2012 19th Working Conference on Reverse Engineering

On the Evolutionary Nature of Architectural Violations

João Brunet*, Roberto Almeida Bittencourt†, Dalton Serey*, Jorge Figueiredo*
*Federal University of Campina Grande, Brazil
{jarthur,dalton,abrantes}@dsc.ufcg.edu.br
†State University of Feira de Santana, Brazil
roberto@uefs.br

***Abstract*—Architectural conformance checking is the process of verifying whether a given software implementation conforms to the rules and decisions in an architectural model. Different conformance checking techniques have been proposed, and both academic and commercial tools use them for architecture evaluation. In this paper, we present an exploratory and longitudinal study on architectural violations of software systems. As an exploratory study, our aim has been to reveal patterns, raise hypotheses and form an initial body of knowledge on architectural violations, rather than test specific hypotheses. We have studied the evolution of four widely known open-source systems for which we have trusted architectural models. The study encompasses the analysis of 19 bi-weekly versions of each system. In total, we analyzed more than 3,000 violations. From our observations, we have derived a series of facts. Four of them, deserve attention: 1) development teams of all studied projects seem to be aware of the presence of architectural violations in the code and all of them do perform perfective maintenance aimed at eliminating such violations; 2) despite all effort, the number of architectural violations, in the long term, is continuously growing; 3) in all studied systems there is a critical core, i.e., just a few design entities are responsible for the majority of violations; and 4) some violations seem to be "respawning", i.e., they are eliminated, but are likely to be back in future versions of the system.**

I. Introduction

Architectural decisions are regarded as fundamental by most software practitioners to assure a successful software project. Good architectures are frequently credited for easy to maintain and evolve software systems and as a sign of internal quality.

Still, implementations that do not follow designed architectures are frequent enough to be considered the norm, not the exception. Terms like design erosion [1], architectural drift [2], code decay [3], and architecture degeneration [4] are often used to express the process by which progressive changes deviate an implementation from the designed architecture.

An architectural violation is a particular piece of code that does not follow its specified architectural constraints. Usually, an architectural constraint establishes illegal dependencies between design entities. For example, suppose a web application in which classes of the `gui` component should not directly access classes of the `model` component. In this case, a method call coming from any class of the `gui` component to a class of the `model` component is an architectural violation. While not considered as critical as a bug or as a functional violation, an architectural violation can be seen as a software defect, in the sense that it indicates a specific code entity that fails to have a specified property.

While there are techniques and tools to detect violations and check whether an implementation conforms to a given architectural reference model [5] - [12], many violations go totally unobserved by development teams — sometimes even unsuspected. In fact, major releases of large and relevant software products have meaningful amounts of architectural violations [13], [14].

Previous studies on this subject concentrate on presenting conformance checking techniques and tools, and how effective they are. In this study, we take a different approach. We focus on architectural violations lifecycle and location over time rather than on identifying them in a single version of the software. In order to do so, we performed a longitudinal and exploratory study on the evolutionary nature of the architectural violations of four open source systems. Our main goal is to better understand how violations unfold as time passes and to build empirical knowledge regarding their temporal behavior. We use the reflexion model technique [5] as an example of static architecture checking technique, due to the easiness of deriving its required high-level module views from existing documentation of open source systems.

Our study revealed a series of facts about architectural violations, how they evolve over time, their impact on architectural drift and how the different development teams deal with them. Four derived facts are particularly worth mentioning. First, we found that a number of violations are solved over time, which reveals that the studied projects tried to cope with the problem. Second, despite all effort, a number of violations are also introduced over time and, as a result, the number of architectural violations is continuously growing in the long term. Third, we found that most violations are located in a few design entities, which we named critical core, and that this core does not significantly change as software evolves. At last, we found that a meaningful number of violations are solved, but reappear in future versions of the system. In this work we named them recurring violations.

The remainder of this paper is organized as follows.

1095-1350/91 $25.00 © 4891 IEEE
DOI 10.1109/WCRE.2012.35

Section II briefly describes our study design. In Section III, we present the experimental results. After that, in Section IV and Section V, we discuss our results and threats to validity, respectively. In Section VI we discuss related work, and finally, Section VII concludes the paper with our final remarks and directions for future research.

II. Study Design

This section describes the experimental design of this work, which was conceived to guide the exploratory study. First, we present the research questions. Then, we introduce the subjects. After that, we describe the data collection. Finally, we present the applied experimental procedures and provide information about the replicability of this study.

A. Research Questions

In this exploratory study, our aim is to investigate architectural violations, how they evolve over time, their location, and how the development teams deal with them. In this context, we formulated the following research questions:

- `RQ1`: **How does the gap between code and architecture evolve over time?**
  We investigated the number of introduced and solved architectural violations over time.
- `RQ2`: **Are the violations equally spread over the design entities or they concentrate on a few ones?**
  We investigated the ratio between classes with violations and the total of classes, the distribution of violations per class and the classes with most violations.
- `RQ3`: **Once violations are solved in a given version, do they appear again in future versions?**
  We investigated the presence of recurring violations – violations that are solved, but reappear from time to time.

