Broken Windows: Exploring the Applicability of a Controversial Theory on Code Quality

Diomidis Spinellis*, Panos Louridas†, Maria Kechagia‡, Tushar Sharma§
Athens University of Economics and Business†, University College London‡, Dalhousie University§*
Athens, Greece*†, London, UK‡, Halifax, Canada§
dds@aueb.gr*, louridas@aueb.gr†, m.kechagia@ucl.ac.uk‡, tushar@dal.ca§

***Abstract*—Is the quality of existing code correlated with the quality of subsequent changes? According to the (controversial) broken windows theory, which inspired this study, disorder sets descriptive norms and signals behavior that further increases it. From a large code corpus, we examine whether code history does indeed affect the evolution of code quality. We examine C code quality metrics and Java code smells in specific files, and see whether subsequent commits by developers continue on that path. We check whether developers tailor the quality of their commits based on the quality of the file they commit to. Our results show that history matters, that developers behave differently depending on some aspects of the code quality they encounter, and that programming style inconsistency is not necessarily related to structural qualities. These findings have implications for both software practice and research. Software practitioners can emphasize current quality practices as these influence the code that will be developed in the future. Researchers in the field may replicate and extend the study to improve our understanding of the theory and its practical implications on artifacts, processes, and people.**

***Index Terms*—code quality, software evolution, broken windows, mining software repositories, software analytics, empirical study, software smells**

I. Introduction

In the late 1960s Stanford professor Philip Zimbardo and his research team ran a fascinating field study demonstrating the ecological effects of community and anonymity on vandalism [1]. They removed the license plates from two used cars and abandoned them on the street with the hood slightly raised: one in leafy Palo Alto, California and one in New York City's gritty Bronx. Within two days they recorded 23 instances where people tore apart or wrecked the Bronx car. In contrast, in Palo Alto in a five day period the only person who touched the car was a passerby who on a rainy day caringly closed the hood to protect the motor. In his description of the experiment Zimbardo argues that in environments where anonymity and a lack of community sense are the rule, individuals resort to vandalism and graffiti to gain personal recognition.

In 1982 George Kelling and James Wilson used a news report of Zimbardo's demonstration [2], together with an evaluation of New Jersey's police foot-patrol program and their personal observations of Newark foot-patrol officers, to discuss policies for maintaining safe communities. In a long, influential, and somewhat controversial article, titled *Broken Windows* [3], they argued that the maintenance of public order can lead to safer communities.

The views of Kelling and Wilson, termed as the *broken windows* theory, have been used to explain the variation of crime among neighbourhoods [4], support theories linking disorder with crime [5, pp. 281–281], and set public policy, most famously in the 1990s by William Bratton as Rudy Giuliani's New York City police commissioner [6, pp. 47–50]. There is no agreement on the results of the corresponding policies [4], [7], [6], [5], mainly because it is difficult to perform controlled studies on the subject. A large carefully-controlled study of Chicago neighbourhoods found that social cohesion among neighbors and their willingness to intervene for the common good is associated with reduced violence [8]. More recently, six clever field experiments demonstrated that when people are exposed to the violation of observed (descriptive) social norms and rules, they are significantly more likely to break prescribed (injunctive) norms and rules [9]. On the other hand, a study published in the same decade [5] independently recreated and examined Kelling and Wilson's data and attributed the original attributed New York's crime reduction to mean reversion. The same study also examined a randomized social experiment that moved families to less disorderly neighbourhoods. The study failed to find a corresponding reduction in those people's criminal behavior. Furthermore, a more recent meta-analysis of 96 studies [10] failed to find consistent evidence that disorder increases aggression or deteriorated attitudes toward the neighborhood, while a meta-analysis of 198 studies by the same authors [11] identified methodological weaknesses that have inflated evidence for the broken windows theory and identified an association from disorder to lower mental health, but not to physical health or risky behavior.

Despite these mixed findings in social contexts, the concept of broken windows theory has intrigued researchers in various fields, including software development. The *objective* of this study is to investigate the broken windows theory in the context of software development, *i.e.,* examine whether developers become more or less diligent regarding their coding, depending on the internal quality of the code they operate on. Internal quality comprises the aspects of software quality that are experienced only by its developers rather than its users. It includes the code's formatting, structure, and identifier naming. On the other hand, internal code quality does not cover the software's functionality, reliability, or performance; the things that are often the topics of defect or bug reports. In an analogy to the broken windows theory, we consider code of high internal quality as "order" and changes that reduce it as "crime". There is also an analogy with the seriousness of the crime: code style infractions [12] can be considered as petty crime, whereas structural problems are more serious.

Though known and often anecdotally referenced, the broken windows theory has not been explored adequately in our field. A motivation for the study is to justify devoting effort to maintaining specific attributes of internal code quality because of their indirect, signalling, effects. The findings can have implications regarding the software development process in general and also specific aspects such as tooling, refactoring, code reviews, and continuous integration.

This work, based on the statistical analysis of metrics derived from two million code commits in 118 constantly evolving projects for long time, comprising 5.5 million lines of code (LOC) contributes two types of findings. First, history's weight on the evolution of internal code quality: it seems that a body's existing code quality is related to the quality of its subsequent evolution. Second, the relationship between the commits' code quality in areas covered by injunctive norms and some of the corresponding code's descriptive norms: adherence to coding guidelines is related to the look of the existing code. Both findings provide (qualified) support for the application of the broken windows theory to software development.

We **contribute** the following to the state of the art. First, we present a method to systematically explore the applicability of the broken window theory on code quality. Next, we outline a theoretical model concerning code quality and the broken window theory, contributing towards the understanding of their potential connection. Finally, we make publicly available a replication package comprising time-series code quality data (metrics and smells) from 118 open-source projects as well as corresponding analysis scripts.[1]

II. Theoretical Model

The impact of norms on human behavior can be productively studied by distinguishing two norm types. *Injunctive* norms encompass what others approve or disapprove, in a formal way (through rules) or informally (through social pressure). *Descriptive* norms illustrate what others actually do [13], and are established when a subject observes the environment. Both norms provide information, what is the expected and the common behavior, and in a particular situation they can be in agreement or in conflict. In the context of programming, an injunctive norm would be the disapproval of using the *goto* statement [14], while a (conflicting) injunctive norm would be its common use to jump to a function's error exit routine [15], [16, pp. 43–44]. In the physical world, it has been found that injunctive norms are observed more when they are in agreement with descriptive norms, associated with the corresponding or even another type of behavior [17], [9]. Persons take into account these norms in order to accurately model reality and their reactions, to have meaningful social relationships, and to maintain their self-concept [17].

For the purposes of our study we have devised a model that describes our understanding of the factors affecting the evolution of a module's internal quality from a time point T_N to a later time point T_{N+1}. First, there are factors external to our study that affect the quality at both time points, without however establishing a causal correlation via the code. These may include the developer ability, the development process (which can set diverse injunctive norms through rules, guidelines, processes, and tools), the application domain, the global distribution of developers, and many others; see Section VI. All these can affect the code quality at both time points, and thereby result in a correlation. There are also factors that may establish a direct causal correlation between the code at the two time points, *i.e.,* the code quality at T_N directly affecting the code quality at T_{N+1}. The most important factor is clearly the legacy of inherited code: at time point T_{N+1} the majority of the code, and therefore its quality, will consist of code from T_N with some changes.

