Automatic Assessment of Architectural Anti-patterns and Code Smells in Student Software Projects

Marco De Luca
marco.deluca2@unina.it
University of Naples Federico II
Naples, Italy

Sergio Di Meglio
sergio.dimeglio@unina.it
University of Naples Federico II
Naples, Italy

Anna Rita Fasolino
fasolino@unina.it
University of Naples Federico II
Naples, Italy

Luigi Libero Lucio Starace
luigiliberolucio.starace@unina.it
University of Naples Federico II
Naples, Italy

Porfirio Tramontana
ptramont@unina.it
University of Naples Federico II
Naples, Italy

ABSTRACT

When teaching Programming and Software Engineering in Bachelor's Degree programs, the emphasis on creating functional software projects often overshadows the focus on software quality, a trend consistent with ACM curricula recommendations. Dedicated Software Engineering courses take typically place in the later stages of the curriculum, and allocate only limited time to software quality, leaving educators with the difficult task of deciding which quality aspects to prioritize. To educate students on the importance of developing high-quality code, it is important to introduce these skills as part of the assessment criteria. To this end, we have implemented a pipeline based on advanced frameworks such as ArchUnit and SonarQube. It was successfully tested on a class of students engaged in the Object Oriented Programming course, demonstrating its usefulness as a resource for educators and providing some concrete evidence of quality problems in student projects.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; **Student assessment**; • **Software and its engineering** → *Software creation and management*.

KEYWORDS

oop courses, code quality, quality criteria, architectural anti-patterns

ACM Reference Format:

Marco De Luca, Sergio Di Meglio, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. 2024. Automatic Assessment of Architectural Anti-patterns and Code Smells in Student Software Projects . In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3661167.3661290

This work has been partially funded by ENACTEST (European innovation alliance for testing education), ERASMUS+ Project number 101055874, 2022-2025 and by GATT (GAmification in Testing Teaching), funded by the University of Naples Federico II Research Funding Program (FRA)..

This work is licensed under a Creative Commons Attribution International 4.0 License.

EASE 2024, June 18–21, 2024, Salerno, Italy

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1701-7/24/06
https://doi.org/10.1145/3661167.3661290

1 INTRODUCTION

The teaching of Programming and Software Engineering in Bachelor's Degree programs, such as Computer Science and Computer Engineering, often emphasizes the ability to create functional projects rather than focusing on software quality. This approach is consistent with the ACM Computer Science and Computer Engineering curricula recommendations for Bachelor degrees [2]. According to these recommendations, software quality is only marginally addressed in typical three-year Bachelor's degree programs, with introductory CS1 courses focusing mainly on programming aspects, and some preliminary software quality concepts being introduced only later in the program, in Software Engineering courses.

Still, Software Engineering courses can allocate only limited time to delve into software quality aspects, and deciding which quality aspects to prioritize within the limited time frames is a key challenge for educators. Indeed, to date, there is a noticeable gap in the literature regarding research into the foundational software quality challenges specifically encountered by intermediate-level students when developing software projects. The scarcity of such insights leaves educators in more advanced Software Engineering courses with limited guidance on which quality aspects to prioritize within their courses. A further difficulty for teachers who want to make students more aware of the quality flaws in their projects is related to the difficulty of automatically assessing these aspects. The objective of this paper is to aim for a methodological and technological framework that provides teachers with a quick assessment of certain quality aspects of student projects.

To this aim, we have used automated state-of-the-art static analysis tools to investigate the prevalent architectural anti-patterns [14] and code smells. Our results may represent a useful resource for educators to enhance the efficacy of their teaching at an introductory level and in later stages of the curriculum. The applicability of such automation was assessed by examining 26 software projects produced by students in an Object Oriented Programming course taught by one of the authors at the University of Naples Federico II. The quality analysis of these projects revealed frequent types of issues that should be addressed by teachers of future Object Oriented Programming courses.

The remainder of the paper is structured as follows. Section 2 presents background information about software quality and related work. Section 3 presents the proposed evaluation framework, while Section 4 describes a case study we conducted involving students of an Object Oriented Programming course, with a preliminary discussion of the most common quality issues encountered. Finally, Section 5 provides conclusions and future work.

