Technical Debt Resulting from Architectural Degradation and Code Smells: A Systematic Mapping Study

Dipta Das
Baylor University
Waco, TX 76798
dipta_das1@baylor.edu

Abdullah Al Maruf
Baylor University
Waco, TX 76798
maruf_maruf1@baylor.edu

Rofiqul Islam
Baylor University
Waco, TX 76798
rofiqul_islam1@baylor.edu

Noah Lambaria
Baylor University
Waco, TX 76798
noah_lambaria1@baylor.edu

Samuel Kim
Baylor University
Waco, TX 76798
samuel_kim1@baylor.edu

Amr S. Abdelfattah
Baylor University
Waco, TX 76798
amr_elsayed1@baylor.edu

Tomas Cerny
Baylor University
Waco, TX 76798
tomas_cerny@baylor.edu

Karel Frajtak
Czech Technical University
Prague, Czechia
frajtak@fel.cvut.cz

Miroslav Bures
Czech Technical University
Prague, Czechia
buresm3@fel.cvut.cz

Pavel Tisnovsky
Red Hat
Brno, Czechia
ptisnovs@redhat.com

ABSTRACT

Poor design choices, bad coding practices, or the need to produce software quickly can stand behind technical debt. Unfortunately, manually identifying and managing technical debt gets more difficult as the software matures. Recent research offers various techniques to automate the process of detecting and managing technical debt to address these challenges. This manuscript presents a mapping study of the many aspects of technical debt that have been discovered in this field of study. This includes looking at the various forms of technical debt, as well as detection methods, the financial implications, and mitigation strategies. The findings and outcomes of this study are applicable to a wide range of software development life-cycle decisions.

CCS Concepts

•**Software and its engineering** → **Maintaining software; Software design tradeoffs; Software evolution;**

Keywords

Technical Debt, Architectural Degradation, Code Smells, Architectural Debt, Design Debt, Code Debt

1. INTRODUCTION

Intense competition in the modern software industry forces companies to produce their products and release new versions under strict time constraints. To meet these deadlines, companies often adopt shortcuts in software development and maintenance.

Technical debt refers to these shortcuts, and the resultant poor software quality [79]. It applies a financial metaphor to express the tradeoffs between short-term benefits and long-term costs of the Software Development Life-Cycle (SDLC) [68, 4]. Studies found that technical debt is primarily created as a result of deliberate decisions made to satisfy customers [79]. Technical debt can be a worthwhile investment as long as the project team is aware of its presence and the higher risks it entails [61]. If effectively managed, technical debt can assist the project in achieving its objectives sooner or more inexpensively [61]. Technical debt can also occur unintentionally due to poor design choices and bad coding practices [21, 74]. While code-level debt can be easily identified and fixed with minimal effort, design debt is difficult to detect and resolve [74]. Also, as software increases in complexity during development, technical debt becomes harder to manually detect and manage [34]. To overcome these challenges, recent studies propose several tools to automate the process of identifying, measuring, and managing technical debt. However, due to different definitions of technical debt and the lack of agreement among these tools, automated technical debt assessments remain complex and hard to generalize [4, 39].

The purpose of this paper is to categorize different aspects of technical debt discovered in this field of research. These include analyzing everything from the types of technical debt, detection tools, and mitigation strategies. We summarized the obstacles of efficient measurement of technical debt. Furthermore, we discussed the detection strategies of architectural smells and code smells, two principal subsets of technical debt. Also, we analyzed the financial impact of technical debt described in recent studies.

The findings and conclusions of this study are relevant for a variety of decisions in the software development lifecycle, e.g., restructuring architecture, refactoring codebase, adopting new technologies where cost and time analysis are crucial.

The rest of the paper is organized as follows. Section 2 gives a general background and lists related work on technical debt, architectural smells, architectural degradation, and code smells. Section 3 describes how the authors collected and analyzed the relevant papers on technical debt. Section 4 answers the research questions listed in section 3. The validity of our systematic study is discussed in Section 5. Finally, we conclude the paper in Section 6 with a general summary of our contributions along with future works.

2. BACKGROUND AND RELATED WORK

Technical debt is defined in many ways in different literature. The most common definition is that Technical debt is a future cost attribute due to code smells, architectural flaws, or any other reason in production-level code that needs to be fixed [30, 19, 41, 48, 73, 69, 5, 43, 42]. Technical debt can occur for many reasons, although the most common reason is the speeding-up process of software development [30, 41, 73, 69, 5, 43]. Developers often ignore some low-profile requirements to finish the project on deadline and postpone them for future work. It is a common practice in the software development process, since clients require updates in every assessment and a working project by the deadline. Moreover, there are also some unintentional and undiscovered flaws that only can be identified when the project runs on the production platform [43].

Technical debt is a metaphorical representation introduced by Ward Cunningham to describe a specific kind of problem that deteriorates software day by day. In 1992, Ward Cunningham, in his experience report [19, 41, 43] for the OOPSLA conference, describes technical debt as the following:

> *"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite…The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."*

Technical debt is considered similar to financial debt [22]. In financial debt, we have to return some extra financial credit as interest with the capital. Similarly, in technical debt, we have to pay some extra labor for code or design refactoring when we attempt to recover it [43]. Oftentimes technical debt can be inescapable, with developers taking shortcuts in the hopes of having some form of savings [60]. Likewise to financial debt, it is also crucial to evaluate payback strategies in further assessing the cost [55].

The importance of studies on technical debt is significant for the maintenance of running software on production. There is a vital influence of technical debt on software quality and lifetime [30]. No client wants a great working project which will no longer work perfectly or break in the future. The study on technical debt needs to be continued since there are multiple questions left that are not answered yet. Firstly, there are multiple tools for identifying and measuring architectural degradation, but we can not select any of them as standard [7, 62, 30, 67, 37]. Secondly, there is no standard measurement unit for architectural degradation which makes measuring technical debt very difficult [4, 40]. Thirdly, there is no standardized mapping between the priority scale of a technical debt and its type. [4]. Such a mapping is very important because it can give us a guideline when we face multiple technical debts in the same project by suggesting what kind of technical debt needs to be fixed first and which can be solved later. Even sometimes holding technical debt is better for software, so we need to find when it is beneficial to keep technical debt [61]. Fourthly, the relation between technical debt and economical debt is also an important concern that involves estimating the cost of different kinds of technical debts and determining when and by whom those will be paid [10, 20].

Technical debt arises from many sources. Architectural degradation and Code smells are two main factors for occurring technical debt [48, 73, 26, 67, 43, 37, 78].

The term code-smell was first coined by Kent Beck in the 1990s. Unlike traditional bugs, code smells are violations that can potentially impact the performance of code [18]. Furthermore, code smells can be subjective as there are variations of what is considered both harmful and non-harmful smells.

Code smell detection is essential in order to prevent future flaws or issues that could occur during the SDLC. As Mandić et. al mentioned in [49], researchers are able to grasp and identify technical debt and architectural degradation by examining code smells. Based on data provided from a research questionnaire [14], some of the most prevalent code smells that software developers encounter are duplicated code, large classes, long methods, etc. These are just a few of many smells that contribute to architectural degradation and erosion. When encountering these types of code smells, it is typically common that some form of refactoring occurs to dispose of the smell. Not only does this enhance the code, but it also improves source code maintainability and comprehension, which developers can keep track of [1]. There is also a significant amount of tools and static analyzers on the market which assist in identifying code smells [23].

Architectural degradation is the process of the actual architecture of a system deviating from the intended architecture [8]. This, like technical debt, occurs when faulty code is added to a system without consideration for its long-term consequences [78]. Architectural degradation can manifest in the form of architectural smells such as cyclic dependencies, hub-like dependencies, unstable dependencies, cyclic hierarchies, scattered functionality, god components, abstraction without decoupling, multipath hierarchies, ambiguous interfaces, unutilized abstractions, implicit cross-module dependencies, and architecture violations [7].

