

UNDERSTANDING CODE SMELLS AND REFACTORING  
PRACTICES IN SIMULATION MODELLING SYSTEMS - A  
COMPREHENSIVE STUDY

by

Riasat Mahbub

Submitted in partial fulfillment of the requirements  
for the degree of Master of Computer Science

at

Dalhousie University  
Halifax, Nova Scotia  
August 2025

© Copyright by Riasat Mahbub, 2025

*To Baba and Mamoni for supporting me through all hardships and  
loving me unconditionally*

# Table of Contents

<b>List of Tables</b> . . . . .	<b>vii</b>
<b>List of Figures</b> . . . . .	<b>viii</b>
<b>Abstract</b> . . . . .	<b>ix</b>
<b>Acknowledgements</b> . . . . .	<b>x</b>
<b>Chapter 1 Introduction</b> . . . . .	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem Statement . . . . .	2
1.3 Research Contribution . . . . .	3
1.4 Related Publications . . . . .	5
1.5 Outline of the Thesis . . . . .	5
<b>Chapter 2 Background</b> . . . . .	<b>7</b>
2.1 Code Smells . . . . .	7
2.1.1 Implementation Smells . . . . .	7
2.1.2 Design Smells . . . . .	8
2.1.3 Architectural Smells . . . . .	9
2.2 Refactoring Techniques . . . . .	10
2.2.1 Method-level Refactoring Techniques . . . . .	10
2.2.2 Class-level Refactoring Techniques . . . . .	15
2.2.3 Package-level Refactoring Techniques . . . . .	17
2.3 Maintainability Metrics . . . . .	18
2.3.1 Method Level Metrics . . . . .	18
2.3.2 Class Level Metrics . . . . .	19
2.3.3 Package Level Metrics . . . . .	19
2.4 Contingency Table . . . . .	20
2.5 Statistical Significance . . . . .	21
2.5.1 Mann-Whitney U Test . . . . .	21
2.5.2 Cliff's Delta . . . . .	21
2.6 Association Tests . . . . .	22

2.6.1	Chi Squared Test . . . . .	22
2.6.2	Cramer’s V Test . . . . .	23
2.7	Survival Analysis . . . . .	23
2.8	Nelson-Aalen Estimator . . . . .	24
2.9	Mann Kendall Test . . . . .	25
2.9.1	Stouffer’s Method . . . . .	26
2.10	Summary . . . . .	26
<b>Chapter 3</b>	<b>On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software . . . . .</b>	<b>28</b>
3.1	Introduction . . . . .	28
3.2	Methodology . . . . .	30
3.2.1	Project Selection Criteria . . . . .	32
3.2.2	Data Collection . . . . .	32
3.2.3	Comparative Analysis . . . . .	33
3.2.4	Survivability Analysis . . . . .	33
3.2.5	Mining Bug Inducing and Fixing Commits . . . . .	34
3.2.6	Association between Code Smells and Bugs . . . . .	35
3.2.7	Replication Package . . . . .	35
3.3	Results and Analysis . . . . .	36
3.3.1	RQ1: Do simulation software systems smell like traditional software systems? . . . . .	36
3.3.2	RQ2: How long do code smells last in simulation systems? . . . . .	42
3.3.3	RQ3: Do code smells co-occur with bugs in simulation software systems? . . . . .	48
3.4	Implication of Findings . . . . .	49
3.5	Threats to Validity . . . . .	50
3.6	Related Work . . . . .	51
3.6.1	Prevalence of code smells . . . . .	51
3.6.2	Evolution of code smells . . . . .	52
3.6.3	Impact of code smells . . . . .	52
3.7	Summary . . . . .	52

<b>Chapter 4</b>	<b>On the Effectiveness, Risks, and Impact of Refactoring Practices in Simulation Modelling Software . . . . .</b>	<b>54</b>
4.1	Introduction . . . . .	54
4.2	Methodology . . . . .	57
4.2.1	Project Selection Criteria . . . . .	57
4.2.2	Mine Refactoring Commits . . . . .	59
4.2.3	Code Smell Detection . . . . .	59
4.2.4	Effectiveness of Refactoring . . . . .	60
4.2.5	Risks of Refactoring . . . . .	61
4.2.6	Impact of refactoring . . . . .	62
4.2.7	Replication Package . . . . .	63
4.3	Results and Analysis . . . . .	63
4.3.1	RQ1: Do refactoring practices effectively remove code smells from simulation software systems? . . . . .	63
4.3.2	RQ2: Do refactoring practices risk developing new code smells in simulation software systems? . . . . .	69
4.3.3	RQ3: Do refactoring practices improve the maintainability of simulation software systems? . . . . .	74
4.4	Implication of Findings . . . . .	76
4.5	Threats to Validity . . . . .	78
4.5.1	Threats to internal validity . . . . .	78
4.5.2	Threats to conclusion validity . . . . .	78
4.5.3	Threats to external validity . . . . .	78
4.6	Related Works . . . . .	79
4.6.1	Effectiveness of Refactoring . . . . .	79
4.6.2	Risks of Refactoring . . . . .	79
4.6.3	Impacts of Refactoring . . . . .	80
4.7	Summary . . . . .	80
<b>Chapter 5</b>	<b>Conclusion and Future Work . . . . .</b>	<b>82</b>
5.1	Conclusion . . . . .	82
5.2	Limitations . . . . .	84
5.2.1	Code Smell Analysis . . . . .	84
5.2.2	Refactoring . . . . .	84
5.3	Future Work . . . . .	85
5.3.1	Code Smell Analysis . . . . .	85
5.3.2	Refactoring Practices . . . . .	85

<b>Appendix A</b>	<b>Complementary Materials</b>	<b>102</b>
A.1	Replication Package	102
A.1.1	On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software	102
A.1.2	On the Effectiveness, Risks, and Impact of Refactoring Practices in Simulation Modelling Software	102
A.2	Copyright Permissions	102
A.2.1	On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software	102

## List of Tables

2.1	An example of contingency table . . . . .	20
2.2	Interpretation of Cliff's delta effect sizes . . . . .	22
2.3	Interpretation of Cramer's V association strengths . . . . .	23
3.1	Keywords for detecting bug-fix commits . . . . .	34
3.2	Significance tests for code smell's prevalence . . . . .	40
3.3	Median survival times (MST) of all code smells . . . . .	46
3.4	Value of Cramer's V and Chi-square p values for each code smell	48
4.1	Example of contingency table for removed <i>Implementation</i> smells and Method-Level refactoring techniques . . . . .	60
4.2	Example of contingency table for new <i>Implementation</i> smells and Method-Level refactoring techniques . . . . .	61
4.3	Association between refactoring technique and removed smells .	66
4.4	Association between refactoring and newly introduced code smells	72
4.5	Trend of method level metrics after refactoring . . . . .	74
4.6	Trend of class level metrics after refactoring . . . . .	75
4.7	Trend of package level metrics after refactoring . . . . .	75

## List of Figures

3.1	Schematic diagram of our empirical study . . . . .	31
3.2	Prevalence of code smells in simulation and traditional systems	36
3.3	Distribution of normalized frequencies of code smells in simulation and traditional systems . . . . .	37
3.4	Prevalence of normalized frequencies of code smells in simulation and traditional systems (by smell type) . . . . .	38
3.5	Distribution of normalized frequencies of code smells in simulation and traditional systems (by smell type) . . . . .	39
3.6	Prevalence of normalized frequencies of code smells in simulation and traditional systems (for significantly different code smells)	41
3.7	Survivability curves of code smells in simulation and traditional systems . . . . .	43
3.8	Survivability curves of code smells in simulation systems (by abstraction level) . . . . .	44
3.9	Survivability curves of code smells in simulation systems (by smell type) . . . . .	45
3.10	Number of code smells in bug-inducing and bug-fixing commits	47
4.1	Schematic diagram of our study . . . . .	58
4.2	Comparison between removed simulation and traditional smells	64
4.3	Breakdown of removed code smells in simulation systems . . .	65
4.4	Kaplan-Meier survival curves for simulation systems . . . . .	67
4.5	Top refactoring techniques for all code smells . . . . .	68
4.6	Removed and newly introduced smells in simulation systems .	70
4.7	Nelson-Aalen survival curve for simulation systems . . . . .	70
4.8	Comparison of composition of new and removed smells for simulation systems . . . . .	72
4.9	Risk of introducing new code smells for each refactoring technique	73

## Abstract

Simulation software systems play a critical role in many areas, including scientific research, planning, and industrial operations. They provide controlled environments for testing and understanding complex real-world scenarios (e.g., Traffic simulation, GHG emission modelling). Despite their importance, they have received limited attention from the software engineering community to date. In particular, their code quality issues, such as code smells and their remediation's, are not well investigated, indicating a major gap in the literature. To address the gap, we conducted two large-scale empirical studies in this thesis. Our first study examines the prevalence, evolution, and impact of code smells in 155 simulation and 327 traditional software systems from GitHub. We find that many implementation-level smells, such as Magic Number and Long Statement, are more prevalent in simulation systems than in traditional systems. We also find that some code smells (e.g., Broken Hierarchy) last longer in simulation systems. Interestingly, we found no correlation between bugs and code smells in simulation systems. Building upon these findings, our second study investigates the effectiveness, risks, and impact of refactoring practices in 104 simulation and 272 traditional software systems from GitHub. Our findings suggest that refactoring removes approximately 35% of Implementation, Design and Architecture smells. However, method-level refactoring activities introduce 2.5 times more code smells than they remove, making them risky. We also found that refactoring activities reduce Structural and Cyclomatic complexity in simulation software systems over time. Thus, our overall findings shed light on the simulation systems code quality issues (e.g., code smells) and refactoring activities, equip software developers and simulation modellers with actionable insights, and also advance the current state of knowledge.

## Acknowledgements

In the name of Allah, the Most Gracious, the Most Merciful. First and foremost, I thank Allah (SWT) for granting me good health, intellectual capacity, and the fortitude to carry out my thesis. His blessings have guided me throughout this journey and made this accomplishment possible.

I want to express my immense gratitude to my supervisors, Dr. Masud Rahman and Dr. Ahsan Habib, for allowing me to join their research groups and supporting me throughout my Master's studies. I thank them for motivating me and believing in me constantly, especially when I could not. Their unwavering support, insightful feedback, and patience have been invaluable to me throughout my thesis work.

I would like to sincerely thank my thesis committee members, Dr. Tushar Sharma and Dr. Hamid Afshari, for their time, expertise, and valuable suggestions which have significantly improved this work. I am also grateful to Dr. Alexander Brandt, the chair of my committee, for his guidance and for ensuring the smooth progress of my thesis defense process.

To my beloved parents, Mahbubul Alam and Mashrura Akter, for their support and love in every capacity throughout my life. Their guidance, encouragement, and sacrifices have been instrumental in allowing me to achieve this milestone in my life and shaping the person I am today.

I would like to acknowledge the irreplaceable support and friendship of Asif Atiq and Faisal Rahman. I would especially like to thank Asif Atiq for helping me navigate all the procedures and hurdles with being a newcomer in Canada as well as introducing me to the joys of low budget horror movies. I would also like to thank Faisal Rahman for helping me out with housework, cheering me up when I felt down and introducing me to digital drawing. I thank both of them for being my friends, and I consider myself lucky to have them by my side.

To all of the members of the Intelligent Automation in Software Engineering (RAISE) Lab and Dalhousie Transportation Collaboratory (DalTRAC) Lab with whom I have had the opportunity to grow as a researcher. I appreciate their efforts

in making our work in the lab enjoyable and for engaging in meaningful discussions. Their camaraderie and intellectual stimulation have made this journey both rewarding and enjoyable. I would especially like to thank Asif and Mehil from the RAISE Lab for patiently helping me develop a stronger foundation as a researcher. A special thanks also goes to Anik and Venkata from DalTRAC for generously sharing their domain knowledge, which was instrumental in helping me understand the practical aspects of our work.

I would also like to thank the Climate Action Awareness Fund (CAAF) and Natural Sciences and Engineering Research Council of Canada (NSERC) Alliance, NSERC Discovery Grant for their funding support during my graduate study.

Furthermore, I would like to thank Dalhousie University and the Faculty of Computer Science for creating a stimulating learning environment and providing the necessary resources for my research. The outcome of this thesis has been greatly influenced by their drive for intellectual progress and their commitment to excellence. In particular, I would like to thank Dr. Michael McAllister and Megan Baker.

Lastly, I extend my heartfelt thanks to all those who have supported me in various ways throughout this research endeavour. Your encouragement, advice, and assistance have been invaluable, and I am deeply grateful for your presence in my life.

# Chapter 1

## Introduction

### 1.1 Motivation

Code smells are issues that do not prevent a software program from working as expected but indicate deeper problems in the software [1]. They often slow down the development efforts and lead to increased technical debt in software over time. According to studies by Yamashita et al, the presence of code smells can reflect the maintainability challenges of software systems [2] and can hinder the evolution of their source code [3]. Similarly, Abbas et al [4] found that the presence of multiple code smells can make code less understandable. Moreover, according to Jamar et al. [5], Object-Oriented classes with anti-patterns and code clones, a frequent code smell, are three times more likely to contain faults or bugs than non-smelly classes. Furthermore, Alkandari et al [6] found that code smells in Android systems can lead to a 7.1% increase in memory usage as well as a 12.7% increase in CPU usage.

Beyond suggesting deep-rooted issues, code smells can highlight specific roadblocks towards the development of software used in various domains. For example, Hamdi et al [7] contrasted 15 Android-specific and 10 traditional code smells and found that both are very likely to occur simultaneously in the source code. Similarly, Jebnoun et al [8] found that deep learning code involves more complex or longer expressions than the traditional code and is more prevalent across releases. Furthermore, a study by Bessghaier et al [9] on 223 releases of PHP web applications found that large class code smells are more prone to change.

To address code smells, software developers regularly apply refactoring to their code. Refactoring is a small change to the structure of the software programs without affecting their functionality. Kaur et al. [10] show that refactoring code smells can lead to lower Cyclomatic Complexity and improved cohesion within methods and classes, making the software easier to maintain. Kataoka et al [11] found that Extract Method and Extract Class refactoring techniques can result in 16.2% and 12.6% fewer

code smells in the source code, respectively. Furthermore, Zheng et al [12] found that automatically refactoring Android systems could also lead to performance benefits, namely 27% decreased power consumption and a 46% decrease in execution time. Given these benefits, many studies investigate the refactoring practices across various domains, such as database systems [13], [14], [15], android systems [12], [16], [17], and even deep-learning systems [18], [19].

Although the existing work on code smells and refactoring practices advances the current state of knowledge, there is a marked lack of research targeting them in simulation modelling systems.

## 1.2 Problem Statement

Simulation software systems represent a class of software that can imitate real-world processes or systems in controlled, virtual environments. They have found many applications across major domains, including military operations [20], transportation [21], [22], space [23], [24], and medicine [25], [26]. They facilitate useful training without risking human safety [27], [28], support critical decision-making through scenario analysis [29], [30], and allow for modelling complex systems [31], [32]. These systems are also found to be useful in modelling certain real-world scenarios, such as GHG emission modelling and Urban transport modelling [33], [34]. However, despite the strong importance of simulation software systems, they have received limited attention from the software engineering community. These software systems are highly complex by design [35], and thus could be vulnerable to various code quality issues, including code smells and poor design [36].

Code smell has been an active topic of research for decades. There have been many studies on the prevalence [37], [38], [39], evolution [40], [41], [42], and impact [43], [44], [45], [46] of code smells in software systems. A few recent works have also shed light on code smells in various domains, including Machine Learning [8], [47], Android applications [48], [49], and Data Intensive systems [50], [51].

Given their negative effects, refactoring of code smells is a natural choice for software practitioners and researchers. Several studies report the benefits of refactoring code smells. For example, Hecht et al. [52] suggest that refactoring of code smells can lead to 3.6% memory and 12.4% UI performance improvements in Android applications.

Similarly, Kaur et al. [10] show that refactoring code smells can lead to lower Cyclomatic Complexity and improved cohesion within methods and classes, making the software easier to maintain. Chug et al. [53] also demonstrate that refactoring certain code smells, such as Spaghetti Code and Functional Decomposition, can lead to 5-20% reduced defects.

Given the above evidence, both code smells and their refactoring practices are important facets and can provide valuable insights to support the maintenance or evolution of any software system. However, despite their strong importance, there has been a marked lack of research investigating the code smells or the refactoring practices adopted in simulation software systems. This is a major gap in the literature, and our work attempts to fill the gap.

### 1.3 Research Contribution

In this thesis, we conduct two independent but complementary studies targeting code smells and refactoring practices in simulation modelling systems, respectively.

In our first study, we conduct an empirical investigation into the prevalence, evolution, and impact of code smells in simulation software systems. We analyze 155 simulation and 327 traditional software systems, collected from GitHub, using static analysis tools such as Designite [54] and detect code smells. We categorise the code smells into multiple categories according to their abstraction levels (e.g. Implementation, Design, and Architecture) as well as Martin Fowler’s catalogue [55] for our analysis. To examine the evolution of code smells, we select commits at 4-week intervals and analyze their survival over time using the Kaplan-Meier Estimator [56]. Finally, we mine bug-inducing and bug-fixing commits from each simulation system using PyDriller [57] and to investigate any association between code smells and bugs by employing statistical tests such as the Chi-square coefficient and Cramer’s V.

Our findings from the first study reveal significant differences in the distribution of code smells between simulation and traditional software systems. Simulation systems contain 62.77%, 19.1%, and 26.7% more Magic Number, Long Statement, and Long Parameter List code smells, respectively, per line compared to traditional systems. We also observe that a typical code smell in a simulation system lasts 815 more days than its counterpart in traditional systems, with Implementation smells lasting the longest

(median survival time of 3,513 days). The Broken Hierarchy smell has the longest survival time of 3,661 days. Interestingly, while Design and Architecture smells are introduced simultaneously with bugs in simulation systems, our statistical tests show no significant association between code smells and bugs in these systems.

Being intrigued by the findings on code smells, we target the refactoring practices in our second study. In particular, we investigate the effectiveness, risks, and impact of refactoring practices in simulation software systems. We analyze 104 simulation and 272 traditional repositories from GitHub using several state-of-the-art tools: Designite, RefactoringMiner, and JaSoMe. We mine these repositories for specific refactoring commits and their previous versions to analyze refactoring practices. First, we employ Designite [58] to detect code smells in commits before and after refactoring operations, and evaluate the effectiveness of refactoring through comparative analysis. We examine the removal rates of different types of code smells (e.g. Implementation, Design, and Architecture) across various refactoring techniques. Following this, we gauge the risks of refactoring practices by analysing newly introduced code smells after refactoring operations and calculating the smell introduction ratio for each refactoring type. In particular, we focus on identifying risky refactoring patterns that may introduce more smells than they remove. Finally, we also examine the impact of refactoring practices by computing various maintainability metrics with JaSoMe [59] at method level (Cyclomatic Complexity, Lines of Code), class level (Coupling Between Objects, Lack of Cohesion), and package level (Abstractness, Instability) before and after refactoring operations. Then, we statistically analyze the changes in these metrics to quantify maintainability improvements.

Our findings from the second study demonstrate that refactoring in simulation systems removes more code smells than in traditional systems. First, we find that refactoring is especially effective in the long term, reducing the probability of smell survival from 95% to around 65% within five years. In particular, Package-level refactoring techniques are most effective for removing Implementation and Design smells. Second, we find that Method-level refactoring techniques are weakly associated with introducing new code smells. However, some specific techniques such as Split Conditional and Move Code introduce 2.5 times more code smells than they remove.

Furthermore, we find that subsequent refactoring commits increase the risk of introducing new Design and Architectural smells over time, with the risk increasing from 15% to 21% within the same five years. This suggests that while refactoring can be beneficial, it also carries risks that need to be managed carefully. Finally, we find that refactoring significantly improves the maintainability of simulation modelling systems by reducing Structural and Cyclomatic complexity. We also find that refactoring often preserves the core logic integrity and system architectures, as evidenced through stable measures of Data Complexity and Depth of Inheritance trees. These complexity reductions lower technical debt while maintaining algorithmic fidelity, confirming refactoring as a low-risk strategy for maintaining simulation systems.

#### 1.4 Related Publications

Several parts of this thesis have been published or are ready to be submitted to different conferences and journals. We provide a list of papers here. In each of these papers, I am the primary author, and all the studies were conducted by me under the supervision of Dr. Masud Rahman and Dr. Ahsan Habib. While I wrote these papers, the co-authors took part in advising, editing, and reviewing the papers.

- *Riasat Mahbub*, M. Masudur Rahman and Muhammad Ahsanul Habib. *On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software*. In Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2024), pp 154-165, Flagstaff, Arizona, June 2024
- *Riasat Mahbub*, M. Masudur Rahman and Muhammad Ahsanul Habib. *On the Effectiveness, Risks, and Impact of Refactoring Practices in Simulation Modelling Software* Journal of Systems and Software(JSS)(Pre Submission)

#### 1.5 Outline of the Thesis

The thesis contains five chapters in total. To advance our understanding of code smells and refactoring practices in simulation software systems, we conduct two independent but interrelated studies, and this section outlines different chapters of the thesis.

- Chapter 1 introduces the motivation, problem statement, contributions, related publications, and outline of the thesis.
- Chapter 2 provides the necessary background on code smells, refactoring practices, and simulation software systems.
- Chapter 3 discusses our first empirical investigation into the prevalence, evolution, and impact of code smells in simulation software systems.
- Chapter 4 discusses our second investigation targeting the effectiveness, risks, and impact of refactoring practices in simulation software systems.
- Chapter 5 concludes the thesis with a list of directions for our future works.

## Chapter 2

### Background

In this chapter, we introduce the required terminologies and concepts to follow the remainder of the thesis. Section 2.1 describes code smells and their types, whereas Section 2.2 focuses on refactoring and various techniques to perform refactoring. Section 2.3 describes various maintainability metrics. Section 2.4 explains contingency tables for analyzing frequency distributions, Section 2.5 covers statistical significance tests, and Section 2.6 describes association tests for measuring association between variables. Sections 2.7 and 2.8 discuss survival and risk analysis techniques, which predict the chances of survival and gauge the risk of various factors, respectively. On the other hand, Section 2.9 explains trend analysis, the process of extrapolating trends in existing data.

#### 2.1 Code Smells

Code smells indicate deeper issues within a codebase. They are not bugs and do not prevent the software program from functioning correctly. However, they slow down the development process and increase the risk of software bugs or failures. They also increase technical debt over time. Over the years, many static analysis tools (e.g. Designite [54], Understand [60], SonarLint [61]) have been designed to detect code smells in software systems. Traditionally, code smells have been categorized into three groups according to their level of abstraction [62]. We provide a brief outline of these code smells below.

##### 2.1.1 Implementation Smells

Implementation smells indicate issues in the code-level implementation of a software system. Examples of Implementation smells are as follows.

- **Magic Number** refers to hard-coded, unexplained numbers in the source code.

- **Long Statement** refers to a program statement that is too long and hard to understand.
- **Long Parameter List** refers to methods or constructors containing too many parameters (e.g. more than three [63]).
- **Complex Method** refers to a method with high Cyclomatic complexity (e.g. more than 20) [64].
- **Empty Catch Clause** refers to try-catch blocks that do not properly handle an encountered error or exception.
- **Long Method** Refers to a method containing too many lines of code.
- **Missing Default** refers to a missing default clause in a switch statement.
- **Long Identifier** refers to variables with unnecessarily long names.

### 2.1.2 Design Smells

Design Smells indicate design and organization issues in the classes of a software system. Examples of Design smells are as follows.

- **Broken Hierarchy** refers to parent and child classes that do not share an *IS-A* relationship.
- **Broken Modularization** refers to multiple methods being scattered among different classes when they should be kept under a single class.
- **Cyclic Hierarchy** refers to the dependencies of a parent class on its child classes.
- **Deep Hierarchy** refers to excessively long hierarchical chains among the classes.
- **Deficient Encapsulation** refers to classes that have greater access to other classes than what is required.
- **Feature Envy** refers to methods that are more interested in accessing data from other classes than from their own class.

- **Unexploited Encapsulation** refers to client classes that use explicit type-checking via long if-else or switch chains instead of exploiting the *polymorphism* principle.
- **Unutilized Abstraction** refers to an abstraction that is left unused or not directly used.
- **Wide Hierarchy** refers to inheritance hierarchies that have a large number of sub-types at the same level, leading to wide hierarchical structures. Due to the high number of subtypes, most subtypes might implement similar functionality. This might also indicate missing intermediate types, which can be used to implement common functionalities across different subtypes.

### 2.1.3 Architectural Smells

Architectural Smells indicate deeper problems within the architecture of a software system. Examples of Architectural smells are as follows.

- **Cyclic Dependency** refers to two or more abstractions that depend on each other. For example, in a system with components A and B, component A depends on a part of component B, and component B depends on a part of component A. In this situation, components A and B are said to be cyclically dependent.
- **Dense Structure** refers to modules or packages with deep and complex hierarchies.
- **Feature Concentration** refers to components that implement more than one feature.
- **God Component** refers to a component or module that has too many classes and lines of code.
- **Unstable Dependency** refers to a component that depends upon another less stable component.
- **Scattered Functionality** refers to a system where multiple components are responsible for realizing the same high-level concern.

Besides the abstraction level, Fowler et al. [55] also divide the code smells into five categories as follows.

