A Preliminary Evaluation on the Relationship among Architectural and Test Smells

Manuel De Stefano[1], Fabiano Pecorelli[2], Dario Di Nucci[1], Andrea De Lucia[1]
[1]*SeSa Lab - University of Salerno*, Fisciano, Italy
[2]Tampere University, Tampere, Finland
madestefano@unisa.it, fabiano.pecorelli@tuni.fi, ddinucci@unisa.it, adelucia@unisa.it

***Abstract*—Software maintenance is the software life cycle's longest and most challenging phase. Bad architectural decisions or sub-optimal solutions might lead to architectural erosion, i.e., the process that causes the system's architecture to deviate from its original design. The so-called architectural smells are the most common signs of architectural erosion. Architectural smells might affect several quality aspects of a software system, including testability. When a system is not prone to testing, sub-optimal solutions may be introduced in the test code, a.k.a. test smells. This paper explores the possible relations between architectural and test smells. By mining 798 releases of 40 open-source Java systems, we studied the correlation between class-level architectural and test smells. In particular, Eager Test and Assertion Roulette smells often occur in conjunction with Cyclically-dependent Modularization, Deficient Encapsulation, and Insufficient Encapsulation architectural smells.**

***Index Terms*—Architectural Smells; Code quality; Test Smells; Correlation; Inter-smell Relationships.**

I. Introduction

Software maintenance is the most extended phase of the software life cycle [1]. Once delivered to the customer, it is necessary to proceed with continuous and periodic changes to preserve software usefulness [1]. However, developers are often forced to adopt sub-optimal solutions in the maintenance activities to speed up the work and delivery the product in time. Thus, good programming principles are often sacrificed for a short-term benefit, leading to a cost to be paid back, namely "technical debt". The symptoms of these sub-optimal decisions are called *code smells*, which were defined and collected in a catalog by Martin Fowler in his book on *refactoring* [2]. Afterward, further studies proved that code smells affect code maintainability and understandability [3].

However, sub-optimal solutions might also occur when designing software architecture. Bad architectural decisions might lead to "architectural drift", i.e., the process that causes the system's architecture to deviate from its original design [4]. When this deviation worsens, going against the conceived architecture, we face "architectural erosion" [4], whose symptoms are described by architectural smells. Similarly to their code counterpart, they represent the same problem, but at a higher level of abstraction [5]–[8].

Architectural smells may impact all system quality attributes, e.g., maintainability and understandability [5], [9], [10], including testability. On the one hand, good architecture should foster and facilitate the testability of a system [9]. On the other hand, tangled dependencies among components and other smells make it difficult to design and develop tests [11].

Test smells are signs of a degraded test code quality [12] that represent less-than-ideal solutions and poor design choices that impact test code, just as their equivalents in code and architecture. Previous studies have looked at these smells from various perspectives, highlighting that tests that are affected by them have less ability to discover faults [13]. In addition, they need more work to be maintained or improved [14], [15].

Since architectural smells are signs of degraded architecture and test smells are signs of degraded test code, it could be possible to spot issues in the test code by looking at the signs of a degraded architecture. In other words, there might be a relationship between architectural and test smells. To explore this possibility, we conducted a preliminary study that considered 11 class-level architectural and seven test smells on 798 releases of 40 Java open-source systems hosted on GitHub. We detected the architectural smells affecting these systems and the test smells affecting the corresponding test class. Then we mined association rules to discover potential relationships among them. The obtained results showed that Cyclically-dependent Mdularization, Deficient Encapsulation, and Insufficient Encapsulation architectural smells often imply Eager Test and Assertion Roulette in the test code.

The paper is then organized as follows. Section II depicts related work on architectural smells and test smells. Section III describes the empirical investigation we conducted, while Section IV presents the results we achieved. Section V depicts potential threats to the study's validity and how we managed them. Finally, Section VI wraps up the paper and presents future research directions.

II. Related Work

The concept of architectural smell was introduced by Lippert *et al.* [8]. They pointed out the concept of architectural smells and defined them as *bad smells* that occur at a higher of the system's granularity.