Architectural degradation can be caused by architectural debt, also known as architectural technical debt, a subset of technical debt [78]. Architectural technical debt can be understood as an erroneous architectural relation between files that accumulates a maintenance cost over time. [78]. A systematic literature review on architectural degradation in open source software [8] found its main causes to be rushed system evolution, recurring changes, lack of awareness on the part of developers, time pressure, and design decision accumulation.

3. MAPPING STUDY METHOD

In this study, we carried out a structured procedure to accumulate and synthesize the research works on technical debt in SDLC. Although there are many recent studies discussing technical debt, in this paper, we narrowed our focus on architectural debt and code smells, two primary constituents of technical debt. To find existing groundwork relevant to the specific field of interest, we followed a software engineering approach of systematic mapping studies [57].

In the first phase of our mapping study, we defined a set of research questions and refined them over time such that the answers to those questions represent a comprehensive understanding of the topic under consideration: technical debt resulting from architectural degradation and code smells. Next, we identified the search terms for querying across different indexing sites and portals. This phase required a trial-and-error process to finalize the search terms relevant to our topic. Once all papers were gathered, we manually filtered out out-of-scope papers by reading through the abstracts and prepared a shortlist. Finally, we rigorously analyzed the shortlisted papers to answer the research questions.

The questions we examined in this study are as follows:

RQ1 What is the trend in research over time? Has more or less been done recently?

RQ2 Which countries and people are most active in this field of research?

RQ3 What are the types of technical debt?

RQ4 What are the common strategies to detect technical debt? What tools are used?

RQ5 How to measure technical debt? What are the limitations?

RQ6 What are the common ways to manage technical debt?

RQ7 What is the relationship between architectural degradation and technical debt? What approaches and tools are used?

RQ8 How do code smells cause technical debt? How to detect them?

RQ9 How can we estimate the economical impact of architectural degradation?

RQ10 What are the future directions for detecting and managing technical debt?

Table 1: Search Query Results for Various Indexer Sites

Indexer	Search Results	Filtered
ACM DL	191	33
IEEE Xplore	75	16
SpringerLink	46	8
ScienceDirect	25[1]	4
Total	337	61

We used four indexing sites and portals (indexers), including IEEE Xplore, ACM Digital Library (DL), ScienceDirect, and SpringerLink. We tailored our search query to look for papers related to technical debt caused by architectural degradation and code smells. We divided our search query into three parts. In the first part of our query, we included the search term *"technical debt"*; however, we also considered related terms like *"code debt"* and *"design debt"*. In the second part, we used similar terms that represent architectural degradation. For the last part, we injected the terms relevant to code smells. The full search query is presented in the Listing 1.

Each paper after the initial filtering was then manually evaluated on its title and abstract to determine if it was a fit for the scope we are looking for. Our search query returned a large number of papers; however, after the manual processing, we found that most of them are related to failure detection instead of failure prediction. Apart from that, we have also discarded non-English papers, papers without available full-text, opinion-based papers, and short papers with less than four pages. Then we went through the related work section of the remaining papers to include relevant studies that the search query omitted.

The results of our search queries and manual filtering are listed in Table 1 along with the papers we found by exploring related work sections. Once we narrowed down the relevant works to about 61 papers, we thoroughly studied them to discover current trends in technical debt in relation to architectural degradation and code smells.

4. ANALYSIS RESULTS

Out of 337 papers returned by the search, we recognized a very small number of relevant works. The majority of the works focused on technical debt in general. Only 61 papers that discuss technical debt as a result of code smells and architectural anomaly were considered for the final analysis. In this section, we present the findings of the study by answering the research questions in separate subsections. The validity of our systematic study is discussed in the last subsection.

4.1 Trends

The idea of technical debt has been around since the 1990s. Even though research started in the 1990s, most of them were focused on technical debt in general until around 2011 when a larger emphasis was placed on code smells and architectural degradation. This emphasis occurred naturally as a reaction to the ever-growing size and complexity of soft-

Listing 1: The Search Query for the Indexers

```
  ("technical debt" OR "code debt" OR "design debt" OR "architectural debt")
AND ("architectural smell" OR "architectural anomaly" OR "architectural degradation"
  OR "architectural erosion" OR "architectural decay" OR "architectural drift"
  OR "code smell" OR "code refactoring" OR "code quality")
```

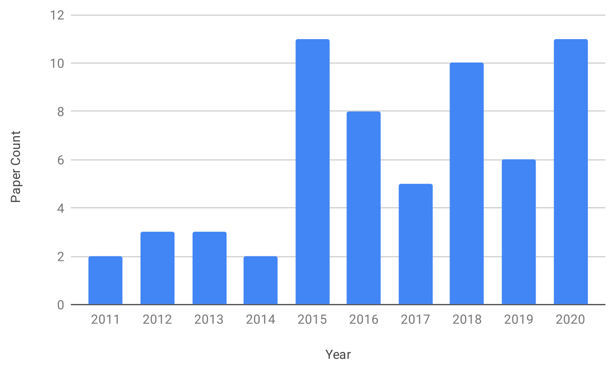


Figure 1: Number of papers found per year

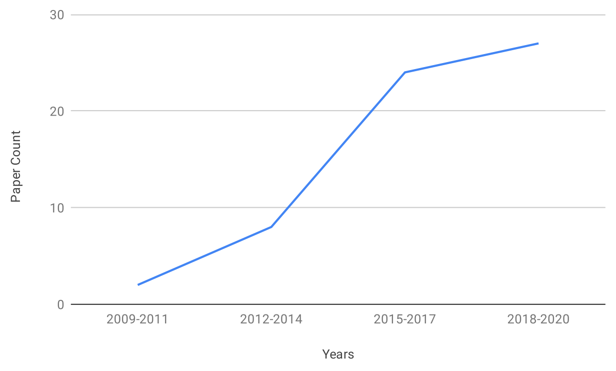


Figure 2: Number of papers found per 3 years

ware systems and the escalating risk technical debt creates. Figure 1 illustrates all of the papers the authors included in this study. As can be seen from the graph above, the general trend of research on technical debt has been increasing over time, especially over the last few years. For the 61 selected papers, Figure 2 shows an sharp increase in the number of papers written after 2014.

4.2 Countries and People

We found several common authors in the selected research papers. Francesca Arcelli Fontana from the University of Milan is the most active researcher in this field, publishing more than 10 articles on technical debt. The other authors who contributed to more than 5 papers are Philippe Kruchten, Ipek Ozkaya, and Lu Xiao. However, in general, each person contributed to one or two papers. This shows us that the research is fairly widespread, and not contained to one group of people. This conclusion can also be seen

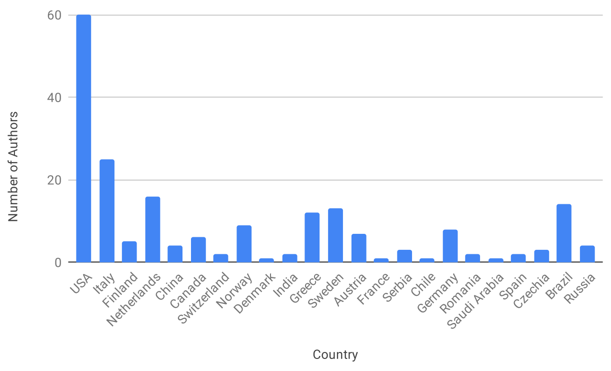


Figure 3: Number of researchers involved per country

in Figure 3, which outlines the number of people in each country that are involved in researching technical debt, architectural degradation, and code smells. Out of the 61 papers analyzed, the United States currently has the most people involved in this research. The other countries that contributed the most to the research of technical debt are Italy, Netherlands, Brazil, Sweden, and Greece.

