ISR Institute for Software Research
University of California, Irvine

Architectural Decay Prediction from Evolutionary History of Software

Joshua Garcia
University of California, Irvine
joshug4@uci.edu

Ehsan Kouroshfar
George Mason University
ekouroshfar@gmail.com

Sam Malek
University of California, Irvine
malek@uci.edu

November 2018
ISR Technical Report # UCI-ISR-18-7

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
isr.uci.edu

Architectural Decay Prediction from Evolutionary History of Software

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

Abstract—As a software system evolves, its architecture tends to decay, leading to the occurrence of defects or architectural elements that become resistant to maintenance. 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, Decay, Defects, Prediction.

1 Introduction

In a software system's life cycle, software maintenance tends to dominate other activities, in terms of time, effort, and cost. Throughout that life cycle, a major artifact that must undergo maintenance is a software system's architecture, which determines the key properties of a software system. 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* [1], 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 must be able to predict 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 [2]–[4]. However, defects are not the only forms of architectural decay [5], [6]. Futhermore, packages represent a structural view of the architecture [7]. 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.

To stem architectural decay, techniques need to be constructed that predict a variety of constructs related to architectural quality, including indicators of architectural decay, i.e., *architectural bad smells* [5], [6], and the *quality of an architecture's modularization* [8]. 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 an error that should be fixed in all cases, unlike a defect. Determining that an architectural module is decaying, even before it is involved in an architectural smell or exhibits low modularization quality, can reduce maintenance time and effort.

To forecast architectural decay, we constructed *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 [9], [10] shown to work accurately based on the latest evaluations of techniques for recovering a software system's architecture [10], [11]. 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 five systems. 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 results of 0.76-0.88 and the occurence 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.

- *Joshua Garcia and Sam Malek are with the Department of Informatics, Institute for Software Research, University of California Irvine, Irvine, CA.*
  E-mail:{joshug4,malek}@uci.edu
- *Ehsan Kouroshfar is with the Department of Computer Science, George Mason University, Fairfax, VA.*
  E-mail: ekourosh@gmu.edu

- We study the degree of change for architectural smells that modules may suffer from and observe that some smells exhibit little change across releases, while others significantly change.
- Although at most 12% of modules exhibit *smell emergence*—which represent sudden occurrence of smells in modules—we are able to predict them 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.

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.

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. 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 between two consecutive releases?*

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?*

3 Prediction Model Construction

Figure 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.

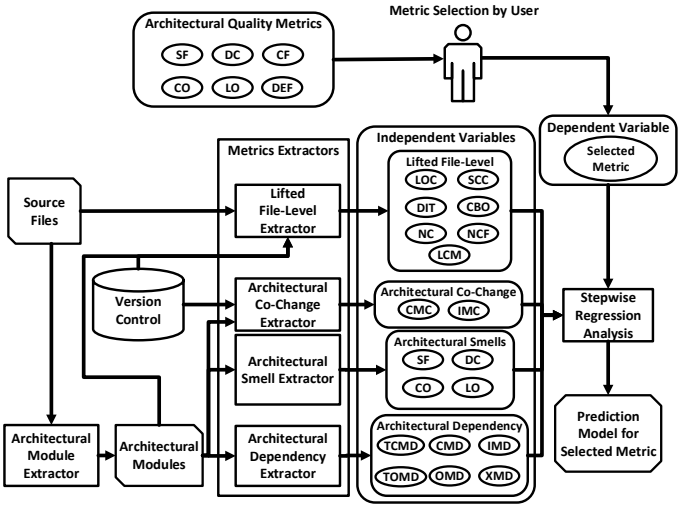


Fig. 1: Overview of our approach for architectural-quality metric prediction

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 [12], 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 [13]–[15]. 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 our previous work, we showed that high-level packages are not suitable for studying the evolution of architecture—due to the coarse granularity—and low-level packages should be used instead [16]. Therefore, we use low-level packages in this study. In low-level packages, architectural modules correspond to packages that only contain Java classes and no sub-packages.

In addition to packages, we include a semantic view of modules obtained using an architecture-recovery technique called *Architectural Recovery using Concerns (ARC)* [9], [10], [17], which utilizes hierarchical clustering and information retrieval to produce modules. ARC leverages a statistical language model, Latent Dirichlet Allocation (LDA) [18], 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.

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 [17].

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 [19] for building LR and NBR and the *randomForest* library for RF [20].

Although LR is popular and widely used in the literature, some have argued that NBR is a more appropriate regression model for defect prediction [21]. 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) [22]. We also chose RF since it has been shown to perform best for software defect prediction [23], 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 [22].

We do not want our prediction metrics to exhibit multi-collinearity, a phenomenon where prediction metrics are correlated, since this can cause our prediction models to become unstable [24]. 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 [25], 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 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)* [8], a metric used widely in previous architectural studies [8], [10], [26], [27] that represents the coupling and cohesion of a module. We calculate CF for a module m as follows:

$$CF_m = \frac{\mu_i}{\mu_i + 0.5 \times \sum_j \varepsilon_{ij} + \varepsilon_{ji}}$$

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 [5], [6], [28].

