2020

Risk Assessment of Architecture Technical Debt

Mrwan Omar Kh. Ben Idris
mobenidris@mix.wvu.edu

Follow this and additional works at: <https://researchrepository.wvu.edu/etd>

Part of the [Other Computer Engineering Commons](#)

Recommended Citation

Ben Idris, Mrwan Omar Kh., "Risk Assessment of Architecture Technical Debt" (2020). *Graduate Theses, Dissertations, and Problem Reports*. 7897.

<https://researchrepository.wvu.edu/etd/7897>

This Dissertation is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Dissertation in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Dissertation has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Risk Assessment of Architecture Technical Debt

Mrwan Omar Kh. Ben Idris

Dissertation submitted to the
Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in
Computer Engineering

Hany H. Ammar, Ph.D., Chair
Dale Dzielski, M.B.A., PMP, CMA, Co. Advisor
Katerina Goseva-Popstojanova, Ph.D.
Brian Woerner, Ph.D.
Saiph Savage, Ph.D.
Marjorie Darrah, Ph.D.

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
Fall -2020

Keywords: Architecture Technical Debt, Architecture Smell, Architecture Smell Risk,
Refactoring, Refactoring Effort

Copyright 2020 Mrwan Ben Idris

Abstract

Risk Assessment of Architecture Technical Debt

Mrwan Omar Kh. Ben Idris

Technical Debt (TD) is a metaphor that refers to short-term solutions in software development that may affect the software development life cycle cost. Researchers have found many TD types. These TD types include but are not limited to code debt (CD), design debt (DD), and architecture technical debt (ATD). Several methods have been used to detect technical debt, such as bad smells, software metrics, and code comments. Although TD has received many researchers' attention, ATD has received less attention compared with CD and DD. We found a lack of tools to deal with ATD in contrast to CD and DD. Avoiding TD altogether is impossible but identifying its risk can help to reduce the impact of the technical debt. However, tracking and assessing the ATD's risk level with the aim of making the refactoring decisions has not been studied in the open literature.

In this dissertation, we systematically study TD and apply multiple case studies to find the research gaps and study technical debt. We survey practicing software engineers to inspect the likelihood, the impact of ATD, and the refactoring benefits and challenges. We will propose a methodology to identify the ATD risk level on software components and apply machine learning techniques to find the ATD risk level. This thesis's main contribution is a novel methodology for assessing the ATD risk level based on architecture smells and software metrics. The proposed methodology based on an empirical approach is used to estimate the ATD risk level by tracking 5,179 architecture smell instances identified in 40 C# project releases collected from the GitHub repository.

The method is validated using a dataset that contains 45 apache java projects and 3,480 packages. We compared our results with related works and assessed our results' accuracy compared to related works results. First, we compared the ATD risk classification with the Quality Depreciation Index Rule (QDIR); the average classification accuracy was 77% (80% with the Critical Severity level). Next, the ATD risk levels were compared with Refactoring Effort levels; the average classification accuracy was 88% and 81% for 3 and 5 levels, respectively. The ATD risk levels were compared with the architecture smell levels; the average classification accuracy was 90% and 89% for 3 and 5 levels, respectively. In addition, we used the Wilcoxon rank-sum test ($\alpha=0.01$) to verify whether the proposed method results are statistically different or not.

Dedication

To my Father and Mother.

To my Wife.

To my Daughters and Son.

To my Brothers and Sister.

To my Friends, and the memory of my best friends:

Essam Elfetori and Akram Elbasyoni

Acknowledgments

I would like to express my greatest gratitude to my committee members. I am especially grateful for the guidance and support of my dissertation committee chairman, Professor Hany Ammar, who gave me the chance to work under his supervision. I offer my special grateful to my co-advisor, Mr. Dale Dzielski, for his persistent, positive guidance. My sincere thanks go to my committee members, namely, Professor Katerina Goseva-Popstojanova, Professor Brian Woerner, Professor Saiph Savage, and Professor Marjorie Darrah. Finally, my gratitude extends to my family, friends, and special thanks go to Dr. Yasser Alshehri, Mr. Mohammed Jamil Ahmad and my research colleague; M.Sc. Samir Deeb.

Contents

1	Introduction	1
2	Problem Statement and Research Objectives	4
2.1	Problem Statement:.....	5
2.2	Research Objectives:	5
2.3	Research Contributions:	6
3	Related Works	8
3.1	Technical Debt Systematic Mapping Studies Related Works	9
3.2	Technical Debt Case Studies Related Works	12
3.3	Architecture Technical Debt Surveys Related Works	14
3.4	Architecture Technical Debt Machine Learning Related Works	16
3.5	Architecture Technical Debt Risk Related Works	19
4	Investigate, Identify and Estimate the Technical Debt: A systematic mapping study ...	22
4.1	Introduction	23
4.2	Goals and Research Questions	23
4.3	Systematic Mapping Implementation	24
4.4	Empirical Studies Metadata	25
4.5	Classification Scheme.....	27
4.5.1	Classification by type of Technical Debt	27
4.5.2	Classification by researchers' investigation	30
4.5.3	Classification by Technical Debt Estimator	32
4.5.4	Classification by Technical Debt Indicator	36
4.5.5	Classification by amount of used tools	41

4.6	Results and Comparison	46
4.6.1	Technical Debt types (RQ1).....	46
4.6.2	Technical Debt investigators (RQ2).....	47
4.6.3	Technical Debt estimators and indicators (RQ3)	47
4.6.4	Used Tools (RQ4)	49
4.7	Threats to Validity	49
4.8	Conclusions	50
5	The Technical Debt Density over Multiple Releases and the Refactoring Story.....	51
5.1	Introduction	52
5.2	Methodology.....	53
5.2.1	Research Goals	53
5.2.2	Research Questions	53
5.2.3	Data Extraction.....	54
5.3	Data Analysis, Results and Discussion.....	56
5.3.1	Research Question 1 (Smell density)	58
5.3.2	Research Question 2 (Smell density increments).....	62
5.3.3	Research Question 3 (Percentage of the eliminated smell density)	66
5.4	Threats to Validity	68
5.5	Conclusion	69
6	Investigating ATD Risk and Refactoring, the Perspective of Software Developers	71
6.1	Introduction	72
6.2	Research Methodology	72
6.3	Result and Discussion.....	74
6.3.1	ATD Likelihood, Impact, and Risk	76
6.3.2	Refactoring preventions	82

6.3.3	Refactoring Benefits.....	84
6.3.4	Refactoring Challenges	85
6.4	Threats to Validity	89
6.5	Conclusions	90
7	Prioritizing Software Components Risk: Towards a Machine Learning-based Approach	92
7.1	Introduction	93
7.2	Methodology.....	93
7.3	Data Analysis and Results	97
7.4	Threats to Validity	105
7.5	Conclusions	106
8	Architecture Technical Debt -Based Risk Assessment	107
8.1	Introduction	108
8.2	Methodology.....	109
8.2.1	Identify the Architecture Technical Debt (ATD).....	109
8.2.2	Assessing the ATD risk level on software components	109
8.2.3	An example explains the proposed methodology	112
8.3	Case Studies.....	115
8.4	Data Analysis, Results, and Discussion.....	118
8.4.1	Architecture Smell Likelihood	119
8.4.2	Architecture Smell Impact	122
8.4.3	Architecture Smell Risk	129
8.5	Threats to Validity	135
8.6	Conclusions	135
9	Methodology Validation	137

9.1	Description of Related Works Methodologies	138
9.1.1	Prioritization of classes for refactoring: A step towards improvement in software quality	138
9.1.2	Estimating refactoring efforts for Architecture Technical Debt	140
9.2	Dataset Used in the Verification Phase	140
9.3	Methodologies Verification	145
9.3.1	Verifying our Methodology by comparing our results with Malhotra's approach	146
9.3.2	Verifying our Methodology by comparing our results with Deeb's approach	148
9.4	Threats to Validity	154
9.5	Conclusions	155
10	Dissertation Conclusions and Future Work	156
10.1	Conclusions	157
10.2	Future Work	160
11	Appendix A. Selected studies	161
12	Appendix B: Survey Questionnaires	164
13	Appendix C: C# Projects	171
14	Appendix D: Apache Java Projects	174
15	Bibliography	178

List of Figures

Figure 4.1 Systematic mapping study method.....	25
Figure 4.2 Number of case studies per database.....	25
Figure 4.3 Case studies and publication year.....	26
Figure 4.4 Case studies and number of citations	26
Figure 4.5 Most studied technical debt types.....	29
Figure 4.6 Investigation methodology	31
Figure 4.7 Technical Debt Elements.....	33
Figure 4.8 Technical debt indicators.....	37
Figure 4.9 Most used tool	42
Figure 4.10 Most TD Estimators	48
Figure 5.1 Architecture Smells types for the studied open-source projects	59
Figure 5.2 Design Smells types for the studied open-source projects	60
Figure 5.3 Architecture Smells types for the studied open-source projects	61
Figure 5.4 The density of architecture smells in the studied open-source projects	63
Figure 5.5 The density of design smells in the studied open-source projects.....	64
Figure 5.6 The density of code smells in the studied open-source projects	65
Figure 5.7 The percentage of the eliminated smells in the studied open-source projects	66
Figure 6.1 Participants experiences	74
Figure 6.2 The most probable AS type from the perspective of all participants	77
Figure 6.3 The most probable AS type from the perspective of different software development roles.....	78
Figure 6.4 The AS with the most significant impact from the perspective of all participants	79
Figure 6.5 The AS with the most significant impact from the perspective of different software development roles	80
Figure 6.6 ATD risk from the perspective of all participants	81
Figure 6.7 Software roles responsible to detect ATD (Perspective of all participants).....	87
Figure 6.8 Software roles responsible to perform refactoring ATD (Perspective of all participants).....	88

Figure 7.1 Information about the components risk for the four projects	101
Figure 7.2 Tracking the level of risk – project 1.....	102
Figure 7.3 Tracking the level of risk – project 2.....	102
Figure 7.4 Tracking the level of risk – project 3.....	103
Figure 7.5 Tracking the level of risk – project 4.....	103
Figure 7.6 MCC for the machine learning.....	104
Figure 7.7 F-Measure for the machine learning.....	105
Figure 8.1 Number of architecture smells for the studied projects.....	119
Figure 8.2 The spearman correlation test between AS likelihood of occurrences.....	122
Figure 8.3 Five level classification for the impact of the architecture smells	123
Figure 8.4 Impact of the Unstable Dependency and Classification.....	124
Figure 8.5 Impact of the Scattered Functionality and Classification.....	124
Figure 8.6 Impact of the God Component and Classification	125
Figure 8.7 Impact of Future Concentration and Classification.....	126
Figure 8.8 Impact of Dense Structures and Classification.....	126
Figure 8.9 Impact of Cyclic Dependency and Classification	127
Figure 8.10 Impact of Ambiguous Interface and Classification	127
Figure 8.11 Architecture Smells with the highest impact.....	128
Figure 8.12 Risk of the Unstable Dependency and Classification.....	129
Figure 8.13 Risk of the Scattered Functionality and Risk Classification	130
Figure 8.14 Risk of the God Component and Risk Classification.....	130
Figure 8.15 Risk of the Feature Concentration and Risk Classification.....	131
Figure 8.16 Risk of the Dense Structure and Risk Classification.....	132
Figure 8.17 Risk of the Cyclic Dependency and Risk Classification	132
Figure 8.18 Risk of the Ambiguous Interface and Risk Classification.....	133
Figure 8.19 Architecture Smells with Very High/High Risk.....	133
Figure 9.1 Number and percentage of affected packages in the apache projects used in the validation phase	142
Figure 9.2 Number of detected architecture smells in the apache projects used in the validation phase	144
Figure 9.3 Confusion Matrix of the ATD Risk and Quality Depreciation Index Rule.....	147

Figure 9.4 Confusion Metrix of the ATD Risk and Refactoring Effort (3 levels).....	149
Figure 9.5 Confusion Matrix of the ATD Risk and Refactoring Effort (5 levels).....	149
Figure 9.6 Confusion Matrix of the ATD Risk and Architecture Smells (3 levels)	151
Figure 9.7 Confusion Matrix of the ATD Risk and Architecture Smells (5 levels)	151

List of Tables

Table 4.1 the exclusion criteria	24
Table 4.2 TD Estimators.....	34
Table 4.3 The exclusion criteria	45
Table 4.4 Papers used more than one tools	46
Table 5.1 Architecture, Design and Code smells definition as defined by Designite	55
Table 5.2 the studied open-source projects from GitHub.....	56
Table 6.1 Answers rate for research questions (RQ1.1, RQ1.2, and RQ1)	75
Table 6.2 Answers rate for research questions (RQ2, RQ3, and RQ4)	76
Table 6.3 ATD Risk from the perspective of different software development roles.....	82
Table 6.4 ATD refactoring preventions that the survey participants experienced	83
Table 6.5 ATD refactoring benefits that the survey participants experienced	85
Table 6.6 ATD refactoring challenges that the survey participants experienced	86
Table 7.1 Information about the four projects used to learn machine learning	93
Table 7.2 Machine learning classifiers	98
Table 7.3 Machine learning classifiers used to classify the component risk.....	99
Table 7.4 Information about the projects used in the case studies	99
Table 8.1 Internal Structure Metrics used to assess the AS severity	111
Table 8.2 Total number of the Internal Structure (Example)	113
Table 8.3 Internal Structure Metrics (ISM) for every architecture smell instance (Example)	113
Table 8.4 Assess the ATD Impact (Severity) (Example)	114
Table 8.5 Assess the AS Risk (Example).....	115
Table 8.6 Statistical characteristics of the selected software projects	116
Table 8.7 Type and number of the identified Architecture smells in the 40 project releases.....	118
Table 8.8 The likelihood of the architecture smells per release	120
Table 9.1 Code smells used by Malhotra	138
Table 9.2 Chidamber and Kemerer Metrics as defined by Malhotra	139
Table 9.3 Apache Java Projects used in the validation Phase	140
Table 9.4 The classification levels used by the Malhotra method.....	146
Table 9.5 Nemenyi post hoc test to determine the difference	154

Chapter 1

Introduction

The “Technical Debt” (TD) metaphor was first used by Ward Cunningham to refer to shortcuts taken during the design and implementation of software. These shortcuts ultimately lead to a difficulty in maintenance [1]. The “code smells” metaphor was first introduced by Fowler [2] who describes a code smell as an indication of a weak solution that leads to a problem impacting the code maintainability. Smells, such as design and code smells, are imperfect solutions to recurring implementation and design problems. Each type can be identified by using a set of metrics and their threshold values. Software development companies are suffering from the TD that negatively impacts internal software quality. This causes high maintenance costs, ultimately leading to internal software quality issues [3]–[7]. TD can be generated either intentionally or unintentionally. The non-strategic consequence of doing a poor job leads to unintentional debt. Intentional debt is more of a cost of doing business and serves as a means to an end [8]. Technical debt can be reduced by refactoring, which has many benefits. However, many challenges can be faced during refactoring. Also, many reasons can prevent developers from paying off the TD. Technical Debt has a negative impact on the software quality and the risk of that impact could be enhanced by increasing the software size. Importantly, the technical debt that was caused intentionally or unintentionally needs to be paid off. Technical debts can be found in any software and it is impossible to pay off all that debt. Project managers and other decision makers must decide which TD needs to be refactored first. For that reason, we have to find the risk of every TD on the software.

In this dissertation we proposed a new methodology to estimate the risk of the Architecture Technical Debt in software. First, we carried out a systematic mapping study to explore the TD phenomena. We looked at four electronic Databases: IEEE Xplore, ACM Digital Library, Springer Link, and Science Direct. We examined the empirical studies that were published from January 2014 to December 2017. This period was selected because it was not investigated in the literature. Many methods were used to identify the technical debt. Bad smells are one of those methods and the most used as a TD indicator. However, there are many types of smells such as code, design, and architecture smells. As a first step, we decided to deeply study these three smells to detect the

technical debt. Second, by conducting multiple case studies, approximately 2 million LOC were gathered from 42 releases composed from 5 open sources from the GitHub repository. Our goals were (1) to find the average smells density for every smell type on the software, (2) to determine if the technical debt is increased over multiple releases, and (3) to look at the percentage of each eliminated TD type. Subsequently, we investigated the technical debt from the perspective of different software development roles by conducting a survey. Thirty three developers, eleven architects, and six project managers participated in this study. Our goals (1) to find out how software development teams rate the ATD risk with respect to each of their roles, (2) to identify the benefits that can be gained from refactoring, (3) to discern the reasons that prevent refactoring activities, and (4) to understand challenges that might arise during the refactoring process. We then used machine learning techniques to identify and classify the ATD severity. In this chapter, we used internal structure metrics to identify and classify ATD severity. Our goal was to show that machine learning techniques can be used to identify and classify the severity of ATD on software components which can help the decision-makers to prioritize the refactoring decisions based on the level of the risk. From the systematic mapping study that we conducted, we observed a lack of methods and tools used to assess the ATD risk. This lack of knowledge hampers the project manager's decision-making process related to ATD. For that reason, we propose a new methodology to assess the ATD risk by tracking the AS types that caused ATD on the software. The goal of this study was to develop a new methodology to help project managers make proper decisions regarding ATD refactoring. The new methodology defines the ATD-based risk as a product of two factors: the likelihood of AS and the impact (severity) of this AS. The impact of an AS instance estimated by using the internal structure metrics (ISM) of the classes causing the AS. By using multiple case studies, we demonstrated the methodology in 40 C# software releases. The applied case studies' results showed us that Cyclic Dependency and God Component have the highest risk on software components. However, a high negative correlation between Cyclic Dependency and God Component was found. We found the impact of the AS is more correlated to the level of the AS risk than to the AS likelihood. Finally, to seek the proposed methodology validity, an apache Java dataset that has 45 project releases was used, and the results were compared with the related works. We need to find how well was our method able to prioritize the refactoring packages? We reached 80 % (at critical ATD risk levels), based on the Quality Depreciation Index Rule (QDIR). We reached 81% (at high risk 3 levels) and 61% (at very high-

risk 5 levels) based on refactoring effort level. We reached 81% (at High risk 3 levels) and 79% (at very high-risk 5 levels) based on the level of the architecture smell. The Wilcoxon rank-sum test ($\alpha = 0.01$) verify there is no statistical difference between the related works and the proposed methodology.

The rest of this dissertation is organized as follows. In Chapter 2, the problem statement, research objectives, and contributions will be presented. In Chapter 3 we highlight the related studies. In Chapters 4 and 5, the conducted systematic mapping study, Investigate, Identify and Estimate the Technical Debt: a systematic mapping study, and the empirical study, The Technical Debt Density over Multiple Releases and the Refactoring Story, are discussed.. In Chapter 6, using a different perspective of software development roles, we explore the risk of the ATD and refactoring benefits, challenges, and preventions. Chapter 7 explains the way we identified and classified the ATD risk using machine learning techniques. Our new methodology is discussed in Chapter 8. The methodology evaluation and verification are discussed in Chapter 9. Finally, Chapter 10 concludes the dissertation and highlighted the future work.

Chapter 2

Problem Statement and Research Objectives

This Chapter highlights our problem statement and research objectives: In Section 2.1, the problem statement will be presented. The research objectives are explained in Section 2.2. Section 2.3 highlights the research contributions.

2.1 Problem Statement:

Tracking the technical debt between releases is not given appropriate consideration in the current practice of software developers. Taking shortcuts to release software on time and refactoring after delivery is still a dominant software development practice. Developers prefer adding new features or fixing bugs instead of refactoring because refactoring does not provide immediate benefits [9]. In fact, refactoring may introduce new bugs. However, these shortcuts ultimately cause difficulties in maintenance and may lead to meeting non-functional requirements. Technical debt needs to be paid off because technical debt has a negative impact on the software quality. The risk of that impact could be amplified by increasing the software size. Undoubtedly, decision-makers and project managers need to decide which technical debt candidate needs to be refactoring. Notably, their decisions must be based on the level of technical debt risk because paying all the technical debt is not beneficial and is also time-consuming. The risk of a technical debt instance can be defined as a combination of the likelihood of a technical debt indicator, such as architecture smells, and the impact of that architecture smells on the internal structure.

The problem addressed in this PhD dissertation is: how to define practical architecture technical debt risk assessment methodologies that can assess the technical debt risk level on the software components based on quantitative measurements? To reach our goal and assess the technical debt risk we need to:

1. Estimate the likelihood of architecture smells types.
2. Estimate the impact (severity) of architecture smell instances on the internal structure of the components.
3. Assess the technical debt risk level of architecture smells instances.

2.2 Research Objectives:

In order to make proper refactoring decisions to pay off the technical debt, improve software quality and reduce maintenance costs, decision-makers and project managers need methodologies to help them make proper decisions regarding ATD refactoring. One methodology could involve tracking the risk of technical debt through multiple releases.

Assessing the risk of technical debt should be considered between releases, and we hypothesize that tracking architecture smells as a technical debt indicator has the capacity to pinpoint the risk

of the technical debt in a software component and can help to mitigate its impacts. We will address about the technical debt risk, taking into consideration its likelihood of appearance and its impacts on the software components.

Baying the technical debt by performing refactoring helps avoid the consequences of that debt but not paying that debt and keep accumulating technical debts may lead the software to “technical bankruptcy” [10]. Adding new components and features to the system may increase the software complexity, and postponing the refactoring leads to accumulating technical debts. All that increases the maintainability costs and may lead to software bankruptcy.

To reduce the risk of the technical debt, we need a methodology that can help - make proper decisions regarding paying off the technical debt, and reducing the software complexity and maintainability costs.

Our objective is to develop a technical debt risk assessment methodology by tracking and extracting data such as architecture smells type, number of classes caused the architecture smell, and number of components depended on the effected package from multiple software releases. The methodology will be based on the likelihood of the appearance of the architecture smells and their impacts on the software components.

The main objective of this research is:

To develop architecture technical debt risk assessment methodology based on factors that could be tracked and extracted from software releases. We need to convert our objective into a problem that can be solved using a quantitative approach to assess the architecture technical debt risk level in the software components.

2.3 Research Contributions:

In this research, we introduce a novel approach for architecture technical debt-based risk assessment methodology to assess the architecture technical debt risk level on software components (packages).

1. We perform a systematic mapping study to get an overview of the research on TD. We explore the technical debt types, the TD indicators and estimators, the methods and tools used to indicate, and we quantify TD.
2. We conduct multiple case studies to investigate three types of technical debt, namely code, design and architecture technical debt, to investigate the smells' density over releases, and to estimate the percentage of the eliminated smell.
3. We survey practicing software engineers to (1) find how the software development teams rate the risk of the architecture smells with respect to their roles; (2) identify the reasons that prevent software development teams from refactoring their projects to pay off ATD; (3) explore the benefits that can be gained from refactoring that is targeted to overcome the architecture debt; and (4) examine challenges that might arise during the refactoring process.
4. We use machine learning to prioritize the refactoring decisions based on the level of the severity of the architecture technical debt.
5. We developed a methodology to assess the architecture technical debt risk level by assessing the likelihood of the architecture smells and their impacts on software components using 40 C# project releases.
6. We used a confidant dataset that contains 45 apache java projects to validate the proposed methodology.

Chapter 3

Related Works

This Chapter highlights works related to the technical debt: In Section 3.1, the related technical debt systematic mapping studies are discussed. The work related to the technical debt case studies is presented in Section 3.2. The related works that use the perspective of software developers to investigate technical debt phenomena are explained in Section 3.3. In Section 3.4, the related works that use machine learning to deal with severity of the technical debt are described. The discussion in Section 3.5 is focused on existing works that assess the ATD risk on the software systems.

3.1 Technical Debt Systematic Mapping Studies Related Works

Even though TD is a research area, we found just five mapping studies or literature reviews related to TD. Li et al. [11] published a systematic mapping study that collected 75 TD studies which were published between 1992 and 2013. After they classified and analyzed these studies, they found ten TD types. They are Requirements TD, Architectural TD, Design TD, Code TD, Test TD, Build TD, Documentation TD, Infrastructure TD, Versioning TD, and Defect TD. They determined that Code TD is the most studied TD type in the selected studies, while Versioning TD is the least studied TD type. Most of the selected studies claimed that TD negatively affects software maintainability. They found that TD identification, measurement, prioritization, prevention, monitoring, repayment, representation/documentation, and communication are technical debt management (TDM) activities used to manage TD. TD repayment, identification, and measurement have received the most attention, while TD representation/documentation has received the least attention. Significantly, many studies present approaches to deal with code, design, and architecture technical debt. However, Li et al. [11] did not find any studies from the selected study that proposed a TDM methodology for documentation and versioning TD. Most of the tools take source code as input, except three tools that take NET Assemblies, requirements and solutions, and compiled binaries as input. The TD identification tools have received much attention, but there was not a TD prevention tool. In addition, they found that most TDM tools used the selected studies to deal with code and design TD. The authors suggested that a TDM tool should be integrated into the daily work environment of the software teams. Finally, they found that there is a lack of empirical studies with a high enough evidence level to make TD stakeholders more confident in managing TD by applying various TDM approaches and tools.

Tom et al. [12] gathered the TD studies that were published before 2011 and performed semi-structured interviews in conjunction with a literature review focusing on the dimensions and causes of TD. In addition, the authors concentrated on the benefits and drawbacks of allowing TD. This paper was published in 2013. The authors answered three research questions related to the dimensions of technical debt, the rise of technical debt, and the benefits and drawbacks of technical debt. They found that the Code, Design and Architectural, Environmental, Knowledge Distribution and Documentation, and Testing Debt are five dimensions of technical debt. They also determined that monetary cost, debt amnesty, bankruptcy, interest and principal, leverage, and repayment and

withdrawal are TD attributes. Technical debt is associated with a real monetary cost because it has a negative impact on morale, productivity, and quality. However, it does not need to be paid off in case of a failure of a product or feature, and this is known as debt amnesty. Usually, technical debt is produced for leverage to increase productivity in the short term, but may lead to bankruptcy if refactoring demand is necessary. Technical debt may increase for many of the following reasons: pragmatism, such as being the first to market; prioritization, such as preferring to add new features, even those that reduce the quality; procedures that are adopted by a team such as poor communication and collaboration processes; attitudes such as the fear of introducing new bugs; and ignorance and oversight, such as being unaware of the mistakes that caused the technical debt. Finally, this study investigated the consequences of technical debt. The study's authors found that technical debt has risks that impact software productivity, quality, and the developers' morale.

In 2015, a related systematic literature review was conducted by Ampatzoglou et al. [13]. The authors focused on the financial aspects of TD. The authors' goals were (1) to present the technical debt financial glossaries and definitions; and (2) to classify economic techniques used to manage technical debt. They introduced a glossary of technical debt financial terminology and derived a definition for each term. Finally, this systematic literature review provided a financial approach classification schema that consists of three levels. The first level presents TDM categories, the second level represents the financial techniques, and the third level describes the tools, methods, and technologies used by software engineering. This schema provided details and the evolution of each TD technique, term, method, and tool. They found that principal and interest are the most common financial terms used in TD research. They also found that real options, portfolio management, cost/benefit analysis, and value-based analysis are the most frequently applied financial strategies.

In 2016, Alves et al. [14] conducted a systematic mapping study to investigate strategies proposed to identify and manage TD in the software lifecycle. The authors collected 100 studies that were published between 2010 and 2014. Their goals were to (1) illustrate technical debt types; (2) identify technical debt indicators; (3) identify technical debt management strategies; (4) understand the maturity level of each strategy; and (5) investigate how visualization techniques have been proposed to identify and manage technical debt. The authors found fifteen types of TD. They are Design TD, Architectural TD, Documentation TD, Test TD, Code TD, Defect TD,

Requirements TD, Infrastructure TD, People TD, Test automation TD, Process TD, Build TD, Service TD, Usability TD, and Versioning TD. They found Code Smells is the most cited and analyzed TD indicator, and God Class is the most investigated type of Code Smells. On the other hand, indicators for Process TD, Infrastructure TD, People TD, and Usability TD were not identified by the selected studies. Using the source code was the most utilized methodology to identify TD. Studies that were viewed proposed visualization techniques. Finally, in this study, the authors declared the necessity of proposing new techniques and tools that could support developers with the control of TD.

In 2017, by concentrating on the Architecture Technical Debt (ATD) regarding principal, interest, and management, Besker et al [15] gathered TD studies published before December 2015. The authors developed a novel ATD model for managing and raising awareness about ATD. The model explores different aspects and relationships to illustrate ATD in a unified and comprehensive way. They found that ATD has a negative impact on the overall software lifecycle. The study grouped ATD into five categories which are; dependency violation, non-uniform usage of pattern and policies, code-related issues, interdependent resources, and absence of addressing nonfunctional requirements. They found that ATD could negatively affect software maintenance and evaluability, performance, reliability, flexibility, and system growth. Regarding the management of ATD, they stated that there is a lack of guidelines on how to manage ATD successfully in practice, and showed the need for methods to monitor and evaluate ATD. They indicated that there are benefits to performing a partial refactoring, and stated that overlooking refactoring could lead to a development crisis in the long term.

There were no systematic mappings focused on studying TD in the empirical studies between 2014 and 2017. Consequently, we collected 43 empirical studies from four databases to investigate and analyze them. The purpose of our systematic mapping study (in Chapter 4) is to explore and understand TD by identifying and analyzing the empirical studies published from January 2014 to December 2017 in TD. We plan to conduct a systematic mapping study of TD by thoroughly reviewing published TD empirical studies. Based on our research goal, we formulated four research questions. The procedure and results of the systematic mapping study are explained in detail in Chapter 4.

3.2 Technical Debt Case Studies Related Works

Various empirical studies have been conducted to detect and study smells in software systems. The studies apply different techniques and use various tools to identify smells in software projects. Marinescu [16] conducted a case study to measure the performance of detection strategies. Various detected code smells were examined by humans; they found that the success rate of the automatic detection strategies was around 70%. In another study, the life cycle of God Classes was observed in two systems through their releases to discover when they were introduced, removed or modified, and what their prevalence was [17]. The objective of the study was to understand whether God Classes affected software systems for long periods, or they were refactored through the releases. They concluded that God Classes that are not modified very much do not cause problems.

However, Olbrich et al. [18] used Marinescu's detection strategies to find the correlation between code and the growth of the maintainability effort. They found God Classes needed more of a maintenance effort than other classes. Li and Shatnawi [19] conducted an empirical study to investigate the association between bad smells and class error probability. They found a correlation between code smells and the number of defects in three releases of the Eclipse project. For comparison, Zhang et al. [20] investigated the relationship between software faults and code smells. Their study suggested that Duplicated Code should be prioritized for refactoring because source code that has Duplicated Code is likely to be related with more faults than source code which contains the other five code smells. Khomh et al. [21] investigated the relationship between the presence of code smells and software change proneness. They found that classes that have code smells tend to be significantly more change prone than other classes.

Finally, Sharma (one of the Designite tool developers) and his research team [22] conducted an empirical study to investigate the fundamental characteristics of code and design smells that frequently occurred in Open source. They found that magic number is the most frequently occurring code smell, while unutilized abstraction is the most frequently occurring design smell in C#, respectively. A related study investigated six types of smells in reference [19], [20], 29 types of smells in reference [21], 19 types of smells in reference [22] and, nine types of smells in reference [23].

In conducting comparisons with the current studies, the objective of this chapter is to analyze 35 types of smells, as well as to include the C# programming language which has been given less

attention in related work compared with other programming languages like Java. We will fill these gaps by conducting a case study to analyze TD in five OSSPs with respect to the average smells density for each type of smell in an OSSP, smells increment over multiple releases, and the percentage of smells density (SD) that is removed by refactoring. The smell density is defined as the average of the number of smells found per one thousand lines of code (KLOC).

In our empirical study, we used code, design, and architecture smells to identify technical debt in software. Our goal is to use smells as indicators to analyze TD in OSSPs for the purpose of understanding how TD is affected with respect to the average smells density for each type of smell in an OSSP. We will also investigate smells increment over multiple releases and the percentage of smells density removed by refactoring. The case study is presented in detail in Chapter 5.

