Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

Correlating Pattern Grime and Quality Attributes

Daniel Feitosa[1], Apostolos Ampatzoglou[1], Paris Avgeriou[1], Senior Member, IEEE, and Elisa Y. Nakagawa[2], Member, IEEE

[1]Department of Mathematics and Computing Science, University of Groningen, Groningen 9700 AK, The Netherlands

[2]Department of Computer Systems, University of São Paulo, São Carlos - São Paulo, Brazil

Corresponding author: Daniel Feitosa (e-mail: d.feitosa@rug.nl).

This work was financially supported by the Brazilian and Dutch agencies CAPES/Nuffic (Grant N.: 034/12), CNPq (Grant N.: 204607/2013-2), as well as the INCT-SEC (Grant N.: 573963/2008-8 and 2008/57870-9).

ABSTRACT The GoF design patterns are widely adopted in industry as best practices and their effect on software quality has been long investigated in academia, with both positive and negative consequences being observed. One important parameter that relates to the effect of patterns on quality is the deterioration of pattern instances due to the buildup of artifacts unrelated to the pattern structure. This is called pattern grime and can potentially diminish some of the benefits of using patterns in the first place. In this paper we investigate the relation between pattern grime and three qualities, namely performance, security and correctness. To this end, we conducted a case study with five industrial projects (approx. 260,000 lines of code) implemented by 16 developers. Our findings suggest a correlation between the accumulation of grime and decreased levels of performance, security, and correctness. Moreover, factors such as the project itself, pattern type and the developer can influence this relation. The obtained results can benefit both researchers and practitioners, as we provide evidence on the accumulation of pattern grime and its correlation to performance, security and correctness, and how different factors affect these correlations.

INDEX TERMS Design patterns, pattern grime, quality attributes, industrial case study

I. INTRODUCTION

The popular GoF (Gang of Four - Gamma, Johnson, Helm, and Vlissides) design patterns catalog consists of 23 solutions to recurring problems of object-oriented design [1]. Practitioners often adopt them as good design practices, but at the same time they are concerned with their impact on the system under development, particularly their effect on quality attributes [2]. This concern is reasonable, as patterns can occur in a significant part of software systems (from 15% to 65% of the classes) [3], [4]. Additionally, the state of the research suggests that this effect of patterns on software quality is not uniform, but it depends on a number of parameters [5]. Several works have concluded that a pattern can be beneficial in some cases and harmful in others, with respect to a specific quality attribute, by studying the structural characteristics of patterns, such as the number of pattern participating classes, number of methods, etc [6]–[9].

One significant aspect of patterns' instantiation that might incurs negative consequences on software quality is the presence of artifacts (e.g., methods or classes) that are not related to the pattern rationale. This phenomenon has been defined by Izurieta and Bieman [10] as **pattern grime**, which is *the "degradation of a design pattern instance due to accumulation of artifacts unrelated to the instance"*. For example, in a Decorator pattern instance, the addition of public methods to the class playing the Decorator role that are not invoked inside the class playing the Component role introduces grime into the instance as this new responsibility is not compliant with the original definition of the pattern [1]. Such a change could reduce the cohesion of the class, as well as hinder its understandability [9]. In general, accumulating pattern grime contributes to the degradation of quality in pattern instances [10]–[12]. Given the aforementioned high percentage of class participation in GoF patterns, the effects of ever-growing grime can be detrimental to the overall quality of those systems.

Despite ongoing research on identifying the impact of pattern grime on software quality [10], [11], [13], there are still three shortcomings. First, only a few quality attributes have been addressed so far, namely testability, adaptability and understandability. Second, despite the existence of industrial case studies examining how pattern grime accumulates [14], [15], there is a lack of industrial studies regarding how the accumulation of grime relate to levels of quality attributes; the existing studies are limited to open source software. Finally, even these studies on open source have limited depth regarding the investigation of factors that contribute to this relation between grime and qualities. For example, developers with different levels of involvement in a project may accumulate grime differently. Identifying the factors related to higher levels of grime can improve the impact of design patterns on quality, as well as to a more adequate allocation of resources in a project.

In this paper, we address the aforementioned shortcomings through an industrial case study that examines the relation between the accumulation of pattern grime and quality. The study was designed according to the guidelines of Runeson et al. [16], reported based on the Linear Analytic Structure [16]. Thus, we offer three advancements compared to the state of the art (which is further elaborated in Section II). First, we focus on three qualities that have not been studied: performance, correctness and security. Second, we consider five industrial software systems for our investigation, instead of open source. Finally, we investigate three factors that may influence the underlying relations:

- the *projects* under development have several characteristics such as application domain and type of systems (e.g., user application, library), which may influence the usage of patterns and development practices. Studies have already shown that projects can accumulate pattern grime differently [10], [15]. Thus, we seek to investigate if this may reflect on the relation between grime and levels of quality as well;
- the *types of pattern* (e.g., Template Method, Singleton, etc.) have also been pointed out as a factor on how pattern grime is accumulated [10], [14], [15]. The different structural and behavioral characteristics of patterns may also be related to how exactly quality is affected; and
- the *developers* often have different traits such as background and experience, which may affect their behavior and productivity [17]. Besides, developers have also been found to accumulate grime differently [15], which corroborates the relevance of also investigating if this factor relates to a varying level of quality.

The study is executed based on the commits performed by 16 developers during the implementation of five projects that sum up to approx. 260,000 source lines of code. The studied qualities are assessed through the number of violations of various coding practices, each one mapped to one of the qualities (for more details see Section III.C.4).

The remainder of this paper is organized as follows. In Section II, we present related work. The design of our case study is described in Section III. In Sections IV and V, we report on our results and discuss the most important findings. We present the identified threats to validity in Section VI, together with actions taken to mitigate them. In Section VII, we conclude the paper and present some interesting extensions for this study.

II. RELATED WORK

In this section, we focus on the terminology related to pattern grime, and address empirical studies that investigate the relation between accumulation of grime and quality attributes.

A. DESIGN PATTERN GRIME AND QUALITY ATTRIBUTES

Pattern grime concerns the degradation of pattern instances without breaking down the original structure on the pattern definition [10]. This degradation occurs through the addition of associations that do not comply with patterns' responsibilities (e.g., addition of a public method that is not in the definition), which can accumulate along the evolution of the instance and obscure their design [11]. Izurieta and Bieman [18] established that the added associations can be assessed from three base perspectives, i.e., there are three forms of pattern grime. *Class grime* regards class-related elements (e.g., number of attributes, methods, or children) that are unrelated to the role of a class in the pattern instance. *Modular grime* regards relationships (e.g., dependency, generalization) between classes of the pattern instance and other classes, which are not predicted in the definition of the pattern. *Organizational grime* regards how pattern-participant classes are distributed into packages and/or namespaces. This threefold classification was further refined by Schanz and Izurieta [14], who provided a taxonomy of subtypes for modular grime, and by Griffith and Izurieta [13], with a taxonomy of subtypes for class grime.

Regarding the relation between the accumulation of pattern grime and the levels of quality attributes, we identified three empirical studies. Izurieta and Bieman [11] investigated how grime is associated with the testability of pattern instances. For that, they considered instances of Singleton, Visitor and State patterns obtained from an open-source system and assessed their testability by the number of test cases necessary to cover them. By analyzing the testability against the accumulation of modular grime, Izurieta and Bieman found that testability decreases (i.e., more test cases are needed) as grime accumulates. Moreover, other issues such as the appearance of code smells also aggravate. In a complementary study, Izurieta and Bieman [10] explored how pattern grime affects the testability and adaptability (measured by pattern instability) of instances from three open-source systems. They examined all three forms of pattern grime (i.e., class, modular and organizational) and again observed a negative impact. Both testability and adaptability decreased with the accumulation of grime, although the results regarding organizational grime were inconclusive due to lack of more data. Finally, Griffith and Izurieta [13] investigated how the understandability of pattern instances changes due to the accumulation of grime. To this end, they focused on class grime and randomly collected pattern instances from a database of open-source components [19]. By correlating the accumulated grime with understandability (assessed according to the QMOOD quality model [20]), they found that this quality attribute is also affected negatively.

B. COMPARISON TO STATE OF THE RESEARCH

In Table I, we compare the main parameters that differentiate our study from related work. In particular, we emphasize that: (a) we investigated three quality attributes (i.e., performance, security and correctness) that have not been addressed in this context; (b) we studied five industrial non-trivial projects (in contrast to open-source ones) that collectively provided 36,571 units of analysis (i.e., modifications to the source code of pattern instances, see Section III); and (c) we investigated factors that, although have been explored with regards to the accumulation of grime, have not still been examined with regards to the relation between grime and quality attributes.

