Empirical Evidence of Code Decay: A Systematic Mapping Study

Ajay Bandi, Byron J. Williams, and Edward B. Allen
Department of Computer Science and Engineering
Mississippi State University
Mississippi State, Mississippi 39762, USA
ab1370@msstate.edu, williams@cse.msstate.edu, and edward.allen@computer.org

***Abstract*—Code decay is a gradual process that negatively impacts the quality of a software system. Developers need trusted measurement techniques to evaluate whether their systems have decayed. The research aims to find what is currently known about code decay detection techniques and metrics used to evaluate decay. We performed a systematic mapping study to determine which techniques and metrics have been empirically evaluated. A review protocol was developed and followed to identify 30 primary studies with empirical evidence of code decay. We categorized detection techniques into two broad groups: human-based and metric-based approaches. We describe the attributes of each approach and distinguish features of several subcategories of both high-level groups. A tabular overview of code decay metrics is also presented. We exclude studies that do not use time (i.e., do not use evaluation of multiple software versions) as a factor when evaluating code decay. This limitation serves to focus the review. We found that coupling metrics are the most widely used at identifying code decay. Researchers use various terms to define code decay, and we recommend additional research to operationalize the terms to provide more consistent analysis.**

***Index Terms*—Code Decay; Metrics; Coupling; Design Rules; Architecture Violations; Software Evolution**

I. Introduction

This systematic mapping study of code decay aims to give a classification and thematic analysis of the literature with respect to code decay detection procedures or methods by aggregating information from empirical evidence. In contrast, the purpose of a systematic review is often to "identify, analyze and interpret all available evidence related to a specific research question" [20], [46, p.45]. We chose a mapping study to find the empirical evidence of code decay because of our broad research questions.

Research in software evolution shows that violations in architecture and design rules cause code to decay [8], [11], [23]. These violations are due to new interactions between modules that were originally unintended in the planned design [23], [43]. The violations include adding new functionality, modifying existing functionality due to changing requirements and repairing defects, which are all inconsistent with the planned architecture and design principles. As a result, the system becomes more complex, hard to maintain, and defect prone [8], [30], [48]. Often, redesign or reengineering of the whole system is the solution for this problem [11]. The phenomenon of gradual increase in software complexity due to unintended interaction between modules that is hard to maintain has been termed code decay and architectural degeneration [8], [15]. In this paper, code decay refers to the violations of architecture, design rules and coding standards over time that make software more difficult to modify.

Eick et al. [8] defined code decay as code being harder to change than it should be after assessing code decay in a 15 year old real-time telephone switching software system using change management data. The system consisted of 50 major subsystems and about five thousand modules in C and C++. They used measures such as number of changes to a file, number of files touched to implement a change, sizes of modules, average age of constituent lines of modules, fault potential, and change effort. Their analysis confirmed that the system decayed due to successive modifications.

As another example, Godfrey and Lee [11] analyzed the open source project of the Mozilla web browser release M9 by extracting architectural models using reverse engineering tools. Mozilla web browser (M9) consisted of more than 2 million lines of source code in more than seven thousand header and implementation files in C and C++. After a thorough assessment of architecture models, Godfrey and Lee concluded that either Mozilla's architecture has decayed significantly in its relatively short lifetime or it was not carefully architected in the first place [11].

Software metrics characterize attributes of software. Product metrics measure attributes of development artifacts, such as source code and design diagrams. Lines of code and McCabe complexity are two of the best know metrics in this category. Process metrics measure attributes of the development process and events associated with the product, such as effort expended, defects discovered, and number of changes to code. Considerable research has modeled relationships between attributes that can be measured early and those measured later [12], [35]. For example, a statistical model might predict which modules are more likely to have bugs in the future, based on attributes measured early [35].

Code decay is an attribute that is evident only in retrospect. It is usually assumed that "decay" is a gradual process that goes unnoticed until a crisis occurs. One can detect decay by comparing measured attributes from the past with current values, and determine that quality has "decayed." A challenge for researchers is to find ways of detecting incipient "decay" well before a crisis develops.

Identifying and minimizing code decay is important to engineering practitioners focused on design and development issues thereby improving software quality. This systematic mapping study identifies detection techniques and metrics used to measure code decay. We followed Kitchenham and Charters [21] approach to perform this study.

The contributions of this review include: presentation of various terms used in the literature to describe decay, a categorization of code decay detection techniques, and description of metrics used to identify code decay. This paper is organized as follows: Section II describes the details of our research methodology. Section III reports the results for the research questions. Section IV discusses the implications and limitations of the study and section V presents conclusions.

II. Study Methodology

A systematic mapping study follows a similar process as systematic literature reviews, but the quality evaluation is not essential [20]. This study on code decay provide a comprehensive overview of the literature and topic categorization in a variety of dimensions (such as architecture violations and design rule violations). The steps in our study are: 1) Plan the study 2) Conduct the study and 3) Report the study.

A. Plan the Study

Planning a mapping study includes the following actions.

1) Identify the Need for the Study: The need for this mapping study is to identify and understand the scope of the empirical research on code decay and its forms. This study helps researchers to understand the research and define future research questions.

2) Specify the Research Questions: The major focus of this review is identifying techniques and metrics to assess code decay without including a general literature on fault prediction performance in software engineering [12], or the literature on fault prediction metrics [35]. Our research questions follow.

RQ1: *What are the techniques used to detect code decay (i.e., how is it discovered)?* To answer this question, we reviewed the literature and presented the categorization of code decay detection techniques.

RQ2: *Given code decay is detected, what metrics are used to quantify the extent of code decay (i.e., how is it measured)?* We extracted metrics used to evaluate code decay from our list of primary studies. A tabular overview of code decay metrics is presented that helps practitioners to assess code decay.

3) Develop the Study Protocol: Following the Kitchenham and Charters [21] guidelines, Dybå and Dingsøyr [6], [7] proposed a review protocol in the systematic review of empirical studies of agile software development. We followed a similar approach to develop our study protocol because our focus is on identifying empirical evidence of code decay. All authors participated in designing the review protocol. This protocol includes data sources and search strategy, inclusion and exclusion criteria, quality assessment criteria, a data extraction form, and data mapping. The details of inclusion and exclusion criteria, quality assessment criteria, and the data extraction form are given in a technical report [1]. Quality evaluation is not essential in mapping studies, but we applied quality criteria assessment when selecting our primary studies.