4.3 Identification of Technical Debt

Different kinds of technical debt identification mechanisms are available on market. Analyzing code, code comments, commits, and architecture are the most used mechanisms for identifying technical debt. Some papers solely worked on the identification of technical debt [48, 73, 78, 71]. Aside from that, most of the related papers identify technical debt for the necessity of it in their work. Solely analyzing code is a widely used method for technical debt identification [7, 62, 13, 26, 67, 20, 43, 56, 12] and there are multiple tools for this which are quite popular, including SonarQube, Arcade, Arcan, Designite, Hotspot detector, etc. Analyzing only commits on the version controlling platform is another used strategy to identify technical debt [66, 64]. Moreover, combining both code and commit analysis is also a popular strategy [48, 9]. Architecture analysis [73, 5, 37] and a combination between code and architecture analysis [30, 41] are also used to identify technical debt in many studies in the literature. Finding patterns [78] is another strategy to identify technical debt.

4.4 Categorization of Technical Debt

Categorizing the technical debt is an important task and several studies tried to do so based on different criteria. Using multiple tools is a common and well-accepted way to categorize technical debt. Azadi and Umberto [7] proposed a catalog of architectural debt which is a subset of technical debt. This paper used 9 tools and established 12 different kinds of architectural debt. Roveda and Riccardo [62] analyzed around 109 open-source projects and defined a new Architectural Debt Index. Another indexing of technical debt was done by Fontana and Francesca [62], which used five tools, some of them being free and others being commercial.

Martini and Stray [51] identified five main types of technical debt in their work: Code debt, Architectural debt, Test Debt, Documentation debt, and Infrastructure debt.

- Code debt mainly arises from different kinds of code smells.
- Architectural debt occurs due to design flaws in project architectures; for example, monolithic architectures are a substandard choice for large projects since dependency handling will cause technical debt.
- Test debt mainly happens due to insufficient test sets and the lack of structured and automated tests.
- Documentation debt arises due to poor levels of code or project documentation like an insufficient description of APIs, an inadequate use-case or domain-model diagram, etc.
- Infrastructure debt mainly occurs due to poor resource management.

Martini and Stray also mentioned non-technical debts such as social debt and process debt. These kinds of debts are mainly generated due to lack of communication and faulty project leading strategy.

4.5 Challenges in Measuring Technical Debt

Measuring technical debt properly is a great challenge for every researcher since there is no fixed unit of technical debt [4, 40, 6] to serve as the analog for money in the economical debt analogy. Different technical debt measurement tools use different criteria as the unit of debt which are not mutually approved by all. When we recover from an economical debt, we pay the principal as well as the interest, totaling to a measurable compensation for the debt. However, because there is no fixed unit for technical debt, completely measuring the compensation of technical debt recovery remains a difficulty. For this reason, there is no fixed scale for an available tool's quality, which is a great challenge to the research in this area.

4.6 Reduction Management of Technical Debt

The study of reducing technical debt is also a hot topic for research. Strategic management [41] is the first approach that we will discuss. It mainly focuses on distinguishing different technical debts and strategies to overcome those debts. It uses various tools, multiple industrial-level source projects, published literature, and other activities on technical debt as the resources for this study. The next approach is getting a better understanding of technical debt. Many papers [43, 56, 49, 37] have discussed the required technical knowledge to reduce technical debt. Some of them also finalized some questionnaires which need to be passed to guarantee the minimal technical debt. Another approach is using automated tools. There are different kinds of tools available in the market, some of them being free, though most are commercial. Designite [67] is a software design quality assessment tool that not only supports comprehensive design smell detection but also provides a detailed metrics analysis. Other similar tools include SonarQube, Arcade, Arcan, Designite, Hotspot detector, etc. Using these types of tools in specific steps of SDLC will help developers to minimize technical debt.

4.7 Strategies to Detect Architectural Degradation

Multiple strategies for addressing architectural degradation involve the use of static code analysis tools which detect code smells [31, 65, 29, 7, 26, 52]. However, one 2017 paper [46] presents issues with the use of code smell detection tools in addressing architectural degradation, such as the finding that Naive Bayes classification models cannot use software-detected code smells to identify classes with architectural inconsistencies and that a relation between architectural inconsistencies and code smells only exists when code smells are detected manually. The study therefore argues that reflexion models are currently more effective than the use of code smell detection tools for determining architectural inconsistencies until detection tools improve.

Other papers mainly present tools that are developed specifically to identify or mend architectural degradation [65, 13, 62, 64]. The Architecture Recovery, Change, and Decay Evaluator (ARCADE) is an example of a tool developed for addressing architectural degradation that also incorporates existing static code analysis tools [65].

In another group of papers, rather than focusing on static code analysis tools or presenting software specifically for architectural degradation, other strategies or models are introduced, such as the ”HCP Matrix”, the ”Design Rule Space”, and a predictive model for identifying the significance of architectural issues [78, 76, 66, 50, 73]. Thus, as summarized in Table 2, the papers can be grouped into three (not mutually exclusive) categories based on whether the strategy presented involves static code analysis tools, involves the use of a tool specifically for architectural degradation, or is a process or model.

The strategies for detecting architectural degradation that involve the use of static code analysis tools are varied. One method is to detect cyclic dependencies through Sonargraph, Sonarqube, and inFusion [31]. As mentioned earlier, static code analysis tools – specifically Dependency Finder and PMD – are also used in ARCADE and are visible in the repository mentioned by Laser et. al [65]. Another paper introduces a method of using of SonarQube, inFusion, Structural Analysis for Java, and Structure 101 to detect the following Quality Indexes: SQALE, technical debt, QDI, Stability Index, and SoC [29]. Numerous static code analysis tools are also mentioned in a catalog that maps common architectural smells to the code analyzers that can detect them [7]. Additionally, Arcan uses the static analysis of compiled source code to detect architectural smells [28].

Multiple papers mainly present software that deal specif-

Table 2: Categories of architectural degradation detection tools

Category	Reference
Tools that involve the use of static code analysis	[31, 65, 29, 7, 28]
Tools developed specifically for architectural degradation	[65, 13, 62, 28]
Presents a process or model	[46, 78, 76, 66, 50, 73, 24, 26]

ically with architectural degradation. One paper explains how ARCADE offers solutions for addressing architectural decay [65], and this is discussed further in section 4.9. Another paper presents Arcan, an open-source tool for detecting architectural smells through analyzing the evolution of individual architectural smell instances [64]. Further papers in this group describe research which expands on Arcan, such as the development a C/C++ port for Arcan which detects five architectural smells [13] as well as the proposal to incorporate into Arcan an Architectural Debt Index, based on measurable factors, for assigning priorities to architectural smells and rating the architectural debt of a project over time [62].

Several papers focus more so on a process or model rather than the use of a specific piece of software for dealing with architectural degradation. One such model is the HCP matrix, a history model introduced in 2016 which is used for estimating the chances for change propagation in files [78]. Another model is the Design-Rule Space (DRSpace), an architecture model that posits that overlapping DRSpaces compose a software architecture, and this is used for architectural debt identification [76]. Furthermore, a different paper proposes the Active Hotspot model for monitoring architecture degradation through analyzing the changes in source files and their relations to other files over time for a given issue [24]. Another paper proposes a predictive model which allows for automatic detection of architecturally significant issues and identification of the significance of new issues. Shahbazian et. al in their paper [66] explain this prediction process by describing how they both gather the textual information and the architectural significance of project issues and apply their machine learning approach. In another paper, a holistic framework is proposed for identifying and estimating architectural debt through the use of a measurement system and estimation formula [50]. Moreover, a different paper proposes the use of ”Self-admitted” Architectural Technical Debt, ”abstracted code evolution analysis”, and an approach for gathering information from multiple sources to identify architectural technical debt [73]. In another study, multiple Views, including a Related Code Smell View, are proposed for analyzing architectural issues based on detected code smells [26]. Finally, the Reflexion Modeling process [46, 54] can be used as a method for finding architectural inconsistencies by creating modules that describe the software's intended architecture and mapping the source code onto those modules. This allows for comparing the software's intended architecture with its actual architecture, revealing architectural debts.

4.8 Code Smells in Relation to Technical Debt

Harmful code smells are typically a result of bad design or quality. This results in the lack of maintainability, leading to both technical debt and architectural degradation as software evolves. Code smells are an indication of design flaws; however, they can be eradicated through continuous refactoring. Based on our mapping study, only a small portion of the total papers mentioned code smells.