TABLE I
COMPARISON WITH RELATED WORK

Parameter	Study			
	[11]	[10]	[13]	Ours
Context	open-source	open-source	open-source	industrial
Projects	1	3	not clear	5
Patterns	3	7	16	9
Instances	2	"small number"	not clear	2,329
Forms of grime	modular	class, modular and organizational	class	class, modular and organizational
Quality attributes	testability	testability and adaptability	understandability	correctness, performance and security
Factors	pattern	lines of code	none	project, pattern, developer

III. STUDY DESIGN

In this section, we present the protocol of our case study, designed according to the guidelines of Runeson et al. [16], reported based on the Linear Analytic Structure [16].

A. OBJECTIVES AND RESEARCH QUESTIONS

We formulated the goal of this study using the Goal-Question-Metric (GQM) approach [21], as follows: "*analyze* the accumulation of grime on GoF pattern instances *for the purpose of* evaluation *with respect to* its relationship with the levels of performance, security and correctness, *from the point of view of* software designers *in the context of* industrial software development". To accomplish this goal, we proposed three research questions (RQs), which are elaborated as follows.

RQ$_1$ Does the accumulation of pattern grime correlate with changes in the investigated quality attributes?

RQ$_{1.1}$ Is a correlation observed for class grime?

RQ$_{1.2}$ Is a correlation observed for modular grime?

RQ$_{1.3}$ Is a correlation observed for organizational grime?

RQ$_1$ aims at acquiring initial evidence of the relationship between the accumulation of pattern grime and changes in the levels of correctness, performance and security. We note that we address each quality attribute in isolation. To more comprehensively answer this question, we investigated all three forms of grime proposed by Izurieta and Bieman [18], i.e., class, modular and organizational grime.

RQ$_2$ Which factors affect the aforementioned relation?

RQ$_{2.1}$ Does it vary for different projects?

RQ$_{2.2}$ Does it vary for different patterns?

RQ$_{2.3}$ Does it vary for different developers?

Next, we extend our analysis to factors that may influence the relation between pattern grime and quality attributes. In this study, we examined three factors. First, we investigated if the correlation between grime and quality attributes differs for different projects (RQ$_{2.1}$). Second, we were interested in answering this question, but for different patterns (RQ$_{2.2}$). Although these two factors were briefly addressed in related work, they have not been empirically explored so far. To complement the analysis, we also investigated whether the relationship varies depending on the developer (RQ$_{2.3}$), in the sense that the expertise or experience of developers may be reflected in the accumulated grime and/or quality attribute.

B. CASE SELECTION AND UNIT OF ANALYSIS

To answer the posed research questions, we designed an exploratory case study [16]. Since *related work is limited to studying only open-source applications*, we decided to fill the gap and perform an industrial case study with five industrial projects from a company in the domain of web and mobile applications development. Moreover, these projects provided us with a diverse and comprehensive sample of developers (and projects) to investigate: one team of six people worked on three of the projects, while the other two projects were developed by two other teams (of five people each) independently.

The cases of our study comprise pattern instances of the aforementioned projects. Based on the evolution of these instances, we assemble our units of analysis, which consist of the changes that they undergo (i.e., the source code change between two successive commits). We perform our analyses and answer our research questions based on this unit of analysis and, thus, we selected this particular unit due to its granularity, which allows us to isolate all necessary variables. In particular, *collecting data regarding individual developers* *is facilitated, as commits regard changes authored distinctively*.

We clarify that the usage of such unit of analysis entail the collection of multiple data points for individual pattern instances. However, each data point concerns a different snapshot of the pattern instance, i.e., there is no repetition. Moreover, the snapshot of a pattern instance is collected only when a change is made, i.e., the collection is not made for every commit. Nevertheless, it is paramount to avoid bias by having an excessive number of units from a few instances. For that, we verified the balance of our population on this regard (see Section IV).

Other main sources of bias regard the authoring of commits and the size of change, which may compromise our analyses. To address these concerns, we consulted with the company, which informed us that their developers are not allowed to commit for each other neither exchange source code to commit it. Moreover, a practice of small commits is encouraged to avoid the aforementioned bad practices.

C. VARIABLES AND DATA COLLECTION

To address the research questions, we recorded four sets of variables for each unit of analysis. Each set regards one of the major steps in the data collection: 1) Characterize commits; 2) Collect patterns instances; 3) Assess pattern grime; 4) Assess quality attributes. In the following we describe each step, the variables collected in them (highlighted between parentheses), and tools we used. A summary of the recoded variables is presented in Table II.

TABLE II
LIST OF RECORDED VARIABLES

Step	Variable	Description
1	project	Project from which the PI was extracted
	commit	Hash of the commit in the git repository
	dev	Developer author of the commit
2	inst_id	ID of the PI that the class belongs to
	pattern	GoF design pattern of the instance
3	cg-napm	# alien public methods in all PI classes (Class grime)
	cg-naa	# alien attributes in all PI classes (Class grime)
	mg-ca	Afferent coupling of the PI (Modular grime)
	mg-ce	Efferent coupling of the PI (Modular grime)
	og-np	# packages within the PI (Organizational grime)
	og-ca	Afferent coupling at the package level (Organizational grime)
4	cor-viol	# correctness violations in all PI classes
	per-viol	# performance violations in all PI classes
	sec-viol	# security violations in all PI classes

PI stands for "pattern instance"
# stands for "Number of"

1) STEP 1: CHARACTERIZE COMMITS

The versioning of the five projects was managed using Git. For each commit we recorded the project name (*project*), the commit hashcode (*commit*) and the developer responsible for the commit (*dev*). We also recorded the files that were modified in order to filter out undesired commits. In particular, we ignored merges (as no modifications to source code are applied) and commits that did not modify pattern instances. We clarify that the latter filtering is performed in the next step.

2) STEP 2: COLLECT PATTERN INSTANCES

This collection was performed for every commit, which is a time-consuming task. Hence, we automated this task using two tools. We first used the Design Pattern Detector (DPD, v4.12) [22], which is able to identify 12 GoF patterns: Adapter/Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State/Strategy, Template Method, and Visitor. We selected this tool because it covers a fair amount of design patterns that can be detected and it has adequate performance, as reported in Tsantalis et al. [22], also when compared to similar tools [23], [24]. To further validate the performance of the tool, we manually assessed 50 instances, which were all true positives.

Despite the performance of DPD, it detects only the main pattern-participant classes (i.e., those that provide the main structure of the pattern solution, commonly abstract classes). To detect the extended pattern-participants classes (i.e., the other classes that play a role in the pattern), we employed a tool developed in our group, name SSA+[1] (v1.0). This tool can detect and complement the output of DPD with ten extended pattern participants: *Concrete Creator and Product*, for Factory Method pattern; *Concrete Prototype*, for Prototype pattern; *Leaf*, for Composite pattern; *Concrete Decorator* and *Concrete Component*, for Decorator pattern; *Concrete Observer*, for Observer pattern; *Concrete State/Strategy*, for State/Strategy pattern; *Concrete Class*, for Template Method pattern; and *Subject*, for Proxy pattern. For that, SSA+ queries the abstract syntax tree (AST) of the system according to a set of rules to identify each extended pattern participant (e.g., inherit from a main pattern-participant class). As the task performed by SSA+ is deterministic (i.e., it identifies classes that comply with a set of rules), we validated it by manually checking the output for 50 randomly selected pattern instances, and no error was detected. In addition, SSA+ was similarly validated in another study [15].

Based on the collected information, we assign an ID (*inst_id*) for every instance and record it together with the type of the pattern (*pattern*). We note that IDs are assigned when instances are first detected and then reused when the same instance is detected again in later versions, i.e., they are persistent across versions of the project. Instances were considered equivalent if the main pattern participants had the same class name or matched a renamed version of the class (obtained from Git).

3) STEP 3: ASSESS PATTERN GRIME

For every unit of analysis (i.e., change to a pattern instance), we assessed the amount of pattern grime accumulated with regards to its three forms (i.e., class, modular and

[1] https://github.com/search-rug/ssap

organizational). For that, we selected six metrics, two for each form of grime, which were previously used and validated to assess pattern grime in non-trivial systems [10], allowing us to analyze its accumulation from various perspectives.

To assess class grime, we calculate: (a) number of alien attributes (*cg-naa*), i.e., that are not described in the original pattern; and (b) number of alien public methods (*cg-napm*). We clarify that we consider only public methods, as they are responsible for exposing functionality of the pattern instance to the whole system. For modular grime, we calculate: (a) afferent pattern coupling (*mg-ca*) that is the amount of in-coming dependencies (or fan-in), representing the responsibility of the pattern instance [10]; and (b) efferent pattern coupling (*mg-ce*) that is the amount of dependencies on classes external to the pattern instance (or fan-out), representing the instability of the pattern instance [10]. Organizational grime is assessed by calculating: (a) number of packages (*og-np*) that contain classes participating on the pattern instance; and (b) afferent coupling at package level (*og-ca*). Although afferent coupling is also calculated for modular grime (*mg-ca*), og-ca will depict the responsibility at a higher level of abstraction. For example, mg-ca may increase within the same package containing the pattern instance, which would not affect og-ca.

