Forecasting Architectural Decay From Evolutionary History

Joshua Garcia, *Member, IEEE*, Ehsan Kouroshfar, Negar Ghorbani, and Sam Malek, *Member, IEEE*

Abstract—As a software system evolves, its architecture tends to decay, leading to the occurrence of architectural elements that become resistant to maintenance or prone to defects. To address this problem, engineers can significantly benefit from determining which architectural elements will decay before that decay actually occurs. Forecasting decay allows engineers to take steps to prevent decay, such as focusing maintenance resources on the architectural elements most likely to decay. To that end, we construct novel models that predict the quality of an architectural element by utilizing multiple architectural views (both structural and semantic) and architectural metrics as features for prediction. We conduct an empirical study using our prediction models on 38 versions of five systems. Our findings show that we can predict low architectural quality, i.e., architectural decay, with high performance—even for cases of decay that suddenly occur in an architectural module. We further report the factors that best predict architectural quality.

Index Terms—Software architecture, prediction model, architectural smell, architectural decay

1 Introduction

In a software system's life cycle, software maintenance tends to dominate other activities in terms of time, effort, and cost [1], [2], [3], [4]. Throughout that life cycle, a major artifact that must undergo maintenance in a long-lived software system is its architecture, which determines the key properties of a software system. Such maintenance activities include determining the system's current architecture, refactoring or restructuring it, or assessing its current ability to achieve its non-functional properties. Architectural elements abstract away unnecessary complexity (e.g., details of source-code constructs), allowing engineers to focus on higher-level design decisions. However, a software system's architecture is known to commonly undergo the phenomenon of *architectural decay* [5], where design decisions are added to and may even violate an architecture, leading to defects and other major architectural problems.

Although decay is typically treated once its detrimental effects (e.g., highly defective component or one that is highly resistant to change) are detected in a system, engineers can benefit from stemming architectural decay before such effects occur. To make such a determination, engineers would significantly benefit from predicting which architectural elements are most likely to undergo decay so that they can allocate resources to those elements in the most effective manner. Previous work has produced models for predicting only defects for packages or directories [6], [7], [8]. However, defects are not the only forms of architectural decay [9], [10]. Futhermore, packages represent a structural view of the architecture [11]. A structural view is an architectural view drawn from packages, directories, or control- or data-flow based dependencies among code-level elements [12], [13], [14], [15]. This structural view is one of several architectural views that represent code organization structure in software architecture, dating back to Kruchten's seminal 4+1 view model of software architecture [11], [16], [17]. For example, Kruchten's 4+1 view model referred to this view as the Development view, while Clements *et al.* referred to this view as the Module Viewtype.

Although such a view is valuable for determining decay, a semantic view of the architecture is needed to identify decay involving the concerns attributed to different architectural elements (e.g., one component handles job tracking, while another component handles filesystem management). A semantic view is an architectural view drawn from the words and terms in a software system, often obtained using some form of information retrieval or natural language processing [12], [13], [14], [15].

To stem architectural decay, engineers can significantly benefit from predicting a variety of constructs related to architectural quality—including indicators of architectural decay, i.e., *architectural bad smells* [9], [10], and the *quality of an architecture's modularization* [18]. Architectural bad smells—which are patterns of architectural constructs that may negatively affect the maintenance of software systems—reduce the quality of a software system's architecture but do not constitute a defect that should be fixed in all cases. We take a software defect to be a mistake that does not meet the software system's specification and can result in unintended behavior [19]. Smells do not introduce functional bugs or effect software behavior, instead they affect maintainability. Determining that an architectural module is decaying, even before it is involved in an architectural smell or exhibits low modularization quality, can reduce

- *Joshua Garcia, Negar Ghorbani, and Sam Malek are with the Department of Informatics, Institute for Software Research, University of California Irvine, Irvine, CA 92697 USA. E-mail: {joshug4, negargh, malek}@uci.edu.*
- *Ehsan Kouroshfar is with the Amazon.com, New York, NY 10012 USA. E-mail: ekouroshfar@gmail.com.*

Manuscript received 12 Mar. 2020; revised 17 Jan. 2021; accepted 1 Feb. 2021. Date of publication 18 Feb. 2021; date of current version 18 July 2022. (Corresponding author: Joshua Garcia.) Recommended for acceptance by R. Holmes. Digital Object Identifier no. 10.1109/TSE.2021.3060068

maintenance time and effort. By predicting architectural decay of modules, an engineer or architect can proactively use information about specific decay to prevent its occurrence. For instance, if the prediction technique indicates that a module will soon handle too many concerns or be involved in too many dependencies, engineers and architects can work together to ensure that best practices (e.g., use of design patterns that encourage separation of concerns) are followed to prevent that module from being bloated.

To forecast architectural decay, we construct novel models that predict the quality of an architectural element (i.e., architectural module) by utilizing multiple architectural views (both structural and semantic) and architectural metrics as features for prediction. To obtain multiple architectural perspectives, we utilize two module-level views: a package-level view and a semantic view, obtained by leveraging an information retrieval-based technique [20], [21] shown to work accurately based on the latest evaluations of techniques for recovering a software system's architecture [21], [22], [23]. Our architectural-quality prediction models utilize an effective set of prediction metrics (i.e., file-level metrics, smell-based metrics, and architectural metrics) and metrics for representing architectural quality at the module level (i.e., defects, smell-based metrics, and modularization quality). Each architectural view provides an alternative perspective that can be used to prioritize architectural modules and allocate resources to them for maintenance purposes.

We conduct our study on 38 versions of 5 open-source Java systems from the Apache Software Foundation. The overarching findings of our experiments are as follows:

- Our models can predict low architectural quality, indicating decay, with high performance. Specifically, our models can predict defectiveness of modules with AUC (area under the curve of the receiver operating characteristic, which is a measure of a predictive model's performance) results of 0.76-0.88 and the occurrence of architectural smells in modules with AUC of 0.84-1.0. Furthermore, our models can rank modules with high accuracy based on their numbers of defects (as represented by Spearman correlation results of 0.48-0.73) and their modularization quality (as represented by a Spearman correlation of 0.70-0.98). All the Spearman correlations we report are significant at the 0.01 level.
- Although at most 12 percent of modules exhibit *smell emergence*—which represent sudden occurrence of smells in modules—we are still able to predict these instances of smell emergence with AUC of 0.79-0.96.
- We investigate which factors are important for predicting different aspects of architectural decay. Our findings suggest that to predict each aspect of architectural decay, different combinations of factors are needed. In particular, file-level metrics are not enough to comprehensively predict architectural quality.
- To facilitate replication of our experiments and reuse of our tools and data, we make both our tools and dataset for our experiments publicly available online [24].

The remainder of this paper is organized as follows: Section 2 introduces the research questions we study. Section 3 describes our approach for predicting architectural quality. That section is followed by a description of our experiments' design and setup (Section 4), the results of our experiments (Section 5), practical importance of our findings (Section 6), and the threats to validity (Section 7). A discussion of related work (Section 8) and conclusions round out the paper (Section 9).

2 Research Questions

For our study, we seek to answer research questions that assess the effectiveness of our architectural-quality prediction models. To that end, we study different regression models, the extent of change of each architectural-smell metric, the ability of our models to predict the emergence of an architectural smell, and the metrics that work best for each of our models. Architectural-quality metrics measure a software system's architecture in terms of its non-functional properties, especially those related to maintainability. Smells metrics are a type of architectural-quality metric. We focus on smells because they give specific instances of low architectural quality with concrete mechanisms for repairing them (e.g., specific architectural restructurings or refactorings).

We produce a different prediction model for each architectural-quality metric. To ensure high performance of these prediction models, we intend to determine the most effective regression models for making these predictions. Note that *performance* in this context means the correctness of a prediction model—i.e., performance in the sense used in prediction-model literature. Consequently, we study the following research question:

RQ1: *What is the performance of each prediction model for the different architectural-quality metrics?*

To better understand the applicability of our models for predicting architectural smells, the architectural-smell metrics we predict should exhibit change. Each architectural-smell metric measures whether a module has a particular architectural smell. To that end, we must determine the extent of change for each architectural-smell metric in our study. As a result, we study the following research question:

RQ2: *What is the amount of change across releases for each architectural-smell metric?*

Potentially, predicting architectural smells is most important in the case of *smell emergence*, i.e., the addition of smells to a software system. For example, if a module has not had a type of smell in the current release but will have that smell in the next release, our models should predict this occurrence, allowing an engineer to take preventive measures to stem that decay. To that end, we aim to answer the following research question:

RQ3: *Can we effectively predict architectural-smell emergence?*

Although we select prediction metrics that intuitively determine architectural quality, the exact combinations of metrics that best predict architectural quality must be assessed empirically. For our study, we select combinations of metrics that are (1) obtained at the file level and aggregated to modules, and (2) are architectural in nature. Thus, we investigate the following final research question:

RQ4: *What are the important metrics for predicting each architectural quality metric?*

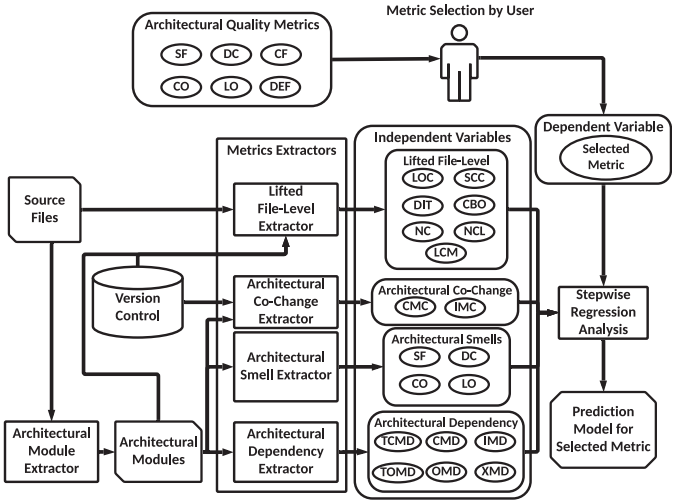


Fig. 1. Overview of our approach for architectural-quality metric prediction.