Garcia *et al.* in 2009 [10] defined *architectural bad smell* as "frequently recurring software designs that can have non-obvious and significant detrimental effects on system life-cycle properties, such as understandability, testability, extensibility, and re-usability".

To what concerns test smells, Beck [16] was the first to stress the value of having well-designed test code. He stated that test cases adhering to sound design principles are desirable since these test cases are simpler to understand, maintain, and successfully use to identify issues in the production code.

Based on these arguments, van Deursen et al. [17] created the term "test smells" and the very first list of 11 bad design choices for writing tests, along with refactoring procedures intended to eliminate them.

The first controlled laboratory experiment to determine the effect of test smells on program comprehension during maintenance operations was carried out by Bavota *et al.* [14]. According to their results, test smells detrimentally impact the understandability and maintainability of test code.

Spadini *et al.* [13] investigated the relationship between the presence of test smells and the change- and defect-proneness of both these test methods and the production code they intend to test. They discovered that test code with smells is more prone to changes and defects. Moreover, production code tested by smelly tests is more defect-prone as well. Among the studied test smells, "Indirect testing", "Eager Test", and "Assertion Roulette" are those associated with the highest change-proneness; moreover, the first two are also related to a higher defect-proneness of the exercised production code.

III. Empirical Study Design

The *goal* of our work is to study the correlation between architectural and test smells with the *purpose* of assessing whether the degradation of software architectures detrimentally impacts test code quality. The *perspective* is of both researchers and practitioners interested in discovering the potential impact of architectural smells on the emergence of test smells, a possible cause-effect relationship, and new ways of predicting their emergence. Given this, we aim to answer the following research question:

RQ. Is there any relationship between architectural smells and test smells?

A. Context selection

The context of our study is composed of architectural smells, test smells, and software systems. Table I depicts the architectural smells we considered in our study. The rationale behind these smells is that they are all class-level defined architectural smells [5] which can be detected by tools [5], [18]. In particular, we relied on Designite [18], a static analysis tool for assessing technical debt. Designite can detect seven module- (package-) level architectural smells, 20 class level (design) architectural smells, and 11 implementation smells in software systems implemented in C# and Java. We limited our choice to class-level defined architectural smells to have the same granularity of test smells defined at the (test) class level. However, we excluded the "Feature Envy" smell since it is considered a code smell rather than an architectural smell.

Table II depicts the well-known test smells we considered in our empirical study. [12], [17], [19] which can be automatically detected in source code by tools [20]. In this case, we relied on VITRuM, a code analysis tool that can detect test smells in Java systems. It was initially designed to allow developers to see static and dynamic test-related data (alongside test smells) in an enhanced visual interface, potentially improving their ability to diagnose code flaws. We relied on a fork of VITRuM, developed to run in a CLI environment to make it useful to mining studies.

Finally, to what concerns the systems we took into consideration, we selected java systems from GitHub that (1) had at least ten releases, (2) had at least 10,000 stars, (3) were not forks, (4) had at least one test class, and (5) followed the standard test directory structure `src/test`. We employed the first three criteria to select mature systems (i.e., highlighted by the number of releases and stars), which we could analyze with the available tools (i.e., only Java projects). In particular, we leveraged "Github Search", which allowed us to apply the above criteria all at once.[1] The output of this tool was a list of 90 GitHub repositories matching the criteria. Criterion 4 was included to ensure having at least one test to analyze to measure test smells. Finally, the last criterion was due to VITRuM, which only works with a standard test directory structure. As a result, we obtained a final number of 40 java systems and 798 releases.

For the sake of space limitations, we include the complete list of the considered systems, along with their characteristics, in our online appendix.[2]

B. Data Collection and Analysis

In the following, we describe the procedure we performed to gather the necessary data and the steps we performed to analyze it. First, we collected all repositories of the systems from GitHub. Then, we mined all the releases (tagged commits) for each system by using RepoDriller [21]. On each release, we ran Designite [18] to detect the architectural smells. Once we collected the production classes, we gathered the corresponding test classes, and by leveraging on VITRuM [20], we determined whether they were affected by test smells. The final output of this step was a collection of records where it was possible to find components affected by architectural smells and the test classes affected by test smells.

