Software Qual J (2019) 27:241–274
https://doi.org/10.1007/s11219-018-9404-z

# Exploring the suitability of source code metrics for indicating architectural inconsistencies

Jörg Lenhard[1] · Martin Blom[1] · Sebastian Herold[1]

Published online: 8 March 2018

**Abstract** Software architecture degradation is a phenomenon that frequently occurs during software evolution. Source code anomalies are one of the several aspects that potentially contribute to software architecture degradation. Many techniques for automating the detection of such anomalies are based on source code metrics. It is, however, unclear how accurate these techniques are in identifying the architecturally relevant anomalies in a system. The objective of this paper is to shed light on the extent to which source code metrics on their own can be used to characterize classes contributing to software architecture degradation. We performed a multi-case study on three open-source systems for each of which we gathered the intended architecture and data for 49 different source code metrics taken from seven different code quality tools. This data was analyzed to explore the links between architectural inconsistencies, as detected by applying reflexion modeling, and metric values indicating potential design problems at the implementation level. The results show that there does not seem to be a direct correlation between metrics and architectural inconsistencies. For many metrics, however, classes more problematic as indicated by their metric value seem significantly more likely to contribute to inconsistencies than less problematic classes. In particular, the fan-in, a classes' public API, and method counts seem to be suitable indicators. The fan-in metric seems to be a particularly interesting indicator, as class size does not seem to have a confounding effect on this metric. This finding may be useful for focusing code restructuring efforts on architecturally relevant metrics in case the intended architecture is not explicitly specified and to further improve architecture recovery and consistency checking tool support.

✉ Jörg Lenhard
joerg.lenhard@kau.se

Martin Blom
martin.blom@kau.se

Sebastian Herold
sebastian.herold@kau.se

[1] Department of Mathematics and Computer Science, Karlstad University, Karlstad, Sweden

**Keywords** Architectural inconsistencies · Software architecture degradation · Reflexion modeling · Source code metrics · Code anomalies · Multi-case study

## 1 Introduction

Literature refers to the phenomenon of software architecture degradation as the process of continuous divergence between the prescriptive, "as intended" software architecture and the descriptive, "as implemented" software architecture of a system (Perry and Wolf 1992; Taylor et al. 2009). Many commonly accepted definitions of software architecture state that the prescriptive software architecture of a system manifests the intended principal design decisions made to address the desired quality attributes of the software system (Bachmann et al. 2005; Taylor et al. 2009; Rozanski and Woods 2005; Sommerville 2010). Since software architecture degradation leads to increasing inconsistency between a system's implementation and the intended principal design decisions manifested in the prescriptive architecture, one can assume that software systems suffering from this phenomenon are less likely to follow and implement those principal design decisions and the desired quality attributes connected to them. For example, diverging from the desired module structure and the allowed dependencies described in an intended software architecture may lead to undesired and unforeseen dependencies that decrease the maintainability and evolvability of a system.

Indeed, several studies discuss cases in which systems suffer from decreased maintainability due to degradation. Godfrey and Lee describe the case of the Netscape web browser in which the development of a new product version was scrapped due to a degraded code base used as starting point of the development (Godfrey and Lee 2000). Other cases report on complex, time-consuming, and most likely very expensive restructuring efforts due to degradation (van Gurp and Bosch 2002; Sarkar et al. 2009; Eick et al. 2001; Dalgarno 2009). Although reliable means for quantifying the cost that degradation may cause are missing (Herold et al. 2016), these studies suggest that software architecture degradation can have severe impacts on software systems and can cause enormous problems for the organization developing them.

In addition to these studies, further research shows that instances of degraded systems exist in industrial and open-source software development practice and across different sectors and application domains, e.g., van Gurp et al. (2005), Deiters et al. (2009), Herold and Rausch (2013), Buckley et al. (2015), and Brunet et al. (2015). Buckley et al., for example, performed an in vivo multi-case study with five systems from the financial sector, developed by four different organizations, and found symptoms of degradation in each of them (Buckley et al. 2015). Deiters et al. identified a high degree of degradation in a system of a national administration agency (Deiters et al. 2009). Brunet et al. observed the Eclipse project over a period of five years and documented significant software architecture degradation as well (Brunet et al. 2015). Even though a statement about the general state and degree of software architecture degradation in the software industry may not be possible based on the quantity of empirical research (Herold et al. 2016), these and other case studies provide strong evidence that the phenomenon is prevalent in practice.

One strategy to mitigate software architecture degradation is to check for those inconsistencies between the prescriptive architecture of a software system and its implementation that become visible when a software system degrades architecturally (de Silva and Balasubramaniam 2012). However, to do so, an up-to-date specification of the prescriptive software architecture is required. In practice, many systems lack an explicit and sufficiently detailed description of the prescriptive architecture. A study by Ding et al., in which the authors investigated how software architectures were documented in ca. 2000 open-source projects, reported that only about 5% of the projects contain architecture documentation at all (Ding et al. 2014). This means that architectural specifications usually need to be created from scratch. This task can require substantial effort for existing systems of realistic size, with hundreds of thousands of lines of code. Moreover, the original developers of a system, the people with tacit knowledge of the originally intended architecture of a system, are often no longer available. Thus, there are cases where the creation of an architectural specification is simply infeasible.

In these situations of missing architectural documentation, code is often the single most important source of information about potential infringements of the desired, yet not explicitly formulated architectural structures. The occurrence of code anomalies, also known as code smells (Fowler 1999), is often considered a phenomenon in source code that can lead to software architecture degradation if tolerated for too long (Hochstein and Lindvall 2005).

Assuming such a relationship between source code anomalies and software architecture degradation symptoms exists, automating the detection of architecturally relevant anomalies could reduce the effort of counteracting degradation. Only few studies have investigated this relationship and the suitability of automated approaches to source code anomaly detection techniques for mitigating software architecture degradation (Macia et al. 2012a, b). They suggest that a lack of accuracy of automated techniques might be problematic.

Many source code anomaly/code smell detection techniques are based on source code quality metrics. The studies mentioned above, for example, apply detection strategies (Marinescu 2004). These strategies consist of logical constraints composed of metrics and threshold values. For example, in order to detect god classes—classes that do too much work themselves instead of relying on the functionality of others—such a constraint could specify that a class is likely to be a god class if it scores high in a metric for the class' complexity, low in a metric for its cohesion, and high in a metric measuring its usage of other classes' attributes (for details, see (Lanza et al. 2005)). Detection strategies provide a binary decision in a concrete source code context on whether a code smell is present or not. However, it might be possible that source code metrics on their own are suitable for discriminating classes that are more likely to contribute to architecture degradation than others or to estimate the likelihood to which a class may be involved in architecture degradation. To the best of our knowledge, this aspect has not yet been investigated.

This article focuses specifically on inconsistencies between intended architecture and source code as detected as symptoms of architectural degradation in reflexion modeling, which is one approach to software architecture recovery and consistency checking (Murphy et al. 2001). In this approach, the intended software architecture of a system is modeled in terms of modules, expected/allowed dependencies between modules, and a mapping between modules and units of the source code. Deviations from the intended architecture and the contained allowed dependencies can then be automatically detected by analyzing the source code (see Section 2). A source code dependency may be classified as architecturally disallowed if it introduces undesired dependencies between architectural modules.

The motivating question for this study is to which extent source code elements being sources or targets of such dependencies can be identified through source code metrics provided by metric suites and other software development infrastructure tools that are commonly used in software development.

Using such tools for this task has the potential to increase efficiency not only when detecting software architecture degradation-related classes while appropriate architecture documentation is absent; it could also foster and improve approaches for the analysis of architecture violation causes in which source code metrics are used to classify architectural inconsistencies in order to effectively repair software architecture degradation (Mair et al. 2014; Herold et al. 2015).

In order to address the motivating question, we performed a multi-case study on three open-source systems. Two of these systems and their validated architectural models have been taken from the replication package of a different study (Brunet et al. 2012). In case of the third system, the intended architecture has been developed and validated as part of this work during a number of video-conferencing sessions together with the current development team. For each system, the intended software architecture was captured using the JITTAC tool, which implements the reflexion modeling approach (Buckley et al. 2013). Next, we computed 49 different source code metrics provided by six different publicly available metric suites and retrieved information from the version control systems used for the studied systems. The data was statistically analyzed to investigate the relationship between architectural inconsistencies and the gathered source code metrics.

In the next section, in Section 2, we outline the reflexion modeling approach used here and introduce fundamental terminology. Following this, we explain the study design in detail in Section 3. In Section 4, the statistical analyses performed and results obtained are described. Our interpretation of the results and potential validity issues are discussed in Section 5. Thereafter, we discuss related work in Section 6. The paper is concluded in Section 7 with a summary and a sketch of potential future work.

## 2 Foundations

This section provides a short overview of software architecture consistency checking and will introduce the key concepts of reflexion modeling. The reflexion modeling tool used in the study is introduced as well.

### 2.1 Software architecture consistency checking

- *Conformance by construction*: These approaches attempt through automatic or semi-automatic generation of lower level artifacts, such as source code, to make sure that these artifacts are consistent with the higher level architecture description. A prominent example of conformance by construction is model-driven development, e.g., Mattsson et al. (2012).
- *Conformance by extraction*: These approaches extract information from artifacts of the implementation process, such as source code dependencies, and compare this information with a specification of an intended architecture based on specified mappings or rules.

The approach used in this paper falls into the category of conformance by extraction. Further categorizations distinguish approaches to conformance by extraction according to whether they leverage information from static (code) analysis, from dynamic analysis, such as executions traces, or both (Knodel and Popescu 2007). Passos et al. provide an overview of three static approaches to conformance by extraction, each of which is a typical representative of a group of techniques (Passos et al. 2010):

- *Dependency matrices*: These approaches capture source code dependencies in a matrix of source code elements complemented by architectural rules that constrain possible values in the matrix and hence the allowed dependencies (Sangal et al. 2005).

- *Source code query language (SCQL)-based approaches*: In these approaches, SCQLs are used to specify constraints that the architecture imposes on the source code and which need to be followed in order to consider the source code architecturally consistent (de Moor et al. 2008; Herold and Rausch 2013).
- *Reflexion modeling-type approaches*: These approaches focus on the creation of graphical, box-and-lines models that define intended architectures in terms of modules and allowed dependencies (Murphy et al. 2001).

The authors of that overview see the main advantage of reflexion modeling-type approaches in the focus on creating an explicit model of the intended architecture that becomes a central element in the consistency checking process. Even if there exist SCQL-based approaches in which the constraints specified are attached to a model capturing the intended architecture, the majority of mature tools adopted in practice falls into the category of reflexion modeling-type techniques, e.g., Hello2morrow (2017), Structure101 (2017), Duszynski et al. (2009), Raza et al. (2006), and Buckley et al. (2013). These techniques have also undergone intensive empirical evaluation, e.g., Knodel et al. (2008),Rosik et al. (2011), Ali et al.(2012, 2017), Brunet et al. (2015), and Buckley et al. (2015).

In this study, we focus on *reflexion modeling* for three reasons. Firstly, as stated, the technique is widely used compared to other techniques and is well supported by tools. Secondly, it is quite a lean technique and users can learn how to use it quickly (e.g., Buckley et al. 2015) which was important in the case of recovering the architecture with the users unfamiliar with this type of tools (see the JabRef case, Section 3.2). Thirdly, even if the expressiveness is comparably low, reflexion modeling covers a basic yet important concern of software architecture which is of particular relevance for this study: the definition of desired structures and dependencies of a code base. In the following, reflexion modeling will be explained in more detail.

## 2.2 Reflexion modeling foundations and terminology

Architectural inconsistencies, as understood in this work, are essentially divergences between the intended architecture of a software system and the structures that are actually implemented in source code.

Reflexion modeling starts with manually constructing the intended architecture in terms of architecture modules and dependencies between them. Modules essentially represent logical partitions of the source code. By explicitly specifying dependencies between modules, the architect expresses which dependencies are allowed. In a second step, the actual implementation elements, e.g., classes and packages of the software system, are manually mapped to the architectural modules specified in the intended architecture. In a third step, the intended architecture and the source code are automatically analyzed, i.e., the allowed dependencies between the architectural modules are compared with the actual dependencies in the source code and investigated for inconsistencies. The results are visualized in the so-called reflexion model.

Three types of dependencies may exist in reflexion models:

1. *Convergences* are dependencies that have been modeled as allowed/expected in the intended architecture and which are also manifested as source code dependencies in the code base.
2. *Absences* are dependencies that have been modeled as allowed/expected in the intended architecture but which are not manifested as source code dependencies in the code base.

3. *Divergences* are dependencies between modules, manifested as source code dependencies, which have not been modeled as allowed/expected by the architect.

In the following, we call a source code dependency a *divergent dependency* if it contributes to a divergence in a reflexion model, i.e., if it connects two implementation elements that are mapped to architectural modules in a way that it would not be considered allowed or expected in the intended architecture.

Divergent dependencies and absences mark cases where the source code is not consistent with the architectural model. However, as absences are not directly manifested in source code (but rather in the non-existence of source code dependencies), we call only divergent dependencies *architectural inconsistencies* in the remainder of this paper. The exploration of the relation between these inconsistencies and source code metrics is the goal of this article.