3 Prediction Model Construction

Fig. 1 overviews our approach for predicting architectural quality. Our approach begins with a set of *source files*, a *version control* repository, and *architectural modules* identified by an *Architectural Module Extractor* from the source files. Given those three artifacts, four *Metrics Extractors—Lifted File-Level Extractor*, *Architectural Co-Change Extractor*, *Architectural Smell Extractor*, and *Architectural Dependency Extractor*—compute 19 metrics that are used as *independent variables* for a *stepwise regression analysis*. A user selects a metric among 6 architectural-quality metrics to be predicted, which serves as the *dependent variable* inputted to the stepwise regression analysis. The result of regression analysis is a prediction model for the selected quality metric. Each prediction model produced by our approach utilizes independent variables of release k of system s and predicts the selected architectural-quality metric for $k+1$ of system s.

In the remainder of this section, we describe the major parts of our approach: the techniques we leveraged to obtain architectural modules, our selected regression models, the six quality metrics to be predicted, and the metrics extracted and used as independent variables.

3.1 Obtaining Architectural Modules

We consider two different techniques for recovering architectural modules, which are used by *Architectural Module Extractor*. As a result, we obtain multiple architectural views [25], allowing an engineer to obtain architectural-quality metrics from different perspectives. This maximizes the possibility of identifying architectural-quality problems throughout a software system. Note that an architecture-recovery technique can be substituted for a ground-truth architecture verified as correct by a software system's architects. In such a situation, our prediction models would likely achieve better performance, since they would not need to correct for improperly recovered modules.

The package structure of a system can be treated as a proxy for the decomposition of the system into architecturally significant elements, as packages are created by the developers of the system. In fact, package structuring has been used as a decomposition reference in prior research [26], [27], [28].

Packages and their sub-packages can be represented in a tree structure corresponding to the packaging hierarchy. Each leaf of the tree is a Java class contained in a package, which itself may belong to a higher level package. The root of the tree is the top-level package. In addition to packages, we include a semantic view of modules obtained using an architecture-recovery technique called *Architectural Recovery using Concerns (ARC)* [15], [20], [21], which utilizes hierarchical clustering and information retrieval to produce modules. ARC leverages a statistical language model, Latent Dirichlet Allocation (LDA) [29], to represent each source file of a system as textual documents consisting of concerns, which are extracted from the identifiers and comments of each file. A concern could be a role, concept, or responsibility of a system. The number of modules recovered by ARC is selectable by an engineer, enabling the consideration of recovered modules at a high level and low level, just as in the case of packages. For ARC's implementation, every entity in a module is a Java source file. Note that for both recovery techniques, the entirety of a file is mapped to a single module.

Once modules have been identified or recovered, we must be able to determine which module m_k in release k is the same module m_{k+1} in release $k+1$. This determination allows us to make predictions for m_{k+1} based on our metrics for m_k. We leverage a technique described in prior work that traces modules across releases based on the degree of overlap among them [15].

3.2 Regression Analysis Selection

We constructed the prediction models in this study using the releases of each project. We use three well-known regression models in this study and compare the results: linear regression (LR), negative binomial regression (NBR), and random forest (RF). We used the *MASS* library in R [30] for building LR and NBR and the *randomForest* library for RF [31].

Although LR is popular and widely used in the literature, some have argued that NBR is a more appropriate regression model for defect prediction [32]. Unlike LR, NBR makes no assumptions about the linearity of the relationship between the variables, or the normality of the variable distributions. NBR is applicable to non-negative integers and, more importantly, can be used for over-dispersed count data (i.e., when the conditional variance of the data exceeds the conditional mean) [33]. We also chose RF since it has been shown to perform best for software defect prediction [34], making RF potentially suitable for predicting architectural quality.

For NBR, we use the log_2 transformation of our metrics to reduce the influence of extreme values, similar to prior work [33].

We do not want our prediction metrics to exhibit multicollinearity, a phenomenon where prediction metrics are correlated, since this can cause our prediction models to become unstable [35]. To avoid the multicollinearity problem, we use stepwise regression to build the models. We leverage the *stepAIC* function in the *MASS* library of R for this purpose. *Akaike Information Criteria (AIC)* is a commonly

used static measure for goodness of fit. Models can be built in two ways: forward and backward. Forward stepwise regression begins with no variable in the model. The variable that improves the model the most is identified and added to the model. The process continues until none of the remaining variables can improve the model. Backward stepwise regression starts with the full model, improves the model by deleting variables, and repeats this deletion until no further improvement is possible. To determine the optimal model, we ran both forward and backward stepwise regression. We used stepwise regression when building models with LR and NBR. We utilized all of the metrics when building models using RF because it works well with a large number of independent variables [36], where our model includes only 19 such variables.

3.3 Dependent Variables

We selected the following six metrics that serve as representations of architectural decay: the number of defects in a module; four architectural-smell metrics, where each metric indicates whether a module has a specific type of smell; and a metric that indicates a module's quality in terms of coupling and cohesion. Each of these metrics is a dependent variable for a single architectural-quality prediction model.

The number of defects per module, shown as *DEF* in Fig. 1, are determined by summing up the defects in each file contained within an architectural module.

The coupling and cohesion of a module is a strong indicator of the module's quality. To that end, we included *Cluster Factor (CF)* [18], a metric used widely in previous architectural studies [18], [21], [37], [38] that represents the coupling and cohesion of a module. We calculate CF for a module m as follows:

$CF_i = \frac{\mu_i}{\mu_i + 0.5 \times \sum_j \epsilon_{ij} + \epsilon_{ji}}$, where μ_i is the number of dependencies between entities within a module, ϵ_{ij} is the number of dependencies from module i to module j, and ϵ_{ji} is the number of dependencies from module j to module i.

The presence or absence of architectural bad smells in a module may inform our prediction models as to the future occurrence of architectural decay. To that end, we select four architectural smells for our study that represent structural or semantic maintainability problems of a module. Each smell falls into one of two categories: *concern-based smells* or *dependency-based smells*. Concern-based smells are caused by inappropriate or inadequate separation of concerns; dependency-based smells arise due to module interactions resulting from code relationships among entities within a module.

We identify the following smells that a module may suffer from, which have been studied in previous work [9], [10], [39].

- *Scattered Functionality (SF)* is a concern-based architectural smell that describes a system in which multiple modules are responsible for realizing the same high-level concern, while some of those modules are also responsible for additional, orthogonal concerns.
- *Concern Overload (CO)* is a concern-based architectural smell that occurs for a module when it implements an excessive number of concerns. For practical identification of such a smell, a given number of concerns is excessive if that number exceeds the mean plus standard deviation of the number of concerns across the modules of the software system in question. This selection of a threshold representing "excessiveness" minimizes the bias of making such a determination [39].
- *Dependency Cycle (DC)* is a dependency-based architectural smell that occurs when a set of modules are linked in such a way that they form a cycle, causing changes to one module to possibly affect all other modules involved in the cycle.
- *Link Overload (LO)* is a dependency-based smell that occurs when a module is involved in an excessive number of dependencies to other modules. A module can have an excessive number of incoming links, outgoing links, or both. Similar to CO, a given number of links is excessive if that number exceeds the mean plus standard deviation of the number of links across the modules of the software system in question.

To represent each of these smells as an architectural-quality metric to be predicted, we create a binary metric for each smell: s_{sf}, s_{co}, s_{dc}, and s_{lo}. If a module m has a smell s, then $s = 1$. Otherwise, $s = 0$. For example, if a module m_1 has CO, then $s_{co} = 1$ for m_1.

3.4 Independent Variables

We use four types of metrics extractors to obtain a combination of *file-level* and *architectural-level* metrics for predicting architectural quality. Many prediction models from existing literature have focused on predicting software defects [32], [40], [41], [42]. We chose a subset of metrics from the prior literature, particularly at the file level, as independent variables for prediction, since they may be indicators of architectural problems.

Lifted File-Level Extractor obtains the following file-level metrics:

- The *lines of code (LOC)* of a file is a measure of the size of a file determined by counting the number of non-empty non-comment lines.
- *Sum cyclomatic complexity (SCC)* of any structured program with only one entry point and one exit point is equal to the number of decision points contained in that program plus one.
- The *depth of inheritance tree (DIT)* is the depth of a class within an inheritance hierarchy calculated as the maximum number of nodes from the class node to the root of the inheritance tree.
- *Coupling between objects (CBO)* for a class is the number of other classes to which it is coupled. Class A is coupled to class B if class A uses a type, data, or member from class B.
- *Lack of cohesion in methods (LCM)* is calculated as 100 percent minus average cohesion for class data members. Average cohesion is calculated as the percentage of pairs of methods in a class that have at least one field in common. A lower percentage means higher cohesion between class data and methods.
- *Number of changes (NC)* is the number of times that a file is committed to a repository.

TABLE 1
Studied Projects and Release Information

Project	Description	# Rel	KSLOC	# Mod	# Def	# Smells
HBase (Hb)	Distributed Database	11	39-246	12-118	29-267	3-86
Hive (Hi)	Data Warehouse Facilities	6	6-226	32-204	15-115	19-84
OpenJPA (Op)	Java Persistence Framework	6	153-407	63-257	24-157	23-99
Camel (Cam)	Message-Oriented Middleware	9	99-390	187-545	178-457	33-212
Cassandra (Cas)	Distributed DBMS	6	50-90	18-75	17-259	13-86

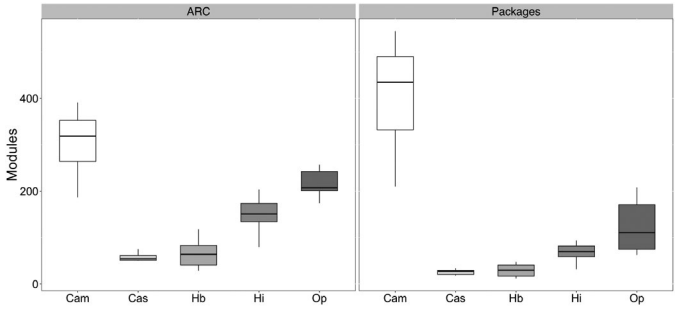


Fig. 2. Number of modules across projects.

