Are Code Smell Detection Tools Suitable For Detecting Architecture Degradation?

Jörg Lenhard
Department of Mathematics and Computer Science
Karlstad University
651 88 Karlstad, Sweden
joerg.lenhard@kau.se

Mohammad Mahdi Hassan
Department of Computer Science
Al Qassim University
Al Malida, Buraydah Saudi Arabia
mo.hassan@qu.edu.sa

Martin Blom
Sebastian Herold
Department of Mathematics and Computer Science
Karlstad University
651 88 Karlstad, Sweden
firstname.lastname@kau.se

ABSTRACT

Context: Several studies suggest that there is a relation between code smells and architecture degradation. They claim that classes, which have degraded architecture-wise, can be detected on the basis of code smells, at least if these are manually identified in the source code.

Objective: To evaluate the suitability of contemporary code smell detection tools by combining different smell categories for finding classes that show symptoms of architecture degradation.

Method: A case study is performed in which architectural inconsistencies in an open source system are detected via reflexion modeling and code smell metrics are collected through several tools. Using data mining techniques, we investigate if it is possible to automatically and accurately classify classes connected to architectural inconsistencies based on the gathered code smell data.

Results: Results suggest that existing code smell detection techniques, as implemented in contemporary tools, are not sufficiently accurate for classifying whether a class contains architectural inconsistencies, even when combining categories of code smells.

Conclusion: It seems that current automated code smell detection techniques require fine-tuning for a specific system if they are to be used for finding classes with architectural inconsistencies. More research on architecture violation causes is needed to build more accurate detection techniques that work out-of-the-box.

CCS CONCEPTS

•Software and its engineering → Software architectures; Abstraction, modeling and modularity; Software maintenance tools;

KEYWORDS

architecture erosion, code smells, data mining, case study

ACM Reference format:
Jörg Lenhard, Mohammad Mahdi Hassan, Martin Blom, and Sebastian Herold. 2017. Are Code Smell Detection Tools Suitable For Detecting Architecture Degradation?. In *Proceedings of ECSA '17, Canterbury, United Kingdom, September 11–15, 2017,* 7 pages.
DOI: 10.1145/3129790.3129808

ECSA '17, Canterbury, United Kingdom

DOI: 10.1145/3129790.3129808

1 INTRODUCTION

When software is developed and maintained over a longer period of time, there is a chance that changes to the source code do not conform to the initially intended architecture [24]. This divergence from the intended architecture by the actual implementation is called *software architecture degradation* or *software architecture erosion* [6]. The phenomenon can be considered as problematic, since it may complicate maintenance and also lead to a reduction in software quality [14]. Hence, there is a need for approaches that help practitioners to mitigate the effects of architecture erosion.

One way of combating architectural erosion is to explicitly model the intended architecture of a software system and to regularly check if the source code is consistent with that model [6]. Unfortunately, this straight-forward approach faces obstacles in practice, since documentation or models of a system's intended architecture are often missing [7].

To address this common situation of absent architectural documentation, some research has targeted the question of how to deal with software architecture erosion if an explicit specification of the system's intended architecture is missing. Studies suggest that other symptoms of decaying software quality, in particular *code smells* [12], might be used to identify architecturally relevant code anomalies and, thus, to act as a surrogate for explicit architectural information [17, 18]. The basic assumption is that classes which contain many code smells, or particular types of code smells, are also those that tend to violate the intended architecture. Therefore, those classes should be prioritized for architectural repair.

This approach is promising, since today code smells are relatively well-known as a concept and there are many software suites and tools for automatic code smell detection [9]. If code smells are a sufficiently good indicator for architectural inconsistencies in classes, they could provide relatively cheap and easy-to-use support for architectural repair, even if architectural documentation is missing. So far, studies have found such a sufficient relation only between architectural inconsistencies and code smells that are identified manually [17, 18]. This is, however, problematic for the application scenario described here, as manually identifying code smells in a large code base might be a very laborious task. Instead, one might as well invest the effort into building an architectural model and

use this model directly to identify inconsistencies between source code and architecture. The benefit of using code smells as surrogate information might hence be very limited in terms of reduced effort.

Our goal in this work is *to investigate whether code smells that are detected automatically by contemporary tools without any fine-tuning can be used in combination to identify classes that contain architectural inconsistencies at a sufficient accuracy.* To this end, we perform a case study on an open source system for which we could develop and validate the intended architecture, which is needed as a ground truth about architectural inconsistencies. We identify architectural inconsistencies in the source code via the reflexion modeling approach [20]. Several code smell detection tools are applied to compute a variety of different code smell metrics for the investigated system. Thereafter, we try to see if a combination of these metrics can be used to classify classes as inconsistency-free or inconsistency-containing. We build these classification models using Naïve Bayes [25] and we perform an exhaustive search of the most accurate models, in terms of their precision and recall. The results suggest that even the most optimal models are currently not accurate enough to identify classes with architectural inconsistencies based on code smell data. Therefore, the challenge is to build better detection tools, a task which requires more research on the causes of architectural inconsistencies.