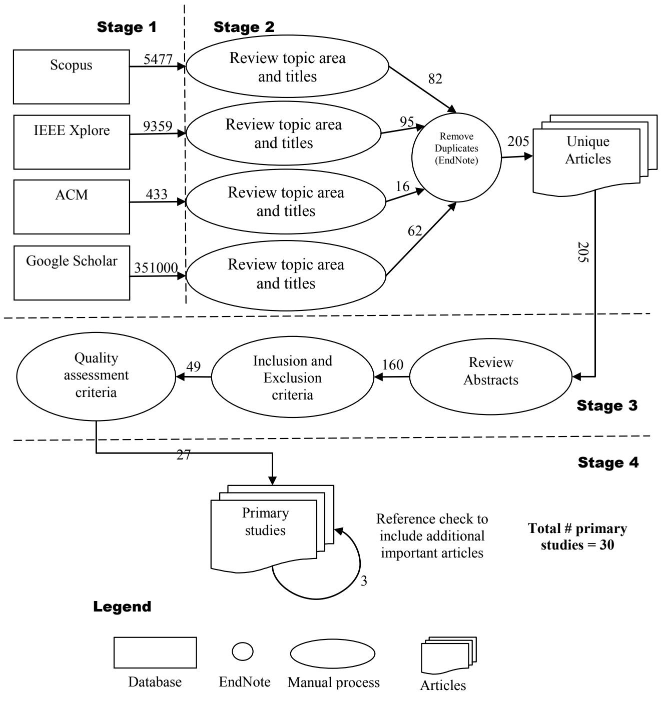
Stage 1
Stage 2
Scopus
IEEE Xplore
ACM
Google Scholar
5477
9359
433
351000
Review topic area and titles
Review topic area and titles
Review topic area and titles
Review topic area and titles
82
95
16
62
Remove Duplicates (EndNote)
205
Unique Articles
205
Quality assessment criteria
49
Inclusion and Exclusion criteria
160
Review Abstracts
Stage 3
27
Stage 4
Primary studies
3
Reference check to include additional important articles
Total # primary studies = 30
Legend
Database
EndNote
Manual process
Articles

Fig. 1. Overview of article selection

B. Conduct the Study

Conducting the study means performing the study protocol.

1) Data Sources and Search Strategy: The goal of the search is to identify relevant papers describing code decay (and related concepts) detection and measurement techniques. We searched peer-reviewed articles in these electronic databases: ACM Digital Library, Google Scholar, IEEE Xplore Digital Library, and Scopus (includes Science Direct, Elsevier, and Springer). In addition, we performed a bibliography check of every primary study to include any additional relevant articles that focused on our research questions. The high-level search string with keywords and their synonyms is shown below.

(((software OR code OR architecture OR system OR design) AND (erosion OR drift OR degeneration OR decay OR smell OR aging OR grime OR rot OR violation*) AND (detect* OR measure* OR metric* OR assess* OR evaluat*)))

The search strategy is a trade-off between finding relevant and irrelevant studies from the results of the search string. Figure 1 shows the review stages and number of studies selected at each stage. The different stages in our study are:

1) Search electronic databases using the search string.
2) Eliminate irrelevant studies reviewing the topic area and titles and remove duplicate articles using EndNote.
3) Filter articles based on the abstracts, inclusion and exclusion criteria, and quality assessment criteria.
4) Obtain primary studies and check their bibliographies of to include any additional appropriate studies.

TABLE I
STUDIES BY RESEARCH METHOD

Research method	Number	Percent
Case studies and archival studies	26	86
Controlled and quasi experiments	2	7
Experience reports and surveys	2	7
Total	30	100

To the best of our knowledge there are no earlier mapping studies or systematic reviews conducted on code decay. Therefore the timeframe for our search was not limited and articles up to May 2013 were considered. After searching peer-reviewed papers from the data sources and reviewing their titles, we exported only the relevant articles to EndNote to organize our bibliography. We excluded several false positives that are not related to software engineering subject. We manually excluded papers with a focus on biology, and mechanical, chemical and electrical engineering topics. EndNote has an advanced feature to remove duplicate articles based on the title, author names and conference titles. After eliminating duplicate articles, there were 205 unique articles remaining.

2) Inclusion and Exclusion Criteria: The basis for selection of primary studies is the inclusion and exclusion criteria. During this stage, we read all the abstracts to determine if the paper focused on identification or empirical evaluation of code decay. We included a paper if it is focused on either on identification or empirical evaluation of code decay. Of the 205 unique studies, 160 papers were selected after reviewing abstracts. Studies were included if they presented empirical evidence of code decay that includes architecture violations and design rule violations over time. Studies were included if they presented the detection of code decay at any level of abstraction ((i.e., class, package, subsystem, architecture and software system.) The term "decay" refers to the gradual decrease in quality. Therefore, only studies that evaluated decay on more than one version of a system developed over a period of time were included. We emphasize time as an important factor of decay. So papers that only took a snapshot of a single version of a system were excluded. Studies that concentrate on code decay detection techniques and analysis of metrics on proprietary or open source systems are included. Invited talks and expert opinions without empirical evidence were excluded. Studies that focused on just one version of the system, introductions, and tutorials were excluded. From the 140 papers after abstract review, we applied our inclusion and exclusion criteria and ended up with 49 papers.