- *Number of co-changed files (NCL)* is the number of other files that a given file is changed with [43].

To represent file-level metrics at the module-level, we *lift them up* to the architectural level by summing up the values of each file-level metric across all files inside each module. The resulting sum is then used as a representation of each file-level metric for a module. For example, in the case of SCC, a module m with four files can have the following SCC values, one for each file: 2, 5, 6, and 9. The SCC for module m is the sum of all SCCs of its constituent files, i.e., 22. This approach has been used for predicting defects for packages [6], [7].

Among our architectural metrics, we include metrics involving *co-changes* between modules that are extracted by *Architectural Co-Change Extractor*. Co-changes are process metrics that represent modifications that occur simultaneously within or across modules. Prior work has demonstrated that architectural co-changes correlate with defects [44], [45]. Consequently, architectural co-change metrics may potentially improve our prediction models. We select the following architectural co-change metrics:

- *Cross-module co-changes (CMC)* is the number of co-changes for a file, where the co-changes are made across more than one architectural module.
- *Inner-module co-changes (IMC)* is the number of co-changes for a file, where there is at least another co-changed file in the same architectural module.

A number of our selected architectural-quality metrics are based on dependencies between modules, which are code relationships among source-level entities within a module (e.g., method invocations, field accesses, import statements, etc.). To predict architectural quality based on such dependencies, *Architectural Dependency Extractor* obtains module-dependency metrics.

We consider two methods for measuring the dependencies between modules. The first method models the dependencies as a binary variable, meaning that we only measure whether a module has a dependency on another module. The second method is to count all of the dependencies between the modules, which considers the number of dependencies between the files inside each of the modules. Using these two methods, we select the following dependency-based metrics:

- *Incoming module dependency (CMD)* is a binary metric for a module m_1 with a value of 1 if there is at least one dependency from another module m_2 to m_1, and 0 otherwise.
- *Outgoing module dependency (OMD)* is a binary metric for a module m_1 with a value of 1 if there is at least one dependency from m_1 to another module m_2, and 0 otherwise.
- *Total incoming module dependencies (TCMD)* is the total number of dependencies to a module m_1 and originating from other modules in a software system.
- *Total outgoing module dependencies (TOMD)* is the total number of dependencies from a module m_1 to other modules in a system.
- *Internal module dependencies (IMD)* is the total number of dependencies among all files within a module.
- *External module dependencies (XMD)* is the total number of incoming and outgoing dependencies of a module.

The existence of architectural smells in a module may indicate further architectural decay in the future for that module. For example, a module with CO may be more likely to exhibit LO in the future. As another example, LO may be an indicator of future reductions in a module's CF. To that end, *Architectural Smell Extractor* identifies the four architectural smells described in Section 3.3 and computes the corresponding metrics.

Note that CF and DEF are not used as independent variables because the process we describe to select independent variables (i.e., stepwise regression as discussed in Section 3.2) demonstrated that these variables did not contribute to the regression models.

4 Experimental Setup

To evaluate our prediction models, this section discusses the experimental setup we use to answer our research questions.

4.1 Projects Studied

Table 1 depicts the five projects used in our experiments, including the number of releases, size of the projects, numbers of modules, numbers of defects, and numbers of smells. We selected projects that (1) are written in Java; (2) are maintained by Apache Software Foundation (ASF) because they maintain links between issues and code—allowing us to link defects with specific modules; and (3) vary across application domains and size. Further statistical information about the five studied projects are provided in the following paragraphs.

Fig. 2 shows the number of modules across different releases and projects for both ARC (a semantic or concern-

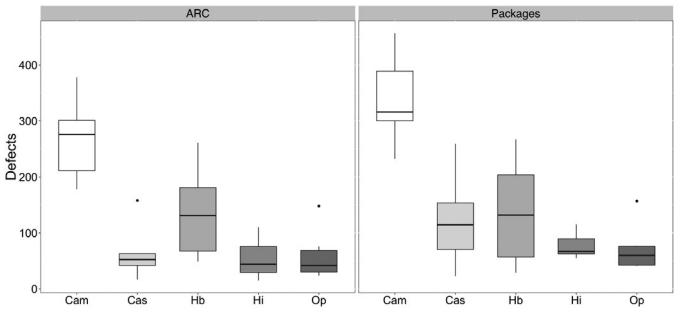


Fig. 3. Number of defects per module across projects and releases.

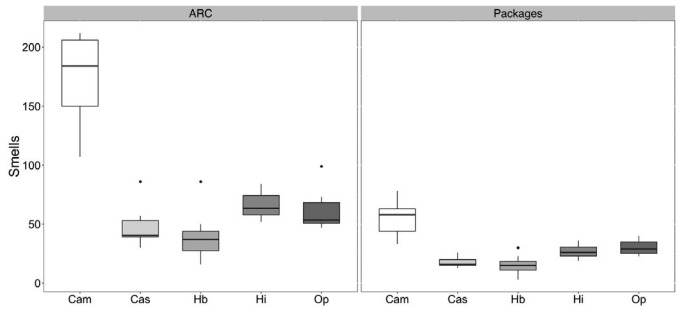


Fig. 4. Number of smells across projects and releases.

based view) and packages (a structural view). We set ARC to produce a number of modules equivalent to 20 percent of the classes in a version of a project, which is the number of modules for which ARC obtained accurate results in a comparative analysis of recovery techniques [21]. Across the five projects, we obtained 29-391 modules for ARC and 12-545 modules for packages. Except for Camel, most of the projects contained more ARC modules than packages.

Fig. 3 illustrates the number of defects across releases and projects, and for both architectural views. The figure indicates that the number of defects tends to be greater for packages than for ARC modules across projects and releases. Specifically, the ARC view contains 15-378 defects, while the package view has 23-457 defects, due to the number of packages being fewer than the number of ARC modules.

Fig. 4 depicts the number of architectural smells obtained from the ARC and package views across releases and projects. The number of smells are greater in the ARC view—containing 16-212 smells—than the package view—which has 3-78 smells. This result is unsurprising since concern-based smells (i.e., CO and SF) are not obtainable from the package view, as that view does not represent a software system's concerns (recall Section 3.1).

4.2 Data Collection and Metric Measurement

To enable prediction of architectural quality, we collect data about bug fixes and metrics at both the code and architectural levels. We utilize different tools for that purpose.

We obtain code-level metrics per file and for each release. The first five file-level metrics (*LOC, SCC, DIT, CBO and LCM*) are measured using UNDERSTAND from Scitools[1].

The change metrics (*NC, NCL, CMC and IMC*) are calculated by processing the developer commits from an SVN repository and extracting the groups of files in the same commit transaction that have been modified together (i.e., co-changes). We use *SVNKit*, a Java toolkit providing APIs to subversion repositories.

To obtain architectural metrics, we leverage *Architecture Recovery, Change, And Decay Evaluator (ARCADE)* [15], [39], a workbench containing tools for addressing architectural decay. Specifically, ARCADE consists of algorithms for detecting architectural smells and computing architectural dependency information, enabling the extraction of our four selected architectural smell metrics (*SF*, *CO*, *DC*, and *LO*) and six architectural dependency-based metrics (*CMD*, *OMD*, *TCMD*, *TOMD*, *IMD*, and *XMD*). To parameterize ARC for this experiment, we simply used the default parameters provided by ARC, which is part of ARCADE, and used in prior studies [15], [20], [22], [39].

In the ASF software repositories and, by extension, the projects studied in this paper, the commits that are bug fixes are identifiable since bugs are referred to by a project name and bug number in SVN commit logs. For example, all of the bug fixes in HBASE begin with *HBASE-<bug number>* (e.g., *HBASE-3172*). This enabled us to find all bug fixes by just parsing the log of commits in SVN and finding the keyword *HBASE-<bug number>*. To determine the number of defects for each module, we sum up the number of bug fixes in all files within each module.

We chose releases so that the period of time between each release is 3 to 4 months. Choosing releases with near-equal time intervals reduces the effects of wide disparities between releases. For example, if one pair of releases in our study are weeks apart, while another pair are years apart, our prediction models may be affected by the large difference in time between the pairs of releases. As a result, we control for time to an extent. Our chosen approach for dealing with time intervals between releases is consistent with previous literature on prediction models for software engineering [46] and empirical studies on architectural co-change [45]. Additionally, this release interval resulted in obtaining the most number of releases as possible while still trying to account for time in a balanced manner for this study.

4.3 Data Splitting and Evaluation Metrics

We first discuss the splitting strategy we select for training our models and testing them. We then cover the two criteria we chose to evaluate the performance of our prediction models: predictive power and ranking.

Data Splitting. In order to evaluate the performance of the models, we use *data splitting*, a commonly used evaluation technique, where a data set is divided into subsets for building and evaluating the model. For evaluating the performance of our prediction models on release k, we use the data of all releases up to but not including that release to train the models, and then we use the data of release k as test data. We assess the performance of our prediction models for multiple releases depending on the number of releases for a project. For HBase and Camel, we evaluate our prediction models for the last three releases. For the remaining projects, we test the models on the last two releases.

1. http://www.scitools.com/

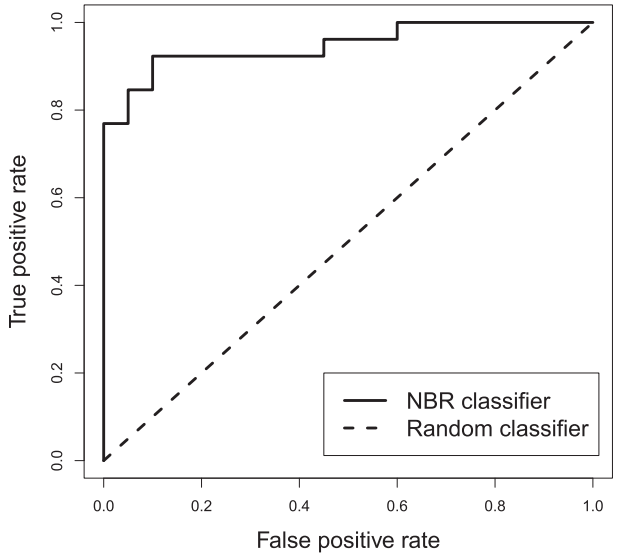


Fig. 5. ROC curve for defect prediction.