- *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 [28].
- *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.

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. As another example, if module m_2 is involved in a DC with other modules, $s_{dc} = 1$ for m_2.

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 [21], [29]–[31]. 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 C is the number of other classes to which C 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% 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.
- *Number of co-changed files (NCF)* is the number of other files that a file f is changed with [32].

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 [2], [3].

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. Our previous work has demonstrated that architectural co-changes correlate with defects [16], [33]. 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.

TABLE 1: Studied Projects and Release Information

Project	Description	Releases	SLOC
HBase	Distributed Scalable Data Store	0.1.0 ,0.1.3 ,0.18.0 ,0.19.0, 0.19.3, 0.20.2, 0.89.20100621, 0.89.20100924, 0.90.2, 0.90.4, 0.92.0	39K-246K
Hive	Data Warehouse System for Hadoop	0.3.0, 0.4.1, 0.5.0, 0.6.0, 0.7.0, 0.8.1	66K-226K
OpenJPA	Java Persistence Framework	1.0.1, 1.0.3, 1.1.0, 1.2.0, 2.0.0-M3, 2.0.1	153K-407K
Camel	Enterprise Integration Framework	1.6.0 ,2.0.M ,2.2.0 ,2.4.0, 2.5.0, 2.6.0, 2.7.1, 2.8.0, 2.8.3	99K-390K
Cassandra	Distributed Database Management System	0.3.0 ,0.4.1 ,0.5.1 ,0.6.2, 0.6.5, 0.7.0	50K-90K

- *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.

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 and Data Collection

Our experimental subjects include five projects, listed in Table 1. They are all written in Java and are maintained by Apache Software Foundation (ASF). However, they vary in their sizes and application domains, allowing us to draw broader conclusions. For our experiments, we excluded test code, since such code is generally not part of the system's architecture, from a traditional architectural perspective [7].

Project Statistics. As part of our overview of the projects that we studied, we provide statistics showing the number of modules, smells, and defects as box plots that present these numbers across our five studied projects and recovered architectural views. In the following paragraphs we introduce and discuss those plots.

Figure 2 shows the number of modules across different releases and projects for both ARC and packages. We set ARC to produce a number of modules equivalent to 20% 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 [10]. 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.

Figure 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.

Figure 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

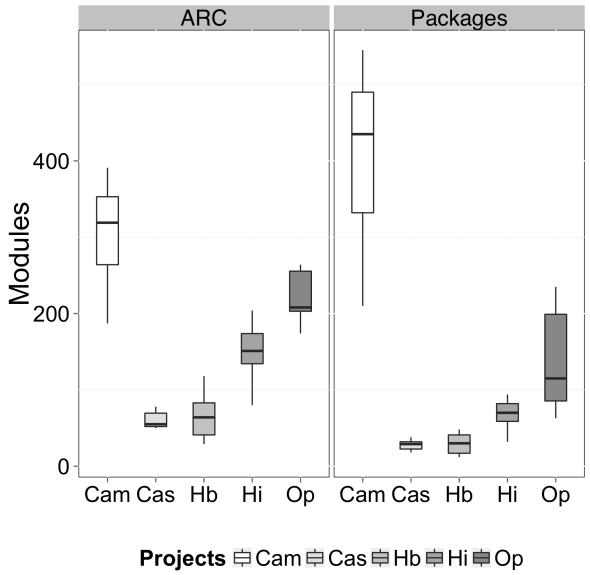


Fig. 2: Number of Modules Across Projects.

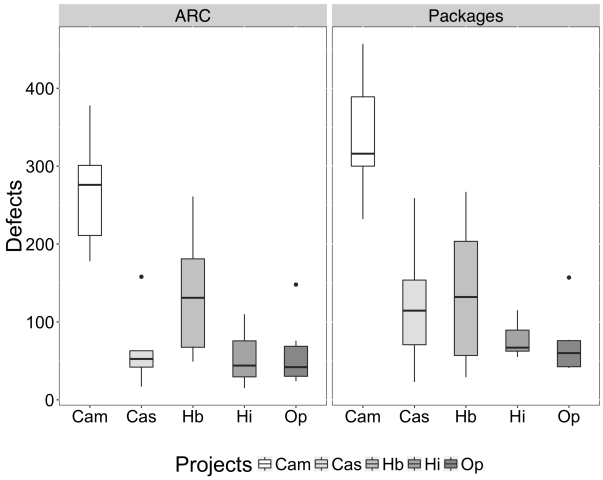


Fig. 3: Number of Defects Across Projects.

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).