There are also other more interesting ways in which the code quality at T_N affects the code quality at T_{N+1}. First, come the interfaces that the code has to use, either to interface with the rest of the system or to use third party components. If their design is substandard, they can be detrimental to the code quality [18], [19], [20], because they can impose naming convention violations or, worse, an ineffective module structure. Then, comes the module's design structure. If this imposes bad traits, such as lack of appropriate layering or encapsulation, subsequent code additions that build upon that design will naturally add to the problem. Also consider identifier naming. Badly named identifiers (overly short, long, inaccurate, or violating coding conventions) existing at T_N have to be used at T_{N+1}, thus perpetuating the problem. Finally, any of the preceding aspects of the internal code quality may act as a descriptive norm, signalling to its developers a lack of care regarding its quality, contributing to the commission of further sins in the future. This last point is the essence of the broken windows theory applied to software development.

Although our examples showed how bad code can lead to worse, note that all the factors associated with the code can also improve the code quality: a sound design, suitably-named identifiers, high-quality third party components, and a code state that signals love and care can make subsequent additions maintain or improve the code's quality. Also note that, although the module structure, the naming conventions, and the used interfaces at T_{N+1} are bound by those used at point T_N, this binding can be broken through refactoring [21]. However, in practice, refactoring tools, which can aid these tasks, are underused [22], and it is doubtful whether refactoring actually improves code quality metrics [23], [24], [25].

In common with the real world, the signalling effect associated with the broken windows theory in software development is about communicating expectations regarding community standards. These can be associated with *injunctive and descriptive norms* (this is how we write code around here),

[1]https://doi.org/10.5281/zenodo.10060518

incentives (rewarding or reprimanding developers based on the quality of their code), and *processes* that flag or correct quality problems (code reviews, commit hooks, and continuous integration checks).

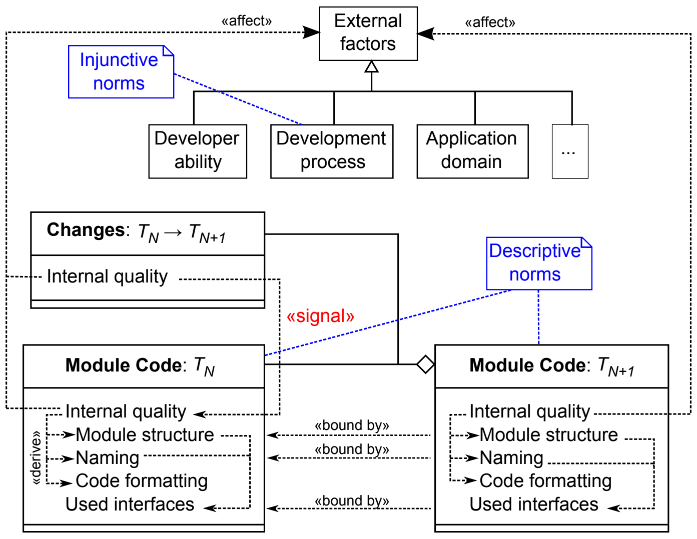

Fig. 1. Factors affecting the evolution of internal code quality.

The code quality evolution model we use in our study is illustrated as a UML diagram in Figure 1. Note that UML, confusingly for some, has dependency arrows point from the dependent (client) entity to the independent (supplier) one. So in the figure the internal quality of the changes $T_N \rightarrow T_{N+1}$ may depend on the signalling effect of the module's internal quality at T_N; the internal quality is derived from the module structure, naming, and formatting; the external factors affect the internal quality; and the structure naming and interfaces at T_{N+1} are bound by the choices at T_N.

III. Methods

A. Overview

The goal of this study is to examine the applicability of the broken windows theory to source code quality. Specifically, we investigate whether the adherence to code quality practices within a source code project impacts its internal quality. In other words, we explore whether the historical internal quality of the code influences developers' adherence to code quality practices. To achieve this goal, we formulate the following two research questions.

RQ1. *Does future code quality relate to past code quality?*

In this research question, we investigate whether the code quality at time T_N correlates to the code quality at time T_{N+1}.

RQ2. *How does existing code quality relate to developers' behavior and practices concerning it?*

With this research question, we check whether a developer behaves better or worse on source code that starts with better or worse quality characteristics. Exploring these research questions will help us establish the applicability (or, non-applicability) of the broken window theory on code quality.

To investigate the research questions, we conduct an empirical analysis. First, we identify a set of repositories written mainly in C and Java. We then collect the required code quality metrics and code smells indicating the code quality of the repositories. We apply statistical analysis to the collected code quality data to investigate the addressed research questions. The rest of the section describes each of the steps in detail.

B. Data Collection and Processing

We studied the possible correlation between the quality of an existing code body and additions or changes made to it as follows. After selecting two popular languages exhibiting different aspects of code quality characteristics (C and Java), we employed stratified sampling to obtain a random representative sample of open source code repositories to study. We selected various metrics and smells covering size, structure, code style, documentation, and adherence to design principles. These metrics and smells are commonly used in code quality analysis studies and fairly represent the state of software code quality. We chose two tools that can reliably produce quality measures for diverse projects written in these languages, namely *cmcalc* [26] for C and DesigniteJava [27] for Java. Based on the capabilities of these tools, we then collected data regarding the evolution of C code quality metrics and Java design, implementation, and testability smells for more than two million file revisions. Finally, we analyzed the obtained data using statistical autocorrelation techniques. We have made publicly available on the Zenodo repository a 260 MB replication package with the code used for extracting and analyzing the data as well as the obtained results.[1]

1) Project Sampling: Following proposed guidelines for the systematic mining of software repositories [28], [29] we established the following *inclusion criteria* for selecting repositories.

1) More than 10 stars or forks, to select projects relevant to the software engineering community, and avoid personal projects and student exercises. These two metrics are considered to be the most useful GitHub project popularity metrics [30].
2) Code in the C or the Java programming language. These two languages cover the imperative and object-oriented programming paradigms, and are among the most popular languages according to the TIOBE index.[2]
3) At least ten years of history with at least one commit on every half-year interval, in order to examine long-evolving software, where code quality may act as a repository of tacit norms.

For *repository selection* we employed *random sampling*, aiming for a sample size N in the range 50–100, which is what we could process with the computational and storage resources at our disposal. To aid the generalizability of our findings, we selected a random stratified sample of the N projects based on community interest and third-party involvement. Specifically,

[2]https://www.tiobe.com/tiobe-index/

TABLE I
EXAMINED REPOSITORY METRICS

	Total	**Min**	**Median**	**Avg**	**Max**	s
Created		2008-07	2011-12	2011-10	2013-08	16
Commits	1 979 092	227	3 902	21 748	872 689	92 520
Committers		13	144	550	24 960	2 614
Stars		25	5 495	12 066	159 725	20 627
Forks		20	1 219	3 362	50 841	6 848
Files	304 743	53	961	3 349	65 697	7 768
Lines	89 426 359	5 755	208 555	982 707	27 533 925	3 012 417
C files	56 389	3	183	842	27 626	3 373
C lines	35 306 678	6 475	92 137	526 965	18 904 362	2 302 884
C file revisions	2 195 510	38	3 884	32 769	1 148 732	140 756
C analysis time (s)	145 644	1	38	2 174	132 335	16 148
Java files	28 393	46	722	1 183	4 069	1 310
Java lines	3 361 394	3 747	74 753	140 058	459 952	148 277
Java file revisions	310 333	506	4 208	12 931	51 120	15 128
Java analysis time (s)	862 747	834	7 169	35 948	280 502	63 836

we defined our sampling method to reflect the fact that it is more likely for developers to interact with popular projects.