The remainder of the paper is structured as follows. In the next section, we discuss related work. Thereafter, in Sect. 3, we outline the theoretical foundations and in Sect. 4, we describe the software used in the case study, as well as our method for data collection and analysis. This is followed by a description of the findings in Sect. 5. We interpret the findings and outline threats to validity in Sect. 6. Finally, the paper is summarized and concluded in Sect. 7.

2 RELATED WORK

There are a number of studies that target the relation between architectural inconsistencies and code smells, or that use the same case study application as we do here.

When it comes to the application scenario of this work, most highly related are a number of studies by Macia et al. [17, 18]. In these studies, the authors analyzed code smells found in a number of systems and tried to relate these code smells to architectural problems. To this end, they fine-tuned detection strategies for a set of code smells and each specific system and tried to see if particular smells could be used for predicting architecture erosion. As an example, they tried to find if classes that they classified as "god class" also contain a given number of architectural inconsistencies. They found that the accuracy of using code smells for this task is generally low [18], but improves if code smells are identified by human intuition instead of tooling [17]. We take this work as a motivation for our study here, but we differ in several ways. First of all, we are not trying to use a particular type of code smell, such as "god class", for prediction. Instead, we aggregate code smells at a higher level into categories as they are provided by several widely-used tools, such as "all code smells related to code size". Moreover, we do not define our own notion of code smells and customize detection strategies to our case study system, but we rely on the notions and configurations available in said tools. What is more, we are testing the combination of different code smell categories to perform the classification. The idea is that the combination of this information could lead to an improved prediction. Lastly, the case study application we use here is substantially larger in terms of size than the applications used in [17, 18] that were publicly accessible. In another study [21] on largely the same systems, the authors also consider the combination of different code smells to find a relation to so-called design problems, such as "fat interface". This work is similar since we also combine code smells here, but the design problems in [21] are not strictly focused on software architecture.

Also Fontana et al. [10] build on the assumption that architecture erosion and code smells are related. Like [17, 18], they are considering specific smells such as "god class", but on top of that they investigate the relationships among smells, such as co-occurrence, in a large set of projects. Their motivational idea is that combinations of smells might allow for a better prioritization of classes for architectural repair. We follow the same line of reasoning, but try to combine smell categories, as they are present in tools, instead of combining specific smells. Moreover, Fontana et al. [10] do not relate smells to actual data on architectural inconsistencies.

Our case study application, JabRef, has been used in several other software engineering studies, some of which are also concerned with architecture. For instance, Constantinou et al. [5] use JabRef to evaluate an approach for architecture recovery on the basis of structural metrics. Here, we are not building on structural metrics, but on the higher-level notion of code smells. Moreover, we are not trying to develop an architecture from selected indicators, but rather to investigate the relation between such indicators and the actual intended architecture. A number of further studies use JabRef to evaluate specific metrics [8], quality models and quality assessment methodologies [16], or bug detection strategies [1]. However, these studies have no specific architectural focus.

3 FOUNDATIONS

The main foundations for this work are architectural inconsistencies as detected by the reflexion modeling approach, the concept of code smells, and the Naïve Bayes classification algorithm.

3.1 Architectural Inconsistencies and Reflexion Modeling

In this work, we refer to architectural inconsistencies as divergences between the architecture of a software system, as it is in intended by its principal architects, and the actual source code. To obtain data on these divergences, we apply the reflexion modeling approach [20]. We need this data as a "ground truth", so that we are able to see if we can predict classes with divergences based on code smell data.

Reflexion Modeling starts out with building the intended architectural model of a software system. This model essentially corresponds to a set of modules, i.e. partitions of the source code, and the dependencies that are envisaged between them. Next, the actual source code as it is implemented in the software is mapped onto the modules of the architecture. Thereafter, it is possible to compare the actual dependencies as they are implemented in the source code with the intended dependencies as they are specified in the architectural model. This typically uncovers dependencies where the source code is not consistent to the intended architecture.

Dependencies that exist in the source code, but that are not envisaged in the architectural model are usually viewed as problematic,

essentially as violations of the model. A classic example of such an unwanted dependency would be a class from the data layer of a system that directly accesses a class from the user interface layer. These are the dependencies that we consider as *architectural inconsistencies* in the remainder of the paper. Dependencies are always directional, i.e., they have a *source* (e.g. a class accessing another class by calling a method on an instance of this class) and a *target* (the class being accessed from a different context). A source code class can be the source and target of multiple inconsistencies at the same time.