3) Quality Assessment Criteria: Assessing quality criteria of studies is important for rigor in selecting primary studies. We defined quality assessment criteria similar to Dybå and Dingsøyr [6], [7] and applied these criteria to the 49 papers that resulted from inclusion and exclusion. We established quality criteria based on the types of studies that will be included in the review. There are quality assessment criteria to assess case studies and archival analysis, controlled and quasi-experiments, and peer-reviewed experience reports and surveys from industrial examinations. Based on the guidelines

TABLE II
DISTRIBUTION OF PRIMARY STUDIES

Publication channel	Number
International Symposium on Empirical Software Engineering and Measurement	4
Working Conference on Reverse Engineering	4
International Conference on Software Maintenance	4
European Conference on Software Maintenance and Reengineering	3
The Journal of Systems and Software	2
International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)	2
IEEE Transactions on Software Engineering	1
IEEE Software	1
Software–Practice and Experience	1
Communications in Computer and Information Science	1
International Conference on Software Engineering	1
IEEE International Software Metric Symposium	1
Asia Pacific Software Engineering Conference	1
IEEE Aerospace Conference	1
International Conference on Software Engineering and Knowledge Engineering	1
International Workshop on Software Aging and Rejuvenation	1
Brazilian Symposium on Software Components, Architectures and Reuse	1
Total	30

and examples [6], [7], [17], [38] we developed quality criteria checklists for different types of research studies [1]. The criteria for the experience report papers are not as rigorous as the other criteria for experiments and case studies. The primary focus for experience report is peer-review and research value. We applied the quality criteria uniformly for all the papers. Most of the papers excluded during this stage were idea-based and short papers. The accepted papers pass a specified number questions for the paper to be included in the primary studies. Applying the quality criteria resulted in 27 primary studies. The acceptance criteria is as follows: case studies, 6 of the 8 criteria is required; controlled/quasi experiments 9 of the 11 criteria is required; and experience reports 4 out of 4 criteria is required [1]. A bibliography check of the primary studies led to 3 additional primary studies.

4) Data Extraction: Once the list of primary studies is decided, the data from the primary studies is extracted. The details of the data extraction form is given in our technical report [1]. If the same study appeared in more than one publication, we included the most recent or the most comprehensive version (i.e. the journal article). After applying the quality assessment criteria, the first two authors studied papers in detail, independently extracted the data, and then independently reviewed a sample of each other's data extraction forms for consistency. We used the third author's opinion to resolve any inconsistencies. As a result there were no disagreements on extracted data. During this process, after extracting the data from a sample of papers, new keywords are included in the search string if necessary.

5) Data Mapping: The extracted data from the primary studies is mapped by identifying similarities in the detection techniques and the metrics used in the detection process. The level of potential automation was a natural distinction among the techniques. Section III answers our research questions and shows the classification of code decay forms, detection techniques and metrics used to identify code decay.

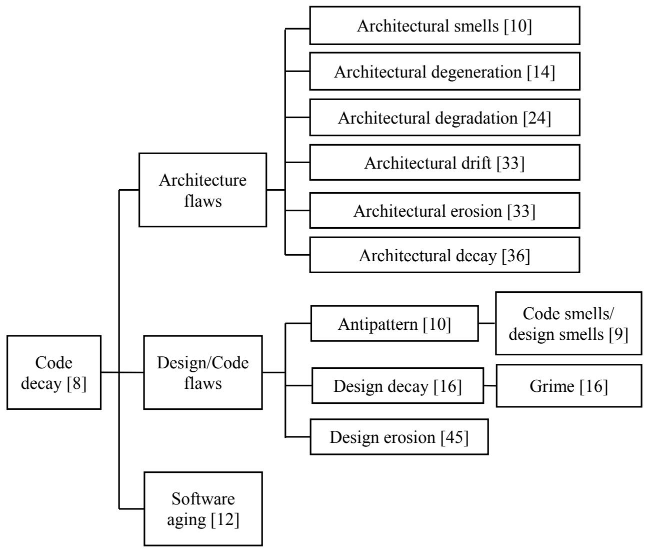

Fig. 2. Different forms of code decay

C. Report the Study

Like any other empirical study, this mapping study is reported to the researchers and practitioners in software engineering. We identified 30 primary studies based on our research questions and review protocol. These studies cover a range of research topics on architecture violations, design defects and problems with source code. The number of publications using a particular research method is listed in Table I. Of these 30 primary studies, 21 were performed on open source projects and 9 on proprietary systems. Table II presents the publication channels of our primary studies. Most of the studies (75%) were published in conferences, while others appeared in journals.

III. Results

The results of our review are presented as answers to each research question defined in section II.A.2. During our review of papers we encountered various terms in the literature that relate to code decay. These terms are organized on the basis of architecture, design, and source code. This terminology of code decay is shown in Figure 2. The definitions of these terms are given in our technical report [1]. Because code clones are considered a code smell and may well evolve from version to version, they may contribute code decay. However, our study did not include code clones, because it is a more mature research area.

A. Research Question 1: What Are the Techniques Used to Detect Code Decay (i.e. How Is it Discovered)?

Table III gives the summarized view of the different strategies to detect code decay. The motivation for the high level classification was the level of potential automation is a natural distinction among the techniques. Research techniques can be broadly categorized as human-based (manual) and metric-based (semi-automated) approaches.

1) Human-Based Approach: Human-based detection techniques consist of manual visual inspection of source code and architectural artifacts.

a) Inspection of Source Code: Source code inspections are performed subjectively and are guided by questionnaires. In the technique presented by Mäntylä et al. [26], developers manually inspected source code to identify code smells. They identified three different code smells (duplicate code, god class, and long parameter list) by filling out a web-based questionnaire. The assessment was based on subjective evaluation on a seven-point numeric Likert scale. These results do not correlate with code smells found using source code metrics. Similarly, Schumacher et al. [41] detected code smells (god class) manually by inspecting source code. In this study the subjects were encouraged to "think aloud" as they filled out the questionnaires. They compared the subjective results with the metric values from the automated classifiers. The results of their study increased the overall confidence in automatic detection of smells. They also found that god classes require more maintenance effort.

b) Inspection of Architectural Artifacts: Inspection of architecture artifacts is done by subjective evaluations using checklists and by comparing architecture models. Bouwers and van Deursen [2] proposed the lightweight sanity check for implemented architectures (LiSCIA) to identify architecture erosion. They provide a checklist of 28 questions based on units of modules, module functionality, module size, and module dependencies. Developers evaluate implemented architectures by inspecting the architecture artifacts. The evaluation phase consists of answering a list of questions concerning the architecture elements. LiSCIA provides corresponding actions to the questions to help identify the erosion in an implemented architecture. Rosik et al. [37] assessed architectural drift by comparing the implemented architecture with the original architecture of a system using a reflexion model (RM) [28]. Developers create and update the code base and associated mappings to the original and implemented architectures. This model displays the architecture in a pictorial representation with nodes and edges. The participants "think aloud" and assess the inconsistencies to identify violations from the results of the model. Their case study confirms that architectural drift occurred during the evolution of the system.