Architectural inconsistencies are always directional. A class (or an interface) in an object-oriented code base is *contributing* to an architectural inconsistency if it is either the *source* or the *target* of the corresponding source code dependency. It is the source of the inconsistency if it depends upon a class that it should not access according to the intended architecture, for instance through an import statement, a method call, or an instance creation. A class is the target of an inconsistency if it is referenced by another class in the system, but the dependency is disallowed by the architectural model. Although it may seem that the sources of inconsistencies are also their cause, this does not hold in general. Even the target of an inconsistency can be its cause, if for instance the target class is misplaced in the wrong module of the code. This is why we consider it important to investigate the sources and targets of architectural inconsistencies alike.

## 2.3 The reflexion modeling tool used—JITTAC

As stated in Section 1, the tool used to model the intended architecture and analyze the implementing source code is JITTAC (Buckley et al. 2013) which implements the reflexion modeling approach.[1] JITTAC is implemented as a plug-in for the Eclipse IDE. The software architect can create a model of the intended architecture as a box-and-lines diagram using a drawing pane. Source code units, such as packages or classes, can be mapped to the modules of the intended architecture by dragging and dropping them to the boxes representing the modules of the system. The reflexion model, which is visualized on the same drawing pane as the corresponding architectural model, shows convergences, divergences, and absences as described above. The software architect/developer can investigate these elements in more detail by inspecting a table of source code dependencies contributing to a convergence or divergence selected in the reflexion model. Moreover, JITTAC is able to export architectural inconsistencies as defined above as a comma-separated list for further data analysis.

JITTAC implements "just-in-time" reflexion modeling functionality. This means that changes in the source code of a system, in the model of its intended architecture, or in the mapping between the two are immediately propagated and visualized. For example, as soon as a developer adds new code containing a call to a method that the current code he is working on is not allowed to invoke according to the intended architecture, divergences in the architectural model will be updated and the corresponding code will be marked in the editor. While this "just-in-time" functionality is not crucial for the presented study, it has

[1] See also http://actool.sourceforge.net/.

been found useful in cases of interactively recovering software architectures (Buckley et al. 2015), a scenario similar to the JabRef case in this study (see Section 3.2).

JITTAC is currently able to apply reflexion modeling for systems written in Java. In order to extract the dependencies required for reflexion modeling, the tool analyzes Java source code for constructs representing dependencies between classifiers, such as usage of types, import of packages/classes, call of methods and constructors, and inheritance. To do so, it makes use of the code parsing functionality provided by the Eclipse Java Development Tools. They allow JITTAC to hook into the compile and build process of Eclipse and to populate JITTAC's internal model of source code dependencies during the build of a system.

## 3 Study design

The study protocol consisted of four basic steps, following guidelines for conducting case study research in software engineering (Runeson and Höst 2009). Firstly, we scoped the study and formulated hypotheses. Secondly, we selected the cases, the sample of systems to investigate, and created architectural models following the reflexion modeling approach based on available or newly created architecture specifications. Thirdly, we collected quantitative source code metric data from six different metric suites and the version control system. Fourthly, we performed statistical analyses that inform on the relation between architectural inconsistencies and source code metrics. The following subsections describe each of these steps in more detail.

### 3.1 Research questions

The research questions we are addressing in this paper concern the relationship that source code metrics, as reported by common software quality tools, have to architectural inconsistencies and if metrics can be used for discriminating between classes that participate in inconsistencies and those that do not.

Our starting point is to see whether metric values increase or decrease with architectural inconsistencies. For example, we try to see if the amount of architectural inconsistencies increases with the size of a class. As such, our first research question (RQ1) is: *Do metric values increase or decrease with the amount of architectural inconsistencies that a class participates in?*

Even if metric values increase or decrease consistently together with the amount of architectural inconsistencies a class participates in, it is not clear if they can meaningfully discriminate such classes from classes that do not participate in inconsistencies. This leads to our second research question (RQ2): *Can metrics be used to discriminate between classes that participate in architectural inconsistencies and those that do not?*

Lastly, a confounding effect of class size has been found for many source code metrics (Emam et al. 2001). Given that there are metrics which can be used to discriminate between classes that participate in inconsistencies and those that do not, it is not clear whether this might be due to a size bias.

### 3.2 Case selection

For this work, we chose to investigate three different large size open-source systems. The systems were chosen, because (i) they are sufficiently large, consisting of several tens of thousands of lines of code, (ii) they have been under active development for several years,

**Table 1** Descriptive data for investigated systems at source code and architecture level

| | JabRef | Lucene | Ant |
|---|---|---|---|
| Source code statistics | | | |
| No. of classes | 736 | 507 | 774 |
| LOC | 82,783 | 60,298 | 92,140 |
| No. of inconsistencies | 1459 | 638 | 377 |
| No. of classes with inconsistencies | 186 | 115 | 80 |
| No. of classes with outgoing inconsistencies | 119 | 75 | 38 |
| No. of classes with incoming inconsistencies | 86 | 49 | 42 |
| Architecture statistics | | | |
| No. of modules | 6 | 7 | 15 |
| No. of allowed dependencies | 15 | 16 | 77 |
| No. of divergences | 12 | 15 | 14 |

which means that there is enough time for architecture degradation to occur, (iii) they are actively used in practice and not just toy systems or proof-of-concept prototypes, and (iv) we were able to obtain a specification of their intended architecture or could create and validate such a specification ourselves. The last point is clearly the most limiting factor. We investigated a number of further systems, e.g., from (Macia et al. 2012a; Brunet et al. 2012), but discarded them from our study, because they were either too small in terms of system size or an application of the reflexion modeling approach did not uncover architectural inconsistencies. Apart from these requirements, the source code of the systems is also available, as required for performing reflexion modeling and for applying some of the metric suites we used. The projects we selected are three widely known and frequently used systems: the *JabRef reference manager*,[2] the *Apache Lucene search engine library*,[3] and the *Apache Ant build system*.[4] Table 1 shows descriptive data for the three systems and also the number of classes with architectural inconsistencies. Classes can be the source of outgoing inconsistencies and the target of incoming inconsistencies at the same time and in the case of JabRef and Lucene, we found such an overlap. To compute architectural inconsistencies, one author modeled the intended architectures using the JITTAC tool described in Section 2. The remaining authors cross-checked these models independently to ensure their validity.

We need to emphasize that the creation of architectural models with JITTAC can only be done if some form of architectural specification is available. The creation of this initial specification is a daunting task that requires intricate knowledge of a system and needs to be informed by its architects. If those architects are no longer available, it may well be impossible to create valid architectural models. This is why we are exploring the application of software metrics as a substitute to architectural specifications in this article.

JabRef is an open-source and cross-platform reference management tool written in Java that is widely used in the academic community. Its development started in 2003 and today

[2]The project home page can be found at http://www.jabref.org/.

[3]Lucene's home page is located at https://lucene.apache.org/.

[4]The home page of Ant can be found at http://ant.apache.org/.

JabRef consists of more than 80,000 lines of code (LOC)[5] and almost 750 classes.[6] The version used in this paper is version 3.5, released on 13 July 2016. We modeled the intended architecture of JabRef and the mapping between this architecture and the classes in the source code in a number of video-conferencing sessions together with the current development team, of which the first author is a member. During these sessions, JITTAC was used to manually model JabRef's intended architecture and the mapping to source code; both were adapted and refined based on the development team's comments until the resulting model reflected the intended architecture and the mapping to source code was correct.

The architecture developed during these sessions is available in the project documentation.[7] Based on this architecture, the source code of JabRef exhibits 1459 architectural inconsistencies, distributed over 186 classes.

Apache Lucene is an open-source and high-performance search engine library implemented in Java and hosted by the Apache foundation. It is used in many projects worldwide for searching large-scale data, one example being Twitter. Lucene is an Apache project since 2001. Its core search engine, which is the system in focus here, consists of more than 500 classes with more than 60,000 LOC. We modeled the architecture of Lucene using JITTAC based on an article by Brunet et al. (Brunet et al. 2012) which describes Lucene's architecture in terms of modules and allowed dependencies between modules as well as a mapping of modules to the source code of the system. One of the authors manually created a JITTAC model according to this description which was cross-checked for correctness and soundness by the two other authors. The model revealed 638 inconsistencies, distributed over 115 classes, in Lucene.

Apache Ant is a foundational build tool for the Java ecosystem and more recent build tools rely on it. As for Lucene, we use the latest version documented by Brunet et al. (Brunet et al. 2012), which describes Ant's intended architecture and the mapping to source code. This version is dated from 14 October 2007 and consists of more than 750 classes with roughly 92,000 LOC. The JITTAC model and the required mapping were created by one of the authors again and cross-checked by the co-authors. The analysis through JITTAC uncovered 377 architectural inconsistencies distributed over 80 classes.

All models describing the intended architecture as well as architectural inconsistencies and the mappings between the architecture and source code of the systems are included in the replication package of this study (see Section 3.5). A few quantitative figures of the architectural models are summarized in Table 1.

## 3.3 Data collection—gathered source code metrics

As stated before, our goal is to investigate if architecturally problematic classes can be identified through source code metrics provided by metric suites and other software development infrastructure tools that are commonly used in software development. Today, a wide range of tools for the quantification of structural source code properties and the detection of source code anomalies is available. Whereas some of these tools are limited to a single programming language, others are multi-language. Proprietary and free systems do exist.

[5]For reporting LOC here, we rely on the measurements made by SonarQube, which computes actual source code lines, excluding lines that consist of whitespace characters or comments only.

[6]All reported class counts are top-level classes only, i.e., without nested or inner classes defined in the scope of another class.

[7]This documentation can be found at https://github.com/JabRef/jabref/wiki/High-Level-Documentation.

Here, we rely on tooling for the Java ecosystem, since this is where the selected case study systems belong to. We exclusively use free tooling, which is common for the open-source projects we are looking at. In fact, some of the tools we use here are actively used by these projects.

By applying a set of tools, we try to obtain data for a wide range of different structural properties and try to avoid a single-instrument bias. Even most basic source code metrics, such as LOC, can be operationalized in different ways, for instance including comment lines or excluding them. By comparing the results that different tools provide for the same concept, we can mitigate this effect to a certain degree. For instance, if we find that there is a relation between LOC and architectural inconsistencies for multiple different ways of computing LOC, this can be seen as evidence for an actual relation between inconsistencies and the concept underlying LOC. In total, we investigated 49 source code metrics and anomaly counts, computed using seven tools, which are briefly described in the following paragraphs.

The complete list of metrics and their categorization can be found in Table 2. We categorized the metrics according to the type of concept they are trying to measure, e.g., size or complexity, or, if present, used the categorization of the metric suite they are part of. The metrics cover aspects of size, complexity, coupling, cohesion, technical debt, code churn,[8] age, and anomalies. We tried to include all metrics from the tools if possible for all of the systems. For example, in the case of CKJM, all metrics that the tools provide could be collected from the code. In contrast, in the case of SonarQube, we did not use all metrics. In particular, metrics regarding unit tests and coverage, or metrics that could not be measured based on a single point in time, were excluded. When it comes to PMD, FindBugs, and git, the metrics we consider are not available directly. Instead, we computed them from the raw data that the tools produced using automated scripts, e.g., we built a script that counts the amount of issues of a certain category in a single class and reports the total number for that class. We focused on categories that are related to structural aspects and excluded several categories that were focused on unrelated aspects, for example on localization issues.

FindBugs is one of the first and probably the most well-known anomaly detection tool for Java (Foster et al. 2007). It detects a variety of different types of anomalies based on custom thresholds and heuristics. For this study, we used the Eclipse plug-in of FindBugs and configured it to report all anomalies found, regardless of their type or severity. We then captured the *total amount* of anomalies in a class (i.e., the numeric value thereof), as well as the amount of anomalies in the *style*, *bad practice*, and *scary* categories separately. These categories encompass the vast majority of all issues reported in the systems, but exclude a number of very specific issues, such as localization issues, that we did not investigate further. We rely on the categorization of anomalies as it is provided by FindBugs, because this categorization has been constructed over a long period of time as a consensus of the reviews of many contributors. We expect this to be more valid than a custom categorization that is built by a single team of authors. For a more detailed description of the issues that fall into the respective categories, we refer the interested reader to the documentation of FindBugs (Foster et al. 2007).

Similar to FindBugs, PMD[9] is an issue detection library that reports different categories of anomalies. As before, we used the Eclipse plug-in for PMD to analyze the projects and aggregated the following categories of issues for every class: the *total number* of issues, the number of *coupling* issues, the number of *design* issues, the number of *code size* issues,

[8]Code churn intends to capture the amount of change in a system (Hall and Munson 2000).

[9]The project page is available at https://pmd.github.io/.