B. Subjects

The subjects of our study comprise four Java systems. Table I shows, for each system, the studied period, their size (KLOC) and frequency of commits. Ant [1] is a popular Java-based build automation tool. ArgoUML [2] is an open source UML modeling tool. Lucene [3] is a text search engine library written entirely in Java. And, SweetHome3D [4] is an interior design application that allows placing furniture in 2D plants with 3D previews.

Our requirements for the subjects were: systems from medium to large size that contained architectural documentation; systems with frequent short-term commits; commits should happen on a daily or short-term basis to allow the generation of meaningful bi-weekly data; software versions should be available from software repositories using version control systems. In addition, there should be an adequate time frame for extracting empirical data (we used a nine-month development period for adequate longitudinal observation). Finally, system had to have compilable source code in Java due to our design extraction tool, which reports facts from the bytecodes of Java systems [7].

[1] ant.apache.org
[2] argouml.tigris.org
[3] lucene.apache.org
[4] www.sweethome3d.com

C. Data Collection

To analyze architectural violations, some choices and assumptions were made about the experimental design, software versions and evolution period.

Assuming that an architectural module view remains stable for a longer period (e.g., some months between software releases), and that source code changes very frequently, sometimes more than once a day, we mined source code from software repositories at different time instants (bi-weekly), and computed architecture checks for each of these instants. Sampling should not be too frequent (commits) neither too sparse (releases). Too frequent sampling leads to noise because they are more likely to be unstable changes. On the other hand, too sparse sampling implies very few data points to analyze. Furthermore, analyzing releases could raise another threat. The longer the period the more likely for the architecture reference model to change. Thus we produced bi-weekly violation lists for each bi-weekly source code version. In this work, we extracted 20 bi-weekly versions for each subject system in a period of nine months. The first version is used as a reference, and the other 19 versions have their violations analyzed, i.e., when they first appear and whether they are solved, if that happens.

The violation lists were produced by applying the Reflexion Model technique [5]. In a nutshell, this technique consists in extracting high-level models, mapping the implementation entities onto these models and comparing the two artifacts, i. e., the high level models and the implemented design, checking where they agree and where they disagree. In this work, the high-level models were extracted from system documentation. SweetHome3D had design tests in the JDepend tool with packages as modules and assertions as the allowed dependencies between modules. Ant had a module view based on packages in the Lattix LDM tool. Lucene had a layered view diagram and we performed the mapping ourselves, using the package names as the basis for module attribution. Finally, ArgoUML had the most detailed design documentation: a set of module views and the packages that made up each module. The high-level models represent relevant features of the systems, but they are not intended to be complete. Thus, some features can be missing in the models. The mapping for the four systems was performed through regular expressions based on the names of packages that made up those modules. Section II-E contains instructions to obtain access to these models and mappings.

Table I
SUBJECT SYSTEMS

Subject	Studied period (first / last)	Revisions (first / last)	Size (KLOC) (min / max)	# Monthly Commits (min / max)
Ant	Jan-2007 / Oct-2007	500,752 / 584,500	232 / 239	22 / 164
ArgoUML	Feb-2007 / Nov-2007	12,103 / 13,713	397 / 813	120 / 286
Lucene	Jun-2010 / Feb-2011	978,784 / 1,075,001	247 / 336	58 / 173
SweetHome3D	Jun-2009 / Feb-2010	2,069 / 2,382	75 / 96	6 / 99

In order to better understand the quantitative data collected from violation lists, we also collected data from other sources. This activity was performed following two strategies: i) using SVN visual diff tool to compare subsequent versions of the software repositories and ii) manual inspection of the developers' public mailing list by date of interest. The former provided the code changes between two subsequent versions, which were used to explain some quantitative data. The latter provided data about architectural discussions and decisions of the development team, which were used to confirm some of our findings.

D. Procedures and Measures

Because our goal was to analyze the evolutionary nature of violations, we had to compute their lifetime. To accomplish this, we uniquely identified a violation through an *id*. This *id* is a tuple that contains the following information:

- *caller*: fully qualified name of the source code entity that violates the architectural rule;
- *callee*: fully qualified name of the source code entity used by the caller;
- *type*: violation dependency type. We considered the following types of dependency: method calls, field access, generalization, realization, catched and thrown exceptions, returned types and received parameters.

For the sake of clarity, let us analyze an example of an architectural violation:

- **caller**: `main.ConditionTask.execute()`
- **callee**: `gui.ConditionBase.countCond()`
- **type**: `calls`

This violation means that the `execute()` method from the `ConditionTask` class calls the `countCond()` method from the `ConditionBase` class. Given that this violation was detected in version *i*, it is trivial to find out whether it was solved or not in the following versions (*i+1, i+2 ...*). This allows us to compute a violation's lifecycle. For example, consider the following lifecycle for a hypothetical violation `P`:

`P`'s lifecycle: v1 v2 v9 v10 v11

Analyzing the violations' lifecycle for all the studied versions, we were able to compute:

- introduced violations per version;
- solved violations per version;
- architectural debt – the difference between introduced and solved violations per version;
- amount of recurring violations;
- degree of recurrence – the amount of times that a violation reappears in the system.

As we can see, the violation `id` also allows us to identify not only the method that causes the violation, but also its class, package and module, once this hierarchy is defined in the high-level module view. Using this information, we also measured:

- the amount of classes with violations;
- the amount of violations per class.