2 BACKGROUND AND RELATED WORKS

2.1 Software Quality

The concept of software quality is defined as the extent to which a software product or system adheres to predefined requirements and meets user expectations [5]. ISO 25010 [8] provides a framework that outlines various quality characteristics that should be addressed in software products, like modularity, reusability, and testability. These quality characteristics depend on two pivotal aspects: the soundness of the underlying architectural design and the quality of the source code.

2.1.1 Software Architecture Design. The quality of a software system also depends on its architecture, i.e., its organization as a collection of components, their interconnections, and constraints between their interactions [1]. A well-designed architecture lays the foundation for a robust, scalable, and maintainable system, while poor architecture can introduce a lot of challenges that impact performance, reliability, and overall user satisfaction.

Software architectural patterns provide generalized and reusable solutions to common software design problems [17]. These patterns are fundamental guidelines for structuring the components of a software system to enhance the overall quality of the software architectural design like its modularity, reusability, and scalability [3, 15]. To ensure that the implemented architecture does not deviate from the intended architectural design, architecture conformity checks (ACC) can be performed [18]. These checks can be performed automatically, leveraging existing techniques and libraries such as ArchUnit.

2.1.2 Source Code Quality. Source code quality is a concept that often lacks a universal definition, leading to diverse interpretations. A common term used to refer to code quality is *Clean Code*. The definition of *Clean Code* is provided in the work of Martin et al. [13] and refers to code that is easy to read, to write, and to maintain, and where the developer's intent is clear not just in the compiler but to humans who read it. Many *static analysis* tools have been proposed and are widely used to ensure that source code satisfies *Clean Code* principles. Lenarduzzi et al., in [11], carried out a comparative analysis of the most popular static analysis tools (Better Code Hub, CheckStyle, Coverity Scan, FindBugs, PMD, and SonarQube). Their results indicate that SonarQube was the most effective in identifying the majority of quality issues.

2.2 Software Quality Issues in Student Code

In the literature, many works have dealt with evaluating the quality of software developed by university students at the introductory level or by novice programmers.

Several works (e.g. [4, 9, 12]) applied techniques and tools for automatically analyzing the quality of student-developed code to provide continuous feedback during the course and support the learning process. While these approaches proved to have a positive impact on the quality of student-developed code, their analyses focus on assessing the extent of the quality improvement when using such tools rather than exploring what are the foundational challenges encountered by introductory-level students.

Other works specifically focused on analyzing code developed by university students and novice programmers *ex-post*, to identify the most common quality flaws [7, 10]. Recently, Sun et al. [16] investigated the learning performance of students in object oriented programming courses, evaluating the quality of the code and tests produced by the students on simple assignments.

Most of this research is limited to evaluating the software quality of non-graduates from online programming platforms or introductory CS1 university courses. However, these works are affected by two key limitations. First, their analyses are based on simple assignments that, in most cases, could be solved with a few lines of code. It is not clear whether, and to what extent, the findings could apply when novice programmers develop more complex object oriented projects, involving a database and a Graphical User Interface (GUI). Second, due to the simplicity of the programming tasks involved, existing studies have, to the best of our knowledge, neglected quality issues arising from the presence of architectural anti-patterns. To the best of our knowledge, these limitations constrain the applicability of such studies to students in object oriented programming courses, where complexities in project scope and architectural considerations are markedly different.

3 AUTOMATIC ASSESSMENT OF QUALITY ISSUES IN STUDENTS' PROJECTS

In this section, we present the evaluation framework we developed to automatically assess quality issues of student software projects.

The proposed framework submits the student software projects to two analysis activities, namely Architectural Analysis and Code Quality Analysis, as reported in Figure 1. The former analysis aims at the identification of the violations of a fixed architectural pattern, while the latter aims at the detection of code smells. Both analyses deliver as output a report listing all the found issues.

The main features of the tools used to carry out the analyses and details of how we configured them to evaluate the student software projects will be explained in the following subsections.