In common with most studies, we sampled projects from GitHub, which contains millions of open source repositories, including mirrors of popular projects hosted elsewhere. We used GitHub stars as a proxy for community interest [30] and GitHub forks as a proxy for personal involvement. We defined five strata on an exponential progression of star or fork engagement ranges: 11–100, 101–1 000, 1 001–10 000, 10 001–100 000, 100 001–1 000 000. (Through experiments we determined that this stratification yields same order of magnitude number of projects in the first four strata. We also found that there are no projects with more than 1 000 000 stars or forks that satisfied our other criteria.) For each stratum we obtained with a GitHub API query the number of projects in it that were written in one of the languages we studied, and had at least ten years of history, including a change in a six-month period in the period's middle. In each stratum i for the number of obtained projects P_i in it, we estimated the total engagements (forks or stars) T_i in the range 10^i to 10^{i+1} as

$$T_i = P_i \frac{10^i + 10^{i+1}}{2}$$

Based on it, we calculated a random selection probability to obtain one of the N projects if it had a single engagement as

$$S = \frac{N}{\sum T_i}$$

Finally, we obtained the number of projects N_i to select in each stratum i as $N_i = ST_i$. For example, in the Java projects stratum with 10 001–100 000 stars there are 51 projects, sharing an estimated total of 2.8 million stars, which results in the requirement to select 30 projects from it. We then selected projects at random from each stratum i, checked whether they satisfied the outlined criteria, and kept those that did, until we reached the required number of projects N_i. Doing this for both stars and forks and taking into account overlaps yielded a number of projects in the desired range 50–100: 93 for Java and 83 for C.

We obtained all data using GitHub API queries, utilizing features such as range selection, sorting, and the conjunction of multiple selection criteria in a single query to obtain the required results in the most efficient manner. This required 36 queries for obtaining the strata metrics (6 strata $\times$ 2 engagement metrics $\times$ 3 languages — we also provide C# projects in the dataset to facilitate future work) and more than 6 000 for deriving the projects sample (18 queried commit intervals for 304 accepted projects in all three languages, plus average of 9 intervals for 139 rejected projects). As a last step we we examined the list of randomly selected repositories for potential issues and removed two repositories that were clones of the (also selected) Linux kernel (*Xilinx/linux-xlnx* and *raspberrypi/linux*) and one that contained mostly patches rather than code (freebsd/pkg). Indicative descriptive statistics of the examined project repositories are listed in Table I. The file and line metrics refer to the contents of each repository at the head of its default branch.

2) C Quality Metrics Collection: Calculating quality metrics on large C code bodies is tricky for technical and operational reasons [31], [32]. On the technical side, dependencies of code associated with its compilation environment, as well as code portability issues, make it difficult to establish the context required to parse and semantically analyze the code. This is especially true for programs written in C, where the compilation depends on compile-time flags and macro definitions passed through the build process, system header files, the search paths for these files, and macros internally defined by the compiler [33], [34], [35]. Then comes the required throughput: with build times for large projects taking minutes, analyzing thousands of revisions of hundreds of projects with a full-fledged compile can take an impractically long time.

We addressed both problems choosing to use, in common with other studies [36], [37], [38], [39], *cmcalc*, an open source tool that efficiently calculates C code quality metrics, without requiring full access to the compilation environment's parameters [26]. The *cmcalc* tool operates as a filter, receiving

TABLE II
C CODE SIZE AND QUALITY METRICS PER FILE

Metric and initials		**Mean**		**25%**		**50%**		**75%**	
		Begin	**End**	**Begin**	**End**	**Begin**	**End**	**Begin**	**End**
Number of statements		119.8	168.4	11	14	49	65	133	181
Number of characters		13079	18568	2488	2783	5748	7245	13085	17749
Number of comment characters		2551	3382	435	409	1101	1159	2469	2823
Number of comments		30.76	39.72	2	3	8	11	25	33
Comment density %	CD	67.46	57.85	6.383	7.692	14.68	15.44	29.89	30.49
Comment size	CS	191.2	154.4	60.47	53.88	107.4	88.8	208	157
Number of functions	FN	10.48	14.27	1	2	5	6	12	16
Function size	FS	12.59	13.05	6	6.333	9.75	10.06	14.71	15.09
Goto density %	GD	1.689	1.863	0	0	0	0	1.695	2.128
Mean unique identifier length	IL	10.2	10.71	8.374	8.915	10.04	10.6	11.75	12.31
Mean line length	LL	27.24	26.73	22.67	22.98	25.39	25.57	29.04	29.3
Number of lines	LN	445.5	619	90	101	212	266	474	635
Questionable word density %	QD	0.1162	0.08946	0	0	0	0	0	0
Style inconsistency %	SI	2.075	1.767	0.09494	0.1293	0.9804	0.8386	2.708	2.254
Mean statement nesting	SN	0.5738	0.5916	0.2727	0.3023	0.4945	0.5188	0.7679	0.781

on its standard input C source code, and printing on its standard output a line of metrics associated with that code. As such it can be efficiently tied to the output of a `git show` command, so that successive versions of a file can be analyzed without the performance degradation of code touching the secondary storage. The tool's operation is based on a state machine logic lexical analyzer for a superset of C code. The analyser combines the functionality of the C preprocessor and C language-proper lexical analysis with rudimentary parsing, so as to recognize C preprocessor directives, functions, statement nesting, indentation, other spacing, comments, identifiers, keywords, and operators. The provided metrics do not require semantic analysis of the code, allowing *cmcalc* to sidestep its cost; thus *cmcalc* dodges the complexity of C's pointer aliasing. By treating the C preprocessor's function-like (those defining entities that can be called like a function) and object-like (e.g. those defining constants) macros as C-proper functions and objects, *cmcalc* will produce reasonably accurate results without requiring access to header files and the compilation environment. As an example, *cmcalc* will not stop processing with an error due to missing include files, declarations, or definitions.

The *cmcalc* tool calculates size, structural, quality, and code style metrics; see references [40], [41, pp. 326–333] for more details. A representative selection of metrics, along with the quartile points calculated on the first version and last version of each file in our data set, are listed in Table II. In our study we provided *cmcalc* with successive versions of each C file, and obtained from its output the corresponding metrics: one set of metrics for each version of each C file. Most metrics are self explanatory; here are details for the rest. The comment density (CD) is the ratio between the number of comments and statements in a file. The comment size (CS) is the ratio between the number of comment characters and the number of comments. The function size (FS) is the ratio between the number of statements and the number of functions. The goto density (GD) is the ratio between the number of *goto* statements and the number of statements. The questionable word density (QD) is the number of words that may indicate problems in the code as well as swearwords divided by the file's number of lines. The following whole words were searched for in a case-insensitive manner: bugbug, buggy, bullshit, crap, crash, damn, damned, doom, doomed, fixme, fuck, fucker, fucking, hack, hacked, hackery, hacks, hell, kludge, kludges, lame, lameness, poop, screwed, screws, shit, shits, suck, sucks, todo, xxx. Finally, the mean statement nesting (SN) is measured as the sum of the nesting of all lines within code blocks (e.g., 1 after a *while* statement and 2 after an *if* statement nested within the *while* one) divided by the number of those lines.

The *cmcalc* tool calculates code style infractions from commonly agreed formatting guidelines. As there are a number of different approaches for formatting C code, *cmcalc* allows us to measure the *consistency* of their application, rather than adherence to a specific formatting style. Specifically, for each way to format a particular construct (for example putting a space after the `while` keyword) *cmcalc* measures the times a the rule is applied in the one way (e.g., putting a space) and the times b the rule is applied in the other way (omitting the space).

Each metric represents the number of occurrences of the corresponding phenomenon in a file.

- A space, a (or a lack of it, b) before or after the following symbols: binary operator, closing brace, comma, keyword, opening brace, opening square bracket, semicolon, struct operator. ($4 \times 8 = 32$ metrics.)
- A space, a (or a lack of it, b): before a closing bracket, after a unary operator, before a closing square bracket. ($3+3 = 6$ metrics.) Note that the rules regarding spacing on the opposite side of the preceding three symbols are context-specific, and therefore they were not checked.
- A space, a at end of a line. (One metric; no style convention puts a space at the end of a line, therefore $b = 0$ in this case.)