**Table 2** Overview of tools and metrics used

| Tool | Metric | Type |
|---|---|---|
| PMD | Total number of issues | Anomaly count |
| | Number of coupling issues | Anomaly count |
| | Number of design issues | Anomaly count |
| | Number of code size issues | Anomaly count |
| | Number of high-priority issues | Anomaly count |
| FindBugs | Total number of issues | Anomaly count |
| | Number of style issues | Anomaly count |
| | Number of bad practice issues | Anomaly count |
| | Number of scary issues | Anomaly count |
| SourceMonitor | Lines of code | Size |
| | Number of statements | Size |
| | Number of method calls | Size |
| | Number of classes and interfaces | Size |
| | Methods per class | Size |
| | Average method size | Size |
| | Percentage of comments | Size |
| | Branch percentage | Complexity |
| | Maximum method complexity | Complexity |
| | Average block depth | Complexity |
| | Average complexity | Complexity |
| CKJM | Weighted methods per class | Complexity |
| | Number of public methods | Size |
| | Response for a class | Coupling |
| | Depth of inheritance tree | Inheritance coupling |
| | Number of children | Inheritance Coupling |
| | Fan-out | Coupling |
| | Fan-in | Coupling |
| | Lack of method cohesion | Cohesion |
| SonarQube | Lines of code | Size |
| | Number of statements | Size |
| | Public API | Size |
| | Number of functions | Size |
| | Public undocumented API | Size |
| | Number of comment lines | Size |
| | Comment line density | Size |
| | Cyclomatic complexity | Complexity |
| | Cyclo. complexity excl. inner classes | Complexity |
| | Number of duplicated lines | Anomaly count |
| | Number of issues | Anomaly count |
| | Number of code smells | Anomaly count |
| | Number of bugs | Anomaly count |

**Table 2** (continued)

| Tool | Metric | Type |
|---|---|---|
| | Number of security vulnerabilities | Anomaly count |
| | Sqale index | Technical debt |
| | Sqale debt ratio | Technical debt |
| VizzAnalyzer | Message Passing Coupling | Coupling |
| | Data Abstraction Coupling | Coupling |
| | Locality of Data | Coupling |
| Git | Number of commits | Code churn |
| | Timestamp of creation | Age |

and the number of *high-priority* issues. As for FindBugs, we excluded issue categories that were very specific, such as J2EE issues, migration issues, or logging issues. PMD provides a much more fine-grained level of reporting and the amount of issues is usually an order of magnitude higher than for FindBugs. Again, we rely on the categorization of the PMD development team, because they are specialists with regard to the issues that PMD detects. A closer description of the categories can be found on the web pages of PMD, which are linked above.

SourceMonitor[10] is a stand-alone metric suite that computes size and complexity metrics and we used all metrics that the tool reported on. It provides us with data on a number of structural properties, such as the number of *lines*, *statements*, *classes and interfaces*, *methods per class*, and aggregated measures, such as *average method size* or the *percentage of comments*. The complexity measures it computes are the *branch percentage*, *average complexity*, *maximum method complexity*, and *average block depth* in a class.

SonarQube is a sophisticated quality assurance suite that features many source code metrics, ranging from anomaly counts, similar to FindBugs or PMD, to traditional size and complexity metrics (Campbell and Papapetrou 2013). For this study, we set up an analysis server and performed the analysis with the help of SonarQube's command line tooling. The set of metrics that we investigate is only a subset of what SonarQube offers and we excluded a number of metrics that were not applicable here. The final set includes size metrics, i.e., LOC, the number of *statements*, *functions*, *comment lines*, the *public API* (which includes all public members, i.e., methods and attributes, in a class), and the *public undocumented API*. Moreover, SonarQube computes complexity metrics, referring to the *cyclomatic complexity of a class* with and without taking inner classes into account and different anomaly counts, namely, *bugs*, *code smells*, *code duplications*, *violations*, and *security vulnerabilities*. In contrast to FindBugs or PMD, SonarQube provides numeric metrics for these aspects out of the box and as before we rely on the anomaly categorization provided by the development team. Finally, SonarQube provides measures for technical debt, called *sqale index* and *sqale debt ratio*, as defined by the SQALE method (Letouzey 2012).

CKJM (Spinellis 2005) is a command line tool that builds on the well-known Chidamber and Kemmerer metric suite for object-oriented design (Chidamber and Kemerer 1994) and also reports on a few additional metrics, such as the fan-in. We used all metrics reported by the tool. It computes a size metric, *the number of public methods*, and the complexity metric *weighted methods per class* (the sum of the complexities of all methods in a class,

[10]SourceMonitor's project page can be found at http://www.campwoodsw.com/sourcemonitor.html.

defaulting to a value of one per method). Moreover, it reports on coupling, with the *response for a class* metric (the transitive closure of the call graph of all methods of the class), the *coupling between object classes* (often called CBO or fan-out). In this paper, we opt for the latter), *afferent couplings* (fan-in) and on inheritance coupling, with the *depth of the inheritance tree* and the *number of children*. Lastly, CKJM also covers cohesion, with the *lack of method cohesion* metric.

VizzAnalyzer is a an academic code quality tool that has been used in several research reports (Ericsson et al. 2012). Our primary reason for using this tool is that it provides a number of more advanced coupling metrics. In particular, these are *message passing coupling* (the number of method calls in a class to the methods of other classes), *data abstraction coupling* (Briand et al. 1999) (the number of abstract data types defined in a class), and the *measure of aggregation metric* (the ratio of the amount of attributes local to a class to the total amount of attributes) (Bansiya and Davis 2002), which is called *locality of data* in VizzAnalyzer. Support for such metrics is lacking in the other tools. The development of the tool has been discontinued, but it is still available as an Eclipse plug-in that is called VizzMaintenance.

Finally, we read data from the git version control system which is used in all three projects (Spinellis 2012). Though git is not a metric suite, it contains extensive information on class history. We extracted the *number of commits* that changed a class, as a notion of change rate, and the *timestamp of a class' creation*, as a notion of class age. Such metrics can be used to analyze code churn. Code churn has been linked to maintainability problems in general (Faragó et al. 2015) and, recently, also to architectural violations in the same sense as here (Olsson et al. 2017).

## 3.4 Data analysis

The protocol during data analysis follows the three research questions, outlined in Section 3.1. Essentially, we are evaluating a number of metric validity criteria as defined by IEEE Std 1061-1998 (R2009) (IEEE 1998), which refer to the metrics' correlation, consistency, discriminative power, and reliability. In this work, we evaluate if source code metrics consistently correlate with architectural inconsistencies and if they can be used to discriminate between classes that participate in inconsistencies and those that do not.

Our first step is to investigate the correlation between inconsistencies and metric values. This also provides insights on consistency of the metrics, e.g., if classes with higher metric values also participate in higher numbers of inconsistencies. As a second step, we try to see which metrics can be used for categorizing classes, i.e., we try to find if certain metric thresholds can be used for discriminating classes that do contain inconsistencies from those that do not. This refers to their discriminative power. Third, we take the short list of metrics resulting from the second step and investigate if these metrics correlate with size to see if their discriminative power might be a side effect of a size bias. To be worth reporting, the relation between a metric and architectural inconsistencies should be reproducible for all three systems. For instance, to speak of a strong correlation between a metric and architectural inconsistencies, a strong correlation should have been shown in all of the three cases. This refers to the reliability of the relationship.

The purpose of a correlation analysis is to see if the number of inconsistencies that a class contributes to increases with the value of a metric, e.g., if the number of architectural inconsistencies increases with class complexity. To this end, we compute the Spearman rank correlation coefficient (Daniel 1990) for the three systems for all combinations of metrics and the number of inconsistencies that a class is source, target, or any type of endpoint for. Spearman's rank correlation was favored over the Pearson product-moment correlation because normality of the variables' distribution, as required for significance testing over the Pearson correlation coefficient, could not be ensured. Visual inspection of some of the plots showed that the data contains outliers and that the distribution is skewed because the number of classes contributing to violations is low compared to the total number. Moreover, we performed a Shapiro-Wilk test on the distributions of inconsistencies in the data which let us refute the assumption of a normal distribution at the highest possible level of significance[11] (Shapiro and Wilk 1965). In such cases, the use of the Spearman rank correlation coefficient instead of the Pearson product-moment correlation seems more appropriate. Additionally, the rank correlation coefficient is also the recommended technique for evaluating consistency according to IEEE Std 1061-1998 (R2009) (IEEE 1998). From the resulting correlation data, we filter out all correlation values that belong to the interval of $[-0.3; 0.3]$. We consider values in this interval too low to indicate a meaningful correlation.

Next, we are trying to see whether source code metrics can discriminate classes that are prone to architectural inconsistencies. This objective does not necessarily require a correlation, although such a relation would certainly be helpful. Instead, it would be sufficient if we could find a threshold value for a metric that separates classes that are more likely to contribute to inconsistencies (regardless of the absolute number of inconsistencies in the class) from classes that are less likely to do so. However, metric values, and hence potentially thresholds, are typically very specific to a particular system and setting a custom value manually would reduce the generalizability of the results. To mitigate this problem, we opted for using distribution-based thresholds instead of a concrete absolute value (Fontana et al. 2015). We test different percentiles of the distribution of metric values as thresholds, namely the 50th, the 75th, and 90th percentile.[12] For instance, using the 75th percentile as threshold separates classes that have a metric value higher than 75% of the remaining classes from the rest. We analyze if, and to which extent, classes above these different thresholds have a significantly higher chance of contributing to architectural inconsistencies than classes below the thresholds. If this is the case for a metric for all three systems and for all three threshold levels, the metric can be seen as a practical indicator for inconsistencies that has shown its applicability in several contexts.

We use Fisher's exact test to see if classes above a certain threshold are significantly more likely to contain architectural inconsistencies (Upton 1992). To apply this test, we sort the classes of a system into a $2\times2$ contingency table, depending on whether they contain architectural inconsistencies and on whether the computed metric value is below or above the percentile selected as threshold value. The test then reports the exact $p$ value, as well as the odds ratio that informs on the likelihood of inconsistencies in the different groups. In this setting, the odds ratio represents the factor by which the likelihood of contributing to architectural inconsistencies is multiplied for classes with metric values greater than the selected threshold compared with classes with values below the threshold.

Fisher's exact test is usually applied as an alternative to the $\chi^2$ test, the test that IEEE Std 1061-1998 (R2009) (IEEE 1998) recommends for evaluating discriminative power. It is

[11] In all cases, the resulting $p$ values were as close to zero as the precision of the statistical software would allow.

[12] Note that we do not apply approaches for outlier elimination, such as outlier fences based on the interquartile range. In our case, outliers do not form incorrectly measured data, but instead classes with very high or low metric values. Hence, it is not acceptable to simply drop outliers. Nevertheless, by considering the 90th percentile, we are specifically considering outliers separated from the remaining data.

more suitable in case of very small samples or if the distribution of variables is very skewed. Even though we have sufficiently large samples for a $\chi^2$ test, the distribution is very skewed since most of the classes for all systems do not contribute to architectural inconsistencies. Hence, Fisher's exact test seems more appropriate in our context.

Last, we investigate the relationship between promising metrics for discrimination and metrics for size. The reason for this is that the metrics considered here are not normalized regarding the size of the class that they measure. Nevertheless, a higher metric value for a metric not directly related to size, such as fan-in, might still be influenced by how large a class is. To determine if such a size bias exists, we compute the Spearman rank correlation coefficient between selected metrics and size metrics in the same fashion as described above.

### 3.5 Replication package

A replication package for the study is available at https://github.com/lenhard/arch-metrics-replication. This package includes the models of the intended architecture we used, the metric data gathered, and the code written in R to compute the results presented in the following section (R Core Team 2016).

## 4 Results

In this section, we present the results of our statistical analysis on the relation between architectural inconsistencies and the aforementioned source code metrics. The correlation analysis is described in the next subsection. Thereafter, the results of the categorical analysis on discriminative power are shown in Section 4.2 and the results of the analysis of the confounding effect of class size are presented in Section 4.3.

### 4.1 Correlation analysis

We tested the correlation between all metrics described in Section 3.3 and any type of architectural inconsistency, i.e., source, target, or total. There are no moderate or strong correlations between architectural inconsistencies and source code metrics in the data that apply to all three systems at the same time. Nevertheless, all correlation coefficients reported below are significant at the level of 0.05.

In the case of Ant, there is no single metric for which the Spearman rank correlation coefficient is outside of the interval of $[-0.3; 0.3]$, regardless of the type of architectural inconsistency that is tested.

In the case of JabRef, there is a moderate correlation between the total amount of inconsistencies in a class and the number of public methods as measured by CKJM ($\rho = 0.32$). This also applies to the number of functions as measured by SonarQube ($\rho = 0.30$). For the number of times a class is a target of an architectural inconsistency, there is a moderate correlation to fan-in ($\rho = 0.33$).

When it comes to Lucene, several moderate correlations can be observed for all types of inconsistencies. The number of source inconsistencies in a class is moderately correlated with the public API (i.e., the number of publicly accessible methods and attributes, $\rho = 0.31$), with the weighted method count ($\rho = 0.31$), and with the number of public methods as calculated by CKJM ($\rho = 0.31$). The public API metric is also moderately correlated with the total amount of architectural inconsistencies ($\rho = 0.35$). Finally, the number of times a class is the target of an inconsistency is moderately correlated with fan-in ($\rho = 0.34$). The fan-in is the only metric for which a moderate correlation to architectural inconsistencies exists in more than one system.

## 4.2 Categorical analysis

For the categorical analysis using Fisher's exact test, as described in Section 3.4, we look at the results for the different types of inconsistencies separately. That is, we report on classes being sources of inconsistencies in Section 4.2.1, on classes being targets of inconsistencies in Section 4.2.2, and on classes contributing to inconsistencies ignoring their specific end-point role, i.e., source or target, in Section 4.2.3. More specifically, we discuss metrics that were found to be significant for all three observed systems and for all three threshold levels.

### *4.2.1 Classes being sources of architectural inconsistencies*

In this case, several metrics significantly discriminate between classes that are sources of inconsistencies and those that are not, for all three systems and for all the thresholds. In all of these cases, classes that have a metric value above the threshold are significantly more likely to contain architectural inconsistencies. The metrics mainly belong to the group of size metrics, but include also cohesion and complexity metrics, as well as anomaly counts.