3.3 Architecture Technical Debt Surveys Related Works

Unlike other types of technical debt, we noticed that few empirical studies have focused on practitioners' perception of architecture technical debt and architecture smells. Tian et al. [24] performed an empirical study from the perspective of the developers that aims to find the level of practical knowledge regarding the descriptions of architecture smells, their causes, and their impacts on software qualities, and how developers detect and refactor architecture smells. The authors found that (1) Developers use general terms to describe ASs. (2) Violating architecture patterns, violating design principles, or misusing architecture anti-patterns are the mean reasons of architecture smells. (3) Developers tend to employ static code analysis tools to cover the absence of tools for detecting and refactoring ASs. (4) The main concerns of the developers are the maintainability and performance of the systems affected by ASs. (5) There are a few approaches and tools for quantifying the cost of ASs. In contrast to our work, the authors studied the architecture smells just from the perspective of the developers to reach aims that are different than ours.

An empirical study at Microsoft was introduced by Kim et al. [25]. The researchers held a survey and semi-structured interviews with professional software engineers to identify refactoring benefits and challenges. According to this study, 75.41% of the participants cited regression bugs and built breaks as a critical risk factor associated with refactoring. Many developers reported an improvement of readability and maintainability in their code after refactoring. Jain et al.[26] presented an explanatory survey to investigate refactoring performed by IT professionals. The authors aimed to reveal facts regarding; refactoring risks, benefits, limitations of tools, and how a team manages consistency between different artifacts while practicing refactoring. Availability, usability, and trust issues are limiting developers' ability to use refactoring tools. More than 80% of the participants indicated maintainability and readability improvements as benefits of refactoring. Around 72% of the participants found refactoring time-consuming, and more than half of them stated merge conflict as a critical risk factor associated with refactoring. In contrast to our work, the authors only considered refactoring benefits and challenges.

Martini et al.[27] carried out an empirical study using questionnaires, interviews, and thorough inspection of the code with the practitioners. The authors used architecture smells to detect ATD and evaluated the undesirable impact of the ATD, refactoring difficulties, and the usefulness of

the Arcan tool. Their results show that practitioners valued the help of the Arcan tool. Besides, despite its higher refactoring effort, the participants prioritized refactoring ATD associated with negative impacts on the software system. Roveda et al. [28] proposed an Architecture Debt Index (ADI) to evaluate the internal quality of software systems based on the history of their architectural smells, their severity, and architecture design metrics. In contrast to our work, these two papers have only considered dependency architecture smells.

Emerson et al. [29] performed an empirical study to explore the usage of refactoring tools. They found that professional programmers underused refactoring tools. They do not support frequent bursts of refactoring interleaved with other programming activities. An empirical study was conducted by Sharma et al. [30] to target software architects to discover the specific problems architects and their teams face when using refactoring tools. Sharma's results show that many development teams are not sufficiently aware of either the benefits of refactoring or the relevant refactoring tools. As compared with our work, Sharma has addressed refactoring tools and challenges while Emerson has only considered refactoring tools.

Unlike the stated related works, we studied seven types of architecture smells from the perspective of developers, architects, and project managers to find how the software development teams rated the risk of the architecture smells. Also, we aim to identify the reasons that prevent software development teams from refactoring their projects to reduce ATD and explore the benefits and challenges of refactoring. The empirical study is detailed in chapter 6.

3.4 Architecture Technical Debt Machine Learning Related Works

In the literature, we found around eight works-related to exploiting machine learning techniques. Five of them used machine learning algorithms to detect and predict the code smells [31]–[35]. One related work used machine learning techniques to detect code smell and classify the smell severity [36]. Another related work used machine learning techniques to estimate smells severity [37]. The deep learning method is a machine learning technique that can be used to detect code smells. Sharma et al. [38] is another related work that used Convolution Neural Networks and Recurrent Neural Networks to train smell detection models. They prove that deep learning techniques can be used to detect code smells.

Maneerat et al.[35] used machine learning to predict bad-smells from the software design model. They collected seven data sets which offer twenty-seven design model metrics and seven bad-smells. Seven different machine learning algorithms were used to predict bad-smells. The study results showed that bad-smells could be predicted from the software design model. However, some algorithms were not suitable for bad-smell prediction. Naive Bayes, VFI and J48 had prediction average rate of less than 90% while Logistic regression, IBI, IBk and Random Forest achieved more than a 90% average prediction rate. They found that the machine learning algorithm selection should be considered based on the type of bad-smell because there is no machine learning algorithm can accurately able to predict all bad smells types.

Moha et al.[39] introduced a DETEX method to detect code, and design smells. DETEX uses domain-specific language (DSL) to detect smells. In DETEX, domain experts, engineers, and quality experts can specify and modify manually the detection rules using high-level abstractions. To detect anti-patterns, Magia et al. [34] introduced a novel approach; named SVMDetect. The approach is based on a machine learning technique. Using the same three java projects (ArgoUML v0.19.8, Azureus v2.3.0.6, and Xerces v2.7.0) and the four anti-patterns (Blob, Functional Decomposition, Spaghetti code, and Swiss Army Knife) that studied by Moha. Magia showed that the accuracy of SVMDetect is greater than of DETEX where SVMDetect was able to find 143 anti-patterns while DETEX found 102.

Fontana et al.[32] used 76 systems from 111 systems written in Java that were collected by Azadi et al.[40] and named the Qualitas Corpus of systems. The authors used the 20120401r version, and as independent variables to their machine learning approach, they selected metrics at

class and method level. iPlasma, PMD, Anti-Pattern Scanner, Marinescue rule and the Fluid Tool were used to detect the rules to identify God Class, Data Class, Long Method and Feature Envy smell. They classified smell severity into 4 levels: No smell, Non-severe smell, Smell, and Severe smell based on the value of the size, complexity, and coupling. Six classifiers which are SMO, LibSVM, J48, Random Forest, Naive Bayes, and JRip were applied. Four classifier that had accuracy greater than 90% for all datasets were named J48, Random Forest, JRip and SMO. In addition, on average, those four classifier have the best performances. Compared with the other classifiers, Naive Bayes has slightly lower performance on Data Class and Feature Envy than on God Class and Long Method. Finally, this study provided a large dataset containing source code, metrics, detection, and smell severity.

Palomba et al. [37] built a bug prediction model that defended using code smell intensity by Fontana et al. [41]. The intensity index is defined in the range from 1 to 10. Code smell detection strategy, metric thresholds, the statistical distribution of the metric values, and the actual values of the metrics were used to estimate the severity index (intensity index). They studied six kinds of code smells which were God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, and Message Chains. Six large Apache Java systems were used in this study, named, Apache Xerces 1.4.4, Apache Xalan 2.7, Apache Velocity 1.6.1, Apache Tomcat 6.2, Apache Lucene 2.4, and Apache Log4j 1.2. Their results indicated that adding code smell intensity as predictor increased bug prediction model accuracy. Even after comparing the accuracy of the intensity index prediction model against other structural metrics in the model including the ones used to compute the code smell intensity, they observed that the intensity index is more important for predicting the buggy components than the other used metrics used.

Fontana et al [33] applied 16 machine learning algorithms (JRip, Naïve Bayes, Random Forest, J48 Reduced Error Pruning, J48 Pruned, J48 Unpruned, SMO Poly Kernel, SMO RBF Kernel, LibSVM C-SVC Linear Kernel, LibSVM v-SVC Linear Kernel, LibSVM C-SVC Radial Kernel, LibSVM C-SVC Polynomial Kernel, LibSVM C-SVC Sigmoid Kernel, LibSVM v-SVC Polynomial Kernel, LibSVM v-SVC Radial Kernel, and LibSVM v-SVC Sigmoid Kernel) to detect Data Class, Large Class, Feature Envy, and Long Method code smell. 74 software systems were used. They found that J48 and Random Forest have the highest performances, while support vector machines have the worst performance.

In this paper, Fontana et al [36] used several machine learning techniques to classify the severity of the code smell. The severity of code smells as a factor helps prioritize refactoring efforts whereas high severity code smells affects negatively the maintainability of software a system. To evaluate the performance of the ordinal classification for the classifiers, the model accuracy and different performance measures were used. Despite that the accuracy of the severity classification is not as high as in the binary classification of presence or absence of code smells a strong correlation (0.88–0.96) between the actual and predicted severity was found using the Spearman correlation.

Di Nucci et al. [31] replicated Fontana et al. [33] after modifying the dataset to reduce the differences in the metrics distribution, (ii) balance the number of smell instances, and have different types of smells in the same dataset to be more realistic. They merged the God Class and Data Class for class-level code smells in one dataset and the Feature Envy and Long Method for method level ones in another dataset. After that they duplicated the datasets so they can have 4 datasets. On average, their model's accuracy was 76% while it was 96% in Fontana et al. [33]. The reason behind the differences is that in the Di Nucci et al. [31] study the number of non-smelly instances were 83% in each dataset while it was 67% in the Fontana et al. [33] study. That means the selected dataset was the reason behind the high performance reached by Fontana et al. [33] and not the real capabilities of the classifiers. Both studies found that J48 and Random Forest have higher performances compared with other classifiers. However, Di Nucci et al. [31] suggest conducting more research regarding using machine learning models to detect smells because detecting smells using machine learning is not solved.

We aim to fill gaps related to the type of TD. From the above, we can see that there have been no related works that use machine learning to indicate and classify the severity of the ATD in software components. In our approach, we focus our attention on four types of internal structure metrics that can indicate ATD. We consider 4 projects for analysis; to build ML models and select the model that has the highest performance rate for detecting and classifying ATD severity while prioritizing the components. Our machine learning models are explained in detail in Chapter 7.

3.5 Architecture Technical Debt Risk Related Works

Martini et al. [27], [42] conducted a case study in a large company. They compared a component before and after it was refactored to achieve modularity. To identify and estimate ATD in the form of non-modularized components, they developed a semi-automated holistic framework. They evaluated their technique by comparing the differences in maintenance and development costs between the refactored and non-refactored components. To quantify the changes, they measured the committed lines of code (LOC) for any change, but as we know, many changes are made for fixing bugs or adding new features in the system. These changes must not be included in estimating the refactoring effort. They did not use several software systems and compare them.

Martini et al. [27] conducted a survey to identify and prioritize the ATD. They investigated three types of architecture smells, which are Cyclic Dependencies, Unstable Dependency, and Hub-Like Dependencies. The authors used “Arcan tools” to detect the tools and calculate the severity index. Arcan measures the severity as the number of vertices involved in the cycles and the probability of every edge which forms the cycle.

Kazman et al. [43] conducted a case study to identify and quantify ATD in one system called SoftServe. They clustered the files in regard to the ATD. Afterward, the authors visualized the architectural flaws which were confirmed by architects, among these files, to find the propagated errors. They extracted data from the development process to quantify the ATD penalty and estimated the potential benefits of refactoring. The authors used three ASs: Unstable Interface, Implicit Cross-module Dependency, and Unhealthy Inheritance Hierarchy, to identify the ATD in SoftServe. The authors were not able to collect true effort data. They used (1) the number of resolved defects per file, (2) the number of completed changes per file, and (3) the number of modified/added/deleted LOC per file to fix defects and make changes to estimate the effort of the refactoring.

Nord et al. [44] developed a metric for managing ATD. This metric can be used to optimize the cost of development by implementing new components in for instance, the current release and the rework of a previous release. By evaluating and comparing the cumulative values of the two paths (current release and the rework on the previous release), the larger value is the one that suffers from more ATD. The authors stated, “We use software architecture as a means to identify

and monitor architecture technical debt.” Nord et al. did not state which ATD can be identified. Neither did they explain how to find the effected components.

Li et al. [6] proposed an ATD identification approach based on architecture decisions and change scenarios. To identify the ATD, the authors assumed that prior to identifying ATD in a software system using this approach, architects recorded information related to the architecture decisions such as (1) why the decision was chosen over alternatives, (2) the decision’s advantages, (3) the decision’s disadvantages, and (4) a diagram that illustrates the affected part. After that, the changing scenario needs to be established similarly. This approach needs to document everything related to the ADs. Documentation like this is believed to be hard to find in many projects or companies, especially if the projects were developed a long time ago. This approach expects all developers to be familiar with architecture decisions and ATD, which is not true.

Öztürk et al. [45] proposed an approach and a toolset for predicting the refactoring effort needed to isolate the software architecture causing the problem. First, the authors applied a Module Dependency Analyzer tool to identify module inter-dependencies in the source code. Second, they built a graph to represent all the identified module’s inter-dependencies. Third, from the graph database, they created and evaluated various decomposition alternatives. In terms of LOC, they calculated the effort needed for every decomposition alternative. The authors identified; every redirected interface, the parameters complexity, the returned value, and nesting level of classes. Öztürk et al. measured the effort by a summed-up number of LOC for a redirected interface, the parameters complexity, the returned value, and nesting level of classes.

Guo et al. [46] proposed a new approach to managing TD to help software managers make informed decisions. They followed a risk management method to manage TD. The approach consists of three steps, which are TD identification, TD Measurement, and TD Monitoring. The authors did not mention which TD type will be identified. The authors estimated the principal, interest amount, and interest probability for every TD item on the list. They assumed that some quantification had been done with the list of the TD items. They assumed the initial estimate for the principal and interest is high, medium, or low. After that, they conducted a case study to apply their approach, and the project team used their own experience to carry it out. After that, the authors recorded copious information about the decisions themselves. The developers who trained for just

40 minutes identified the risk. The project leader measured the impact “Severity” of the TD by using his/her experience. This study focuses on the costs of managing technical debt.

Xiao et al. [47] defined ATD as a group of architecturally connected files and the maintenance cost growth for those files. The authors estimated the probability of changing a file when another file was changed by using a history coupling probability matrix (HCP). After that, they indexed file groups using dependency and coupling files’ histories. They quantified the maintenance costs using the number of modified LOC to fix the bugs for the files involved in the ATD. They estimated the ATD interest using four regression models. Finally, they ranked ATD according to accumulated maintenance costs. By conducting a case study, they evaluated their approach using seven open-source projects. Xiao et al. estimated the maintenance costs based on the number of modified LOC to fix the bugs, which is believed to be related to defect debt more than ATD.

In our research, we will estimate the ATD risk level. The ATD probability and the ATD severity will be used to estimate the level of the ATD risk. The methodology is explained in detail in Chapter 8, while Chapter 9 verified the proposed method.

Chapter 4

Investigate, Identify and Estimate the Technical Debt: A systematic mapping study

In this chapter, a systematic mapping study is conducted to get an overview of the research on TD. This chapter is based largely upon a paper published by M. BenIdris, H. Ammar, and D. Dzielski, in International Journal of Software Engineering & Applications (IJSEA) [48]. The main motivation for this chapter is to understand the phenomena and find how to identify and estimate the technical debt.

Chapter four is organized as follows. Introduction to the technical debt is explicated in Section 4.1 while Section 4.2 stated our goals and research questions. The execution of the study is detailed in Section 4.3. Section 4.4 and 4.5 showed the study metadata and the publications classification, respectively. In section 4.6, we answered our research questions. Threats to the validity of this study were discussed in Section 4.7. Finally, section 4.8 concluded the study.

4.1 Introduction

For the first time, the concept of the technical debt (TD) was introduced by Ward Cunningham in 1992. Ward Cunningham said, “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” [1]. Usually, while trying to keep the quality up to standard, software developers face the challenge of delivering a software system under tight schedules [46]. Technical debt can occur if developers take shortcuts and focus only on functional requirements and ignore some non-functional requirements. This shortcut taken by a decision reflects the technical debt metaphor, which may affect the software in the long term. Many TD types can occur during the software lifecycle [49]. It is essential to know the reason that leads to that particular TD type and its indicator [50] to identify it. Before we manage the TD, the first task is to identify it in the software [46]. In many empirical studies, the source code can be used to detect the TD. However, the code comments also can help developers to understand the code source [51], [52] and find the technical debt stated by the developers, which is called Self-Admitted Technical Debt (SATD) [50], [53], [54].

The principal of TD (cost of fixing the debt) and the interest of TD (the future cost if the debt is not fixed) are two terms used to assess TD. The principal is considered as a function of three variables: (1) the number of the problems needed to solve, (2) the time, and (3) the cost required to fix each problem [18]. The effort/cost that is needed to eliminate the debt from a given system known as the TD principal [14].

In this study, we selected 43 empirical studies. Even so many tools are available to use to detect technical debt, some researchers developed tools to identify TD as well as develop their own approaches, methods to evaluate and manage TD [55]–[59].

4.2 Goals and Research Questions

The purpose of this work is to understand technical debt by conducting a systematic mapping study to analyze the empirical studies published from January 2014 to December 2017 in technical debt. We formulated four research questions based on our research goal.

RQ1: What are the types of technical debt in the selected empirical studies?

RQ2: How did researchers investigate the technical debt in the selected empirical studies?

RQ3: What are the methods used to identify and assess the technical debt in the selected empirical studies?

RQ4: What are the tools used in the selected empirical studies, and which tools are the most commonly used by researchers?

4.3 Systematic Mapping Implementation

To get a general idea of the research on technical debt, a systematic mapping study is carried out. In this study, to search for publications, six steps are performed.

1. We determined the scope of this study. As shown in Figure 4.1, exactly four electronic Databases: Springer Link, Science Direct, ACM Digital Library, and IEEE Xplore were used, and the period: from January 2014 to December 2017 is our study scope.
2. Technical debt as a search string was used in the abstract to find the publication that related to the technical debt.
3. An exclusion criterion which is shown in Table 4.1, was applied.

Table 4.1 the exclusion criteria

#	The exclusion Criteria
1	Publication without unavailable full text such as abstracts
2	Not English publication
3	Panel summary publication

4. The results from the four databases were merged, and the duplicates' studies were removed.
5. The publications were filtered by reading the abstract, and any unrelated paper or any study that dependent on just the developer or student, such as surveys was excluded
6. Forty-three empirical studies, which are shown in Appendix A, were selected after reading the full text for all publications.

Figure 4.1 Systematic mapping study method

4.4 Empirical Studies Metadata

As shown in Figure 4.2, IEEE Xplore DB has the majority of the chosen empirical studies. One reason is that many of the publications duplicated between ACM and IEEE Xplore. We classified them with IEEE Xplore instead of ACM. Another reason, we also noticed that the International Workshop on Managing Technical Debt (MTD) workshop, which was initiated in 2010, raised the number of publications and increased the difference between IEEE Xplore and the other three DBs in the publication number.

Figure 4.2 Number of case studies per database

Figure 4.3 presented the total number of selected empirical studies, which have nearly doubled from 2014 to 2016. While Figure 4.4 presents the citations of the selected papers. For instance, study number 43 has 55 citations in four years, while study number 17 has 44 citations in three years. Now in September 2020, S43 has 107 citations in 6 years, while S17 has 100 citations in five years.

Figure 4.3 Case studies and publication year

Figure 4.4 Case studies and number of citations

4.5 Classification Scheme

For a better understanding of the technical debt phenomena and helping us to answer our four research questions, we will have five categories of classification: technical debt types, researchers' investigation, technical debt indicators, technical debt evaluators, and used Tools.

4.5.1 Classification by type of Technical Debt

A large number of TD types were studied in the selected empirical studies.

- *Architecture Technical Debt* refers to a problem on the software architecture decisions such as violation of modularity [60].
- *Code Debt* refers to poorly written code such as brain method [60] or code duplication [61].
- *Documentation Technical Debt* refers to a problem on the software documentation such as incomplete documentation [11].
- *Versioning Technical Debt* refers to a problem in source code versioning such as unnecessary code forks [11].
- *Requirements Technical Debt* refers to the distance between the optimal requirements specification and the actual system implementation under domain assumptions and constraints such as implemented requirements that do not cover all cases [49].
- *Build TD* refers to issues that make building the system harder and unnecessarily time-consuming such as manual building process [62].
- *Defect TD* refers to defects, bugs, or failures that should be fixed but instead have to be deferred to a later time for some reason such as deferred bugs because of the competing priorities [14].
- *Design Debt* refers to violations of the principles of good object-oriented design such as a complex class [11], [49].
- *Test Debt* refers to issues found in testing activities that can affect the quality of those activities such as planned tests that were not run [49].
- *Infrastructure TD* refers to infrastructure issues that can delay or hinder some development activities such as use of an old technology [62].
- *Database Debt* refers to flaws in software databases such as at able without primary key[63] or foreign key [64].
- *Performance Debt* refers to flaws in a software system performance that make the software performance is poor such as memory bottleneck [65].

- *Usability Debt* refers to inappropriate usability decisions that will need to be adjusted later such as the lack of a common user interface template [65].
- *Service debt* refers to selecting unsuitable web services that lead to a mismatch of the service features and applications' requirements such as the service capacity is underutilized and the cost of maintenance outweighs the revenue streamed from using the service [66].
- *People debt* refers to personnel issues that can delay or hinder some development activities such as expertise concentrated in too few people, as an effect of delayed training and/or hiring [54].
- *Process debt* refers to inefficient processes such as what the process was designed to handle may be no longer appropriate [14], [67].

As shown in Figure 4.5, Code Debt, Design Debt, and Architecture Debt were the most frequent types in these studies. On the other hand, some TD types such as Version debt, Service, Process, and People debt were the least frequent types in the selected studies. Some papers, S4, S5, S6, S8, S10, S11, S12, S13, S14, S17, S18, S19, S23, S24, S26, S29, S30, S31, S32, S33, S35, S37, S38, S39, and S41 studied one type of TD, whereas other papers studied more than one type. S1, S21, S22, S28, and S34 studied two technical debt types of TD. S40 and S43 studied three types, while S3 studied four types of TD. S20 and S42 investigated five types of TD, while S2 and S16 studied six types. Three papers, S7, S8, and S9, studied 7, 8, and 12 TD types, respectively.

TD Type: Paper Reference as indexed

Figure 4.5 Most studied technical debt types

4.5.2 Classification by researchers' investigation

Figure 4.6 shows the methods that the researchers used to investigate the technical debt in these selected case studies. We found that some authors explored how much self-admitted technical debt is removed and who removed it, the relationship between self-admitted technical debt and software quality, the relationship between the quality model (QMOOD), and the different technical debt tools, and the correlation between software architecture and maintenance cost. Additionally, they study the software history.

In detail, S2 investigated the relation between TD Principal and developers. S2 explores the relationship between the diffusion and evaluation of SATD with software quality. S10 explored the relationship between some project characteristics and amount of TD. The relation between metrics and monetized TD assessments have been investigated by S18. S26 investigated the relation between changes metrics and types of refactoring. S31, S38, and S43 investigated the interest of the technical debt. S31 investigated the TD interest to find if it has correlation with developers' activities, changes metrics, and code smells. The relation between the amount of technical debt and the interest that has to be paid during corrective maintenance has been investigated in S38. S43 investigated the correlation between TD that identified by analyzing the source code and the interest payments in the form of increased defect- and change-proneness. S34 and S36 investigated the software defects where S34 investigated the relationship between the defects and SA rules while S36 investigated the correlation between SATD and defects on the files to examine the relationship between SATD and software quality. Finally, S39 investigated the relationship between system architecture and maintenance costs.

Another study, S3, tried to map or find the link between debt and quality while another study. S19 linked code metrics to the software product quality characteristics of reliability and maintainability and to the most commonly identified sources of TD. However, many papers investigated the ability to identify technical debt in software. S9, S20, S27, and S42 investigated the ability to identifying the technical debt from the code comments. Other papers investigated the ability of the metrics to identify TD, such as S5 and S30. S5 used software modularity metrics as a substitute for modified components per commit (ANMCC) to indicate ATD. While S30 proposed a technique based on eight software metrics, namely Afferent Couplings, Efferent Couplings,

Depth of Inheritance Tree, Lack of Cohesion in Methods, Number of Children, Cyclomatic Complexity, and Code Duplication.

Figure 4.6 Investigation methodology

Documentation related measures can be used to identify TD. S14 proposed Intensity Index to determine the most critical TD instances while S29 introduced a software design quality assessment tool, Designite, to detect design smell. S21, S22, S27, S42 investigated the ability of machine learning to identify TD. Other researchers investigated the ability to assess or predict the principal or the interest of the technical debt in software. The authors in S6 used KNN-regression to predict the principal (standard time for bug fixing time). In S8, the authors proposed an approach for estimating the breaking point while S23 developed a method to assess the severity of the interest. S11 and S24 used the software quality to assess the TD. S11 assessed the software quality of a DB schema through static analysis. They defined 11 rules that are able to identify database debt in the database, while in S35, the authors explored the ability of using the foreign keys as an indicator to identify database TD. S24's proposed approach was to assess the ideal quality of software based on the benchmarking concept. S31 presented a framework to assess the interest while S13 implemented a what-if scenario analyzing the estimated number of defective files in the absence of smells. Based on experts' knowledge and baseline project data, S40 created a quality

and prediction model; ProDebt, that measured the productivity to estimate the saved effort. Finally, as we mentioned above, S18 investigates the relation between metric-based and monetized TD assessments. In addition to that, the authors in S18 built a model based on metrics that can predict the TD principal.

Studies such as S4, S7, S21, and S33 answered important questions that were addressed by the studies' authors. In S4, the authors estimated how much technical debt is incurred by code generators and tested if TD affects the maintenance of the Model-Driven Engineering (MDE) projects. The expected lifetime of components that have TD is an important question that is investigated by S7. S21 studied the removal of self-admitted technical debt by examining the following questions: How much SATD gets removed? Who removes SATD? Is it most probable to be self-removed or eliminated by others? How long does SATD survive in a project? What activities lead to remove of SATD? However, when code smells start manifesting themselves and whether this happens suddenly or gradually is an important question that addressed by the S33's authors. Managing TD is investigated by S12, S16, S17, and S25. The authors in S12 introduced an approach for visualizing TD. S16 proposed a TD formulation to decide when-to-release decisions while S17 built economic models of the before and predicted after states. Finally, S25, by using Scrum, describes a practical evaluation of the Seaman and Guo's TD management framework. The authors in S15 evaluated three methods that are able to estimate technical debt principal. The evaluation compares each technique against an external quality model. Evaluation of the software system, Apache Ecosystem, by using the tool, SonarQube, was conducted by S41's authors. By focusing on the maintainability aspect of the configuration code quality, S28 proposed a catalog for implementation and design configuration smells. Finally, by integration testing and extensive unit testing, S32 applied an approach to reduce and avoid TD.

4.5.3 Classification by Technical Debt Estimator

Figure 4.7 and Table 4.2 show Technical Debt Elements and the Technical Debt Estimator, respectively. We found five Technical Debt Elements, which are Principal, Interest, Interest Probability, Probability of Refactoring, and TD lifespan.

Figure 4.7 Technical Debt Elements

As shown in Table 4.2, to assess the TD principal, S8 and S18 used the software quality as an estimator. S8 used the distance of an actual object-oriented design to the corresponding optimum one to assess the breaking point, which equals to the principal over the average interest. S18 assesses the TD principal using seven structural metrics that quantify different aspects of quality (i.e., coupling, cohesion, complexity, size, and inheritance). The time needed to fix the violation is used to estimate the principal in S1, S11, S24, S38, and S41. Besides estimating the principal, In S38, the authors assessed the interest (corrective maintenance effort) and the interest probability (corrective maintenance frequency).

Other studies used commits to estimate the principal. S2 used code commits to estimate number of SATD instances introduced, removed, and left KLOC from commits. From code commits, the authors in S4 classified source files into three groups: (1) generated files, (2) modified generated files, and (3) developer-created files. They extracted the total number of files of every group and # LOC of generated code, of modified generated files, and of developer-created files. Also, they extracted the Proportion of Generated Files and Generated Code that need maintenance to estimate the principal. From historical data, the authors in S6 and S21 extracted the time to fix bugs and the time to remove SATD, respectively.

Table 4.2 TD Estimators

Paper	Elements	TD estimator	Method
S1	Principal	Maintainability	LOC modified during each commit
S2		Effort	LOC
S4		Effort	LOC
S6		Maintainability	Machine learning (amount of time to fix bug)
S8		Quality	Distance between an actual and optimum design
S10		Effort	LOC
S11		violations	Time to fix violations
S13		Maintainability	Machine learning (16 change metrics)
S16		Effort	LOC
S17		Maintainability	LOC to fix defects and make changes (project data)
S18		Quality	Seven structural metrics that quantify different aspects of quality
S24		violations	Time to fix violations
S25		Effort/Quality	Classify the effort to levels based on # of hours/Classify the complexity to points 10 points
S26		Quality	Architecture complexity
S30		Quality/Effort	Decrease the coupling between the two layers/persons per hour
S31		Effort	Time
S33		Quality	The growth of LOC,WMC,RFC,CBO,LOCM,NOA,NOM
S34		Personnel	Experience
S39		Maintainability	Defect-related activity (\$/LOC).
S40		Effort	Machine learning (Real Effort data which recorded per day and per feature)
S12	Interest	Quality	Impact of misplaced logical components
S23		Quality	Growth of the code complexity and it's impacts such as impact on development speed
S25		Effort	Classify the effort to Levels based on # of hours
S25	Interest probability	Time	Classify in 3 levels. Happened >1, Happened =1 and Happened =0
S38		Maintainability	Corrective maintenance frequency
S26	Probability of Refactoring	Maintainability	Fault Repairing Modification, Feature Introduction Modification, and General Maintenance Modification
S7	TD lifespan	Personnel	Experience

In S31, to estimate the TD principal, the author extracted the developer activity logs. They extracted activity metrics which were (1) the percentage of the time spent in a file over whole logged time, (2) the percentage of the time spent editing files over whole logged time, (3) the percentage of the time spent in a file overall sessions, (4) the percentage of the time spent editing files overall sessions, (5) Ratio of number references to file over number references in all sessions and the total number of sessions. In S33, from the commits, the authors took the days needed to fix a code smell in their consideration to estimate the TD principal. In S40, the authors measured the productivity (P0) to the current quality and productivity (Px) to improve quality x%. After that, they estimated the TD principal = $P0 (1 - P0/Px)$.

Defect resolution and maintenance efforts were used as TD estimator by S13, S17, S26, and S39. In S13, the authors are assessing the principal as a decreasing number of defective classes to avoid smells. In S17, the principal is assessed as a number of LOC to resolve defects, a number of LOC to complete changes, and a number of mod/add/del LOC to fix defects and make changes. S39 assessed the TD principal from the maintenance efforts by capturing the level of defect-related activity (\$/LOC). Lastly, in S26, the probability of the refactoring is estimated using the Fault Repairing Modification, Feature Introduction Modification, and General Maintenance Modification. In S12, the authors assess the ATD interest by assessing the impact (Misplaced Logical Components) of ATD. In S19, they measured the architectural complexity based on five metrics, namely Lines of Code, Weighted Method Count - Unweighted, Weighted Method Count – McCabe, Response for Class, and Coupling Between Objects to identify TD. Each of these metrics was defined based on existing calculations performed by the Understand tool. Some studies assessed the TD based on professional opinion. In S7, the lifespan of the TD and the importance of quality attributes (A rating, in a scale from one (lowest) to five (highest), of the importance of the following QAs: Functional Suitability, Reliability, Performance, Usability, Security, Compatibility, Maintainability, and Portability.) throughout TD is answered by practitioners without the use of any documentation. In S34, the TD principal, the reduction in effort, is computed based on an opinion from project teams. In S10, linear regression was used to estimate the TD debt where $\text{debt} = \text{Time to write all source code} / (\#LOC * 30) * 100$. In S16, the authors estimated the effort of the refactoring with the help of the COCOMO II model (To assess the Effort Adjustment Factor (EAF)) and the ordered weighted average of all criteria that determine the TD (TDAF). $\text{Effort} = (\text{KLOC})^{\text{TDAF}} * \text{EAF}$.