The file's style inconsistency for n style rules (20 in our case) as a percentage of possible cases is defined as follows.

$$\mathrm{SI} = \frac{\sum_{i=1}^{n} \min(a_i, b_i)}{\sum_{i=1}^{n} a_i + b_i} \times 100 \tag{1}$$

Thus, through *cmcalc* and the preceding definition we identify the prevalent coding style used in each file (e.g. putting spaces around a binary operator), and obtain a metric of inconsistency regarding the coding style found within the file. We obtained the rules from the Google,[3] FreeBSD,[4] GNU,[5] and the updated Indian Hill[6] style guidelines.

From the metrics we gathered five are associated with code style quality: commenting (CD, CS), naming (IL), and layout (SI, LL). Another six are proxies for code structure quality: modularity at the file level (FN, SN, LN) and the function level (FS), code complexity (SN, GD), and questionable coding practices (QD).

Metrics calculation was performed through a series of nested loops, expressed as *bash* [42] shell scripts that run *cmcalc* for each revision, of each file, of each repository. The revisions were obtained through the `git log` command, run with a custom output format to obtain the revision's hash code, committer email, and machine-readable time stamp. Then `git show` was invoked on the filename and hash code associated with each revision to pipe to *cmcalc* the source code to be analyzed. Thus a single 96-field line was produced for each file's revision (410 million values in total), which other programs could use to analyze the results.

The metrics calculation of all revisions of all files (35 million lines) took more than 40 hours to run. Caching and checkpointing were used to allow the efficient execution of incremental runs while the processing code was debugged. A considerable speedup was achieved by parallelising the analysis of each file using GNU parallel [43]. This gave the calculation a throughput of 25 thousand lines per second on an 8-core computer.

3) Java Code Smells Collection: We extended the code quality analysis by detecting and analyzing commonly used code smells [44], [45] on repositories mainly written in Java. For a comprehensive coverage of code quality, we required a tool that supports code smells detection at different granularities such as implementation, design, and testability. We chose DesigniteJava [27], which detects a variety of code smells and computes common code quality metrics. The tool has been validated by its authors [46], [47] and has been used in diverse studies [48], [46], [49], [50], [51].

We included in our analysis three types of smells listed in Table III: design smells (DS) [52], implementation smells (IS) [44], and testability smells (TS) [47]. The latter, taking into account that testability is the degree to which the development of test cases can be facilitated by the software design choices, are the practices that impact the testability of a software system. We selected the listed commonly occurring smells for our analysis because, given their scope and characteristics they may get influenced by other existing smells.

TABLE III
Java Code Size and Smell Instances

Type	Name	Begin	End
	Lines of code	5 872 257	6 321 769
	Number of classes	143 850	147 279
DS	Broken hierarchy	10 935	11 183
DS	Broken modularization	3 721	4 111
DS	Deep hierarchy	13	14
DS	Deficient encapsulation	12 719	13 706
DS	Feature envy	5 504	5 994
DS	Hublike modularization	97	153
DS	Insufficient modularization	6 840	8 092
DS	Missing hierarchy	245	311
DS	Multifaceted abstraction	181	214
DS	Multipath hierarchy	185	201
DS	Wide hierarchy	314	371
IS	Complex conditional	6 788	8 450
IS	Complex method	11 869	14 188
IS	Empty catch clause	14 841	16 345
IS	Long method	1 261	1 575
IS	Long parameter list	14 506	17 419
IS	Long statement	118 374	138 959
IS	Magic number	301 830	334 784
IS	Missing default	3 644	4 110
TS	Excessive dependency	7 479	8 773
TS	Global state	12 310	13 753
TS	Hard-wired dependency	204	234
TS	Law of Demeter violation	437	628

In terms of its architectural design, DesigniteJava is organized into three layers. Eclipse Java Development Toolkit (JDT) forms the bottom layer. The tool utilizes JDT to parse the source code, prepare ASTs, and resolve symbols *i.e.,* associate type information with variable declarations. The middle layer, *source model*, maintains a source code model created from the extracted information from an AST with the help of JDT. The business logic *i.e.,* the smell detection and code quality metrics computation logic resides in the top layer. The layer accesses the source model, identifies smells and computes metrics, and outputs the inferred information in either .CSV or .XML files. As its user, we utilized the tool for each repository selected for analysis in *multi-commit* analysis mode.[7] In this mode the tool takes the path of a Git repository as input, switches the repository to a commit, analyzes the code, and produces a set of .CSV files containing smells and metrics data for the commit. This produces CSV files with smell details for each file at each commit. We subsequently filtered these to create timeline series specific to changed files. The total number of smells detected at the beginning and at the end of all time series is listed in Table III. (The total number of lines in Table III is larger than the one shown for Java files in Table I, because a file can appear in many time series as it moves around the repository.) Our processing excludes

[3]http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
[4]http://www.freebsd.org/cgi/man.cgi?query=style&sektion=9
[5]https://www.gnu.org/prep/standards/html_node/Formatting.html
[6]http://www.cs.arizona.edu/\%7Emccann/cstyle.html

[7]DesigniteJava documentation—https://www.designite-tools.com/docs/commands.html

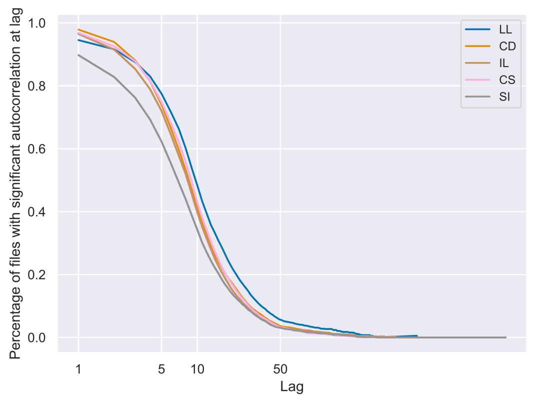

Fig. 2. Percentage of files with autocorrelation > 0.5 at each lag for code style metrics.

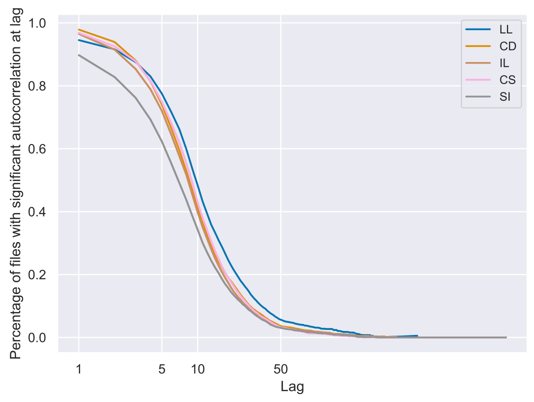

Fig. 3. Percentage of files with autocorrelation > 0.5 at each lag for code structure metrics.

a few Java projects with faulty repositories (*apache/camel*) or with processing requirements that exceeded our computing resources (e.g. checkstyle/checkstyle, which run out of heap space despite getting allocated 40 GB of RAM).

IV. Results

We present our observations corresponding to both research questions exploring the broken windows theory in software using our corpus.

A. ***RQ1***: *Relationship of quality of existing code on code quality evolution*

To set the scene we inquired into the way code quality evolves over time; in particular, the way existing code quality relates to future quality. After all, if existing quality does not relate to future quality in a significant way, it does not make much sense to check the effect of existing code in developers' coding practices.

We started our analysis by looking at the autocorrelation of C code style metrics and C code structure metrics. We calculated the autocorrelation for files that have more than 50 commits and a non-constant value of the metric (because otherwise the autocorrelation is undefined). From 55,523 files, that left us with 10,530 files with autocorrelations calculated for at least one of the code style metrics and 10,404 files for code structure metrics. We took into account up to 50 lags and judged as important autocorrelations greater than 0.5 that were found to be statistically significant, having p-value < 0.05 for the Ljung-Box test. The results can be seen in Figure 2 and Figure 3.