Figure 1 shows the odds ratios from the Fisher tests for metrics where $p$ values were below the significance level of 0.05 for all three systems and all thresholds. We do not list the concrete $p$ values to keep the figure more readable, but the concrete $p$ values, odds ratios, and confidence intervals of the metrics presented in the figure are listed in the Appendix. In case multiple metrics for similar concepts were significant, for instance if different variants of computing LOC from different tools were significant, we limit the presentation to a single one of them. Furthermore, we opted for a graphical instead of a tabular presentation of the odds ratios since we think that this enhances understandability, given the amount of data points to depict.

A metric that could serve as a good indicator should have high values for all thresholds. Also a monotonic increase or decrease is positive in this regard, because it indicates that the metric is consistent when it comes to discriminating between classes (IEEE 1998). In addition to the visualization, we mainly report mean odds ratios over all systems and thresholds in the following paragraphs. $\overline{OR}_{src}(metric)$, $\overline{OR}_{trg}(metric)$, and $\overline{OR}_{any}(metric)$ refer to the mean odds ratios of a metric w.r.t. sources, targets, or endpoints of dependencies of any type contributing to architectural inconsistencies.

As discussed in Section 3.4, the odds ratio represents the factor by which the likelihood of contributing to architectural inconsistencies is multiplied for classes with metric values greater than the selected threshold compared with classes with values below the threshold. For Lucene, for example, classes that have a larger public API than at least 50% of the classes in the system (50th percentile, medium red line in bottom-left chart in Fig. 1) are 5.41 times as likely to contain architectural inconsistencies as classes with a smaller public API. The contingency table for this case is given in Table 3. The odds ratio, $OR$, can be computed from the contingency table as follows: $OR = (a/b)/(c/d) = (56/152)/(19/280) \approx 5.41$. Moreover, Fig. 1 shows that this likelihood increases with the metric value, i.e., classes whose public API value belongs to the top 25% (75th percentile) are 5.53 times more likely to be the source of inconsistencies, which increases to a factor of 6.57 for classes that are above the 90th percentile for this metric.

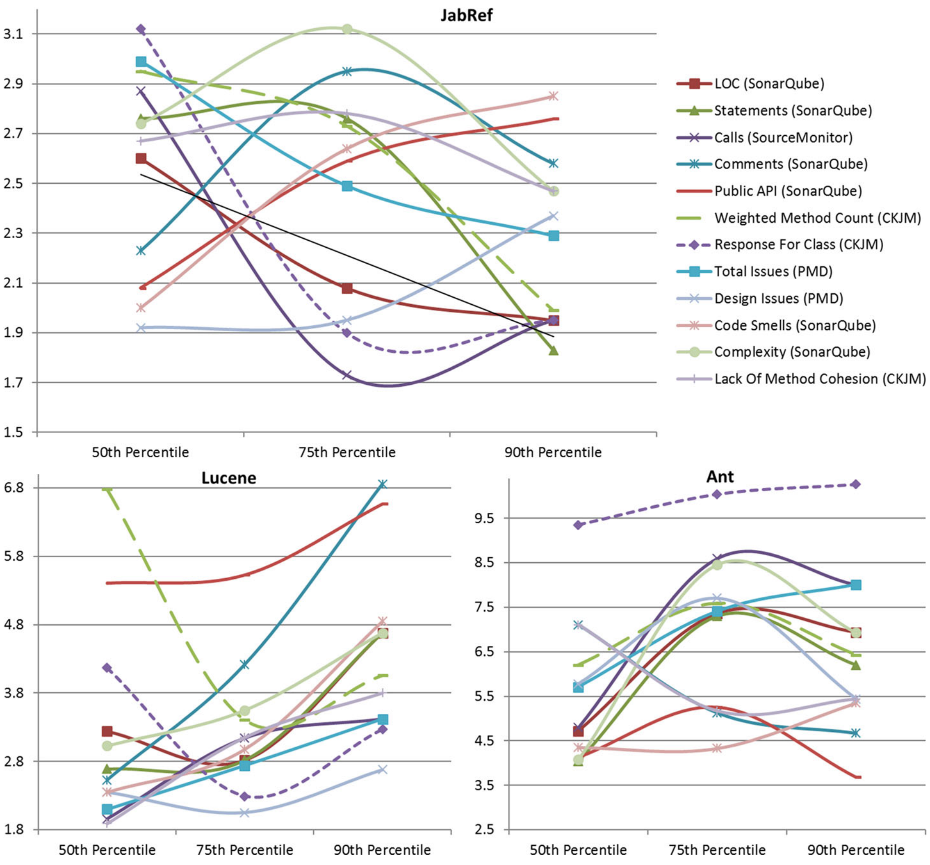


**Fig. 1** Line charts showing Fisher test odds ratios (*y*-axis) for sources of inconsistencies for different metric thresholds (*x*-axis)

When it comes to size metrics, line counts and method counts reach the significance level most frequently. To begin with, LOC as computed by SonarQube (excluding whitespaces and comments, $\overline{OR}_{src}(LOC_{SQ}) = 4.04$), reaches the significance level in all cases. The same applies to the number of statements as calculated by SonarQube (excluding class attributes, $\overline{OR}_{src}(NOS_{SQ}) = 3.90$), but not to the number of statements as calculated by SourceMonitor (including class attributes). Moreover, different versions of method counts, such as the weighted method count ($\overline{OR}_{src}(WMC) = 4.68$) as calculated by CKJM and the public API metric by SonarQube (public methods and attributes, $\overline{OR}_{src}(PublicAPI) = 4.22$), reach the significance level. Also the absolute number of comment lines ($\overline{OR}_{src}(Comments) = 4.25$) and the number of method calls in a class ($\overline{OR}_{src}(Calls) = 4.05$) seem to be significantly related to architectural inconsistencies. Next, several anomaly counts significantly discriminate between architecturally consistent and inconsistent classes. This includes the total number of PMD issues ($\overline{OR}_{src}(TotalIssues_{PMD}) = 4.13$) and the number of PMD design issues ($\overline{OR}_{src}(DesignIssues_{PMD}) = 3.58$) in a class, as well as code smells ($\overline{OR}_{src}(CodeSmells) = 3.52$) as reported by SonarQube. Also the cyclomatic complexity ($\overline{OR}_{src}(Complexity) = 4.34$) and the response for class metric ($\overline{OR}_{src}(RFC) = 5.15$)

**Table 3** Contingency table for the public API of Lucene at a threshold of 50%

| | $Classes_{Inconsistencies}$ | $Classes_{NoInconsistencies}$ |
|---|---|---|
| $PublicAPI > 50\%$ | a: 56 | b: 152 |
| $PublicAPI \leq 50\%$ | c: 19 | d: 280 |

reach the significance level in all cases. Finally, a cohesion metric, lack of method cohesion ($\overline{OR}_{src}(LCOM) = 3.83$), also shows a significant relation.

For the concrete values of the odds ratios presented in Fig. 1, there is no consistent picture for the different systems and thresholds. In case of JabRef, there is no particular trend with odds ratios rising or falling for different threshold values. For Lucene, odds ratios are highest in general at a 90th percentile threshold. For Ant, odds ratios go up at the 75th percentile threshold for most metrics, but drop again at the 90th percentile.

### 4.2.2 *Classes being targets of architectural inconsistencies*

When it comes to classes being targets of architectural inconsistencies, the number of cases that reach the significance level is much smaller. Of the 49 metrics we investigated, only in two cases $p$ values were significant for all the three systems and all the thresholds. These are two metrics that also stood out with regard to correlation as discussed in Section 4.1, the public API and fan-in.

The odds ratios in the cases where all tests resulted in $p$ values less than 0.05 are depicted in Table 4. As before, the complete test results can be found in the Appendix. Odds ratios increase consistently for the public API metric ($\overline{OR}_{trg}(PublicAPI) = 3.32$) with the thresholds for all three systems. The same cannot be said for the fan-in metric ($\overline{OR}_{trg}(FanIn) = 8.40$), but ratios are comparably high. The odds ratios for fan-in are the highest for all observations.

Two further metrics reach the significance level for all three systems for a threshold at the 75th percentile, but not for the other thresholds, and are therefore not included in Table 4. Firstly, this is the number of public methods as computed by CKJM, which is similar to the public API metric. The second one is the cohesion metric, lack of method cohesion.

### 4.2.3 *Classes being sources or targets of architectural inconsistencies*

When we ignore the specific role that a class can play w.r.t an architectural inconsistency, i.e., we do not separate sources from targets, the metrics which are significant are largely identical to the metrics that have been significant for the other two categories. The odds ratios for this case are displayed in Fig. 2, which can be read similar to Fig. 1.

**Table 4** Fisher's test odds ratios for targets of inconsistencies

| Metric | JabRef | | | Lucene | | | Ant | | |
|---|---|---|---|---|---|---|---|---|---|
| Percentile | 50th | 75th | 90th | 50th | 75th | 90th | 50th | 75th | 90th |
| Public API | 3.29 | 3.90 | 5.65 | 2.06 | 3.06 | 3.60 | 2.11 | 2.48 | 3.69 |
| Fan-in | 8.02 | 5.36 | 10.02 | 12.32 | 10.80 | 13.44 | 7.75 | 3.61 | 4.31 |

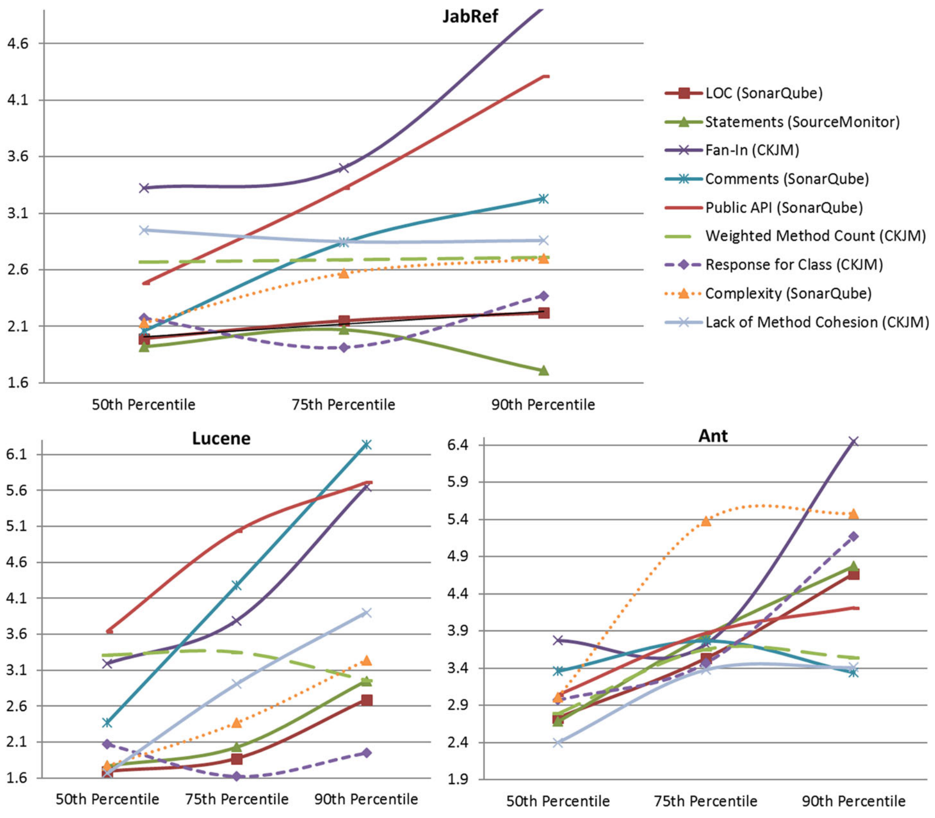


**Fig. 2** Line charts showing Fisher's test odds ratios (*y*-axis) for sources and targets of inconsistencies for different metric thresholds (*x*-axis)

All in all, the observations are comparable to the ones in Section 4.2.1: LOC as measured by SonarQube ($\overline{OR}_{any}(LOC_{SQ}) = 2.62$), the weighted method count by CKJM ($\overline{OR}_{any}(WeightedMethodCount) = 3.07$), the response for class metric ($\overline{OR}_{any}(RFC) = 2.63$), the cyclomatic complexity ($\overline{OR}_{any}(Complexity) = 3.18$), and the lack of method cohesion ($\overline{OR}_{any}(Complexity) = 2.93$) are significant. Moreover, the metrics that were found to be significant in the case of targets of inconsistencies, public API ($\overline{OR}_{any}(PublicAPI) = 3.96$) and fan-in ($\overline{OR}_{any}(FanIn) = 4.26$), also reach the significance level here. However, several of the metrics that we observed to be significant in all cases for classes that are the source of architectural inconsistencies do not reach the significance level for all thresholds for one of the systems, but remain significant in case of the rest.