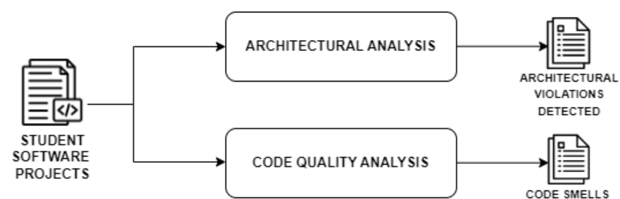

Figure 1: Evaluation Process

3.1 Assessment of Architectural Patterns Violations

The activity in the upper part of the process reported in Figure 1 aims to ensure that the architecture implemented by the students

in their projects follows the pattern that was presented by the teacher during the lectures. To automate this verification process there are many available tools and libraries, in particular, we have used ArchUnit[1]. ArchUnit is a free and easy-to-use library designed for architecture testing in Java projects, its simplicity lies in the possibility of defining architectural rules in the form of JUnit tests. It allows specific tests that can be performed to verify dependencies between packages and classes, as well as to examine the presence of cyclic dependencies and other relevant architectural aspects.

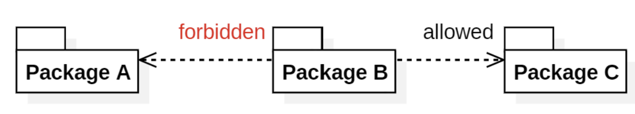

Figure 2: Package Dependencies example.

Listing 1: Sample ArchUnit test to ensure that dependencies are met, as shown in Figure 2

```
@ArchTest
public static final ArchRule enforce_package_dependencies =
  noClasses().that().resideInAPackage("..Package B..")
  .should().dependOnClassesThat().resideInAPackage("..Package A");
```

For example, suppose we want to check that dependencies between packages follow the constraint shown in Figure 2, i.e., no classes within package B should have dependencies on classes within package A. By exploiting ArchUnit it is possible to implement this check by means of a single, very readable test case such as the one shown in Listing 1.

3.1.1 ArchUnit Configuration: To enable architectural pattern violation detection, it is needed to add the ArchUnit library to the build automation script (e.g. pom.xml for Maven projects) and implement a test case expressing the defined architectural constraints by referring to the actual package names used in the project under analysis. Of course, ArchUnit test cases can be executed in the context of the adopted build automation tool and they automatically generate reports of the observed architectural pattern violations.

3.2 Assessment of Code Quality

The activity in the lower part of Figure 1 leverages the functionalities of SonarQube[2], one of the most popular and effective static analysis tools [11]. The tool compares the code against a comprehensive set of rules and best practices. In our experiment, we configured the tool to consider the set of 677 Java rules[3] and eventually filtered out only the rule violations of interest. SonarQube reports an issue each time one of these rules is violated, categorizing them into three types: *Bug*, *Vulnerability* and *Code Smell*. Each issue is assigned a levels of severity. The most severe issues, labelled as *Blocker*, can affect the application's behaviour and must be resolved before deployment. Next in severity is *Critical* issues and then *Major*, *Minor* and lastly *Info*. SonarQube code smells may affect different quality factors like maintainability, reusability, understandability, security, performance, and adherence to coding rules.

[1]ArchUnit documentation, available at https://www.archunit.org/
[2]SonarQube documentation, available at https://docs.sonarsource.com/sonarqube/latest/
[3]Java Rules list available at https://rules.sonarsource.com/java/

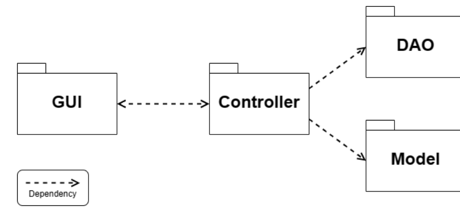

Figure 3: UML Package diagram representing the proposed reference architecture for student's projects

3.2.1 SonarQube Configuration: To enable automated code smell analysis, we deploy a dedicated SonarQube instance and instrument the build automation script to link it to the SonarQube instance. As a result, each time a project build is executed, SonarQube analysis is automatically performed.

As the output of the code quality analysis phase, SonarQube produces an Issues Report, namely a table containing all identified rule violations, classified by type and severity, and an estimation of the effort required, in terms of working hours, to correct the identified issue (i.e., the *technical debt*).