To automate the data collection of the aforementioned metrics, we used a tool developed in our group, namely *spoon-pttgrime*[2] (v0.1.0). This tool takes as input Java source files of a project and an XML file describing the pattern instances in the project (i.e., the output from SSA+). For each pattern instance, *spoon-pttgrime* calculates the six aforementioned metrics by querying the project's AST using the Spoon library [25]. To validate the tool, we verified the calculated metrics for 50 pattern instances that were randomly selected, and the results were all correct. Based on the collected information, we record the amount of grime accumulated according to each metric, i.e., the difference between two consecutive versions of the pattern instance.

We note that other indicators of grime have been proposed in the literature, which are based on taxonomies of modular and class grime [13, 14]. However, they are not independent grime indicators in the sense that they are subtypes of the indicators that we already investigate. Moreover, these additional indicators have been so far validated only through synthetic experiments [12, 13, 14], and there is no tool to automate their measurement. At the same time, the size of the population of our study makes it infeasible to assess them manually. Therefore, we decided to consider such indicators in our future work, and not include them in this study setup.

4) STEP 4: ASSESS QUALITY ATTRIBUTES

As mentioned in Section I, we estimated the studied quality attributes based on the number of violations of various coding practices. For that, we used FindBugs (v3.0.1), which considers bug patterns as rules to identify violation of good coding practices [26]. In particular, FindBugs organizes its rules (i.e., bug patterns) into nine high-level categories[3], from which five can be mapped into the studied quality attributes: correctness (Correctness and Multithreaded Correctness categories), performance (Performance category), and security (Security and Malicious Code categories). We note that, despite the name of the tool, we do not consider its output as bugs but simply as warnings, i.e., violations of good coding practices, and take them as indicators of quality. A similar approach was used by Kahlid et al. [27], who correlated the violations of three categories (one being performance) to quality as perceived by end-users. They found the data to be closely related, which supports the violations as quality indicators.

We selected FindBugs due to its collection of rules (252 of them regarding the considered categories), the possibility to map them into the studied quality attributes, as well as due to its adequate precision when compared to similar tools [26], [28], [29], which reflects on the relevance of the offered rules. Moreover, we analyzed and validated FindBugs in a previous study [30] and found that the precision can be noticeably improved by excluding violations with low level of confidence. To estimate the level for each quality attribute, we calculate the amount of rules violations in the pattern-participant classes of a unit of analysis (*cor-viol*, *per-viol*, and *sec-viol*). We clarify that lower numbers of violations reflect a higher level of quality.

D. ANALYSIS PROCEDURE

To investigate the collected data, we performed various statistical analyses. First, to answer RQ_1, we calculated the correlation between every pair of <*grime metric, quality indicator*> (e.g., pattern efferent coupling vs. performance violations). We assess the strength of the correlation according to the guidelines of Evans [31]: 'very weak' (0.00-0.19); 'weak' (0.20-0.39); 'moderate' (0.40-0.59); 'strong' (0.60-0.79); and 'very strong' (0.80-1.00). To select the most fitting method for correlation analysis, we first tested the normality of our data, using the Kolmogorov-Smirnov test [32], which is more appropriate for large samples. We clarify that for normally distributed variables, we used Pearson correlation method [32], otherwise, we used the Spearman's rank correlation method [32]. Moreover, the correlations calculated in this study do not entail bias from consecutive measurements with same value, also known as artificial boost. This is because every unit of analysis regards different states of a particular pattern instance. Therefore, consecutive measures with same value suggest that a specific metric is not designed to capture this particular change, and this information is relevant to our study.

[2] https://github.com/search-rug/spoon-pttgrime

[3] The categories are: Security, Correctness, Multithreaded Correctness, Performance, Malicious Code, Bad Practice, Internationalization, Experimental and Dodgy Code

To answer RQ$_2$, we performed the following steps for each factor (i.e., project, pattern, developer) that might affect the relation between pattern grime and quality. First, we grouped the dataset according to the factor. Next, we verified whether the groups differentiate between themselves with regards to the measured variables. For that, we performed an Analysis of Variance (ANOVA) [32] to confirm a disparity among groups, followed by post-hoc tests for pairwise comparisons. We note that we applied Levene's test [32] to assess the assumption of equal variances of the tested populations. When the assumption was met, we used regular ANOVA, followed by Tukey's Honestly Significant Differences (HSD) tests [32]. Otherwise, we used Welch's ANOVA, followed by Games-Howell tests (which are more appropriate for large samples). Finally, for the groups that are statistically different, we calculate the correlation for each pair of grime and quality metrics and identify statistically relevant correlations.

IV. RESULTS

In this section, we present a summary of the collected data, as well as the results of the analysis performed to answer the research questions posed in Section III.A. During the data collection, we identified 1,422 commits that contain the creation or modification of pattern-participating classes of the five investigated projects, from which the majority (94%) regard the modification of one or more pattern instances. Based on the commits, we isolated 2,329 pattern instances of eight different GoF patterns: (Object) Adapter / Command, Decorator, Factory Method, Observer, Singleton, State / Strategy, and Template Method. In Table III we present a summary of the units of analysis by project and patterns.

TABLE III
SUMMARY OF DATASET

Project	Pattern	Number of instances	Number of units of analysis
P1	(Object) Adapter/Command	284	8150
	Singleton	80	155
	State/Strategy	351	11586
P2	(Object) Adapter/Command	86	484
	Observer	3	21
	Singleton	16	42
	State/Strategy	275	2113
	Template Method	1	6
P3	(Object) Adapter/Command	327	3090
	Factory Method	53	545
	Singleton	29	152
	State/Strategy	375	3995
P4	(Object) Adapter/Command	136	2144
	Decorator	1	1
	Factory Method	13	266
	Singleton	21	73
	State/Strategy	230	3275
P5	(Object) Adapter/Command	16	206
	Factory Method	2	33
	Singleton	5	10
	State/Strategy	25	224

In Section III.B, we highlighted the necessity of having a balanced population (i.e., pattern instances should have similar number of modifications) to avoid bias from pattern instances with excessive number of units of analysis. After studying the history of commits, we assessed that each pattern instance underwent a maximum of 178 modifications. Moreover, 87% of the pattern instances (i.e., 2,039) were modified at least once, and 64% (i.e., 1,500) at least five times. Our analysis suggests that although the population is not evenly balanced, the discrepancies are not enough to harm the statistical analysis of our study nor the answers to the research questions.

In summary, we collected a total of 36,571 units of analysis (i.e., creation/modification of a pattern instance in a commit). For each unit, we recorded the amount of pattern grime that was accumulated according to six metrics (*cg-**, *mg-** and *og-**) and the number of violations regarding the three studied quality attributes (**-viol*). We clarify that due to a non-disclosure agreement signed with the company in this case study, we cannot share the created dataset, nor certain details regarding specific projects and developers.

To characterize our population, in Table IV we present the descriptive statistics for these variables. We notice that pattern efferent coupling (*mg-ce*) is the grime metric that changes the most, which may be a sign of bad practices since it represents the dependency of the pattern instance on other classes. On the counterpart, number of packages (*og-np*) is the metric that changes the least, which is expected given that pattern instances normally grow within the same package. Furthermore, we notice that violations of good practices regarding correctness appear to be considerably more frequent than regarding performance and security. This observation may be partially related to the fact that the majority of the rules checked by FindBugs concern correctness: out of all the rules for the three studied qualities, correctness accounts for approx. 70%, while performance and security correspond to approx. 15% each. Nevertheless, we could detect considerably fewer violations concerning security rather than performance, which suggest that other parameters are also relevant, such as the type of application or even the specific security-related violations that FindBugs checks.