## 4.3 Significant metrics and size

Considering the results from the prior two sections, there are in total 13 unique types of metrics for which we could find a significant relation to architectural inconsistencies at some point. These refer to several size-related aspects, i.e., LOC, the number of statements, the number of calls, the number of comments, and the public API, but also to aspects that are

**Table 5** Spearman's rank correlation coefficients for selected metrics and lines of code—the data shows the correlation coefficients and $p$ values for the different systems

| Metric | JabRef | Lucene | Ant |
|---|---|---|---|
| WMC | 0.77 ($p < 2.2e^{-16}$) | 0.76 ($p < 2.2e^{-16}$) | 0.86 ($p < 2.2e^{-16}$) |
| Total issues (PMD) | 0.94 ($p < 2.2e^{-16}$) | 0.96 ($p < 2.2e^{-16}$) | 0.94 ($p < 2.2e^{-16}$) |
| Design issues (PMD) | 0.46 ($p < 2.2e^{-16}$) | 0.7 ($p < 2.2e^{-16}$) | 0.71 ($p < 2.2e^{-16}$) |
| Code smells (SQ) | 0.71 ($p < 2.2e^{-16}$) | 0.81 ($p < 2.2e^{-16}$) | 0.77 ($p < 2.2e^{-16}$) |
| Complexity | 0.9 ($p < 2.2e^{-16}$) | 0.96 ($p < 2.2e^{-16}$) | 0.97 ($p < 2.2e^{-16}$) |
| RFC | 0.91 ($p < 2.2e^{-16}$) | 0.87 ($p < 2.2e^{-16}$) | 0.90 ($p < 2.2e^{-16}$) |
| LCOM | 0.56 ($p < 2.2e^{-16}$) | 0.33 ($p : 5.65e^{-14}$) | 0.71 ($p < 2.2e^{-16}$) |
| Fan-in | 0.14 ($p : 5.08e^{-5}$) | 0 ($p : 0.96$) | 0.09 ($p : 0.01$) |

not directly related to size, i.e., the weighted method count, the total number of PMD issues, the number of PMD design issues, the number of code smells reported by SonarQube, the cyclomatic complexity, response for class, lack of method cohesion, and fan-in. For all of the latter metrics, there is no normalization regarding class size and, hence, it is not clear if their ability to discriminate between classes with architectural inconsistencies and classes without is just a side effect biased by class size.

To investigate this, we computed the Spearman rank correlation coefficient between the metrics not directly related to size and selected size metrics. To be precise, we computed the correlation coefficient to LOC and the number of statements, since these are the significant size metrics that are not focused on a specific aspect of the code, such as comments or public methods. The correlation coefficients have been nearly identical for both cases, which is why we just report on the correlation coefficients to LOC here. The results are listed in Table 5.

It can be seen from Table 5 that all metrics that represent anomaly counts have a strong correlation with LOC, with the notable exception of design issues in JabRef ($\rho = 0.46$) that is comparably smaller. The same applies to the weighted method count, complexity, and the response for class for all systems. For lack of method cohesion, correlation coefficients vary from relatively weak in the case of Lucene ($\rho = 0.33$) to rather strong in the case of Ant ($\rho = 0.71$). For fan-in, correlation values are close to zero, but in the case of Lucene, the correlation coefficient is not significant. Nevertheless, based on the results for JabRef and Ant, this still indicates that there is hardly a correlation between fan-in and LOC. To sum up, fan-in seems to be the only metric that is independent of a size bias.

## 5 Discussion

In this section, we discuss the results of our analysis and highlight key findings and implications for research and practice. Furthermore, possible threats to validity and reliability are discussed.

### 5.1 Interpretation of the results

Two of the metrics, a size metric and a coupling metric, stick out in every part of the results: fan-in ($\overline{OR}_{trg}(FanIn) = 8.40$, $\overline{OR}_{any}(FanIn) = 4.26$) and the public API

($\overline{OR}_{src}(PublicAPI) = 4.22$, $\overline{OR}_{trg}(PublicAPI) = 3.32$, $\overline{OR}_{any}(PublicAPI) = 3.96$) have been significant for all systems and for all thresholds, with the only exception of the fan-in for classes being the source of architectural inconsistencies. For these metrics in particular, classes with higher values are consistently more likely to contribute to divergences from the intended architecture. In some cases, such as for the fan-in for classes being the target of an inconsistency in case of Lucene (see Table 7, Lucene threshold 90th percentile), this likelihood is more than 13 times as high as for classes with lower values. Moreover, the very same metrics show a moderate correlation with the number of architectural inconsistencies for some of the systems, as reported in Section 4.1. On top of that, fan-in is not correlated with size, as reported in Section 4.3, which means that it can help to identify architectural inconsistencies without a size bias. From a conceptual point of view, it seems plausible that both metrics are related to architectural inconsistencies: The higher the number of publicly visible elements that a class provides is and the more often the class is referenced elsewhere, the higher is the chance that it is used by classes that should not access it architecture-wise. Classes with many public members are designed for being used, as opposed to classes with visibility restrictions. It seems likely that classes with a lot of public members that are also accessed from a lot of other classes form the interface, or the boundary, of an architectural module. Therefore, it is not surprising if classes with such properties participate more frequently in architectural inconsistencies than other classes. An example of such a class is the *Globals* class in JabRef, which participates in the second highest number of inconsistencies of the complete system. This class contains solely public methods, 14 of its 18 variables are public, and it is in the top 10% when it comes to the size of public API. It also has the second highest fan-in in the system. The class is intended as a communication point between only two of JabRef's modules, but it is being used from most other modules as well, although this is not intended according to the architectural model. This discussion can be summarized in the following way:

*Finding 1: The fan-in and the public API metrics are the most suitable indicators for classes with architectural inconsistencies found in this study.*

Interestingly, the public API is also suitable for indicating classes that are the sources of inconsistencies ($\overline{OR}_{src}(PublicAPI) = 4.22$). Again, the reason for this might be that such classes stand at the boundary of an architectural module, like the *Globals* class in JabRef mentioned above, and connect to classes from other modules, which in some cases they should not. Curiously, the counterpart of fan-in, fan-out, did neither show a moderate correlation nor did it consistently reach the significance level. Since inconsistencies link a source and a target class, it could have been expected that this metric would be equally indicative. Nevertheless, based on the reported results, we cannot confirm such a relation. The reason for this might be that the metric as computed by the tooling also includes dependencies to classes of the Java API. This could be considered as noise in the metric value for our use case, since such dependencies are typically not restricted. We have found a similar lack of significance for the remaining coupling metrics, e.g., message passing coupling, data abstraction coupling, locality of data, or the inheritance-related coupling metrics. Message passing coupling and data abstraction coupling reached the significance level for classes being the source of inconsistencies in many cases, but not in all, and failed to reach this threshold in nearly all cases for classes being the target of inconsistencies. Since coupling concerns the relationships among classes, it could have been expected that coupling metrics are more suitable indicators for architectural inconsistencies than other types of metrics. However, our data provides no evidence for such a claim. Overall, it is a surprising finding that not more metrics show an equally strong relation to architectural inconsistencies as fan-in and the public API. From the findings reported in related work, a much larger set of metrics could have been expected.

Moreover, the results for method counts are striking.

*Finding 2: Method counts seem to be suitable for indicating architectural inconsistencies.*

Although the presentation in Section 4.2 is limited to the weighted method count from CKJM ($\overline{OR}_{src}(WMC) = 4.68$, $\overline{OR}_{any}(WMC) = 3.07$), basically all method count metrics we considered were significant, including the public API metric discussed above. The relation between method counts and architectural inconsistencies is strengthened by the fact that we could reproduce this relationship for similar metrics from different tools with small differences, such as counting all methods in a class or only the public ones. Taken together with the findings discussed above, this allows us to answer our first research question:

*RQ1: Method counts and the fan-in show a moderate positive correlation with the amount of architectural inconsistencies that a class participates in.*

Interestingly, method counts seem more suitable for indicating source classes of architectural inconsistencies than for indicating target classes. This is surprising, because classes with many methods have a higher potential for reuse and could therefore be expected to be the targets of inconsistencies. In a similar vein, the amount of metrics that are suitable for indicating classes that are the source of inconsistencies is much larger.

When it comes to classes that are the source of architectural inconsistencies, the following can be said:

*Finding 3: Classes which violate various design principles (being very large, complex, and containing many code smells) are significantly more likely to be the source of architectural inconsistencies.*

This confirms general development principles that emphasize avoidance of such design issues. In contrast, classes that are the target of an inconsistency do not generally violate many design principles. This seems plausible, since classes with design issues are probably not easier to access and reuse than well-engineered classes. In this case, a placement of the class in the wrong architectural module is more likely.

A curious observation is that the number of comments ($\overline{OR}_{src}(Comments) = 4.25$, $\overline{OR}_{any}(Comments) = 3.50$) seems to be related to architectural inconsistencies. A possible explanation for this phenomenon is that classes which contain architecture problems are commented more often, where the need to comment is a symptom of the problem. An example where this is the case is the *BasePanel* class of JabRef, which is one of the top-level UI containers in the system. It is the fourth most heavily commented class and also third in terms of the total amount of inconsistencies in which it participates. Some of the comments in the class relate to architectural problems, for example there are comments that: (i) state that if a chunk of code in the class is changed, then a number of other classes in different modules need to be changed as well, (ii) explain the functioning of different parts of the system for asynchronous processing to justify why a chunk of code looks as it does, or (iii) state that a solution is a “quickfix/dirty hack,” because a better solution is not directly possible in the current structure.

Finally, there are a number of metrics that did not stick out during the analysis at any particular point, like the coupling metrics already mentioned above. This includes all anomaly counts reported by FindBugs and several anomaly counts reported by SonarQube and PMD.

*Finding 4: Anomaly counts and detection strategies are less suitable indicators than more traditional source code metrics.*

Anomaly counts were only found to be significantly related to classes that are the source of architectural inconsistencies. Ultimately, also classes that are targeted by an inconsistency can be the original cause, for instance by being misplaced in the wrong module. Due to this blind spot, anomaly counts seem to be less suitable for indicating architecture degradation. In addition to that, Section 4.3 shows that anomaly count and code smell metrics are strongly correlated with code size. This suggests that it might be more worthwhile to consider size metrics directly when trying to identify classes with architectural inconsistencies.

It could also be expected that a relation between technical debt and architectural inconsistencies exists, as the violation of the architecture for short-term gains is a typical symptom of technical debt. However, the results suggest the opposite.

*Finding 5: The metrics related to technical debt applied in this study are not suitable to indicate classes contributing to architectural inconsistencies.*

The relevant metrics used in this study are sqale index and sqale debt ratio implemented in SonarQube. Since the concept of technical debt is still new, current ways of quantifying it might still be too immature to uncover such a relation. Moreover, for aggregated metrics, such as the average block depth in a class, we were also unable to find any reliable significant relationship for the three systems.

The number of commits that touched a class showed a significant difference in some cases, indicating that classes that are changed more often than their peers seem to be more prone to architectural inconsistencies. Frequent changes in a class might indicate that the implemented functionality is not stable and also its purpose may change over time. If, in the face of such changes, the class is not realigned to the intended architecture, inconsistencies are the result. However, the number of commits was not a significant discriminator in all cases. Lastly, the age of a class did not stick out either. Overall, the findings discussed so far let us answer our second research question:

*RQ2: A variety of metrics can be used to discriminate between classes that are more likely to participate in architectural inconsistencies and those that are less likely to do so, including size, coupling, cohesion, and complexity metrics, as well as anomaly counts, but the best discriminators are the fan-in and the size of the public API of a class.*

When it comes to the relationship of metrics to code size, as discussed in Section 4.3, there are only two metrics that are not either size metrics, such as the public API, and that are not strongly correlated with size: fan-in and lack of method cohesion. This lets us answer our final research question:

*RQ3: There is a confounding effect of class size on most metrics that are suitable discriminators, but especially the fan-in and, to a limited degree, lack of method cohesion are exceptions to this effect.*

This result firstly suggests that more suitable architectural quality metrics are indeed needed, because most of the current body of metrics boil down to size. Nevertheless, especially the results for fan-in provide empirical evidence that it might be possible to identify architectural inconsistencies automatically without a size bias.

## 5.2 Implications for research and practice

Based on the discussion of findings in the previous section, it is hard to form a practically usable recommendation regarding a suitable threshold value that can be used to separate architecturally consistent from inconsistent classes, based on the current data. Odds ratios are rising and falling for different thresholds and systems in inconsistent ways. This demonstrates, unsurprisingly, that there is no single best threshold that works for all systems. At the very least, it can be seen that in all cases, regardless of threshold or system, *higher metric values correspond to classes being more prone to architectural inconsistencies*. Put the other way around, smaller and less complex classes with fewer methods, i.e., classes that are well-designed according to established design principles, are less likely to diverge from the intended architecture of a system. Thus, our recommendation for practitioners who are trying to prioritize classes for architectural repair is to select the top-most classes in terms of size, fan-in, and complexity as it is likely to find the architecturally critical ones among these.

However, our results also show that these general metrics are far from being able to give satisfyingly precise advice on which parts of the source code are involved in architectural degradation. For software development practice, this means that software developers, designers, and architects should not rely on these metrics alone when trying to avoid architecture degradation. Making the intended architecture explicit by modeling it and detecting inconsistencies through tools like JITTAC seem to be more effective and efficient.

Our recommendation for research, and our goal for future work, is to develop architectural quality metrics that are not affected by a size bias. This requires more insights on the actual causes of architectural inconsistencies and also more empirical work with realistic systems, such as the one presented here.

## 5.3 Validity and reliability

The assessment of the validity of the study follows the categorization by Brewer which distinguishes external, construct, and internal validity (Brewer and Crano 2014).

Several threats exist regarding external validity, which concerns the generalizability of a study to a larger population. All systems under study are similar-sized systems implemented in Java. Hence, the results might not be generalizable to systems of very different size, for instance with several million lines of code, or to systems implemented in different, or several, languages. Further studies are required to complement the presented results. It is also an open question whether the results are generalizable to commercial, closed source systems.