In S23, the authors assessed the TD by using a set of factors with properties. Growth of interest is determined by propagation factors (internal: “Growth of the source and complexity” and external: “number of ATD-related increments planned and number of external users in the ecosystem”) and/or other impacts (Impact on development speed (# of increments), on maintainability, on quality on learning, on revenues, and other costs that depend on the organization). S25’s authors conducted an empirical study on two companies to assess the interest, principal, and interest probability. The First company classified the interest and principal to high “>8 hours”, medium “between 4 and 8 hours” or low “<4 hours”. And the Interest Probability to high “80%”, medium “50%”, or low “20%”. On the other hand, the second company defined complexity points (From 1 to 10) as an estimation of the effort of refactoring. Where $\text{Effort} = \text{Complexity point} * 8 \text{ (hours)}$. For instance, if the refactoring code is classified as complexity point = 4 that means the $\text{Effort} = 4 * 8 = 32 \text{ hours}$. Finally, S6, S13, and S40 used machine learning techniques to assess technical debt.

4.5.4 Classification by Technical Debt Indicator

As presented in Figure 4.8, the most used indicator to identify technical debt is Smells, while the second most commonly used one is Code Comments and Defect/Bug. Smells were used as a TD indicator by many papers. To indicate technical debt, S4 used ten types of bad smell, which are God Class, Excessive Class Length, Excessive Method Length, Excessive Parameter List, Duplicate Code, Cyclomatic Complexity, Coupling Between Objects, Excessive Imports, Too Many Methods, and Too Many Fields. S14 used code smells to indicate technical debt. The authors in this study used different strategies to detect every studied code smell.

In S28, the authors presented a list of implementation, and design configuration smells and then gave a brief description for every type. To detect implementation smells, the authors used the Puppet-Lint tool. On the other hand, to detect design configuration smells, the authors developed a tool, Puppeteer. In S29, the authors developed a tool, Designite, to detect Design smell. S33 focused on the five types of smells named Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, and Spaghetti Code. To identify those types they developed a tool named HistoryMiner to identify smell introducing commits and run the DÉCOR tool. S30 used the Sonar tool to detect code smells as indicators to TDs, while S31 used code smells, namely God Class and Dispersed Coupling, to indicate TD. Finally, the authors in S43 compared between

4 methods used to detect TD. One of those methods uses smell to detect the TD. The authors used the Codevizard tool to detect Code smells. They classified the code smells in two levels (Class-level and Method-level). God Class, Brain Class, Refused Parent Bequest, Tradition Breaker, Feature Envy, and Data Class were classified as Class-level code smell. While, Brain Method, Intensive Coupling, Dispersed Coupling, and Shotgun Surgery were classified as Method-level code smells.

Figure 4.8 Technical debt indicators

Commits and Code Comments are used to detect SATD. In S2, the authors were able to identify 273 SATDs. They found 81, 34,27,55,21, and 55 comments that related to code debt, design debt, documentation debt, defect debt, test debt, and requirement debt, respectively. S9 proposed a model to support the detection of TD through code comments analysis. To identify technical debt on code comments, the authors developed a Contextualized Vocabulary Model. The model uses word classes and code tags to support TD identification. The model provides a vocabulary that may be used to detect 12 types of TD, namely Architecture Technical Debt, Build Debt, Code

Debt, Defect Debt, Design Debt, Documentation Debt, Infrastructure Debt, People Debt, Process Debt, Requirement Debt, Service Debt, and Test Debt. From code comments, S20 identified different types of technical debt, namely design debt, defect debt, documentation debt, requirement debt, and test debt. The most common type of SATD was design debt. S21 used SrcML to parse the source code and extract the comments. They trained the Natural Language Processing (NLP) classifier by using the manually classified SATD dataset provided by S22. The S22's authors proposed an approach to automatically identify design and requirement from code comments using NLP. S26 performed a deep analysis of the change history of the subject systems by manually inspecting the commit messages and the source code related to commits having as a goal the fixing of bugs. They extracted three changes metrics, namely Fault Repairing Modification, Feature Introduction Modification, and General Maintenance Modification. S27 developed a tool to parse code comments which extracted by srcML and stored the comment nodes of the code into separate text files for each project. After that the authors analyzed the comments manually by reading through each of them to identify those that indicated SATD. S33 developed a tool, HistoryMiner, to identify smell-introducing commits. HistoryMiner tool mines the entire change history of each repository, checks out each commit in chronological order and runs the DÉCOR tool to detect smells. The authors in S36 applied 62 comment patterns proposed by S27 to identify SATD in file-level and change level. In S42, the authors automatically detect SATD in source code comments by using a text mining-based approach. To evaluate their approach performance, they used a manual dataset that was provided by S20's authors. However, the commit history of a project is not always available. For that reason, the S5' authors suggested two modularity metrics, namely Index of Package Changing Impact and Index of Package Goal Focus, that can be used to indicate ATD instance of using source code comments. The modularity metrics can be directly calculated based on source code.

Software quality and metrics are used to identify technical debt in a software project. S8's captured the optimum design by using an Entity Placement metric that ranks the refactoring suggestions according to their effect on the design based on two principles of software design (high cohesion and low coupling). S10 used the SonarQube tool to identify TD of the selected projects. SonarQube tool uses Software Quality Assessment based on Lifecycle Expectations (SQALE). SQALE identifies and organizes non-functional requirements related to code quality. SQALE defines the software quality by estimating the distance between the current and the optimum state

of the software's non-functional requirements that are defined by using rules. S18 used two approaches that use Software Quality Attribute such as Inheritance, Coupling, Size, Complexity, Cohesion, and Encapsulation to identify TD. Every attribute is assessed by using one or more than one metric. In S15, the authors used the Understand tool to assess the quality attribute of the ten open source systems. S19 used two architecture metrics to identify ATD. (1) Propagation cost (the proportion of software files that are directly or indirectly linked to each other), (2) Core size metrics (classifying every component into one of five architecture groups based on the number of direct and indirect links of the components: core, shared, control, peripheral, and isolated). S31 identified God Class, and Dispersed Coupling smells based on four static metrics: (1) weighted method count, (2) tight class cohesion, (3) access to foreign data, and (4) fan-out. To detect ATD, S39's authors used two metrics (Fan-in and Fan-out) to measure the degree to which components are coupled to each other.

Violations caused by the developers were used as an indicator of TD in S1, and to find the violation, the SonarQube tool was used. S12 developed a visualization tool to identify ATD. The tool looks for architectural violations in the detailed design that impact the efficiency of the communication between components. To identify ATD, the S17's authors used the Titan tool to calculate architecture hotspots within the source code and identify architecture issues (ATD) within these hotspots. After that, the architecture hotspots were represented as Design Structure Matrices (DSM). Then, they asked project architects to identify significant problems. They were able to recognize three types of ATD, namely Unstable Interface, Implicit Cross-module Dependency (modularity violations), and Unhealthy Inheritance Hierarchy. Based on the rules of the analysis tools PMD and FindBugs, the authors in S24 used violations to detect TD. S38 used the SonarQube tool to identify TD. The debt for each file represents various aspects of TD quantification, ranging from programming convention violations to structural characteristics of the software. S40 conducted two case studies. In the first case study (QAware project), the SonarQube tool was used to identify TD while in the second case study (Insiders Technologies project), the software quality is managed ad-hoc without any specific tool support. S41 used the SonarQube tool to identify TD. SonarQube detects the violations of a large variety of rules for software quality. These issues are classified by the tool into three categories: bugs, violations, and code smells.

In S6, the authors identify Defect debts from extracting defects, failures, bugs, which are reported in one release but not fixed in the same release. The authors in S34 used a Static Analysis tool to identify TD and test reports to defects. Their main goal was to have a mapping between code and design debt and defect/bug/issue. An issue tracking system was used to identify TD. Other researchers used other methods to identify technical debt in their studied software systems. In S7 and S25, the technical debt items were detected by the research participants. S11 identified Database debt by analyzing the SQL source code of a database schema and based on eleven rules, while S35 used the absence of foreign keys (FKs) as an indicator of Database debt. Based on a combination of integration testing and extensive unit testing, the author in S32 presented an approach to detect and avoid technical debt. In S16, the authors used different dimensions and every dimension use one or more than one criterion to identify TD, such as Documentation Completeness, Code Coverage, Components Coupling, and Reusability.

Some papers used many approaches or tools to identify TD, such as S43 and S3. In S43, the authors used four approaches to identify TD and compare between these approaches. However, in the first method, the authors used the CLIO tool to find modularity violations. In the second approach, software design patterns decay was used to detect TD. In the third approach, Code smells were used to identify technical debt. In the fourth method, the Automatic Static Analysis (ASA) tool was used to analyze source or compiled code to find violations (“issues”) that might cause faults or degrade some dimensions of software quality (e.g., maintainability, efficiency). The author in S3 executed static and runtime analysis tools, namely Findbugs, Coverity, and FxCoP, to identify a list of TD. The type of TD is classified based on the type of Error, such as Memory and Performance. Finally, other researchers used machine learning techniques to identify SATD. S21 and S22 applied machine learning to identify SATD by using Natural Language Processing (NLP) classifier. S37 developed a machine learning approach that uses source code (1) structural metrics, (2) readability metrics and, (3) warnings raised by static analysis tools as independent variables to identify design debt. S42 used Naive Bayes Multinomial (NBM), Support Vector Machine (SVM), k-Nearest Neighbor (kNN), and NLP classifier to identify SATD from source commits.

4.5.5 Classification by amount of used tools

Many tools were used in the selected empirical studies, but some tools are not for technical debt. However, as shown in Figure 4.9, SonarQube was the most used technical debt tool from 48 noticed tools which used in the selected studies. S1, S10, S13, S14, S15, S18, S30, S38, S40, and S41 used SonarQube tool to detect or assess the TD. SonarQube uses Software Quality Assessment based on Lifecycle Expectations (SQALE). SQALE identifies and organizes the non-functional requirement related to code quality. SQALE defines the software quality by estimating the distance between the current and the optimum state of the software's non-functional requirements that are defined by using rules. Besides, SonarQube classifies the violations of a large variety of rules for software quality into three categories: bugs, violations, and code smells. Findbugs tool was used by S3, S15, and S43. The PMD tool was used by S4, S15, and S37. However, the Findbugs tool scans java code to detect defects and/or suspicious code. PMD tool analyzed source code to identify warnings such as unused variables and empty catch blocks. CheckStyle tool used by S37. The authors used the CheckStyle tool for analyzing source code and identify warnings. CheckStyle, a static analysis tool that can be used to check the adherence to coding standards and identify code smells. S3 used Coverity, and FxCop, while S43 used CLIO, and Codevizard tool. Coverity tool identifies critical software quality defects and security vulnerabilities in code. Coverity is able to detect coding errors, and runtime errors that have the potential to break the live system. It supports 20 languages and over 70 different frameworks. FxCop tool analyzes the compiled object code to detect more than 200 different code violations in different areas such as design, Performance, Security, Maintainability, and Reliability. The Codevizard tool was used in S43 to find the Modularity violations and code smells, respectively. CLIO finds modularity violation by comparing the change of the components (from revision history) and how those components should change together based on the modular structure. CodeVizard calculates more than seventy software metrics from reading subversion or CVS repositories to provide views to the system, code, and metrics. The system view visualizes the infected components (10 types of code smells, namely God Class, Brain Class, Refused Parent Bequest, Tradition Breaker, Feature Envy, Data Class, Brain Method, Intensive Coupling, Dispersed Coupling, and Shotgun Surgery), and the modified components and who modified them. The code View shows the development of a single file. The metric view shows the various software metrics as line graphs. Titan was used by S17' authors. Titan can help to bridge the gap between architecture and defect prediction by

capturing the architecture and evolutionary structure. Titan® uses file dependency reports that are generated by tools such as Understand®, which was used in S15, S17, S19, S39, and S40. Understand® generates graphs that show the hierarchy of architecture, or just a sub-hierarchy. Understand® provides quality reports and structure reports. Graphs such as Treemaps® (visually metric), UML Class diagram, and Control Flow Graph, Hierarchical Graph, and dependency graphs are provided by Understand®. S39 used another tool, Call Graph Extractor, to capture important dependencies.

Figure 4.9 Most used tool

In S15, the authors used many tools to identify TD as reported before. In addition, they used inFusion tool to assess TD. InFusion able to detect more than 20 object-oriented design flaws and code smells. SrcML tool is used in many papers. S2, S21, S27, S31, and S37 used the SrcML tool. SrcML is able to parse source code in multiple languages. The source code is represented as XML where the markup tags identify elements of the abstract syntax for the language. DECOR tool was used in S33 and S37 while JDeodorant tool was used in S20 and S22. DECOR identifies code and design smells using detection rules based on the values of internal quality metrics while JDeodorant is an Eclipse plug-in that able to parse the source code and extract the code comments and to identify code smells, namely Feature Envy, Type Checking, Long Method, God Class, and Duplicated Code in software, and suggest the appropriate refactorings for every smell. JDeodorant statically analyzes source code to detect structural anomalies and does not rely on any metric thresholds. Puppet-Lint tool was used by the S28' authors to identify implementation configuration smells. Puppet-lint checks for style violations and Puppet code that may lead to errors. To obtain an estimate of the interest that is accumulated between versions, S8 used the SEAgile tool to extract the repository activity and source code metrics. SEAgile is able to extract three types of metrics, namely Commit-Related Metrics (Authors, Commits, Added Files, Deleted Files, Modified Files, Added Lines, Deleted Lines, Added Test Files, and Modified Test Files), Source-Code Metrics (Coupling Between Objects, Number of Attributes, Weighted Method Complexity, Lack of Cohesion of Methods, and Number of Methods), and Graph-Based Metrics (Number of nodes, Number of edges, Diameter, Density, Clustering Coefficient, Number of edges to new nodes, Number of edges between existing nodes, Number of edges between new nodes, Number of edges to existing nodes, and Number of deleted edges).

Researchers used other tools and methods to reach their goals. S20's authors used SLOCCount to calculate the number of the source lines of code (SLOC), while Unified Code Count tool was used by S30's authors to find number of lines of code (LOC) in the studied system. In S25, the author conducted two case studies (two teams). The first team (SoftOne) used the Vtiger tool to register their technical debt and the Trello tool as a support tool to create the technical Debt Backlog. The second team (SoftTwo) used Jira as a tool for managing activities, tasks, monitoring, and reporting bugs. S26 used the Buse and Weimer tool that evaluates the readability of code. Buse and Weimer were able to predict the code readability metric by applying machine learning. S31 used the Blaze tool to capture and record developer activity logs in Visual Studio. Blaze captures

events from a menu, window-view, and command-key actions, and for each event, it records a timestamp, the event name, the event type, and the name of the file with focus. S36 developed a tool to identify comments and the type of each comment (i.e., single-line or block comments). In addition, the tool shows, for each comment, the name of the file where the comment appears, as well as the line number of the comment. To ensure the accuracy of the developed tool, the Count Lines of Code (CLOC) tool was used to count the entire number of lines of comments. In S40, while the Code measurement is based on SonarQube and on the Seerene measurement tool (German company Seerene), the Effort and user stories were managed with the help of the custom ticket (storyteller) tool.

Many studies used Git repository and Issue tracking systems such as JIRA, Bugzilla, and SVN to extract important historical information. S6 proposed a model measuring the defect debt by analyzing the issue tracking system. They classified the bugs into regular bugs and debt prone bugs. In S13, the authors used JIRA and SVN to gather metrics that are able to describe the maintenance activities of a file during a release. To implement their methodology, when-to-release planning–TD, S16’s authors plugged their prototype into the ReleasePlanner tool. ReleasePlanner is able to perform analytical release planning based on a set of an initial pool of features. In S17, the authors used the project’s revision history from Git repository. S20 used JDeodorant tool to parse the source code and extract the code comments S36 looked for keywords such as crash, freeze, breaks, bug, fix, or defect to determine whether the change was to fix a defect. After that, by using the issue tracking system, they identify the corresponding defect report for each defect and extract relevant information from each report. In S40, the authors extracted the revision data of the first case study (QAware project) from Git or SVN and effort data from JIRA.

Despite the fact that many of the used tools are free to use, some researchers were preferred to develop their own tools to reach their goals. Many studies used one tool, and seven studies used at least three tools. Five tools (FindBugs, Infusion, PMD, SonarQube, and Understand) were used by S15. Table 4.4 presents details about empirical studies that used more than two TD tools. Table 4.3 presents the newly developed tools which aim to detect technical debt or SATD. The authors in S2 developed a tool to identify the SATD from source commits and the change history.

Table 4.3 The exclusion criteria

Developed tool name	Paper
No name found for the tool	S2
CommitAnalyzer tool	S5
ModularityCalculator tool	S5
Jcaliper tool	S8
DBCritics tool	S11
Dependency tool	S12
JCodeOdor tool	S14
Puppeteer tool	S28
Designite tool	S29
GithubGrabber tool	S38

The authors in S5 developed two tools. ModularityCalculator tool to calculate six modularity metrics, namely (1) Index of Inter-Package Usage (IIPU), (2) Index of Inter-Package Extending (IIPPE), (3) Index of Package Changing Impact (IPCI), (4) Index of Inter-Package Usage Diversion (IIPUD), (5) Index of Inter-Package Extending Diversion (IIPED), and (6) Index of Package Goal Focus (IPGF)) while the CommitAnalyzer tool calculates the Average Number of Modified Components per Commit (ANMCC). S8 developed a software tool named JCaliper, which is capable of finding the ‘distance’ between the actual and the optimum design in terms of their coupling and cohesion. JCaliper automatically extracts the number, type, and sequence of refactorings required to obtain the optimum design. S11 developed a tool, DBCritics, based on 11 rules, which are (1) Detect the use of * in SELECT request (2) Foreign key referencing a non-primary key. (3) Too many columns in SELECT request (4) Table without primary key (5) Column not key (PK/FK) using the name convention for key (e.g. “k_” in name) (6) Stub entities are used but not defined in the DB schema. (7) Isolated table. (8) Unused functions (9) View using another view (10) View using only one table (11) too many columns in a table. S12 developed a visualization tool, Dependency, that provides a suitable representation of the ATD items and their interest previously elicited. Using collected data from models and databases, and the high-level architecture with the detailed design, the tool calculates and visualizes the technical debt. The authors in S14 developed a tool, JCodeOdor, which can be used as a library in Eclipse to analyze Java systems. JCodeOdor applies detection strategies that are based on number of metrics to detect

code smells. S28 developed a tool, namely Puppeteer to identify design configuration smells. The developed tool uses different detection strategies such as metrics and other tools to detect design configuration smell. The first authors of S28 with other authors in the S29 developed a software design quality assessment tool, Designite, to detect design smells. The tool provides a Dependency Structure Matrix (DSM) to visualize the dependencies among the source code entities. Moreover, the tool supports code-clone detection. Finally, the first author in S38, developed the githubGrabber tool to analyze GitHub repository.

Table 4.4 Papers used more than one tools

Paper	Tools
S15	FindBugs, Infusion, PMD, SonarQube and Understand
S30	CheckStyle, CodePro AnahtiX, FindBugs,JoCoCo,PMD,SonarQube, and Squal
S37	CheckStyle,PMD, and TEDIUS
S28	PMD, Pppet-Lint, and Puppeteer
S3	Coverity, FindBugs, FxCop, GDI Handle, GDI View, IBM Rational Purify , and Perfmon
S40	Seerene meaurment, SonarQube, and Understand
S43	CLIO, Codevizard, and FindBugs

4.6 Results and Comparison

In this section, we present our results and compare them with the other related works (Section 3.1) to determine if there is an overlap between the results.

4.6.1 Technical Debt types (RQ1)

Motivation: related works showed many types of technical debt that have been discovered and investigated by researchers. Therefore, we expect to find other technical debt types that are presented which concern researchers. We need to find this type of technical debt. Besides, we need to know which types have been studied extensively.

Approach: To answer this question, we will go through the selected studies and find if the researchers mentioned the type of technical debt that they are studying. If the type is not mentioned

clearly, we need to go through the indicators (RQ3), or we need to go through the used tool (RQ4) to find the TD type.

Results: Figure 4.5 presents 16 types of technical debt. We found Code and Design debt are the most studied types. We found Database debt (S3, S11, and S35) and Performance Debt (S3, S16, and S25) are new two types that are addressed for the first time and never mentioned by the related work (Section 2.1).

4.6.2 Technical Debt investigators (RQ2)

Motivation: related works showed many methods used to investigate the studied phenomena. Therefore, we need to find those methods.

Approach: To answer this question, we want to go through the research questions and goals of the researchers in the selected studies and find how they investigated the TD.

Results: Figure 4.6 gives an idea about how the researchers in the selected studies investigated the TD. We found that the academics investigate the TD in various ways.

- Find the relation between TD (principal and interest), and software quality, metrics (static and change), and defects/bugs.
- Discover the ability to identify TD from code comments, metrics, and databases.
- Find out how to manage TD by visualizing it, deciding when to release, and how to avoid it.
- Investigate the ability of machine learning to identify and assess the TD.
- Evaluate some existing methods that were able to assess the TD as investigated by researchers

4.6.3 Technical Debt estimators and indicators (RQ3)

Motivation: We need to look at the methods that the researchers followed or proposed to assess or identify technical debt. Therefore, we can classify them and pick a method to help us to identify or assess the TD in our research.

Approach: To answer this question, we will go through the selected papers and find what are the methods used to detect and assess the technical debt in the selected studies?

Results 1 (Assess TD): Figure 4.7 and Table 4.2 show Technical Debt Elements and the Technical Debt Estimator, respectively. We found five Technical Debt Elements, which are

Principal, Interest, Interest Probability, Probability of Refactoring, and TD lifespan. We found that the authors used Maintainability, Effort, Quality, Violation, and experience opinion to assess the principal. Interest was assessed by using Quality and Effort. Interest probability was estimated by classifying the number of occurrences and corrective maintenance frequency. Probability of Refactoring was assessed based on three factors, namely Fault Repairing Modification, Feature Introduction Modification, and General Maintenance Modification. Finally, an experienced opinion was used to assess the lifespan of technical debt. To assess TD, machine learning was used by some researchers. However, the most common TD estimator was Effort, as showed in Figure 4.10. As mentioned in by Ampatzoglou et al.[13] found Effort as the first technical debt estimator.

Figure 4.10 Most TD Estimators

Results 2 (Identify TD): Figure 4.8 shows Technical Debt Indicators. Ten methods were used as indicators to identify TD in the selected studies. However, Alves et al.[14], and we found that smells were the most commonly used indicator. We found code comments and Defect/Bug were the second most commonly used as indicators to technical debt, while Alves found the Automatic Static Analysis (ASA) and Software Architecture Issues was the second most commonly used. Violation and software quality as TD indicators were the third most commonly TD indicators.

4.6.4 Used Tools (RQ4)

Motivation: We need to look at the tools that were used by the researchers to identify TD, assess TD, or that helped them to answer their research questions. Therefore, we can pick a tool to help us to identify or assess the TD in our research.

Approach: To answer this question, we will go through the selected papers and find what are the tools used to detect and estimate the TD in the selected studies.

Results: As shown in Figure 4.9, SonarQube and PMD were the most used TD tools where 9 and 6 papers used them, respectively. Ampatzoglou et al. [13] and Li et al. [11] found the same. However, we found around 48 tools used in the selected studies. Many of them were not to identify TD but helped the researchers to reach their goals. Ampatzoglou et al. [13] found seven tools, and Li et al. [11] 29 tools for TD. Even so, sixteen tools able to detect TD and used in the selected studies, ten new tools were developed to identify TD. That means the available tools are not enough to identify all the TD types. Besides, some Static Analyzer tools have ‘False Positive’ errors, as mentioned by S3’s authors.

4.7 Threats to Validity

This section address the possible threats to validity that may affect our study. First of all, our research questions do not cover all technical debt areas. Second, our domain covers only four years (2014 - 2017) and four databases. Despite that, those years have not covert by the related works; this domain explicitly does not include all the technical debt empirical studies. Third, we used “Technical Debt” as a search string in the selected database, which excluded empirical studies that use a specific name of technical debt, such as bad smells or design debt [11].

4.8 Conclusions

In this chapter, we inspected relevant studies in four databases. We selected forty-three empirical studies. Our objective was to find the types of the Technical Debt, the Technical Debt indicators, the Technical Debt estimators, and recognize the methods and tools used to investigate, indicate, and quantify Technical Debt from the selected empirical studies. In this study, our research questions' answers were collected. This data was summarized and analyzed to draw our main conclusions, which are compressed to the following points:

- The number of published technical debt empirical studies has been significantly increasing from 2014 to 2017.
- Code and Design debt is the most studied types. However, Database debt and Performance Debt are new two types that are addressed for the first time and never mentioned by the related work.
- Researchers investigated the ability to
 - Find a relationship between technical debt and software quality, metrics (static and change), and defects/bugs.
 - Identify technical debt from code comments, metrics, and databases.
 - Use machine learning to Identify and assess technical debt.
- The smells were the most used indicator to identify technical debt, and special attention was paid to study self-admitted Technical Debt throughout the code comments.
- Principal, Interest, Interest Probability, Probability of Refactoring, and TD lifespan are five Technical Debt Elements. Maintainability, Effort, Quality, Violation, and experience opinion are different methods that can be used to assess Technical Debt Elements
- Precisely, Forty-eight tools were identified from the selected empirical studies. The SonarQube tool was the most used tool, and the PMD tool was the second most used tool where PMD was used in six empirical studies. Precisely, ten new tools have been developed between 2014 and 2017. In seven studies, the authors used at least three tools to reach their goals

Chapter 5

The Technical Debt Density over Multiple Releases and the Refactoring Story

In this chapter, multiple case studies are conducted to study the code, design, and architecture technical debt. This chapter is based on a paper that has been written by M. BenIdris, H. Ammar, and D. Dzielski and accepted on Jul 19, 2020 for publication in the International Journal of Software Engineering and Knowledge Engineering.

The main objective of this work is to analyze TD in open source projects with respect to the average smells density, smells increment over multiple releases, and the percentage of removed smells density.

The Introduction to the technical debt is presented in Sections 5.1. Section 5.2 explains our methodology. The data analysis, results, and discussion are provided in Section 5.3. The threats to the validity of this chapter are deliberated in Section 5.4. Lastly, section 5.5 concludes the case study.

5.1 Introduction

Smells, such as Design and Code smells, are imperfect solutions to recurring implementation and design problems. Each type of smell can be identified by using a set of metrics and their threshold values. Software development companies are adversely affected by TD that negatively impacts internal software quality.

There are several types of TD. We will study three categories, ATD, DD, and CD. Researchers have most frequently investigated these three types of TD [48]. Architecture decisions that make compromises in some internal quality aspects, such as maintainability, lead to ATD [11]. The most important source of ATD is Architectural decisions[68]. DD arises when a shortcut in the principles of good design is taken. CD refers to code that is poorly written, such as code duplication [11], [14].

There are many indicators of TD, such as smells, code comments, defects/bugs, automatic static analysis, and software architecture issues. Smells are not just related to TD, but smells are related to Defect-proneness [69], Micro-patterns, and Nano-patterns [70]. Micro-patterns are known as class-level traceable patterns, while Nano-patterns are referred to as Method-level traceable patterns. Based on the structure of Java classes, twenty-seven Micro-patterns have been defined and organized into eight categories [71]. Their code quality can be improved by using Micro-patterns to detect code smells [72]. Singer et al. [73] suggested that there are seventeen types of Nano-patterns. LocalReader, a known type of Nano-pattern, is highly present in defective methods [74]. Smells can be used as indicators of technical debt. They are beneficial in detecting code complexity and fault-proneness, and they decrease maintainability [75]. In this study, we have used the smells as an indicator of TD because smells are the most commonly used TD indicators [14], [75]. Metrics and threshold rules can be utilized to detect bad smells. Many tools use metrics-based techniques such as Checkstyle [76], PMD [77], iPlasma [78], Designite [57], and Stench Blossom [79], while tools such as DÉCOR [39] use a dedicated specification language. The JDeodorant [80] tool analyzes the program code to recognize refactoring opportunities. Other tools such as MLCS [32] and Fica [81] use a machine learning approach to detect/classify smells. Code comments can be used to assist in identifying the Self-Admitted Technical Debt (SATD) [82]. In this study, we have used the Designite [83] tool to detect ATD, DD, and CD. Developers will be assisted in bringing their attention to these types of smells by observing and identifying

which type of smell has the most occurrence, by determining which behaviors are shown through multiple releases, and by detecting which smell is more reduced. These are notable and important contributions that have been incorporated into this chapter.

5.2 Methodology

In this Section, we describe our research goal, research questions, data collection methods, and research methodology.

5.2.1 Research Goals

Our goal is to use smells as indicators to analyze TD in OSSP for the purpose of understanding: (1) how TD is affected with respect to the average smells density of each type of smell in an OSSP, (2) the smells increment over multiple releases, and (3) the percentage of smells density that is removed by refactoring. The research questions presented in Section 5.2.2 aim to reach these goals.

5.2.2 Research Questions

Research Question 1 (RQ1): What is the average smell density for each type of smell in an OSSP? This question is used as an indicator of the different types of TD. It is important to address this topic because finding the average smell density for each type will help us identify which type of smell has the most occurrence in the open-source. Results of RQ1 will encourage developers to pay attention to the most common occurrence of a smell.

Research Question 2 (RQ2): Does the density of smells increase over multiple releases? This question is being addressed to determine how TD indicators change over multiple releases. It is important to address and observe the present behaviors of smells through multiple releases. This strategy will assist developers in maintaining considerations about the smell types that tend to increase through releases to reduce their impact before releasing new versions.

Research Question 3 (RQ3): What percentage of each smell type density is eliminated by refactoring? This question will be addressed to determine how TD changes through the use of refactoring. We addressed this question to find out which smell is reduced the most. The developers prioritize reducing this type because it makes the impact of their projects higher than the other types. For RQ3, we assume that any change that leads to a decrease in the number of smells implies refactoring activity. Even if the change is not a conscious or explicit refactoring

activity (say a bug fix), the change has refactored a smell. We aim to focus on the number of smells regardless of the purpose of refactoring.

5.2.3 Data Extraction

In this study, we analyze five C# open source projects selected from GitHub. GitHub is an online software repository that hosts millions of repositories of active projects and developers, and the scientific community has used GitHub as a source of data for researchers in software engineering [84] [85]. In this study, to get the OSSPs that we needed to study, we searched for OSSPs C# in GitHub. Next, the projects were sorted by “most stars,” and we selected the first five projects that: (1) had more than five releases in at least two years, (2) had 80% of the code written in C#, and (3) had the designate tool that was able to analyze the source code. We reviewed the literature for papers related to the identification of TD. We found many tools, such as Designite, Arcan, Sonargraph, ARCADE, and Structure 101. Because we needed to use an available and free tool, we applied the exclusion criteria of (1) excluding the unavailable tools, and (2) excluding tools that were not free. After we applied our excluded criteria, we discovered that Designite is the only tool that is able to discover architecture, smell, and code smell. Designite supports C# language and is able to identify 7 common architecture, 17 design, and 11 code smells. Each type of smell is defined in Table 5.1. Designite provides a simple and interactive implementation of DSM to help analyze the dependencies among the source code entities [57]. Around 55 universities from different countries have used the Designite tool, and it has been utilized in several published papers. This is important because it implies that the Designite tool is trustworthy to use.

The Designite tool uses different methods to detect different smells. Some of the smells are easier to detect than simply using metrics such as Lack of Cohesion between Methods, Number of Public Methods, and Weighted Methods per Class. Other smells require a deeper source code analysis beyond the standard software metrics. For instance, feature concentration occurs when a component is realizing more than one architectural concern/feature. This means that the component is not cohesive. Lack of Component Cohesion (LCC) is used to measure the component cohesion. To estimate LCC, Designite detects relationships such as association and inheritance among classes. It then forms the related classes in specific groups. LCC is subsequently estimated by dividing the number of groups by total classes in the namespace. A component will suffer from dense structure smell if it forms a highly dense dependency graph. To detect dense structure, the

Designite tool creates a dependency graph among all the namespaces to compute the average degree. The average degree of a graph can be computed by using the vertexes (V) and the edges (E) among the vertexes. Average degree = $2|E| / |V|$, where E is the set of all the edges, and V is the set of all the vertexes belonging to the graph. Other smell detection rules are available in reference [83].

