Developers' Perspectives on Architecture Violations: A Survey

Ajay Bandi
School of Computer Science and Information Systems
Northwest Missouri State University
Maryville, MO, 64468, USA
ajay@nwmissouri.edu

Abstract

Architecture violations are indicators of code decay. Software developers have to follow the original or planned architecture. However, several systems often require redesign or reengineering. This paper aims to understand the knowledge of software developers architecture violations and anti-design patterns. I used an exploratory survey to collect responses from 30 professional software developers. Analysis of the responses found 25% of respondents were not aware of about architecture violations while 36% were aware of them but were not sure of their significance. Based on the responses, I also identified five categories of developers' perspectives on architecture violations and design anti-patterns: non-adherence to original architecture, lack of software quality, poor design decisions, lack of developer's skills, and cost-benefit considerations.

keywords: architecture constraints, architecture violations, survey, reverse engineering, software evolution

1 Introduction

Properly implemented, code should follow specified architectural constraints and conform to the conceptual architecture. However, architectural violations are often due to new interactions between modules that were originally unintended in the design [11, 15, 17, 18]. Such violations may be caused by adding new functionality, by modifying existing functionality to implement changing requirements, or by repairing defects. When such changes are inconsistent with the planned architecture and design principles, the system becomes more complex, hard to maintain, and defect-prone [6, 13, 20]. Often, redesign or reengineering of the whole system is the only practical solution for this problem [9]. Such gradual increase in software complexity due to unintended interactions between hard-to-maintain modules is called "architectural degeneration" and "code decay" [3, 4, 6, 10]. Research shows that violations of architecture and design rules cause code to decay [6, 9, 11]. My research focuses on exploring the developer's awareness of architecture violations or design anti-patterns.

The remainder of this paper is organized as follows. Section 2 presents the background on architectural violations. Section 3 details the methodology of my survey. Section 4 presents the preliminary results and analysis of my survey. Section 5 presents conclusions and future work.

2 Background on Architecture Violations

This section presents part of the related work section from Bandi's dissertation [2]. Software systems often refer to an architectural model and are organized into several subsystems and modules that follow some design rules. These design rules constitute the constraints on architectural styles and software design patterns. Developers may violate these constraints from one version to another. This is the starting point of architecture degeneration that causes code decay and makes maintenance difficult. Managing architectural violations for each version during software development and maintenance can prevent architectural degeneration. Below are a couple of examples that show the violations of design rules in architectural styles and design patterns.

Figure 1 shows a simple Layered architecture and modules within those layers. Following are some of the constraints imposed on the layered architecture [5, 12].

- Layer dependencies are not transitive. If layer A is allowed to use layer B, and layer B is allowed to use layer C, it does not automatically follow that layer A can access layer C.
- Each module within a layer is allowed to access other modules within the layer.
- If one layer accesses another layer, all modules

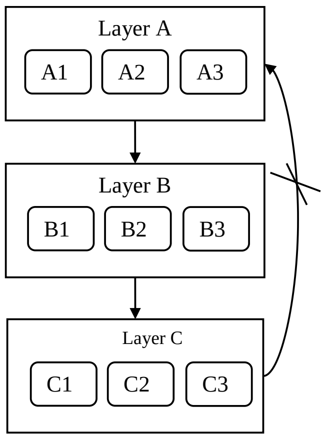

Figure 1: Violations in Layered architecture style

defined with public visibility in the accessed layer are visible within the accessing layer.

A violation can occur when a developer attempts to allow a module from Layer C to access data from Layer A, which is not defined in our rules (marked with an "X" in Figure 1). This small violation represents an initial sign of architectural degeneration.

The Mediator design pattern is often used when interactions among objects are unstructured, complex, and their reuse is difficult [5, 8]. Figure 2 shows an example of a Mediator pattern and potential violations (marked as "X"). If tasked to implement new functionality for aCheckbox, a developer might complete the task using several interactions between the other colleagues (e.g., aListBox, aButton, anEntryField), but these interactions would violate the rules of the Mediator design pattern. This implementation results in tight coupling between colleagues and makes it more difficult to understand the architecture of the system. A correct implementation is marked with a dashed line in Figure 2. All these constraints can be represented using can-use/cannot-use phrases. My research's main goal is to understand the developer's knowledge of these violations.