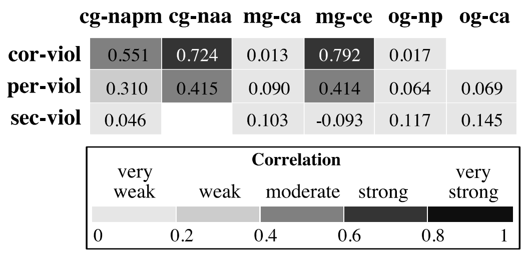
TABLE IV
DESCRIPTIVE STATISTICS PER COMMIT

Variable	Minimum	Maximum	Mean	Std. Deviation
cg-napm	0.00	52.80	13.66	8.27
cg-naa	0.00	41.50	7.95	5.00
mg-ca	0.00	107.00	8.01	10.72
mg-ce	0.00	325.00	100.18	61.22
og-np	1.00	5.00	2.34	0.53
og-ca	0.00	54.00	6.75	8.06
cor-viol	0.00	35.00	2.39	3.19
per-viol	0.00	8.00	0.97	1.45
sec-viol	0.00	20.00	0.25	1.40

A. RQ$_1$ – CORRELATION BETWEEN GRIME AND QUALITY ATTRIBUTES

To answer RQ$_1$, we calculated the correlation between all pairs of <*grime metric, quality indicator*> (e.g., *cg-ce* vs.

per-viol) as explained in Section III.D. We note that we could not assume normal distribution for all variables and, thus, we used Spearman's rank correlation method. Moreover, 'artificial boost' is not a concern in this population (see Section III.D). Fig. 1 depicts a heatmap with the results of our analysis, in which darker shades of gray denote stronger correlation. The coefficients are written within each cell except for correlations that are not statistically significant (which are blank). Based on Fig. 1 we can make several observations.



	cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca
cor-viol	0.551	0.724	0.013	0.792	0.017	
per-viol	0.310	0.415	0.090	0.414	0.064	0.069
sec-viol	0.046		0.103	-0.093	0.117	0.145

Correlation
very weak
weak
moderate
strong
very strong
0
0.2
0.4
0.6
0.8
1

FIGURE 1. **Correlation between grime metrics (*cg-*, mg-*, og-**) and quality attributes indicators (**-viol*).**

The accumulation of grime seems to be related with the depreciation of correctness and performance (i.e., more violations), as we observed strong correlations (i.e., above 0.6) and moderate correlations (i.e., between 0.4 and 0.59) respectively [31]. Furthermore, the very weak correlation with security violations (i.e., below 0.2) does not imply that a link does not exist. This only shows a lack of evidence.

Another observation is that **metrics for assessing *class grime, namely number of alien public methods (cg-napm), alien attributes (cg-naa), and pattern efferent coupling (mg-ce)* displayed the strongest correlations regarding every quality attribute**. This outcome can be considered intuitive in the sense that, as structural elements at the class level, patterns are expected to be more influential at lower levels of granularity (e.g., class rather than module). The degradation of another quality, namely maintainability, due to the existence of alien methods is also reported in related work [9].

The aforementioned observations are based on how grime accumulates in pattern instances. However, one may wonder if changes in the quality levels can be simply explained by natural evolution of the source code, i.e., any type of change to the pattern instance rather than pattern grime. To investigate this possibility, we assessed the correlation between lines of code (*LOC*) and both grime metrics and quality indicators. The results show that grime is strongly correlated (0.81) with *LOC*, i.e., most maintenance activities in pattern instances entail accumulation of grime. However, the correlation between grime and quality indicators was often slightly stronger compared to the correlation between *LOC* and quality indicators. For example, the correlation between *cor-viol* vs. *mg-ce* (0.792) is marginally stronger than *cor-viol* vs. *LOC* (0.785), *per-viol* vs. *mg-ce* (0.414) is stronger than *per-viol* vs. *LOC* (0.359), and *sec-viol* vs. *mg-ce* (-0.093) is stronger than sec-viol vs. *LOC* (-0.074). Therefore, although the difference between correlation values may be marginal at times, the overall analysis consistently shows that ***grime matches the degradation of quality better than natural evolution***.

B. RQ$_2$ – ANALYSIS OF FACTORS

To further explore the relation between the accumulation of pattern and the three studied quality attributes, we investigated three factors that may influence the observed correlations as described in Section III.D: projects, patterns and developers.

1) COMPARISON OF PROJECTS

We collected data from five different industrial projects, here referred to as P1 to P5. From the 36,571 units of analysis, 19,891 regard P1, 2,667 regard P2, 7,781 regard P3, 5,759 regard P4, and 473 regard P5. Moreover, P1 and P2 were developed by two different teams of developers while a third team developed P3, P4 and P5. In Table V, we show the descriptive statistics of all variables for each project independently.

We notice that the projects are considerably distinct from each other with regard to these variables. For example, P2 has the highest mean for most grime metrics but not for quality indicators, while P5 has the lowest means for grime metrics but present the highest average of performance violations. To verify the observed differences, we compared the means between projects by performing an analysis of variances (ANOVA) for each variable, followed by one post-hoc test for each pairwise comparison (i.e., 90 in total). The results of the tests are publicly available online in a supplementary material[4]. The results show that 91% of the tests are statistically significant, i.e., the means differ between the two compared projects. Based on these findings, we hypothesize that the different characteristics of projects are indeed reflected on the relationship between the accumulation of pattern grime and the indicators of correctness, performance and security.

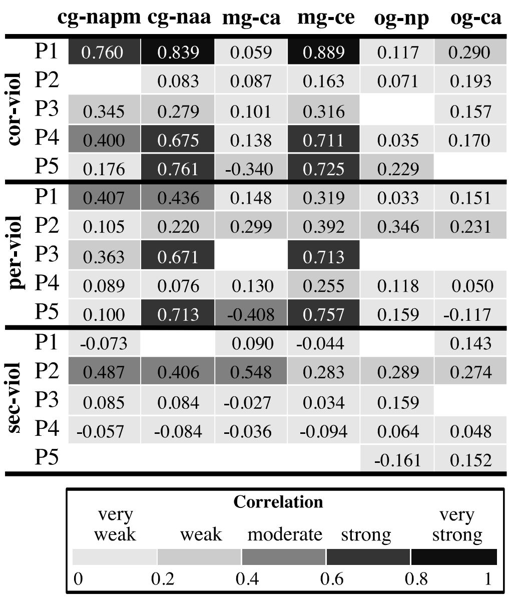
To verify how the accumulation of grime in projects relates to the levels of quality, we calculated the correlation between every pair of grime metric and quality indicator for each project. The results are presented in Fig. 2 (which is interpreted as Fig. 1), from which we observe that the correlations are noticeably different based on the projects. For example, similar to the results observed for the general population, P1 exhibits a strong correlation (i.e., above 0.6) between class grime metrics and the correctness indicator (*cor-viol*). The opposite is observed for P2, for which the data suggest a correlation between pattern grime and the security indicator (*sec-viol*), which have not been observed for the general population.

[4] https://doi.org/10.5281/zenodo.1133552

TABLE V
DESCRIPTIVE STATISTICS PER PROJECT

Variable	Project	Minimum	Maximum	Mean	Std. Deviation
cg-napm	P1	0.00	44.67	13.94	8.26
	P2	0.00	48.00	14.92	11.90
	P3	0.00	52.80	11.79	6.47
	P4	0.00	42.00	14.89	8.13
	P5	0.00	29.75	10.36	5.73
cg-naa	P1	0.00	22.33	8.97	4.87
	P2	0.00	30.33	5.87	3.72
	P3	0.00	30.50	5.90	3.81
	P4	0.00	41.50	8.45	6.13
	P5	0.00	13.75	4.72	2.50
mg-ca	P1	0.00	49.00	6.52	8.48
	P2	0.00	78.00	17.69	18.65
	P3	0.00	107.00	7.44	10.29
	P4	0.00	102.00	10.02	10.85
	P5	0.00	5.00	1.00	1.07
mg-ce	P1	0.00	325.00	132.05	62.91
	P2	0.00	140.00	55.07	28.23
	P3	0.00	162.00	63.31	29.65
	P4	0.00	141.00	65.50	27.99
	P5	0.00	95.00	43.13	19.02
og-np	P1	1.00	3.00	2.31	0.48
	P2	1.00	3.00	2.64	0.53
	P3	1.00	5.00	2.30	0.60
	P4	1.00	4.00	2.35	0.54
	P5	1.00	3.00	2.21	0.46
og-ca	P1	0.00	54.00	5.55	8.32
	P2	0.00	29.00	15.42	8.98
	P3	0.00	44.00	7.71	7.50
	P4	0.00	19.00	6.03	3.95
	P5	0.00	5.00	0.88	0.55
cor-viol	P1	0.00	35.00	3.74	3.68
	P2	0.00	5.00	0.16	0.55
	P3	0.00	7.00	0.66	1.19
	P4	0.00	10.00	1.24	1.25
	P5	0.00	2.00	0.63	0.65
per-viol	P1	0.00	7.00	1.16	1.58
	P2	0.00	8.00	1.12	1.50
	P3	0.00	6.00	0.72	1.15
	P4	0.00	5.00	0.57	1.10
	P5	0.00	8.00	1.16	1.66
sec-viol	P1	0.00	2.00	0.03	0.18
	P2	0.00	14.00	1.31	2.65
	P3	0.00	14.00	0.31	1.33
	P4	0.00	20.00	0.42	2.41
	P5	0.00	8.00	0.28	0.89