Manual detection of code decay and its categories is tedious work. Moreover, this process is time consuming, non-repeatable, and non-scalable. Manual detection of code smells do not correlate with the results of source code metrics derived from automated classifiers [26], [41].

2) Metric-Based Approach: The metric-based approach is divided into four subcategories. They are: historical data analysis, interpretation of rules, and model based techniques. These are discussed in the next subsections. We derived these subcategories by grouping each of the papers based on the unique approach presented. The four subcategories represent an orthogonal classification of the approaches used in the primary studies. The metric details used in these techniques are presented in section III.B.

TABLE III
CODE DECAY DETECTION TECHNIQUES

Detection Technique	Category	Subcategory	Reference
Human-based (manual) approach	Inspection of source code	Filling out questionnaires	[26], [41]
	Inspection of architectural artifacts	Answering checklist questions	[2], [37]
Metric-based (semi-automated) approach	Historical data analysis	Change management data	[8]
		Architecture history	[3], [13]
		Defect-fix history	[22], [29]
		Source code metrics	[36], [43], [44]
	Rule-based interpretations	Heuristics with threshold filter rules	[23], [27], [30], [31], [34], [42]
		Domain specific language rules	[4], [19]
	Model-based techniques	Probabilistic model	[45]
		Graphical model	[18], [39]
		Modularity model	[47]

a) Historical Data Analysis: The categories of historical data analysis used for discovery of code decay are change management data, architecture history, defect-fix history, and using source code metrics.

Change Management Data: Eick et al. [8] dealt with the history of change management data to detect code decay using code decay indices. Change management history includes source code of the feature, modification request, delta, and change to severity levels. Their statistical analysis on this data showed that an increase in the number of files touched per change and decrease in modularity over time yields strong evidence of code decay.

Architecture History: Brunet et al. [3] used the architectural diagrams from several versions of four different open source systems. They found violations in the architecture by applying the reflexion model technique [28]. They extracted JDepend, Lattix reverse engineering tools and design documentation to extract the high level architecture. They identified more than 3000 architecture violations.

Hassaine et al. [13] proposed a quantitative approach called ADvISE to detect architectural decay in software evolution. They used the architectural histories of three open source systems. The architecture history consists of architectural diagrams of different versions that are extracted from source code using a tool. The extracted architecture is represented as a set of triplets (triplet (S, R, T) where S and T are two classes and R is the relationship between two classes). They performed pair-wise matching of the subsequent architectures to identify deviations in the actual architecture from the original architecture by tracking the number of common triplets. This procedure was accomplished by matching architectural diagrams using a bit-vector algorithm. An increase in the number of classes and number of common triplets over time is a good indicator of architecture decay.

Defect-Fix History: Li and Long [22] used defect-fix history to measure architecture degeneration. Defect-fix history consists of information about the release, component in which the defect occurred, and the number of files changed to fix the defect. To analyze the defect history, they used multiple component defect metrics (e.g., percentage of defects that affected multiple components in a system and the average quantity of files changes to fix a defect). After analyzing the defect-fix history of a compiler system they found that an increase in the value of these metrics between two versions of the system indicates that the architecture has degenerated. Ohlsson et al. [29] performed historical data analysis on the defect-fix reports or source change notices of large embedded mass storage system to identify code decay. The defect-fix reports consist of description of release and the defect that has to be corrected. Average number of changes, size, effort and coupling are used to identify code decay. Average number of changes and coupling metrics play a major role in identifying code decay in this system. Their results showed that increases in values of these metrics indicate code decay.

Source Code Metrics: Researchers [43], [36], [44] compares the metrics of the source code over different versions of system using original version to identify code decay. Tvedt el al. [43] compares the interactions between the mediator and the colleagues in the mediator pattern between the two versions of Visual Query Interface (VQI) system (VQI1 and VQI2). They used coupling between modules (CBM) to identify unintended violations in the mediator design pattern and other misplaced violations. They concluded that the actual design of the system veered from the planned design. Riaz et al. [36] used coupling related metrics to compare two versions of a system. They found that an increase in the value of coupling related metrics indicates architecture decay. Van Gurp and Bosch [44] compared UML design diagrams of the ATM simulator from one version to another version by calculating metrics related to packages, functions and inner classes. Increases in the values of metrics, design decisions, and new requirements during the evolution of the system cause design erosion.

b) Rule-Based Interpretations: Rule-based interpretations are divided into two types. They are 1) Metric heuristics with threshold filters and 2) Domain specific language rules.

Metric Heuristics with Threshold Filters: Several researchers [27], [30], [31], [34] used metric-based heuristics to detect code/design smells. Marinescu [27] proposed a metric-based approach to detect code/design flaws in an object-oriented system. He detected code smells (god class and data class) using the values of metrics as heuristics. An example of one metric is given here: 1) The lower the Weight of Class (WOC) value, the more the class is expected to be a Data class and 2) The higher the WOC value, the more the class is expected to be god class. Similarly, he used metric heuristics on other metrics to detect both of these classes in an industrial case study. The threshold values are based on expert opinion.

Raţiu et al. [34] used heuristics and threshold values for each metric to detect god classes and data classes. These threshold values are based on the experience of the analyst. The results of this detection technique are suspected code smells. They analyzed different versions of software to obtain class and system history using the Evolution Matrix method. Their results highlight that this method improves accuracy in detecting god classes and data classes. Olbrich et al. [30], [31] used heuristics with threshold filter rules to detect god classes, brain classes, and shotgun surgery code smells. The threshold values used in the filtering rules are based on expert opinion.

An analysis of different versions of Lucene and Xerces found that there is a large correlation between the size of the system and the number of god classes, number of shotgun classes, and the number of brain classes. The preceding code smell techniques are indicators of code decay.

Lindvall et al. [23] compares the interactions between the modules in two versions of Experience Management System (EMS1 and EMS2) to avoid architectural degeneration. They measured architecture degeneration using coupling between modules (CBM) and coupling between module classes (CBMC). The values of CBM and CBMC are lower for the ESM2 version than ESM1 version, which indicates developers avoided architecture degeneration in the system.