Some further anomalies that were mentioned while discussing code smells were the following:

- Design Smells [59]
- Architectural smells [62, 27, 15, 32]
- Structural Anti-patterns [26]
- Abstraction Smells [35]

Although architectural smells and code smells differ, they sometimes can be uncovered within the same files. As mentioned in [26], architectural smells are also caused due to violations of design principles. Based on the work by Plösch et. al [59], both code smells and design smells are interchangeable terms that refer to the same concepts and violations.

Some prevalent code smells as mentioned in three papers [14, 26, 3] that software developers encounter and researchers discuss are the following:

- Duplicated Code [14, 38]
- Large Class [14]
- Long Method [14, 38]
- God Class [26, 3, 59, 38, 25]
- Data Class [26]
- Brain Method [26, 25]
- Shotgun Surgery [26]
- Dispersed Coupling [26]
- Message Chains [26]

This is a non-exhaustive list of some of the types of code smells that contribute to technical debt, hindering the maintainability of software. The authors in each of the papers [14, 26, 3, 59] provide a detailed analysis as well as definitions of each type of smell, including statistics on how frequently they appear.

4.8.1 Relationship with Technical Debt

Bad code smells can sometimes result in technical debt. From our findings, code smells coincide with technical debt significantly more than they coincide with architectural degradation. Only a minor percentage of papers mentioned both architectural degradation and code smells, demonstrating that it typically is more associated with technical debt. To manage technical debt, it typically is essential to have some form of interaction with static analyzer tools, which will be further discussed. Fontana et. al [26] assumes that specific code anomalies are more inclined to be indicators of architectural degradation and technical debt.

A majority of papers discussed both technical debt and code smells together. As seen in [3], research suggests that code smells are the most analyzed and mentioned indicators of technical debt. Furthermore, Alves et. al elaborated that on when beginning to detect technical debt, one of the best candidates is god classes. This is because they are thirteen times more likely to be affected with faults [3] and are the most conceptually identifiable. Further papers, such as in [62], suggest that architectural issues are a substantial source of technical debt. Code smells also result in design debt, leading to potential refactoring.

4.8.2 Tools Used

Our findings suggest that there are several commonly available tools to detect code smells. Several papers mentioned static analyzers, which assist in the process of identifying various code smells [44]. These tools are also efficacious at revealing quality rule violations. Static analyzers that are used to identify technical debt and architectural degradation as previously examined are also beneficial for recognizing code smells. As Falessi and Kruchten mentioned in [23], these static analyzers provide analysis of code smells such as god classes or god clones. Some of the more commonly mentioned tools to examine code smells were Kaleidoscope, HistoryMiner, Arcan, Arcade, and CodeVizard. These tools will be further discussed later in the paper.

4.8.3 Obstacles and Challenges

If bad code smells are not caught early on, it can lead to significant technical debt and oftentimes go undetected. This also significantly impacts the performance of software systems. Most papers highlight challenges while managing code smells. Although static analyzers are beneficial and speed up the process, Haendler et al. [35] mentions that most code smells must be manually observed and inspected. Another obstacle that Fontana et al. [26] indicates is that there is no specific standard set of base measures that evaluate software architecture quality. Similarly, Li et. al [47] discusses how it is challenging to know ahead of time what technical debt items (e.g., code smells) will have the highest cost.

4.9 Architectural Degradation, Code Smells, and Technical Debt Tools

There are many tools on the market that are beneficial in examining architectural degradation, technical debt, and code smells. These tools differ in many ways, whether by input type, output type, analysis strategy, objective, etc. In this work, we have analyzed 22 tools that address these topics and categorize them into four different groups, those being Code-based, Code and design-based, Commit-based, and Architecture and design-based. Table 3 presents the categorization for each tool based on its usefulness in a particular area.

4.9.1 Code-Based Tools

In our study, the Code-based category covers a large set of tools. Normally, Code-based tools take source code as input and analyze it to find out architectural degradation, technical debt, or code smells.

SonarQube [70, 63] is a code-based tool that checks code quality against a set of coding rules. It can operate over around 27 programming languages including Java, C/C++, C#, Python, etc. It finds out code smells by reporting about duplicate code, coding standards, unit tests, code coverage, code complexity, comments, bugs, security vulnerabilities, and other significant flaws in code. Also, it is able to detect the cyclic dependency architectural smell [31].

Designite [67] is another code-based tool that only works on the C# programming language and is used as an assessment tool for software design and quality. It gives detailed metric analysis along with comprehensive design smell detection by accessing the AST to prepare a hierarchical meta-model which is used for inferring design smell. The main features of this tool are metric analysis, dependency analysis, hotspot analysis, code-clone detection. Additionally, this tool can detect nine architectural smells [7].

ARCADE [45] is the next code-based tool and works only on the Java programming language. It is a software workbench that employs a suite of architecture recovery techniques and a set of metrics for measuring different aspects of architectural change. It can detect both architectural degradation and code smells. ARCADE has five subsystems – Recovery, Decay Detection, Measurement, Visualization, and Prediction – and the first four only require source code as input.

Sonargraph [36] is another code-based tool that works on Java, C#, C, and C++. It is a prominent tool for modernizing legacy applications, such as converting monolithic projects to microservices. The main features of this tool are code comprehension, quality assessment, detection and elimination of architectural debt, addressing technical debt, and reducing maintenance costs. Furthermore, it can detect cyclic dependencies and architecture violations [7].

The next code-based tools are Cast MRI for Software and Cast Highlight, which are software analysis tools that detect the cyclic dependency architectural smell [13]. Along with detecting that smell, Cast Highlight outputs other code smells for improving resiliency and reducing technical debt as well as an analysis of which applications to prioritize based on their business impact, where the maintenance effort should be prioritized, and where technical debt reduction is best spent [16]. Cast MRI for Software additionally outputs an interactive map of the software architecture and reports of the architectural flaws [17].

CodeVizard [47, 80] is another code-based tool that allows the visualization of code smells within C# and Java projects. Within their work [47], Li et. al highlighted CodeVizard as a potential tool for recognizing code smells. Provided by the University of Maryland, this instrument allows individuals to investigate data from software repositories. As Zazworka and Ackermann discussed in [80], the tool demonstrates that specific code smells that linger in software components "were more change-prone than non-infected components". Like HistoryMiner, CodeVizard is beneficial for empirical studies.

Structure 101 [58, 72] is also a code-based tool that works on C, C++, Java, and several other programming languages. The tool takes the output of the QA C and QA C++ parsing engines as input. This software provides deep structural analysis, dependency management, impact analysis, and dependency analysis.

InFusion [31, 26] is another code-based software analysis tool that detects three architectural smells: cyclic dependencies, SAP breakers, and unstable dependencies. It works on Java, C, and C++ programming languages.

The next code-based tool is Lattix [13, 77]. It is a software analysis tool that creates a Dependency Structure Matrix (DSM) to detect dependencies between architectural components. Lattix is available for multiple programming languages, takes source code as input, and is mostly used for finding architectural degradation.

The Breaking Point Calculator (BPC) [6] is a tool developed in Java for the purpose of validating the FITTED framework and calculating the Technical Debt Breaking Point, the point at which the accumulated interest resulting from the technical debt equals the principal. This program uses SonarQube and Percerons Client to aid in project analysis. The input for this desktop application is the source code of a Java project and the output is, among other information related to the FITTED framework, the principal, interest, and breaking point.

Arcan [13, 7, 28], originally made for Java, is a tool for detecting architectural smells and is composed of four components: the system reconstructor which extracts dependencies, the graph manager which builds dependency graphs, the metrics engine which computes software metrics, and the AS engine which detects architectural debt.

A C/C++ port for Arcan exists [13] which detects five architectural smells: unstable dependencies, hub-like dependencies, cyclic dependencies, multiple architectural smells, and specification implementation violation.

AI Reviewer [7, 2] has the capability to conduct automated code reviews, specifically in C and C++. Furthermore, the tool examines numerous code metrics, which it takes in as its input. Using S.O.L.I.D. design principles, AI Reviewer detects violations. The tool also heavily focuses detection with concrete classes, generating code reviews as well as measurement reports at ease.