However, we also noticed that higher values of accumulated grime are related to higher depreciation of quality (i.e., higher number of violations), which is often reflected in higher correlation coefficients. ***This evidence strengthens our finding that the relationship between pattern grime and quality attribute indicators is project-dependent***. It also suggests that the observed difference is connected to how grime accumulates in the different projects. This finding is in accordance to those of Vasquez et al. [33], which suggest that other indirect quality indicators (such as anti-patterns or code smells) vary among projects of different application domains, as well as with Izurieta and Bieman [10], who observed varied levels of grime and quality on the studied projects.



		cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca
cor-viol	P1	0.760	0.839	0.059	0.889	0.117	0.290
	P2		0.083	0.087	0.163	0.071	0.193
	P3	0.345	0.279	0.101	0.316		0.157
	P4	0.400	0.675	0.138	0.711	0.035	0.170
	P5	0.176	0.761	-0.340	0.725	0.229	
per-viol	P1	0.407	0.436	0.148	0.319	0.033	0.151
	P2	0.105	0.220	0.299	0.392	0.346	0.231
	P3	0.363	0.671		0.713		
	P4	0.089	0.076	0.130	0.255	0.118	0.050
	P5	0.100	0.713	-0.408	0.757	0.159	-0.117
sec-viol	P1	-0.073		0.090	-0.044		0.143
	P2	0.487	0.406	0.548	0.283	0.289	0.274
	P3	0.085	0.084	-0.027	0.034	0.159	
	P4	-0.057	-0.084	-0.036	-0.094	0.064	0.048
	P5					-0.161	0.152

FIGURE 2. Correlation between grime metrics (*cg-, *mg-**, *og-**) and quality attributes indicators (**-viol*) for individual projects (*P**).**

2) COMPARISON OF PATTERNS

During the data collection, we identified instances of eight different patterns. From the 36,571 units of analysis, 14,074 regard the (Object) Adapter / Command (AC) patterns, 844 regard the Factory Method (FM) pattern, 432 regard the Singleton (Si) pattern, 21,193 regard the State/Strategy (SS) patterns, 21 regard the Observer pattern, six regard the Template Method pattern, and one regards the Decorator pattern. Due to the limited amount of units, we do not present results concerning the last three patterns, which are available in the supplementary material.

In Table VI, we present the descriptive statistics of all variables for each pattern independently. We notice that this factor also seems to influence the relations between pattern grime and indicators of the studied quality attributes. In particular, we observe that the means for every metric varies considerably among patterns. Moreover, we could not observe clear trends, i.e., patterns that consistently display the highest or lower means. For example, Factory Method displays the highest mean of security violations (*sec-viol*) but one of the lowest of correctness violations (*cor-viol*).

To verify our observations, we computed the ANOVA for each variable and performed the post-hoc tests (i.e., 48 in total). The results of the tests are available in the supplementary material. We note that Singleton instances had no variance with regards to number of packages (*og-np*) and security indicator (*sec-viol*), and, thus, these variables were not considered in the analyses for this pattern. The results show that 93% of the tests are statistically significant.

To further investigate this factor, we calculated the correlation between every pair of grime metric and quality indicator for the investigated patterns. In Fig. 3 (which is interpreted as Fig. 1), we present the results of the calculations, which show clearly varying correlations depending on the pattern. We notice that, as for projects, we could identify a pattern, namely Factory Method, for which the accumulation of grime is moderately correlated with the depreciation of quality indicators. Again, we observed that the combination of ***higher accumulation of grime and quality indicators often reflects in higher correlation coefficients***. All this information suggests that the ***relationship between pattern grime and quality attribute indicators also depends on the pattern type of the instance***. This finding is in accordance with the literature, which suggests that different patterns have different effects on the same quality attribute (e.g., [4], [34]).

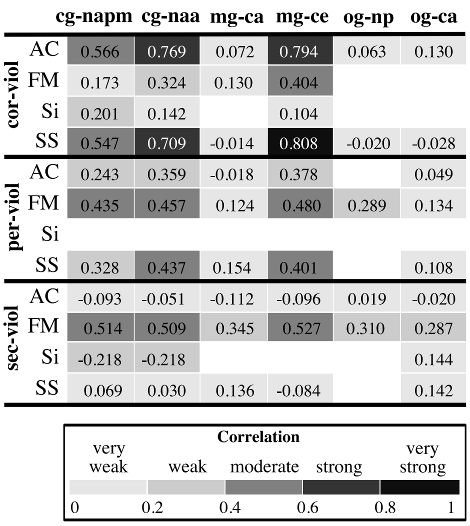
TABLE VI
DESCRIPTIVE STATISTICS PER PATTERN

Variable	Pattern	Minimum	Maximum	Mean	Std. Deviation
cg-napm	AC	0.00	43.00	12.59	7.70
	FM	2.40	52.80	15.63	8.89
	Si	0.00	6.00	0.82	1.35
	SS	0.67	48.00	14.55	8.39
cg-naa	AC	0.50	41.50	8.53	5.78
	FM	1.00	23.20	6.86	4.43
	Si	0.00	3.00	0.44	0.73
	SS	0.00	30.33	7.77	4.32
mg-ca	AC	0.00	49.00	6.57	8.71
	FM	1.00	107.00	19.70	23.46
	Si	0.00	52.00	7.39	10.54
	SS	0.00	78.00	8.51	10.80
mg-ce	AC	5.00	236.00	90.24	51.97
	FM	10.00	129.00	58.00	25.67
	Si	0.00	17.00	1.68	2.52
	SS	1.00	325.00	110.56	64.80
og-np	AC	1.00	2.00	1.99	0.10
	FM	1.00	5.00	3.37	0.68
	Si	1.00	1.00	1.00	0.00
	SS	1.00	4.00	2.56	0.50
og-ca	AC	0.00	46.00	5.82	7.72
	FM	1.00	44.00	13.34	8.15
	Si	0.00	54.00	13.06	8.03
	SS	0.00	46.00	6.96	8.07
cor-viol	AC	0.00	13.00	2.19	2.84
	FM	0.00	5.00	0.69	1.02
	Si	0.00	3.00	0.02	0.21
	SS	0.00	35.00	2.64	3.44
per-viol	AC	0.00	5.00	0.82	1.30
	FM	0.00	6.00	0.56	1.22
	Si	0.00	0.00	0.00	0.00
	SS	0.00	8.00	1.11	1.54
sec-viol	AC	0.00	6.00	0.04	0.26
	FM	0.00	20.00	4.71	5.81
	Si	0.00	12.00	0.48	1.67
	SS	0.00	14.00	0.21	1.05

AC = (Object)Adapter/Command; FM = Factory Method; Si = Singleton; SS = State/Strategy

3) COMPARISON OF DEVELOPERS

The case study involved 16 developers, here referred to as D1 to D16, which account for various amounts of units of analysis[5]. Due to the low number of data points, we did not include D4, D7, D8 and D10 in our analyses. In Table VII, we show the mean value of all variables, for each developer. We note that we do not present the complete descriptive statistics, which are available in the supplementary material. Similar to the previous factors, we observe that **mean *values regarding all variables differ among developers, i.e., they exhibit different characteristics***. For example, both D11 and D15 show higher tendency to pollute pattern instances with alien methods (i.e., higher cg-napm values) than other developers. However, D11 seems much less prone to pollute instances with external dependencies (i.e., lower mg-ce).



		cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca
cor-viol	AC	0.566	0.769	0.072	0.794	0.063	0.130
	FM	0.173	0.324	0.130	0.404		
	Si	0.201	0.142		0.104		
	SS	0.547	0.709	-0.014	0.808	-0.020	-0.028
per-viol	AC	0.243	0.359	-0.018	0.378		0.049
	FM	0.435	0.457	0.124	0.480	0.289	0.134
	Si						
	SS	0.328	0.437	0.154	0.401		0.108
sec-viol	AC	-0.093	-0.051	-0.112	-0.096	0.019	-0.020
	FM	0.514	0.509	0.345	0.527	0.310	0.287
	Si	-0.218	-0.218				0.144
	SS	0.069	0.030	0.136	-0.084		0.142

FIGURE 3. Correlation between grime metrics (*cg-*, mg-*, og-) and quality attributes indicators (**-viol*) for individual patterns (AC, FM, Si, and SS).**