Domain Specific Language Rules: Khomh et al. [19] used the DEtection and CORrection (DECOR) technique to detect code smells (god class). This technique generates automatic detection algorithms to detect code smells or antipatterns using rule cards. Rule cards are designed in a domain specific language with the combination of metrics and threshold values. The threshold values are defined based on in-depth domain analysis and empirical studies. The authors analyzed the relation between code smells and the changes in 9 releases of Azureus and 13 releases of Eclipse and concluded that code smells do have higher change-proneness. Ciupke [4] proposed automatic detection of design problems by specifying queries to the information gathered from the source code. The result of the query is the location of the problem in the system. These design queries can be implemented using logical propositions. The heuristics used to build these queries are based on the experience of the author. The author presented design violations in different versions of industrial and academic systems.

c) Model-Based Techniques: The three model-based techniques are: 1) Probabilistic model 2) Graph model and 3) Modularity model

Probabilistic Model: Vaucher et al. [45] used a Bayesian network approach to detect the presence of god classes. They built a Bayesian network model of the design detection rules. This model is based on metrics used to characterize specific classes and compute the probability that these specific classes are god classes. Metrics such as number of methods, number of attributes of a class and other cohesion values are used as inputs to the model. This probabilistic model predicts all the occurrences of god classes with a few false positives in different versions of Xerces and EclipseJDT.

Graph Model: Sarkar et al. [39] detected back-call, skip-call and dependency cycle violations in a layered architecture using a module dependency graph. The metrics used to detect these violations are: back-call violation index, skip-call violation index, and dependency violation index. In a dependency graph, back-call violations can be detected if the modules of one layer call the modules in another layer except the top layer. Skip-call violations can be detected by identifying the modules of one layer calls the modules existing in other layers but not the modules in adjacent layers. Dependency cycle violations are detected by the identifying strongly connected components in the module dependency graph. In a strongly connected graph, there exists a path from each vertex to every other vertex in a graph. The authors analyzed MySql 4.1.12 and DSpace and identified these violations using module dependency graphs. Johansson and Host [18] identified an increase in violations of design rules using graph measures of the architecture of a software product lines. Increase in the design rule violations from one version to other version of the software is a good indicator of code decay.

Modularity Model: Wong et al. [47] detected software modularity violations using their CLIO tool. This tool computes the differences between predicted co-change patterns and actual co-change patterns to reveal modularity violations (co-change patterns reflect classes that are often changed simultaneously). The co-change patterns are predicted by a logical model called Augmented Constraint Network (ACN) according to the Baldwin and Clarks design rule theory. They analyzed 10 releases of Eclipse JDT and 15 releases of Hadoop and identified four types of modularity violations that contribute code decay. They are: cyclic dependency, code clone, poor inheritance hierarchy, and unnamed coupling.

B. Research Question 2: Given Code Decay is Detected, What Metrics Are Used to Quantify the Extent of Code Decay (i.e., How Is it Measured)?

The results regarding metrics are summarized in Table IV. Eick et al. [8] defined code decay indices: history of frequent changes, span of changes, size, age, and fault potential to analyze historical change management data. An increase in values for history of frequent changes for a class and, span of changes for modification records are indicators of code decay. Ohlsson et al. [29] found empirical evidence of code decay using average number of changes in a module, and 'coupling' (how often a module involved in defects that required corrections extended to other modules). Increase in the value of coupling and average number of changes in a module is a good indicator of code decay. Lindvall et al. [23], [43] uses coupling between modules (CBM) and coupling between module classes (CBMC) to avoid architecture degeneration by identifying violations in the mediator pattern. The increase in value the of CBM and CBMC from one version of the system to another indicates degeneration in architecture. Li and Long [22] used various metrics related to defects spanning multiple components in a system. The greater the values of these metrics, the more significant is the architecture degeneration.