4.9.2 Code and Design Based

There are multiple tools in the market which use the code and design of a project jointly to identify architectural degradation and technical debt. In this section, we will talk about some of them.

Titan [77] is a tool for creating and visualizing DRSpaces. The Design Rule Space (DRSpace) is an architectural model that is based on Baldwin and Clark's design rule theory. This model allows for an algorithm that computes the minimal set of DRSpaces which represent the software's erroneous files, and this set is known as the architecture roots. The architecture roots, which are groups of erroneous files that are architecturally related, are analyzed across multiple versions, and if they are consistently erroneous, the group of files is identified as an architectural debt.

STAN [7] is an Eclipse-based structural analysis tool for Java that combines development and quality assurance to visualize the design, help understand code, and measure quality reports and design flaws. STAN can detect the cyclic dependency architectural smell.

Kaleidoscope [35] is a software design analysis tool that assesses smell candidates. Haendler et al. used Kaleidoscope to incorporate a test framework that collects test execution trace data, placing it into a corresponding trace model. Lastly, this tool bundles the trace data to corresponding matrices and generates UML2 sequence diagrams. This is advantageous as the tool assists in the visualization of code smells.

4.9.3 Commit Based

Some of the tools encountered in this study are noteworthy for the way they incorporate commit information as input.

One such tool is the Prediction subsystem of ARCADE [45]. To produce prediction models, ARCADE first extracts the project issues from an issue repository and the architectural smell instances from the Decay Detection subsystem. Relation analyzers then combine this information into correlation data that the model constructors use to output predictions for a system, examples being future issues or the system's proneness to change.

Another tool in this category is HistoryMiner, which Behnamghader et al. utilized [9]. HistoryMiner provides the capability of analyzing every commit in a specific software's history and checking if a code smell is discovered. This tool is beneficial for conducting large-scale empirical studies.

4.9.4 Architecture and Design Based

This category is for a tool which uses the architecture and design of a project as input.

Hotspot Detector [13, 7] is an architecture and design-based tool which is used for Java. It detects five architectural smells identified as hotspot patterns: unstable interface, implicit cross-module dependency, unhealthy interface inheritance hierarchy, cross-module cycle, and cross-package cycle. As input, it takes structural dependencies, coupling information, and clustering information.

4.10 Economical Debt and Its Measurement

In the case study of Besker et al. [11], the authors discussed the influence of technical debt on the productivity of software developers. They conducted interviews with industry professionals to determine how much time was spent overall as a result of technical debt, as well as specific tasks developers spend this time on. According to their results, technical debt is said to be responsible for wasting almost 25% of all developer's working time and this additional time was wasted on additional source code analysis and refactoring, as well as additional testing. Besker et. al [10] also conducted a similar survey. In their findings, on average 36% of all development time is wasted because of technical debt.

The case study of Martini et al. [50] provided a measurement system for architectural technical debt along with a mathematical relationship for calculating interests in terms of additional work and development that was put in. Their case study was based on finding architectural technical debt due to lack of modularization. They utilized the ISO Standard [33] for source code measurement. They estimated both the present extra costs incurred as a result of architectural technical debt and the long-term cost savings achieved by modularization. Authors used Developer Work Months

Table 3: Tools Examined

Category	Tool Name	Input	Output	Domain	Supported Language(s)	Citations
Code based	SonarQube	Source Code	Reports code analysis, Cyclic dependency detection	Arch Debt, Code Smells	27+ [70]	[70, 63, 31]
	Designite	Source Code	Reports code analysis, Arch. smell detection	Arch Debt, Code Smells	C#	[67, 7]
	ARCADE	Source Code	Architectural recovery and change detection	Arch Debt, Code Smells	Java	[45]
	Sonargraph	Source Code	Reports code analysis, Arch. smell detection	Arch Debt, Code Smells, Technical Debt	Java, C#, C, C++	[36, 7]
	Cast Highlight	Source Code	Reports code analysis, Arch. smell detection	Arch Debt, Code Smells	35+ [16]	[13]
	Cast MRI for Software	Source Code	Reports code analysis and architectural flaws	Arch Debt, Code Smells	60+ [17]	[13]
	CodeVizard	Source Code	Visualization of code smells	Code Smells	Java, C#	[47, 80]
	Structure101	Source Code	Analysis of structural metrics	Arch Debt	Java, C, C++, and others	[58, 72]
	InFusion	Source Code	Reports code analysis, Arch. smell detection	Arch Debt, Code Smells	Java, C, C++	[31, 26]
	Lattix	Source Code	DSM	Arch Debt	Java, C, C++, C#, Javascript, Python, and others	[13, 77]
	BPC	Source Code	Technical Debt Breaking Point	Technical Debt	Java	[6]
	Arcan	Source Code	Architectural smell detection	Arch Debt, Code Smells	Java, C, C++	[13, 7, 13]
	AI Reviewer	Source Code	Generates Code reviews and Measurement Reports	Arch Debt, Arch Smells	C, C++	[7, 2]
Code and Design based	Titan	DSM and clustering files	Creates and visualizes DRSpaces	Arch Debt	N/A	[77]
	STAN	Source Code	Structure Analysis	Code Smells	Java	[7]
	Kaleidoscope	Execution trace data	Trace models and UML2 Sequence Diagrams	Code Smells	N/A	[35]
Commit based	ARCADE (Prediction Subsystem)	Issue Information and Source Code	Predictions of future architectural significance for existing issues	Arch Debt	Java	[66]
	HistoryMiner	Commit Information	Code smell detection	Code Smells	N/A	[9]
Architecture and Design based	Hotspot Detector	Structural dependencies, coupling information and clustering information	Architectural smell detection	Arch Debt	N/A	[13, 7]

(DWM) as a unit to represent man-hours of developers. In their case study, after refactoring, the developers were able to save 2.279 DWM/month, which is essentially the interest that is paid each month due to the architectural technical debt.

4.11 Future Directions

The definition and measurement of technical debt are becoming finer over recent years, yet there are many challenges in this field of research [43]. Technical debt depends heavily upon the perspective of how the study was conducted and which tools were used. Many authors aim to investigate technical debt from an additional point of view and experiment with more tools in their future work [31, 53]. It is possible to elaborate on existing models to estimate the mitigation costs of technical debt [59]. Automation of technical debt management can be improved by directly mining artifacts from version control systems like Git [64]. Since many organizations are utilizing research works in different ways based on their context, it is desirable to define a baseline practice for technical debt management [43]. Some authors have also expressed their interest in refining the classification of technical debt, preparing a catalog of detection tools, and collecting examples [56, 7]. Also, most of the research works verified their approach on small projects. Those approaches can be tested against real-world industry-standard benchmark projects [59, 53, 62, 48, 13, 31].

5. THREATS TO VALIDITY

Mapping studies are frequently challenged by several validity issues that must be addressed. We attempted to reduce the impact of issues on the quality of the results and the study's outcome. From the standpoint of Wohlin's taxonomy [75], we explore four possible validity threats: external validity, concept validity, internal validity, and conclusion validity.

5.1 Construct Validity

Construct validity takes into account the studied area concerning the research questions. The primary term technical debt, as well as its immediate extensions code debt and design debt, are coupled with secondary terms as described in Section 3. All of the primary and secondary keywords are widely used as search strings.

Omitting important research from our review might jeopardize the validity of our study. We attempted to mitigate the impact of this threat by choosing and analyzing different search phrases, as well as doing pilot searches for many papers. We set our query to be as broad as feasible to find all relevant work from our original search.

However, another point of view must be taken into account. Our study included four major research databases, including ACM Digital Library, IEEE Xplore, SpringerLink, and ScienceDirect, although the authors only had limited access to indexed full texts on SpringerLink. Other publishers may index and publish additional articles that we did not include. To ensure the impartiality and credibility of the information sources, we included only peer-reviewed papers published by journals or conferences. It does not include reprints of articles submitted to or accepted for publication in arXiv.org, researchgate.net, or individual personal sites. These reprints may feature innovative ideas, techniques, and challenges related to the topic of the articles under consideration.