Data Collection. 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

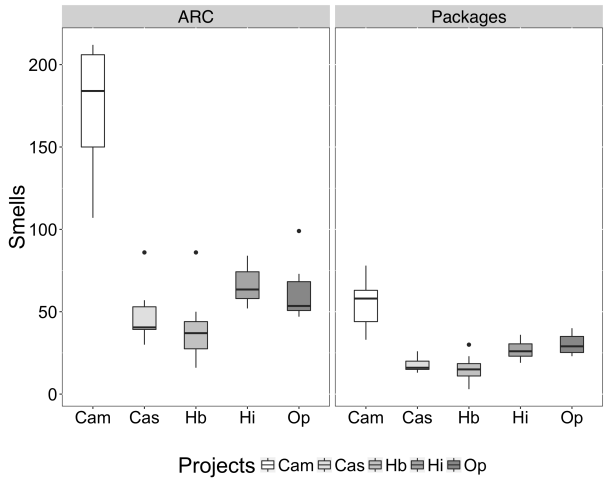


Fig. 4: Number of Smells Across Projects.

measured using UNDERSTAND from Scitools[1] for each release.

The change metrics (*NC, NCF, 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)* [17], [28], 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*).

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 [34] and empirical studies on architectural co-change [16].

4.2 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.

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 [35]. 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 [23], and further described below.

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—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. 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 [23], [30]. 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, Figure 5 shows an ROC curve corresponding to one of our models for predicting defects in architectural modules of OpenJPA project. 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

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

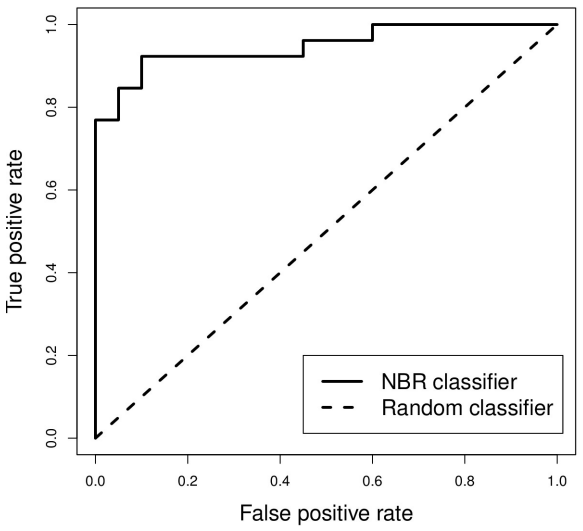


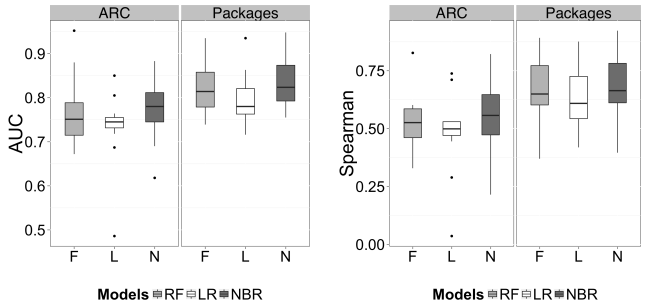
Fig. 5: ROC Curve for Defect Prediction.

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. As another 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 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



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

Fig. 6: Defect Prediction Performance

correlation values greater than 0.4 to be sufficiently strong [2], [36]. 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.

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

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*. Figure 6a shows AUC results for defect existence prediction for RF (F), LR (L), and NBR (N), using both ARC and packages. 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.

We further observe that AUC results for module-level defect prediction are higher for packages than ARC. This higher performance for packages may result from the fact that no special technique is needed to obtain packages and are, thus, less susceptible to error. However, given that our models can obtain at least 0.76 AUC, they exhibit resilience to errors that may exist in ARC.

Only predicting which modules have defects in future releases does not help in prioritizing modules for defect analysis and removal. Particularly, roughly 50% 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, rather than simply whether a module has a defect. Predicting the magnitude of a module's defectiveness allows an engineer to prioritize modules for defect analysis and removal.

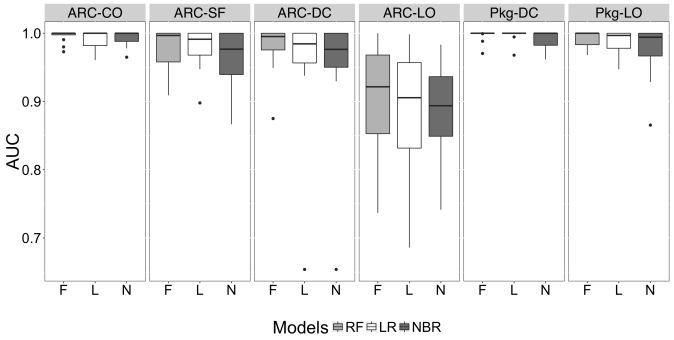
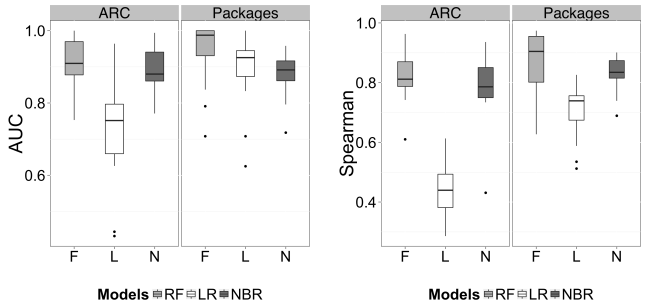