Regardless of the direction of an architectural inconsistency that a class participates in, it might also be the *cause* of the inconsistency. For instance, if a class calls a method on another class that it should not access according to the architectural model, it is the cause of an inconsistency. In contrast, a class might also be misplaced in the wrong architectural model and therefore be accessed by other classes that need to do so, but which are not allowed by the architectural model. In this case, the misplaced target class is the cause of the inconsistency.

3.2 Code Smells and their Detection

Code smells [12] are anomalies in the source code that do not usually represent a factual error or fault, but instead represent a violation of coding standards that make the code harder to change or more error-prone. Oftentimes, they are considered as indicators for maintainability problems and also architecture erosion [14].

Typical examples are classes that are very large or complex (the "god class" smell), methods with very long parameter lists (the "long parameter list" smell), or classes that excessively use the methods of other classes (the "feature envy" smell). Although, the concept of code smells is relatively well-known in software engineering research and practice, there is no generally agreed-upon and clear-cut definition of what constitutes a smell. For example, there is no uniform definition of when a class is too large and complex. Nevertheless, there are many code quality and code smell detection tools that provide custom notions for smells and also many different smells. These tools apply detection strategies [19], i.e., logical constraints that are composed of software metrics and permitted metric thresholds, which allow to detect if a code smell is present in a given context. Examples are specific settings for the complexity value at which a class turns into a "god class".

Here, we rely on the facilities as they are implemented and configured out-of-the-box in several detection tools to compute code smell data. Since the goal is to apply the tools in their default configuration, we do not attempt to unify the varying notions that different tools have for code smells. Instead, we consider the different code smell categories that a tool provides. For instance, the PMD tool detects a variety of smells such as "Excessive Method Length" or "Excessive Parameter List" and sorts these smells into the *code size* category. Here, we count the amount of code smells present in a class based on their category[1]. That is, we do not base the analysis on, for instance, the amount of "Excessive Parameter List" smells in a class, but instead on the amount of code size smells in a class. That way, we combine different but related smells into a smell metric. Due to page restrictions and the huge amount of different smells that the tools consider, for instance PMD detects 13 different types of smells in the *code size* category alone for Java programs, we cannot present a mapping of smells to categories here, but have to refer the reader to the documentation of the tools.

[1]To do so, we parse the result files produced by the detection tools. These files essentially list smell occurrences by stating the class and line of code (if available), smell name, and smell category in comma-separated or XML format.

3.3 Data Mining and Naïve Bayes

In this work, we connect data on architectural inconsistencies and automatically-detected code smells and try to predict the former using the latter. We use data mining techniques, in particular the Naïve Bayes algorithm, for building and evaluating classification models that perform this prediction [25].

Naïve Bayes is a popular and simple classification algorithm that learns a classification scheme from the features of a set of observations. Each feature allows for a given set of values. The algorithm learns the classification scheme by considering the probabilities of values in the features of the observations. We are interested in classifying classes as to whether they participate in architectural inconsistencies, respectively whether they are the cause of architectural inconsistencies. Feature values correspond to the values of particular code smell metrics, i.e. in what frequency a certain category of code smells is present in a class. For example, one feature is the PMD code size smell metric and possible feature values for observations (classes) are based on the amount of smells in a class: *very low*, *low*, *medium*, *high*, and *very high*. Thus, the algorithm builds a classification model that identifies classes as inconsistency-containing if certain thresholds for code smells are present, and inconsistency-free otherwise. The amount of true-positives and false-positives, as well as true-negatives and false-negatives, can be used to compute the accuracy of a classifier in terms of its precision and recall.

Naïve Bayes assumes a normal distribution and an independence of the attributes used to build the classification scheme. These assumptions are often violated, but it has been found that the algorithm is quite robust regarding such violations and often outperforms more sophisticated classifiers nevertheless [25, 26]. Its robustness and simplicity are the reasons why we apply the algorithm here. For more details on Naïve Bayes, we refer to [25].

4 STUDY DESIGN

We perform a case study of a single open source system from which we derive quantitative data through the application of reflexion modeling and four different code smell detection tools. From the resulting data, we build Naïve Bayes classifiers for separating classes that participate in architectural inconsistencies from those that do not. The following subsections describe these steps in more detail.