5.2 Internal Validity

Internal validity questions the techniques used to collect and evaluate data. The search query viewpoint is to ensure that we have acquired all relevant articles on the specified topic. We examined common publication databases for peer-reviewed literature (excluding gray literature). In terms of dependability and repeatability, we specified search keywords and used synonyms that others might duplicate.

Another most likely threat is connected to the extent of the paper's inclusion and exclusion. We spent a lot of time screening and reading the selected articles to assure that they are within the scope of the study considering the vast range of publications on technical debt and the wide variety of research objectives. Section 3 describes the selection criteria in full. For example, we excluded papers that did not provide a specific output, papers that offered suggestions or opinions about technical debt analysis but did not include experiments or robust proposed methods, and literature that focused on topics other than technical debt cause, impact, categorization, measurement, or tools.

A *data extraction bias* is one possible issue. The extraction method, in which just one individual pulls information from the articles, might be the source of this bias. To lessen the impact of this issue, we dispersed data extraction across multiple authors. Furthermore, we had at least three authors examine each search result and double-check the extraction of other authors. The usage of shared spreadsheets for grading and result verification was part of this procedure. This comprised a cursory assessment of the title, abstract, and keywords of the paper, followed by a more in-depth investigation of the whole text with extracts and classification.

To address concerns about *categorization bias*, we created a mind map that was sent to all authors for discussion, comments, and expansion. While the classification is not exhaustive, it represents our perspective on the recognized literature. Given the overlapping perspectives, additional alternative categorizations might be established.

The interpretation of the results may be influenced by *data synthesis bias*. To reduce this risk, the gathered data was synthesised by several authors in many rounds including review sessions.

5.3 External Validity

Knowledge generalization is concerned with external validity. In this study, we gathered data from a wide range of online databases. Our findings and observations apply to technical debt, code smells, and architectural deterioration. They may, however, be partially relevant to other SDLC concerns. Because we evaluated and classified given publications based on the topic of the research, our categorization cannot be implicitly generalized. Furthermore, there is a possibility that missing related work will influence generality since our findings are only the outcome of a peer-reviewed literature search published during 20XX-2021 at four indexing sites.

5.4 Conclusions Validity

The validity of findings is concerned with whether the conclusions are justified on the available evidence. To reduce author bias, extraction bias, and interpretation bias, we had many authors participate in addressing this danger, double-checking the publication's rating and extracts. Several brainstorming sessions resulted in the conclusions. Furthermore, they were all resolved separately by all writers.

6. CONCLUSION

Code debt and architectural debt are the two forms of technical debt that have attracted the most research. One of the consequences of architectural debt is architectural degradation. Both architectural degradation and code smells may originate from violations of design principles. The most used techniques to identify technical debt are analyzing code, comments, commits, and architecture. We divided these strategies into four categories in our mapping study: code-based, code and design-based, commit-based, and architecture and design-based. We looked at 22 tools, and 50 percent of them are code-based. Java, C, C++, and C# are the most commonly supported languages in these code-based tools.

In addition to these four languages, only 34% of code-based tools support other languages. This demonstrates a lack of available tools for other languages. The remaining tools in each category made up about 10% to 13% of the total we considered. This also demonstrates the potential for study in the categories of commit-based as well as architectural and design-based research.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research (https://research.redhat.com).

8. REFERENCES

[1] M. Abidi, M. Grichi, F. Khomh, and Y.-G. Guéhéneuc. Code smells for multi-language systems. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19. Association for Computing Machinery, New York, NY, USA, 2019. doi: 10.1145/3361149.3361161

[2] AI Reviewer. AI Reviewer Features. Accessed on Dec. 9, 2021. [Online]. Available: `https://www.aireviewer.com/features/`, 2021.

[3] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016. doi: 10.1016/j.infsof.2015.10.008

[4] T. Amanatidis, N. Mittas, A. Moschou, A. Chatzigeorgiou, A. Ampatzoglou, and L. Angelis. Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities. *Empirical Software Engineering*, 25(5):4161–4204, Sept. 2020. doi: 10.1007/s10664-020-09869-w

[5] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, P. Abrahamsson, A. Martini, U. Zdun, and K. Systa. The perception of technical debt in the embedded systems domain: An industrial case study. In *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, pp. 9–16, 2016. doi: 10.1109/MTD.2016.8

[6] A. Ampatzoglou, A. Michailidis, C. Sarikyriakidis, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. A framework for managing interest in technical debt: An industrial validation. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, p. 115–124. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3194164.3194175

[7] U. Azadi, F. A. Fontana, and D. Taibi. Architectural smells detected by tools: A catalogue proposal. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops. IEEE Press, 2019. doi: 10.1109/TechDebt.2019.00027

[8] A. Baabad, H. B. Zulzalil, S. Hassan, and S. B. Baharom. Software architecture degradation in open source software: A systematic literature review. *IEEE Access*, 8:173681–173709, 2020. doi: 10.1109/ACCESS.2020.3024671

[9] P. Behnamghader, P. Meemeng, I. Fostiropoulos, D. Huang, K. Srisopha, and B. Boehm. A scalable and efficient approach for compiling and analyzing commit history. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3239235.3239237

[10] T. Besker, A. Martini, and J. Bosch. The pricey bill of technical debt: When and by whom will it be paid? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 13–23, 2017. doi: 10.1109/ICSME.2017.42

[11] T. Besker, A. Martini, and J. Bosch. Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, p. 105–114. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3194164.3194178

[12] T. Besker, A. Martini, J. Bosch, and M. Tichy. An investigation of technical debt in automatic production systems. In *Proceedings of the XP2017 Scientific Workshops*, XP '17. Association for Computing Machinery, New York, NY, USA, 2017. doi: 10.1145/3120459.3120466

[13] A. Biaggi, F. Arcelli Fontana, and R. Roveda. An architectural smells detection tool for c and c++ projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 417–420, 2018. doi: 10.1109/SEAA.2018.00074

[14] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Limiting technical debt with maintainability assurance: An industry survey on used techniques and differences with service- and microservice-based systems. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, p. 125–133. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3194164.3194166

[15] C. Carrillo, R. Capilla, O. Zimmermann, and U. Zdun. Guidelines and metrics for configurable and sustainable architectural knowledge modelling. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15. Association for Computing Machinery, New York, NY, USA, 2015. doi: 10.1145/2797433.2797498

[16] Cast. Cast Highlight. Accessed on Dec. 9, 2021. [Online]. Available: `https://www.castsoftware.com/products/highlight/outputs-analytics`, 2021.

[17] Cast. Cast MRI for Software. Accessed on Dec. 9, 2021. [Online]. Available: `https://www.castsoftware.com/products/MRI-for-Software`, 2021.

[18] Codegrip. What are code smells? how to detect and

remove code smells? Accessed on Dec. 9, 2021. [Online]. Available: `https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-smells/`, 2019.

[19] W. Cunningham. The WyCash Portfolio Management System. Accessed on Dec. 9, 2021. [Online]. Available: `http://c2.com/doc/oopsla92.html`, Mar. 1992. publisher: OOPSLA.

[20] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the size, cost, and types of technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 49–53, 2012. doi: 10.1109/MTD.2012.6226000

[21] G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. On the Temporality of Introducing Code Technical Debt. In M. Shepperd, F. Brito e Abreu, A. Rodrigues da Silva, and R. Pérez-Castillo, eds., *Quality of Information and Communications Technology*, Communications in Computer and Information Science, pp. 68–82. Springer International Publishing, Cham, 2020. doi: 10.1007/978-3-030-58793-2_6

[22] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, p. 50–60. Association for Computing Machinery, New York, NY, USA, 2015. doi: 10.1145/2786805.2786848

[23] D. Falessi and P. Kruchten. Five reasons for including technical debt in the software engineering curriculum. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15. Association for Computing Machinery, New York, NY, USA, 2015. doi: 10.1145/2797433.2797462

[24] Q. Feng, Y. Cai, R. Kazman, D. Cui, T. Liu, and H. Fang. Active hotspot: An issue-oriented model to monitor software evolution and degradation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, p. 986–997. IEEE Press, 2019. doi: 10.1109/ASE.2019.00095

[25] C. Fernández-Sánchez, J. Garbajosa, C. Vidal, and A. Yagüe. An analysis of techniques and methods for technical debt management: A reflection from the architecture perspective. In *Proceedings of the Second International Workshop on Software Architecture and Metrics*, SAM '15, p. 22–28. IEEE Press, 2015.