Moreover, it is important to note that systems were studied for which the intended software architecture was recovered after being unavailable for a long period of development, as such scenarios motivated the research. Hence, the results might not be generalizable to systems in which a specification of the intended architecture has been available and has at least been partially followed during development.

Each of the projects developing one of the three sample systems uses at least one of the tools from which we gathered metric data for this study (none of them uses them all, though). One might consider this a threat to validity because potential correlation between architectural inconsistencies and metric values might be skewed by code being optimized towards the available metrics and hence argue that the results are not generalizable to systems being analyzed by the same or similar tools. An analysis of open-source repositories by Bholanath investigated 122 projects and found evidence of the use of static analysis tools comparable to the ones used in this study in 59% of all projects (Bholanath 2015). In a questionnaire-based investigation of 36 projects, the same study revealed usage of such tools in 77% of the projects. We are hence confident that investigating the research questions for systems already applying such tools is a relevant and representative scenario. However, it is certainly an interesting line for future research to investigate other scenarios, including systems not being analyzed through static code quality tools, as well.

We identified several threats related to construct validity, which refers to the degree to which measurements relate to the phenomenon under study. The first is connected to the numbers of architectural inconsistencies in classes. These inconsistencies are derived from the specification of an intended architecture and, therefore, errors in the structure of this specification might be a threat to validity. For two of the systems (Ant and Lucene), we used the intended architectures documented in the replication package of a study by Brunet et al., claiming to have evaluated the specifications (Brunet et al. 2012). The translation of these models into JITTAC was performed by one of the authors and cross-checked independently by the others. For JabRef, we developed the intended architecture in video-conferencing sessions with the current development team, of which the first author is a member. Hence, we are confident that all of the architectural specifications are valid and conform to the consensus held by the respective development teams.

A second threat to construct validity lies in the various metric suites we used to obtain quantitative data of the different systems, including our way of calculating architectural inconsistencies. There is always a chance that software tools contain faults that impact the results they produce or that their usage leads to errors. For instance, it could be the case that, due to a software fault, the tool misses a certain type of architectural inconsistency. The same might apply to our configuration of the metric suites or our scripts for transforming metric values. As before, the configuration and the development of the scripts for an automated metric transformation was performed by one of the authors and cross-checked independently by the others.

Another threat to construct validity lies in the selection of the statistical methods for the analysis. As Fisher's exact test (Upton 1992) makes no assumptions regarding distribution or sample size and also the Spearman rank correlation coefficient (Daniel 1990) is resilient regarding the distribution, we are confident that their interpretation in this study is valid.

It should also be noted that the phenomenon under study in this work is rather architectural inconsistencies as defined in Section 2 than software architecture degradation in general; whether an architectural inconsistency really constitutes a violation of the software architecture or whether it is an exception from the dependency rules specified by a reflexion model is not differentiated in this work.

Internal validity refers to the extent to which a study makes sure that a factor and only this factor causes an effect. As this study is not about providing evidence for such a cause-effect relationship but about exploring a correlation, we do not further discuss this type of validity.

Reliability refers to consistency in data gathering and analysis, such that performing the measurements and computation twice or more often will produce the same results. To increase reliability, manual steps performed by the first author of the study, such as modeling the intended architectures and writing the scripts for the statistical computations, were independently cross-checked by the co-authors. Moreover, the data gathered and procedures to perform the analysis are available as a replication package for cross-checks and replication by other researchers.

## 6 Related work

There are a couple of studies investigating the relationship between source code properties, as for example measured through source code metrics, and architectural inconsistencies. These studies are most closely related to the focus of this study and considered as the relevant related work.

Using various kinds of metrics to judge and compare the quality of software architectures is a method of architecture analysis (Dobrica and Niemela 2002). Here, we are not considering the quality of an intended architecture, but are trying to relate inconsistencies between such an architecture and the actual implementation to source code metrics. With this regard, the work by Macia et al. is highly related (Macia et al. 2012a, b). In these papers, the authors investigated anomalies in six systems and their relation to architectural problems. They found that automatically detected code anomalies that are computed based on metric thresholds have a poor relation to architectural violations (Macia et al. 2012b), but the relationship gets stronger if code anomalies are identified manually (Macia et al. 2012a).

Here, we also include automatically detected code anomalies and code smells in our analysis, but in addition, we focus on a more fine-grained level of abstraction: source code metrics in isolation. These metrics form the input for detection strategies. In the aforementioned study, smell detection, also in the automated case, requires a fine tuning of metric thresholds for each investigated system with manually selected values (Macia et al. 2012b). We refrained from such fine tuning, since it reduces the generalizability of the findings and is hard to achieve in practice (Fontana et al. 2015).

This is also in line with how the development teams of at least one of our case study applications apply metric tooling. The developers of JabRef deliberately decided against adjusting metric thresholds. Nevertheless, we are able to generally confirm the results by Macia et al. stating that automatically detected anomalies can hardly be used as a predicator variable for estimating the absolute number of architectural inconsistencies (Macia et al. 2012a). Although the prediction of such an absolute number might be too much to ask for, we found that source code metrics can still serve as indicators for classes that contain inconsistencies.

Moreover, two of the systems used in those studies that we were able to access, Health-Watcher and MobileMedia, were comparably small and data on the remaining systems unavailable. While the accessible systems consist of about 100 and 50 classes, respectively, the systems investigated in this article consist of more than 500 and 700 classes each. Largely based on the same systems, the authors also investigated if, and confirmed that, groups of code anomalies "flocking together" are more correlated with higher level design problems (such as "fat interface" or "overused interface") than single code anomalies (Oizumi et al. 2016; Garcia et al. 2009).

As architectural inconsistencies could be considered one of those design problems (which the authors do not touch upon), our research complements that work by focusing on this particular aspect. Moreover, we are interested in the link between code anomalies and architectural inconsistencies with the intended architecture (in which even a fat interface may still be valid with regard to the defined structure) whereas the authors of the study mentioned above state themselves that they "... focus on the relation among code anomalies rather than the architectural connection..." (Oizumi et al. 2016), saying that the focus is on the relationship between lower level and higher level anomalies in the “as implemented” architecture.

Another work on architectural inconsistencies, on which we partly base here, is presented by Brunet et al. (2012). The authors investigate the evolution of architectural inconsistencies over time, but do not relate their numbers to other structural properties. In this paper, we do not consider a period-wise development of inconsistencies, but try to relate them to source code metrics. Most importantly, the study has been the source for the intended architectures of Ant and Lucene (Brunet et al. 2012). We need to note that the properties we compute of the systems differ from the ones reported in Brunet et al. (2012). Notions such as lines of code are known to differ strongly with different methodologies. The same applies to class counts, depending on how inner or anonymous classes are treated, and also architectural inconsistencies. This is not a problem here, as we are not trying to replicate the said study (Brunet et al. 2012). What matters for us is consistency in computing metrics for the different systems here.

In or own prior work (Lenhard et al. 2017), we used JabRef to test the suitability of code smell detection tools for predicting architectural inconsistencies in classes. The fact that code smells alone were not found to be suitable was one of the motivations for exploring the suitability of source code metrics in this work. In addition, we gathered data on two additional systems for this study.

Metrics for identifying the causes of architectural inconsistencies have been proposed by Herold et al. (2015). These metrics try to characterize the reason why an architectural inconsistency occurred and do so by relying on other source code metrics. The focus of that study is hence slightly different but related in that such source code metrics are used in combination to identify different potential causes of architectural inconsistencies. However, the metrics are not used to tell apart classes contributing to architectural inconsistencies from those that do not.

De Oliviera Barros et al. use Ant as a case study for observing architectural degradation (de Oliveira Barros et al. 2015). They characterize the source code of the system over a longer period using various metrics that we also rely on here. Using search-based techniques of module clustering, they try to sort the classes of the latest release in a way that complies to the original architecture. This automatic repair of the system is not what we aim for, and the authors of the study concede that the final result of their approach would hardly be acceptable to the developers.

## 7 Conclusion and future work

In this paper, we explored the suitability of source code metrics for identifying classes that contribute to inconsistencies between the prescriptive “as intended architecture” of a software system and its descriptive “as implemented” software architecture. We determined these inconsistencies for three widely used open-source software systems by creating reflexion models based on their intended architectures. Subsequently, we computed 49 different source code metrics for all classes in the systems’ code bases using six popular and freely available source code measurement tools and the version control system. The data obtained was analyzed to investigate the relationship between architectural inconsistencies and source code metrics.

The results do not provide evidence for a strong correlation between metric values indicating potentially negative properties of classes, such as low cohesion, high coupling, or high complexity, and their contribution to architectural inconsistencies. However, the categorical analysis shows interesting results regarding the likelihood of classes to contribute to those inconsistencies. For many metrics, the likelihood for classes being in the category above the higher percentiles—which indicate the most critical classes w.r.t. the particular metrics—is significantly higher. This is particularly true for the fan-in metric and the size of the public API of a class. In other words, these results suggest that classes in a system having “worse” metric values are significantly more likely to contribute to architectural inconsistencies than classes indicating good design properties. Thus, it can be said that these metrics might help characterize and identify architecturally critical classes. Moreover, the results suggest that traditional source code metrics are better indicators of architectural inconsistencies than anomaly counts, because the latter ones do not appropriately capture classes that are the target of inconsistencies.

However, class size seems to have a confounding effect on most metrics, except for the fan-in metric and, to a lesser degree, the lack of method cohesion metric. The latter two metrics show that there is potential for identifying classes with architectural inconsistencies without a size bias. In summary, it seems that more work is needed before metric suites are really usable for the task of architectural repair. To this end, the development of and support for more architecture-specific metrics, which are not confounded by class size, is desirable.

Several further paths for future work follow from these results. First, more empirical work is needed to investigate the generalizability of the findings. We aim at analyzing more large-scale systems that are implemented in different languages. Further analysis will ideally also involve proprietary systems developed in industry.

A second area of future work that we have started addressing is the transition from this study to concrete techniques and tools. The presented results only indicate an increased likelihood of architectural inconsistencies for a certain category of classes; however, the underlying classification does not yet perform well enough to identify such classes with satisfiable precision and recall. To improve this situation, we are going to look at combinations of metrics by combining various metrics in a regression model instead of looking at them in isolation. For instance, metrics above a certain threshold for fan-in *and* the public API might be much more likely to contain inconsistencies than if the metrics are considered in isolation. This also involves looking at metrics on package level which might provide relevant information about a class’ context in the code base, such as metrics about afferent and efferent coupling of packages (Martin 2003). We are going to analyze whether individual thresholds for particular metrics and metric combinations result in a more accurate disambiguation between classes that contribute to inconsistencies and classes that do not. This also involves taking metrics into account that provide information about the codebase on a more coarse-grained level, e.g., at package level, which might be useful. Due to the large amount of metrics, and the resulting large amount of metric thresholds and combinations thereof, we are currently applying data mining and machine learning techniques to search for promising combinations.

**Acknowledgments** The authors would like to thank the members of the JabRef development team who participated in the construction of the intended architecture of the system, in particular Simon Harrer, Tobias Diez, Matthias Geiger, Oliver Kopp, Stefan Kolb, and Christoph Schwentker.

## Appendix

Tables 6, 7, and 8 list the complete results of the Fisher tests for the metrics that were referenced in Section 4: *p* values, odds ratios, and confidence intervals.