4.1 Data Collection

We investigate a long-living open source software system for which we could create and validate a reflexion model. This software is the JabRef reference manager[2]. Jabref is a cross-platform reference manager for BibTeX and BibLaTeX bibliographies written in Java. Initiated in 2003, it consists of approximately 750 top-level classes with 80 000 physical lines of code and almost 100 developers contributed to its source code over time.

[2]The project homepage is located at http://www.jabref.org.

The data in this paper is based on JabRef 3.5, released on 2016-07-13. We built the reflexion model for this version in several video-conferencing sessions together with the current development team, of which the first author is a member. The intended architecture created in these session is now available in the project documentation[3]. The reflexion model was created and computation of architectural inconsistencies was done using the JITTAC tool [3]. This uncovered 186 classes participating in architectural inconsistencies with a number of 1459 inconsistencies in total.

Although the reflexion model informs on which classes participate in inconsistencies, it does not carry information on which of the classes are the actual *cause* of an inconsistency. To obtain data on causality, the first author investigated the code of every class participating in inconsistencies, no matter whether it is the source or target. The classes were then classified as to whether they are the cause of at least one architectural inconsistency or not. This classification was strengthened by comparing the source code of the version of JabRef used here, v3.5, with a subsequent release, v4.0-beta released on 2017-04-17. In the cases, where an inconsistency had been resolved in the newer version, it could be seen how the code had been changed. For instance, if a class was moved to another package between v3.5 and v4.0-beta, thereby resolving an architectural inconsistency, this class can be considered as misplaced in v3.5, being the original cause of the inconsistency. We identified a total of 90 classes as inconsistency causes this way.

Next to data on architectural violations and their causality, we applied a number of code quality tools to the source code. More precisely, we applied the code quality and code smell detection tools Findbugs [11], PMD[4], SonarQube [4], and Sonargraph[5]. Each of these tools provides a variety of metrics and indicators for problems in source code. From the resulting data, we selected a number of metrics that inform on code smells and aggregated this data at the level of a class. For Findbugs, we considered the amount of code smells in the categories of *style*, *bad practice*, and *scary*, which we also aggregated to a *total amount* of code smells per class. In the case of PMD, we calculated the amount of *design*, *code size*, *coupling* and *high priority* code smells in a class. SonarQube provides metrics for the number of *code smells* in a class, the number of *violations*, which, similar to its *code smells* metric refers to general purpose issues in the source code, as well as the number of *duplicated lines*, which corresponds to a particular type of code smell [12]. Furthermore, the tool provides metrics for the number of *bugs* and the number of security *vulnerabilities* in a class. Since these aspects are arguably more critical than code smells, we consider them as worth investigating here as well. On top of that, SonarQube computes a metric for *technical debt*, which corresponds to the amount of time needed to fix issues and code smells, as well as a *technical debt ratio*, which compares the effort required to remedy existing technical debt with the effort of rewriting the code. These two metrics seem interesting in our context, since they provide an aggregated and more abstract notion of code smells. From Sonargraph, we obtained a metric for the *instability* of a class, which detects classes as instable depending on their incoming and outgoing dependencies.

[3]See https://github.com/JabRef/jabref/wiki/High-Level-Documentation.
[4]The project page of PMD is available at https://pmd.github.io/.
[5]Sonargraph's project page is available at https://www.hello2morrow.com/products/sonargraph/explorer.

Needless to say, there is a high variation as to what the tools consider as code smells in classes for the different metrics. In all cases, we applied as little custom configuration as possible to the tools, since our goal is to see if we can find a relation between architectural inconsistencies and code smells that have been detected out-of-the-box and with little configuration effort.

4.2 Data Analysis

During the data analysis phase, we merged all code smell data and inconsistency data into a single data frame and normalized all variables to make them fit for building classification models.

Response variables were coded on a binary scale, indicating that a class either participates in architectural inconsistencies or not, and that it is the cause of inconsistencies or not. The predictor variables, i.e. all code smell metrics, were transformed to a five-point likert scale, ranging from one (zero or very low number of code smells) to five (very high number of code smells). We performed this transformation based on the quintiles of each particular variable. For example, all observations with a metric value lower than 20% of all observed values (the classes within the first quintile for a variable, i.e the classes with the least amount of code smells) were assigned a value of one on the likert scale. Similarly, all observations with a metric value higher than the 20% lowest of all observed values, but lower than 40% (the classes within the second quintile for a metric), were assigned a value of two on the likert scale, and so on. For some metrics, there were so few code smells detected in the data overall that the transformation led to a normalization of all observations to the same value in the likert scale. As an example, if more than 80% of all classes do not contain code smells of a certain category, all classes will have the same value on the likert scale. A variable where all observations are identical is not useful for building classification models, so we dropped it in the following. This was the case for all metrics from Findbugs, except for the total amount of code smells in a class, ignoring their specific category. Also for the bugs, security vulnerabilities, and duplicated lines metrics from SonarQube, there were too few observations for a meaningful classification. After these transformations, we arrived at a set of ten metrics coming from all four code smell detection tools as the input to building classification models.