- **Bloaters** are methods, classes, and components that have increased to such an extent that they are very hard to work with. Common examples include Long Statements, Long Methods, Long parameter lists, and Large Classes.
- **Object-Orientation Abusers** refer to code containing incorrect or incomplete applications of Object-Oriented principles. Code smells such as Switch Statements, Refused Bequest, and Unutilized Abstraction are common examples of Object-Oriented Abusers.
- **Change Preventers** refer to poorly designed classes and modules that prevent further code-level changes. Some examples of Change Preventers include Shotgun Surgery and Divergent Change.
- **Dispensables** refer to redundant parts of the code that can be removed without breaking any existing functionality. Common examples include the Dead Code, Code Comments, Duplicate Code, Lazy Class, and Data Class.
- **Couplers** contribute to excessive coupling among classes. Common examples of Couplers include Feature Envy and Inappropriate Intimacy.

## 2.2 Refactoring Techniques

Refactoring is the process of restructuring existing code without changing its external behavior which could lead to improved readability, maintainability, and extensibility of the code. Code smells—indicators of deeper design or implementation flaws—can be addressed using appropriate refactoring techniques. As outlined by Tsatanis et al [65], we divide refactoring techniques by abstraction level (method, class, package) to align with the code elements being refactored.

### 2.2.1 Method-level Refactoring Techniques

Method-level refactoring techniques focus on improving the structure and readability of individual methods. They do not alter a method's external behavior (e.g., return

type, method signatures) but make it's logic more maintainable and flexible.

- **Rename Method:** Renames a method to make its intent clear and improve its code readability.
- **Remove Parameter:** Deletes unnecessary or unused parameters from method signatures.
- **Rename Variable:** Improves code understanding by assigning more meaningful names to existing variables.
- **Add Parameter Modifier:** Adds modifiers to parameters (e.g., `final`) to indicate their constraints.
- **Remove Thrown Exception Type:** Removes unused exception declarations to simplify method signatures.
- **Add Variable Modifier:** Adds modifiers such as `final`, `static`, or `volatile` to variables to control their access behaviors.
- **Extract Method:** Creates a new method by extracting code from an existing method to improve its modularity and reuse.
- **Change Variable Type:** Changes the declared type of a variable to a more general or appropriate type.
- **Rename Parameter:** Renames method parameters to improve clarity and maintain consistency.
- **Change Return Type:** Alters the return type to make a method more general or compatible with the calling code.
- **Inline Variable:** Replaces a variable with its value or expression to simplify code when the variable is redundant.
- **Change Parameter Type:** Changes a method's parameter type to a superclass or interface for generalization.

- **Parameterize Variable:** Converts global constants to method parameters to increase flexibility. For example, instead of always taking  $g=9.81$  as the gravitational constant, we can take it as a parameter to account for different gravity on other planets.
- **Extract Variable:** Introduces a temporary variable to hold a complex expression to enhance code readability.
- **Extract And Move Method:** Extracts a method and moves it to a class where it is more relevant.
- **Invert Condition:** Rewrites a conditional logic in its opposite form to simplify complex conditions.
- **Add Method Annotation:** Adds annotations (e.g., `@Override`) to describe a methods behavior or constraints.
- **Move Method:** Transfers a method to another class to maintain proper responsibility distribution.
- **Change Method Access Modifier:** Modifies visibility (e.g., from `private` to `public`) for better access control.
- **Add Parameter:** Adds a parameter to a method to make it more flexible or to pass necessary data.
- **Remove Parameter Modifier:** Removes modifiers to simplify method parameter declarations.
- **Reorder Parameter:** Changes the order of parameters in the method signature for consistency or clarity.
- **Split Conditional:** Breaks a complex `if` statement into smaller conditional branches for clarity.
- **Remove Variable Modifier:** Removes unnecessary or redundant modifiers to simplify code.

- **Remove Method Annotation:** Deletes annotations that are no longer relevant or necessary.
- **Inline Method:** Replaces a method call with the method body when the method is simple or no longer reused.
- **Add Method Modifier:** Adds modifiers like `static`, `synchronized` to a method for behavioral control.
- **Pull Up Method:** Moves a method from a subclass to a superclass to promote code reuse.
- **Split Method:** Divides a large or complex method into smaller ones to improve modularity.
- **Move And Rename Method:** Moves a method to a new location and renames it to reflect its new context.
- **Merge Method:** Combines multiple methods into a single method to reduce duplication.
- **Move And Inline Method:** Moves a method and makes its implementation inline at the new location to avoid unnecessary method calls.
- **Replace Loop With Pipeline:** Replaces traditional loop constructs (e.g., `for`, `while`) with functional-style stream pipelines.
- **Replace Conditional With Ternary:** Simplifies a conditional logic into a ternary operator when appropriate.
- **Remove Method Modifier:** Removes modifiers such as `abstract` or `static` to change method behavior.
- **Move Code:** Relocates one or more lines of code to a more appropriate method or class to improve structure and cohesion. This refactoring requires careful handling of data and control dependencies to ensure behavior remains correct.
- **Add Thrown Exception Type:** Adds an exception to the method's `throws` clause when new failure scenarios are introduced.

- **Replace Pipeline With Loop:** Reverts functional pipeline chains into traditional loop structures. This can allow for more complex control flow by being easier to debug.
- **Split Parameter:** Divides a single parameter (typically a composite object or structure) into multiple, more specific parameters.
- **Modify Method Annotation:** Updates the method annotations to reflect changes in their requirements or behavior.
- **Push Down Method:** Moves a method from a superclass to its subclasses where it's more appropriate. It can help avoid the refused bequest code smell.
- **Merge Parameter:** Combines multiple parameters into one if they are always passed together or represent a single concept.
- **Merge Conditional:** Consolidates multiple conditionals that lead to the same outcome to reduce duplication.
- **Localize Parameter:** Reduces the scope of a parameter to limit side effects or simplify logic.
- **Replace Anonymous With Lambda:** Converts anonymous classes into lambda expressions for brevity.
- **Merge Variable:** Combines multiple variables into one to reduce redundancy.
- **Change Thrown Exception Type:** Updates the thrown exception types for better granularity or correctness.
- **Try With Resources:** Encloses resource-handling code with the try-with-resources pattern for automatic cleanup.
- **Merge Catch:** Consolidates multiple catch blocks into one to reduce code duplication.

### 2.2.2 Class-level Refactoring Techniques

Class-level refactoring techniques aim at improving the internal structures and external interface of classes. They help make class responsibilities more coherent, promote reuse, and facilitate easier extension and maintenance.

- **Change Class Access Modifier:** Modifies visibility (e.g., from `public` to `package-private`) to improve encapsulation.
- **Rename Attribute:** Improves attribute names to better reflect their role in the class.
- **Change Attribute Type:** Changes the type of a class attribute to a more suitable type.
- **Rename Class:** Assigns a class a more meaningful name to a class to enhance its readability and understanding.
- **Change Attribute Access Modifier:** Adjusts the access level of class attributes (e.g., making a private field protected) to allow or restrict their visibility.
- **Change Type Declaration Kind:** Changes the kind of type (e.g., from class to interface) to better reflect its intended use.
- **Move And Rename Class:** Moves a class to a different package and renames it for better organization.
- **Remove Attribute Modifier:** Removes redundant modifiers (e.g., removing `static` where unnecessary) of a class attribute.
- **Add Attribute Modifier:** Adds modifiers to class attributes such as `static` or `final` to enforce constraints.
- **Extract Superclass:** Moves shared behavior to a new superclass to promote reuse and reduce duplication.
- **Move Attribute:** Relocates an attribute to a class where it belongs in terms of responsibility.

- **Push Down Attribute:** Moves an attribute from a superclass to its relevant subclass(es). It can help avoid the refused bequest code smell.
- **Extract Class:** Breaks a large class into smaller, more focused ones to improve cohesion.
- **Pull Up Attribute:** Moves a duplicated attribute across multiple subclasses to a common superclass.
- **Extract Interface:** Creates an interface to define a common contract among different classes.
- **Move Class:** Relocates a class to another package or module to improve modularity.
- **Add Class Modifier:** Adds class-level modifiers such as `abstract` or `final`.
- **Remove Class Modifier:** Removes unnecessary or incorrect class-level modifiers.
- **Extract Subclass:** Moves specific behavior from a class into a new subclass.
- **Add Class Annotation:** Adds metadata annotations (e.g., `@Entity`, `@Deprecated`) to the class.
- **Extract Attribute:** Separates part of an object's data into a new attribute or object.
- **Split Class:** Divides a class into multiple classes to separate responsibilities and improve maintainability. It can help avoid the Divergent change code smell.
- **Modify Class Annotation:** Updates class annotations to reflect changes in behavior or configuration.
- **Replace Generic With Diamond:** Uses the diamond operator (`<>`) to simplify generic instantiations. So instead of `Map<String, String> map = new HashMap<String, String>()` we can use `Map<String, String> map = new HashMap<>()` to simplify object creation.

- **Add Attribute Annotation:** Adds annotations to fields (e.g., @Inject, @Column) to control their behavior or to document intent.
- **Remove Class Annotation:** Deletes annotations no longer required by the class.
- **Replace Anonymous With Lambda:** Refactors anonymous inner classes with lambda expressions, making overriding existing functions easier.
- **Merge Class:** Combines two related classes into one to reduce overhead and improve cohesion. It can help address the Shotgun Surgery code smell.
- **Collapse Hierarchy:** Flattens a class hierarchy by merging classes when their inheritance is no longer beneficial.
- **Replace Anonymous With Class:** Converts an anonymous inner class into a named class to enhance code readability and testability.
- **Merge Attribute:** Merges multiple fields that serve similar purposes into one better.
- **Replace Attribute:** Replaces an existing attribute with a different one that serves the same purpose.
- **Replace Attribute With Variable:** Converts a class attribute into a local variable within a method to limit scope.

### 2.2.3 Package-level Refactoring Techniques

Package-level refactoring techniques aim to improve the modular structure of a software system by reorganizing or renaming its packages and source folders. They can improve navigation, logical grouping, and maintainability of the codebase.

- **Move Package:** Relocates a package to another module or directory to align with the architectural structure or reduce coupling.
- **Rename Package:** Updates the name of a package to better reflect its purpose or content.

- **Split Package:** Divides a large or complex package into multiple smaller packages to improve cohesion and separation of concerns.
- **Move Source Folder:** Relocates an entire source directory, often for restructuring modules or standardizing project layout.
- **Merge Package:** Combines two or more packages into one when their responsibilities overlap or are too fine-grained.

## 2.3 Maintainability Metrics

Maintainability metrics are quantitative measures that often reflect how easily software can be understood, modified, extended, and corrected. They provide objective insights into code complexity, cohesion, coupling, and other structural properties that influence long-term software quality.

### 2.3.1 Method Level Metrics

Method-level metrics help us measure the impact of both local and global changes on the maintainability of individual methods within a software system. A list of these metrics are given below:

- **Cyclomatic Complexity (VG)** Number of independent paths through a method's source code.
- **Nested Block Depth (NBD)** Maximum depth of nesting within a method.
- **Fan-in** Number of other methods that call the target method.
- **Fan-out** Number of other methods called by the target method
- **Structural Complexity (Si)**  $Fan-out^2$  per method.
- **Data Complexity (Di)**  $\frac{IOVars}{Fan-out+1}$  per method, where IOVars is the number of input/output variables.
- **System Complexity (Ci)**  $Si + Di$  per method.

- **TLOC** Total lines of code per method.
- **NCOMP** Number of comparisons (conditional checks) per method.

### 2.3.2 Class Level Metrics

Class-level metrics help us gauge the impact of both global and local changes on the maintainability of a higher-level abstraction, namely classes or interfaces. We use the metrics defined by Chidamber and Kemerer [66] to measure class level metrics. A brief description of these are given below:

- **Weighted Methods per Class (WMC)** Sum of cyclomatic complexities of all methods within a class.
- **Depth of Inheritance Tree (DIT)** Depth of a class in the inheritance hierarchy.
- **Lack of Cohesion in Methods (LCOM)** Measure of dissimilarity between class methods.
- **Class Total Complexity (CITCi)** The sum of the system complexity values of all methods within a class. It reflects the overall complexity of a class.
- **Class Relative Complexity (CIRCi)** The average complexity per method in the class. It provides insights into the complexity of individual methods within a class.
- **Number of Children (NOCh)** Number of direct subclasses of a class.

### 2.3.3 Package Level Metrics

Package-level metrics capture the top-down architectural concerns across all the components of a software system. For our purposes, we use the metrics defined by Robert C. Martin [67] as our Package level metrics. A brief description of these are given below:

- **Afferent Coupling (Ce)** Number of external classes dependent on the package.

- **Efferent Coupling (Ca)** Number of package classes dependent on external classes.
- **Instability (I)**  $Ce + Ca$  (package dependency index).
- **NOC** Number of classes in package.
- **NOI** Number of abstract classes and interfaces in package.
- **Abstractness (A)** Ratio of Number of abstract classes(NOA) and interfaces(NOI) to the total number of classes in the package (NOC),  $\frac{NOI}{NOC}$ .
- **Package Total Complexity (PkgTCi)** Sum of system complexity values across all methods within a package.
- **Package Relative Complexity (PkgRCi)** Average of system complexity values across all methods within a package.

## 2.4 Contingency Table

Contingency Tables show the frequency distribution of multiple variables using a matrix representation. Each cell of a contingency table represents the Observed Frequency  $O_i$  of a combination of variables. Table 2.1 shows an example of a contingency table for two categorical variables – Smoker (Smoker and Non-Smoker) and Lung Cancer (Lung Cancer and No Lung Cancer). Here, we have 50 Smokers and 30 Non-Smokers with Lung cancer as well as 70 Smokers and 150 Non-Smokers without any Lung Cancer. These values show the frequency of both Smokers and Non-Smokers with and without Lung cancer, where each cell represents the Observed Frequency  $O_i$  for both variables.

Table 2.1: An example of contingency table

	Lung Cancer	No Lung Cancer	Total
Smoker	50	70	120
Non-Smoker	30	150	180
Total	80	220	300

## 2.5 Statistical Significance

The null hypothesis suggests that there is no relationship between two samples under analysis [68]. In other words, if the hypothesis is true, then any apparent relationship between them can be left up to chance alone. A statistical significance test must be conducted to determine whether to reject or accept the null hypothesis. In this thesis, we conduct two types of statistical test as discussed below.

### 2.5.1 Mann-Whitney U Test

Mann-Whitney U test [69] is a non-parametric statistical test that compares two independent samples for statistical significance without assuming their underlying distribution. The test gives a  $p$  value that measures the probability of the null hypothesis being true. The obtained  $p$  values are usually contrasted against a predefined significance threshold  $\alpha$ , which is generally accepted as 0.05 [70].

### 2.5.2 Cliff's Delta

Cliff's delta [71] is another non-parametric statistical test that quantifies the differences between two samples and delivers a value between -1 and +1. The values closer to +1 indicate that all items in the first sample are higher than those of the second sample, while values closer to -1 indicate the opposite. On the other hand, values closer to 0 indicate little to no difference between the two samples. The Equation 2.1 shows that the outcome  $\delta$  can be calculated using the number of items,  $n1$ , and  $n2$ , from two samples, respectively. Here,  $\delta_{ij}$  represents the number of times one sample's value exceeds the other.

$$\delta = \frac{1}{n1 \times n2} \sum_{i=1}^{n1} \sum_{j=1}^{n2} \delta_{ij} \quad (2.1)$$

The value of  $\delta$  ranges from -1 (complete dominance of group Y) to +1 (complete dominance of group X), with 0 indicating complete overlap between the groups. The absolute value  $|\delta|$  is commonly interpreted using the following thresholds [72]:

Table 2.2: Interpretation of Cliff’s delta effect sizes

<b>Effect Size</b>	<b>Range of <math> \delta </math></b>
Negligible	$ \delta  < 0.15$
Small	$0.15 \leq  \delta  < 0.33$
Medium	$0.33 \leq  \delta  < 0.47$
Large	$ \delta  \geq 0.47$

## 2.6 Association Tests

Association tests are statistical tests that determine if there is a significant relationship between two or more categorical variables. They help us understand whether the observed frequencies of combinations of categories differ from what would be expected if the variables were independent.

### 2.6.1 Chi Squared Test

We use Pearson’s Chi-Squared test [73] to find if two categorical variables are associated or not. This test uses the contingency table (see Section 2.4) to capture the observed frequencies  $O_i$  and the expected frequencies  $E_i$  of two categorical values. Here,  $O_i$  is the number of times a particular category was measured while  $E_i$  refers to the frequency of a combination assuming that both variables are independent. From Equation 2.2, we see that the expected frequencies  $E_i$  can be calculated by dividing the product of *row* (i.e sum of frequencies in each row) and *col* (i.e sum of frequencies in each column) by  $n$ , the total number of samples from all categories. As seen in Equation 2.3, we derive the Chi-squared statistic  $\chi^2$  from the Observed frequency  $O_i$  and expected frequency  $E_i$  for each variable.

$$E_i = \frac{row \times col}{n} \quad (2.2)$$

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i} \quad (2.3)$$

We use a significance threshold of 0.05 for the Chi-squared test, which is a common threshold for statistical significance [70]. If the  $p$  value is less than 0.05, we reject the null hypothesis and conclude that there is a statistically significant association

between the two variables. If the  $p$  value is greater than 0.05, then the null hypothesis cannot be rejected.

### 2.6.2 Cramer's V Test

Pearson's Chi-squared test indicates whether two categorical variables are associated or not. However, it does not tell us how strong the association is. To find the strength of associations, many existing studies [74], [75], [76] use the Cramer's V test [77]. The Cramer's V test returns a value between 0 and 1, where 0 indicates no association and 1 indicates complete association. In Equation 2.4, we see how Pearson's Chi-square  $\chi^2$  is used to derive the value of Cramer's V. It is divided by the product of  $n$  (total samples) and the smaller number between  $row - 1$  and  $col - 1$ .

$$V = \sqrt{\frac{\chi^2}{n \times \min(row - 1, col - 1)}} \quad (2.4)$$

The value of  $V$  ranges from 0 (No association between variables) to 1 (perfect association between variables). We interpret the value of  $V$  using the the following thresholds [78]:

Table 2.3: Interpretation of Cramer's V association strengths

Association Strength	Range of V
Weak	$V < 0.2$
Moderate	$0.2 \leq V < 0.6$
Strong	$V \geq 0.6$

## 2.7 Survival Analysis

Survival analysis is a statistical technique that analyzes the expected delay of an event. An event can be anything that is clearly defined, such as death during a hospital stay or mechanical failure in a machine. To determine the probability of survival, events experienced by subjects within a given period must be tracked. If the subjects do not experience any event or leave before the event within the period of observation, then the event is censored at the end of the period.

Kaplan-Meier Estimator [79] is a survival analysis technique that returns a probability of survival given a time to event and the status of a subject. The *time to event* is defined as the interval between the start of observation and the occurrence of an event. The time interval can be measured in any positive unit. On the other hand, *status* is a boolean value that indicates whether a subject has experienced an event or the data is censored. *The survival function*  $S(t)$  returns the probability of survival of a subject after time  $t$ . As shown in Equation 2.5, the *survival function*  $S(t)$  is calculated within the time when at least one event occurred ( $t_i$ ). Here,  $d_i$  is the number of events that occurred within the time  $t_i$  and  $n_i$  is the number of individuals that survived up to time  $t_i$ .

$$S(t) = \prod_{i:t_i \leq t} \left[1 - \frac{d_i}{n_i}\right] \quad (2.5)$$

We use the Kaplan-Meier Estimator in both first and second studies. We employ this non-parametric analysis method since the underlying distribution of code smells in these systems might be unknown. Similarly, since the relative risks of removing or introducing code smells might not remain constant over time, we cannot use semi-parametric tests such as Cox-proportional test [80].

## 2.8 Nelson-Aalen Estimator

The Nelson-Aalen Estimator is a non-parametric method to estimate the cumulative hazard function over time [81]. Unlike the Kaplan-Meier Estimator, which focuses on estimating the survival probability, the Nelson-Aalen approach quantifies the accumulated risk of experiencing the event up to a certain point in time.

The Nelson-Aalen Estimator provides the cumulative hazard function, denoted as  $\hat{H}(t)$ , which estimates the total hazard experienced by a subject from the start of observation up to time  $t$ . It is defined as the sum of hazard contributions from each event time  $t_i$  where at least one event occurred. At each such time  $t_i$ ,  $d_i$  represents the number of observed events, and  $n_i$  is the number of subjects at risk immediately before  $t_i$ .

The cumulative hazard function is computed as shown in Equation 2.6:

$$\hat{H}(t) = \sum_{i:t_i \leq t} \frac{d_i}{n_i} \quad (2.6)$$

We use the Nelson-Aalen function in our second study, in particular, to find the cumulative risk of introducing new code smells by refactoring activities in simulation systems.

## 2.9 Mann Kendall Test

The Mann-Kendall test [82] is a non-parametric statistical method for identifying monotonic trends in time-series data. It assesses whether later observations systematically exceed or precede earlier values.

$$S = \sum_{k=1}^{n-1} \sum_{j=k+1}^n \text{sgn}(X_j - X_k) \quad (2.7)$$

Here,  $X$  represents the observations across the timestamps  $j$  and  $k$ . We use the test statistic  $S$  to find the difference between the number of increasing pairs (i.e. pairs where  $X_j > X_k$ ) and decreasing pairs (i.e. pairs where  $X_j < X_k$ ). Positive values of  $S$  indicate an upward trend, while negative values indicate a downward trend. However, values close to 0 suggest no monotonic trends. On the other hand, for large sample sizes ( $n > 10$ ), the distribution of  $S$  begins to follow a normal distribution due to the central limit theorem.

The test statistic  $S$  compares the number of increasing and decreasing pairs. A positive  $S$  indicates an upward trend, while a negative  $S$  suggests a downward trend. Values near zero imply no clear trend. For large sample sizes ( $n > 10$ ), the distribution of the test statistic  $S$  approximates a normal distribution under the null hypothesis of no trend. To facilitate hypothesis testing,  $S$  is standardized to a  $Z$ -score using its expected value and variance. This transformation allows us to use the standard normal distribution to calculate p-values, making it easier to determine the statistical significance of trends. Standardizing also puts results on a common scale, enabling consistent interpretation across different datasets. This  $Z$  statistic follows a similar pattern where positive values exhibit increasing trends while negative values describe decreasing trends.

$$Z = \begin{cases} \frac{S-1}{\sqrt{\text{Var}(S)}} & \text{if } S > 0 \\ 0 & \text{if } S = 0 \\ \frac{S+1}{\sqrt{\text{Var}(S)}} & \text{if } S < 0 \end{cases} \quad (2.8)$$

We use the Mann-Kendall test in our second study to find the trends for maintainability metrics after refactoring simulation systems. As we don't know the underlying distribution of software metrics in simulation systems, which can even be non-linear or irregular, we require mathematically robust models to detect these trends. This makes the non-parametric Mann-Kendall test the best choice for our study, as it does not make any underlying assumptions about the distribution of maintainability metrics.

### 2.9.1 Stouffer's Method

To determine the overall significance of multiple independent hypothesis tests, we use Stouffer's method [83]. It transforms the p-values from each test into a z-score using the inverse cumulative distribution function. The individual z-scores are then averaged (normalized by the square root of the number of tests) to produce a combined z-score as shown in Equation 2.9.

$$Z = \frac{\sum_{i=1}^k \Phi^{-1}(1 - p_i)}{\sqrt{k}} \quad (2.9)$$

Here,  $k$  is the number of tests and  $\Phi^{-1}$  is the inverse Cumulative Distribution Function of the standard normal distribution. The resulting  $Z$  score follows a standard normal distribution under the null hypothesis and can be converted back to a p-value using the standard normal Cumulative Distribution Function. Then the p-value can be used to determine the statistical significance of those tests.

## 2.10 Summary

In this chapter, we introduce the required terminologies and concepts to follow the remainder of the thesis. Section 2.1 and 2.2 focus on code smells and refactoring techniques respectively. Section 2.3 describes various maintainability metrics. Sections 2.4, 2.5, and 2.6 cover contingency tables, statistical significance tests, and association

tests for analyzing association between variables. Sections 2.7 and 2.8 discuss survival and risk analysis techniques, while Section 2.9 explains trend analysis using the Mann-Kendall test. In the next chapter, we present our first study discussing the prevalence, evolution, and impact of code smells in simulation software systems.

## Chapter 3

# On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software

Simulation models imitate complex real-world systems and are widely used in research, engineering, and industry. Due to their complexity, simulation software systems often suffer from code quality issues (e.g. code smells), yet no prior work has examined their issues in depth. In this chapter, we present the first empirical study on the prevalence, evolution, and impact of code smells in simulation software systems.

The rest of this chapter is organized as follows. Section 3.1 introduces our study and discusses the novelty of our work. Sections 3.2 and 3.3 discuss our methodology and analyze the results. Sections 3.4 and 3.5 present the implications of our results and discuss threats to validity. Section 3.6 discusses related works that examine the prevalence, evolution and impact of code smells. Finally, Section 3.7 presents a summary of our study.

### 3.1 Introduction

Code smells are symptoms that indicate deeper code quality issues in software systems [55]. They do not prevent a software program from working correctly, but increase its technical debt over time. They could also lead to performance issues [84], lack of understandability [4], and lack of maintainability in software systems [2], [3]. According to a study by Jaafar et al. [5], Object-Oriented classes with anti-patterns and code clones are three times more likely to contain faults or bugs than non-smelly classes. Moreover, Hecht et al. [84] suggest that refactoring of code smells can lead to 3.6% memory and 12.4% UI performance improvements in Android applications. Thus, the study of code smells has been an active research topic in the software engineering community [85], [86], [87].