Predictive Power. We assess the predictive power of a model by selecting an appropriate performance measure. We considered a variety of measures often utilized to evaluate the performance of predictive models for software-engineering purposes. We will briefly discuss some commonly used measures—*accuracy*, *precision*, and *recall*—and why they are undesirable for our study. We then follow that discussion with an introduction and justification of our chosen measure for predictive performance: *area under the curve (AUC)* of the *receiver operating characteristic (ROC)*.

Precision and recall are pairs of performance measures commonly used together for prediction models. Precision is a measure of a model's ability to predict modules without falsely marking them as having low architectural quality. Recall is a measure of a model's ability to correctly predict all modules with low architectural quality. A prediction model should have a high precision and recall; however, increasing one often decreases the other.

Accuracy is the proportion of correct predictions, which can be a bad performance measure for imbalanced data [47]. For example, if we only have a few defective modules in our data set, a model that considers all modules as clean would have a high accuracy.

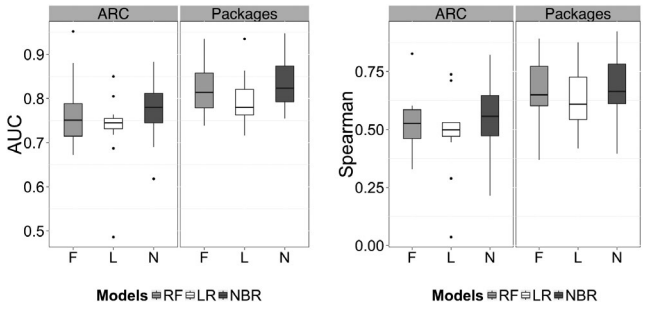
Precision, recall, and accuracy all require the arbitrary setting of discrimination thresholds to declare a module as having low architectural quality. To avoid arbitrary setting of thresholds in our experiments, we utilize AUC of ROC as the performance measure for comparing prediction models, as suggested by [34], and further described below.

The Receiver operating characteristic (ROC) is a curve that plots true positive rates (y-axis) against false positive rates (x-axis) for all possible thresholds between 0 and 1 that are used to convert a prediction model score to a class label—precluding the need to arbitrarily set thresholds. AUC is a scalar performance measure derived from ROC and is the area enclosed by the curve and the x-axis. AUC separates predictive performance from class and cost distributions, which are based on characteristics of projects. A class distribution represents the balance of class instances in the dataset (e.g., they can be uniform or imbalanced). A cost distribution represents the tradeoff between the true positive rate and false positive rate of a prediction model. In other words, AUC computes a predictive performance that is independent of the balance of class instances in the dataset or the tradeoffs between the false positive rate and true positive rate. The best possible model is a curve close to $y = 1$ with AUC of 1.0; a random classifier would obtain AUC of 0.5. In code-level defect prediction literature, an AUC of 0.7 or above is considered a high level of performance for a prediction model [34], [41]. Given the similarity of architectural decay and defects, we also consider AUC of 0.7 and above as a high level of performance for architectural-quality prediction.

For illustration, Fig. 5 shows an ROC curve corresponding to one of our models for predicting defects in architectural modules of the OpenJPA project. Every point on a ROC curve represents a threshold tradeoff between the true positive rate and false positive rate. For example, at the top-right end of the curve we identify all modules as defective, resulting in also a 100 percent false positive rate. Ideally, even on the left end of the curve, where we select thresholds that push the false positive rate to zero, we obtain high true positive rates. As a result, the ROC curve shows all the different tradeoffs between the rates that can act as discrimination thresholds. By choosing a different discrimination threshold for declaring a module defective, the prediction model would produce a different performance, as shown in this curve. Rather than reporting the results using an arbitrary threshold, we use AUC to holistically compare the classification performance of different prediction models under all possible thresholds.

Our approach for evaluating the prediction models is orthogonal to how the engineers would use the models in software projects. In practice, the engineer can choose a discrimination threshold that achieves the desired balance of precision and recall based on the characteristics of a project. For instance, if a project is understaffed and there are insufficient resources to thoroughly review the system's architecture/code, the engineer may choose a threshold that achieves a higher precision and a lower recall, meaning less wasted effort investigating false positives, at the expense of not fixing all architectural issues in time. On the other hand, if a project has the necessary staff and resources to thoroughly review the system's architecture/code, the engineer may choose a threshold that achieves a lower precision and a higher recall, meaning more wasted effort of investigating false positives, but increased likelihood of fixing all architectural concerns. For example, in a safety-critical software project, the engineers may choose to use thresholds that maximize the recall to reduce architectural decay factors, and thereby improve the quality of software, as much as possible.

Ranking. Determining the modules with the lowest architectural quality allows engineers to prioritize their efforts to those modules first. To that end, we assess if a model can correctly predict the order of modules according to their architectural-quality metrics. Ranking is not applicable to architectural smells since they are binary variables. However, we can obtain ranking results for defects and CF. In defect ranking, we build the prediction models using data splitting, predict the number of faults for each module, and



(a) AUC Performance Defects (b) Spearman Correlation for Ranking Defective Modules

Fig. 6. Defect prediction performance

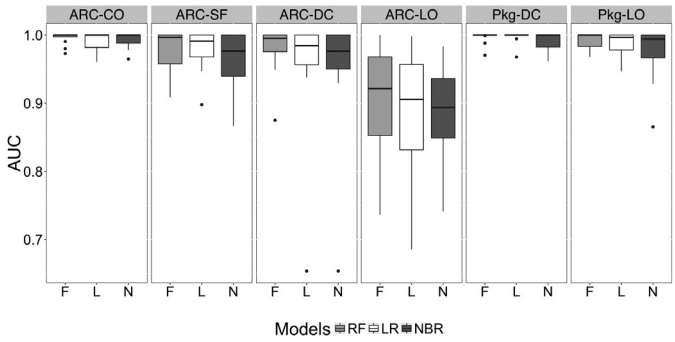


Fig. 7. AUC performance architectural smells. CO stands for concern overload; SF stands for scattered functionality; DC stands for dependency cycle; and LO stands for link overload.

compare the ordering of the predicted defect numbers with actual defect numbers using Spearman correlation. Similarly, we predict CF values for each module and compare the ranking of predicted CF values with the ranking of actual CF values.

We consider a Spearman correlation greater than 0.4 that is statistically significant at the 0.01 level to be a reliable ranking of modules. A correlation of 1.0 denotes a perfect ranking. Previous work on code-level defect prediction has considered Spearman correlation values greater than 0.4 to be sufficiently strong [6], [48]. Given the similarity of predicting code-level defects and architectural decay, this consideration is sensible for our prediction models. Note that all the Spearman correlations that we report are significant at the 0.01 level.

5 Experimental Results

Given our approach and the experimental design described in the previous sections, we now discuss the results obtained for each of our research questions. We begin by assessing the overall performance of our prediction models for each architectural-quality metric. We follow that study by assessing the degree of change for each architectural-smell metric. Afterwards, we focus on prediction results for smell emergence. Lastly, we determine the metrics that best predict architectural quality. For readers interested in extra details of results, we have provided an online appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2021.3060068 [2] that shows 58 additional box plots for each project displaying results for six predicted architectural metrics (cluster factor, defects, and smells)—associated with the different research questions we studied.

5.1 RQ1: Performance for Architectural-Quality Metrics

Defects. We first assess our model's ability to predict whether a module has at least one defect, which we refer to as *defect existence prediction*. Fig. 6a shows AUC results for defect existence prediction for RF (F), LR (L), and NBR (N), using both ARC and packages. Every box plot for a particular regression model (i.e., random forest, linear regression, or negative binomial regression) and architecture-recovery technique (i.e., ARC or Packages) represents a variety of statistics (e.g., minimum and maximum AUC values, different quartiles, and the median). Hence, we simply compute these different statistics and visualize them as those box plots. The results show that the prediction performance of NBR is higher than LR and RF. Particularly in the case of NBR, our models predict module defectiveness with AUC of at least 0.76.

Only predicting which modules have defects in future releases does not help in prioritizing modules for defect analysis and removal. Particularly, roughly 50 percent of modules in our study tend to have defects, which provides engineers with little information as to which modules should be allocated more maintenance resources. To address this issue, our models can predict the amount of defects a module may have. Predicting the magnitude of a module's defectiveness allows an engineer to prioritize modules for defect analysis and removal.

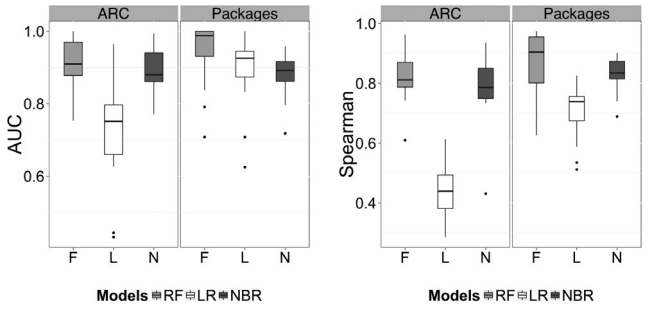
We assess our model's ability to predict a module's defectiveness by using Spearman correlation to compare the actual ranking of defective modules with our model's predicted rankings. Fig. 6b shows these results. As before, NBR outperforms LR and RF: Prediction for ARC modules obtains a Spearman correlation of 0.48-0.69; for packages, our models obtain a Spearman correlation of 0.62-0.73.