4 CASE STUDY

4.1 Study Design

To validate the proposed framework, we experimented with it on a set of Java projects produced by students of an Object Oriented Programming course. This kind of course typically represents the first introduction to the design and implementation of software systems using the object-oriented paradigm. This study aims to assess the usefulness of the proposed framework for identifying the prevalent issues and smells of the design and code of student projects. Two research questions were investigated:

RQ1: How common are architectural pattern violations in student projects?
RQ2: How common are code quality issues in student projects?

The subjects of the experiment were the students of an Object Oriented Programming course held by one of the authors in the second year of a Computer Science bachelor's degree program during the academic year 2021-22 at the University of Naples Federico II. The Object Oriented Programming course included 48 hours of lectures, corresponding to 6 ECTS, in the standard European Credit Transfer and Accumulation System (ECTS) way of defining the academic characteristics of courses.

As for the course topics, besides the basics of the object orientation paradigm, the course also presented an introduction to the UML language and the basics of the Java programming language. Moreover, the course hinted at the basics of code quality and architecture design principles, providing a reference architectural pattern for GUI-based applications, separating into four modules its different responsibilities. More in detail, the reference architectural pattern included: (1) a GUI package (2) a Controller package, (3) a Model package, (4) a Data Access Object (DAO) package, with, possibly, another package with utility classes. The proposed reference architecture is sketched in Figure 3. The GUI Package includes GUI classes representing the user entry point to the application.

Table 1: Percentage of Projects presenting disallowed Dependencies between GUI (G), Controller (C), Model (M), and DAO (D) Packages

Couplings							
G→M	**M→G**	**G→D**	**D→G**	**M→C**	**D→C**	**M→D**	**D→M**
23 (88%)	0	8 (31%)	1 (4%)	1 (4%)	1 (4%)	0	25 (96%)

These classes should depend on Controller classes, implementing the application logic. Controller classes, on the other hand, depend on Model classes that are responsible for the transient storage of the information domain data and on DAO classes for accessing the persistent data stored in the database. Controller classes are allowed to open GUI instances.

We carried out a post-evaluation of the quality issues of each project, searching both for architectural pattern violations (RQ1) leveraging ArchUnit, and for code smells (RQ2), using SonarQube. ArchUnit test cases were implemented for each project to measure the presence of each of the eight dependencies between packages that were forbidden by the reference architecture, i.e. from GUI to Model or Database and vice-versa, from Model to Database and vice-versa, from Model or Database to Controller. An example of the implemented test case is reported in the supporting material [6]. Next, the ArchUnit library was added to library dependencies in the Maven build automation script of each project. As for the code smell analysis, it was carried out by using a SonarQube instance installed in a Docker container accessible from the Maven build automation script of each project.

We collected in total 26 projects developed by 50 students. The projects featured, on average, 41 classes and 4,647 non-commenting lines of code. Note that these projects are remarkably more complex than the simple assignments employed in related works investigating software quality in student-developed code.

It is worth noting that setting up and tailoring the pipeline to automatically analyze the collected projects proved to be a straightforward process, as the students were already instructed to use Maven as a build tool, which simplified the integration operations. The biggest effort was required to manually tailor ArchUnit tests to the specific package naming used by students in each project and to integrate the SonarQube analysis in the student project's build phase. This step requires the creation of a new SonarQube project and the inclusion of an additional metric computation step in the project build pipeline, by editing Maven `pom.xml` files. More in detail, it took two of the authors of this paper approximately 7 minutes per project, on average, to complete the setup and run the analyses.

4.2 Results

4.2.1 RQ1 - How common are architectural pattern violations in student projects? Table 1 reports the number and percentage of projects from each considered group that presented at least one occurrence of a disallowed type of coupling (respectively from GUI to Model, from Model to GUI, from GUI to DAO, from DAO to GUI, from Model to Controller, from DAO to Controller, from Model to DAO and from DAO to Model).