Fig. 7: AUC Performance Architectural Smells



(a) AUC Performance Cluster Factor

(b) Spearman Correlation Cluster Factor

Fig. 8: Cluster Factor Prediction Performance

We assess our model's ability to predict the extent of a module's defectiveness by using Spearman correlation to compare the actual ranking of defective modules with our model's predicted rankings. Figure 6b shows these results. As in the case of defect existence prediction of modules, NBR outperforms LR and RF: Prediction for ARC modules obtains Spearman correlation of 0.48-0.69; for packages, our models obtain a spearman correlation of 0.62-0.73. Similar to defect existence prediction for modules, ranking results are higher for packages than ARC. Again, this is likely due to error introduced by ARC when recovering modules.

For smell prediction, we determine whether our models can predict the occurrence of different types of smells. To that end, we utilize AUC as our performance measure. Figure 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. 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 Figure 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. Overall, prediction results are better for packages than ARC modules, which is consistent with the prediction results for defects.

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 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.

Figure 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. Just as with AUC performance for defects, AUC values for CF are higher for packages than ARC. This result further indicates that our models are resilient to errors that may exist in ARC.

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.

Figure 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. The superior performance of NBR and RF is significantly more pronounced for ARC. This difference in CF may be due to ARC ignoring dependency-based coupling and cohesion, which is what CF is based on.

To illustrate how the results of this research might be used by the engineers, we describe one of the prediction models from Figure 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 is 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).

In summary, 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.

Next we report the results of the amount of change for each architectural-smell metric.

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

Figure 9a shows the percentages of changes across all releases and systems for each architectural smell. We compute smell change σ_Δ for release r using the following equation:

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

$$\sigma_\delta(m, r, r+1) = \sigma_{em}(m, r, r+1) \cup \sigma_{re}(m, r, r+1)$$

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

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

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

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 exhibit relatively little change; LO changes drastically across all releases of our systems.

Our results for smell changes indicate that, for our selected systems, modules that suffer from concern-based smells (CO and SF) tend to retain those smells across releases—with little addition or removal of such smells afterwards. At the same time, change for SF is significantly higher than CO.

For each dependency-based architectural smell (DC and LO), change across releases varies significantly. The amount of change represented by LO varies drastically between ARC and packages. This difference is likely due to the fact that ARC does not take dependencies into account, which are used to compute LO.

DC exhibits a similar amount of architectural change, across releases and systems, for both ARC and packages. Furthermore, the amount of change for DC is quite low (largely between 1%-26%). Consequently, across releases, the same modules tend to be involved in a DC, for our selected systems.

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. To that end, we examine the results for our next research question:

> ***RQ3:*** *Can we effectively predict architectural-smell emergence between two consecutive releases?*

As part of answering this research question, we first assess the frequency of smell emergence. Figure 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:

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

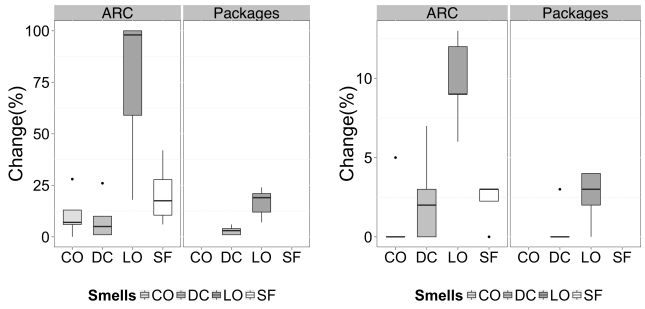
This equation is highly similar to σ_{Δ}; however, σ_{Δ}^{em} does not utilize σ_{re} and, thus, only accounts for smell emergence.

LO is the most frequent type of smell emergence with a median of 9% occurring for modules. SF and DC smell emergence occurs less than 5% 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 one 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.

Figure 10 shows the AUC prediction results for smell emergence for all systems and releases. Although the number of smell-emergence instances are low, we predict those instances with AUC of 0.83 on average using NBR.

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 [37]; however, stepwise regression with LR and NBR are less susceptible to imbalanced data.