Table 5.1 Architecture, Design and Code smells definition as defined by Designite

Architecture Smells	This smell arises when
Cyclic Dependency	2 or more architecture components depend on each other directly or indirectly.
Unstable Dependency	A component depends on other components that are less stable than itself.
Ambiguous Interface	A component offers only a single, general entry-point into the component.
God Component	A component is excessively large either in terms of LOC or number of classes.
Feature Concentration	A component realizes more than one architectural concern/feature.
Scattered Functionality	Multiple components are responsible for realizing the same high-level concern.
Dense Structure	Components have excessive and dense dependencies without any particular structure.
Design Smells	This smell arises when
Imperative Abstraction	An operation is turned into a class
Unnecessary Abstraction	An abstraction that is actually not needed (and thus could have been avoided) gets introduced in software design.
Multifaceted Abstraction	An abstraction has more than one responsibility assigned to it.
Deficient Encapsulation	The declared accessibility of one or more members of abstraction is more permissive than actually required.
Unexploited Encapsulation	A client class uses explicit type checks (using chained if-else or switch statements that check for the type of the object) instead of exploiting the variation in types already encapsulated within a hierarchy.
Unutilized Abstraction	An abstraction is left unused.
Broken Modularization	Data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions.
Insufficient Modularization	An abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both.
Hub-like Modularization	An abstraction has dependencies (both incoming and outgoing) with a large number of other abstractions.
Cyclically-dependent Modularization	Two or more abstractions depend on each other directly or indirectly.
Wide Hierarchy	An inheritance hierarchy is "too" wide, indicating that intermediate types may be missing.
Deep Hierarchy	An inheritance hierarchy is "excessively" deep.
Multipath Hierarchy	A subtype inherits both directly as well as indirectly from a super-type, leading to unnecessary inheritance paths in a hierarchy.

Cyclic Hierarchy	A super-type in a hierarchy depends on any of its subtypes.
Rebellious Hierarchy	A subtype rejects the methods provided by its super-type(s).
Unfactored Hierarchy	There is unnecessary duplication among types in a hierarchy.
Missing Hierarchy	A design segment uses conditional logic to explicitly manage variation in behavior where a hierarchy could have been created and used to encapsulate those variations.
Code Smells	This smell arises when
Long Method	A method is too long to understand.
Complex Method	A method is complex (in terms of cyclomatic complexity).
Long Parameter List	A method accepts a long list of parameters.
Long Identifier	An identifier is long.
Long Statement	A statement is long.
Complex Conditional	A conditional expression is complex.
Virtual Method Call from Constructor	A constructor calls a virtual method.
Empty Catch Block	A catch block is empty.
Magic Number	A potentially unexplained literal is used in an expression.
Duplicate Code	A method has a code clone-set.
Missing Default	A switch statement does not contain a default case.

5.3 Data Analysis, Results and Discussion

Table 5.2 reports the statistical characteristics of the selected OSSPs. These characteristics are release, KLOC, number of classes, and the number of methods. After the data is collected, the number of smells is analyzed. In this Section, we present information about the five projects and their number of smells. Exploratory Data Analysis helps to explore data for further statistical analysis and to summarize their main characteristics.

Table 5.2 the studied open-source projects from GitHub

Project #	Project name	Release	KLOC	# of Classes	# of Methods
1	Simple.Data	0.6.3	18	245	1263
		0.12.2	29	417	2282
		1.0.0-beta6	34	451	2643
		1.0.0-rc3	36	487	2796
		2.0.0-alpha1	37	508	2814

2	OpenRA	20140405-2	82	1471	4712
		20141029	98	1703	5462
		20150531	110	1866	6503
		20151224	120	1965	6398
		20160508	126	2045	6686
		20160904	132	2116	6988
		20171014	143	2281	7547
		20180307	149	2339	7831
		20180729	155	2479	8242
3	Entitas	0.1.0	1	22	109
		0.4.2	1.2	25	126
		0.9.1	3.2	70	326
		0.10.0	3.3	66	270
		0.22.3	6	135	430
		0.26.0	7	160	510
		0.30.3	12	257	807
		0.35.0	15	322	1076
		0.36.0	14	287	1031
		0.42.3	13	286	813
		0.42.4	14	290	826
		0.47.0	11	231	643
4	ZeroFormatter	1.5.2	12	251	698
		1.0.0	22	328	1163
		1.2.0	27	380	1339
		1.3.0	28	391	1350
		1.4.0	29	397	1358
		1.5.0	30	454	1398
		1.6.0	31	484	1422
		1.6.4	32	486	1423

5	ILRuntime	0.5	11	88	403
		0.8	18	181	851
		0.85	19	181	852
		1	33	230	1445
		1.1	34	238	1492
		1.2	38	354	1728
		1.3	40	368	1801
		1.4	49	444	2090

Now the results of the case study will be presented, discussed, and formulated. First, for every release, we normalized the smells by calculating the smells' density (SD). SD is equal to the number of smells for that type divided by the number of KLOC. Second, we found the average of the SD for every type of smell in every OSSP (all releases). Third, the grand mean of SD for every type of smell in all five OSSPs was calculated.

5.3.1 Research Question 1 (Smell density)

To provide an answer for RQ1, all the architecture, design, and code smells instances in all releases were detected using Designite tools. After that, the average number of each smells type in each project was estimated.

5.3.1.1 Architecture smell density

Figure 5.1 shows that Cyclic Dependency is the highest type of architecture smell in projects 1, 2, and 5. Features Concentration is the highest AS type in projects 3 and 4. Scattered Functionality has the second-highest number in OSSP in projects 2, 4, and 5. In project 3, there are 16 Ambiguous Interface instances while there are not any Ambiguous Interface instances in other projects. There are 2, 3, 14, 10, and 6 God Components while there are 18, 8, 6, 7, and 20 Unstable Dependency in projects 1, 2, 3, 4, 5, respectively. However, we found that the Cyclic Dependency has the highest density, with an average of about 0.36 in every KLOC. The Features Concentration has the second-highest density with an average of 0.22 in every KLOC.

Figure 5.1 Architecture Smells types for the studied open-source projects

5.3.1.2 Design smell density

Figure 5.2 shows the average number of design smells in the five projects. Cyclically-dependent Modularization is the highest number in projects 1, 2, and 3, with 1106, 1104, and 688 instances on average, respectively. Alternatively, there are 583 and 741 DS types of Unutilized Abstraction, which is the highest number of DS types in projects 3 and 4. - Cyclically-dependent Modularization is the highest type of DS in project 1, while the number of Unutilized Abstraction instances is 1006. In addition, there are 178, 127, 194, 404, and 178 Unnecessary Abstraction instances while there are 156, 54, 22, 21, and 41 Rebellious Hierarchy instances in projects 1, 2, 3, 4, and 5, respectively. We found that project 4 has 254 and 100 instances of Duplicate Abstraction and Broken Modularization, respectively, while there are 172, 99, 84, and 166 instances of Deficient Encapsulation in projects 2, 3, 4, and 5, respectively. Overall, Cyclically-Dependent Modularization, Unutilized Abstraction, and Unnecessary Abstraction have the highest density, respectively, with 33%, 28%, and 12% of the total density of the smells.

Figure 5.2 Design Smells types for the studied open-source projects

5.3.1.3 Code smell density

Figure 5.3 shows the average number of code smells in our five projects. Project 1, 2, 3, 4, and 5 have 381, 1614, 220, 6671, and 1129 instances of Magic Number. We found 113, 240, 105, 797, and 331 instances of Long Statement while there are 19, 130, 11, 176, and 290 instances of Long Parameter List. There are 1, 76, 22, 79, and 90 instances of Complex Method. On the other hand, project one does not have any instance of Long Method, while projects 2, 3, 4, and 5 have 22, 2, 46, and 21 instances of Long Method, respectively. Unexpectedly, there is just one instance of Duplicate code in projects 2 and 5. Projects 2, 4, and 5 have 40, 44, and 21 instances of Complex Conditional, respectively.

Figure 5.3 Architecture Smells types for the studied open-source projects

In general, Magic Number has the highest density. The average of the overall smells density for all the smells indicate CD is around 25 per KLOC. More than 70% of them are Magic Number. We also found 14% of the smells indicate CD is a Long statement. This is in comparison with Sharma et al. [22], who studied the design and code smells and found that unutilized abstraction was the most frequently occurring design smell, with around 22% of the design smells. Further, they found that the most frequently occurring code smells were the Magic number and Long statement, which constituted 80%, and 12% of code smells. They excluded the architecture smells in their study. As we mentioned, the most common ATD smell is Cyclic dependency, and the most common DD smell is Cyclically-Dependent modularization. A pertinent

question to ask here is: what exactly is the difference between -cyclic dependency and -Cyclically-Dependent modularization? Cyclic dependency arises when two or more components depend on each other directly or indirectly. Cyclically-Dependent modularization occurs when two or more abstractions depend on each other directly or indirectly.

Finding 1: Architecture Technical Debt: The Cyclic Dependency has the highest density, with an average of more than 0.36 in every KLOC. Features Concentration has the second-highest density with an average of 0.22 in every KLOC. Design Debt: Cyclically-Dependent Modularization and Unutilized Abstraction have the highest density, respectively, with 33% and 28% of the total design smells density. Code Debt: Magic Number has the highest density, with an average of more than 70% of the code smells. In addition, 14% of the total code smells is the Long statement.

5.3.2 Research Question 2 (Smell density increments)

To answer RQ2, we used regression to describe the statistical relationship between the changes in the smells density over multiple releases.

5.3.2.1 Architecture smell density increments

Figure 5.4 shows that the density of architecture smells increased in projects 2, 3, and 4 while it decreased in project 5. In project 1, the architecture smells density is almost the same. In project 1, the number of architecture smells was increased, and the number of lines of codes was also increased, nearly by the same percentage between the releases. Even though the number of architecture smells was increased, the architecture smells density (number of architecture smells/number of lines of codes) stayed almost the same for all the releases in this project. In projects 2, 3, and 4, the architecture smells density increased because the amount of increase in the project size was less than the amount of increase in the number of AS. Alternatively, in project 5, the amount of increase in the project size was large compared to the increase in AS.

Figure 5.4 The density of architecture smells in the studied open-source projects

5.3.2.2 Design smells density increments

Figure 5.5 shows that the density of design smells increased in projects 1, 2, and 4 while it decreased in project 5. In project 3, the design smells density is slightly decreased. The same can be noticed for the DS. The density of design smells increased in projects 1, 2, and 4 because the amount of increase in the number of DS was more significant than the growth in project size except release 2 for project 2. The density of design smells decreased in project five and slightly reduced in project 3 because the amount of increase in the number of DS was less than the growth in project size.

Figure 5.5 The density of design smells in the studied open-source projects

5.3.2.3 Code smells density increments

Figure 5.6 shows the CS density is increased in projects 1, 2, 3, and 5, while in project 4 it is slightly decreased. In general, the coefficient indicates that for every new release, we can expect the architecture smells, design smells, and code smells density to increase by an average of 0.44 %, 12%, and 14%, respectively. For the code smells density, CS density is increased in projects 1, 2, 3, and 5 because the amount of increase in the number of CS was larger than the amount of increase in project size. Project 4 is slightly decreased because the amount of increase in the number of CS was smaller than the amount of growth in project size, especially in release 4.

However, we estimated the correlation between the release size and the number of introduced smells. For the three types of AS, DS, and CS, the correlation was 0.73, 0.78, and 0.63, respectively. On the other hand, we found a weak correlation between the size and number of removed smells and AS, DS, and CS. The correlation was 0.46, 0.32, and 0.24, respectively.

Figure 5.6 The density of code smells in the studied open-source projects

Finding 2: The architecture smells, design smells, and code smells tend to increase. To find why the smells tend to increase, we went through the comments for each project, and we made the following observations. Refactoring activities were only reported in the second project. To give more detail in the first project, we found 314 issues. From those, 240 of them were closed. 8% of the closed cases were regarding adding new features and 4.6% to fix bugs. There were no refactoring activities reported in the comments. In the second project, we found 8,189 issues. 6,630 of them were closed issues. 20% of the closed issues had been implemented to fix bugs and 11% to add new features. Compared with other reported issues in the second project, the refactoring percentage was only 2.6% of the closed. We found 6.4% of the open issues suggested a need for refactoring. We found 845 issues reported in the third project. 82% of them were closed, and none of them were refactoring activities. In the fourth and fifth projects, we found 105 and 338 issues, respectively. 51% of the 105 issues in the fourth project were closed. No refactoring activities were found. 87% of the issues that have been found in the last project were closed, and no refactoring activities were found.

5.3.3 Research Question 3 (Percentage of the eliminated smell density)

To answer this question, we estimated the percentage of the eliminated smells for every type in every project. Figure 5.7 shows the percentage of refactoring that related to architecture smells as the highest percentage, while the design smells are shown as the lowest percentage. In general, the average percentage of the eliminated architecture smells, design smells, and code smells are 46%, 15.8, and 32.6%, respectively.

The question may arise: What is the evidence present in determining whether or not refactoring has been performed? It could be deduced that the number of smells was reduced from one version to the next. Numerous other factors can contribute to the change in the number of smells. Smells are removed as a side effect of regular maintenance rather than by intentional refactoring activities. The Designite tool classifies the smells as refactored smells if the smells that have been detected in the commit n is eliminated in the commit $n+1$. However, we do not need to know if the smells are eliminated by fixing bugs. We are more interested to know if the change has refactored a smell.

Figure 5.7 The percentage of the eliminated smells in the studied open-source projects

The results of RQ3 are determined to be the most interesting results in this chapter. ATD was the most removed type of TD. We have found that the developers were aware of issues in the project and removed them. Specifically, developers appear to be concerning themselves with architectural smells more than other smells. However, ATD was not only the most removed form of TD but also was the lowest density type found.

For project 1, the architectural smells have a higher decrease in the percentage of smells refactored. This number is 52% out of 116 architecture smell instances on the average. Alternatively, for design smells, this percentage went down to 10% out of over 2,732 design smell instances. Comparatively, the percentage of code smell instances went down to 20% out of over 528 code smell instances. However, 61 architecture smell instances, 278 design smell instances, and 107 code smell instances were removed, which means that more design smells were removed, even though their percentage was lower.

For project 2, the architectural smells have a higher decrease in the percentage of smells refactored. This number is 56% out of 89 architecture smell instances on the average. Conversely, for design smells, this percentage went down to 23% out of over 1,757 design smell instances. For code smells, this percentage went down to 38% out of over 2,184 code smell instances. However, 50 architecture smell instances, 401 design smell instances, and 826 code smell instances were removed, which means that more code smells were removed, even though their percentage was lower.

For project 3, the architectural smells have a higher decrease in the percentage of smells refactored. This number is 46% out of 66 architecture smell instances on the average. For design smells instead, this percentage went down to 35% out of over 1,071 design smell instances. On the other hand, for code smells, this percentage went down to 38% out of over 405 code smell instances. However, 30 architecture smell instances, 373 design smell instances, and 159 code smell instances were removed, which means that more design smells were removed, even though their percentage was lower.

For project 4, the architectural smells have a higher decrease in the percentage of smells refactored. This number was 46% out of 87 architecture smell instances on the average. Alternatively, for design smells, this percentage went down to 7% out of over 1,963 design smell

instances. For code smells, instead, this percentage went down to 11% out of over 7,903 code smell instances. However, 41 architecture smell instances, 132 design smell instances, and 907 code smell instances were removed, which means that more code smells were removed, even though their percentage was lower.

Lastly, for project 5, the code smells have a higher decrease in the percentage of smells refactored. This number is 56% out of 1,850 code smell instances on the average. Comparatively, for architecture smells, this percentage went down to 30% out of over 177 architecture smell instances. For design smells, instead, this percentage went down to 4% out of over 1,602 design smell instances. However, 53 architecture smell instances, 64 design smell instances, and 1,036 code smell instances were removed, which means that more code smells were removed. Furthermore, more design smells were removed than architecture smells, although the 30% from the architecture smells were refactored while only 4% of the code smells were refactored.

A strong correlation (0.75) has been found between the TD and the maintainability effort (number of removed smells). This correlation can illustrate how technical debt is connected to the effort.

Finding 3: Even though the density of the architecture smells had the lowest density compared with the density of the design and code smells, the density of the architecture smells is decreased more than the density of the design and code smells. This is because the numbers of lines of code that have been added are a large amount compared with the number of architecture smells that have been newly introduced. However, the developers aimed to eliminate the code and design smells more than the architecture smells because architecture smells were challenging to deal with [68].

5.4 Threats to Validity

Construct Validity: The smells were automatically detected using tools to avoid subjective bias. In this chapter, we defined each type of smell detected by the used tool to eliminate any ambiguity or confusion that may arise. Nevertheless, the recognized smells depend upon metrics that are implemented by the tools. This could be a possible threat to validity because other smell detection tools could detect a different number of smells for every type of smell in our studied projects.

Internal Validity: This term is related to the ability to identify the architecture, debt and code smells. The tool that was used to detect the smells may identify smell instances that are not considered as such by a human expert. Other threats to internal validity are data quality. To ensure the data quality, we went through the tool's results to determine if there were missing fields. Also, we checked if there was any duplication related to the same smell instance.

External Validity: This relates to the ability to generalize given results. This study is based on analyzed data collected from five open-source software projects, written in C#. Consequently, we could not generalize the results because they were not based on multiple studies that had replicated the same research on different open source projects using different programming languages. In addition, we could not generalize the results with respect to a commercial software project.

Conclusion validity: This term refers to the ability to draw correct conclusions. One threat to conclusion validity is related to the data sample sizes of one project from the selected projects. We selected only five releases from the first project. However, our selection was based on the duration between releases. To answer RQ2, we used multiple regression models. We described the regression models as being other threats to conclusion validity. However, the least number of releases we had were five releases for the first project. This number is the least ratio that can be used to find a model regression. To answer RQ3, we assumed that any change that led to eliminating a smell was known as a refactoring activity. We adopted this assumption based on our goal in the chapter which was not to know the reason behind eliminating the smell instance. Overall, we were more interested in knowing whether the change had refactored a smell or not. Even that change was applied due to fixing a bug.

5.5 Conclusion

This empirical study aimed to analyze architecture technical debt, design debt, and code debt created by developers. More than 25 kinds of smells in five C# open source projects were detected in this study. The obtained study results allowed us to distill the following three points:

- Architecture smells such as Cyclic Dependency and Feature Concentration had the highest density, which indicated that the ATD had 36% and about 22% of the total architecture smells. Two design smells types, Cyclically-dependent Modularization, and Unutilized Abstraction, as design debt indicators had the highest density with 33%, and 28% of the

total design smells were present. From 11 types of code smells, the Magic-Number had the highest density. More than 70% of the entire CD is Magic Number.

- According to our data, the developers refactored some of the debt in their projects. However, because they added new features (which add new debt to the un-refactored debt), the density of the TD is increased in most of the cases between releases.
- The density of the architecture smells was the least identified density. Nevertheless, it decreased more than the density of the design and code smells. However, the developers tended to eliminate the code and design smells more than the architecture smells because architecture smells are difficult to deal with.

Chapter 6

Investigating ATD Risk and Refactoring, the Perspective of Software Developers

In this chapter, a survey of developers, architects, and project managers is examined in order to learn the following: (1) To find out how software development teams rate the risk of the architecture smells with respect to each of their roles. (2) Identify the reasons that prevent software development teams from refactoring their projects to pay off ATD. (3) The benefits that can be gained from refactoring are targeted to overcome the architecture debt. (4) Understanding challenges that might arise during the refactoring process.

The main motivation for this survey was to analyze the participants to find the above from the perspective of the industry. Chapter 6 is organized as follows: Section 6.1 presents the introduction to this chapter. The research methodology is explicated in Section 6.2. Section 6.3 detailed the discussion and the survey results. Section 6.4 deliberated the threats to the validity of the survey. The participants' answers were concluded in Section 6.5.

6.1 Introduction

Refactoring can be used to eliminate bad smells. The point of refactoring is to improve the software's internal structure without changing its external behavior [2]. Refactoring improves software quality and reduces maintenance costs. A software system can be negatively affected by bad smells. Bad smells are an indicator of imperfect solutions to recurring implementation and design problems[9]. Code, design, and architecture smells are three types of smells that can harmfully affect software quality. Bad smells can be identified by using a set of metrics and their corresponding threshold values [9]. However, release deadlines make it difficult for developers to perform refactoring since they prefer to add new features or fix bugs instead of refactoring because refactoring as it does not provide an immediate benefit [9]. Nonetheless, if the bad smell starts to be an obstacle to further software evolution and maintenance, refactoring becomes necessary [86]. Unlike other types of smells, we notice that few empirical studies focus on practitioners' perception of architecture smells (ASs) and architecture technical debt (ATD). ATD can be detected by architecture smells, code comments, and software architecture issues. However, architecture smells are the most common method used to detect ATD [14], [48]. ATD is incurred by an architecture decision that compromises the software quality aspects [11]. In this study, we focus our attention on ASs and ATD from the perspective of software developers, architects, and project managers. We need to know if developers, architects, and project managers are aware of the risk of the ASs. If they are, which AS types have high risk? Also, what obstacles prevent them from refactoring? Finally, we are interested in discovering what benefits can be gained from refactoring, and what challenges one might face during refactoring?

We conducted an online survey to examine the above questions—the participants in the study consisted of 33 developers, 11 architects, and 6 project managers. We analyzed the responses to evaluate the perceived AS risk, find refactoring benefits and challenges, and identify the reasons that prevent them from refactoring. Based on participants' answers, we concluded some facts about architecture smell risk and the benefits, challenges, and refactoring risk. We also pointed out some reasons that prevent developers from refactoring.

6.2 Research Methodology

An online cross-section survey was developed and utilized to evaluate AS risk, identify the reasons behind preventing refactoring, and explore refactoring benefits and challenges. Snowball

sampling was used to increase the number of professional software engineers that could be reached. The survey had three parts. Part 1 was general questions such as gender, level of education, software developer roles, experience, etc. In part 2, we introduced the ATD definition and explained how it could be detected. We presented seven types of architecture smells along with their descriptions. Following this, we asked them about the likelihood and the impact of every AS. In part 3, we defined the refactoring and displayed refactoring examples. We then asked questions regarding refactoring, such as refactoring benefits, refactoring challenges, and the reasons preventing refactoring. However, before executing the study and in order to improve our questionnaires, a pilot study was conducted with three developers. Appendix B shows the survey questions.

In total, 66 participants responded to our survey. Because we considered the different software professional roles, 50 responses were interesting because they were from the software industry. Software designers (3 participants) were included with software architects, and software testers (3 participants) were added with the developers because the number of designers and testers were small. In total, 50 responses are used in this chapter. 66% of them are software developers, 22% are software architects, and 12% are software project managers.

Our research goals are (1) Discovering how the software development teams rate the risk of the architecture smells with respect to their roles. (2) Identifying the reasons that prevent software development teams from refactoring their projects to pay ATD. (3) Exploring the benefits that that can be gained from refactoring targeted to overcome the architecture debt. (4) Examining challenges that might arise during the refactoring process. To achieve our goals, we have formulated the following research questions.

RQ1: which type of architecture smells has the highest risk on the software components?

RQ1.1: What is the most probable type of architecture smells that might occur in the software components?

RQ1.2: Which type of architecture smell has the most significant impact on software components?

RQ2: What prevents software development teams from refactoring their projects for ATD?

RQ3: Which benefits can be obtained by software development teams through refactoring?

RQ4: What are the major challenges that developers encounter while refactoring their ATD?

6.3 Result and Discussion

In total, there were 50 participants from 9 different countries. Most of the participants are from Libya, 19 (38%), and the USA 10 (20%). Six participants (12%) were from Iraq and five (10%) from Canada. Four participants (8%) were from Palestine, and two (4%) from Norway. Two participants (4%) were from Saudi Arabia and one (2%) from Kuwait. Finally, one participant (2%) was from the UK. The respondents' experience ranged from 1 to 40 years (an average of 12.4 years and a median of ten years). Figure 6.1 shows the experience of the participants. Eight participants have fewer than five years' experience. Eighteen participants have experienced between 5 and 10 years. Sixteen participants had the experience of more than ten years and less than 21 years. Five participants had more than 20 years and less than 31 years' experience—finally, one participant had forty years' experience in software development.

Figure 6.1 Participants experiences

All the participants had at least an undergraduate degree. The level of education was high, 30%, 46%, and 24% of the participants had a Ph.D., Master, and bachelor degree, respectively. Almost all of them were male, with 49 (98%) males and just 1 (2%) female. Before discussing our results, we present the participants' knowledge about the concept of ATD. From the entire set of respondents, 4 (8 %) had never heard of ATD, 9 (18%) had inadequate knowledge, 24 (48%) had

adequate knowledge, 10 (20%) had good knowledge, and 3 (6%) had very good knowledge, However, in the survey, we defined the concept of ATD, and explained how to detect ATD. Also, we included “Don’t know” as an answer to avoid the possible selection of a random answer.

Table 6.1 Answers rate for research questions (RQ1.1, RQ1.2, and RQ1)

Research question	Software developer roles	% of the participants							Average	Median
		Cyclic Dependency	Unstable Dependency	Ambiguous Interface	God Component	Feature Concentration	Scattered Functionality	Dense Structure		
RQ1.1	Developers	97%	97%	94%	91%	97%	94%	94%	95%	94%
	Architects	82%	100%	100%	82%	82%	82%	82%	87%	82%
	Project managers	100%	100%	100%	100%	100%	100%	100%	100%	100%
RQ1.2	Developers	79%	97%	91%	91%	91%	88%	91%	90%	91%
	Architects	82%	91%	100%	100%	82%	82%	91%	90%	90%
	Project managers	100%	100%	100%	100%	100%	100%	100%	100%	100%
RQ1	Developers	94%	94%	88%	85%	91%	88%	91%	90%	90%
	Architects	82%	91%	91%	73%	73%	73%	82%	81%	81%
	Project managers	100%	100%	100%	100%	100%	100%	100%	100%	100%

Table 6.1 and Table 6.2 shows the answer rate of the participant. First, for RQ1.1 and RQ1.2, we classified the answer rate by architecture smells type and software role. The project managers answered all the questions. As we see in Table 6.2, the developers answered all the questions related to RQ3, RQ4, and RQ5. The developers’ average and median answers rate for other questions that related to RQ1.1, RQ1.2, and RQ1 was at least 90%. The architects’ average and median answers rate for questions related to RQ1.1, RQ1.2, and RQ1 was at least 81% while their answer rate for other questions related to RQ2, RQ3, and RQ4 was 91%.

Table 6.2 Answers rate for research questions (RQ2, RQ3, and RQ4)

Research question	Software developer roles	% of the participants
RQ2	Developers	100%
	Architects	91%
	Project managers	100%
RQ3	Developers	100%
	Architects	91%
	Project managers	100%
RQ4	Developers	100%
	Architects	91%
	Project managers	100%

Here we present our results, organized according to the research questions stated in Section 6.2:

6.3.1 ATD Likelihood, Impact, and Risk

Before answering RQ1, we need to find the likelihood (RQ1.1) and the Impact (RQ1.2) of every architecture smell. Hence, we asked the participant to rate them. From the collected answers, we are able to estimate AS risk.

RQ1.1: What is the most probable type of architecture smells that might occur in the software components?

Motivation: In Chapter 5, we noticed that the architecture smells have a different likelihood, from project to project. We expect to find which type of architecture smells the participants are most likely to face in their daily work.

Approach: To answer this question, we need to go through the participants' answers and analyze their responses first in general, as seen in Figure 6.2 and with respect to their software development roles, as seen in Figure 6.3.

Figure 6.2 The most probable AS type from the perspective of all participants

Results: Figure 6.2 shows the most likely type of architecture smell that might occur in the software components from the perspective of all the participants without looking at their software roles. We can see that 32 (64%), 31 (62%), and 29 (58%) participants found cyclic dependency, Feature Concentration, and God Components as the most probable type of architecture smells, respectively. Figure 6.3 shows the most probable kind of architecture smell from the perspective of different software development roles. From the standpoint of developers, we recognized that more than two-thirds of developers reported Cyclic Dependency, Feature Concentration, and God Components as the architecture smells with the highest probability of occurrence in software components. In contrast, almost two-thirds pointed to Dense Structure as the smell with the highest likelihood of occurrence in the software components.

Figure 6.3 The most probable AS type from the perspective of different software development roles

From the perspective of architects, we can easily see that most of the architects have the same view as the developers regarding Cyclic Dependency and Feature Concentration likelihood. More than three-quarters (78%) and two-third (67%) of the architects pointed to Cyclic Dependency and Feature Concentration as the most probable architecture smell types, respectively. From the perspective of project managers, we can understand that the project managers have different perspectives. Most project managers, 83% and 67%, pointed to God Component and Scattered Functionality as the most probable architecture smell type. On the other hand, just 33% and 50% of the project managers share the perspective of the other professions regarding the probability of Cyclic Dependency and the Feature Concentration, respectively.

RQ1.2: Which type of architecture smell has the most significant impact on software components?

Motivation: We noticed that the architecture smells have a different likelihood, and we are sure that they have a different impact. However, finding the AS type with the highest impact on

the software components from the perspective of the participants is expected to help other practicing software engineers to make attention to this type during their daily work.

Approach: To answer this question, we will follow the same process as in RQ1.1. First, the participants' answers will be analyzed in general, as seen in Figure 6.4. Second, we will explore their results from the different perspectives of various software development roles, as seen in Figure 6.5.

Figure 6.4 The AS with the most significant impact from the perspective of all participants

Results: Figure 6.4 shows the architecture smell with the highest impact from the perspective of all the participants without looking at their software roles. We can see more than half of the participants share the same fear regarding the impact of the Dense Structure, Cyclic Dependency, Unstable Dependency, Feature Concentration, and Scattered Functionality. On the other hand, only 48% and 40% of the participants pointed to God Component and Ambiguous Interface, respectively as the architecture smell with the highest impact on their software system.

Figure 6.5 The AS with the most significant impact from the perspective of different software development roles

Figure 6.5 shows the impact of architecture smells from the perspectives of different software development roles. 67%, 66%, and 65 of the Developers pointed to Dense Structure, Scattered Functionality, and Cyclic Dependency, respectively, as the architecture smells with some of the highest impacts on the software components. More than three-fourths of the architects found Cyclic Dependency has some of the highest impacts on their software components. Simultaneously, 70% of them saw Dense Structure and Unstable Dependency have some of the highest impacts on the software components. From the perspective of the project managers, Feature Concentration has the highest impact, where 83% of them pointed to it. However, 67% and 60% of the project Managers found Cyclic Dependency and Dense Structure have a high degree of impact on the software system.

RQ1: which types of architecture smells have the highest risk from the software components?

Motivation: Finding the impact and the likelihood of every architecture smell will help to assess the risks of every architecture smell, which may help in making refactoring decisions.

Consequently, after analyzing the results of RQ1.1 and RQ1.2, we can find which ATD type has the highest risk, based upon expert judgment.

Approach: To answer this question, we use the RQ1.1 and RQ1.2 results to assess the ATD risks for every architecture smell type. We convert the participant's answers for RQ1.1 and RQ1.2 to values between 0 and 1. After that, we look at every participant's answers and multiple the impact answer with the probability to assess the risk value. In RQ1.1 and RQ1.2, every response answer has a scale of 1 to 5. For instance, Participant number five answers that the magnitude of the impact and the likelihood of the impact the architecture smell was a very high impact (5) and likely to occur (4), respectively. So the risk= $(5 * 4)/25 = 0.8$. This was done for each participant individually. After that, we classified the risk at three levels: (1) If the value of the risk is larger than 0.6 $\rightarrow$ High Risk, (2) If the risk value is between 0.59 to 0.2 $\rightarrow$ Moderate Risk, (3) If the risk value is less than 0.2 $\rightarrow$ Low risk)

Figure 6.6 ATD risk from the perspective of all participants

Results: Figure 6.6 displays the architecture risk in general without looking into the differences between the software development roles. Precisely 52% of the participants' answers indicate that Cyclic Dependence and Future Concentration have high risk. 45% of the participants' responses indicate the God Component has a high risk to their software components. Table 6.3

presents the participants' answers with respect to their software development roles. Precisely 78% of the architects' responses indicate cyclic dependency as the architecture smell type with the highest risk, and just 50% and 48% from the project managers and developers answers' show the same. Precisely 83% of the project managers' answers indicate Feature Concentration as the highest risk on the software components, and 57% of the developers share the perspective of the project managers regarding the risk of Feature Concentration.