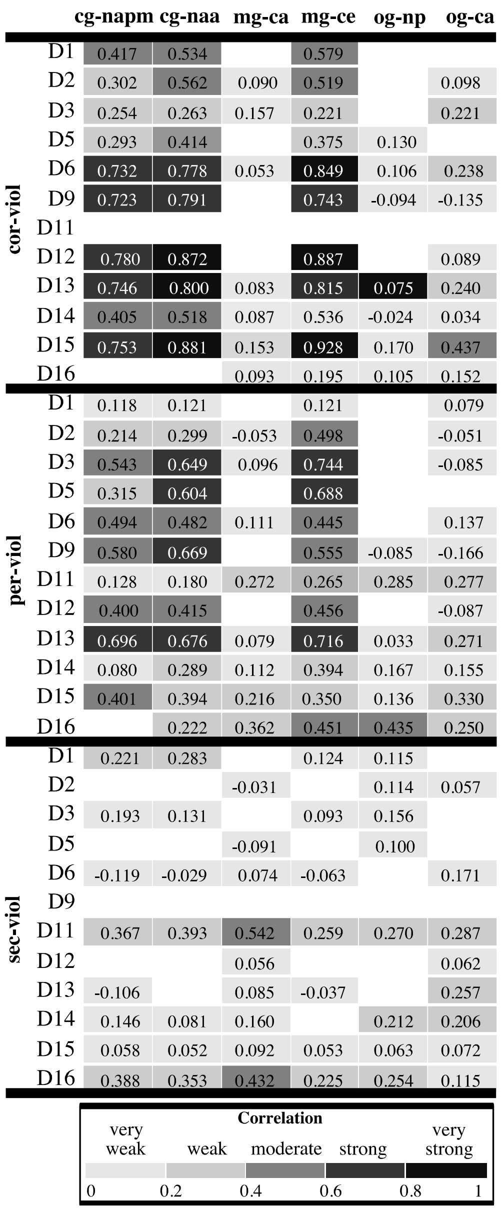
TABLE VII
DESCRIPTIVE STATISTICS PER DEVELOPER

	cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca	cor-viol	per-viol	sec-viol
D1	9.50	3.17	8.40	38.09	2.28	7.07	0.47	0.00	0.25
D2	12.53	7.17	8.02	63.75	2.32	7.21	1.03	0.56	0.34
D3	9.83	7.11	7.68	70.78	2.30	8.59	0.26	1.14	0.29
D5	12.58	5.84	5.48	56.39	2.27	5.08	0.45	0.97	0.27
D6	14.01	9.15	7.00	127.50	2.30	5.65	3.57	1.59	0.05
D9	9.62	6.76	6.41	99.90	2.27	5.02	2.21	0.87	0.00
D11	16.36	5.88	14.27	51.18	2.59	13.07	0.07	0.83	0.96
D12	12.83	7.92	8.03	122.20	2.38	7.20	2.99	1.11	0.01
D13	12.42	8.36	7.21	112.66	2.26	5.49	2.61	1.83	0.08
D14	14.81	6.99	11.12	64.40	2.40	8.06	0.89	0.88	0.65
D15	15.65	9.78	5.57	152.28	2.34	5.28	4.83	0.64	0.01
D16	14.31	5.66	16.25	53.53	2.60	15.48	0.17	0.75	0.96

To validate the differences observed in the measurements, we performed ANOVA on all variables, followed by the post-hoc tests (583 in total), which are all available in the supplementary material. We note that no variance in the security indicator (*sec-viol*) was observed for D9 and, thus, we discarded this variable for analyses regarding the developer. The results show that 80% of the tests are statistically significant. The majority of the comparisons that were not significant, concern the number of packages, which is intuitive, as pattern instances do not tend to be spread across multiple packages/namespaces.

[5] The number of units by each developer is: D1 - 810; D2 - 5662; D3 - 1535; D4 - 8; D5 - 470; D6 - 5368; D7 - 21; D8 - 62; D9 - 1464; D10 - 11; D11 - 811; D12 - 1648; D13 - 3565; D14 - 6748; D15 - 7825; D16 - 563.

We also calculated the correlation between variables, which are shown in Fig. 4 (which is interpreted as Fig. 1). The results suggest that developers accumulate grime differently and that this may reflect on the quality indicators. We also observed that **although we found that correlations differ among developers, they are mostly consistent in the sense that more grime is correlated with more violations** (i.e., depreciated quality). In summary, all collected information strengthens our finding that developers comprise a factor to how the accumulation is related to the depreciation of correctness, performance and security in pattern instances. Our results are in accordance with those by Amanatidis et al. [17], who studied the accumulation of technical debt and observed an imbalance regarding the number of violations among developers.



		cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca
cor-viol	D1	0.417	0.534		0.579		
	D2	0.302	0.562	0.090	0.519		0.098
	D3	0.254	0.263	0.157	0.221		0.221
	D5	0.293	0.414		0.375	0.130	
	D6	0.732	0.778	0.053	0.849	0.106	0.238
	D9	0.723	0.791		0.743	-0.094	-0.135
	D11						
	D12	0.780	0.872		0.887		0.089
	D13	0.746	0.800	0.083	0.815	0.075	0.240
	D14	0.405	0.518	0.087	0.536	-0.024	0.034
	D15	0.753	0.881	0.153	0.928	0.170	0.437
	D16			0.093	0.195	0.105	0.152
per-viol	D1	0.118	0.121		0.121		0.079
	D2	0.214	0.299	-0.053	0.498		-0.051
	D3	0.543	0.649	0.096	0.744		-0.085
	D5	0.315	0.604		0.688		
	D6	0.494	0.482	0.111	0.445		0.137
	D9	0.580	0.669		0.555	-0.085	-0.166
	D11	0.128	0.180	0.272	0.265	0.285	0.277
	D12	0.400	0.415		0.456		-0.087
	D13	0.696	0.676	0.079	0.716	0.033	0.271
	D14	0.080	0.289	0.112	0.394	0.167	0.155
	D15	0.401	0.394	0.216	0.350	0.136	0.330
	D16		0.222	0.362	0.451	0.435	0.250
sec-viol	D1	0.221	0.283		0.124	0.115	
	D2			-0.031		0.114	0.057
	D3	0.193	0.131		0.093	0.156	
	D5			-0.091		0.100	
	D6	-0.119	-0.029	0.074	-0.063		0.171
	D9						
	D11	0.367	0.393	0.542	0.259	0.270	0.287
	D12			0.056			0.062
	D13	-0.106		0.085	-0.037		0.257
	D14	0.146	0.081	0.160		0.212	0.206
	D15	0.058	0.052	0.092	0.053	0.063	0.072
	D16	0.388	0.353	0.432	0.225	0.254	0.115

FIGURE 4. Correlation between grime metrics (*cg-, *mg-**, *og-**) and quality attributes indicators (**-viol*) for individual developers (*D**).**

V. DISCUSSION

In this section, we revisit the findings of our study and present their connection to related work. Next, we elaborate on the main implications to researchers and practitioners.

A. INTERPRETATIONS OF RESULTS

1) CORRELATION BETWEEN GRIME AND ATTRIBUTES

The findings discussed in this paper suggest that, as pattern grime accumulates, **classes that participate in pattern instances become more prone to quality depreciation**. In particular, such classes are more susceptible to source code that violates good practices that promote correctness, performance and security of software systems. These findings corroborate those by related work that analyzes the relations between grime and quality [10], [11], [13], in the sense that ***we also found that grime goes hand in hand with diminished quality***.

In our study, ***we noticed that three metrics, namely number alien attributes (cg-naa), number of alien public methods (cg-napm) and instance efferent coupling (mg-ce), were the most likely to be appropriate indicators of bad quality***; these same metrics have shown similar relevance in the related work. Moreover, these metrics correspond to structural characteristics of pattern instances (e.g., efferent coupling), and similar metrics (at class level rather than instance level) have been largely explored in the literature (e.g. [6]–[9]) and found to be good estimators of the benefit (or harmfulness) of pattern instances to quality attributes. In a previous study, we found that the degradation of certain well-known design metrics can be used as hints of the accumulation of pattern grime [15], as it is assessed based on design propertied of pattern participants. In particular, we investigated the metric suits proposed by Chidamber and Kemerer [35], Li and Henry [36], and Bansiya and Davis [20]. Results of that study showed that the metrics data abstraction coupling (DAC) [36] and measure of aggregation (MOA) [20] may help identifying accumulation of *cg-naa*; the metrics weighted methods per class (WMC) [35] and class interface size (CIS) [20] may help identifying accumulation of *cg-napm*; and the metrics coupling between object classes (CBO) [35] and response for a class (RFC) [35] may help identifying accumulation of *mg-ce*.

2) CONTRIBUTING FACTORS

The way pattern grime builds up in pattern-participant classes can depend on several factors. Our empirical investigation confirmed that three such factors indeed play a role: project, pattern and developer. With regard to projects, we observed that the difference may be related to two sub-factors. The ***type of the project seems relevant on determining the relation between grime and quality***. Two of the studied projects (P1 and P4) provide services to other applications (e.g., libraries or API's) and showed to be more prone to grime and violations; this aligns with the suggestion by

Vasquez et al. [31] that parameters such as application domain can be relevant. However, we also noticed that these projects had more pattern instances (i.e., a bigger pattern code base) and that a second sub-factor, *namely* ***lines of code was also correlated with both grime and quality indicators***; this has also been discerned by Izurieta and Bieman [10].