E. Replication Package

The architecture module views for Ant, ArgoUML, Lucene and SweetHome3D were obtained from Bittencourt's Ph.D. dissertation [15]. We provide these models, raw and processed data, and the scripts used to obtain the results of this work in the URL: http://code.google.com/p/on-the-nature-dataset/wiki/ReplicabilityOfTheStudy.

III. RESULTS

In this section, we present and analyze the results of our experiment in face of the questions raised during the experimental design. We address each question separately in both quantitative and qualitative perspectives. The quantitative analysis is based on the interpretation of the collected data whereas the qualitative analysis is derived from manual inspection of the repositories and the developers' public mailing list.

A. Addressing RQ1: How does the gap between code and architecture evolve over time?

In order to answer `RQ1`, our first step was to identify and count the amount introduced and solved violations per version. Figure 1 shows the data collected for the four selected subjects. Each point in the figure represents either the number of introduced violations or the number of solved violations (vertical axis) for a given version (horizontal axis).

The second step was to observe how the architectural debt behaves over time. Given a software version, we define architectural debt as the difference between the amount of solved and introduced violations. Figure 2 shows, as vertical bars, the architectural debt for each version. In addition, the line represents the cumulative architectural debt as software evolves.

Considering the amount of introduced violations per version shown in Figure 1, we can observe, in most versions,

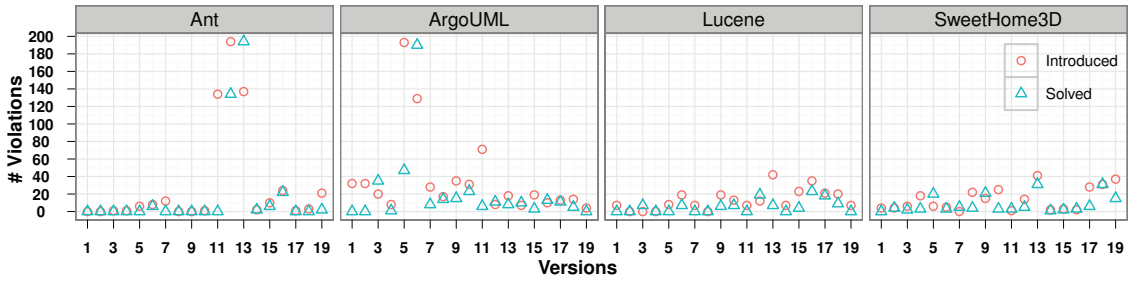

Figure 1. Introduced and Solved Violations per Version

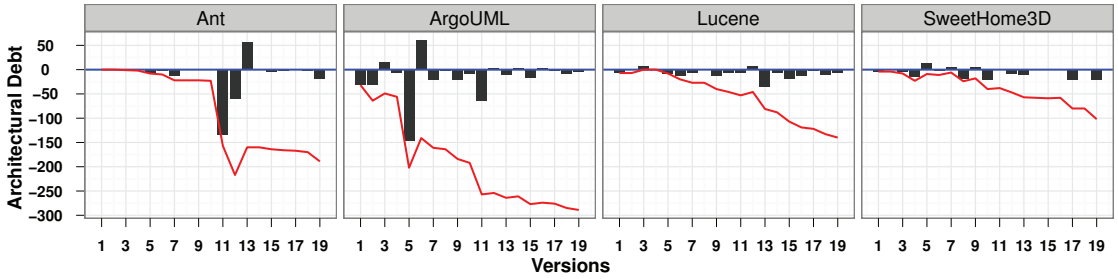

Figure 2. Architectural debt per version. The line represents the Cumulative Architectural Debt.

that few violations were introduced. The same occurs with the amount of solved violations. It is worth noting that both for Ant and ArgoUML, the data collected shows that only a few versions introduce and solve a large number of violations. One can also notice that this occurs in consecutive versions, meaning that violations introduced in one version are usually solved in the following version. As a consequence, the cumulative architectural debt increases in one version followed by a reduction in the following version (see versions 11, 12, and 13 for Ant, and versions 5 and 6 for ArgoUML in Figure 2).

1) Analysis: To conduct the qualitative analysis, we performed a manual inspection focused on adjacent versions in the repository. We conducted a more detailed inspection in versions with a large number of introduced and removed violations. For instance, versions 11, 12, and 13 for Ant, versions 5 and 6 for ArgoUML, and version 13 for both Lucene and SweetHome3D.

In fact, the qualitative analysis revealed a major restructuring period in Ant and ArgoUML. In ArgoUML, the analysis of the discussions between the developers during this period was quite enlightening. First, we detected that one developer performed a major change and communicated it to the rest of the development team, as we can see by his transcribed message: "*Yesterday I removed the old directory* (`org.argouml.uml.profile`) *and modified Argo code to use the code in the new directory* (`org.argouml.profile`)." This important change in the code introduced several violations since new classes with forbidden relationships were added into the (`org.argouml.profile`) package.

As Figure 1 shows, a significant number of architectural violations were solved in version 6 of ArgoUML. Our qualitative analysis revealed that two major changes in the code were responsible for this. Again, analyzing the mailing list and the commit messages (revision 12,455), we first found that one developer moved a class to its correct module, as can be seen by his transcribed message: "*ProgressMonitor does not belong in the GUI subsystem...Move the ProgressMonitor into its own subsystem.*" This change, combined with the removal of a cyclic dependence between two modules of ArgoUML (revision 12,407), decreased the cumulative architectural debt.