3 Methodology

In order to investigate the developers' awareness of architectural violations and anti-patterns, I chose to use the survey method [7, 14, 19]. I used exploratory studies of code smells [20] as a model for my study. My study focuses on two research questions (RQ):

- RQ1: Are software engineering practitioners familiar with architecture/design violations?

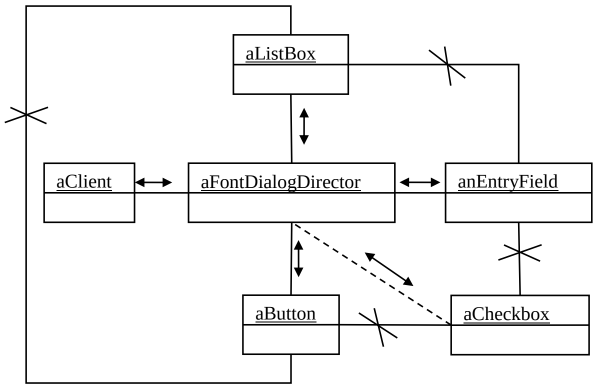

Figure 2: Violations in Mediator design pattern

- RQ2: How aware are software engineering practitioners of architecture/design violations?

I prepared survey questions consisting of background information on the respondents, including predominant role, expertise in programming languages, familiarity with programming-languages paradigms, and software development experience in the number of lines of source code. A five-point Likert scale questions their familiarity with architectural violations and design anti-patterns; the sources from which developers learn about these concepts; and how concerned they are about the presence of architecture violations in their software systems; and the specific design patterns that developers focused on during the implementation, detection, and refactoring of those patterns or architecture constraints. The complete survey is not included in this paper because of space limitations. The survey was prepared using Survey Monkey and can be found in the website [1]. The survey was forwarded to the members of the Professional Advisory Team (PAT) of the School of Computer Science and Information Systems at Northwest Missouri State University. The PAT members then forwarded the survey to their software development teams. Upon completion of the survey, I exported the data from Survey Monkey to be used for my analysis.

4 Results and Analysis

This section presents the results and analysis of the data I collected in the survey.

Table 1: Predominant role of the respondents

Category	Number	Percentage
Developer	15	50%
Architect	11	36.67%
Tester	2	6.67%
Project Manager	1	3.33%
Self-employed	1	3.33%
Total	30	100%