[26] F. A. Fontana, V. Ferme, and M. Zanoni. Towards assessing software architecture quality by exploiting code smell relations. In *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*, pp. 1–7, 2015. doi: 10.1109/SAM.2015.8

[27] F. A. Fontana, I. Pigazzini, C. Raibulet, S. Basciano, and R. Roveda. Pagerank and criticality of architectural smells. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, p. 197–204. Association for Computing Machinery, New York, NY, USA, 2019. doi: 10.1145/3344948.3344982

[28] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto. Arcan: A tool for architectural smells detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 282–285, 2017. doi: 10.1109/ICSAW.2017.16

[29] F. A. Fontana, R. Roveda, S. Vittori, A. Metelli, S. Saldarini, and F. Mazzei. On evaluating the impact of the refactoring of architectural problems on software quality. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops. Association for Computing Machinery, New York, NY, USA, 2016. doi: 10.1145/2962695.2962716

[30] F. A. Fontana, R. Roveda, and M. Zanoni. Technical debt indexes provided by tools: A preliminary discussion. In *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, pp. 28–31, 2016. doi: 10.1109/MTD.2016.11

[31] F. A. Fontana, R. Roveda, and M. Zanoni. Tool support for evaluating architectural debt of an existing system: An experience report. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, p. 1347–1349. Association for Computing Machinery, New York, NY, USA, 2016. doi: 10.1145/2851613.2851963

[32] F. A. Fontana, W. Trumler, C. Izurieta, and R. L. Nord. Ninth international workshop on managing technical debt: Report on the mtd 2017 workshop. In *Proceedings of the XP2017 Scientific Workshops*, XP '17. Association for Computing Machinery, New York, NY, USA, 2017. doi: 10.1145/3120459.3120461

[33] I. O. for Standardization/International Electrotechnical Commission et al. Systems and software engineering—measurement process. *ISO/IEC 15939: 2007*, 1, 2007.

[34] Y. Guo, R. O. Spínola, and C. Seaman. Exploring the costs of technical debt management – a case study. *Empirical Software Engineering*, 21(1):159–182, Feb. 2016. doi: 10.1007/s10664-014-9351-7

[35] T. Haendler, S. Sobernig, and M. Strembeck. Towards triaging code-smell candidates via runtime scenarios and method-call dependencies. In *Proceedings of the XP2017 Scientific Workshops*, XP '17. Association for Computing Machinery, New York, NY, USA, 2017. doi: 10.1145/3120459.3120468

[36] Hello2morrow. Sonargraph: Tools to control technical debt and empower software craftsmanship.

[37] C. Izurieta, G. Rojas, and I. Griffith. Preemptive management of model driven technical debt for improving software quality. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '15, p. 31–36. Association for Computing Machinery, New York, NY, USA, 2015. doi: 10.1145/2737182.2737193

[38] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka. A case study in locating the architectural roots of technical debt. In

Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, p. 179–188. IEEE Press, 2015.

[39] I. Khomyakov, Z. Makhmutov, R. Mirgalimova, and A. Sillitti. An Analysis of Automated Technical Debt Measurement. In J. Filipe, M. Śmiałek, A. Brodsky, and S. Hammoudi, eds., *Enterprise Information Systems*, Lecture Notes in Business Information Processing, pp. 250–273. Springer International Publishing, Cham, 2020. doi: 10.1007/978-3-030-40783-4_12

[40] I. Khomyakov, Z. Makhmutov, R. Mirgalimova, and A. Sillitti. An analysis of automated technical debt measurement. In J. Filipe, M. Śmiałek, A. Brodsky, and S. Hammoudi, eds., *Enterprise Information Systems*, pp. 250–273. Springer International Publishing, Cham, 2020.

[41] P. Kruchten. Strategic management of technical debt: Tutorial synopsis. In *2012 12th International Conference on Quality Software*, pp. 282–284, 2012. doi: 10.1109/QSIC.2012.17

[42] P. Kruchten, R. L. Nord, and I. Ozkaya. 4th international workshop on managing technical debt (mtd 2013). In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, p. 1535–1536. IEEE Press, 2013.

[43] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *SIGSOFT Softw. Eng. Notes*, 38(5):51–54, Aug. 2013. doi: 10.1145/2507288.2507326

[44] P. Kruchten, R. L. Nord, I. Ozkaya, and J. Visser. Technical debt in software development: From metaphor to theory report on the third international workshop on managing technical debt. *SIGSOFT Softw. Eng. Notes*, 37(5):36–38, Sept. 2012. doi: 10.1145/2347696.2347698

[45] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 235–245, 2015. doi: 10.1109/MSR.2015.29

[46] J. Lenhard, M. M. Hassan, M. Blom, and S. Herold. Are code smell detection tools suitable for detecting architecture degradation? In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, ECSA '17, p. 138–144. Association for Computing Machinery, New York, NY, USA, 2017. doi: 10.1145/3129790.3129808

[47] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015. doi: 10.1016/j.jss.2014.12.027

[48] Z. Li, P. Liang, and P. Avgeriou. Architectural technical debt identification based on architecture decisions and change scenarios. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pp. 65–74, 2015. doi: 10.1109/WICSA.2015.19

[49] V. Mandić, N. Taušan, and R. Ramač. The prevalence of the technical debt concept in serbian it industry: Results of a national-wide survey. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, p. 77–86. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3387906.3388622

[50] A. Martini, E. Sikander, and N. Madlani. A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component. *Information and Software Technology*, 93:264–279, 2018. doi: 10.1016/j.infsof.2017.08.005

[51] A. Martini, V. Stray, and N. B. Moe. Technical-, social- and process debt in large-scale agile: An exploratory case-study. In R. Hoda, ed., *Agile Processes in Software Engineering and Extreme Programming – Workshops*, pp. 112–119. Springer International Publishing, Cham, 2019.

[52] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pp. 51–60, 2015. doi: 10.1109/WICSA.2015.12

[53] A.-J. Molnar and S. Motogna. Long-term evaluation of technical debt in open-source software. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3382494.3410673

[54] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001. doi: 10.1109/32.917525

[55] I. Ozkaya, P. Kruchten, R. Nord, and N. Brown. Second international workshop on managing technical debt (mtd 2011). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, p. 1212–1213. Association for Computing Machinery, New York, NY, USA, 2011. doi: 10.1145/1985793.1986051

[56] I. Ozkaya, P. Kruchten, R. L. Nord, and N. Brown. Managing technical debt in software development: Report on the 2nd international workshop on managing technical debt, held at icse 2011. *SIGSOFT Softw. Eng. Notes*, 36(5):33–35, Sept. 2011. doi: 10.1145/2020976.2020979

[57] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, Aug. 2015. doi: 10.1016/j.infsof.2015.03.007

[58] Phaedrus Systems Ltd. QA Structure 101. Accessed on Dec. 9, 2021. [Online]. Available: `https://www.phaedsys.com/principals/programmingresearch/pr-structure.html`, 2021.