Table 6.3 ATD Risk from the perspective of different software development roles

Architecture Smell Type	Developers	Architects	Project Managers
Cyclic Dependency	48%	78%	50%
Unstable Dependency	39%	20%	67%
Ambiguous Interface	31%	20%	50%
God Component	50%	25%	67%
Feature Concentration	57%	38%	83%
Scattered Functionality	38%	25%	67%
Dense Structure	53%	11%	67%

6.3.2 Refactoring preventions

RQ2: What prevents software development teams from refactoring their projects for ATD?

Motivation: Many reasons can prevent performing refactoring in an industrial contexts. Asking the different industrial software developer roles, the reasons that prevent them from performing the refactoring will highlight the obstacles so that decision-makers can consider them.

Approach: To answer this question, the responses's answers will be analyzed from the perspective of various software development roles, as seen in Table 6.4.

Results: A large majority; 83% of the project managers and 73% of the Architects, indicate a lack of awareness about refactoring's impact on product quality is a reason that prevents them from performing the refactoring. On the other hand, only 30% of the developers share the same fear, which means that developers are more aware of the impact of refactoring on product quality.

However, 55% of them cannot perform refactoring because they are unable to predict the impact of the changes, such as adding bugs to their product.

Table 6.4 ATD refactoring preventions that the survey participants experienced

Prevent ATD Refactoring	Developers	Architects	Project Managers
Emphasis on new feature implementation	51%	55%	50%
Unable to predict the impact of the changes caused by refactoring	55%	55%	17%
Lack of awareness about refactoring's impact on product quality	30%	73%	83%
Unable to convince higher management about the need for refactoring	24%	64%	50%
Unable to estimate the effort required for refactoring	37%	36%	33%
Lack of support from project management	18%	64%	33%
Unable to prioritize refactoring candidates	24%	28%	0%
Unable to clearly show the improvement in software quality	22%	28%	33%
Unable to identify refactoring candidates	12%	28%	33%

We noticed that 64% of the architects are not able to perform refactoring because they are unable to prove the necessity for refactoring to the managers, and the lack of support from their managers. We can realize that the project managers and the architects are paying more attention to the impact of refactoring on software quality. We summarized the participants' answers regarding what prevents them from performing refactoring in Table 6.4.

However, around 74% of the participants (25 Developers, 8 Architects, and 3 Project Managers) do not use a refactoring tool, while 26% of the participants (8 Developers, 3 Architects, and 3 Project Managers) use refactoring tool(s). For participants who use refactoring tool(s), we found that the average and median result of the refactoring decision is "usually perform refactoring" On the other hand, it is "sometimes perform refactoring" to participants who do not use a refactoring tool (With Correlation = 0.3, P-value < 0.05). From this, we can see that using a refactoring tool can increase the likelihood of refactoring, and the absence of the refactoring tool will prevent software development teams from performing the refactoring. Compared with Sharma et al. [30], who surveyed 39 software architects, they found that fear of breaking the working code

(37 participants) and emphasizing feature implementation (28 participants) were the primary two reasons that prevent architects from performing refactoring tasks.

6.3.3 Refactoring Benefits

RQ3: Which benefits can be obtained by software development teams through refactoring?

Motivation: Many benefits can be gained by performing the refactoring. By highlighting refactoring benefits, we hope to reduce the reasons preventing refactoring in order to get the benefits of refactoring.

Approach: For this question, we analyzed participants' answers from different software developer roles. Table 6.5 presents the participants' responses.

Results: Most, 65% of the developers pointed to the fact that refactoring will improve the readability and performance, while 64% of them indicated that refactoring improves software usability. On the other hand, reducing code duplication and improving maintainability can be gains of refactoring, as stated by 58% of the developers. Precisely, 64% of the Architects pointed to reducing code duplication and reducing bugs as the first benefit of refactoring, while 55% of them pointed to improve usability, help adding new features, reduce complexity, reduce code size, and reduce maintainability cost as refactoring benefits. Nevertheless, helping to add new features is stated by 83% of the Project Managers as a refactoring benefit, while 67% of them indicated that refactoring improves performance and reduces maintenance cost.

We comprehend that developers are refactoring their code, expecting to increase their software readability, performance, and usability. Project Managers hoping performing refactoring will help in adding new features, improve performance, and reduce maintenance costs. However, reducing duplication and the number of bugs are some expected refactoring benefits from the perspective of the software Architect. Other refactoring benefits are reported in Table 6.5. Compared with related works, around 43% and 30% of the participants, respectively [25] pointed to improving readability and maintainability as the first two refactoring benefits. Similarly, 82% and 85% of the participants on [26] shared the same thought. However, 68% and 63% of the participants on [26] pointed at reduced duplication and helping in adding new features as refactoring benefits.

Table 6.5 ATD refactoring benefits that the survey participants experienced

ATD refactoring Benefits	Developers	Architects	Project Managers
Improve readability	65%	46%	50%
Improve performance	65%	45%	67%
Improve usability	64%	55%	17%
Reduce duplication	58%	64%	50%
Improve maintainability	58%	46%	33%
Reduce bugs	55%	64%	50%
Improve testability	52%	46%	50%
Help adding new features	49%	55%	83%
Reduce complexity	48%	55%	50%
Reduce code size	48%	55%	50%
Reduce maintenance cost	42%	55%	67%

We asked the participants how often they refactor their projects. We noticed that the highest percentage of participants who answered with; always perform refactoring, were Developers (8 Developers). This looks axiomatic because Developers are unlike Project Managers and Architects since they deal with the code every day, and performing refactoring helps developers to improve the code's readability, performance, and usability, as around 65% of the developers said in Table 6.5. A large minority; 45% of Architects usually perform refactoring to fix bugs and 64% reduce code duplication, as they stated. All the Project managers at least sometimes perform refactoring because refactoring helps with adding new features, as stated by 83%, or to improve the project performance and reduce maintenance cost, as stated by 67% of Architects.

6.3.4 Refactoring Challenges

RQ4: What are the major challenges that developers encounter while refactoring their ATD?

Motivation: Many challenges may have to be faced while performing the refactoring. We hope that finding the refactoring challenges will allow them to be taken into consideration to reduce their consequences before making refactoring decisions.

Approach: To answer this question, we analyzed the participants' from different software developer roles answers.

Table 6.6 ATD refactoring challenges that the survey participants experienced

ATD refactoring Challenges	Developers	Architects	Project Managers
Taking more time than expected	66%	55%	83%
Adding bugs	61%	55%	50%
Disrupting adding new feature	37%	55%	67%

Results: The refactoring challenges are presented in Table 6.6. Developers, Architects, and Project Managers are having difficulties predicting the expected time of the refactoring correctly. Taking more time than expected was a significant concern, as pointed by 83% and 66% of the project managers and developers. Compared with Jain's empirical study[26], 43% of the respondents pointed to the fact that refactoring takes time from other tasks, and nearly 19% of the participants on [25] believed the same. Precisely, 61% of the Developers, 55% of the Architects, and half of the Project Managers have a fear of adding bugs in performing the refactoring. 74% of the participants on [25] share the same concern. Disrupting adding new features is a significant concern. It can be a challenge that gets the attention of project managers during refactoring, even though 63% of the developers and almost half of the Architects did not have a significant concern regarding that. We realize that time is critical in the software development process because of delivering deadlines, and that is why most of the participants have a concern about the expected refactoring time. Approximately 72% of the participants on [26] found refactoring time-consuming. We can conclude that project Managers, in contrast, were more concerned about that the expected refactored time and the fear of disrupting the adding of new features. This fear is understood because delivering software on time with good quality and including new features to satisfy the customer is the project managers' objective. The results indicate that 62% of the participants who do refactoring without using any tool cannot accurately predict the time required for refactoring. Unexpectedly, 85% of the participants who used the refactoring tool could not predict the time of the refactoring correctly. This means the refactoring tools that they used are only able to *detect* ATD, or that the refactoring tool is not accurately estimating the refactoring time. However, we found 62% of them do not have a fear of disrupting the adding of new features. Therefore, using the factoring tool can help to reduce the refactoring time. On the other hand, 50% of the participants who do refactoring without using refactoring tools do have a fear of disrupting

the adding of new features. This is understandable because they need more time to detect ATD than participants who use refactoring tools who just need time to refactor the selected candidate.

Most (72% of the participants) do not use tools to help them in detecting or refactoring ATD. That means they likely don't find themselves to be responsible for detecting ATD or even in performing the refactoring. Therefore, who do you think is responsible for detecting the ATD, and who do you think is responsible for performing refactoring ATD are questions that were asked. Figure 6.7 presents the participants' responses to this question, while Figure 6.8 presents the participants' response to whom is responsible for performing the refactoring of ATD.

Figure 6.7 Software roles responsible to detect ATD (Perspective of all participants)

Figure 6.7 shows that a few, (14% of the participants, 7) participants stated that Developers are the ones who are responsible for detecting ATD. 86% of them are Developers, and the rest are Architects. Some, (24% of the participants, 12) participants stated that Architects are responsible for detecting ATD. Where 54%, 25%, and 17% of them were Developers, Architects, and Project Managers, respectively. Only (8% of the participants, 7) participants stated that project managers are the ones who are required to detect ATD. Half of them were Developers, and the rest were Project Managers. Some, (28% of the participants, 9 Developers and 5 Architects) found themselves sharing the responsibility of detecting ATD. Only 6% (3 Developers) of the participants stated that Project Managers should be sharing with them the responsibility of detecting ATD. On the other hand, only 4% (2 Developers) of the participants found that Architects and Project Managers were sharing the responsibility of detecting ATD. Finally, 16% (4 Developers, 2 Architects, and 2 Project managers) of the participants stated that all of them as one team shared the responsibility for detecting ATD.

Figure 6.8 Software roles responsible to perform refactoring ATD (Perspective of all participants)

Figure 6.8 shows that a few (14%) of the participants (100% Developers) stated that Developers are the one who are responsible for paying off the ATD by performing the refactoring. A small fraction, (12%) of the participants (3 Developers and 3 Architects) stated that Architects are responsible for the refactoring of ATD. Only 4% of the participants stated (1 Developer and 1 Project manager) that project managers are the ones who are responsible for performing refactoring. A large minority, (40%) of the participants (20 participants: 11 Developers, 6 Architects, and 3 Project managers) found that Developers, and Architects were responsible for paying off the ATD by performing the refactoring. Only 6% (2 Developers and 1 Project manager) found themselves sharing the responsibility of performing the refactoring. On the other hand, only 4% (1 Developer and 1 Project manager) of the participants found that Architects and Project Managers shared the responsibility for detecting ATD. Finally, 20% (80% of them were Developers, and 20% were Architects) of the participants stated all of them as one team shared the responsibility for paying off the ATD.

Detecting and managing ATD is a very important task, but there is conflict in the participant's answers. We found that the task of detecting and managing ATD needed to be assigned to one software role or two software roles before the team starts the development of the project. Finally, in this study, 90% of the participants' answered ATD could affect their project. Almost all the Architects (answers rate 82%) and the Project Managers (answers rate 100%) were certain that ATD could affect their project. In addition, 85% of the Developers (answers rate 82%) found that ATD could affect their project.

6.4 Threats to Validity

Construct, internal, external, and conclusion validity are four essential aspects of the quality of any empirical study. This section discusses these four aspects.

Construct Validity is related to whether we measured what we intend to measure. We formulated research questions to reach our goals. For each research question, a number of questions will be answered by the participants. The misunderstanding of our questions can be a threat to Construct Validity. To reduce this threat, the survey was divided into three sections. Section 2 and 3 started with explanations and definitions. For instance, in the second section, we defined the studied phenomena and explained how ATD could be detected using architecture

smells, which were also described. A short description of refactoring and its types were given. To avoid biased answers, the questions were carefully worded. Finally, in some questions, the respondent could have more than one response. For that reason, in addition to multiple choices that were allowed, the category “Other” was included [87].

Internal Validity is related to influences that can affect variables/measurements without our knowledge. The participants’ answers could be a threat to Internal Validity if the participants did not understand our explanation of the studied phenomena. However, this threat can be reduced by including “Don’t know” as an answer to avoid possible misinterpretation.

Conclusion Validity is related to our ability to draw correct conclusions. One threat to conclusion validity is connected to the number of participants. Through email, 66 participants responded to our survey with an answer rate of around 62.8% and a 95% average complete rate. Sixteen participants were excluded because they were academics. Fifty participants who were in the software industry were interested. As a result, the responses from 50 participants that had different software roles, expertise, education, and different nationalities were used in this study. Another threat to conclusion validity is the descriptive statistics of the participants’ information, such as expertise and level of education. For that reason, the median and mean experiences of the participants were reported.

External Validity is related to the ability to generalize the results. The participants who were involved in this study could be biased. For instance, 39% of the participants from Libya and 20% from the US. On the other hand, just one participant from the UK and one from Kuwait. For that reason, and even though the participants were from nine countries, generalizing this study is not possible.

6.5 Conclusions

It is essential to assess the ATD risk since estimating the ATD risk can help project managers and decision-makers make good refactoring decisions. Determining the architecture smells likelihood and impact, the refactoring benefits, the refactoring challenges, and the risk of refactoring are the main goals of this study based on a survey with 50 respondents, the survey results can be distilled in the following seven points:

- The most likely type of architecture smell that might occur in the software components from the software Developers and Architects is Cyclic Dependency, and from the perspective of the software Project Managers, it is God Component (RQ1.1).
- The architecture smell with the highest impact from the perspective of developers is Dense Structure, Scattered Functionality, and Cyclic dependency. Cyclic dependency and Feature Concentration has the highest impact from the perspective of the software Architects and project managers, respectively (RQ1.2).
- The developer's responses and project managers' responses indicated Feature Concentrations has a high risk to their software component, while Architect's responses showed Cyclic Dependency (RQ1).
- Furthermore, this study discloses the reasons behind avoiding performing the refactoring. We found that the unpredictable impact of refactoring to adding new features prevents Developers from performing the refactoring. On the other hand, the effect of refactoring on software quality is the main concern of Project Managers. Besides sharing the same fears as the project managers', Architects indicated adding new features, convincing higher management, and lack of support from project management as other reasons to prevent refactoring (RQ2).
- This study reveals the readability, maintainability, and usability of code are some refactoring benefits the Developers can gained. Reducing duplication and bugs are some refactoring benefits mentioned by Architects. Helping to add new features, improve performance, and reduce maintenance cost is an enormous refactoring benefit, according to Project Managers (RQ3).
- This study shows that having difficulties accurately predicting the refactoring time is the largest refactoring challenge for all participants. In addition to that, Developers and Architects have a fear of adding bugs through performing the refactoring. On the other hand, refactoring may disrupt adding a new feature and this was a significant concern for participated Project Managers (RQ4).
- Finally, most of the participants believe that ATD affect software systems, and we found using a refactoring tool can increase the likelihood of refactoring.

Chapter 7

Prioritizing Software Components Risk: Towards a Machine Learning-based Approach

This chapter explains using machine learning to prioritize software components based on the severity of the expected ATD. This chapter is based on paper that has been written by M. BenIdris, H. Ammar, D. Dzielski, and W. H. Benamer and accepted in the 6th International Conference on Engineering & MIS 2020. The paper has been published in the Association for Computing Machinery (ACM) database [88]. Our goal is to use machine learning techniques to help project managers and decision-makers make accurate and more informed decisions by building a model that can detect and prioritize based on the ATD severity on the software components. This chapter is structured as follows. Section 7.1 presents the introduction to this chapter. Section 7.2 describes the methodology. Section 7.3 details data analyses of the results and case studies. Section 7.4 presents the threats that may be affecting the findings. Finally, Section 7.5 gives the conclusion.

7.1 Introduction

Release deadlines prevent developers from refactoring. Generally, developers prefer adding new features or fixing bugs instead of refactoring because refactoring does not provide immediate benefits [2]. Problems will appear if different tools are used to detect smells because every tool depends on detection rules based on various metrics and related [40]. For instance, Designite, Arcan, Sonargraph, and Structure 101 are tools used to detect ATD in software components. Decisions taken about refactoring may be limited by smell detectors tools. Different tools identify different smells. Different tools use different thresholds, which affect the accuracy of the detector results [31], [32]. To overcome these limitations, we will use machine learning techniques to identify and classify the severity of the architecture technical debt in the software components. Due to the relationship between data representation and machine learning models, different machine learning techniques will be used on different representations to obtain the highest classification accuracy [89].

7.2 Methodology

To build our ML model, we needed to go through three steps:

STEP #1: (GET THE ML INPUTS)

To get the inputs for machine learning, we analyzed four C# open-source system projects selected from GitHub. We used “C#” as a search string to find all the related sources. We sorted the projects by “most stars,” and we selected four projects with multiple releases. After that, we used Designite to find and save the four types of internal structure metrics (ML inputs) to use them to indicate the ATD.

Table 7.1 Information about the four projects used to learn machine learning

	Project name			
	OpenID, OAuth protocols	Xamarin android	Hawk	Sharp compress
Release	4.0.012030	9.1.5.1	5.0	0.11.6
KLOC	85.2	74	27.5	162.5
Methods	3396	3594	1315	7162
Classes	790	742	301	1344

We used four types of internal structure metrics to indicate three types of architecture smells in software components. We selected those four metrics based on previous researches [90]–[93]. Lack of Component Cohesion (LCC), the increased value of LCC reveals the Feature Concentration smell [94]. Weighted Method per Component (WMC): It can be measured either by counting the number of methods associated with a component or summing the complexities (CC) of the methods. High WMC indicates the complexity of a component. Fan-in and Fan-out: These indicate Hub-Like Dependency [90]. Fan-in is the number of modules that call a given module, and Fan-out is the number of modules that are called by a given module. To produce a ground truth for machine learning. First, we established criteria to be used for machine learning. The initial criteria were obtained from related work and the Designite tool. We examined the effect of the architecture smell on the internal structure which can be measured by software metrics. We designed a criteria for examining the internal structure. The package's severity was categorized in four levels using static analysis extracted from C# open source projects. We classified severity in four categories for different internal qualities that packages have Complexity, Cohesion, and Dependency. The values and the thresholds were extracted using the Designite tool. Higher amounts in the metrics indicate an urgent need for refactoring to this package. Our ground truth is theoretical. We expected that the Lack of Component Cohesion (LCC) leads to Feature Concentration smell [94], indicating the component's lack of cohesion. We also expect that the Weighted Method per Component (WMC) shows the complexity of a component. Complexity is a quality factor which identifies critical design elements of the software. Excessive software complexity makes code harder to read and understand and also makes it more difficult to change [95] and tends to lead to more defects [96]. We also see that the number of modules that call a given module (Fan-in) and the number of modules that are called by a given module (Fan-out) show Hub-Like Dependency [90]. Hub-Like Dependency overloads a part of the software with too much responsibility for the whole (violating modularity). Also, any change in a hub class will affect the dependent classes (at least), and any change in the dependent classes will affect the hub class (violating the healthy dependency structure) [90]. This theoretical ground truth is in the form of 288 architecture smells that have been extracted by running the Designite tool on four open source project releases that are shown in Table 7.1

STEP #2: (GET THE ML OUTPUTS)

We used machine learning algorithms to model architecture debt detection. We had four independent variables which were LCC, WMC, Fan-in, and Fan-out. The dependent variable is a component risk, then severity of the ATD as a whole entity was calculated. Following that, we calculated the severity of every component. Determining the severity of TD in every component helped us prioritize refactoring and could help project managers make better decisions. The severity of every type of ATD was estimated. The severity level for each metric was classified based on the threshold [91] of every metric.

1. LCC

<i>if $LCC < 0.2$</i>	<i>No Severity</i>
<i>if $0.2 \leq LCC \leq 0.49$</i>	<i>Low Severity</i>
<i>if $0.49 < LCC \leq 0.79$</i>	<i>Moderate Severity</i>
<i>if $0.79 < LCC$</i>	<i>High Severity</i>

2. WMC

<i>if $WMC < 300$</i>	<i>No Severity</i>
<i>if $300 \leq WMC \leq 350$</i>	<i>Low Severity</i>
<i>if $350 < WMC \leq 400$</i>	<i>Moderate Severity</i>
<i>if $400 < WMC$</i>	<i>High Severity</i>

3. Fan-In and Fan-Out

<i>if $Fanin < 20$</i>	<i>No Severity</i>
<i>if $20 \leq Fanin \leq 25$</i>	<i>Low Severity</i>
<i>if $25 < Fanin \leq 30$</i>	<i>Moderate Severity</i>
<i>if $30 < Fanin$</i>	<i>High Severity</i>
<i>if $Fanout < 20$</i>	<i>No Severity</i>
<i>if $20 \leq Fanout \leq 25$</i>	<i>Low Severity</i>
<i>if $25 < Fanout \leq 30$</i>	<i>Moderate Severity</i>
<i>if $30 < Fanout$</i>	<i>High Severity</i>

We created four levels of severity; we categorized them as High Severity (HS =3), Moderate Severity (MS=2), Low Severity (LS=1), and No Severity (NS=0). Consecutively severity was normalized to give us the ability to calculate the component priority.

Example: Normalizing and calculating the component priority for components X and Y.

Component X: LCC=HS=3, WMC=NS=0, Fan-in=LS=1 and Fan-out=NS=0.

Estimate component X priority

$$\begin{bmatrix} LCC \\ WMC \\ Fanin \\ Fanout \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} HS \\ MS \\ LS \\ NS \end{bmatrix} = \begin{bmatrix} HS \\ NS \\ LS \\ NS \end{bmatrix} \quad (1)$$

From (1), we can see that component X has very high severity related to LCC and low severity related to Fan-in. (XPriority =3+0+1+0=4).

Component Y: LCC=HS=2, WMC=NS=0, Fan-in=VHS=3 and Fan-out=VHS=3.

Estimate component Y priority

$$\begin{bmatrix} LCC \\ WMC \\ Fanin \\ Fanout \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} HS \\ MS \\ LS \\ NS \end{bmatrix} = \begin{bmatrix} MS \\ NS \\ HS \\ HS \end{bmatrix} \quad (2)$$

From (2), we can see that Component Y has very high severity related to Fan-in and Fan-out. Also, it has a high severity related to LCC. (YPriority = 2+0+3+3=8).

In general, the component priority will be a value from 0 to 12. The component risk will be classified into four categories.

<i>if Periority = 0</i>	<i>No risk</i>
<i>if 0 < Periority ≤ 4</i>	<i>Low risk</i>
<i>if 4 < Periority < 8</i>	<i>Moderate risk</i>
<i>if 8 ≤ Periority</i>	<i>High risk</i>

Priority = 0 (No risk) that means all the metrics (LCC, WMC, Fan-in, and Fan-out) are below the threshold. $0 < \text{Priority} \leq 4$ (Low risk) that means in the worst case (1) Two metrics are classified as moderate severity (2). At most, one metric is classified as high severity. $4 < \text{Priority} < 8$ (moderated risk) that means in the worst case at most (1) three metrics are classified as moderate severity (2) two metrics are classified as high severity (3) one metric as high severity and two as moderate severity. $8 \leq \text{Priority}$ (High risk) that means in the best case at least (1) All metrics are classified as moderate severity (2) Two metrics are classified as high severity and one as moderate severity.

STEP #3: (FIND THE BEST ML MODEL)

For every model, predicted performance was estimated by using 10-fold cross-validation. It has the best accuracy since its estimation is known to have less bias, as mentioned by Kohav [97]. The data was separated into ten segments (folds), and the first segment was tested, and the others were used as training data. The performance was calculated, and the same steps were repeated with all segments

7.3 Data Analysis and Results

Data analysis was performed using Weka version 3.8.3. Thirteen machine learning algorithms were used to detect architecture smells in software components. In addition, the ML algorithms were used to classify architecture smells impact on the internal structure of the software components. The ML inputs were extracted from four OSPPs from the GitHub repository. Table 7.2 shows what performance measures were taken to evaluate the correctness and performance of the applied machine learning models where:

- **Accuracy** indicates the best-detected machine learning model. The accuracy expresses the model for how it is correctly trained and how it performs.
- **ROC** stands for Receiver Operating Characteristics Curve. It evaluates the performance of the model by considering the true-positive and false-positive. It leaves out the true-negative and false-negative [98].
- **F-measure** is the sub-contrary mean of recall and precision where recall measures the percentage of total relevant results correctly classified by the machine learning models algorithm, while precision measures the percentage of our results that are relevant.

- **MCC** stands for Matthews Correlation Coefficient. It is the correlation coefficient between the predicted and the observed values. It uses a confusion matrix to estimate the model performance.

We applied the thirteen machine learning algorithms to the extracted internal structure metrics. By looking at the ROC values, we can determine if a ML model is a good model. The best performance measurement is the model accuracy, where it is the ratio of the correctly predicted observation to the total observations. From Table 7.2, Meta Random Committee has the highest accuracy. Our data shows that Trees Random Forest has the second-highest accuracy. On the other hand, Lazy LWL has the lowest accuracy between the thirteen models. It is worth noting that accuracy is a great measure if we have symmetric datasets. Our dataset is asymmetric. Consequently, F-measure and MCC are necessary for evaluating the performance of our models.

Table 7.2 Machine learning classifiers

Classifier	Accuracy	F-Measure	Roc Area	MCC
Meta Random Committee	97%	0.97	0.99	0.95
Trees Random Forest	96%	0.96	0.99	0.95
Trees J48	96%	0.96	0.97	0.93
Meta Classification Via Regression	94%	0.94	0.98	0.91
Rules PART	92%	0.92	0.97	0.89
Rules JRip	92%	0.92	0.96	0.88
Lazy IBK	86%	0.86	0.92	0.80
Lazy KStar	86%	0.85	0.98	0.79
Rules Decision Table	85%	0.85	0.96	0.79
Meta Multi Class Classifier	73%	0.72	0.89	0.61
Naive Bayes	71%	0.7	0.9	0.57
Rules OneR	65%	0.64	0.73	0.48
Lazy LWL	61%	0.53	0.84	0.37

By looking at the F-measure, Meta Random Committee has the highest F-measure value with 0.97. Trees Random Forest and Trees J48 have the second-highest F-measure values with 0.96. By looking at the Matthews Correlation Coefficient, Meta Random Committee and Trees Random Forest have the highest MCC values with 0.95. Trees J48 has the second-highest MCC value with 0.93. Table 7.3 displays more information for the best three machine learning models, which were used to classify component risk through multiple releases.

Table 7.3 Machine learning classifiers used to classify the component risk

Classifier	Risk	F-Measure	MCC
Meta Random Committee	High	0.987	0.985
	Moderated	0.947	0.931
	Low	0.969	0.944
	No	0.986	0.982
Trees Random Forest	High	0.987	0.985
	Moderated	0.933	0.914
	Low	0.962	0.932
	No	0.986	0.982
Trees J48	High	1	1
	Moderated	0.932	0.912
	Low	0.952	0.914
	No	0.972	0.964

Case Studies

We used the best three machine learning models to track ATD risk over multiple releases. Table 7.4 shows information about the releases that were used by the best three models for tracking ATD risk.

Table 7.4 Information about the projects used in the case studies

Project 1: OpenID, OAuth protocols			
Release	2.5.2.9007	3.4.3.10103	4.2.0.13024
KLOC	25.8	73.33	93
Classes	230	626	902
Methods	1217	2911	3709
Project 2: Xamarin-android			

Release	9.0.0.0	9.2.0.5	9.4.0.51
KLOC	72.26	77.4	80.7
Classes	733	752	797
Methods	3553	3721	3915
Project 3: Hawk			
Release	2.0	2.1	3.0
KLOC	19.9	20	24.75
Classes	244	243	282
Methods	1070	1072	1216
Project 4: Sharpcompress			
Release	0.9	0.10.3	0.11.3
KLOC	73.44	120.1	158
Classes	625	989	1327
Methods	3151	5397	6995

Figure 7.1 shows that the ATD risk in project number one was high in 5 components in the first and second releases, but in the third release, ATD risk was high in only 4 components. The ATD risk was moderated in the 4, 18, and 23 components while was low in the 8, 4, 18, and 23 components in releases 1, 2, and 3, respectively. The ATD risk in project two had nearly no change over the three releases. In project 3, ATD risk increased from low to moderate in 3 components, but only in one component was it high through all three releases. In project 4, the number of components that were classified as high risk increased from 6 to 9 to 13 through the three releases.

Figure 7.1 Information about the components risk for the four projects

Figures 7.2, 7.3, 7.4, and 7.5 show the ATD risk histories after tracking each component created in the 1st release through the 2nd and 3rd releases. The figure does not display the components that did not change. The Figures 7.2 helped us to find refactored components and the components where ATD risk increased. Developers in the first project refactored 9 components, (ID: From 4 to 12) while 3 components' ATD risk increased (ID: 1, 2, and 3). In the second project, 2 components were refactored (ID: 4 and 5), and the ATD risk was increased in 3 components (ID: 1, 2, and 3). The developers in the third and fourth projects did not refactor any component. On the other hand, the ATD risk increased in three and thirteen components in Project 3 and 4, respectively.

Figure 7.2 Tracking the level of risk – project 1

Figure 7.3 Tracking the level of risk – project 2

Figure 7.4 Tracking the level of risk – project 3

Figure 7.5 Tracking the level of risk – project 4

What does this mean to the decision-makers? Let us take project one as an example. Components 1, 2, and 3 need to be refactored because their ATD risk was increased through releases. However, if there is no time to refactor the three components, and since component 1 has the highest risk a decision regarding refactoring this component needed to be taken. On the other hand, developers refactored other components. Therefore, if project managers or decision-makers did not make these decisions, they could meet the developers to inquire about why they refactored these components and did not refactor those components? This procedure helps the corporation monitor refactoring activities. We conclude that tracking the ATD risk on software components helps the corporation to (1) Prioritize the refactoring decisions based on the level of the ATD risk and (2) Monitor the refactoring activities.

Our best three models were tested using 12 releases. The F-measure and MCC of each machine learning model compared using boxplots. Our datasets were unbalanced, and to evaluate our machine learning MCC and F-measure values were used. Figure 7.6 shows a boxplot MCC of the three machine learning models. We used MCC to estimate the models' performance because MCC is use True-positive, True-negative, False-positive, and False-negative. Taking in its account all the values in the Confusion Matrix helps MCC to evaluate the unbalanced dataset. Tress Random Forest has the best performance with 0.91, 0.84, and 0.93 MCC values for the median, first quarter, and third quarter, respectively.

Figure 7.6 MCC for the machine learning

Figure 7.7 shows the boxplot F-measure of the three machine learning models. We used F-measure to test our models' accuracy. Tress Random Forest has the best accuracy with 0.92, 0.88, and 0.95 F-measure values for the median, first quarter, and third quarter, respectively.

Figure 7.7 F-Measure for the machine learning

After testing the best three models, we found that the median performance of the Tress Random Forest model is 91%. We tested the accuracy of the Tress Random Forest, and we found that the median accuracy of the Tress Random Forest model is 92%. We conclude that the decision-makers can use machine learning techniques to classify the ATD risk which can help them take decisions related to refactoring ATD risk.

7.4 Threats to Validity

Construct Validity: We built models based on previous rules based Designite tool. The smells recognized depended upon metrics that were implemented by the Designite tool, which could have been a possible threat to validity because other smell detection tools could have detected a different number of smells for every type of smell in our studied projects.

Internal Validity: The rules that were used to detect and classify the architecture smells might identify smell instances that are not considered as such by a human expert. However, in the case of our work, the rules were built based on previous rules based Designite tool.

External Validity: This is related to the ability to generalize the results. Machine learning input came from analyzed data collected from open-source software projects, written in C#. Consequently, we cannot generalize the results.