For the ten remaining metrics, we computed all possible combinations of a given size, ranging from one to ten. For instance, we computed all combinations of two out of the ten metrics, of three out of the ten metrics, and so on, independent of the order of metrics. Each of the different combinations corresponds to a potential classification model for predicting that a class participates in architectural inconsistencies, or that it is the cause of such inconsistencies. Given the set of ten metrics, there are $2^{10} - 1 = 1023$ combinations in total (leaving out the trivial combination of zero metrics) that correspond to potential classification models. For every combination, we computed a Naïve Bayes classifier using R [22] and its data mining extension RWeka [15]. To evaluate the model, we used the same inital seed for all models, as well as ten-fold cross-validation.

From the set of all classification models, we are interested in the "best" models overall, as well as the "best" model for a given level, e.g., for models that consist of exactly two metrics. As indicated in

Sect. 3.3, model quality can be determined by *precision* and *recall*. Here, we are interested in classifying classes that participate in inconsistencies as such. Hence, precision is calculated based on whether the classes that are classified as inconsistency-related also actually participate in inconsistencies, i.e. the percentage of true positives in all positives. A high value for recall is based on whether a large part of all the classes that contain inconsistencies are also classified as such.

For our application scenario, using code smell metrics as a surrogate for architectural inconsistencies, it is important to take precision as well as recall into account. If we disregard one of the criteria, we risk to select classifiers that are very skewed. For instance, we might end up with a classifier that is 100% precise, because it only classifies a single class as inconsistency-related. In this case, most of the actual architecture erosion in a system goes unnoticed by the classifier. Similarly, when only considering recall, we might arrive at classifiers that rate all classes as inconsistency-related. To consider precision and recall equally, we used the F_1-score, F-score for short, which corresponds to the harmonic mean of the two variables [23]. The F-score is the metric, which we are trying to maximize when searching for optimal models. Ideally, a suitable classifier should work for a large majority of the data, i.e. at least two thirds of the classes that are identified as inconsistency-related should also fulfill this property and a majority of all classes containing inconsistencies should also be classified as such. Put differently, a threshold for a desirable F-score is 0.66 or higher.

4.3 Replication Package

A replication package for the study is available at https://github.com/lenhard/saerocon2017-replication. This package includes the data frame on which the computations in the paper are based, as well as the R code used to perform the computations.

5 FINDINGS

In the following, we describe our findings for models that predict whether a class participates in architectural inconsistencies in Sect. 5.1. In Sect. 5.2, we report our findings for models that predict whether a class is the cause of an inconsistency. All values are rounded to two decimal places.

5.1 Classifying Classes Participating in Inconsistencies

Tab. 1 lists the optimal classification models with regard to their F-score for classifying classes that participate in architectural inconsistencies for different levels of n.

There is no proper classifier (F-score equals zero) with only one metric. In this case, all models classify all classes as inconsistency-free. This results in precision and recall values of zero.

Overall, the classifiers for rating whether a class participates in architectural inconsistencies are the most accurate ones we have found. Classifiers that use eight or nine metrics respectively have the highest F-score of 0.38. However, classifiers using five, six, or seven metrics are very close, with an F-score of 0.37. The maximum precision in all classifiers is at the level of five or six metrics with 0.5. The same precision value is almost reached for the optimal classifier using three metrics, but this model results in a lower recall. The maximum value for recall of 0.35 is reached with the optimal classifier using nine of the metrics.

When considering the metrics that are part of the optimal classifiers, it can be seen that high priority smells reported by PMD are part of every optimal classifier. The same almost applies to the code smells metric reported by SonarQube, which is only missing in one optimal model. In contrast, smells reported by Findbugs are only part of the single classifier that uses all metrics. Also the remaining metrics regarding technical debt by SonarQube or instability by SonarGraph are only part of around half of the classifiers, and mostly of those with a high amount of metrics.

Next to the results presented in Tab. 1, we also computed classifiers for rating that a class is the *source* of inconsistencies or the *target* of inconsistencies. We refrain from reporting detailed results for reasons of page space. When it comes to classifiers that rate whether a class is the source of inconsistencies, the results largely mirror the ones depicted in Tab. 1, although at a lower level of accuracy. The F-score is highest with a value of 0.35 for models with eight or nine metrics. Precision is highest with 0.37 for classifiers with three metrics and recall reaches its maximum of 0.38 for classifiers with eight or more metrics. When it comes to classification models that predict whether a class is the target of architectural inconsistencies, there is no single model, regardless of the amount of metrics used, that performs a meaningful classification. In all cases, all classes are classified as not being the target of an inconsistency.