Smell Prediction. We determine whether our models can predict the occurrence of different types of smells by utilizing AUC as our performance measure. Fig. 7 shows the AUC results for predicting smells in ARC. We have the results of all four smells from ARC; however, two of the smells are concern-based and only applicable to ARC. The recovered architectures obtained by using packages as modules do not provide a representation of the system concerns needed to identify concern-based smells. Thus, for packages, we have results for DC and LO only.

Recall from Section 3.1 that ARC represents each source file as containing a set of concerns. These concerns are needed to identify SF and CO in a software system's architecture, precluding these types of smells from being determined from the package view. As shown in Fig. 7, we can predict the occurrences of smells in modules with a high AUC of 0.84 or above. Furthermore, LR, NBR, and RF obtain similar prediction results, in terms of AUC, for smells.

Cluster Factor. The overwhelming majority of modules in projects have low architectural quality as measured by CF. We consider a module m as having a low CF

2. http://tiny.cc/arch-prediction-appendix



(a) AUC Performance Cluster Factor (b) Spearman Correlation Cluster Factor

Fig. 8. Cluster factor prediction performance.

when CF < 0.2 for m. This CF value indicates that the vast majority of m's dependencies are with entities outside of m, as opposed to within m, indicating high coupling and low cohesion. Using the threshold of CF $<$ 0.2, we created a binary, independent variable that we used to assess the CF prediction performance in terms of AUC. We need to make the continuous variable CF into a discrete variable because AUC is applicable to classification problems. We achieve this by converting CF to a binary variable representing low and high CF.

Fig. 8a shows AUC results for predicting the CF values of modules. Similar to our previous results, RF and NBR outperform LR. Both RF and NBR obtain AUC values for CF of at least 0.71, demonstrating high effectiveness for predicting CF.

Given that modules mostly have low CF values, it is particularly important that engineers identify the modules with the worst CF. With such information, engineers can allocate maintenance resources to those modules first. To that end, we further assess the ranking results of CF.

Fig. 8b depicts the ranking results for CF values compared using Spearman correlation. For both ARC and packages, NBR and RF perform similarly, achieving more than 0.7 correlation, with RF performing slightly better than NBR. Both models outperform LR.

To illustrate how the results of this research might be used by the engineers, we describe one of the prediction models from Fig. 8b in more detail. We show the CF prediction results for a subset of packages in HBase version 0.92. Table 2 shows the actual values of CF for packages, the predicted value of CF, and also the corresponding ranking. As shown, the predicted values of CF are very close to the actual values of CF. Out of 15 modules, 12 modules are ranked correctly by the prediction model, while for the 3 remaining modules (i.e., handler, executor, and replication) the actual and predicted rankings are quite close. Engineers could use such information to identify architectural problems (e.g., identify the modules with low CF) and prioritize their effort (e.g., refactor the modules with lowest CF). For instance, the top-3 most decayed packages in Table 2 (i.e., `.thrift.generated, .client.coprocessor, and .io`) should be refactored to reduce coupling and increase cohesion.

RQ1 Summary. Overall, the results show that our models can effectively predict the different architectural-quality metrics. For most cases, NBR provides superior results and is the best overall model for predicting architectural quality.

5.2 RQ2: Changes for Architectural Smells

Fig. 9a shows the percentages of changes across all releases and systems for each architectural smell. We compute smell change σ_Δ for release r, which represents the ratio of smell emergence or removal of modules in release $r+1$ to all modules with smells in r, using the following equation:

$$\sigma_\Delta(M_r, r, r+1) = \frac{|\{m_a \in M_r : \sigma_\delta(m_a, r, r+1)\}|}{|\{m_b \in M_r : has\sigma(m_b)\}|} \times 100$$
$$\sigma_\delta(m, r, r+1) = \sigma_{em}(m, r, r+1) \vee \sigma_{re}(m, r, r+1),$$

M_r is the set of modules for release r. σ_{em} is true when module m has no smell in release r but has a smell in release $r+1$, and false otherwise—representing a smell emergence. σ_{re} is true when a module m has a smell in release r but does not have that same smell in release $r+1$, and is false otherwise—representing a smell being removed or changed to another smell. $has\sigma(m)$ returns true if module m has any smell, and false otherwise. Intuitively, the denominator calculates the number of modules for a release that have any smells; the numerator calculates the number of modules in the current release that will change in the next release.

Although all types of architectural smells change across releases, the amount of change varies: SF, DC, and CO

TABLE 2
Prediction of Cluster Factor (CF) for Packages in HBase (version 0.92)

Package Name	CF	Predicted CF	Rank of CF	Rank of Predicted CF
org.apache.hadoop.hbase.thrift.generated	0.00	0.01	1	1
org.apache.hadoop.hbase.client.coprocessor	0.01	0.02	2	2
org.apache.hadoop.hbase.io	0.03	0.03	3	3
org.apache.hadoop.hbase.replication	0.05	0.04	6	4
org.apache.hadoop.hbase.executor	0.04	0.04	4	5
org.apache.hadoop.hbase.master.handler	0.05	0.05	5	6
org.apache.hadoop.hbase.util.hbck	0.06	0.06	7	7
org.apache.hadoop.hbase.replication.regionserver	0.09	0.08	8	8
org.apache.hadoop.hbase.rest.client	0.11	0.10	9	9
org.apache.hadoop.hbase.mapreduce	0.11	0.11	10	10
org.apache.hadoop.hbase.mapred	0.13	0.14	11	11
org.apache.hadoop.hbase.regionserver	0.18	0.20	12	12
org.apache.hadoop.hbase.rest	0.22	0.24	13	13
org.apache.hadoop.hbase.io.hfile	0.25	0.28	14	14
org.apache.hadoop.hbase.filter	0.27	0.31	15	15

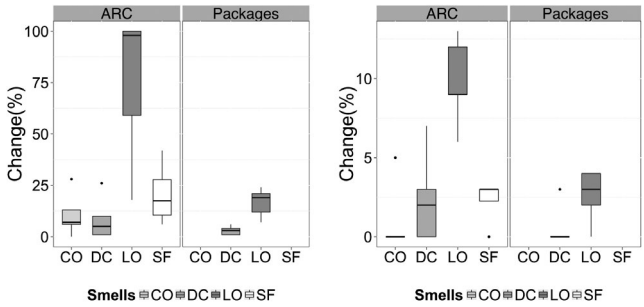


Fig. 9. Percentages of changes for architectural smells. CO stands for concern overload; DC stands for dependency cycle; LO stands for link overload; and SF stands for scattered functionality.

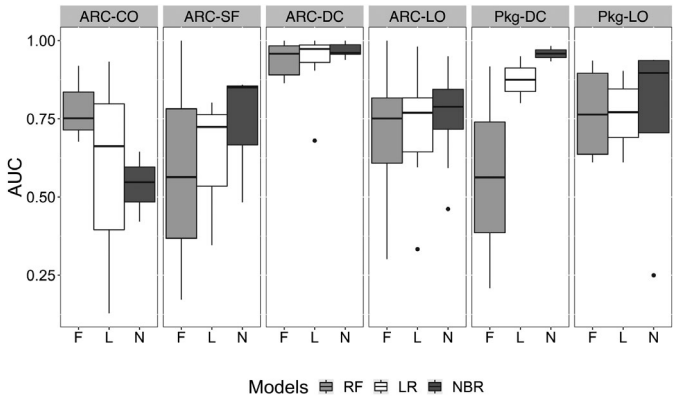


Fig. 10. AUC performance for architectural smell emergence. CO stands for concern overload; SF stands for scattered functionality; DC stands for dependency cycle; and LO stands for link overload.

exhibit relatively little change; LO changes drastically across all releases of our systems. The amount of change undergone by smells SF, DC, and CO is relatively small ranging from about 5 to 35 percent. However, LO change is highly substantial ranging from about 60 to 99 percent.

Overall, we find that architectural smells do exhibit significant change worth predicting. However, we would like to determine if our prediction models can forecast a particular type of architectural-quality change, i.e., smell emergence, so that engineers can possibly take action before a smell occurs—resulting in possible savings of future time and effort.

5.3 RQ3: Predicting Architectural-Smell Emergence

For this research question, we first assess the frequency of smell emergence. Fig. 9b shows the percentages of smell emergence in architectural modules across all systems and releases. We compute the percentage of smell emergence σ_{Δ}^{em} for release r using the following equation:

$$\sigma_{\Delta}^{em}(M_r, r, r+1) = \frac{|\{m_a \in M_r : \sigma_{em}(m_a, r, r+1)\}|}{|\{m_b \in M_r : has\sigma(m_b)\}|} \times 100$$

This equation is highly similar to σ_{Δ}; however, σ_{Δ}^{em} does not utilize σ_{re} and, thus, only accounts for smell emergence. Intuitively, σ_{Δ}^{em} computes the ratio of modules that experience smell emergence in release $r+1$ to all modules with smells for release r.

LO is the most frequent type of smell emergence with a median of 9 percent occurring for modules. SF and DC smell emergence occurs less than 5 percent in ARC; DC smell emergence does not occur in most projects. Although smell emergence occurs infrequently, this phenomenon is intuitively difficult to predict and preventing its occurrence may reduce future maintenance issues.

To build a model for predicting smell emergence cases, we created new binary variables for each smell: se_{co}, se_{dc}, se_{lo}, se_{sf}. se variables are equal to 1 whenever the value of the corresponding smell is 0 in the current release and 1 in the next release—meaning that the smell does not exist in the previous release, but it emerges in the next release. We created models for predicting smell emergence using these new dependent variables.

Fig. 10 shows the AUC prediction results for smell emergence for all systems and releases. Despite the number of smell-emergence instances being low, NBR predicts those instances with AUC of 0.83 on average. The performance of RF drops considerably for smell-emergence prediction compared to LR and NBR. This occurs because RF can lose significant performance when a dataset is extremely imbalanced [49]; however, stepwise regression with LR and NBR are less susceptible to imbalanced data.

In summary, our models can predict smell emergence—and architectural-quality metrics in general—with high performance.