Conclusion validity: Is related to our ability to draw correct conclusions. One threat to conclusion validity is in the datasets. The datasets were unbalanced and to evaluate our machine learning F-measure, and MCC values were used.

7.5 Conclusions

The results show that it is possible to use machine learning to classify the severity of the ATD in software components. Meta Random Committee, Trees Random Forest, and Trees J48 algorithms were able to classify ATD risk with extraordinary accuracy. Machine learning models were able to classify software components as no, low, moderate, and high risk, and this helped to prioritize performing the refactoring. We conducted multiple case studies by applying our best models to track the ATD risk through multiple releases. The models' F-measure and MCC values were reported to show the performance and accuracy of the models. Tracking the ATD risk helps decision-makers monitor the refactoring activities and make proper decisions related to ATD refactoring.

Chapter 8

Architecture Technical Debt -Based Risk Assessment

This Chapter explains a proposed methodology to assess ATD risk level in software components. Several steps of procedure are created to assess the ATD risk level, and the methodology was established using multiple case studies. For this work, the main motivation is that there is a need to develop a new method to help project managers make better decisions regarding ATD refactoring. Section 8.1 presents instructions for this Chapter. The proposed methodology is described in detail in Section 8.2. In Section 8.3, the methodology is demonstrated using multiple case studies. The data analysis, results, and discussion are detailed in Section 8.4. The threats to the validity of the method is deliberated in Section 8.5. Section 8.6 concludes the proposed methodology.

8.1 Introduction

Any software system can be recognized as several components and connectors. The connectors help the components to communicate with each other to accomplish their functions. Any problem with architecture can affect the developed system.

Architecture Technical Debt may occur due to unintended design decisions that violate the original architecture. When elements of an architecture violate modularity principles, architectural problems arise. Due to the lack of formal architecture documentation, developers frequently need to detect architecture problems in source code. Architecture smells such as God Component indicates maintenance problems because of complexity, this component (package) has reduced its understandability and maintainability. Developers can reduce the number of classes or LOCs in the God Component to reduce its complexity [99]. This change done by developers is known as refactoring. Refactoring improves the internal structure of the software without changing its external behavior [2]. Refactoring improves software quality and limits maintenance costs, but deadlines to release prevent developers from refactoring. Adding new features or fixing bugs instead of refactoring may be preferred by developers because refactoring does not provide immediate benefits [9]. Consequently, determining when to perform refactoring is important part of managing software development. Risk Assessment is also an important part of software management [100]. Using risk assessment, we can identify troublesome components and prioritize them by making the technical debt picture clear to the decision-makers to help them make better refactoring decisions.

In this Chapter, we propose a risk assessment method to deal with decisions related to ATD. We believe the ATD needs to be managed as a risk because:

- It is impossible to find the precise cost of refactoring.
- The decision-makers need to look at ATD from the risk perspective because refactoring does not provide immediate benefits.
- As we know, developers will face software components with ATDs, but which ATD has the highest risk? Risk management is a much more effective approach when dealing with many problems requiring prioritization because there is no time to solve them all [100].

Through tracking the architecture smells, we estimate their likelihood and impact (severity) on the components' internal structures. By estimating the likelihood and severity of the ASs, we will assess software components' ATD risk level.

8.2 Methodology

In this Section, a method for assessing the ATD risk level on software components will be proposed. Before using case studies, an example is presented to illustrate how the proposed approach can be applied to estimate the risk of the architecture technical debt in software components. Our methodology has four steps: (1) identify the architecture smell; (2) assess likelihood for each AS type; (3) estimate the impact of every AS instance; (4) estimate the ATD risk of each AS instance in the software components.

8.2.1 Identify the Architecture Technical Debt (ATD)

An established tool will be used to identify. To find such a tool, we reviewed the literature for papers relating to the identification of ATD. We found nine tools in our search that can identify the ATD, which include Designite, Arcan, Sonargraph, ARCADE, Structure 101, STAN, AI Reviewer, Hotspot Detector, and Massey Architecture Explorer. Because we need to use an available and free tool, we applied the exclusion criteria of (1) excluding the unavailable tools and (2) excluding tools that are not free. After we applied our excluded criteria, Designite and Arcan are the only tools that could be used.

However, we need to use a tool that can discover more architecture smells. Designite supports C#, and Java language and can identify seven architecture smell types, as presented in Table 5.1 (Chapter 5). Arcan supports Java and can identify only three architecture smell types. For the preceding reasons, we decided to use Designite.

8.2.2 Assessing the ATD risk level on software components

Risk Assessment is used to manage software development projects. "According to the NASA-STD-8719.13A standard, the risk is a function of the anticipated frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity" [100]. Risk assessment and risk control are two steps to manage the risk [101]. Using our methodology, we will assess the risk of the architecture technical debt

on the software components. In order to estimate the risk, we need to identify the risk, analyze it, and prioritize it. In their work, Yacoub and Ammar [102] adopted the risk definition from “NASA’s Definition of Risk Matrix,” which defines risk as a combination of two factors: the probability of malfunctioning (failure) and the consequence of malfunctioning (severity).

We define risk as a combination of two factors. The first factor is the probability: The likelihood of occurrence of the ASs. The second factor is the severity: The impact of the ASs on the internal structure of the classes causing the ASs on the software components. The ATD Risk can be estimated by using equation (1).

$$ATD_{Risk} = ATD_{Probability} \times ATD_{Impact} \dots \dots \dots (1)$$

Where ATD is CD,, UD, AI, GC, FC, SF, DS

Where CD, UD, AI, GC, FC, SF, and DS are the abbreviation of the Cyclic Dependency, Unstable Dependency, Ambiguous Interface, God Component, Feature Concentration, Scattered Functionality, and Dense Structure.

The $ATD_{Probability}$ is the likelihood of occurrence for an architecture smell in software components. The likelihood is the percentage chance that a type of architecture smell occurs, and can be estimated using equation (2).

$$ATD_{Probability} = \frac{Number\ of\ AS\ Type}{Number\ of\ All\ AS} \dots \dots \dots (2)$$

In order to estimate the ATD_{Impact} (severity) on the software components, a tool was developed to extract the internal structure metrics for the classes causing the architecture smells. Table 8.1 shows the Internal Structure Metrics (ISMs) that we will use to estimate the severity of every architecture smell in the software components.

Table 8.1 Internal Structure Metrics used to assess the AS severity

ISMs	Definition
NOC	The total number of classes caused the architecture smell (AS).
NOM	The total number of methods in the class caused AS.
LOC	The total number line effected by the AS.
NC	The total number of children in the class caused AS.
WMC	The sum of Cyclomatic Complexities of all the methods belonging to the class caused AS.
DIT	It is the maximum inheritance path from the class caused AS to the root class.
Fan-in	The total number of classes that reference the class caused AS.
Fan-out	The total number of classes referenced by the class caused AS.

To estimate the impact of every architecture smell instance on the software components, we need to apply those steps:

1. Extract the Total Internal Structure (TIS) of every project release. TIS_{ISM_I} is estimated as shown in equation (3). For instance TIS_{LOC} can be obtained by summing up all number lines of code in this project release.

$$TIS_{ISM_I} = \sum ISM_I \dots \dots \dots (3)$$

Where ISM_I is $NOC, NOM, LOC, NC, WMC, DIT, Fanin, Fanout$

2. Estimate the ISM for every architecture smell instance. ATD_{ISM_I} is estimated as shown in equation (4). For instance CD_{LOC} for that CD instance can be obtained by summing up all number line of code affected by this CD instance $CD_{LOC} = \sum LOC$.

$$ATD_{ISM_I} = \sum ISM_I \dots \dots \dots (4)$$

Where ATD is $CD, UD, AI, GC, FC, SF, DS$

3. Assess the severity of each architecture smell instance on each internal structure metric by finding the percentage of the internal structure metric affected by that architecture smell instance. ATD_{Impact} is estimated as shown in equation (5). For instance, Impact of the CD_{Impact} for that instance on the software based on LOC metric is $CD_{Impact_{LOC}} = \frac{CD_{LOC}}{\sum loc}$.

$$ATD_{Impact} = \sum ATD_{Impact_{ISM_I}} = \sum \frac{ATD_{ISM_I}}{TIS_{ISM_I}} \dots \dots \dots (5)$$

Where $ATD_{Impact_{ISM_I}}$ is

$$ATD_{Impact_{NOC}}, UD_{Impact_{NOM}}, AI_{Impact_{LOC}}, GC_{Impact_{NC}}, FC_{Impact_{WMC}}, \\ SF_{Impact_{DIT}}, DS_{Impact_{Fanin}}, DS_{Impact_{Fanout}}$$

4. After estimating $ATD_{Probability}$ and ATD_{Impact} for each architecture smell instance on the software component, the ATD_{Risk} of every architecture smell can be estimated by using equation number (1). For instance, the severity of the CD1 is equal to:

$$CD1_{Risk} = CD_{Probability} \times CD1_{Impact}$$

5. The ATD risk on the software component $Component_{ATD_{Risk}}$ can be easily estimated as showed in equation 6 by adding up all the ATD_{Risk} that occurred in this component.

$$Component_{ATD_{Risk}} = \sum_{ATD=CD}^{DS} ATD_{Risk} \dots \dots \dots (6)$$

8.2.3 An example explains the proposed methodology

We will give an example to explain how we will estimate the ATD risk on software components for one release. The same explained steps in this example will be applied automatically for each release. One can easily estimate the probability for each architecture smell by calculating the fraction of every AS type. For instance, suppose that only three types of architecture smells (Cyclic Dependency, Scattered Functionality, and God Component) occurred. If there are ten architecture smell instances in this project release, Cyclic Dependency occurred five times. The Cyclic Dependency likelihood will be estimated by using equation (2).

$$CD_{Probability} = \frac{5}{10} = 0.5$$

The same will be applied to other architecture smell types to assess their likelihood. Scattered Functionality and God Component occurred three and two times, respectively.

$$SF_{Probability} = \frac{3}{10} = 0.3 \text{ and } GC_{Probability} = \frac{2}{10} = 0.2$$

To assess the severity (impact) of every architecture smell instance, we apply these steps:

1. Extract the total number of the internal structure values (TIS) of the software. After applying equation 3, Table 8.2 shows the TIS values for this example. $TIS_{ISM_I} = \sum ISM_I$

Table 8.2 Total number of the Internal Structure (Example)

NOC	NOM	LOC	WMC	NC	DIT	Fan-Out	Fan-In
202	300	62,397	1302	712	57	421	369

2. Extract the values of the Internal Structure Metrics for all classes caused the architecture smell. After applying equation (4), Table 8.3 presents the real values for every ISM. $ATD_{ISM_I} = \sum ISM_I$.

Table 8.3 Internal Structure Metrics (ISM) for every architecture smell instance (Example)

Comp	Architecture smell Type	AS ID	NOC	NOM	LOC	WMC	NC	DIT	Fan Out	Fan In
A	Cyclic Dependency	1	14	57	691	102	94	2	38	40
	Cyclic Dependency	2	18	64	932	111	70	2	28	81
	Cyclic Dependency	3	23	75	1186	133	23	1	46	6
	Scattered Functionality	4	15	34	485	66	14	1	24	12
B	God Component	5	6	32	341	56	12	1	5	6
	Cyclic Dependency	6	3	5	306	5	7	3	9	8
C	Cyclic Dependency	7	2	4	54	6	7	1	4	7
	Scattered Functionality	8	13	13	246	29	5	1	22	5
	Scattered Functionality	9	11	22	673	83	5	3	54	8
D	God Component	10	7	1	21	1	5	1	2	6

To assess every architecture smell instance's severity, we need to normalize the ISM values by applying equation 5, as seen in Table 8.4. We divided every ISM value in Table 8.3 by the total value in Table 8.2. For instance, in this example, the project release has 202 classes. CD1 impacted 7% (14/202) of the classes in this project release. However, Dependency with AS-ID = 2 has the highest impact (0.82).

$$ATD_{Impact} = \sum ATD_{Impact_{ISM_I}} = \sum \frac{ATD_{ISM_I}}{TIS_{ISM_I}}$$

Table 8.4 Assess the ATD Impact (Severity) (Example)

Comp	AS ID	AS Type	NOC	NOM	LOC	WMC	NC	DIT	Fan Out	Fan In	ATD Impact
A	1	CD	0.07	0.19	0.01	0.08	0.13	0.04	0.09	0.11	0.71
	2	CD	0.09	0.21	0.01	0.09	0.10	0.04	0.07	0.22	0.82
	3	CD	0.11	0.25	0.02	0.10	0.03	0.02	0.11	0.02	0.66
	4	SF	0.07	0.11	0.01	0.05	0.02	0.02	0.06	0.03	0.37
B	5	GC	0.03	0.11	0.01	0.04	0.02	0.02	0.01	0.02	0.25
	6	CD	0.01	0.02	0.00	0.00	0.01	0.05	0.02	0.02	0.15
C	7	CD	0.01	0.01	0.00	0.00	0.01	0.02	0.01	0.02	0.08
	8	SF	0.06	0.04	0.00	0.02	0.01	0.02	0.05	0.01	0.22
	9	SF	0.05	0.07	0.01	0.06	0.01	0.05	0.13	0.02	0.41
D	10	GC	0.03	0.00	0.00	0.00	0.01	0.02	0.00	0.02	0.08

Table 8.5 shows the ATD risk for every architecture smell instance. The risk for a particular architecture smell instances is equal to the likelihood of this architecture smell type's occurrence multiplied by the impact of this specific instance (equation 1). The ATD risk for ATD risk on the software component ($Component_{ATDRisk}$) can be directly estimated by using equation 6.

$$Component_{ATDRisk} = \sum_{ATD=CD}^{DS} ATDRisk$$

$$\begin{aligned}
 Component(A)_{ATDRisk} &= \sum_{ATD=CD}^{DS} ATDRisk = CD_{Risk} + CD_{Risk} + CD_{Risk} + SF_{Risk} \\
 &= 0.36 + 0.41 + 0.33 + 0.11 = 1.21
 \end{aligned}$$

$$Component(C)_{ATD_{Risk}} = \sum_{ATD=CD}^{DS} ATD_{Risk} = GD_{Risk} + CD_{Risk} = 0.05 + 0.07 = 0.12$$

$$\begin{aligned} Component(C)_{ATD_{Risk}} &= \sum_{ATD=CD}^{DS} ATD_{Risk} = CD_{Risk} + SF_{Risk} + SF_{Risk} = 0.04 + 0.07 + 0.12 \\ &= 0.23 \end{aligned}$$

$$Component(D)_{ATD_{Risk}} = \sum_{ATD=CD}^{DS} GC_{Risk} = 0.02$$

Table 8.5 Assess the AS Risk (Example)

Component	AS ID	AS Type	AS Risk (Impact x Probability)
A	1	Cyclic Dependency	0.36
	2	Cyclic Dependency	0.41
	3	Cyclic Dependency	0.33
	4	Scattered Functionality	0.11
B	5	God Component	0.05
	6	Cyclic Dependency	0.07
C	7	Cyclic Dependency	0.04
	8	Scattered Functionality	0.07
	9	Scattered Functionality	0.12
D	10	God Component	0.02
Likelihood: Cyclic Dependency =0.5,Scattered Functionality =0.3,God Component = 0.2			

8.3 Case Studies

In this Section, multiple case studies are presented to verify our methodology. Exactly, 40 C# open source project releases were downloaded from the GitHub. All the project releases were analyzed, and their statistical characteristics are reported in Table 8.6. We applied the described

risk methodology in Section 8.2 to estimate the ATD risk of all architecture smell instances and to answer the following questions:

RQ1: Which type of architecture smell has the highest probability of occurring in the software components?

RQ2: Which type of architecture smell has the highest impact on the software components?

RQ3: Which type of architecture smell has the highest risk in the software components?

Table 8.6 Statistical characteristics of the selected software projects

Project name: Simple.Data - Project ID: 1				
Release ID	1	2	3	4
Release	0.6.3	0.12.2	1.0.0-beta6	2.0.0-alpha1
KLOC	18	29	34	37
Classes	245	417	451	508
Methods	1263	2282	2643	2814
Project name: ILRuntime- Project ID: 2				
Release ID	1	2	3	4
Release	0.8	1.0	1.2	1.4
KLOC	18	33	38	49
Classes	181	230	354	444
Methods	851	1445	1728	2090
Project name: Hawk - Project ID: 3				
Release ID	1	2	3	4
Release	2.0	2.1	3.0	5.0
KLOC	19.9	20	24.75	27.51
Classes	244	243	282	301
Methods	1070	1072	1216	1315
Project name: ZeroFormatter - Project ID: 4				
Release ID	1	2	3	4

Release	1.0.0	1.3.0	1.5.0	1.6.0
KLOC	22	28	30	30.2
Classes	328	391	454	484
Methods	1163	1350	1398	1422
Project name: Xamarin-android - Project ID: 5				
Release ID	1	2	3	4
Release	9.0.0.0	9.1.5.1	9.2.0.5	9.4.0.51
KLOC	72.26	74	77.4	80.7
Classes	733	742	752	797
Methods	3553	3594	3721	3915
Project name: OpenID, OAuth protocols - Project ID: 6				
Release ID	1	2	3	4
Release	2.5.2.9007	3.4.3.10103	4.0.012030	4.2.0.13024
KLOC	25.8	73.33	85.2	93
Classes	230	626	790	902
Methods	1217	2911	3396	3709
Project name: OpenRA - Project ID : 7				
Release ID	1	2	3	4
Release	20150614	20160508	20180307	20190314
KLOC	109.7	126	148	162.3
Classes	1866	2045	2339	2569
Methods	6054	6686	7831	8417
Project name: Sharpcompress - Project ID: 8				
Release ID	1	2	3	4
Release	0.9	0.10.3	0.11.3	0.11.6
KLOC	73.44	120.1	158	162.5
Classes	625	989	1327	1344
Methods	3151	5397	6995	7162
Project name: NRefactory - Project ID: 9				

Release ID	1	2	3	4
Release	5.0.1	5.2.0	5.4.0	5.5.0
KLOC	232.2	242.7	280.7	383.7
Classes	1631	1693	2541	2549
Methods	12138	12754	18854	18964
Project name: UnityCsReference - Project ID: 10				
Release ID	1	2	3	4
Release	2017.1.0a3	2018.1.0b12	2019.2.0a9	2019.3.0f4
KLOC	409	501.86	615.25	658
Classes	3551	5283	6748	7164
Methods	25385	31830	39332	42521

8.4 Data Analysis, Results, and Discussion

More than 5.4 million LOC, 54 thousand classes, and 3 million methods were analyzed to assess the architecture smells risk on software components. Table 8.7 shows the 5,183 architecture smell instances were detected in the forty releases. Appendix C has more information about the used C# open sources.

Table 8.7 Type and number of the identified Architecture smells in the 40 project releases

Ambiguous Interface	Cyclic Dependency	Dense Structure	Feature Concentration	God Component	Scattered Functionality	Unstable Dependency
24	2274	24	916	298	1072	575

Figure 8.1 Number of architecture smells for the studied projects

8.4.1 Architecture Smell Likelihood

RQ1: Which type of architecture smell has the highest likelihood of occurring in the software components?

Motivation: From Chapter 5 and Chapter 6, we noticed that the architecture smells have a different likelihood. Therefore, we expect to assess the likelihood of the seven types of architecture smells in the chosen projects.

Approach: To answer this question, we will (1) use a tool to track every architecture smell instance. (2) We need to write a script to go through the information saved as comma-separated values (CSV) file with the tool to measure the likelihood of every architecture smell type. (3) The results of the architecture smell likelihood will be presented per project-release in a table. (4) The mean, median, min, max, and standard deviation will be reported in the same table.

Results: Table 8.8 shows the likelihood of each architecture smell type in the software components. The Cyclic Dependency had the highest probability in 30 (75%) project releases. In 7 (17.5%) project releases, Feature Concentration had the highest probability. Scattered Functionality had the highest likelihood in 2 project releases. Scattered Functionality and Cyclic

Dependency shared the highest probability in one project's release. In general, we can see that Cyclic Dependency had the uppermost average, median, and maximum likelihood of appearance. Cyclic Dependency had a minimum probability (0.05), and it was only in two project releases while it had the second probability in 6 project releases. Feature Concentration had the second uppermost average and the maximum likelihood of appearance and shared the second uppermost median likelihood with Scattered Functionality, which had the third, uppermost average and the maximum likelihood of appearance.

Table 8.8 The likelihood of the architecture smells per release

Project ID	Release ID	Unstable Dependency	Scattered Functionality	God Component	Feature Concentration	Dense Structure	Cyclic Dependency	Ambiguous Interface
1	1	0.14	0.10	0.10	0.37	0	0.29	0
	2	0.10	0.10	0.10	0.48	0	0.21	0.01
	3	0.12	0.12	0.12	0.44	0	0.19	0.01
	4	0.10	0.10	0.12	0.32	0	0.36	0
2	1	0.13	0.11	0.04	0.13	0.02	0.57	0
	2	0.13	0.19	0.04	0.09	0.02	0.53	0
	3	0.12	0.20	0.05	0.14	0.02	0.47	0
	4	0.20	0.10	0.06	0.07	0.01	0.56	0
3	1	0.14	0.17	0	0.16	0	0.53	0
	2	0.14	0.17	0	0.14	0	0.55	0
	3	0.13	0.02	0.06	0.23	0.02	0.54	0
	4	0.12	0.02	0.06	0.19	0.02	0.59	0
4	1	0.15	0.20	0.15	0.45	0	0.05	0
	2	0.10	0.20	0.15	0.50	0	0.05	0
	3	0.07	0.15	0.19	0.40	0	0.19	0
	4	0.11	0.07	0.22	0.41	0	0.19	0
5	1	0.21	0.05	0.08	0.19	0	0.45	0.02
	2	0.20	0.05	0.08	0.19	0	0.47	0.01
	3	0.19	0.04	0.07	0.18	0	0.49	0.03
	4	0.19	0.04	0.09	0.19	0	0.48	0.01

6	1	0.16	0.13	0.03	0.11	0.02	0.54	0.01
	2	0.12	0.19	0.03	0.14	0.01	0.51	0
	3	0.13	0.28	0.02	0.17	0.01	0.39	0
	4	0.15	0.27	0.02	0.18	0.01	0.36	0.01
7	1	0.10	0.23	0.07	0.14	0.01	0.45	0
	2	0.11	0.20	0.06	0.12	0	0.50	0.01
	3	0.10	0.20	0.07	0.14	0	0.48	0.01
	4	0.10	0.22	0.06	0.14	0	0.46	0.02
8	1	0.14	0.08	0.02	0.19	0	0.57	0
	2	0.12	0.12	0.02	0.29	0	0.45	0
	3	0.10	0.10	0.02	0.35	0	0.43	0
	4	0.10	0.10	0.02	0.36	0	0.42	0
9	1	0.13	0.27	0.09	0.13	0.01	0.35	0.02
	2	0.12	0.33	0.11	0.11	0.01	0.33	0
	3	0.10	0.46	0.11	0.10	0.01	0.21	0.01
	4	0.10	0.46	0.11	0.10	0.01	0.21	0.01
10	1	0.10	0.24	0.03	0.14	0	0.46	0.03
	2	0.08	0.23	0.05	0.16	0	0.47	0.01
	3	0.08	0.27	0.07	0.14	0	0.44	0
	4	0.08	0.27	0.06	0.12	0	0.46	0.01
Median		0.12	0.17	0.06	0.17	0	0.46	0
Average		0.13	0.17	0.07	0.22	0.01	0.41	0.01
MIN		0.07	0.02	0	0.07	0	0.05	0
MAX		0.21	0.46	0.22	0.50	0.02	0.59	0.03
STD		0.03	0.10	0.05	0.12	0.01	0.14	0.01

We tested the correlation between the different architecture smells probabilities using the non-parametric Spearman correlation. As it appears in Figure 8.2: First, a high negative correlation between Cyclic Dependency and God Components ($r = -0.77$) has been detected. Second, a highly negative correlation had been found between Cyclic Dependency and Feature Concentration ($r = -0.68$). Third, a moderate positive correlation has been detected between God Component and Feature Concentration ($r = 0.53$). Fourth, Unstable Dependency and Scatter Functionality had a moderate negative correlation ($r = -0.46$).

Figure 8.2 The spearman correlation test between AS likelihood of occurrences

8.4.2 Architecture Smell Impact

RQ2: Which type of architecture smell has the highest impact on the software components?

Motivation: We noticed that each architecture smell has a different impact on the software component from the participants' perspective in our survey (Chapter 6). Therefore, we expect to assess their real impacts on the internal structure of the software components.

Approach: To answer this question, we will (1) use a tool to extract the internal structure of the classes caused the architecture smell; (2) we will write a script to go through that information

to assess the impact of every architecture smell type; (3) To analyze the results we present every AS type in the graph that shows the normalized impact (0:1) and the impact classification. The impact value will be classified into five levels: 10% Very High (VH), 20% High (H), 40% Moderate (M), 20% Low (L), 10% Very Low (VL), as seen in Figure 8.3

Figure 8.3 Five level classification for the impact of the architecture smells

Results: Figure 8.4 to Figure 8.10 show the architecture smells' severity (impact). Figure 8.4 shows the impacts of all the Unstable Dependency (575 instances in 40 project releases) and the classification of the UD impact. We notice that only 44 (7.65%) UD instances had a Very High impact. 55 (9.57%) UD instances were classified as having a High impact. Exactly 163 and 172 UD instances (28.35% and 29.91%) were classified as Moderated and Low impact, respectively. 141 UD instances (24.52%) were classified as Very Low impact.

Figure 8.4 Impact of the Unstable Dependency and Classification

Figure 8.5 Impact of the Scattered Functionality and Classification

Figure 8.5 shows the impacts of all the Scattered Functionality, (1,072 instances in 40 project releases) and the SF impact classification. Around 4.57% (49 instances) of the SF instances had a Very High impact. Some 59 SF instances (5.50%) were classified as having a High impact. Most of the Scattered Functionality instances (470 and 394 instances) were classified as Moderated

(43.84%) and Low impact (36.75%), respectively. Exactly 100 SF instances (9.33%) were classified as Very Low impact.

Figure 8.6 displays the impacts of all the God Component smells, (298 instances in 40 project releases) and the classification of the GC impact. We notice most of the God Component smells 115 GC instances (38.59%) had a Very High impact, and 28 GC instances (9.40%) were classified as High impact. Exactly, 83 and 42 GC instances (27.85% and 14.09%) respectively were classified as Moderate and Low impact. Last and least 30 (10.07%) of the GC instances were classified as Very Low impact.

Figure 8.6 Impact of the God Component and Classification

The impact and the impact classifications of 916 Future Concentration instances are presented in Figure 8.7. The expensive cases High and Very High impact amounted to 10.81% and 13.43% (99 and 123 instances) respectively of the Future Concentration smell. The most numerous 341 (37.23%) FC instances were classified as having Moderate impact. The least expensive Low impact FC instances amounted to 188 (20.52%). Finally 18.01 % of the FC instances (165 instances) were classified as Very Low impact.

Figure 8.7 Impact of Future Concentration and Classification

Figure 8.8 Impact of Dense Structures and Classification

The Dense Structure instance's impacts and the impact classification of 24 DS instances are offered in Figure 8.8. Almost all the Dense Structure smell instances (22 instances) were Very High, while the other two remaining DS instances were classified as Moderate and Low impact respectively. On the other hand, zero FC instances were classified as a High or Very Low impact.

Figure 8.9 Impact of Cyclic Dependency and Classification

Figure 8.9 shows the impact and the impact classification of all the Cyclic Dependency (2274 instances in 40 project release). We found 188 CD instances (8.27%) had a Very High impact. Most of the Cyclic Dependency smells (769 and 1025 instances) were classified as High (33.82%) and Moderate (45.07%) impact. Approximately 10.3% and 2.6% (233 and 59 instances) of the CD instances were classified as Low and Very Low impact.

Figure 8.10 Impact of Ambiguous Interface and Classification

The Ambiguous Interface instance's impacts and the impact classification of 24 AI instances are offered in Figure 8.10. Almost all the AI instances (22 instances) had a Very Low classification. The other AI instances were indicated as Low impact.

To answer our RQ2, which is “Which type of AS has the highest impact on the software components?” we merged the Very High and High impact level and plotted in Figure 8.11.

Figure 8.11 Architecture Smells with the highest impact

We compared the percentages of the (VH/H) impacts between the seven architecture smells types to find which one had the highest impact. The rare Dense Structure occurred 24 times (0.5%) from 5183 architecture smell instances that occurred in 40 project releases. Most of the DS instances (around 92%) had VH/H impact on the software component. Almost half of the God Component (48%) had a VH/H impact on the software components. Cyclic Dependency had the third-highest impact, where around 42% of the Cyclic Dependency had VH/H impact. We can conclude that Dense Structure, God Component, and Cyclic Dependency smells have the highest impact on the software components, while Ambiguous Interface has no impact.

8.4.3 Architecture Smell Risk

RQ3: Which type of architecture smell has the highest risk in the software components?

Motivation: Assessing the risk of every architecture smell instance can help in prioritizing the refactoring decisions based on risk levels. We believe we can assess ATD risk after using the assessed likelihood, and the severity of the architecture smells on the software components.

Approach: To answer this question, we will use the use likelihood assessed in RQ1 and the impact calculated in RQ2. The results will be analyzed by using a Scatter. The values of the risk of architecture smell instances will be classified into five levels: 10% Very High Risk (VH), 20% High Risk (H), 40% Moderate Risk (M), 20% Low Risk (L), 10% Very Low Risk (VL).

Results: Figure 8.12 to Figure 8.18 show the architecture smells' Risk (Likelihood x Impact). Figure 8.12 shows the risk of the 575 Unstable Dependency and the classification of the UD risk. Around 3% and 3.8 % (18 and 22 instances) of the UD instances had a Very High and High risk, respectively. Exactly 131 of the UD instances were classified as a Moderate risk. The largest number, 40.5% of UD instances had Low impact. Approximately 30% of the UD instances (171 instances) were classified as Very Low impact.

Figure 8.12 Risk of the Unstable Dependency and Classification

Figure 8.13 shows the risk of 1,072 Scattered Functionality instances and the classification of their risk. Around 3.45% (37 instances) of the SF instances had very high risk. Exactly 60 (5.6%) SF instances had a High risk. Most of the Scattered Functionality instances (479 and 398 instances) had either Moderate (44.68%) or Low (37.13%) risk respectively. Slightly over 9% (98 instances) of the SF were Very Low risk.

Figure 8.13 Risk of the Scattered Functionality and Risk Classification

Figure 8.14 Risk of the God Component and Risk Classification

Figure 8.14 shows the risk and the risk classification of all the God Component smells (298 instances) that were detected in the 40 project-releases. Around 13.09 % and 12.42% (39 and 37 instances) of the God Components smells had Very High and High risk respectively. Most of the GC instances (39.26%, 117 instances) had Moderate risk. Last 50 and 55 GC instances (16.78% and 18.46%) had Low or Very Low risk.

Figure 8.15 Risk of the Feature Concentration and Risk Classification

The risks and the risk classification of the detected 916 Future Concentration instances are presented in Figure 8.15. Around 6.67% and 9.51 % (61 and 87) of the FC instances had Very High or High impact respectively. The greatest quantity 380 (41.53%) FC instances had Moderate impact. Low impact accounted for 235. (25.68%), and 152 (16.61%) FC instances had a Very Low impact.

Figure 8.16 shows the Dense Structure instances' risk and the risk classification. Of the 24 distinguished DS instances in the 40 project-releases just four Dense Structure smell instances had Very High risk and High risk (2 instances, 8.33% for each class). Most of the DS instances had Moderate or Low risk (9 instances, 37.5% for each class). Two FC instances were classified as having Very Low risk. As it appears in Figure 8.16, most of the DS instances had VH impact but on average, the DS smell has 0.01% likely to happen.

Figure 8.16 Risk of the Dense Structure and Risk Classification

The risk and risk classification of 2,274 detected Cyclic Dependency smells are presented in Figure 8.17. The most expensive CD instances 360 (15.83%) and 826 (36.32%) were Very High or High risk respectively. Most of the Cyclic Dependency smells (967 instances, 42.52%) were classified as a Moderate risk. CD instances were classified as Low or Very Low risk in 103 and 18 instances (4.53% and 0.79%) respectively.