Over the last two decades, there have been numerous studies on the prevalence [37], [38], [39], evolution [40], [41], [42], and impact [43], [44], [45], [46] of code smells in

software systems. Several recent works focus on domain-specific code smells, including code smells from Machine Learning [7], [8], [47], Android [48], [49], and Data Intensive systems [50], [51]. These works, a source of our inspiration, shed light on domain-specific issues relating to code smells and reveal their unique maintenance challenges. However, to the best of our knowledge, no prior work focuses on investigating the code smells in simulation modelling software.

Simulation modelling provides an imitative representation of real-world processes or systems in a controlled, virtual environment. It has found numerous applications in many critical areas, including aviation, transportation, and medicine, which help their stakeholders with training, testing, and decision-making. Simulation modelling systems are also highly complex due to their extensive domain logic, which abstracts real, physical systems [88]. As a result, they could be susceptible to various quality issues including code smells. An investigation of code smells in simulation software systems can reveal important insights about their code quality issues and maintenance challenges.

In this chapter, we conduct an empirical study to investigate the prevalence, evolution, and impact of code smells found in simulation software systems. Our study relies on the analysis of 155 simulation and 327 traditional software systems collected from GitHub. We apply static analysis tools (e.g. Designite [54]) to detect code smells in these systems. We also categorize the code smells into multiple categories according to their abstraction levels as well as Martin Fowler’s catalog [55] for our analysis. Through our experiments, we answer three important research questions.

- (a) **RQ1: Do simulation software systems smell like traditional software systems?** We use open-source static analysis tools (e.g. Designite [54]) and determine the prevalence of code smells in both simulation and traditional software systems. From our experiments, we find that there are significant differences in the distribution of code smells between simulation and traditional software systems. In particular, simulation systems contain 62.77%, 19.1%, and 26.7% more *Magic Number*, *Long Statement*, and *Long Parameter List* code smells, respectively per line compared to those of traditional systems. On the other hand, several smells such as *Unutilized Abstraction*, *Feature Concentration*, and *Broken Modularization* are more prevalent in traditional software systems.

(b) **RQ2: How long do code smells last in simulation software systems?**

We examine the evolution of code smells throughout the development stage of simulation software systems. Specifically, we inspect the probability and duration of survival of each type of code smell in simulation software systems and contrast them with their counterparts from traditional software systems. We observe that a typical code smell in a simulation system lasts 815 more days than its counterpart in traditional systems. Moreover, the *Implementation smells* last the longest in simulation systems with a median survival time of 3,513 days. Our findings also show that most code-level problems are ignored during development.

(c) **RQ3: Do code smells co-occur with bugs in simulation software systems?**

We mine both bug-inducing and bug-fixing commits from our collected systems using an open-source tool namely *PyDriller* [57], and examine the code smells from the mined commits. We conduct appropriate statistical tests (e.g. Chi-square coefficient and Cramer’s V) to determine if code smells have any association with software bugs. We observed no significant association between the occurrence of code smells and bugs in simulation software systems.

### 3.2 Methodology

Figure 3.1 shows the schematic diagram of our conducted study. First, we select hundreds of repositories hosting simulation and traditional software systems from GitHub. After performing the necessary data preprocessing and cleanup, we detect code smells in the repositories using static analysis tools. Then, we perform a comparative study on the prevalence of code smells to answer our first research question. We also select commits from simulation software to perform a survivability analysis of the code smells. Finally, we attempt to find the association between code smells and bugs in the simulation systems. In the following sections, we discuss different steps of our methodology.

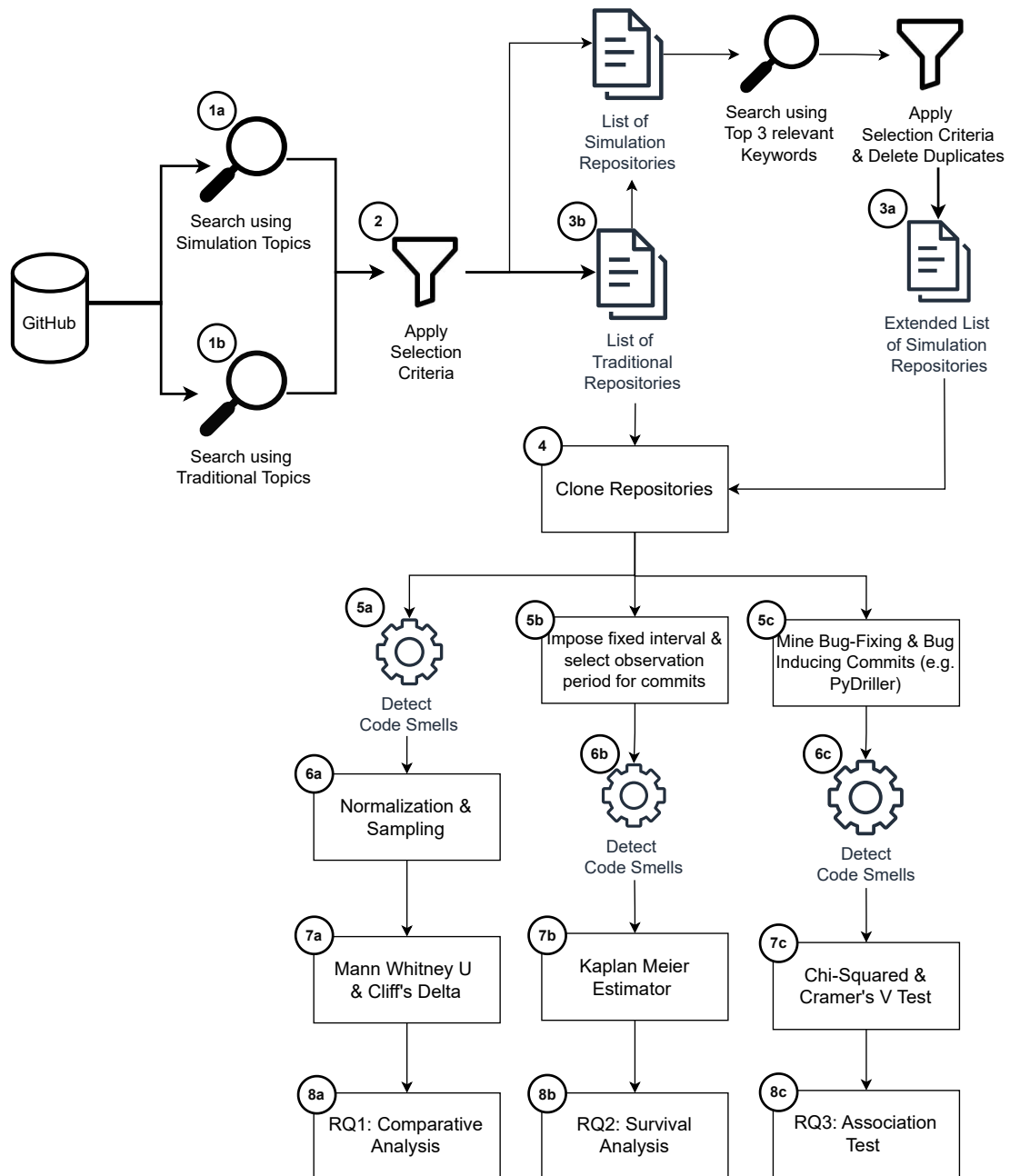


Figure 3.1: Schematic diagram of our empirical study

### 3.2.1 Project Selection Criteria

We use GitHub as a source of our simulation and traditional software systems. We only select the repositories that meet the following criteria:

1. We only consider repositories with *open-source licenses* (MIT, GPL, and Apache licenses) as they do not require explicit permissions for any third-party re-use.
2. We do not consider any repositories *disabled* by the repository owner since we do not have permission to access them.
3. We do not consider *archived* repositories since they have stopped all active development.
4. We do not consider any *forks* of original repositories. This minimizes code duplication as forked projects share code with their original repository.

In addition to automatic filtering, we also manually analyze the documentation (e.g., README) of each automatically captured repository and attempt to reduce false positives. We note that, simulation repositories from popular languages such as C++ and Python contain a large number of false positives even after filtering efforts. This results in a significant imbalance between the quality and quantity of simulation systems compared to their traditional counterparts, making comparisons difficult. So, we use only Java systems to justify the comparisons between simulation and traditional repositories with similar quality and quantity. This results in a final dataset of 327 traditional and 155 simulation software repositories.

### 3.2.2 Data Collection

As shown in Step 1a and 1b of Figure 3.1, we use the GitHub search API [89] to collect repositories using GitHub Topics and keywords. First, we search using the following topics – *simulation*, *simulator* and *simulation modelling* – and collect a list of simulation systems. Similarly, we use several traditional topics – *web*, *framework*, *android*, *HTTP*, and *desktop* – to collect a diverse set of traditional software systems. We also apply the selection criteria above (Step 2) and collect 67 simulation systems and 327 traditional systems.

Since we have significantly less number of simulation systems, we augment our existing list of simulation systems by following the process shown in Step 3a of Figure 3.1. First, we retrieve the top three keywords (e.g. simulation, modelling, cloudsim) by analyzing the README documents of the retrieved simulation systems and using TF-IDF and RAKE scores [90]. Then we perform a keyword search using these keywords to collect 775 new repositories. After applying our selection criteria above and manually analyzing repositories, we get a total of 88 new simulation systems, which leads to a total of 155 simulation systems. Finally, as shown in Step 4, Figure 3.1, we clone the final list of 155 simulation and 327 traditional systems for our analysis.

### 3.2.3 Comparative Analysis

As shown in Step 5a of Figure 3.1, we start our comparative analysis by detecting code smells in simulation and traditional systems using the Designite tool [54]. It gives us a detailed report on code smells for each repository using the same thresholds and heuristics for both simulation and traditional systems. This makes comparisons between traditional and simulation systems more meaningful, at the cost of avoiding code smells specific to simulation systems.

Since repositories could have different sizes, to make the comparison fair, we normalize the number of smells in each repository against the total lines of code. This gives us an average number of smells from each repository, which accounts for repository size. After normalization, we perform a random sub-sampling on traditional software repositories (Step 5b, Figure 3.1) to equalize the number of simulation modelling and traditional software systems. We then perform the *Mann-Whitney U test* (see Section 2.5) to determine if there are any significant differences in the distribution of smells between the simulation and traditional systems.

### 3.2.4 Survivability Analysis

For our survivability analysis, we first collect all commits from each repository. Since we have 327 traditional systems against 155 simulation systems, we take a random sub-sample of 155 traditional systems for our analysis to reduce bias. Furthermore, since our selected simulation and traditional repositories contain thousands of commits, analyzing them all is computationally expensive. To remedy this, we use Sas et al's

Table 3.1: Keywords for detecting bug-fix commits

Keyword	Detection of bug-fixing Commits
fix	44.67%
fixed	24.19%
bug	9.35%
issue	4.18%
except	3.96%

[91] technique to select one commit every 4 weeks, which reduces the computational complexity. We then select the observation period by retrieving the earliest and latest commit times common to both simulation and traditional systems. This leaves us with an observation period of 4,949 days (Step 5b, Figure 3.1).

After selecting our observation period, we ran the Designite tool on our selected commits to detect their code smells. This is done by using the *multi-commit analysis* mode to analyze multiple selected commits at once [92] (Step 5b, Figure 3.1). Then we checked if any code smell was able to survive until the last commit by observing their probabilities of survival. To calculate the probability of survival of each smell, we use the Kaplan-Meier Estimator (see Section 3.2.4). We collect the estimates for each type and sub-type of code smell across multiple repositories and determine their chances of survival during the development or evolution of a system.

### 3.2.5 Mining Bug Inducing and Fixing Commits

To determine any association between software bugs and code smells in simulation systems, their buggy and bug-fix commits should be analyzed. To this end, we use an appropriate list of keywords (e.g. Table 3.1) and find the bug-fix commits (Step 5c, Figure 3.1). Table 3.1 shows how each keyword helped us identify the bug-fix commits. These keywords were widely used by the existing works of Antoniol et al. [93], Mockus et al. [94], and Zhong et al. [95] to find bug-fix commits. We search for these keywords in the title of each commit to detect the bug-fix commits from each repository.

After capturing the bug-fixing commits, we then use PyDriller [57] to find the corresponding bug-inducing commits. bug-inducing commits inject the bugs in the source documents that are later resolved by the bug-fixing commits. PyDriller relies

on the SZZ algorithm [96] for its operation. The SZZ algorithm looks for potential bug-inducing commits by selecting bug-fixing commits and traversing through the version history to find commits that introduced changes in the currently fixed code.

### 3.2.6 Association between Code Smells and Bugs

As shown in Step 6c of Figure 3.1, we run Designite on both the bug-fixing and bug-inducing versions of code using the mined bug-fixing and bug-inducing commits. We use Designite’s *multi-commit analysis* mode to analyze both the bug-inducing and bug-fixing versions of the code to detect their code smells. After analyzing code smells in each bug-fixing and bug-inducing version, we organize the data into a contingency table with high and low frequencies of code smells for bug-inducing and bug-fixing categories (see Section 2.4). Here, we discretize the frequencies using quantile binning to account for any imbalanced distribution of code smell frequencies [97]. Then we calculate the Observed and Expected Frequencies  $O_i$  and  $E_i$  from our contingency table (see Section 2.4) to find any associations between them.

After getting our Observed and Expected Frequencies  $O_i$  and  $E_i$ , we perform the Pearson’s Chi-square test and Cramer’s V test respectively as outlined in Step 7c of Figure 3.1. Here, Pearson’s Chi-square test is used to determine the association between code smells and software bugs while Cramer’s V test shows the strength of the association between code smells and buggy code.

According to an existing survey of Cairo et al [45], most studies suggest that the presence of code smells leads to more bugs in traditional software systems. Thus, their relationship was well established for traditional software systems. However, this relationship was not as clear for simulation systems, leading to us only using simulation systems for our experiments.

### 3.2.7 Replication Package

We made our replication package [98] publicly available for any third-party reuse or replication.

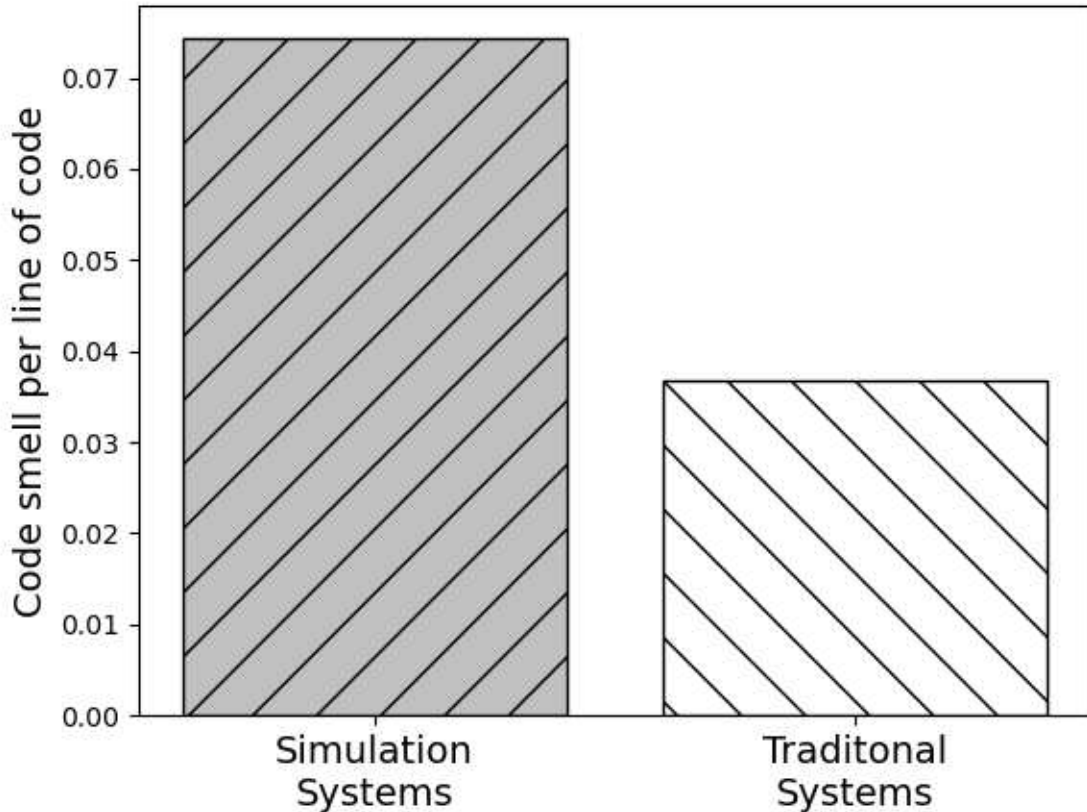


Figure 3.2: Prevalence of code smells in simulation and traditional systems

### 3.3 Results and Analysis

#### 3.3.1 RQ1: Do simulation software systems smell like traditional software systems?

After collecting the code smell statistics from simulation and traditional systems, we compare their prevalence of code smells. In particular, we normalize the code smell frequencies against the lines of code (LOC) in each repository for our comparative analysis.

As shown in Figure 3.2, simulation software systems have more code smells per line than traditional ones. It indicates that simulation systems are more prone to code smells, with a significantly higher median than their traditional counterparts (check Figure 3.3). Upon further analysis, we find that not only do simulation systems have more code smells, but also their distribution is different from that of traditional systems

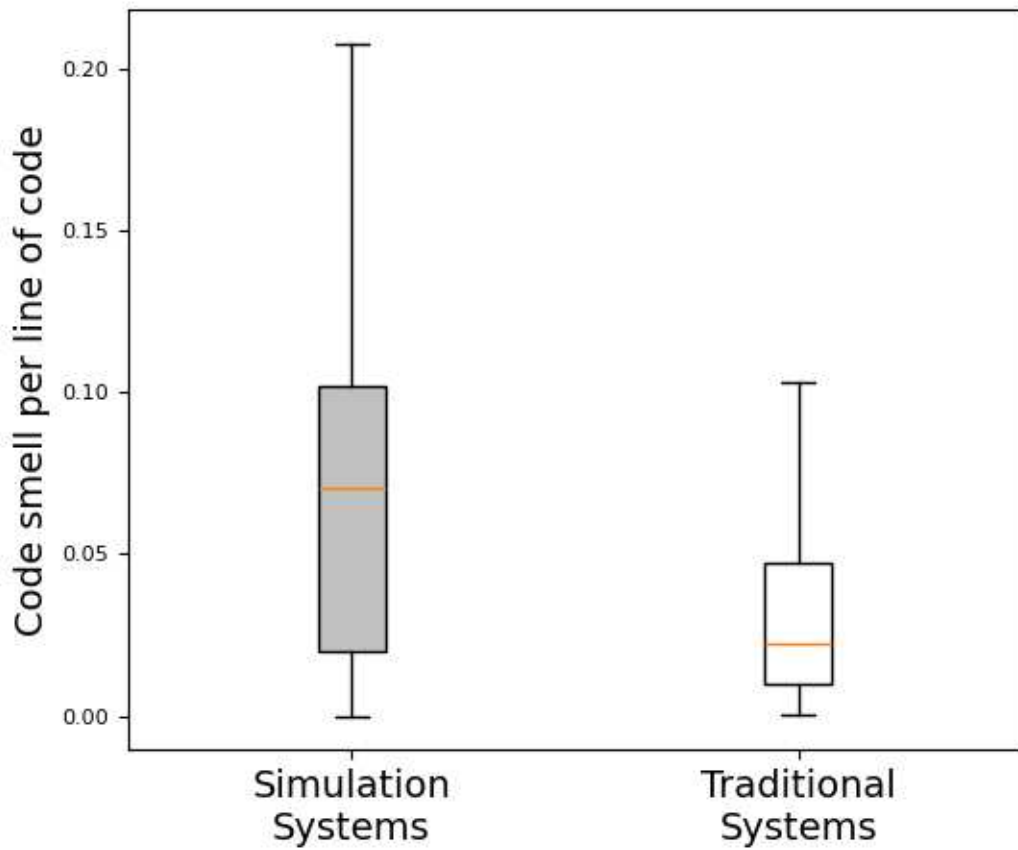


Figure 3.3: Distribution of normalized frequencies of code smells in simulation and traditional systems

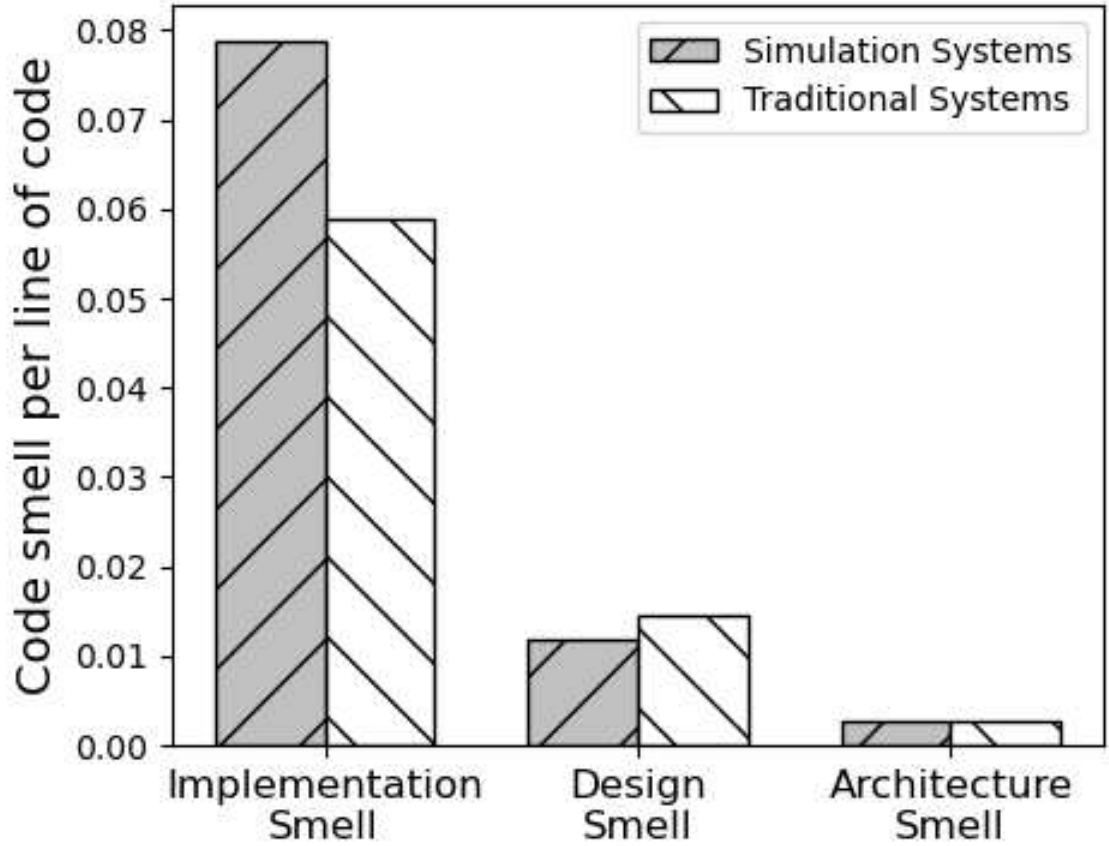


Figure 3.4: Prevalence of normalized frequencies of code smells in simulation and traditional systems (by smell type)

(Figure 3.3). This is supported by our non-parametric Mann-Whitney U test, which shows a p-value of  $8.61 \times 10^{-12}$ , denoting a statistically significant difference between the distribution of code smells. We also found Cliff’s delta,  $\delta = 0.44$ , suggesting that simulation systems have significantly more code smells per line than traditional systems with a *medium* effect size.