(a) Any Changes (b) Smell Emergence

Fig. 9: Percentages of Changes for Architectural Smells

In summary, our models can predict smell emergence—and architectural-quality metrics in general—with high performance. To obtain such prediction models, it is important to identify the metrics that best improve our prediction models. We make that determination as we answer the following research question:

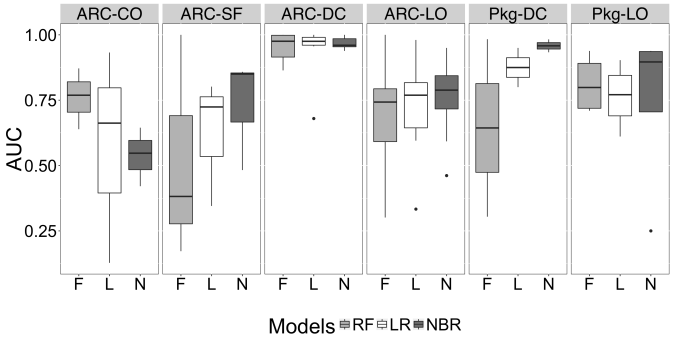


Fig. 10: AUC Performance for Architectural Smell Emergence

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

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. [2]

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 architectural smells (SF, CO, SF-SE and CO-SE), 5 is the maximum value, because the package view does not include such smells. For example, LOC contributes to 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 factors for predicting CF are mainly from the architectural-dependency metrics. Given that CF is a measure of coupling and cohesion based on architectural dependencies, this result is intuitive and expected.

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 [4], [38], 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 Discussion

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 LOC, 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 studied.

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 found many cases corroborating the validity of our quality metrics through the developers' commit logs and changes that involved restructuring of the system's architecture. As a case in point, our metrics identified the following four packages to have DC on 2/17/09: *component.cxf*, *component.cxf.util*, *converter.stream* and *converter*. But those packages did not have a DC in a version that was released two months later. To confirm our DC metric is indeed 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.`

2. Readers may find the study artifacts, including the prediction models and results, at: http://www.ics.uci.edu/~seal/projects/archprediction

TABLE 3: Factors contributing to each model

	LOC	SCC	DIT	CBO	LCM	NC	NCF	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

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 many 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.

Our experiences were not limited to DC. As another case in point, we were able to predict *component.log* will not have the LO smell in a future release, even though it had that smell in preceding releases. When we investigated the commit logs, we found evidence 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 software 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.

Another threat to construct validity has to do with the accuracy of the architectural modules we obtain. We address this threat in several ways: We selected a technique, ARC, that has exhibited higher accuracy when compared to other techniques in previous work [10]. We further complement the semantic view provided by ARC with a structural view obtained through packages. Additionally, any inaccuracies in our identification of architectural modules would only degrade the results of our predictions, by potentially reducing the possibility of accurately relating similar modules across releases. However, our models still achieve high performance. Nevertheless, to ensure that the architectural modules we obtained are meaningful, we attempted to use our prediction models on randomly generated modules for each of our five subject systems. Given that the resulting modules have random files in them, no traceability can be achieved between modules—i.e., there is no longer a module m_{k+1} in release $k+1$ that is similar to a module m_k in release k. This result further validates that we obtain meaningful architectural modules.

Another manner of constructing an architecture to consider inaccuracy of identified architectural modules is to simply take any release $r+1$ and, for any new entities not existing in the previous

release r, randomly assign such entities to an existing module. However, there is no reason to believe such modules would be accurate, in the general case: An architecture in industrial or open-source software systems is not created at random; instead, it is affected by explicit design decisions, and design decisions influenced by implementation decisions made in, largely non-random fashions, by developers [39]. Therefore, there is no reason to believe that such modules would be meaningful, whether for simply understanding a system's architecture or performing a maintenance task requiring modification to the existing system.

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 involves the fact that we only include open-source projects. However, ASF projects are widely used, even in industrial settings, which allows our study's results to generalize further, because our study's projects are more likely to represent software that is actually built and used in real-world industrial settings.

8 Related Work

We overview prior work covering three areas: defect prediction, one of the most commonly studied prediction models in software-engineering literature; 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 areas [29], [32], [40]–[48].

There are several studies that use different learning techniques and statistical methods in order to predict the location or number of faults in a software system [31]. Ostrand et al. [21] developed a model based on NBR to predict the number of faults in files. Menzies et al. [30] demonstrated that the method for building prediction models is significantly more important than the attributes selected for those models.

While most of the bug prediction studies are at the file-level, some studies focus on the subsystem level. Mockus and Weiss [41] 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. [38] 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 [49] 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 [50] 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 the impact of it on software defects. Zimmermann et. al [2] showed that complexity metrics are indicators of defects in Eclipse using files and packages. Kamei et. al [3] 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 [4] showed that import dependencies can predict defects using both files and packages. Bouwers et. al [51] investigated twelve architecture metrics for their ability to quantify the encapsulation of an implemented architecture and used packages for evaluation.

In summary, while the majority of existing studies on defect prediction are at the file level, our study is at the architectural level. We further examine other 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 recovery techniques for identifying modules from source code. These recovered architectural views enable us to build architectural prediction models for any system, even if a ground-truth architecture is unavailable.

8.2 Architectural Evolution and Decay

Several 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 [52], 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. [53] studied the evolution of architectural violations from four subject systems. Rosik et al. [54] 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.

Four additional studies investigate different facets of architectural decay. Hassaine et al. [55] present a recovery technique, which they use to study decay in three systems. van Gurp et al. [56] conduct two qualitative studies of software systems to better understand the nature of architectural decay and how to prevent it. D'Ambros et al. [57] present an approach for studying software evolution that focuses on the storage and visualization of evolution information at the code and architectural levels. Mo et al. [58] 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.