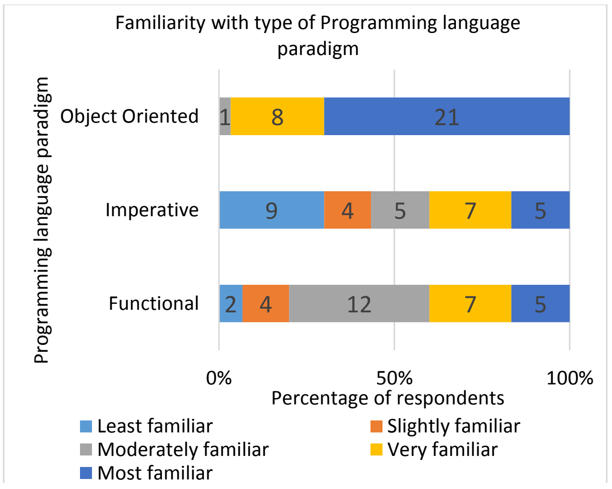

Figure 3: Respondents familiarity with the type of programming language paradigm

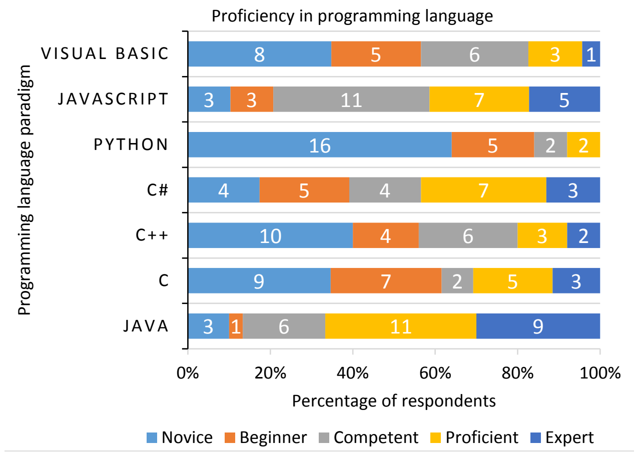

Figure 4: Respondents Proficiency in programming languages

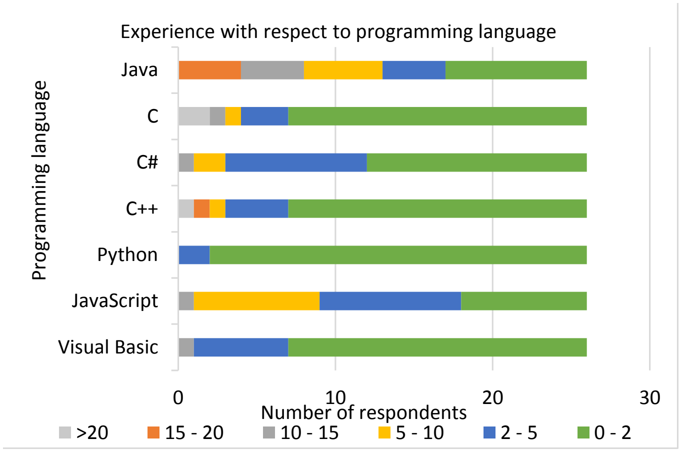

Figure 5: Experience with respect to programming language

4.1 Background and skills of respondents

Out of 45 respondents, 30 respondents fully completed the survey, yielding a response rate of 67%. All the respondents are software developers from the Mid-west region of the United States. Table 1 shows a summary of the different roles of respondents and their frequency.

Figure 3 presents the self-assessed proficiency of the respondents with respect to the programming language paradigm. The majority of the respondents 21 or 70% mentioned that they are most familiar with Object-Oriented programming paradigm. Interestingly, 5 (16%) respondents mentioned that they are most familiar with both Imperative and Functional language paradigms. When compared to Imperative, a larger group of respondents are confident with a Functional programming paradigm (40% for moderately familiar, 23% for very familiar, and 16% for most familiar).

Figure 4 presents the self-assessed proficiency of the respondents with respect to the programming languages. While analyzing the data, I found 16 (64%) respondents are novices with respect to Python programming language. Nine 9 (30%) respondents mentioned that they are most familiar with Java programming language. A larger group of respondents are confident in Java (40% competent, 23% proficient, and 17% experts) and JavaScript (20% competent, 37% proficient, and 30% experts) programming languages.

Figure 5 presents the respondents' number of years of experience with respect to the programming language. The majority of the respondents have experience in JavaScript, Java, and C++ programming languages varied from 2 to 20 years of experience. If a value was not entered for the respective programming language, the number of years of experience for that respondent was valued at zero.

Figure 6 represents the work experience of respondents with respect to the programming language in terms of lines of source code (LOC). If a value was not entered for the respective programming language, the