5.4 RQ4: Factors Enabling Architectural Prediction

To obtain our prediction models, it is important to identify the metrics that best improve our prediction models. Our previous results show that prediction models using NBR tend to perform as well or outperform LR and RF in the majority of cases. Consequently, to answer RQ4 we focus on identifying the best metrics, obtained through stepwise regression, for NBR. We produced 50 prediction models for architectural quality using NBR. These were obtained from the combination of five systems, two architectural views (ARC and packages), and six dependent variables (defects, SF, CO, DC, LO and CF), where SF and CO are only applicable for ARC. Similarly, we constructed several prediction models for smell emergence. Due to the number of prediction models, we do not report the coefficient values and significance level of all of the independent variables in each model.[3]

Table 3 showcases the factors, i.e., independent variables, that contribute to prediction models for each quality metric: Each column represents an independent variable; each row represents a dependent variable. Factors for smell-emergence models are denoted by *-SE*. Values in the table depict the number of times each independent variable contributes to a prediction model. The maximum value in each cell is 10 (the combination of two architectural views and five systems). However, for concern-based smells (SF, CO, SF-SE and CO-SE), 5 is the maximum value, because the package view does not include such smells (denoted by highlighted row headers in Table 3). For example, LOC contributes to

3. Readers may find the study artifacts, including the prediction models and results, at: https://sites.google.com/view/forecasting-arch-decay

TABLE 3
Factors Contributing to Each Model

	LOC	SCC	DIT	CBO	LCM	NC	NCL	CMC	IMC	SF	CO	DC	LO	CMD	OMD	TCMD	TOMD	IMD	XMD
Defects	10	2	4	7	1	8	4	6	7	0	3	5	1	1	6	1	4	4	6
SF									1	4			1						
CO								1			5				1	1			
DC												10							
LO	3		1	4	1	3	3		1	1	2	2	9	2	1	4	3	4	4
CF	1			1												2	1	9	5
SF-SE	1	2	1	1			2		1				2	1	1	1			1
CO-SE	1							1		1		1				1			
DC-SE	1	3		1	1		1	3			1			2	3	3	1	2	2
LO-SE	4		1	4	1	3	1	2	2		1	2	1	4	3	3	2	3	8

all models for predicting defects and, thus, is included in all 10 models.

A wide variety of metric types, from all categories, are important factors–with values of at least 5—for predicting defects: lifted file-level metrics (LOC, CBO, and NC), architectural co-changes (CMC and IMC), architectural smells (DC), and architectural-dependency metrics (OMD and XMD).

In general, for three of the four types of architectural smells (SF, CO, and DC), the important factor for predicting those smells is if the smell exists for a module in the current release. For example, if a module has CO, it is likely to continue having CO in the next release. However, a wider variety of metrics are important factors for predicting LO.

Overall, these smell results indicate that architectural smells are rarely restructured, meaning that smell-oriented decay tends to remain in a system once it emerges. This result further motivates the need to predict smell emergence and prevent smell occurrence.

The important factors for predicting smell emergence are starkly different from predicting the general case of architectural quality: A wide variety of metrics predicted each type of smell emergence. This result indicates that smell emergence originates from a complex set of factors that warrants further research.

Overall, our results indicate that all categories of independent variables are important for predicting architectural quality. Unlike previous work for predicting defects in packages [8], [50], which only used lifted file-level metrics, we show that both lifted file-level metrics and architectural metrics are important for predicting architectural quality. Futhermore, stepwise regression using NBR provides the best results for such prediction.

6 Example Case

In the previous section, we relied on statistical criteria to empirically assess the performance of our prediction models. To determine the usefulness of these predictions from a practical perspective, we also manually studied some of the results produced by our models. Without being exhaustive, here we describe some of our findings in the case of the Camel project, providing concrete evidence as to how the prediction models can be useful in practice for identifying the architectural problems. We focus on Camel as a case study for two key reasons. First, Camel is a popular project with many commits and users, making it particularly interesting as a case study. Second, Camel is one of the larger projects in our study, with a higher number of LOC and versions.

We manually investigated whether architectural quality metrics, such as architectural smells, used in the construction of our prediction models, are indeed architectural problems the developers care about and aim to resolve. We conducted this investigation by reading through commit logs. We only report architectural problems described in the commit history by the developers. We found cases corroborating the validity of our quality metrics through the developers' commit logs and changes that involved restructuring of the system's architecture. For instance, our metrics identified the following four packages to have DC on 2/17/09: *component.cxf*, *component.cxf.util*, *converter.stream* and *converter*. Two months later, those packages no longer had a DC. To confirm our DC metric is properly capturing an issue in the architecture of the system, we looked at the log commits of Camel, filtered the changes that include those packages, and found the following messages:

- `revision: 749227, date: 3/2/2009, log message: CAMEL-588: LoggingLevel moved from model to root pacakge to improve API package.`
- `revision: 749236, date: 3/2/2009, log message: CAMEL-588: Fixed bad package tangle.`
- `revision: 749561, date: 3/3/2009, log message: CAMEL-588: Removed package dependency and using the type converter API to find the right converter instead of direct usage.`

We also looked at CAMEL-588 in Jira; the description of issue starts as follows: `"Currently there is a bad dependency cycle between camel, spi and model..."`. These comments clearly describe the same phenomenon intended to be measured by the DC metric (recall Section 3.3). Experiences such as this provide concrete evidence that architectural smell metrics can be effective in practice with helping the practitioners identify architectural problems and decaying elements.

We also found cases in which our smell emergence predictions were found to be issues that the developers had acknowledged in their commit logs and had attempted to resolve. A concrete example of this situation occurred with the *language.simple* package, which did not have DC for multiple releases, but our model predicted that it will start to have DC from version 2.5 (10/31/2010). When we manually investigated the commit logs, excerpts of which are shown below, not only did we find evidence of DC emergence, but also attempts by the developers to fix the problem afterwards:

- `revision: 1150991, date: 7/26/2011, log message: CAMEL-3961: Polished and reduced some package tangling`
- `revision: 1171490, date: 9/16/2011, log message: CAMEl-4457 Move types of the simple language to a new package simple.types to avoid dependency cycle`

The description of CAMEL-4457 in Jira summarizes the issue: "`Currently we have a big dependency cycle between language.simple and language.simple.ast`".

We believe using our smell-emergence prediction models, Camel developers could have identified and refactored the decaying architectural modules earlier.

These phenomena were not limited to DC. For example, we were able to predict *component.log* will not have the LO smell in a future release, even though it had that smell in previous releases. We found evidence in commit logs that the architecture of the system had been refactored in between the releases:

- `revision: 749193, date: 3/2/2009, log message: CAMEL-588: Package tangle fixes. Tokenizer in spring renamed to Tokenize. And fixed a CamelCase.`
- `revision: 749212, date: 3/2/2009, log message: CAMEL-588: Moved LoggingLevel from model to core package, to fix bad tangle.`

In summary, our analysis suggests that we can accurately predict many architectural quality concerns and that such concerns are indeed taken seriously by the developers of open-source software, as evidenced by commit logs showcasing their attempts to fix degraded architectural modules. We believe our prediction models could help developers detect architectural decay in a systematic fashion, possibly prior to its full manifestation in code.

7 Threats to Validity

We now describe the main threats to validity of our findings.

Construct validity is concerned with whether we are actually or accurately measuring the constructs we are interested in studying. One such threat involves the correctness of our linking of modules and their constituent files with defects. However, recall from Section 4.1 that the process used by engineers in ASF to link bug-fixing commits and issues significantly mitigates this threat. We chose fixed bugs instead of reported bugs because fixed bugs are verified to be legitimate by developers while reported bugs may not be verified by developers.

Another threat to construct validity involves the accuracy of the architectural modules we obtain. We address this threat in several ways: We selected a technique, ARC, that has exhibited high accuracy when compared to other techniques in previous work [21] and used the settings for ARC that worked well in previous work [15], [21] We further complement the semantic view provided by ARC with a structural view obtained through packages. The package-based view is often considered architectural [6], [7], [8], [26], [27], [28], [51], [52]. Both of these techniques obtained highly accurate architecture recoveries in previous work [15], [21] —even when the recovered architectures were compared to manually recovered architectures obtained with the actual architects of widely used software systems (e.g., Hadoop and the Bourne-again shell or Bash, for short) [53]. We choose these two different views because they are strikingly different views, resulting in highly different modules. Any inaccuracies in the identification of architectural modules would degrade the results of our predictions, reducing the possibility of accurately relating similar modules across releases. However, our models still achieve high performance.

The final threat to construct validity involves whether our selected metrics actually represent architectural decay or the factors that predict architectural quality. To ensure that we have a comprehensive set of metrics that represent architectural decay, we included three types of architectural-quality metrics: architectural defects, architectural smells, and CF. For the factors that may indicate architectural decay, i.e., the independent variables of our models, we selected a wide variety of metrics that do not overlap, in order to avoid the multicollinearity problem.

Threats to *external validity* involve the generalizability of our findings. One such threat is that all our projects are from ASF and are implemented in Java. To mitigate this threat, we selected projects from different application domains that vary in their sizes. Furthermore, Java is a widely used language, making our results more generalizable. Specifically, our results become particularly generalizable to the many software projects worldwide that are implemented in Java, or similar languages.

Another threat to external validity relates to the architectural views this paper utilizes for recovery. This paper uses a structural view corresponding the Development view in Kruchten's 4+1 view model [11], [17] or the Module view [16] from Clements *et al.* An architecture recovery for this view does not generalize to the execution-oriented views such as the Component-and-Connector view of Clements *et al.* or the Process view of Kruchten's model. Nevertheless, there is currently no architectural decay prediction technique based on the Development view or Module view which already has well-established and widely used and studied techniques (e.g., ACDC or ARC). As a result, this paper contributes a novel and important first step toward performing architectural decay prediction for other views, such as the Process view or the Component-and-Connector view.

Threats to *conclusion validity* are concerned with the correctness of relationships among variables. One potential threat to conclusion validity is the correctness of using an

architectural smell in an older version to help predict the introduction of or change to an architectural smell in a later version. Essentially, architectural smells can be both dependent and independent variables in our models. To mitigate this potential threat, we carefully construct prediction models using three regression models and show that, in fact, architectural smells do contribute to the ability to predict such smells in later versions (see Table 3).