At last, we found a message from an important ArgoUML developer that summarizes the whole period of restructuring: "*I think it is time to start planning a 0.25.4 release to get all this together. I hope you agree.*"

Looking at Ant data shown in Figure 1 we identified a major restructuring period during versions 11, 12, and 13. An interesting fact that first caught our attention is that the number of solved violations in version 12 is identical to the introduced in version 11 (its previous version). Analyzing the repository and performing the diff between versions 10, 11, 12, and 13, we could visualize and understand

what happened during this period. The restructuring was conducted as follows:

- 10 - 11: Class `FileUtils$3` was added to the project. This explains the large number of violations introduced.
- 11 - 12: The same class (`FileUtils$3`) was removed from the project and its code was moved to another class. This explains the identical number of introduced and solved violations between these two consecutive versions and the large number of introduced violations as well.
- 12 - 13: Class `ProjectHelper` was restructured. The change comprised splitting its code in six other classes, which explains the large number of introduced and solved violations.

Considering Lucene, various classes were added to the system in version 13 (revision 1,048,879). To be more precise, 26 classes were added to `util.automaton.fst` package, which is part of the `util` module. These classes have forbidden method calls to the `store` module. As a result, 42 architectural violations were introduced in the system.

Finally, in SweetHome3D, three classes were removed and their code were moved to `AppletApplicationClass`, which explains the number of introduced and solved violations in version 13 (revision 2,210).

Another important aspect to point out is that, for all systems, developers usually perform perfective maintenance that aims to solve architectural violations. Moreover, when we analyze in each version the difference between solved and introduced violations, the numbers suggest that, in most cases, the problem of architectural deviation is feasible to solve. Figure 3 shows the boxplot of the absolute value of the difference between the number of solved and introduced violations per version. As we can see, in all systems, tackling up to the third quartile of violations seems to be feasible in a period of two weeks. However, as software evolves and the problem is not properly faced, the cumulative architectural debt, shown by the line in Figure 2, grows and the code tends to increasingly diverge from the intended architecture.[5]

B. Addressing RQ2: Are the violations equally spread over the design entities or they concentrate on a few ones?

To answer `RQ2`, we collected data considering different relations between classes and violations. First, we investigated the ratio between classes with violations and the total of classes. Figure 4 shows the amount of classes with and without violations per system in version 1. It is important to mention that, in our experiment, we also looked at the other versions and found that the mean of classes with violations

[5]It is important to say that, due to the few restructuring moments, this function is not monotonically decreasing.

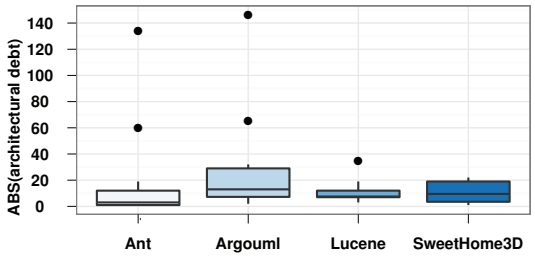

Figure 3. Quantiles for the architectural debt

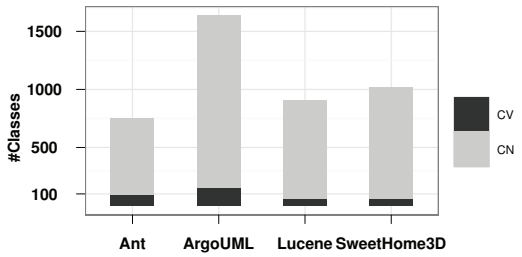

Figure 4. Distribution of violations per system. CV = classes with violations and CN = classes with no violations.

over time is 11%, 9%, 6% and 5% for Ant, ArgoUML, Lucene and SweetHome3D, respectively. Second, we analyzed the distribution of violations per class. Figure 5 shows the histogram of classes per violation. In each plot, the horizontal axis represents the amount of violations whereas the vertical axis stands for the frequency of classes. As we can see, the heavy-tailed distribution means that few classes have many violations, while most classes contain very few violations. Again, these data regard only version 1 of each system. However, we have considered all the versions and found no significant variation among them.

The histograms suggest that the classes in the distribution tail are responsible for most of the violations. Figure 6 shows the cumulative proportion of violations per class, ordered by classes with most violations. Each curve in each plot represents one version. Figure 6 confirms the idea that few classes are responsible for most violations. Moreover, this behavior repeats in time, since the curves are very close to each other.

One last important aspect analyzed to answer `RQ2` is shown in Table II. For each system, the table presents: i) the mean proportion of violations caused by the Top-10 classes; ii) the number of different classes that appeared at least once in the Top-10 classes in the studied period (DC); iii) the mean ratio between DC and the total number of classes.

