1996. [Online]. Available: http://books.google.com/books?id=8Ca2AAAAIAAJ

[36] J. R. Landis and G. G. Koch, “The Measurement of Observer Agreement for Categorical Data,” *Biometrics*, vol. 33, no. 1, pp. 159–174, Mar. 1977.

[37] D. G. Altman, *Practical Statistics for Medical Research (Statistics texts)*, 1st ed. Chapman & Hall/CRC, Nov. 1990. [Online]. Available: http://www.worldcat.org/isbn/-0412276305

[38] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, “Comparing bug finding tools with reviews and tests,” in *Proc. of Int’ Conf. on Testing of Communications Systems*, 2005, pp. 40–55.

[39] M. D’Ambros, A. Bacchelli, and M. Lanza, “On the impact of design flaws on software defects,” in *Quality Software (QSIC), 2010 10th International Conference on*, July 2010, pp. 23 –31.

[40] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000

[41] Bieman, J.M.; Straw, G.; Wang, H.; Munger, P.W.; Alexander, R.T.; , "Design patterns and change proneness: an examination of five evolving systems," *Software Metrics Symposium, 2003. Proceedings. Ninth International* , vol., no., pp. 40- 49, 3-5 Sept. 2003

[42] Khomh, F.; Di Penta, M.; Gueheneuc, Y.-G.; , "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on* , vol., no., pp.75-84, 13-16 Oct. 2009

[43] Shull, F. “Perfectionists in a World of Finite Resources,” *IEEE Software*, vol. 28, no. 2, pp. 4-6, March / April 2011.