Another threat to validity involves our choice of regression models and their parameterization. To ensure validity of these selections, we chose widely-used and well-established prediction models (i.e., LR, RF, and NBR). These models were parameterized using standard or widely-used parameter values or determined using standard algorithms (e.g., Akaike Information Criteria for stepwise regression).

8 Related Work

We overview prior work covering three areas: defect prediction; studies focused on architectural evolution or architectural decay; and studies concerned with architectural-quality metrics.

8.1 Defect Prediction

Several studies have shown that metrics mined from code change history can be effective in locating defect-prone code [40], [43], [54], [55], [56], [57], [58], [59], [60], [61], [62]. A number of studies use different statistical methods in order to predict the location or number of faults in a software system [42]. Ostrand *et al.* [32] developed a model based on NBR to predict the number of faults in files. Menzies *et al.* [41] demonstrated that the method for building prediction models is significantly more important than the attributes selected for those models. Similarly, prior work has demonstrated that architectural co-changes correlate with defects [44], [45].

While most of the bug prediction studies are at the file-level, some studies focus on the subsystem level. Mockus and Weiss [55] found that in a large switching software system, the number of subsystems modified by a change can be a predictor of whether the change results in a fault. Nagappan *et al.* [50] used post-release defect history of five Microsoft software systems and found that failure-prone software entities are statistically correlated with code complexity measures.

Zimmermann and Nagappan [63] investigated the architecture and dependencies in Windows Server 2003, demonstrating how the complexity of a subsystem's dependency graph can be used to predict the number of failures.

Several studies used packages as modules. Martin and Martin [64] introduced the Common Closure Principle (CCP) as a design principle about package cohesion. This principle implies that a change to a component may affect all the classes in that component, but should not affect other components. Although the authors introduce CCP as a guideline for good decomposition of architecture, they do not investigate its impact on software defects. Zimmermann *et al.* [6] showed that complexity metrics are indicators of defects in Eclipse using files and packages. Kamei *et al.* [7] showed that package-level predictions do not outperform file-level predictions when the effort needed to review or test the code is considered. Schroter *et al.* [8] showed that import dependencies can predict defects using both files and packages. Bouwers *et al.* [65] investigated twelve architecture metrics for their ability to quantify the encapsulation of an implemented architecture and used packages for evaluation.

While the majority of existing studies on defect prediction are at the file level, our study is at the architectural level. We further examine indicators of architectural decay and quality other than defects (i.e., architectural smells and modularization quality). Furthermore, existing studies of prediction models at the subsystem level used either packages as architectural modules or other pre-defined modules (e.g., studies on Windows that used binaries as architectural modules). In this work, we use packages and a recovery technique for identifying modules from source code. These recovered architectural views enable us to build architectural prediction models for any system.

8.2 Architectural Evolution and Decay

A wide variety of studies are concerned with architectural decay across multiple versions of a software system. None of the following studies aim to predict architectural quality or decay.

Two studies have examined architectural decay by using the reflexion method [66], a technique for comparing descriptive architectures (i.e., architectures as designed by its architects) and recovered architectures (i.e., architectures as represented by implementation-level artifacts). Brunet *et al.* [67] studied the evolution of architectural violations from four subject systems. Rosik *et al.* [68] conducted a case study using the reflexion method to assess whether architectural drift, i.e., unintended design decisions, occurred in their subject system and whether instances of drift remain unsolved.

A number of studies investigate architectural decay without using the reflexion method. In terms of novel techniques for investigating architectural decay, Hassaine *et al.* [69] present a recovery technique, which they use to study decay in three systems. van Gurp *et al.* [70] conduct two qualitative studies of software systems to better understand the nature of architectural decay and how to prevent it. D'Ambros et al. [71] present an approach for studying software evolution that focuses on the storage and visualization of evolution information at the code and architectural levels.

Other studies of architectural decay are more exploratory or descriptive in nature. Two studies [45], [51] examine the effects of code changes on architectural modules and architectural decay. Ernst *et al.* [72] surveyed 1,831 participants, mostly software engineers and architects, on technical debt, finding that architectural decisions are the most important form of technical debt.

Two studies focus on patterns that represent architectural decay. Mo *et al.* [73] study patterns of recurring architectural problems at the file and package level, finding evidence of proneness to errors and changes for such entities involved in such patterns. Le *et al.* [74] investigate the nature and impact of architectural smells.

Three other studies [75], [76], [77] focus on the impact of code anomalies, their relations, and co-occurrences on the possibility of architecture degradation.

8.3 Architectural-Quality Metrics

A variety of metrics have been established in the software-engineering literature that quantify architectural quality and are applicable to architectural modules. Most of the metrics focus on representing coupling and cohesion between architectural entities. Other metrics consider the concerns (i.e., concepts, roles, or responsibilities) of the software system. Furthermore, some metrics have been applied to studies of architectural evolution.

Several studies focus on coupling and cohesion metrics for architectural modules. Allen and Khoshgoftaar [78] define coupling and cohesion metrics based on information theory. Briand *et al.* [79] present coupling and cohesion metrics based on object-oriented design princples. Sarkar *et al.* [80], [81] defined a series of metrics concerned with quality at the module and object-oriented levels. Most of these metrics highly overlap with previous metrics and are based on coupling and cohesion. Many of these metrics overlap with constructs measured by our selected metrics, while others are dependent on specific technologies or are not fully automatable—precluding their inclusion in our study.

Sant'Anna *et al.* [82] present architectural metrics based on concerns. These metrics are highly similar to concern-based architectural smells and focus on aspect-oriented systems. They do not provide mechanisms for identifying concerns that are not aspect-oriented, precluding the use of these metrics for our study.

Wermelinger *et al.* [83] apply architectural-decay metrics across multiple releases of Eclipse, with a focus on coupling, cohesion, and stability metrics. Sangwan *et al.* [84] apply architectural complexity metrics to multiple versions of Hibernate. Finally, Zimmerman *et al.* [85] propose that true coupling is determined by studying revision histories and code-level entities rather than the decomposition of modules or files. None of this previous work aims to predict architectural quality, which is the focus of our research.

9 Conclusion

Architectural decay is a phenomenon of software systems that leads to defects and increases maintenance time and effort. To address this issue, we constructed models for predicting three types of architectural decay: architectural defects, architectural smells, and modularization quality. For 38 versions of five software systems, we can predict architectural decay with high performance across two architectural views—one semantic view and another structural view. Even when architectural smells suddenly emerge in a module, we can predict these rare cases with high performance (AUC of 0.79-0.96). We further discovered that architectural smells tend to remain in modules once they emerge. Lastly, we discovered that a wide variety of metrics—of which file-level metrics are only a subset—are needed to predict architectural decay.

To enable replication of our results and improvement over our approach for architectural-quality prediction, we make our prediction models and results available online [24].

Acknowledgments

This work was supported in part by Award CNS-1823262 from National Science Foundation. We would like to thank the anonymous reviewers and associate editor for their valuable feedback, which helped us to improve this work.

References

[1] A. Telea and L. Voinea, "Visual software analytics for the build optimization of large-scale software systems," *Comput. Statist.*, vol. 26, no. 4, pp. 635–654, Dec. 2011. [Online]. Available: http://link.springer.com/10.1007/s00180–011-0248-2

[2] T. A. Standish, "An essay on software reuse," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 5, pp. 494–497, Sep. 1984.

[3] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Syst. J.*, vol. 28, no. 2, pp. 294–306, 1989.

[4] S. Yau and J. Collofello, "Some stability measures for software maintenance," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 6, pp. 545–552, Nov. 1980.

[5] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[6] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. Thi3 Int. Workshop Predictor Models Softw. Eng.*, 2007, Art. no. 9.

[7] Y. Kamei, S. Matsumoto, A. Monden, K.-I. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proc. IEEE Int. Conf. Software. Maintenance*, 2010, pp. 1–10.

[8] A. Schroter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proc. ACM/IEEE Int. Symp. Empirical. Softw. Eng.*, 2006, pp. 18–27.

[9] J. Garcia *et al.*, "Identifying architectural bad smells," in *Proc. 13th Eur. Conf. Software. Maintenance Reeng.*, 2009, pp. 255–258.

[10] J. Garcia *et al.*, "Toward a catalogue of architectural bad smells," in, *Proc. 5th Int. Conf. Quality. Softw. Architectures: Architectures Adaptive Softw. Syst.*, 2009, pp. 146–162.

[11] P. Kruchten, "The 4+1 view model of architecture," *Softw. IEEE*, vol. 12, no. 6, pp. 42–50, Nov. 1995.

[12] M. Risi, G. Scanniello, and G. Tortora, "Architecture recovery using latent semantic indexing and k-means: An empirical evaluation," in *Proc. 8th IEEE Int. Conf. Softw. Eng. Formal Methods*, 2010, pp. 103–112.

[13] G. Bavota *et al.*, "Using structural and semantic measures to improve software modularization," *Empir. Softw. Eng.*, vol. 18, no. 5, pp. 901–932, 2013.

[14] G. Bavota *et al.*, "Software re-modularization based on structural and semantic metrics," in *Proc. 17th Work. Conf. Reverse Eng.*, 2010, pp. 195–204.

[15] D. M. Le *et al.*, "An empirical study of architectural change in open-source software systems," in *Proc. IEEE/ACM 12th Working Conf. Mining Software. Repositories*, 2015, pp. 235–245.

[16] P. Clements *et al.*, "Documenting software architectures: Views and beyond," in *Proc. 25th Int. Conf. Software. Eng.*, 2003, pp. 740–741, iSSN: 0270–5257.

[17] P. Kruchten, "Architectural Blueprints—The "4+1" View Model of Software Architecture," pp. 15, 1995.

[18] B. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, Mar. 2006.

[19] *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12–1990*, pp. 1–84, Dec. 1990.