1) Analysis: The distribution of violations among classes revealed different issues. First, analysis of Figure 4 suggests that a small proportion of the whole system is responsible for the architectural violations. In fact, in the worst scenario

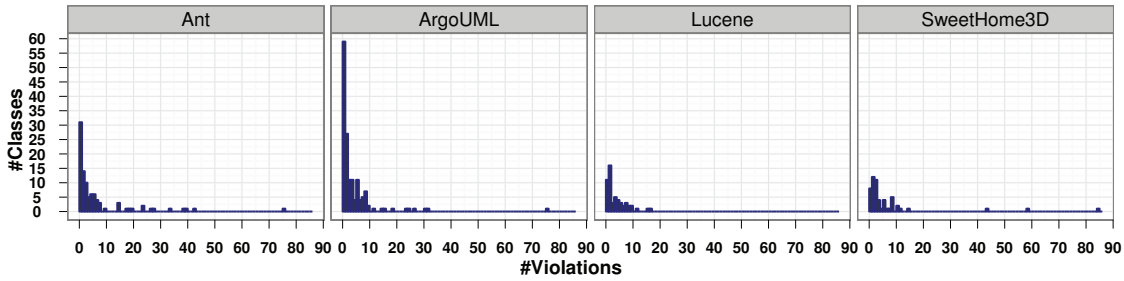

Figure 5. Frequency of classes per amount of violations (Version 1)

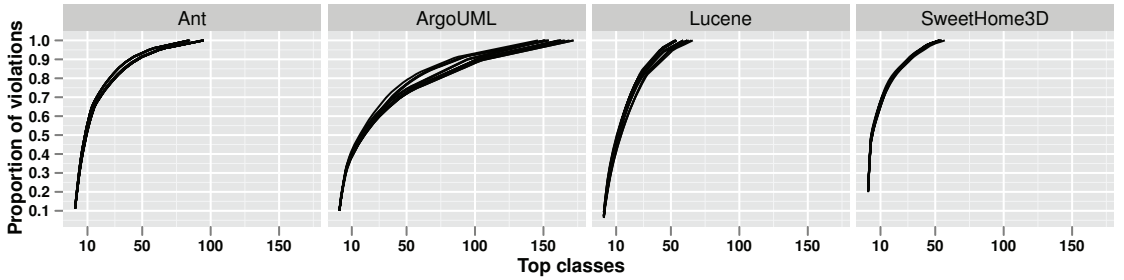

Figure 6. Distribution of violations per class

Table II
TOP-10 DATA. DC = NUMBER OF DIFFERENT CLASSES IN TOP-10 DURING THE STUDIED PERIOD.

Subject	Top-10 proportion	DC	DC / Total
Ant	54%	12	1.8%
ArgoUML	40%	17	2.7%
Lucene	45%	16	6.2%
SweetHome3D	66%	12	2.9%

(ArgoUML), at most 11% of the classes contain forbidden architectural relationships. Still, this proportion means, in this case, that 186 classes are responsible for architectural violations, which leads us to believe that such a large number of classes inhibits developers to cope with the problem. For this reason it is important to analyze the distribution of violations inside the classes with violations. Analysis reveals a heavy-tailed distribution shown in Figure 5, which leads to the conclusion that very few classes contain a large number of violations. Therefore, the analysis of the violations inside the classes suggests the existence of a small number of classes responsible for a large proportion of the violations, since few classes in the distribution tail are responsible for a large proportion of the violations (Figure 6).

One important concern during the experiment was to restrict the analysis to a smaller group of Top-10 violating classes. If the Top-10 classes do not vary much in time, maintaining conformance might be easier, since the scope of architectural problems would be confined to a small number of classes. Confirming our intuition, Table II shows that the number of different classes (DC) that appear at least once in the Top-10 group is small. Moreover, it represents a rather small proportion of the total number of classes. In summary, the critical core comprises a small number of classes and does not change much in time.

Results from the qualitative analysis point that a core of classes is critical not only for the large number of violations that they contain, but also for two aspects: i) their role in the architecture; and ii) how restructuring changes in the core have a high impact on the amount of solved relations.

In this context, regarding the role of the critical core, we found, for ArgoUML and SweetHome3D, that classes inside the `gui` and `swing` modules correspond to the majority of Top-10 classes. For example, we found that six classes in ArgoUML Top-10 group are responsible for the graphical interface. In fact, the class with most violations in ArgoUML is `ProjectBrowser`, which is part of the `gui` component. The large number of violations occurs because `ProjectBrowser` is a presentation class and therefore needs information about various business logic objects, which leads to non-allowed coupling with various classes.

Another valuable information that reinforces our observation about the critical core is that, although we have analyzed ArgoUML data for 2007, since 2002 the `ProjectBrowser` class is an architectural concern to the

development team, as the discussion below shows:

1) Developer A: "I just refactored `ProjectBrowser` to take out the construction of the Menubar. Are there any objections against this?" (2002-10-10)
2) Developer B: "Take out the references to the project. Let the project be managed by another singleton class. Decouple Main and `ProjectBrowser`." (2002-10-12)
3) Developer C: "Refactor suggestion for `ProjectBrowser`: Take out anything to do with current themes and place this in its own singleton class." (2002-10-12)

Due to their application domain, Lucene and Ant do not have graphical interface modules. However, analyzing violations in these two projects, we found that the `util` module of both applications is critical. That happens because an util abstraction receives objects from many different classes to perform its actions.