To achieve an in-depth insight, we analyze the prevalence of each category of code smells. As shown in Figure 3.4, simulation systems have a larger share of implementation code smells than their traditional counterparts. When frequency distribution is considered, Figure 3.5, shows a bigger median for *Implementation* code smells in simulation systems than that of traditional repositories. This observation is supported by a p-value of  $4.05 \times 10^{-9}$  and Cliff’s  $\delta$  of 0.4569. They suggest that the difference is statistically significant with simulation systems containing more

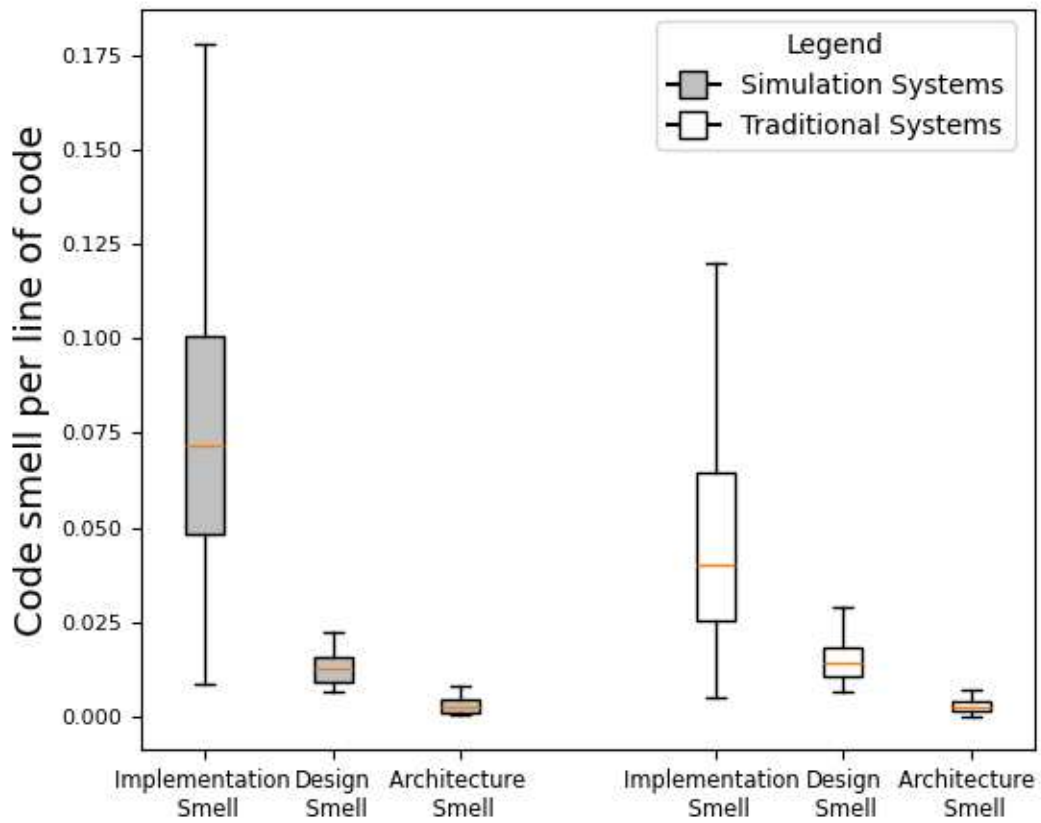


Figure 3.5: Distribution of normalized frequencies of code smells in simulation and traditional systems (by smell type)

Table 3.2: Significance tests for code smell’s prevalence

Code Smell	Cliffs $\delta$	p-value
Long Parameter List	0.1748	0.0231
Long Statement	0.1518	0.03703
Magic Number	0.5001	$1.19 \times 10^{-8}$
Empty Catch Clause	-0.4004	$8.25 \times 10^{-5}$
Unutilized Abstraction	-0.6384	$3.24 \times 10^{-9}$
Broken Modularization	-0.2783	0.0039
Feature Concentration	-0.3284	$6.59 \times 10^{-6}$

implementation smells per line of code. Similarly, the distribution of Design smells is also different in both simulation and traditional repositories, as suggested by a p-value of 0.00634 and Cliff’s  $\delta$  of  $-0.1859$ . Although this difference is much smaller, it still shows that simulation programs are home to fewer design smells than traditional repositories. Finally, the number and distribution of architectural smells in both repositories are similar. This is evidenced by the p-value of 0.6917 and Cliff’s delta of  $-0.0311$ , which suggest negligible differences between the prevalence of smells in both systems.

We also compare the prevalence of each smell individually between the two types of systems. We employ Mann-Whitney-U, and Cliff’s delta tests and Table 3.2 summarizes our comparative analysis. From the 33 types of detected smells, only 7 were found to have a significantly different distribution ( $p - value < 0.05$ ). As shown in Table 3.2, the *Long Parameter List*, *Long Statement*, *Magic Number*, and *Empty Catch clause* belong to the Implementation Smell category, which confirms our findings in Figures 3.4, 3.5. On the other hand, the *Unutilized Abstraction* and *Broken Modularization* smells belong to the *Design Smell* category, whereas *Feature Concentration* is a type of *Architecture Smell*. We also see that the *Long Parameter List*, *Long Statement* and *Magic Number* code smells have positive values of Cliff’s delta. This indicates that these smells are more prevalent in simulation systems than in traditional systems. On the other hand, the *Empty Catch Clause*, *Unutilized Abstraction*, *Broken Modularization*, and *Feature Concentration* code smells all have negative values of Cliff’s delta, indicating their higher prevalence in traditional software systems.

From Figure 3.6, we see that the Magic Number and Long Statement smells

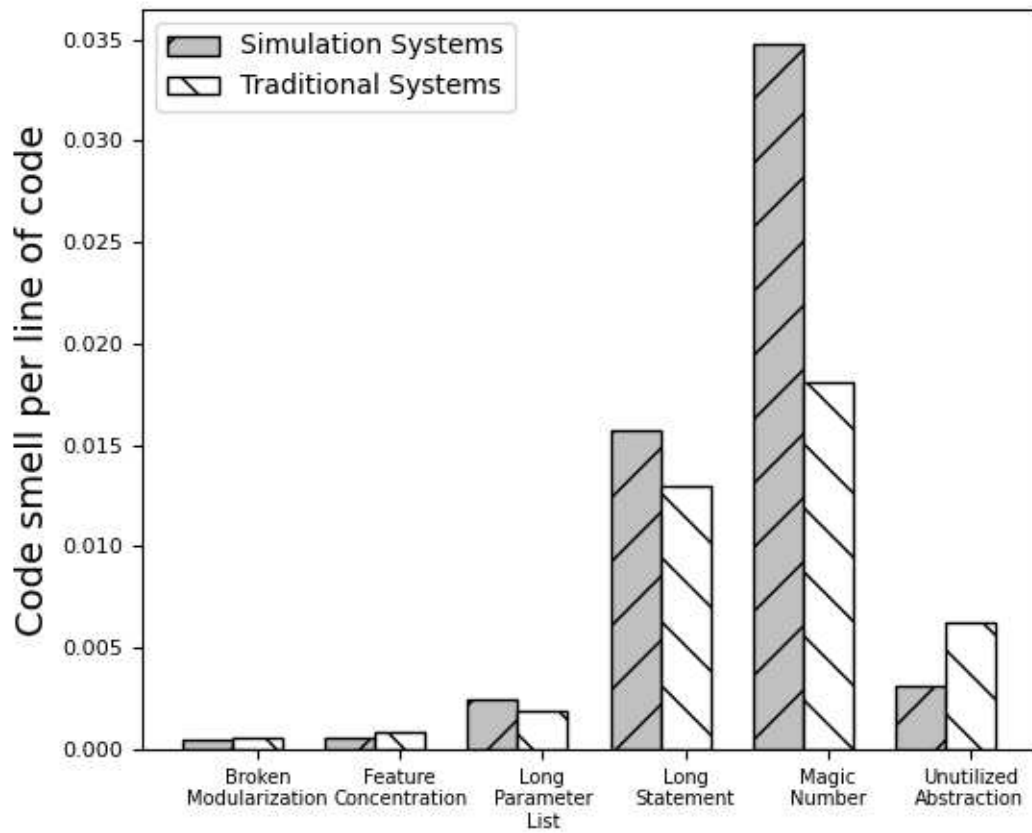


Figure 3.6: Prevalence of normalized frequencies of code smells in simulation and traditional systems (for significantly different code smells)

occur 62.77% and 26.72% more times per line of code (LOC) in simulation systems, which supports our prior observation from Table 3.2. On the other hand, the Broken Modularization, Feature Concentration, and Unutilized Abstraction code smells occur 24.56%, 38.22%, and 66.42% less times per LOC respectively. According to the above findings, simulation systems are more resistant to Design and Architectural code smells but significantly more vulnerable to implementation smells. Thus, while developing simulation software systems, developers should continuously refactor their implementation code to prevent the accumulation of technical debt throughout the system.

**Summary of RQ1:** We observe that code smells are more prevalent in simulation systems compared to traditional systems (Figure 3.2). This is especially true for *Implementation Smells* such as *Magic Number* and *Long statement*, which occur much more frequently in simulation systems (Figure 3.6). These findings suggest that developers should adopt practices such as well named constants and properly structured classes in simulation systems to avoid introducing code smells.

### 3.3.2 RQ2: How long do code smells last in simulation systems?

We perform our survivability analysis on 155 simulation and 327 traditional systems by first selecting commits with a 4-week interval. This leaves us with 2,263 and 5,402 commits from simulation and traditional systems respectively. Since we have more traditional systems, we use a random subsample of 155 traditional systems, leaving us with around 3007 traditional system commits. We select our observation period of 4,949 days by selecting common start and end dates between two systems (check Section 3.2.4). Then we plot the Kaplan-Meier survivability curves for code smells detected in both simulation and traditional systems.

From Figure 3.7, we see that for the first 2,000 days, code smells in both simulation and traditional systems have similar probabilities of survival. However, after 2,000 days, their survivability curves start to diverge, with the probability of survival in traditional systems taking a nosedive. Although we observe a similar phenomenon in simulation systems after 3,000 days, their overall probability of survival remains higher when compared to traditional counterparts. At the end of the observation period,

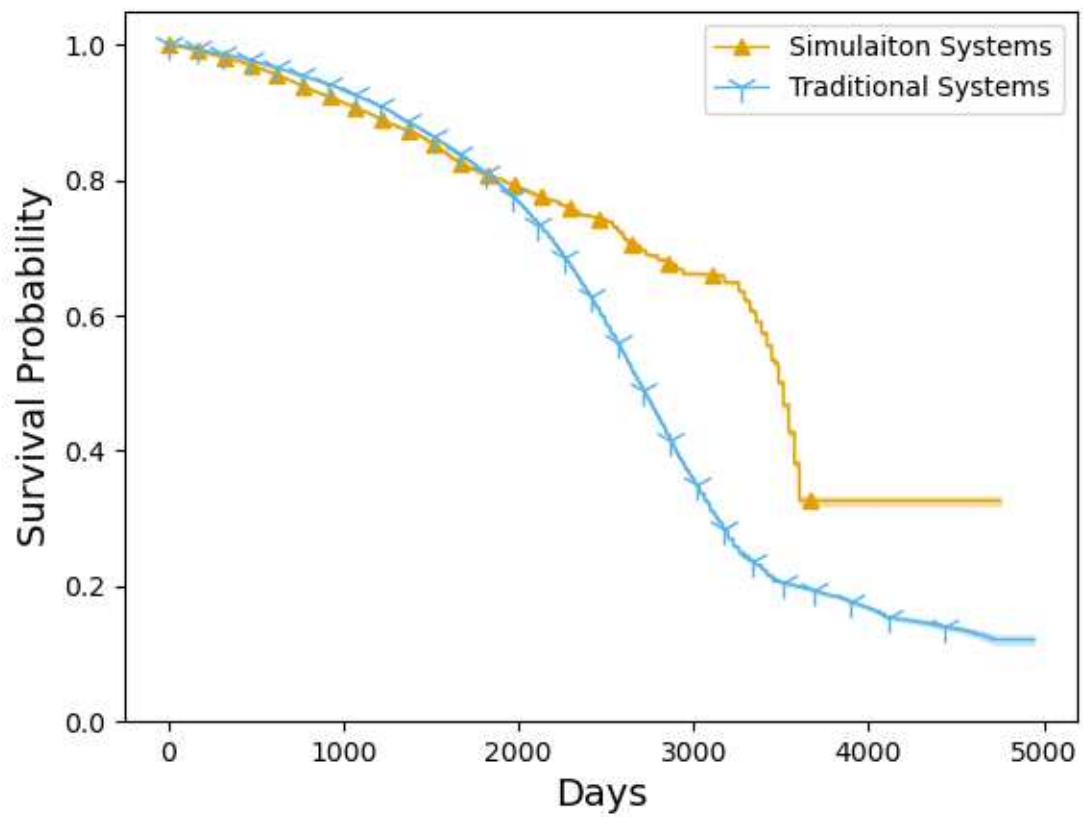


Figure 3.7: Survivability curves of code smells in simulation and traditional systems

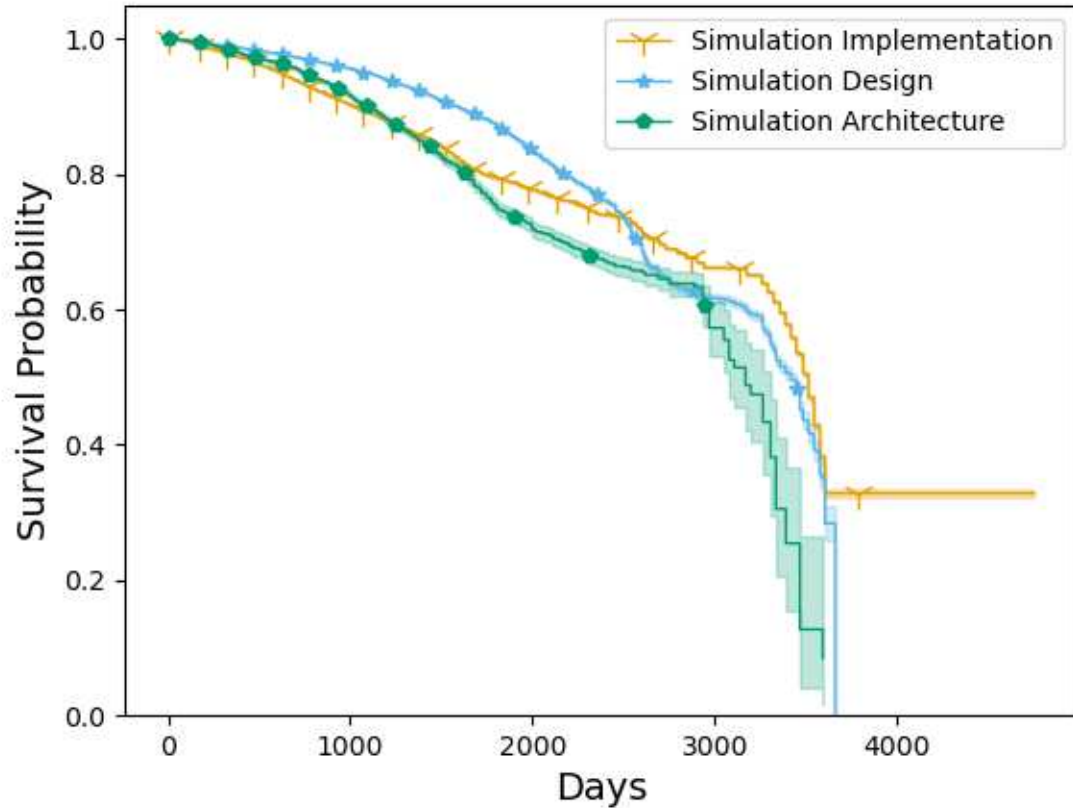


Figure 3.8: Survivability curves of code smells in simulation systems (by abstraction level)

we observed that code smells in simulation and traditional systems have a median survival time of 3,513 and 2,698 days respectively. This shows that code smells can last longer in simulation systems than their counterparts in traditional systems.

Figure 3.8 shows the survivability curves for three types of code smells –Implementation, Design, and Architecture smells – from simulation systems. We notice that all share a similar probability of survival for around the first 3,000 days. However, after this, the probabilities of survival for *Design* and *Architecture Smells* decrease drastically, implying that most *Design* and *Architecture Smells* are refactored at the end of the observation period. However, the curve for implementation smells stays the same even after 3,500 days. This indicates that most implementation smells in simulation systems might not get refactored even after a long development time.

We see a similar phenomenon in Figure 3.9, where all five types of code smells

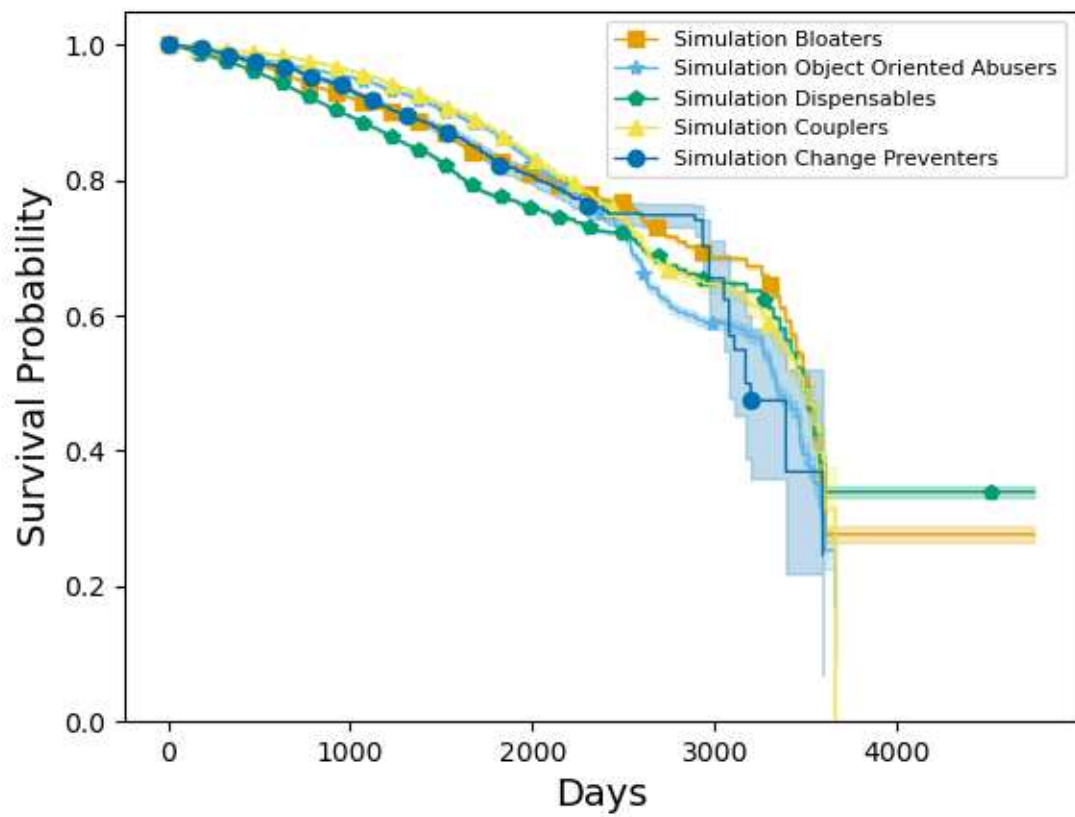


Figure 3.9: Survivability curves of code smells in simulation systems (by smell type)

Table 3.3: Median survival times (MST) of all code smells

Code Smell	MST (days)
Long Parameter List	3,513
Long Statement	3,467
Complex Method	3,513
Magic Number	3,484
Complex Conditional	3,513
Long Method	3,604
Empty catch clause	3,322
Long Identifier	3,576
Abstract Function Call From Constructor	1,733
Unutilized Abstraction	3,336
Deficient Encapsulation	3,513
Insufficient Modularization	3,604
Broken Modularization	3,107
Imperative Abstraction	3,306
<b>Broken Hierarchy</b>	<b>3,661</b>
Feature Envy	3,390
Missing Hierarchy	3,446
Unexploited Encapsulation	1,809
Feature Concentration	2,934
Unstable Dependency	3,079
Scattered Functionality	1,920

(i.e., Martin Fowler’s catalog) have similar survivability curves for the first 2,500 days. After this, The *Change Preventer* category of code smells shows a steeper survivability curve, ending with a 24.58% chance of survival at the end of the observation period. These steeper curves suggest a longer duration of survival before refactoring. Similarly, the survivability curve of *Object-Oriented Abusers* also diverges with a lower survival rate after 2,500 days, ending with a 16.85% chance of survival at the end of the observation period. On the other hand, *Bloaters*, *Dispensables* and *Couplers* share very similar survivability curves up to the first 3,500 days. After this, we observe that the survivability curves for both *Bloaters* and *Dispensables* stagnate, having around 27.65% and 33.93% chance of survival respectively. Conversely, we see the survivability curve of *Couplers* experience a sharp drop off after 3,500 days to almost 0% chance of survival at the end of the observation period.

Table 3.3 further shows the median survival times of all code smells detected in

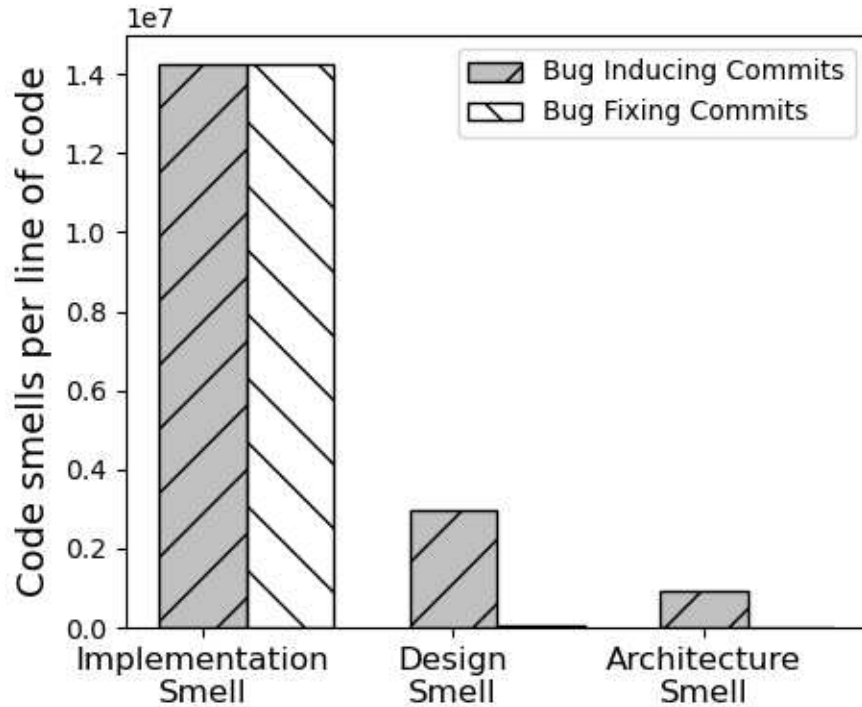


Figure 3.10: Number of code smells in bug-inducing and bug-fixing commits

simulation software systems. Here, we see that the *Broken Hierarchy* smell lasts the longest with a median survival time of 3,661 days followed by the *Long Method* smell that has a survival time of 3,604 days. We also observe that most code smells last around 3,000 days in simulation systems. However, the *Abstract Function Call From Constructor* and *Scattered Functionality* are exceptions to this trend, lasting about 1,733 and 1,920 days respectively.

**Summary of RQ2:** We observe that the code smells in simulation systems can survive longer than those in traditional systems (Figure 3.7). In particular, *Implementation*, *Bloaters* and *Dispensable* code smells exhibit higher survival rates in simulation systems (Figure 3.8, 3.9). These results suggest that developers of simulation systems should be more proactive removing code smells during development, thereby making them less persistent in the codebase.

Table 3.4: Value of Cramer’s V and Chi-square p values for each code smell

Code Smell	Cramer’s V	p-value
Imperative Abstraction	0.1963	0.0061
Multifaceted Abstraction	0.1268	0.3899
Feature Envy	0.1233	0.0818
Broken Modularization	0.1134	0.1306
Broken Hierarchy	0.1108	0.1583
Feature Concentration	0.1028	0.0587
Unstable Dependency	0.0916	0.2207
Rebellious Hierarchy	0.0861	0.4469
Scattered Functionality	0.0841	0.7886
Missing Hierarchy	0.0829	0.6217
Unutilized Abstraction	0.0758	0.2779
Deficient Encapsulation	0.0584	0.7044
Cyclic Hierarchy	0.0566	0.8843
Unexploited Encapsulation	0.0456	0.9192
Dense Structure	0.0451	0.6064
Cyclic Dependency	0.0343	0.9428
Insufficient Modularization	0.0311	0.9615
Multipath Hierarchy	0.0299	0.7646
Wide Hierarchy	0.0286	0.6285
God Component	0.0059	0.9136

### 3.3.3 RQ3: Do code smells co-occur with bugs in simulation software systems?

We investigate whether code smells co-occur with software bugs by analyzing the prevalence of code smells in bug-inducing and bug-fixing commits of simulation systems. According to Figure 3.10, there are more *Implementation Smells* than *Design* and *Architecture Smells* in both bug-inducing and bug-fixing commits. Interestingly, the implementation smells are equally present in both types of commits. This shows that either most *Implementation smells* are introduced before bugs or they are not refactored even after bugs are fixed, implying that *Implementation Smells* do not occur with bugs. This is backed up by our results from *RQ2*, which shows that Implementation smells can survive the longest in simulation systems. However, as shown in Figure 3.10, bug-inducing commits contain a much larger amount of *Design* and *Architecture* smells than bug-fix commits. This shows that most *Design* and *Architecture Smells* are introduced at the same time as bugs in simulation systems.

We also determine the association between code smells and bugs from a contingency table capturing observed and expected frequencies of each code smell (see Section 3.2.6). In particular, we perform Pearson’s Chi-Squared and Cramer’s V tests to determine the association and strength of any associations between code smells and bugs. We see that the values of Cramer’s V for bugs and all code smells remain below 0.2, which indicates that they are very weakly associated with each other. Among them, the *Imperative Abstraction* smell has the highest value of Cramer’s V (0.1963), which still suggests a weak association. Similarly, p-value is always greater than 0.05 for all code smells and bugs, which shows that this weak association is not statistically significant. Thus, our results suggest that the presence code smells does not always indicate the occurrences of bugs in simulation systems.

**Summary of RQ3:** We observe that bug-inducing commits contain more *Design* and *Architecture* smells, while the number of *Implementation Smells* remains static (Figure 3.10). However, results of the Chi-Square and Cramer’s V test show no significant association between code smells and bugs in simulation systems. This suggests that developers should not use code smell as indicators of bugs (Figure 3.4). Rather, their primary purpose should serve to improve code quality.

### 3.4 Implication of Findings

From *RQ1*, we see that the prevalence of code smells in simulation systems is significantly different than in traditional systems. Two code smells – *Magic Number* and *Long Statement* – are 62.77% and 26.72% more frequent respectively in simulation systems. To avoid technical debt in simulation systems, developers should focus on adopting symbolic constants rather than *Magic Numbers* in their code. Similarly, the Long Statements in code should be broken down into multiple manageable, smaller statements.