**Table 6** Fisher's test data for source inconsistencies. The data is listed in the form *p* value, odds ratio, and confidence interval

| Metric | 50% | 75% | 90% |
|---|---|---|---|
| | JabRef | | |
| LOC | $5.41e^{-6}$, 2.60, [1.69; 4.08] | 0.00075, 2.08, [1.34; 3.22] | 0.023, 1.95, [1.05; 3.51] |
| Statements | $1.23e^{-6}$, 2.76, [1.78; 4.34] | 0.0023, 1.96, [1.25; 3.03] | 0.044, 1.83, [0.97; 3.32] |
| Calls | $4.51e^{-7}$, 2.87, [1.85; 4.54] | 0.01, 1.73, [1.11; 2.69] | 0.039, 1.95, [1.05; 3.51] |
| Comments | $8.77e^{-5}$, 2.23, [1.46; 3.43] | $6.21e^{-7}$, 2.95, [1.91; 4.54] | 0.0011, 2.58, [1.42; 4.57] |
| Public API | 0.00036, 2.08, [1.37; 3.16] | $1.08e^{-5}$, 2.59, [1.67; 4.01] | 0.00050, 2.76, [1.51; 4.92] |
| WMC | $1.52e^{-7}$, 2.95, [1.93; 4.56] | $3.78e^{-6}$, 2.73, [1.76; 4.20] | 0.024, 1.99, [1.05; 3.63] |
| RFC | $7.74e^{-8}$, 3.12, [2.00; 4.96] | 0.0034, 1.90, [1.21; 2.95] | 0.029, 1.95, [1.05; 3.51] |
| Total issues | $2.33e^{-7}$, 2.99, [1.92; 4.74] | $2.53e^{-7}$, 2.49, [1.61; 3.83] | 0.0043, 2.29, [1.25; 4.06] |
| Design issues | 0.0017, 1.92, [1.27; 2.91] | 0.0087, 1.95, [1.16; 3.19] | 0.013, 2.37, [1.13; 4.73] |
| Code smells | 0.00074, 2.00, [1.32; 3.03] | $9.26e^{-6}$, 2.64, [1.71; 4.09] | 0.00028, 2.85, [1.58; 5.03] |
| Complexity | $1.31e^{-6}$, 2.74, [1.77; 4.31] | $1.03e^{-7}$, 3.12, [2.03; 4.81] | 0.0022, 2.47, [1.37; 4.37] |
| LCOM | $2.11e^{-6}$, 2.67, [1.73; 4.17] | $2.05e^{-6}$, 2.78, [1.79; 4.28] | 0.0022, 2.47, [1.37; 4.37] |
| | Lucene | | |
| LOC | $9.81e^{-6}$, 2.60, [1.83; 5.92] | 0.00013, 2.82, [1.64; 4.84] | $4.70e^{-6}$, 4.68, [2.36; 9.17] |
| Statements | 0.00016, 2.69, [1.55; 4.78] | 0.00013, 2.82, [1.64; 4.84] | $4.70e^{-6}$, 4.68, [2.36; 9.17] |
| Calls | 0.0080, 1.96, [1.16; 3.39] | $1.70e^{-5}$, 3.14, [1.82; 5.40] | 0.00052, 3.42, [1.68; 6.78] |
| Comments | 0.00041, 2.53, [1.47; 4.46] | $6.49e^{-8}$, 4.22, [2.57; 7.28] | $1.01e^{-8}$, 6.86, [3.47; 13.56] |
| Public API | $2.48e^{-10}$, 5.41, [3.04; 10.01] | $1.08e^{-10}$, 5.53, [3.20; 9.64] | $6.20e^{-8}$, 6.57, [3.24; 13.27] |
| WMC | $1.36e^{-11}$, 6.78, [3.56; 13.87] | $3.81e^{-6}$, 3.41, [1.98; 5.87] | $7.46e^{-5}$, 4.06, [1.96; 8.20] |
| RFC | $1.68e^{-10}$, 4.17, [2.31; 7.91] | 0.0021, 2.29, [1.31; 3.96] | 0.00095, 3.27, [1.58; 6.56] |
| Total issues | 0.00040, 2.10, [1.23; 3.66] | 0.00021, 2.74, [1.58; 4.72] | 0.00052, 3.42, [1.68; 6.78] |
| Design issues | 0.0010, 2.35, [1.38; 4.07] | 0.0086, 2.05, [1.16; 3.55] | 0.0085, 3.42, [1.68; 6.78] |
| Code smells | 0.0010, 2.35, [1.38; 4.07] | $5.75e^{-5}$, 2.98, [1.72; 5.12] | $3.22e^{-6}$, 4.85, [2.43; 9.53] |
| Complexity | $2.85e^{-5}$, 3.03, [1.74; 5.34] | $1.84e^{-6}$, 3.54, [2.06; 6.10] | $4.70e^{-6}$, 4.68, [2.36; 9.17] |
| LCOM | 0.013, 1.89, [1.11; 3.26] | $2.32e^{-5}$, 3.15, [1.82; 5.44] | $8.98e^{-5}$, 3.80, [1.89; 7.50] |
| | Ant | | |
| LOC | $7.93e^{-5}$, 4.72, [2.01; 12.87] | $7.94e^{-9}$, 7.36, [3.50; 16.39] | $7.19e^{-7}$, 6.94, [3.19; 14.70] |
| Statements | 0.00019, 4.04, [1.78; 10.34] | $8.99^{-9}$, 7.30, [3.47; 16.25] | $4.09e^{-6}$, 6.20, [2.82; 13.23] |
| Calls | $4.21e^{-5}$, 4.80, [2.04; 13.07] | $6.81e^{-10}$, 8.60, [4.02; 19.65] | $7.78e^{-8}$, 8.01, [3.72; 16.94] |
| Comments | $2.53e^{-6}$, 7.11, [2.71; 23.58] | $1.66e^{-6}$, 5.13, [2.50; 10.83] | 0.00015, 4.67, [2.05; 10.13] |
| Public API | $6.74e^{-5}$, 4.13, [1.91; 9.68] | $1.22e^{-6}$, 5.25, [2.56; 11.09] | 0.002026, 3.68, [1.52; 8.23] |
| WMC | $3.13e^{-6}$, 6.20, [2.52; 18.36] | $4.79e^{-9}$, 7.59, [3.60; 16.90] | $2.91e^{-6}$, 6.43, [2.92; 13.73] |
| RFC | $1.52e^{-7}$, 9.35, [3.28; 36.64] | $5.61e^{-11}$, 10.04, [4.62; 23.67] | $9.15e^{-10}$, 10.26, [4.83; 21.72] |
| Total issues | $1.56e^{-5}$, 5.71, [2.32; 16.52] | $7.01e^{-9}$, 7.41, [3.52; 16.51] | $7.78e^{-8}$, 8.01, [3.72; 16.94] |
| Design issues | $3.17e^{-6}$, 5.78, [2.45; 15.75] | $3.70e^{-9}$, 7.71, [3.66; 17.17] | $2.46e^{-5}$, 5.44, [2.43; 11.69] |
| Code smells | $6.95e^{-5}$, 4.35, [1.97; 10.59] | $1.74e^{-5}$, 4.33, [2.12; 8.97] | $2.86e^{-5}$, 5.35, [2.39; 11.48] |
| Complexity | 0.00018, 4.08, [1.80; 10.45] | $8.90e^{-10}$, 8.46, [3.96; 19.34] | $7.20e^{-7}$, 6.93, [3.19; 14.70] |
| LCOM | $2.53e^{-6}$, 7.11, [2.72; 23.58] | $1.50e^{-6}$, 5.17, [2.52; 10.92] | $2.46e^{-5}$, 5.44, [2.43; 11.69] |

**Table 7** Fisher's test data for target inconsistencies. The data is listed in the form *p* value, odds ratio, and confidence interval

| Metric | 50% | 75% | 90% |
|---|---|---|---|
| | JabRef | | |
| Public API | $6.41e^{-7}$, 3.29, [2.00; 5.52] | $2.10e^{-8}$, 3.90, [2.39; 6.38] | $1.19e^{-8}$, 5.65, [3.09; 10.19] |
| Fan-in | $2.65e^{-16}$, 8.02, [4.49; 15.16] | $2.31e^{-12}$, 5.36, [3.28; 8.84] | $2.48e^{-15}$, 10.02, [5.61; 17.93] |
| | Lucene | | |
| Public API | 0.021, 2.06, [1.09; 3.94] | 0.00052, 3.06, [1.59; 5.87] | 0.0018, 3.60, [1.52; 8.01] |
| Fan-in | $3.25e^{-12}$, 12.32, [5.07; 36.18] | $8.82e^{-14}$, 10.80, [5.43; 22.43] | $6.57e^{-13}$, 13.44, [6.49; 28.16] |
| | Ant | | |
| Public API | 0.024, 2.11, [1.08; 4.25] | 0.0085, 2.48, [1.23; 4.85] | 0.0012, 3.69, [1.60; 8.00] |
| Fan-in | $7.07e^{-8}$, 7.75, [3.18; 22.78] | 0.00012, 3.61, [1.82; 7.14] | 0.00026, 4.31, [1.91; 9.20] |

**Table 8** Fisher's test data for source inconsistencies. The data is listed in the form *p* value, and odds ratio, and confidence interval

| Metric | 50% | 75% | 90% |
|---|---|---|---|
| | JabRef | | |
| LOC | $6.57e^{-5}$, 1.99, [1.40; 2.85] | $5.37e^{-5}$, 2.15, [1.47; 3.13] | 0.0027, 2.22, [1.30; 3.75] |
| Statements | 0.00018, 1.92, [1.35; 2.74] | 0.00012, 2.07, [1.41; 3.02] | 0.047, 1.71, [0.99; 2.91] |
| Fan-in | $5.00e^{-12}$, 3.32, [2.32; 4.77] | $2.37e^{-11}$, 3.50, [2.39; 5.11] | $1.00e^{-9}$, 4.92, [2.88; 8.46] |
| Comments | $2.97e^{-5}$, 2.06, [1.45; 2.94] | $2.27e^{-8}$, 2.84, [1.95; 4.15] | $5.81e^{-6}$, 3.23, [1.90; 5.47] |
| Public API | $1.38e^{-7}$, 2.48, [1.74; 3.53] | $1.59e^{-10}$, 3.35, [2.29; 4.90] | $2.18e^{-8}$, 4.31, [2.52; 7.44] |
| WMC | $1.14e^{-8}$, 2.67, [1.88; 3.82] | $1.26e^{-7}$, 2.69, [1.84; 3.92] | 0.00021, 2.71, [1.57; 4.63] |
| RFC | $9.50e^{-6}$, 2.17, [1.52; 3.11] | 0.00075, 1.91, [1.30; 2.79] | 0.0010, 2.37, [1.39; 4.00] |
| Complexity | $1.41e^{-5}$, 2.13, [1.50; 3.06] | $4.43e^{-7}$, 2.57, [1.76; 3.74] | 0.00012, 2.70, [1.59; 4.53] |
| LCOM | $7.09e^{-10}$, 2.94, [2.05; 4.27] | $3.17e^{-8}$, 2.85, [1.95; 4.16] | $5.35e^{-5}$, 2.86, [1.69; 4.82] |
| | Lucene | | |
| LOC | 0.015, 1.69, [1.09; 2.65] | 0.0098, 1.87, [1.15; 3.00] | 0.0022, 2.69, [1.40; 5.11] |
| Statements | 0.0081, 1.77, [1.14; 2.78] | 0.0030, 2.03, [1.25; 3.26] | 0.0062, 2.95, [1.54; 5.60] |
| Fan-in | $9.65e^{-8}$, 3.19, [2.03; 5.06] | $1.81e^{-8}$, 3.79, [2.34; 6.15] | $2.09e^{-8}$, 5.65, [2.97; 10.87] |
| Comments | $8.02e^{-5}$, 2.37, [1.51; 3.76] | $2.72e^{-10}$, 4.28, [2.67; 6.87] | $5.65e^{-9}$, 6.24, [3.24; 12.25] |
| Public API | $3.36e^{-9}$, 3.64, [2.31; 5.81] | $4.15e^{-12}$, 5.03, [3.12; 8.16] | $8.75e^{-8}$, 5.71, [2.90; 11.45] |
| WMC | $7.50e^{-8}$, 3.31, [2.08; 5.33] | $2.22e^{-7}$, 3.35, [2.09; 5.38] | 0.0013, 2.96, [1.50; 5.78] |
| RFC | 0.00097, 2.07, [1.32; 3.26] | 0.047, 1.62, [0.99; 2.63] | 0.047, 1.95, [0.97; 3.79] |
| Complexity | 0.0080, 1.77, [1.14; 2.77] | 0.00020, 2.37, [1.48; 3.81] | 0.00016, 3.24, [1.70; 6.14] |
| LCOM | 0.020, 1.67, [1.07; 2.61] | $5.18e^{-6}$, 2.91, [1.80; 4.68] | $1.19e^{-5}$, 3.90, [2.05; 7.40] |
| | Ant | | |
| LOC | $7.95e^{-5}$, 2.73, [1.61; 4.75] | $3.82e^{-7}$, 3.53, [2.14; 5.83] | $4.21e^{-7}$, 4.67, [2.55; 8.41] |
| Statements | $8.09e^{-5}$, 2.68, [1.59; 4.62] | $5.95e^{-8}$, 3.84, [2.32; 6.35] | $3.22e^{-7}$, 4.77, [2.60; 8.59] |
| Fan-in | $1.12e^{-7}$, 3.77, [2.20; 6.69] | $1.80e^{-7}$, 3.72, [2.24; 6.18] | $6.95e^{-10}$, 6.45, [3.55; 11.58] |

**Table 8** (continued)

| Metric | 50% | 75% | 90% |
|---|---|---|---|
| Comments | $2.73e^{-6}$, 3.36, [1.94; 6.02] | $7.24e^{-8}$, 3.77, [2.28; 6.24] | 0.00012, 3.34, [1.76; 6.14] |
| Public API | $4.30e^{-6}$, 3.04, [1.83; 5.16] | $5.54e^{-8}$, 3.87, [2.34; 6.41] | $4.21e^{-6}$, 4.21, [2.25; 7.66] |
| WMC | $4.66e^{-5}$, 2.79, [1.66; 4.82] | $1.53e^{-7}$, 3.65, [2.21; 6.04] | $6.57e^{-5}$, 3.54, [1.86; 6.52] |
| RFC | $1.63e^{-5}$, 2.97, [1.75; 5.22] | $6.42e^{-7}$, 3.47, [2.10; 5.74] | $6.32e^{-8}$, 5.17, [2.83; 9.21] |
| Complexity | $1.58e^{-5}$, 3.01, [1.77; 5.28] | $6.49e^{-12}$, 5.38, [3.24; 9.01] | $1.57e^{-8}$, 5.48, [3.02; 9.79] |
| LCOM | 0.00055, 2.40, [1.43; 4.11] | $8.19e^{-7}$, 3.38, [2.05; 5.60] | $9.81e^{-5}$, 3.41, [1.80; 6.26] |

## References

Ali, N., Rosik, J., Buckley, J. (2012). Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study. In *Proceedings of the 8th international ACM SIGSOFT conference on quality of software architectures QoSA '12* (pp. 23–32). New York: ACM. https://doi.org/10.1145/2304696.2304702.