Regarding changes in the critical core, we first hypothesized that the number of solved relations is highly impacted by corrective changes in the Top-10 classes. Then, we analyzed peaks of solved violations and identified these changed classes. Not surprisingly, for all the systems, peaks of solved violations were caused by changes in the classes on the distribution tail. In other words, by changes in the critical core.

C. Addressing RQ3: Once violations are solved in a given version, do they appear again in future versions?

In order to answer `RQ3`, we identified the number of recurring violations (RV) and their proportion over the total of solved violations (RV / Solved). Besides that, we counted the amount of times that a violation reappears in the system (degree of recurrence). After that, we identified the statistical modal value of the degree of recurrence (MDR) considering the recurring violations. We use the modal value instead of mean or median because it is the most representative descriptive statistic of the data. Table III summarizes the data collected for each system.

Table III
RECURRING VIOLATIONS DATA. RV = #RECURRING VIOLATIONS AND MDR = THE MODAL VALUE OF THE DEGREE OF RECURRENCE.

Subject	RV	Total Solved	RV/Solved	MDR
Ant	343	366	94%	1
ArgoUML	44	400	11%	1
Lucene	21	107	20%	1
SweetHome3D	37	162	23%	4

1) Analysis: Our quantitative analysis revealed that all the systems have recurring violations. The high number of recurring violations for Ant is explained by the rollback occurred during versions 11 and 12, as we have previously identified. Yet, recurring violations represent a meaningful number when compared to the total of solved violations.

For all systems, analyzing the recurring violations' lifecycle, we found that a small number of violations were not definitely solved during the studied period. For example, for Ant, 9 of the 343 violations were not solved. It is worth saying that, due to the limitation in our timeline, we cannot assure that these violations were definitely removed from the system.

It is worth noting that, except for SweetHome3D, the modal value of degree of recurrence (MDR) is 1. It means that most of the recurring violations were solved and appeared only once again.

One common approach to regard a problem as relevant is to identify whether it was addressed by the development team in earlier versions [16]. What is clear when analyzing recurring data is that a meaningful proportion of architectural violations that were solved ealier, i.e., relevant violations, are likely to reappear in the future.

It is not simple to assert why violations reappear over time. For example, analyzing Ant data, we observed that one of the factors that might cause this is a mistaken restructuring activity followed by a series of correction steps. Besides that, another aspect to take into account is the degree of knowledge of a developer in a specific area of the code. Unfortunately, due to limitation of our data, it is not possible to identify the authors who were responsible for the introduced violations.

In summary, the number of recurring violations suggests that the problem exists and it cannot be ignored during the software evolution. This kind of behavior may indicate, for example, mistaken restructuring changes and the lack of architectural awareness by some developers [17].

IV. DISCUSSION

A. Do not live with broken windows

Through this exploratory study, we could gather some interesting insights about architectural drift. In particular, although our data does not empirically demonstrate such a conclusion, we believe that architectural drift seems to be related to the "Do not live with broken windows" [18] principle. Hunt and Thomas used this metaphor to highlight the importance of not letting small problems unrepaired in the code. In the context of architectural debt, our results suggest that the gap between code and architecture is tractable when violations are checked and then solved in a short period (e.g., bi-weekly). However, as software evolves and small problems are not properly faced, tackling architectural drift can become unfeasible, as can be seen in Table IV. The table shows the total number of violations in version 19. This number regards the violations introduced and not solved during all the systems' lifecycle. That is to say, it represents the overall gap, not only the one observed during our studied timeline.

Table IV
TOTAL NUMBER OF VIOLATIONS IN VERSION 19

Subject	#Violations
Ant	637
ArgoUML	641
Lucene	305
SweetHome3D	429

B. Human factors

Qualitative analysis performed in this study revealed valuable information about: i) how developers deal with architectural issues; and ii) how changes in the code impact architectural drift. Although quantitative analysis reveals important facts about architecture erosion, we found that other sources of information improve our understanding about it. For example, developers' mailing list records gave us detailed descriptions of architectural discussions and decisions. In summary, analyzing architectural issues involves observing not only source code and models, but also the human factors involved.

C. Critical core first

Changes in the critical core can produce great impact on architectural debt. For instance, Figure 7 shows one of the peaks of solved violations in ArgoUML. As we can see, the number of violations of the classes in the distribution tail (critical core) had significantly decreased, i.e., they were moved to the left in the distribution.

Based on our results, we can state that by addressing the critical core, developers can concentrate on the largest part of the violations, while having to deal with a small number of entities. However, it is important to make clear that this does not imply that less work has to be done.

D. Recurring violations

In our study, we also found that some violations are solved, but reappear in future versions of the system. Analyzing this issue helps to reveal recurring architecture problems. These problems may be caused by several reasons, of which we highlight two: lack of architectural awareness [17] and lack of conceptual integrity [19]. In a nutshell, the former refers to the awareness of a developer about several aspects, including architectural decisions, while the latter refers to the uniformity of a mental model that the developers have about the architecture. We put these two terms in perspective because, if the recurrence was caused by the same developer that solved it earlier, it may suggest that this developer is losing architectural awareness over time. On the other hand, if its recurrence was caused by a different developer, it may suggest lack of conceptual integrity.