Lastly, we also tested the computation of classifiers optimized for their precision, instead of their F-score, to find the most precise classifiers. We can confirm that more precise classifiers do exist, most notably at the level of four or five metrics. Here, the optimal classifiers are 100% precise, because the amount of classes they rate as inconsistency-related is very low, i.e. one class for the classifier using four metrics and three classes for the one using five.

5.2 Classifying Classes Causing Inconsistencies

The optimal classification models for predicting whether classes are the cause of an architectural inconsistency are shown in Tab. 2.

It can be seen in the table that classifiers for rating classes as the cause of architectural inconsistencies are less accurate overall than the ones described in Sect. 5.1. Furthermore, at least five metrics are required to build a useful classifier and classifiers using a lower amount of metrics rate all classes as inconsistency-free.

The classifiers with the highest F-score of 0.09 use seven, eight, or nine metrics respectively. These are also the classifiers with the highest recall of 0.07. Precision is highest for the optimal classifier involving six metrics with a value of 0.16.

When looking at the metrics that are part of the optimal classifiers, it can be seen that the amount of high priority and code size smells by PMD, as well as code smells and violations by SonarQube are part of every classifier. In constrast, the technical debt ratio reported by SonarQube and the instability reported by SonarGraph are the least frequent part of classifiers.

Also for the classifiers that rate classes as causes of architectural inconsistencies, we tested an optimization for precise models. Again, the results are similar to the ones reported in Sect. 5.1, but at a lower level. There is a single classifier using six metrics with a precision value of one. As before, this classifier rates a single class of the actual 90 as inconsistency-related.

Table 1: Optimal Models for Predicting Classes that Participate in Architectural Inconsistencies

N	PMD priority	PMD design	PMD coupling	PMD code size	SonarQube code smells	SonarQube violations	SonarQube technical debt	SonarQube technical debt ratio	SonarGraph instability	Findbugs smells	F-Score	Recall	Precision
2	X			X							0.27	0.21	0.38
3	X				X		X				0.33	0.25	0.49
4	X	X			X	X					0.35	0.30	0.43
5	X	X	X		X			X			0.37	0.29	0.50
6	X	X			X	X		X	X		0.37	0.30	0.50
7	X	X			X	X	X	X	X		0.37	0.32	0.46
8	X		X	X	X	X	X	X	X		0.38	0.34	0.42
9	X	X	X	X	X	X	X	X	X		0.38	0.35	0.40
10	X	X	X	X	X	X	X	X	X	X	0.36	0.33	0.39

Table 2: Optimal Models for Predicting Classes that Cause Architectural Inconsistencies

N	PMD priority	PMD design	PMD coupling	PMD code size	SonarQube code smells	SonarQube violations	SonarQube technical debt	SonarQube technical debt ratio	SonarGraph instability	Findbugs smells	F-Score	Recall	Precision
5	X			X	X	X	X				0.02	0.01	0.14
6	X		X	X	X	X	X				0.08	0.06	0.16
7	X	X	X	X	X	X				X	0.09	0.07	0.15
8	X	X	X	X	X	X	X			X	0.09	0.07	0.14
9	X	X	X	X	X	X	X		X	X	0.09	0.07	0.13
10	X	X	X	X	X	X	X	X	X	X	0.07	0.04	0.12

6 DISCUSSION

In the following, we interpret the results in Sect. 6.1 and discuss threats to validity in Sect. 6.2.

6.1 Interpretation of the Results

Overall, the suitability of code smell metrics for classifying classes with architectural inconsistencies seems debatable, no matter if the goal is to rate whether a class is the cause of an inconsistency or whether it just participates in one. As the exhaustive search of all possible classifiers shows, there is no single classifier that correctly classifies a majority of the inconsistency-related classes and meets the criteria we set out in Sect. 4.2. Practically all of the classifiers rate most of the classes as inconsistency-free. Also when ignoring a balance between precision and recall and optimizing for precision, resulting classifiers are hardly suitable since they ignore nearly all classes with architecture-related problems. This finding indicates that there is no sufficient relation between code smells as they are computed by contemporary tooling and architectural inconsistencies. Although there are studies that have found a relation between code smells and architectural inconsistencies [17], it seems that this relation does not hold if code smells are identified by a tool instead of a human. Whereas prior studies [18] made this assertion for particular code smells, such as god class, in isolation, we are able to add the following: Even if automatically-detected code smells are aggregated to smell categories, such as size-related smells, and multiple categories are combined with each other, it is not possible to classify classes as inconsistency-related or inconsistency-causing with a suitable accuracy. To generalize this statement: *It seems that it is not possible to use contemporary code smell detection tools out-of-the-box to aid in the task of architectural repair.* For practice, this means that instead of investing effort into the fine-tuning of tools or manual detection of smells, it seems more sensible to invest the effort into building a reflexion model and to obtain information about architectural inconsistencies directly. This can be interpreted as a challenge to the research community to build better detection tools and respectively to provide more insights into the actual causes of architectural inconsistencies as a prerequisite to building better tools. A lack of empirical research with regard to this challenge is also confirmed by a prior mapping study [13].