For both classes of metrics, we see that a very high percentage of files have significant autocorrelations for small lags. Moving forward in time, we see that for all metrics about 40% of the files have significant autocorrelations for lags up to 10. **Overall, history does relate to the evolution of code style and structure metrics, as the value of a metric at a particular commit exerts a considerable effect to more than 80% of files for all metrics in the commits that follow.** Moreover, in a significant portion of the files the relationship

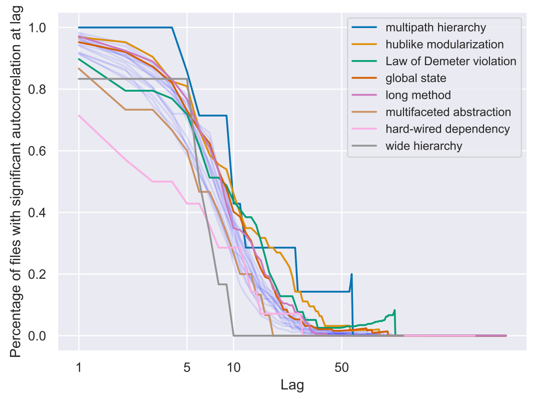

Fig. 4. Percentage of files with autocorrelation > 0.5 at each lag for smells.

goes deep, to many commits and not just the one that comes next.

A general pattern that we can discern from Figures 2 and 3 is that metrics that correspond to advice given frequently in software engineering matter a lot in history. Comments, both in what regards their density and their size, make a mark in the history of many files. Rules on identifier length and line length are also found in many style guides, and they are sticky in that previous values to a large extent relate to future values. The same goes for style inconsistency, which aggregates different kinds of inconsistencies. Advice on function size and number of functions in a file is frequently drilled to software developers; so is advice against too much nesting and the use of *gotos*. It seems that it is not just history, but what we might call tradition, in the form of old, time-honoured programming tenets, that manifests itself along the evolution of software.

We can discern a similar pattern in Figure 4, which shows the autocorrelation of Java smell metrics over different time lags. Out of 114,519 files, we found 832 with more than 50 commits and a non-constant metric value. The autocorrelations show again that **about 40% of the files we examined exhibit significant autocorrelations for lags up to ten**. We have highlighted the top five and bottom three of the smells, in terms of the average percentage. Three of the top five, *i.e.,* multipath hierarchy, hub-like modularization, and Law of Demeter violation, seem to follow distinctive paths, as do the bottom three metrics, *i.e.,* multi-faceted abstraction, hard-wired dependency, and wide hierarchy. This may be because for all the highlighted metrics, the number of files where the autocorrelation could be calculated with statistical significance was small: the median number of files, for the different lags, for which we could calculate autocorrelation > 0.5 with p-value < 0.05 was less than 10.

Our analysis indicates that history does relate to the evolution of code quality, for code style and structure metrics and for code smells.

B. ***RQ2:*** *Developers' behavior and historical code quality*

It is easy to spot a real broken window, but as there is no *a priori* indication of what is a good or a bad file, we used quantiles for ascribing categories to files. For each project, we identified the first commit and the corresponding metric for each of the code style and the code structure metrics. We computed the 25% and the 75% quantiles for those metrics and we grouped the files at the top quantile as top files and the files at the bottom quantile as bottom files. Note that top and bottom are not equivalent to good and bad, as in some metrics higher values are better whereas in other metrics the opposite is true. In total we formed 11 groups containing top and bottom files, one for each selected metric.

To see whether developers behave differently in top files than they do in bottom files, we investigated whether their commits in top files are quantitatively different, in terms of our metrics, than their commits in bottom files. In effect, we looked whether developers tailor the quality of their commits based on the quality of the file they commit to. We measured the quality of a particular commit as the difference of the selected metric for the commit from the value of the metric in the previous commit.

To work out the numbers, we grouped each project's data by developer and for each group we selected the commits made in top files and the commits made in bottom files. To see if the developers perform different kinds of commits depending on the file's quality (top or bottom), we calculated the Kolmogorov-Smirnov two samples test for those developers that had at least 10 commits in top files and 10 commits in bottom files. We have 11 metrics, so in total we have 121 cases. As these are multiple tests, we used a Family-Wise Error Rate of 0.05 to adjust the p-values using the Benjamini-Hochberg prodecure [53].

The results for C code are shown as a heat map in Figure 5. The y-axis of the heat map represents the different groupings; the x-axis represents the metric we are using for the statistical test. For example, the bottom left square corresponds to files being grouped to top and bottom quartiles depending on mean line length (y-axis), and the percentage of developers that display different commit behaviour in what regards mean line

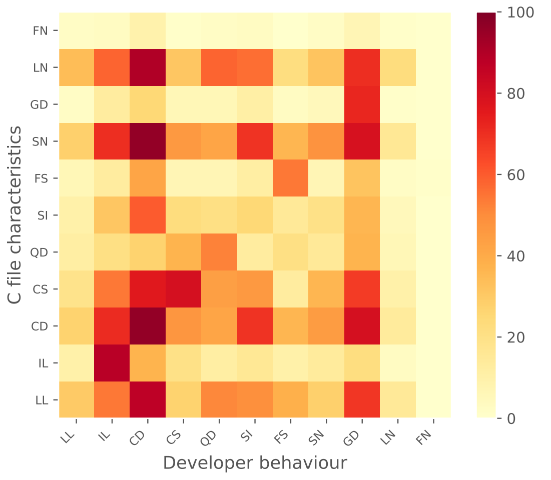

Fig. 5. Percentage of developers with different behaviour in top vs bottom C files.

length (x-axis). The square at the intersection of SI on the x-axis and CD on the y-axis corresponds to the percentage of developers that display different commit behaviour in what regards style inconsistency measured in files grouped according to the classification of the file as top or bottom based on the code density. In other words, what is the percentage of developers whose commits exhibit a different style inconsistency on files coming out top in comment density against files coming out at the bottom in comment density?

Looking the at columns of Figure 5, CD, GD, and IL stand out. **Developers appear to behave very differently in what regards comment density, goto density, and to long or short identifier lengths, along files grouped in the top or bottom with respect to a variety of different metrics.** The laggards are LL, LN, and FN; it seems that developers do not behave with respect to line lengths and high or low line and function counts on files with different top or bottom metric counts.

Going row-wise, one way to interpret each row is as the context the file provides regarding the value of a metric in the file. **So, LN, SN, CS, CD, and LL seem to provide particularly strong contexts and, taking the whole picture, stronger than others.** That is reasonable. Developers do different kinds of commits depending on the file they are into, but not every single characteristic of the file influences every single commit metric to the same extent. The row for GD illustrates that the heatmap is not symmetric: even though developers behave differently in terms of goto density on files grouped to the top or bottom on many different metrics, files that come top or bottom in goto density seem to be have high percentage of different goto density behavior on the part of the developers.

We repeated the same analysis for the Java smells; we omit the resulting heatmap, as it is bright yellow nearly all over. That means that we were not able to detect a statistical significance in the Java smells metrics of developers' commits in files rated top and bottom with regards to those smells. That does not entail that, in contrast to C code and structure, developers do not show a difference in their behavior regarding design smells, implementation smells, and testability smells; it may be that developers adhere to norms regarding these smells independently of the context. We plan to investigate this further in the future.

Our results indicate that some historical code characteristics (such as comment density and identifier length) strongly relate to the developers' behavior. However, code smells do not show a similar statistically significant relationship.