The prevalent architectural violations were related to dependencies from DAO to Model (D→M). They were mostly due to cases in which classes from the DAO package extract data from the database and create instances of the Model classes to store them. These issues occurred on average in 96% of projects. *Students often coupled DAO and Model as another shortcut instead of passing serialized data to the Controller, which should have this responsibility.*

The direct dependencies from GUI to Model (G→M) were not allowed because Model classes and objects should be managed only by Controllers and not directly by GUI classes in order to make the GUI implementation independent from the information modelling. We observed this architectural issue very often (in 88% of projects). *These dependencies were shortcuts used by students to avoid the need to serialize data from model classes in data structures for the GUI.*

Another common architectural violation was the direct request of data from the GUI classes (G→D) that was observed in 31% of projects. The other violations were observed rarely: only in one project were found calls from DAO to GUI (D→G), from Model to Controller (M→C) or from DAO to Controller (D→C). Interactions from Model to GUI (M→G) and from Model to Data (M→D) were never observed.

4.2.2 RQ2 - How common are code quality issues in student's projects? The SonarQube analysis was carried out on the source code of each project and returned an average of 36.8 different typologies of detected issues per project. Tables including the overall list of smell typologies observed in all the student projects are available in the supporting material [6].

To have a summary view of the most common issues in the selected student projects, we limited the scope of our analysis to the top ten in order of frequency of observation, as reported in Table 2. For each issue type, the table reports its description as provided by SonarQube. In the other columns, we reported the classification of the issue (Bug or Code Smell), its severity (Blocker, Critical, Major, or Minor) and the number (and percentage) of students' projects in which the issue type was detected.

Among these 10 most frequently occurring issues, only one is classified as Blocker, 2 as Critical, 3 as Major and the remaining 4 as Minor. Moreover, only one of them is considered a Bug, whereas all the other ones are classified as Code Smells. No issues labelled with the Vulnerability tag were in the top ten list. 9 out of 10 issues affect the maintainability of the project, whereas the remaining one (i.e., the Bug) affects its reliability.

The Blocker category issue concerns "*Resources that were not properly closed*". It was found in 85% of projects. The issues were generally due to student inexperience: leaving open a stream or connection to an external resource can cause concurrency problems or memory leaks.

The two Critical issues are "*String literals should not be duplicated*" and "*Cognitive Complexity of methods should not be too high*". These issues are classified as Code Smells and are present in, respectively, 100% and 96% of projects. All of them negatively affected the maintainability of the projects. In particular, the first one may cause inconsistencies when string literals have to be modified (e.g., translated into another language), while the second one may make hard the comprehension of methods behaviour.

Table 2: List of the more common code issues found in student projects

Sonar Rule Description	Issue Type	Severity	Occurrence
Resources should be closed	BUG	BLOCKER	22 (85%)
String literals should not be duplicated	SMELL	CRITICAL	26 (100%)
Cognitive Complexity of methods should not be too high	SMELL	CRITICAL	25 (96%)
Standard outputs should not be used directly to log anything	SMELL	MAJOR	24 (92%)
Unused assignments should be removed	SMELL	MAJOR	21 (81%)
Branches in a conditional structure should not have the same implementation	SMELL	MAJOR	23 (88%)
Variables and method parameters should comply with naming conventions	SMELL	MINOR	25 (96%)
Field names should comply with a naming convention	SMELL	MINOR	23 (88%)
Unnecessary imports should be removed	SMELL	MINOR	25 (96%)
Method names should comply with a naming convention	SMELL	MINOR	18 (69%)

Three issues in the list are SonarQube Major issues and correspond to other Code Smells negatively affecting understandability and modifiability of the source code. For example, the use of `System.out` for logging (found in 92% of the projects) is not recommended because log outputs may mix with standard outputs and error outputs. Other frequent issues are related to unused assignments (81%) or branches of the same conditional structure with the same implementation (88%). All these bad practices appeared to be due to programmers' lack of attention to the quality of their code.

Similarly, the 4 most common Minor issues corresponded to Code Smells. Many of them represented a lack of coherence in the use of naming conventions about local variables, method parameters, field and method names (e.g., starting with a capital letter). In addition, in 96% of the projects unused imports were found. All these smells denoted a lack of knowledge or attention about naming conventions and best practices that were explicitly presented during lectures.