V. THREATS TO VALIDITY

We must state some aspects that might influence our observations. In this context, the main threat is the architecture

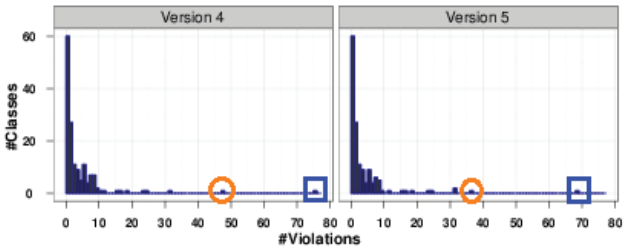

Figure 7. Peak of ArgoUML solved violations. Classes in the distribution tail were restructured.

module views that we have extracted from the subjects. Despite the fact that they were based on systems' architectural documentation, we still have to validate them with the architects and developers. However, in our qualitative analysis of the peaks of introduced and solved violations, we manually inspected two sources of information to support our quantitative data: the commits and discussions in the developers' mailing list. Through this analysis, we did not find inconsistencies between our observations and what really happened in the systems, which leads us to believe that the architectural module views seem to be consistent.

Another important aspect regarding the architecture module views is that we assume that the architectural decisions remain stable during the studied period. This might affect our observations because a change in the module view of a system during the period of our experiment could generate different results. Nevertheless, as far as we could observe, the architectural decisions remained stable for all the studied systems.

It is also worth mentioning the renaming problem, i.e., given that a violation *id* is based on entity names, if this name changes, we consider the violation as solved. We performed a qualitative analysis of the amount of renamed entities and found that, in only one of the subjects, namely ArgoUML, one of the top ten classes was renamed. Regardless of this exception, our approach was still correct in considering the violation solved, because the renaming was caused by moving the class to its appropriate package. Nonetheless, we do realize that this is a particular case. Hence, we understand that further analysis is required, to account for name changes.

Finally, results cannot be generalized to contexts different from the subject systems. We tried to reduce external validity threats, though, by choosing popular and long-life systems in an industrial-strength language (Java).

VI. RELATED WORK

Lehman [20] have built an initial body of knowledge on software evolution establishing, among other concepts, the so called Lehman's laws [21]. In this context, several studies have been performed to analyze software in an evolutionary

perspective. Godfrey and Tu [22], for example, investigated the growth of the Linux operating system kernel over time. The authors examined 96 kernel versions measuring their size in terms of the distribution package, LOC, number of functions, variables and Macros. As an important observation, the authors found that all measures revealed that development releases grow at a super-linear rate over time, contrasting Lehman's hypothesis of an inverse square growth rate [23], [24]. Gall et al. [25] performed a similar work on a large telecom switching system (TSS). As a major result, the authors found divergences between the whole system growth and its subsystems.

Some of the works in the software evolution area specifically focus on the architectural evolution over time. For example, van Gurp et al. [26] analyzed two case studies in order to investigate the common causes for design erosion, how the stakeholders identify this scenario and what are the common activities performed to address design erosion.

Hassaine et al. [27] proposed a quantitative approach to study the evolution of the architecture of object-oriented systems. The authors conceived a representation of an architecture based on classes and their relationships and used this representation to measure architectural decay by comparing with a subsequent program architecture. Hassaine and colleagues use the term architectural decay to refer to the deviation of the actual architecture from the original design. This work is related to ours because we both analyze architectural decay/drift over the time. However, instead of assuming that the first version of a system is the intended architecture, we use explicit architectural models extracted from the systems' documentation, which reveals that the inconsistencies are in fact architectural violations.

Another work closely related to ours is a case study performed by Rosik et al. [4]. The researchers assessed the architectural drift during the *de novo, in vivo* development of a commercial system, named DAP. The authors describe their experience in applying conformance checking during software evolution and developers' actions in face of the feedback given by the process of conformance checking. As a result, they identified that the analyzed system diverged from the intended architecture and that developers tend to keep a number of violations unsolved. According to the authors, in most cases this is due to the risk of the changes, which comprise a number of restructuring activities. The work of Rosik et al. is similar to ours in the sense that it aims to analyze the evolution of the gap between code and architecture. However, it is important to clarify some differences. First, we have analyzed four mature and architecturally stable systems, while they performed architectural conformance checking in one system since the beginning of its development. Moreover, our focus was not restricted only to architectural drift, but we also observed the location of the architectural violations and their lifecycle to respectively identify critical cores and recurring architectural problems. On the other hand, our results corroborate with theirs in that implementation of a system tends to diverge from its intended architecture.

Wermelinger et al. [28] proposed an architectural evolution assessment framework based on quality metrics, laws, principles, and guidelines to address important questions about the architecture behavior over time. They focus on assessing architecture by analyzing quality principles, such as the Acyclic Dependency Principle and the Open Close Principle. We, however, did not evaluate the architecture through quality metrics. We were more concerned on how far it is from the intended one.

VII. Summary and Future Work

This paper addressed the lack of knowledge on the evolutionary nature of architectural violations. We focused our effort on investigating the architectural drift over time, the location of the violations and their lifecycle. In particular, we addressed in this paper three main research questions:

- How does the gap between code and architecture evolve over time?
- Are the violations equally spread over the design entities or they concentrate on a few ones?
- Once violations are solved in a given version, do they appear again in future versions?