A similar observation also holds for developers: ***those that wrote more code (i.e., provided more units of analysis) were more prone to incur both grime and violations***. Finally, our main observation concerning the difference among patterns is that those using ***more complex mechanisms (e.g., State, Strategy and Factory Method, which have polymorphic calls) tend to accumulate both more grime and violations***; this is intuitive given that more complex designs are less understandable and harder to maintain.

Investigating the factors in isolation allowed us to observe that the ***correlations in different groups (based on factors) differ from the ones concerning the entire dataset***. However, although the differences may look random at first, we noticed a recurrent motif. In particular, we observed that the majority (approx. 80%) of ***moderate or strong correlations*** *(i.e., more than 0.4) [31]* ***have been identified when grime and qualities metrics are at a similar level***. For example, projects that on average concentrate few violations and have low levels of accumulated grime, or the opposite. Among those, 56% regard higher values on both grime and quality indicators.

3) ANALYSIS OF VIOLATIONS

Finally, since we estimated the levels of quality attributes through the number of violations of good coding practices, it is relevant to dig deeper into these violations. In Table VIII, we present the most recurrent violations, assessed according to the addressed research questions, i.e., the overall dataset, per project, per pattern and per developer. We note that some developers have not violated any rules for certain quality attributes in pattern-participant classes; those are marked with "-". We observe that this ***list of violations comprises issues that are clearly harmful to the respective quality attributes***, e.g., calling unsafe methods in a multithreaded context can lead to race conditions or unpredictable states.

Thus, if these violations are among the recurrent ones, they can pose a serious threat to the system. Furthermore, the ***top issues vary among projects, patterns and developers***. The differences that we observe between developers is aligned with the findings by Amanatidis et al. [17], who not only observed an imbalance on how developers accumulate violations but also a difference on the recurrence. Nevertheless, ***it is possible to discern the connection between groups.*** For example, the two recurrent performance issues that appear for the most among developers (i.e., "Comparison of different types" and "Possible null pointer dereference"), also appear frequently among projects and patterns, and one of them is the most recurrent in the entire dataset.

TABLE VIII
MOST RECURRENT VIOLATIONS

		Correctness	Performance	Security
Overall		Comparison of different types	Class member should be static	Exposed inner representation by incorporating mutable object
Project	**P1**	Comparison of different types	Class member should be static	Method invocation without proper security check
	P2	Unsafe call in for multithreading	Class member should be static	Exposed inner representation by returning mutable object
	P3	Possible null pointer dereference	Private method is never called	Exposed inner representation by incorporating mutable object
	P4	Unsafe call for multithreading	Unnecessary value unboxing	Exposed inner representation by incorporating mutable object
	P5	Unsafe call for multithreading	Unnecessary call to toString()	Exposed inner representation by incorporating mutable object
Pattern	**AC**	Comparison of different types	Class member should be static	Field should be package protected
	FM	Possible null pointer dereference	Unnecessary value unboxing	Exposed inner representation by incorporating mutable object
	Si	Comparison of different types	Inefficient use of map iterator	Exposed inner representation by incorporating mutable object
	SS	Comparison of different types	Class member should be static	Exposed inner representation by returning mutable object
Developer	**D1**	Comparison to null	-	Exposed inner representation by incorporating mutable object
	D2	Possible null pointer dereference	Unnecessary value unboxing	Exposed inner representation by incorporating mutable object
	D3	Possible null pointer dereference	Class member should be static	Exposed inner representation by incorporating mutable object
	D5	Unsafe call for multithreading	Unnecessary value unboxing	Exposed inner representation by incorporating mutable object
	D6	Comparison of different types	Private method is never called	Method invocation without proper security check
	D9	Possible null pointer dereference	Private method is never called	-
	D11	Variable self-assignment	Class member should be static	Exposed inner representation by incorporating mutable object
	D12	Possible null pointer dereference	Class member should be static	-
	D13	Nullcheck on dereferenced variable	Invoke of inefficient constructor	Method invocation without proper security check
	D14	Unsafe call for multithreading	Class member should be static	Exposed inner representation by returning mutable object
	D15	Comparison of different types	Private method is never called	-
	D16	Unsafe call for multithreading	Class member should be static	Exposed inner representation by incorporating mutable object

AC = (Object)Adapter/Command; FM = Factory Method; Si = Singleton; SS = State/Strategy

B. IMPLICATIONS TO RESEARCHERS AND PRACTITIONERS

GoF patterns are popular among practitioners as established and valuable design solutions. However, the consequences of using them often become a matter of concern, especially regarding quality. This paper sheds some light on this respect, suggesting the following implications to practitioners. We encourage the ***conscious usage of GoF patterns***, in the sense that knowledge about the patterns being applied, as well as the pattern instances in the system under development, should be disseminated within the team of developers.

In addition, ***monitoring the pattern instances is of paramount importance to maintain desired levels of quality***, especially correctness, performance and security. Moreover, practitioners can take advantage of the tool *spoon-pttgrime* in order to track the accumulation of grime and plan maintenance activities. Conversely, if practitioners already use FindBugs within their development process, the number of violations (for correctness, performance and security) can be used as indicators of grime accumulation, helping the team on identifying pattern instances with potentially deteriorated design.

The findings in this paper can also benefit researchers. Our work joins the small pool of studies that investigate pattern grime, especially its relation to quality attributes, and further demonstrate the relevance of researching this phenomenon and the underlying relations. In particular, ***we provide evidence that encourages the investigation of other quality attributes, as well as factors related to it***. The presented information also builds up on the body of knowledge on pattern grime, and we hope it will support future research. Particularly, we envisage confirmatory studies to seek more evidence to explain the observed variations in the relationship between grime and the studied quality attribute, as well as others. We also demonstrate that the usage of static analysis tools such as FindBugs can provide valuable information regarding the accumulation of grime. Finally, the design of our case study and used tools used can be exploited for future research efforts.

VI. THREATS TO VALIDITY

In this section, we discuss threats to the validity of the study reported on this paper; in particular, construct validity, reliability and external validity. Construct validity concerns to what extent the objects of the study are connected to the research questions. Reliability regards the extent to which the study can be replicated with the same observed results. External validity pertains to the limitations to generalize our findings to the entire population. We note that we do not analyze internal validity, as we empirically study the correlation between variables without establishing causal relations.

Regarding *construct validity*, we identified the following threats. First, the DPD and FindBugs tools are limited by their precision and recall, which may bias our results due to false positives and negatives. We note that, to the best of our knowledge, these tools have adequate performance and good reputation (see Sections III.C.2 and III.C.4). Nevertheless, to mitigate this threat, we randomly selected 50 pattern instances and verified the output from each tool manually. In addition, we acknowledge that the list rules provided by FindBugs is by no means exhaustive and additional rules could affect our results. However, we reiterate that the diverse list of bug patterns (i.e., 252 rules) and evidence provided by other studies that used FindBugs to estimate quality attributes [27], [30] suggest that the tool is adequate for the purpose used in this study. Finally, concerning the tools developed in our group (SSA+ and *spoon-pttgrime*), which although perform deterministic tasks, may contain bugs and bias the results of the study. To mitigate this threat, we also checked their output for 50 randomly selected pattern instances. In addition, our tools have been used in previous studies, where they were also validated.

To address *reliability threats*, at least two researchers were involved in both data collection and analysis. Samples of the output were checked by both researchers and the verification followed a checklist to avoid irregularities. Furthermore, most tasks were automated by the tools referenced in this paper, which are all publicly available. Despite our effort, we acknowledge that non-disclosure agreements do not allow us to share the collected dataset. However, replications studies can be carried out to attempt to replicate our results.

Concerning *external validity*, the main threat is that we explored projects from the same company, from which three were developed by the same team. Such uniformity (e.g., developers subject to same company practices) may lessen the generalizability of our findings to other companies or teams. However, we note that the accumulation of grime that we observed aligns with the results of other studies, e.g., class and modular grime are the main indicators of grime. Moreover, we also aimed at identifying variations in the relationship between pattern grime and quality attributes based on project and developer, which we identified successfully despite the “uniformity” of our subjects. The other threats regard limitations of our study design. In particular, we investigated a limited number of patterns and subjects, and we acknowledge that additions to the population may affect our findings. Furthermore, we investigated projects developed in Java and our observations cannot be generalized to other languages without additional analyses. Finally, the grime metrics and quality indicators are estimators, and the usage of different variables may affect the observed results. Specifically, the inclusion of metrics based on subtypes of grime could provide more refined observations.