Figure 8.17 Risk of the Cyclic Dependency and Risk Classification

The Ambiguous Interface instances' risk and the risk classification of 24 identified AI instances are available in Figure 8.18. Almost all the AI instances (22 instances) were Very Low risk. In addition, the other AI instances were indicated to be Low risk.

Figure 8.18 Risk of the Ambiguous Interface and Risk Classification

RQ3: Which type of architecture smell has the highest risk in the software components?

To answer this, we will compare the merged Very High and High classifications. The results of the merged two classes are plotted in Figure 8.19.

Figure 8.19 Architecture Smells with Very High/High Risk

In comparing the percentage of the (VH/H) risks between the seven architecture smells types we observe some patterns. We notice that more than half of the CD instances, 1,186 (52%) had at least High risk in the software components. From 298 detected GC instances, a quarter of them (76 GC instances) have a VH/H risk on the software components. From this, we can conclude that Cyclic Dependency smells and God Component are the two architecture smells with the highest risk in the software components. The risk of the architecture smells on the software component can be easily assessed by adding up all the risk value for all the AS in this component. The AS Risk on the components (packages) will be assessed, and the methodology will be validated in Chapter 9 by applying the proposed methodology to assess the risk of the architecture smell per package and compare the results with related works. We will use a dataset (Java projects) collected by Lenarduzzi et al. [103] and used by Samir Deeb et al. “*Estimating Refactoring Efforts for Architecture Technical Debt*” Deeb’s work has been accepted on Oct 19, 2020 for publication in the International Journal of Software Engineering and Knowledge Engineering.

We tested the correlation between the architecture smells probability, impact, and risk. We need to see which factor (Probability or Impact) is more correlated with risk. We used the non-parametric Spearman correlation. We found that AS impact is more correlated with AS risk. A high positive correlation between the impact of the architecture smell and risk of the architecture smell ($r = 0.80$, $p\text{-value} = 0$) was found, while the correlation between the likelihood and the risk of the architecture smell was only 0.43 ($p\text{-value} = 0$).

The main advantages of the assessing ATD risk technique proposed in this new methodology are three-fold. First tracking every architecture smell instance to estimate the likelihood of occurrence for every AS in the software components. Second by estimating the AS impact on the internal structure, which is a sound and objective approach for measuring quality level and an important factor in creating ATD risk. Assessing the risk of ATD on software components helps the decision-makers assign a priority to refactoring decisions based on level of risk. The second advantage is that assessing ATD risk is based on an empirical approach, which provides confidence in the estimates’ reliability. The last advantage is that the proposed methodology does not rely on assumptions and does not require many inputs.

The main contribution of this Chapter represents a novel methodology to assess the risk of ATD. The new method shows the ability to assess the risk of the architecture technical debt on

software components by tracking the AS instances to estimate the likelihood of each AS type and severity of AS instances.

8.5 Threats to Validity

Construct Validity: We used Designite tool to detect architecture smells to avoid subjective bias. In order to eliminate any ambiguity that may face the reader types of architecture smell have been defined.

Internal Validity: Requires the ability to track the architecture smell through multiple releases. The developed tool tracked architecture smells to estimate their probability severity. In order to ensure quality randomly selected results were compared manually.

External Validity: We did not use different programming languages, and we do not include commercial software projects. Consequently, the results cannot be generalized to other languages such as Java. They could be generalized to C# Open source projects after other researchers applied the same methodology using other C# projects provided they get the same results.

Conclusion validity: Is related to our ability to draw correct conclusions. We studied forty releases from different project sizes: small, medium, and big projects, and the project's releases were randomly selected.

8.6 Conclusions

The proposed methodology aims to assess the risks of architecture technical debt in software components to help the decision-maker make a valid decisions about ATD refactoring. Forty software releases with more than 5.4 million LOC, 54 thousand classes, and 3 million methods were analyzed to assess architecture smell risks on software components. More than 5,179 architecture smells instances have been tracked. The applied case studies' results allowed us to infer the following: In C# programming language, (1) Cyclic Dependency and Feature Concentration have the first and second highest probability of occurrence. (2) Dense Structure, God Component, and Cyclic Dependency have the highest impact on the software components' internal structure. (3) Cyclic Dependency and God Component have the highest risk on software components.

We studied the correlation between the likelihoods of the existence of different architecture smells. A high negative correlation between Cyclic Dependency and God Component, and between Cyclic Dependency and Feature Concentration have been found. A positive moderate correlation between God Component and Feature Concentration exists while a negative moderate correlation between Unstable Dependency and Scatter Functionality has been identified.

As we proposed, the risk of any AS instance can be estimated by finding the likelihood and the impact of this AS. For that reason, we tested the correlation between the architecture smell risk and the probability and the impact of the architecture smells. We found the impact is more correlated to the architecture smell risk than to the likelihood.

Chapter 9

Methodology Validation

This Chapter validates our proposed risk methodology. In order to validate the results of our methodology that based on assessing the architecture technical debt risk level we compare the methodology results with the results of related studies. This Chapter is structured as follows. Section 9.1 explains the related works' methods. Section 9.2 describes the dataset used to validate our approach. The verification of our methodology and the results validating it will be presented in Section 9.3. Section 9.4 discusses threats to the validity of the methodology validation. In section 9.5 we give our conclusions for this Chapter.

9.1 Description of Related Works Methodologies

9.1.1 Prioritization of classes for refactoring: A step towards improvement in software quality

Malhotra et al [104] used quality to prioritize the refactoring. The software's quality is estimated using two factors, namely bad smell and Chidamber and Kemerer (C&K) Metrics. They assigned equal weight to the two factors. After that, they estimated the quality of the class by finding the quality index, namely Quality Depreciation Index Rule (QDIR), which is calculated according to equation (1)

$$\text{Quality Depreciation Index Rule (QDIR)} = \frac{BoB}{2} + \frac{BoM}{2} \dots \dots \dots (1)$$

The larger value of QDIR indicates that the class should be given high priority for the refactoring as it contains critical design flaws.

Calculation of Base of Bad Smell (BoB)

They identified four types of bad smells: God Class, Long Method, Type Checking, and Feature Envy. Table 9.1 describes the four bad smells as had been defined by this related work.

Table 9.1 Code smells used by Malhotra

Bad Smell	Definition
God Class	A class tends to centralize the system intelligence.
Long Method	A method that is long, which makes it difficult to modify and understand.
Type Checking	The function is split into multiple functions.
Feature Envy	A method is interested more in other class than the one where it currently located

They assigned equal smell weight to each smell (0.25). The BoB can be estimated using equation (2)

$$Base\ of\ Bad\ Smell\ (BoB) = \frac{1}{4} \sum_{i=1}^4 (Priority_{badsmell}) \dots \dots \dots (2)$$

Where $Priority_{badsmell}$ is $Priority_{GodClass, LongMethod, FeatureEnvy, and LongMethod}$

Calculation of Base Metric (BoM)

Malhotra used six C&K Metrics, which are described in Table 9.2.

Table 9.2 Chidamber and Kemerer Metrics as defined by Malhotra

Metric	Definition
Weighted Methods per Class (WMC)	It is the total number of methods in a particular class
Response for a Class (RFC)	It is the total number of the methods that are executed in response to any message received by any instance of that class
Coupling between Objects (CBO)	It is the total number of dissimilar classes that are not inheritance related and with which a given class is coupled
Lack of Cohesion of Method (LOCM)	It is all probable pairs of a class method not shared by any probable pairs of class methods that share instances.
Depth of Inheritance Tree (DIT)	It is the longest path length between the root node and given node
Number of Children (NOC)	It is the total number of classes that inherit the given class.

To estimate the BoM, they need to assess each metric (Metric Value). The Metric Value (MV) is estimated as it appears in equation (3). The BoM is estimated as presented in equation (4) where the MV is the average of all six C&K metrics.

$$Metric\ Value\ (MV) = \frac{Calculated_{Metric}}{Threshold_{Metric}} \dots \dots \dots (3)$$

$$Based\ of\ Metric\ (BoM) = \frac{1}{6} \sum_{j=1}^6 MV_{Metric_j} \dots \dots \dots (4)$$

Where $Metric_j$ is $Metric_{WMC}, Metric_{RFC}, Metric_{CBO}, Metric_{LOCM}, Metric_{DIT}, Metric_{NOC}$,

They decided to prioritize the classes based on the value of the QDIR. They broke down the values of the QDIR into four classes. The highest 10% of the QDIR values are considered as a Critical Severity Bad Smells class. The 25% of the QDIR below the Critical Severity Bad Smells class are considered High Severity Bad Smell class. The 25% of the QDIR values below the High Severity Bad Smells class are considered Mild Severity Bad Smell. The lowest 40 % of the QDIR values are considered to be the Low Severity Bad Smell class.

9.1.2 Estimating refactoring efforts for Architecture Technical Debt

Deeb et al. estimated refactoring efforts using the COCOMO II: 2000 model. The refactoring efforts were calculated in person-month units per release. They used the Designite-Java tool to detect the architecture smells. Next, they ranked the packages based on two factors: Architecture smells (based on the number of the AS) and Refactoring Cost (based on the number added to LOC). After that, they divided the ranking into five (Very Low, Low, Medium, High, Very High) and three (Low, Medium, High) levels. In order to perform this division, they classified the ranks into five and three levels of normal distribution. Into the five levels, they distributed the number of packages at 10%, 20%, 40%, 20%, and 10% while they distributed the packages into the three levels at 25%, 50%, and 25%. After that, they tested the correlation between architecture smells and refactoring cost. Also, they applied machine learning to predict the results of the refactoring effort. They were able to achieve a good accuracy at the higher levels: 93% for the “Very High level” in the five level classification and 91.5% for the 'High level' in the three level classification.

9.2 Dataset Used in the Verification Phase

The dataset we will use in this Chapter was collected by Lenarduzzi et al. [103] and used by Deeb. It includes data extracted from apache projects. 44 project release are used in our verification phase. Table 9.3 shows the number of architecture smells, packages, classes Methods, and LOC in each project. Appendix D has more information about the used Java open sources.

Table 9.3 Apache Java Projects used in the validation Phase

Project name	Release number	# of AS	# of Packages	# of Classes	# of Methods	# of LOC
Atlas	0.7.1	41	60	579	4749	47243
Atlas	0.8.1	66	84	972	9620	92278

Atlas	0.8.2	72	91	1016	10475	100595
Aurora	0.9.0	35	41	670	5295	40069
Aurora	0.10.0	49	66	942	6795	52596
Aurora	0.12.0	50	64	933	6025	49410
Aurora	0.17.0	57	67	1027	6570	54030
Aurora	0.18.0	58	67	1057	6836	56619
Aurora	0.18.1	58	67	1057	6836	56619
Batik	1.5	64	76	2175	15406	194331
Batik	1.5.1	66	76	2209	15726	197275
Batik	1.6	69	81	2282	16399	204295
Batik	1.8	82	87	2595	20511	266609
beam	2.6.0	156	189	5438	42887	305578
beam	2.7.0	157	192	5519	43935	312440
beam	2.8.0	179	212	6512	51216	374879
beam	2.9.0	196	232	6920	53796	392678
beam	2.10.0	196	225	6998	55355	402873
beam	2.11.0	200	228	7105	56235	409789
cocoon	2.1.7	48	52	714	5329	68678
cocoon	2.1.8	52	54	761	5814	75321
cocoon	2.1.9	54	54	769	5844	75936
cocoon	2.1.10	50	55	775	5934	76810
commons-collections	3.3	19	12	664	7837	81368
commons-configuration	1.3	5	8	221	2668	33051
httpcomponents-client	4.0.2	24	27	374	2329	27408
httpcomponents-client	4.1	27	32	534	4019	45957
httpcomponents-client	4.1.3	30	32	552	4153	47906
httpcomponents-client	4.5.11	32	38	832	6722	70248
httpcomponents-core	4.1.4	32	34	510	3769	41220
httpcomponents-core	4.4.12	33	34	668	5552	58241
mina-sshd	0.14.0	41	55	534	3602	34105
mina-sshd	1.0.0	65	73	681	5301	50375
mina-sshd	1.1.0	73	84	846	6941	66575
mina-sshd	1.2.0	81	90	908	7688	74627
mina-sshd	1.3.0	87	91	937	7849	76790
mina-sshd	2.1.0	101	115	1092	9276	90145
santuario	1.5.2	36	58	537	3743	50873
santuario	1.5.3	37	61	550	3871	52302
santuario	1.5.4	37	62	554	3911	52820
zookeeper	3.4.5	21	30	660	4528	56625
zookeeper	3.4.10	21	33	748	5298	66300
zookeeper	3.4.11	23	33	753	5347	66838
zookeeper	3.4.12	23	33	761	5398	67235
zookeeper	3.4.14	22	32	787	5621	69352

Figure 9.1 shows the number of packages, the number of affected packages, and the percentage of the packages affected by the ATD for each project. On average, at least 50% of the packages were affected by AS. Project “*Beam 2.6.0*” has the minimum percentage of effectiveness with 8%, while project “*Commons-Collection 3.3*” has the maximum percentage of effectiveness with 92%.

Figure 9.1 Number and percentage of affected packages in the apache projects used in the validation phase

Figure 9.2 shows the types of architecture smell detected in each project. Project “*Atlas*” has the highest number of architecture smell instances of the projects we examined. As instances numbering 200, 196, and 196 AS were found in “*Atlas*” project releases 0.7.1, 0.8.1, and 0.8.2, respectively. This means the three releases of the “*Atlas*” project have 20.24% of the total number of the architecture smells. Aurora has the second-highest number of ASs. Project “*Aurora*”, releases 0.9.0, 0.10.0, 0.12.0, 0.17.0, 0.18.0, and 0.18.1 have 179, 157, 156, 101, 87, 82 architecture smell instances respectively. In general, the six “*Aurora*” project releases together have 762 AS instances, which is 26.05% of the total number of the architecture smell instances. The “*Zookeeper*” project had the least number of AS instances. We found 5, 19, 21, 21, and 22 AS instances in “*Zookeeper*” project releases 3.4.14, 3.4.12, 3.4.11, 3.4.10, and 3.4.5, respectively. The five “*Zookeeper*” project releases together have 88 AS instances, which is 3.01% of the total number of AS instances. To make a fair comparison between the projects, we estimated AS density and used it to compare between projects.

$$\text{Where AS density} = \frac{\text{Numebr of the Architecture smell instances}}{LOC}.$$

We found these results: The “*Atlas*” project has the highest AS density. The AS density in “*Atlas*” project releases 0.7.1, 0.8.1, and 0.8.2 is 0.45%, 0.42%, and 0.32%, respectively. Aurora has the second-highest AS density. The AS density in Project “*Aurora*”, releases 0.9.0, 0.10.0, 0.12.0, 0.17.0, 0.18.0, and 0.18.1 is 0.30%, 0.21%, 0.19%, 0.19%, 0.18% and 0.15% respectively. The “*Zookeeper*” project has the lowest AS density. The density of AS in all “*Zookeeper*” project releases is between 0.01% and 0.02%, respectively. The “*Batrik*” project releases have an AS density of between 0.09% and 0.15%. The “*Beam*” project releases have an AS density of between 0.07% and 0.09%. The “*Cocoon*” project releases have an AS density of between 0.05% and 0.07%. The “*commons-collections 3.3*” and “*commons-configuration 1.3*” project releases have an AS density of 0.05. The “*Cocoon*” project releases have an AS density of between 0.05% and 0.07%. Finally, all the “*httpcomponents-client*” and “*httpcomponents-core*” project releases have an AS density of 0.04.

Figure 9.2 Number of detected architecture smells in the apache projects used in the validation phase

9.3 Methodologies Verification

To validate our approach, we need to answer this research question:

RQ: To what extent was the method able to prioritize refactoring the packages?

Motivation: We were able to assess the ATD risk by applying our proposed methodology in Chapter 8. Assessing the ATD Risk can help decision-makers make a proper decision to prioritize refactoring the components (packages) based on the ATD risk level. We need to validate our results so the methodology can be trusted.

Approach: To answer this research question, we compare our methodology to related works. In Chapter 8, we extracted and collected our data from GitHub. All the projects in the dataset were C# projects. This Chapter will use a different dataset (Java projects) to apply our methodology to, and assess the ATD risk level per package. The dataset that will be used is trustworthy because many researchers use it. To verify our methods: (1) we will use Malhotra's methodology to assess the ATD risk level. We need to make some changes in the method mentioned because Malhotra's methodology was applied at the class level and our work is at the package level. To apply Malhotra's approach at the package level, different smells and some metrics will be added. We will use the same level classification of Malhotra's, and compare their results with ours. (2) We will use Deeb's dataset which already classified the packages based on the architecture smell level, and the refactoring effort level. We will use the two related works to estimate the accuracy of our methodology. (3) We will apply the Wilcoxon rank-sum test to check whether our method's results are statistically different from the two related works. Below are the test's null hypothesis and the test's alternative hypothesis statements to find any significant differences in the packages' classification based on the ATD risk levels.

H₀: There is no significant difference between our methodology and related work methodology.

H₁: There is a significant difference between our methodology and related work methodology.

Assumption: The actual values are the values that were generated by applying the related work methods.

We will answer our research question first by comparing our results with the result of Malhotra's methodology. After that we will answer it by comparing our results with the outcome of Deeb's approach.

9.3.1 Verifying our Methodology by comparing our results with Malhotra's approach

As we said in Section 9.1.1, we will prioritize the packages based on the value of the QDIR. As shown in Table 9.4, we will use the same four levels used by Malhotra, which are 10%, 25%, 25%, and 40% with regard to Critical Severity Architecture Smells, High Severity Architecture Smells, Mild Severity Architecture Smells, and Low Severity Architecture Smells class. The classification will consider only the packages that are affected by architecture smells, 1,855 packages.

Table 9.4 The classification levels used by the Malhotra method

Classifier	Package Classification
10%	Critical Severity Architecture Smell (CSAS)
25%	High Severity Architecture Smell (HSAS)
25%	Mild Severity Architecture Smell (MSAS)
40%	Low Severity Architecture Smell (LSAS)

To prioritize the packages based on the value of the QDIR we will apply equations (5, 6, 7, and 8) after making some changes to equations (1, 2, 3, and 4) to make the approach suited to prioritizing packages instead of classes. We will use architecture smells instead of code smells and metrics related to architecture smells.

$$Quality\ Depreciation\ Index\ Rule\ (QDIR) = \frac{BoAS}{2} + \frac{BoM}{2} \dots \dots \dots (5)$$

$$Base\ of\ Architecture\ Smell\ (BoAS) = \frac{1}{7} \sum_{i=1}^7 (Priority_{architecture\ smell}) \dots \dots \dots (6)$$

Where $Priority_{architecture\ smell}$ is $Priority_{CD,UD,AI,GC,FC,SF,DS}$

$$Metric\ Value\ (MV) = \frac{Calculated_{Metric}}{Threshold_{Metric}} \dots \dots \dots (7)$$

$$Based\ of\ Metric\ (BoM) = \frac{1}{8} \sum_{j=1}^8 MV_{Metric_j} \dots \dots \dots (8)$$

Where $Metric_j$ is $Metric_{NOC}$, $Metric_{NOM}$, $Metric_{LOC}$,

$Metric_{NC}$, $Metric_{WMC}$, $Metric_{DIT}$, $Metric_{Fanin}$, $Metric_{Fanout}$,

Figure 9.3 shows the confusion matrix results of the comparison between our classification and the related work's classification. The accuracy of each classification level can be estimated from the confusion matrix. The accuracy of the critical risk level and the high risk level is 80% and 70%, respectively. Less accuracy is found in the mild risk level 64%. Finally, our methodology has its high accuracy in the low-risk level with 88%.

		Our methodology			
		Critical risk	High risk	Mild risk	Low risk
Malhotra's methodology	Critical Severity	155	34	3	1
	High Severity	36	323	76	25
	Mild Severity	2	102	296	61
	Low Severity	0	1	86	654

Figure 9.3 Confusion Matrix of the ATD Risk and Quality Depreciation Index Rule

We tested the difference between our classification and Malhotra classification. We will use a non-parametric test, namely Wilcoxon test. The reason for using a non-parametric test is that our

data is ordinal variables including Likert items from four levels. Anyway, we applied the Wilcoxon rank-sum test using $\alpha = 0.01$ to verify whether the proposed method results are statistically different or not.

Our hypothesis:

H₀: There is no significant difference between the ATD risk level and QDIR.

H₁: There is a significant difference between the ATD risk level and QDIR.

Results: The null hypothesis (H₀) states that ATD risk levels and QDIR levels have no significant differences at $\alpha = 0.01$ (p-value > 0.01). The larger the p-value, the more it supports H₀. We found the p-value equals 0.96, which means we cannot reject the null hypothesis, because if we do so, the chance of type I error is too high, 96%.

9.3.2 Verifying our Methodology by comparing our results with Deeb's approach

We will prioritize the packages based on the level of the architecture smells and the refactoring effort. We will use the same two classifications used by Deeb. They used five and three levels, as reported in Sections 9.1.2. Contrary to what Malhotra did. This classification will consider all the packages and not just the affected packages. That means Deeb's and our methodology will be used to classify 3,480 packages. Both classification (5 and 3 levels) will be compared, and accuracy will be assessed based on Deeb's methodology.

9.3.2.1 Comparing ATD Risk with Refactoring Effort

Figure 9.4 shows the confusion matrix results of the comparison between ATD risk levels and the refactoring effort levels (3 levels classification). From the confusion matrix, the high and moderate risk classification accuracy is 81% and 89, respectively. More than 92 % of the low-risk class is right classified.

		Our methodology		
		High risk	Moderate risk	Low risk
Deeb's methodology	High Risk	713	116	49
	Moderate Risk	163	1541	20
	Low Risk	2	67	809

Figure 9.4 Confusion Metrix of the ATD Risk and Refactoring Effort (3 levels)

Figure 9.5 displays the confusion matrix results of the comparison between ATD risk levels and the refactoring effort levels (5 levels classification). From the confusion matrix, the very high and high-risk classification accuracy is 61% and 71%, respectively. The accuracy of the Moderate risk classification is around 88%. The accuracy of the very low and low-risk classification is 89% and 94%, respectively.

		Our methodology				
		Very High risk	High risk	Moderate risk	Low risk	Very Low Risk
Deeb's methodology	Very High risk	216	85	41	29	9
	High Risk	129	501	46	17	6
	Moderate Risk	10	114	1187	16	5
	Low Risk	1	3	77	635	0
	Very Low Risk	0	0	1	19	333

Figure 9.5 Confusion Matrix of the ATD Risk and Refactoring Effort (5 levels)

We tested the difference in the comparison between our classification and the refactoring effort classification used by Deeb. We used a non-parametric test because our data is ordinal data includes Likert items from three and five levels. We applied the Wilcoxon test using $\alpha = 0.01$ to verify whether the ATD risk and the refactoring effort classification results are statistically different or not.

Our hypothesis for three levels of classification:

H₀: There is no significant difference between the ATD risk levels and refactoring effort levels using three classifications levels.

H₁: There is a significant difference between the ATD risk levels and refactoring effort levels using three classifications levels.

Results: The null hypothesis (H_0) states that the ATD risk level and refactoring effort (3 levels) have no significant differences at $\alpha = 0.01$ (p-value > 0.01). We cannot reject the null hypothesis, because the p-value equals 0.79 ($> \alpha = 0.01$).

Our hypothesis for five levels of classification:

H₀: There is no significant difference between the ATD risk levels and refactoring effort levels using five classifications levels.

H₁: There is a significant difference between the ATD risk levels and refactoring effort levels using five classifications levels.

Results: The null hypothesis (H_0) states that the ATD risk level and refactoring effort levels (5 levels) have no significant differences at $\alpha = 0.01$ (p-value > 0.01). We cannot reject the null hypothesis, because the p-value equals 0.42 ($> \alpha = 0.01$).

9.3.2.2 Comparing ATD Risk with Architecture Smells levels

Figure 9.6 shows the confusion matrix results of the comparison between ATD risk levels and the architecture smells levels (3 levels classification). The accuracy of the high and moderate risk classification is 81% and 90, respectively. Almost all (99.9%) of the low-risk class is correctly classified.

		Our methodology		
		High risk	Moderate risk	Low risk
Deeb's methodology	High Risk	713	166	0
	Moderate Risk	165	1557	1
	Low Risk	0	1	877

Figure 9.6 Confusion Matrix of the ATD Risk and Architecture Smells (3 levels)

		Our methodology				
		Very High risk	High risk	Moderate risk	Low risk	Very Low Risk
Deeb's methodology	Very High risk	282	71	5	0	0
	High Risk	74	521	113	0	0
	Moderate Risk	0	111	1233	2	0
	Low Risk	0	0	1	714	0
	Very Low Risk	0	0	0	0	353

Figure 9.7 Confusion Matrix of the ATD Risk and Architecture Smells (5 levels)

Figure 9.7 shows the results of five levels of comparison between ATD risk levels and the architecture smells levels. By looking at the confusion matrix, we can realize that the very high-risk classification accuracy is 79%, while 90% of the high-level classes are correctly classified. More than 91% of the accuracy has been found on the moderate level. Finally, the low and very low-risk accuracy is 99.7% and 100%, respectively.

We tested the difference in comparison between our classification, and the architecture smells classification. Using $\alpha = 0.01$, we applied the Wilcoxon test to verify whether the ATD risk classification and the architecture smell classification results are statistically different or not.

Our hypothesis for three levels classification:

H₀: There is no significant difference between the ATD risk levels and architecture smell levels using three-level classifications.

H₁: There is a significant difference between the ATD risk levels and architecture smell levels using three-level classifications.

Results: The null hypothesis (H_0) states that, at three levels classifications, the ATD risk level and architecture smells have no significant differences at $\alpha = 0.01$ (p-value > 0.01). We cannot reject the null hypothesis; hence p-value equals 0.96 ($> \alpha = 0.01$).

Our hypothesis for five levels of classification:

H₀: There is no significant difference between the ATD risk levels and architecture smell levels using five classifications levels.

H₁: There is a significant difference between the ATD risk levels and architecture smell levels using five classification levels.

Results: The null hypothesis (H_0) states that the architecture smells level and refactoring effort levels (5 levels) have no significant differences at $\alpha = 0.01$ (p-value > 0.01). The p-value equals to 0.61 ($> \alpha = 0.01$), which means the null hypothesis cannot be rejected.

We used the Wilcoxon rank-sum test for two reasons.

Every method used different classification levels, which prevents us from performing the Friedman rank-sum test to compare the Effort level, Architecture Smells level, Quality Index, and the Architecture Technical Debt Risk level.

By following the related works' classification, we will avoid any subjective bias from our side. We can apply the Friedman rank-sum test if we use the same classification system for all the related methods. We decided to use a 3 level classification (low risk, moderate risk, high risk) used by Deeb et al. We saw QDIR classified by Malhotra et al. In four levels. We categorize the QDIR

results into three classes instead of the four-level classification. Then we can apply the Friedman rank-sum test using $\alpha = 0.01$ to verify whether the ATD risk, the refactoring effort, the Architecture Smells level, and the QDIR classification results are statistically different or not.

Our hypothesis for three levels of classification:

H₀: There is no significant difference between the results of the four methods

H₁: There is a significant difference between the results of the four methods

Results: The null hypothesis (H₀) states that the ATD risk level and refactoring effort (3 levels) have no significant differences at $\alpha = 0.01$ (p-value > 0.01). We cannot reject the null hypothesis because the p-value equals 0.033 ($> \alpha = 0.01$).

However, at $\alpha = 0.05$, the H₀ can be rejected. For that reason, we applied Nemenyi post hoc test on each pair of two methods to determine the difference.

Table 9.5 Nemenyi post hoc test to determine the difference

Nemenyi post hoc test between		Experiment wise error rate
Refactoring Efforts	Architecture Smells	0.83
Refactoring Efforts	QDIR	0.96
Refactoring Efforts	ATD Risk	0.84
Architecture Smells	QDIR	0.53
Architecture Smells	ATD Risk	1
QDIR	ATD Risk	0.55

From the Table 9.5, we conclude that:

1. The Refactoring Efforts and the Architecture Smells have no significant differences at $\alpha = 0.05$, where p-value = 0.83.

2. The Refactoring Efforts and the QDIR have no significant differences at $\alpha = 0.05$, where p-value = 0.96.
3. The Refactoring Efforts and the ATD Risk have no significant differences at $\alpha = 0.05$, where p-value = 0.84.
4. The Architecture Smells and the QDIR have no significant differences at $\alpha = 0.05$, where p-value = 0.53.
5. The Architecture Smells and the ATD Risk have no significant differences at $\alpha = 0.05$, where p-value = 1.
6. The Architecture Smells and the ATD Risk have no significant differences at $\alpha = 0.05$ where p-value = 0.55.

9.4 Threats to Validity

Construct Validity: To assess the ATD risk level, we used architecture smells as an indicator of ATD and seven internal metrics to determine each AS instance's impact. To avoid subjective bias, a dataset with a high degree of confidence was used. The approaches, factors, and classification levels used by the related works were defined to avoid any ambiguity.

Internal Validity: Poor data quality can be a threat to internal validity. The validation phase included a large and confident dataset used by other researchers. The projects that have architecture smells were included to validate our methodology. However, to ensure the dataset quality regarding the architecture smells and the software metrics, randomly selected projects were analyzed by applying the DesigniteJava tool. We performed this step to avoid threats to the internal validity

External Validity: We used datasets containing information extracted from Java rather than the dataset used in Chapter 8. The results cannot be generalized to other Java projects. The results could be generalized to apache Java after other researchers applying the same methodology, with other apache Java projects, if they get the similar results.

Conclusion validity: Is our ability to draw correct conclusions. Using uncorrected statistical tests will lead to that type of validity. We used a non-parametric test because our data is ordinal and includes Likert items in three, four, and five levels. The Wilcoxon rank-sum test was used to verify whether the proposed method results are statistically different compared to the related work

results. If we compared our results with only one method it could be a threat to the validity of our conclusion. For that reason, we compared our method with two related works that have three classification approaches (Quality Depreciation Index Rule classification, refactoring effort classification, and architecture smell classification). This is important for assessing our methodological accuracy by comparing it to multiple classification methods.

9.5 Conclusions

This Chapter seeks to validate the approach we proposed to assess the risks of architecture technical debt in software components. Around 3,480 packages (1,855 (53%) affected packages) from 45 apache java projects were used to test our methodology. After comparing the results of our method with the related works' results we were able to answer our research question: "How well was our method able to prioritize the refactoring packages?" and our main conclusions are the following:

- We reached 80 % accuracy at critical ATD risk levels, based on the Quality Depreciation Index Rule (QDIR).
- We reached 81% accuracy at the high-risk class in three classification levels and 61% at the very high risk class in the five classification levels, based on the level of the refactoring effort.
- We reached 81% and 79% in classification accuracy for the high-risk class in 3 and very high-risk in 5 level classifications, respectively. Evaluation was based on the level of the architecture smell.
- The Wilcoxon test between our methodology and related works indicated that there are no statistically significant differences.

Additionally, in Chapter 8 we assessed the likelihood of occurrence for each architecture smell type. Using C# projects we found Cyclic Dependency and Feature Concentration have the first and second highest probability of occurrence. We found that Unstable Dependency and Feature Concentration have the first and second highest probability of occurrence in Java projects.

Chapter 10

Dissertation Conclusions and Future Work

This Chapter completes this dissertation and presents planned future work. Section 10.1 describes the dissertation in the overview and shows how it contributes to the field. An outline of future work will be presented in Section 10.2.

10.1 Conclusions

We studied the Architecture Technical Debt. We proposed a methodology to assess the ATD risk level. This methodology can be used by project managers and decision-makers in refactoring decisions to assign priority based on the risk level. We examined relevant studies published between 2014 and 2017. We conducted an empirical study to analyze architecture technical debt, design debt, and code debt created by developers. Several developers, architects, and project managers studied ATD and refactoring during our journey. We used machine learning techniques to prioritize software components refactoring based on the severity of the expected ATD. We applied the proposed methodology on 40 C# project releases we extracted from GitHub. Finally, the proposed methodology's accuracy was tested using 45 apache java projects and related works.