In order to provide answers to the aforementioned questions, we conducted a longitudinal and exploratory study. We performed conformance checking on four widely known open-source systems for which we have architectural models. In total, we analyzed more than 3,000 violations. From our quantitative and qualitative analysis, we observed that, in face of the questions raised during the study design: i) the number of architectural violations, in the long term, is continuously growing; ii) in all studied systems there is a critical core and this core does not change much over time; and iii) some violations are solved, but reappear over time.

As future work, we intend to use the initial body of knowledge produced in this work to improve architectural maintenance and evolution tasks. For example, future work in this area includes the automated identification of critical cores.

We also intend to perform a study on the relevance of architectural violations detected by conformance checking. As can be seen in this paper, a number of violations are detected and are not solved over time. Therefore, further research could focus on the reasons that make developers ignore such violations.

Another research track is to perform statistical studies in order to correlate the amount of introduced and solved violations with a number of other variables. For example, we are interested in finding out whether peaks of solved violations cause a positive impact in code quality metrics, such as coupling, cohesion, and instability.

ACKNOWLEDGEMENTS

Special thanks go to Marco Tulio Valente, Ricardo Terra, Gail Murphy, and the reviewers for the fruitful discussions and comments about this work. We also would like to thank Tercio de Melo for the technical support on scripts development. This work has been supported by National Council for the Improvement of Higher Education (CAPES) and ePol/SETEC/DPF.

REFERENCES

[1] J. Van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of systems and software*, vol. 61, no. 2, pp. 105–119, 2002.

[2] D. Perry and A. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[3] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Software Engineering, IEEE Transactions on*, 2001.

[4] J. Rosik, A. Le Gear, J. Buckley, M. Babar, and D. Connolly, "Assessing architectural drift in commercial software development: a case study," *Software: Practice and Experience*, vol. 41, no. 1, pp. 63–86, 2011.

[5] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *ACM SIGSOFT Software Engineering Notes*, no. 4, 1995.

[6] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, 2002.

[7] J. Brunet, D. Serey, and J. Figueiredo, "Structural conformance checking with design tests: An evaluation of usability and scalability," in *Proceedings of 27th IEEE International Conference on Software Maintenance*, 2011.

[8] M. Sefika, A. Sane, and R. Campbell, "Monitoring compliance of a software system with its high-level design models," in *Proceedings of ICSE*, 1996.

[9] R. Tesoriero Tvedt, P. Costa, and M. Lindvall, "Evaluating software architectures," *Advances in Computers*, vol. 61, 2004.

[10] R. Terra and M. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 39, no. 12, 2009.

[11] J. Knodel, M. Lindvall, D. Muthig, and M. Naab, "Static evaluation of software architectures," in *Proceedings of CSMR*, 2006.

[12] P. Lam and M. Rinard, "A type system and analysis for the automatic extraction and enforcement of design information," *ECOOP 2003–Object-Oriented Programming*, pp. 235–268, 2003.

[13] M. Feilkas, D. Ratiu, and E. Jurgens, "The loss of architectural knowledge during system evolution: An industrial case study," in *Proceedings of ICPC'09*. IEEE, 2009.

[14] G. Murphy and D. Notkin, "Reengineering with reflexion models: A case study," *Computer*, vol. 30, no. 8, 1997.

[15] R. A. Bittencourt, "Enabling Static Architecture Conformance Checking of Evolving Software," Ph.D. dissertation, 2012.

[16] J. Araujo, S. Souza, and M. Valente, "Study on the relevance of the warnings reported by java bug-finding tools," *Software, IET*, vol. 5, no. 4, pp. 366–374, 2011.

[17] H. Unphon and Y. Dittrich, "Software architecture awareness in long-term software product evolution," *Journal of Systems and Software*, vol. 83, no. 11, pp. 2211–2226, 2010.

[18] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[19] F. Brooks, "The mythical man-month," *Reading, MA: Addison-Westey*, 1977.

[20] M. Lehman, "Program evolution," *Information Processing & Management*, 1984.

[21] M. Lehmann, "Laws of software evolution revisited," in *Software process technology*, 1998.

[22] M. Godfrey and Q. Tu, "Evolution in open source software: a case study," in *Software Maintenance, International Conference on*, 2000.

[23] M. Lehman, D. Perry, and J. Ramil, "Implications of evolution metrics on software maintenance," in *Software Maintenance, International Conference on*, 1998.

[24] W. Turski, "Reference model for smooth growth of software systems," *IEEE Transactions on Software Engineering*, 1996.

[25] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth, "Software evolution observations based on product release history," in *Software Maintenance, International Conference on*. IEEE, 1997.

[26] J. van Gurp, S. Brinkkemper, and J. Bosch, "Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles," *J. Softw. Maint. Evol.*, 2005.

[27] S. Hassaine, Y. Guéhéneuc, S. Hamel, and G. Antoniol, "ADvISE: Architectural decay in software evolution," in *Proceeding of the 16th European Conference on Software Maintenance and Reengineering*, 2012.

[28] M. Wermelinger, Y. Yu, A. Lozano, and A. Capiluppi, "Assessing architectural evolution: a case study," *Empirical Software Engineering*, pp. 1–44, 2011.