Our findings from *RQ2* show that code smells in simulation software systems survive longer than those in traditional software systems. Smells such as *Long Method*, and *Broken Hierarchy* can survive 3,604 and 3,661 days respectively, on average, in a simulation system. This implies that these smells are possibly ignored during the development of simulation systems and have low priority during refactoring. From a

subsequent analysis, we observe that *Bloaters* and *Dispensable* code smells are often ignored during development, as evidenced by their relatively high chances of survival - 33.93% and 27.65%. Thus, according to our findings, simulation systems could contain large complex code structures that are not properly maintained, leading to a significant amount of redundancy. To avoid accumulating technical debt, developers should prioritize refactoring these code smells in the simulation software systems.

Our *RQ3* investigates the possible co-occurrences of bugs and code smells in simulation systems. Here, we found that *Design* and *Architecture Smells* occur more frequently in the bug-inducing commits of simulation software systems. However, there is no significant association between code smells and bugs, as evidenced by our Pearson’s Chi-Squared and Cramer’s V tests. Thus, future investigations could focus on the impact of code smells in simulation systems and their refactoring strategies.

### 3.5 Threats to Validity

**Threats to internal validity.** Threats to *internal validity* relate to any experimental errors or biases [99]. Since we select repositories from different domains, the quality of their code bases might not be comparable. To mitigate this threat, we used several metrics such as star count, number of contributors, and number of issues and attempted to select the repositories that are of comparable quality.

We also use the SZZ algorithm to capture bug-inducing commits against their bug-fixing commits, where the algorithm has its limitations. Since we used a set of keywords to find bug-fixing commits, it may introduce false positives [100]. To remedy this, we manually checked 25 randomly sampled commits and found only 2 (8%) false positive commits. Thus, any problems related to the SZZ algorithm might not significantly affect our overall findings. Besides, we have only considered differences, associations, or co-occurrences between two variables. As we do not claim any causation relationship between the two variables, the relevant threats might be minimal and might not affect our overall findings.

**Threats to conclusion validity.** Threats to *conclusion validity* relate to the accuracy of conclusions [101]. During repository selection, we end up with 155 simulation systems and 327 traditional systems. Due to the differences in the sample sizes, the statistical results might be hard to infer. To mitigate this threat, we take a

random subsample of equal size to make the comparison fair and unbiased. We also use non-parametric statistical tests to avoid any assumptions behind the underlying distribution of the samples.

**Threats to External validity.** Threats to *external validity* relate to the generalizability of any findings. To mitigate these threats and achieve diversity in our dataset, we select our traditional systems from a variety of domains, including web development, mobile development, and desktop applications. Moreover, we collect simulation systems based on both topic-based and keyword-based searches. Thus, our selected systems might represent the general population of simulation projects.

## 3.6 Related Work

### 3.6.1 Prevalence of code smells

Many existing works study the prevalence of code smells in different software systems. Sabóia et al. [38] analyzed 25 C# systems and suggest that *Implementation smells* are the most prevalent in C# systems. In particular, they found the *Magic Number* and *Long Statement* smells to be the most common in C# systems. Similarly, Cardozo et al [37] analyzed 24 Reinforcement Learning systems and found *Long Method* and *Long Method Chain* as the most common code smells. On the other hand, Jebnoun et al [8] found no statistically significant difference in the prevalence of code smells between 59 deep learning and 59 traditional systems. Mannan et al [48] analyzed 500 Android and 750 Desktop applications and showed similar variety and densities of code smells for both systems. However, they observed that Desktop systems are dominated by *External Duplication* and *Internal Duplication* code smells, while Android systems contain an equal distribution of both smell types. In our study, we analyze the prevalence and distribution of code smells in 155 simulation software systems. Furthermore, we also measure the probabilities of survival for significantly different code smells from *RQ1* for both simulation and traditional systems (see Section 3.3.1).

### 3.6.2 Evolution of code smells

Tufano et al [40] found that very few code smells are introduced during software evolution. They also found 400 cases where refactoring operations introduced code smells in the system. Chatzigeorgiou et al [42] found that very few smells are removed from a system after their introduction. They observed that most code smells are removed through adaptive maintenance rather than active refactoring efforts. Muse et al [50] also analyzed 150 open-source Java projects and suggested that most SQL code smells can persist through multiple project versions without getting refactored. Unlike the above studies, we investigate the differences in the evolution of code smells from simulation and traditional systems. We also find the survivability of each code smell in simulation systems across our observation period of 4,949 days (see Section 3.3.2).

### 3.6.3 Impact of code smells

A literature survey of eighteen studies by Cairo et al [45] found sixteen studies suggesting an association between bugs and code smells. The remaining two studies found no associations between bugs and code smells. According to Jaafar et al [5], Object-Oriented classes with anti-patterns and code clone smells are three times more likely to contain faults than non-smelly classes. In particular, up to 64% classes contain co-occurrences of *anti patterns* and *Code Clone* smells, resulting in a higher fault proneness ratio. Hecht et al [84] found that refactoring of code smells can lead to memory and UI performance improvements in Android Systems. In particular, refactoring the *Member Ignoring Method* smell leads to a 12.4% improvement in UI performance. On the other hand, refactoring *Garbage Collection* smells shows a 3.6% improvement in memory performance. Our study attempts to detect any associations between bugs and code smells in simulation systems. We also measure the strength of these associations to analyze the impact of code smells on bugs in simulation software.

## 3.7 Summary

The presence of code smells in a software system indicates its deeper code quality issues. Thus, many studies focus on the prevalence and effects of code smells in various types of software systems. However, despite their enormous importance, there has not

been any work on the code smells of simulation systems. In this study, we aim to fill this gap by analyzing 155 simulation and 327 traditional systems and investigating the prevalence, evolution, and impact of code smells in simulation modelling software. First, our analysis of simulation and traditional systems code smells shows that code smells are more prevalent in simulation systems compared to traditional systems. In particular, the *Magic Number* and *Long Statement* smells occur more frequently in simulation systems. We also draw survivability curves to observe that code smells in simulation systems last longer than in traditional systems. Moreover, we find that *Implementation Smells*, *Bloaters*, and *Dispensable* code smells have the highest survival rates in simulation systems. Lastly, after performing both Pearson's Chi-Square and Cramer's V test, we find no significant association between code smells and bugs. Overall, our study is one of the first to extensively investigate the nature of code smells in simulation software systems. By analyzing the prevalence, evolution, and impact of code smells in simulation modelling software, we shed light on the code quality of simulation systems. The results of our study can inform both users and developers of simulation software about specific threats to code quality and thus could impact their development practices.

Refactoring is a natural choice to improve smelly code. Being intrigued by our findings on code smells, we present our second study in the next chapter investigating the refactoring practices in simulation software systems.

## Chapter 4

### On the Effectiveness, Risks, and Impact of Refactoring Practices in Simulation Modelling Software

Our first study in Chapter 3 investigates the code smells in simulation modelling systems. Refactoring is a natural choice for addressing code smells. However, the refactoring practices in simulation systems are not well understood to date, indicating a significant knowledge gap. In this chapter, we conduct an empirical study to investigate the effectiveness, risks, and impact of refactoring practices in simulation systems, aiming to bridge the existing knowledge gap.

The rest of this chapter is organized as follows. Section 4.1 introduces our study and discusses the novelty of our work. Section 4.2 discusses our methodology employed in our empirical study. Section 4.3 analyzes the results of refactoring simulation systems in detail. Section 4.4 presents the implications of our results and how they might impact the development of simulation systems. Section 4.6 discusses related works, targeting the effectiveness, risks and impacts of refactoring. Finally, Section 4.7 presents a summary of our study.

#### 4.1 Introduction

Code smells do not directly impair software functionality, but they serve as indicators of underlying software quality concerns [102]. Existing research has demonstrated that code smells correlate with various software issues including degraded performance [52], reduced maintainability [103], [104], and impaired comprehension [105]. Furthermore, Jaafar et al. [106] found that Object-Oriented classes containing anti-patterns and code clones have tripled the likelihood of defects compared to smell-free classes. Given these implications, code smell remains an active topic of research in software engineering.

Given the negative effects of code smells, refactoring them is a natural choice for software developers. Refactoring is a small change to the internal structure of the source code that does not affect its external behaviour. For example, if a method is

refactored, its implementation might change, but the signature, return type or formal parameters should not change. Refactoring has the potential to improve the non-functional properties of the code (e.g., maintainability, portability). Several studies report the benefits of refactoring code smells. For example, Hecht et al.[52] suggest that refactoring of code smells can lead to 3.6% memory and 12.4% UI performance improvements in Android applications. Similarly, Kaur et al [10] show that refactoring code smells can lead to lower Cyclomatic Complexity and improved cohesion within methods and classes, making the software easier to maintain. Moreover, Chug et al. [53] demonstrate that refactoring code smells, such as Spaghetti Code and Functional Decomposition, can lead to a 5-20% reduction in defects. Similarly, Mumtaz et al. [107] suggest that refactoring Feature Envy code smells could lead to reduced security defects. While these studies demonstrate the benefits of refactoring, some studies also present a more nuanced view. For instance, Alshayeb et al [108] found that refactoring may not always lead to significant improvements in adaptability, maintainability, understandability, reusability, and testability metrics. Similarly, Yamashita et al. [2] observed that certain types of refactoring, such as clone refactoring, could potentially decrease overall code quality in some cases. This mix of results have inspired numerous studies investigating refactoring practices across different domains, including general purpose software systems [11], [109], database systems [13], [14], [15], android systems [12], [16], [17], and even deep-learning systems [18], [19]. However, there is a notable lack of research focusing on refactoring practices in simulation software systems.

Simulation models are an imitative representation of real world systems in a controlled, virtual environment. They play a crucial role in numerous fields including military [20], scientific research [110], [111], transportation [21], [22], and medicine [25], [26]. Given the importance of simulation software systems, understanding their refactoring practices tackling any code quality issues is essential.

In this chapter, we conduct an empirical study to investigate the effectiveness, risks, and impacts of refactoring practices in simulation software systems. To this end, we analyze 104 simulation and 272 traditional software repositories from GitHub, and employ three widely used tools – Designite [54], RefactoringMiner [65] and JaSoMe [59] for our analysis. Through our experiments, we thus answer three important research questions as follows:

- (a) **RQ1: Do refactoring practices effectively remove code smells from simulation software systems?** We employ multiple static analysis tools (e.g., RefactoringMiner [65], Designite[54]), establish connections between code smells and their refactored code, and determine the effectiveness of refactoring activities. We find that refactoring is more effective (i.e., removes more code smells) in simulation systems than in traditional systems. Refactoring activities reduce the survival rate of code smells from 95% to 65% in simulation systems. We also found that, Package level refactoring are effective for addressing Implementation and Design smells, whereas, Method level refactoring activities are effective against Architecture smells.
- (b) **RQ2: Do refactoring practices risk developing new code smells in simulation software systems?** We incorporate newly introduced and unchanged code smells into our dataset and analyze the risks of refactoring activities in simulation systems. We find that refactoring introduces fewer code smells in simulation systems compared to traditional systems. Refactoring activities can increase the risk of introducing new code smells from 15% to 18% in simulation systems. We also found that, Method level refactoring techniques have overall weak statistical association according to Cramer’s V test. However, Split Conditional, Move Code and Assert Throws refactoring techniques introduce 2.5 times more code smells than they remove, making them significant outliers.
- (c) **RQ3: Do refactoring practices improve the maintainability of simulation software systems?** We also calculate 23 software maintainability metrics for the refactored code by employing JaSoMe, a static analysis tool, for all three levels of code abstraction (Method, Class and Package). We find that refactoring activities result in reduced Cyclomatic and Structural complexity of code. They also reduce the cumulative complexity in both Class and Package level metrics. Finally, although refactoring increases the number of abstract classes and interfaces, it does not significantly change the abstractness of the code as the total number of classes remain largely stable throughout the development period.

## 4.2 Methodology

Figure 4.1 shows the schematic diagram of the conducted study. We filter and preprocess 104 and 272 simulation and traditional software repositories from GitHub. First, we rank each refactoring technique based on their effectiveness at removing code smells. Next, we capture and analyze newly introduced code smells after refactoring to assess the risk of each refactoring technique. Finally, we calculate 23 maintainability metrics to check the maintainability of simulation software systems. In the following sections, we discuss the major steps of our methodology as follows.

### 4.2.1 Project Selection Criteria

We choose GitHub as the source of our simulation and traditional software repositories (Step 1, Fig. 4.1). We use the GitHub search API [89] to collect repositories based on GitHub Topics and keywords. First, we search using the following topics – *simulation*, *simulator* and *simulation modeling*– and capture a collection of 1,214 of simulation systems. Similarly, we use several traditional topics – *web*, *framework*, *android*, *HTTP*, and *desktop* – to collect a diverse set of 2,722 traditional software systems. We then select the repositories from this list that meet the following criteria:

1. Only the repositories with *open-source licenses* (MIT, GPL, and Apache licenses) are selected as they do not require explicit permissions for any third-party re-use.
2. We do not consider any repositories *disabled* by their owners since we do not have permission to access them.
3. We also do not select *archived* repositories, as they have ceased all active development.
4. Forks of original repositories are also removed to avoid code duplication in multiple repositories.

After applying the above filtration criteria (Step 2, Fig. 4.1), we also manually select repositories with a large history of refactoring efforts. This allows us to avoid false positives from the automatic filtering process and leaves us with 104 simulation and 272 traditional software systems. Since, popular languages such as C++ and Python

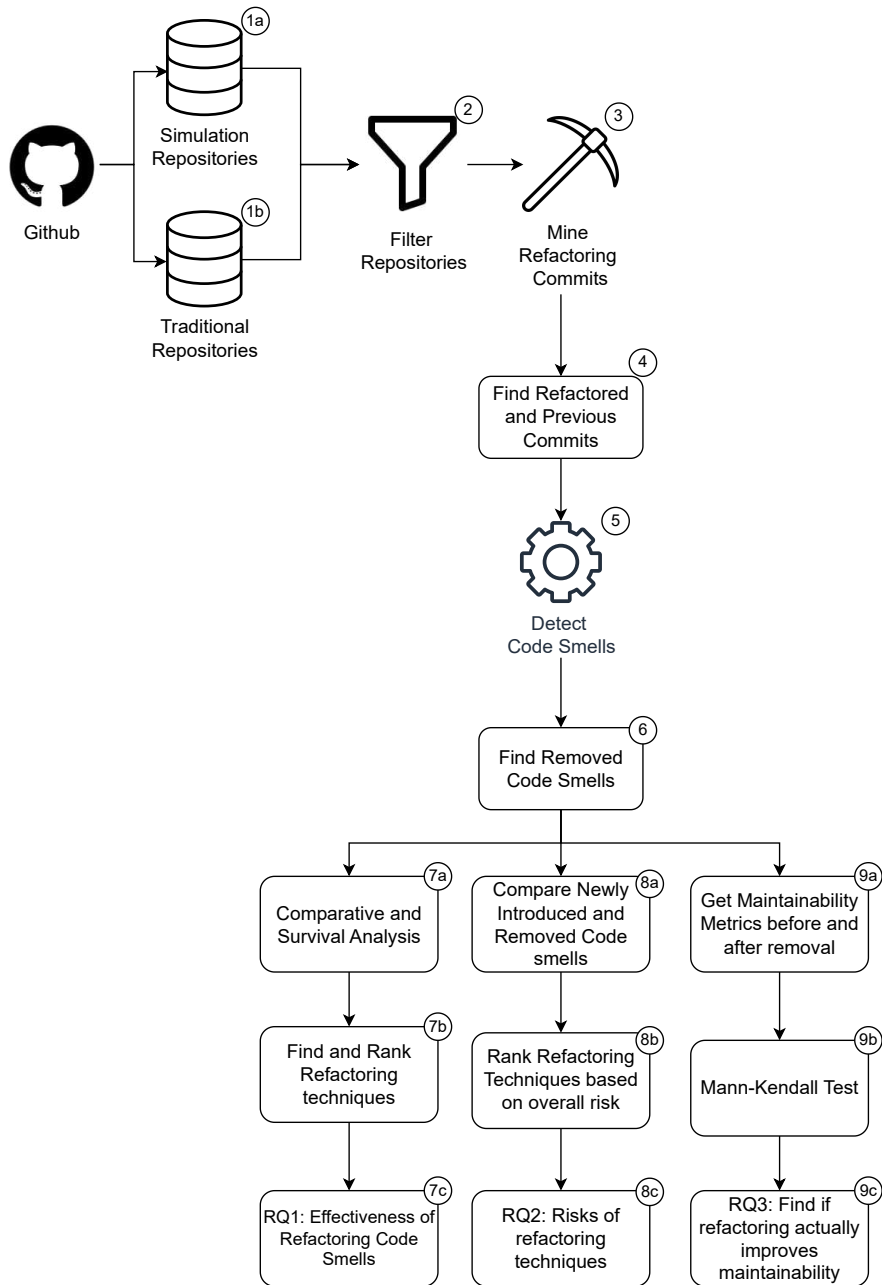


Figure 4.1: Schematic diagram of our study

have a large number of false positives for simulation systems, it makes comparing them with traditional systems difficult. Thus, we only select Java repositories for our study.

### 4.2.2 Mine Refactoring Commits

We mine our selected repositories for specific refactoring commits (Step 3, Fig. 4.1). Our goal is to analyze the refactoring practices of simulation and traditional systems. To this end, we apply RefactoringMiner [65] to each repository, which yields a list of refactored commits along with their corresponding refactoring patterns. We also employ PyDriller [57] to find the previous commits of these refactoring commits (Step 4, Fig. 4.1) and gather further insights on the refactoring practices in both simulation and traditional systems.

### 4.2.3 Code Smell Detection

After capturing a list of refactoring commits and their parent commits from each repository, we scan them for code smells (Step 5). To get the source code, we reset the repository to the refactoring commit and its parent commit. We then employ Designite [54] to detect code smells from both the refactoring commit and the parent commit, which reports code smells in each project across three levels of abstraction (Implementation, Design and Architecture Smells). As the size of simulation and traditional systems could differ, we normalize the number of code smells in each repository using their total lines of code. This gives us an average number of smells per line for each repository, which accounts for repository size.

After collecting the code smell reports for both sets of commits, we attempt to detect the successful refactoring commits (Step 6, Fig. 4.1). That is, we consider a refactoring commit successful if the code smells only exist in its previous commit. We look at the method, class and filenames of detected smells in the previous commit to determine successful cases. If we find the same code smell exists in the same method, class and file in the refactored commit, we consider it as an unchanged code smell. However, since method, class and file names can be changed between commits, we rely on the names in the refactored commits for matching. This step informs us of the removed and unchanged code smells, across different levels of abstractions, by the

Table 4.1: Example of contingency table for removed *Implementation* smells and Method-Level refactoring techniques

<b>Removed Implementation Smell</b>	<b>Split Parameter</b>	<b>Merge Conditional</b>	<b>Change Return Type</b>
Magic Number	16	21	57
Complex Conditional	12	34	9
Long Parameter List	24	13	17

refactoring activities.

#### 4.2.4 Effectiveness of Refactoring

After gathering our data, we perform comparative analysis to show the differences among the removals of Implementation, Design, and Architecture Smells per refactoring commit (Step 7a, Fig. 4.1). This is done by comparing the code smell frequencies in each commit across three levels of abstraction (Implementation, Design, Architecture) for simulation and traditional systems. We also break down the removed and remaining code smells in each abstraction level into their respective categories to find which code smells are better removed by refactoring activities. These analyses help us understand the effectiveness of refactoring practices for each commit in simulation systems. We also find the long-term effectiveness of refactoring practices across multiple commits using Kaplan-Meier survival curves (see Section 2.7) for a period of 5 years.

To find which refactoring technique is the most effective at removing code smells, we rank them based on their effectiveness at removing each type code smell (Step 7b, Fig. 4.1). We divide the refactoring techniques into three levels of abstraction (Method, Class and Package). To validate our rankings, we create a contingency table (See section 2.4) by placing refactoring techniques into each row and the number of removed smells in each column. This results in a table where each cell contains the number of removed code smells for each refactoring technique.

Table 4.1 shows an example of contingency table, with the rows containing the number of *Implementation* smells and the columns containing the number of *Implementation* smells being removed for each Method-Level refactoring technique. We create similar contingency tables for code smells and refactoring techniques for all layers of abstraction (Implementation, Design and Architecture) and refactoring scopes

Table 4.2: Example of contingency table for new *Implementation* smells and Method-Level refactoring techniques

<b>New Implementation Smell</b>	<b>Split Parameter</b>	<b>Merge Conditional</b>	<b>Change Return Type</b>
Magic Number	11	12	19
Complex Conditional	6	11	15
Long Parameter List	14	3	16

(Method, Class and Package) respectively. We then calculate the results of the Chi-Square test (See section 2.6.1) by using the number of removed smells in the table to find any significant associations between applied refactoring techniques and removed code smells. We also expand these results to find the strength of any significant association via the Cramer’s V test (See section 2.6.2), which is calculated from the Chi-Square statistic and number of removed smells from the table.

#### 4.2.5 Risks of Refactoring

To determine the risks of refactoring activities in simulation systems, we compare the frequencies of newly introduced code smells with the removed code smells in each refactoring commit (Step 8a, Fig. 4.1). We compare the distribution of each type of code smell before and after their refactoring to see how refactoring can affect their maintainability. We also calculate the long-term risk of introducing new code smells across multiple refactoring commits using Nelson-Aalen estimators (see Section 2.8). This helps us determine the probability of contracting new code smells by refactoring activities over an observation period of 5 years.

To determine which refactoring techniques are more likely to cause new code smells than removal, we rank them based on their calculated risk above (Step 8b, Fig. 4.1). We then support each ranking by putting the refactoring techniques in the row and the number of newly introduced code smells in the columns of a contingency table. This results in each cell of the table containing the number of new code smells introduced by each refactoring technique. We then perform Chi Squared and Cramer’s V tests (see Section 2.6.1 and 2.6.2) to determine associations between refactoring techniques and newly introduced code smells.

Table 4.1 shows an example of the described contingency table, with the rows containing the number of *Implementation* smells and the columns containing the number of new *Implementation* smells introduced by each Method-Level refactoring technique. Similar to the effectiveness of refactoring techniques, we create tables for new code smells and refactoring techniques for all layers of abstraction (Implementation, Design and Architecture) and refactoring scopes (Method, Class and Package) respectively. Now, we calculate the results of the Chi-Square and the Cramer’s V test (See section 2.6.1 and 2.6.2) similarly to our previous study on the effectiveness of refactoring techniques. This shows us any existing associations between different refactoring techniques and the new code smells introduced by them along with their strengths.

#### 4.2.6 Impact of refactoring

To investigate whether refactoring in simulation systems improves their maintainability, we conducted a multi-step empirical analysis. First, we collected maintainability metrics for original and refactored code using static analysis tools such as JaSoMe. We use metrics such as Cyclomatic Complexity, Structural Complexity, Data Complexity, Fan-in, Fan-out and Nested Block Depth to get a rounded view of the control flow, code and dependency organization of different methods. Similarly, we employ metrics by Chidamber and Kemerer [66] and Robert C. Martin [55] to assess the class and architectural design of entire systems. These metrics capture the issues in structure and modularity of both small functions and system-wide packages and thus can serve as a suitable proxy for measuring overall software maintainability.

To ensure consistency in our analysis, we select the same starting and ending date for all repositories. As different repositories can have different number of commits in the same timeframe, we represent each commit as a percentage of overall progress for each repository. For example, if repository A has 10 commits and repository B has 20 commits, then each single commit in repository A represents 10% of its overall progress, while each commit in repository B represents 5%. This means we account for the varying commit histories when analyzing maintainability changes. We select 23 maintainability metrics, including Cyclomatic Complexity, Structural Complexity, Abstractness, and Stability, to assess the impact of refactoring on simulation systems. We then calculate the average values of these metrics for two code versions – before

and after refactoring –of each repository (Step 9a, Fig. 4.1).

We then analyze trends in maintainability metrics over time using the Mann-Kendall test (Step 9b, Fig. 4.1), applied to each repository individually. As this results in multiple p-values for each repository, we use Stouffer’s method (see Section 2.9.1) to aggregate the results. This method gives an overall p and z value , indicating the significance of the impact of refactoring on multiple simulation systems(Step 9c, Fig. 4.1).

#### 4.2.7 Replication Package

We made our replication package [112] publicly available for any third-party reuse or replication.

### 4.3 Results and Analysis

#### 4.3.1 RQ1: Do refactoring practices effectively remove code smells from simulation software systems?

We gauge the effectiveness of refactoring practices in simulation systems by comparing them with traditional systems. In Figure 4.2, we see that code smells are removed similarly in both simulation and traditional software systems, with most removals happening at the Implementation level. Architectural smells has the least removals in both simulation and traditional systems. However, looking at the smells removed per line, we see that code smells are far more effectively removed in simulation systems than in traditional systems. In particular, we see that simulation systems remove 0.007 Implementation, 0.005 Design and 0.0006 Architecture smells per line of code, whereas traditional systems only remove  $8.56 \times 10^{-7}$  Implementation,  $3.52 \times 10^{-7}$  Design and  $5.4 \times 10^{-8}$  Architecture smells per line of code. This stark differences of smell removals per line highlight that refactoring efforts in simulation systems are more focused on removing code smells than traditional systems.