When looking at the optimal classifiers, two code smell metrics stick out. These are the amount of high priority smells as reported by PMD and the amount of code smells as reported by SonarQube. It seems that among the metrics we tested, these are the most appropriate for a classification. What is more, also other metrics that represent rather basic smell counts, i.e. all metrics reported by PMD and violations reported by Sonarqube, seem to perform better than newer and more specialized notions, such as the ones related to technical debt. It seems that there is only little relation between technical debt measures, as computed by the tools we used, and architectural inconsistencies. Architectural problems are common forms of technical debt, and therefore a stronger relation could have been expected. More work on metrics regarding concepts such as technical debt is needed to produce more accurate measurements.

Lastly, some smell metrics, in particular the ones reported by Findbugs, perform relatively bad when compared to others, such as those reported by PMD. This is likely the case because of the granularity in which code smells are identified. In general, PMD is much more verbose than Findbugs, reporting smells in numbers that are an order of magnitude higher than for Findbugs. This results in a stronger deviation of smell counts for different classes, which makes it easier for a data mining algorithm to use the metrics for building a classification model. Thus, our suggestion for improving detection tools for the task at hand is to make them verbose.

6.2 Threats to Validity

Our discussion of threats to validity distinguishes between external, construct, and internal validity, as well as reliability, according to the categorization by Brewer and Crano [2].

External validity concerns the generalizabilty of the findings to a larger population. Our results build upon a case study of a single system, which is clearly a threat to generalizability. Classifiers and code smell counts can be different for different systems, especially

if they differ significantly in size, if they are written in a different programming language, and possibly also if they come from a closed source context instead of the open source context of our case study application. Furthermore, there is a risk of overfitting the classifiers to the current system when using too many variables. Essentially, this means that the classifiers with more variables presented here might be less generalizable. Further studies on different systems are needed to complement the presented results.

Threats to construct validity concern the degree to which the measurements made relate to the phenomenon under study and there are several such threats here. Firstly, data on architectural inconsistencies is computed based on an architectural model. Errors in this model are a threat to construct validity. Since we built the model together with the current development team, we are confident that the amount of such errors is limited. A next threat lies in the code quality tools we applied to obtain code smell data. Our objective is to test the suitability of these tools, but we cannot guarantee that their notion of code smells is ultimately valid. Moreover, the statistical methods that we applied might be a source of threats to construct validity. This refers to the Naïve Bayes algorithm we used to build the classifiers. We cannot prove that the independence assumption of the algorithm holds for all the variables that we used here. Nevertheless, Naïve Bayes has been shown to be robust even when used with dependent variables, as discussed in Sect. 3.3.

Threats related to internal validity concern cause-effect relationships. Since this study is not about observing a cause-effect relationship, but about the relation between two factors, such threats are not applicable here.

Reliability concerns the reproducability of the results when performing the measurements and the computations multiple times. To improve reliability here, the artifacts and code produced by one author were independently cross-checked by the others. Moreover, the data and the scripts for computing the results are made publicly available as a replication package to make it easier for other researchers to double-check our results.

7 CONCLUSION

In this work, we performed a case study using one open source system with the goal to see if code smells that are computed automatically by code quality tools can be combined to rate whether classes participate in architectural inconsistencies. We built and validated a reflexion model for the case study system to obtain data on architectural inconsistencies. Thereafter, we applied data mining techniques using Naïve Bayes to build classifiers and performed an exhaustive search of all possible classifiers for the most accurate results. Since the accuracy even of the most optimal classifiers is low, it seems that code smells as detected by contemporary tooling are not suitable for rating classes as inconsistency-related.

The results presented here demonstrate that more sophisticated detection mechanisms for architectural inconsistencies are needed. Practitioners could benefit from tooling that helps them to identify architecturally-problematic classes without requiring too much manual effort. To make such tools a reality, it seems that more research on causes of architectural inconsistencies is required. To this end, we are currently working on a classification scheme for categorizing the causes of architectural inconsistencies. This might enable us to build more accurate prediction techniques for certain types of architecture violation causes.