[59] R. Plösch, J. Bräuer, M. Saft, and C. Körner. Design debt prioritization: A design best practice-based approach. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, p. 95–104. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3194164.3194172

[60] K. Rindell, K. Bernsmed, and M. G. Jaatun. Managing security in software: Or: How i learned to stop worrying and manage the security technical debt. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19. Association for Computing Machinery, New York, NY, USA, 2019. doi: 10.1145/3339252.3340338

[61] N. Rios, R. O. Spínola, M. Mendonça, and C. Seaman. The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from Brazil. *Empirical Software Engineering*, 25(5):3216–3287, Sept. 2020. doi: 10.1007/s10664-020-09832-9

[62] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni. Towards an architectural debt index. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 408–416, 2018. doi: 10.1109/SEAA.2018.00073

[63] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, and S. Romano. On the accuracy of sonarqube technical debt remediation time. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 317–324, 2019. doi: 10.1109/SEAA.2019.00055

[64] D. Sas, P. Avgeriou, and F. Arcelli Fontana. Investigating instability architectural smells evolution: An exploratory case study. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 557–567, 2019. doi: 10.1109/ICSME.2019.00090

[65] M. Schmitt Laser, N. Medvidovic, D. M. Le, and J. Garcia. Arcade: An extensible workbench for architecture recovery, change, and decay evaluation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, p. 1546–1550. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3368089.3417941

[66] A. Shahbazian, D. Nam, and N. Medvidovic. Toward predicting architectural significance of implementation issues. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, p. 215–219. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3196398.3196440

[67] T. Sharma, P. Mishra, and R. Tiwari. Designite - a software design quality assessment tool. In *2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE)*, pp. 1–4, 2016. doi: 10.1109/Bridge.2016.009

[68] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman. Technical Debt: Showing the Way for Better Transfer of Empirical Results. In J. Münch and K. Schmid, eds., *Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach*, pp. 179–190. Springer, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-37395-4_12

[69] S. Soares de Toledo, A. Martini, A. Przybyszewska, and D. I. Sjøberg. Architectural technical debt in microservices: A case study in a large company. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 78–87, 2019. doi: 10.1109/TechDebt.2019.00026

[70] SonarQube. Code Quality and Code Security | SonarQube. Accessed on Dec. 9, 2021. [Online]. Available: `https://www.sonarqube.org/`, 2021.

[71] P. Strečanský, S. Chren, and B. Rossi. Comparing maintainability index, sig method, and sqale for technical debt identification. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, p. 121–124. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3341105.3374079

[72] Structure101. Structural analysis. Accessed on Dec. 9, 2021. [Online]. Available: `https://structure101.com/legacy/structural-analysis/`, 2021.

[73] R. Verdecchia. Architectural technical debt identification: Moving forward. In *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 43–44, 2018. doi: 10.1109/ICSA-C.2018.00018

[74] R. Verdecchia, P. Kruchten, and P. Lago. Architectural Technical Debt: A Grounded Theory. In A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann, eds., *Software Architecture*, Lecture Notes in Computer Science, pp. 202–219. Springer International Publishing, Cham, 2020. doi: 10.1007/978-3-030-58923-3_14

[75] C. Wohlin. *Experimentation in Software Engineering: An Introduction.* International Series in Engineering and Computer Science. Kluwer Academic, 2000.

[76] L. Xiao. Quantifying architectural debts. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, p. 1030–1033. Association for Computing Machinery, New York, NY, USA, 2015. doi: 10.1145/2786805.2803194

[77] L. Xiao, Y. Cai, and R. Kazman. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, p. 967–977. Association for Computing Machinery, New York, NY, USA, 2014. doi: 10.1145/2568225.2568241

[78] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, p. 488–498.

Association for Computing Machinery, New York, NY, USA, 2016. doi: 10.1145/2884781.2884822

[79] J. Yli-Huumo, A. Maglyas, and K. Smolander. The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company. In A. Jedlitschka, P. Kuvaja, M. Kuhrmann, T. Männistö, J. Münch, and M. Raatikainen, eds., *Product-Focused Software Process Improvement*, Lecture Notes in Computer Science, pp. 93–107. Springer International Publishing, Cham, 2014. doi: 10.1007/978-3-319-13835-0_7

[80] N. Zazworka and C. Ackermann. Codevizard: A tool to aid the analysis of software evolution. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10. Association for Computing Machinery, New York, NY, USA, 2010. doi: 10.1145/1852786.1852865

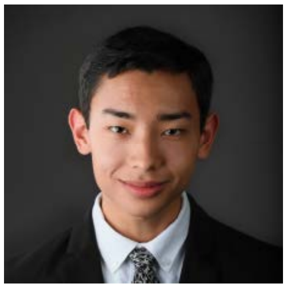
ABOUT THE AUTHORS:

Dipta Das is a software engineer at Amazon. He completed his Master's degree from Baylor University in 2021 and a bachelor's degree from Chittagong University of Engineering and Technology in 2017. In 2021, he received the Outstanding Graduate Student Award from Baylor University. His research interests include software engineering, microservice security, and code analysis.

Abdullah Al Maruf is a graduate student of Computer Science at Baylor University. His research interests include software engineering, code analysis, and runtime log analysis. He received his Bachelor's degree from the Department of Computer Science and Engineering at the Chittagong University of Engineering and Technology in Bangladesh. He has worked in the industry for four years as both a software developer and a DevOps engineer. He is an open-source enthusiast.

MD Rofiqul Islam is a graduate student of Computer Science at Baylor University. His research area includes software engineering, microservices architecture, code analysis, and log analysis. He received his Bachelor's degree from the Department of Computer Science and Engineering at the University of Dhaka in Bangladesh. He has experience as a researcher for more than three years in multiple research groups in different countries. He also served as a software engineer in TigerIT in Bangladesh for more than two years while he worked on SOA and Microservice Architecture.

Noah Lambaria is an undergraduate student of Computer Science at Baylor University. He has worked under the guidance of Dr. Cerny during the summer and fall periods of 2021 in research related to architectural degradation. He aspires to continue research in software engineering, particularly in code analysis.



Samuel Kim is an undergraduate student of Computer Science at Baylor University. He has worked under Dr. Cerny during the summer and fall periods of 2021 in research related to architectural degradation.

Amr S. Abdelfattah is a Ph.D. student and Research Assistant of Computer Science at Baylor University in the USA. His research interests are Cloud Computing, Mobile Cloud Computing, the Internet Of Things, and Software Engineering. He received his Bachelor’s and Master's degrees from the Faculty of Computer and Information Science at Ain Shams University in Egypt. Amr worked 9+ years in the industry as a mobile technical lead for international companies. In addition to being a Lecturer Assistant at Ain Shams University.

Tomas Cerny is a Professor of Computer Science at Baylor University. His area of research is software engineering, code analysis, security, and cloud-based system design. He received Master's and Ph.D. degrees from the Czech Technical University in Prague, and an M.S. degree from Baylor. Dr. Cerny served 10+ years as the lead developer of the International Collegiate Programming Contest and authored nearly 100 publications related to code analysis. Among his awards are the Outstanding Service Award ACM SIGAPP or the 2011 ICPC Joseph S. DeBlasi Outstanding Contribution Award. He chaired multiple conferences, including ACM SAC/RACS, and ICITCS.

Karel Frajtak is a lecturer and researcher with the System Testing IntelLigent Lab (STILL) at the Dept. of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague. He received his Master's and Ph.D. degrees from the Faculty of Electrical Engineering at the Czech Technical University in Prague. His lectures and area of research focus on software testing methods, test automation approaches, and software architectures.

Miroslav Bures leads the System Testing IntelLigent Lab (STILL) at the Computer Science, FEE, Czech Technical University in Prague, where he was appointed in 2010 and currently serves as the Associate Professor of Computer Science. His research interests include quality assurance and reliability methods, model-based testing, path-based testing, combinatorial interaction testing and test automation for software, Internet of Things, and mission-critical systems. He leads several projects in the field of test automation for software and Internet of Things systems, covering the topics of automated generation of test scenarios as well as automated execution of the tests.

Pavel Tisnovsky is known for the in-depth articles he writes on various technical topics for the Czech Linux magazine root.cz. He taught computer graphics at Brno Technical University and worked as a C, C++, and Java developer in various companies before he joined Red Hat, where he was a quality assurance engineer in the OpenJDK team. Now he works as SW developer and tech lead using Python and Go programming languages to develop scalable data processing pipelines and notification systems. He also teaches professional Java and Go training.