5 CONCLUSIONS AND FUTURE WORK

This paper has presented an ongoing experience about the automatic assessment of architectural pattern violations and code smells in Java projects realized by students. To this aim, we have implemented a pipeline by exploiting two state-of-the-art tools respectively ArchUnit to find architectural pattern violations and SonarQube for code smell analysis.

We presented the results of a case study conducted by analyzing 26 Java projects realized by students in an object-oriented programming course as part of a degree course in Computer Science at the University of Naples Federico II. The results showed that students produced frequent architectural pattern violations due to (1) bad practices adopted to avoid serialization and de-serialization operations for passing data between classes of different packages, and (2) inadequate understanding of the principles for correctly assigning responsibilities to classes. As regards code smells, we have observed frequent bad smells negatively affecting mainly reliability and maintainability. This work provides a pipeline for the automatic analysis of some architectural pattern violations and code smells that could be useful for teachers both to understand which quality aspects are more neglected by students and to support the quality assessment process of a large number of student projects.

In future work, we intend to replicate this study in the context of other editions of this course or other similar courses offered in other universities or degrees (e.g., in Software Engineering courses) to extend the general applicability of the automatic quality assessment.

REFERENCES

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, 187–197.

[2] Alison Clear, Allen S Parrish, John Impagliazzo, and Ming Zhang. 2019. Computing Curricula 2020: introduction and community engagement. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 653–654.

[3] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford. 2003. Documenting software architectures: views and beyond. In *25th International Conference on Software Engineering, 2003. Proceedings.* 740–741.

[4] Pedro Henrique de Andrade Gomes, Rogério Eduardo Garcia, Gabriel Spadon, Danilo Medeiros Eler, Celso Olivete, and Ronaldo Celso Messias Correia. 2017. Teaching software quality via source code inspection tool. In *2017 IEEE Frontiers in Education Conference (FIE)*. 1–8. https://doi.org/10.1109/FIE.2017.8190658

[5] Peter J Denning. 1992. What is software quality? *Commun. ACM* 35, 1 (1992), 13–15.

[6] Sergio Di Meglio, Luigi Libero Lucio Starace, Marco De Luca, Porfirio Tramontana, and Anna Rita Fasolino. 2024. Automatic assessment of architectural anti-patterns and code smells in Student Software Projects. https://zenodo.org/records/10800604

[7] Tomáš Effenberger and Radek Pelánek. 2022. Code Quality Defects across Introductory Programming Topics. In *53rd ACM Technical Symposium on Computer Science Education - Volume 1 (SIGCSE 2022)*. ACM, 941–947.

[8] John Estdale and Elli Georgiadou. 2018. Applying the ISO/IEC 25010 quality models to software product. In *Systems, Software and Services Process Improvement: 25th European Conference, EuroSPI 2018, Bilbao, Spain, September 5-7, 2018, Proceedings 25*. Springer, 492–503.

[9] Julian Jansen, Ana Oprescu, and Magiel Bruntink. 2017. The impact of automated code quality feedback in programming education. *CEUR Workshop Proceedings* 2070 (2017).

[10] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACm, 110–115.

[11] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023), 111575. https://doi.org/10.1016/j.jss.2022.111575

[12] Yao Lu, Xinjun Mao, Tao Wang, Gang Yin, and Zude Li. 2019. Improving students' programming quality with the continuous inspection process: a social coding perspective. *Frontiers of Computer Science* 14, 5 (2019). https://doi.org/10.1007/s11704-019-9023-2

[13] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[14] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2021. Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles. *IEEE Transactions on Software Engineering* 47, 5 (2021), 1008–1028.

[15] Mary Shaw and David Garlan. 1996. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc.

[16] Qing Sun, Ji Wu, and Kaiqi Liu. 2020. Toward understanding Students' learning performance in an object-oriented programming course: The perspective of program quality. *IEEE Access* 8 (2020), 37505–37517.

[17] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. 2010. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons.

[18] Alla Zakurdaeva, Michael Weiss, and Steven Muegge. 2020. Detecting architectural integrity violation patterns using machine learning. In *35th Annual ACM Symposium on Applied Computing (SAC '20)*. ACM, 1480–1487.