After the mining procedure, we performed association rule mining by exploiting the aPriori algorithm [22] to find which architectural smells and test smells co-occur. Association rule mining, in particular, is an unsupervised learning technique for detecting local patterns, highlighting attribute value conditions that occur together in a dataset [22]. In our specific case, in our case, the dataset contained the set of architectural smells and test smells discovered in each release of the considered systems. An association rule in the form of $R_{left} \rightarrow R_{right}$ indicates that the presence of an architectural smell in a class implies the occurrence of that particular test smell in the relative test class. For each association rule mined, we

[1]https://seart-ghs.si.usi.ch
[2]Online Appendix https://figshare.com/s/72d71fb1f99e1e8c0080

TABLE I
ARCHITECTURAL SMELLS CONSIDERED IN THE CONTEXT OF OUR STUDY.

Name	Definition
Deficient Encapsulation (DE)	This smell occurs when the declared accessibility of one or more members of abstraction is more permissive than required.
Unutilized Abstraction (UNA)	This smell arises when an abstraction is left unused (either not directly used or not reachable)
Broken Hierarchy (BH)	This smell arises when a super-type and its sub-type conceptually do not share an "IS-A" relationship resulting in broken substitutability
Broken Modularization (BM)	This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions
Insufficient Modularization (IM)	This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both
Wide Hierarchy (WH)	This smell arises when an inheritance hierarchy is "too" wide, indicating that intermediate types may be missing
Unnecessary Abstraction (UA)	This smell occurs when an abstraction that is not needed (and thus could have been avoided) gets introduced in a software design
Multifaceted Abstraction (MA)	This smell arises when an abstraction has more than one responsibility assigned to it
Cyclically-dependent Modularization (CDM)	This smell arises when two or more abstractions depend on each other directly or indirectly
Cyclic Hierarchy (CH)	This smell arises when a super-type in a hierarchy depends on any of its sub-types
Rebellious Hierarchy (RH)	This smell arises when a sub-type rejects the methods provided by its super-type(s)

TABLE II
TEST SMELLS CONSIDERED IN THE CONTEXT OF OUR STUDY.

Name	Definition
Ignored Test (It1)	A test method or class that contains the `@Ignore` annotation
General Fixture (Gf1)	Not all fields instantiated within the `setUp()` method of a test class are utilized by all test methods in the same test class
Resource Optimism (Ro1)	A test method utilizes an instance of a `File` class without calling the `exists()`, `isFile()` or `notExists()` methods of the object
Assertion Roulette (Ar1)	A test method contains more than one assertion statement without an explanation/message (parameter in the assertion method)
Eager Test (Et1)	A test method contains multiple calls to multiple production methods
Mystery Guest (Mg1)	A test method containing object instances of files and databases classes
Sensitive Equality (Se1)	A test method invokes the `toString()` method of an object

computed two metrics, namely *support* and *confidence* [22], which are defined as follows:

$$support = \frac{|R_{left} \cup R_{right}|}{T} \quad (1)$$

$$confidence = \frac{|R_{left} \cup R_{right}|}{R_{left}} \quad (2)$$

where T is the total amount of co-occurrences among architectural and test smells in our dataset. These metrics gave us insights concerning the strength of the association. We filtered out all the rules having a support value of 0.1 and a confidence value of 0.8 as defined in previous studies [22]. Moreover, we calculated the lift metric, which assesses the ability of a rule to accurately detect a relationship when compared to a random choice model [22]. A lift value greater than one indicates that the left-hand and right-hand operators of an association rule occur together more frequently than predicted, implying that the existence of the left-hand operator frequently suggests the presence of the right-hand operator. Finally, to statistically evaluate the significance of the mined rules, we computed Fisher's exact test [23] on the lift value. This test measures the significance of deviation between the association rule model and the random choice models [23]. If the *p-value* is lower than 0.05, we consider the deviation statistically significant.