V. Threats to Validity

In terms of **external validity**, it is clear that we have limited ourselves to projects using the C and Java programming languages. Although these languages still enjoy rude health, programmers have a wealth of other languages at their disposal, and it does not automatically follow that our findings transfer to them. In our defense, the syntax and overall style of C and Java have influenced many other newer programming languages, so there is no *a priori* reason to indicate our results would not carry over to other languages, at least those with a similar structure. However, it may be difficult to replicate our findings in environments, such as Visual Basic, that by default perform many formatting tasks automatically. At the same time, programs written in dynamic languages may rely more on a consistent code style to make up for the lack of a compiler that can catch trivial errors.

The metrics we have used are **internal quality** ones we could efficiently measure through the two tools we employed. More sophisticated measures, such as the actual defect density, may yield different results. Looking at external quality metrics, like those measuring reliability, accuracy, and performance, may also differentiate the resulting picture.

In projects code style may not be a developer's choice, but may be imposed through tools and processes. We manually searched the developer documentation of 25 popular projects (Blender, CPython, ffmpeg, FreeBSD, gcc, gdb, gecko-dev, Git, illumos, ImageMagick, KDE, Linux, MySQL server, Nagios, OpenCV, Perl, PHP, PostgreSQL, VLC, WINE, as well as the Apache, freedesktop.org, GNOME, GNU Git, and Sourceware projects) looking for style guidelines and how these are enforced. We found that 7 out of 25 projects (28%) are using automated methods (mandatory and voluntary) to ensure code style conformance. Linux, gcc, CPython, and illumos-gate instruct committers to check their source code for style inconsistencies before each commit, whereas other projects suggest the use of automatic style checks, such as adapting the configuration of source code editors and IDEs, and executing third-party scripts. We also found that 17 out of 25 projects (68%) prescribe specific mandatory or voluntary coding guidelines. Projects with mandatory checks, such as FreeBSD, Blender, Perl, and PHP, have extensive guidelines, and encourage committers to conform to them in order to have their commits accepted by reviewers. Projects with voluntary checks, such as PostgreSQL, VLC, and KDE, provide shorter guidelines, and do not impose strict code style checks.

Our choice of projects may also be a limiting factor. Our random stratified sampling resulted in the inclusion of some big, successful projects, where quality standards may be above and beyond those entertained by other projects. It is possible that our findings may apply less to such other projects. On the other hand, it is likelier that quality in such projects will show increased variation, and that developers will enjoy greater freedom in their coding. Both factors would amplify the quality's signalling effects.

Note also that we only examine the changes that show up in the main development branch. Development also takes place in other branches as well [54]; changes on other branches may be merged with the current branch, so these will show in the data we examine, but their own history will be lost, as we do not examine the history of the separate branches. This may hide from our analysis quality problems that were identified and fixed through a code review.

Moving on to **internal validity**, we have been careful not to propose any causal relationships in the analysis of our results, which is a key criticism leveled against the broken windows theory [11]. We report results of statistical relations, or point out the absence of relations, but we do not attribute cause and effect. That would require a more detailed examination of the model we proposed in Section II, and, possibly, putting forward a concrete mathematical formulation encompassing dependent and independent variables of software quality in relation to the existing quality context. This can be the subject of further research.

VI. Related Work

Broken windows theory in software engineering: To the best of our knowledge there is limited work that uses the broken windows theory to explain factors that affect software quality. In particular, Deissenboeck and Pizka have referred to the broken windows theory when examining the inconsistent naming of identifiers in software projects [55]. The same authors in another study [56] have also called for studying psychological effects, such as those associated with the broken windows theory. Brunet *et al.* refer to the theory when suggesting that the gap between code and architecture is tractable provided that violations are checked and solved in a short time period [57]. The preceding studies referred to the broken windows theory in order to explain or justify software development phenomena. However, none of the studies attempted to use empirical evidence in order to validate or disprove the broken windows theory in the context of software development. A related theory concerns contagious technical debt, which has so far been studied mainly qualitatively [58], [59]. In the similar vein, the human inclination to imitate behavior of others is known as the Bandwagon effect. Such cognitive biases have been explored in the software engineering context [60], [61], [62], [63].

Factors affecting software quality: On the other hand, there is a considerable body of work on the factors that affect software quality. These studies can be categorized into the areas of management, the software development process, the developers' characteristics, and product properties. A survey on factors that affect software quality [64] examined organizational, technical, and individual factors. In the following paragraphs we will briefly describe some representative findings from each area.

In the field of *management* a number of studies found that clear source code ownership results in fewer failures [65] and defects [66], that unfocused teams working on central modules can increase post-release failures [67], that well-coordinated teams can reduce software failures [68], that organizationally (rather than geographically) distributed teams lead to an increase of software failures [69], and that low staff morale and excessive turnover can decrease software quality [70].

Regarding the *software development process*, major factors that have been reported to affect software quality include the software's architecture and design [70], [71], the quality of the requirements [72], [70], tool use (in some cases) [72], [70], test-driven development [73], [74], code reviews [75], [76], scheduling [70], (maybe) refactoring [24], [23], and the existence of software processes in general [72], [70].

On the *developer* front, studies have found that the capabilities and experience of the personnel can ensure the software's conformance quality [72], while their absence can lead to software decay [70].

Finally, the properties of the developed product can also affect its quality. Factors that have been identified include the product's size [72], [77], [70], the age of the source code [70], [68], code porting and reuse [70], [78], the chosen programming language [79], and the application domain [80].

VII. Discussion and Implications

In experimented boundary, factors, and context, we have shown the following in the preceding sections.

- The quality of an existing, initial, code body and its subsequent evolution are related. Code quality carries its history with it, and its current state is strongly dependent on that history. The exact nature of the dependence varies on how we define history. It appears that traditional software engineering advice has a particularly strong effect on history.
- The developers' behavior associated with a variety of code style and structure metrics is significantly related to a file's commenting (comment size and density—CS, CD), size (number of lines and statements—LN, SN), and, rather unexpectedly, length of lines (LL), as seen by studying Figure 5 row-wise.
- The developers' behavior associated with identifier length (IL), comment density (CD), and style inconsistency (SI) is related with a number of file characteristics, as seen by studying Figure 5 column-wise.

In effect, we have seen evidence that some descriptive norms that apply to software (how the software is actually written) are associated with variations in developer behavior in areas that are often covered by injunctive norms (guidelines and best practices). In the context of the broken windows theory, we did not find as high a significant relationship between public order (style consistency) and more serious crimes (increased statement indentation, fewer comments, shorter identifiers, or more *goto* statements). Consequently, our results are nearer to Zimbardo's original demonstration [2] — a community's effect on descriptive norms – than to Kelling and Wilson's interpretation [3] — the escalation of violations from descriptive to injunctive norms.

Surprisingly, we saw that a file's style consistency which is a ubiquitous indicator of order in a file and thereby forms a strong descriptive norm, is associated with the developers' behavior at a lesser extent; certainly not as strongly as we would have expected when we started this study.

The apparent lack of a strong behavioral link between style inconsistency and other measures of software quality surprised us at first; but then on closer inspection less so. The fact that it does not appear to be correlated with other measures suggests that it is an independent quality variable, and not one that can be readily calculated from structural quality metrics. Programming style is a distinct quality attribute, and our style inconsistency density metric may be one way to measure it.

In the context of the broken windows theory in software development, style inconsistency is special for a number of reasons. First, style infractions can be easily determined by inspection, and committed with the sure knowledge that they will not affect the software's external quality. Therefore, stylistic infractions are both a more noticeable signal and a more sensitive effect. This situation resembles the actual broken windows in the real world.