Figure 4.3 shows a breakdown of all removed code smells in simulation systems in the form of a Sankey diagram. We see that Implementation Smells are the most frequently removed code smells, followed by Design and Architecture Smells. Among them, the Magic Number and the Long Statement smells are removed the most frequently.

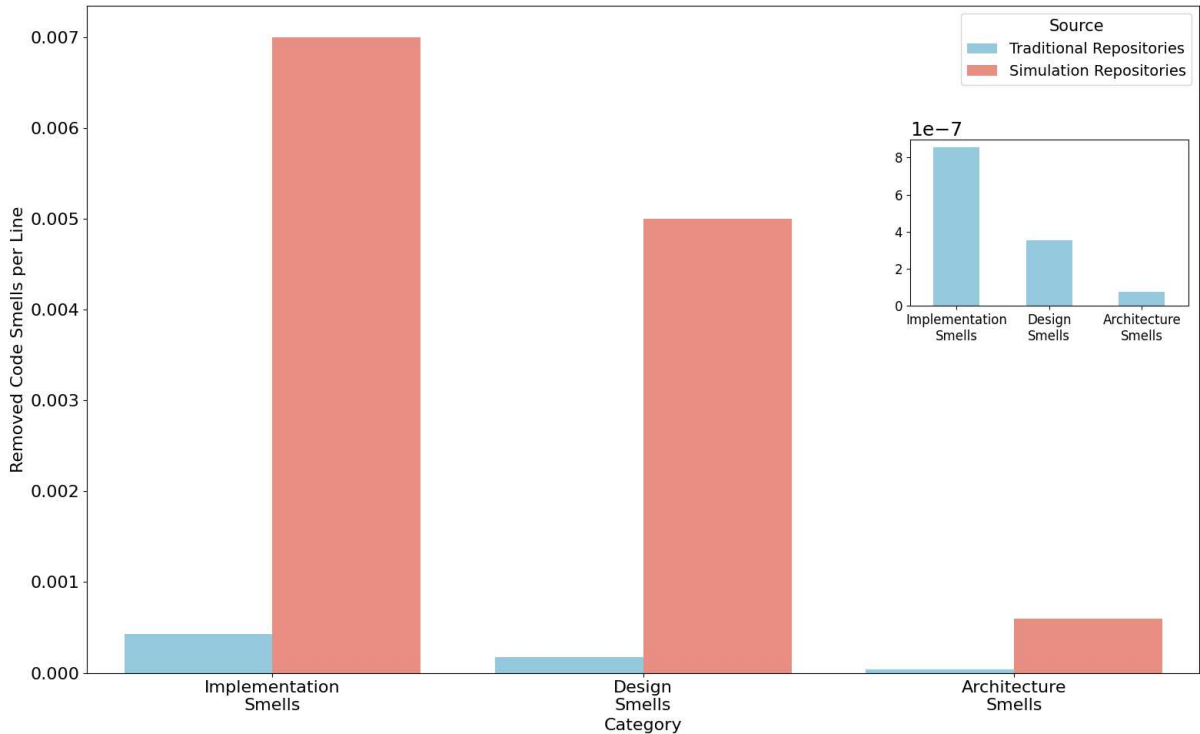


Figure 4.2: Comparison between removed simulation and traditional smells

Similarly, removals of Design Smells are dominated by the Unutilized Abstraction and Broken Hierarchy smells. However, we see that different types Architectural smells are removed in similar proportions. This shows us that refactoring practices in simulation systems are skewed towards removing specific Implementation and Design smells but no single Architectural smell dominates the refactoring efforts. We also found that unlike Implementation and Design smells which are removed deliberately, Architectural smells are often removed incidentally as part of larger refactoring efforts, leading to a more even distribution.

We also analyze the long-term effectiveness of refactoring activities in simulation systems by employing the Kaplan-Meier Survival Curves analysis for an observation period of five years. From Fig 4.4 we see that Implementation smells initially showed the highest survival rate at 95%, while Architecture smells demonstrated the lowest initial survival rate of 84%. Over the 5-year period, Architecture smells proved to be the least resilient, experiencing multiple sharp declines and finally dropping to the lowest survival rate of 64%. In contrast, Implementation and Design smells

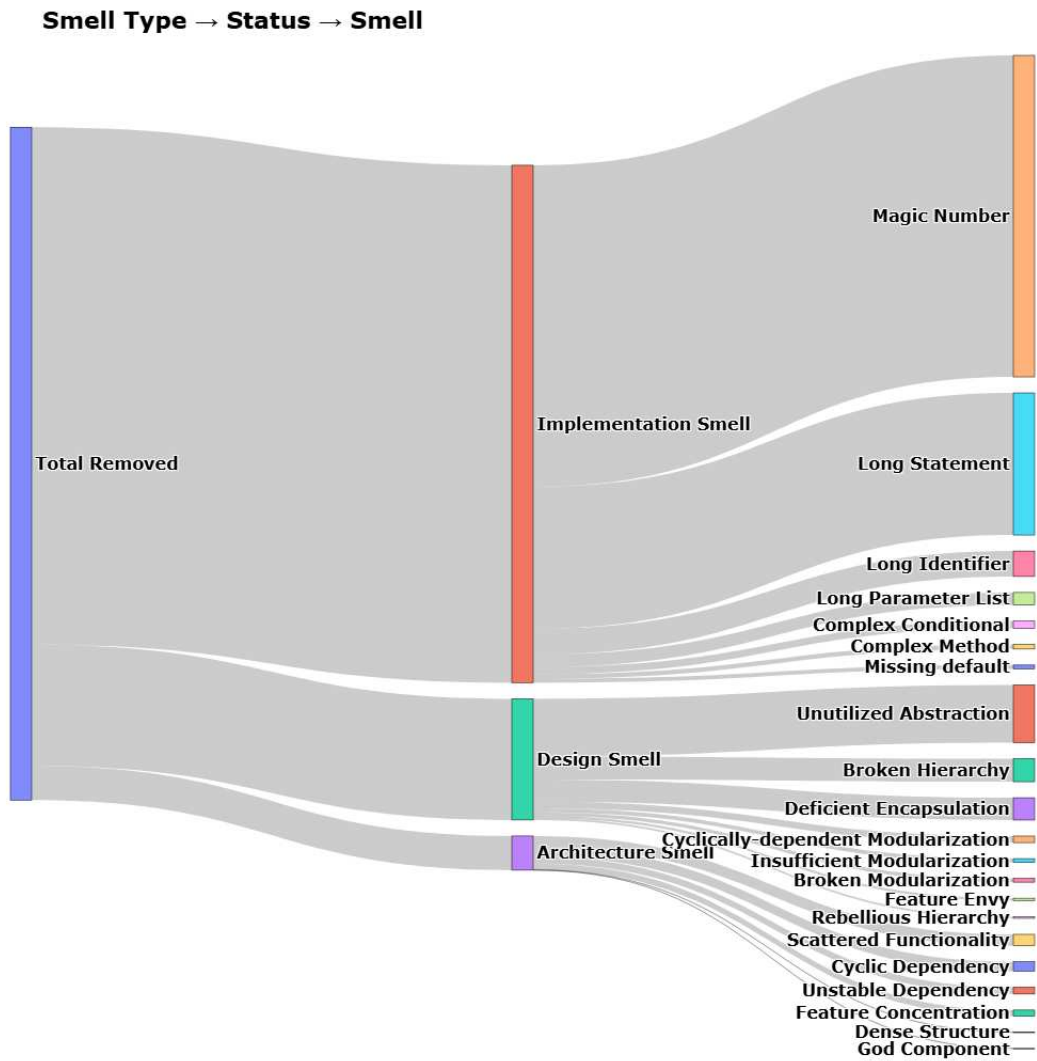


Figure 4.3: Breakdown of removed code smells in simulation systems

Table 4.3: Association between refactoring technique and removed smells

Removed Code Smell	Scope of Refactoring	p < 0.05	Cramer's V
<b>Implementation Smell</b>	Method Level	True	0.1256
	Class Level	True	0.1241
	Package Level	True	0.2460
<b>Design Smell</b>	Method Level	True	0.1735
	Class Level	True	0.1951
	Package Level	True	0.3380
<b>Architecture Smell</b>	Method Level	True	0.3581
	Class Level	True	0.2642
	Package Level	True	0.2678

maintained relatively higher survival rates through most of the period, though both eventually declined to around 66% by the end. The most dramatic decline was seen in Implementation smells, which fell from an initial 95% to 66% survival rate - a 29% drop that represents the largest overall decrease among all smell types. Notably, the survival rates of all three smell types converge around 1750 days, which is close to the end of the observation period (1825 days). This convergence suggests that, regardless of the initial differences in survival rates among smell types, long-term refactoring efforts tend to have a similar overall impact on all types of code smells, with approximately 65% of smells surviving after 5 years of development. This is in stark contrast to our previous findings in Figure 3.7, which shows that most code smells are not refactored during the course of development. This suggests that while dedicated refactoring efforts are more successful at removing code smells in simulation systems, their applications might be more sparse. We also note that developers often neglect refactoring during other developmental tasks such as feature additions and bug-fixes, resulting in more code smells surviving these tasks. Furthermore, the sudden drop in code smells at the end of development might be a result of dedicated refactoring efforts after a large contingent of code smells accumulate in the system.

We also conduct association tests to determine if refactoring methods are associated with the removal of code smells or not. Table 4.3 shows the results of our Cramer's V tests. We see that all refactoring techniques are significantly associated with the removal of code smells from different levels of abstraction (Implementation, Design and Architecture). For Implementation smells, Package level refactoring shows the strongest association with a Cramer's V of 0.2460, while Method and Class level

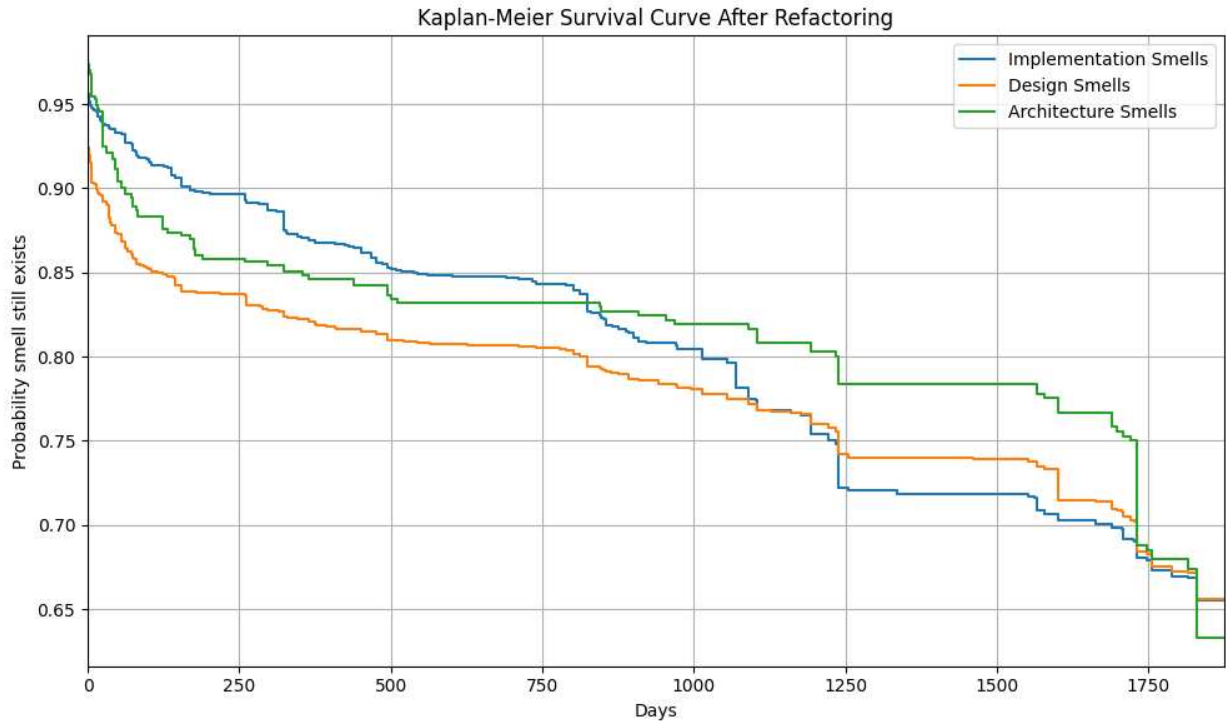
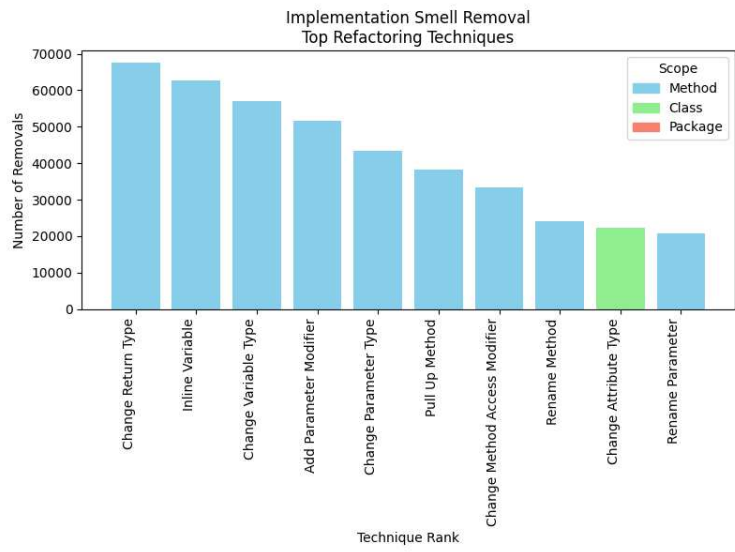


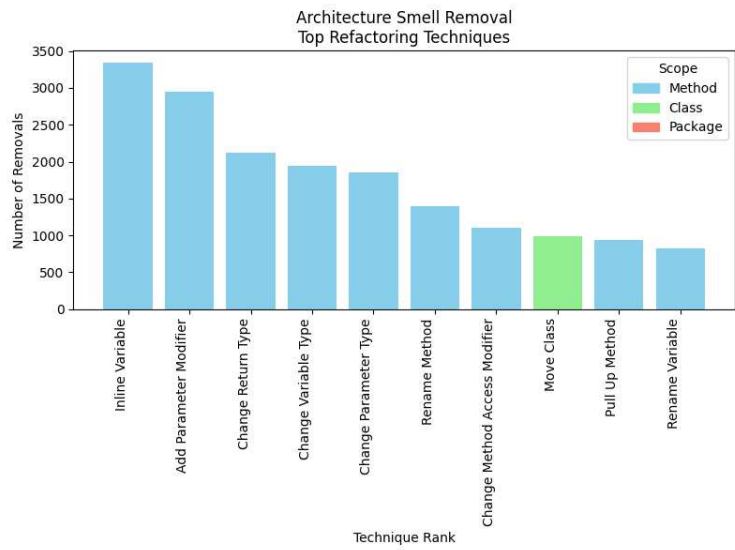
Figure 4.4: Kaplan-Meier survival curves for simulation systems

refactoring show much weaker associations of 0.1256 and 0.1241, respectively. Similarly for Design smells, Package level refactoring demonstrates the strongest association with a Cramer’s V of 0.3380, followed by Class level (0.1951) and Method level (0.1735) refactoring. However, for Architecture smells this pattern reverses; Method level refactoring shows the strongest association with a Cramer’s V of 0.3581, while Class and Package level refactoring show moderate associations of 0.2642 and 0.2678 respectively. This suggests that while all refactoring techniques can help remove code smells, their effectiveness varies considerably by abstraction level. Package level refactoring is most effective for Implementation and Design smells, while Method level refactoring works best for Architecture smells. The associations range from very weak (0.1241) to moderately strong (0.3581), indicating significant variation in effectiveness across different combinations of refactoring scope and smell type.

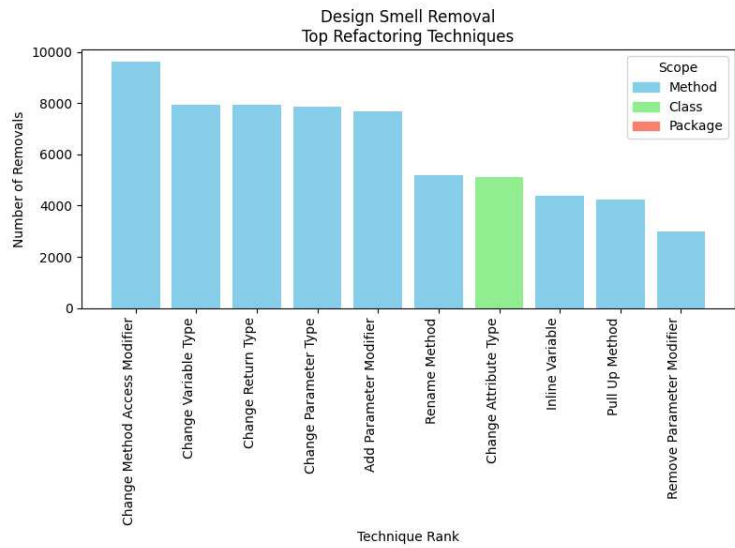
Looking at the effectiveness of top 10 refactoring techniques in Fig. 4.5, we can see that most refactoring techniques responsible for removing code smells directly work in the method level of a codebase. Only the Change Attribute Type and Move Class refactoring techniques directly work on the class level. This is in direct contrast



(a) Top refactoring techniques for removing implementation smell



(b) Top refactoring techniques for removing architecture smell



(c) Top refactoring techniques for removing design smell

Figure 4.5: Top refactoring techniques for all code smells

to Table 4.3 which shows that package level refactoring techniques have the highest degree of association with removed Implementation and Design Smells. On the other hand, removal of Architectural smells have the highest degree of association with Class level refactoring techniques. This suggests that while Method-level refactoring techniques are the most common, they tend to make localized improvements rather than addressing systemic issues. In contrast, while Package and Class level refactoring techniques are used less frequently, they have a broader impact in removing code smells properly by addressing higher order architectural concerns. This indicates that the frequency of use does not necessarily correlate with effectiveness at removing code smells.

**Summary of RQ1** Refactoring removes more code smells in simulation systems than in traditional systems, with only 65% of them surviving after 5 years (Figure 4.2, 4.4). Although Method level refactoring techniques are popular, Package and Class level refactoring techniques are more effective at removing code smells (Figure 4.5, Table 4.3). These findings suggest that developers should apply more Class and Package-Level refactoring techniques to maximize code smell removal.

#### 4.3.2 RQ2: Do refactoring practices risk developing new code smells in simulation software systems?

To analyze the risk of refactoring activities in simulation software systems, we compare the removed code smells with newly introduced code smells. Figure 4.6 shows the number of removed and newly introduced code smells per line after refactoring in simulation systems. We see that refactoring removes more code smells than it introduces at three different abstraction levels. In particular, refactoring affects Implementation smells most, with 0.007 smells being removed and 0.002 smells being introduced per line of code, indicating a relatively low risk. Design Smells show a similar pattern, with 0.005 smells being removed for 0.001 smells being introduced per line of code. Architecture smells demonstrate the smallest impact, with 0.0006 smells being removed for 0.00025 smells being introduced per line of code, but notably have the highest risk ratio of 0.42. This suggests that architectural changes are more likely to introduce new problems relative to the number of smells they remove. This

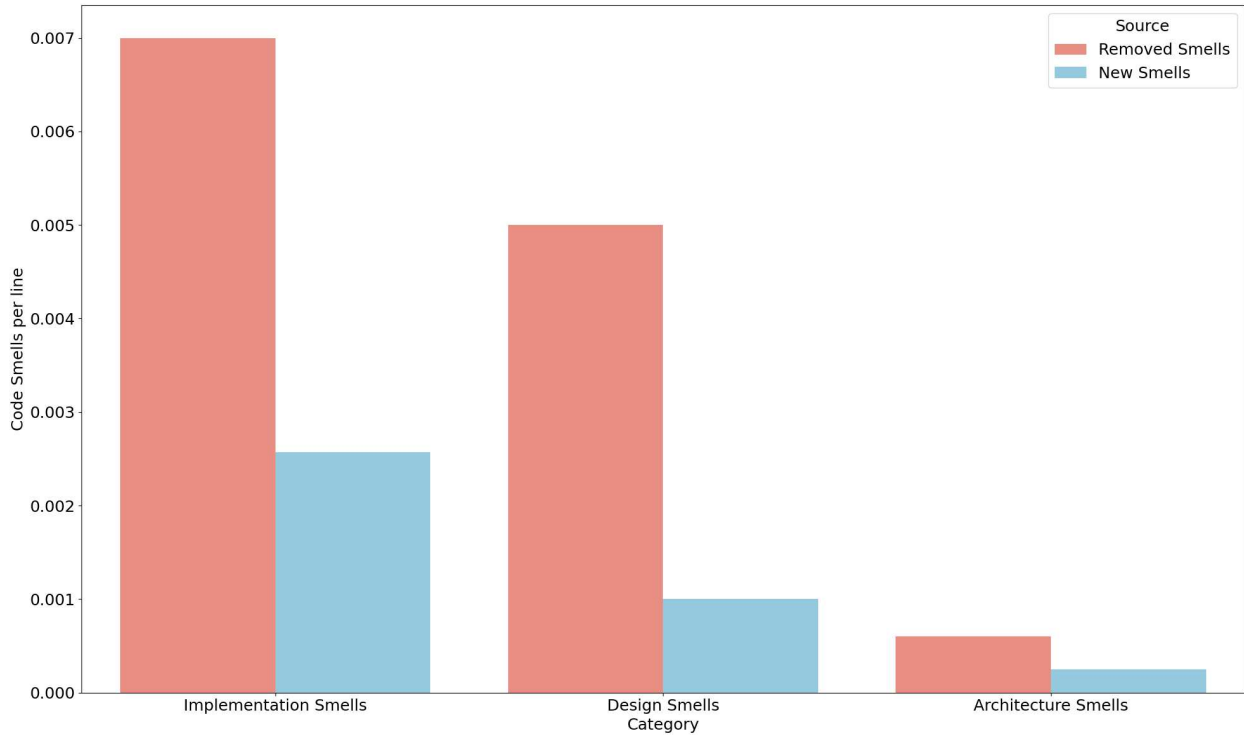


Figure 4.6: Removed and newly introduced smells in simulation systems

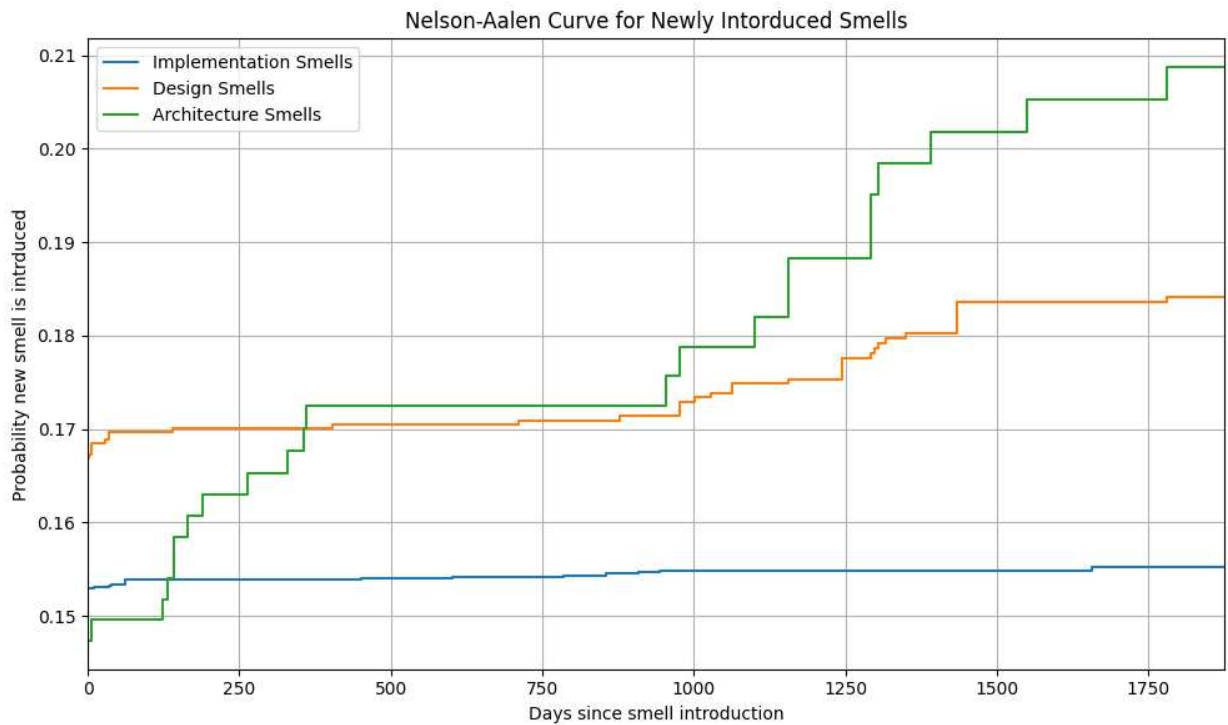


Figure 4.7: Nelson-Aalen survival curve for simulation systems

difference in risk ratios is due to the fact that refactoring Implementation smells tends to be more localized, focusing on specific methods or classes, while Design and Architecture smells often involve broader structural changes that can inadvertently introduce new issues. These large changes can lead to introducing new architectural smells, even when the intention is to improve the overall design.