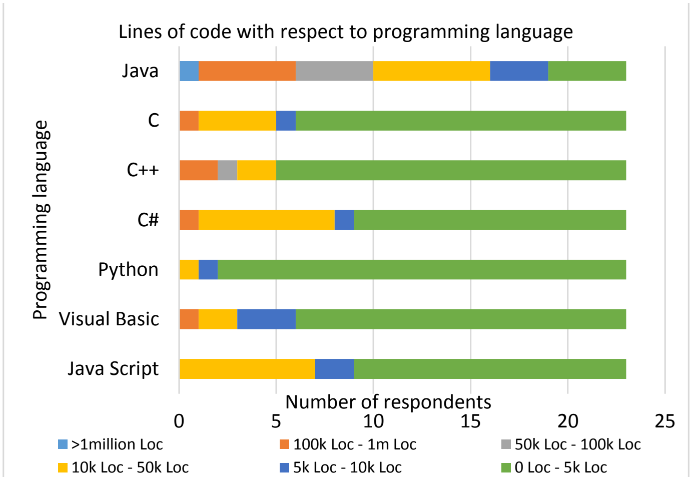

Figure 6: Lines of code with respect to the programming language

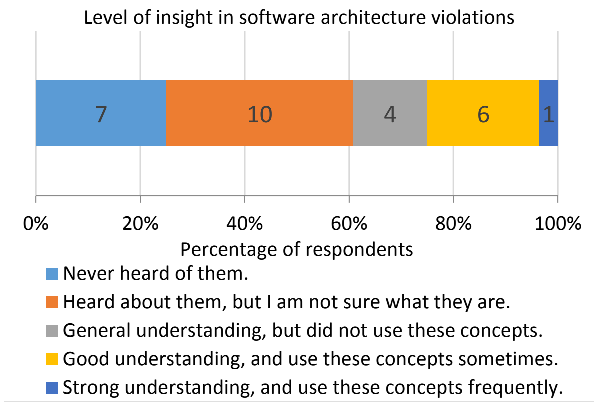

Figure 7: Level of insight of architectural violations

years of work experience for the respondent was valued at zero. Two respondents answered "Other" category of programming language, one respondent with four years' experience working with 8kLOC in PeopleSoft, and another respondent with one year's experience working with 500kLOC in Objective-C programming language.

4.2 Level of insight in software architecture violations

This section addresses our first research question. RQ1: Are software engineering practitioners familiar with architecture/design violations?

From the total set of respondents, 7 (25%) answered that they have never heard about the architecture violations or design anti-patterns. In the remaining answers, 10 (36%) responded that they had heard about architecture violations or design anti-patterns, but the respondents were not sure what these terms mean. A total of 4 (16%) respondents replied that they have a general understanding but did not use these concepts. A total of 6 (21%) respondents answered that they have a good understanding of these concepts and use them sometimes. Only one respondent (4%) answered that they have a strong understanding of these concepts and use them frequently in writing software. These findings suggest that there is a need for research in the software industry about architectural violations and design anti-patterns.

4.3 Awareness of software architecture violations

This section addresses our second research question. RQ2: How aware are software engineering practitioners of architecture/design violations?

We used coding techniques in ground theory [16] to synthesize the survey responses. Table 2 shows a few examples of what the respondents mean by architecture or design violations with codes. The technical report has the complete list of statements along with the codes. In total, we have the following five categories with respective codes.

(1) *Non-adherence to original or planned architecture:* This category relates to the comparison of the designed or conceptual architecture and the implemented architecture of the system. Codes belonging to this category are 'conflict,' 'contradict,' and 'not follow' the architecture rules.

(2) *Lack of software quality:* This category covers the rationale in connection to adding functionality, repairing defects, and modifying the existing software. Codes fitting this category are 'ease of maintenance,' 'flexibility,' 'durability,' and 'security' of the later software versions.

(3) *Poor design decisions:* This category relates to the architect or project manager's decisions during the architecture and design stages. Codes related to this category are 'decision' and 'choice.'

(4) *Lack of developers' skills:* This category relates to the respondent's knowledge of the design principles. Codes belonging to this category are 'lack of knowledge,' 'poor abstraction,' and 'poor documentation.'

(5) *Cost-benefit considerations:* This category consists of statements relating to deadlines, costs, and release dates. Codes fitting this category are 'time pressure,' 'deadlines,' and 'no/less budget.'