In our previous work [16], we studied the effects of changes spanning architectural modules and within architectural modules

on the likeliness of introducing defects in those modules. Our study considered multiple architectural views and identified that, except for high-level packages, changes spanning architectural modules are more likely to introduce architectural defects than those restricted to individual modules.

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 [59] define coupling and cohesion metrics based on information theory. Briand et al. [60] present coupling and cohesion metrics based on object-oriented design princples. Sarkar et al. [61], [62] 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. [63] 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. [64] apply architectural-decay metrics across multiple releases of Eclipse, with a focus on coupling, cohesion, and stability metrics. Sangwan et al. [65] apply architectural complexity metrics to multiple versions of Hibernate. Finally, Zimmerman et al. [66] 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 40 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.

In the future, we intend to move beyond prediction of architectural decay by determining the specific actions that can be taken to prevent decay once it occurs. One possible direction is the utilization of the important factors of our prediction models to identify specific preventative measures. As an example, a promising possibility is applying architectural restructurings that minimize coupling between modules. Once such preventive measures are applied, we can perform further assessment. For example, we can conduct a study to examine whether engineers can more quickly or easily perform maintenance tasks, after restructurings are applied.

To enable replication of our results and improvement over our approach for architectural-quality prediction, we make our prediction models and results available online at http://www.ics.uci.edu/~seal/projects/archprediction.

10 Acknowledgment

We would like to acknowledge Nenad Medvidovic, Duc Le, Pooyan Behnamghader, Daniel Link, and Arman Shahbazian from the University of Southern California for their invaluable assistance with this study, specifically their feedback on the formulation of experiments and their help with running the ARCADE environment.

References

[1] D. E. Perry, A. L. Wolf, Foundations for the Study of Software Architecture, SIGSOFT Softw. Eng. Notes 17 (4) (1992) 40–52. doi:10.1145/141874.141884.

[2] T. Zimmermann, R. Premraj, A. Zeller, Predicting Defects for Eclipse, in: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, PROMISE '07, IEEE Computer Society, Minneapolis, MN, USA, 2007, pp. 9–. doi:10.1109/PROMISE.2007.10.

[3] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, A. E. Hassan, Revisiting Common Bug Prediction Findings Using Effort-aware Models, in: Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM '10, IEEE Computer Society, Timisoara, Romania, 2010, pp. 1–10. doi:10.1109/ICSM.2010.5609530.

[4] A. Schroter, T. Zimmermann, A. Zeller, Predicting Component Failures at Design Time, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE '06, ACM, Rio de Janeiro, Brazil, 2006, pp. 18–27. doi:10.1145/1159733.1159739.

[5] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying Architectural Bad Smells, in: 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 2009, pp. 255–258. doi:10.1109/CSMR.2009.59.

[6] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Toward a Catalogue of Architectural Bad Smells, in: Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems, QoSA '09, Springer-Verlag, East Stroudsburg, PA, USA, 2009, pp. 146–162. doi:10.1007/978-3-642-02351-4_10.

[7] P. Kruchten, The 4+1 view model of architecture, Software, IEEE 12 (6) (1995) 42–50. doi:10.1109/52.469759.

[8] B. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the Bunch tool, IEEE Transactions on Software Engineering 32 (3) (2006) 193–208. doi:10.1109/TSE.2006.31.

[9] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, Y. Cai, Enhancing Architectural Recovery Using Concerns, in: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, IEEE Computer Society, Lawrence, KS, USA, 2011, pp. 552–555. doi:10.1109/ASE.2011.6100123.

[10] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), Palo Alto, CA, USA, 2013, pp. 486–496. doi:10.1109/ASE.2013.6693106.

[11] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, R. Kroeger, Comparing software architecture recovery techniques using accurate dependencies, in: Proceedings of the 37th International Conference on Software Engineering, 2015.

[12] P. Kruchten, Architecture blueprints–the '4+1' view model of software architecture, in: Tutorial Proceedings on Ada's Role in Global Markets: solutions for a changing complex world, TRI-Ada '95, ACM, Anaheim, CA, USA, 1995, pp. 540–555. doi:10.1145/216591.216611.

[13] F. Beck, S. Diehl, Evaluating the Impact of Software Evolution on Software Clustering, in: 17th Working Conference on Reverse Engineering, Beverly, Massachusetts, 2010, pp. 99–108. doi:10.1109/WCRE.2010.19.

[14] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, A. Matsuo, Feature-gathering dependency-based software clustering using dedication and modularity, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, 2012, pp. 462–471. doi:10.1109/ICSM.2012.6405308.

[15] A. Corazza, S. Di Martino, V. Maggio, G. Scanniello, Investigating the use of lexical information for software system clustering, in: Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, 2011, pp. 35–44. doi:10.1109/CSMR.2011.8.