We also determine the risk of refactoring activities by employing Nelson-Aalen survival analysis. Figure 4.7 shows the risk of introducing new code smells by refactoring activities in simulation software systems. We can see that the risk of introducing new Implementation smells remains fairly static at 15% over the whole 5 year period. Similarly, the risk of introducing new Design smells remains at 17% till the 100 day mark. However, we see a gradual increase for the next 500 days, settling at 18% at the end. On the other hand, the risk of refactoring architectural smells remain the highest. It starts from 15% and moves to 17% in the first 500 days, where it stabilizes for the next 500 days. After that, we see a sharp increase, going from 17% to 21%. This higher risk for architectural smells can be attributed to the fact that refactoring them involves complex system-wide modifications that affect multiple components and their interactions. This complexity increases the likelihood of introducing new code smells compared to more localized changes for refactoring Implementation and Design smells.

We also analyze the compositions of new and removed code smells in Figure 4.8, we see that the compositions of both groups are structurally similar. For Implementation smells, we see that 67.8% of Removed smells and 62.1% of New smells are Magic Number smells. Similarly, the Long Statement smell also constitutes 25.2% of removed and 27.5% of new code smells. This means that 93% of all removed smells and 89.6% of new smells belong to two categories – Magic Number and Long Statement code smells. However, for Design smells, we see that only 30.8% of removed smells belong to the Unused Abstraction category, which increases to 47.8% in the case of newly introduced smells. Deficient encapsulation makes up 28.6% of all removed smells, making it the second most removed smell. However, Broken Hierarchy takes the second spot with 19% of all new smells being from this category. Finally, for Architecture smells we see that Scattered Functionality makes up 52.9% of all removed and 33.3% of all newly introduced smells. The Cyclic Dependency is a close second, constituting

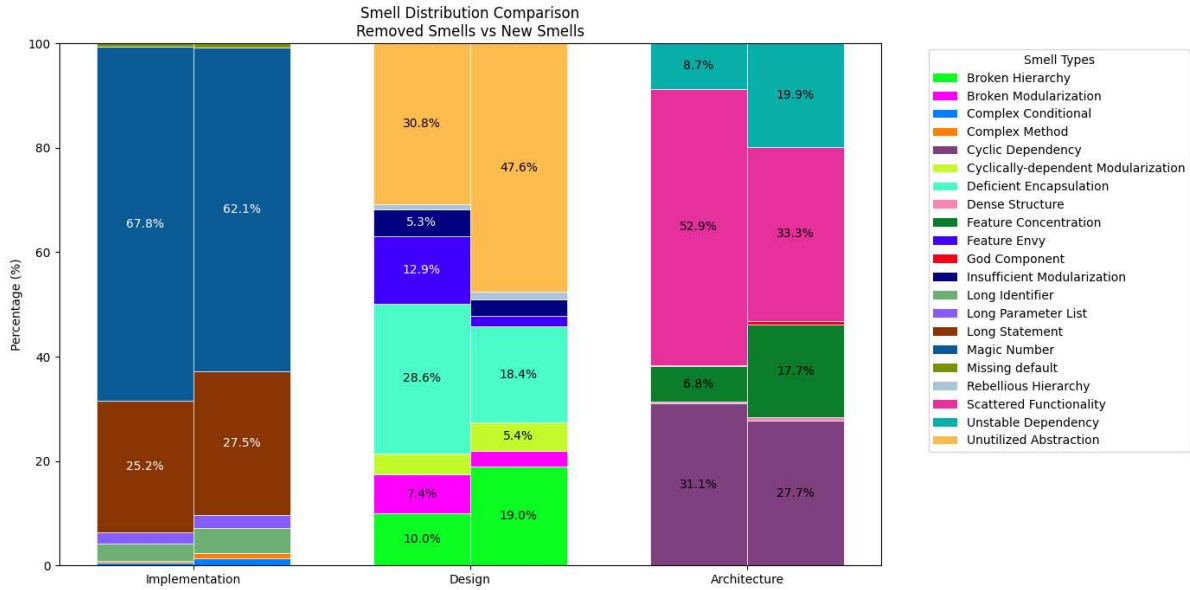


Figure 4.8: Comparison of composition of new and removed smells for simulation systems

31.3% of all removed and 27.7% of all new smells.

Table 4.4 examines the association between newly introduced code smells and the scope of refactoring, based on Cramer’s V we found statistically significant association between these two variables. For Implementation Smells, we found weak associations at both Method (0.1303) and Class (0.1429) levels, with only Package level (0.2152) refactoring techniques showing moderate association. Similarly, Design Smells showed consistently weak associations across all levels, ranging from 0.1536 for Method level to 0.1781 for Class level refactoring. On the other hand, Architecture

Table 4.4: Association between refactoring and newly introduced code smells

Code Smell	Scope of Refactoring	p < 0.05	Cramer’s V
<b>Implementation Smell</b>	Method Level	True	0.1303
	Class Level	True	0.1429
	Package Level	True	0.2152
<b>Design Smell</b>	Method Level	True	0.1536
	Class Level	True	0.1781
	Package Level	True	0.1766
<b>Architecture Smell</b>	Method Level	True	0.2451
	Class Level	True	0.2615
	Package Level	True	0.3591

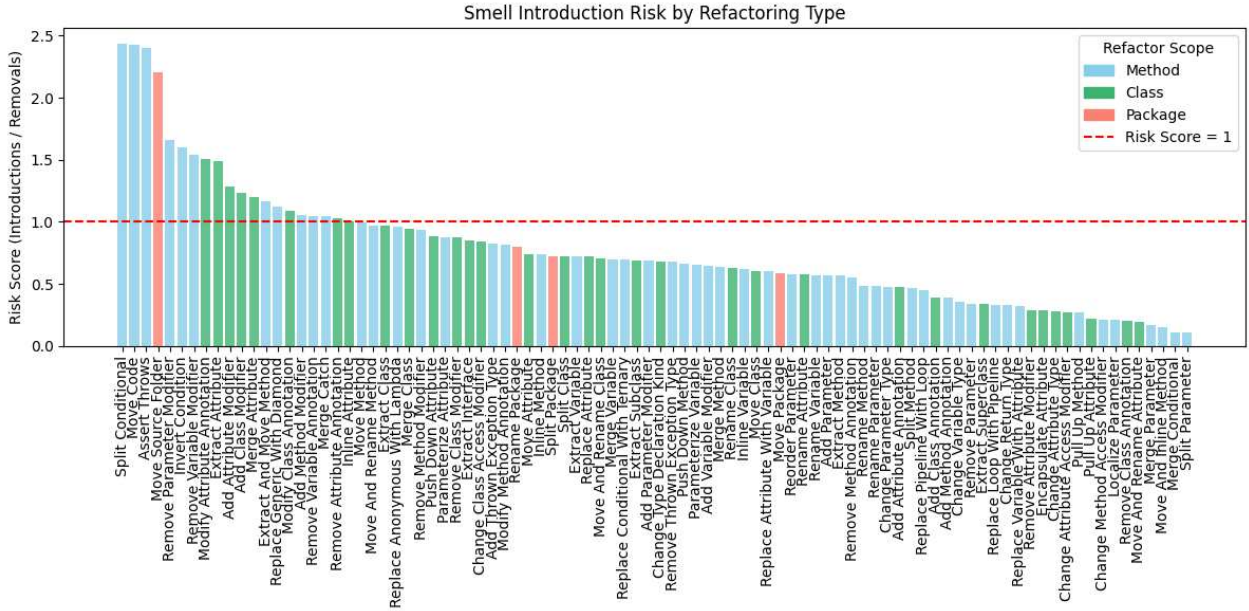


Figure 4.9: Risk of introducing new code smells for each refactoring technique

Smells exhibited the strongest associations overall, with moderate values at Method (0.2451) and Class (0.2615) levels, reaching their peak at Package level (0.3591). Comparing with Table 4.3, we see that while Package-level refactoring techniques are effective at removing Design and Implementation smells, they also introduce the most Architecture and Implementation smells. This highlights that while larger-scale changes can more effectively eliminate simpler code smells, they risk introducing more complex architectural issues.

Figure 4.9 shows the risk of introducing new code smells for each refactoring technique, where refactoring techniques have a risk score less than 1, signaling that they are more effective at removing code smells than introducing new ones.

We also found that, the Method level refactoring techniques – Split Conditional, Move code and Assert Throws have the highest risk of introducing new code smells. These refactoring techniques introduce 2.5 times more code smells than they remove, lowering overall code quality. The Move Source Folder refactoring technique is equally risky introducing new code smells. However, most other Method level and Package level refactoring techniques have a risk score of less than 1, indicating that they are more effective at removing code smells than introducing new ones. On the other hand, the Class level refactoring techniques are the least risky, with only Modify Attribute

Table 4.5: Trend of method level metrics after refactoring

Metric	$p < 0.05$	$z$	Trend
System Complexity	True	-2.770695	decreasing
Data Complexity	False	0.495953	no trend
Fan-In	True	-2.671305	decreasing
Fan-Out	True	-2.671305	decreasing
Nested Block Depth	True	-2.915064	decreasing
Number of Comparisons	False	-0.587698	no trend
Structural Complexity	True	-2.696153	decreasing
Total Lines of Code	True	-2.188566	decreasing
Cyclomatic Complexity	True	-2.761839	decreasing

Annotation and Extract Attribute refactoring techniques introducing 1.5 new smells for each removed smell. Contrasting with the data from Table 4.4, we see that Method level refactoring techniques show weak overall associations with introducing new smells, as evidenced by most of them having a low risk score. However, Split Conditional, Move Code and Assert Throws are significant outliers, introducing more code smells than they remove. This suggests that these specific techniques may require additional scrutiny to mitigate their impact on code quality.

**Summary of RQ2** We find that, refactoring introduces fewer code smells in simulation systems than in traditional systems, with an average 18% risk of new smells (Figure 4.6, 4.7). Most Method-level techniques show weak links to smell introduction, but outliers such as *Split Conditional*, *Move Code*, and *Assert Throws* introduce 2.5 times more smells than they remove (Table 4.4, Figure 4.9). This suggests that while most refactoring techniques are beneficial, developers should apply certain techniques cautiously.

### 4.3.3 RQ3: Do refactoring practices improve the maintainability of simulation software systems?

We analyze the impact of refactoring in simulation software systems by calculating several maintainability metrics (Section 2.3). Table 4.5 shows the trends of method level maintainability metrics, which suggest statistically significant improvements in maintainability of code due to refactoring activities. We see that, System Complexity ( $C_i$ ) decreased significantly ( $z = -2.77$ ), thanks to the reductions in Structural

Table 4.6: Trend of class level metrics after refactoring

Metric	$p < 0.05$	$z$	Trend
Class Average System Complexity	False	-1.843061	no trend
Class Total System Complexity	True	-2.882508	decreasing
Depth of Inheritance	False	-1.646778	no trend
Weighted Methods per Class	False	-1.097780	no trend
Lack of Cohesion of Methods	False	-0.451749	no trend
Number of Children	False	-1.511902	no trend

Table 4.7: Trend of package level metrics after refactoring

Metric	$p < 0.05$	$z$	Trend
Abstractness	False	-1.634290	no trend
Efferent Coupling	False	0.039661	no trend
Afferent Coupling	False	-0.544279	no trend
Instability	True	2.218516	increasing
Number of Classes	False	-0.662878	no trend
Number of Interfaces	True	-2.676063	decreasing
Package Average System Complexity	False	-0.778767	no trend
Package Total System Complexity	True	-2.420475	decreasing

Complexity (Si) ( $z = -2.70$ ). Cyclomatic Complexity (VG) ( $z = -2.76$ ) and Nested Block Depth (NBD) ( $z = -2.92$ ). Thus, our findings suggest that refactoring leads to simpler control logic and shallower nesting. In simulation systems we also see that Coupling metrics (Fan-in and Fan-out,  $z = -2.67$ ) and Total Lines of Code (TLOC) ( $z = -2.19$ ) declined significantly, indicating lower coupling. However, we do not notice any change in data complexity and NCOMP after the refactoring activities.

Table 4.6 shows the changes in maintainability metrics for classes in simulation systems. We see that the Class Total System Complexity (CITCi) decreased significantly ( $z = -2.88$ ), reflecting reduced cumulative method complexity. However, Class Relative System Complexity (CIRCi) ( $z = -1.84$ ), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Lack of Cohesion (LCOM), and Number of Children (NOCh) remained stable. This might indicate that refactoring practices in simulation systems often target internal method structures rather than altering class size or inheritance hierarchies.

We also see the changes for Package Level maintainability metrics in Table 4.7. This shows us a significant decrease in Package Total System Complexity (PkgTCi) ( $z$

= -2.42), which is consistent with our previous method and class-level trends. Similarly, Instability (I) across packages increased ( $z = 2.22$ ) despite stable Efferent Coupling (Ca) ( $z = 0.04$ ). And while the change in Afferent Coupling (Ce) ( $z = -0.54$ ) was not statistically significant, its decreasing value shifted the ratio to more outgoing dependencies. Since, Instability (I) is defined as the ratio of Efferent Coupling to Afferent Coupling, this shift in dependencies lead to increasing instability. However, we see that the Number of Interfaces (NOI) declining ( $z = -2.68$ ), while Abstractness (A) ( $z = -1.63$ ), Number of Classes (NOC) ( $z = -0.66$ ), and Package Relative Complexity (PkgRCi) ( $z = -0.78$ ) were unaffected by refactoring simulation systems.

**Summary of RQ3** We find that refactoring reduces Cyclomatic and Structural complexity in simulation systems, as well as cumulative complexity in both Class- and Package-level metrics (Table 4.5, 4.6). However, while Afferent Coupling does not change significantly ( $z = -0.54$ ,  $p \geq 0.05$ ), the increase in Instability ( $z = 2.22$ ) suggests that refactoring introduces more external dependencies (Table 4.7). Overall, refactoring is useful for developers of simulation systems as it helps simplify code structures and improves maintainability.

#### 4.4 Implication of Findings

The results of RQ1 have various important implications. First, we see that refactoring removes more code smells in simulation systems than in traditional systems, suggesting that periodic refactoring is more important for keeping simulation systems healthy. We also note that refactoring removes a small portion of the overall code smells in simulation systems. However, it is effective at significantly reducing code smells in a system over time, as very few smells survive after multiple rounds of refactoring efforts. This suggests that developers should dedicate more time to refactoring simulation systems as repeated refactoring efforts result in significantly less code smells surviving during development. In terms of refactoring techniques, we see that developers tend to make more small method-level localized changes, leading to their popularity. However, this popularity does not always translate to effectiveness, as evidenced by their overall low associations with removing Implementation and Design smells. However, larger Package level refactoring techniques with their stronger associations with removing

Implementation and Design smells are much less popular. These results imply that developers of simulation systems should be more willing to make larger refactoring changes to the codebase to maximize the removal of code smells.

Similarly, the results of RQ2 have important implications regarding the risk of refactoring activities in simulation systems. First, we see that refactoring introduces fewer code smells than it removes, suggesting refactoring as a net positive practice for simulation systems. However, the risk of introducing new Architecture and Design smells increases slightly from 15% to 21% over subsequent refactoring commits, resulting in degraded system design over time. We also see that there weak association between new smells and Method level refactoring techniques, suggesting that they are less risky than Package and Class level refactoring techniques. These findings imply that developers should not be afraid of refactoring simulation systems due to its low risk of introducing new smells. However, significant outliers such as Split Conditional, Move Code and Assert Throws, which introduce 2.5 times more code smells than they remove. These refactoring techniques should be used sparingly by developers in simulation systems due to the higher than usual risks involved.

From RQ3, we see that refactoring practices enhance the maintainability of simulation systems by reducing their structural complexity. Method-level complexity such as Cyclomatic Complexity and Structural Complexity decrease due to refactoring, indicating simpler control flow and concise logic respectively. Similarly, Class and Package level complexity metrics also decline and demonstrate cumulative benefits of refactoring activities in simulation systems. And although the decrease in Afferent Coupling is not significant by itself, its change compared to Efferent Coupling leads to an increase in Package Instability (I). This suggests that refactoring practices in simulation systems lead to more outgoing dependencies, making them harder to maintain. These findings show that developers should refactor simulation systems to decrease their complexity and make them easier to understand and maintain. However, developers should also be aware of the wary of achieving this reduced complexity by shifting responsibilities to outgoing dependencies, as it can make the simulation more unstable.

## 4.5 Threats to Validity

### 4.5.1 Threats to internal validity

Threats to *internal validity* stem from errors or biases in the experimental setup [99]. Since we select repositories from different domains, the quality of their code bases might not be comparable. To mitigate this threat, we used several metrics such as star count, number of contributors, and number of issues and attempted to select the repositories that are of comparable quality.

As we also analyze the refactoring practices over time for multiple different projects, they might have significantly different development timelines, making comparisons difficult. To combat this, we set a specific observation period common to all projects. Since some projects might have different number of commits over the same observation period, we measure the developmental progress as a percentage of the total number of commits in the observation period. This allows us to compare projects with different commit frequencies on a common scale.

### 4.5.2 Threats to conclusion validity

Threats to *conclusion validity* concern the accuracy of conclusions [101]. During repository selection, we end up with 104 simulation systems and 272 traditional systems after filtering. Due to the differences in sample sizes, the statistical results might be hard to infer. To mitigate this threat, we randomly sample an equal number of traditional repositories to make the comparison fair and unbiased. We also use multiple non-parametric statistical tests to avoid any assumptions behind the underlying distribution of the samples.

### 4.5.3 Threats to external validity

Threats to *external validity* describe concerns regarding the generalizability of any findings. To mitigate these threats and achieve diversity in our dataset, we select simulation systems using both topic-based and keyword-based searches from GitHub, covering various simulation domains. Similarly, our traditional systems span multiple domains, including web development, mobile development, and desktop applications.

## 4.6 Related Works

### 4.6.1 Effectiveness of Refactoring

There have been numerous empirical studies investigating the effectiveness of refactoring software systems, with varying results. Kataoka et al [11] found that Extract Method and Extract Class refactoring techniques can result in 16.2% and 12.6% less code smells in the source code respectively. Similarly, Habchi et al [113] found that 79% of Android code smells are successfully removed through constant maintenance activities. However, Cedrim et al looked at 10 different types of refactoring changes in 23 projects and found that 57% of refactorings do not remove any code smells. They also found that only 9.7% of refactoring techniques are responsible for removing any code smells. Similarly, Yoshida et al [114] found that only 42% of refactoring operations were done on code elements with code smells, where only 7% of these operations removed any code smells. Similar experiments were conducted by Hamdi et al [115] on 652 android releases considering 10 Object Oriented and 15 Android specific code smells. They found that only 5% of Object Oriented smells and 1.5% of Android specific smells were removed by refactoring. In our study, we analyze 104 simulation and 272 traditional software systems and analyze the effectiveness of refactoring practices in these systems. Our findings contrast these studies by showing that refactoring simulation systems is more effective at removing code smells than traditional systems, with approximately 35% of them being eliminated over time. While individual refactoring commits may address only a small portion of existing code smells, the cumulative effect over five years significantly reduces smell survival rates from 95% to approximately 65%.

### 4.6.2 Risks of Refactoring

Prior research on the risks of refactoring has produced mixed results. A study by Tufano et al [1] on 200 open source projects found that Enhancement and Feature requests introduce more code smells than refactoring operations. However, Bibiano et al. [116] analyzed 4607 batch refactoring operations and found that 51% of them were responsible for introducing new code smells. Moreover, Tavares et al [117] analyzed seven different systems and found that refactoring practices replacing existing elements

are the riskiest, being responsible for introducing Lazy Class and Refused Bequest code smells on 57.14% of systems. In our research, we analyze the risks of refactoring changes in simulation systems by detecting their introduced code smells. Moreover, we also find the risks of introducing new code smells through refactoring over a span of 5 years. However, our findings reveal that refactoring has net positive impact on simulation systems, removing more code smells than it introduces across all abstraction levels. However, developers should be aware that the risk of introducing new Design and Architecture smells increases over time, reaching 18% and 21% respectively.

### 4.6.3 Impacts of Refactoring

Prior research has used maintainability metrics to determine the impacts of refactoring in simulation systems. Kannangara et al [118] reported a significant increase in the Halstead Volume and Lines of code of refactored code. However, they found no positive changes in coupling or complexity of code. Similarly, Geppert et al [119] found that after a systematic refactoring phase, customer-reported defects and maintenance efforts for their resolution declined in subsequent releases. However, an investigation of three different systems from Ashayeb et al [108] found that common refactoring techniques – Extract method, Replace Temp with Query, Inline class, and Extract Superclass– did not necessarily enhance external quality attributes such as: adaptability, maintainability or testability. Our study attempts to describe the link between refactoring simulation systems and maintainability metrics of their code by investigating metric trends over time.

## 4.7 Summary

This empirical study analyzes refactoring practices in 104 simulation and 272 traditional software repositories. Our findings demonstrate that refactoring simulation systems effectively improves code quality with manageable risks. First, we find that refactoring simulation systems removes code smells more effectively than in traditional systems, eliminating approximately 35% of smells over time. While individual commits have limited impact, the cumulative effect over five years reduces smell survival rates from 95% to 65%. Package-level refactoring was effective at removing Implementation and Design smells, while Method-level changes work better for Architecture smells.

Furthermore, our risk analysis shows that refactoring removes more smells than it introduces across all abstraction levels. However, the risk of new Design and Architecture smells increases over time to 18% and 21% respectively. And although Method level refactoring techniques are not strongly associated with an introduction of new code smells, some techniques like Split Conditional, Move Code, and Assert Throws are significant outliers, introducing 2.5 times more smells than they remove. Finally, our investigation into maintainability metrics after refactoring shows significant reductions in complexity and coupling of simulation software systems while preserving core logic through stable Data Complexity and inheritance measures. This translates to reduced technical debt without compromising simulation fidelity. In conclusion, our evidence supports refactoring as a valuable maintenance strategy for simulation systems, providing guidance for improving their long-term sustainability.

## Chapter 5

### Conclusion and Future Work

#### 5.1 Conclusion

Code smells are an indication of deeper problems in a software system. They do not prevent a software system from working but slow down its development and increase technical debt. Refactoring is a natural choice to address code smells. It performs small changes to the internal structure of code without altering the external behavior.

Code smells and refactoring practices have been studied for not only traditional software systems [10], [11], [12] but also specialized systems such as database systems [13], [14], [15], android systems [12], [16], [17], and even deep-learning systems [18], [19]. However, no studies investigate the code smells or refactoring practices from simulation software systems.

As simulation systems imitate real world systems and have found application in many domains including military operations [20], transportation [21], [22], aviation [23], [24], and medicine [25], [26]. Despite their importance, these systems have received limited attention from the software engineering community, particularly regarding code quality issues and refactoring practices. This thesis addresses this gap through two comprehensive empirical studies as follows.

- The first study examines the prevalence, evolution, and impact of code smells in 155 simulation and 327 traditional software systems from GitHub. Our findings reveal significant differences in the distribution of code smells between simulation and traditional systems, with simulation systems containing substantially more Magic Number, Long Statement, and Long Parameter List code smells per line of code. We also observe that code smells in simulation systems tend to survive considerably longer than in traditional systems, with some smells like Broken Hierarchy lasting for over 3,661 days. Interestingly, despite the prevalence of these smells, our statistical tests show no significant association between code

smells and bugs in simulation systems, suggesting that these smells may impact other quality attributes (e.g., non-functional attributes) rather than directly causing bugs.

Our study contains valuable insights for both developers and modelers involved in simulation systems. First, we see that currently simulation systems contain significantly more code smells which are only refactored late into development. This can be remedied by a more proactive approach to developing simulation systems, where code smells are removed other development activities such as feature additions or bug fixes. This can decrease the amount of code smells in both the short and long term, making code easier to maintain. Moreover, our observation regarding significantly larger amounts of Magic Number, Long Statement, and Long Parameter List smells in simulation systems shows a clear focus on adding new features over code quality. Thus, modelers should also consider the existing code structures such as named constants and classes when proposing new features. This can make future additions easier, as they would not be hampered by existing code smells from previous additions.

- Being intrigued by these findings, the second study investigates the effectiveness, risks, and impact of refactoring practices in 104 simulation and 272 traditional software systems. Our findings indicate that refactoring removes approximately 35% of Implementation, Design, and Architecture smells in simulation systems. Package-level refactoring techniques prove to be the most effective at removing Implementation and Design smells. And although Method-level refactoring techniques have the lowest associations with introducing new code smells, techniques such as Split Conditional, Move Code and Assert Throws are significant outliers, introducing 2.5 times more code smells than they remove. According to our findings, refactoring significantly improves simulation system's maintainability by reducing Structural and Cyclomatic complexity while maintaining core logic integrity and architectural foundations.

This study illuminates valuable actionable insights for both developers and project managers of simulation systems. First, developers of simulation systems should not be afraid refactoring simulation systems, as they are very effective

at removing code smells with little risk of introducing new ones. However, applications of certain techniques such as Split Conditional, Move Code and Assert Throws should be carefully monitored and only applied to the codebase when necessary. Similarly, Project managers should also allocate more time for dedicated refactoring efforts during development to remove code smells. As repeated refactoring efforts are very effective at removing code smells while only introducing few Design and Architectural smells, a higher allocation of dedicated refactoring efforts can help sustain software quality.

## **5.2 Limitations**

Despite our best efforts, there exist several limitations that were not addressed in our thesis. We present the limitations of each study below.