TABLE IV
METRICS

Category/Metrics	Relationship
Code decay	
History of frequent changes: Number of changes to a module over time [8].	Increase in number of changes to a module is an indicator of code decay.
Span of changes: Number of files touched by a change [8].	Increase in span of changes is an indicator of code decay.
Coupling: How often a module involved in defects that required corrections extended to other modules [29].	Increase in coupling between modules is an indicator of code decay.
Size: Number of non-commented source code lines from all the files in a module [8] (OR) Sum of added LOC, deleted LOC, added executable LOC and deleted executable LOC [29].	Growth in size of the system over time alone does not tell about the code decay. It represents the complexity of the system.
Fault potential: Number of faults that will have to be fixed in a module over time [8].	Number of faults need to fixed itself does not reveal evidence of code decay. It is the likelihood of changes to induce faults in the system.
Effort: Man hours required to implement a change [8], [29].	This depends on the total number of files touched to implement a change.
Architecture degeneration (modular level)	
Coupling-between-modules (CBM): Number of non-directional, distinct, inter-module references [23], [43]. *Coupling-between-module-classes(CBMC):* Number of non-directional, distinct, inter-module, class-to-class references [23].	Increase in the values of CBM and CBMC from one version to other version of the system indicates architectural degeneration.
Architecture degeneration (defect perspective)	
The average quantity of strong fix relationships that a component has in a system, The percentage of multiple -component defects (MCD) in a system, The average MCD density of components in a system, The average quantity of components that an MCD spans in a system, The average quantity of code changes (fixes) required to fix an MCD in a system MCD means defects spanning multiple components in a system. Fixing an MCD requires changes in the associated components. The relationship among these components is a fix relation. [22]	The greater the values of these metrics are, the more serious the architectural degeneration is.
Architecture decay	
Number of classes: Growth in size of the application [13]. *Number of triplets:* Triplet(S,R,T) S and T are two classes. R is the relation between S and T. [13].	Increase in the number of classes and number of common triplets from one version to another version by architectural diagram matching is a good indicator of architectural decay. Matching of architecture diagrams is automated using bit-vector algorithm.
Data Abstraction Coupling(DAC): Number of instantiations of other classes within a given class [36]. *Message Passing Coupling (MPC):* Number of method calls defined in methods of a class to methods in other classes [36]. *Coupling between objects (CBO):* Average number of classes used per class in a package [36].	Increase in the values of DAC, MPC and CBO from old version to latest version of the system becomes harder to maintain and indicates architecture decay.
Design pattern decay (Modular grime)	
Strength of coupling: Determined by removing the coupling relationship between classes (can be persistent or temporary) [40].	Persistent relationship between classes is more prone to decay compared to temporary association.
Scope of coupling: Demarcates the boundary of a coupling relationship (can be internal or external) [40].	Grime originating from external classes is more prone to decay than internal classes.
Design pattern decay (Modular grime)	
Direction of coupling: Number of inbound and out-bound relationships [40].	Increase in number of in-bound classes is more difficult to remove than out-bound classes.
Design erosion	
Number of packages, Number of inner classes, Number of functions, Non-commented source code statements, New (inner) classes, New functions, Removed (inner) classes. Metrics related to packages, functions, and inner classes. [44].	Increase of these metrics between different versions of system indicates design erosion. However, not all changes are reflected in the metrics. It also depends on how design decisions accumulate and become invalid because of new requirements.
Software aging	
LOC, CountCodeDel, countLineCodeExe, CountLineComment, CountDeclFileCode, CountDeclFileHeader, CountDeclClass, CountDeclFunction, CountLineInactive, CountStmtDecl, CountStmtExe, RatioCommentToCode Metrics related to program size (amount of lines of code, declarations, statements, and files) [5].	Program size metrics are positively correlated with software aging.
Architecture Violations	
Back-call violation index (BCVI), Skip-call violation index (SCVI), Dependency cycle violation index (DCVI): These are metrics used to detect back-call, skip-call and dependency cycle violations in layered style architecture. [39].	If BCVI/SCVI/DCVI is 1, then no violation. If BCVI/SCVI/DCVI is 0, then there is violation.
Code smells (god class)	
Access to Foreign Data (ATFD): The number of external classes from which a given class access attributes, directly or via accessor methods. Inner classes and super classes are not counted. [27], [34], [30], [31], [42] *Weighted Method Count (WMC):* WMC is the sum of statical complexity of all methods in a class. [27], [34], [30], [31], [42] *Tight Class Cohesion (TCC):* TCC is defined as the relative number of directly connected methods. [27], [34], [30], [31], [42] *Number of Attributes (NOA):* Number of attributes in a class. [34]	Increase in the number of god classes over time is an indicator of code decay. However, there are some harmless god classes also. (Ex: Class that has functionality of parser.)
Code smells (data class)	
Weight of Class (WOC): Number of attributes in a class. The number of non-accessor methods in a class divided by the total number of members of the interface[27], [34], [42] *Number of Public Attributes (NOPA):* The number of non-inherited attributes that belong to interface of a class. [27], [34], [42] *Number of Accessor Methods (NOAM):* The number of non-inherited accessor methods declared in the interface of a class [27], [34], [42]. *Weighted Method Count (WMC):* The sum of the statical complexity of all methods in a class [42]	Increase in the number of data classes over time is an indicator of code decay.
Code smells (brain class)	
WMC and TCC are same as described under god class detection. [31] *Number of brain methods (NOM):* Number of methods identified as brain methods in class. LOC in a method, cyclomatic complexity of a method, maximum nestinng level of control structures within the method and number of accessed variables in a method.[31]	Increase in the number of brain classes over time is an indicator of code decay.

TABLE IV
(CONTINUED)

TABLE IV
(CONTINUED)

Category/Metrics	Relationship
Code smells (shotgun surgery)	
Changing Methods (CM): The number of distinct methods that call a method of a class. [27], [34]	Increase in the number of shotgun surgery smells over time is an indicator of code decay.
Changing Class (CC): The number of classes in which the methods that call the measured method are defined. [27], [34]	
Code smells (Feature envy)	
Access to Foreign Data (ATFD): The number of external classes from which a given class access attributes, directly or via accessor methods. Inner classes and super classes are not counted. [42]	Increase in the number of feature envy type code smells over time is an indicator of code decay.
LAA: The number of attributes from the method's definition class, divided by total number of variables accessed.[42]	
Design smells (Extensive coupling and intensive coupling)	
CINT: The number of distinct operations called from the measured operation. [42]	Increase in coupling over time is an indicator of code decay.
CDISP: The number classes in which the operations called from the measured operations are defined in, divided by CINT.[42]	
Architecture degradation	
Graph measure: It is a function that denotes the deviation of the architecture structure compared to the wanted structure defined by design rules. [18]	Increase in the number of design rule violations makes architecture degraded and an indicator of code decay.

Hassaine et al. [13] used metrics such as the number of classes and the number of triplets to identify architecture decay by analyzing the architecture history using architectural diagram matching. Riaz et al. [36] used coupling related metrics such as Data Abstraction Coupling (DAC), Message Passing Coupling (MPC), and Coupling between objects (CBO) and by comparing these values between two versions of the system. The increase in the values of these metrics indicate architecture decay of the system. Grime is the phenomenon of accumulating unnecessary code in the design pattern. It is a form of design pattern decay. The three levels of grime are class grime, modular grime and organizational grime [16]. Schanz and Izurieta [40] use metrics of strength, scope, and direction of coupling to classify modular grime. Van Gurp and Bosch [44] assessed design erosion using metrics related to packages, functions, and inner classes. Increase in the values of these metrics between different versions of the systems indicate design erosion. Design erosion is not fully explained by the metrics. They found that design erosion is also based on the accumulation of design decisions that are not implemented due to new requirements. Sarkar et al. [39] used violation indices (BCVI, SCVI, and DCVI) to detect back-call, skip-call, and dependency cycle violations in a layered architectural style. If the value of BCVI/SCVI/DCVI indices is 1 then, there is no corresponding violation in the architecture. If BCVI/SCVI/DCVI value is zero, then there is corresponding violation in the architecture.

Our primary studies presented empirical evidence that an increase in the number of code smells from one version to another is an indicator of code decay. The code smells were identified using using well-defined metrics [30], [34], [31], [27]. These metrics are listed in Table IV. The threshold values of the metrics is based on expert opinion and empirical analysis. An increase in the number of code smells during the evolution of software is an indicator of code decay. Cotroneo et al. [5] used the program size metrics (such as amount of lines of code, declarations, statements and files) to predict the relation between software aging trends and software metrics.

IV. Discussion

In this mapping study, we identified 30 primary studies related to our research questions. This section discusses the implications of our results, weaknesses of our primary studies, research issues, and the limitations of our study.