[16] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, Y. Cai, A study on the role of software architecture in the evolution and quality of software, in: Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 246–257.
URL http://dl.acm.org/citation.cfm?id=2820518.2820548

[17] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, N. Medvidovic, An empirical study of architectural change in open-source software systems, in: Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 235–245.
URL http://dl.acm.org/citation.cfm?id=2820518.2820547

[18] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent Dirichlet Allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.

[19] CRAN - Package MASS, http://cran.r-project.org/web/packages/MASS/index.html.

[20] CRAN - Package randomForest, http://cran.r-project.org/web/packages/randomForest/index.html.

[21] T. Ostrand, E. Weyuker, R. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (4) (2005) 340–355. doi:10.1109/TSE.2005.49.

[22] J. Cohen, J. Cohen, Applied multiple regression/correlation analysis for the behavioral sciences, L. Erlbaum Associates, Mahwah, N.J., 2003.

[23] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings, IEEE Transactions on Software Engineering 34 (4) (2008) 485–496. doi:10.1109/TSE.2008.35.

[24] D. E. Farrar, R. R. Glauber, Multicollinearity in regression analysis: The problem revisited, The Review of Economics and Statistics 49 (1) (1967) pp. 92–107.

[25] A. Prinzie, D. Van den Poel, Random multiclass classification: Generalizing random forests to random mnl and random nb, in: R. Wagner, N. Revell, G. Pernul (Eds.), Database and Expert Systems Applications, Vol. 4653 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 349–358. doi:10.1007/978-3-540-74469-6_35.

[26] J. Wu, A. Hassan, R. Holt, Comparison of clustering algorithms in the context of software evolution, in: 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, 2005, pp. 525–535. doi:10.1109/ICSM.2005.31.

[27] K. Praditwong, M. Harman, X. Yao, Software module clustering as a multi-objective search problem, Software Engineering, IEEE Transactions on 37 (2) (2011) 264–282. doi:10.1109/TSE.2010.26.

[28] J. Garcia, A unified framework for studying architectural decay of software systems, Ph.D. thesis, University of Southern California (2014).

[29] M. Cataldo, A. Mockus, J. Roberts, J. Herbsleb, Software Dependencies, Work Dependencies, and Their Impact on Failures, IEEE Transactions on Software Engineering 35 (6) (2009) 864–878. doi:10.1109/TSE.2009.42.

[30] T. Menzies, J. Greenwald, A. Frank, Data Mining Static Code Attributes to Learn Defect Predictors, IEEE Transactions on Software Engineering 33 (1) (2007) 2–13. doi:10.1109/TSE.2007.256941.

[31] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, in: 7th IEEE Working Conference on Mining Software Repositories, Cape Town, South Africa, 2010, pp. 31–41. doi:10.1109/MSR.2010.5463279.

[32] E. Shihab, A. Mockus, Y. Kamei, B. Adams, A. E. Hassan, High-impact defects: a study of breakage and surprise defects, in: 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11, ACM, Szeged, Hungary, 2011, pp. 300–310. doi:10.1145/2025113.2025155.

[33] E. Kouroshfar, Studying the effect of co-change dispersion on software quality, in: ACM Student Research Competition, 35th International Conference on Software Engineering (ICSE), San Francisco, CA, 2013, pp. 1450–1452. doi:10.1109/ICSE.2013.6606741.

[34] N. Nagappan, T. Ball, Using software dependencies and churn metrics to predict field failures: An empirical case study, in: Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, 2007, pp. 364–373. doi:10.1109/ESEM.2007.13.

[35] P.-N. Tan, M. Steinbach, V. Kumar, Introduction to Data Mining, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[36] T. Zimmermann, N. Nagappan, Predicting Defects Using Network Analysis on Dependency Graphs, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, Leipzig, Germany, 2008, pp. 531–540. doi:10.1145/1368088.1368161.
URL http://doi.acm.org/10.1145/1368088.1368161

[37] C. Chen, A. Liaw, L. Breiman, Using random forest to learn imbalanced data, University of California, Berkeley.

[38] N. Nagappan, T. Ball, A. Zeller, Mining Metrics to Predict Component Failures, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, Shanghai, China, 2006, pp. 452–461. doi:10.1145/1134285.1134349.

[39] J. Garcia, I. Krka, C. Mattmann, N. Medvidovic, Obtaining ground-truth software architectures, in: Proceedings of the International Conference on Software Engineering, ICSE '13, IEEE Press, San Francisco, CA, USA, 2013, pp. 901–910.
URL http://dl.acm.org/citation.cfm?id=2486788.2486911

[40] T. Graves, A. Karr, J. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Transactions on Software Engineering 26 (7) (2000) 653–661. doi:10.1109/32.859533.

[41] A. Mockus, D. M. Weiss, Predicting risk of software changes, Bell Labs Technical Journal 5 (2) (2000) 169–180. doi:10.1002/bltj.2229.

[42] A. E. Hassan, Predicting faults using the complexity of code changes, in: 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Vancouver, Canada, 2009, pp. 78–88. doi:10.1109/ICSE.2009.5070510.