[20] J. Garcia *et al.*, "Enhancing architectural recovery using concerns," in *Proc. 26th IEEE/ACM Int. Conf. Automated. Softw. Eng.*, 2011, pp. 552–555.

[21] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. IEEE/ACM 28th Int. Conf. Automated. Softw. Eng.*, 2013, pp. 486–496.

[22] T. Lutellier *et al.*, "Comparing software architecture recovery techniques using accurate dependencies," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 69–78.

[23] T. Lutellier *et al.*, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 159–181, Feb. 2018.

[24] "arch-prediction-tools on GitHub," 2020. [Online]. Available: https://github.com/jgarci40/arch-prediction-tools

[25] P. Kruchten, "Architecture blueprints–the '4+1' view model of software architecture," in *Proc. Ada's Role Global Markets: Solutions Changing Complex World*, 1995, pp. 540–555.

[26] F. Beck and S. Diehl, "Evaluating the impact of software evolution on software clustering," in *Proc. 17th Work. Conf. Reverse Eng.*, 2010, pp. 99–108.
[27] K. Kobayashi *et al.*, "Feature-gathering dependency-based software clustering using dedication and modularity," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 462–471.
[28] A. Corazza *et al.*, "Investigating the use of lexical information for software system clustering," in *Proc. 15th Eur. Conf. Softw. Maintenance Reeng.*, 2011, pp. 35–44.
[29] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.
[30] "CRAN - Package MASS," 2020. [Online]. Available: http://cran.r-project.org/web/packages/MASS/index.html
[31] "CRAN - Package randomForest," 2020. [Online]. Available: http://cran.r-project.org/web/packages/randomForest/index.html
[32] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.
[33] J. Cohen and J. Cohen, *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Mahwah, NJ, USA: L. Erlbaum Associates, 2003.
[34] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 2008.
[35] D. E. Farrar and R. R. Glauber, "Multicollinearity in regression analysis: The problem revisited," *Rev. Econ. Statist.*, vol. 49, no. 1, pp. pp. 92–107, 1967.
[36] A. Prinzie and D. Van den Poel, "Random multiclass classification: Generalizing random forests to random mnl and random nb," in *Proc. Int. Conf. Database Expert Syst. Appl.*, 2007, pp. 349–358.
[37] J. Wu, A. Hassan, and R. Holt, "Comparison of clustering algorithms in the context of software evolution," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance*, 2005, pp. 525–535.
[38] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 264–282, Mar./Apr. 2011.
[39] J. Garcia, "A unified framework for studying architectural decay of software systems," Ph.D. dissertation, Univ. Southern California, Los Angeles, CA, USA, 2014.
[40] M. Cataldo *et al.*, "Software dependencies, work dependencies, and their impact on failures," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 864–878, Nov./Dec. 2009.
[41] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.
[42] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories*, 2010, pp. 31–41.
[43] E. Shihab *et al.*, "High-impact defects: A study of breakage and surprise defects," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Foundations Softw. Eng.*, 2011, pp. 300–310.
[44] E. Kouroshfar, "Studying the effect of co-change dispersion on software quality," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 1450–1452.
[45] E. Kouroshfar *et al.*, "A study on the role of software architecture in the evolution and quality of software," in *Proc. 12th Working Conf. Mining Softw. Repositories*, 2015, pp. 246–257. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820548
[46] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *Proc. 1st Int. Symp. Empir. Softw. Eng. Meas.*, 2007, pp. 364–373.
[47] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.
[48] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 531–540. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368161
[49] C. Chen, A. Liaw, and L. Breiman, "Using random forest to learn imbalanced data," *Univ. California, Berkeley*, vol. 110, no. 24, pp. 1–12, 2004.
[50] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 452–461.
[51] M. Paixao *et al.*, "Are developers aware of the architectural impact of their changes?" in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 95–105. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155578
[52] M. Paixao *et al.*, "The impact of code review on architectural changes," *IEEE Trans. Softw. Eng.*, early access, Apr. 23, 2019, doi: 10.1109/TSE.2019.2912113.
[53] J. Garcia *et al.*, "Obtaining ground-truth software architectures," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 901–910. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486911
[54] T. Graves *et al.*, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.
[55] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Tech. J.*, vol. 5, no. 2, pp. 169–180, 2000.
[56] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.
[57] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *Proc. 16th Work. Conf. Reverse Eng.*, 2009, pp. 135–144.
[58] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292.
[59] S. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 1–12, Jan. 2001.
[60] S. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Visualizing software changes," *IEEE Trans. Softw. Eng.*, vol. 28, no. 4, pp. 396–412, Apr. 2002.
[61] D. Poshyvanyk *et al.*, "Using information retrieval based coupling measures for impact analysis," *Empir. Softw. Eng.*, vol. 14, no. 1, pp. 5–32, Feb. 2009.
[62] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 432–441.
[63] T. Zimmermann and N. Nagappan, "Predicting subsystem failures using dependency graph complexities," in *Proc. 18th IEEE Int. Symp. Softw. Rel.*, 2007, pp. 227–236.
[64] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C#*. Upper Saddle River, NJ, USA: Prentice Hall, 2007.
[65] E. Bouwers, A. van Deursen, and J. Visser, "Quantifying the encapsulation of implemented software architectures," in *Proc. 30th IEEE Int. Conf. Softw. Maintenance Evolution.*, 2014, pp. 211–220.
[66] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Trans. Softw. Eng.*, vol. 27, no. 4, pp. 364–380, Apr. 2001.
[67] J. Brunet *et al.*, "On the evolutionary nature of architectural violations," in *Proc. 19th Working Conf. Reverse Eng.*, 2012, pp. 257–266.
[68] J. Rosik *et al.*, "Assessing architectural drift in commercial software development: A case study," *Softw.: Practice Experience*, vol. 41, pp. 63–86, 2011.
[69] S. Hassaine, Y. Guéhéneuc, S. Hamel, and G. Antoniol, "Advise: Architectural decay in software evolution," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, 2012, pp. 267–276.
[70] J. van Gurp, S. Brinkkemper, and J. Bosch, "Design preservation over subsequent releases of a software product: A case study of baan erp," *J. Softw. Maintenance Evol.: Res. Practice*, vol. 17, no. 4, pp. 277–306, 2005.
[71] M. D'Ambros *et al.*, *Analysing Software Repositories to Understand Software Evolution*. Berlin, Germany: Springer, 2008.
[72] N. A. Ernst *et al.*, "Measure it? manage it? Ignore it? software practitioners and technical debt," in *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 2015, pp. 50–60. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786848
[73] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Proc. 12th Working IEEE/IFIP.*, 2015, pp. 51–60.
[74] D. M. Le *et al.*, "An empirical study of architectural decay in open-source software," in *Proc. IEEE Int. Conf. Softw. Archit.*, 2018, pp. 176–17 609.
[75] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *Proc. 2nd Int. Workshop Softw. Archit. Metrics*, 2015, pp. 1–7.
[76] I. Macia, R. ArcoverdeA. Garcia, C. Chavez, and Arndt von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, 2012, pp. 277–286.
[77] W. Oizumi, A. Garcia, L. Da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 440–451.

[78] E. Allen and T. Khoshgoftaar, "Measuring coupling and cohesion: an information-theory approach," in *Proc. 6th Softw. Metrics Symp.*, 1999, pp. 119–127.
[79] L. Briand, S. Morasca, and V. Basili, "Measuring and assessing maintainability at the end of high level design," in *Proc. Conf. Softw. Maintenance*, 1993, pp. 88–87.
[80] S. Sarkar, G. Rama, and A. Kak, "API-based and information-theoretic metrics for measuring the quality of software modularization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 14–32, Jan. 2007.
[81] S. Sarkar, A. Kak, and G. Rama, "Metrics for measuring the quality of modularization of large-scale object-oriented software," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 700–720, Sep./Oct. 2008.
[82] C. Sant'Anna *et al.*, "On the modularity of software architectures: A concern-driven measurement framework," in *Proc. Eur. Conf. Softw. Archit.*, 2007, vol. 4758, pp. 207–224.
[83] M. Wermelinger *et al.*, "Assessing architectural evolution: A case study," *Empir. Softw. Eng.*, vol. 16, pp. 623–666, 2011.
[84] R. S. Sangwan, P. Vercellone-Smith, and C. J. Neill, "Use of a multidimensional approach to study the evolution of software complexity," *Innovations Syst. Softw. Eng.*, vol. 6, pp. 299–310, 2010.
[85] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Proc. 6th Int. Workshop Princip. Softw. Evol.*, 2003, pp. 73–83.

Joshua Garcia (Member, IEEE) received the BS degree in computer engineering and computer science from the University of Southern California (USC), and the MS and PhD degrees in computer science from University of Southern California. He is an assistant professor with the School of Information and Computer Sciences at the University of California, Irvine. He conducts research in software engineering with a focus on software analysis and testing, software security, and software architecture. He is a member of the ACM and ACM SIGSOFT.

Ehsan Kouroshfar received the BS dehree in computer engineering from the Amirkabir University of Technology - Tehran Polytechnic, the MS degree in computer software engineering from Sharif University of Technology, and the PhD degree in computer science from George Mason University. He is a software engineer with Amazon.com.. His research interests include software engineering, particularly software architecture and empirical software engineering.

Negar Ghorbani received the BSc degree in software engineering from the Computer Engineering Department of Sharif University of Technology. She is currently working toward the PhD degree in the software engineering program at the School of Information and Computer Sciences at the University of California, Irvine. Her research interests include software engineering with a focus on software analysis and testing, software security, and software architecture decay.

Sam Malek (Member, IEEE) received the BS degree in information and computer science from University of California, Irvine, and the MS and PhD degrees in computer science from the University of Southern California. He is a professor with the School of Information and Computer Sciences at the University of California, Irvine (UCI). He is also the director of the Institute for Software Research at UCI. His main research interests include software engineering, and to date his focus has spanned the areas of software architecture, software security, and software analysis and testing. He has received numerous awards for his research contributions, including the US National Science Foundation CAREER Award. He is a member of the ACM and ACM SIGSOFT.