In the first step, we studied the literature between 2014 and 2017 to find TD types, TD indicators, TD estimators, and to understand the methods and tools used to investigate, indicate, and quantify TD. We found that Code and Design debt is the most studied types. Database debt and Performance Debt are two types that were addressed for the first time by our studies (or recently) and never mentioned by the related work in Section 3.1. Smells were the most used indicator to identify TD. Special attention was paid to study SATD throughout the code comments. We found 5 Technical Debt Elements: Principal, Interest, Interest Probability, Probability of Refactoring, and TD lifespan. Different methods can be used to assess Technical Debt Elements, which are: Maintainability, Effort, Quality, Violation, and Experienced Professional Opinion. The SonarQube, and PMD tools were the most used tools. We know ten new tools were developed between 2014 and 2017.

In the second step, it was important to find a tool to study and analyze the TD, specifically, architecture technical debt (ATD), design debt (DD), and code debt (CD). This empirical study explored more than 25 kinds of smells created by developers. We applied the Designite tool to 42 C# open source project releases. We found that ASs, such as Cyclic Dependency and Feature Concentration, had the highest density. Developers refactored some of the technical debt in their projects, but because they added new features, TD's density increased in most cases between releases of the projects that we studied. The AS was the least common type of smell. Nevertheless, it decreased more than the DS density and CS.

Discovering how the software development teams rate the AS risk with respect to their roles and identifying the refactoring benefits gained and the challenges they face are very important. For that reason, we conducted a survey and emailed it to practicing software engineers. We found that the developer and project managers' responses indicated Feature Concentrations have a high risk to their software components, while Architect's responses showed Cyclic Dependency. We found that the impact of refactoring on adding new features prevents Developers from performing the refactoring. The main concern of Project Managers was the effect of refactoring on software quality. Besides sharing the same concern as the project managers, Architects indicated adding new features, convincing higher management, and lacking support from project management as other reasons preventing refactoring. Developers can increase code readability, maintainability, and usability by performing refactoring. Software architects were performing refactoring to reduce code duplication and bugs. Project Managers order refactoring to add new features, improve performance, and reduce maintenance costs. The largest challenge for all participants is the ability to compute the refactoring time. We found that software developers and architects have a fear of adding bugs through performing the refactoring. The main concern of participating Project Managers was that refactoring might disrupt adding new features. Most of the participants believe that ATD affects software systems. We conclusively found using a refactoring tool can increase the likelihood of choosing refactoring activities.

We used machine learning techniques to identify and classify ATD severity. We used four types of internal structure metrics to indicate ATD and classify every ATD instances based on the metrics' values. We created four levels of severity: high, moderate, low, and no severity. Machine learning algorithms were used to detect AS instances and classify their impacts on software components. The results showed that it is possible to use machine learning to measure ATD severity in software components. Meta Random Committee, Trees Random Forest, and Trees J48 algorithms can classify ATD severity with extraordinary accuracy. Categorizing the software components as no, low, moderates, and high severity can help in prioritizing refactoring. Multiple case studies were conducted applying the best models to track ATD severity through various releases. Tracking the severity of the ATD can help decision-makers monitor the need for refactoring and make proper refactoring decisions.

We propose a methodology to assess the ATD risk level in software components. The method was applied to 40 C#. We tracked and assessed the risk of more than 5,179 AS smells. In C# programming language, we found that Cyclic Dependency and God Component have the highest risk on software components. We found a high negative correlation between Cyclic Dependency and God Component, and between Cyclic Dependency and Feature Concentration. Also, we tested the correlation between the ATD risk and the AS probability and the AS impact. We found the AS impact is more correlated to the architecture smell risk than to the AS likelihood.

Lastly, we applied our methodology on a confident dataset that contains 45 apache java projects to estimate its accuracy. We compared the ATD risk classification with the Quality Depreciation Index Rule (QDIR); the average classification accuracy at critical ATD risk levels was 80%. We compared the ATD risk levels with refactoring effort levels; the average accuracy was 81% at the high level for three classifications and 61% at the very high level for five classifications. Also, ATD risk levels were compared with the AS levels; the average classification accuracy at the high-risk level was 81% (3 classifications) and 79% at the very high-risk level for five categories. Finally, the Wilcoxon rank-sum test ($\alpha = 0.01$) verified that there is no statistical significant difference between our method results and the related works' results.

10.2 Future Work

In the future, we plan to estimate the average refactoring effort implied for each architecture smell type. We intend to show the link between all the refactoring types and the architecture smell types. We will compare the results of each classification link with the change metrics extracted from the commits. The refactoring effort of each AS type can be assessed using the change metrics. We can use the same dataset we used in Chapter 9. This dataset has information about refactoring types, such as move class. For instance, to refactor God Component, we may perform a move class. But move class can also be performed to fix Future Concentration. However, we need to go through the literature review and survey practicing software engineers to examine all the possibility for refactoring each AS type. In addition, to distinguish between different refactoring reasons, we need to use issue tracking systems such as JIRA or SVN to extract the commit message. The potential benefits of the proposed study could help us to assess the ATD costs.

The development of tool to support the proposed ATD risk assessment methodology will also be important future work. The tool would depend on current tools to identify AS. It should take the project code files as an input, parses it, and display the packages as colored graphs. The color darkness shows the level of the ATD risk. A package classified as high, moderate, or low level of risk will be colored red, orange, or green, respectively.

We propose to conduct an empirical study at the university level. In this study, we will apply the proposed methodology to actual development projects written by students. We will divide the students into two groups based on their grades on the programming courses. We hope this can balance the two groups' programming abilities. The two groups will be given the same task. Randomly, one group will be selected to apply the proposed methodology while developing the project. We will explain the technical debt and teach the chosen group a way to detect TD. An available tool could be present to the selected group and guide the group to apply the proposed methodology. The two groups will be asked not to commentate and helps each other. They will make more than release by giving them new features to be added to the project. After that, we will analyze and compare the two groups' projects. The goal of this experiment could help us to assess the benefit of the proposed methodology.

Appendix A. Selected studies

This appendix shows information regarding the selected studies. The 43 studies were selected from different databases using the [WVU library website](#).

Study	Study title	Year of publication
S1	Who is Producing More Technical Debt? A Personalized Assessment of TD Principal	2017
S2	A large-scale empirical study on self-admitted technical debt	2016
S3	Pragmatic Approach for Managing Technical Debt in Legacy Software Project	2016
S4	Technical debt in MDE: a case study on GMF/EMF based projects	2016
S5	An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt	2014
S6	Measuring the principal of defect debt	2016
S7	The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study	2016
S8	Estimating the breaking point for technical debt	2015
S9	A Contextualized Vocabulary Model for identifying technical debt on code comments	2015
S10	Technical Debt and the Software Project Characteristics	2017
S11	CodeCritics applied to database schema: Challenges and first results	2017

S12	Identifying and visualizing Architectural Debt and its efficiency interest in the automotive domain: A case study	2015
S13	What if I Had No Smells?	2017
S14	Towards a prioritization of code debt: A code smell intensity index.	2015
S15	The correspondence between software quality models and technical debt estimation approaches	2014
S16	When-to-release decisions in consideration of technical debt	2014
S17	A case study in locating the architectural roots of technical debt	2015
S18	Technical Debt Principal Assessment Through Structural Metrics	2017
S19	Compiling Static Software Metrics for Reliability and Maintainability from GitHub Repositories	2017
S20	Detecting and quantifying different types of self-admitted technical debt	2015
S21	An empirical study on the removal of self-admitted technical debt	2017
S22	Using natural language processing to automatically detect self-admitted technical debt	2017
S23	An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt	2016
S24	A benchmarking-based model for technical debt calculation	2014
S25	Managing technical debt in software projects using scrum: An action research	2015
S26	An exploratory study on the relationship between changes and refactoring	2017
S27	An exploratory study on self-admitted technical debt	2014

S28	Does your configuration code smell?	2016
S29	Designite-A Software Design Quality Assessment Tool." Bringing Architectural Design Thinking Into Developers	2016
S30	Applying metrics to identify and monitor technical debt items during software evolution	2014
S31	A case study of program comprehension effort and technical debt estimations	2016
S32	How "Specification by Example" and Test-Driven Development Help to Avoid Technical Debt	2016
S33	When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away).	2017
S34	Assessing the Effectiveness of Static Analysis through Defect Correlation Analysis	2015
S35	Managing technical debt in database schemas of critical software	2014
S36	Examining the impact of self-admitted technical debt on software quality	2016
S37	Recommending when Design Technical Debt Should be Self-Admitted	2017
S38	The relation between technical debt and corrective maintenance in PHP web applications	2017
S39	Technical debt and system architecture: the impact of coupling on defect-related activity	2016
S40	Lessons Learned from the ProDebt Research Project on Planning Technical Debt Strategically	2017
S41	The Evolution of Technical Debt in the Apache Ecosystem	2017
S42	Identifying self-admitted technical debt in open source projects using text mining	2017
S43	Comparing four approaches for technical debt identification	2014

Appendix B: Survey Questionnaires

This appendix shows the survey questionnaires sent to the participants. We used Google Forms. The first response was received on February 27, 2020, and the last response was on March, 25 2020.

Part 1: General Information

1. What is your gender?

- ☐ Female
- ☐ Male
- ☐ Prefer not to say

2. What is your highest level of education?

- ☐ High School
- ☐ Bachelor degree
- ☐ Master degree
- ☐ PhD degree
- ☐ Other:

3. What is your job?

- ☐ Developer
- ☐ Designer
- ☐ Software Architect
- ☐ Software tester
- ☐ Project manager
- ☐ Other:

4. How many years of experience in software development do you have?

.....

5. Place of work?

- ☐ USA
- ☐ Canada
- ☐ Norway
- ☐ UK
- ☐ Libya
- ☐ Palestine
- ☐ Saudi Arabia
- ☐ Other:

6. Type of work?

- ☐ Self-Employ
- ☐ Academia
- ☐ Industry(Company)
- ☐ Other:

8. Based upon your experience, which architecture smells do you think has the highest impact on your project?

	Very high impact	high impact	Moderate impact	Low impact	Very low impact	I don't know
Cyclic Dependency	☐	☐	☐	☐	☐	☐
Unstable Dependency	☐	☐	☐	☐	☐	☐
Ambiguous Interface	☐	☐	☐	☐	☐	☐
God Component	☐	☐	☐	☐	☐	☐
Feature Concentration	☐	☐	☐	☐	☐	☐
Scattered Functionality	☐	☐	☐	☐	☐	☐
Dense Structure	☐	☐	☐	☐	☐	☐

9. How would you describe your knowledge about the concept of ATD?

- ☐ Very well
- ☐ Well
- ☐ Adequate
- ☐ Inadequate
- ☐ None

10. Should you, as a team member, detect ATD?

- ☐ Yes
- ☐ No

11. Should you, as a team member, manage ATD?

- ☐ Yes
- ☐ No

12. Do you think that the ATD can affect your project?

- ☐ Yes
- ☐ No

13. Who do you think is responsible for detecting the ATD? Please check all that apply.

- ☐ Developer
- ☐ Designer
- ☐ Software Architect
- ☐ Tester
- ☐ Project manager
- ☐ Other:

Part 3: Refactoring ATD

Please read before answering!

To remove the ATD, developers need to refactor their source code. Refactoring means rewrite your code to improve the internal structure of the software without changing its external behavior. **Refactoring examples:**

- Move Class, Method, Field, or Parameter.
- Rename Class, Method, Field, or Parameter
- Replace delegation with Inheritance.
- Delete Duplicated code or unused code.

14. How often do you refactor your project?

- ☐ Always
- ☐ Usually
- ☐ Sometimes
- ☐ Rarely
- ☐ Never (0%)

15. Do you use a refactoring tool?

- ☐ Yes
- ☐ No

16. What are some benefits gained from refactoring your project? Please check all that apply.

- ☐ Improve readability.
- ☐ Improve maintainability.
- ☐ Improve testability.
- ☐ Improve performance.
- ☐ Improve usability.
- ☐ Help adding new features.
- ☐ Reduce complexity.
- ☐ Reduce duplication.

- Reduce code size.
- Reduce the bugs.
- Reduce maintenance costs.
- Other:

17. What are some challenges faced by refactoring your project? Please check all that apply.

- Adding bugs
- Disrupting adding new features.
- Taking more time than expected
- Other:

18. Which of the following issues prevent you from refactoring your project? Please check all that apply.

- Unable to perform refactoring tasks because of an emphasis on feature implementation
- Unable to predict the impact of the changes caused by refactoring
- Unable to clearly show the improvement in software quality once refactoring is complete
- Unable to estimate the effort required for refactoring
- Unable to prioritize refactoring candidates
- Unable to identify refactoring candidates
- Unable to convince higher management about the need for refactoring
- Lack of awareness about refactoring's impact on product quality
- Lack of support from project management
- Other:

19. Who do you think is responsible for refactoring the ATD? Please check all that apply

- Developer
- Designer
- Software Architect
- Tester
- Project manager
- Other:

Appendix C: C# Projects

This appendix shows some information regarding the selected open-source projects. The ten C# open source projects were selected from GitHub. We picked appropriate open-source projects and we applied our methodology to them. We searched for C# open source projects in GitHub. After that, the C# open source projects were sorted by most stars. Most stars indicated the project's popularity. We feel that there could be a better metric that could be used to find the best candidate. For instance, we could use the number of users, but many projects do not have this information. As a best effort we selected the ten “most stars” C# open source projects with multiple releases, with source code could be analyzed using the Designate tool.

C# OPEN SOURCE PROJECTS

Information depends on accessed Oct. 27, 2020

1. Project name: Simple.Data

Link: <https://github.com/markrendle/Simple.Data>

Number of Release: 22

Used by: 375 users

C# %: 99.8%

Date of the First Release: May 24, 2011

2. Project name: ILRuntime

Link: <https://github.com/Ourpalm/ILRuntime>

Number of Release: 15

Used by: unknown

C# %: 100%

Date of the First Release: Oct 13, 2016

3. Project name: Hawk

Link: <https://github.com/ferventdesert/Hawk>

Number of Release: 7

Used by: unknown

C# %: 100%

Date of the First Release: Nov 13, 2016

4. Project name: ZeroFormatter

Link: <https://github.com/neuecc/ZeroFormatter>

Number of Release: 25

Used by: 151 users

C# %: 99.8%

Date of the First Release: Nov 7, 2016

5. Project name: Xamarin-android

Link: <https://github.com/xamarin/xamarin-android>

Number of Release: 61

Used by: unknown

C# %: 88%

Date of the First Release: Jul 3, 2018

6. Project name: OpenID, OAuth protocols

Link: <https://github.com/DotNetOpenAuth/DotNetOpenAuth>

Number of Release: 107

Used by: 20,240 users

C# %: 97.9%

Date of the First Release: Jun 20, 2006

7. Project name: OpenRA

Link: <https://github.com/OpenRA/OpenRA>

Number of Release: 374

Used by: unknown

C# %: 83%

Date of the First Release: Apr 16, 2010

8. Project name Sharpcompress

Link: <https://github.com/adamhathcock/sharpcompress>

Number of Release: 48

Used by: 2,620 users

C# %: 71.1%

Date of the First Release: May 4, 2013

9. Project name NRefactory

Link: <https://github.com/icsharpcode/NRefactory>

Number of Release: 7

Used by: unknown

C# %: 100%

Date of the First Release: May 18, 2012

10. Project name UnityCsReference

Link <https://github.com/Unity-Technologies/UnityCsReference>

Number of Release: 450

Used by: unknown

C# %: 100%

Date of the First Release: Mar 13, 2017

Appendix D: Apache Java Projects

This appendix presents information regarding the 12 apache java projects which selected from the dataset and used to validate our methodology.

APACHE JAVA PROJECTS:

Information depends on accessed Oct. 27, 2020

1. Project name: Atlas

Link: <https://github.com/apache/atlas>

Number of Release: 35

Used by: 4 users

Java %: 67.1%

Date of the First Release: Jun 17, 2015

2. Project name: Aurora

Link: <https://github.com/apache/attic-aurora>

Number of Release: 40

Used by: Unknown

Java %: 63.6%

Date of the First Release: Dec 17, 2013

3. Project name: Batik

Link: <https://github.com/apache/xmlgraphics-batik>

Number of Release: 42

Used by: Unknown

Java %: 99.3%

Date of the First Release: Oct 10, 2000

4. Project name: Beam

Link: <https://github.com/apache/beam>

Number of Release: 146

Used by: Unknown

Java %: 72.2%

Date of the First Release: May 31, 2016

5. Project name: cocoon

Link: <https://github.com/apache/cocoon>

Number of Release: 77

Used by: Unknown

Java %: 76.7%

Date of the First Release: Apr 29, 2003

6. Project name: commons-collections

Link: <https://github.com/apache/commons-collections>

Number of Release: 49

Used by: 86,012 users

Java %: 99.3%

Date of the First Release: Jul 28, 2007

7. Project name: commons-configuration

Link: <https://github.com/apache/commons-configuration>

Number of Release: 71

Used by: 86,012 users

Java %: 99.9%

Date of the First Release: Jul 28, 2007

8. Project name: httpcomponents-client

Link: <https://github.com/apache/httpcomponents-client>

Number of Release: 141

Used by: 279 users

Java %: 99.9%

Date of the First Release: Jul 24, 2008

9. Project name: httpcomponents-core

Link: <https://github.com/apache/httpcomponents-core>

Number of Release: 146

Used by: 48 users

Java %: 99.9%

Date of the First Release: Aug 22, 2010

10. Project name: mina-sshd

Link: <https://github.com/apache/mina-sshd>

Number of Release: 31

Used by: 33 users

Java %: 99%

Date of the First Release: Jun 12, 2009

11. Project name: santuario

Link: <https://github.com/apache/santuario-java>

Number of Release: 65

Used by: Unknown

Java %: 50.6%

Date of the First Release: Dec 15, 2010

12. Project name: zookeeper

Link: <https://github.com/apache/zookeeper>

Number of Release: 126

Used by: 124 users

Java %: 74.4%

Date of the First Release: Nov 24, 2010

Bibliography

- [1] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.
- [2] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [3] C. Fernández-Sánchez, J. Díaz, J. Pérez, and J. Garbajosa, "Guiding flexibility investment in agile architecting," in *2014 47th Hawaii International Conference on System Sciences*, 2014, pp. 4807–4816.
- [4] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee Softw.*, vol. 29, no. 6, pp. 18–21, 2012.
- [5] Z. Li, P. Liang, and P. Avgeriou, "Architectural debt management in value-oriented architecting," in *Economics-Driven Software Architecture*, Elsevier, 2014, pp. 183–204.
- [6] 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*, 2015, pp. 65–74.
- [7] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the 4th Workshop on Refactoring Tools*, 2011, pp. 33–36.
- [8] M. Chicote, "Startups and technical debt: managing technical debt with visual thinking," in *2017 IEEE/ACM 1st International Workshop on Software Engineering for Startups (SoftStart)*, 2017, pp. 10–11.
- [9] W. Opdyke, "Refactoring, reuse & reality," *Lucent Technol. Labs*, 1999.
- [10] G. Samarthayam, M. Muralidharan, and R. K. Anna, "Understanding test debt," in *Trends in Software Testing*, Springer, 2017, pp. 1–17.
- [11] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, pp. 193–220, 2015.
- [12] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *J. Syst. Softw.*, vol. 86, no. 6, pp. 1498–1516, 2013.

- [13]A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Inf. Softw. Technol.*, vol. 64, pp. 52–73, 2015.
- [14]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," *Inf. Softw. Technol.*, vol. 70, pp. 100–121, 2016.
- [15]T. Besker, A. Martini, and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," *J. Syst. Softw.*, vol. 135, pp. 1–16, 2018.
- [16]R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 350–359.
- [17]S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *2009 16th Working Conference on Reverse Engineering*, 2009, pp. 145–154.
- [18]S. 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 *2009 3rd international symposium on empirical software engineering and measurement*, 2009, pp. 390–400.
- [19]W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, no. 7, pp. 1120–1128, 2007.
- [20]M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Prioritising refactoring using code bad smells," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 458–464.
- [21]F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *2009 16th Working Conference on Reverse Engineering*, 2009, pp. 75–84.
- [22]T. Sharma, M. Frangkoulis, and D. Spinellis, "House of cards: code smells in open-source C# repositories," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 424–429.
- [23]R. Mo, J. Garcia, Y. Cai, and N. Medvidovic, "Mapping architectural decay instances to dependency models," in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013, pp. 39–46.

- [24]F. Tian, P. Liang, and M. A. Babar, "How developers discuss architecture smells? an exploratory study on Stack Overflow," in *2019 IEEE International Conference on Software Architecture (ICSA)*, 2019, pp. 91–100.
- [25]M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, 2014.
- [26]S. Jain and A. Saha, "An Empirical Study on Research and Developmental Opportunities in Refactoring Practices.," in *SEKE*, 2019, pp. 313–418.
- [27]A. Martini, F. A. Fontana, A. Biaggi, and R. Roveda, "Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company," in *European Conference on Software Architecture*, 2018, pp. 320–335.
- [28]R. Roveda, F. A. Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018, pp. 408–416.
- [29]E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Softw.*, vol. 25, no. 5, pp. 38–44, 2008.
- [30]T. Sharma, G. Suryanarayana, and G. Samarthayam, "Challenges to and solutions for refactoring adoption: An industrial perspective," *IEEE Softw.*, vol. 32, no. 6, pp. 44–51, 2015.
- [31]D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?," in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (saner)*, 2018, pp. 612–621.
- [32]F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code smell detection: Towards a machine learning-based approach," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 396–399.
- [33]F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, 2016.
- [34]A. Maiga *et al.*, "Support vector machines for anti-pattern detection," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 278–281.

- [35]N. Maneerat and P. Muenchaisri, “Bad-smell prediction from software design model using machine learning techniques,” in *2011 Eighth international joint conference on computer science and software engineering (JCSSE)*, 2011, pp. 331–336.
- [36]F. A. Fontana and M. Zanoni, “Code smell severity classification using machine learning techniques,” *Knowl.-Based Syst.*, vol. 128, pp. 43–58, 2017.
- [37]F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, “Smells like teen spirit: Improving bug prediction performance using the intensity of code smells,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 244–255.
- [38]T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “On the feasibility of transfer-learning code smells using deep learning,” *ArXiv Prepr. ArXiv190403031*, 2019.
- [39]N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, 2009.
- [40]U. Azadi, F. A. Fontana, and M. Zanoni, “Poster: machine learning based code smell detection through WekaNose,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2018, pp. 288–289.
- [41]F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, “Towards a prioritization of code debt: A code smell intensity index,” in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 16–24.
- [42]A. Martini, E. Sikander, and N. Medlani, “Estimating and quantifying the benefits of refactoring to improve a component modularity: a case study,” in *2016 42th Euromicro conference on software engineering and advanced applications (SEAA)*, 2016, pp. 92–99.
- [43]R. Kazman *et al.*, “A case study in locating the architectural roots of technical debt,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 2, pp. 179–188.
- [44]R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, “In search of a metric for managing architectural technical debt,” in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2012, pp. 91–100.
- [45]F. Öztürk, E. Sarılı, H. Sözer, and B. Aktemur, “Effort estimation for architectural refactoring to introduce module isolation,” in *European Conference on Software Architecture*, 2014, pp. 300–307.

- [46]Y. Guo, R. O. Spínola, and C. Seaman, “Exploring the costs of technical debt management—a case study,” *Empir. Softw. Eng.*, vol. 21, no. 1, pp. 159–182, 2016.
- [47]L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, “Identifying and quantifying architectural debt,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 488–498.
- [48]M. Benlhdri, “Investigate, identify and estimate the technical debt: a systematic mapping study,” *Available SSRN 3606172*, 2020.
- [49]N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, “Towards an ontology of terms on technical debt,” in *2014 Sixth International Workshop on Managing Technical Debt*, 2014, pp. 1–7.
- [50]G. Bavota and B. Russo, “A large-scale empirical study on self-admitted technical debt,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 315–326.
- [51]S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, “Which documentation for software maintenance?,” *J. Braz. Comput. Soc.*, vol. 12, no. 3, pp. 31–44, 2006.
- [52]D. Steidl, B. Hummel, and E. Juergens, “Quality analysis of source code comments,” in *2013 21st international conference on program comprehension (icpc)*, 2013, pp. 83–92.
- [53]A. Potdar and E. Shihab, “An exploratory study on self-admitted technical debt,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 91–100.
- [54]M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. da Silva, and R. O. Spínola, “A contextualized vocabulary model for identifying technical debt on code comments,” in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 25–32.
- [55]A. Martini and J. Bosch, “An empirically developed method to aid decisions on architectural technical debt refactoring: AnaConDebt,” in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 31–40.
- [56]T. Sharma, M. Frangkoulis, and D. Spinellis, “Does your configuration code smell?,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 189–200.
- [57]T. Sharma, P. Mishra, and R. Tiwari, “Designite: A software design quality assessment tool,” in *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers’ Daily Activities*, 2016, pp. 1–4.

[58]W. Trumler and F. Paulisch, “How ‘Specification by Example’ and test-driven development help to avoid technical debt,” in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, 2016, pp. 1–8.

[59]M. Ciolkowski, L. Guzmán, A. Trendowicz, and F. Salfner, “Lessons learned from the ProDebt research project on planning technical debt strategically,” in *International Conference on Product-Focused Software Process Improvement*, 2017, pp. 523–534.

[60]N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, “Comparing four approaches for technical debt identification,” *Softw. Qual. J.*, vol. 22, no. 3, pp. 403–426, 2014.

[61]G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, “The evolution of technical debt in the apache ecosystem,” in *European Conference on Software Architecture*, 2017, pp. 51–66.

[62]A. Ampatzoglou *et al.*, “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)*, 2016, pp. 9–16.

[63]J. Delplanque, A. Etien, O. Auverlot, T. Mens, N. Anquetil, and S. Ducasse, “CodeCritics applied to database schema: Challenges and first results,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 432–436.

[64]J. H. Weber, A. Cleve, L. Meurice, and F. J. B. Ruiz, “Managing technical debt in database schemas of critical software,” in *2014 Sixth International Workshop on Managing Technical Debt*, 2014, pp. 43–46.

[65]F. Oliveira, A. Goldman, and V. Santos, “Managing technical debt in software projects using scrum: An action research,” in *2015 Agile Conference*, 2015, pp. 50–59.

[66]E. Alzaghouli and R. Bahsoon, “CloudMTD: Using real options to manage technical debt in cloud-based service selection,” in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013, pp. 55–62.

[67]Z. Codabux and B. Williams, “Managing technical debt: An industrial case study,” in *2013 4th International Workshop on Managing Technical Debt (MTD)*, 2013, pp. 8–15.

[68]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*, 2015, pp. 50–60.

- [69]D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, “On the relation of test smells to software code quality,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 1–12.
- [70]Z. Codabux, K. Z. Sultana, and B. J. Williams, “The Relationship between Traceable Code Patterns and Code Smells,” in *SEKE*, 2017, pp. 444–449.
- [71]J. Gil and I. Maman, “Micro patterns in Java code,” in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005, pp. 97–116.
- [72]F. A. Fontana, B. Walter, and M. Zanoni, “Code smells and micro patterns correlations,” 2013.
- [73]J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yiapanis, “Fundamental nano-patterns to characterize and classify java methods,” *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 7, pp. 191–204, 2010.
- [74]A. Deo and B. J. Williams, “Preliminary study on assessing software defects using nano-pattern detection,” 2015.
- [75]W. Fenske and S. Schulze, “Code smells revisited: A variability perspective,” in *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, 2015, pp. 3–10.
- [76]“checkstyle – Checkstyle 8.36.2.” <https://checkstyle.sourceforge.io/> (accessed Oct. 26, 2020).
- [77]“PMD.” <https://pmd.github.io/> (accessed Oct. 26, 2020).
- [78]M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.
- [79]E. Murphy-Hill and A. P. Black, “An interactive ambient visualization for code smells,” in *Proceedings of the 5th international symposium on Software visualization*, 2010, pp. 5–14.
- [80]N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, “JDeodorant: Identification and removal of type-checking bad smells,” in *2008 12th European Conference on Software Maintenance and Reengineering*, 2008, pp. 329–331.

[81]J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Filtering clones for individual user based on machine learning analysis," in *2012 6th International Workshop on Software Clones (IWSC)*, 2012, pp. 76–77.

[82]E. da S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 238–248.

[83]"Designite - Reduce Technical Debt of your Software." <http://www.designite-tools.com/> (accessed Oct. 26, 2020).

[84]L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on computer supported cooperative work*, 2012, pp. 1277–1286.

[85]J. S. de Jesus and A. C. V. de Melo, "Technical debt and the software project characteristics. A repository-based exploratory analysis," in *2017 IEEE 19th Conference on Business Informatics (CBI)*, 2017, vol. 1, pp. 444–453.

[86]F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment.," *J Object Technol*, vol. 11, no. 2, p. 5:1-38, 2012.

[87]F. Shull, J. Singer, and D. I. Sjøberg, *Guide to advanced empirical software engineering*. Springer, 2007.

[88]M. BenIdris, H. Ammar, D. Dzielski, and W. H. Benamer, "Prioritizing Software Components Risk: Towards a Machine Learning-based Approach," in *Proceedings of the 6th International Conference on Engineering & MIS 2020*, 2020, pp. 1–11.

[89]E. A. O. Elfallah and W. H. Benamer, "Studying the classification accuracy performance when representation is changed on several classifier techniques," in *Proceedings of the International Conference on High Performance Compilation, Computing and Communications*, 2017, pp. 50–54.

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

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

[92]F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zaroni, "Automatic detection of instability architectural smells," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 433–437.

[93]S. G. Ganesh, T. Sharma, and G. Suryanarayana, "Towards a Principle-based Classification of Structural Design Smells.," *J Object Technol*, vol. 12, no. 2, p. 1:1-29, 2013.

[94]T. Sharma, "Extending maintainability analysis beyond code smells," Οικονομικό Πανεπιστήμιο Αθηνών. Σχολή Διοίκησης Επιχειρήσεων. Τμήμα ..., 2019.

[95]R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study," *Manag. Sci.*, vol. 44, no. 4, pp. 433–450, Apr. 1998, doi: 10.1287/mnsc.44.4.433.

[96]T. Honglei, S. Wei, and Z. Yanan, "The research on software metrics and software complexity metrics," in *2009 International Forum on Computer Science-Technology and Applications*, 2009, vol. 1, pp. 131–136.

[97]R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, 1995, vol. 14, no. 2, pp. 1137–1145.

[98]Y. Liu, J. Cheng, C. Yan, X. Wu, and F. Chen, "Research on the Matthews correlation coefficients metrics of personalized recommendation algorithm evaluation," *Int. J. Hybrid Inf. Technol.*, vol. 8, no. 1, pp. 163–172, 2015.

[99]W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. V. Staa, "On the relationship of code-anomaly agglomerations and architectural problems," *J. Softw. Eng. Res. Dev.*, vol. 3, no. 1, p. 11, 2015.

[100] G. E. Apostolakis, "How useful is quantitative risk assessment?," *Risk Anal. Int. J.*, vol. 24, no. 3, pp. 515–520, 2004.

[101] S. A. Sherer, "The three dimensions of software risk: technical, organizational, and environmental," in *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, 1995, vol. 3, pp. 369–378.

[102] S. M. Yacoub and H. H. Ammar, "A methodology for architecture-level reliability risk analysis," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 529–547, 2002.

[103] V. Lenarduzzi, A. Martini, D. Taibi, and D. A. Tamburri, “Towards surgically-precise technical debt estimation: early results and research roadmap,” in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 37–42.

[104] R. Malhotra, A. Chug, and P. Khosla, “Prioritization of classes for refactoring: A step towards improvement in software quality,” in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, 2015, pp. 228–234.