Table 2: Predominant role of the respondents

ID	Statement	Code
2	Software implementations that contradict architecture principles or good design principles.	Contradict
5	Architecture/Design violations are code, algorithms, or whole applications that are not designed according to established patterns and design principles that enable efficient operation or ease of maintenance for the application.	Ease of maintenance
12	Decisions which negatively impact non-functionals or contradict the prescribed architecture or design	Decisions
16	One of a couple things. First, code that is structured poorly or doesn't use good abstraction. Second, the application of software patterns in situations where they are not warranted.	Poor abstraction
25	Spaghetti code due to time pressure on a project with no budget to refactor. Asp classic templates and SQL server dts package templates still being reused because no budget to modernize them.	Time pressure, no budget

5 Conclusion and Future Work

In this paper, I reported the preliminary results of my survey on the perspective of software developers on architecture or design violations. I analyzed the data from 30 respondents (software developers) on their background and their level of insight about software architectural violations. The data showed that 25% of the respondents had no awareness of architecture or design violations, and 36% of the respondents had heard about them but were not sure what the terms mean. These findings suggest that there is need to increase awareness of architectural violations and design anti-patterns in the software industry.

I also analyzed data on how the software developers understand the concepts of architecture or design violations. Based on the statements provided by the respondents, I identified five categories of perceived violations, including non-adherence to original or planned architecture, lack of software quality, poor design decisions, lack of developers' skills, and cost-benefit considerations.

In the future, I will analyze and present the results of the data about the level of concern of the architecture violations in the source code; the sources from which the developers learn about the concepts, detection, and refactoring of architecture violations; and the immediate need for developers to take courses in software architecture violations and design anti-patterns.

Acknowledgments

I thank all the respondents of this survey. I also thank Dr. Edward B. Allen and Dr. Tanmay Bhowmik for their valuable suggestions to this survey. My sincere thanks to Ms. Giuli Coniglio for proofreading and editing this paper.

References

[1] Ajay Bandi. *Architecture/design violations survey.* `https://www.surveymonkey.com/r/ArchitectureViolationsSurvey`.

[2] Ajay Bandi. *Assessing code decay by detecting architectural violations.* PhD thesis, Mississippi State University, December 2014.

[3] Ajay Bandi, Edward B. Allen, and Byron J. Williams. Assessing code decay: A data-driven approach. In *Proceedings of ISCA 24th International Conference on Software Engineering and Data Engineering.*, 2015.

[4] Ajay Bandi, Byron J. Williams, and Edward B. Allen. Empirical evidence of code decay: A systematic mapping study. In *Proceedings: 20th Working Conference on Reverse Engineering*, pages 341–350.

[5] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architecture: Views and Beyond.* Addison-Wesley, Boston, MA, 2003.

[6] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[7] Arlene Fink. *The Survey Handbook.* SAGE, Calfornia, 2003.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, MA, 1994.

[9] Michael W. Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Proceedings: 2nd International Symposium on Constructing Software Engineering Tools*, 2000.

[10] L. Hochstein and M. Lindvall. Diagnosing architectural degeneration. In *Proceedings: 28th Annual NASA Goddard Software Engineering Workshop*, pages 137–142, 2003.

[11] M. Lindvall, R. Tesoriero, and P. Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *Eighth IEEE Symposium on Software Metrics*, pages 77–86, 2002.

[12] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[13] Steffen M. Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings: 3rd International Symposium on Empirical Software Engineering Measurement*, 2009.

[14] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, 2009.

[15] Santonu Sarkar, Girish Maskeri, and Shubha Ramachandran. Discovery of architectural layers and measurement of layering. *The Journal of Systems and Software*, 82(11), 2009.

[16] A. Strauss and J. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* SAGE, 1988.

[17] Adrian Trifu and Radu Marinescu. Diagnosing design problems in object oriented systems. In *Proceedings: 12th Working Conference on Reverse Engineering*, 2005.

[18] Roseanne Tesoriero Tvedt, Patricia Costa, and Mikael Lindvall. Does the code match the design? A process for architecture evaluation. In *Proceedings: International Conference on Software Maintenance*, pages 393–401, 2002.

[19] Claes Wohlin, Per Runeson, Martin Host, Magnus C. Ohlsson, Bjorn Regnell, and Anders Wesslen. *Experimentation in Software Engineering.* Springer, Berlin, 2012.

[20] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *Proceedings: IEEE International Conference on Software Maintenance*, pages 306–315, 2012.