Furthermore, code style is also a matter of personal taste and opinion. Some developers hold these opinions with such a conviction that it may give rise to so-called holy wars [81, p. 35]. Therefore, it may be easy for developers lacking self-discipline to commit stylistic infractions, especially when editing a file where their "religion" appears to be tolerated.

Assuming that the effect of code on code is indeed casual, it has important implications for software developers and their managers. The basic message that should be carried away is that keeping basic code hygiene can not only help the software's maintainability, but also improve the quality of subsequent code additions. Developer behavior regarding the code quality measures we examined can be improved by a few, rather unglamorous, actions: keeping modules (files in the case of C code) short and focused, writing plenty of descriptive comments, and avoiding long code lines.

Furthermore, given that a file's number of functions (FN) and their size (FS) seem to be strongly related with developers' behavior regarding the file's size, it seems important to invest effort in designing the appropriate decomposition of the code into modules (files and functions in the case of C code), for once this structure is set in code it is apparently difficult to escape from it.

VIII. Further Work

The study of social influence in the context of software development can be expanded in two fronts: those of the empirical data and the mechanisms at work.

The empirical basis can be extended by studying more and smaller projects where external quality-setting factors are less prevalent. This can be done by using data sets that contain metadata from many such repositories, such as the *RepoReapers Data Set* [82], or *GitHub Search* [29], in conjunction with actual commit data from the corresponding repositories. The work can also be easily extended to cover more programming languages, especially given the fact that the analysis performed by *cmcalc* is mostly programming language agnostic. In particular, it would be interesting to study object-oriented metrics [83], resource leaks, code duplication, and security vulnerabilities. It would also be particularly interesting to check the effect of style infractions in languages where these often lead to errors in programming logic; for example, one could examine the role of the optional semicolons in JavaScript.

Studying the mechanisms through which social influence theory applies to software development as well as the effects of these mechanisms on software quality and process is more challenging. Some topics worthy of further examination are the following.

- Are the signals communicated and acted upon subconsciously, or are developers making conscious rational choices on the quality of their work based on a file's perceived quality?
- Are developers using the signals to optimize where they will direct their efforts?
- How does this optimization affect external code quality?
- How should the software development process be adjusted to take into account these factors?
- What are the reasons and the import in the differences of the results between code and structure metrics on the one hand, and design, implementation, and testability smells on the other?

These are clearly questions whose answers can change the way we view software development.

Acknowledgements

This research has been co-financed by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thalis — Athens University of Economics and Business — Software Engineering Research Platform. In addition, NSERC Discovery grant (#R35616) has also partially supported the study. Dedicated to the memory of the criminologist Calliope D. Spinellis (1933–2024).

References

[1] P. Zimbardo, "Anonymity of place stimulates destructive vandalism," Online http://www.lucifereffect.com/about_content_anon.htm. Current September 2014. Archived through WebCite at http://www.webcitation.org/6SL1IvBEp, 2006.

[2] "Diary of a vandalized car," *Time*, vol. 93, no. 9, p. 68, Feb. 1969.

[3] G. L. Kelling and J. Q. Wilson, "Broken windows: The police and neighborhood safety," *Atlantic Monthly*, vol. 249, no. 3, pp. 29–38, Mar. 1982.

[4] W. G. Skogan, *Disorder and Decline: Crime and the Spiral of Decay in American Neighborhoods*. University of California Press, 1992.

[5] B. E. Harcourt and J. Ludwig, "Broken windows: New evidence from New York City and a five-city social experiment," *The University of Chicago Law Review*, pp. 271–320, 2006.

[6] B. Harcourt, *Illusion of Order : The False Promise of Broken Windows Policing*. Cambridge, Mass: Harvard University Press, 2004.

[7] G. Kelling and C. M. Coles, *Fixing Broken Windows: Restoring Order and Reducing Crime in our Communities*. New York: Simon & Schuster, 1997.

[8] R. J. Sampson, S. W. Raudenbush, and F. Earls, "Neighborhoods and violent crime: A multilevel study of collective efficacy," *Science*, vol. 277, no. 5328, pp. 918–924, 1997.

[9] K. Keizer, S. Lindenberg, and L. Steg, "The spreading of disorder," *Science*, vol. 322, no. 5908, pp. 1681–1685, Dec. 2008.

[10] D. T. O'Brien, C. Farrell, and B. C. Welsh, "Looking through broken windows: The impact of neighborhood disorder on aggression and fear of crime is an artifact of research design," *Annual Review of Criminology*, vol. 2, no. 1, p. 53–71, Jan. 2019.

[11] ——, "Broken (windows) theory: A meta-analysis of the evidence for the pathways from neighborhood disorder to resident health outcomes and behaviors," *Social Science & Medicine*, vol. 228, p. 272–292, May 2019.

[12] D. Spinellis, "elyts edoc," *IEEE Software*, vol. 28, no. 2, pp. 104–103, Mar. 2011. [Online]. Available: https://www.dmst.aueb.gr/dds/pubs/jrnl/2005-IEEESW-TotT/html/v28n2.html

[13] R. B. Cialdini, R. R. Reno, and C. A. Kallgren, "A focus theory of normative conduct: Recycling the concept of norms to reduce littering in public places." *Journal of Personality and Social Psychology*, vol. 58, no. 6, p. 1015, 1990.

[14] E. W. Dijkstra, "Go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968.

[15] W. Gellerich, M. Kosiol, and E. Ploedereder, "Where does GOTO go to?" in *Reliable Software Technologies — Ada-Europe '96*, ser. Lecture Notes in Computer Science 1088. Springer, 1996, pp. 385–395.

[16] D. Spinellis, *Code Reading: The Open Source Perspective*. Boston, MA: Addison-Wesley, 2003. [Online]. Available: https://www.spinellis.gr/codereading

[17] R. B. Cialdini and N. J. Goldstein, "Social influence: Compliance and conformity," *Annual Review of Psychology*, vol. 55, pp. 591–621, 2004.

[18] D. Spinellis, "A critique of the Windows application programming interface," *Computer Standards & Interfaces*, vol. 20, no. 1, pp. 1–8, Nov. 1998. [Online]. Available: https://www.dmst.aueb.gr/dds/pubs/jrnl/1997-CSI-WinApi/html/win.html

[19] D. Bermbach and E. Wittern, "Benchmarking web API quality," in *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 188–206.

[20] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, "API designers in the field: Design practices and challenges for creating usable APIs," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Oct. 2018.

[21] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 2000.

[22] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[23] K. Stroggylos and D. Spinellis, "Refactoring: Does it improve software quality?" in *5th International Workshop on Software Quality*, B. Boehm, S. Chulani, J. Verner, and B. Wong, Eds. ACM Press, May 2007, pp. 1–6. [Online]. Available: https://www.dmst.aueb.gr/dds/pubs/conf/2007-WoSQ-Refactor/html/SS07.html

[24] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, vol. 51, no. 9, pp. 1319–1326, 2009.

[25] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas, and I. Stamelos, "Evaluating the quality of open source software," *Electronic Notes in Theoretical Computer Science*, vol. 233, pp. 5–28, 2009.

[26] D. Spinellis, P. Louridas, and M. Kechagia, "The evolution of C programming practices: A study of the Unix operating system 1973–2015," in *ICSE '16: Proceedings of the 38th International Conference on Software Engineering*, W. Visser and L. Williams, Eds. New York: Association for Computing Machinery, May 2016, pp. 748–759. [Online]. Available: https://www.dmst.aueb.gr/dds/pubs/conf/2016-ICSE-ProgEvol/html/SLK16.html

[27] T. Sharma, "DesigniteJava," Dec. 2018, https://github.com/tushartushar/DesigniteJava. [Online]. Available: https://doi.org/10.5281/zenodo.2566861

[28] M. Vidoni, "A systematic process for mining software repositories: Results from a systematic literature review," *Information and Software Technology*, vol. 144, p. 106791, 2022.