TABLE III
STATISTICALLY SIGNIFICANT ASSOCIATION RULES MINED FROM THE OBSERVED DATA, ORDERED BY DESCENDING LIFT VALUE.

Rule	Support	Confidence	Lift
CDM $\rightarrow$ et1	0.147	0.916	1.165
IM $\rightarrow$ et1	0.254	0.958	1.219
DE $\rightarrow$ et1	0.246	0.846	1.075
DE $\rightarrow$ ar1	0.251	0.864	1.072

IV. Analysis of Results

Table III depicts the association rules we mined from the analyzed data, which were evaluated as statistically significant by Fisher's exact test [23]. The first and foremost aspect that we can note from the table is that three out of four association rules have the "Eager Test" (et1) as the right-hand side of the rule. This observation is quite reasonable since "Eager Test" represents a test class that violates the single responsibility rule [24] because it tests multiple production methods, i.e., more behaviors.

The first rule, which associates the presence of CDM with et1 might be explained by the fact that cyclic dependencies among classes do not fully allow for isolating a production class in a test environment. Indeed, a class involved in a cyclic-dependent modularization will always require some other class to be instantiated, requiring other classes recursively. This architectural flaw makes it difficult to create test stubs, and it might be that the actual production classes are used in tests. Eventually, some methods of the other classes might also be tested in the same test method. This rule shows the highest lift value, which means it is very likely that a class affected by CDM will have its test class affected by et1.

The second rule states that when a class suffers from insufficient modularization (IM), its test class will exhibit an eager test smell. This rule shows a reasonably high lift value and higher support than the former (0.254 vs. 0.157), meaning it frequently happens. This relationship might be explained by the cumbersome nature of classes affected by IM. Since these classes are composed of non-fully decomposed abstractions, some internal dependencies might force developers to test multiple production methods, as it was an end-to-end test.

The third and fourth rules regard Deficient Encapsulation (DE) architectural smells implying eager test (et1) and assertion roulette (ar1). They showed the smallest lift value (1.075 and 1.072, respectively), which means that the presence of the former implies the presence of the latter slightly often. A possible explanation behind these two rules might be given by the nature of the architectural smell itself. A deficient encapsulation occurs when some aspects of a class, i.e., methods or attributes, have too permissive visibility. Therefore, methods or attributes that should not be accessible from external classes, including test classes, are visible. When testing a production method, developers might have tested methods (which should be private) invoking the method under test or included some attribute of the external class to ensure its correct state after executing the method under test. This situation leads to eager tests (since multiple production methods are testes) and assertion roulette (since multiple assertions are used to check some class attributes as well).

V. Threats To Validity

In the following, we discuss the potential threats to the validity of our study, alongside the strategies we applied to mitigate them.

A. Threats To Construct Validity

The relationships between theory and observation are a threat to construct validity. This threat typically alludes to measurement accuracy issues. In our case, it impacts all data collected because it may be "biased" by the sloppiness of the used tools. However, Designite and VITRuM are both well-established tools [18], [20], demonstrating strong detection abilities. Therefore, we are confident in the veracity of the information gathered.

B. Threats To Conclusion Validity

Using the aPriori algorithm to identify connections between the observable events poses the most significant threat in this area. On the one hand, researchers have commonly used this method to examine unknown connections between two occurrences ([25]–[27]). On the other hand, we only considered and discussed the most solid regulations, i.e., those that were most trustworthy and achieved great support and confidence and were found as significant by Fisher's exact test [23].