[43] M. D'Ambros, M. Lanza, R. Robbes, On the Relationship Between Change Coupling and Software Defects, in: 16th Working Conference on Reverse Engineering, Lille, France, 2009, pp. 135–144. doi:10.1109/WCRE.2009.19.

[44] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: 27th International Conference on Software Engineering, St. Louis, Missouri, 2005, pp. 284–292. doi:10.1109/ICSE.2005.1553571.

[45] S. Eick, T. Graves, A. Karr, J. Marron, A. Mockus, Does code decay? Assessing the evidence from change management data, IEEE Transactions on Software Engineering 27 (1) (2001) 1–12. doi:10.1109/32.895984.

[46] S. Eick, T. Graves, A. Karr, A. Mockus, P. Schuster, Visualizing software changes, IEEE Transactions on Software Engineering 28 (4) (2002) 396–412. doi:10.1109/TSE.2002.995435.

[47] D. Poshyvanyk, A. Marcus, R. Ferenc, T. Gyimthy, Using Information Retrieval Based Coupling Measures for Impact Analysis, Empirical Softw. Engg. 14 (1) (2009) 5–32. doi:10.1007/s10664-008-9088-2.

[48] F. Rahman, P. Devanbu, How, and Why, Process Metrics Are Better, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, San Francisco, CA, USA, 2013, pp. 432–441.

[49] T. Zimmermann, N. Nagappan, Predicting Subsystem Failures using Dependency Graph Complexities, in: The 18th IEEE International Symposium on Software Reliability, Trollhattan, Sweden, 2007, pp. 227–236. doi:10.1109/ISSRE.2007.19.

[50] R. C. Martin, M. Martin, Agile principles, patterns, and practices in C#, Prentice Hall, Upper Saddle River, NJ, 2007.

[51] E. Bouwers, A. van Deursen, J. Visser, Quantifying the Encapsulation of Implemented Software Architectures, in: 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), Victoria, BC, Canada, 2014, pp. 211–220. doi:10.1109/ICSME.2014.43.

[52] G. Murphy, D. Notkin, K. Sullivan, Software reflexion models: Bridging the gap between design and implementation, IEEE TSE 27 (4) (2001) 364–380.

[53] J. Brunet, R. A. Bittencourt, D. Serey, J. Figueiredo, On the evolutionary nature of architectural violations, in: Reverse Engineering (WCRE), 2012 19th Working Conference on, IEEE, 2012.

[54] J. Rosik, A. Le Gear, J. Buckley, M. A. Babar, D. Connolly, Assessing architectural drift in commercial software development: a case study, Software: Practice and Experience.

[55] S. Hassaine, Y. Guéhéneuc, S. Hamel, G. Antoniol, Advise: Architectural decay in software evolution, in: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE, 2012.

[56] J. van Gurp, S. Brinkkemper, J. Bosch, Design preservation over subsequent releases of a software product: a case study of baan erp, Journal of Software Maintenance and Evolution: Research and Practice.

[57] M. D'Ambros, H. Gall, M. Lanza, M. Pinzger, Analysing software repositories to understand software evolution, Springer, 2008.

[58] R. Mo, Y. Cai, R. Kazman, L. Xiao, Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells, in: 2015 12th

Working IEEE/IFIP Conference on Software Architecture (WICSA), 2015, pp. 51–60. doi:10.1109/WICSA.2015.12.

[59] E. Allen, T. Khoshgoftaar, Measuring coupling and cohesion: an information-theory approach, in: Software Metrics Symposium, 1999. Proceedings. Sixth International, 1999, pp. 119–127. doi:10.1109/METRIC.1999.809733.

[60] L. Briand, S. Morasca, V. Basili, Measuring and assessing maintainability at the end of high level design, in: Proceedings of the Conference on Software Maintenance, Montreal, Canada, 1993. doi:10.1109/ICSM.1993.366952.

[61] S. Sarkar, G. Rama, A. Kak, Api-based and information-theoretic metrics for measuring the quality of software modularization, Software Engineering, IEEE Transactions on 33 (1) (2007) 14–32. doi:10.1109/TSE.2007.256942.

[62] S. Sarkar, A. Kak, G. Rama, Metrics for measuring the quality of modularization of large-scale object-oriented software, Software Engineering, IEEE Transactions on 34 (5) (2008) 700–720. doi:10.1109/TSE.2008.43.

[63] C. Sant'Anna, E. Figueiredo, A. Garcia, C. Lucena, On the modularity of software architectures: A concern-driven measurement framework, in: F. Oquendo (Ed.), Software Architecture, Vol. 4758 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 207–224. doi:10.1007/978-3-540-75132-8_17.

[64] M. Wermelinger, Y. Yu, A. Lozano, A. Capiluppi, Assessing architectural evolution: a case study, Empirical Software Engineering.

[65] R. S. Sangwan, P. Vercellone-Smith, C. J. Neill, Use of a multidimensional approach to study the evolution of software complexity, Innovations in Systems and Software Engineering.

[66] T. Zimmermann, S. Diehl, A. Zeller, How history justifies system architecture (or not), in: Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of, IEEE, 2003.