REFERENCES

[1] Nasir Ali, Aminata Sabané, Yann-Gaël Guèhèneuc, and Guilian Antoniol. 2012. Improving Bug Location Using Binary Class Relationships. In *12th International Working Conference on Source Code Analysis and Manipulation*.

[2] M. B. Brewer and W. D. Crano. 2014. Research Design and Issues of Validity. *Handbook of Research Methods in Social and Personality Psychology, Second Edition* (2014), 11–26.

[3] J. Buckley, S. Mooney, J. Rosik, and N. Ali. 2013. JITTAC: A Just-In-Time Tool for Architectural Consistency. In *Proceedings of the 35th International Conference on Software Engineering*. San Francisco, USA.

[4] G. Ann Campbell and Patroklos P. Papapetrou. 2013. *SonarQube in Action*. Manning Publications Co.

[5] Eleni Constantinou, George Kakarontzas, and Ioannis Stamelos. 2011. Open Source Software: How Can Design Metrics Facilitate Architecture Recovery?. In *4th Workshop on Intelligent Techniques in Software Engineering*.

[6] Lakshitha de Silva and Dharini Balasubramaniam. 2012. Controlling Software Architecture Erosion: A Survey. *J. Syst. Softw.* 85, 1 (Jan. 2012), 132–151.

[7] W. Ding, P. Liang, A. Tang, H. v. Vliet, and M. Shahin. 2014. How Do Open Source Communities Document Software Architecture: An Exploratory Survey. In *2014 19th International Conference on Engineering of Complex Computer Systems*.

[8] Andi Wahju Rahardjo Emanuel, Retantyo Wardoyo, Jazi Eko Istiyanto, and Khabib Mustofa. 2011. Modularity Index Metrics for Java-Based Open Source Software Projects. *International Journal of Advanced Computer Science and Applications* 2, 11 (2011).

[9] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11, 2 (2012), 5–1.

[10] Francesca Arcelli Fontana, Vincenzo Ferme, and Marco Zanoni. 2015. Towards assessing software architecture quality by exploiting code smell relations. In *3nd International Workshop on Software Architectures and Metrics*. Florence, Italy.

[11] Jeffrey Foster, Michael Hicks, and William Pugh. 2007. Improving Software Quality With Static Analysis. In *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 83–84.

[12] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] Sebastian Herold, Martin Blom, and Jim Buckley. 2016. Evidence in Architecture Degradation and Consistency Checking Research - Preliminary Results from a Literature Review. In *10th European Conference on Software Architecture Workshops*. Copenhagen, Denmark.

[14] Lorin Hochstein and Mikael Lindvall. 2005. Combating architectural degeneration: a survey. *Information and Software Technology* 47, 10 (2005), 643–656.

[15] Kurt Hornik, Christian Buchta, and Achim Zeileis. 2009. Open-Source Machine Learning: R Meets Weka. *Computational Statistics* 24, 2 (2009), 225–232.

[16] Klaus Lochmann, Jasmin Ramadani, and Stefan Wagner. 2013. Are comprehensive quality models necessary for evaluating software quality?. In *9th International Conference on Predictive Models in Software Engineering*. Baltimore, USA.

[17] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In *16th European Conference on Software Maintenance and Reengineering*.

[18] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. 2012. Are Automatically-detected Code Anomalies Relevant to Architectural Modularity?: An Exploratory Analysis of Evolving Systems. In *11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. ACM, 167–178. DOI: http://dx.doi.org/10.1145/2162049.2162069

[19] R. Marinescu. 2004. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 350–359. DOI: http://dx.doi.org/10.1109/ICSM.2004.1357820

[20] G. C. Murphy, D. Notkin, and K. J. Sullivan. 2001. Software reflexion models: bridging the gap between design and implementation. *IEEE Trans Software Eng* 27, 4 (2001), 364–380. DOI: http://dx.doi.org/10.1109/32.917525

[21] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code Anomalies Flock Together. In *38th IEEE International Conference on Software Engineering*. Austin, TX, USA.

[22] R Core Team. 2017. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

[23] C. J. Van Rijsbergen. 1979. *Information Retrival* (2nd ed.). Butterworth.

[24] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing.

[25] Ian H. Witten, Eibe Frank, and Mark A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques* (3rd ed.). Morgan Kaufmann, San Francisco.

[26] Harry Zhang. 2004. The Optimality of Naive Bayes. In *17th International Florida Artificial Intelligence Research Society Conference*. Miami Beach, FL, USA.