Ali, N., Baker, S., O'Crowley, R., Herold, S., Buckley, J. (2017). Architecture consistency: state of the practice, challenges and requirements. *Empirical Software Engineering*. https://doi.org/10.1007/s10664-017-9515-3.

Bachmann, F., Bass, L., Klein, M., Shelton, C. (2005). Designing software architectures to achieve quality attribute requirements. *IEE Proceedings - Software*, *152*(4), 153–165. https://doi.org/10.1049/ip-sen:20045037.

Bansiya, J., & Davis, C.G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, *28*(1), 4–17.

Bholanath, R. (2015). *Analyzing the state of static analysis: a large-scale evaluation in open source soft*. Master's thesis, Delft University of Technology.

Brewer, M.B., & Crano, W.D. (2014). *Research design and issues of validity. Handbook of research methods in social and personality psychology, 2nd edn.* (pp. 11–26).

Briand, L.C., Daly, J.W., Wüst, J. K. (1999). A unified framework for coupling measurement in object oriented systems. *IEEE Transactions on Software Engineering*, *25*(1), 91–121.

Brunet, J., Bittencourt, R.A., Serey, D., Figueiredo, J. (2012). On the evolutionary nature of architectural violations. In *2012 19th Working conference on reverse engineering* (pp. 257–266). IEEE.

Brunet, J., Murphy, G.C., Serey, D., Figueiredo, J. (2015). Five years of software architecture checking: a case study of eclipse. *IEEE Software*, *32*(5), 30–36. https://doi.org/10.1109/MS.2014.106.

Buckley, J., Mooney, S., Rosik, J., Ali, N. (2013). JITTAC: a just-in-time tool for architectural consistency. In *Proceedings of the 35th international conference on software engineering*. San Francisco.

Buckley, J., Ali, N., English, M., Rosik, J., Herold, S. (2015). Real-time reflexion modelling in architecture reconciliation: a multi case study. *Information and Software Technology*, *61*, 107–123. https://doi.org/10.1016/j.infsof.2015.01.011. http://www.sciencedirect.com/science/article/pii/S0950584915000270.

Campbell, G.A., & Papapetrou, P.P. (2013). *SonarQube in action*. Manning Publications Co.

Chidamber, S.R., & Kemerer, C.F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895.

Dalgarno, M. (2009). When good architecture goes bad. *Methods and Tools*, *17*, 27–34.

Daniel, W.W. (1990). Applied nonparametric statistics. PWS-kent, chap Spearman rank correlation coefficient, pp 358–365. ISBN: 0-534-91976-6.

Deiters, C., Dohrmann, P., Herold, S., Rausch, A. (2009). Rule-based architectural compliance checks for enterprise architecture management. In *2009 IEEE International enterprise distributed object computing conference* (pp. 183–192). https://doi.org/10.1109/EDOC.2009.15.

de Oliveira Barros, M., de Almeida Farzat, F., Travassos, G.H. (2015). Learning from optimization: a case study with apache ant. *Information and Software Technology*, *57*, 684–704.

de Moor, O., Sereni, D., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Tibble, J. (2008). *QL: object-oriented queries made easy*, (pp. 78–133). Berlin: Springer.

de Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: as survey. *Journal of Systems and Software*, *85*(1), 132–151. https://doi.org/10.1016/j.jss.2011.07.036.

Ding, W., Liang, P., Tang, A., Vliet, H., Shahin, M. (2014). How do open source communities document software architecture: an exploratory survey. In *2014 19th International conference on engineering of complex computer systems* (pp. 136–145). https://doi.org/10.1109/ICECCS.2014.26.

Dobrica, L., & Niemela, E. (2002). A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, *28*(7), 638–653. https://doi.org/10.1109/TSE.2002.1019479.

Duszynski, S., Knodel, J., Lindvall, M. (2009). Save: software architecture visualization and evaluation. In *2009 13th European conference on software maintenance and reengineering* (pp. 323–324). https://doi.org/10.1109/CSMR.2009.52.

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

Emam, K.E., Benlarbi, S., Goel, N., Rai, S. (2001). The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, *27*(7), 630–650.

Ericsson, M., Wingkvist, A., Löwe, W. (2012). The design and implementation of a software infrastructure for IQ assessment. *International Journal of Information Quality*, *3*(1), 49–70.

Faragó, C., Hegedűs, P., Ferenc, R. (2015). Impact of version history metrics on maintainability. In *8th International conference on advanced software engineering & its applications (ASEA)*. Jeju Island.

Fontana, F.A., Ferme, V., Zanoni, M., Yamashita, A. (2015). Automatic metric thresholds derivation for code smell detection. In *Proceedings of the Sixth international workshop on emerging trends in software metrics* (pp. 44–53). IEEE Press.

Foster, J., Hicks, M., Pugh, W. (2007). Improving software quality with static analysis. In *7th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering* (pp. 83–84).

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Boston: Addison-Wesley Longman Publishing Co., Inc.

Garcia, J., Popescu, D., Edwards, G., Medvidovic, N. (2009). *Toward a catalogue of architectural bad smells*, (pp. 146–162). Berlin: Springer.

Godfrey, M.W., & Lee, E.HS. (2000). Secrets from the monster: extracting Mozilla's software architecture. In *Proc. of 2000 Intl. symposium on constructing software engineering tools (Co SET 2000)* (pp. 15–23).

Hall, G.A., & Munson, J.C. (2000). Software evolution: code delta and code churn. *Journal of Systems and Software*, *54*(2), 111–118.

Hello2morrow (2017). Sonargraph webpage. www.hello2morrow.com, Accessed 24 July 2017.

Herold, S., & Rausch, A. (2013). Complementing model-driven development for the detection of software architecture erosion. In *Proceedings of the 5th international workshop on modeling in software engineering MiSE '13* (pp. 24–30). Piscataway: IEEE Press. http://dl.acm.org/citation.cfm?id=2662737.2662744.

Herold, S., English, M., Buckley, J., Counsell, S., Cinnéide, M.Ó. (2015). Detection of violation causes in reflexion models. In *IEEE 22nd International conference on software analysis, evolution, and reengineering (SANER)* (pp. 565-569). IEEE.

Herold, S., Blom, M., Buckley, J. (2016). Evidence in architecture degradation and consistency checking research: preliminary results from a literature review. In *Proccedings of the 10th European conference on software architecture workshops ECSAW '16* (pp. 20:1–20:7). New York: ACM. https://doi.org/10.1145/2993412.3003396.

Hochstein, L., & Lindvall, M. (2005). Combating architectural degeneration: a survey. *Information and Software Technology*, *47*(10), 643–656. https://doi.org/10.1016/j.infsof.2004.11.005.

IEEE (1998). IEEE Std 1061-1998 (R2009). IEEE Standard for a Software Quality Metrics Methodology. Revision of IEEE Std 1061–1992.

Knodel, J., & Popescu, D. (2007). A comparison of static architecture compliance checking approaches. In *2007 Working IEEE/IFIP conference on software architecture (WICSA'07)* (pp. 12–12). https://doi.org/10.1109/WICSA.2007.1.

Knodel, J., Muthig, D., Haury, U., Meier, G. (2008). Architecture compliance checking—experiences from successful technology transfer to industry. In *2008 12th European conference on software maintenance and reengineering* (pp. 43–52). https://doi.org/10.1109/CSMR.2008.4493299.

Lanza, M., Marinescu, R., Ducasse, S. (2005). *Object-oriented metrics in practice*. Secaucus: Springer.

Lenhard, J., Hassan, M.M., Blom, M., Herold, S. (2017). Are code smell detection tools suitable for detecting architecture degradation? In *Proceedings of the 11th European conference on software architecture: companion proceedings ECSA '17* (pp. 138–144). New York: ACM. https://doi.org/10.1145/3129790.3129808.

Letouzey, J.L. (2012). The SQALE method for evaluating technical debt. In *3rd International workshop on managing technical debt*. Zurich.

Macia, I., Arcoverde, R., Garcia, A., Chavez, C., von Staa, A. (2012a). On the relevance of code anomalies for identifying architecture degradation symptoms. In *16th European conference on software maintenance and reengineering*.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., von Staa, A. (2012b). 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* (pp. 167–178). ACM https://doi.org/10.1145/2162049.2162069.

Mair, M., Herold, S., Rausch, A. (2014). Towards flexible automated software architecture erosion diagnosis and treatment. In *Proceedings of the WICSA 2014 companion volume, WICSA '14 Companion* (pp. 9:1–9:6). ACM https://doi.org/10.1145/2578128.2578231.

Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International conference on software maintenance, 2004. Proceedings.* (pp. 350–359). https://doi.org/10.1109/ICSM.2004.1357820.

Martin, R.C. (2003). *Agile software development: principles, patterns, and practices*. Upper Saddle River: Prentice Hall.

Mattsson, A., Fitzgerald, B., Lundell, B., Lings, B. (2012). An approach for modeling architectural design rules in UML and its application to embedded software. *ACM Transactions on Software Engineering and Methodology*, *21*(2), 10,1–10,29. https://doi.org/10.1145/2089116.2089120.

Murphy, G.C., Notkin, D., Sullivan, K.J. (2001). Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, *27*(4), 364–380. https://doi.org/10.1109/32.917525.

Oizumi, W., Garcia, A., da Silva Sousa, L., Cafeo, B., Zhao, Y. (2016). Code anomalies flock together. In *38th IEEE International conference on software engineering*. Austin.

Olsson, T., Ericsson, M., Wingkvist, A. (2017). The relationship of code churn and architectural violations in the open source software JabRef. In *Proceedings of the 11th European conference on software architecture: companion proceedings ECSA '17* (pp. 152–158). New York: ACM. https://doi.org/10.1145/3129790.3129810.

Passos, L., Terra, R., Valente, M.T., Diniz, R., das Chagas Mendonca, N. (2010). Static architecture-conformance checking: an illustrative overview. *IEEE Software*, *27*(5), 82–89. https://doi.org/10.1109/MS.2009.117.

Perry, D.E., & Wolf, A.L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*(4), 40–52. https://doi.org/10.1145/141874.141884.

R Core Team. (2016). *R: a language and environment for statistical computing*. Vienna: R Foundation for Statistical Computing. https://www.R-project.org/.

Raza, A., Vogel, G., Plödereder, E. (2006). *Bauhaus—a tool suite for program analysis and reverse engineering*, (pp. 71–82). Berlin: Springer.

Rosik, J., Le Gear, A., Buckley, J., Babar, M.A., Connolly, D. (2011). Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience*, *41*(1), 63–86. https://doi.org/10.1002/spe.999.

Rozanski, N., & Woods, E. (2005). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley Professional.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*, 131.

Sangal, N., Jordan, E., Sinha, V., Jackson, D. (2005). Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications OOPSLA '05* (pp. 167–176). New York: ACM, https://doi.org/10.1145/1094811.1094824.

Sarkar, S., Ramachandran, S., Kumar, G.S., Iyengar, M.K., Rangarajan, K., Sivagnanam, S. (2009). Modularization of a large-scale business application: a case study. *IEEE Software*, *26*(2), 28–35. 10.1109/MS.2009.42.

Shapiro, S., & Wilk, M.B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, *52*(3–4), 591–611.

Sommerville, I. (2010). *Software engineering*, 9th edn. USA: Addison-Wesley.

Spinellis, D. (2005). Tool writing: a forgotten art? *IEEE Software*, *22*(4), 9–11.

Spinellis, D. (2012). Git. *IEEE Software*, *29*(3), 100–101.

Structure101 (2017). Structure101 webpage. www.structure101.com, Accessed 24 July 2017.

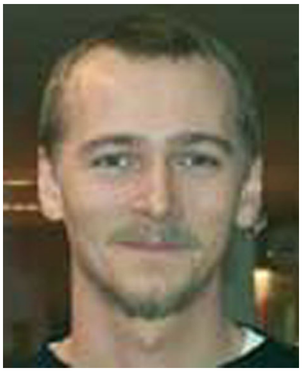
Taylor, R.N., Medvidovic, N., Dashofy, E.M. (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing.

Upton, G.J. (1992). Fisher's exact test. *Journal of the Royal Statistical Society Series A (Statistics in Society)*, 395–402.

van Gurp, J., & Bosch, J. (2002). Design erosion: problems and causes. *Journal of Systems and Software*, *61*(2), 105–119.

van Gurp, J., Brinkkemper, S., Bosch, J. (2005). Design preservation over subsequent releases of a software product: a case study of Baan ERP. *Journal of Software Maintenance Research and Practice*, *17*(4), 277–306. https://doi.org/10.1002/smr.313.

**Jörg Lenhard** is a postdoctoral research fellow in the Software Engineering Research Group at the Department of Mathematics and Computer Science of Karlstad University, Sweden. He received his Ph.D. from the University of Bamberg, Germany. Jörg's research interests include the software quality, software architecture, service-oriented computing, and process-aware information systems.



**Martin Blom** is an associate professor in the Software Engineering Research Group at the Department of Mathematics and Computer Science of Karlstad University, Sweden, where he also received his Ph.D. Martin's research interests include empirical software engineering, software architecture, software testing, and data mining.

**Sebastian Herold** is an assistant professor in the Software Engineering Research Group at the Department of Mathematics and Computer Science of Karlstad University, Sweden. He received his Ph.D. from Clausthal University of Technology, Germany, and worked at Lero - The Irish Software Research Centre in Limerick, Ireland. Sebastian's research interests focus on software architecture and design, software evolution, and software quality.