[29] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in GitHub for MSR studies," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2021.

[30] H. Borges and M. T. Valente, "What's in a GitHub star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, Dec. 2018.

[31] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *MSR '09: 6th IEEE International Working Conference on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 11–20.

[32] G. Gousios and D. Spinellis, "Conducting quantitative software engineering studies with Alitheia Core," *Empirical Software Engineering*, vol. 19, no. 4, pp. 885–925, Aug. 2014. [Online]. Available: https://rdcu.be/b7FyD

[33] D. Spinellis, "Global analysis and transformations in preprocessed languages," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1019–1030, Nov. 2003. [Online]. Available: https://www.dmst.aueb.gr/dds/pubs/jrnl/2003-TSE-Refactor/html/Spi03r.html

[34] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *AOSD '11: 10th International Conference on Aspect-Oriented Software Development*, 2011, pp. 191–202.

[35] P. Gazzillo and R. Grimm, "SuperC: Parsing all of C by taming the preprocessor," in *PLDI '12: Programming Language Design and Implementation*, 2012, pp. 323–334.

[36] D. Spinellis, P. Louridas, and M. Kechagia, "An exploratory study on the evolution of C programming in the Unix operating system," in *ESEM '15: 9th International Symposium on Empirical Software Engineering and Measurement*, Q. Wang and G. Ruhe, Eds. IEEE, Oct. 2015, pp. 54–57. [Online]. Available: https://www.dmst.aueb.gr/dds/pubs/conf/2015-ESEM-CodeStyle/html/SLK15.html

[37] T. Johnston, "Toolkit for automatic collection and predictive modelling of software metrics," Master's thesis, McMaster University, Department of Computing and Software, Hamilton, ON, Canada, 2016. [Online]. Available: http://hdl.handle.net/11375/19705

[38] J. Walden, "The impact of a major security event on an open source project," in *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, Jun. 2020.

[39] ——, "OpenSSL 3.0.0: An exploratory case study," in *Proceedings of the 19th International Conference on Mining Software Repositories*. ACM, May 2022.

[40] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Boston, MA: Addison-Wesley, 2002.

[41] D. Spinellis, *Code Quality: The Open Source Perspective*. Boston, MA: Addison-Wesley, 2006. [Online]. Available: https://www.spinellis.gr/codequality

[42] C. Newham, *Learning the bash Shell: Unix Shell Programming*. O'Reilly Media, Inc., 2009.

[43] O. Tange, "GNU parallel: The command-line power tool," *;login:*, vol. 36, no. 1, pp. 42–47, Feb. 2011.

[44] M. Fowler, *Refactoring: Improving the Design of Existing Programs*, 1st ed. Addison-Wesley Professional, 1999.

[45] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[46] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smells," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4020–4068, 2020.

[47] T. Sharma, S. Georgiou, M. Kechagia, T. A. Galeb, and F. Sarro, "Investigating developers' perception on software testability and its effects," *Empirical Software Engineering*, vol. 28, no. 120, 2023.

[48] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira, "On the density and diversity of degradation symptoms in refactored classes: A multi-case study," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 346–357.

[49] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, and A. Oliveira, "Removal of design problems through refactorings: Are we looking at the right symptoms?" in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 148–153.

[50] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, and C. Bezerra, "How does modern code review impact software design degradation? an in-depth empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 511–522.

[51] M. Alenezi and M. Zarour, "An empirical study of bad smells during software evolution using designite tool," *i-Manager's Journal on Software Engineering*, vol. 12, no. 4, p. 12, 2018.

[52] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. Morgan Kaufmann, 2014.

[53] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995. [Online]. Available: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1995.tb02031.x

[54] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining Git," in *MSR'09: 6th IEEE International Working Conference on Mining Software Repositories*, IEEE. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10.

[55] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Control*, vol. 14, no. 3, pp. 261–282, Sep. 2006.

[56] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard, "An activity-based quality model for maintainability," in *ICSM '07: IEEE International Conference on Software Maintenance*, Oct 2007, pp. 184–193.

[57] J. Brunet, R. Bittencourt, D. Serey, and J. Figueiredo, "On the evolutionary nature of architectural violations," in *WCRE '12: 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 257–266.

[58] A. Martini and J. Bosch, "On the interest of architectural technical debt: Uncovering the contagious debt phenomenon," *Journal of Software: Evolution and Process*, vol. 29, no. 10, Jul. 2017.

[59] F. Bi, B. Vogel-Heuser, Z. Huang, and F. Ocker, "Characteristics, causes, and consequences of technical debt in the automation domain," *Journal of Systems and Software*, vol. 204, p. 111725, Oct. 2023.

[60] K. Borowa, A. Zalewski, and S. Kijas, "The influence of cognitive biases on architectural technical debt," in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, 2021, pp. 115–125.

[61] P. Ralph, "Possible core theories for software engineering," in *2013 2nd SEMAT Workshop on a General Theory of Software Engineering (GTSE)*, 2013, pp. 35–38.

[62] ——, "Toward a theory of debiasing software development," in *Research in Systems Analysis and Design: Models and Methods*, S. Wrycza, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 92–105.

[63] W. Levén, H. Broman, T. Besker, and R. Torkar, "The broken windows theory applies to technical debt," *arXiv preprint arXiv:2209.01549*, 2022.

[64] N. Gorla and S.-C. Lin, "Determinants of software quality: A survey of information systems project managers," *Information and Software Technology*, vol. 52, no. 6, pp. 602–610, 2010.

[65] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 4–14.

[66] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 521–530.

[67] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 2–12.

[68] M. Cataldo and J. Herbsleb, "Coordination breakdowns and their impact on development productivity and software failures," *Software Engineering, IEEE Transactions on*, vol. 39, no. 3, pp. 343–360, March 2013.

[69] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality?: An empirical case study of Windows Vista," *Commun. ACM*, vol. 52, no. 8, pp. 85–93, Aug. 2009.

[70] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 1–12, Jan 2001.

[71] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," *Management Science*, vol. 44, no. 4, pp. 433–450, 1998.

[72] M. S. Krishnan, C. H. Kriebel, S. Kekre, and T. Mukhopadhyay, "An empirical analysis of productivity and quality in software products," *Manage. Sci.*, vol. 46, no. 6, pp. 745–759, Jun. 2000.

[73] N. Nagappan, E. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Software Engineering*, vol. 13, no. 3, pp. 289–302, 2008.

[74] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?" *IEEE Softw.*, vol. 25, no. 2, pp. 77–84, Mar. 2008.

[75] C. Kemerer and M. Paulk, "The impact of design and code reviews on software quality: An empirical study based on PSP data," *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 534–550, July 2009.

[76] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR '14. New York, NY, USA: ACM, 2014, pp. 192–201.

[77] M. Agrawal and K. Chari, "Software effort, quality, and cycle time: A study of CMM level 5 projects," *Software Engineering, IEEE Transactions on*, vol. 33, no. 3, pp. 145–156, March 2007.

[78] C. Kapser and M. Godfrey, ""cloning considered harmful" considered harmful," in *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*, Oct 2006, pp. 19–28.

[79] R. Subramanyam and M. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 297–310, April 2003.

[80] P. D. Edwards and R. S. Rivett, "Towards an automative 'safer subset' of C," in *16th International Conference on Computer Safety, Reliability and Security: SAFECOMP '97*, P. Daniel, Ed., European Workshop on Industrial Computer Systems: TC-7. Berlin: Springer Verlag, Sep. 1997, pp. 185–195.

[81] P. Goodliffe, *Code Craft: The Practice of Writing Excellent Code*. San Francisco: No Starch Press, 2007.

[82] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," in *FSE 2016: Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 2016.

[83] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.