VII. CONCLUSIONS

In this paper, we reported on an exploratory case study with five industrial software systems, in which we examined the relationship between the accumulation of pattern grime and the levels of three quality attributes, namely correctness, performance and security. For that, we considered six metrics regarding the three forms of grime (i.e., class, modular and organizational), and one indicator of each studied quality attribute, estimated by the amount of violations of coding practices in pattern-participant classes. We investigated the evolution of 2,329 pattern instances over 1,422 commits, totalizing 36,571 units of analysis, in which we assessed the correlations between the grime metrics and quality indicators. Moreover, we sought to analyze factors that might influence the observed correlations, in particular, projects, pattern types, and developers.

The results suggest that pattern grime is related to the depreciation of correctness, performance and security in pattern instances. These findings are based on both class and modular grime, whilst no strong evidence is observed based on organizational grime. The results also suggest that all three examined factors can influence the relationship between pattern grime and quality attributes.

Based on our findings, we envisage several opportunities for future work. Confirmatory empirical studies could investigate one or more of the explored factors in more details, and seek evidence to explain the observed variations in the relationship between grime and quality attributes. Furthermore, a replication study with open-source systems could increase the external validity of the results reported on this paper. Finally, investigation of additional grime metrics and factors could enhance the understanding over the consequences of accumulating pattern grime. In particular, metrics regarding subtypes of grime have been proposed in the literature and it would be interesting to investigate the interplay between indicators of the types and subtypes of grime in similar study settings.

REFERENCES

[1] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, vol. 206. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] B. Bafandeh Mayvan, A. Rasoolzadegan, and Z. Ghavidel Yazdi, "The state of the art on design patterns: A systematic mapping of the literature," *J. Syst. Softw.*, vol. 125, pp. 93–118, Mar. 2017. DOI: 10.1016/j.jss.2016.11.030.

[3] F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "Playing roles in design patterns: An empirical descriptive and analytic study," in *Proc. 25th IEEE Int. Conf. Software Maintenance (ICSM '09)*, Edmonton, AB, Canada, 2009, pp. 83–92. DOI: 10.1109/ICSM.2009.5306327.

[4] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou, "The effect of GoF design patterns on stability: A case study," *IEEE Trans. Softw. Eng.*, vol. 41, no. 8, pp. 781–802, Aug. 2015. DOI: 10.1109/TSE.2015.2414917.

[5] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Research state of the art on GoF design patterns: A mapping study," *J. Syst. Softw.*, vol. 86, no. 7, pp. 1945–1964, Jul. 2013. DOI: 10.1016/j.jss.2013.03.063.

[6] B. Huston, "The effects of design pattern application on metric scores," *J. Syst. Softw.*, vol. 58, no. 3, pp. 261–269, Sep. 2001. DOI: 10.1016/S0164-1212(01)00043-7.

[7] T. Muraki and M. Saeki, "Metrics for applying GOF design patterns in refactoring processes," in *Proc. 4th Int. Workshop Principles of Software Evolution (IWPSE '02)*, Vienna, Austria, 2002, pp. 27–36. DOI: 10.1145/602461.602466.

[8] N.-L. Hsueh, P.-H. Chu, and W. Chu, "A quantitative approach for evaluating the quality of design patterns," *J. Syst. Softw.*, vol. 81, no. 8, pp. 1430–1439, Aug. 2008. DOI: 10.1016/j.jss.2007.11.724.

[9] S. Charalampidou, A. Ampatzoglou, P. Avgeriou, S. Sencer, E.-M. Arvanitou, and I. Stamelos, "A theoretical model for capturing the impact of design patterns on quality," in *Proc. 32nd ACM SIGAPP Symp. Applied Computing (SAC '17)*, Marrakech, Morocco, 2017, pp. 1231–1238. DOI: 10.1145/3019612.3019781.

[10] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Softw. Qual. J.*, vol. 21, no. 2, pp. 289–323, Jun. 2013. DOI: 10.1007/s11219-012-9175-x.

[11] C. Izurieta and J. M. Bieman, "Testing consequences of grime buildup in object oriented design patterns," in *Proc. 1st Int. Conf. Software Testing, Verification, and Validation (ICST '08)*, Lillehammer, Norway, 2008, pp. 171–179. DOI: 10.1109/ICST.2008.27.

[12] M. R. Dale and C. Izurieta, "Impacts of design pattern decay on system quality," in *Proc. 8th ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement (ESEM '14)*, Torino, Italy, 2014, pp. 37:1–37:4. DOI: 10.1145/2652524.2652560.

[13] I. Griffith and C. Izurieta, "Design pattern decay: the case for class grime," in *Proc. 8th ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement (ESEM '14)*, Torino, Italy, 2014, pp. 39:1–39:4. DOI: 10.1145/2652524.2652570.

[14] T. Schanz and C. Izurieta, "Object oriented design pattern decay," in *Proc. 4th ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement (ESEM '10)*, Bolzano-Bozen, Italy, 2010, pp. 7:1–7:8. DOI: 10.1145/1852786.1852796.

[15] D. Feitosa, P. Avgeriou, A. Ampatzoglou, and E. Y. Nakagawa, "The evolution of design pattern grime: An industrial case study," in *Proc. 18th Int. Conf. Product-Focused Software Process Improvement (PROFES '17)*, Innsbruck, Austria, 2017, pp. 165–181. DOI: 10.1007/978-3-319-69926-4_13.

[16] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Blackwell, 2012.

[17] T. Amanatidis, A. Chatzigeorgiou, A. Ampatzoglou, and I. Stamelos, "Who is producing more technical debt? A personalized assessment of TD principal," in Proc. 9th *Int. Workshop Managing Technical Debt (MTD '17)*, Cologne, Germany, 2017, pp. 4:1–4:8. DOI: 10.1145/3120459.3120464.

[18] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *Proc. 1st Int. Symp. Empirical Software Engineering and Measurement (ESEM '07)*, Madrid, Spain, 2007, pp. 449–451. DOI: 10.1109/ESEM.2007.55.

[19] A. Ampatzoglou, O. Michou, and I. Stamelos, "Building and mining a repository of design pattern instances: Practical and research benefits," *Entertain. Comput.*, vol. 4, no. 2, pp. 131–142, Apr. 2013. DOI: 10.1016/j.entcom.2012.10.002.

[20] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, 2002. DOI: 10.1109/32.979986.

[21] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric (GQM) Approach," in *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., 2002, pp. 528–532. DOI: 10.1002/0471028959.sof142

[22] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, 2006. DOI: 10.1109/TSE.2006.112

[23] G. Kniesel and A. Binun, "Standing on the shoulders of giants - A data fusion approach to design pattern detection," in *Proc. IEEE 17th Int. Conf. Program Comprehension (ICPC '09)*, Vancouver, BC, Canada, 2009, pp. 208–217. DOI: 10.1109/ICPC.2009.5090044.

[24] N. Pettersson, W. Löwe, and J. Nivre, "Evaluation of Accuracy in Design Pattern Occurrence Detection," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 575–590, Jul. 2010. DOI: 10.1109/TSE.2009.92.

[25] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of Java source code," *Softw. Pract. Exp.*, vol. 46, no. 9, pp. 1155–1179, Sep. 2016. DOI: 10.1002/spe.2346.

[26] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004. DOI: 10.1145/1052883.1052895.

[27] H. Khalid, M. Nagappan, and A. E. Hassan, "Examining the relationship between FindBugs warnings and app ratings," *IEEE Softw.*, vol. 33, no. 4, pp. 34–39, Jul. 2016. DOI: 10.1109/MS.2015.29.

[28] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. S. E. I. T. on Vouk, "On the value of static analysis for fault detection

in software," *Softw. Eng. IEEE Trans.*, vol. 32, no. 4, pp. 240–253, 2006. DOI: 10.1109/TSE.2006.38.

[29] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, 2008. DOI: 10.1109/MS.2008.130.

[30] D. Feitosa, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, "Investigating quality trade-offs in open source critical embedded systems," in *Proc. 11th Int. ACM SIGSOFT Conf. Quality of Software Architectures (QoSA '15)*, Montréal, QC, Canada, 2015, pp. 113–122. DOI: 10.1145/2737182.2737190.

[31] J. D. Evans, *Straightforward statistics for the behavioral sciences*. Pacific Grove: Brooks/Cole Pub. Co., 1996.

[32] A. Field, *Discovering Statistics Using SPSS*, 3rd ed. SAGE Publications Ltd, 2009.

[33] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc, "Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps," in *Proc. 22nd Int. Conf. Program Comprehension (ICPC '14)*, Hyderabad, India, 2014, pp. 232–243. DOI: 10.1145/2597008.2597144.

[34] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes," in *Proc. 19th Working Conf. Reverse Engineering (WCRE '12)*, Kingston, ON, Canada, 2012, pp. 437–446. DOI: 10.1109/WCRE.2012.53.

[35] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994. DOI: 10.1109/32.295895.

[36] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, 1993. DOI: 10.1016/0164-1212(93)90077-B.