A. Implications

The detection strategies we found in this review are categorized into human-based (manual) and metric-based (semi-automated) approaches. In manual processes, code decay is typically identified by answering questionnaires and using checklists. This approach is time consuming and non repeatable for larger systems. Moreover, it is expensive.

Metric-based approaches involve less human intervention in identifying code decay. Among the metric-based approaches, historical data analysis is useful only if the history of the system is available by comparing the architecture of one version to the subsequent architecture version. Source code metrics (Coupling between modules, coupling between module classes etc.) are compared to one another at the modular level. These metric values help to understand and avoid architecture degeneration. Modular metrics are helpful in identifying structural violations in design patterns and architectural styles. Applying heuristics with threshold filtering rules is a prominent technique to identify code/design smells. The disadvantage of this technique is threshold values are determined by expert opinion. Using expert opinion for threshold values does not apply to all the systems uniformly in identifying code decay. A model-based approach uses Bayesian models where the probability is computed using manually validated data. In metric-based approaches there is less human intervention and they are scalable to larger systems. From our observations, historical data analysis is a predominant technique to identify code decay when compared to other techniques.

Metrics that identify module and class coupling are predominantly used in the literature to detect code decay. Our review found that complexity metrics alone did not provide evidence of code decay. Coupling related metrics such as coupling between modules, coupling between module classes, data abstraction coupling, message passing coupling, coupling between objects, number of files coupled for a change, strength of coupling, scope of coupling, and direction of coupling do give evidence of code decay. It is important to measure coupling when assessing code decay.

Code decay degrades the quality attributes of the system. Some of the quality attributes include: maintainability (effort to change the code) [8], [30], [41], [43], [48], understandability [23], [30], [41], and extendability (effort to add new functionality) [43]. We also observed different factors, both developer-driven and process-driven lead to code decay. Developer-driven decay involves inexperienced/novice developers [41], developers focused on pure functionality [41], lack of system's architecture knowledge [37], developers apprehension due to system complexity [37], and impure hacking (carelessness of the developers). Process-driven decay includes difficulties related to missing functionality [37], violation of object-oriented concepts (data abstraction, encapsulation, modularity and hierarchy) [41], project deadline pressures [30], [41], changing and adding new requirements [8], [23], [43], updating new software and hardware components [8], and ad-hoc modifications without documentation [39].

Studies that concentrated on the relation between the design/code smells and architecture degradation [18], [24], [25] provide evidence of how design/code smells affect the architecture degradation. In aspect-oriented programming, modularity anomalies scattered among different classes is usually an architecturally-relevant smell. Such architecturally-relevant smells are difficult and expensive to fix in the later stages of software development [25]. Macia et al. [24] suggested that developers should promptly identify and address the code smells upfront, otherwise code anomalies increase the modularity violations and cause architecture degradation.

B. Weaknesses

Our primary studies did not focus on maintenance effort of the code decay with respect to architecture violations or the code smells. Primary studies focused only on the human-based and semi-automated approaches and not automated approaches. Researchers presented rule-based interpretations using heuristics with thresholds and domain specific languages but not focused on rules that violate architectural styles and design patterns.

C. Research Issues

From the current state-of-the art of code decay detection techniques, we can infer that there is an opportunity in the following research areas.

1) Automated Classifiers: There is need for research on automated detection techniques of code decay. Automated detection means automatic decision-making in identifying violations in architectural rules, design rules, and source code standards. There is a need to build automated classifiers that support developers in locating architecturally-relevant code smells and detecting the violations in architecture.

2) Deriving Architecture Constraints: Research should be conducted in evaluating the implemented architecture of the software system with the goal of deriving rules for architecture styles and design patterns used in the system. Research and evaluation techniques are needed to prevent code decay by automatically identifying architecture and design rule violations at the time of check-in to the version control system during the implementation phase of software development life cycle.

3) Representation of Architecture: van Gurp and Bosch [44] indicate expressiveness of representing the architecture is one of the research challenges of the large and complex systems. Applying the research on visual analytics to represent the software architectures and to track the violations of the architecture over different versions of the system is another important area of research.

4) Cost–Benefit Analysis: Another challenge is to measure the maintainability of large and complex software systems. Research should be conducted on the cost of refactoring, based on prioritization of categories of architecture violations and design/code smells. The relationship between code decay and the maintenance effort over different versions deserves investigation.

5) Identifying Best Practices: There is a need to identify the best practices for identifying architectural violations and design paradigms. The research focus must be on identifying and minimizing code decay with respect to procedures, technologies, methods or tools, and by aggregating information from empirical evidence.

6) Terminology: Research should be conducted to operationalize the various code decay related terms to move toward a consensus in defining the phenomenon of code decay at various levels of abstraction.

D. Limitations

One of the limitations of this review is possible bias in selection of our primary studies. To ensure that the selection process was unbiased, we developed a research protocol based on our research questions. We selected our data sources and defined a search string to obtain the relevant literature. Since the software engineering terms are not standardized, there is a risk that the search results might omit some of the relevant studies. To reduce this risk, we did a bibliography check of every article we selected for primary studies. Another limitation of this study is that only authors participated in the selection and analysis of the papers. We mitigated this risk by having discussions on the inconsistencies raised while conducting our study. Another potential limitation is papers that do not emphasize time or successive versions of a system were excluded in this study.

V. Conclusions

This paper described a systematic mapping study that targeted empirical studies of detection techniques and metrics used in code decay. A total of 30 primary studies were selected using a well-defined review protocol. The three contributions of this paper are the following. First, we categorize different terms used in the literature that leads to code decay with respect to the violations in architectural rules, design rules and source code standards. Second, we classify the code decay detection techniques into human-based and metric-based approaches. Subcategories of these approaches are also discussed. Finally, we present a comprehensive tabular overview of metrics used to identify code decay and their relationship with code decay. Metrics identified to detect code decay help to assess the severity of code decay and to minimize it. Coupling related metrics are widely used and helpful at identifying code decay.

References

[1] A. Bandi, B. J. Williams, and E. B. Allen, "Empirical evidence on code decay: A systematic mapping study," Mississippi State University, Tech. Rep. 06152013, 2013.