C. Threats to External Validity

The main threats falling into this category might be related to the number of analyzed systems and the choice of smells. To what concerns the systems, we analyzed a dataset of 40 Java systems that met certain criteria (e.g., being old enough to have ten releases or having a minimum number of stars to exclude toy systems). On the one hand, 40 Java systems on the whole amount of systems might seem a small number. Even the selection criteria (i.e., number of stars and releases) were highly selective to maximize the possibility of not selecting toy projects, which would have been irrelevant for the study. Hence, on the other hand, it must be considered that the purpose of this study is a preliminary evaluation of the possible effect of architectural erosion and not an extensive empirical study on the whole architectural conditions of all the systems hosted on GitHub. Moreover, it must also be considered that we were limited by the tools we employed, i.e., Designite and VITRuM, which can analyze only systems written in Java. Further studies will consider a more extensive set of systems and different programming languages whenever tools supporting them are available. To what concerns the considered smells, we were also limited by the tools at our disposal. Further studies will certainly consider other smells as soon as we can detect them.

VI. Conclusion

In this paper, we explored the potential effect of architectural degradation on some quality aspects of a software system. In particular, we examined how architectural degradation (seen as the presence of architectural smells) can impact the testability of the system (seen as the presence of test smells in the test code). We detected architectural and test smells occurring on 798 releases coming from 40 Java projects hosted on GitHub by leveraging association rule mining. We found that the presence of some architectural smells (i.e., Cyclically-dependent Modularization, Insufficient encapsulation, and Deficient Encapsulation) often implies the presence of test smells in their test classes (i.e., Eager Test and Assertion Roulette).

Further, more in-depth studies will be conducted to discover not only *how* this occurs, but also *why*. In particular, we plan to conduct historical, observational, and case studies to make us understand the causality relationship between these smells. We considered only class-level architectural smells. Thus, in the future, we will also consider the impact of module-level (or package-level) architectural smells on the software testability.

In the long run, we aim to develop an automatic architectural refactoring strategy to improve software testability. In this scenario, refactoring will primarily improve the overall quality of the architecture and the side effect of facilitating testing activities, which eventually could improve other non-functional requirements, such as the system dependability. Hence, further studies will consider other quality aspects that architectural degradation might affect, not only at the product level but also at the process level.

References

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] M. Fowler and K. Beck, "Refactoring: Improving the design of existing code," *Addison-Wesley Longman Publishing Co., Inc.*, 1999.

[3] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 403–414.

[4] N. Medvidovic and R. N. Taylor, "Software architecture: foundations, theory, and practice," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2. IEEE, 2010, pp. 471–472.

[5] H. Mumtaz, P. Singh, and K. Blincoe, "A systematic mapping study on architectural smells detection," *Journal of Systems and Software*, vol. 173, p. 110885, 2021.

[6] U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2019, pp. 88–97.

[7] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *International conference on the quality of software architectures*. Springer, 2009, pp. 146–162.

[8] M. Lippert and S. Roock, *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

[9] R. C. Martin, J. Grenning, S. Brown, K. Henney, and J. Gorman, *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall, 2018, no. s 31.

[10] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258.

[11] M. Feathers, *Working Effectively With Legacy Code: Work Effect Leg Code _p1*. Prentice Hall Professional, 2004.

[12] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[13] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 1–12.

[14] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[15] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019.

[16] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[17] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 2001, pp. 92–95.

[18] T. Sharma, P. Mishra, and R. Tiwari, "Designite: A software design quality assessment tool," in *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, 2016, pp. 1–4.

[19] L. Moonen, A. v. Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software evolution*. Springer, 2008, pp. 173–202.

[20] F. Pecorelli, G. Di Lillo, F. Palomba, and A. De Lucia, "Vitrum: A plug-in for the visualization of test-related metrics," in *Proceedings of the International Conference on Advanced Visual Interfaces*, 2020, pp. 1–3.

[21] M. Aniche, "Repodriller," 2012.

[22] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993, pp. 207–216.

[23] R. A. Fisher, "On the interpretation of χ 2 from contingency tables, and the calculation of p," *Journal of the royal statistical society*, vol. 85, no. 1, pp. 87–94, 1922.

[24] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[25] M. De Stefano, F. Pecorelli, D. A. Tamburri, F. Palomba, and A. De Lucia, "Splicing community patterns and smells: A preliminary study," in *Proceedings of the ieee/acm 42nd international conference on software engineering workshops*, 2020, pp. 703–710.

[26] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[27] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.