### **5.2.1 Code Smell Analysis**

First, we considered different simulation systems collectively to contrast them with traditional systems. However, simulation systems can employ very different methods (e.g., numerical, event-based, or agent-based modeling), which were not accounted for in our analysis. Second, our primary goal was to examine whether code smells in simulation systems differ from those in traditional systems, by focusing on code smells that were common to both. While our study demonstrated that simulation systems do indeed exhibit distinct code smells, it overlooked the fact that code smells are subjective and can be different for each systems. For instance, we did not investigate whether smells such as Long Statement or Long Parameter List naturally occur within simulation systems rather than representing poor design choices. Moreover, we did not consider anti-patterns specific to simulation systems. While some of these issues might be harmless for traditional systems, they might be problematic for simulation systems.

### **5.2.2 Refactoring**

Our second study is also limited by various factors. First, we examine the refactoring practices of simulation systems by focusing solely on refactoring commits. While

this approach highlights the effectiveness and risks of explicit refactoring efforts, it overlooks other commits such as feature implementation or bug fixes. These changes can also restructure code and eliminate code smells while not being a part of dedicated refactoring efforts. Second, we evaluate the impact of refactoring practices on the maintainability of simulation software through a range of software metrics. Our study uses a diverse set of metrics for all three levels of abstractions, which are presented as numeric values. While they can indicate code quality and maintainability concerns, they miss other relevant factors such as developer experience, tooling and quality of documentation. Which could affect the maintainability of a software system.

### **5.3 Future Work**

We have identified several directions for future research from both our studies. We present the potential future work for each study below.

#### **5.3.1 Code Smell Analysis**

There are several avenues for future work from our first study. Simulation systems (e.g. iTLE, Sumo) have unique characteristics such as complex mathematical models, time-dependent behavior, stochastic processes, and intricate state management which leads to domain-specific code quality issues. These systems often require extensive parameter configurations, numerical computations, and specialized algorithms that can manifest in ways different from traditional software systems. And while our study provides a understanding of the prevalence, evolution, and impact of code smells in simulation systems, there is still much to explore. First, we plan to investigate domain-specific code smells that might be unique to simulation software systems. Second, we aim to develop specialized detection tools that can identify simulation-specific code smells with higher precision. Finally, we plan to investigate the relationship between code smells and other non-functional quality attributes such as performance, state management, and security, which are particularly critical for simulation systems.

#### **5.3.2 Refactoring Practices**

We envision several future research directions from our second work.

First, we plan to develop refactoring recommendation systems specifically tailored for simulation software systems. This system should be able to suggest optimal refactoring techniques based on code smells specific to simulation systems. This is important because existing refactoring tools do not account for simulation-specific concerns (e.g., complex mathematical models, time-dependent behaviors, and stochastic processes) that require specialized detection and refactoring approaches. Second, we aim to investigate the impact of refactoring on simulation performance metrics, as performance is often a critical concern in simulation systems. In particular, we plan to analyze how refactoring techniques can optimize numerical computations, memory usage, and parallel execution in large-scale simulations. Finally, we plan to develop guidelines and best practices for refactoring simulation systems that can help developers maintain high-quality code while preserving the complex algorithmic integrity, essential for accurate simulations. This is particularly critical since even minor changes to simulation logic can propagate errors through the system, potentially invalidating results or introducing subtle behavioral changes that may go undetected in standard testing procedures.

## Bibliography

- [1] M. Tufano et al., “When and why your code starts to smell bad (and whether the smells go away),” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.
- [2] A. Yamashita and L. Moonen, “Do code smells reflect important maintainability aspects?” In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 306–315. DOI: 10.1109/ICSM.2012.6405287
- [3] A. Yamashita and L. Moonen, “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 682–691. DOI: 10.1109/ICSE.2013.6606614
- [4] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, “An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension,” in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 181–190. DOI: 10.1109/CSMR.2011.24
- [5] F. Jaafar, A. Lozano, Y.-G. Guéhéneuc, and K. Mens, “On the analysis of co-occurrence of anti-patterns and clones,” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2017, pp. 274–284.
- [6] M. A. Alkandari, A. Kelkawi, and M. O. Elish, “An empirical investigation on the effect of code smells on resource usage of android mobile applications,” *IEEE Access*, vol. 9, pp. 61 853–61 863, 2021.
- [7] O. Hamdi, A. Ouni, E. A. AlOmar, and M. W. Mkaouer, “An empirical study on code smells co-occurrences in android applications,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2021, pp. 26–33. DOI: 10.1109/ASEW52652.2021.00018

- [8] H. Jebnoun, H. Ben Braiek, M. M. Rahman, and F. Khomh, “The scent of deep learning code: An empirical study,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 420–430.
- [9] N. Bessghaier, A. Ouni, and M. W. Mkaouer, “On the diffusion and impact of code smells in web applications,” in *International Conference on Services Computing*, Springer, 2020, pp. 67–84.
- [10] A. Kaur and M. Kaur, “Analysis of code refactoring impact on software quality,” in *MATEC Web of Conferences*, EDP Sciences, vol. 57, 2016, p. 02 012.
- [11] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, “A quantitative evaluation of maintainability enhancement by refactoring,” in *International Conference on Software Maintenance, 2002. Proceedings*, IEEE, 2002, pp. 576–585.
- [12] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, “Refactoring android java code for on-demand computation offloading,” *ACM Sigplan Notices*, vol. 47, no. 10, pp. 233–248, 2012.
- [13] G. Vial, “Database refactoring: Lessons from the trenches,” *IEEE Software*, vol. 32, no. 6, pp. 71–79, 2015.
- [14] P. Khumnin and T. Senivongse, “Sql antipatterns detection and database refactoring process,” in *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, IEEE, 2017, pp. 199–205.
- [15] A. M. Boehm, D. Seipel, A. Sickmann, and M. Wetzka, “Squash: A tool for analyzing, tuning and refactoring relational database applications,” in *International Conference on Applications of Declarative Programming and Knowledge Management*, Springer, 2007, pp. 82–98.
- [16] O. Hamdi, A. Ouni, M. Ó. Cinnéide, and M. W. Mkaouer, “A longitudinal study of the impact of refactoring in android applications,” *Information and Software Technology*, vol. 140, p. 106 699, 2021.

- [17] O. Hamdi, A. Ouni, E. A. AlOmar, M. Ó. Cinnéide, and M. W. Mkaouer, “An empirical study on the impact of refactoring on quality metrics in android applications,” in *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, IEEE, 2021, pp. 28–39.
- [18] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu, “A survey of deep learning based software refactoring,” *arXiv preprint arXiv:2404.19226*, 2024.
- [19] S. Wang, X. Hu, B. Wang, W. Yao, X. Xia, and X. Wang, “Insights into deep learning refactoring: Bridging the gap between practices and expectations,” *arXiv preprint arXiv:2405.04861*, 2024.
- [20] R. R. Hill and J. O. Miller, “A history of united states military simulation,” in *2017 Winter Simulation Conference (WSC)*, IEEE, 2017, pp. 346–364.
- [21] C. Osorio and L. Chong, “A computationally efficient simulation-based optimization algorithm for large-scale urban transportation problems,” *Transportation Science*, vol. 49, no. 3, pp. 623–636, 2015.
- [22] D. P. Möller, *Introduction to Transportation Analysis, Modeling and Simulation* (Simulation Foundations, Methods and Applications). London: Springer London, 2014.
- [23] E. Salas, C. A. Bowers, and L. Rhodenizer, “It is not how much you have but how you use it: Toward a rational use of simulation to support aviation training,” *The international journal of aviation psychology*, vol. 8, no. 3, pp. 197–208, 1998.
- [24] R. Page, “Lessons from aviation simulation,” in *Manual of simulation in healthcare*, Oxford University Press, Oxford, 2008, pp. 37–49.
- [25] B. Chakravarthy, E. T. Haar, S. S. Bhat, C. E. McCoy, T. K. Denmark, and S. Lotfipour, “Simulation in medical school education: Review for emergency medicine,” *Western Journal of Emergency Medicine*, vol. 12, no. 4, p. 461, 2011.

- [26] M. D. Beal, J. Kinnear, C. R. Anderson, T. D. Martin, R. Wamboldt, and L. Hooper, “The effectiveness of medical simulation in teaching medical students critical care medicine: A systematic review and meta-analysis,” *Simulation in Healthcare*, vol. 12, no. 2, pp. 104–116, 2017.
- [27] H. Higham and B. Baxendale, “To err is human: Use of simulation to enhance training and patient safety in anaesthesia,” *BJA: British Journal of Anaesthesia*, vol. 119, no. suppl\_1, pp. i106–i114, 2017.
- [28] R. Aggarwal et al., “Training and simulation for patient safety,” *BMJ Quality & Safety*, vol. 19, no. Suppl 2, pp. i34–i43, 2010.
- [29] E. H. Page, “Simulation modeling methodology: Principles and etiology of decision support,” 1994.
- [30] T. Østergård, R. L. Jensen, and S. E. Maagaard, “Building simulations supporting decision making in early design—a review,” *Renewable and Sustainable Energy Reviews*, vol. 61, pp. 187–201, 2016.
- [31] C. A. Aumann, “A methodology for developing simulation models of complex systems,” *Ecological Modelling*, vol. 202, no. 3-4, pp. 385–396, 2007.
- [32] R. Fujimoto, C. Bock, W. Chen, E. Page, and J. H. Panchal, *Research challenges in modeling and simulation for engineering complex systems*. Springer, 2017.
- [33] G. P. Marcilio, J. J. de Assis Rangel, C. L. M. de Souza, E. Shimoda, F. F. da Silva, and T. A. Peixoto, “Analysis of greenhouse gas emissions in the road freight transportation using simulation,” *Journal of Cleaner Production*, vol. 170, pp. 298–309, 2018.
- [34] C. Osorio and M. Bierlaire, “A simulation-based optimization framework for urban transportation problems,” *Operations Research*, vol. 61, no. 6, pp. 1333–1345, 2013.

- [35] F. E. Cellier and E. Kofman, *Continuous System Simulation*. Springer Science & Business Media, 2006.
- [36] E. E. Ogheneovo et al., “On the relationship between software complexity and maintenance costs,” *Journal of Computer and Communications*, vol. 2, no. 14, p. 1, 2014.
- [37] N. Cardozo, I. Dusparic, and C. Cabrera, “Prevalence of code smells in reinforcement learning projects,” in *2023 IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI (CAIN)*, IEEE, 2023, pp. 37–42.
- [38] A. L. Sabóia, A. D. F. Martins, C. S. Melo, J. M. Monteiro, C. T. de Souza, and J. de Castro Machado, “Prevalence of bad smells in c# projects,” in *ICEIS (2)*, 2020, pp. 424–431.
- [39] F. Goncalves de Almeida Filho et al., “Prevalence of bad smells in pl/sql projects,” in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 116–121. DOI: 10.1109/ICPC.2019.00025
- [40] M. Tufano et al., “When and why your code starts to smell bad,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 403–414. DOI: 10.1109/ICSE.2015.59
- [41] A. Rani and J. K. Chhabra, “Evolution of code smells over multiple versions of softwares: An empirical investigation,” in *2017 2nd International Conference for Convergence in Technology (I2CT)*, 2017, pp. 1093–1098. DOI: 10.1109/I2CT.2017.8226297
- [42] A. Chatzigeorgiou and A. Manakos, “Investigating the evolution of code smells in object-oriented systems,” *Innovations in Systems and Software Engineering*, vol. 10, pp. 3–18, 2014.
- [43] W. Li and R. Shatnawi, “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution,” *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

- [44] 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. DOI: 10.1109/ESEM.2009.5314231
- [45] A. S. Cairo, G. d. F. Carneiro, and M. P. Monteiro, “The impact of code smells on software bugs: A systematic literature review,” *Information*, vol. 9, no. 11, p. 273, 2018.
- [46] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, “Are test smells really harmful? an empirical study,” *Empirical Software Engineering*, vol. 20, pp. 1052–1094, 2015.
- [47] B. Van Oort, L. Cruz, M. Aniche, and A. Van Deursen, “The prevalence of code smells in machine learning projects,” in *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*, IEEE, 2021, pp. 1–8.
- [48] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, “Understanding code smells in android applications,” in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft ’16, Austin, Texas: Association for Computing Machinery, 2016, pp. 225–234, ISBN: 9781450341783. DOI: 10.1145/2897073.2897094 [Online]. Available: <https://doi.org/10.1145/2897073.2897094>
- [49] S. G. Carvalho, M. Aniche, J. Veríssimo, R. S. Durelli, and M. A. Gerosa, “An empirical catalog of code smells for the presentation layer of android apps,” *Empirical Software Engineering*, vol. 24, pp. 3546–3586, 2019.
- [50] B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, “On the prevalence, impact, and evolution of sql code smells in data-intensive systems,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR ’20, Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 327–338, ISBN: 9781450375177. DOI: 10.1145/3379597.3387467 [Online]. Available: <https://doi.org/10.1145/3379597.3387467>

- [51] J. Bernard and T. Kintziger, “Mongodb code smells: Defining, classifying and detecting code smells for mongodb interactions in java programs,” Ph.D. dissertation, Master’s thesis, Faculté d’informatique. Université de Namur, Belgium, 2021.
- [52] G. Hecht, N. Moha, and R. Rouvoy, “An empirical study of the performance impacts of android code smells,” in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft ’16, New York, NY, USA: Association for Computing Machinery, 2016, pp. 59–69. DOI: 10.1145/2897073.2897100 [Online]. Available: <https://doi.org/10.1145/2897073.2897100>
- [53] A. Chug and M. Gupta, “A quality enhancement through defect reduction using refactoring operation,” in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, IEEE, 2017, pp. 1869–1875.
- [54] 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*, ser. BRIDGE ’16, Austin, Texas: Association for Computing Machinery, 2016, pp. 1–4, ISBN: 9781450341530. DOI: 10.1145/2896935.2896938 [Online]. Available: <https://doi.org/10.1145/2896935.2896938>
- [55] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [56] E. L. Kaplan and P. Meier, “Nonparametric estimation from incomplete observations,” in *Breakthroughs in Statistics: Methodology and Distribution*, S. Kotz and N. L. Johnson, Eds. New York, NY: Springer New York, 1992, pp. 319–337, ISBN: 978-1-4612-4380-9. DOI: 10.1007/978-1-4612-4380-9\_25 [Online]. Available: [https://doi.org/10.1007/978-1-4612-4380-9\\_25](https://doi.org/10.1007/978-1-4612-4380-9_25)

- [57] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, New York, New York, USA: ACM Press, 2018, pp. 908–911, ISBN: 9781450355735. DOI: 10.1145/3236024.3264598 [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3236024.3264598>
- [58] T. Sharma, *Designite - A Software Design Quality Assessment Tool* — *doi.org*, <https://doi.org/10.5281/zenodo.2566832>, [Accessed 27-01-2024].
- [59] *GitHub - rodhilton/jasome: JaSoMe (Java Source Metrics) - Object Oriented Metrics analyzer for Java code* — *github.com*, <https://github.com/rodhilton/jasome>, [Accessed 21-05-2025].
- [60] *Understand Download Free (Windows) - 7.1 Build 1231* — *softpedia.com*, <https://www.softpedia.com/get/Programming/Coding-languages-Compilers/Understand.shtml>, [Accessed 28-07-2025].
- [61] *Sonar: Methods and Tools*, <https://www.methodsandtools.com/PDF/mt201001.pdf>, [Accessed 28-07-2025].
- [62] T. Sharma and D. Spinellis, “A survey on software smells,” *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.
- [63] R. C. Martin, *The clean coder: a code of conduct for professional programmers*. Pearson Education, 2011.
- [64] M. Thomas, “Software quality metrics to identify risk,” *Department of Homeland Security Software Assurance Working Group*, 2008.

- [65] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinianian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18, Gothenburg, Sweden: ACM, 2018, pp. 483–494, ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180206 [Online]. Available: <http://doi.acm.org/10.1145/3180155.3180206>
- [66] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.
- [67] C. M. Robert, “Agile software development,” *Martin: Prentice Hall*, 2003.
- [68] A. M. Helmenstine, *Null hypothesis definition and examples*, <https://www.thoughtco.com/definition-of-null-hypothesis-and-examples-605436>, Accessed: 2024-6-4, Nov. 2010.
- [69] H. B. Mann and D. R. Whitney, “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other,” *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. DOI: 10.1214/aoms/1177730491 [Online]. Available: <https://doi.org/10.1214/aoms/1177730491>
- [70] J. Neyman, *The emergence of mathematical statistics: A historical sketch with particular reference to the united states. on the history of statistics and probability*, ed. db owen, 1976.
- [71] N. Cliff, “Dominance statistics: Ordinal analyses to answer ordinal questions,” *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.
- [72] K. Meissel and E. S. Yao, “Using cliff’s delta as a non-parametric effect size measure: An accessible web app and r tutorial,” *Practical Assessment, Research, and Evaluation*, vol. 29, no. 1, 2024.

- [73] K. Pearson, “X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling,” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900.
- [74] F. Pellegrin, Z. Yücel, A. Monden, and P. Leelaprute, “Task estimation for software company employees based on computer interaction logs,” *Empirical Software Engineering*, vol. 26, no. 5, p. 98, 2021.
- [75] E. Klotins et al., “Use of agile practices in start-ups,” *arXiv preprint arXiv:2402.09555*, 2024.
- [76] V. Gupta and L. B. Singh, “Perception of it professionals towards root causes of software bugs,” *International Journal of Computer Applications*, vol. 975, p. 8887,
- [77] H. Cramér, *Mathematical methods of statistics*. Princeton university press, 1999, vol. 26.
- [78] S. J. Oishwee, Z. Codabux, and N. Stakhanova, “An exploratory study on the relationship of smells and design issues with software vulnerabilities,” in *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, 2022, pp. 16–20.
- [79] E. L. Kaplan and P. Meier, “Nonparametric estimation from incomplete observations,” *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 457–481, 1958. DOI: 10.1080/01621459.1958.10501452 eprint: <https://www.tandfonline.com/doi/pdf/10.1080/01621459.1958.10501452>. [Online]. Available: <https://www.tandfonline.com/doi/abs/10.1080/01621459.1958.10501452>
- [80] D. R. Cox, “Regression models and life-tables,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 34, no. 2, pp. 187–202, 1972.

- [81] W. Nelson, "Hazard plotting for incomplete failure data," *Journal of Quality Technology*, vol. 1, no. 1, pp. 27–52, 1969.
- [82] P. D. Valz, A. I. McLeod, and M. E. Thompson, "Cumulant generating function and tail probability approximations for kendall's score with tied rankings," *The Annals of Statistics*, vol. 23, no. 1, pp. 144–160, 1995.
- [83] S. A. Stouffer, E. A. Suchman, L. C. DeVinney, S. A. Star, and R. M. Williams Jr, "The american soldier: Adjustment during army life.(studies in social psychology in world war ii), vol. 1," 1949.
- [84] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16, Austin, Texas: Association for Computing Machinery, 2016, pp. 59–69, ISBN: 9781450341783. DOI: 10.1145/2897073.2897100 [Online]. Available: <https://doi.org/10.1145/2897073.2897100>
- [85] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018, ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2017.12.034> [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121217303114>
- [86] M. Agnihotri and A. Chug, "A systematic literature survey of software metrics, code smells and refactoring techniques," *Journal of Information Processing Systems*, vol. 16, no. 4, pp. 915–934, 2020.
- [87] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110 610, 2020.
- [88] F. E. Cellier and E. Kofman, *Continuous system simulation*. Springer Science & Business Media, 2006.

- [89] *GitHub REST API documentation - GitHub Docs* — *docs.github.com*, <https://docs.github.com/en/rest?apiVersion=2022-11-28>, [Accessed 20-06-2024].
- [90] S. Rose, D. Engel, N. Cramer, and W. Cowley, “Automatic keyword extraction from individual documents,” *Text mining: applications and theory*, pp. 1–20, 2010.
- [91] D. Sas, P. Avgeriou, I. Pigazzini, and F. Arcelli Fontana, “On the relation between architectural smells and source code changes,” *Journal of Software: Evolution and Process*, vol. 34, no. 1, e2398, 2022. DOI: <https://doi.org/10.1002/smr.2398> eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2398>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2398>
- [92] *Commands &#x2014; designite documentation* — *designite-tools.com*, <https://github.com/tushartushar/DesigniteJava>, [Accessed 23-06-2024].
- [93] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, “Is it a bug or an enhancement? a text-based approach to classify change requests,” in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, 2008, pp. 304–318.
- [94] Mockus and Votta, “Identifying reasons for software changes using historic databases,” in *Proceedings 2000 international conference on software maintenance*, IEEE, 2000, pp. 120–130.
- [95] H. Zhong and Z. Su, “An empirical study on real bug fixes,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 913–923.
- [96] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

- [97] L. Zhang, H. Ray, J. Priestley, and S. Tan, “A descriptive study of variable discretization and cost-sensitive logistic regression on imbalanced credit data,” *Journal of Applied Statistics*, vol. 47, no. 3, pp. 568–581, 2020.
- [98] R. Mahbub, M. M. Rahman, and M. A. Habib, *On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software*, version 1.0.0. [Online]. Available: <https://zenodo.org/records/13621760>
- [99] Y. Tian, D. Lo, and J. Lawall, “Automated construction of a software-specific word similarity database,” in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, IEEE, 2014, pp. 44–53.
- [100] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, “Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm,” *Information and Software Technology*, vol. 99, pp. 164–176, 2018.
- [101] M. A. García-Pérez, “Statistical conclusion validity: Some common threats and simple remedies,” *Frontiers in psychology*, vol. 3, p. 29584, 2012.
- [102] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.
- [103] A. Yamashita and L. Moonen, “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 682–691.
- [104] A. Yamashita and L. Moonen, “Do code smells reflect important maintainability aspects?” In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 306–315.
- [105] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, “An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension,” in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 181–190.

- [106] F. Jaafar, A. Lozano, Y.-G. Guéhéneuc, and K. Mens, “On the analysis of co-occurrence of anti-patterns and clones,” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2017, pp. 274–284.
- [107] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, “An empirical study to improve software security through the application of code refactoring,” *Information and Software Technology*, vol. 96, pp. 112–125, 2018.
- [108] M. Alshayeb, “Empirical investigation of refactoring effect on software quality,” *Information and Software Technology*, vol. 51, no. 9, pp. 1319–1326, 2009.
- [109] E. H. Vashisht, S. Bharadwaj, and S. Sharma, “Impact of clone refactoring on external quality attributes of open source softwares,” *International Journal of Scientific Research in Computer Science, Engineering, and Information Technology*, vol. 3, no. 8, pp. 86–94, 2018.
- [110] R. Greenberg, J. F. Wacker, W. K. Hartmann, and C. R. Chapman, “Planetesimals to planets: Numerical simulation of collisional evolution,” *Icarus*, vol. 35, no. 1, pp. 1–26, 1978.
- [111] A. H. Larsen et al., “The atomic simulation environment—a python library for working with atoms,” *Journal of Physics: Condensed Matter*, vol. 29, no. 27, p. 273 002, 2017.
- [112] H. Mahbub Rahman, *Riasat-mahbub/refactor-replication* — *github.com*, <https://github.com/riasat-mahbub/refactor-replication>, [Accessed 30-07-2025], 2025.
- [113] S. Habchi, N. Moha, and R. Rouvoy, “Android code smells: From introduction to refactoring,” *Journal of Systems and Software*, vol. 177, p. 110 964, 2021.
- [114] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, “Revisiting the relationship between code smells and refactoring,” in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, IEEE, 2016, pp. 1–4.

- [115] O. Hamdi, A. Ouni, M. Ó. Cinnéide, and M. W. Mkaouer, “A longitudinal study of the impact of refactoring in android applications,” *Information and Software Technology*, vol. 140, p. 106 699, 2021.
  
- [116] A. C. Bibiano et al., “A quantitative study on characteristics and effect of batch refactoring on code smells,” in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, 2019, pp. 1–11.
  
- [117] C. Tavares, M. Bigonha, and E. Figueiredo, “Analyzing the impact of refactoring on bad smells,” in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 2020, pp. 97–101.
  
- [118] S. Kannangara and W. Wijayanayake, “An empirical evaluation of impact of refactoring on internal and external measures of code quality,” *arXiv preprint arXiv:1502.03526*, 2015.
  
- [119] B. Geppert, A. Mockus, and F. Robler, “Refactoring for changeability: A way to go?” In *11th IEEE International Software Metrics Symposium (METRICS’05)*, IEEE, 2005, 10–pp.

# Appendix A

## Complementary Materials

### A.1 Replication Package

#### A.1.1 On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software

Zenodo Archive: <https://zenodo.org/records/13621760>

#### A.1.2 On the Effectiveness, Risks, and Impact of Refactoring Practices in Simulation Modelling Software

GitHub Repository: <https://github.com/riasat-mahbub/refactor-replication>

### A.2 Copyright Permissions

#### A.2.1 On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2. In the case of illustrations or tabular material, we require that the copyright line © 2024 IEEE appear prominently with each reprinted figure and/or table.
3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1. The following IEEE copyright/ credit notice should be placed prominently in the references: © 2024 IEEE. Reprinted, with permission, from Riasat Mahbub, Mohammad Masudur Rahman, Muhammad Ahsanul Habib, On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software, IEEE International Workshop on Source Code Analysis and Manipulation, October 2024.
2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Dalhousie University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to [http://www.ieee.org/publications\\_standards/publications/rights/rights\\_link.html](http://www.ieee.org/publications_standards/publications/rights/rights_link.html) to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.