[2] E. Bouwers and A. van Deursen, "A lightweight sanity check for implemented architectures," *IEEE Software*, vol. 27, no. 4, pp. 44–50, 2010, [Primary Study].

[3] J. Brunet, R. A. Bittencourt, D. Serey, and J. Figueiredo, "On the evolutionary nature of architectural violations," in *19th WCRE*, 2012, [Primary Study].

[4] O. Ciupke, "Automatic detection of design problems in object-oriented reengineering," in *TOOLS*, 1999, pp. 18–32, [Primary Study].

[5] D. Cotroneo, R. Natella, and R. Pietrantuono, "Is software aging related to software metrics?" in *2nd International Workshop on Software Aging and Rejuvenation*, 2011, [Primary Study].

[6] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and Software Technology*, vol. 50, 2008.

[7] ——, "Strength of evidence in systematic reviews in software engineering," in *Second ESEM*, 2008, pp. 178–187.

[8] S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, 2001, [Primary Study].

[9] M. Fowler and K. Beck, *Refactoring: Improving the design of existing code*. Addison Wesley, 1999.

[10] J. Garcia *et al.*, "Identifying architectural bad smells," in *European Conference on Software Maintenance and Reengineering*, 2009.

[11] M. W. Godfrey and E. H. S. Lee, "Secrets from the monster: Extracting Mozilla's software architecture," in *2nd International Symposium on Constructing Software Engineering Tools*, 2000.

[12] T. Hall *et al.*, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, 2012.

[13] S. Hassaine, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "Advise: Architectural decay in software evolution," in *16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 267–276, [Primary Study].

[14] L. Hochestein and M. Lindvall, "Combating architectural degeneration: A survey," *Information and Software Technology*, vol. 47, no. 10, 2005.

[15] L. Hochstein and M. Lindvall, "Diagnosing architectural degeneration," in *28th Annual NASA Goddard Software Engineering Workshop*, 2003.

[16] C. Izurieta and J. M. Bieman, "Software designs decay: A pilot study of pattern evolution," in *1st ESEM*, 2007, pp. 449–451, [Primary Study].

[17] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, *Reporting experiments in software engineering guide to advanced empirical software engineering*. London: Springer, 2008.

[18] E. Johansson and M. Host, "Tracking degradation in software product lines through measurement of design rule violations," in *SEKE*, 2002, [Primary Study].

[19] F. Khomh, M. D. Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th WCRE*, 2009, [Primary Study].

[20] B. A. Kitchenham, D. Budgen, and O. P. Brereton, "Using napping studies as the basis for further research–a participant-observer case study," *Information and Software Technology*, vol. 53, 2009.

[21] B. A. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University and University of Durham, Tech. Rep. EBSE, 2007.

[22] Z. Li and J. Long, "A case study of measuring degeneration of software architectures from a defect perspective," in *18th Asia Pacific Software Engineering Conference*, 2011, pp. 242–249, [Primary Study].

[23] M. Lindvall, R. Tesoriero, and P. Costa, "Avoiding architectural degeneration: An evaluation process for software architecture," in *Eighth IEEE Symposium on Software Metrics*, 2002, [Primary Study].

[24] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *16th European Conference on Software Maintenance Reengineering*, 2012, [Primary Study].

[25] I. Macia, A. Garcia, A. von Staa, J. Garcia, and N. Medvidovic, "On the impact of aspect-oriented code smells on architecture modularity: An exploratory case study," in *Fifth Brazilian Symposium on Software Components, Architectures and Reuse*, 2011, [Primary Study].

[26] M. V. Mäntylä, J. Vanhanen, and C. Lassenius, "Bad smells–humans as code critics," in *20thICSM*, 2004, pp. 399–408, [Primary Study].

[27] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," in *39th TOOLS*, 2001, pp. 173–182, [Primary Study].

[28] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, 2001.

[29] M. C. Ohlsson *et al.*, "Code decay analysis of legacy software through successive releases," in *IEEE Arerospace Conference*, 1999, pp. 69–81, [Primary Study].

[30] S. M. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *3rd ESEM*, 2009, [Primary Study].

[31] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells are harmful? a study of God classes and Brain classes in the evolution of three open source systems," in *26th ICSM*, 2010, [Primary Study].

[32] D. L. Parnas, "Software aging," in *16th ICSE*, 1994, pp. 279–287.

[33] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, 1992.

[34] D. Raţiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws," in *8th European Conference on Software Maintenance and Reengineering*, 2004, [Primary Study].

[35] D. Radjenović *et al.*, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, 2013.

[36] M. Riaz, M. Sulayman, and H. Naqui, "Architectural decay during continous software evolution and impact of 'design for change' on software architecture," *Communications in Computer and Info. Science*, vol. 59, 2009, [Primary Study].

[37] J. Rosik, A. L. Gear, J. Buckley, M. A. Babar, and D. Connolly, "Assessing architectural drift in commercial software development: A case study," *Software–Practice and Experience*, vol. 41, pp. 63–86, 2011, [Primary Study].

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

[39] S. Sarkar, G. Maskeri, and S. Ramachandran, "Discovery of architectural layers and measurement of layering," *The Journal of Systems and software*, vol. 82, no. 11, pp. 1891–1905, 2009, [Primary Study].

[40] T. Schanz and C. Izurieta, "Object oriented design pattern decay: A taxonomy," in *ESEM*, 2010, [Primary Study].

[41] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *ESEM*, 2010, [Primary Study].

[42] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *12th WCRE*, 2005, pp. 7–11, [Primary Study].

[43] R. T. Tvedt, P. Costa, and M. Lindvall, "Does the code match the design? A process for architecture evaluation," in *ICSM*, 2002, [Primary Study].

[44] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *The Journal of Systems and Software*, vol. 61, pp. 105–119, 2002, [Primary Study].

[45] S. Vaucher *et al.*, "Tracking design smells: Lessons from a study of God classes," in *16th WCRE*, 2009, pp. 145–154, [Primary Study].

[46] C. Wohlin *et al.*, *Experimentation in Software Engineering*. Berlin: Springer, 2012.

[47] S. Wong, M. Kim, and M. Dalton, "Detecting software modularity violations," in *16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 411–420, [Primary Study].

[48] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *IEEE ICSM*, 2012, [Primary Study].